Report

Q1：Data processing (2%)

- The pre-trained embedding I used：`glove.840B.300d.txt`

- How do you tokenize the data：
  - In "`preprocess_intent.py`" (sample code)：
    1. 使用 set() 建立 `intents`, 使用 `Counter()` 建立 `words`
    2. 利用 `set()` 和 `Counter()` 的 `update()` 函數，讓重複的內容只會被記錄一次，分別抓出 "共有多少種 intent" → 存入 `intents` 和 "訓練資料集中所使用到的所有單字" → 存入 `words`
    3. 利用自動編號 `enumerate()` 幫 `words` 內所有的單字編號即可得到第一個檔案 `intent2idx.json`
    4. 使用 `most_common(n)` 篩選 `words` 中 最常出現的前 n 種單字作為 token，雖然 token 應具有代表性的之意，但 sample code 預設 `vocabe_size = 10000`，而 `words` 裡面只有 6489 個字，因此在該條件下，全部單字都被當成 token

5. 將 `most_common(n)` 取出的 token 放入事先寫好的 `Vocab(class)` 中幫單字進行編號，並存入 `vocab` 中，以便稍後與 `glove.840B.300d.txt` 進行比對，並使用`pickle.dump()`將`vocab`封裝起來，得到第二個檔案`vocab.pkl`
6. 讀取 `glove.840B.300d.txt`裡的內容，glove的組成方式是 "一個單字 + 該單字的向量（ Dim = glove.840B.**300d** ）"，利用 python 本身的資料結構`Dict`將每個字作為查詢的 key ，pre-trained vector 作為 value 存起來 → 存入 `glove[str: List[float]]`
7. 從`glove[str: List[float]]`中查詢收錄在`vocab`裡所有單字的 vector，如果 glove 有收錄，那就直接把 pre-trained vector 拿來使用，如果沒有收錄則使用 `random()` 隨機產生一組 [-1,1] , Dim = 300 的一維向量 將這些 vector 按照 vocab 裡的編號儲存在 `embeddings`中，裡用`torch.save()`把`embeddings`封裝起來，得到第三個檔案`embeddings.pt`，size [6489, 300]

- In "`IntentClsDataset(Dataset)`"：
  1. `self.data = json.loads(data_path.read_text())` ：以讀入訓練資料
  2. `def __getitem()__(self, index):`：index 是 `dataloader(class)`要拿第 i 筆資料的意思，根據`batch_size`決定一次要拿幾筆，所以`__getitem()__`要設計成一次 return 一組"訓練資料" [text, intent]
     a. `self.idx2label = {idx: label for label, idx in self.label2idx.items()}`：利用原本已經有的`intent2idx`產生 index to intent
     b. `sentence = self.data[index]["text"].split()`：把整個句子依單字分割，`data[index]["text"]`是第 i 筆句子（ text ）
     c. 複製一個`sentence`取名叫`sentence_in_index`：方便進行單字跟數字調換
     d. `sentence_in_index[i] = self.vocab.token_to_id(sentence[i])`：利用`vocab.token_to_id()`將分割好的單字依序替換成數字
     e. 因為送入 RNN model 的句子不能忽長忽短，因此要將句子都補到一樣的長度（最大長度`max_len = 128` ）
     `while len(sentence_in_index) < self.max_len:`

`sentence_in_index.append(0)`

f. `intent = self.data[index]["intent"]`：同 2. 提取第 i 筆 intent

g. `intent_in_index = self.label2idx[intent]`：藉由`preprocess.sh`所得到的 `intent2idx`將 intent 轉成對應的編號

h. `return sentence_in_index, intent_in_index`：return 以數字表達的句子（text） 和 intent

Q2：Describe your intent classification model. (2%)

a. model

   a. h_0 = torch.Size([4, 256, 512])：initialize a t-1 hidden_layer to begin, all element in h_0 is zeros, and c_0 is same as h_0 but auto create by pytorch

   b. embedding = embedding_layer(input)：input passing to embedding_layer to replace word-index to word-embedding vector

   c. lstm_out, hidden_n = lstm(embedding_layer, (h_0, c_0))：passing all word-embedding vector through model, h_t-1 and c_t-1 start with (h_0,c_0) and auto pass to the end become (h_n, c_n). For one lstm cell, h_t, c_t = LSTM(w_t, h_t-1, c_t-1), where w_t is a single word-embedding vector and h_t, c_t will become h_t-1, c_t-1 for next word/moment of the t-th token.

   d. output = fc(lstm_out)：lstm_out is a list store all h_t in each moment, passing through a Linear_layer will map its dimension from h_t to num_classes

   e. output = logsoftmax(output)：passing to softmax to get distribution of probability, and its sum equals to one, and use a log function to make big_num bigger & small_num smaller

f. output = output[:,-1,:]： pick up the last h_t(h_n) to represent whole sentence

b. performance on kaggle：**Score: 0.89244**

c. loss function：CrossEntropyLoss()

d. optimization algorithm：Adam（learn_rate=1e-3）, batch_size=128

Q3：Describe your slot tagging model. (2%)

來不及做完，但概念應該會參照Q2，因為 lstm_output 存的是每個時間點下產生的 h_t，而且資訊會一直往下一個時間點傳遞，因此各時間點的 predict tag 也可能因為先前幾個單字的變化而產生改變，因此把 Q2. a-c. 的 lstm_output 裡的每個 h_t 各自接上一個 Linear_layer ，接著套用 softmax，最後進行 predict，應該就可以得到正確的 predict ，跟 HW1 最後 "Guides" 裡提示的一樣。

Q4：Sequence Tagging Evaluation (2%)

因為Q3沒做完，所以Q4也沒答案，老師、助教抱歉QQ

Q5：Compare with different configurations (1% + Bonus 1%)

因為GPU 是筆電版1050Ti ，VRAM只有4GB基本上調什麼都會 run out of GPU memory，想調其他參數batch_size又只能開很小，但是發現 batch size 開小跑出來的 model 成果都不太好，所以最後是稍微調大一點 hidden size 才跑過 base line。

第一次過 base line，batch_size=64，hidden_size=768，後來考慮記憶體太容易炸開的問題（前前後後也試了蠻多組合，都還是直接不能動），只好試看看 batch size 調大一點點會不會 backpropagation 的計算結果會比較好，因此多用 batch size=128，hidden_size=768 重跑一次，Local train/eval Accuracy 從 88.87% 提升至 90.38%，Kaggle Accuracy 從 88.000% 提升至 89.466%，稍微有點進步。

雖然沒辦法往上調，但是在用LSTM之前，一開始有試過RNN，但是效果也是不太好 Accuracy 最高也是只有 77% 左右，不排除是因為我的 model 太簡單，但是前面真的研究太久所以後來沒什麼時間再測其他玩法了，後來用LSTM的時候因為考慮到電腦的問題，有拿去 colab 跑跑看 batch_size=256，hidden_size=1024，跑出最高分的 Public Score=90.577%，雖然是免費的colab但也是一定比我的電腦強，看來有好的硬體真的蠻重要的，不僅高分還可以省時，因此有考慮下次作業要買 colab-pro，不然顯卡真的太貴買不下手，嗚嗚。

- origin：
  - batch_size = (64)
  - hidden_size = (768)
  - num_layers = 2
  - dropout = 0.1

    - bidirectional = True
    - num_classes = 150

> best_result @ epoch 5 (5 in 1:100):
> best_avg_acc = 88.87%
> best_avg_loss = 0.4735032820955236

→ Kaggle Public Score = 88.000%

- improve：
    - batch_size = (128)
    - hidden_size = (768)
    - num_layers = 2
    - dropout = 0.1
    - bidirectional = True
    - num_classes = 150

> best_result @ epoch 11 (11 in 1:100):
> best_avg_acc = 90.38%
> best_avg_loss = 0.4650507140904665

→ Kaggle Public Score = 89.466%