Report - ADL HW1

網媒所碩一 R11944026 柯婷文

Q1. Data Processing

建立字典

我使用助教提供的sample code來建立字典(vocabulary)。
首先，用 `json.loads()` 去讀 `train.json` 、 `eval.json` 後，會各自讀出一個list of dictionary，每個dict都是一筆data，欄位包含 `text` (存一個由許多token組成的句子), `intent` (這個句子的目的), 跟 `id` 。

而我們用一個名為 `intents` 的set來存出現過的intent，由於set的特性，不同的intent只會被紀錄一次。之後幫每個intent編號，做成dict，寫入 `intent2idx.json` 。

再來，需要計算每個token出現的次數以便做成Vocab。這裡我們先將所有data的 `text` 都用 `split()` 切分成許多token，再把他們放進一個list。之後把這個蒐集了 `train.json` 、 `eval.json` 裡所有token的list放進一個叫 `words` 的 `Counter` ，計算每個token出現的次數，並使用 `most_common()` 取出最常出現的 `vocab_size` 個token，放在一個叫 `common words` 的set。接著定義一個class `Vocab` ，他的第0個字跟第1個字分別是unknown及padding token，剩餘部份放進我們剛剛取出的 `common words` 。同時這個 `Vocab` 的class也定義了 `tokens` 、 `encode` 、 `encode_batch` 等property function，方便我們取得所有token、將一串token轉成id、將一個batch的token轉成id。如此一來，我們便可以利用此字典查找token對應的id，但要注意的是，那些沒被列在 `common words` 裡的token將會被視為unknown來處理。

Slot tagging的部分也使用幾乎一樣的方式進行。

進行 Tokenize

建立好字典後，便是要將token轉換為id，這部分主要寫在 `dataset.py` 裡各自的 `collate_fn` ，也就是處理一個batch的data的部分。

Intent Classification

先將每個句子(text)用 `split()` 切成許多token，而每個batch又是由多個句子組成，所以切完之後會是List[List[str]]。接著使用 `vocab` 裡的 `encode_batch` 幫我們把整個batch的token轉換成id，此時資料型態由List[List[str]]轉為List[List[int]]。Intent的部分先把 `intent2idx.json` 讀出來轉換成dict，當

作mapping，再將整個batch的intent取出(一筆data有一個intent)，再轉換成idx (List[int])。
值得注意的是， `encode_batch()` 會以batch裡面最長的seq_len為基準，幫我們做padding，所以不用額外做。

Slot Tagging

句子處理的部分跟Intent Classification一樣，較不同的是tag的處理。slot tagging任務裡每個token都會有一個tag，所以整個batch裡的每筆data會有不同數量的tag。因此除了用mapping將tag轉換成編號，還使用了 `utils.py` 裡的 `pad_to_len` 做padding。而Padding value要設多少呢？原有的tag編號是編到num_class - 1(因為0-index)，所以我將padding value設為num_class。如此一來，經過模型預測的tag值將會介在0~num_class。但為了不考慮這些padding位置的預測值，我還用了一個 `ignore` 的matrix紀錄padding的位置，有pad的地方會=1。

Embedding

使用GloVe提供的 `glove.840B.300d.txt` 這個已經pretrain好的embedding，他是用了Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download)的版本。每個token丟進去後會轉換成300-d的向量，而對於不在GloVe裡的token，其每個element的值將隨機設為random()*2-1。

Q2. Intent Classification

我使用LSTM的模型，參數分別設定為hidden_size(512), num_layers(2), dropout(0.3), bidirectional(True)。
先將batch data轉換為一個embeds的embedding，此時維度是seq_len x batch_size x num_class*2，然後放進LSTM。

$$lstm_{out}, (h_{n,}) = LSTM(embeds)$$

$lstm_{out}$是LSTM每個時間點的output，一個seq_len x batch_size x 2*hidden_size 的tensor。而h_n則為最後一個時間點的所有hidden states，維度是2 * num_layer x batch_size x hidden_size，我們用最後一個時間點出來的結果來當整筆data的預測值。

為了等等計算attention，將top-most layer的forward和backward的output concat起來。

$$h_n = concat(h_n[-1], h_n[-2])$$

得到h_n的維度是batch_size x hidden_size*2。

接著，就可以把$lstm_{out}$的維度轉換成batch_size x seq_len x 2*hidden_size，去計算Attention。也就是在比較需要關注的地方給予比較高的權重，計算加權後的分數。

$$attn_{out} = attention(lstm_{out}, h_n)$$

此時維度是batch_size x hidden_size*2。
再接上fully connected layer，幫助我們判斷每筆data的判斷類別。

$$out=FC(attn_{out})$$

如此一來便得到一個batch_size x num_class的輸出，裡面每筆data被分到每個類別的機率分布。之後用cross entropy計算跟正確答案(label)間的loss。

$$batch\_loss=CrossEntropy(out,label)$$

我使用Adam作為optimizer，learning rate設為0.2，batch size是256。

最終得到的public score為0.91688。

Q3. Slot Tagging

我使用LSTM的模型，參數分別設定為hidden_size(512), num_layers(2), dropout(0.3), bidirectional(True)。
先將batch data轉換為一個embeds的embedding，此時維度是seq_len x batch_size x num_class*2，然後放進LSTM。

$$lstm_{out}, \_=LSTM(embeds)$$

$lstm_{out}$是LSTM每個時間點(也就是每個token)的output，一個seq_len x batch_size x 2*hidden_size 的tensor。這裡我們保留所有時間點的輸出，因為每個token都需要預測出一個類別。

再接上fully connected layer，幫助我們判斷每筆data的判斷類別。

$$out=FC(lstm_{out})$$

如此一來便得到一個batch_size x seq_len x (num_class+1) 的輸出，裡面每筆data被分到每個類別的機率分布，這邊num_class有+1是因為padding的tag也會被視為一個類別。之後用cross entropy計算跟正確答案(label)間的loss。

$$batch\_loss=CrossEntropy(out,label)$$

我使用Adam作為optimizer，learning rate設為0.2，weight_decay設為0.0001，batch size是16。

最終得到的public score為0.78659。

Q4. Sequence Tagging Evaluation

令TP(True Positive), TN(True Negative), FP(False Positive), FN(False Negative)
seqeval提供四個指標，分別為

- Precision(準確率) = TP/(TP+FP)
- Recall(召回率) = TP/(TP+FN)
- F1-score = 2 * Precision * Recall / (Precision + Recall)
- Support = 該tag實際出現的次數

而下方的幾個平均數分別代表

- micro avg: 將所有tag的TP都加起來後，若是算precision就除以(TP+FP)的總數；若是算recall就除以(TP+FN)的總數。
- macro avg: 每個類別各自計算自己的precision、recall後，取平均。
- weighted avg: 每個類別各自計算自己的precision、recall後，根據每個類別的樣本多寡，取加權平均

而兩種accuracy分別代表

- Joint accuracy: 以句子為單位，一個句子裡所有token都被預測正確才算對。計算正確句子占所有句子的比例。
- Token accuracy: 以token為單位，計算正確的token占所有token的比例。

也就是說，在token accuracy相同的情況下，若錯的token都集中在同一句話，joint accuracy可能還是可以不錯；但若錯的token分布在不同句話，將導致joint accuracy很低。

Q5. Compare with different configurations

參數調整

在Slot tagging的部分，因為觀察到資料筆數並不多，所以我先將batch_size調到16。然而在epoch=40的情況下，都有發生Training loss穩定下降，validation loss卻往上爬升，很明顯的overfit了。所以我加入L2 Regularization，將optimizer的weight_decay設為0.0001，發現這麼做可以顯著降低validation loss，然而也不能設太小，否則Accuracy又會降低。此外我也實驗了將LSTM改用GRU或RNN的結果。
以下為batch_size(16), epoch(40)的實驗結果

RNN-based layer	Weight decay	Dropout	Train acc	Val acc	Test acc
LSTM	0	0.3	0.993236	0.801000	0.72064
LSTM	0	0.4	0.995030	0.788000	0.72493
LSTM	0.0001	0.3	0.895638	0.790000	0.78659
LSTM	0.0005	0.3	0.809221	0.752000	0.73780
LSTM	0.00005	0.3	0.937051	0.758000	-
GRU	0	0.3	0.856295	0.781000	-
RNN	0	0.3	0.929183	0.757000	-

後來發現batch_size設這麼小容易overfit，所以將batch_size改回128。以下為batch_size(128), epoch(60)的實驗結果

RNN-based layer	Weight decay	Dropout	Train acc	Val acc	Test acc
LSTM	0.0001	0.3	0.918001	0.796000	0.79517
GRU	0.0001	0.3	0.934567	0.791000	-
RNN	0.0001	0.3	0.921452	0.784000	-

仿照Intent classification的結果，在embedding layer後多加了一層Dropout layer的結果

RNN-based layer	Weight decay	Dropout	Train acc	Val acc	Test acc
LSTM	0.0001	0.3	0.869685	0.802000	-
LSTM	0.0001	0.6	0.876311	0.808000	0.80911

Bonus

在Intent classification中，嘗試加入attention的機制，讓句子的不同部分對其他部分可以有不同程度的關注。也調整不同的dropout rate，觀察實驗結果。

Attention Layer	Dropout	Batch size	Epoch	Train acc	Val acc	Test acc
沒有	0.1	128	100	1.000000	0.902333	-
沒有	0.15	128	100	1.000000	0.911667	0.90044
沒有	0.2	128	100	1.000000	0.908667	-
有	0.1	128	100	1.000000	0.929333	0.87733
*有	0.3	128	100	1.000000	0.925333	-
有	0.3	15	100	1.000000	0.910667	-
*有	0.3	256	100	1.000000	0.927333	0.91688
有	0.3	256	20	0.997400	0.916000	-
有	0.3	256	10	0.993333	0.911000	-

發現加了Attention後表現有變好。為了解決overfit，也嘗試在Embedding後/FC層前額外加入了Dropout Layer
(Batch size = 128)

Dropout Layer(Embed後/FC前)	Dropout	Epoch	Train acc	Val acc	Test acc
X	0.3	100	1.000000	0.925333	-
X/O	0.3	80	1.000000	0.924000	-
X/O	0.4	80	1.000000	0.925000	-
O/O	0.4	80	0.996400	0.932000	-
O/O	0.6	80	0.994467	0.944667	0.92577

(Batch size = 256)

Dropout Layer(Embed後/FC前)	Dropout	Epoch	Train acc	Val acc	Test acc
X	0.3	100	1.000000	0.927333	0.91688
X/O	0.3	100	0.999733	0.927667	-
X/O	0.4	100	1.000000	0.925667	-
O/O	0.4	80	0.998600	0.937667	0.92888
O/O	0.6	80	0.995600	0.943000	-

結果發現在Embedding layer後加一層dropout layer的效果很好，都可以讓val acc提升0.1左右，也讓模型較不容易overfit。