深度視覺 HW7

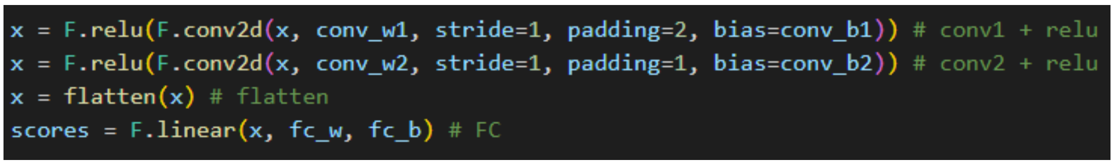
M113040064 李冠宏

Part I. Preparation

這邊是做一些事前工作，因此不再贅述。

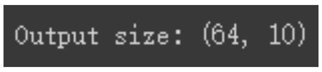
Part II. Backbones Pytorch

這部分是用 pytorch 裡 nn 的一些套件來直接做神經網路，不用已定義好的模組。首先是 three_layer_convnet，我們透過 torch.nn.functional 裡面的套件來定義網路層。過程為先將 input 放入卷積層，做一次 ReLU，再重複一次，最後拉平並送入全連接層輸出結果。



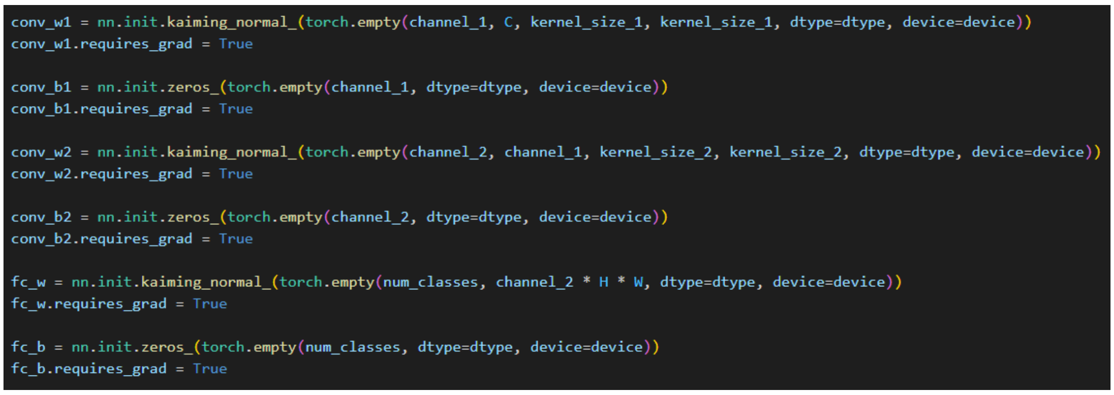
```python
x = F.relu(F.conv2d(x, conv_w1, stride=1, padding=2, bias=conv_b1)) # conv1 + relu
x = F.relu(F.conv2d(x, conv_w2, stride=1, padding=1, bias=conv_b2)) # conv2 + relu
x = flatten(x) # flatten
scores = F.linear(x, fc_w, fc_b) # FC
```

最後有得到正確的 shape



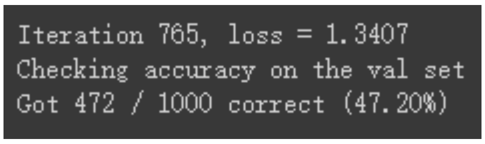
```
Output size: (64, 10)
```

接下來是初始化參數的部分，定義了 initialize_three_layer_conv_part2。
這邊主要是用 nn.init 裡面的函式來初始化 kernel 和 bias



```python
conv_w1 = nn.init.kaiming_normal_(torch.empty(channel_1, C, kernel_size_1, kernel_size_1, dtype=dtype, device=device))
conv_w1.requires_grad = True

conv_b1 = nn.init.zeros_(torch.empty(channel_1, dtype=dtype, device=device))
conv_b1.requires_grad = True

conv_w2 = nn.init.kaiming_normal_(torch.empty(channel_2, channel_1, kernel_size_2, kernel_size_2, dtype=dtype, device=device))
conv_w2.requires_grad = True

conv_b2 = nn.init.zeros_(torch.empty(channel_2, dtype=dtype, device=device))
conv_b2.requires_grad = True

fc_w = nn.init.kaiming_normal_(torch.empty(num_classes, channel_2 * H * W, dtype=dtype, device=device))
fc_w.requires_grad = True

fc_b = nn.init.zeros_(torch.empty(num_classes, dtype=dtype, device=device))
fc_b.requires_grad = True
```

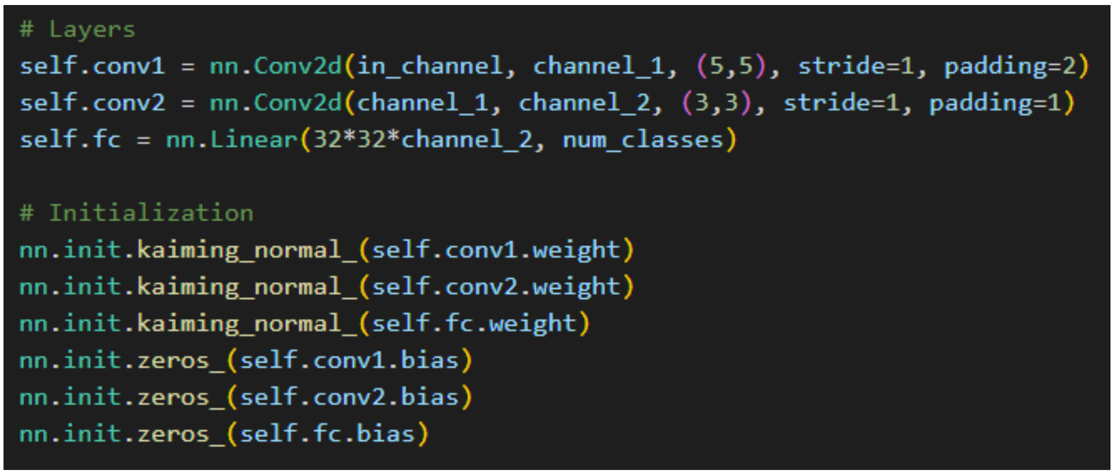
以下為完成上述過程後的訓練結果



```
Iteration 765, loss = 1.3407
Checking accuracy on the val set
Got 472 / 1000 correct (47.20%)
```

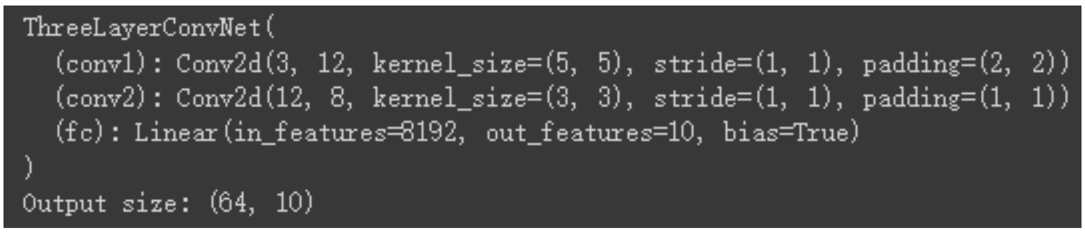
Part III. Pytorch Module API

這部份多使用了 nn.Module，能夠讓我們可以更快的定義 model。
我們自行定義了 ThreeLayerConvNet 類別，和剛剛不同的是，我們要定義的 layer 會寫在建構式內，並在裡面做初始化



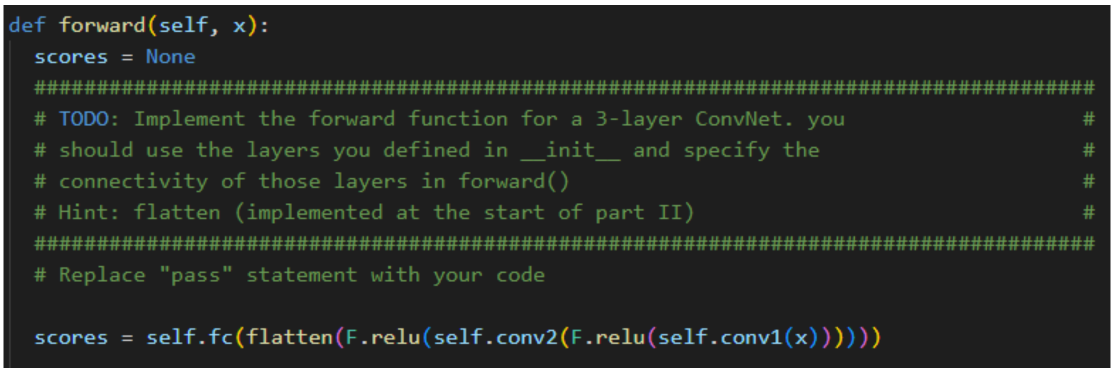
```
# Layers
self.conv1 = nn.Conv2d(in_channel, channel_1, (5,5), stride=1, padding=2)
self.conv2 = nn.Conv2d(channel_1, channel_2, (3,3), stride=1, padding=1)
self.fc = nn.Linear(32*32*channel_2, num_classes)

# Initialization
nn.init.kaiming_normal_(self.conv1.weight)
nn.init.kaiming_normal_(self.conv2.weight)
nn.init.kaiming_normal_(self.fc.weight)
nn.init.zeros_(self.conv1.bias)
nn.init.zeros_(self.conv2.bias)
nn.init.zeros_(self.fc.bias)
```

Output 為正確的結果



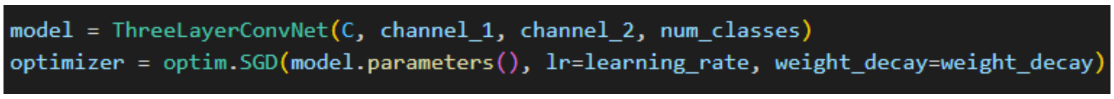
```
ThreeLayerConvNet(
  (conv1): Conv2d(3, 12, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(12, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc): Linear(in_features=8192, out_features=10, bias=True)
)
Output size: (64, 10)
```

下面則是這個類別的 method forward，其中定義了要如何將 input 往下傳遞，也就是我們實際上要做的動作



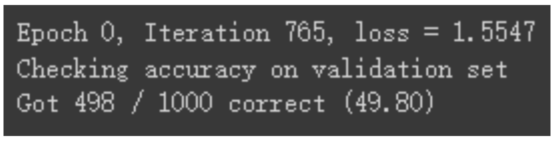
```
def forward(self, x):
    scores = None
    ########################################################################
    # TODO: Implement the forward function for a 3-layer ConvNet. you      #
    # should use the layers you defined in __init__ and specify the        #
    # connectivity of those layers in forward()                            #
    # Hint: flatten (implemented at the start of part II)                  #
    ########################################################################
    # Replace "pass" statement with your code

    scores = self.fc(flatten(F.relu(self.conv2(F.relu(self.conv1(x))))))
```

除此之外，我們還要定義一個初始化函式，來初始化 model 和 optimizer



```
model = ThreeLayerConvNet(C, channel_1, channel_2, num_classes)
optimizer = optim.SGD(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
```

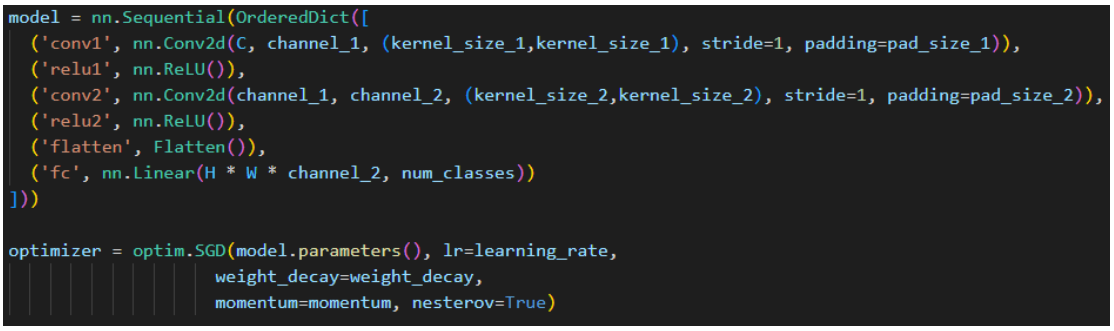
以下為最後訓練的結果



```
Epoch 0, Iteration 765, loss = 1.5547
Checking accuracy on validation set
Got 498 / 1000 correct (49.80)
```

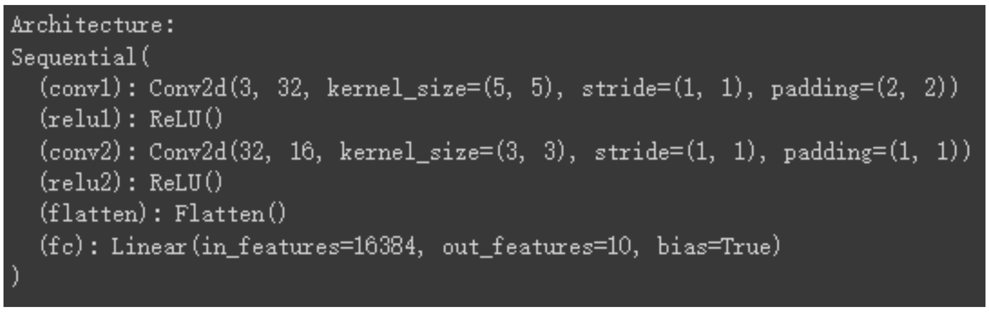
Part IV. Pytorch Sequential API

這個部份我們運用到了 nn.Sequential 這個套件，和 nn.Module 相比，他又更方便了一點，我們不需要去定義建構式和 method，直接可以將要的屬性打包在一起。在此我們定義了 initialize_three_layer_conv_part4

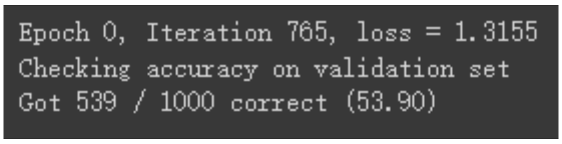


```
model = nn.Sequential(OrderedDict([
    ('conv1', nn.Conv2d(C, channel_1, (kernel_size_1,kernel_size_1), stride=1, padding=pad_size_1)),
    ('relu1', nn.ReLU()),
    ('conv2', nn.Conv2d(channel_1, channel_2, (kernel_size_2,kernel_size_2), stride=1, padding=pad_size_2)),
    ('relu2', nn.ReLU()),
    ('flatten', Flatten()),
    ('fc', nn.Linear(H * W * channel_2, num_classes))
]))

optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      weight_decay=weight_decay,
                      momentum=momentum, nesterov=True)
```

最後 output 為我們定義的架構和訓練結果



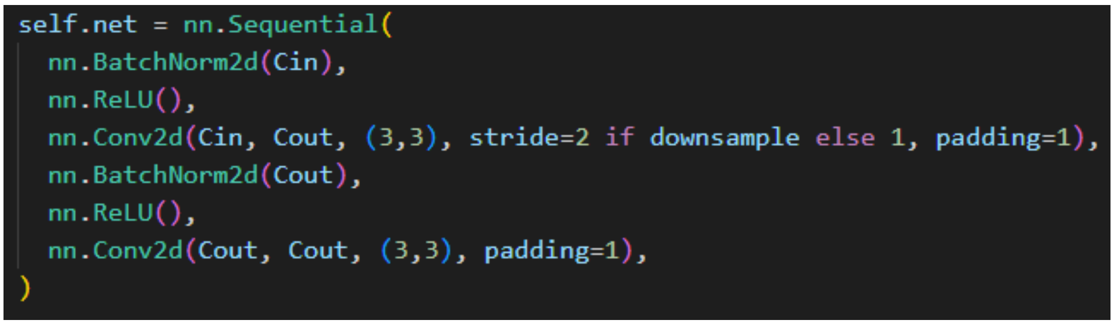
```
Architecture:
Sequential(
  (conv1): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (relu1): ReLU()
  (conv2): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu2): ReLU()
  (flatten): Flatten()
  (fc): Linear(in_features=16384, out_features=10, bias=True)
)
```



```
Epoch 0, Iteration 765, loss = 1.3155
Checking accuracy on validation set
Got 539 / 1000 correct (53.90)
```

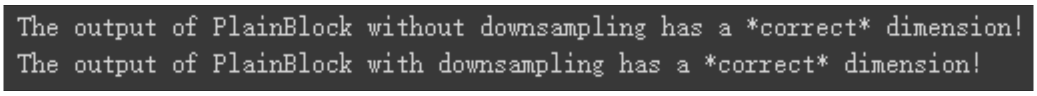
Part V. ResNet for CIFAR-10

這個部份我們要來定義 residual network，簡稱 ResNet。
首先要先定義 Plain block，也就是沒有 skip connection 的部分



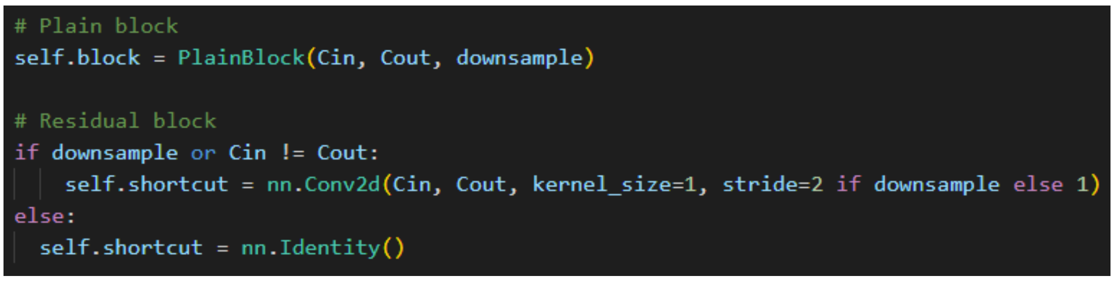
```
self.net = nn.Sequential(
  nn.BatchNorm2d(Cin),
  nn.ReLU(),
  nn.Conv2d(Cin, Cout, (3,3), stride=2 if downsample else 1, padding=1),
  nn.BatchNorm2d(Cout),
  nn.ReLU(),
  nn.Conv2d(Cout, Cout, (3,3), padding=1),
)
```

Output 結果表示為正確



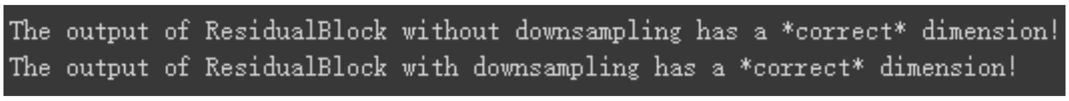
```
The output of PlainBlock without downsampling has a *correct* dimension!
The output of PlainBlock with downsampling has a *correct* dimension!
```

接下來是 residual block，也就是 skip connection 的部分



```
# Plain block
self.block = PlainBlock(Cin, Cout, downsample)

# Residual block
if downsample or Cin != Cout:
    self.shortcut = nn.Conv2d(Cin, Cout, kernel_size=1, stride=2 if downsample else 1)
else:
  self.shortcut = nn.Identity()
```

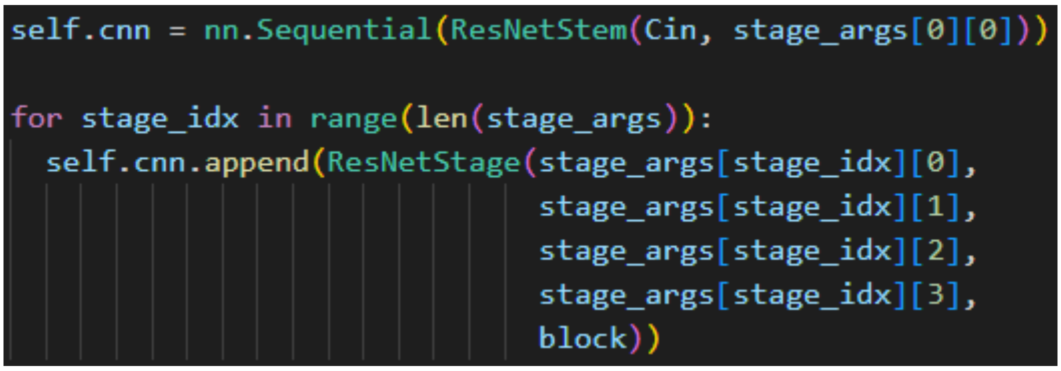
Output 結果表示為正確



```
The output of ResidualBlock without downsampling has a *correct* dimension!
The output of ResidualBlock with downsampling has a *correct* dimension!
```

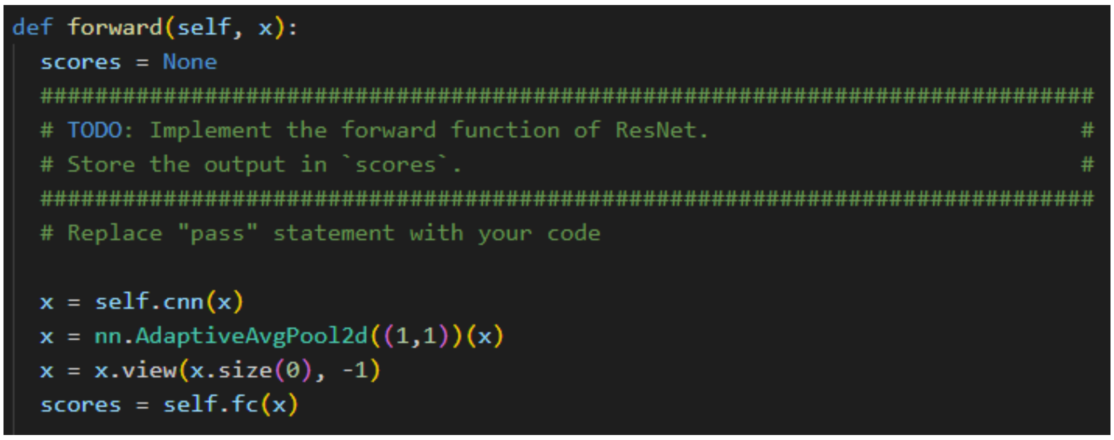
接下來，使用預先定義好的 residual stage 和 residual stem，我們就可以使用這兩個 class 和剛剛定義好的 block 來建構 ResNet 了。

首先是建構式的部分，首先使用 nn.Sequential 來將 ResNetStem 加在第一層，接下來再用迴圈將包含多個 layers 的 ResNetStage 逐一 append 到 Sequential 裡面



```
self.cnn = nn.Sequential(ResNetStem(Cin, stage_args[0][0]))

for stage_idx in range(len(stage_args)):
  self.cnn.append(ResNetStage(stage_args[stage_idx][0],
                              stage_args[stage_idx][1],
                              stage_args[stage_idx][2],
                              stage_args[stage_idx][3],
                              block))
```

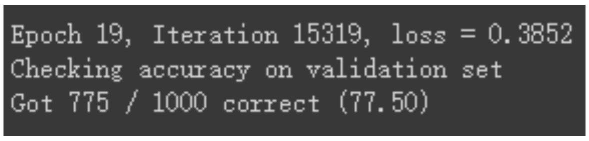
Forwarding 的話則是先將 output 送進 cnn，經過 average pooling 後，將其形狀轉換成符合 Batch size 的，最後再送進全連接層以得到輸出



```
def forward(self, x):
  scores = None
  ##########################################################################
  # TODO: Implement the forward function of ResNet.                        #
  # Store the output in `scores`.                                          #
  ##########################################################################
  # Replace "pass" statement with your code

  x = self.cnn(x)
  x = nn.AdaptiveAvgPool2d((1,1))(x)
  x = x.view(x.size(0), -1)
  scores = self.fc(x)
```

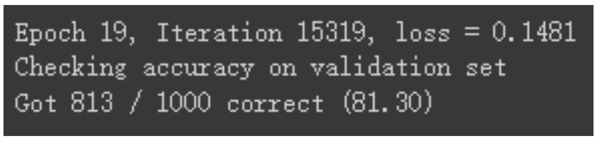
訓練結果:

Plain block:



```
Epoch 19, Iteration 15319, loss = 0.3852
Checking accuracy on validation set
Got 775 / 1000 correct (77.50)
```

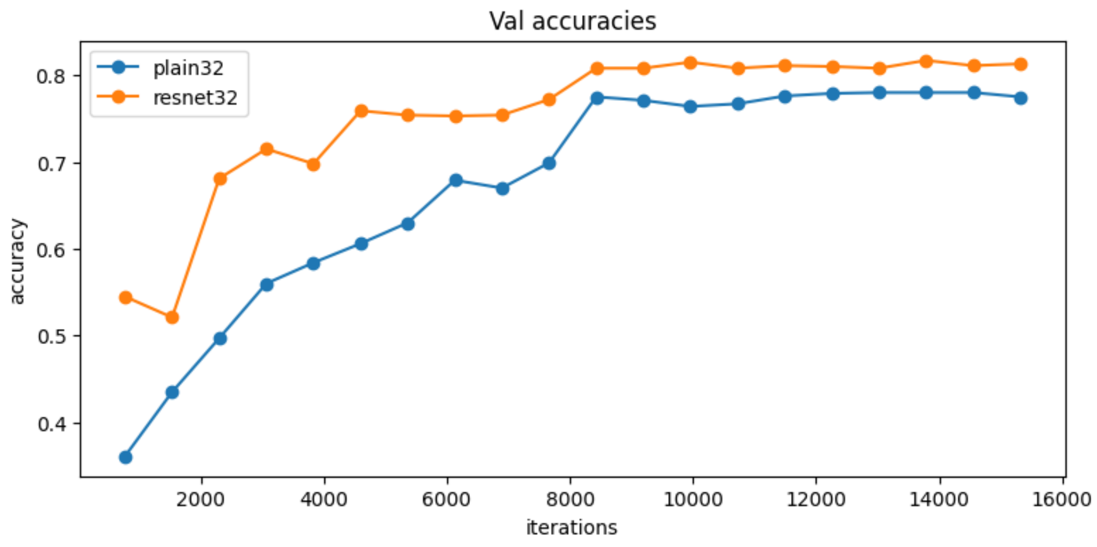
Residual block:



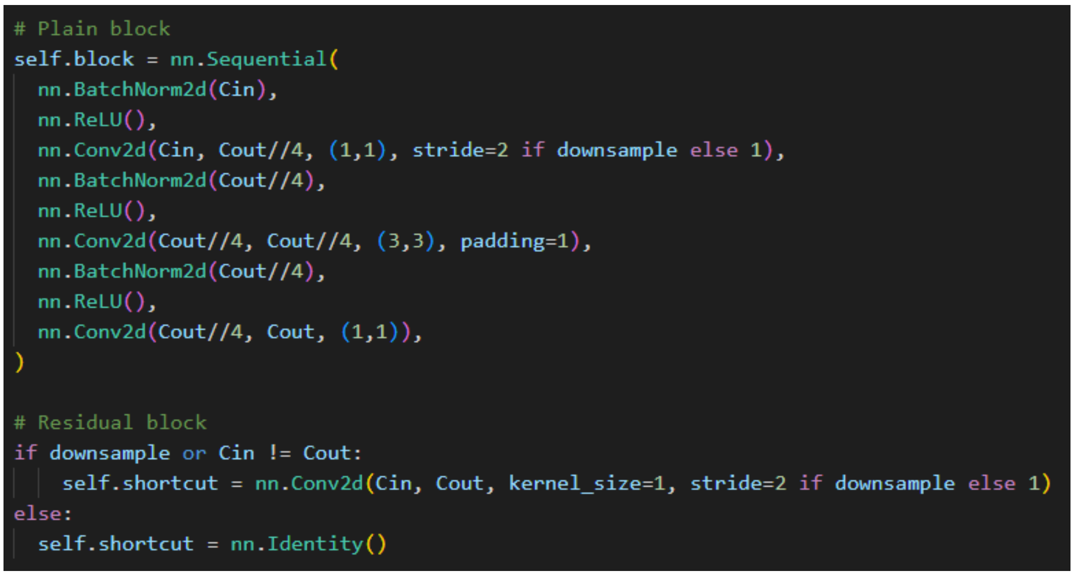
```
Epoch 19, Iteration 15319, loss = 0.1481
Checking accuracy on validation set
Got 813 / 1000 correct (81.30)
```

圖表:

我們可以發現 ResNet32 (有使用 residual block)的結果比 plain32 還要好

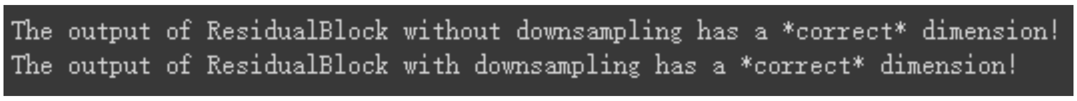


最後是 residual bottleneck block，他是一種更改 plain block 的替代方式，最終也能使得訓練結果更好，定義如下，主要是從 residual block 類別延伸，更改 plain block 的部分



```
# Plain block
self.block = nn.Sequential(
  nn.BatchNorm2d(Cin),
  nn.ReLU(),
  nn.Conv2d(Cin, Cout//4, (1,1), stride=2 if downsample else 1),
  nn.BatchNorm2d(Cout//4),
  nn.ReLU(),
  nn.Conv2d(Cout//4, Cout//4, (3,3), padding=1),
  nn.BatchNorm2d(Cout//4),
  nn.ReLU(),
  nn.Conv2d(Cout//4, Cout, (1,1)),
)

# Residual block
if downsample or Cin != Cout:
    self.shortcut = nn.Conv2d(Cin, Cout, kernel_size=1, stride=2 if downsample else 1)
else:
  self.shortcut = nn.Identity()
```

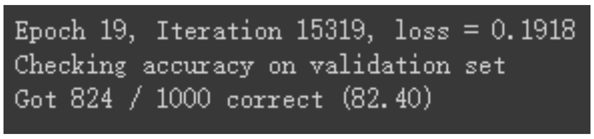
測試結果:



```
The output of ResidualBlock without downsampling has a *correct* dimension!
The output of ResidualBlock with downsampling has a *correct* dimension!
```

最後我們將定義好的 residual bottleneck block 套用上去做訓練。

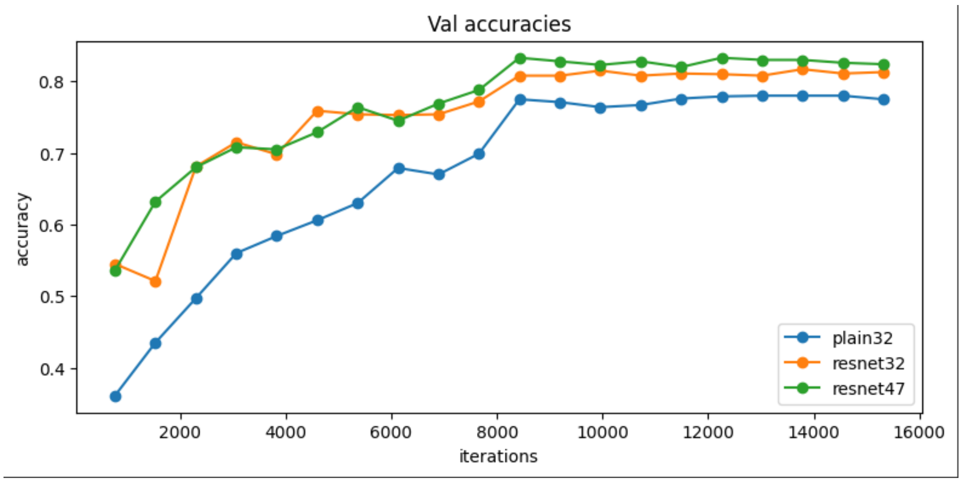
訓練結果:



```
Epoch 19, Iteration 15319, loss = 0.1918
Checking accuracy on validation set
Got 824 / 1000 correct (82.40)
```

圖表:

我們可以發現他又比我們剛剛定義的 ResNet32 更好一些。


Val accuracies
accuracy
iterations
plain32
resnet32
resnet47