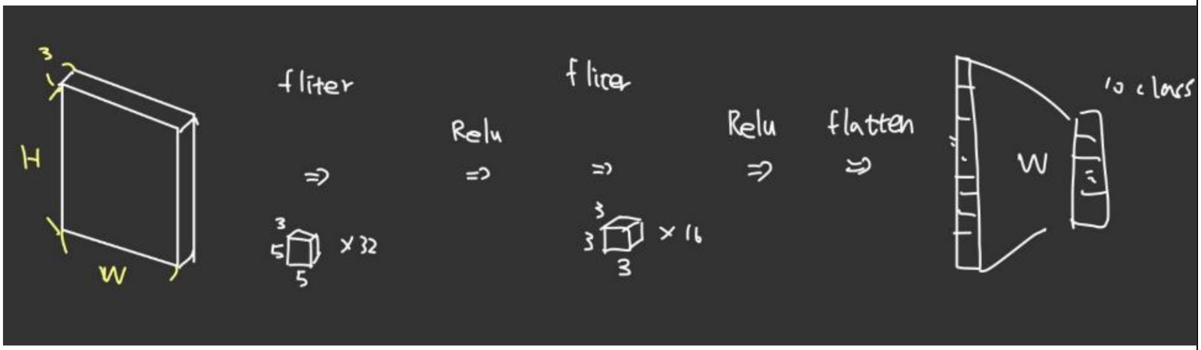
HW9 Pytorch Autograd and NN

M113040105 劉東霖

壹.Part II.Barebones Pytorch:

一.three_layer_convent:

整個程式碼的架構圖如下:



首先，計算第一層 convolution。進入一個 conv，輸入權重和 bias 為 conv_w1 和 conv_b1，padding=2 和 stride=1，再進入 relu。程式碼如下。

```
buff=F.conv2d(input=x,weight=conv_w1,bias=conv_b1,padding=2,stride=1)
buff=F.relu(input=buff)
```

再來計算第二層 convolution。首先，進入一個 conv，輸入權重和 bias 為 conv_w2 和 conv_b2，padding=1 和 stride=1，再進入 relu。程式碼如下。

```
buff=F.conv2d(input=buff,weight=conv_w2,bias=conv_b2,padding=1,stride=1)
buff=F.relu(input=buff)
```

再來計算第三層全連接層。首先，先把第二層算出來的東西坦平，再用 linear 函式算出輸出 scores，權重和 bias 為 fc_w 和 fc_b。程式碼如下。

```
buff=flatten(buff)
scores=F.linear(buff,fc_w,fc_b)
```

二.initialize_three_layer_conv_part2:

如下圖所示，利用 Kaiming 初始化 conv_w1，size 為(channel_1,C,kernel_size_1,kernel_size_1)，裝置和資料型態都跟輸入的一樣。最後把 require_grad 設為 true 讓這個參數可以反向傳播。

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

如下圖所示，把 conv_b1 初始化所有元素為 0，size 為 channel_1，裝置和資料型態都跟輸入的一樣。最後把 require_grad 設為 true 讓這個參數可以反向傳播。

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

如下圖所示，利用 Kaiming 初始化 conv_w2，size 為(channel_2, channel_1, kernel_size_2, kernel_size_2)，裝置和資料型態都跟輸入的一樣。最後把 require_grad 設為 true 讓這個參數可以反向傳播。

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

如下圖所示，把 conv_b2 初始化所有元素為 0，size 為 channel_2，裝置和資料型態都跟輸入的一樣。最後把 require_grad 設為 true 讓這個參數可以反向傳播。

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

如下圖所示，利用 Kaiming 初始化 fc_w，size 為(num_classes, channel_2*H* W)，裝置和資料型態都跟輸入的一樣。最後把 require_grad 設為 true 讓這個參數可以反向傳播。

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

如下圖所示，把 fc_b 初始化所有元素為 0，size 為 num_classes，裝置和資料型態都跟輸入的一樣。最後把 require_grad 設為 true 讓這個參數可以反向傳播。

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

三. 程式執行結果:

1. 如下圖所示，在所有權重和 bias 和輸入都是全 0 的 tensor 的情況下，輸出 size 為[64,10]。

```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=to_float)  # minibatch size 64, image size [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=to_float)  # [out_channel, in_channel, kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=to_float)  # [out_channel, in_channel, kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, before the fully-connected layer
    fc_w = torch.zeros((10, 9 * 32 * 32))
    fc_b = torch.zeros(10)

    # YOUR_TURN: Impelement the three_layer_convnet function
    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b])
    print('Output size:', list(scores.size()))  # you should see [64, 10]
three_layer_convnet_test()

Output size: [64, 10]
```

2. 如下圖所示，拿前面寫的 function 拿來做訓練，發現準確率大多落在四十幾%左右。

```
Iteration 0, loss = 2.6007
Checking accuracy on the val set
Got 109 / 1000 correct (10.90%)
Iteration 100, loss = 1.9974
Checking accuracy on the val set
Got 349 / 1000 correct (34.90%)
Iteration 200, loss = 1.7885
Checking accuracy on the val set
Got 392 / 1000 correct (39.20%)
Iteration 300, loss = 1.6607
Checking accuracy on the val set
Got 418 / 1000 correct (41.80%)
Iteration 400, loss = 1.5935
Checking accuracy on the val set
Got 451 / 1000 correct (45.10%)
Iteration 500, loss = 1.6623
Checking accuracy on the val set
Got 449 / 1000 correct (44.90%)
Iteration 600, loss = 1.6291
Checking accuracy on the val set
Got 469 / 1000 correct (46.90%)
Iteration 700, loss = 1.7864
Checking accuracy on the val set
Got 483 / 1000 correct (48.30%)
Iteration 765, loss = 1.2412
Checking accuracy on the val set
Got 461 / 1000 correct (46.10%)
```

貳.Part III.Pytorch Module API:

一.ThreeLayerConvNet:

1.__init__:

首先，定義了一個二維卷積層 self.conv1，它具有 in_channel 個輸入通道，channel_1 個輸出通道，卷積核大小為 5，步長為 1，並在輸入邊界上進行填充，以維持輸出尺寸不變。接著，使用 Kaiming 初始化 self.conv1 的權重，並將偏差設置為零。程式碼如下:

```
self.conv1=nn.Conv2d(in_channels=in_channel,out_channels=channel_1,kernel_size=5,stride=1,padding=2)
nn.init.kaiming_normal_(self.conv1.weight)
nn.init.zeros_(self.conv1.bias)
```

再來，定義了一個二維卷積層 self.conv2，它具有 channel_1 個輸入通道，channel_2 個輸出通道，卷積核大小為 3，步長為 1，並在輸入邊界上進行填充，以維持輸出尺寸不變。接著，使用 Kaiming 初始化 self.conv2 的權重，並將偏差設置為零。程式碼如下:

```
self.conv2=nn.Conv2d(in_channels=channel_1,out_channels=channel_2,kernel_size=3,stride=1,padding=1)
nn.init.kaiming_normal_(self.conv2.weight)
nn.init.zeros_(self.conv2.bias)
```

最後，定義了一個全連接層 self.fully，輸入大小為 channel_2*32*32 ，輸出大小為 num_classes。同樣地，使用 Kaiming 初始化 self.fully 的權重，並將偏差設置為零。程式碼如下:

```
self.fully=nn.Linear(channel_2*32*32,  num_classes)
nn.init.kaiming_normal_(self.fully.weight)
nn.init.zeros_(self.fully.bias)
```

2. forward:

架構的流程為:x->conv1->relu->conv2->relu->flatten->fully->scores，scores 為輸出。程式碼如下:

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

二. initialize_three_layer_conv_part3:

首先創立一個 ThreeLayerConvNet model，in_channel 為 C，channel_1 為 channel_1, channel_2 為 channel_2, 輸出數為 num_classes。程式碼如下:

```
model=ThreeLayerConvNet(in_channel=C, channel_1=channel_1, channel_2=channel_2, num_classes=num_classes)
```

再來是優化器的部分。題目要求用 sgd，參數為 model 的參數，並設置好 learning rate 和 weight decay。程式碼如下:

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

三. 程式執行結果:

1. 如下圖所示，在輸入是個 size 為(64, 3, 32, 32)全為 0 的 tensor，in_channel=3, channel_1=12, channel_2=8, num_classes=10 的情況下帶入 ThreeLayerConvNet model，輸出 size 為[64, 10]，並把模型架構印出來。

```
def  test_ThreeLayerConvNet():
    x  =  torch.zeros((64,  3,  32,  32),  dtype=to_float)   #  minibatch  size  64,  image
    #  YOUR_TURN:  Impelement  the  functions  in  ThreeLayerConvNet  class
    model  =  ThreeLayerConvNet(in_channel=3,  channel_1=12,  channel_2=8,  num_classes=10)
    scores  =  model(x)
    print(model)  #  printing  `nn.Module`  shows  the  architecture  of  the  module.
    print('Output  size:',  list(scores.size()))   #  you  should  see  [64,  10]
test_ThreeLayerConvNet()

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))
  (fully): Linear(in_features=8192, out_features=10, bias=True)
)
Output size: [64, 10]
```

2.如下圖所示，拿前面寫的 initialize_three_layer_conv_part3 函式裡面的 model 和優化器拿來訓練，發現準確率最後落在 47%。

```
Epoch 0, Iteration 0, loss = 3.2691
Checking accuracy on validation set
Got 126 / 1000 correct (12.60)

Epoch 0, Iteration 100, loss = 1.9328
Checking accuracy on validation set
Got 318 / 1000 correct (31.80)

Epoch 0, Iteration 200, loss = 1.7352
Checking accuracy on validation set
Got 372 / 1000 correct (37.20)

Epoch 0, Iteration 300, loss = 1.6908
Checking accuracy on validation set
Got 407 / 1000 correct (40.70)

Epoch 0, Iteration 400, loss = 1.4226
Checking accuracy on validation set
Got 417 / 1000 correct (41.70)

Epoch 0, Iteration 500, loss = 1.5764
Checking accuracy on validation set
Got 436 / 1000 correct (43.60)

Epoch 0, Iteration 600, loss = 1.3392
Checking accuracy on validation set
Got 439 / 1000 correct (43.90)

Epoch 0, Iteration 700, loss = 1.5629
Checking accuracy on validation set
Got 457 / 1000 correct (45.70)

Epoch 0, Iteration 765, loss = 1.7755
Checking accuracy on validation set
Got 470 / 1000 correct (47.00)
```

參.Part IV.Pytorch Sequential API:

一. initialize_three_layer_conv_part4:

這裡使用了 PyTorch 的 nn.Sequential() 方法來構建模型。

模型的第一層為卷積層，具有 C 個輸入通道、channel_1 個輸出通道，卷積核大小為 kernel_size_1，步長為 1，填充大小為 pad_size_1，並接著一個 ReLU 激活函數。

模型的第二層也是卷積層，具有 channel_1 個輸入通道、channel_2 個輸出通道，卷積核大小為 kernel_size_2，步長為 1，填充大小為 pad_size_2，並接著一個 ReLU 激活函數。

最後，把卷積完的結果坦平，並使用全連接層 nn.Linear 輸出結果。輸入大小為 channel_2*32*32 ，輸出大小為 num_classes。

程式碼如下:

```
model = nn.Sequential(
    nn.Conv2d(in_channels=C, out_channels=channel_1, kernel_size=kernel_size_1, stride=1, padding=pad_size_1),
    nn.ReLU(),
    nn.Conv2d(in_channels=channel_1, out_channels=channel_2, kernel_size=kernel_size_2, stride=1, padding=pad_size_2),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32, num_classes)
)
```

再來是優化器的部分。題目要求用 nesterov momentum，參數為 model 的參數，並設置好 learning rate 和 weight decay 和 momentum。程式碼如下:

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

二.程式執行結果:

3.如下圖所示，拿前面寫的 initialize_three_layer_conv_part4 的 function 拿來做訓練，發現準確率最後落在 53.4%，並把模型架構印出來。

```
Architecture:
Sequential(
  (0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (1): ReLU()
  (2): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU()
  (4): Flatten()
  (5): Linear(in_features=16384, out_features=10, bias=True)
)
```

```
Epoch 0, Iteration 0, loss = 2.2960
Checking accuracy on validation set
Got 136 / 1000 correct (13.60)

Epoch 0, Iteration 100, loss = 1.6668
Checking accuracy on validation set
Got 382 / 1000 correct (38.20)

Epoch 0, Iteration 200, loss = 1.4602
Checking accuracy on validation set
Got 479 / 1000 correct (47.90)

Epoch 0, Iteration 300, loss = 1.7123
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 0, Iteration 400, loss = 1.6086
Checking accuracy on validation set
Got 490 / 1000 correct (49.00)

Epoch 0, Iteration 500, loss = 1.4320
Checking accuracy on validation set
Got 498 / 1000 correct (49.80)

Epoch 0, Iteration 600, loss = 1.5051
Checking accuracy on validation set
Got 544 / 1000 correct (54.40)

Epoch 0, Iteration 700, loss = 1.3422
Checking accuracy on validation set
Got 541 / 1000 correct (54.10)

Epoch 0, Iteration 765, loss = 1.3148
Checking accuracy on validation set
Got 534 / 1000 correct (53.40)
```