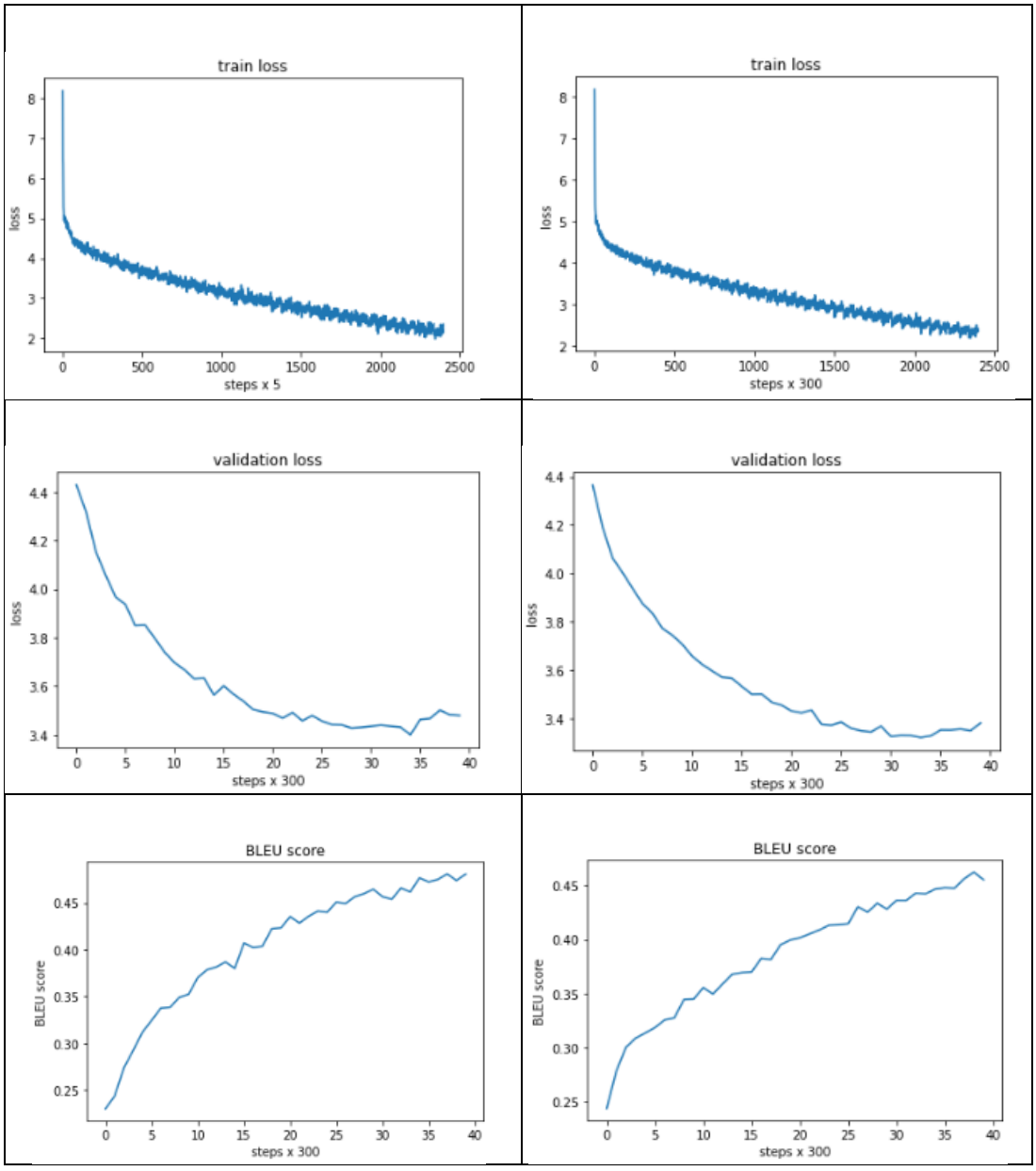
學號：**R08946006**　　系級：資料科學學程碩一　姓名：　周逸平

1. **(20%) Teacher Forcing:**

請嘗試移除 Teacher Forcing，並分析結果。

啟用 Teacher Forcing 0.5 圖　　未啟用 Teacher Forcing 圖

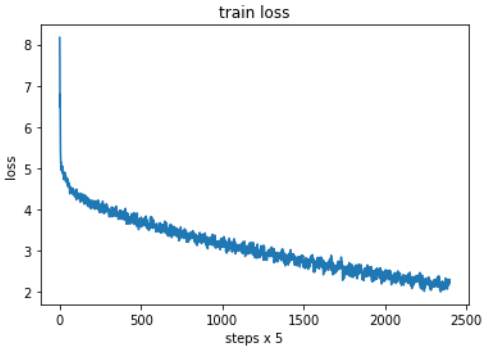


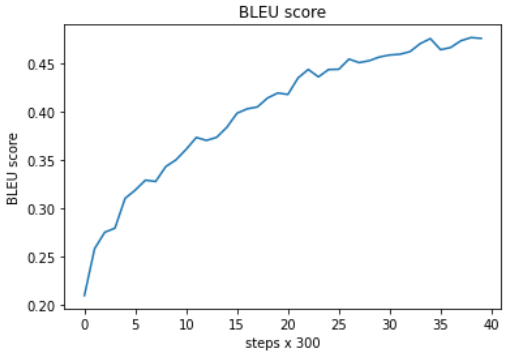
右上 Train 的 x 軸寫錯了，是 steps * 5 才對。

啟用 teacher forcing 的 train loss 相較未啟用的震盪較小，因為中間有 5 成都是直接吃正解的，所以每次 iteration 相較未啟用的都可以保證 5 成的輸入絕對正確，但是到 val loss 時較為顛簸，而這可能是因為 val 時不再有正解作為輸入，而 model 本身吃到的自身產出也不是正解，造成某些 val loss 稍微上升，亦或是 model 產出了更好的解，使其下降，而 BLEU 同理。

2. (30%) Attention Mechanism:

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





先分析結果，使用與 Q1 Teacher Forcing 0.5 相同參數，從 train loss 上雖然看不出大變化，但可以看到 val loss 的曲線稍微平滑了點，而 BLEU 的曲線也是圓滑了許多，尚未使用 Attention 的 BLEU 可以感受到在第 5 次 valid 前後其斜率有差別，但使用了 Attention 之後的比較像是一條曲線。

實做 Attention 的方式以下圖 code 截圖說明。

使用 encoder_outputs 以及 decoder_hidden 分別做為 queue 以及 keys，而 decoder_hidden 因為有三個 layer 故直接將其相加。

其後將 encoder_outputs 以及相加後的 decoder_hidden 做 concat 之後經過對應的 fc 層並以 tanh 作為激發層，最後為了對應上 value 維度再過一層 fc。

```python
class Attention(nn.Module):
  def __init__(self, hid_dim):
    super(Attention, self).__init__()
    self.hid_dim = hid_dim
    self.attn = nn.Linear((hid_dim * 2) + (hid_dim * 2), hid_dim)
    self.v = nn.Linear(hid_dim, 1, bias = False)

  def forward(self, encoder_outputs, decoder_hidden):
    # encoder_outputs = [batch size, sequence len, hid dim * directions]
    # decoder_hidden = [num_layers, batch size, hid dim]
    # 一般來說是取 Encoder 最後一層的 hidden state 來做 attention
    # num_layers = 3
    # TODO
    src_len = encoder_outputs.shape[1]
    decoder_hidden = decoder_hidden[0] + decoder_hidden[1] + decoder_hidden[2]
    decoder_hidden = decoder_hidden.unsqueeze(1).repeat(1, src_len, 1)

    energy = torch.tanh(self.attn(torch.cat((encoder_outputs, decoder_hidden), dim = 2)))
    attention = self.v(energy).squeeze(2)

    return F.softmax(attention, dim = 1)
```

我們得到 attention 的 output 之後，便與 encoder_outputs 做 bmm 來得到權重，再將其與 embedded 做 concat 後送入 rnn 中，此舉希望 rnn 在看到 embedded 以及 weight 後，可以以對應的 weight 來評斷輸入字串中各詞的重要性，此處 rnn 的輸入維度也有做對應調整。

```python
if self.isatt:
    # TODO: 在這裡決定如何使用 Attention，e.g. 相加 或是 接
    attn = self.attention(encoder_outputs, hidden)
    attn = attn.unsqueeze(1)
    weighted = torch.bmm(attn, encoder_outputs)
    rnn_input = torch.cat((embedded, weighted), dim=2)
    output, hidden = self.rnn(rnn_input, hidden)
```

3. (30%) Beam Search:

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：

以 python 之 heapq 並令為 pq 實現。以 code 截圖說明。

從右圖 code 中可以發現，我將 beam search 分為兩個部分，即 t==1 時，由於只有單一個 node (<bos>)，故此處基於 bos 選擇出 topk，共 k 個候選。

t>=2 時，基於 k 個候選在分出 topk，共 k*k 個候選，並從之選出 topk。

```
preds = []
for t in range(1, input_len):
  if t == 1:
    # first
    output, hidden = self.decoder(input, hidden, encoder_outputs)
    outputs[:,t] = output
    output = F.log_softmax(output, dim = 1)
    log_prob, indexs = torch.topk(output, BEAM_WIDTH)

    for k, (log_p, idx) in enumerate(zip(log_prob[0], indexs[0])):
      idx = idx.view(1)
      pq.heappush(preds, [-log_p.clone().item(), [idx], hidden.clone(), outputs.clone()])

  else:
    temp = []
    for i in range(BEAM_WIDTH):
      cur_p, cur_tokens, cur_hidden, cur_outputs = preds[i]
      input = cur_tokens[-1]
      output, cur_hidden = self.decoder(input, cur_hidden, encoder_outputs)
      cur_outputs[:,t] = output
      output = F.log_softmax(output, dim = 1)
      log_prob, indexs = torch.topk(output, BEAM_WIDTH)
      # print(log_prob)

      for j, (log_p, idx) in enumerate(zip(log_prob[0], indexs[0])):
        temp_p = cur_p
        idx = idx.view(1)
        tmp_token = cur_tokens + [idx]
        temp_p -= log_p.item()
        pq.heappush(temp, [temp_p, tmp_token, cur_hidden.clone(), cur_outputs.clone()])

    preds = [pq.heappop(temp) for i in range(BEAM_WIDTH)]
    # print(len(preds))
    # print(len(preds[0]))
    del temp

preds = pq.heappop(preds)
_, preds, _, outputs= preds
preds = torch.LongTensor(preds).view(1, len(preds))
return outputs, preds
```

操作方法為：設定一 list 用以儲存每一輪之 topk 結果，此處為 preds

每一輪提取出 preds[i]其中包含以下

1. 當前累積機率 (-log 後) – cur_p
2. 當前句子 – cur_tokens
3. 當前 hidden state – cur_hidden
4. 當前總 output – cur_outputs

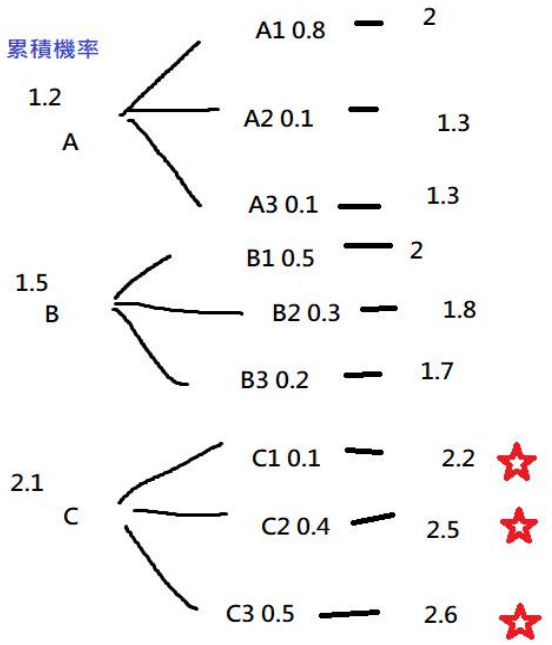
取 cur_tokens[-1]作為當前 input，並與當前 hidden state 以及 encoder_output 作為 decoder 輸入參數。

將 decoder 產出 hidden state 與 output 分別做覆蓋 (cur_hidden = hidden) 以及紀錄 (cur_outputs[:,t] = output)後。

為了計算累積機率，對 output 做 log_softmax，並以 topk 函數取得 topk 值以及 index。使用 log 為避免若是使用機率相乘有可能會收縮到 0。

將對應的機率累積至 cur_p 後，使用 heappush 紀錄同上 4 項，惟注意累積機率當使用 -=，因為 heappop 會吐出 list 首元素中最小項目，最後再使用 k 次 heappop 來取得 topk。

Beam search 結束後再使用 heappop 一次得到累積機率最高者，取出字串以及 output 分布後回傳。付上簡易小畫家做圖作為示意圖(圖中為取-log 後機率和)。



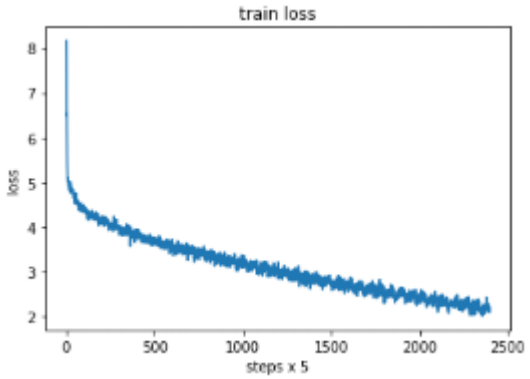
測試結果如下：

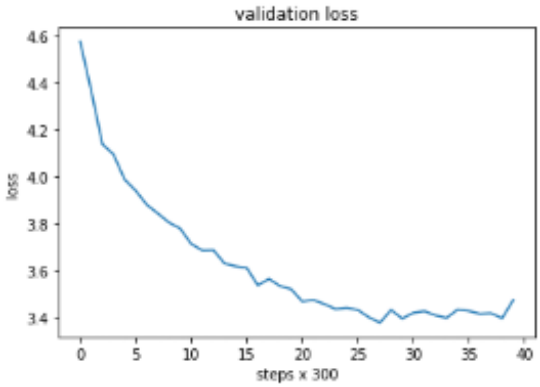
此處使用 b = 3，有 attention 以及 teacher forcing = 0.5。

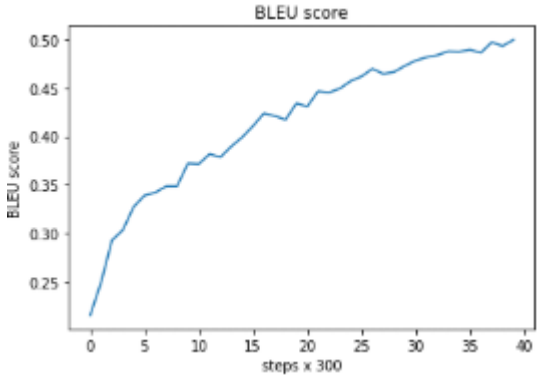
使用 beam search 似乎在 train loss 上的曲線上並無太大變化，但是在 val loss 相較上面幾種方法有更多次達到了 3.4 以下的 loss 成績，而在 BLEU 上則是達到了最高 0.5 的成績，相較上述方法最高只達到 0.45 來說是相當驚人的進步。

可見使用 Beam search 來做 inference 是可以帶來相當可觀的成長的，惟使用 Beam search 需要消耗掉將當大的運算資源，自 t>=2 之後每輪 decoder 都需要被 call k*k 次，並且進行排序才可以選出 topk 對象，這十分的耗時間。

以助教設定的 12000step，每 300steps 做一次 val 來說，得到上述結果共花費 9903 秒，且是於 TITAN V 單張運算以及 Intel(R) Xeon(R) CPU E5-2623 v4 @ 2.60GHz 上使用 8 threads 做運算，於 inference 上的確是相當可觀的時間耗損。



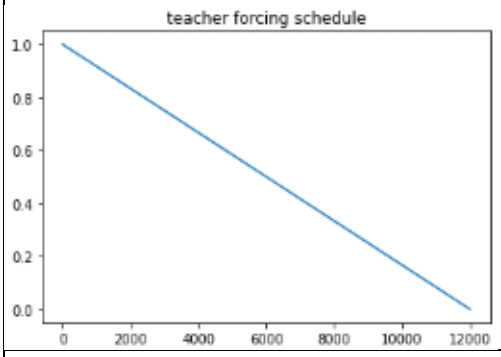
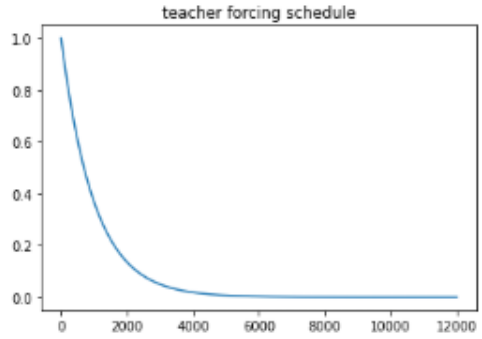
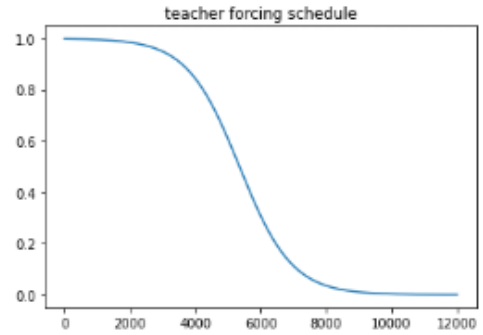
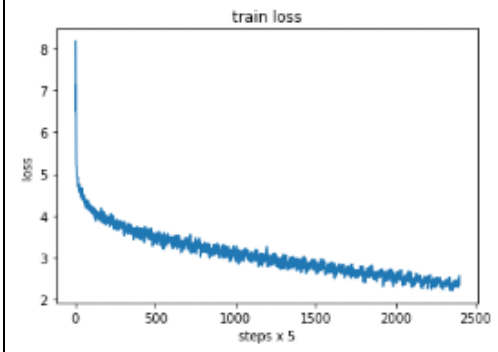
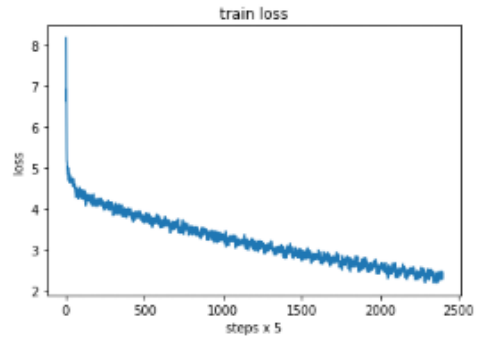
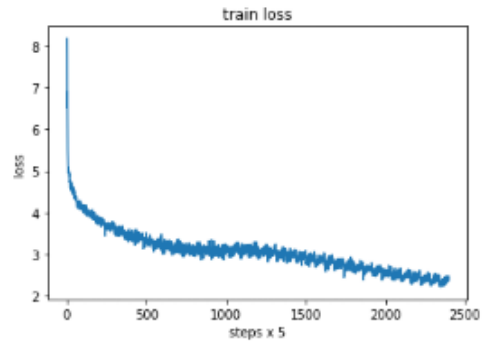
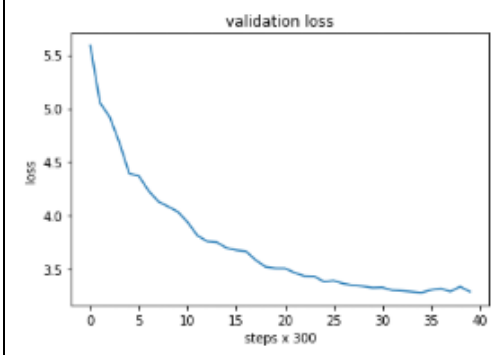
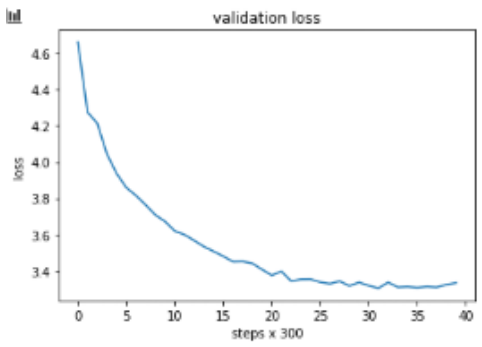
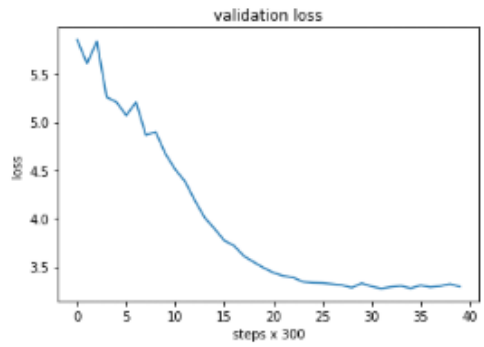
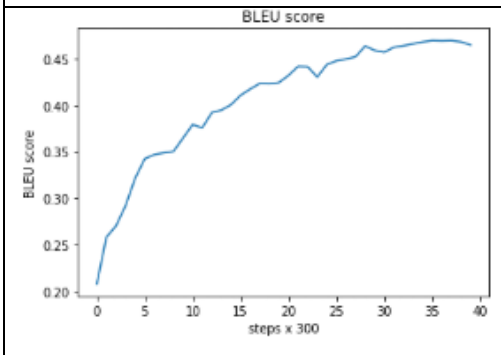
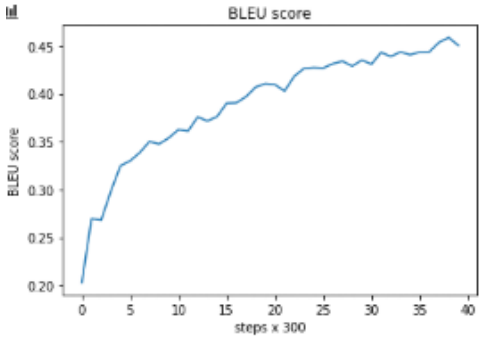
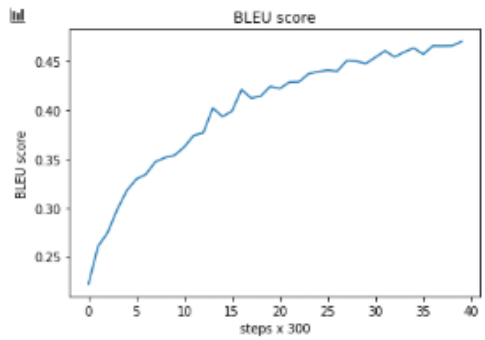




4. (20%) Schedule Sampling:

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

下頁分析結果

Linear	Exponential, k = 0.999	Inverse Sigmoid, k = 800
		
		
		
		
Best bleu 4.70	4.58	4.70

三種 schedule sampling 的方法直接使用於助教 tutorial p10 所附上之 paper 內提到之三種 sampling 方法，公式如下：

- Linear decay: $\epsilon_i = \max(\epsilon, k - ci)$ where $0 \leq \epsilon < 1$ is the minimum amount of truth to be given to the model and k and c provide the offset and slope of the decay, which depend on the expected speed of convergence.
- Exponential decay: $\epsilon_i = k^i$ where $k < 1$ is a constant that depends on the expected speed of convergence.
- Inverse sigmoid decay: $\epsilon_i = k/(k+\exp(i/k))$ where $k \geq 1$ depends on the expected speed of convergence.

其中 exponential decay 之 k 選擇 0.999 而 Inverse sigmoid decay 使用 800，為求於總步數 12000 步時得到較合理的曲線 (見上頁第一列圖示)。全部都無使用 beam search。

使用 Linear decay 的表現與全程使用 0.5 的圖較為相近，但是由於 linear 前期使用了較大的 teacher forcing 值，所以觀察 training loss 下降的速度較快，相較於 exp 以及 inverse sigmoid 在 valid loss 以及 bleu score 的表現上也較為平滑。
Exponential decay 的表現則由於快速的下降 teacher forcing 的值使得 training loss 下降的速度較其他二者慢，而達到的 max bleu score 也較其他二者差勁。
而最後的 inverse sigmoid decay 則是結合了以上二者，前期使用較高的 teacher forcing 而後期則大幅降低之，讓模型在足夠的學習後也更能使用自己的產出做為下一步的預測，而由於 inverse sigmoid 中段有著十分劇烈的 decay 下降幅度，故可以看到表現出了與上述所有圖都相差十分明顯的特殊 training , valid loss 曲線，其中可以看到由於前期使用了大量的 teacher forcing 使得 valid loss 較高，但最後隨著足夠的學習，模型成熟後開始使用自身的產出，而最後同樣的也得到了不錯的 bleu score。

以上實驗超參都按照助教 tutorial，除了 report 要求題目外並無調整，如下：

```
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                # 最後輸出句子的最大
    self.num_steps = 12000                  # 總訓練次數
    self.store_steps = 300                  # 訓練多少次後須儲存
    self.summary_steps = 300                # 訓練多少次後須檢驗
    self.load_model = False                 # 是否需載入模型
    self.store_model_path = "./model_withTeachForceIS_attn"
    self.load_model_path = None             # 載入模型的位置 e.g
    self.data_path = "./cmn-eng"            # 資料存放的位置
    self.attention = True                 # 是否使用 Attention M

BEAM_SEARCH = False
BEAM_WIDTH = 3
TEACHER_FORCE_RATE = 0.5
```