Q1 : Data processing

I use the tokenization and pre-trained embedding from sample code(preprocess.sh)

a. The tokenized data was saved in the vocab.pkl by the function build_vocab in preprocessing.py. The token data uses Vocab that is in utils.py, which the words that are not common will observe as Unknown.
   When I need to use the data to feed into my model, I will call a function in utils.py that is called encode_batch to encode my data then feed it into my model.
b. I used the embeddings from sample code(embeddings.pt), which is from `http://nlp.stanford.edu/data/glove.840B.300d.zip`, the embeddings is made in preprocess.py - build_vocab, output saved as embeddings.pt.
   This embedding is used in model forward, embedded where the data come out from encode_batch.

Q2 : Describe your Intent Classification model.

a. __init__() part :
   Define a multi-layer **long short-term memory (LSTM)** RNN to an input sequence.
   embeddings : from sample code (embeddings.pt)
   hidden_size=512,
   num_layers=2, means stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results.
   bidirectional=True, so becomes a bidirectional LSTM.
   dropout=0.2
   batch_first=True, so the input and output tensors are provided as (batch, seq, feature) instead of (seq, batch, feature).
   Two **Linear Layer** and an activation function of **ReLU()**

   Forward() part :
   In my code :
   embedded = self.embed(batch)
   lstm_out, _ = self.lstm(embedded)
   #output, (h_n, c_n) = LSTM(input, (h_0, c_0))
   LSTM input :
   input (batch size, sequence length, input_size) - tensors of shape for unbatched input, containing the features of the input sequence.
   h_0 - tensors of shape for unbatched input, containing the initial hidden state for each element in the input sequence.
   c_0 - tensors of shape for unbatched input, containing the initial cell state for each element in the input sequence.
   LSTM output :
   output - tensors of shape for unbatched input, containing the output features (h_t) from the last layer of the LSTM, for each t.
   h_n - tensors of shape for unbatched input, containing the final hidden state for each element in the sequence. In our code bidirectional=True, so h_n will contain a concatenation of the final forward and reverse hidden states, respectively.
   c_n - tensors of shape for unbatched input, containing the final cell state for each element in the sequence. In our code bidirectional=True, so

c_n will contain a concatenation of the final forward and reverse cell states, respectively.

In my code :

hidden_state = lstm_out[:,-1,:] - intend to predict the label associated with the input text only using the last hidden state of my LSTM network.

In my code :

output = self.classifier_1(hidden_state)
output = self.classifier_2(output)

Two Linear layers that transfer the dimension of hidden_size to num_classes, using two linear layers that the layers of neurons in neural networks are *not* affine transformations. All commonly used neurons have some kind of non-linearity. The simplest of these is the Rectified Linear Unit (**ReLU**), which is the activation function that I used.

b. Intent Classification : 0.89066

c. CrossEntropyLoss() :
This criterion computes the cross entropy loss between input and target. It is useful when training a classification problem with C classes, also useful when having an unbalanced training set.
   i. **Input** - Shape (num_classes), (batch_size) or (batch_size,num_classed,d_1,d_2,...,d_k) with k>=1 in the case of K-dimensional loss.
   ii. **Target** - If containing class indices, shape (), (batch_size) or (batch_size,d_1,d_2,...,d_k) with k>=1 in the case of K-dimensional loss where each value should be between [0,num_classes).
   iii. **Output** - same shape as the target.

d. torch.optim.Adamax(), learining rate : 1e-3, batch_size : 128

Q3 : Describe your Slot Tagging model.

a. __init__() part :
Extension of Intent Classification model (SeqClassification).
Forward() part :
Extension of Intent Classification model (SeqClassification).
Difference :
   - Only one Linear layer to improve accuracy.
   - CrossEntropyLoss expects the shape of its input to be (N,C,...), so the second dimensions is always the number of classes, in my code it will be (batch_size, num_classes, max_len), so when the LSTM use for slot tagging, one input sentence needs to predict lots of labels for each tokens,hence the tensors of dimensions needs to be permute so that it can feed into CrossEntropyLoss().

b. Slot Tagging : 0.78766

c. CrossEntropyLoss()

d. torch.optim.Adamax(), learining rate : 1e-3, batch_size : 128

Q4 :

	precision	recall	f1-score	support
date	0.67	0.71	0.69	14
first_name	1.00	1.00	1.00	11
last_name	1.00	0.80	0.89	10
people	0.68	0.74	0.71	23
time	0.81	0.76	0.79	29
micro avg	0.79	0.78	0.79	87
macro avg	0.83	0.80	0.81	87
weighted avg	0.80	0.78	0.79	87

Metrics :

Precision : Compute the precision, The precision is the ratio of true positive / (true positive + false positive). The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The best value is 1 and the worst value is 0.

Recall : Compute the recall, The recall is the ratio of true positive / (true positive + false negative). The recall is intuitively the ability of the classifier to find all the positive samples. The best value is 1 and the worst value is 0.

F1-score : Compute the F1 score, F1_score = 2*(precision * recall) / (precision + recall). The F1 score can be interpreted as a harmonic mean of precision and recall, where an F1 score reaches its best value at 1 and worst score at 0.

Token accuracy : The denominator is the number of tokens that all sentences have, calculated correctly when one token is predicted correctly.

Joint accuracy : The denominator is the number of sentences, and only correct when all tokens predict correctly in multi classification tasks, more useful in some tasks just like slot tagging.

Q5 :

I have tried using torch.nn.functional softmax to improve accuracy when model forward output, but when training it seems not better than before.

Also I tried to do layer normalization when passing the output but it seems nothing improved either.