深度學習

HW9

學號 : B103012002

姓名 : 林凡皓

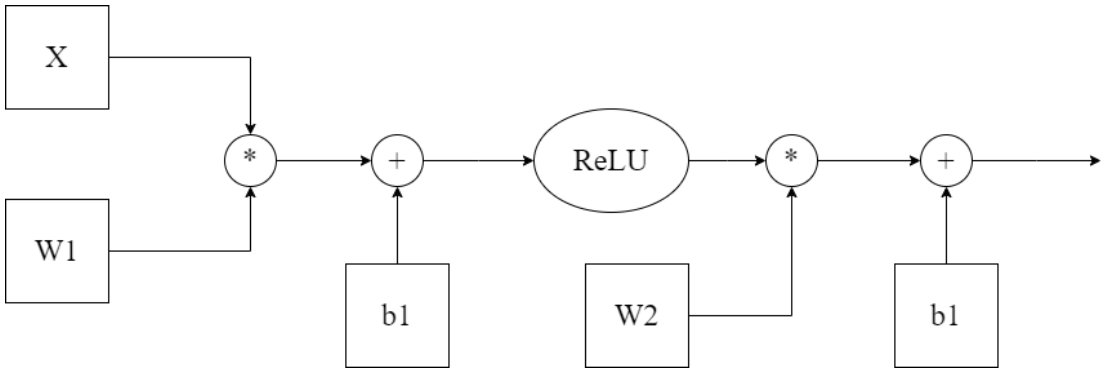
一、 Barebones PyTorch

PyTorch 提供 high-level 的 API 來讓使用者方便定義模型。這部分會從一個簡單的全連接 ReLU 網路架構開始，利用 PyTorch tensor 的操作來計算 forward path，並用 PyTorch autograd 來計算梯度。

在創建 PyTorch tensor 時，如果將參數 requires_grad 設定成 True，則在計算與該 tensor 相關部分的時候 PyTorch 會建立一個 computational graph 來計算梯度，並將梯度存到 x.grad 中。

- Barebones PyTorch: Two-Layer Network

  Two-Layer Network 架構如下



  根據此架構來完成 forward path。

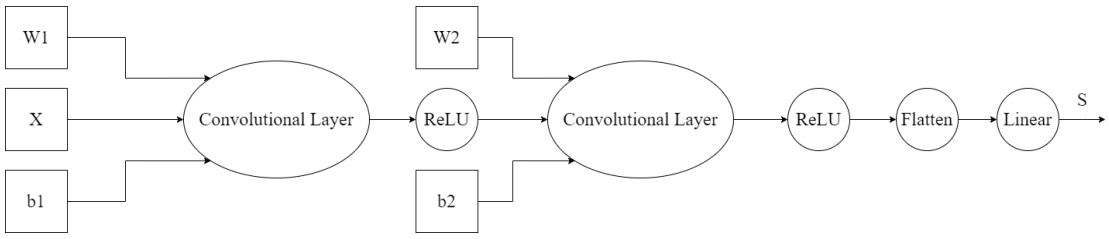
  為了確認網路架構的正確性，我們將輸入與參數全部設定成 0 之後傳入網路中，並將輸出的形狀顯示出來，結果如下

```
Output size: [64, 10]
```

  此結果為正確結果，代表說網路可以正常運作。

- Barebones PyTorch: Three-Layer Network

  Three-Layer Network 架構如下



  根據此架構完成 forward path。

  關於 Convolutional layer，利用 torch.nn.functional.conv2d 來實作。

  為了確認網路架構的正確性，我們將輸入與參數全部設定成 0 之後傳

入網路中，並將輸出的形狀顯示出來，結果如下

```
Output size: [64, 10]
```

- Barebones PyTorch: Kaiming Initialization

  對於 W 的初始化，採用 Kaiming initialization，我們可以直接利用 PyTorch 幫我們定義好的函數 torch.nn.init.kaiming_normal_( )來完成。

  對於 b 的初始化，我們一律將他初始化為 0。

  這邊嘗試利用 nn.init.kaiming_normal_( )與 nn.init.zeros_( )來查看輸出結果，結果如下

```
tensor([[-0.5848, -0.2690, -1.6721,  0.0918, -0.0764],
        [-0.3667, -0.3939, -0.2077, -0.6796, -0.2297],
        [-1.0569,  1.4328,  0.1971, -0.1165,  0.8137]], device='cuda:0')
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]], device='cuda:0')
```

- Barebones PyTorch: Check Accuracy

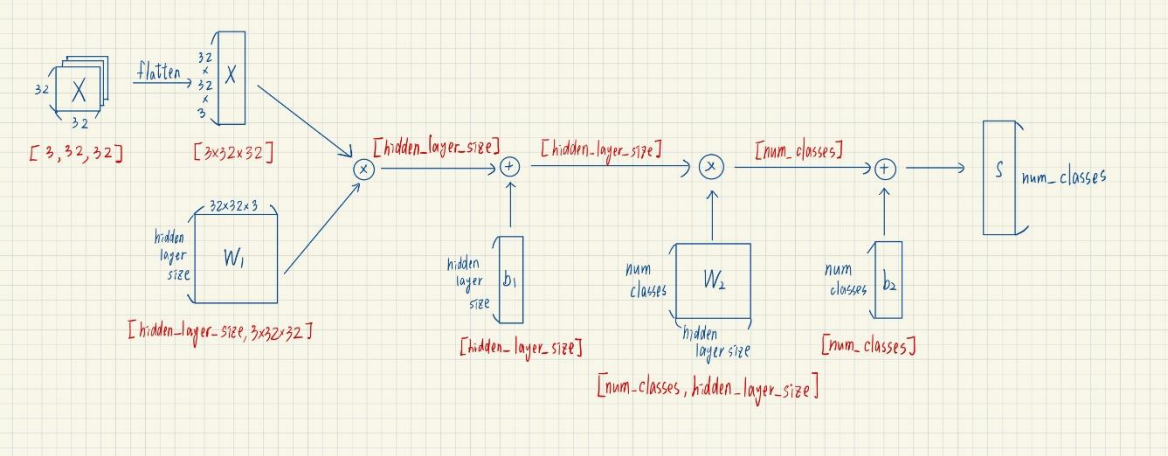
  建立 check_accuracy_part2( )，用來計算準確度。此函數會接收三個參數，分別為 data loader、model、parameters。先利用 data loader 來讀取資料，在訓練階段利用 validation set 來做驗證，在測試階段使用 test set 做測試。接著根據 model 和 parameters 來完成模型，並利用此模型來計算準確度，最後將準確度回傳。

- Barebones PyTorch: Training Loop

  定義 train_part2( )來做訓練。此函數會接收三個參數，分別為 model、parameters、learning rate。此函數主要為一個 for loop，先透過 forward path 預測出結果，並利用 cross entropy 來計算出 loss。利用.backward( )自動幫忙計算梯度，然後根據梯度與 learning rate 來更新權重。完成一個 epochs 後，利用剛才定義好的 check_accuracy_part2( )來協助計算 validation accuracy。

- Barebones PyTorch: Train a Two-Layer Network

  利用先前建立好的 Two-Layer Network 來訓練模型。訓練一開始，我們需要先初始化參數。W 的部分利用 torch.nn.init.kaiming_normal_( )來初始化，b 則利用 torch.nn.zeros( )來做初始化。關於各參數的形狀，參考以下推導



執行結果如下

```
Iteration 0, loss = 3.8489
Checking accuracy on the val set
Got 96 / 1000 correct (9.60%)
Iteration 100, loss = 2.6742
Checking accuracy on the val set
Got 359 / 1000 correct (35.90%)
Iteration 200, loss = 2.1110
Checking accuracy on the val set
Got 391 / 1000 correct (39.10%)
Iteration 300, loss = 1.7854
Checking accuracy on the val set
Got 392 / 1000 correct (39.20%)
Iteration 400, loss = 1.6192
Checking accuracy on the val set
Got 423 / 1000 correct (42.30%)
Iteration 500, loss = 2.0057
Checking accuracy on the val set
Got 422 / 1000 correct (42.20%)
Iteration 600, loss = 1.8463
Checking accuracy on the val set
Got 429 / 1000 correct (42.90%)
Iteration 700, loss = 1.8102
Checking accuracy on the val set
Got 422 / 1000 correct (42.20%)
Iteration 765, loss = 1.5319
Checking accuracy on the val set
Got 414 / 1000 correct (41.40%)
```

可以看到在調整參數之前，準確度大約落在 42 %左右。

- Barebones PyTorch: Train a ConvNet

  這一部分要實作 initialize_three_layer_conv_part2 函數，這個函數主要是在對參數(W、b)做初始化。

  對於 W 的初始化，採用 Kaiming initialization，我們可以直接利用 PyTorch 幫我們定義好的函數 torch.nn.init.kaiming_normal_( )來完成。

  對於 b 的初始化，我們一律將他初始化為 0。

  關於 torch.nn.init.kaiming.normal_( )，他需要傳入一個 tensor，這題我們傳入一個空的 tensor。

  由於第一層 convolutional layer 的參數有 W1 和 b1，根據 kernel_size_1 和 channel 數量，W1 的 shape = (channel_1, C, kernel_size_1, kernel_size_1)，中間會加上一個 C 的主要原因在於輸入 X 有 rgb 三個顏色。b1 的 shape = (channel_1)。第二層 convolutional layer 的參數有 W2 和 b2，根據前一層的 shape，W2 的 shape = (channel_2, channel_1, kernel_size_2, kernel_size_2)，b2 的 shape = (channel_2)。最後還要通過一層全連接層，全連接層的參數有 fc_w 與 fc_b。由於有先經過 flatten 的關係，fc_w 的 shape = (num_classes, channel_2*H*W)，而 fc_b 的 shape = (num_classes)。

  執行結果如下

```
Iteration 0, loss = 2.6887
Checking accuracy on the val set
Got 94 / 1000 correct (9.40%)
Iteration 100, loss = 1.9734
Checking accuracy on the val set
Got 339 / 1000 correct (33.90%)
Iteration 200, loss = 1.8025
Checking accuracy on the val set
Got 395 / 1000 correct (39.50%)
Iteration 300, loss = 1.6301
Checking accuracy on the val set
Got 424 / 1000 correct (42.40%)
Iteration 400, loss = 1.6624
Checking accuracy on the val set
Got 439 / 1000 correct (43.90%)
Iteration 500, loss = 1.7439
Checking accuracy on the val set
Got 451 / 1000 correct (45.10%)
Iteration 600, loss = 1.5853
Checking accuracy on the val set
Got 470 / 1000 correct (47.00%)
Iteration 700, loss = 1.7783
Checking accuracy on the val set
Got 468 / 1000 correct (46.80%)
Iteration 765, loss = 1.3397
Checking accuracy on the val set
Got 455 / 1000 correct (45.50%)
```

由結果可以看到，準確度最高可以到 47%，比起前面訓練的 Two-Layer Network，加入 convolutional layer 後準確度提升大約 5%。

二、 PyTorch Module API

前面採用 barebone PyTorch 時，我們需要自行追蹤參數的 shape，這對於大型神經網路說是不可能的，因此 PyTorch 有提供 nn.Module API 來幫助我們自動追蹤所有參數。此外，在前兩次作中我們自行實作了許多 optimizer，在 PyTorch 中有提供 torch.optim 來讓我們不用每一次都要自己寫 optimizer。

- Module API: Two-Layer Network

  建立一個 Python 繼承類別名為 TwoLayerFC，該類別繼承 torch.nn.Module 的方法。

  TwoLayerFC 為一個兩層全連接層的神經網路。網路中參數 W 都是以 kaiming_normal 做初始化，b 則是初始化為 0。

  此外，雖然說 backward path 可以透過 PyTorch 中的 autograd 計算，但是 forward path 仍然需要自行定義，因此還需要寫一個函數來定義 forward path。

  呼叫 TwoLayerFC 的建構子並將建立好的模型顯示出來，結果如下

```
Architecture:
TwoLayerFC(
  (fc1): Linear(in_features=768, out_features=42, bias=True)
  (fc2): Linear(in_features=42, out_features=10, bias=True)
)
Output size: [64, 10]
```

- Module API: Three-Layer ConvNet

  定義一個 ThreeLayerConvNet 的繼承類別。

  ThreeLayerConvNet 為一個三層的神經網路，其中前兩層為 convolutional layer，最後一層為全連接層。第一層 convolutional layer channel size 為 channel_1、filter size 為 5*5、padding 為 2。第二層 convolutional layer channel size 為 channel_2，filter size 為 3*3、padding 為 1。最後一層全連接層，input features 為 channel_2*H*W(考慮到 flatten 的作用)、output features 為 num_classes。

  此外，跟 TwoLayerNet 相同的是，一樣要定義 forward path。

  呼叫 TwoLayerNet 的建構仔並將建立好的模型顯示出來，結果如下

```
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]
```

- Module API: Check Accuracy

  定義 check_accuracy_part34( )來幫助我們確認模型的準確度。此函數與在 Barebones PyTorch 時建立的 check_accuracy_part2( )大致上相同，差別只在於這次不需要手動傳入參數。

- Module API: Training Loop

  定義兩個函數，分別為 adjust_learning_rate( )和 train_part345( )來幫助我們訓練模型。

  adjust_learning_rate( )為調整 learning rate 的函數，他會根據使用者設定好的 schedule 來調整 learning rate，調整幅度為參數 lrd (learning rate decay)。

  train_part_345( )為訓練模型的函數。此函數會利用到剛才定義的 adjust_learning_rate( )來優化訓練過程。計算梯度時也是利用 PyTorch 的 autograd 來協助計算。與方才在 Barebones PyTorch 時定義的 training_loop 不一樣的是，這次我們不手動更新權重，而是利用 torch.optim 來更新權重。

- Module API: Train a Two-Layer Network

  利用剛才定義好的函數來訓練 Two-Layer Network。這次訓練我們不再需要自行分配參數 tensor，只需要傳入 input size、hidden layer size、number of class 即可。透過剛才定義好的 TwoLayerNet 類別來方便建立神經網路，並利用 torch.optim 來選取 optimizer(這邊使用 SGD)。訓練結果如下

```
Epoch 0, Iteration 200, loss = 2.1086
Checking accuracy on validation set
Got 335 / 1000 correct (33.50)

Epoch 0, Iteration 300, loss = 2.1708
Checking accuracy on validation set
Got 433 / 1000 correct (43.30)

Epoch 0, Iteration 400, loss = 1.9790
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)

Epoch 0, Iteration 500, loss = 1.7888
Checking accuracy on validation set
Got 449 / 1000 correct (44.90)

Epoch 0, Iteration 600, loss = 2.1020
Checking accuracy on validation set
Got 470 / 1000 correct (47.00)

Epoch 0, Iteration 700, loss = 1.6738
Checking accuracy on validation set
Got 475 / 1000 correct (47.50)

Epoch 0, Iteration 765, loss = 1.6793
Checking accuracy on validation set
Got 415 / 1000 correct (41.50)
```

最好的準確度落在 47.5%。

- Module API: Train a Three-Layer ConvNet

  利用剛才定義好的函數來訓練 Three-Layer ConvNet。

  這邊會用到 ThreeLayerConvNet 類別中的方法，

  initialize_three_layer_conv_part3( )。

  Initialize_three_layer_conv_part3( )為初始化模型的方法，該方法會建立一個 Three-Layer ConvNet 並將 optimizer 設定成 SGD。訓練結果如下

```
Epoch 0, Iteration 200, loss = 1.7803
Checking accuracy on validation set
Got 411 / 1000 correct (41.10)

Epoch 0, Iteration 300, loss = 1.7486
Checking accuracy on validation set
Got 460 / 1000 correct (46.00)

Epoch 0, Iteration 400, loss = 1.3971
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)

Epoch 0, Iteration 500, loss = 1.6607
Checking accuracy on validation set
Got 482 / 1000 correct (48.20)

Epoch 0, Iteration 600, loss = 1.3310
Checking accuracy on validation set
Got 486 / 1000 correct (48.60)

Epoch 0, Iteration 700, loss = 1.6023
Checking accuracy on validation set
Got 506 / 1000 correct (50.60)

Epoch 0, Iteration 765, loss = 1.5531
Checking accuracy on validation set
Got 499 / 1000 correct (49.90)
```

可以看到，最佳的準確度為 50.6%，比起 Two-Layer Network 高了 3.1%。

三、 PyTorch Sequential API

透過 PyTorch Module API 的協助，建立一個神經網路比起之前一個函數一個函數慢慢寫方便許多，但是我們仍然會需要去定義一個類別，並將建構子與 forward path 完成。

PyTorch 提供一個 module 叫做 torch.nn.Sequential，此 module 將剛才定義一個類別、建構子、forward path 這幾個步驟合併成一個。雖然說 torch.nn.Sequential 很方便，但是他只提供一些基本功能，更加複雜的功能仍然需要透過 PyTorch Module API 來完成。

- Sequential API: Two-Layer Network

  一開始先利用 nn.Sequential 將模型定義好。

  nn.Sequential 可以接收一個 ordered dictionary，dictionary 中從放 layer 的名字與 layer 的種類。nn.Sequential 會將 ordered dictionary 中的 layer

照著順率串接起來，成為一個神經網路。

定義好模型之後，利用 torch.optim 來選擇 optimizer，這邊使用 SGD。

利用剛才定義好的 train_part345( )來訓練模型，結果如下

```
Epoch 0, Iteration 200, loss = 1.7005
Checking accuracy on validation set
Got 408 / 1000 correct (40.80)

Epoch 0, Iteration 300, loss = 1.6148
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)

Epoch 0, Iteration 400, loss = 1.6338
Checking accuracy on validation set
Got 456 / 1000 correct (45.60)

Epoch 0, Iteration 500, loss = 1.5450
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)

Epoch 0, Iteration 600, loss = 1.4782
Checking accuracy on validation set
Got 474 / 1000 correct (47.40)

Epoch 0, Iteration 700, loss = 1.6625
Checking accuracy on validation set
Got 453 / 1000 correct (45.30)

Epoch 0, Iteration 765, loss = 1.2965
Checking accuracy on validation set
Got 455 / 1000 correct (45.50)
```

最佳的準確度為 47.4 %。

- Sequential API: Three-Layer ConvNet

  步驟基本上與 Two-Layer Network 相同，一開始先利用 nn.Sequential 將模型定義好，接著利用 torch.optim 來選擇 optimizer，這邊選擇使用 SGD。訓練結果如下

```
Epoch 0, Iteration 200, loss = 1.3689
Checking accuracy on validation set
Got 457 / 1000 correct (45.70)

Epoch 0, Iteration 300, loss = 1.8072
Checking accuracy on validation set
Got 483 / 1000 correct (48.30)

Epoch 0, Iteration 400, loss = 1.5623
Checking accuracy on validation set
Got 516 / 1000 correct (51.60)

Epoch 0, Iteration 500, loss = 1.4391
Checking accuracy on validation set
Got 514 / 1000 correct (51.40)

Epoch 0, Iteration 600, loss = 1.4313
Checking accuracy on validation set
Got 534 / 1000 correct (53.40)

Epoch 0, Iteration 700, loss = 1.4228
Checking accuracy on validation set
Got 546 / 1000 correct (54.60)

Epoch 0, Iteration 765, loss = 1.2679
Checking accuracy on validation set
Got 538 / 1000 correct (53.80)
```

最好的準確度為 54.6 %。

四、 額外嘗試

- 嘗試重新訓練 Two-Layer Network

  剛才訓練的 Two-Layer Network 都是未經調整參數訓練出來的結果，我嘗試迭代多組參數來尋找最佳結果。首先我將 optimizer 更改為 adam，並將訓練 epochs 提升到 10。超參數設置部分，hidden layer size 我嘗試[4000, 5000, 6000]、learning rate [1e-1, 1e-2, 1e-3, 1e-4]、weight decay [1e-3, 1e-4, 1e-5]。訓練後最佳結果發生在 hidden layer size = 4000、learning rate = 0.0001、weight_decay = 0.0001 時，準確度為 57.4 %，結果如下圖

```
training for hidden = 4000, lr = 0.0001, weight_decay = 0.0001
Architecture:
Sequential(
  (flatten): Flatten()
  (fc1): Linear(in_features=3072, out_features=4000, bias=True)
  (relu1): ReLU()
  (fc2): Linear(in_features=4000, out_features=10, bias=True)
)
```

```
Epoch 8, Iteration 6500, loss = 0.9701
Checking accuracy on validation set
Got 518 / 1000 correct (51.80)

Epoch 8, Iteration 6600, loss = 0.7962
Checking accuracy on validation set
Got 538 / 1000 correct (53.80)

Epoch 8, Iteration 6700, loss = 0.9810
Checking accuracy on validation set
Got 574 / 1000 correct (57.40)

Epoch 8, Iteration 6800, loss = 0.6367
Checking accuracy on validation set
Got 565 / 1000 correct (56.50)

Epoch 9, Iteration 6900, loss = 0.6024
Checking accuracy on validation set
Got 555 / 1000 correct (55.50)

Epoch 9, Iteration 7000, loss = 0.6301
Checking accuracy on validation set
Got 563 / 1000 correct (56.30)

Epoch 9, Iteration 7100, loss = 0.6675
Checking accuracy on validation set
Got 556 / 1000 correct (55.60)
```

- 嘗試重新訓練 Three-Layer ConvNet

  剛才訓練的 Two-Layer Network 都是未經調整參數訓練出來的結果，我嘗試迭代多組參數來尋找最佳結果。首先，我將 optimizer 調整為 adam，我嘗試使用不同 channel size、learning rate、weight decay，並將 epochs 設為 10 來做做訓練。訓練後，最佳結果發生在 learning rate = 0.0001、weight decay = 1e-5 時，神經網路架構如下

```
training for learning rate = 0.0001, weight_decay = 1e-05
Architecture:
Sequential(
  (conv1): Conv2d(3, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (relu): ReLU()
  (conv2): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (flatten): Flatten()
  (fc1): Linear(in_features=65536, out_features=10, bias=True)
)
```

訓練結果如下

```
Epoch 6, Iteration 4800, loss = 0.6632
Checking accuracy on validation set
Got 640 / 1000 correct (64.00)

Epoch 6, Iteration 4900, loss = 0.6883
Checking accuracy on validation set
Got 624 / 1000 correct (62.40)

Epoch 6, Iteration 5000, loss = 0.8925
Checking accuracy on validation set
Got 639 / 1000 correct (63.90)

Epoch 6, Iteration 5100, loss = 0.6751
Checking accuracy on validation set
Got 663 / 1000 correct (66.30)

Epoch 6, Iteration 5200, loss = 0.7822
Checking accuracy on validation set
Got 651 / 1000 correct (65.10)

Epoch 6, Iteration 5300, loss = 0.7978
Checking accuracy on validation set
Got 629 / 1000 correct (62.90)
```

最佳的準確度來到 66.3 %。

比較 Two-Layer Network 與 Three-Layer ConvNet (調整參數後結果)，Three-Layer Network 的準確度比 Two-Layer Network 高出 11.4 %，比起調整參數前的差距，可以說是差很多。

- 參數初始化

  這次作業中使用到 kaimimg initialization，以下為一些 initialization 的說明。

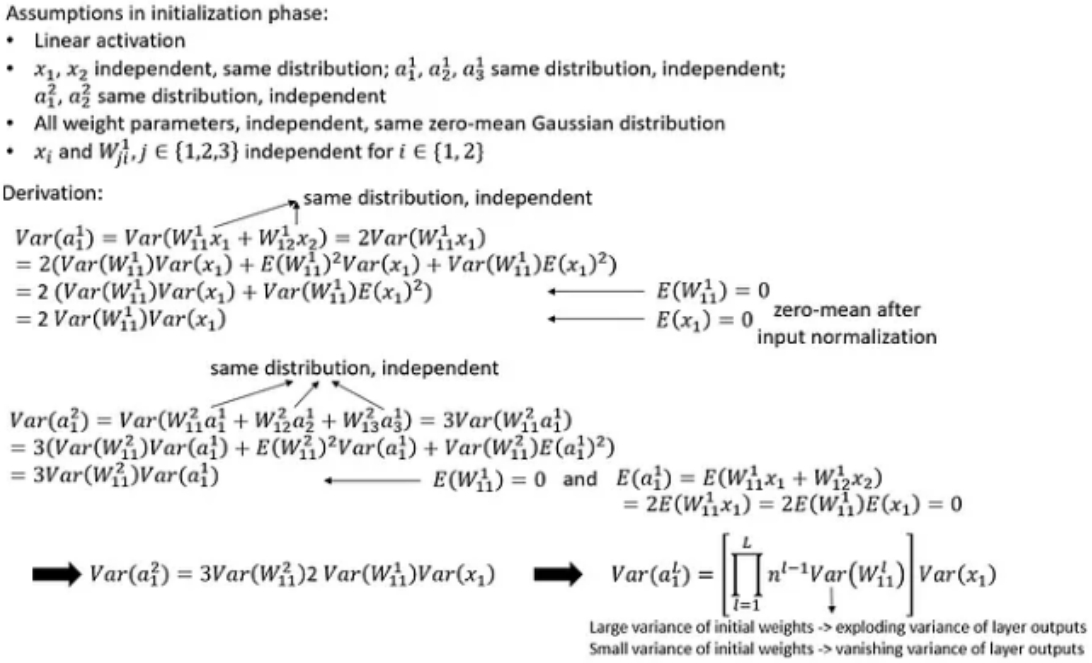
  1. Weight 初始化為 0 :

     如果初始化為 0，forward path 計算結果都會是 0。要注意的是只是輸出結果為 0，模型仍然會根據 back propagation 做學習。

  2. Random initialization :

     隨機給予權重初始值。Random initialization 的問題在於會根據高斯分布的變異數，造成神經網路中 layer output 的消失或是爆炸。

     數學推導如下



可以看到當初始化權重的變異數過大，會造成 layer output 的變異數也過大；當初始化權重變異數很小，layer output 的變異數也會很小。

3. Xavier initialization :

在 PyTorch 中，可以透過 nn.init.xavier_normal_來做 Xavier initialization。Xavier 的主要概念在保持輸入與輸出的變異數一致。

推導如下 (d 為輸入節點數量)

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$$

$$Var(y) = Var(w_1 x_1 + w_2 x_2 + \cdots + w_d x_d)$$

$$= Var(w_1 x_1) + Var(w_2 x_2) + \cdots + Var(w_d x_d) = \sum_i^d Var(w_i x_i)$$

$$Var(w_i x_i) = \mathrm{E}(x_i)^2 Var(w_i) + \mathrm{E}(w_i)^2 Var(x_i) + Var(w_i)Var(x_i)$$

假設 xi 與 wi 的平均數都是 0，且 idd，則

$$Var(w_i x_i) = Var(w_i)Var(x_i)$$

$$Var(y) = \sum_i^d Var(w_i x_i) = \sum_i^d Var(w_i)Var(x_i) = d \times Var(w_i)Var(x_i)$$

如果希望 Var(y) = 1，則

$$Var(y) = d \times Var(w_i)Var(x_i) = 1 \Rightarrow Var(w_i) = \frac{1}{d \times Var(x_i)}$$

$$Var(w_i) = \frac{1}{d} = \frac{1}{n_{inputnode}}$$

利用相同方式將 backward path 的部分也推導一次，可以得到

$$Var(w_i) = \frac{1}{n_{outputnode}}$$

為了要讓 forward path 與 backward path 的變異數相同，因此取

$$Var(w_i) = \frac{2}{n_{inputnode} + n_{outputnode}}$$

假設初始化權重的分布為 Uniform~(a, b)且 a = -b，計算其變異數要等於剛才推導結果

$$var(w_i) = \frac{(b-a)^2}{12} = \frac{2}{n_{inputnode} + n_{outputnode}}$$

$$\Rightarrow b^2 = \frac{6}{n_{inputnode} + n_{outputnode}}$$

$$\Rightarrow b = \sqrt{\frac{6}{n_{inputnode} + n_{outputnode}}}$$

$$a = -\sqrt{\frac{6}{n_{inputnode} + n_{outputnode}}}$$

$$w \sim U\left(-\sqrt{\frac{6}{n_{inputnode} + n_{outputnode}}}, \sqrt{\frac{6}{n_{inputnode} + n_{outputnode}}}\right)$$

最後可以得到初始化權重的分布。

4. He initialization (Kaiming initialization) :

在 PyTorch 中，可以透過 nn.init.kaiming_normal_來呼叫。

剛剛的 Xavier initialization 的問題在於使用 ReLU 作為 activation function 時，輸入輸出的變異數會有所不同。He initialization 解決了這個問題，公式推導如下

$$Var(y_l) = n_l \times Var(w_l)Var(x_l)$$

由於 yl - 1 是對稱 0 的分布，再加上使用 ReLU，所以

$$Var(x_l) = Var\left(f(y_{l-1})\right) = \frac{1}{2}Var(y_{l-1})$$

$$Var(y_l) = \frac{n_l}{2} \times Var(w_l)Var(y_{l-1})$$

照這樣的推理，將公式推導至第一層

$$Var(y_L) = \frac{n_L}{2} \times Var(w_L)Var(y_{L-1}) = \frac{n_L}{2} \times Var(w_L) \times \frac{n_{L-1}}{2}Var(w_{L-1})Var(y_{L-2})$$

$$= \cdots = Var(y_1)\left(\prod_{l=2}^{L}\frac{n_l}{2}Var(w_l)\right)$$

可以看到 Var(yl)為 Var(wl)作連乘，為了避免變異數的改變

$$\frac{n_l}{2}Var(w_l) = 1 \Longrightarrow Var(w_l) = \frac{2}{n_l}, \forall l$$

一樣去假設權重的分布為 Uniform~(a, b)，則根據權重之變異數

$$var(w_i) = \frac{(b-a)^2}{12} = \frac{2}{n_l} \Longrightarrow b^2 = \frac{6}{n_l} \Longrightarrow b = \sqrt{\frac{6}{n_l}}$$

$$a = -\sqrt{\frac{6}{n_l}}$$

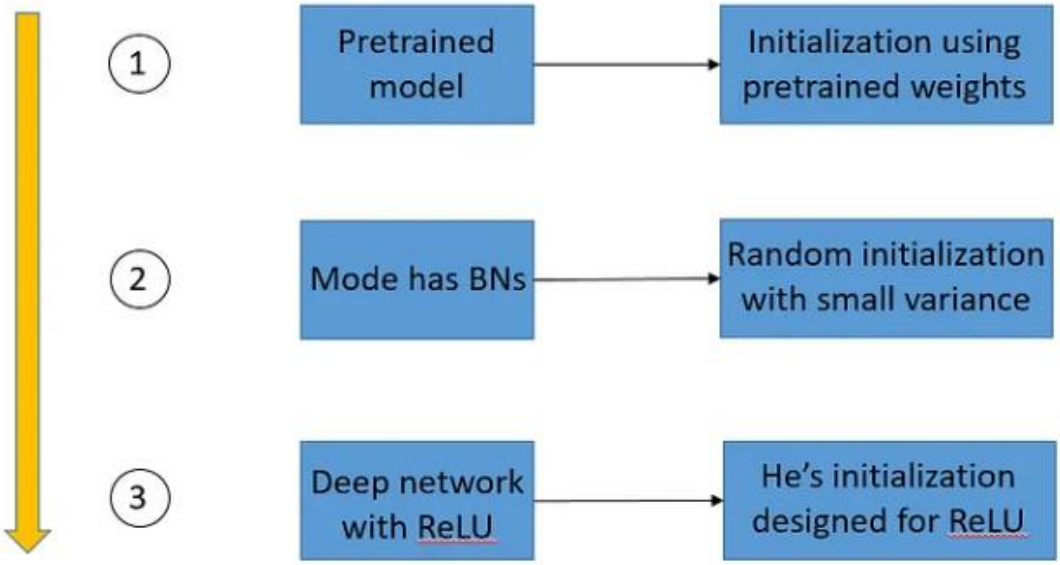
$$w \sim U\left(-\sqrt{\frac{6}{n_l}}, \sqrt{\frac{6}{n_l}}\right)$$

最後推導出初始化權重之分布。

5. Batch Normalization

   Batch normalization 在訓練過程中對 layer output 做 normalization，這樣一來就可以不用考慮如何對參數做初始化

6. 如何選擇這些方法



五、 Reference

[1] Tommy Haung "深度學習:Weight initialization 和 Batch Normalization" https://chih-sheng-huang821.medium.com/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92-weight-initialization%E5%92%8Cbatch-normalization-f264c4be37f5

[2] Huili Yu "Weight Initializaiton for Deep Neural Network" https://medium.com/@freshtechyy/weight-initialization-for-deep-neural-network-e0302b6f5bf3

[3] Wanderer001 "一文稿懂深度網路初始化 (Xavier and Kaiming Initialization" https://blog.csdn.net/weixin_36670529/article/details/104336598

[4] OpenAI. (2023). ChatGPT (Mar 14 version) [Large language model]. https://chat.openai.com/