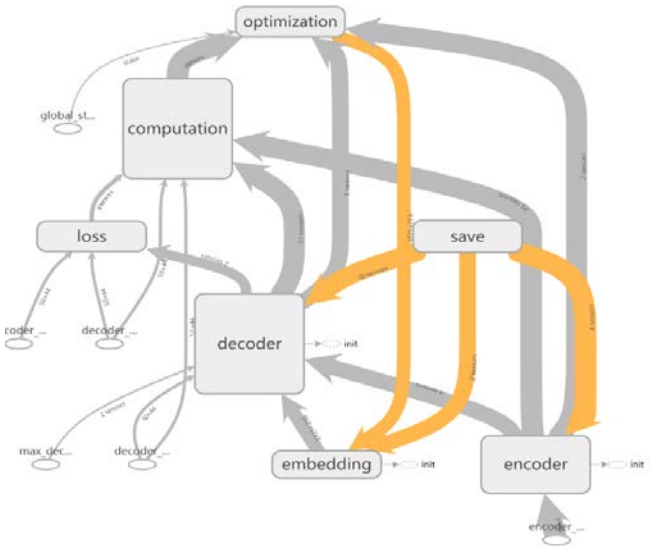
ADLxMLDS-hw2

孫奧 R05922147

1 Model description (2%)

模型采用的是一个encoder加上 decoder 的配置。这与論文中的 encoder 和 decoder 一起運算不同 ,如右圖所示 ,我們將 encode 和 decode 分為了兩個階段，即在 encode 完成以後 ,會把最後一個的 hidden state 的值作為 decoder 的初始，然後開始 decoder 的運算。這也是常用的翻譯系統的架構，這次實作很大程度上參考來 Google 在 github 上 NMT 的教程。



整個的流程大概如下：首先是 padding 整齊的視頻 input (當然不用 padding 可以用 dynamic_rnn 等來替代 ,不過本次作業正好每一段視頻都是 80 個 frame) ,然後丟個 RNN，最後得到最後一期的 hidden state。接著 , 連同<BOS>所對應的向量和這個 hidden state 交給另一個 RNN，可以想象<BOS>所對應的向量是這個 RNN 的 x，而這個 RNN 的 hidden state 與以前的全 0 不一樣，是前面得到的 hidden state，然後就會有一個輸出，這個輸出會被 match 到跟 vocab size 一樣長的的 vector(就是乘上一個矩陣 , 例如 decoder 每一時期的輸出是 100 個長度的向量，vocab size 是 1000，則需要乘上一個 100x1000 的矩陣 , 這個矩陣也會被訓練), 然後取出最大的(softmax)index , 然後從我們的 embedding 這個詞典里去找是那個詞以及那個詞所對應的向量是啥(這邊 embedding 的作用是這樣的：有 vocab size 個 row 以及自定義個數的 column，他將一個整數對應到一個向量，同時這

個整數也對應著一個單詞，所以這個向量就是這個單詞某種程度上的一個表示）當作下一期的輸入（這樣的話就一般的做法，效果很差，普遍的做法是用 ground truth 當作是輸入），然後再重複這個過程，然後 loss 用 cross entropy 來表示訓練。

2 Attention mechanism(2%)

2.1 How do you implement attention mechanism? (1%)

在 tensorflow 中的 seq2seq 中，有提供各種 attention 的 API，所以在 Implement 的時候並沒有什麼難度，如下。

```
self.attention_states = self.encoder_outputs
self.attention_mechanism = tf_seq2seq.LuongAttention(
    hidden_units, self.attention_states)

self.decoder_cell = tf.contrib.seq2seq.AttentionWrapper(
    self.decoder_cell, self.attention_mechanism,
    attention_layer_size=hidden_units)
```

在這邊選擇的是 Luong 的版本，下圖是 NMT 截取來的圖，首先是 Attention 的公式，即每一個 source(encode 階段的輸出)在給定現在 decoder 的 hidden state 時的權重是下面的α，然後用各個權重乘上各個 source 得到新的向量 c，最後 concat 向量 c 和 hidden state 作為一個新的輸入

$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \quad \text{[Attention weights]} \quad (1)$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad \text{[Context vector]} \quad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_\boldsymbol{c}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad \text{[Attention vector]} \quad (3)$$

而計算各個 source 權重的 score 函數有多個定義，這邊選用的是 Luong 的定義。

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_1 \boldsymbol{h}_t + \boldsymbol{W}_2 \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \quad (4)$$

2.2 Compare and analyze the results of models with and without attention mechanism. (1%)

比較下是 Attention 雖然看起來會更強大，但是更難 train 出好的模型，意思是，如果我所有架構不變，然後在上面加個 Attention 的機制，結果反而不會比較好，我不知道原因是什麼，不過我猜測可能是資料不夠多的原因，因為資料只有大概 1500 筆也就是就 1500 個不同的 hidden state（假設網絡固定），用 1500 個訓練一個 attention 可能還是不太夠，此外 1500 筆可能會導致訓練的視頻和測試的視頻不太一樣導致網絡 encode 的東西也很不像（比如 train 的都是運動，test 的都是動畫，也許 network 最後在 test 上 encode 出來的東西用運動上訓練的 Attention 效果會差很多）

3 How to improve your performance (1%)

3.1 Describe the model or technique (0.5%)

加上 dropout，在 encode 和 decode 階段都加上 dropout 以及減少 embedding 的 dimension，此外用 decay 的 learning rate，以及訓練所有的 captain。

3.2 Why do you use it (0.5%)

之所以用 dropout，是為了避免 overfitting，起先不使用的原因是總覺得這個很連續的過程，由於 dropout 會少掉資料害怕會訓練不起來。不過後來就發現根本不是這麼一回事，一開始沒有 dropout 的時候，我的 cross entropy 到過 0.3（只訓練一個 captain 的時候），這意味著每一個視頻是多有描述都對，而這顯然不可能，果然在做 test 的時候結果很大，所以 dropout 必須有。

之所以減少 embedding 的 dimension 是因為，希望藉由 embedding 也能學到一些詞先後的關聯，這些學得好也許語法上帶來的錯誤可以避免（不會出現 A man are 之類的句子）。

Learning rate 的 decay 是實驗出來的，因為一開始網絡 loss 降低的速度還是非常快的，所以一開始選 0.0001 這樣的下降的非常慢，但是後面快要收斂較大的 learning rate 會導致很不穩定。

訓練所有的 captain 是為了契合 blue 這個評分方式的。因為一開始的訓練所有 instance 的第一個 captain 會導致輸出的句子和第一個 captain 很像，但是自己觀察資料集會知道其實會差很大在 captain 之間的表達，我做了實驗，用助教給的 bleu 在訓練集上測試和第一個 captain 完全一樣的預測，結果大概是 bleu 在 0.35 左右，不過後來修正的 bleu 就幾乎 1 了，我覺得後來的比較合理。

4 Experimental results and settings (1%)

4.1 parameter tuning, schedule sampling

Score	schedule sampling	LR-DECAY	embedding	Hidden Size	Dropout
0.228	True	True	1000	256	0.8
0.277	False	True	256	256	0.8
0.265	False	True	512	256	0.8
0.288	False	True	128	256	0.7
0.276	False	False	1000	256	0.8