HW7 成果分析

Part 1. Preparation

使用 CIFAR10 dataset 作為此次訓練資料庫。資料是數筆 32*32 pixels 的彩色影像，並且有數個類別。

Part 2. Barebones Pytorch

模型架構：Three-Layer ConvNet

使用 Barebones 建兩層 Conv_ReLU 和 一層 Linear 的簡單類神經網路。結構如下：

Forward

```
# conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
x = F.conv2d(x, conv_w1, conv_b1, padding=2)
x = F.relu(x)
x = F.conv2d(x, conv_w2, conv_b2, padding=1)
x = F.relu(x)
x = flatten(x)
scores = F.linear(x, fc_w, fc_b)
```

Backward

我們並不需要自己寫反向傳播。作業注釋提到，可以用 `requires_grad = True` 來初始化這些張量。
這就是 pytorch 的 `autograd` 。當我們使用 `requires_grad = True` 來建立張量時，pytorch 會產生一個 Computational Graphs 用來計算反向傳播的梯度值。
所以我們只要再初始化時設定即可。

```
#scores = model_fn(x, params)
#loss = F.cross_entropy(scores, y)

loss.backward() # call torch.tensor.backward() backward function to calculate
grads
```

Initialize

剛剛提到的 autograd, 就是在初始化時定義的：

```
conv_w1 = torch.empty(channel_1, C, kernel_size_1, kernel_size_1, dtype=dtype,
device=device)
conv_b1 = torch.empty(channel_1, dtype=dtype, device=device)
conv_w2 = torch.empty(channel_2, channel_1, kernel_size_2, kernel_size_2,
dtype=dtype, device=device)
conv_b2 = torch.empty(channel_2, dtype=dtype, device=device)
fc_w = torch.empty(num_classes, channel_2 * H * W, dtype=dtype, device=device)
fc_b = torch.empty(num_classes, dtype=dtype, device=device)

nn.init.kaiming_normal_(conv_w1, nonlinearity='relu')
conv_w1.requires_grad = True # requires_grad = True to do autograd
nn.init.kaiming_normal_(conv_w2, nonlinearity='relu')
conv_w2.requires_grad = True
nn.init.kaiming_normal_(fc_w, nonlinearity='relu')
fc_w.requires_grad = True
nn.init.zeros_(conv_b1)
conv_b1.requires_grad = True
nn.init.zeros_(conv_b2)
conv_b2.requires_grad = True
nn.init.zeros_(fc_b)
fc_b.requires_grad = True
```

模型訓練

使用剛剛的三層簡易模型進行訓練，並使用以下參數：

```
learning_rate = 3e-3
params = initialize_three_layer_conv_part2(dtype=to_float, device='cuda') # w/
kaiming initialize
acc_hist_part2 = train_part2(three_layer_convnet, params, learning_rate)
```

得到結果：

```
Iteration 765, loss = 1.2418
Checking accuracy on the val set
Got 461 / 1000 correct (46.10%)
```

Check Accuracy

計算正確率，把 `x` 和 `params` 傳入進行 forward 計算。

```
# in check_accuracy_part2()

    scores = model_fn(x, params)
    _, preds = scores.max(1)
    num_correct += (preds == y).sum()
    num_samples += preds.size(0)
    acc = float(num_correct) / num_samples
```

Part 3. PyTorch Module API

模型架構：Three-Layer ConvNet

Barebones 必須自己追蹤每個張量，當模型較大時，就會變得很不容易。此時，呼叫 Moule API 可以讓程式更加簡潔。

Initialize

使用 Module API 時，先在 `__init__` 裡面定義要使用的 layer object，同時進行權重和偏置值的初始化。

記得執行 `super().__init__()` 讓 Module API 也初始化。

```
# in class TwoLayerFC(nn.Module)

def __init__(self, input_size, hidden_size, num_classes):
    # super().__init__()
    H, W = 32, 32
    self.conv1 = nn.Conv2d(in_channel, channel_1, 5, padding=2)
    self.relu1 = nn.ReLU()
    self.conv2 = nn.Conv2d(channel_1, channel_2, 3, padding=1)
    self.relu2 = nn.ReLU()
    self.fc = nn.Linear(channel_2 * H * W, num_classes)
    self.conv1.weight = nn.init.kaiming_normal_(self.conv1.weight)
    self.conv2.weight = nn.init.kaiming_normal_(self.conv2.weight)
    self.fc.weight = nn.init.kaiming_normal_(self.fc.weight)
    self.conv1.bias = nn.init.zeros_(self.conv1.bias)
    self.conv2.bias = nn.init.zeros_(self.conv2.bias)
    self.fc.bias = nn.init.zeros_(self.fc.bias)
```

Forward

把剛剛初始化的參數和層連接在一起，**這裡只能使用有在 `init` 裡面定義的參數和層，不能再新增其他的**。

```
# in class TwoLayerFC(nn.Module)

def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = flatten(x)
    scores = self.fc(x)
```

一樣， pytorch 會幫我們處理反向傳播的運算，所以我們不用自己寫。所以到這裡，使用 Module API 建構的模型就算完成了。

模型訓練

把剛剛的模型讀進一個變數，並且定義 optimizer 。這邊使用的是 SGD optimizer 。

```
# in initialize_three_layer_conv_part3()

learning_rate = 3e-3
weight_decay = 1e-4

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

開始訓練：

```
# HW7.py

acc_hist_part3, _ = train_part345(model, optimizer)
```

得到結果：

```
Epoch 0, Iteration 765, loss = 1.5552
Checking accuracy on validation set
Got 499 / 1000 correct (49.90)
```

Check Accuracy

在 Module API 當中，不再需要手動傳 `params` 進去了。

```
# in check_accuracy_part34()

    scores = model(x)
    _, preds = scores.max(1)
    num_correct += (preds == y).sum()
    num_samples += preds.size(0)
    acc = float(num_correct) / num_samples
```

Part 4. PyTorch Sequential API

Sequential API 雖然靈活度不及 Module API 但是對於大部份使用來說，更進一步簡化了模型的建立，也更容易撰寫。

模型架構：Three-Layer ConvNet

比起前幾個方式， Sequential API 直接整合了初始化和建立 forward 連接的步驟：

```
# learning_rate = 1e-2
# weight_decay = 1e-4
# momentum = 0.5

model = nn.Sequential(OrderedDict([
    ('conv1', nn.Conv2d(C, channel_1, 5, padding=2)),
    ('relu1', nn.ReLU()),
    ('conv2', nn.Conv2d(channel_1, channel_2, 3, padding=1)),
    ('relu2', nn.ReLU()),
    ('flatten', Flatten()),
    ('fc', nn.Linear(channel_2 * H * W, num_classes))
  ]))

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

模型訓練

```
acc_hist_part4, _ = train_part345(model, optimizer)
```

得到結果：

```
Epoch 0, Iteration 765, loss = 1.3151
Checking accuracy on validation set
Got 534 / 1000 correct (53.40)
```

Part 5. ResNet

與一般的神經網路最大的不同，就是 ResNet 多了恆等映射 (Identity) 的連接，解決了梯度消失的問題。

PreResNet

接下來要做的 PreResNet 則是會在 Conv2d 前先經過一次的 activation。

模型架構

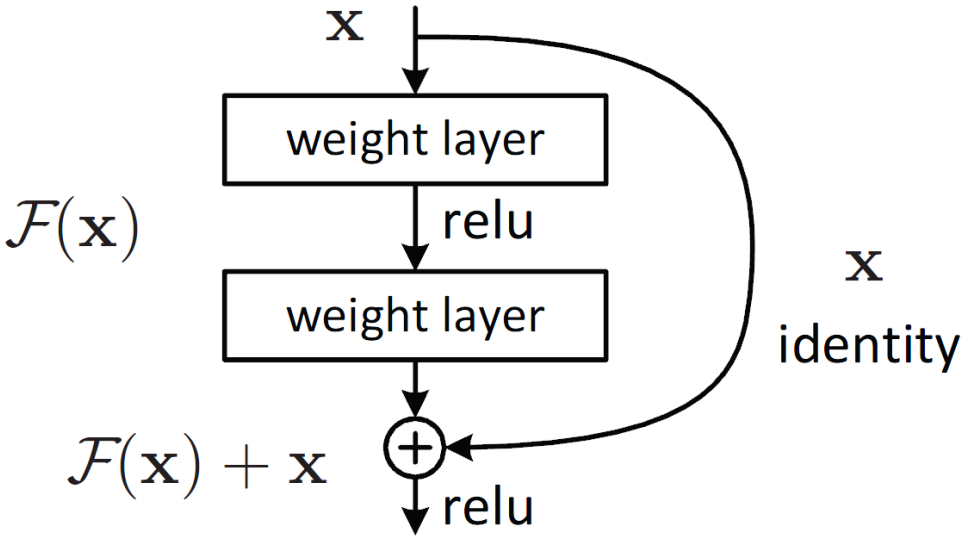
Plain Block

和之前的神經網路不同，先經過 ReLU 之後才進到 Conv2d 裡面做卷積運算。

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

Residual Block

建立恆等映射，如下圖，從輸入多拉一條短路神經元到輸出。這邊要注意如果$C_{in} \neq C_{out}$要另外做處理，才有辦法相加。



1. If $C_{in} = C_{out}$ and $\mathcal{F}$ does not perform downsampling, then $\mathcal{F}(x)$ will have the same shape as x, so $\mathcal{G}$ is the identity function: $\mathcal{G}(x) = x$
2. If $C_{in} \neq C_{out}$ and $\mathcal{F}$ does not downsample, then $\mathcal{G}$ is a 1x1 convolution with C_{out} filters and stride 1.
3. If $\mathcal{F}$ downsamples, then $\mathcal{G}$ is a 1x1 convolution with C_{out} filters and stride 2.

```
self.block = PlainBlock(Cin, Cout, downsample)
if not downsample:
  if Cin != Cout:
    self.shortcut = nn.Conv2d(Cin, Cout, kernel_size=1, stride=1)
  else:
    self.shortcut = nn.Identity()
else:
  self.shortcut = nn.Conv2d(Cin, Cout, kernel_size=1, stride=2)
```

Residual Bottleneck Block

加上一層 1x1 的 kernel 在每個 block 的最尾端，再進行深層運算的時候可以減少參數量，增加效率。

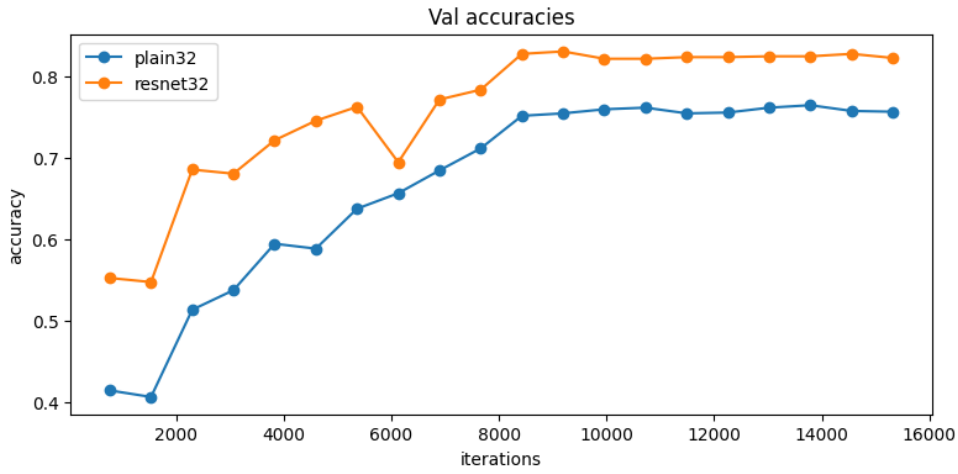
```
model = \
    nn.Sequential(
        nn.BatchNorm2d(Cin),
        nn.ReLU(),
        nn.Conv2d(Cin, Cout // 4, kernel_size=1, stride=(2 if downsample else
1)),
        nn.BatchNorm2d(Cout // 4),
        nn.ReLU(),
        nn.Conv2d(Cout // 4, Cout // 4, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(Cout // 4),
        nn.ReLU(),

        # Bottleneck layer
        nn.Conv2d(Cout // 4, Cout, kernel_size=1, stride=1)
    )

    self.block = model

    if not downsample:
      if Cin != Cout:
        self.shortcut = nn.Conv2d(Cin, Cout, kernel_size=1, stride=1)
      else:
        self.shortcut = nn.Identity()
    else:
      self.shortcut = nn.Conv2d(Cin, Cout, kernel_size=1, stride=2)
```

模型訓練



可以發現，多了 Identity 連接會讓模型有更好的表現，尤其是在深層網路時，理論上差異會愈大。

因為多了 Identity，所以一定程度避免了梯度消失或是梯度爆炸的問題，所以讓日後的模型可以深度發展。

以下是 plain32, resnet32, resnet47 的訓練結果，可以發現

resnet47 因為深度更深的關係，所以又比 resnet32 的表現更好一點點。