CS5120 Homework Assignment #1

Name: 王領崧 Student ID: 107062107

1. Design Concept

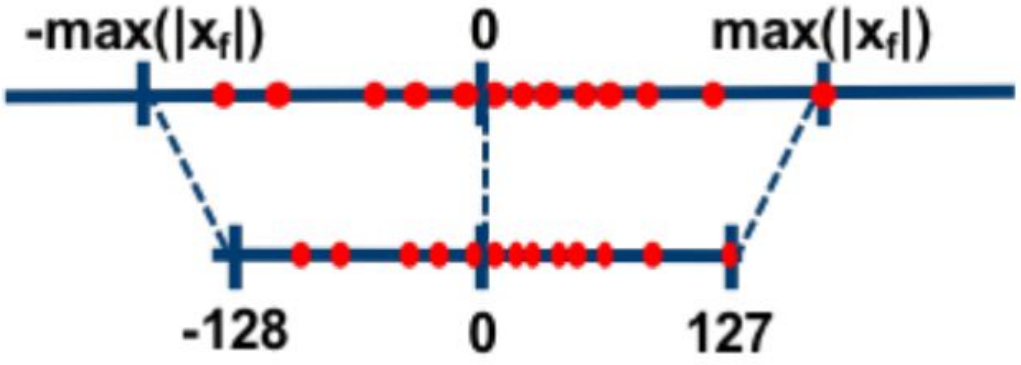
此次實作的 network 為 Lenet for CIFAR10 dataset。

Network Architecture :

Layer	Type	Input Shape	Output Shape	Activation fx
Conv1	Convolution	3*32*32	6*28*28	ReLU
Pool1	Pooling	6*28*28	6*14*14	X
Conv2	Convolution	6*14*14	16*10*10	ReLU
Pool2	Pooling	16*10*10	16*5*5	X
FC1	Fully-connected	400	120	ReLU
FC2	Fully-connected	120	84	ReLU
FC3	Fully-connected	84	10	X

1. quantized_weights()

quantized_weights function 用意為將 floating point weight quantize 成 8-bit integer point。這邊採用最容易實作的symmetric quantization，如下圖(圖片來源:上課講義)所示，先取出weights中的min, max值，接著比較兩者取絕對值後的大小，如果|max|比較大的話，則將range改為[-max, max]，反之則為[min, -min]。然後讓scale = 256 / range，作為兩者區間的比值。接著將weight * scale取round，最後再用clamp確保quantize後的weight都有落在[-128, 127]之間，就完成了。




```
max = torch.max(weights)
min = torch.min(weights)
if torch.abs(max) > torch.abs(min):
  min = max * -1
else:
  max = min * -1
scale = 128 / max
result = (weights * scale).round()
return torch.clamp(result, min=-128, max=127), scale
```

2. **quantize_initial_input()**
   quantize_inital_input function 用意為得到 initial input quantize scaling factor。
   與1.quantized_weights相同，採用 symmetric 的方式實作，不過因為只需要得到scaling factor，所以執行到 scale = 256 / range，計算出 scale 的大小即可。
3. **quantize_activations()**

$$W * I = O$$
$$n_{Wconv1} W_{conv1} * n_I I = n_{Wconv1} n_I O_{conv1}$$
$$n_{Wconv1} n_I O_{conv1} \rightarrow n_{Oconv1} * (n_{Wconv1} n_I O_{conv1})$$
$$n_{Wconv2} W_{conv2} * n_{Oconv1} n_{Wconv1} n_I O_{conv1} = n_{Wconv2} n_{Oconv1} n_{Wconv1} n_I O_{conv2}$$
$$n_{Wconv2} n_{Oconv1} n_{Wconv1} n_I O_{conv2} \rightarrow \dots$$

quantize_activations function 用意為計算出每一層 output activations 的 scaling factor。
完成這個 function 必須先探討一下 weight / activations 的數學式關係。
首先最原始的 convolution 為第一式，當我們要進行 quantization 的時候，因為 1. 和 2 .的關係，左邊會多兩個 scaling factor，一個是 weight，一個是 initial_input，所以在我們做 output activations quantization 之前，等式的右邊已經多了 n_wconv1 和 n_I 兩個 scaling factor，因此我們執行的 output activations quantization，其實不只是原本的 O_conv1，也包含了前面的兩個 factor (也就是第二個式子)。
接下來，當我們完成 quantization 後(第3個式子)，會產生 n_Oconv1 這個 output activations scaling factor。然後開始第2層 convolution 的運算，與上面相同的方式推導，不過要注意的是，這一層的 input activations (也就是上一層的 output)，它所帶的參數有3個，自己的 scaling factor、上一層的 weight scaling factor、initial_input scaling factor。
依照這個規律，要計算 scaling factor，需要先將前面所有層的 weight scaling factor、input scaling factor 與 output activations 相乘，再使用 symmetric 的方式計算其 scaling factor。

```
if not ns:
  output_scale = n_w * n_initial_input
  activations = np.multiply(activations, output_scale.cpu())
  max = torch.max(activations)
  min = torch.min(activations)
  if torch.abs(max) > torch.abs(min):
    min = max * -1
  else:
    max = min * -1
  scale = 128 / max
else:
  output_scale = n_w * n_initial_input
  for idx in range(len(ns)):
    output_scale = output_scale * ns[idx][0] * ns[idx][1]
  activations = np.multiply(activations, output_scale.cpu())
  max = torch.max(activations)
  min = torch.min(activations)

  if torch.abs(max) > torch.abs(min):
    min = max * -1
  else:
    max = min * -1
  scale = 128 / max
```

4. **forward()**

forward function 用意為 network 的 前向傳播。
由於 quantize_activations() 已經計算出每一層 activations scaling factor，因此我們需要在 forward 的階段，將每一層輸出的activation進行處理。
架構上與一開始提供的 forward function 相同，只是需要另外把上一步驟得到的scaling factor 都存進來，然後在每一層的 output activation，乘上相對應的 scaling factor 取 round 去做 quantization，並用 clamp function 確保數值落在 [-128, 127] 之間，就完成了。

```
# ADD YOUR CODE HERE
input_scale = self.input_scale
conv1_output_scale = self.conv1.output_scale
conv2_output_scale = self.conv2.output_scale
fc1_output_scale = self.fc1.output_scale
fc2_output_scale = self.fc2.output_scale
fc3_output_scale = self.fc3.output_scale

# input
x = (x * input_scale).round()
x = torch.clamp(x, min=-128, max=127)

# conv1
x = self.pool(F.relu(self.conv1(x)))
x = (x * conv1_output_scale).round()
x = torch.clamp(x, min=-128, max=127)
```

5. **quantized_bias**

$$W * I + \beta = O$$
$$n_W W * n_I I + \beta = n_O n_W n_I O$$

quantized_bias function 用意為 quantize 最後一層的 bias。
與 3. 所講述的概念類似，原本的數學式為第一式，當我們做完 weight / activations quantization 後，會得到第二個式子，但很顯然地，這個數學式左右並不相等，我們需要將 bias 也乘上 scaling factor (前面所有的 weight / activations factor)，才能讓這個式子平衡，這也是為什麼不做 bias quantization，accuracy 會有明顯地下降的原因。程式實作的部份與 3. 大同小異，只是這邊的 bias 要直接 quantize。

```
# ADD YOUR CODE HERE
scale = n_w * n_initial_input
for idx in range(len(ns)):
  scale = scale * ns[idx][0] * ns[idx][1]

return torch.clamp((bias * scale).round(), min=-2147483648, max=2147483647)
```

2. Simulation and Discussion

Accuracy 的分析比較

1) 尚未 quantize 的 CNN accuracy

```
Accuracy of the network on the test images: 52.07%
```

2) quantize 所有的 weights 之後的 accuracy

```
Accuracy of the network after quantizing all weights: 52.05%
```

3) quantize 所有的 weights / activations 之後的 accuracy

```
Accuracy of the network after quantizing both weights and activations: 52.11%
```

由上面的結果圖可以發現，進行 quantization 後對於 accuracy 的降低微乎其微，甚至有時候還會變高。

4) network with bias

```
Accuracy of the network (with a bias) on the test images: 52.2%
```

5) network with bias，quantize 所有的 weights / activations 之後的 accuracy

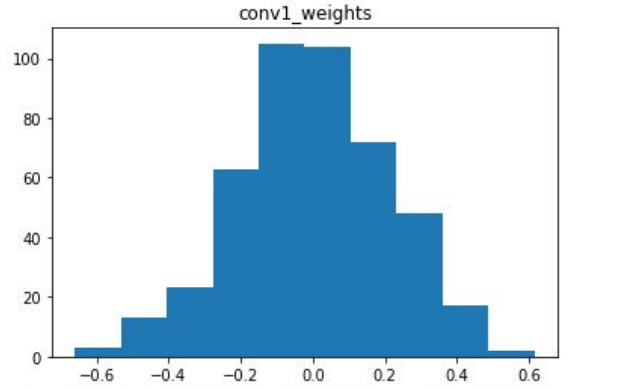
```
Accuracy of the network on the test images after all the weights are quantized but the bias isn't: 47.58%
```

6) network with bias，quantize 所有的 weights / activations / bias 之後的 accuracy

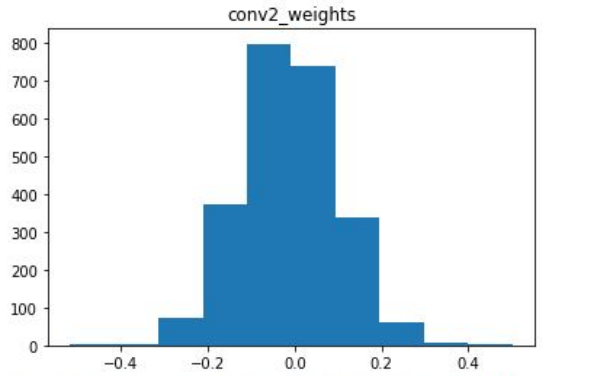
```
Accuracy of the network on the test images after all the weights and the bias are quantized: 52.12%
```

由 network with bias 的結果可以看到，accuracy 一樣沒什麼下降，而沒有做 bias quantization 導致 accuracy 明顯下降的原因，也於 quantized_bias function 有做說明。

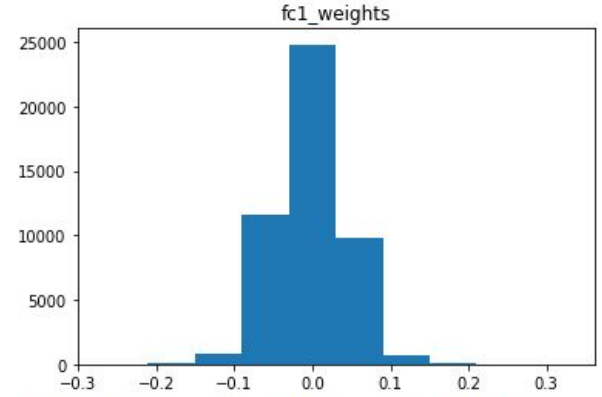
Visualizing weights



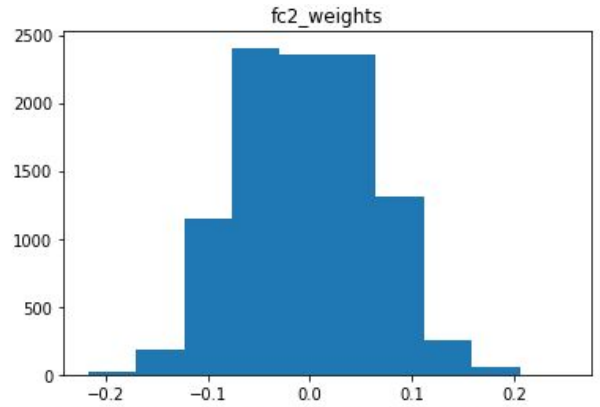
conv1_weights range : tensor(-0.6597) ~ tensor(0.6146)
conv1_weights mean : tensor(0.0049)
conv1_weights std : tensor(0.2071)
conv1_weights 3 sigma range : tensor(-0.6165) ~ tensor(0.6263)



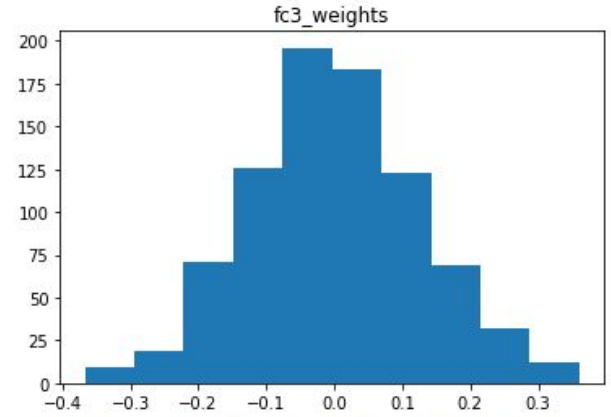
conv2_weights range : tensor(-0.5174) ~ tensor(0.5017)
conv2_weights mean : tensor(-0.0119)
conv2_weights std : tensor(0.1130)
conv2_weights 3 sigma range : tensor(-0.3509) ~ tensor(0.3272)



fc1_weights range : tensor(-0.2712) ~ tensor(0.3308)
fc1_weights mean : tensor(-0.0025)
fc1_weights std : tensor(0.0425)
fc1_weights 3 sigma range : tensor(-0.1301) ~ tensor(0.1250)



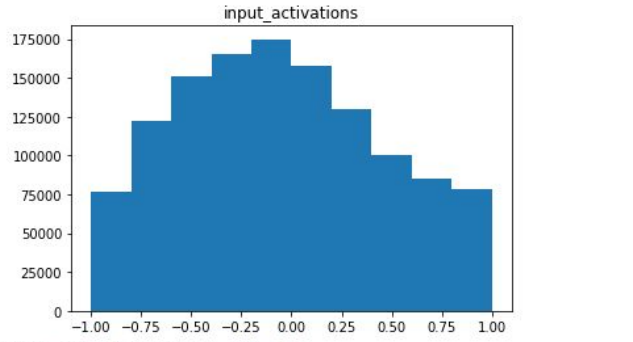
fc2_weights range : tensor(-0.2170) ~ tensor(0.2522)
fc2_weights mean : tensor(-0.0025)
fc2_weights std : tensor(0.0640)
fc2_weights 3 sigma range : tensor(-0.1945) ~ tensor(0.1894)



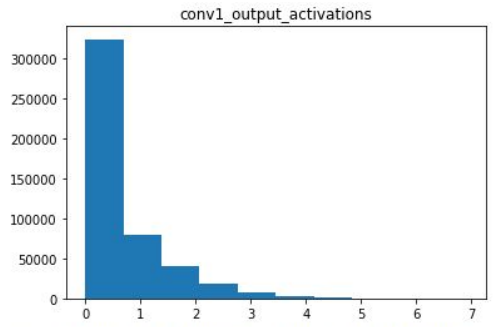
fc3_weights range : tensor(-0.3670) ~ tensor(0.3606)
fc3_weights mean : tensor(7.5531e-05)
fc3_weights std : tensor(0.1266)
fc3_weights 3 sigma range : tensor(-0.3799) ~ tensor(0.3800)

每一層 Layer 的 weight 都在 [-1,1]，平均值非常接近 0，大致都像是常態分佈一樣，也沒有嚴重的 outlier 問題，整體來看非常的不錯，不需要做什麼前處理。
5張圖表中，只有 fc3 的 range 是介在 +- 3 sigma range 之間，其餘的4個都在之外，顯示出另外4張圖中的 weight 仍然還是有一點點的 outlier 存在，但是因為 range 都在[-1,1]，算是可以接受，因此沒有做什麼特別的前處理。
最理想 quantize weights 的 range 應該要像 fc3 一樣，都介於 +- 3 sigma range 之間，因為如果 weight 的分佈是理想的常態分佈，那幾乎所有的點都會落在 +- 3 sigma range 之間，這樣在做 symmetric quantization 的時候，也不會因為特別有分佈偏左或是偏右，導致極大值或是極小值離真實的極大極小有一段距離(因為要對稱)，進而使得 quantization 的效果沒有那麼理想。

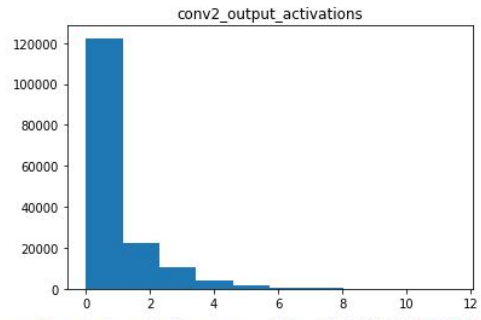
Visualizing activations



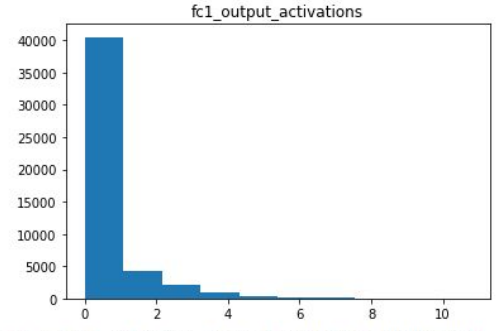
input_activations range : -1.0 ~ 1.0
input_activations 3-sigma : -1.5714041272897252 ~ 1.4711073681488838



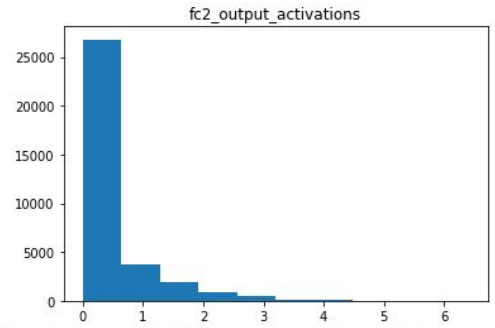
conv1_output_activations range : 0.0 ~ 6.912405967712402
conv1_output_activations 3-sigma : -1.7930922725343836 ~ 2.9546630911564598

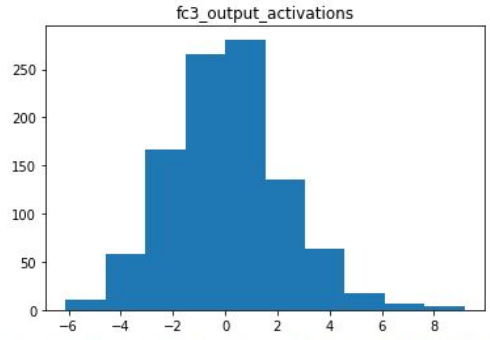


conv2_output_activations range : 0.0 ~ 11.487926483154297
conv2_output_activations 3-sigma : -2.8127516056472923 ~ 4.258603582635015



fc1_output_activations range : 0.0 ~ 10.769229888916016
fc1_output_activations 3-sigma : -2.622349285336012 ~ 3.5832922210044065





fc2_output_activations range : 0.0 ~ 6.400975704193115
fc2_output_activations 3-sigma : -1.6947405020183224 ~ 2.4234621909581273

fc3_output_activations range : -6.104842662811279 ~ 9.169123649597168
fc3_output_activations 3-sigma : -6.660977138419074 ~ 6.799198632918484

因為 ReLU activation 的關係，conv1, conv2, fc1, fc2 數值0的數量算是 dominate 整個分佈，另外兩張則是比較接近常態分佈，但是比起 weights 的分佈，range 相對來說都大了許多。
除了一開始的 input activation 的 range 有在+- 3 sigma range 之間，其餘的都沒有符合，而且像是 conv1, fc1，偏離的情況就蠻嚴重的，顯示出這些分佈存在著數據不平均 & outlier 的問題。
除了 input activations 使用 symmetric quantization 比較好之外，其他的 activations 理論上應該都要使用 asymmetric quantization 會比較好，因為這些 activations 受到 ReLU 的影響，不會有 < 0 的數值出現，而且 0 的數量會十分的多，導致數據很不平衡，如果使用的是 symmetric quantization 的話，有一半的 range 就會被浪費掉，造成原本圖的 range 被壓縮太多，效果就會沒那麼理想。
但在實作的過程中，發現全部都用 symmetric quantization 並沒有讓 accuracy 減少太多，因此最後就沒有改成實作上相對複雜的 asymmetric quantization。

遇到的困難

此次寫 lab 所遇到最大的困難就是在寫 quantize_activations 的部份，一開始一直覺得只要把前一層的 weight factor & input activations factor 和 output 相乘再做 quantize 即可，然而最後的 accuracy 卻一直是失敗的， 感謝後來與同學討論，從 initial_input 重新開始推導，才發現我一直忽略到 input activations factor 所乘的 input activations 是帶上前面的 factor，最後稍微修改 code，也就解決這個最棘手的問題了~

3. Summary

lab1 對整個 LeNet (simple CNN) 進行post-training quantization。 原本為floating point的運算，我們對 weight, activations, bias，一步一步地進行quantization，並在每一階段進行accuracy的分析，發現 accuracy 的下降幅度非常之小，幾次跑下來都不到 1%，有時候甚至會提高。在 training speed 方面，可能是 lab 提供的 network 不夠大，所跑的 epoch 也不算多，因此不太容易看出 training time 的減少，但以理論來看，速度應該會提升不少。
這次 lab 的難易度我覺得算適中，要實作的數學式也都不算太複雜，但是就像老師上課所說，有很多小細節需要去注意，我也因為一開始對這些小細節不夠了解，而遭遇了不少問題和困難，所幸與同學討論後，最後都能順利解決問題、完成作業。很開心能藉由這個 lab，對於 AI 加速的概念又有更進一步的了解與體會。