HW1

r11944008 許智凱

Q1: Data processing (2%)

a.How do you tokenize the data

The input texts are spilt by whitespace, and then the preprocess program in sample code would convert each token to a given word index

b.The pre-trained embedding you used

GloVe.
In sample code, the word index would be converted to the corresponding embedding based on GloVe.
Note that if a word doesn't exist in the conversion dictionary, its index would be given as `UNK`, and its embedding would be assigned to a random vector.

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

a. your model

I use a two-layers bidirectional GRU with given input size, hidden size and dropout as my model.

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

where w_t is the word embedding of the t-th token and $h_t,\ c_t$ are the hidden state, the cell output at the t-th timestamp, respectively.

The hyper parameters of BiGRU

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

After the inputs are encoded by BiGRU, the outputs would be fed into a classifier to generate the predictions.

Architecture of BiGRU & classifier

```
self.rnn = nn.GRU(
    input_size = self.embed.embedding_dim,
    hidden_size = self.hidden_size,
    num_layers = self.num_layers,
    dropout = self.dropout,
    bidirectional = self.bidirectional,
    batch_first = True, # batch * seq * feature
)

self.classifier = nn.Sequential(
```

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

Note that BiGRU's `dropout` are the same as classifier

Weight initailization pass

```
for name, param in self.rnn.named_parameters():
    if name.startswith('weight'):
        nn.init.orthogonal_( param )
    else:
        nn.init.zeros_( param )
```

Forward pass

```
def forward(self, batch) -> Dict[str, torch.Tensor]:
    x = self.embed( batch )

    # if using CNN
    if self.CNN:
        x = x.permute( 0, 2, 1 )
        for conv in self.cnn:
            x = conv( x )
        x = x.permute( 0, 2, 1 )

    x, hidden = self.rnn( x, None )
    if self.bidirectional:
        out = self.classifier( torch.sum( x, dim = 1 ) )
    else:
        out = self.classifier( x[:, -1, :] )
    return out
```

- If using CNN model, pass the embeded batch to it.
  Note that the embeded batch needs to be permuted, because the embeded batch's dimension is *(batch size, max len, embedding dimension)*, convert it to *(batch size, embedding dimension, max len)*.
  Before passing the outputs of CNN to BiGRU, permute it to original dimension *(batch size, max len, embedding dimension)*.
- If it is a bidirectional GRU, I sum the two outputs of BiGRU; otherwise I just decode the output

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

0.92977

c. the loss function you used.

Cross entropy loss

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

Adam

The hyper parameters

```
lr = 2e-4
weight_decay = 1e-5
batch_size = 128
```

scheduler: optim.lr_scheduler.StepLR

The hyper parameters

```
step_size = 10
gamma = 0.1
```

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

a. your model

I use a two-layers CNN and a two-layers bidirectional GRU with given input size, hidden size and dropout as my model.

$h_t,\ c_t = GRU(w_t,\ h_{t-1},\ c_{t-1})$
where w_t is the word embedding of the t-th token and $h_t,\ c_t$ are the hidden state, the cell output at the t-th timestamp, respectively.

The hyper parameters of CNN

```
in_channels = self.embed.embedding_dim # 300
out_channels = self.embed.embedding_dim # 300
kernel_size = 5
stride = 1
padding = 2
padding_mode = 'zeros'
dropout = 0.3
```

The hyper parameters of BiGRU

```
hidden_size = 512
num_layers = 2
dropout = 0.3
bidirectional = True
```

Architecture of CNN & BiGRU & classifier

```
self.num_cnn = 2
self.cnn = []
for i in range( self.num_cnn ):
    conv = nn.Sequential(
                nn.Conv1d(
                    in_channels = self.embed.embedding_dim, # 300
                    out_channels = self.embed.embedding_dim, # 300
                    kernel_size = 5,
                    stride = 1,
                    padding = 2,
                    padding_mode = 'zeros',
            ),
            nn.ReLU(),
            nn.Dropout( self.dropout ),
            nn.BatchNorm1d( self.embed.embedding_dim )
    )
    self.cnn.append( conv )
self.cnn = nn.ModuleList( self.cnn )

self.rnn = nn.GRU(
    input_size = self.embed.embedding_dim, # 300
    hidden_size = self.hidden_size,
    num_layers = self.num_layers,
    dropout = self.dropout,
```

```
    bidirectional = self.bidirectional,
    batch_first = True, # batch * seq * feature
)

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

Note that CNN & BiGRU's `dropout` is the same as classifier

Weight initailization pass

```
for name, param in self.rnn.named_parameters():
    if name.startswith('weight'):
        nn.init.orthogonal_( param )
    else:
        nn.init.zeros_( param )
```

Forward pass

```
def forward(self, batch) -> Dict[str, torch.Tensor]:
    x = self.embed( batch )
    if self.CNN:
        x = x.permute( 0, 2, 1 )
        for conv in self.cnn:
            x = conv( x )
        x = x.permute( 0, 2, 1 )

    x, hidden = self.rnn( x, None )
    out = self.classifier( x )
    return out
```

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

0.81554

c. the loss function you used.

Cross entropy loss

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

Adam

The hyper parameters

```
lr = 5e-4
weight_decay = 1e-5
batch_size = 128
```

scheduler: optim.lr_scheduler.StepLR

The hyper parameters

```
step_size = 30
gamma = 0.1
```

Q4: Sequence Tagging Evaluation (2%)

```
Joint Acc: 0.824000, (824/1000)
Token Acc: 0.969586, (7651/7891)

              precision    recall  f1-score   support

        date       0.81      0.78      0.80       206
  first_name       0.97      0.98      0.98       102
   last_name       0.93      0.88      0.91        78
      people       0.75      0.76      0.75       238
        time       0.82      0.83      0.83       218

   micro avg       0.83      0.82      0.82       842
   macro avg       0.86      0.85      0.85       842
weighted avg       0.83      0.82      0.82       842
```

Note that $support$ refers to the number of actual occurrences of the class in the dataset

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

$$Joint\ Accuracy = \frac{\#\ of\ correct\ predicted\ sentences}{\#\ of\ sentences}$$

The token accuracy counts correct predicted tokens.

$$Token\ Accuracy = \frac{\#\ of\ correct\ predicted\ tokens}{\#\ of\ tokens}$$

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

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

$$F1\ score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

Micro average
Micro averaging computes **global average** Precision、Recall and F1 score by counting the sums of the True Positives (TP), False Negatives (FN), and False Positives (FP).
The formulas are above.

Macro average
First, calculate each class's $Precision$、$Recall\ and\ F1\ score$ and then averages them.

$$Macro\ Precision = \frac{1}{n}\sum_{i=1}^{n} Precision_i,\ where\ i\ means\ class$$

$$Macro\ Recall = \frac{1}{n}\sum_{i=1}^{n} Recall_i,\ where\ i\ means\ class$$

$$Macro\ F1\ score = \frac{1}{n}\sum_{i=1}^{n} F1\ score_i,\ where\ i\ means\ class$$

Weighted average

First, calculate each class's $Precision$ 、 $Recall\ and\ F1\ score$ and then averages them with their proportion in total data.

$$Weighted\ Precision = \sum_{i=1}^{n} \frac{\#\ of\ class_i(support)}{\#\ of\ total\ data} Precision_i,\ where\ i\ means\ class$$

$$Weighted\ Recall = \sum_{i=1}^{n} \frac{\#\ of\ class_i(support)}{\#\ of\ total\ data} Recall_i,\ where\ i\ means\ class$$

$$Weighted\ F1\ score = \sum_{i=1}^{n} \frac{\#\ of\ class_i(support)}{\#\ of\ total\ data} F1\ score_i,\ where\ i\ means\ class$$

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

Intent classification

The default hyper-parameters

optimizer	weight-decay	lr	schaduler	step size	gamma
Adam	1e-5	2e-4	StepLR	10	0.1

batch size	epoch	# of RNN layers	dropout
128	100	2	0.2

CNN

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

`hidden size = 512`

without CNN

method	Val. acc	Public score	Private score
LSTM	0.817333	0.78711	0.77688
BiLSTM	0.936333	0.93244	0.92933
GRU	0.920000	0.91555	0.90844
BiGRU	0.939000	0.92977	0.92933
Average	0.903167	0.89122	0.88600

with one-layer CNN

method	Val. acc	Public score	Private score
CNN-LSTM	0.875000	0.85288	0.84133
CNN-BiLSTM	0.942667	0.93155	0.93200
CNN-GRU	0.910000	0.88977	0.88844
CNN-BiGRU	0.942000	0.92800	0.92622
Average	0.917418	0.90055	0.89700

`hidden size = 256`

without CNN

method	Val. acc	Public score	Private score
LSTM	0.772667	0.75600	0.73822
BiLSTM	0.943667	0.91688	0.91511
GRU	0.902000	0.89688	0.89066
BiGRU	0.929333	0.91288	0.91377
Average	0.886917	0.87066	0.86444

with one-layer CNN

method	Val. acc	Public score	Private score
CNN-LSTM	0.854667	0.82133	0.82444
CNN-BiLSTM	0.939667	0.92844	0.92488
CNN-GRU	0.892667	0.88044	0.87422
CNN-BiGRU	0.933333	0.92133	0.92222
Average	0.905084	0.88788	0.88644

The overall performance of `hidden_size=512` is better than `hidden_size=256`
I think that this task might need larger hidden_size so that the encoder could record the information of the whole sentence.

Slot tagging

The default hyper-parameters

optimizer	weight-decay	lr	schaduler	step size	gamma
Adam	1e-5	5e-4	StepLR	30	0.1

batch size	epoch	# of RNN layers	dropout
128	100	2	0.2

CNN

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

`hidden size = 512`

without CNN

method	Val acc	Public score	Private score
LSTM	0.723000	0.71367	0.71650
BiLSTM	0.803000	0.79356	0.80439
GRU	0.717000	0.71313	0.70739
BiGRU	0.812000	0.78820	0.79635
Average	0.763750	0.75214	0.75616

with two-layers CNN

method	Val. acc	Public score	Private score
CNN-LSTM	0.818000	0.80697	0.79314
CNN-BiLSTM	0.834000	0.81554	0.82529
CNN-GRU	0.811000	0.80857	0.80064
CNN-BiGRU	0.814000	0.81823	0.81457
Average	0.819250	0.81233	0.80841

`hidden size = 256`

without CNN

method	Val acc	Public score	Private score
LSTM	0.718000	0.69812	0.71596
BiLSTM	0.802000	0.78498	0.79581
GRU	0.718000	0.68954	0.70150
BiGRU	0.802000	0.79249	0.80385
Average	0.760000	0.74128	0.75428

with two-layers CNN

method	Val. acc	Public score	Private score
CNN-LSTM	0.817000	0.81286	0.80814
CNN-BiLSTM	0.826000	0.82037	0.81404
CNN-GRU	0.811000	0.80589	0.81028
CNN-BiGRU	0.825000	0.81554	0.80546
Average	0.819750	0.81367	0.80948

For this task, the smaller hidden_size is generally a bit better than the larger one, which is different from the intent classification. I think that this task needs more nearby imformation instead of the whole sentence.

Using CNN + RNN is better than using RNN only. I think that CNN could gather the nearby tokens' imformation before being fed to the RNN, so the performance is better.