CSCI_544_HW4

November 13, 2021

Nidhi Rajkumar Saini USCID: 3730422872

[38]:
```
#path that contains folder you want to copy
from google.colab import drive
drive.mount('/content/drive')
%cd /content/
%cp -r /content/drive/MyDrive/USC/NLP_CSCI_544/HW4/* ./
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content
```

1 Import required libraries

[39]:
```
import pandas as pd
import numpy as np
import os
import random
import tqdm

import torch
import torch.nn as nn
from torch.nn.utils.rnn import pack_padded_sequence,pad_packed_sequence
```

2 Define flags and constants

[40]:
```
# special token for unknown words
UNK = "<UNK>"
# special token for padding
PAD = "<PAD>"

random_seed = 42

model_file_name = 'BLSTM1.pt'

# checks whether to use CNN for character-level representation
```

```
use_cnn_for_char_level = False

# vector dimension for word embeddings
word_embedding_dim = 100

# vector dimension for character embeddings
char_embedding_dim = 30



# None if not using glove embedding else contains glove embedding dictionary
pre_embeddings = None


# setting the hyperparameters as per the task description
lstm_hidden_dim = 256
dropout = 0.33
output_dimension = 128

# number of filters in the CNN layer
out_channels = 30


# setting all the other hyperparameters
num_epochs = 100
batch_size = 16
learning_rate = 0.015
momentum = 0.9
decay_rate = 0.05
grad_clip = 5.0

loss_func = nn.CrossEntropyLoss

# creating dictionary for storing tag and corresponding index
tag_to_idx = {}
idx_to_tag = {}

# creating dictionary for storing word and corresponding index in training data
word_to_idx = {}
idx_to_word = {}

# creating dictionary for storing character and corresponding index in training
 ↪data
char_to_idx = {}
idx_to_char = {}
```

3 Load data files

[41]:
```python
train = pd.read_csv('data/train', header = None, names = ['idx','word','tag'],␣
↪sep ='\s',na_values=['<NAN>'], keep_default_na=False)
train.head(5)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: ParserWarning:
Falling back to the 'python' engine because the 'c' engine does not support
regex separators (separators > 1 char and different from '\s+' are interpreted
as regex); you can avoid this warning by specifying engine='python'.
  """Entry point for launching an IPython kernel.
```

[41]:

	idx	word	tag
0	1	EU	B-ORG
1	2	rejects	O
2	3	German	B-MISC
3	4	call	O
4	5	to	O

[42]:
```python
dev = pd.read_csv('data/dev', header = None, names = ['idx','word','tag'], sep␣
↪= '\s',na_values=['<NAN>'], keep_default_na=False)
dev.head(5)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: ParserWarning:
Falling back to the 'python' engine because the 'c' engine does not support
regex separators (separators > 1 char and different from '\s+' are interpreted
as regex); you can avoid this warning by specifying engine='python'.
  """Entry point for launching an IPython kernel.
```

[42]:

	idx	word	tag
0	1	CRICKET	O
1	2	-	O
2	3	LEICESTERSHIRE	B-ORG
3	4	TAKE	O
4	5	OVER	O

[43]:
```python
test = pd.read_csv('data/test', header = None, names = ['idx','word'], sep =␣
↪'\s',na_values=['<NAN>'], keep_default_na=False)
test.head(5)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: ParserWarning:
Falling back to the 'python' engine because the 'c' engine does not support
regex separators (separators > 1 char and different from '\s+' are interpreted
as regex); you can avoid this warning by specifying engine='python'.
  """Entry point for launching an IPython kernel.
```

[43]:
```
   idx    word
0    1  SOCCER
1    2       -
2    3   JAPAN
3    4     GET
4    5   LUCKY
```

4 Define functions

- The is_word function checks whether a given string contains any valid characters (it returns False for strings which are only numbers or dates).
- This helps in reducing the vocabulary size, since numbers and dates always have an O tag.
- This step was made differently for model 1 and the other 2 models

[44]:
```
def is_word(word,num="1"):
  for char in word:
    if num!="1" and char in␣
↪set(list("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ,.?()\'\"")):
      return True
    elif num=="1" and char in␣
↪set(list("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ,.?;;
↪()\'\"[]{}<>-_=+*")):
      return True
  return False
```

- The *make_data* function creates a list of 3-tuples where first element is list of preprocessed words, second element is list of actual words and third element is list of corresponding tags.

[45]:
```
def make_data(df,has_tags=True,num="1"):
    sentences = []
    actual_sentences = []
    sentence_tags = []
    tags = []
    sentence = None
    actual_sentence = None
    for row in df.values.tolist():
        if row[0] == 1:
            if sentence:
                sentence_tags.append(tags)
                sentences.append(sentence)
                actual_sentences.append(actual_sentence)

            sentence = []
            actual_sentence = []
            tags = []
        if not is_word(row[1],num=num):
```

```
            sentence.append('<0>')
        else:
            sentence.append(row[1])
        actual_sentence.append(row[1])
        if has_tags:
          tags.append(row[2])
        else:
          tags.append("O")
    sentence_tags.append(tags)
    sentences.append(sentence)
    actual_sentences.append(actual_sentence)

    return list(zip(sentences,actual_sentences,sentence_tags))
```

- The *make_tag_to_idx* function creates the tag to index dictionary.
- The *make_word_to_idx* function creates the word to index dictionary.
- Both functions use the output of the *make_data* function.

[46]:
```
def make_tag_to_idx(data):
    if PAD not in tag_to_idx:
      idx_to_tag[len(tag_to_idx)] = PAD
      tag_to_idx[PAD] = len(tag_to_idx)

    for _,_,tags in data:
      for tag in tags:
        if tag not in tag_to_idx:
          idx_to_tag[len(tag_to_idx)] = tag
          tag_to_idx[tag] = len(tag_to_idx)


def make_word_to_idx(data,is_valid_or_test=False):

  if is_valid_or_test:
    for sentence,_,_ in data:
      for word in sentence:
        if word not in word_to_idx:
          idx_to_word[len(word_to_idx)] = word
          word_to_idx[word] = len(word_to_idx)

  else:
    idx_to_word[0] = PAD
    word_to_idx[PAD] = 0
    idx_to_word[1] = UNK
    word_to_idx[UNK] = 1

    idx_to_char[0] = PAD
    char_to_idx[PAD] = 0
```

```
    idx_to_char[1] = UNK
    char_to_idx[UNK] = 1

    for sentence,_,_ in data:
      for word in sentence:
        if word not in word_to_idx:
          idx_to_word[len(word_to_idx)] = word
          word_to_idx[word] = len(word_to_idx)

    for sentence,_,_ in data:
      for word in sentence:
        for char in sentence:
          if char not in char_to_idx:
            idx_to_char[len(char_to_idx)] = char
            char_to_idx[char] = len(char_to_idx)
```

- The *get_embedding_data* function creates the embedding dictionary from the glove.6B.100d text file.
- The *build_embedding_table* function creates a numpy matrix for all word embeddings. If glove embedding dictionary is provided then, that is used for creating a table otherwise the table is created with random entries.

[47]:
```
def get_embedding_data(filename,dim):
  embedding = dict()
  with open(filename,'r',encoding='utf-8') as f:
    for line in f.readlines():
      line = line.strip()
      if len(line) == 0:
        continue
      line_split = line.split() #word followed by dim numbers
      embedd = np.empty([1,dim])
      embedd[:] = line_split[1:]
      word = line_split[0]
      embedding[word] = embedd
  return embedding

def build_embedding_table():
    global embeddings
    scale = np.sqrt(3.0 / word_embedding_dim)
    caps = set(list("ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
    if pre_embeddings is not None:
        embeddings = np.empty([len(word_to_idx), word_embedding_dim])
        for word in word_to_idx:
            if word in pre_embeddings:
                embeddings[word_to_idx[word], :] = np.
↪append(pre_embeddings[word],[[-1]],axis=1)
            elif word.lower() in pre_embeddings:
```

```python
                embeddings[word_to_idx[word], :] = np.
↪append(pre_embeddings[word.lower()],[[1]],axis=1)
            else:
                is_cap = 1 if word[0] in caps else -1
                embeddings[word_to_idx[word], :] = np.append(
                    np.random.uniform(-scale, scale, [1, word_embedding_dim-1]),
                                                        [[is_cap]],axis=1)
    else:
        embeddings = np.empty([len(word_to_idx), word_embedding_dim])
        for word in word_to_idx:
            embeddings[word_to_idx[word], :] = np.random.uniform(-scale, scale,␣
↪[1, word_embedding_dim])
```

- The *make_numeric_data* function uses the word_to_idx, tag_to_idx and char_to_idx dictionaries in order to change the dataset into a list of indices.
- The *data_batching* function creates batches from entire dataset. All sentences in a batch are padded to the same length using the *padded_batching* function.

[48]:
```python
def make_numeric_data(data):
  list_sent_ids = []
  list_tag_ids = []
  list_char_ids = []
  for sentence,_,tags in data:
    sentence_ids = []
    tag_ids = []
    char_ids = []
    for word in sentence:
      if word in word_to_idx:
        sentence_ids.append(word_to_idx[word])
      else:
        sentence_ids.append(word_to_idx[UNK])

      char_id = []
      for c in word:
        if c in char_to_idx:
          char_id.append(char_to_idx[c])
        else:
          char_id.append(char_to_idx[UNK])
      char_ids.append(char_id)

    for tag in tags:
      if tag in tag_to_idx:
        tag_ids.append(tag_to_idx[tag])
      else:
        tag_ids.append("O")
    list_sent_ids.append(sentence_ids)
    list_tag_ids.append(tag_ids)
```

```
    list_char_ids.append(char_ids)

  return list(zip(list_sent_ids,list_char_ids,list_tag_ids))

def data_batching(data):
  num_instances = len(data)
  num_batches = num_instances // batch_size + 1 if num_instances != 0 else␣
↪num_instances//batch_size
  batched_data = []
  for i in range(num_batches):
    batch = data[i*batch_size:(i+1)*batch_size]
    batched_data.append(padded_batching(batch))
  return batched_data

def padded_batching(data):
  batch_size = len(data)
  batch_data = data

  word_seq_len = torch.LongTensor(list(map(lambda x: len(x[0]), batch_data)))
  max_word_seq_len = word_seq_len.max()
  # print(batch_data)
  char_seq_len = torch.LongTensor(
      [list(map(len,x[1])) + [1]* (int(max_word_seq_len) - len(x[1])) for x in␣
↪batch_data]
  )

  max_char_seq_len = char_seq_len.max()

  word_seq_tensor = torch.zeros((batch_size,max_word_seq_len),dtype=torch.long)
  char_seq_tensor = torch.zeros((batch_size,max_word_seq_len,max_char_seq_len),␣
↪dtype=torch.long)
  tag_seq_tensor = torch.zeros((batch_size,max_word_seq_len),dtype=torch.long)

  for idx in range(batch_size):
    word_seq_tensor[idx,:word_seq_len[idx]] = torch.
↪LongTensor(batch_data[idx][0])
    tag_seq_tensor[idx,:word_seq_len[idx]] = torch.
↪LongTensor(batch_data[idx][2])

    for word_idx in range(word_seq_len[idx]):
      char_seq_tensor[idx,word_idx,:char_seq_len[idx,word_idx]] = torch.
↪LongTensor(
          batch_data[idx][1][word_idx]
      )

    for word_idx in range(word_seq_len[idx],max_word_seq_len):
```

```
        char_seq_tensor[idx,word_idx,0:1] = torch.LongTensor([char_to_idx[PAD]])

  return word_seq_tensor,word_seq_len,char_seq_tensor,tag_seq_tensor,data
```

- The *initialize_linear_layer* function initializes the weights and bias of a linear layer using xavier initialization.
- The *initialize_lstm_layer* function initializes the weights of the LSTM layer using an orthogonal matrix. It initializes the bias by sampling a normal distribution.
- The *BLSTM* class defines feed-forward architecture of the model used in tasks 1 and 2. It also takes care of character-level CNN for the bonus task.

[49]:
```
def initialize_linear_layer(layer):
    nn.init.xavier_normal_(layer.weight.data)
    nn.init.normal_(layer.bias.data)

def initialize_lstm_layer(layer):
    for param in layer.parameters():
        if len(param.shape) >= 2:
            nn.init.orthogonal_(param.data)
        else:
            nn.init.normal_(param.data)

class BLSTM(nn.Module):
  def __init__(self):
    super(BLSTM,self).__init__()

    self.vocab_size = len(word_to_idx)
    self.tag_size = len(tag_to_idx)

    if use_cnn_for_char_level:
      self.char_embedd = nn.Embedding(len(char_to_idx),char_embedding_dim)
      nn.init.xavier_uniform_(self.char_embedd.weight)
      self.char_cnn = nn.Conv2d(
          in_channels = 1,
          out_channels = out_channels,
          kernel_size = (3,char_embedding_dim),
          padding = (2,0)
      )

    self.word_embedd = nn.Embedding(self.vocab_size,word_embedding_dim)
    self.word_embedd.weight = nn.Parameter(torch.FloatTensor(embeddings))

    self.dropout = nn.Dropout(dropout)

    if use_cnn_for_char_level:
      self.lstm = nn.
↪LSTM(word_embedding_dim+out_channels,lstm_hidden_dim,num_layers=1,
```

```
                          batch_first = True, bidirectional=True)

    else:
      self.lstm = nn.LSTM(word_embedding_dim,lstm_hidden_dim,num_layers=1,
                          batch_first = True, bidirectional=True)

    initialize_lstm_layer(self.lstm)
    self.dropout = nn.Dropout(dropout)
    self.hidden = nn.Linear(lstm_hidden_dim*2, output_dimension)
    self.ELU = nn.ELU()
    self.out_layer = nn.Linear(output_dimension,self.tag_size)
    initialize_linear_layer(self.out_layer)

  def forward(self,word_seq_tensor,word_seq_len,char_seq_tensor):
    word_embedds = self.word_embedd(word_seq_tensor)

    if use_cnn_for_char_level:
      batch_size = char_seq_tensor.size(0)
      sent_len = char_seq_tensor.size(1)
      char_seq_tensor = char_seq_tensor.view(batch_size*sent_len,-1)
      char_embedds = self.char_embedd(char_seq_tensor).unsqueeze(1)
      cnn_out = self.char_cnn(char_embedds)
      char_embedds = nn.functional.max_pool2d(cnn_out,kernel_size=(cnn_out.
↪size(2),1)).view(
          cnn_out.size(0),
          out_channels
      )

      char_features = char_embedds.view(batch_size,sent_len,-1)
      word_embedds = torch.cat([word_embedds,char_features],axis=2)

    word_embs = self.dropout(word_embedds)
    sorted_seq_len,idx = word_seq_len.sort(0,descending=True)
    _,sorted_idx = idx.sort(0,descending=False)
    sorted_seq_tensor = word_embs[idx]
    packed_words = pack_padded_sequence(sorted_seq_tensor,sorted_seq_len,True)
    output, _ = self.lstm(packed_words,None)
    output,_ = pad_packed_sequence(output,batch_first=True)
    output = output[sorted_idx]
    output = self.dropout(output)
    output = self.hidden(output)
    output = self.ELU(output)
    output = self.out_layer(output)
    return output
```

- The *evaluate_model* function creates the .out prediction file with required output. It also runs the perl script to evaluate F-1 score.

- It returns the best F-1 score until current epoch, the current F-1 score and a flag called save, which tells the training function when to save the model.

[50]:
```
def evaluate_model(model,data,act_data,best_f1_score,name="dev",model_num="1"):
  save = False
  f1_score = 0.0

  pred_file_name = name + model_num + '.out'
  score_file_name = name+model_num+'_score.out'

  pred_file = open(pred_file_name,'w')

  for idx in range(len(data)):
    batch = data[idx]
    rows = batch[4]
    pred_scores = model(*batch[:3])
    pred_tags = pred_scores.argmax(-1)
    orig_data = act_data[idx*batch_size:(idx+1)*batch_size]
    for i,(row,preds) in enumerate(list(zip(rows,pred_tags))):
      for idx,(word,gold,pred) in␣
↪enumerate(zip(orig_data[i][1],row[2],preds),start=1):
        #print(str(idx),word,gold,pred.item())
        pred_file.write(' '.
↪join([str(idx),word,idx_to_tag[gold],idx_to_tag[pred.item()]]))
        pred_file.write('\n')
      pred_file.write('\n')

  pred_file.close()
  os.system('perl conll03eval.txt < %s > %s' % (pred_file_name,score_file_name))
  eval_lines = [l.rstrip() for l in open(score_file_name,'r',encoding='utf-8')]

  for i, line in enumerate(eval_lines):
    print(line)
    if i==1:
      f1_score = float(line.strip().split()[-1])
      if f1_score>best_f1_score:
        best_f1_score = f1_score
        save = True
        print('Best F1 Score:',f1_score)

  return best_f1_score,f1_score,save
```

- The *train_model* function contains the main loop for training the model.
- First, the model back propagates and learns on training batches and after each epoch, the F-1 score on development dataset is calculated. The model is saved whenever it beats its current best F-1 score.

[51]:
```
def train_model(model,train_batches,dev_batches,model_num="1",curr_epoch = 0):
  optimizer = torch.optim.SGD(model.
↪parameters(),lr=learning_rate,momentum=momentum)
  if curr_epoch>0:
    for g in optimizer.param_groups:
      g['initial_lr'] = learning_rate
  lambda_schedule = lambda x: 1/(1+x*0.05)
  scheduler = torch.optim.lr_scheduler.
↪LambdaLR(optimizer,lambda_schedule,verbose=True,last_epoch=curr_epoch-1)

  losses = []
  best_dev_f1 = -1.

  loss_function = loss_func(ignore_index=tag_to_idx[PAD])

  for epoch in range(1,num_epochs+1):
    total_loss = 0
    model.train()
    model.zero_grad()
    for idx in tqdm.notebook.tqdm(np.random.
↪permutation(len(train_batches)),total=len(train_batches)):
      batch = train_batches[idx]
      pred_scores = model(*batch[:3])
      out = pred_scores.view(-1,pred_scores.shape[-1])
      gold = batch[3].view(-1)
      loss = loss_function(out,gold)
      total_loss += loss.data
      loss.backward()
      nn.utils.clip_grad_norm_(model.parameters(),grad_clip)
      optimizer.step()
      model.zero_grad()
    losses.append(total_loss)
    model.eval()
    best_dev_f1 , dev_f1, save =␣
↪evaluate_model(model,dev_batches,validation_data,best_dev_f1,"dev",model_num)

    if save:
      torch.save(model.state_dict(),model_file_name)

    model.zero_grad()
    scheduler.step()
  return None
```

- The *infer* function creates the .out prediction file with required output without the column for gold tags.

[94]:
```
def infer(model,data,act_data,filename):

  pred_file = open(filename,'w')

  for idx in range(len(data)):
    batch = data[idx]
    rows = batch[4]
    pred_scores = model(*batch[:3])
    orig_data = act_data[idx*batch_size:(idx+1)*batch_size]
    pred_tags = pred_scores.argmax(-1)
    for i,(row,preds) in enumerate(list(zip(rows,pred_tags))):
      for idx,(word,pred) in enumerate(zip(orig_data[i][1],preds),start=1):
        pred_file.write(' '.join([str(idx),word,idx_to_tag[pred.item()]]))
        pred_file.write('\n')
      pred_file.write('\n')

  pred_file.close()
```

5 Task 1: Simple Bidirectional LSTM model

- We use all the above functions to preprocess and prepare our data for training the model.
- After forming the initial training and validation data, we sort the sentences (and corresponding tags) according to length.
- The model is trained with the following parameters:
- use_cnn_for_char_level = False
- word_embedding_dim = 100
- pre_embeddings = None
- lstm_hidden_dim = 256
- dropout = 0.33
- output_dimension = 128
- num_epochs = 100
- batch_size = 16
- learning_rate = 0.015
- momentum = 0.9
- decay_rate = 0.05
- grad_clip = 5.0
- loss function = Cross Entropy Loss

- optimizer = SGD

[ ]:
```
word_embedding_dim = 100
pre_embeddings = None
model_file_name = 'BLSTM1.pt'
batch_size = 16
use_cnn_for_char_level = False

word_to_idx = {}
idx_to_word = {}

training_data = sorted(make_data(train),key=lambda x:len(x[0]))
validation_data = sorted(make_data(dev),key=lambda x:len(x[0]))
make_word_to_idx(training_data)
make_tag_to_idx(training_data)
build_embedding_table()

training_data_tensors = make_numeric_data(training_data)
validation_data_tensors = make_numeric_data(validation_data)

print(training_data[0])
print(training_data_tensors[0])
train_batches = data_batching(training_data_tensors)
dev_batches = data_batching(validation_data_tensors)

random.seed(random_seed)
model = BLSTM()
train_model(model,train_batches,dev_batches)
```

- Downloading the model with best F-1 score

[ ]:
```
from google.colab import files
files.download('BLSTM1.pt')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

- Loading the best model and running evaluation of loaded model on dev dataset to verify correctness.

[103]:
```
word_embedding_dim = 100
pre_embeddings = None
model_file_name = 'BLSTM1.pt'
batch_size = 16
```

```
use_cnn_for_char_level = False


word_to_idx = {}
idx_to_word = {}

training_data = sorted(make_data(train),key=lambda x:len(x[0]))
validation_data = sorted(make_data(dev),key=lambda x:len(x[0]))
make_word_to_idx(training_data)
make_tag_to_idx(training_data)
build_embedding_table()

validation_data = make_data(dev)
validation_data_tensors = make_numeric_data(validation_data)
dev_batches = data_batching(validation_data_tensors)
```

[104]:
```
model = BLSTM()
model.load_state_dict(torch.load(model_file_name))
```

[104]:
```
<All keys matched successfully>
```

[105]:
```
model.eval()
best_dev_f1 , dev_f1, save =␣
↪evaluate_model(model,dev_batches,validation_data,-1,"dev","1")
```

```
processed 51578 tokens with 5942 phrases; found: 5155 phrases; correct: 4300.
accuracy:  95.40%; precision:  83.41%; recall:  72.37%; FB1:  77.50
Best F1 Score: 77.5
              LOC: precision:  90.37%; recall:  82.25%; FB1:  86.12  1672
             MISC: precision:  87.81%; recall:  75.81%; FB1:  81.37  796
              ORG: precision:  78.18%; recall:  64.65%; FB1:  70.78  1109
              PER: precision:  77.50%; recall:  66.40%; FB1:  71.52  1578
```

What are the precision, recall and F1 score on the dev data?

- Precision = 83.41%
- Recall = 72.37%
- F1-Score = 77.50

6 TASK 2: Using GloVe word embeddings

- We use all the above functions to preprocess and prepare our data for training the model.
- For Task 2, we also load the glove embeddings for words in the train, dev and test sets into our embedding layer.
- After forming the initial training and validation data, we sort the sentences (and corresponding tags) according to length.
- The model is trained with the following parameters:

- use_cnn_for_char_level = False
- word_embedding_dim = 100 (adding 1 more feature for capitalization makes the dimension 101)
- pre_embeddings = GloVe Embeddings Dictionary
- lstm_hidden_dim = 256
- dropout = 0.33
- output_dimension = 128
- num_epochs = 100
- batch_size = 16
- learning_rate = 0.015
- momentum = 0.9
- decay_rate = 0.05
- grad_clip = 5.0
- loss function = Cross Entropy Loss
- optimizer = SGD

[59]:
```
!gzip -d glove.6B.100d.gz
```

[ ]:
```
word_embedding_dim = 101 #including 1 dimension for capitalization
pre_embeddings = get_embedding_data('glove.6B.100d',100)
model_file_name = 'BLSTM2.pt'
batch_size = 16
use_cnn_for_char_level = False

word_to_idx = {}
idx_to_word = {}

training_data = sorted(make_data(train,num="2"),key=lambda x:len(x[0]))
validation_data = sorted(make_data(dev,num="2"),key=lambda x:len(x[0]))
testing_data = sorted(make_data(test,has_tags=False,num="2"),key=lambda x:
 ↪len(x[0]))

make_word_to_idx(training_data)
make_word_to_idx(validation_data,is_valid_or_test=True)
make_word_to_idx(testing_data,is_valid_or_test=True)
make_tag_to_idx(training_data)

build_embedding_table()

training_data_tensors = make_numeric_data(training_data)
```

```
validation_data_tensors = make_numeric_data(validation_data)

train_batches = data_batching(training_data_tensors)
dev_batches = data_batching(validation_data_tensors)

random.seed(random_seed)
model = BLSTM()
train_model(model,train_batches,dev_batches,model_num="2")
```

- Downloading the model with best F-1 score

[ ]:
```
from google.colab import files
files.download('BLSTM2.pt')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

- Loading the best model and running evaluation of loaded model on dev dataset to verify correctness.

[100]:
```
word_embedding_dim = 101 #including 1 dimension for capitalization
pre_embeddings = get_embedding_data('glove.6B.100d',100)
model_file_name = 'BLSTM2.pt'
batch_size = 16
use_cnn_for_char_level = False

word_to_idx = {}
idx_to_word = {}

training_data = sorted(make_data(train,num="2"),key=lambda x:len(x[0]))
validation_data = sorted(make_data(dev,num="2"),key=lambda x:len(x[0]))
testing_data = sorted(make_data(test,has_tags=False,num="2"),key=lambda x:
↪len(x[0]))

make_word_to_idx(training_data)
make_word_to_idx(validation_data,is_valid_or_test=True)
make_word_to_idx(testing_data,is_valid_or_test=True)
make_tag_to_idx(training_data)

build_embedding_table()

validation_data = make_data(dev,num="2")
validation_data_tensors = make_numeric_data(validation_data)
dev_batches = data_batching(validation_data_tensors)
```

[101]:
```
model = BLSTM()
model.load_state_dict(torch.load(model_file_name))
```

[101]:
```
<All keys matched successfully>
```

[102]:
```
model.eval()
best_dev_f1 , dev_f1, save =␣
↪evaluate_model(model,dev_batches,validation_data,-1,"dev","2")
```

```
processed 51578 tokens with 5942 phrases; found: 6008 phrases; correct: 5394.
accuracy:  98.32%; precision:  89.78%; recall:  90.78%; FB1:  90.28
Best F1 Score: 90.28
              LOC: precision:  94.05%; recall:  93.79%; FB1:  93.92  1832
             MISC: precision:  83.05%; recall:  80.80%; FB1:  81.91  897
              ORG: precision:  81.66%; recall:  86.35%; FB1:  83.94  1418
              PER: precision:  95.00%; recall:  95.98%; FB1:  95.49  1861
```

What are the precision, recall and F1 score on the dev data?

- Precision = 89.78%
- Recall = 90.78%
- F1-Score = 90.28

7 Bonus Task: Using BLSTM-CNN with GloVe word embeddings

- We use all the above functions to preprocess and prepare our data for training the model.
- For this task, we load the glove embeddings for words in the train, dev and test sets into our embedding layer.
- Additionaly, we also make use_cnn_for_char_level = True, this in turn gives the model a CNN layer which takes character level embedding as input and creates a character-level representation for a word. These representations are then concatenated with the word embeddings to create the inputs for the LSTM Layer.
- After forming the initial training and validation data, we sort the sentences (and corresponding tags) according to length.
- The model is trained with the following parameters:
- use_cnn_for_char_level = True
- word_embedding_dim = 100 (adding 1 more feature for capitalization makes the dimension 101)
- pre_embeddings = GloVe Embeddings Dictionary
- lstm_hidden_dim = 256
- dropout = 0.33
- output_dimension = 128

- num_epochs = 100
- batch_size = 16
- learning_rate = 0.015
- momentum = 0.9
- decay_rate = 0.05
- grad_clip = 5.0
- loss function = Cross Entropy Loss
- optimizer = SGD

[ ]:
```
word_embedding_dim = 101 #including 1 dimension for capitalization
pre_embeddings = get_embedding_data('glove.6B.100d',100)
model_file_name = 'BLSTM3.pt'
batch_size = 16
use_cnn_for_char_level = True

word_to_idx = {}
idx_to_word = {}
char_to_idx = {}
idx_to_char = {}

training_data = sorted(make_data(train,num="3"),key=lambda x:len(x[0]))
validation_data = sorted(make_data(dev,num="3"),key=lambda x:len(x[0]))
testing_data = sorted(make_data(test,has_tags=False,num="3"),key=lambda x:
 ↪len(x[0]))

make_word_to_idx(training_data)
make_word_to_idx(validation_data,is_valid_or_test=True)
make_word_to_idx(testing_data,is_valid_or_test=True)
make_tag_to_idx(training_data)

build_embedding_table()

training_data_tensors = make_numeric_data(training_data)
validation_data_tensors = make_numeric_data(validation_data)

train_batches = data_batching(training_data_tensors)
dev_batches = data_batching(validation_data_tensors)

random.seed(random_seed)
model = BLSTM()
train_model(model,train_batches,dev_batches,model_num="3")
```

- Downloading the model with best F-1 score

[ ]:
```
from google.colab import files
files.download('BLSTM3.pt')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

- Loading the best model and running evaluation of loaded model on dev dataset to verify correctness.

[97]:
```
word_embedding_dim = 101 #including 1 dimension for capitalization
pre_embeddings = get_embedding_data('glove.6B.100d',100)
model_file_name = 'BLSTM3.pt'
batch_size = 16
use_cnn_for_char_level = True

word_to_idx = {}
idx_to_word = {}
char_to_idx = {}
idx_to_char = {}


training_data = sorted(make_data(train,num="3"),key=lambda x:len(x[0]))
validation_data = sorted(make_data(dev,num="3"),key=lambda x:len(x[0]))
testing_data = sorted(make_data(test,has_tags=False,num="3"),key=lambda x:
 ↪len(x[0]))

make_word_to_idx(training_data)
make_word_to_idx(validation_data,is_valid_or_test=True)
make_word_to_idx(testing_data,is_valid_or_test=True)

make_tag_to_idx(training_data)

build_embedding_table()

validation_data = make_data(dev,num="3")
validation_data_tensors = make_numeric_data(validation_data)
dev_batches = data_batching(validation_data_tensors)
```

[98]:
```
model = BLSTM()
model.load_state_dict(torch.load(model_file_name))
```

[98]:
```
<All keys matched successfully>
```

[99]:
```
model.eval()
best_dev_f1 , dev_f1, save =␣
 ↪evaluate_model(model,dev_batches,validation_data,-1,"dev","3")
```

```
processed 51578 tokens with 5942 phrases; found: 6024 phrases; correct: 5413.
accuracy:  98.35%; precision:  89.86%; recall:  91.10%; FB1:  90.47
Best F1 Score: 90.47
              LOC: precision:  94.39%; recall:  93.36%; FB1:  93.87  1817
             MISC: precision:  84.99%; recall:  81.67%; FB1:  83.30  886
              ORG: precision:  81.67%; recall:  88.37%; FB1:  84.89  1451
              PER: precision:  94.12%; recall:  95.55%; FB1:  94.83  1870
```

What are the precision, recall and F1 score on the dev data?

- Precision = 89.86%
- Recall = 91.10%
- F1-Score = 90.47

8 References

- https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html
- Xuezhe Ma, & Eduard Hovy. (2016). End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF.

[ ]:
```
!apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-latex-recommended
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

[112]:
```
!jupyter nbconvert --Application.log_level=CRITICAL --to pdf "/content/drive/
↪MyDrive/Colab Notebooks/CSCI_544_HW4.ipynb"
```

[ ]:
```
```