ADL HW1

b08902045 資工三 袁紹奇

Q1: Data processing (2%)

Describe how do you use the data for intent_cls.sh, slot_tag.sh:

1. How do you tokenize the data.
   The input texts of intent classification is split by spaces, and then the program would convert each token to a given word index.
2. The pre-trained embedding you used.
   The word indices would be converted to the corresponding embeddings based on GloVe. The conversion code is given by the TAs. Note that if some word don't exist in the conversion dictionary, the index would be given as `UNK`, and then it would be assigned to a random vector.

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

a. your model

The model I use for this task is a two-layer bidirectional GRU with given input size, hidden size, dropout.

$$h_t, c_t = \mathrm{GRU}(w_t, h_{t-1}, c_{t-1})$$

where the w_t is the t-th token of the input sequence, and h_t, c_t are the hidden state and the cell output at the t-th time stamp, respectively.
The hyper-paramenters are given as follows.

```
hidden_size = 512
num_layers = 2
dropout = 0.2
bidirectional = True
```

After the inputs being encoded by GRU, the result would be fed into a decoder to generate the final prediction. The model structure is as follows.

```
nn.Sequential(
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(in_features=self.encoder_output_size,
out_features=hidden_size//2),
    nn.BatchNorm1d(hidden_size//2),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(in_features=hidden_size//2, out_features=num_class),
)
```

Note that the parameter `dropout` is the same as GRU.

b. performance of your model. (public score on kaggle)

0.93111

c. the loss function you used.

I used cross entropy loss as the loss function.

d. The optimization algorithm (e.g. Adam), learning rate and batch size.

Adam is used as the optimizer. The hyper-parameters are given as follows.

```
lr = 2e-4
weight_decay = 1e-5
batch_size = 1024
```

Note that I implemented clipping in my code. Whenever the L2 Norm of the gradient is larger than 5, it would be set to 5.

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

a. your model

The model I use for this task is a two-layer bidirectional GRU with given input size, hidden size, dropout.

$$h_t, c_t = \mathrm{GRU}(w_t, h_{t-1}, c_{t-1})$$

where the w_t is the t-th token of the input sequence, and h_t, c_t are the hidden state and the cell output at the t-th time stamp, respectively.
The hyper-paramenters are given as follows.

```
hidden_size = 512
num_layers = 2
dropout = 0.2
bidirectional = True
```

After the inputs are encoded by GRU, the result would be feed into a decoder to generate the final prediction. The model structure is as follows.

```
nn.Sequential(
    nn.Dropout(dropout),
    nn.ReLU(),
    nn.LayerNorm(self.encoder_output_size),
    nn.Linear(in_features=self.encoder_output_size, out_features=num_class),
)
```

Note that the parameter dropout is the same as GRU.

b. performance of your model. (public score on kaggle)

0.79624

c. the loss function you used.

I used cross entropy loss as the loss function.

d. The optimization algorithm (e.g. Adam), learning rate and batch size.

Adam is used as the optimizer. The hyper-parameters are given as follows.

```
lr = 5e-4
weight_decay = 1e-5
batch_size = 512
```

Note that I implemented clipping in my code. Whenever the L2 Norm of the gradient is larger than 0.1, it would be set to 0.1.

Q4: Sequence Tagging Evaluation (2%)

```
Joint Accuracy: 0.815 (815/1000)
Token Accuracy: 0.968 (7637/7891)

seqeval Classification Report
              precision    recall  f1-score   support

        date       0.76      0.78      0.77       206
  first_name       0.97      0.87      0.92       102
   last_name       0.76      0.91      0.83        78
      people       0.77      0.72      0.75       238
        time       0.82      0.91      0.86       218

   micro avg       0.80      0.82      0.81       842
   macro avg       0.82      0.84      0.83       842
weighted avg       0.80      0.82      0.81       842
```

The joint evaluation method counts a correct sentence only if all the tokens in the sentence are predicted correctly.

$$\text{joint acc} = \frac{\text{correct predicted sentence}}{\#\text{ of sentence}}$$

The tokens evaluation method counts correct tokens as the unit.

$$\text{token acc} = \frac{\text{correct predicted tokens}}{\#\text{ of tokens}}$$

The seqeval evaluation method first separates each sample to several *chunks* based on the labeling schemes. Here we use IOB2 as the labeling schemes. Each chunk contains the same tag, and the chunk can be represented to a *tuple* of `(tag, begin, end)`. Based on the information, the method calculates the true positive (TP), false positive (FP) and false negative (FN) for the validation dataset. It then calculates the precision, recall, F1 score, support for each tags. The equations are given as follows.

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

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

$$F1\ score = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

$$Support = \#\ of\ tuples$$

The micro-average counts the average of all *tokens*. The macro-average counts the average of all *types of tags*. The weighted average, however, counts the *weight average* of all types of tags.

For example:

```
y_true = ['O', 'O', 'O', 'B-MISC', 'I-MISC', 'I-MISC', 'O', 'B-PER', 'I-PER']
y_pred = ['O', 'O', 'B-MISC', 'I-MISC', 'B-MISC', 'I-MISC', 'O', 'B-PER', 'I-PER']

print("accuary: ", accuracy_score(y_true, y_pred))
print("p: ", precision_score(y_true, y_pred))
print("r: ", recall_score(y_true, y_pred))
print("f1: ", f1_score(y_true, y_pred))
print("classification report: ")
print(classification_report(y_true, y_pred))
```

The output would be

```
accuary:  0.6666666666666666
p:  0.3333333333333333
r:  0.5
f1:  0.4
classification report:
              precision    recall  f1-score   support

        MISC       0.00      0.00      0.00         1
         PER       1.00      1.00      1.00         1

micro avg       0.33      0.50      0.40         2
macro avg       0.50      0.50      0.50         2
```

reference: https://zhuanlan.zhihu.com/p/30953081

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

For the following experiments, if not specified, default hyper-parameters are set for both tasks. The default hyper-parameters are as follows.

Optimizer	weight decay	learning rate	batch size	hidden size	# of epoch	# of layers of RNN	dropout
Adam	1e-5	2e-4	256	512	100	2	0.2

Also, clipping is implemented in my code, the value is 10 and 0.1 for intent classification and slot tagging, respectively.

Intent classification

For the decoder, I use the following model structure:

```
nn.Sequential(
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(in_features=self.encoder_output_size,
out_features=hidden_size//2),
    nn.BatchNorm1d(hidden_size//2),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(in_features=hidden_size//2, out_features=num_class),
)
```

I have tested the different models for the encoder including vanilla RNN, GRU and LSTM. I also tested the difference of the usage of bidirectional RNN, and the method to extract the output of the encoder. For the experiments of bidirectional RNN, one of the method is to feed the decoder with the mean of all encodes. The other is to use the concatenation of the final outputs of both direction.

hidden size = 512

Method	Validation Acc.	Public Test	Private Test
RNN	0.920	0.91022	0.90266
biRNN (mean)	0.921	0.91600	0.91600
biRNN (concat.)	0.921	0.91644	0.91244
GRU	0.938	0.93111	0.92044
biGRU (mean)	0.936	0.92755	0.92222
biGRU (concat.)	0.942	0.93066	0.93066
LSTM	0.934	0.92444	0.91600
biLSTM (mean)	0.935	0.93288	0.92488
biLSTM (concat.)	0.935	0.93244	0.92533

hidden size = 256

Method	Validation Acc.	Public Test	Private Test
RNN	0.912	0.89511	0.88622
biRNN (mean)	0.923	0.91955	0.91377
biRNN (concat.)	0.908	0.88711	0.88488
GRU	0.934	0.91866	0.92222
biGRU (mean)	0.934	0.92666	0.92666
biGRU (concat.)	0.932	0.92577	0.92088
LSTM	0.929	0.92977	0.92133
biLSTM (mean)	0.927	0.91555	0.91066
biLSTM (concat.)	0.931	0.91777	0.92311

The overall performance of `hidden_size=512` is better than `hidden_size=256`. I think that the task might still needs high hidden_size so that the encoders could record the information of the whole sequence. The difference of concatenation and mean is not clear. It seems that both methods can sufficiently represent the whole sequence. Furthermore, generally GRU and LSTM performs better than vanilla RNN. It is intuitive because the encoder can records the more previous information.

Slot tagging

For the decoder, I use the following model structure:

```
nn.Sequential(
    nn.Dropout(dropout),
    nn.ReLU(),
    nn.LayerNorm(self.encoder_output_size),
    nn.Linear(in_features=self.encoder_output_size, out_features=num_class),
)
```

Method	Validation Acc.	Public Test	Private Test
RNN	0.810	0.79195	0.80171
biRNN	0.809	0.78230	0.78510
GRU	0.817	0.80536	0.81672
biGRU	0.810	0.77426	0.80225
LSTM	0.801	0.79088	0.81243
biLSTM	0.808	0.79463	0.81457

hidden size = 256

Method	Validation Acc.	Public Test	Private Test
RNN	0.799	0.76461	0.77974
biRNN	0.799	0.79302	0.78992
GRU	0.805	0.79249	0.81082
biGRU	0.811	0.79946	0.80921
LSTM	0.803	0.79624	0.81028
biLSTM	0.811	0.79731	0.80439

For the slot tagging task, the smaller hidden size is generally better than the larger hidden size, which is different from intent classification. I think that slot tagging needs more nearby information instead of the whole sequence, and the portion of sequence requires less information than the whole sequence.

Bonus

For the bonus part, I have implemented the CNN layer before RNN encoder. The sequence of tokens would first be converted to the word embeddings. Then the sequence of word embeddings would be feed to the CNN layers, such that the output sequence of CNN would have seen the nearby tokens before being fed to the RNN layers. The hpyer-parameters of the CNN layers are as follows.

input size	output size	kernel size	stride	padding	padding mode
300	300	5	1	2	zeros

I have experimented the CNN-RNN model for the following settings. The results are as follows.

Method	Validation Acc.	Public Test	Private Test
RNN	0.803	0.76246	0.78456
biRNN	0.808	0.78981	0.80385
GRU	0.811	0.80332	0.78230
biGRU	0.811	0.80921	0.78445
LSTM	0.816	0.80439	0.78820
biLSTM	0.813	0.78552	0.81243

As the experiments result shows, the difference of using CNN layer is small. I think that it is because the RNN has enough information for slot tagging.