學號：B06901053　系級：電機三　姓名：謝承延

1. (20%) Teacher Forcing:

a. 請嘗試移除 Teacher Forcing，並分析結果。

以下有/無 Teacher Forcing 都選用相同的參數設定

參數設定:

Batch size =60

Embedding dim 為 256

GRU 中的 hidden dimension 為 512

GRU 層數為 3(Encoder 為 Bi-GRU)

Decoder input 前及 FC 的部分設置 dropout =0.5

Learning rate 為 0.00005

Optimizer: Adam

Maximum output length 為 50

Train 12000 次 每 300 次記錄一次 loss 跟 BLEU 變化

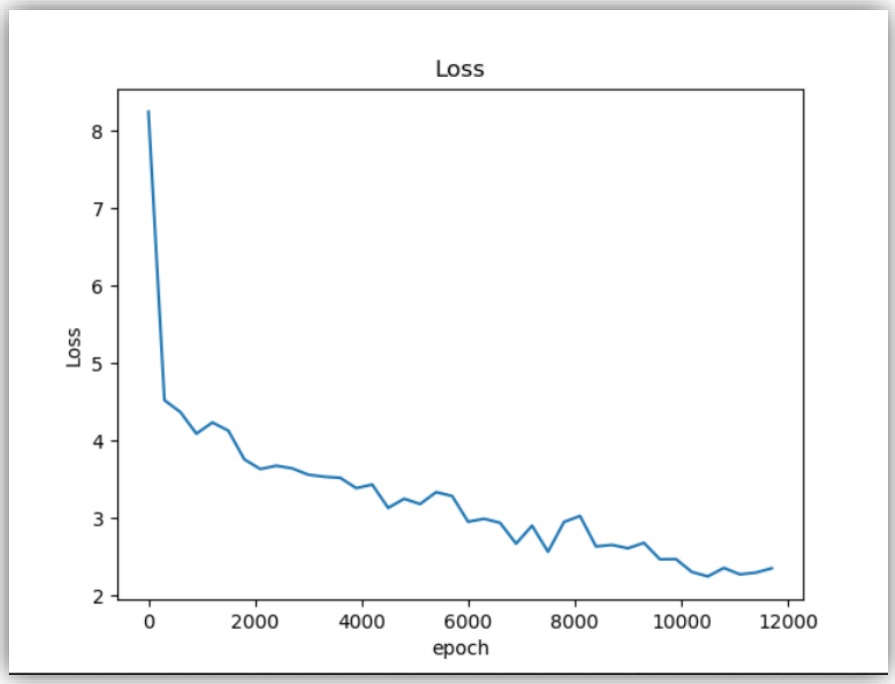
Loss function 為 cross entropy

```
self.batch_size = 60
self.emb_dim = 256
self.hid_dim = 512
self.n_layers = 3
self.dropout = 0.5
self.learning_rate = 0.00005
self.max_output_len = 50              # 最後輸出句子的最大長度
self.num_steps = 12000              # 總訓練次數
self.store_steps = 300                # 訓練多少次後須儲存模型
self.summary_steps = 300              # 訓練多少次後須檢驗是否有overfitting
self.load_model = False               # 是否需載入模型
self.store_model_path = "./ckpt"      # 儲存模型的位置
# 載入模型的位置 e.g. "./ckpt/model_{step}"
self.load_model_path = './ckpt/model_TF'
self.data_path = sys.argv[1]          # 資料存放的位置
self.attention = False                # 是否使用 Attention Mechanism
```

Case1: 移除 teacher forcing（根據自己預測結果做預測）

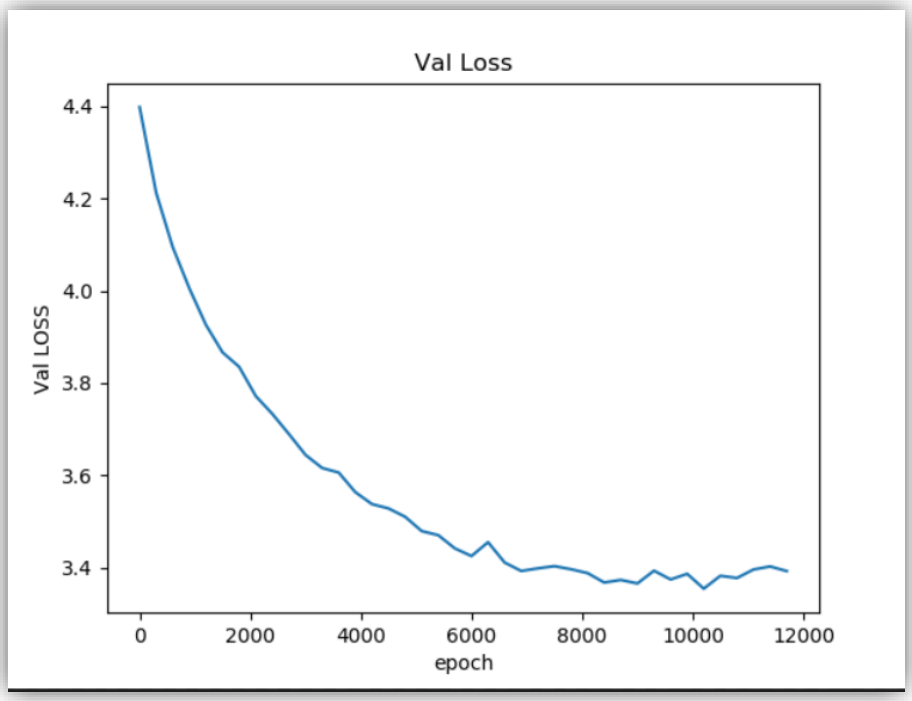
Train LOSS:

最終大約收斂於 2.3 的位置



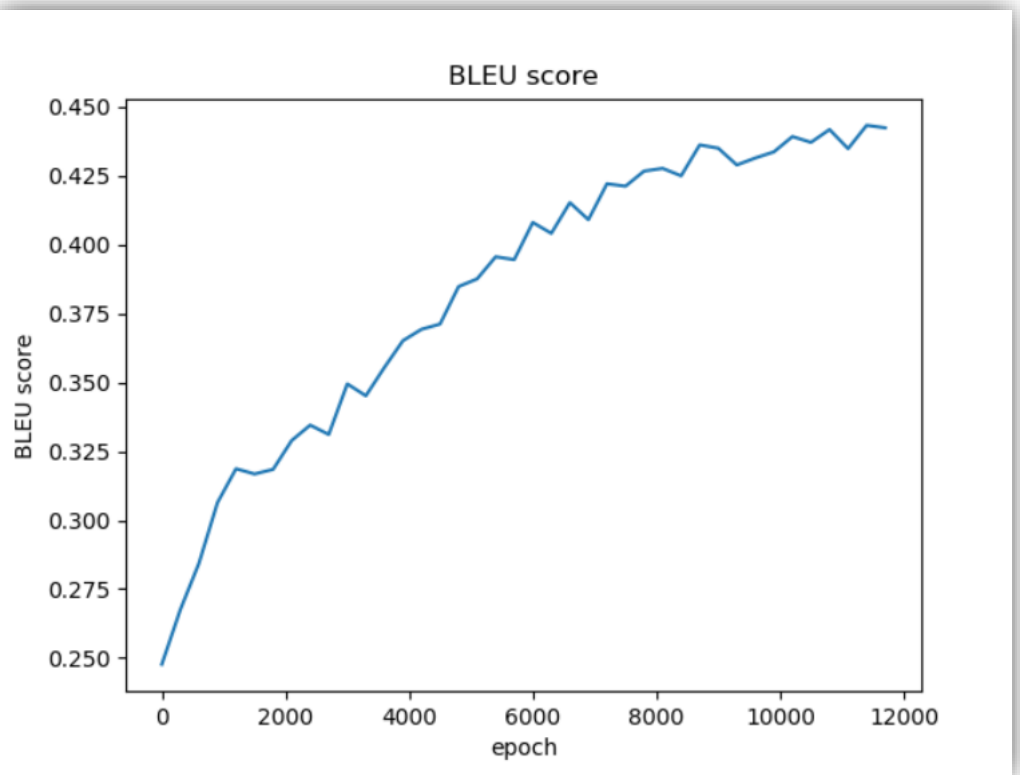
VAL_LOSS:

最終大約收斂於 3.4 的位置



VAL BLEU SCORE:

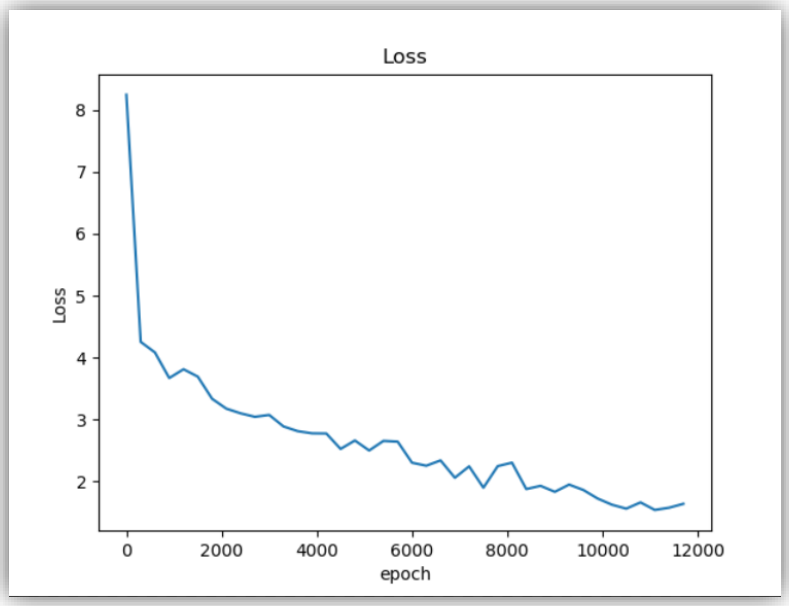
最後 1000 個 epoch score 都大致收斂於 0.43(因圖片 300 個 epoch 記錄一次 所以收斂看起來不明顯)



Case2 : 使用 teacher forcing(用正確 reference 做預測)

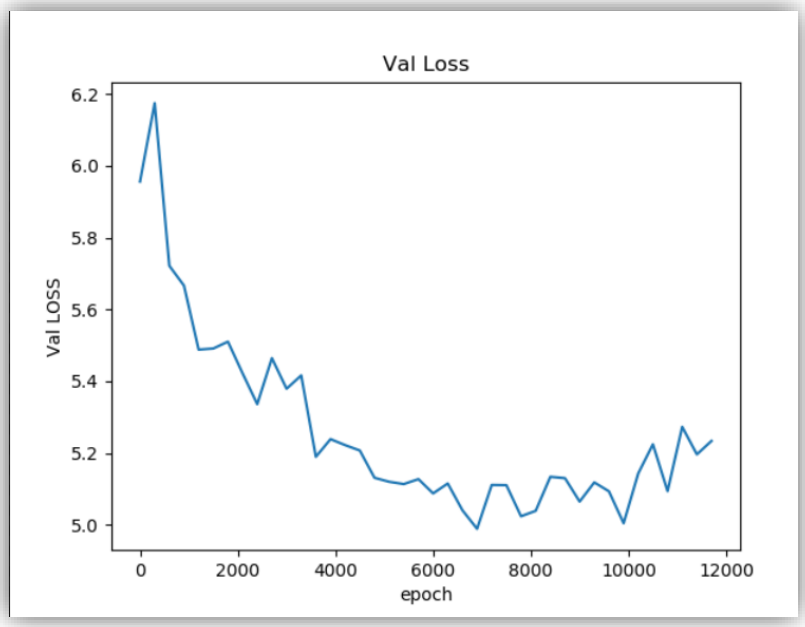
Train LOSS:

最終大約收斂於 1.45 的位置



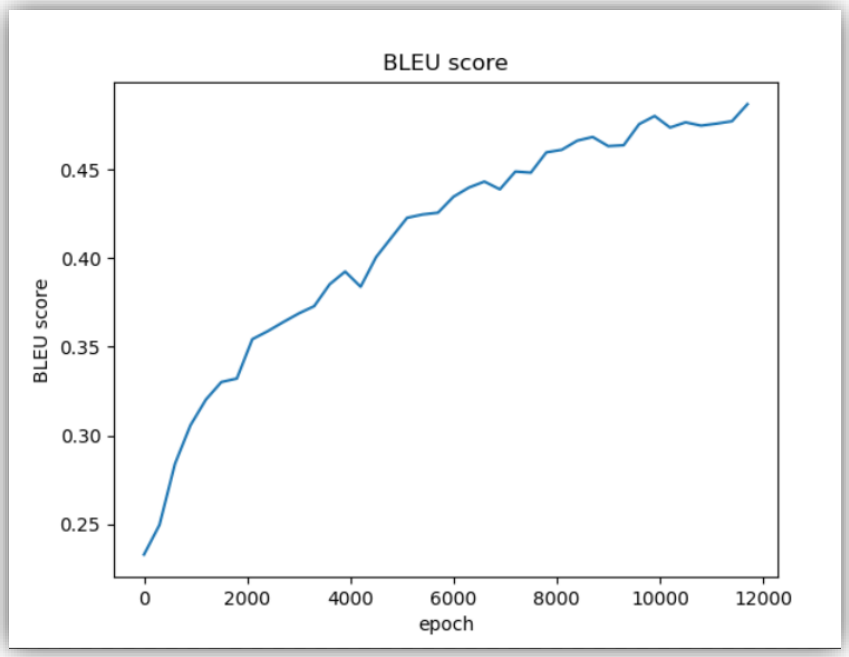
VAL_LOSS:

有些 overfitting 的現象，epoch8000 逐漸上升



VAL BLEU SCORE:

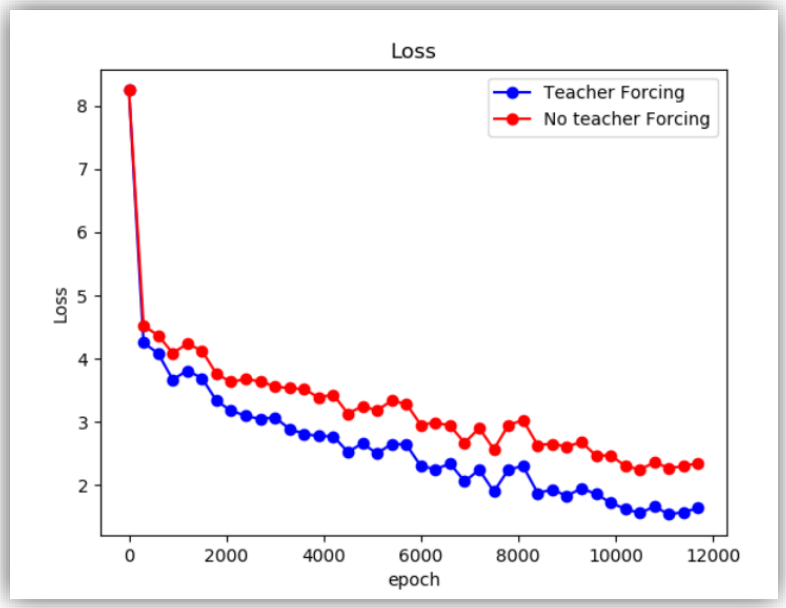
最後 1000 個 epoch 的 BELU score 都約為 0.47(仍有上升趨勢)



比較:

Train loss:

藍色為 teacher forcing，紅色為移除 teacher forcing。

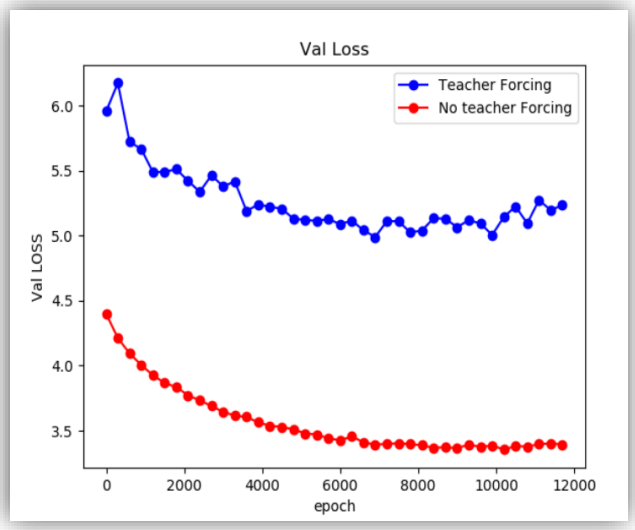


沒有 teacher forcing 的 Model loss 大概高了 1 左右(Cross Entropy)，老師上課曾經提到說 model 若假如完全靠著自己的預測去訓練會比較難 train 起來，這是因為 model 的預測結果可能有誤，當我們根據 gradient 去做改善時，其實是對於一個已經錯誤的狀況(錯誤預測的 input)做改善，這相當於說，如果之後 input 預測變成正確，之前的改善都沒有意義了，因此會比較難 train 的好。

然而 loss 並沒有我想的差距那麼大。

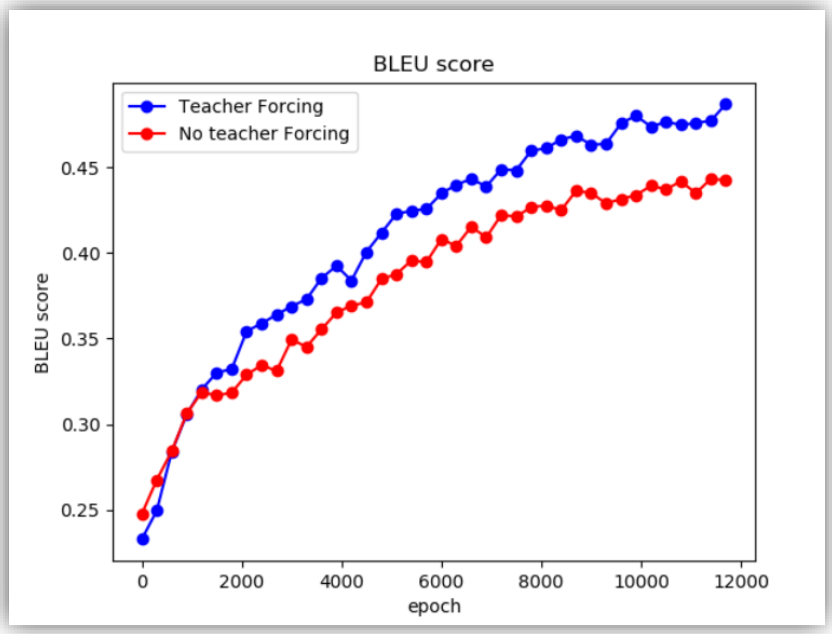
Val_loss:

同樣藍色為 teacher forcing 紅色為移除 teacher forcing



可以清楚看到 teacher forcing 的 validation loss 高出沒有 teacher forcing 的狀況許多。老師上課提到完全根據 reference 去預測的話，容易 train 跟 test set 的表現有很大落差，原因是假如 validation set 的第一個 input 就預測錯誤了，model 等於完全沒有遇過這種錯誤狀況，所以之後的預測就會偏差較大。而從訓練曲線也可以看到，有 teacher forcing 的 model loss 較高，可能就是來自上述原因。

VAL BLEU SCORE:



兩者都還有些微的上升趨勢，但是大體可以看出 teacher forcing 的 score 較高，我想這個原因可能也跟 no teacher forcing 的 model 比較難 train 起來是有關的，雖然整體而言 no teacher forcing 的 model 比較沒有 train test 表現偏差很大的狀況，但 score 的表現較差。

2. (30%) Attention Mechanism:

a. 請詳細說明實做 attention mechanism 的計算方式，並分析結果。

實作方法:

我使用的 match function 為 cosine similarity，將

Encoder 的 outputs(也有些是 Decoder 的 output):

(batch size, sequence length, hidden dim*2 (因 bidirectional))

中每一個 time stamp(1~sequence length)都跟 Decoder 的 hidden state

(batch size, 1, hidden dim)

做 dot product(也就是 cosine similarity)，然後做 softmax

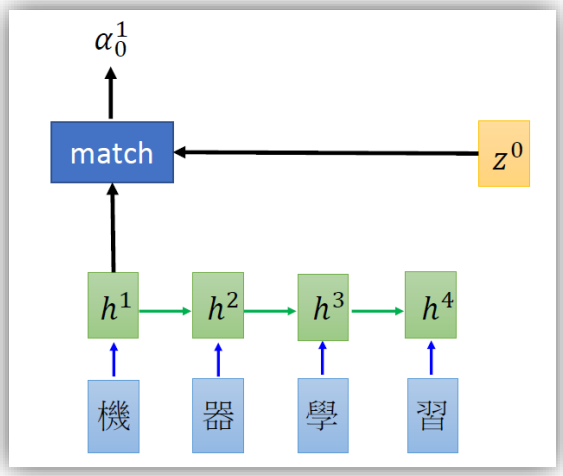
```
for i in range(encoder_outputs.shape[1]):
    # 有sequence length個dot product結果
    # hidden:(batch size, hid dim)
    dot_product1[:, i] = torch.sum(
        encoder_outputs[:, i, :] * hidden, dim=1)

dot_product1 = F.softmax(dot_product1, dim=1)
```

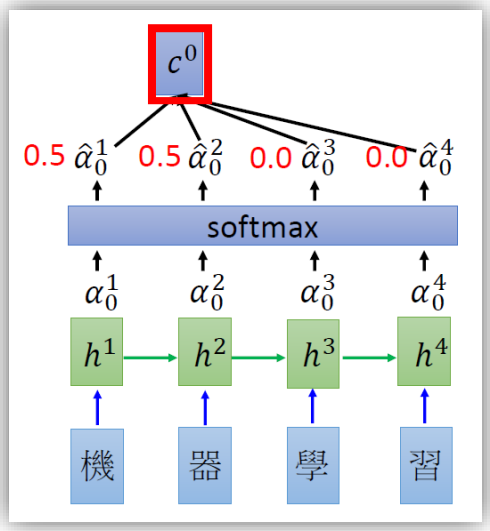
得到的結果(dot_product:(batch size,sentence length))就是每一個 time stamp 的 output 的 weight，將這個 weight 乘上 time stamp 後做加總，

```
attention1 = encoder_outputs[:, :, :]*dot_product1[:, :, None]
attention = torch.sum(attention1, dim=1).view(
    attention1.shape[0], 1, attention1.shape[2])
```

並接到 decoder 的 input 後面，即完成 Attention。

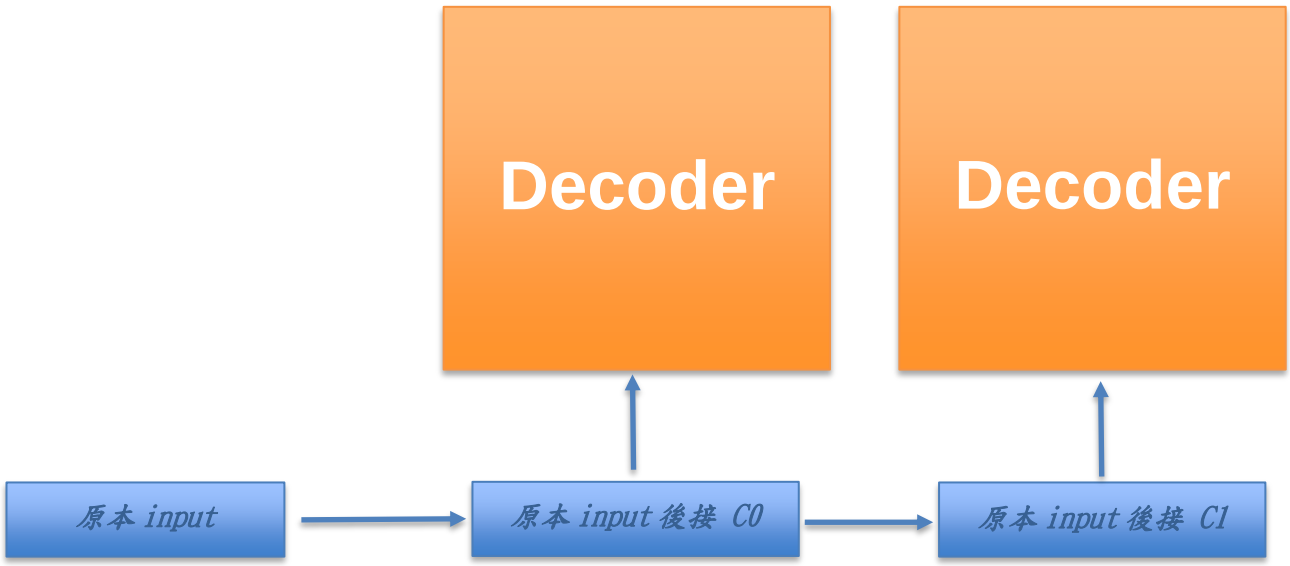
圖解: h1 h2 h3 h4 為 output，將其一一跟 z0 做 dot product(在此 z0 為 decoder 的 hidden state)





得到 dot product 後做 softmax 得到權重，權重乘上 Encoder outputs 並加總得到 c0。

C0 就是 attention vector，將他接在 decoder 原本的 input 後面(embedding 的 vector)



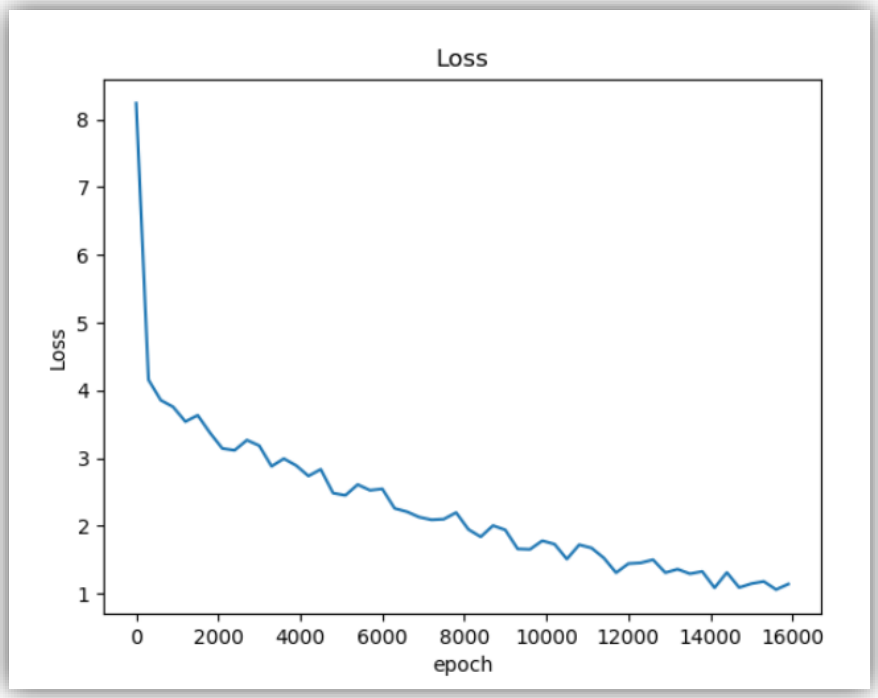

```
embedded = self.dropout(self.embedding(input))
# embedded = [batch size, 1, emb dim]
if self.isatt:
    attn = self.attention(encoder_outputs, hidden)
    # 將 attention接在embedded後
    embedded = torch.cat([embedded, attn], dim=2)
```

之後則接續相同的動作，將 Decoder 的 output 做 attention based(形成新的 attention vector c1 c2 c3….)，接在 input 後面並繼續丟入 Decoder，直到預測完全結束。

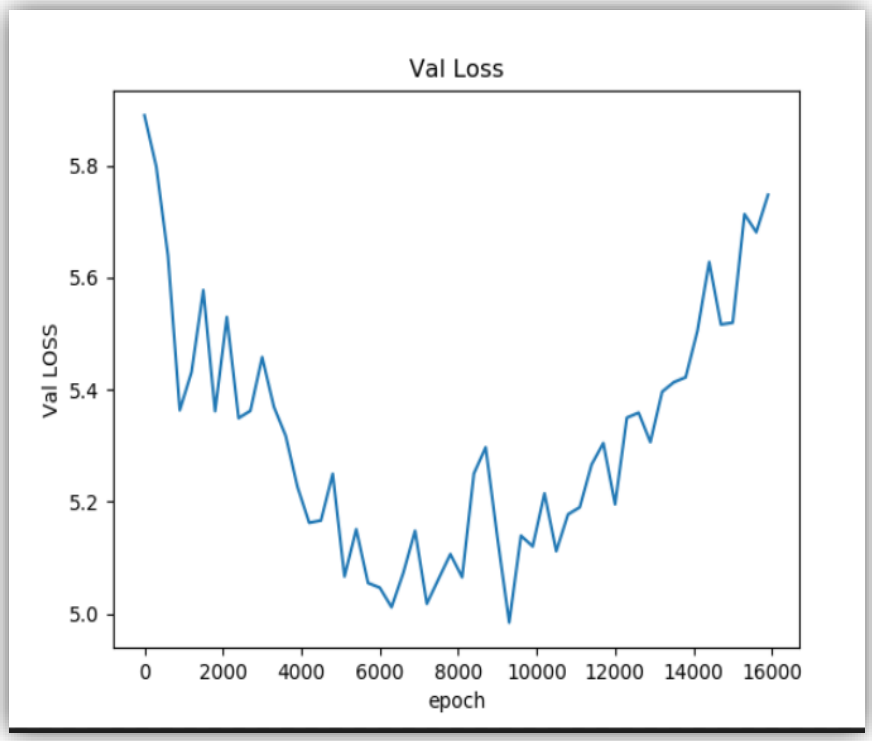
參數設定:
Batch size 為 60
Embedding dimension 設為 256
Hidden state dim:512
GRU 層數為 3
Decoder input 跟 FC 部分 dropout 為 0.5
Learning rate:0.00005
Optimizer: Adam
Maximum output length:50
訓練總次數 16000
每 300 次記錄一次 loss BLEU 變化

```
class configurations(object):
    def __init__(self):
        self.batch_size = 60
        self.emb_dim = 256
        self.hid_dim = 512
        self.n_layers = 3
        self.dropout = 0.5
        self.learning_rate = 0.00005
        self.max_output_len = 50
        self.num_steps = 16000                #
        self.store_steps = 300
        self.summary_steps = 300
        self.load_model = False
        self.store_model_path = "./ckpt"
        # 載入模型的位置 e.g. "./ckpt/model_
        self.load_model_path = './ckpt/mode
        self.data_path = sys.argv[1]
        self.attention = True
```

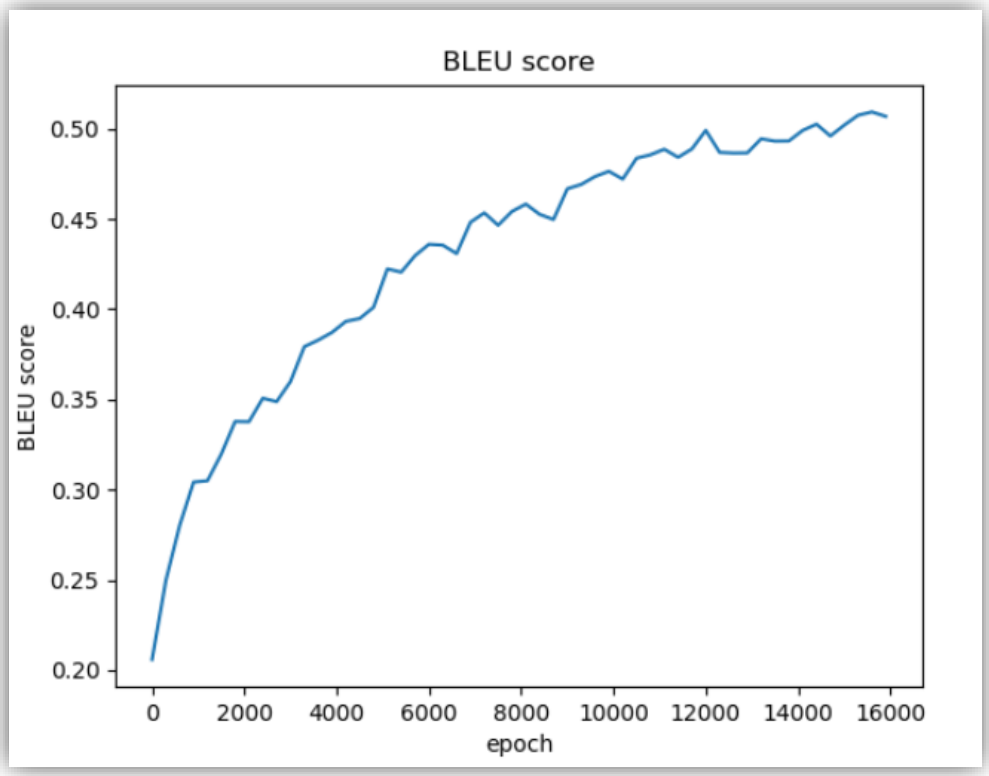
Train loss:
最後 1000 個 epoch loss 都大約落在 15000 的位置



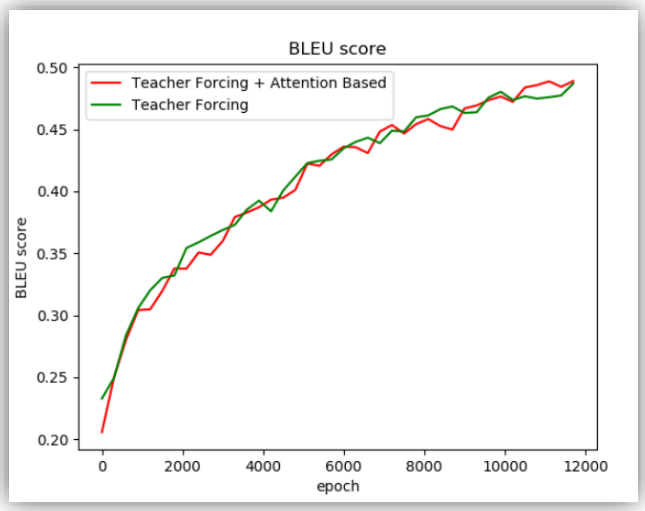
Val Loss:
Validation set 的 loss 在 epoch 8000 後又逐漸上升。



VAL BLEU SCORE:
BLEU SCORE 在 epoch 15000 後超過 0.5。



比較: Teacher Forcing vs Teacher Forcing + Attention Based:
Attention Based 主要是為了讓 Decoder 在預測結果時能更準確的抓到有用的 encoder output 資訊，因此我們可以 focus 在 BLEU SCORE 的變化。
兩者參數設定都相同(同此題)，為求趨勢變化明顯，作圖做到 12000epoch。



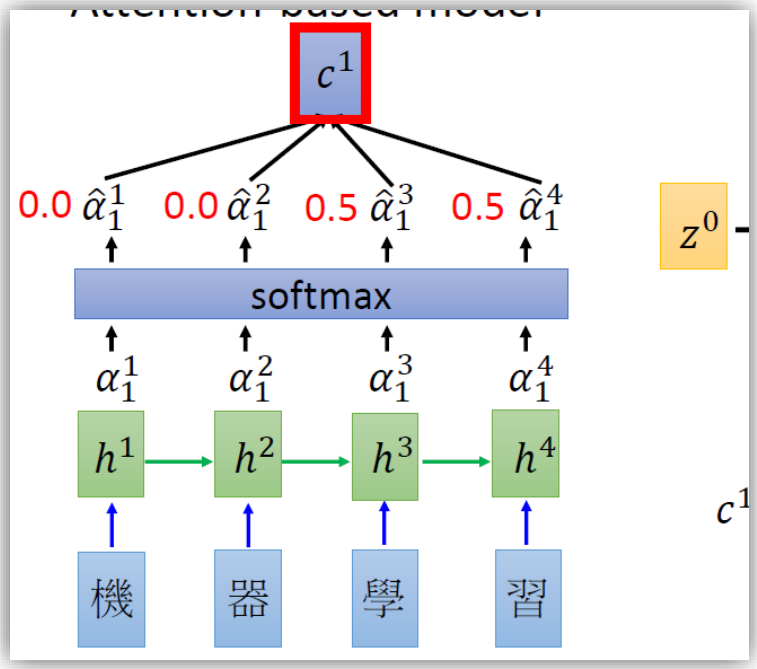
其中紅色為 TF+Attention Based, 綠色為 Teacher Forcing。
整個 SCORE 並沒有甚麼太大的進步或改善…。

以下是我認為可能的原因:

首先，match function 用 cosine similarity 真的能得到適合的 attention weight 嗎? 假設今天我們想在第一個 Decoder 預測得到正確的字，那麼照理來說我們會希望 Encoder 中比較早的 time stamp 的 output 所得到的權重較大(因為通常英文句子前面的字句被翻譯成中文順序也是在前的 例如”Tai Gaun Tsai” 跟”抬棺材” “Tai”應該是對應到”抬”)。

但是今天我們若將 decoder 的 hidden state 跟 Encoder 的 output 做 hidden state(這是第一個 Decoder 在做的事情)，Decoder hidden state 照理來說會比較像 Encoder output 中 time stamp 比較後面的部分(因為在我的實作中，是將 encoder RNN 的 hidden state 接上第一個 Decoder 的 hidden state)。這樣的話做 cosine similarity，就會是 time stamp 較後面的 encoder output 獲得較大的 weight，也就是說 z0 因為跟 h3 h4 比較像而使 h3 h4 獲得較大的 weight，而非 h1 h2 獲得較大 weight。

因此我認為要改善 Attention based 的話，需要再另外初始化一個 vector z0 可能效果會更好，但這裡為了讓之後 decoder 或的事情比較有相似性，我都直接用 Decoder hidden state 當作 z 的角色，可能就是這個原因導致效果並不明顯。



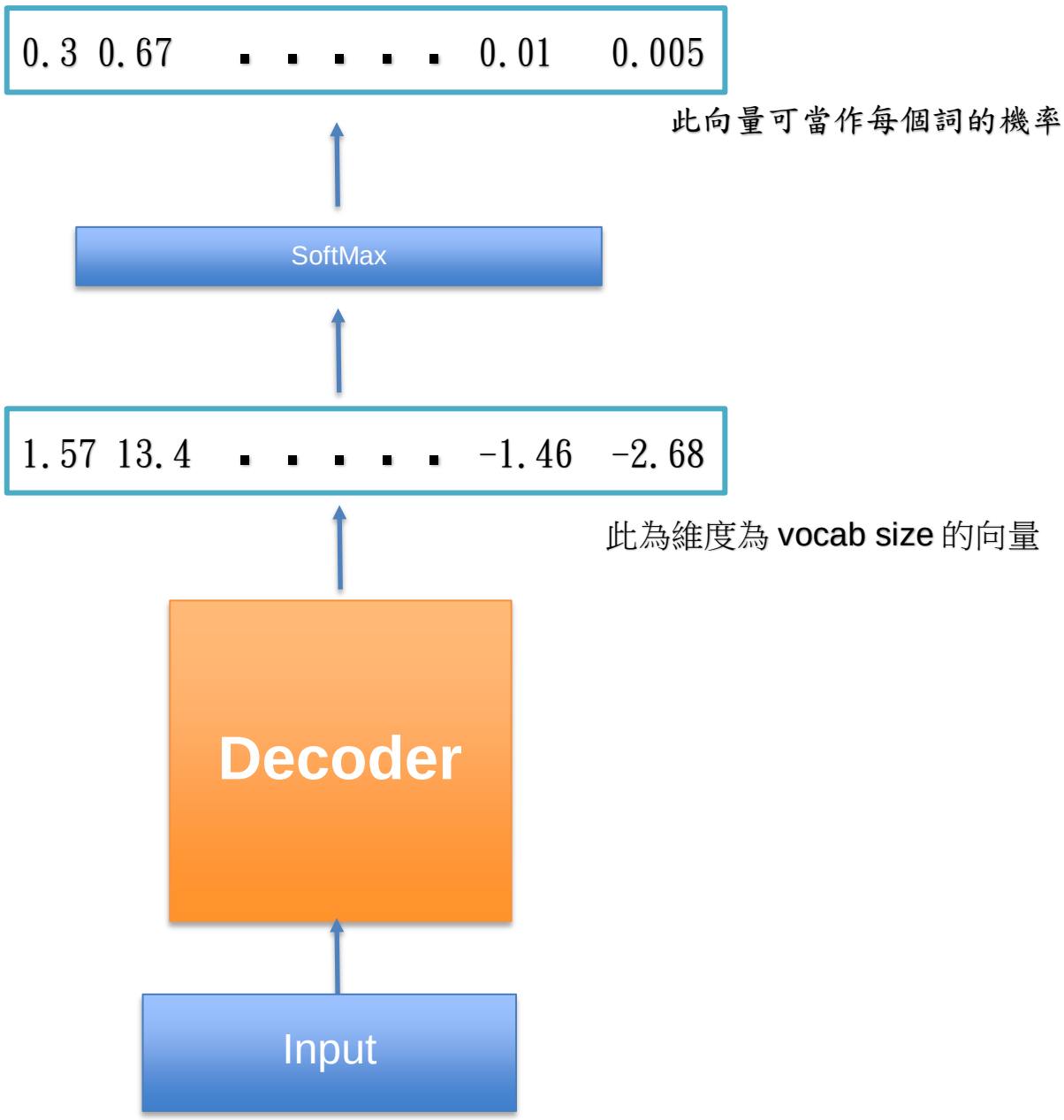
3. (30%) Beam Search:

a. 請詳細說明實做 beam search 的方法及參數設定，並分析結果。

實作方法: Attention + Teacher Forcing + Beam Search(同上傳版本)

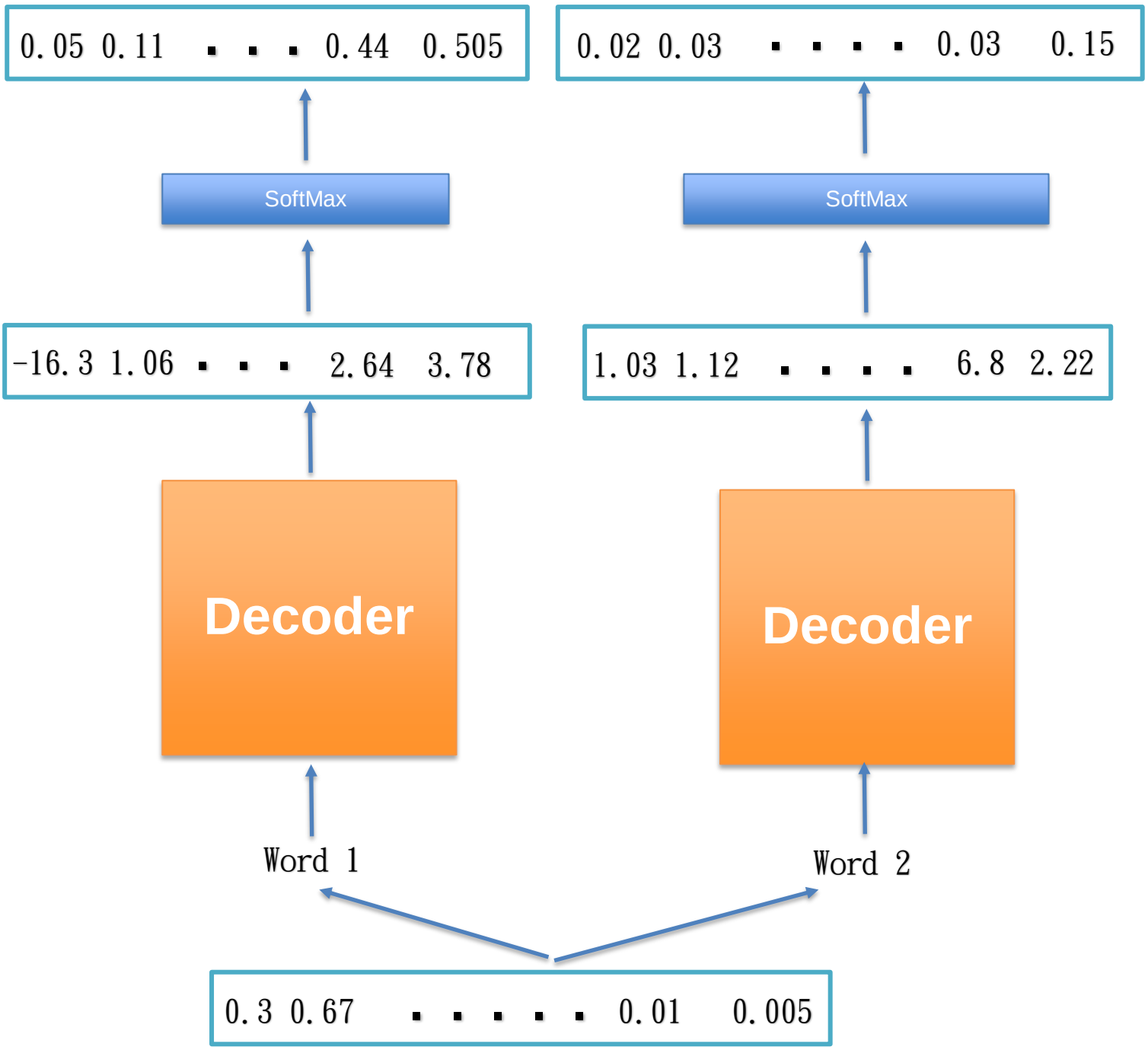
Beam Search 做法:

先將接上 Attention vector 的 input 丟入 Decoder，此時會產生維度 vocab size 的向量，將其做 softmax 後，可以把它視為是每一個詞的預測機率。



在這之中，取前兩大機率的單詞，當作第二個 decoder 的 input，丟入 decoder 後會得到 2*vocab size 個預測機率，將這 2*vocab size 個結果乘上先前 input 的詞的機率，並取最大的兩個當作第三個 decoder 的 input，以此循環直到碰到最後一個 decoder。

例如我們會將這兩個向量的機率乘上 word1 或 word2 的機率，然後選出最大的兩個詞，這兩個詞就是這次 Decoder 的預測結果。

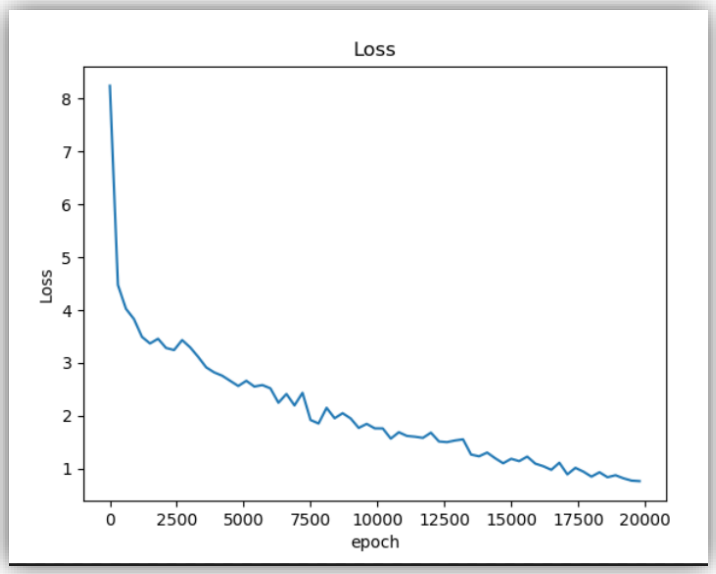


最後全部 Decoder 預測完後會挑最後一個 time stamp 機率最大的那個 word 並 trace back，將中途 Decoder 的預測結果納入，就完成一個句子的預測。

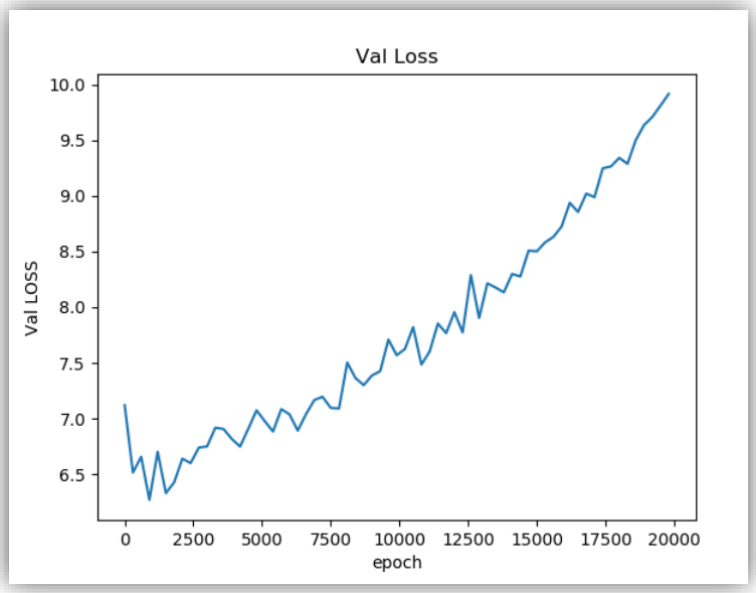
參數設定:
Batch size 為 60
Embedding dimension 設為 256
Hidden state dim:512
GRU 層數為 3
Decoder input 跟 FC 部分 dropout 為 0.5
Learning rate:0.00005
Optimizer: Adam
Maximum output length:50
訓練總次數 20000
每 300 次記錄一次 loss BLEU 變化

```
def __init__(self):
    self.batch_size = 60
    self.emb_dim = 256
    self.hid_dim = 500
    self.n_layers = 3
    self.dropout = 0.5
    self.learning_rate = 0.00005
    self.max_output_len = 50
    self.num_steps = 20000
    self.store_steps = 300
    self.summary_steps = 300
    self.load_model = False
    self.store_model_path = "./ckpt"
    # 載入模型的位置 e.g. "./ckpt/model_
    self.load_model_path = './ckpt/mode
    self.data_path = sys.argv[1]
    self.attention = True
```

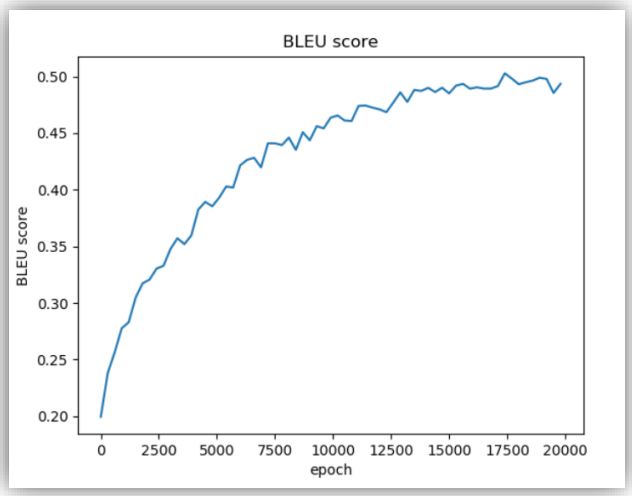
Train loss:最後約落在 0.8~0.9 間



Val Loss: Validation loss 越來越高，很可能是 Beam Search 對於預測產生了負面影響，因為總是猜錯結果，所以造成 loss 偏差越來越大。

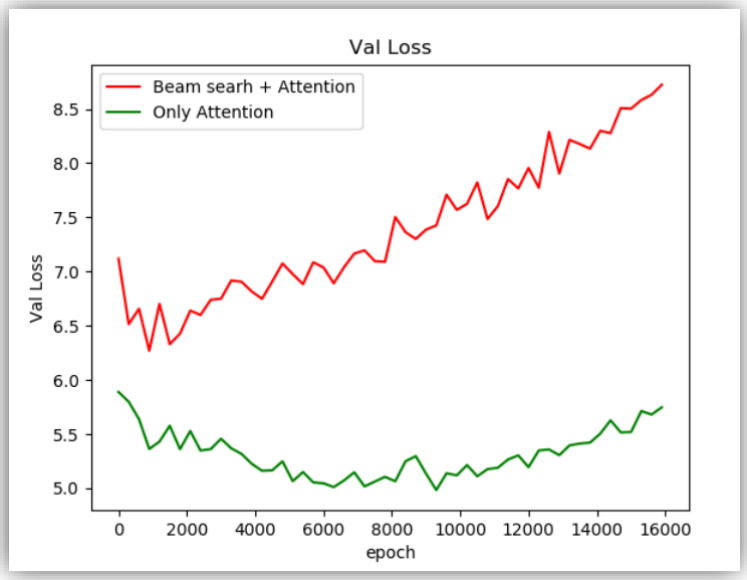


BLEU SCORE:最終約落在 0.49~0.50 之間



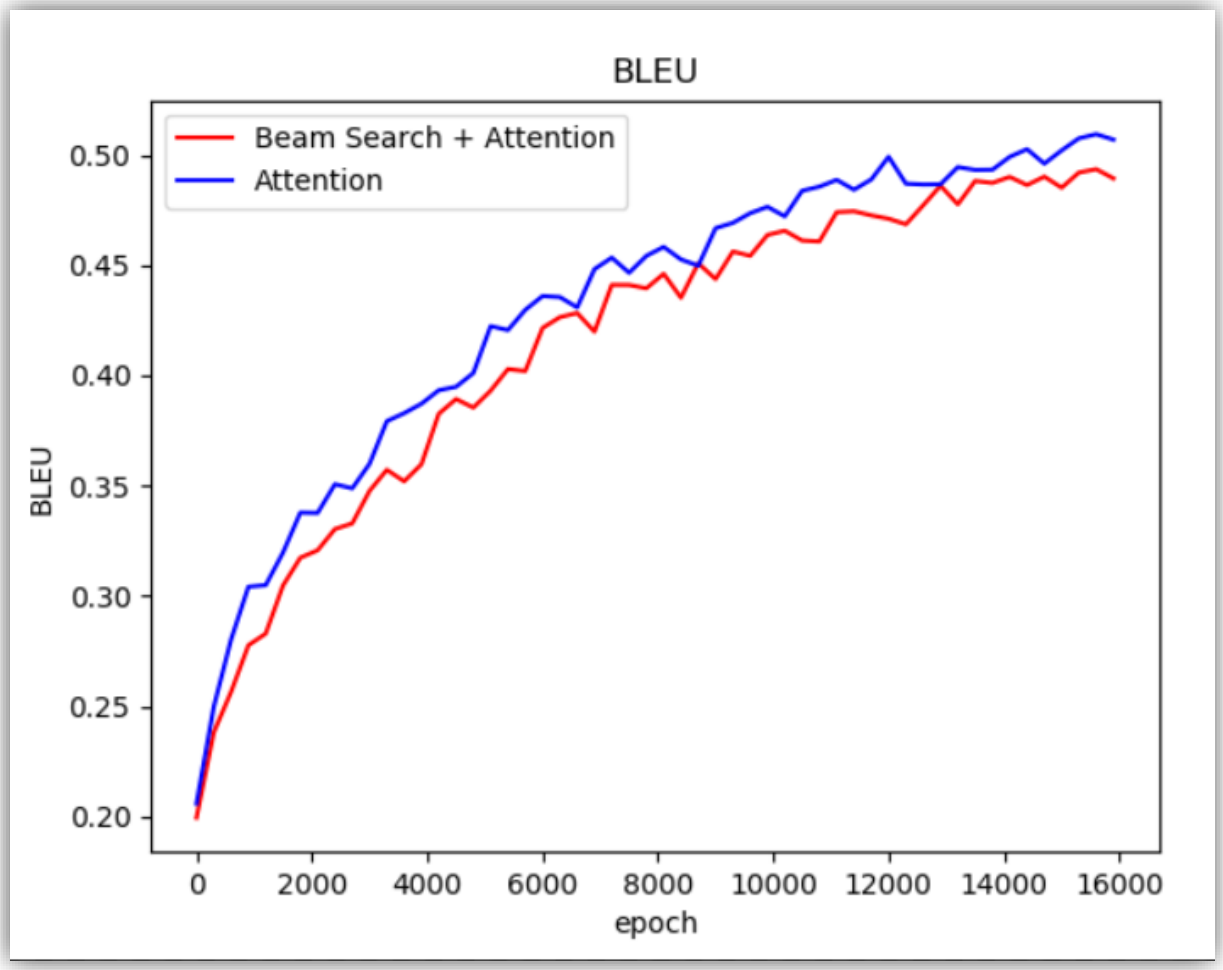
比較:(相同參數)
Attention +Teacher Forcing vs Attention + Teacher Forcing + Beam Search:
Beam search 這個方法主要是用在 testing 的階段，所以我們主要要比較兩者 validation loss 跟 BLEU SCORE 的部分:

Val LOSS:
可以看到使用 Beam search 的 model，validation loss 明顯高出了許多，可見 Beam Search 在預測實產生的結果並不好，因為不斷預測錯誤導致 loss 偏高，這很有可能是因為我實做的方法中，topk 的部分只取前兩大，且可能性探索的深度也不夠(這一點代表窮舉的可能性太少)，很有可能就是因為這樣導致了結果沒有變好反而變爛。



BLEU SCORE:

從這裡也可以看到 Beam search 的表現較差，再再顯示這次實驗中 beam search 對預測沒有造成好的效果。



分析:

然而以上述觀察就推斷 Beam Search 不會造成好的效果是不夠充分的，因為我實做的方法中只考慮每一層前兩大可能性，若能增加考慮的可能性數量，Beam Search 應該能夠發揮更好的效果。

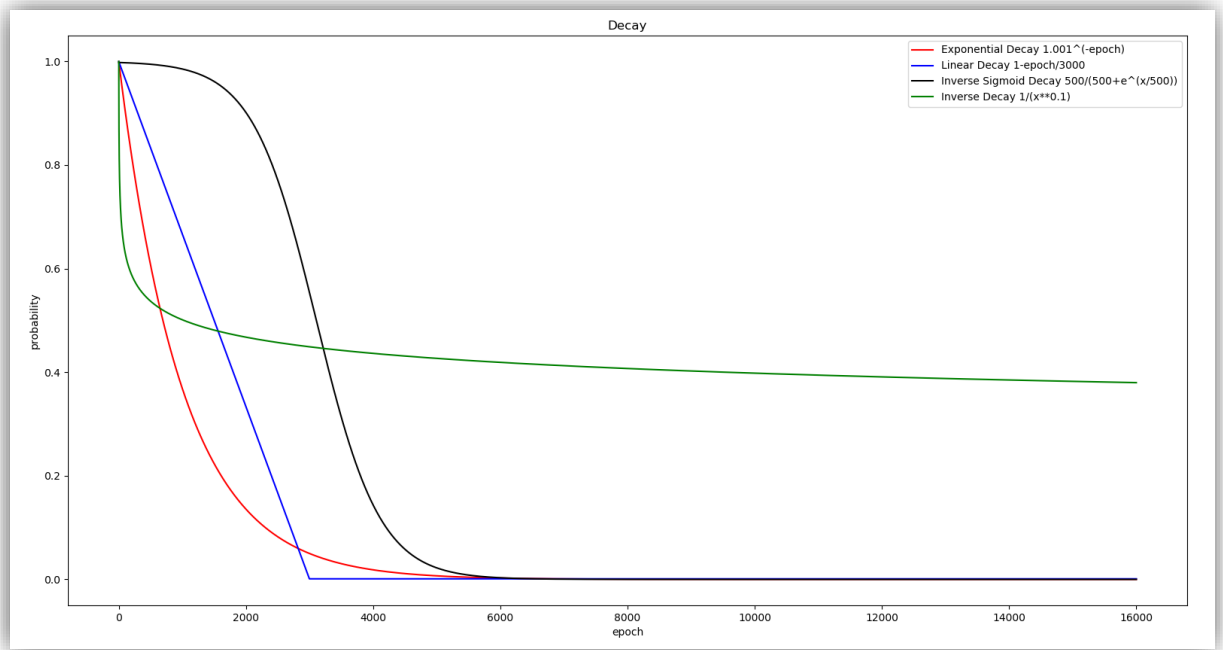
例如說，每次考慮所有預測結果的下一個預測結果的機率，也就是第一個 output 有 vocab size 個預測機率，把每一個字都當作 input 丟到下一個 decoder 並預測所有 output，然後再取機率最大者，如此一來可能可以增加預測準確度，不過 train 一次不知道要幾個小時呢。

4. (20%) Schedule Sampling:

a. 請至少實做 3 種 schedule sampling 的函數，並分析結果。

以下四種 function 使得 teacher forcing 的機率會隨 train 的次數產生變化，可以幫助我們決定是否要以 reference 來做預測。

先看到四種函數的變化趨勢:



紅色:Exponential Decay

藍色:Linear Decay

黑色: Inverse Sigmoid Decay

綠色:Inverse Decay

參數設定(四種 function 的參數皆相同):

Batch size:60

Embedding dim=256

Hidden Dim=512

GRU 層數為 3

Decoder input 前跟 FC 部分 drop out 0.5

Learning Rate:0.00005

Optimizer:Adam

Maximum output length:50

訓練次數 16000

每 300 次記錄一次

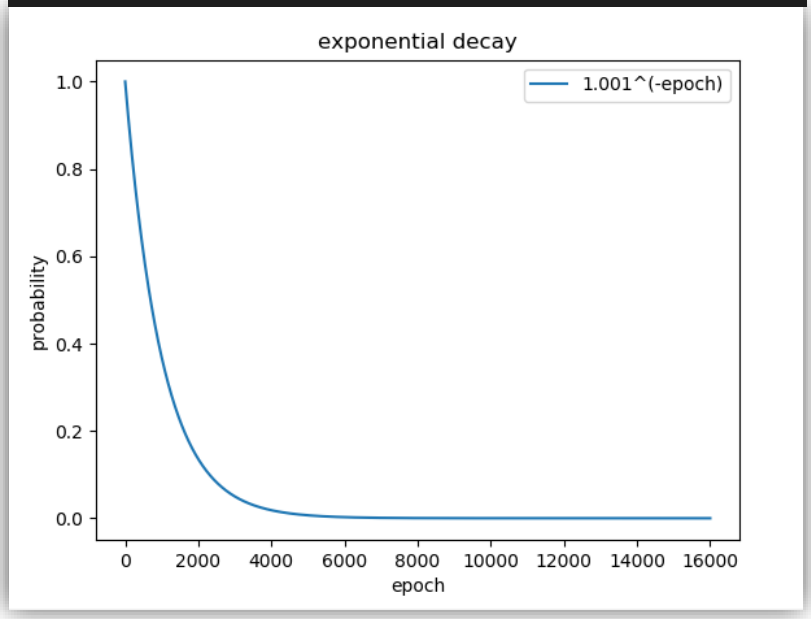
```
self.batch_size = 60
self.emb_dim = 256
self.hid_dim = 512
self.n_layers = 3
self.dropout = 0.5
self.learning_rate = 0.00005
self.max_output_len = 50
self.num_steps = 16000
self.store_steps = 300
self.summary_steps = 300
self.load_model = False
self.store_model_path = "./ckpt"
# 載入模型的位置 e.g. "./ckpt/model_
self.load_model_path = './ckpt/mode
self.data_path = sys.argv[1]
self.attention = False
```

這裡沒有使用 Attention 純粹針對 teacher forcing 得機率的影響做觀察

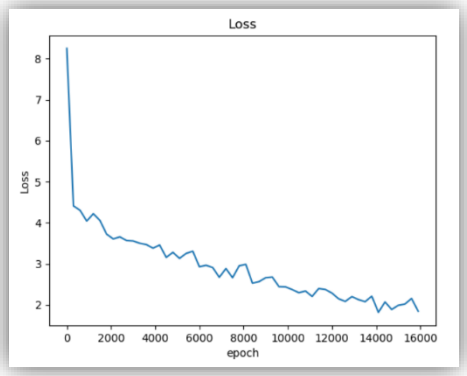
以下將逐一分析

Function1: Exponential Decay $1.001^{(-x)}$

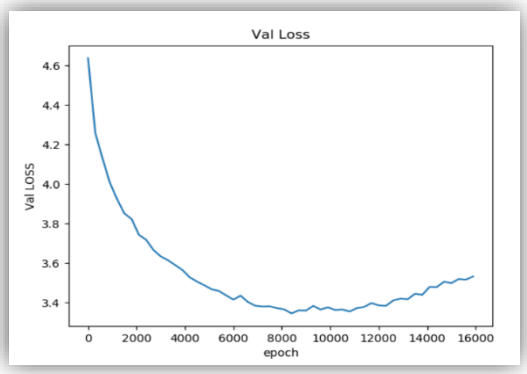
```
def schedule_sampling(steps):
    return torch.pow(1.001, -steps)
```



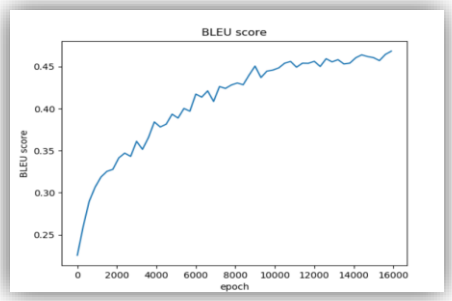
Train Loss: 最後 1000 個 epoch 約為 1.9



Val Loss:最後約為 3.5



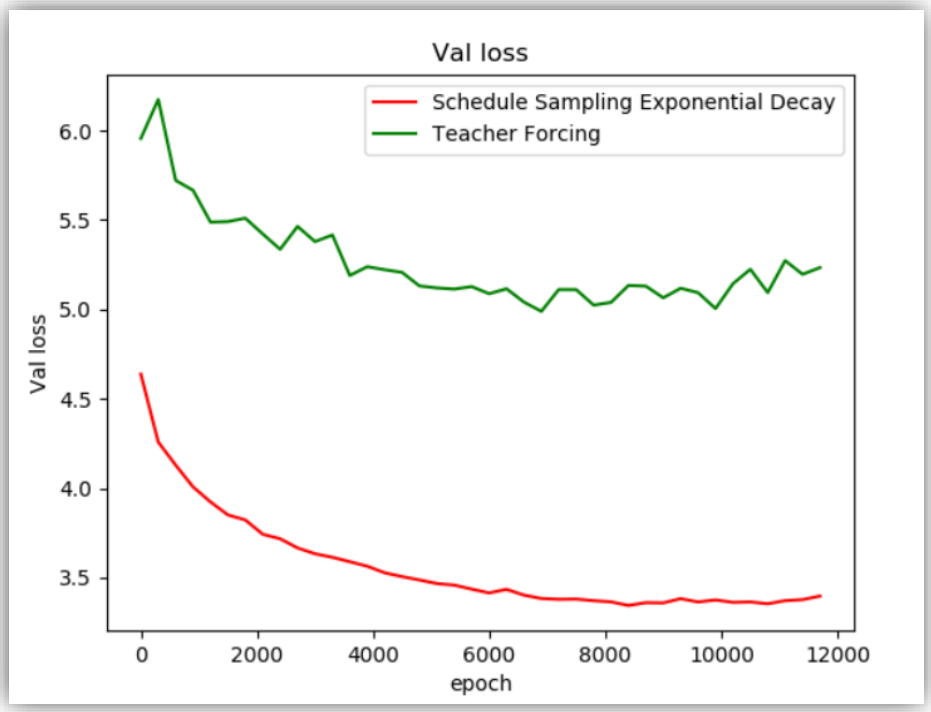
VAL BLEU SCORE:最後約為 0.47



分析:

這個 function 隨 epoch decay 的速度可說是 4 個 function 中最快的，因為 schedule sampling 主要是為了解決 Time Test 不平均的表現，因此我們拿它來跟 Tearcher Forcing 的版本做比較。

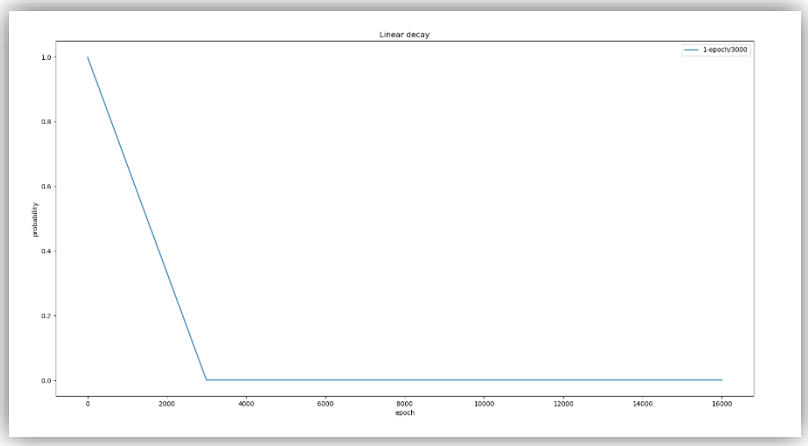
從下圖可以明顯看到 Val loss 的部分，使用 schedule Sampling 的確讓 validation loss 下降了不少。



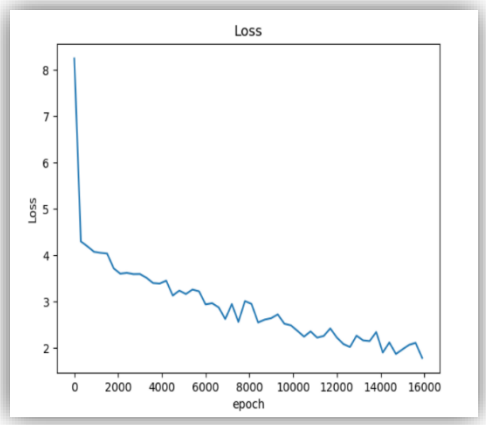
Function 2: Linear Decay max(0.001, $1 - \frac{epcoh}{3000}$)

線性衰減，但是強制修剪所有低於 0.001 的機率為 0.001。

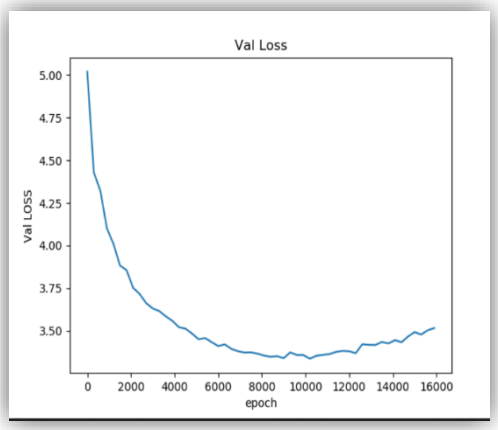
```
def schedule_sampling(steps):
    c = 1/4000
    return torch.max(0.001, 1.-c*steps)
```



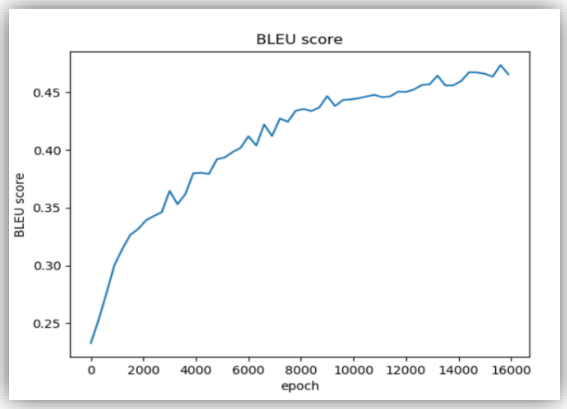
Train Loss:　約為 1.9



Val Loss:約為 3.5

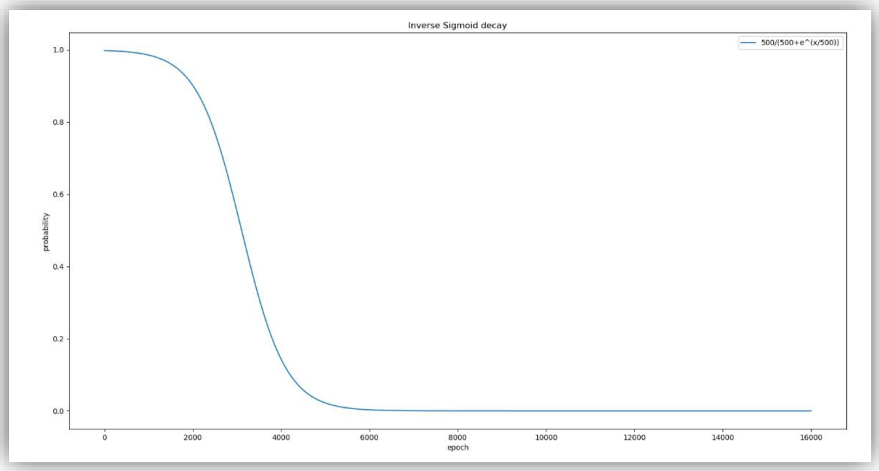


VAL BLEU SCORE:約為 0.47

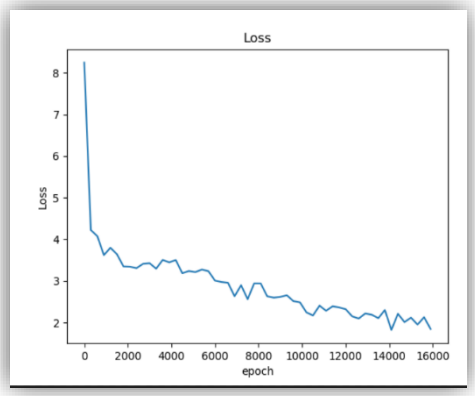


Function 3:Inverse Sigmoid Function: $\frac{500}{500 + e^{x/500}}$

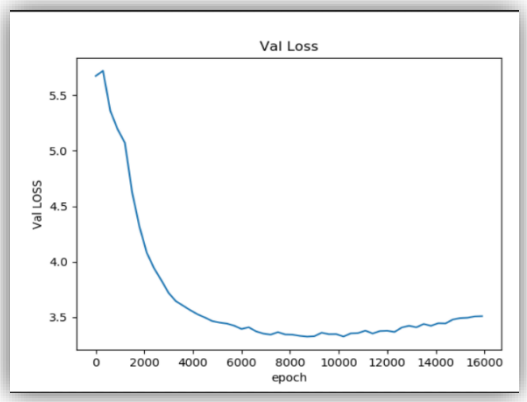
```
def schedule_sampling(steps):
    k = 500.
    return k/(k+torch.exp(steps/k))
```



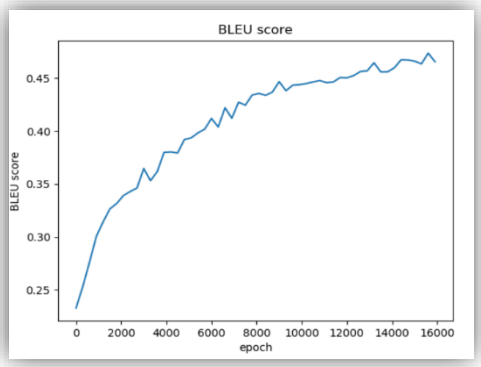
Train Loss: 最後 1000 個 epoch 約為 2



Val Loss:約為 3.5



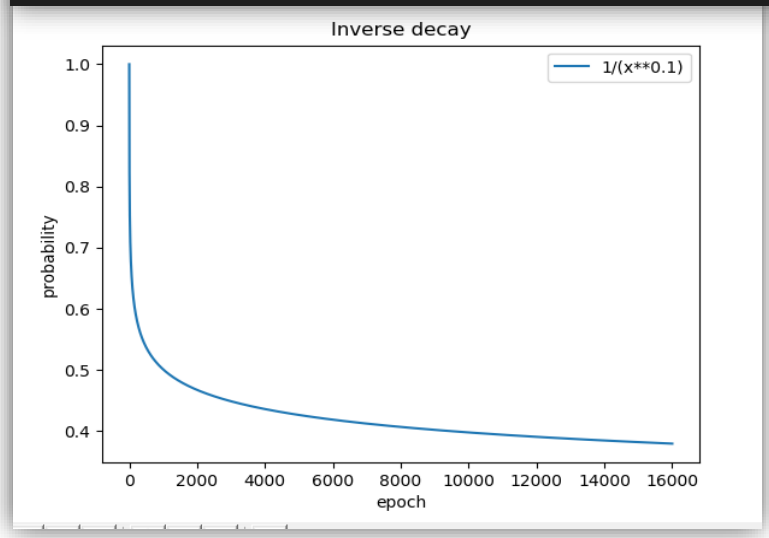
VAL BLEU SCORE: 最後 1000 個 epoch 約落在 0.47



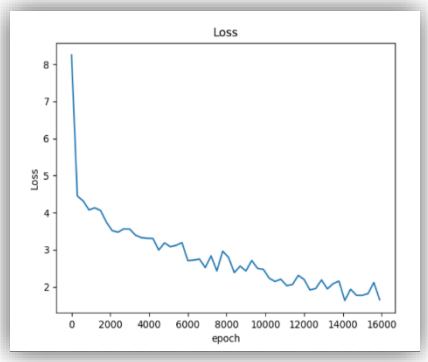
Function 4: Inverse Decay: $\frac{1}{x^{0.1}}$

teacher forcing 的機率為 x^0.1 的 inverse

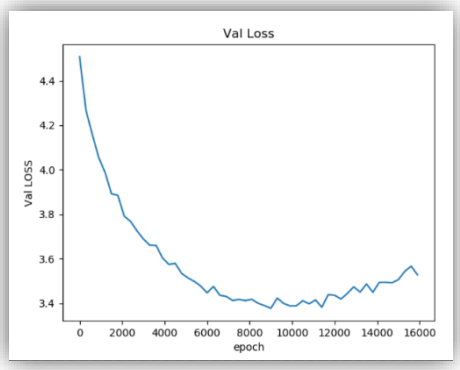
```
def schedule_sampling(steps):

    return 1/(torch.pow(steps, 0.1))
```



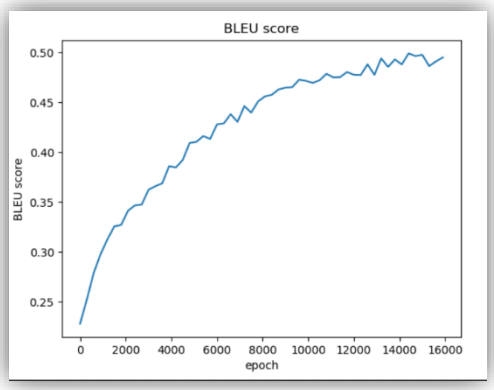
Train Loss:　最終約為 1.8



Val Loss:約為 3.5 過 10000 後反升



VAL BLEU SCORE: 約為 0.495



四者比較:

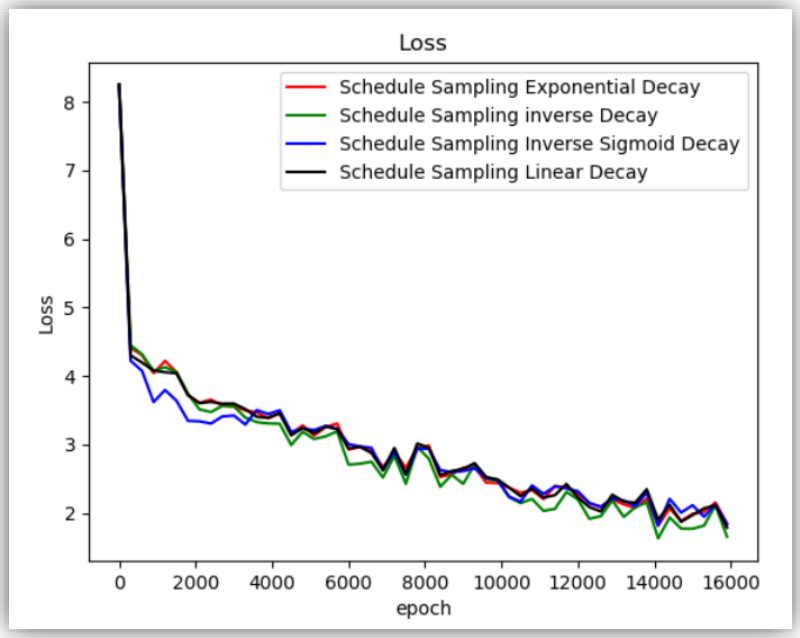
紅色:Exponential Decay

藍色:Inverse Sigmoid Decay

黑色:Linear Decay

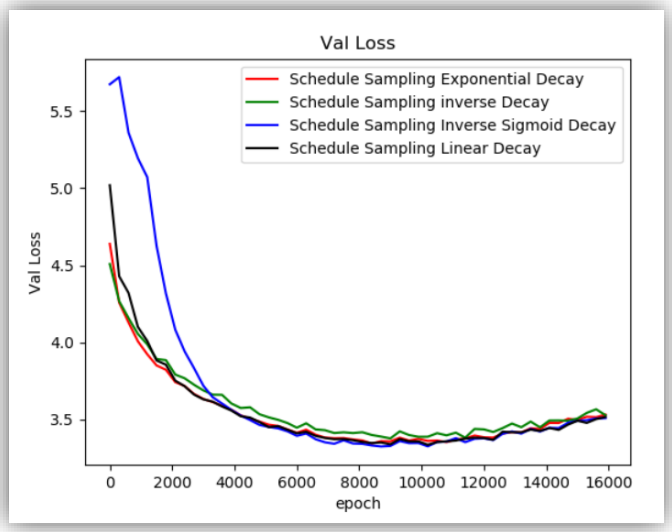
綠色:Inverse Decay

Train Loss: Train loss 的部分四條線差不多，沒有明顯的差異。

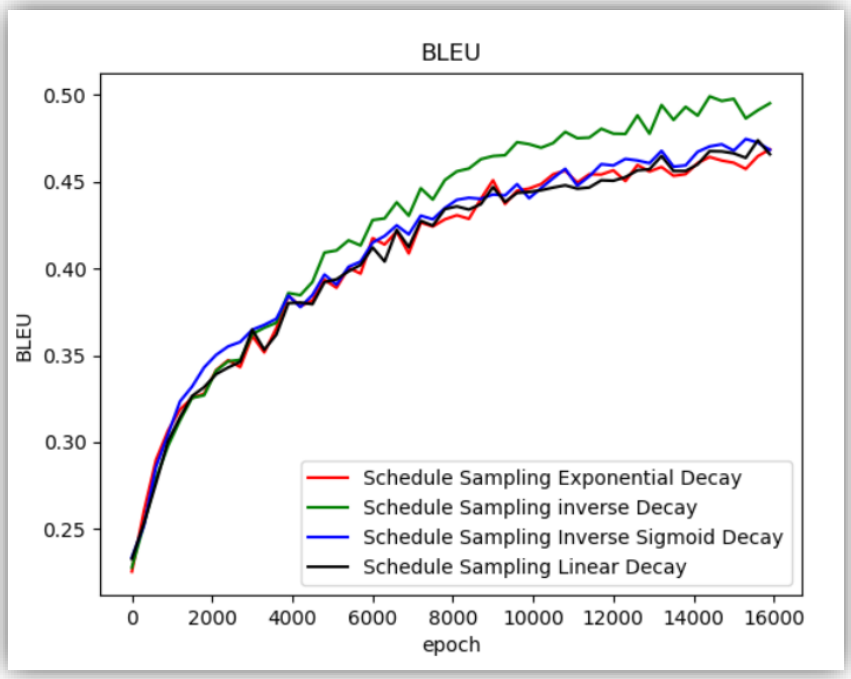


Val Loss:

Validation loss 的部分 可以看到除了 inverse sigmoid decay 以外，其他線條的結果都差不多，而 inverse Sigmoid Decay 的 validation loss 收斂的最慢，且在 epoch 較小的時候 validation loss 最高。

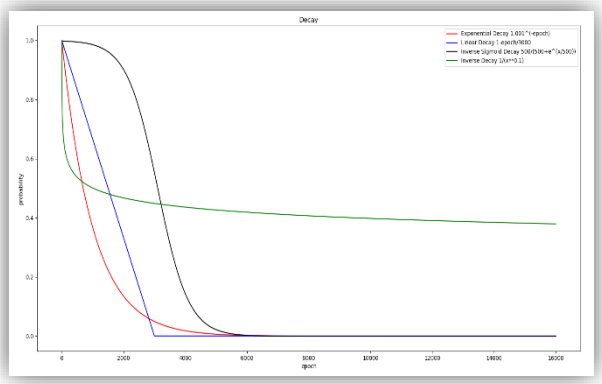


BLEU SCORE:BLEU SCORE 的部分也是大部分都不多，而 inverse Decay 的 BLEU SCORE 最高



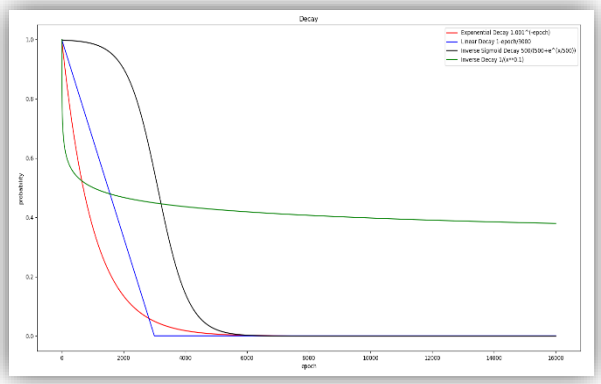
結論:

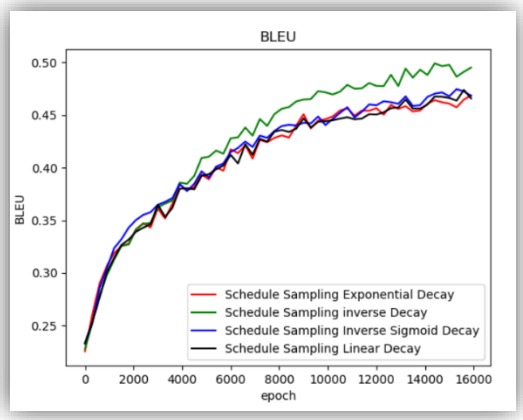
從以上四個函數的比較可以發現，結果其實都很類似。 差異比較大的是 validation loss，Inverse Sigmoid Decay 的函數的 validation loss 收斂的最慢，我們可以回去觀察四種 decay 的機率變化，可以發現 inverse sigmoid Decay 本身的機率收斂也是最慢的，機率收斂的越慢，代表它越晚脫離 teacher forcing 的狀態(因為機率到較高 epoch 才掉下來)，也就是說 train/test 的表現可能會較像 teacher forcing，也許就是這樣的原因導致它的 validation loss 在 epoch 小的時候是四者最高的。



黑線為 inverse sigmoid decay

而因為 BLEU SCORE 的表現都太過相似，因此我又多做了 Inverse Decay 這個函數，發現它的 BLEU SCORE 最高。 綠色線條為 Inverse Decay





可以很清楚觀察到 inverse Decay 跟其他 decay 方式最不一樣的地方在於，在高 epoch 的地方，inverse decay 的 teacher forcing 的機率是收斂於 0.4 左右的，其他每一種 decay 在高 epoch 處幾乎都是收斂於 0。

這樣的現象讓 inverse Decay 的 model 在高 epoch 處仍然能夠有機會以正確的 reference 進行 Decoder 的預測，使用其它種 Decay 的 model 因為最後幾乎都是以自身預測的結果進行訓練，很有可能就是因此導致了表現比 inverse decay 還差。