題目：Chinese QA

隊名：NTU_b03902013_YLC1208

組員&分工：

b03902013 吳克駿:方法三model

b03902084 王藝霖:方法一、二preprocess+方法二model

b03902093 張庭維:方法三preprocess

b03902101 楊力權:方法一、二preprocess+方法一model

For 方法一、二

一、Preprocessing:

(一)將簡體轉成繁體

data其實有部分是簡體字，而簡體字與繁體字語意近乎相同，但是以電腦的角度會將兩個相同的字當成不同的字，而語意判斷也有可能發生誤差。因此使用opencc套件將training與testing data都轉成繁體字。

(二)Word2Vector model

我們使用Gensim訓練一個word2vector的model，data使用維基的中文data，再加上training用的data，維基的data有1G多加上training data的800M幾乎可以排除testing時候OOV的問題。以下是訓練的細節

(1.)先斷詞：在訓練word2vector前，如何知道詞與詞的相鄰關係，斷詞是必須的，因此使用jieba的中文斷詞，先將data從文章或句子分成詞。

(2.)使用Gensim：使用維度400，min_count=5，用400維表示每個詞，且忽略小於出現次數太少的詞。

二、Models, Experiments, and Discussions

(一)方法一：

直接用word embedding找出答案。

因為要使用word2vector model，因此也需要用jieba分詞，細節與preprocessing一樣。以下是不同的嘗試：

(1.)我們初步猜測，針對一個題目，且答案在文章中，則答案的word embedding會與問題十分相近，因此將問題的word embedding做平均，再對文章中的每個詞直接做cosine similarity。

結果：Kaggle F1_score：0.02249
分析：顯然效果很差，是因為這個方法直接依靠word2vector training時的data set，句子與答案的語意關係，相近的詞其實就不是解答了，而是最接近問句的詞，因此幾乎不會是正確答案；而且整個文章的範圍太大，找出來的答案容易被干擾。就像是
問題：黃河流經9個省份最後會注入哪個海？
答案：河流

(2.)假設每個題目的答案皆在文章中，則大膽猜測文章中會有講到類似題目的敘述，而且在同一句話中，極有可能會提到答案。因此我們把文章切分成句子，對每個句子做word embedding的平均，也把問題的word embedding做平均，並且比較文章每個句子與問題的cosine similarity。選出的句子視為文章對問題句答案的一段敘述，而答案就在這個句子裡，因此把範圍縮到這個句子的每個詞，並用詞與問句做cosine similarity找出最近詞。
結果：Kaggle F1_score：0.12902
分析：效果有明顯進步，但是僅只有過simple baseline，後來發現在問答中，有很多不只一個詞的答案。

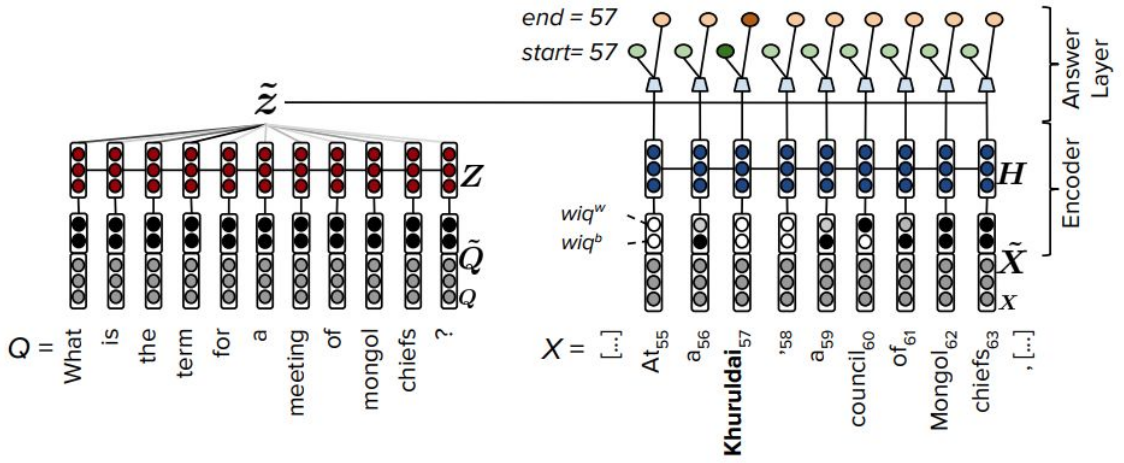
(3.)與(2)的假設相同，改成把找到的最相近句子，整個句子都當成答案。
結果：Kaggle F1_score：0.16528
分析：

(4.)改良(3.)把不重要的詞剔除，如何定義不重要的詞是關鍵，連接詞

(二)方法二：

使用 fastQA (ref: Making Neural QA as Simple as Possible but not Simpler) 方法，參考 github 上用 keras 實作的模型。

以下是 fastQA 的結構示意圖



fastQA 首先是將文字經過 embedding，接著使用 RNN (Bidirectional LSTM) 當作 encoder 來 encode 問題和文章。值得一提的是，問題和文章是使用相同的 RNN (shared weight)，除了一個地方不同，那就是最後的 RNN 結果 projection matrix 不同。如此一來便能省去其他提出的 model 所做的問題和文章的 word-by-word interation，論文中表示也能達到不錯的結果。

fastQA 特殊的地方在於 wiq，wiq 所代表的是 Context Matching，也就是把內容中的每個字有沒有出現在問題中也加進 feature 當中。分為兩個部分 weighted 和 binary，binary 指的是在 X（文章內容）當中，x（每個字）有沒有在 Q（問題）當中出現。而 weight 則是用兩個字的 word embedding dot 的結果再經過 softmax activation。如此一來便能用相對於其他所提出來的模型較為簡單的結構，達到也不錯的結果，並且有著更好的效率，能夠較快的訓練和達到收斂。

以下是 training 和 infering 實驗分別的細節：

(1.) training

使用如一、preprocessing 所述之方法得到文章和問題的 word embedding。再者由於句子和問題分別長度不一，因此有使用 padding 來對齊。padding 的長度選為平均的長度再多一些些，過長的會截掉，過短的會補零。使用的 optimizer 為 adam，batch_size = 32。

模型的細節請見報告最後所附的圖。

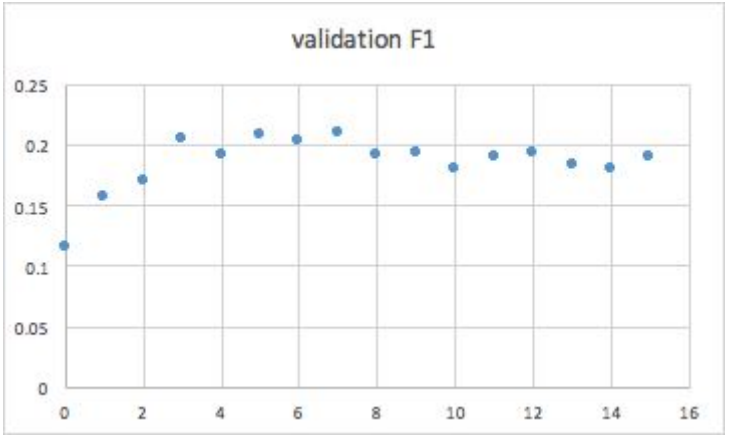
(2.) infering

因為在 fastQA 的 answer layer 中，model 的目的是要學到文章中，對於每個問題每個字是答案開頭 start 的機率，和答案結尾 end 的機率。

最後要預測回答時，尋找一對 start 和 end 的組合使得機率最高，但必須滿足 start < end 的條件。

此外，因為 model 並沒有學得非常好，可能會有回答長度過長的問題，如此會造成 precision 比較低。因此我們使用 rule base 的方式來找較好的答案回答。在長度 > 10 時，改用 start 或 end 其中較高機率的位置附近的一句話來作為回答。

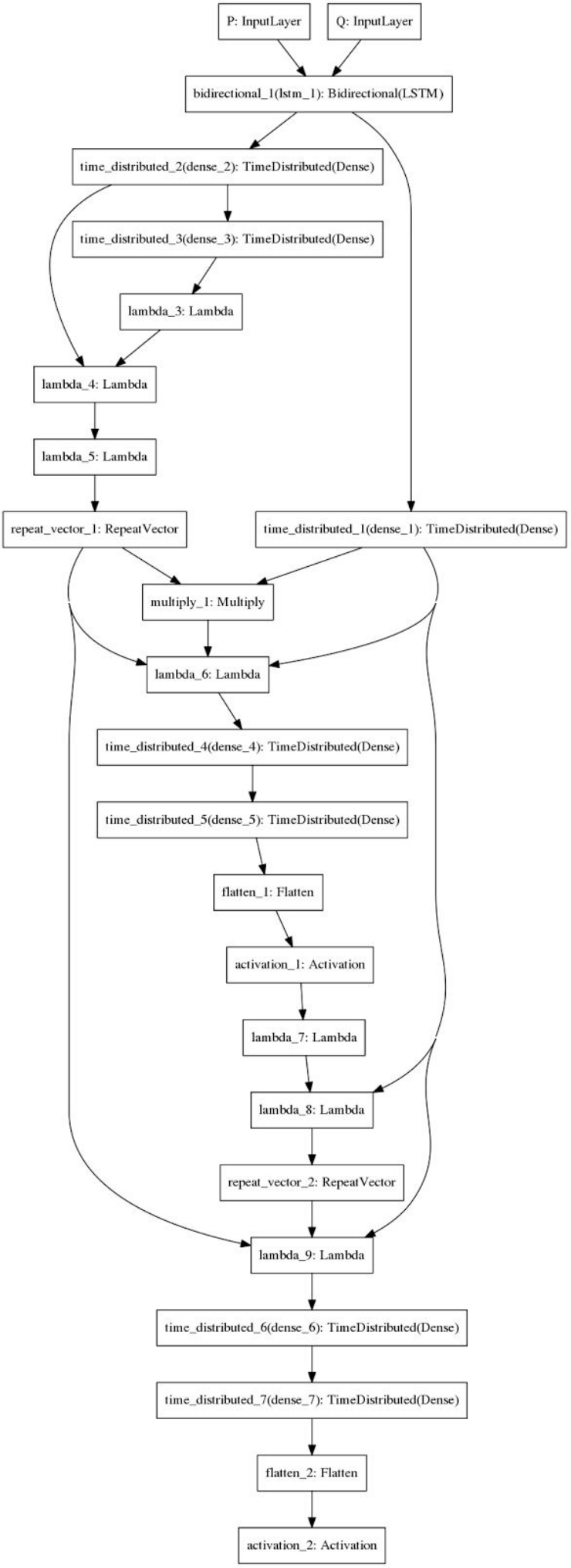
實驗結果討論：



Kaggle F1_score = 0.21716

因為我們使用 keras，有些部分只能用近似的方法，例如其中的 attention 部分。另外

(三) ensemble 以上兩種方法：


P: InputLayer
Q: InputLayer
bidirectional_1(lstm_1): Bidirectional(LSTM)
time_distributed_2(dense_2): TimeDistributed(Dense)
time_distributed_3(dense_3): TimeDistributed(Dense)
lambda_3: Lambda
lambda_4: Lambda
lambda_5: Lambda
repeat_vector_1: RepeatVector
time_distributed_1(dense_1): TimeDistributed(Dense)
multiply_1: Multiply
lambda_6: Lambda
time_distributed_4(dense_4): TimeDistributed(Dense)
time_distributed_5(dense_5): TimeDistributed(Dense)
flatten_1: Flatten
activation_1: Activation
lambda_7: Lambda
lambda_8: Lambda
repeat_vector_2: RepeatVector
lambda_9: Lambda
time_distributed_6(dense_6): TimeDistributed(Dense)
time_distributed_7(dense_7): TimeDistributed(Dense)
flatten_2: Flatten
activation_2: Activation

For 方法三

一、Preprocessing/Feature Engineering:

由於jieba對地名的切字有些不太理想(EX:廣/州市&廣州/市)，且facebook的pre-trained model 似乎沒有數字的data，且將文章切詞過後，對於字詞在原文中的index搜索也是一大挑戰，因此在方法三中，我們不做任何的切詞處理，直接將原文含標點符號一個一個丟入gensim裡面去train word embedding ，確保training data皆可在此embedding model內搜索到。

gensim 訓練細節：

dimension = 64, windows=5, min_count=1,iteration=15

而由於方法三使用的是R-NET的架構，因此在preprocess上，幾乎都是以原作者所使用的架構去建造：

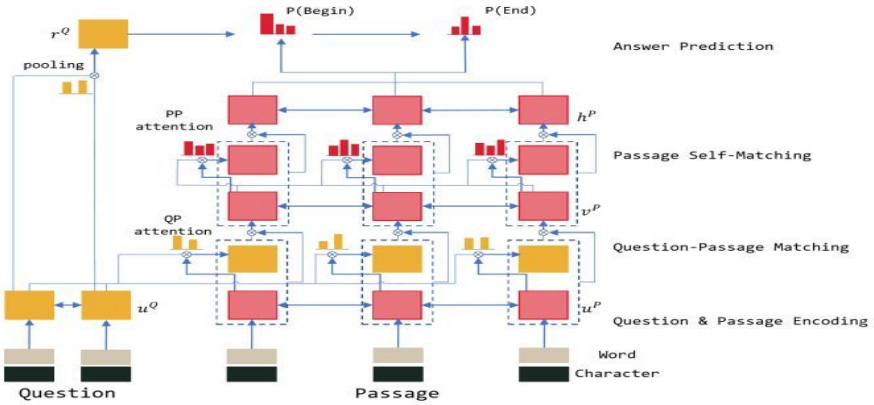
Data

- training data
    - training data:n*64 float
    - training question:n*64 float
- training answer
    - answer start index:int
    - answer end index:int

testing data的package，則是把training answer部分捨棄掉，即可得到。

二、Models, Experiments, and Discussions

在助教的提示下，我們也利用的在SQuAD dataset中表現出色的R-NET model，但由於原本的範疇皆是在英文下去操作，並且有相當強大的stanford coreNLP package 來使用，因此如何將此model轉換成中文來使用，讓我們傷透了腦筋。



上圖是R-net model大致上的架構，就如同人類要回答QA的流程一樣，首先必須看懂文章、題目(encoding)，在來將文章與題目match起來，找到題目大概會在文章的哪一段，最後則是挑出答案機率最高的片段，找出答案開始的位置，及答案結束的位置。

但中文最大的劣勢在每個詞的字數並不相同，對斷詞、尋找index來說有著極大的難處，因此為了簡化問題，我們就以每個字去作單位來train。但此方法會使context data的長度到五百多，甚至近千字，現有的資源並不足以train這麼龐大的data，因此退而求其次，將QA的範圍限制在小範圍，也就是以句號為分界，從包含題目的句子裡找出答案。而獲得句子的方式非常簡單，就是從題目每個字中去跟context的每個句子去比對，hit到數量最高的就是我們認為所含題目的句子。

而在predict testing data部分，我們也試了兩個方法，第一是將整個文本丟下去給他predict，但做出來結果相當差，有許多answer start在answer end後面，也有的是根本predict不出來，直接給index 0當答案，也有的是答案長達100多字，從常理判斷根本不可能是答案的情形，因此我們從中挑選出比較有可能的答案，空缺的部分則與前面兩個方法去做ensemble，得出來結果卻比原先還要差，kaggle score在0.18左右。

因此第二個方法我們也是將test context去做分段，也是用土法煉鋼，將question跟文本的內容去match，match到最多字的就是我們的文本精華，可能由於我們在training時即是短文本的訓練，且training start&end accuracy可以到90%以上，valid data start&end accuracy也大約在55%上下，因此在testing時，kaggle score可以到0.44，雖然不太好，但跟前面的方法比起來有大幅度的躍進。

這個現象我把它解讀成像是小朋友的學習，使用有限且簡單的字彙(超過一個字就算難)，從簡單的data去學習，沒辦法一次負荷太多的文本(GPU不足不能train太大的data)，因此在考試的時候自然沒辦法出太長的題目，這樣小朋友會看不懂，就像我們的model看不懂整個文本一樣，就給你亂猜答案；但如果我們將考試題目內容縮短，小朋友會寫了，自然會好好的作答，答題也就有一定的水準。

但如果答案並不在我們預先切好的句子內，自然也不可能predict出正確的答案，這也是我們一開始使用整個test context最大的原因，因此這一點是我們future work最大的挑戰：如何濃縮整個文本並且對每一段的語意去做分析，找出最適的分段點。再來則是字與字之間的關聯性不足，最終還是要回到詞與詞才能得到更好的效果，因此我們需要做出更縝密的字詞index系統，拿詞去train，最後再依詞的index去找到字的index，得到更準確的答案。

training 細節：

word_dimension:64 hidden_dim:75 epochs:90 dropout:0.1 learning rate:1

optimizer:adadelta loss:categorical cross entropy batch_size:70