Q1: Data processing

1. Describe how do you use the data for intent_cls.sh, slot_tag.sh. If you use the sample code, you will need to explain what it does in your own ways.

   (a) How do you tokenize the data.

   - Intent Classification
     i. Split each text sentence by " " into tokens.
     e.g. "what is our purpose" → ["what", "is", "our", "purpose"]
     (This is performed in *collate_fn* of *SeqClsDataset*)
     ii. Encode the tokens into indices with *vocab*.
     (This can be done with *vocab.encode_batch()*)
     iii. The token indices are transformed into corresponding vectors.
     (This is done during model forwarding by calling *x=self.embed(x)*)
   - Slot Tagging
     i. Encode the tokens into indices with *vocab*.
     (This can be done with *vocab.encode_batch()*)
     ii. The token indices are transformed into corresponding vectors.
     (This is done during model forwarding by calling *x=self.embed(x)*)

   The data is tokenized during preprocessing.

   (b) The pre-trained embedding you used.
   The embedding for tokens in this homework is GloVe. With the pretrained vector representations for words, it would be easier for models to grasp the meanings of the tokens.

Q2: Describe your intent classification model.

1. Describe

   (a) your model
   nn.GRU:
   hidden_size: 512
   num_layers: 2
   dropout: 0.2
   bidirectional: True

   classifier:
   nn.Dropout(0.2)
   nn.Linear(self.encoder_output_size, self.num_class)
   nn.Softmax()

   Idea of the *forward* function:

   i. $x = self.embed(x)$
   Transform indices to vector representations.

   ii. $, h_n = self.rnn(x)$
   Run the rnn model to get the final hidden states for the sequence.
   h_n is of shape: $(2 * num\_layers, batch\_size, hidden\_size)$. Then permute the first two dimensions and reshape h_n into $(batch\_size, 2 * num\_layers * hidden\_size)$

   iii. $pred\_logits = self.classifier(h_n)$
   $pred\_logits$ is of shape $(batch\_size, num\_class)$. $pred\_logit[i][j]$ represents how much the model believes that the i^{th} text corresponds to the j^{th} intention.

   (b) performance of your model. (public score on kaggle)
   Intent Classification: 0.92533

   (c) the loss function you used.
   The loss function is *BCELoss*. It is the loss between *pred_logits* and real tags.

   (d) *Adam*, *lr=1e-3*, *batch_size=128*.

Q3: Describe your slot tagging model.

1. Describe

   (a) your model
   nn.GRU:
   hidden_size: 384
   num_layers: 2
   dropout: 0.7
   bidirectional: True

   cnn(nn.Conv1d):
   dropout(0.7)
   kernel_size=11
   padding=5

   classifier:
   nn.Dropout(0.7)
   nn.Linear(self.encoder_output_size, self.num_class)
   nn.Softmax()

   Idea of the *forward* function:

   i. $x = self.embed(x)$
   Transform indices to vector representations.
   ii. $x = self.cnn(x)$
   Let x go through a layer of CNN before running RNN.
   iii. $x, h_n = self.rnn(x)$
   Run the model to obtain the outputs of GRU.
   iv. $pred\_logits = self.classifier(x)$
   $pred\_logits$ is of shape $(batch\_size, seq\_len, num\_class)$. $pred\_logit[i][j][k]$ represents how much the model believes that the i^{th} text's j^{th} token corresponds to the k^{th} tag.

   (b) performance of your model. (public score on kaggle)
   Slot Tagging: 0.78713

   (c) the loss function you used.
   The loss function is *BCELoss*. It is the loss between $pred\_logits$ and real tags.

   (d) *Adam*, *lr=1e-3*, *batch_size=128*.

Q4: Sequence Tagging Evaluation

Reference: http://nathanlvzs.github.io/Several-Tagging-Schemes-for-Sequential-Tagging.html

- Please use seqeval to evaluate your model in Q3 on validation set and report classification_report(scheme=IOB2, mode='strict').

	precision	recall	f1-score	support
date	0.75	0.77	0.76	206
first_name	0.94	0.96	0.95	102
last_name	0.88	0.87	0.88	78
people	0.74	0.76	0.75	238
time	0.85	0.89	0.87	218
micro avg	0.81	0.83	0.82	842
macro avg	0.83	0.85	0.84	842
weighted avg	0.81	0.83	0.82	842

Classification Report

- Explain the differences between the evaluation method in seqeval, token accuracy, and joint accuracy.
  seqeval:
  - tags explanation:
    - "B-": Begin
    - "I-": Intermediate
    - "E-" / "L-": End/Last
    - "S-" / "U-": Single/Unitary (It is used for single words)
    - "O-": Other
  - modes:
    - default
      Mistagging of "B-", "I-", "E-" but with correct label is fine.
    - strict
      Mistagging of "B-", "I-", "E-" is regarded as wrong tagging.
  - schemes:
    - IOB1
      Tag "I-" for labeling and "O-" for others. "B-" is used to identify subsequently concatenated labels.
    - IOB2
      Tag "I-" for labeling and "O-" for others. "B-" is used to identify each starting character in a label.

* IOE1
  Tag "I-" for labeling and "O-" for others. "E-" is used to identify the end when a subsequently concatenated labels exist.
* IOE2
  Tag "I-" for labeling and "O-" for others. "E-" is used to identify each ending character in a label.
* IOBES(only in strict mode) Tag starting("B-") and ending("E-") characters. "S-" is for single words.
* BILOU(only in strict mode) Tag starting("B-") and ending("L-") characters. "U-" is for single words.

```
* Example:  Bill   works for  Bank   of     America and takes the Boston Philadelphia train.
* IO:       I-PER  O     O    I-ORG  I-ORG  I-ORG   O   O     O   I-LOC  I-LOC        O
* IOB1:     I-PER  O     O    I-ORG  I-ORG  I-ORG   O   O     O   I-LOC  B-LOC        O
* IOB2:     B-PER  O     O    B-ORG  I-ORG  I-ORG   O   O     O   B-LOC  B-LOC        O
* IOE1:     I-PER  O     O    I-ORG  I-ORG  I-ORG   O   O     O   E-LOC  I-LOC        O
* IOE2:     E-PER  O     O    I-ORG  I-ORG  E-ORG   O   O     O   E-LOC  E-LOC        O
* BILOU:    U-PER  O     O    B-ORG  I-ORG  L-ORG   O   O     O   U-LOC  U-LOC        O
* SBEIO:    S-PER  O     O    B-ORG  I-ORG  E-ORG   O   O     O   S-LOC  S-LOC        O
* SBEIO is also called IOBES
```

An example for different tagging schemes.

– metrics:
TP: True Positive, TN: True Negative, FP: False Positive, FN: False Negative

* precision: $\frac{TP}{TP+FP}$
* recall: $\frac{TP}{TP+FN}$
* f1-score: $\frac{2*(precision*recall)}{precision+recall}$
* support: $TP+FN$
* macro avg: take average directly
  e.g. $precision_{macro\_avg} = \frac{1}{N}\sum_i precision_i$
* micro avg: sum up and evaluate.
  e.g. $precision_{micro\_avg} = \frac{\sum_i TP_i}{\sum_i TP_i + FP_i}$
* weighted avg: average with support
  e.g. $precision_{weighted\_avg} = \sum_i (precision_i * \frac{support_i}{\sum_j support_j})$

joint accuracy: $\frac{\#CorrectSentences}{\#TotalSentences}$

token accuracy: $\frac{\#CorrectTokens}{\#TotalTokens}$

Q5: Compare with different configurations

Please try to improve your baseline method (in Q2 or Q3) with different configuration (includes but not limited to different number of layers, hidden dimension, GRU/LSTM/RNN) and EXPLAIN how does this affects your performance / speed of convergence / ... Some possible BONUS tricks that you can try: multi-tasking, few-shot learning, zero-shot learning, CRF, CNN-BiLSTM. This question will be grade by the completeness of your experiments and your findings.

1. **Intent Classification**

   Performance on validation set: 90.885
   My Configurations:

   - RNN: GRU
     I've tried vanilla RNN (validation performance: 88.184), and LSTM (validation performance: 89.551) as well, and it turns out that GRU outperforms on this task.
   - learning rate: 1e-3, with StepLR
     Initially, the learning rate was set to 1e-4, and it learns quite slowly. Afterwards, it was raised to 1e-3. It converges even more quickly. It can reach around 87% accuracy just in around 10 epochs. I also tried to raise the learning rate to 2e-3 but it turned out that it does not attain better performance. With an idea that the learning rate should decline over time, I found *pytorch* has porvided *StepLR* function. The learning rate will be multiplied by 0.8 every 20 epochs under my settings.
   - hidden_size: 512
     I had tried hidden_size to be 256 and it does not perform as well as 512 can be.
   - num_layers: 2
     I've also tried different number of layers (3 layers: 89.128, 1 layer: 88.639). It seems that 2 layers would be a better option.
   - dropout: 0.2
     The default setting is 0.5; however, it does not attain the performance when it was set to 0.2 in my experience.
   - bidirectional: True The task should take the up coming words into consideration. So it is intuitively set to be true. I've tried a non-bidirectional version (validation performance=88.8281). As expected, it does not reach the performance of bidirectional.

2. **Slot Tagging (Old implementation without CNN)**

   Performance on validation set: 79.6
   My Configurations:

   - RNN: GRU
     From Intent Classification, GRU outperforms other RNN models, so I chose GRU to be the model for this task.
   - learning rate: 1e-3, with StepLR
     The configuration is the same as that in Intent Classification. It also converges in reasonable time, so I don't modify the learning rate settings.
   - hidden_size: 384
     I tried hidden_size to be 256 and 512 and it turns out that 384 can reach a better performance.
   - num_layers: 3
     Initially, the number of layers was set to 2; however, I discovered that 3 layers would get a slightly better result. Therefore, I chose 3 layers as my final configuration.
   - dropout: 0.3
     The dropout rate is higher compared to Intent Classification. I've tried 0.2, 0.3, 0.4, 0.5 and found that 0.3 has slightly better performance.
   - bidirectional: True The task should take the up coming words into consideration. So it is intuitively set to be true. I've tried a non-bidirectional version (validation performance=70.3). As expected, it does not reach the performance of bidirectional.

3. **Slot Tagging with CNN-BiLSTM**

Performance on validation set: 82.5
My Configurations:

- CNN: Conv1D
  I only apply 1 layer of CNN because 2 layers of CNN do not outperform 1 layer in my experience. The kernel_size is 11 with a padding of 5 to make the length the same after going through CNN.
- RNN: GRU
  From previous experience, GRU outperforms other RNN models, so I chose GRU to be the model for this task.
- learning rate: 1e-3, with StepLR
  From previous experience this configuration is fine, so I don't modify the learning rate settings.
- hidden_size: 384
  I tried hidden_size to be 256 and 512 and it turns out that 384 can reach a better performance.
- num_layers: 2
  After adding a CNN layer, it turns out that 2 layers achieve better performance than 3 layers, so num_layers is set to be 2.
- dropout: 0.7
  The dropout rate is even higher compared to the previous implementation. I've tried 0.3, 0.4, 0.5, 0.6, 0.7, 0.75, 0.8 and found that 0.7 suits the implementation the best.
- bidirectional: True The task should take the up coming words into consideration. So it is intuitively set to be true. I've tried a non-bidirectional version (validation performance=75.3). As expected, it does not reach the performance of bidirectional.