Q1: Data processing

a. How to tokenize the data

I used the sample code (preprocess) for data processing.

In intent_cls.sh, first, I separated intents and texts from the dataset. Then, I splitted every text into serveral words (tokens) and saved intents with a set and the splitted tokens with a counter. With the counter data type, we would have words without duplicating. Then, load the pre-trained embedding in order to map tokens to a vector (which is a list of float). If a token was not in the pre-trained embedding, then we used `random( ) * 2 - 1` for each value in the vector as the embedding.

In slot_tag.sh, it was quite similar to that in intent_cls.sh. The only different was that the input of slot tagging task was already a list of tokens so that we didn't need to split the text on our own. Then, we also checked whether the token was in the pre-trained embedding. If not, we had to create it.

b. The pre-trained embedding

In the sample code, it used Glove as the pre-trained embedding.

Q2: Describe your intent classification model.

a. Description of model

$$
\begin{aligned}
w_t &= embedding(x_t) \text{ ,where } x_t \text{ is the token of t-th word of the sentence} \\
\overrightarrow{h_t}, \overrightarrow{c_t} &= GRU(w_{t-1}, \overrightarrow{h}_{t-1}, \overrightarrow{c}_{t-1}) \text{ , where } w_t \text{ is the word embedding of the t-th token} \\
\overleftarrow{h_t}, \overleftarrow{c_t} &= GRU(w_{t-1}, \overleftarrow{h}_{t+1}, \overleftarrow{c}_{t-1}) \\
h' &= flatten(h) \text{ , flatten the outputs to 1-dim where } h = U[\overrightarrow{h}; \overleftarrow{h}] \\
output &= Linear(h')
\end{aligned}
$$

Settings:

- GRU:

    - Input_size: dimension of the embeddings
    - hidden_size: 512
    - num_layers: 2
    - bidirectional: True
    - dropout: 0.1
    - batch_first: True
- Linear:
    - One-layer of fully-connected layer
    - Input_feature: max_len * num_direction * hidden_size
    - Output_feature: num_class

b. performance of the model

public score on Kaggle: 0.90044

private score on Kaggle: 0.91244

c. Loss function

I used **CrossEntropyLoss** as my loss function.

d. Optimization Algorithm, learning rate and batch size

- optimization algorithm: Adam
- learning rate: 10^{-3}
- batch size: 512

Q3. Describe your slot tagging model.

a. Description of model

$$
\begin{aligned}
w_t &= embedding(x_t) \text{ ,where } x_t \text{ is the token of t-th word of the sentence} \\
\overrightarrow{h_t}, \overrightarrow{c_t} &= GRU(w_{t-1}, \overrightarrow{h}_{t-1}, \overrightarrow{c}_{t-1}) \text{ , where } w_t \text{ is the word embedding of the t-th token} \\
\overleftarrow{h_t}, \overleftarrow{c_t} &= GRU(w_{t-1}, \overleftarrow{h}_{t+1}, \overleftarrow{c}_{t-1}) \\
output &= Linear(h) \text{ , where } h = U[\overrightarrow{h}; \overleftarrow{h}]
\end{aligned}
$$

Settings:

- GRU:
  - Input_size: dimension of the embeddings
  - hidden_size: 1024
  - num_layers: 2
  - bidirectionalal: True
  - dropout: 0.25
  - batch_first: True
- Linear:
  - 2-layers of fully connected layer
  - Input_feature: max_len * num_direction * hidden_size
  - Output_feature: num_class (9)
  - Model:
    - Linear(input_dim, 2048)
    - BatchNorm1d(2048)
    - Dropout(0.5)
    - ReLU( )
    - Linear(2048, num_class)

b. performance of the model

public score on Kaggle: 0.75924

private score on Kaggle: 0.76420

c. Loss function

I used **CrossEntropyLoss** as my my loss function

d. Optimization Algorithm, learning rate and batch size

- optimization algorithm: Adam
- learning rate: 7×10^{-5}
- batch size: 128

Q4. Sequence Tagging Evaluation

	precision	recall	f1-score	support
date	0.77	0.74	0.75	206
first_name	0.95	0.86	0.90	102
last_name	0.74	0.72	0.73	78
people	0.78	0.73	0.75	238
time	0.85	0.86	0.86	218
micro avg	0.81	0.78	0.80	842
macro avg	0.82	0.78	0.80	842
weighted avg	0.81	0.78	0.80	842

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

The result of true or false is based on a chunk with IOB2 format (B-tag is used in the beginning of every chunk)

Here are the definitions of the three evaluation.

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

The main differences between these methods are that

- Joint Accuracy: the **strictest** method among them, all tags in a **sentence** have to be correct and it'll be counted as correct
- Evaluation method in seqeval: the most **balanced** method, all tags in a **chunk** have to be correct and it'll be counted as correct and a sentence contains one or more than

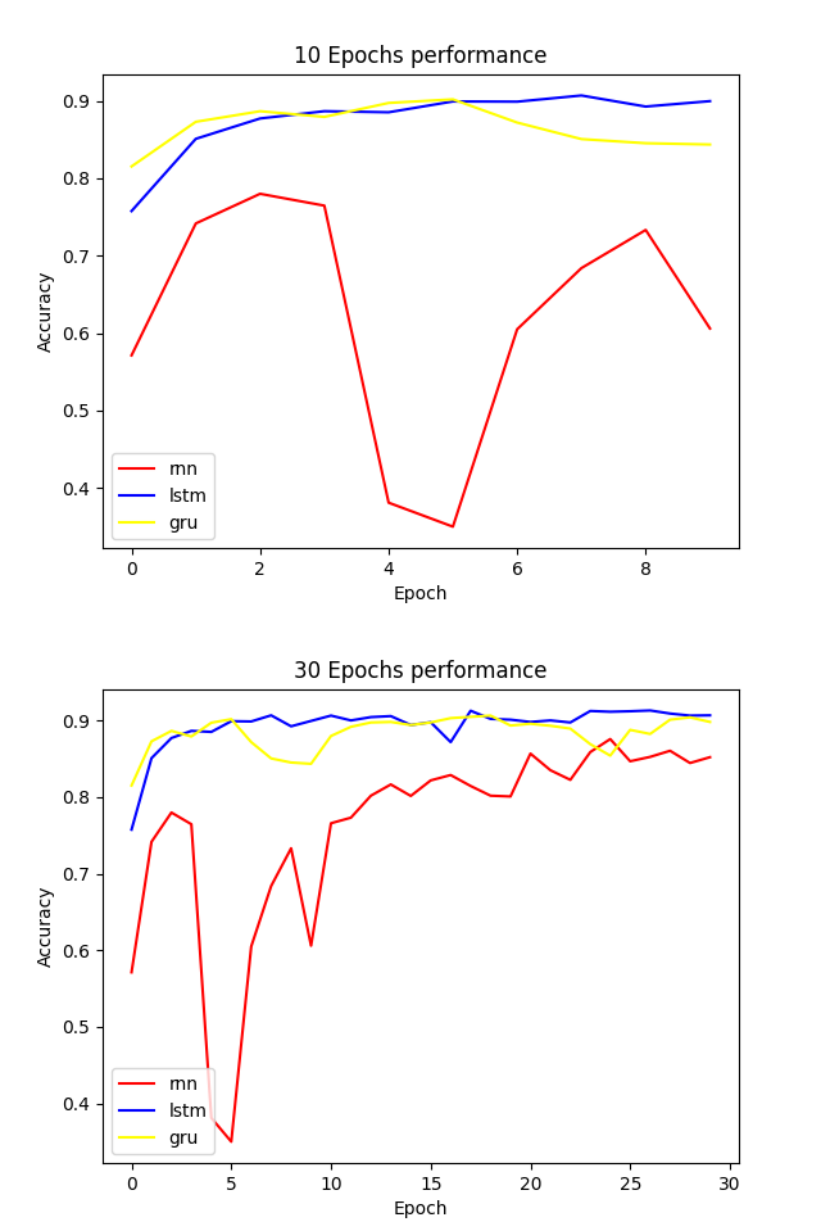
one chunks

- Tag Accuracy: the **loosest** method, when one tag is correct and it'll be counted as correct

Q5. Compare with different configurations

The accuracy below is **validation accuracy**.

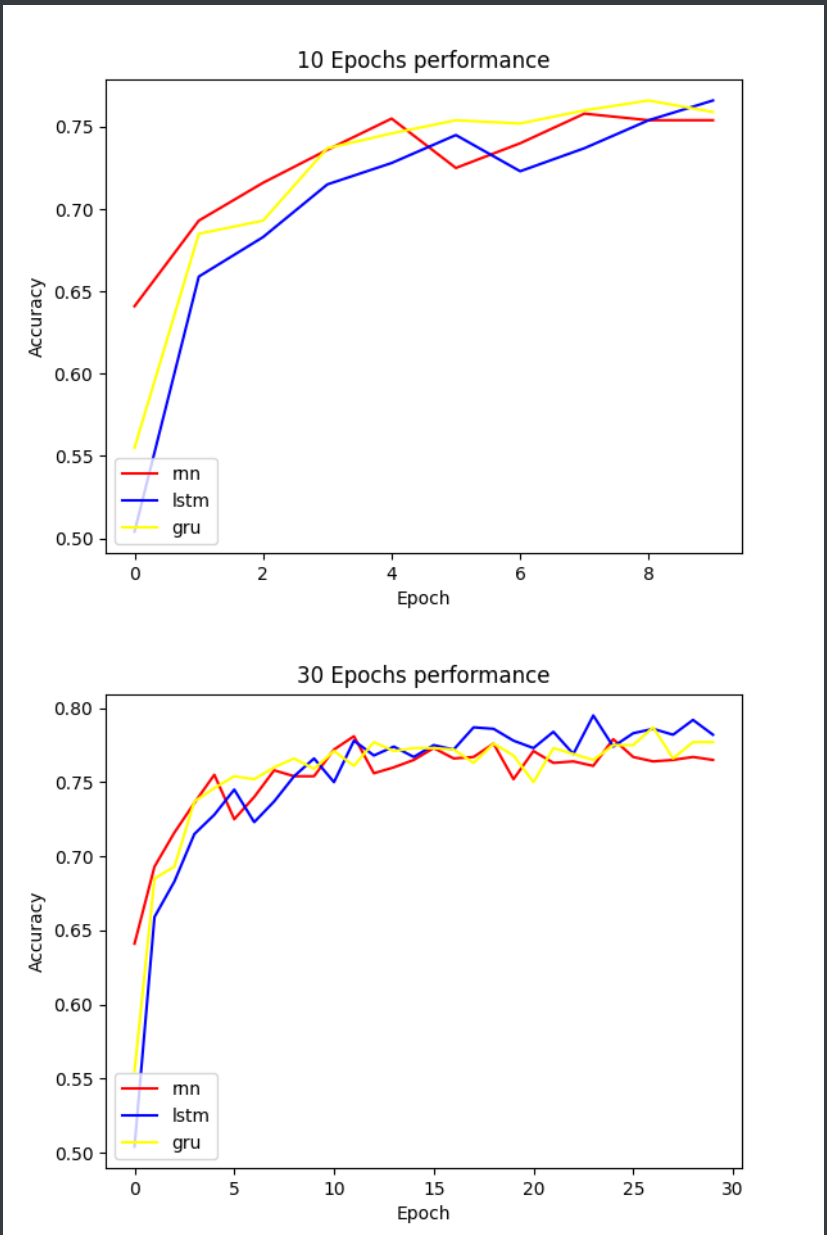
Intent Prediciton



Settings:

- hidden_size: 512
- num_layer: 2
- dropout: 0.1
- learning_rate: 0.001
- batch_size: 128

Slot Tagging



Settings:

- hidden_size: 1024

- num_layer: 2
- dropout: 0.25
- learning_rate: 0.00007
- batch_size: 128

Since the RNN model has the problem of vanishing gradient problem which has been taught in class, it's difficult for RNN to learn to preserve information over many timesteps. Therefore, GRU and LSTM contains the concept of **gates** to prevent it. Hence, we can find out that in both tasks, the performance of RNN is the worst of all of three models. However, it's worth mentioning that the performance of RNN is not that bad for the task of slot tagging. I think it's because the task of slot tagging does not depend on the whole sentence (long term memory). As a result, the problem of vanishing gradient won't hurt too much while the task of intent prediction relies on the semantic of the whole text.