

Deep-Learning-for-Autonomous-Driving

Lab4 : Pruning & Quantization

ID : 310605007

Member : 鄭晴立

Department : 機器人學程

目錄

表目錄	2
找不到圖表目錄。	錯誤! 尚未定義書籤。
圖目錄	2
1. Task 1: Training.....	3
1.1. Result.....	3
1.2. Data augmentation.....	4
1.4. Dirty data.....	5
2. Task 2 : Fine-grained Pruning	7
2.1. Result.....	7
2.2. Method	8
2.3. Conclusion.....	8
3. Task 3 : Coarse-grained Pruning.....	9
3.1. Result.....	9
4. Task 4 : Post Training Static Quantization	10
5. Report	11
5.1. What are the pros and cons of fine-grained pruning and coarse-grained pruning? Which one is more hardware-friendly and why? (5%).....	11
5.2. Give your opinions on the benchmark table. (about speed and accuracy) (5%).....	11
5.3. Why do we fuse modules in quantization flow? (5%).....	11
5.4. In step 3 of post-training static quantization (PTSQ), we insert observers. What's the use of these observers? (5%)	12
5.5. We use PTSQ in task 5. What's the difference between PTSQ and post-training dynamic quantization? Which one is more hardware-friendly and why? (5%)	12
5.6. Other (10%).....	13

表目錄

表 一 clean data 及 dirty data 的 Accuracy 交叉比對.....	6
表 二 clean data 及 dirty data 的 Confusion Matrix 交叉比對	6

圖目錄

圖 一 最佳 model 的 Confusion Matrix.....	3
圖 二 訓練過程曲線	3
圖 三 Data Loader 匯入兩種資料	4
圖 四 利用 librosa 函式做 Data augmentation，(左圖)有提高及降低音頻，有增加雜訊。(右圖)有 8 種 augmentation 的方式，在 load data 時會隨機選擇，而每一種 augmentation 前面都會加上隨機不同程度的 noise.....	4
圖 五 40 個檔案的波型，其中包含部分錯誤檔案，但完全無法從中去判斷	5
圖 六 直接在 speech_command_dataset.py 檔裡面匯入新的 txt 檔	6
圖 七 model 每一層 layer 的 Sparsity table	7
圖 八 Testing Accuracy 和 Training Accuracy	7
圖 九 Testing Accuracy 和 Training Accuracy	9
圖 十 第一次移除 20%連接，得到[43,88,176,512]、91.57%的 Accuracy 及 337.4K 的 parameter	9
圖 十一 第二次移除 26%連接，得到[20,81,144,512]、92.1%的 Accuracy 及 269.4K 的 parameter	9
圖 十二 Quantization 公式.....	12
圖 十三 pytorch 官方文件中的 add_observer 函式	12
圖 十四 計算 register 中的變量	12

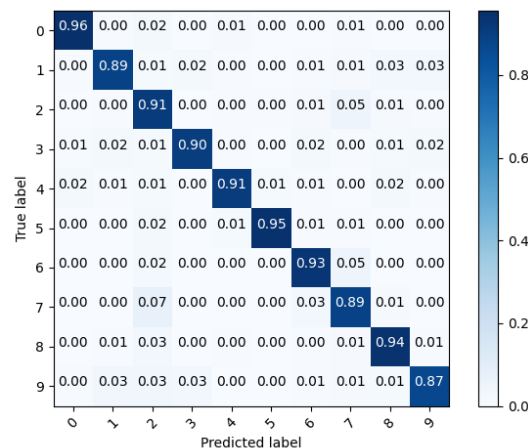
1. Task 1: Training

在本次的實驗中，我在 Training 過程做了兩個調整，一個是 Data augmentation，一個是刪除壞掉的音訊資料，最好的 model 可以到 **91.47%**，以下先展示本次的 training 結果。。

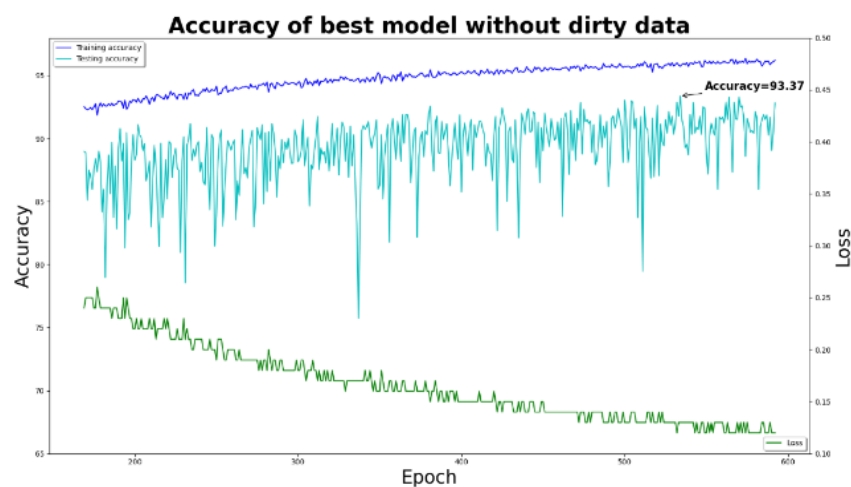
1.1. Result

圖一 是 model 的 Confusion Matrix，可以發現二類、第六類及第七類容易混淆，其分別是 up、on 和 off，其發音非常類似；而第零類及第八類，分別為 yes 和 stop，因為發與其他有巨大差別，所以分辨率特別高。

最高的 Testing Accuracy 達到 **91.47%**，下圖二 看到的 93.37% 是因為在訓練時，我將有問題的 data 刪掉後，所得到的數值。



圖一 最佳 model 的 Confusion Matrix



圖二 訓練過程曲線

1.2. Data augmentation

經過觀察，大多數的聲音檔噪音皆很大，有語調高低的不同、速度的不同。原本我想利用 torchaudio 函式中 sox_effects.apply_effects_tensor 去做處理，但其僅支援 linux 系統，於是我只能改用 librosa 函式做處理，其處理方式是將聲音檔轉為一維的 numpy，再將其作處理，最大的缺點就是他不在 GPU 運算，所以速度會慢很多。

如圖 三，我是直接在做 dataloader 的時候就匯入沒有 augmentation 及有 augmentation 的 data，等於直接將 training data 變成兩倍。

```
train_set = SpeechCommandDataset()
train_loader = DataLoader(train_set, **training_params)

train_set_aug = SpeechCommandDataset(aug=True)
train_loader_aug = DataLoader(train_set_aug, **training_params)

test_set = SpeechCommandDataset(is_training=False)
test_loader = DataLoader(test_set, **testing_params)
```

圖 三 Data Loader 匯入兩種資料

```
def pitchup(data):
    return librosa.effects.pitch_shift(data, sampling_rate, 1.25)
def pitchdown(data):
    return librosa.effects.pitch_shift(data, sampling_rate, -1.25)
def speedup(data):
    return librosa.effects.time_stretch(data, 1.15)
def speeddown(data):
    return librosa.effects.time_stretch(data, 0.85)
def shift(samples):
    y_shift = samples.copy()
    timeshift_fac = 0.3*2*(np.random.uniform()-0.5)
    start = int(y_shift.shape[0] * timeshift_fac)
    if (start > 0):
        y_shift = np.pad(y_shift, (start, 0), mode='constant')[0:y_shift.shape[0]]
    else:
        y_shift = np.pad(y_shift, (0, -start), mode='constant')[0:y_shift.shape[0]]
    return y_shift
def value(samples):
    y_aug = samples.copy()
    dyn_change = np.random.uniform(low=2, high=2)
    return y_aug * dyn_change
def noise(samples, a):
    y_noise = samples.copy()
    noise_amp = a*np.random.uniform()*np.amax(y_noise)
    y_noise = y_noise.astype('float64') + noise_amp * np.random.normal(size=y_noise.shape[0])
    return y_noise
def Stretching(samples):
    input_length = len(samples)
    streching = samples.copy()
    streching = librosa.effects.time_stretch(streching.astype('float'), 1.02)
    if len(streching) > input_length:
        streching = streching[:input_length]
    else:
        streching = np.pad(streching, (0, max(0, input_length - len(streching))), "constant")
    return streching
```

```
def aug(waveform):
    n=random.uniform(0.1,-0.1)
    a=random.randrange(1, 8)
    if a==1:
        waveform=noise(waveform,n)
        waveform=pitchup(waveform)
    elif a==2:
        waveform=noise(waveform,n)
        waveform=pitchdown(waveform)
    elif a==3:
        waveform=noise(waveform,n)
        waveform=speedup(waveform)
    elif a==4:
        waveform=noise(waveform,n)
        waveform=speeddown(waveform)
    elif a==5:
        waveform=noise(waveform,n)
        waveform=shift(waveform)
    elif a==6:
        waveform=noise(waveform,n)
        waveform=value(waveform)
    elif a==7:
        waveform=noise(waveform,n)
    else:
        waveform=noise(waveform,n)
        waveform=Stretching(waveform)
    return waveform
```

圖 四 利用 librosa 函式做 Data augmentation，(左圖)有提高及降低音頻，有增加雜訊。(右圖)有 8 種 augmentation 的方式，在 load data 時會隨機選擇，而每一種 augmentation 前面都會加上隨機不同程度的 noise

1.4. Dirty data

經過觀察，在 Data 裡面，有非常多音訊資料是有問題的，是人耳無法分辨的，包括幾種情況：

- 噪音太大，以至於完全聽不到正確發音
- 聽得懂在念什麼，但完全分類錯誤
- 聽不懂在念什麼，其發音完全無法分辨為該類別

我本來想利用波型去濾除錯誤檔案，但是我發現根本無法判斷，如圖五，因為每一種雜訊都太過隨機了，於是我利用第一次 train 出來的 91% 的 model 去找 Confusion Matrix，將每一次判斷錯的 training data 和 testing data 列出來，減少檔案數量後，再一個一個去聽，找出錯誤檔案，並列出正確的 test_list_new 及 train_list_new，然後直接在 speech_command_dataset.py 檔裡面匯入新的 txt 檔，如圖六所示，以下將有錯誤的資料稱為 dirty data，沒有錯誤的稱為 clean data。

從表一 可以看到我的訓練結果，Accuracy 並沒有預期中的提高，僅從 91.07% 提高到 91.47%，但如果我用 clean data 去 test 的話就會提高到 93.41%。



圖五 40 個檔案的波型，其中包含部分錯誤檔案，但完全無法從中去判斷

表一 clean data 及 dirty data 的 Accuracy 交叉比對

	用 dirty data 去訓練	用 clean data 去訓練
在 dirty data evaluation	Accuracy of best model:91.07%	Accuracy of best model:91.47%
在 clean data evaluation	-	Accuracy of best model:93.41%

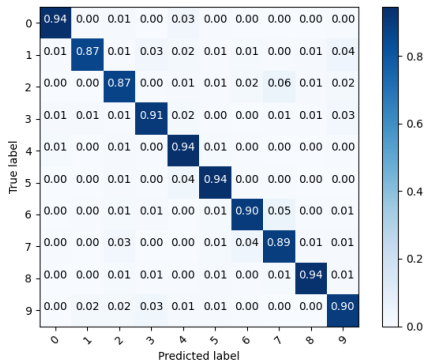
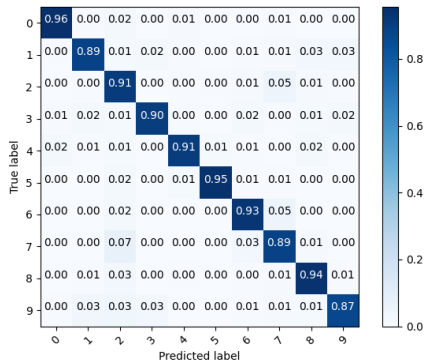
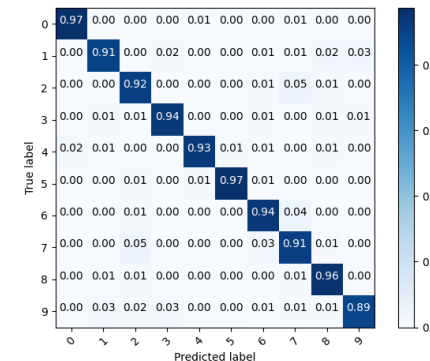
```

if is_training:
    self.data_list_path = os.path.join(
        self.data_path, 'train_list_new.txt')
else:
    self.data_list_path = os.path.join(self.data_path, 'test_list_new.txt')

```

圖六 直接在 speech_command_dataset.py 檔裡面匯入新的 txt 檔

表二 clean data 及 dirty data 的 Confusion Matrix 交叉比對

	用 dirty data 去訓練	用 clean data 去訓練
在 dirty data evaluation		
在 clean data evaluation	-	

2. Task 2 : Fine-grained Pruning

2.1. Result

從下圖 七 可以看到，Fine-grained 之前 Accuracy 是前面所說的 91.47%，sparsity 為 0；而 Fine-grained 之後 Accuracy 僅下降到 90.98%，而 sparsity 可以降到 68.96%。

從圖 八所示，我做了 3 次的 fine grained 第一次降到 85.62%、第二次降到 88.89%，後來發現都可以很順利 train 到 89% 以上，於是最後一次直接看到 70.48%。

Accuracy of best model:91.47%			
Modules	Parameters	zero number	Sparsity
features.0.weight	5120	0	0.0 %
features.0.bias	128	0	0.0 %
features.1.weight	128	0	0.0 %
features.1.bias	128	0	0.0 %
features.4.weight	49152	0	0.0 %
features.4.bias	128	0	0.0 %
features.5.weight	128	0	0.0 %
features.5.bias	128	0	0.0 %
features.8.weight	98304	0	0.0 %
features.8.bias	256	0	0.0 %
features.9.weight	256	0	0.0 %
features.9.bias	256	0	0.0 %
features.12.weight	393216	0	0.0 %
features.12.bias	512	0	0.0 %
features.13.weight	512	0	0.0 %
features.13.bias	512	0	0.0 %
fc.weight	5120	0	0.0 %
fc.bias	10	0	0.0 %
Total Trainable Params: 553994, Sparsity: 0.0 %			

Accuracy before pruning:90.98%			
Modules	Parameters	zero number	Sparsity
features.0.weight	5120	3584	70.0 %
features.0.bias	128	0	0.0 %
features.1.weight	128	0	0.0 %
features.1.bias	128	0	0.0 %
features.4.weight	49152	34406	70.0 %
features.4.bias	128	0	0.0 %
features.5.weight	128	0	0.0 %
features.5.bias	128	0	0.0 %
features.8.weight	98304	68813	70.0 %
features.8.bias	256	0	0.0 %
features.9.weight	256	0	0.0 %
features.9.bias	256	0	0.0 %
features.12.weight	393216	275251	70.0 %
features.12.bias	512	0	0.0 %
features.13.weight	512	0	0.0 %
features.13.bias	512	0	0.0 %
fc.weight	5120	0	0.0 %
fc.bias	10	0	0.0 %
Total Trainable Params: 553994, Sparsity: 68.96 %			

圖 七 model 每一層 layer 的 Sparsity table

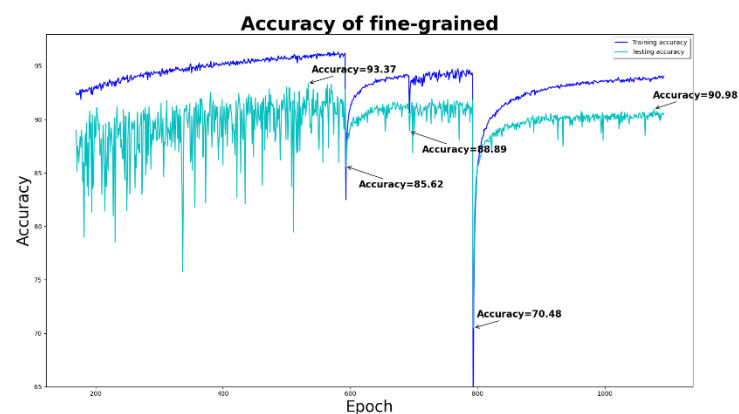


圖 八 Testing Accuracy 和 Training Accuracy

2.2. Method

首先，將原本的 model 做 deepcopy，避免更改到原本的 model，並將需要 Pruning 的 layer 取出來之後，將 weight 轉成 numpy 形式，再利用 numpy 的 percentile 函式取出絕對值最大的前幾百分位，將其作成一個 mask，最後套上原本的 weight 就完成了。

```
def Pruning_fg(model,p_pruning=60):
    model_fg = copy.deepcopy(model)
    length = len(list(model_fg.parameters()))
    mask_list=[]
    for i, param in enumerate(model_fg.parameters()):
        if len(param.size())==3:
            weight = param.detach().cpu().numpy()
            w_mask=np.abs(weight)<np.percentile(np.abs(weight),p_pruning)
            mask_list.append(w_mask)
            weight[w_mask] = 0
            weight = torch.from_numpy(weight).to(device)
            param.data = weight
    return model_fg,mask_list
```

2.3. Conclusion

從圖 八可以發現，當我 Pruning 50%~70% 讓 Training Accuracy 下降到 70%、80%之後，Train 第一個 epoch 就可以快速上升 90%以上，從這裡可以看出，model 其實是被少數的 weight 所支配。

3. Task 3 : Coarse-grained Pruning

3.1. Result

我做了兩次的 Coarse-grained Pruning，其方法就依照助教所提供的 code 去完成。第一次我弄掉 20% 的連接，得到[43,88,176,512]，而有 91.57% 的 Accuracy 和 337.46K 的 parameter：第二次我弄掉 26% 的連接，得到[20,81,144,512]，而有 92.1% 的 Accuracy 和 269.24K 的 parameter

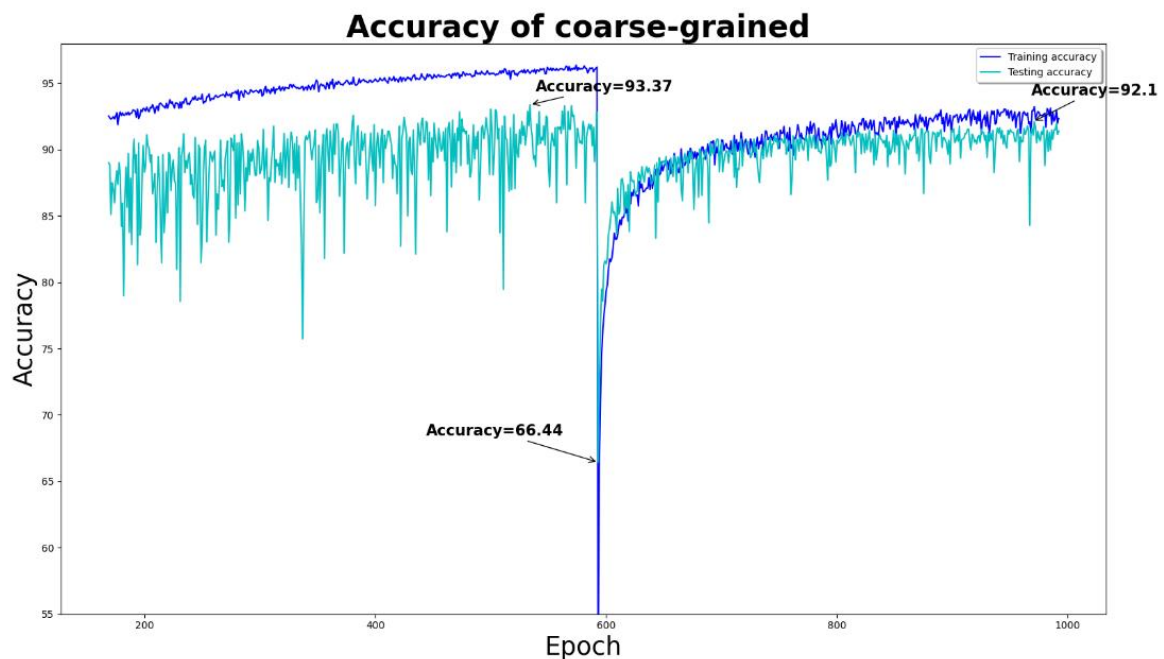


圖 九 Testing Accuracy 和 Training Accuracy

```
Accuracy of coarse grained pruning:91.57%  
Format of model : [43, 88, 176, 512]  
Number of coarse grained pruning: 337.46k
```

圖 十 第一次移除 20% 連接，得到[43,88,176,512]、91.57% 的 Accuracy 及 337.4K 的 parameter

```
Accuracy of coarse grained pruning:92.1%  
Format of model : [20, 81, 144, 512]  
Number of coarse grained pruning: 269.24k
```

圖 十一 第二次移除 26% 連接，得到[20,81,144,512]、92.1% 的 Accuracy 及 269.4K 的 parameter

4. Task 4 : Post Training Static Quantization

```
run_benchmark(quantized_model, NUM_BATCH) 💡
```

```
inference time: 8.134 s
```

```
run_benchmark(best_model, NUM_BATCH)
```

```
inference time: 150.174 s
```

```
run_benchmark(fine_model, NUM_BATCH)
```

```
inference time: 145.694 s
```

```
run_benchmark(coarse_model, NUM_BATCH)
```

```
inference time: 15.019 s
```

5. Report

5.1. What are the pros and cons of fine-grained pruning and coarse-grained pruning? Which one is more hardware-friendly and why? (5%)

在本次的實驗中 fine-grained 僅將部分權重調成 0，並沒有後續的壓縮或其他調整，在運算速度的提升是非常有限的，幾乎沒有提升。

而在 coarse-grained，直接設置 threshold 的方式，也會讓模型架構有點不平均，應該有更多的調整方式，但整體來說效果比 fine-grained 好很多。

5.2. Give your opinions on the benchmark table. (about speed and accuracy) (5%)

如上題所述，fine-grained 在沒有其他調整的情況下，運算速度很難有所提升，而 Quantized 是個非常有效率且簡單的調整方式，只是受限於 pytorch，在其他實作應用上，可能有些限制。

5.3. Why do we fuse modules in quantization flow? (5%)

如同助教在影片所說，pytorch 所提供的 quantization 並不支援 cuda，因此龐大的 torch 在 cpu 裡面會花很多時間去計算，因此 fuse_modules 能將多層 layer 的 model 融合成一個模塊，以提升它的運算速度，但官方文件有說他也可以提升精確度，但我就不太確定是為什麼。

5.4. In step 3 of post-training static quantization (PTSQ), we insert observers. What's the use of these observers? (5%)

Quantization 轉化的方式為圖 十二，需要找出 scale 及 zero-point。從官方文件可以看到，會先利用 `add_parameter` 去加入 observer，加入之後會再去計算 register 中的變量 `histogram`、`min_val`、`max_val`，透過這幾個參數去找出開頭提的 scale 及 zero-point，並計算最終 quantization 的值

$$Q(x, \text{scale}, \text{zero_point}) = \text{round}\left(\frac{x}{\text{scale}} + \text{zero_point}\right)$$

圖 十二 Quantization 公式

```
add_observer_(
    model, qconfig_propagation_list, observer_non_leaf_module_list,
    custom_module_class_mapping=custom_module_class_mapping)
return model
```

圖 十三 pytorch 官方文件中的 `add_observer` 函式

```
self.register_buffer("histogram", torch.zeros(self.bins, **factory_kwargs))
self.register_buffer("min_val", torch.tensor(float("inf"), **factory_kwargs))
self.register_buffer("max_val", torch.tensor(float("-inf"), **factory_kwargs))
```

圖 十四 計算 register 中的變量

5.5. We use PTSQ in task 5. What's the difference between PTSQ and post-training dynamic quantization? Which one is more hardware-friendly and why? (5%)

兩者都是將 `float32` 轉換為 `int8`，而 post-training dynamic quantization 只支援 `nn.Linear` and `nn.LSTM`，實際作用的 layer 比起 PTSQ 更少一些，效果也較差。

5.6. Other (10%)

- Training:

若時間夠的話，多做幾次 Confusion matrix 應該能把所有的錯誤資料挑出來，並且使用 torchaudio 函式做 data augmentation，Accuracy 和效率應該會好很多。

- Pruning:

fine-grained 和 coarse-grained，應該有更多的調整方式，提升 Pruning 的效率，但遺憾沒時間做。而我不太懂 fine-grained 中的 sparsity 的意義，實際上並沒有任何的實際效率的提升，或是 model 的輕量化。

- Quantization

看起來是非常有效率的輕量化選擇，但從 pytoech 的說明看來，目前限制蠻多的，不知道在實際應用上會不會有困難。