CE7455 Assignment 2

Name: PENG HONGYI

Matric No: G2105029E

All code and model checkpoints can be found at PengHongyiNTU/NLP-Assignment-2-Bas (github.com)

Question one (i)

Named Entity Recognition (NER), an important task in NLP attempts to classify predefined entities in a sentence. In our assignment, we use *eng.train* for training, *eng.testa* for validation, and *eng.testb* for testing.

A sentence in the dataset is presented below, where the first column is the input word, and the last column is the output tag. The dataset contains four different types of predefined entities: PERSON, LOCATION, ORGANIZATION, and MISC. As shown in the fourth column, the fourth column contains the ground truth entity name and the BIO tag, separated by '-.'

Word			Tag
EU	NNP	I-NP	I-ORG
rejects	VBZ	I-VP	O
German	JJ	I-NP	I-MISC
call	NN	I-NP	O
to	TO	I-VP	O
boycott	VB	I-VP	O
British	JJ	I-NP	I-MISC
lamb	NN	I-NP	O
.	.	O	O

Question one (ii)

There are 5 preprocessing steps in the code provided:

- Replacing all the digits with 0.
- Convert BIO tagging to BIOES tagging.
- Generate words mapping.
- Generate tag mapping.
- Generate characters mapping.
  Mappings here are dictionaries that assign an integer id to every unique word, character, and tag. After the preprocessing step, we found:

> Found 17493 unique words (203621 in total)
> Found 75 unique characters
> Found 19 unique named entity tags

The preprocessed mapping is stored in *data/mapping.pkl*. We use the code provided to load the dataset:

> 14041 / 3250 / 3453 sentences in train / val / test.

Before evaluating the performance of the **BiLSTM_CRF** model (Refer to as base model later on) provided, we write a function to calculate the total number of parameters in the model:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

The base model contains 2284255 parameters. We load the pre-trained model provided at https://github.com/TheAnig/NER-LSTM-CNN-Pytorch/raw/master/trained-model-cpu and evaluate its performance on the test set. The test f1-score is:

> Pre-trained base model: **0.8737**

Question one (iii)

Either a CNN or an LSTM can be used to perform character-level encoding
In the provided code, the CNN is declared as

```
char_cnn = nn.Conv2d(in_channels=1, out_channels=self.out_channels, kernel_size=(3,
char_embedding_dim), padding=(2,0))
```

Whereas the LSTM is defined as

```
char_lstm = nn.LSTM(char_embedding_dim, char_lstm_dim, num_layers=1,
bidirectional=True)
                init_lstm(self.char_lstm)
```

No matter what character-level encoder is used, the extracted character-level representation will be concatenated with higher-level word embeddings and be fed into a Bidirectional-LSTM. However, the input dimension for the higher-level LSTM is different for the different encoders. If CNN is used, the input dimension is word_embedding_dim + out_channeles. If LSTM is used, the input dimension is word_embedding_dim + char_lstm_dim * 2

Question one (iV)

As mentioned in the previous section, word embeddings, concatenated with the character level embedding, are fed into an LSTM. In this section, we will replace the LSTM with CNN.

First of all, let's check the output dimension of the word-level LSTM:

```
with torch.no_grad():
    x = torch.randn(8, 1, 125).cuda()
    lstm = model.lstm
    print(lstm)
    lstm_y, _ = lstm(x)
    print(lstm_y.shape)
```

```
LSTM(125, 200, bidirectional=True)

 torch.Size([8, 1, 400])
```

Clearly, the input of the word-level lstm has the shape (L, N, H_{emb}) where L is the length of sequences, in our case the length of the input sentences, N is the number of samples in the mini-batch. The provided code adopts single-batch training. Thus, $N = 1$. H_{emb} is the dimension of input embedding. In our case, $H_{emb} = H_{word} + H_{char} = 100 + 25 = 125$. The output of word-level lstm has the shape. $(L, N, 2 * H_{ltsm})$ since our LSTM is bi-directional. (we set $H_{lstm} = 200$.)

nn.Conv1d() takes a (N, C_{in}, L_{in}) tensor as input. To make sure that the output dimension is the same after replacing lstm with CNN, we need to transpose the input tensor and set the output channel C_{out} = 400. Moreover, we set the kernel size equals 3; if no padding is added, the output of CNN will be $(N, C_{out}, L_{in} - 3 + 1)$. To not change L, the padding is set to 1.

```
KERNAL_SIZE = 3
with torch.no_grad():
    cnn = nn.Conv1d(in_channels=125, out_channels=400, kernel_size=3, padding=1).cuda()
    x = x.squeeze()
    x_t = x.transpose(0, 1)
    x_t = x_t.unsqueeze(0)
    print('Input: ', x_t.shape)
    y = cnn(x_t)
    print('Output:', y.shape)
lstm_y= lstm_y.view(8, -1)
y = y.squeeze_().t()
print(lstm_y.shape == y.shape)
```

```
True
```

To replace the word-level LSTM layer, we define a new class that inherits the provided model and modify the **get_lstm_features()** function. Although, it should be called **get_cnn_features()** now.

```
 self.word_cnn = nn.Conv1d(
            in_channels=self.embedding_dim + self.out_channels,
            out_channels=2*self.hidden_dim,
            kernel_size=KERNAL_SIZE,
            padding=1
        )
```

Inside the **get_lstm_features()** function, we add:

```
# Our Code for CNN Features
# The embed size now is (L, 125)
embeds = embeds.transpose(0, 1)
# embede size: (125, L)
embeds = embeds.unsqueeze(0)
# embdes size: (1, 125, L)
embeds = self.word_cnn(embeds)
# embedes size: (1, 400, L)
cnn_out = embeds.squeeze_().t()
# embedes size: (400, L) => (L, 400)
lstm_feats = self.hidden2tag(cnn_out)
return lstm_feats
```

Full detail can be found *Assignment2.ipynb*. The **Char-CNN-Word-CNN** model has 2434655 parameters in total. Due to time and computation constraints, we only train the model for 5 epochs with early stops.

The training is very slow (10 mins per epoch, 20 epochs will take more than 3 hours, so we interrupt the training). Since this is just an assignment that doesn't require comparing our model with the state-of-the-art. Later on, we only train 5 epochs for all models. We only train for 5 epochs, **Char-CNN-Word-CNN** model achieves decent performance (0.873 f1-score) on validation sets. On the test sets, it performs a 0.8556 f1-score.

Question one (v)

In this section, we compare the performance of our previous "CNN-Char-CNN-Word" model with the "LSTM-CHAR-CNN-Word" model. To do so, we change the parameters['char_mode'] to 'LSTM'.

```
parameters['char_mode'] = 'LSTM'
```

This **Char-LSTM-Word-CNN** model has 2483155 parameters in total. We compare its performance with our previous **Char-CNN-Word-CNN** model:

We present the results in the table below. **Char-LSTM-Word-CNN** achieves a better f1-score with more parameters. It's also worth noticing that the training time of the **Char-LSTM-Word-CNN** model is significant longer (97mins) than the training time of the **Char-CNN-Word-CNN** model (70mins)

Model	#Parameters	F1-Score
Char-CNN-Word-CNN	2434655	0.8556
Char-LSTM-Word-CNN	2483155	**0.8725**

Question One (vi)

In this section, we increase the number of CNN layers for the word-level encoder.

Let's try a 3-layer-CNN for word encoding. First of all, let's switch back to the CNN-based character-level encoder.

```
cnns = [
            nn.Conv1d(
            in_channels=self.embedding_dim + self.out_channels,
            out_channels=2*self.hidden_dim,
            kernel_size=KERNAL_SIZE,
            padding=1),
            nn.Conv1d(
            in_channels=self.embedding_dim + self.out_channels,
            out_channels=2*self.hidden_dim,
            kernel_size=KERNAL_SIZE,
            padding=1),
            nn.Conv1d(
            in_channels=self.embedding_dim + self.out_channels,
            out_channels=2*self.hidden_dim,
            kernel_size=KERNAL_SIZE,
            padding=1),

        ]
 self.word_cnn = nn.Sequential(*cnns)
```

The number of parameters increases with the layer of CNN. Now, this **Char-CNN-Word-3-CNN** model has 2735455 parameters. However, a larger model size doesn't improve the performance a lot. This model has a 0.8539 test f1-score, which is even worse than the **Char-CNN-Word-CNN**.

Question one (vii)

In this section, we investigate the effect of replacing CNN with dilated CNN:

```
cnns = [
            nn.Conv1d(
            in_channels=self.embedding_dim + self.out_channels,
            out_channels=2*self.hidden_dim,
            kernel_size=KERNAL_SIZE,
            padding=1,
            dilation=1),
            nn.Conv1d(
            in_channels=self.embedding_dim + self.out_channels,
            out_channels=2*self.hidden_dim,
            kernel_size=KERNAL_SIZE,
            padding=1,
            dilation=2),
            nn.Conv1d(
            in_channels=self.embedding_dim + self.out_channels,
            out_channels=2*self.hidden_dim,
            kernel_size=KERNAL_SIZE,
            padding=1,
            dilation=3),

        ]
```

```
self.word_cnn = nn.Sequential(*cnns)
```

Dilated Convolution layer has the same amount of parameters as the regular one. Thus the total number of parameters of this **Char-CNN-Word-3-DiCNN** remains to be 2735455.

In Dilated CNN, the receptive field grows exponentially with the depth. Introducing Dilated CNN leads to better performance: test f1-score 0.8683

Question one (vii)

In the section, we replace the CRF with a simple softmax layer.

To do so we just need to

```
parameters['crf'] = 0
```

However, the original implementation has some bugs to fix. Removing CRF reduces the total number of parameters: From 2735455 to 2735094. The training time also reduces significantly, from 10mins one epoch to 3mins one epoch (This may be caused by the original implementation is not optimized)

However, this **Char-CNN-Word-3-DiCNN-NoCRF** model performs 0.8418 on the test set.

Question one (ix)

We summarize our experimental results with the table below:

Model	Parameters	F1-Score
Baseline	**2284255**	**0.8737**
Char-CNN-Word-CNN	2434655	0.8556
Char-LSTM-Word-CNN	2483155	0.8725
Char-CNN-Word-3CNNs	2735455	0.8539
Char-CNN-Word-3DiCNNs	2735455	0.8683
Char-CNN-Word-3DiCNNs-NoCRF	2735094	0.8418

First of all, the most significant limitation of our experiments is that we only train 5 epochs for each model. Some models may take longer to converge and don't fulfill their full potential in our investigation.

Based on our experiments, a few observation is made:

- LSTM Character level encoder significantly outperforms the CNN Character level encoder without introducing many parameters. In our experiments, **Char-LSTM-Word-CNN** is comparable with the baseline.
- Comparing the performance of **Char-CNN-Word-CNN** with **Char-CNN-Word-3CNNs** reveals that, at least in our setting (Model can be sensitive to hyperparameters like kernel size), adding the number of layers won't lead to better performance and will increase the number of parameters.

- Comparing **Char-CNN-Word-3CNNs** with **Char-CNN-Word-3DiCNNs** tells us dilated convolution layer is an efficient tool to boost the performance without increasing parameters. This can be caused by the exponential growth of the receptive field. However, additional experiments have to be made to support this argument (Increasing CNN layer is not as efficient as Increasing Dilated CNN layer)
- Comparing **Char-CNN-Word-3DiCNNs-NoCRF** with **Char-CNN-Word-3DiCNNs** tells as CRF is essential in the design of the base model.