MLDS HW1 Report

組別：我才是真的Baseline
組員：f03942038鍾佳豪 / r05942102王冠驊 / d05921018林家慶 / d05921027張鈞閔

Our Environment

OS: Ubuntu 14.04.5 LTS,
CPU: Intel i7-5930K
GPU: GeForce GTX 1080 , GeForce GTX TITAN X
Libraries: tensorflow, numpy, pandas, argparse

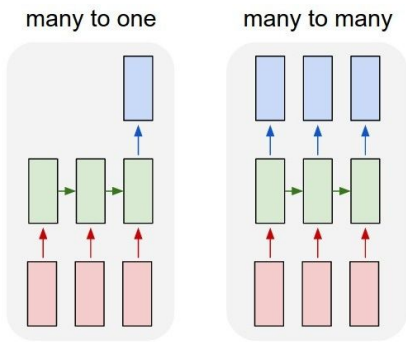
Data pre-processing

這次的 training data 包含了500多個檔案，每個檔案的內容及格式都不太一樣，也未經過完善的斷句的處理。為了方便，我們將所有的檔案都串在一起，並去掉"\n"，形成一個很長字串。

我們對此字串**建立字典**(vocabulary)，並依照出現頻率做排序。我們將出現頻率太低的字以"UNK"取代，得到指定 size 的字典。我們發現 vocabulary size 設定到40000時，才有能夠比較完善的包含 testing data 內出現的字，因此我們運用此設定。

接著再依照指定的 batch size 及 sequence length，將字串轉成大小為 [batch size*sequence length] 的矩陣，並當成我們輸入模型的資料。雖然這樣的做法不能保證每個 sequence 都是完整的句子，但是我們相信在資料很大量的時候，這不會造成太大的問題。

Model



為了實現 language modeling，我們先後實作了兩種RNN模型： many to one RNN 以及 many to many RNN。如右圖所示，many to one RNN 會輸入前幾個字，預測下一個字為何；many to many RNN 輸入一個句子，並在看到每個字時預測下一個字。我們將在 Part A 及 Part B 分別詳細介紹這兩個模型以及討論預測的結果。

[picture source: https://github.com/Vict0rSch/deep_learning/tree/master/keras/recurrent]

Part A: Many to one RNN

在實作中，我們設定模型看前四個字，去預測下一個字為何(sequence length = 5)。為了向助教提供的 baseline 看齊，我們先使用一樣的 configuration: 1 layer

LSTM with 256-D hidden state and standard softmax; AdamOptimizer with learning rate = 0.001; batch_size = 128。

Many to one RNN 在 kaggle 上得到的最好成績為**0.234** (public)以及**0.180** (private)，無法超過助教提供的baseline。因此我們開始思考問題的所在。第一，在我們的實作中，這個模型只學習到固定長度的字串(i.e. 過去的四個字)下一個可能會出現的字，但若考慮在一句較長的句子中，從這樣長度的字串獲得的資訊是不足的；第二，訓練這個模型的時候，誤差只會針對輸出的最後一個字計算，並再向前 propagate，導致模型更新的次數較少，且可能會導致 gradient vanishing 等問題，讓模型訓練效果不佳。因此，我們以下改用 Many to many RNN。

Part B: Many to many RNN

使用 Many to many 的架構，讓 RNN 可以在看到每一個字的時候，都去預測下一個字為何。換句話說，RNN 可以學會看見第一個字，預測第二個字；看見第一、第二個字，預測第三個字；看見第一、第二、第三個字，預測第四個字...以此類推，大大的擴展了 RNN 可以學到的東西。此外，在訓練時，我們可以在每一個 step 都計算誤差，並用來更新模型，減少了 gradient vanishing 等問題的發生。

我們使用以下的 configuration: 1 layer LSTM with 256-D hidden state and standard softmax; AdamOptimizer with learning rate = 0.002; batch_size = 50，並存下不同 epoch 時的模型，並比較效率。我們發現大概在 2 epochs 時 training loss 大概就到達穩定，在 kaggle 上最好可以得到 **0.396** (public)以及**0.430** (private)的成績 (vocabulary size = 40000)。

Model tuning

我們比較使用 one-hot encoding 以及 word embedding 的方式當成 RNN 的輸入。我們發現 one-hot encoding 的表現非常差，因此我們一律使用 word embedding 的方式。我們並沒有使用 pre-train 的 word embedding，而是直接在 RNN 前加入一層 randomly initialized embedding layer，並在訓練 language model 時，一起學習 word embedding。

針對Many to many RNN，我們嘗試增加 LSTM 的層數、 hidden state 的維度、以及使用 Dropout。然而，這些都會使得每個 batch 的訓練時間增加不少，training loss也沒有比較明顯的下降。也許增加模型的複雜度，必須就要更仔細的調整 learning rate，才有可能讓模型的準確度上升。

這次的 testing data 是類似克漏字的形式，在一段句子內會有一個空格，並提供五個選項，讓我們選擇。在文獻上應該有更符合這樣任務的模型，然而，我們還是針對這次作業的題目，使用 language model 來完成此任務。我們觀察到空格大多出現在一段句子的中間，而 language model 主要只能看空格前面的字，去預測空格可能會出現的字，並在從五個選項選出最有可能的選項(五個選項中，選擇機率最大的)。我們嘗試以下兩件事情，看是否能夠增加 language model 在這個任務上的準確度。

首先，我們統計在每筆testing data中的空格之前，大約會有10個字。因此，在模型訓練時，我們設定輸入的句子長度為10以及25，並比較差異。句子長度設定為

10時，其實是比較符合 testing data 的情況，這樣子可達到**0.384** (public)以及**0.373** (private)；而設定為25時，可達到**0.396** (public)以及**0.430** (private)。考慮越長的句子，或許能讓模型學到在句子之間(也就是在 hidden state 不是初始狀態的時候)字跟字之間的預測。

第二，在空格之後其實也還有很多字，但是我們都沒有用到，浪費了許多資訊。因此，我們嘗試考慮這些在空格之後的一個字(以下稱為 next word)來幫助我們預測空格內的字。詳細來說，我們先用空格前的字，去得到五個選項的機率值Pa, Pb, Pc, Pd, Pe，然後我們用空格前的字加上選項(有五種組合)，去得到 next word 出現的機率 Pa_next, Pb_next, Pc_next, Pd_next, Pe_next，最後再將兩個機率相乘當成該選項的最終機率，例如 Pa*Pa_next 為選項 a 的機率。我們發現這樣的方法其實帶來的改變不大，也就是說前面出現過的字才是影響最終機率的關鍵，也許我們可以嘗試考慮更多步之後的字，來看看後面的字是否能造成更大的影響。

Performance

我們最終使用 Many to many RNN 當成我們的 master 以及 best model，預設的參數為：

1 layer LSTM with 256-D hidden state and standard softmax;

AdamOptimizer with learning rate = 0.002;

vocabulary size = 40000; batch_size = 50; sequence length = 25;

並訓練 2 個 epoch，所花的時間約為 100 分鐘。在預測時，也只考慮空格之前的字，最終在 kaggle 上的成績為 **0.396** (public)以及**0.430** (private)。

Team Division

f03942038 鍾佳豪	Build model (Part B); Report
r05942102 王冠驊	Data preprocessing; Training model; Report
d05921027 張鈞閔	Build model (Part A); Report
d05921018 林家慶	Debug; Training model; Report