AI6127: Deep Learning for Natural Language Processing

Ong Jia Hui (G1903467L)
JONG119@e.ntu.edu.sg

Assignment 2

1 Question One [50 marks]

Named Entity Recognition (NER) is an important information extraction task that requires to identify and classify named entities in a given text. These entity types are usually predefined like location, organization, person, and time. In this exercise, you will learn how to develop a neural NER model in Pytorch. In particular, you will learn:

- How to prepare data (input and output) for developing a NER model.
- How to design and fit/train a neural NER model.
- How to use the trained NER model to predict named entity types for a new text.

1.1 Implementation

The baseline model from the code base is the CNN-LSTM-CRF model, which uses CNN for character-level encoding and Bi-directional LSTM (Bi-LSTM) for word-level encoding.

This report will use the following acronyms to denote the different architectures experimented:

- **LSTM** represents a Bi-directional LSTM model.
- **CNN[i]** represents a model with varying number of layers i.
- **DCNN** represents a model with three increasing dilations convolution layers.
- **SFX** represents replacement of the CRF layer with a Softmax layer as output layer.

An example notation would be "LSTM-CNN3-CRF" uses a Bi-LSTM character-level encoder with a three convolution layers word-level encoder and CRF as the output layer.

[CHAR ENCODING]-[WORD ENCODING][LAYERS]-[OUTPUT LAYER]

1.1.1 Parameters Comparison

The following table shows the number of parameters are required in different model architectures. The table is a representative tabulation to show the comparative differences using different modules.

Table 1: Number of parameters between different models

MODEL	*CNN-LSTM-CRF*	*CNN-CNN1-CRF*	*CNN-CNN2-CRF*	*LSTM-CNN1-CRF*	*CNN-DCNN3-CRF*	*LSTM-DCNN3-SFX*
char_embeds	1	1	1	1	1	1
char_layers	2	2	2	8	2	8
word_embeds	1	1	1	1	1	1
word_layers	8	2	4	2	6	6
hidden2tag	2	2	2	2	2	2
TOTAL	**14**	**8**	**10**	**14**	**12**	**18**

Table 1 indicates that the different modules used for character level or word level encoders will alter the number of parameters involved. Bi-LSTM adds a total of 8 parameters (4 weights and 4 biases) to the model, whereas every layer of convolutional network will add 2 parameters (1 weight and 1 bias) to the count.

The following are sample printouts of codes to extract the number of parameters used.

CNN-LSTM-CRF

```
char_embeds.weight : torch.Size([75, 25])
char_cnn3.weight : torch.Size([25, 1, 3, 25])
char_cnn3.bias : torch.Size([25])
word_embeds.weight : torch.Size([17493, 100])
lstm.weight_ih_l0 : torch.Size([800, 125])
lstm.weight_hh_l0 : torch.Size([800, 200])
lstm.bias_ih_l0 : torch.Size([800])
lstm.bias_hh_l0 : torch.Size([800])
lstm.weight_ih_l0_reverse : torch.Size([800, 125])
lstm.weight_hh_l0_reverse : torch.Size([800, 200])
lstm.bias_ih_l0_reverse : torch.Size([800])
lstm.bias_hh_l0_reverse : torch.Size([800])
hidden2tag.weight : torch.Size([19, 400])
hidden2tag.bias : torch.Size([19])
```

CNN-CNN1-CRF

```
char_embeds.weight : torch.Size([75, 25])
char_cnn3.weight : torch.Size([25, 1, 3, 25])
char_cnn3.bias : torch.Size([25])
word_embeds.weight : torch.Size([17493, 100])
word_cnn.0.weight : torch.Size([400, 1, 3, 125])
word_cnn.0.bias : torch.Size([400])
hidden2tag.weight : torch.Size([19, 400])
hidden2tag.bias : torch.Size([19])
```

LSTM-CNN3-CRF

```
char_embeds.weight : torch.Size([75, 25])
char_lstm.weight_ih_l0 : torch.Size([800, 25])
char_lstm.weight_hh_l0 : torch.Size([800, 200])
char_lstm.bias_ih_l0 : torch.Size([800])
char_lstm.bias_hh_l0 : torch.Size([800])
char_lstm.weight_ih_l0_reverse : torch.Size([800, 25])
char_lstm.weight_hh_l0_reverse : torch.Size([800, 200])
char_lstm.bias_ih_l0_reverse : torch.Size([800])
char_lstm.bias_hh_l0_reverse : torch.Size([800])
word_embeds.weight : torch.Size([17493, 100])
word_cnn.0.weight : torch.Size([400, 1, 3, 500])
word_cnn.0.bias : torch.Size([400])
word_cnn.2.weight : torch.Size([400, 400, 3, 1])
word_cnn.2.bias : torch.Size([400])
word_cnn.4.weight : torch.Size([400, 400, 3, 1])
word_cnn.4.bias : torch.Size([400])
hidden2tag.weight : torch.Size([19, 400])
hidden2tag.bias : torch.Size([19])
```

1.1.2 F1 Results

The best F1 scores for training, dev and test sets are tabulated in Table 2:

Table 2: F1 scores of each model after training of 50 epochs

$Model$	$TrainF1$	$DevF1$	$TestF1$
CNN-LSTM-CRF	0.999594	0.928541	0.877675
CNN-LSTM-SFX	0.986734	0.918707	0.874529
CNN-CNN1-CRF	0.998464	0.911263	0.856861
CNN-CNN2-CRF	0.999530	0.926627	0.878879
CNN-CNN3-CRF	0.999637	0.927489	0.880671
CNN-CNN4-CRF	0.999424	0.925577	0.878863
CNN-CNN5-CRF	0.999189	0.928130	0.881349
CNN-CNN5-SFX	0.998358	0.921689	0.876999
LSTM-CNN1-CRF	0.999082	0.919759	0.872361
LSTM-CNN2-CRF	0.999637	0.928420	0.878313
LSTM-CNN3-CRF	0.999658	0.930900	0.881135
LSTM-CNN4-CRF	0.999680	0.927953	0.880078
CNN-DCNN-CRF	0.999765	0.924927	0.875602
CNN-DCNN-SFX	0.999509	0.904004	0.857368
LSTM-DCNN-CRF	0.998655	0.921553	0.881747
LSTM-DCNN-SFX	0.999658	0.903855	0.853424

1.2 Observations

A brief summary of the observations based on the experiment results are as follows:

- The number of parameters required for Bi-LSTM layer is greater than a single CNN layer number of weights and biases due to the additional reverse weights and biases.
- The F1 scores of the models using more than 3 CNN layers for word encoding outperformed the baseline Bi-LSTM word encoding model.
- The model that performs the best on valid (test) set using CNN char encoding and CNN word-level encoding requires 5 layers of CNN (CNN-CNN5-CRF).
- The model that performs the best on valid (test) set using LSTM char encoding and CNN word-level encoding requires 4 layers of CNN (LSTM-CNN4-CRF)
- Using dilated CNN layers trains significantly faster than vanilla CNN. LSTM-DCNN-CRF outperformed all models.
- The F1 scores of CRF models are higher than those using Softmax layer. This could be because Softmax doesn't value any dependencies, whereas CRF takes into account the full sequence to assign the tag and therefore more suitable for NER tasks.

2 Question Two [0 marks]

To help me in future planning with the assignment exercises, please mention how much time you spent on the question.

Training of the different model architectures took a very long time like four days when running on Google Colab. I also spent additional two days to rerun the whole experiment again after a major bug fix, however the results made more sense.

Appendix A Source Code Changes

This section describes the source code changes implemented.

A.1 Additional Model Parameters

```
parameters['crf'] = 0 # 1 to use CRF, 0 to use Softmax
parameters['char_mode']="LSTM"    #LSTM CNN available
parameters['word_mode']="DCNN"    #LSTM CNN DCNN available
parameters['cnn_layers']=1        #For CNN word mode
parameters['conv_dim']=2          #Switch between Conv1d and Conv2d
```

A.2 get_word_features Function

This section shows the alteration made for get_word_features(), renamed from "get_lstm_features()".

```
def get_word_features(self, sentence, chars2, chars2_length, d):
    if self.char_mode == 'LSTM':
        chars_embeds = self.char_embeds(chars2).transpose(0, 1)
        packed = torch.nn.utils.rnn.pack_padded_sequence(chars_embeds, chars2_length)
        lstm_out, _ = self.char_lstm(packed)
        outputs, output_lengths = torch.nn.utils.rnn.pad_packed_sequence(lstm_out)
        outputs = outputs.transpose(0, 1)
        chars_embeds_temp = Variable(torch.FloatTensor(torch.zeros((outputs.size(0), outputs
                                                    .size(2)))))
        if self.use_gpu:
          chars_embeds_temp = chars_embeds_temp.cuda()

        for i, index in enumerate(output_lengths):
          chars_embeds_temp[i] = torch.cat((outputs[i, index-1, :self.char_lstm_dim],
                                            outputs[i, 0, self.char_lstm_dim:]))

        chars_embeds = chars_embeds_temp.clone()

        for i in range(chars_embeds.size(0)):
          chars_embeds[d[i]] = chars_embeds_temp[i]

    if self.char_mode == 'CNN':
        chars_embeds = self.char_embeds(chars2).unsqueeze(1)

        ## Creating Character level representation using Convolutional Neural Netowrk
        ## followed by a Maxpooling Layer
        chars_cnn_out3 = self.char_cnn3(chars_embeds)
        chars_embeds = nn.functional.max_pool2d(chars_cnn_out3,
                                        kernel_size=(chars_cnn_out3.size(2), 1))
                                        .view(chars_cnn_out3.size(0), self.out_channels)

        ## Loading word embeddings
    embeds = self.word_embeds(sentence)

    ## We concatenate the word embeddings and the character level representation
    ## to create unified representation for each word
    embeds = torch.cat((embeds, chars_embeds), 1)

    embeds = embeds.unsqueeze(1)

    ## Dropout on the unified embeddings
    embeds = self.dropout(embeds)

    if self.word_mode == 'LSTM':
        ## Word lstm
        ## Takes words as input and generates a output at each step
        word_out, _ = self.lstm(embeds)
        ## Dropout on the lstm output
        word_out = self.dropout(word_out)
```

```
    elif self.word_mode == 'DCNN':
        ## Word Dilated CNN
        embeds = embeds.permute(1,0,2)
        word_out = self.word_dcnn(embeds.unsqueeze(0))
        word_out = word_out.permute(2, 1, 0, 3)
    elif self.word_mode == 'CNN':
        embeds = embeds.permute(1,0,2)
        word_out = self.word_cnn(embeds.unsqueeze(0))
        word_out = word_out.permute(2, 1, 0, 3)

    ## Reshaping the outputs from the lstm layer
    word_out = word_out.view(len(sentence), self.hidden_dim*2)

    ## Linear layer converts the ouput vectors to tag space
    word_feats = self.hidden2tag(word_out)

    return word_feats
```

A.3 Main Model Class

```
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim,
                 char_to_ix=None, pre_word_embeds=None, char_out_dimension=25,
                                                        char_embedding_dim=25,
                                                        use_gpu=False
                 , use_crf=True, char_mode='CNN', word_mode='CNN', dropout=0, cnn_layers=1):
        '''
        Input parameters:

                vocab_size= Size of vocabulary (int)
                tag_to_ix = Dictionary that maps NER tags to indices
                embedding_dim = Dimension of word embeddings (int)
                hidden_dim = The hidden dimension of the LSTM layer (int)
                char_to_ix = Dictionary that maps characters to indices
                pre_word_embeds = Numpy array which provides mapping from word embeddings to
                                                        word indices
                char_out_dimension = Output dimension from the CNN encoder for character
                char_embedding_dim = Dimension of the character embeddings
                use_gpu = defines availability of GPU,
                    when True: CUDA function calls are made
                    else: Normal CPU function calls are made
                use_crf = parameter which decides if you want to use the CRF layer for
                                                        output decoding
        '''

        super(BiLSTM_CRF, self).__init__()

        #parameter initialization for the model
        self.use_gpu = use_gpu
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag_to_ix = tag_to_ix
        self.use_crf = use_crf
        self.tagset_size = len(tag_to_ix)
        self.out_channels = char_out_dimension
        self.char_mode = char_mode
        self.word_mode = word_mode
        self.cnn_layers = cnn_layers
        self.char_lstm_dim = hidden_dim

        if char_embedding_dim is not None:
            self.char_embedding_dim = char_embedding_dim

            #Initializing the character embedding layer
```

```
        self.char_embeds = nn.Embedding(len(char_to_ix), char_embedding_dim)
        init_embedding(self.char_embeds.weight)

        #Performing LSTM encoding on the character embeddings
        if self.char_mode == 'LSTM':
            self.char_lstm = nn.LSTM(char_embedding_dim, self.char_lstm_dim, num_layers=
                                                        1, bidirectional=True)
            init_lstm(self.char_lstm)

        #Performing CNN encoding on the character embeddings
        if self.char_mode == 'CNN':
            self.char_cnn3 = nn.Conv2d(in_channels=1, out_channels=self.out_channels,
                                                        kernel_size=(3,
                                                        char_embedding_dim), padding=(
                                                        2,0))

    #Creating Embedding layer with dimension of ( number of words * dimension of each
                                                    word)
    self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
    if pre_word_embeds is not None:
        #Initializes the word embeddings with pretrained word embeddings
        self.pre_word_embeds = True
        self.word_embeds.weight = nn.Parameter(torch.FloatTensor(pre_word_embeds))
    else:
        self.pre_word_embeds = False

    #Initializing the dropout layer, with dropout specificed in parameters
    self.dropout = nn.Dropout(dropout)

    if char_mode == 'LSTM':
        kernel_size = (3, embedding_dim + self.char_lstm_dim * 2)
        pad_size = (1, 0)
    else:
        kernel_size = (3, embedding_dim + self.out_channels)
        pad_size = (1, 0)
    if self.word_mode == 'LSTM':
      #Lstm Layer:
      #input dimension: word embedding dimension + character level representation
      #bidirectional=True, specifies that we are using the bidirectional LSTM
      if self.char_mode == 'LSTM':
          self.lstm = nn.LSTM(embedding_dim+char_lstm_dim*2, hidden_dim, bidirectional=
                                                        True)
      if self.char_mode == 'CNN':
          self.lstm = nn.LSTM(embedding_dim+self.out_channels, hidden_dim, bidirectional
                                                        =True)
      #Initializing the lstm layer using predefined function for initialization
      init_lstm(self.lstm)
    elif self.word_mode == 'DCNN':
        dilations = [1, 2, 3]
        self.word_dcnn = nn.Sequential(
                  nn.Conv2d(in_channels=1, out_channels=hidden_dim * 2,
                            kernel_size=kernel_size, padding=pad_size, dilation=(
                                                                    dilations[0],1
                                                                    )),
                  nn.ReLU(inplace=True),
                  # nn.Dropout(dropout),
                  nn.Conv2d(in_channels=hidden_dim * 2, out_channels=hidden_dim * 2,
                            kernel_size=(3, 1), padding=(2, 0), dilation=(dilations[1],1
                                                                    )),
                  nn.ReLU(inplace=True),
                  # nn.Dropout(dropout),
                  nn.Conv2d(in_channels=hidden_dim * 2, out_channels=hidden_dim * 2,
                            kernel_size=(3, 1), padding=(3, 0), dilation=(dilations[2],1
                                                                    )),
                  nn.ReLU(inplace=True),
                  # nn.Dropout(dropout)
          )
    elif self.word_mode == 'CNN':
```

```
            self.word_cnn = nn.Sequential(
                      nn.Conv2d(in_channels=1, out_channels=hidden_dim * 2,
                                kernel_size=kernel_size, padding=pad_size),
                      nn.ReLU(inplace=True)
            )
            seq_ctr = 2
            kernel_size = (3, 1)
            for idx in range(self.cnn_layers-1):
                self.word_cnn.add_module(str(seq_ctr), nn.Conv2d(in_channels=hidden_dim * 2,
                              out_channels=hidden_dim * 2,
                              kernel_size=kernel_size, padding=pad_size))
                self.word_cnn.add_module(str(seq_ctr+1), nn.ReLU(inplace=True))
                seq_ctr = seq_ctr + 2

        # Linear layer which maps the output of the bidirectional LSTM into tag space.
        self.hidden2tag = nn.Linear(hidden_dim*2, self.tagset_size)

        #Initializing the linear layer using predefined function for initialization
        init_linear(self.hidden2tag)

        if self.use_crf:
            # Matrix of transition parameters.  Entry i,j is the score of transitioning *to*
                                                    i *from* j.
            # Matrix has a dimension of (total number of tags * total number of tags)
            self.transitions = nn.Parameter(
                torch.zeros(self.tagset_size, self.tagset_size))

            # These two statements enforce the constraint that we never transfer
            # to the start tag and we never transfer from the stop tag
            self.transitions.data[tag_to_ix[START_TAG], :] = -10000
            self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000

    #assigning the functions, which we have defined earlier
    _score_sentence = score_sentences
    _get_word_features = get_word_features
    _forward_alg = forward_alg
    viterbi_decode = viterbi_algo
    neg_log_likelihood = get_neg_log_likelihood
    forward = forward_calc
```

A.4 Softmax Fix for PyTorch Version

A minor modification is required to populate the output of Softmax for newer PyTorch version in the forward_calc function.

```
else:
    score, tag_seq = torch.max(feats, 1)
    tag_seq = tag_seq.cpu().tolist() #list(tag_seq.cpu().data)
```