DSP Final Project Report

B07902054 資工三 林子權

Topic: Speech enhancement with Deep Autoencoder method

Motivatation

在生活步調非常快的現代社會之中，許多人在通勤之餘常常還得應付工作上的事情，甚至有時還得遠端通話開會。在通勤的環境下，常常都是十分吵雜的，要聽清楚對方說甚麼東西就非常的困難了，沒有辦法擁有良好的開會品質跟效率。因此如果能夠預先 train 好一個不錯的 speech enhancement 的 model，不論到時候要直接使用，或是要拿來做 real-time learning 的 initial model，都是不錯的工具。除了工作開會上的用途之外，speech enhancement 對 speech recognition 也能起到不錯的幫助，透過把使用者講出來的 speech 的 noise 降低，理論上可以有效地降低辨識錯誤率。

Data Preparing

為了要 train 一個 speech enhancement 的 model，首先需要非常大量的含有背景噪音的 speech data。因為網路上比較難找到大量的 noisy speech data，但 clean speech data 和 noise data 卻是還蠻容易就可以獲得，且網路上也有許多不錯的 dataset，所以我們可以蒐集大量的 clean speech data 和少數幾種特定的常見的 noise data，然後將他們合成在一起，就變成了不錯的 training data 了。

而且自己合成 noisy speech dataset 的好處是，這樣就可以同時有 noisy speech data 以及其對應的 clean speech data 跟 noise data，在訓練模型時，即可把 clean speech data 或是 noise data 當作 target 來訓練，希望 model 輸出的結果跟他們其中之一越接近越好。

Clean speech 我使用了下面 Reference 中的 LibriSpeech ASR corpus dataset 來取得，總共有 10 個小時的 clean speech。

Noise 我則是使用了下面 Reference 中的 ESC-50 dataset 來取得，我在其中選擇了 10 種生活中常見的 environment noise。

只選擇 10 種常見的 environment noise 是因為這樣比較容易 train 的起來，缺點就是對於那些和這 10 種 noise 相差比較多的 noise，去除的效果就會沒有那麼的顯著。所以選擇這 10 種 noise 其實是還蠻重要的一個步驟，要盡量選 10 種常見且聲音特性相差比較多的 noise。

這邊我選用以下 10 種 noise voice：

1. 嬰兒的哭聲
2. 敲鐘聲
3. 咳嗽聲
4. 烏鴉的叫聲
5. 蟲鳴聲
6. 笑聲
7. 流水聲
8. 雷聲
9. 車流聲
10. 火車過軌道聲

把蒐集來的 clean speech 和上面 10 種 noise speech 合成成一個長 11 小時的 noisy speech data，10 小時當作 training data，1 小時當作 validation data。這 10 小時的 noisy speech data 的 spectrogram 就是我們的 training data 的 input，其對應的 10 小時的 noise voice data 的 spectrogram 就是我們的 label。

合成的方法，是設定一個 frame size(我設為 1 秒左右)，在 clean speech 裡面 random sample 40000 個 frame，在 noise speech 裡面 random sample 40000 個 frame，然後把這 40000 個 clean speech frame 和 40000 個 noise speech frame 一一對應，每個 noise speech frame 要 random 乘上一個介於 0.2~1 之間的小數，然後再把他們合在一起，就成了 40000 個 random sample 的 noisy speech frame(合在一起的方法可以直接把兩個 audio 轉成 numpy array 後相加就好了)。

Data Preprocessing

跑完了上面 Data preparing 的 code 之後，我們會得到兩個長 40000 秒左右的 audio，一個聽起來會是一大堆長度只有 1 秒的英文句子背景夾雜 noise

拼湊而成的 noisy speech audio，一個聽起來會是一大堆長度只有 1 秒的 noise 拼湊而成的 noise audio。我們把這個長 40000 秒左右的 noisy speech audio 讀進來變成維度(40000 x frame_size)維的 matrix，然後再把這個 matrix 透過 short time furier transform 轉換成 40000 張 spectrogram。對 noise audio 也做一樣的處理。然後就得到了 40000 筆 training data，noisy speech spectrogram 是 input data，noise spectrogram 是 decoder 的輸出要去 minimize loss 的對象。會選用 noise spectrogram 而不是 clean speech spectrogram 做為 decoder 輸出 minimize 的對象，是因為 noise speech spectrogram 比較會有特定的 feature 可以去抓，clean speech spectrogram 則會因為講話的人跟講話的語句不同而有顯著的差別。最後答案就把 noisy speech spectrogram 減掉 decoder 輸出的結果即是預測的 clean speech spectrogram 了。

Reference

data preparing 以及 data preprocessing 的做法，我是依據下面 Reference 中第一個連結的文章來實作。clean speech data 還有 noise data 是我去 Reference 中第三跟第四個連結下載的。

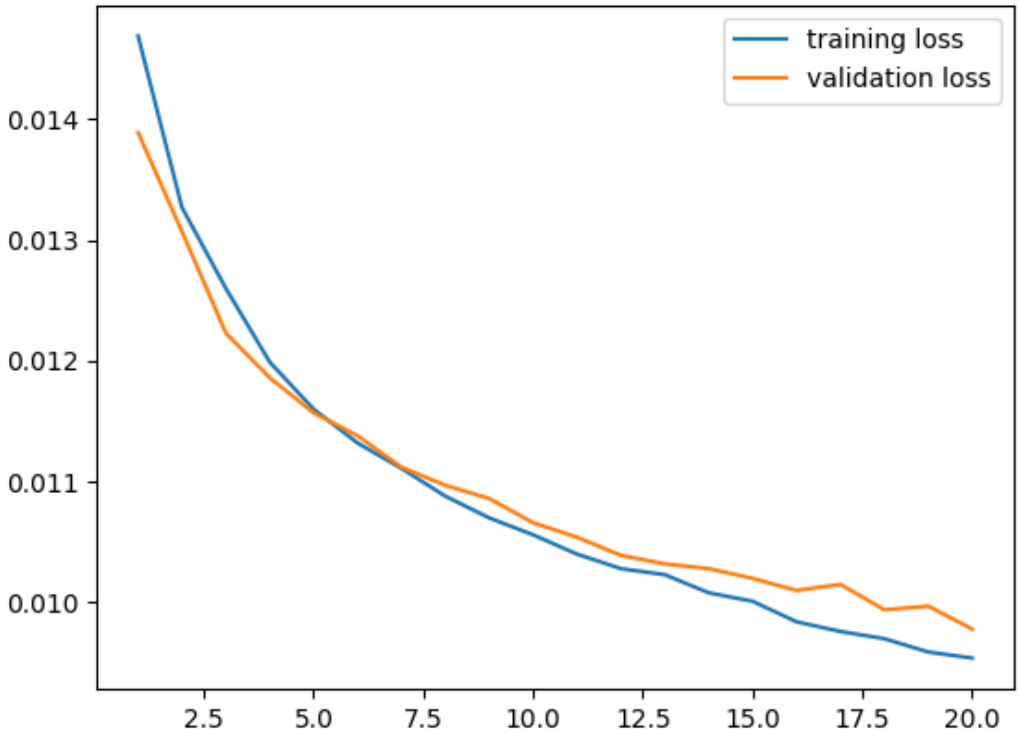
data preparing 以及 data preprocessing 的 code，寫在 /src/generate_data.py 裡面。我部分參考了下面 Reference 中第二個連結的 github，裡面有一些好用的 data preprocess 的函式可以使用。主要功用就是把 audio 轉成 numpy array 再經由 short time furier transform 轉成 spectrogram 等等。參考的 code 都集中在 data_tools.py 這一個檔案裡面區別開來。

Model structure that I have tried

Model 1: Deep Autoencoder

一開始我嘗試使用 encoder 有 10 層 convolution、decoder 有 5 層 deconvolution 的模型架構，input 為 noisy speech 的 spectrogram，output 為 noise 的 spectrogram，用 learning rate 為 1e-5，train 了 20 個 epoch，發現有嚴重的 underfitting 現象，loss 在 0.095 左右就開始慢慢地收斂的感覺。會發生這樣的現象，我認為很有可能是 decoder 還有 encoder 的架構太過於簡單。

用 predict.py 去測試 train 出來的 model，其實多少還是有 denoice 的效果，但在不同的 noice 上面效果有顯著的差別，在流水聲上的效果還可以，能去除掉 50%左右，但在鐘聲、交通車流聲等比較吵的 noice 上，感覺只有降低一些音量的感覺，還是可以清楚地聽到背景有吵雜聲。



這一個 model 我一樣有在每一個 convolution layer 後面加上 batch normalization layer。batch normalization 可以很好的解決 covariance shift 的問題，並且對於 training 速度的提升有很大的幫助。

model structure:

```
class AE(nn.Module):
    def __init__(self):
        super(AE, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 3, 3, stride=1, padding=1),    ## 1x128x128 ->
3x128x128
            nn.BatchNorm2d(3),
            nn.ReLU(True),
            nn.Conv2d(3, 3, 3, stride=1, padding=1),    ## 3x128x128 ->
3x128x128
            nn.BatchNorm2d(3),
```

```
            nn.ReLU(True),
            nn.MaxPool2d(2),                            ## 3x128x128 ->
3x64x64
            nn.Conv2d(3, 32, 3, stride=1, padding=1),   ## 3x64x64 ->
32x64x64
            nn.BatchNorm2d(32),
            nn.ReLU(True),
            nn.Conv2d(32, 32, 3, stride=1, padding=1),  ## 32x64x64 ->
32x64x64
            nn.BatchNorm2d(32),
            nn.ReLU(True),
            nn.MaxPool2d(2),                            ## 32x64x64 ->
32x32x32
            nn.Conv2d(32, 64, 3, stride=1, padding=1), ## 32x32x32 ->
64x32x32
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.Conv2d(64, 64, 3, stride=1, padding=1),## 64x32x32 ->
64x32x32
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.MaxPool2d(2),                            ## 64x32x32 ->
64x16x16
            nn.Conv2d(64, 128, 3, stride=1, padding=1), ## 64x16x16 ->
128x16x16
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),## 128x16x16 ->
128x16x16
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.MaxPool2d(2),                            ## 128x16x16 ->
128x8x8
            nn.Conv2d(128, 256, 3, stride=1, padding=1), ## 128x8x8 ->
256x8x8
            nn.BatchNorm2d(256),
            nn.ReLU(True),
```

```
            nn.Conv2d(256, 256, 3, stride=1, padding=1),## 256x8x8 ->
256x8x8
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.MaxPool2d(2),                            ## 256x8x8 ->
256x4x4
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 128, 5, stride=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 9, stride=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 32, 17, stride=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(32, 3, 33, stride=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(3, 1, 65, stride=1),
            nn.Tanh()
        )

    def forward(self, x):
        x1 = self.encoder(x)
        x  = self.decoder(x1)
        return x1, x
```

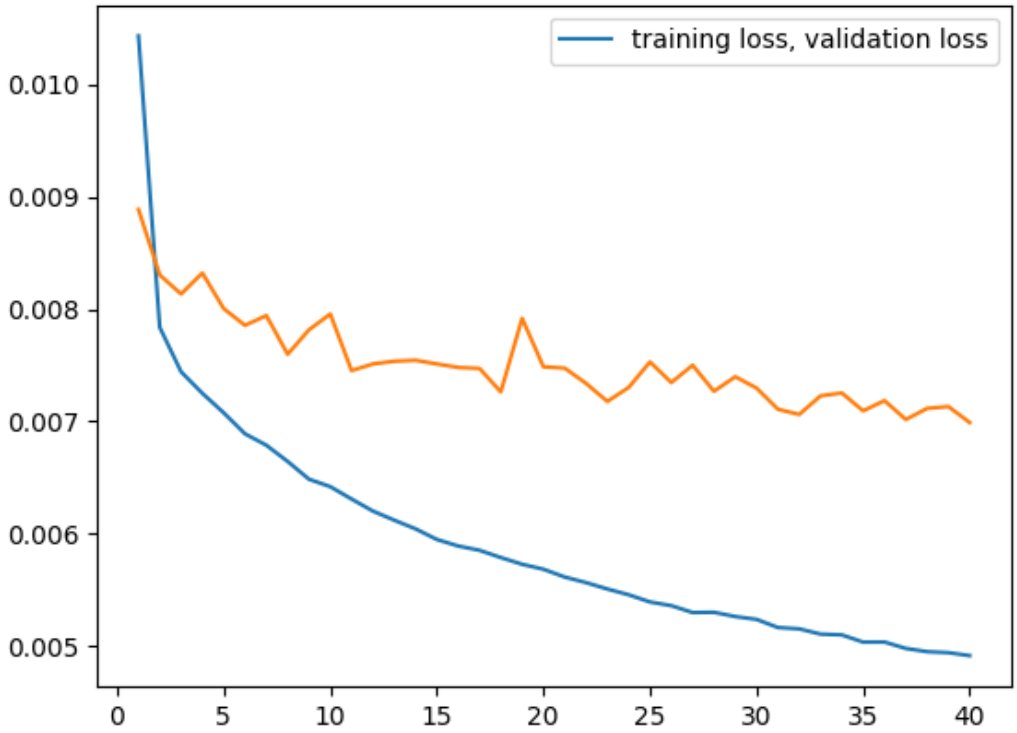
Model 2: Deep Autoencoder with Leaky ReLU activation function

跟第一個 model 的架構大致上是一樣的，只不過在 encoder 和 decoder 各加了一層 convolution layer 還有 deconvolution layer，分別是從 256 維 channel 到 512 維，以及從 512 維 channel 回去 256 維，這其實會讓 model 的參數多了將近 1 倍。和 model 1 一樣，在 decoder 的 convolution 後面都加上 batch normalization layer 來提速，但在我的機器上跑(Nvidia GeForce RTX 2070)，一個 epoch 還是要將近 25 分鐘。

和 Model 1 不一樣的地方還有，這個 model 的 activation function 我改成了 LeakyReLU function，LeakyReLU 能夠解決「Dead ReLU problem」，也就

是 ReLU 在 negative part 的值永遠都是 0，有時會導致 learning 受到阻礙。LeakyReLU 改良了這點，在其 negative part，會是一個離 0 很近的非 0 fraction。

這一次，我先將 learning rate 設為 1e-5，先讓他 train 20 個 epoch，測試看看在這樣的資料量之下，這樣複雜的 model 是否 train 的起來。結果發現其實可以，最後 training loss 大約落在 0.0075、validation loss 大約落在 0.0085 左右，和 Model 1 相比起來進步了一些。但從 loss 下降的趨勢來看，我觀察到此時已經有點逐漸收斂的感覺，可能是卡在了 local minimum 或是 saddle point 之類的，無法再下降。
所以，我接著嘗試把 learning rate 調高一些，看能不能在梯度下降時跳出那些 local minimum 或 saddle point。我把 learning rate 設為 1e-4，train 40 個 epoch，果然提升了不少 performance，training loss 可以降到 0.00520 且還沒有收斂的跡象、validation loss 可以降到 0.00699。雖然 training loss 可以降到很低，但是 validation loss 有點下不去，是典型的 overfitting 現象，解決此問題的最有效的方法應該是 generate 更多的 data 出來並且花更長的時間去 train。下圖的藍線是 training loss 的曲線、橘線是 validation loss 的曲線。



用 predict.py 測試過後，成果比我想像中好非常多，能夠很大程度去除掉背景的雜音了，只有少部分的 noise 去除不乾淨，例如說鐘聲，但即使是這樣還是有成功降低它對 speech 的影響。現在的問題變成，去除掉 noise 之後剩下人聲，但人聲的聲音質量受到了一些影響，有點悶悶糊糊的感覺，像是

蒙了一層東西在講話。其實會有這樣的結果也是可預期的，只要 noise 預測得不夠準確，差了一些些，原本的 speech spectrogram 就會受到影響。因為這個 model 的 decoder 做的事情是試著去還原出這段 voice 裡頭的 noise，再把 voice 減掉 noise 得到預測的 clean speech。因此只要 noise 差了一些，就會損害到原本的 clean speech。雖然不至於嚴重到聽不懂在說甚麼，但仍然可以嘗試看看把這個問題改善。

model structure:

```
class AE2(nn.Module):
    def __init__(self):
        super(AE2, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=1, padding=1),    ## 1x128x128 ->
16x128x128
            nn.BatchNorm2d(16),
            nn.LeakyReLU(True),
            nn.Conv2d(16, 16, 3, stride=1, padding=1),    ## 16x128x128
-> 16x128x128
            nn.BatchNorm2d(16),
            nn.LeakyReLU(True),
            nn.MaxPool2d(2),                            ## 16x128x128 ->
16x64x64
            nn.Conv2d(16, 32, 3, stride=1, padding=1),   ## 16x64x64 ->
32x64x64
            nn.BatchNorm2d(32),
            nn.LeakyReLU(True),
            nn.Conv2d(32, 32, 3, stride=1, padding=1),  ## 32x64x64 ->
32x64x64
            nn.BatchNorm2d(32),
            nn.LeakyReLU(True),
            nn.MaxPool2d(2),                            ## 32x64x64 ->
32x32x32
            nn.Conv2d(32, 64, 3, stride=1, padding=1), ## 32x32x32 ->
64x32x32
            nn.BatchNorm2d(64),
            nn.LeakyReLU(True),
            nn.Conv2d(64, 64, 3, stride=1, padding=1),## 64x32x32 ->
64x32x32
```

```
            nn.BatchNorm2d(64),
            nn.LeakyReLU(True),
            nn.MaxPool2d(2),                            ## 64x32x32 ->
64x16x16
            nn.Conv2d(64, 128, 3, stride=1, padding=1), ## 64x16x16 ->
128x16x16
            nn.BatchNorm2d(128),
            nn.LeakyReLU(True),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),## 128x16x16 ->
128x16x16
            nn.BatchNorm2d(128),
            nn.LeakyReLU(True),
            nn.MaxPool2d(2),                            ## 128x16x16 ->
128x8x8
            nn.Conv2d(128, 256, 3, stride=1, padding=1), ## 128x8x8 ->
256x8x8
            nn.BatchNorm2d(256),
            nn.LeakyReLU(True),
            nn.Conv2d(256, 256, 3, stride=1, padding=1),## 256x8x8 ->
256x8x8
            nn.BatchNorm2d(256),
            nn.LeakyReLU(True),
            nn.MaxPool2d(2),                             ## 256x8x8 ->
256x4x4
            nn.Conv2d(256, 512, 3, stride=1, padding=1), ## 256x4x4 ->
512x4x4
            nn.BatchNorm2d(512),
            nn.LeakyReLU(True),
            nn.MaxPool2d(2),                            ## 512x4x4 ->
512x2x2
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(512, 256, 3, stride=1),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(256, 128, 5, stride=1),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(128, 64, 9, stride=1),
            nn.LeakyReLU(True),
```

```
            nn.ConvTranspose2d(64, 32, 17, stride=1),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(32, 16, 33, stride=1),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(16, 1, 65, stride=1),
            nn.LeakyReLU(True),
            nn.Tanh()
        )

    def forward(self, x):
        x1 = self.encoder(x)
        x  = self.decoder(x1)
        return x1, x
```

以上 training 的 code，我寫在/src/train.py 以及/src/train_tools.py

Experiment

這一部分會針對上面所提及的 Model2，對一些我覺得有趣的現象跟議題進行實驗

對於其它不同的 **noise** 的降噪能力

這個部分我自己預期不會太好，因為從上面的訓練曲線來看，有 overfitting 的現象發生，所以在那些不在我的 training data 的 noise 上，預測的效果會更差一點。不過當然還是取決於 noise 的種類，如果是那些相似的 noise，或許結果會好一點。

我一樣從下面 Reference 的 ESC-50 dataset 找了四種不一樣的 noise，分別是：

1. 鴨子叫聲
2. 公雞叫聲
3. 火焰燃燒聲
4. 打字聲

實驗結果顯示，其中只有火焰燃燒聲的 denoise 效果稍微好一點，但跟上面 10 種 noise 在 Model2 的表現比起來，完全沒有辦法相比。

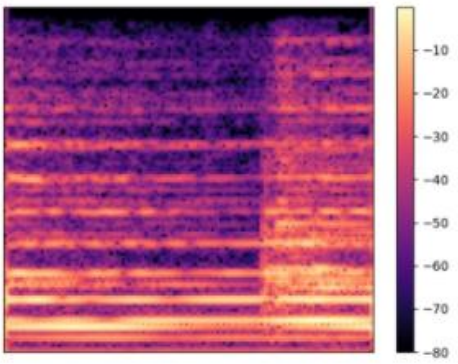
所以如果想要把這項研究推展到現實世界可以使用，必須要收集非常多種類的 noise voice，train 一個很大的 model。並且在使用時最好也能隨著使用者的使用過程，一邊進行 real time 的 training。

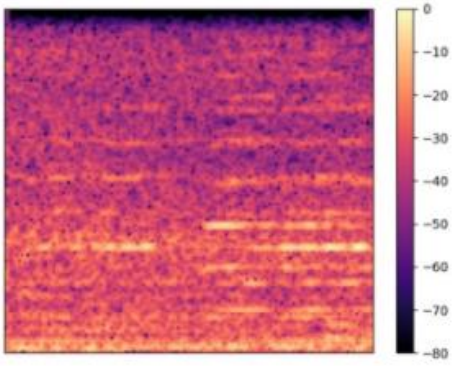
或是可以設計針對不同場合使用的 model：在辦公室使用的就針對打字聲、談話聲、寫字聲等 noise 來進行 training；在交通繁忙的街上使用的，就針對車流聲、腳步聲、喇叭聲等 noise 來進行 training。這樣子就可以不太需要擔心 overfitting 的問題，缺點是 program 使用上功能比較侷限。

為何敲鐘聲的 denoise 效果是最差的

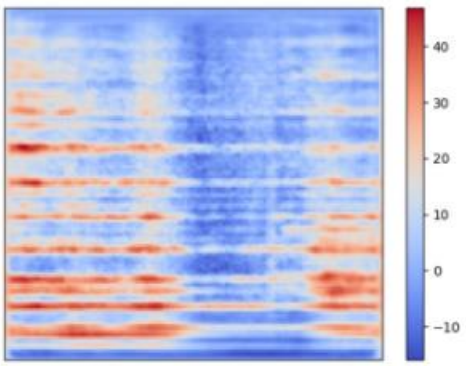
同上面所說，10 種 noise 裡面，敲鐘聲的 denoise 效果是唯一比較沒那麼好的，其它 noise 都能很好的去除掉。

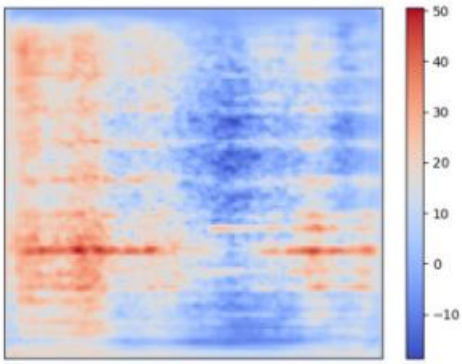
我取了一秒的敲鐘聲還有一秒的車流聲當作對照組，轉成 spectrogram 並輸出成 image，希望能透過分析頻譜圖來了解甚麼樣的 noise 是相對比較難去除掉的，先來看這兩個 noise 的 spectrogram。左邊是敲鐘聲，右邊是車流聲：





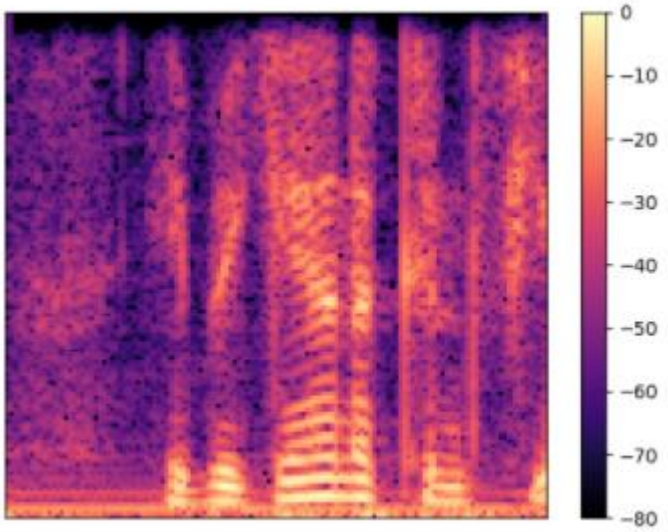
以及 Model2 根據兩者和同一段 clean speech 混合的 noisy voice，輸出的預測的 noise。左邊是敲鐘聲，右邊是車流聲：



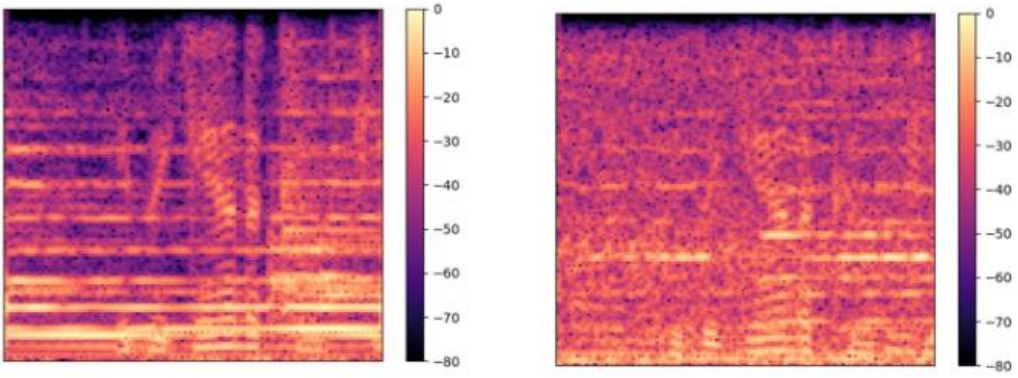


可以看出敲鐘聲的預測結果其實是有成功把低頻跟高頻的 noise 去除掉，但中高頻有一個區段幾乎沒有 predict 出來。反觀車流聲的預測結果，有成功的把分貝最高的 noise predict 出來，並且其它比較沒那麼明顯的 feautre 它也都有抓到那個樣子。

再來，看一下和他們混合在一起的 clean speech 的 spectrogram：



以及他們混合在一起後的 noisy speech 的 spectrogram。左邊是敲鐘聲的，右邊是車流聲的：



和他們混合的 clean speech spectrogram 恰好是中高頻區段比較集中的 spectrogram。並且我觀察了其它一些我 dataset 裡面的 clean speech spectrogram，大部分都有高比例的中高頻，我覺得可能是因為我找的 clean speech data 是朗誦的那種 speech，人在朗誦的時候音調都會不自覺的提高。

而觀察敲鐘聲 noise 的 spectrogram 可以發現，它在中高低頻都有很高分貝的地方，且大致上呈現一條一條的橫線。也許就是它在中高頻的 feature 和 clean speech 的 feature 混合在一起了，導致 decoder 在 decode 的時候無

法將中高頻的 feature decode 回去 noise 的 spectrogram。但這只是我個人的猜測，實際上為何會這樣我沒有一個很好的答案。因為直觀上來說，以人類的眼睛來判斷，我反而覺得敲鐘聲的 noisy speech spectrogram 比較好分辨出哪些地方是敲鐘聲那些是 speech。

以上 experiment 的 code 我放在/src/generate_experiment_data.py 以及 /src/predict_experiment.py。

Conclusion

這次的 project 我學到了許多處理語音 data 的方式，以及了解到調參數的重要性，learning rate 稍微差一些都可能造成巨大的改變。demo 的結果看起來是還不錯的，但是這是建立在我取用的 noise data 不多的情況之下，實際上如果要放在生活上應用，除非是在那種特定的場合，針對少數幾種 noise 下去 training，要不然只要 noise 的種類太多太複雜，我相信要有好的 performance 就得再花一些功夫去調整我的 model。

Reference

- Speech Enhancement with Deep Learning
- Speech Enhancement with Deep Learning Github
- LibriSpeech ASR corpus
- ESC-50