Q1: Data processing

1. **Describe how do you use the data:**

   a. **How do you tokenize the data?**

      i. preprocess:

      The function *build_vocab()* in preprocess_intent.py will help us tokenize our data with the support from pre-trained embedding, glove. First, it will split the "text" data into serveral single words when doing intent classifier, while "tokens" data in slot tagging is already in the form of single words.

      Then, it will start to count the occurrence of the different words in these data, and map them into different embeddings depending on the count of their occurrence. After this, it will use *torch.save()* to store the result.

      ii. collate_fn:

      My 2 collate_fn have similar effect. First, they will aggregate "text"/"tokens" of all data and make it as a list of list of string. During this, it will split each "text" and make it a list of string, while "tokens" is already a list. Then, I use *encode_batch()* to transform all string the final list of "text"/"tokens" to numbers and make it a flow.

      As for "intent"/"tag", I apply *label_mapping()* to them, and similarly I will aggregate all data together to make a list. Additionally, I have to trace the max number of "tags" in each data, and use *pad_to_len()* at last to make sure that all data has the same length, which can be transform to a tensor. Moreover, since the length of each data has changed, I have to remember the original length for fear that print too much redundant word when predicting. Hence, I append a new column named "length" in slot tagging.

      For "id", I just simply aggregate all of them and make it a list. Finally, I will have all lists mentioned above to become a dictionary containing all data.

   b. **The pre-trained embedding you used:**

      I use glove.840B.300d.txt within the sample code.

Q2: Describe your intent classification model

1. Model:

   a. init:
   GRU(
   embedding size
   hidden_layer size: the number of features in the hidden state i.e. h
   num_layers: the number of recurrent layers
   dropout: I set it to 0.001, which is the rate to remove a unit from neural net.
   bidirectional: It denote whether the GRU is bidirectional or not.
   batch_first: I set it to true, which lets the output tensor start with batch.
)
   classifier = torch.nn.sequential(
   Dropout(dropout): apply dropout layer
   Linear(encoder_output_size, num_class): apply linear layer
   Softmax(): map all value within the output tensor to [0, 1] and make sure their sum equals to 1
)

   b. forward:
   $h_i = GRU(w_i, h_{i-1})$, where w_i is word embeding
   $h_i^{'} = classifier(h_i)$ It will make h_i a probability distribution of each type of intent. Then, we can simply return a dictionary whose keys are each "id" with responding probability distribution within h_i as values.

2. Performance: 0.92311

3. Loss function: BCE_loss

   $loss = BCE\_loss(y2, y) + BCE\_loss(y2 \cdot y, y) \cdot num\_class$, where y2 is the prediction of probability distribution, while y is the one_hot of correct answer.

4. Relative Parameter

   a. optimization algorithm: Adam

   b. learning rate: 0.001
   applying ***torch.optim.lr_scheduler.StepL*** with step_size=50, gamma=0.8

   c. batch size: 128

Q3: Describe your slot tagging model

1. Model:

   a. init:
   GRU(
   embedding size
   hidden_layer size: the number of features in the hidden state i.e. h
   num_layers: the number of recurrent layers
   dropout: I set it to 0.001, which is the rate to remove a unit from neural net.
   bidirectional: It denote whether the GRU is bidirectional or not.
   batch_first: I set it to true, which lets the output tensor start with batch.
)
   I extraly append 4 CNN layers.
   CNN(
   sequential(
   ReLU()
   Dropout(dropout): apply dropout layer
   Conv1d(in, out, kernel_size = 5):
   in/out: the number of channels in the input/output layer
   kernel_size: size of the convolving kernel
)
)
   classifier = sequential(
   Dropout(dropout): apply dropout layer
   Linear(encoder_output_size, num_class): apply linear layer
   Sigmoid(): map all value within the output tensor to [0, 1]
)

   b. forward:
   $h_i^1 = GRU(w_i, h_{i-1}^1),\ where\ w_i\ is\ word\ embedding$
   $h_i^{j+1} = CNN(h_i^{j'}),\ where 1 \leq j \leq 4\ and\ h_i^{j'}$ is hidden data with size of **kernel_size** collected from h_i^j multiplying with a matrix from model we trained.
   Since I apply 4 CNN layers before, I can use it for 4 times.
   $h_i^{'} = classifier(h_i^5)$ It will make h_i a probability distribution of each type of tags.
   Then, we can simply return a dictionary whose keys are each "id" with responding probability distribution within h_i as values.

2. Performance: 0.77265

   (I lost my best model, so I use the second highest as my submission.)

3. Loss function:

   $loss = BCE\_loss(y2, y) + BCE\_loss(y2 \cdot y, y) \cdot num\_class$, where y2 is the prediction of probability distribution, while y is the one_hot of correct answer.

4. Relative Parameter

   a. optimization algorithm: Adam

   b. learning rate: 0.001
      applying ***torch.optim.lr_scheduler.StepL*** with step_size=25, gamma=0.8

   c. batch size: 128

Q4: Sequence Tagging Evaluation

$$precision = \frac{true\ positive}{true\ positive + false\ positive}$$

It denotes the ability not to label a negative sample as positive.

$$recall = \frac{true\ positive}{true\ positive + false\ negative}$$

It denotes the ability to find the positive samples.

The best result of joint accuracy I got is 0.78227. It's almost smaller than all the value with in the figure, for seqeval evaluation doesn't only consider the whole sentence but a single tags. Thus, if a tag is matched to one of types listed in the figure it will be justified as correct, which is similar to tokens accuracy, while joint accuracy will justified a prediction as correct only if all tags within the sentence are matched. Speaking of weighted avg of precision in the figure, it is similar to token accuracy. However, it doesn't seem to be much more than joint accuracy I got. I think it's because it ignores the tag "O", which accounts for the majority of the data. Therefore, the accuracy will much less than token accuracy.

	precision	recall	f1-score	support
date	0.75	0.76	0.76	206
first_name	0.94	0.87	0.90	102
last_name	0.76	0.79	0.77	78
people	0.78	0.76	0.77	238
time	0.84	0.83	0.83	218
micro avg	0.80	0.79	0.80	842
macro avg	0.81	0.80	0.81	842
weighted avg	0.81	0.79	0.80	842

Figure 1: seqeval result

Q5: Compare with different configurations

a. Intent Classification:

Initially, I use LSTM as my model, but it didn't perform well. First, the correct rate was much less than my current model. Therefore, I tried to increase it value of hidden_size, num_layers and batch_size. As a result, the correct rate indeed increased, but the execution time was also added, and over-fitting occurs, that is the correct rate would drop after about 70 epochs and the value of loss function is almost 0, which means it couldn't make any improvement at all. I attempted to cope with it by increasing the value of dropout. However, although it can somehow solve such problems, it still couldn't improve the resulting correct rate.

Additionally, its speed of convergence was much slower than my current model, too. The latter requires at least 50 60 epochs to reach the max correct rate, while the former only costs about 10 20 epochs. Last but not least, LSTM has longer execution time than GRU with same parameter. Hence, I switched to use GRU instead of LSTM, and adjustment of parameter mentioned above also didn't make improvement and had the same impact on my GRU, so I don't change their default value eventually.

b. Slot Tagging:

Slot tagging has similar result mentioned above. Differently, the correct rate was just slightly higher than baseline. Therefore, I add several CNN layers after the GRU layer and try to improve it. I had tried 1, 2, 3, 4, 5, and 6 CNN layers and set its kernel_size to 3 or 5. From 1 4 layers of CNN, either kernel_size is 3 or 5, the correct rate had obvious increase. However, for 5 or 6 layers of CNN, the correct rate no longer increased. I think it is because the length of each sentence isn't so long, 5 or 6 layers of CNN would exceed its length, which was in vain. Hence, depending on my experiment, I choose 4 layers of CNN with kernel_size set to 5.

Moreover, I use ***scheduler(num_epoch, ratio)*** to adjust my learning rate periodically, where ***num_epoch*** functions as a timer to adjust my learning rate, and ***ratio*** denotes the ratio how we reduce it. Since I find out that changing default value of learning rate hardly has impact on my result, I think perhaps scheduler could influence more. Therefor, I tried lots of set of scheduler and find out (50, 0.8) could generate the best correct rate.