Hee Ji Park

CSCI544 (NLP)

11.12.2021

Report – HW4 (CSCI544)

Python version	Google Colab / local Python (3.6.12)
Jupyter notebook version	Google Colab / local Jupyter notebook (6.1.4)
Package and libraries	import torch import torch.nn as nn import torch.nn.functional as F import torch.optim as optim import time import random import pandas as pd import numpy as np import string from torch.utils.data import TensorDataset, DataLoader import gzip import os import shutil import pickle
Attached Files	**For Task1 – Simple Bidirectional LSTM model** 1. ./data/vocab_dictionary.pickle 2. ./data/ner_dictionary.pickle 3. ./data/int_vocab_dictionary.pickle 4. ./data/int_ner_dictionary.pickle 5. ./data/loader_train.pickle 6. ./data/loader_dev.pickle 7. ./result/blstm1.pt 8. ./result/dev1.out 9. ./result/test1.out 10. ./result/dev1_for_perl.out : This code is for perl script 11. HeeJiPark_HW4_Task1.py 12. HeeJiPark_HW4_Task1.ipynb 13. HeeJiPark-HW4_Task1_cmd.py
	For Task2 – Using Glove word embedding 1. ./data/vocab_dictionary.pickle 2. ./data/ner_dictionary.pickle 3. ./data/int_vocab_dictionary.pickle 4. ./data/int_ner_dictionary.pickle 5. ./data/loader_train.pickle 6. ./data/loader_dev.pickle

	7. ./data/embedding_vector.pickle 8. ./result/blstm2.pt 9. ./result/dev2.out 10. ./result/test2.out 11. ./result/dev2_for_perl.out : This code is for perl script 12. HeeJiPark_HW4_Task2.py 13. HeeJiPark_HW4_Task2.ipynb 14. HeeJiPark-HW4_Task2_cmd.py
How to run?	- First of all, **you need ‘data’ folder** which contains data files (train, dev, test) and also **need ‘result’ folder** which will contains the result files (dev1.out, dev2.out, etc.) - In case of ’.ipynb’ files (HeeJiPark-HW4_Task1.ipynb & HeeJiPark-HW4_Task2.ipynb), **you can run my code using Jupyter notebook**. (* I wrote these codes using Google Colab.) - In case of ’.py’ files (HeeJiPark-HW4_Task1.py and HeeJiPark-HW4_Task2.py), you can run my code using terminal **- How to run** 1. Command line in the Terminal: **python HeeJiPark_HW4_Task1_cmd.py** => You can get [dev1.out/dev1_for_perl.out] files automatically by running this code. 2. Command line in the Terminal: **python HeeJiPark_HW4_Task2_cmd.py** => You can get [dev2.out/dev2_for_perl.out] files automatically by running this code.
Explanation	- I have subdivided the <unk> token. The <unk> token is subdivided according to whether the unknown word contains a number or has the characteristics of a digit/half_digit/noun/verb/adjective/adverb/contain_digit.
Explorations	**Below is a list of things I've tried to change hyperparameter and model.** 1. I have variously changed batch size to 10, 14, 16, 32, 64. 1. I tried to change the learning rate from 0.05 to 0.5. 2. I tried to use learning rate scheduling, but the performance decreased. (ReduceLROnPlateau, StepLR, CosineAnnealingWarmRestarts) 3. I tried to use various momentum size. 0.6-0.9 4. I tried to set weights for each class, but this did not work. 5. I tried to change the Loss function like nn.CrossEntropyLoss and nn.NLLLoss. 6. I tried to change the dropout position variously when creating the model.

Task1. Simple Bidirectional LSTM mode

Change point	**Accuracy**	**F1**
optim. SGD(model. parameters(), lr= 0.1, momentum= 0.9, nesterov= True) + nn. CrossEntropyLoss(ignore_index= -100) + min_count=2 + batch size=10 + epoch=20	**96.07%**	**78.05%**

Task2. Using GloVe word embedding

Change hyperparameter	Accuracy	F1
optim. SGD(model. parameters(), lr= 0.23, momentum= 0.9, nesterov= True) + nn. CrossEntropyLoss(ignore_index= -100) + min_count=2 + batch size=10 + epoch=60	97.68%	87.20%

Hee Ji Park (4090715830) - CSCI HW4 - Task1

Task1 - Simple Bidirectional LSTM model

Libraries

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchtext import data
from torchtext import datasets
import time
import random
import pandas as pd
import numpy as np
import string
from torch.utils.data import TensorDataset, DataLoader
import pickle
```

Preprocessing for unknown words

In [2]:
```
# If the word is number, return True. Or return False
def isNumber(s):
    try:
        if ',' in s: # ex) 4,800 -> 4800
            s = s.replace(',','')
        float(s)
        return True
    except ValueError:
        return False
```

In [3]:
```
# This code is to classify unknown words
punct = set(string.punctuation)
noun_suffix = ["let",'ie',"kin","action", "ling", "hood", "ship", "ary","age",
               "ery", "ory", "ance", "an","ary","eer","er","ier","herd","cy", "dom",
               "ee", "ence", "ster", "yer", "ant","ar", "ion", "ism", "ist", "ity",
               "ment", "ness", "or", "ry", "scape", "ty"]
```

```
verb_suffix = ["ate", "ify", "ize", "ise"]
adj_suffix = ["able", "ible", 'ant', 'ent', 'ive', "al","ial","an","ian","ish",
              "ern", "ese", "ful", 'ar', 'ary','ly','less','ic','ive','ous', "i", "ic"]
adv_suffix = ["ly","lng","ward", "wards", "way", "ways", "wise"]

def unk_preprocessing(s):
    # If unknown word has number, return <unk_num> token
    num = 0
    for char in s:
        if char.isdigit():
            num += 1

    digitFraction = num / float(len(s))

    if s.isdigit(): #Is a digit
        return "<unk_num>"
    elif digitFraction > 0.5:
        return "<unk_mainly_num>"
    # If unknown word contains characteristics of verb, return <unk_verb> token
    elif any(s.endswith(suffix) for suffix in verb_suffix):
        return "<unk_verb>"
    # If unknown word contains characteristics of adj, return <unk_adj> token
    elif any(s.endswith(suffix) for suffix in adj_suffix):
        return "<unk_adj>"
    # If unknown word contains characteristics of adverbs, return <unk_adv> token
    elif any(s.endswith(suffix) for suffix in adv_suffix):
        return "<unk_adv>"
    elif s.islower(): #All lower case
        return "<unk_all_lower>"
    elif s.isupper(): #All upper case
        return "<unk_all_upper>"
    elif s[0].isupper(): #is a title, initial char upper, then all lower
        return "<unk_initial_upper>"
    elif any(char.isdigit() for char in s):
        return "<unk_contain_num>"
    else:
        return "<unk>"
```

Make a vocabulary and datasets

In [4]:
```
# Make a vocabulary for input data
def make_sequence(file, min_count=2):
    vocab = {}
    ner_set = set()
```

```
    sentence = []
    sentences = []
    with open(file, "r") as train:
        for line in train:
            if not line.split(): # Ignore a blank line
                sentences.append(sentence)
                sentence =[]
                continue
            word_type, NER_type = line.split(" ")[1], line.split(" ")[2].strip('\n')
            if word_type not in vocab:
                vocab[word_type] = 1
            else:
                vocab[word_type]+=1
            sentence.append([word_type,NER_type])
            ner_set.add(NER_type)
        sentences.append(sentence)

        # make <unk> token
        vocab['<unk>'], vocab['<unk_mainly_num>'] = 0,0
        vocab['<unk_num>'], vocab['<unk_contain_num>'] = 0,0
        vocab['<unk_verb>'], vocab['<unk_adj>'] = 0,0
        vocab['<unk_adv>'], vocab['<unk_all_lower>'] = 0,0
        vocab['<unk_all_upper>'], vocab['<unk_initial_upper>'] = 0,0


        delete = []
        for word, occurrences in vocab.items():
            if occurrences >= min_count:
                continue
            else:
                new_token = unk_preprocessing(word)
                vocab[new_token] += occurrences   # If occurrences is lower than 3 : change word name to < unk >
                delete.append(word) # To remove the word in the dictionary (vocab), store 'word' in the delete list

        for i in delete:
            del vocab[i] # Remove the word in the vocab dictionary

    return vocab, ner_set, sentences
```

In [5]:

```
vocab, ner_set, sentences = make_sequence('./data/train')
vocab_sorted = sorted(vocab.items(), key=lambda x:x[1], reverse=True)
```

In [6]:

```
# Make a dictionary
word_to_index = {w: i+1 for i, (w, n) in enumerate(vocab_sorted)}
word_to_index['PAD'] = 0 # This is for padding words
```

In [7]:

```
# Make NER to dictionary. This is for changing the NER tags to number
ner_to_index = {}
i = 0
for ner in ner_set:
    ner_to_index[ner] = i
    i += 1
print(ner_to_index)
```

```
{'I-LOC': 0, 'I-MISC': 1, 'B-LOC': 2, 'I-ORG': 3, 'B-ORG': 4, 'B-PER': 5, 'O': 6, 'I-PER': 7, 'B-MISC': 8}
```

In [8]:

```
# Dictionary: Index to word
index_to_word = {}
for key, value in word_to_index.items():
    index_to_word[value] = key
```

In [9]:

```
# Change index to NER
index_to_ner = {}
for key, value in ner_to_index.items():
    index_to_ner[value] = key
```

In [10]:

```
# This code is for input sequence
data_X = []

for s in sentences:
    temp_X = []
    for w, label in s:
        if w in word_to_index:
            temp_X.append(word_to_index.get(w))
        else:
            unk = unk_preprocessing(w)
            temp_X.append(word_to_index[unk])
    data_X.append(temp_X)
```

In [11]:

```
# This code is for target sequence
data_y = []
for s in sentences:
    temp_y = []
    for w, label in s:
        temp_y.append(ner_to_index.get(label))
    data_y.append(temp_y)
```

In [12]:

```
# Limit the maximum review length to 130
def pad_features_for_word(x, desired_len):
    for i, row in enumerate(x):
```

```
        if len(row) > desired_len: # Turncate longer sentences
            x[i] = row[:desired_len]
        elif len(row) < desired_len: # Padding shorter sentencess with a '0'
            x[i] = row[:len(row)] + [0]*(desired_len-len(row))

    return x
```

In [13]:
```
# Limit the maximum review length to 130
def pad_features_for_NER(x, desired_len):
    for i, row in enumerate(x):
        if len(row) > desired_len: # Turncate longer sentences
            x[i] = row[:desired_len]
        elif len(row) < desired_len: # Padding shorter sentencess with a '-100'
            x[i] = row[:len(row)] + [-100]*(desired_len-len(row))

    return x
```

In [14]:
```
# Make a dataset and dataloader
data_X = pad_features_for_word(data_X, 130)
data_y = pad_features_for_NER(data_y, 130)

X_train = torch.LongTensor(data_X)
Y_train = torch.LongTensor(data_y)

ds_train = TensorDataset(X_train, Y_train)
loader_train = DataLoader(ds_train, batch_size=10, shuffle=False)
```

Set GPU or CPU

In [15]:
```
# If a GPU is available, return True. Else it'll return False
is_cuda = torch.cuda.is_available()

# Set CPU or GPU
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("GPU not available, CPU used")
```

```
GPU is available
```

BLSTM Model

In [16]:
```
class BLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, first_output_dim, output_dim, num_layers, bidirectional, drop
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.blstm = nn.LSTM(embedding_dim, hidden_dim, num_layers = num_layers, bidirectional = bidirectional, batch_first
        self.fc1 = nn.Linear(hidden_dim * 2, first_output_dim)
        self.dropout = nn.Dropout(drop_out)
        self.activation = nn.ELU()
        self.fc2 = nn.Linear(first_output_dim, output_dim)

    def forward(self, text):
        # text = [batch size, sentence length]
        embedded = self.dropout(self.embedding(text)) # embedded = [batch size, sentence length, embedding dim]
        outputs, (hidden, cell) = self.blstm(embedded) # output = [batch size, sentence length , hidden dim * n_layers dire
        outputs = self.dropout(outputs)
        outputs = self.activation(self.fc1(outputs))
        predictions = self.fc2(outputs) # predictions = [batch size, sentence length, output dim]
        return predictions
```

In [17]:
```
# Model BLSTM
INPUT_DIM = len(word_to_index)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
FIRST_OUTPUT_DIM = 128
OUTPUT_DIM = len(ner_to_index)
N_LAYERS = 1
BIDIRECTIONAL = True
DROPOUT = 0.33

model = BLSTM(INPUT_DIM,
              EMBEDDING_DIM,
              HIDDEN_DIM,
              FIRST_OUTPUT_DIM,
              OUTPUT_DIM,
              N_LAYERS,
              BIDIRECTIONAL,
              DROPOUT)

model.to(device)
```

Out[17]:
```
BLSTM(
  (embedding): Embedding(11994, 100, padding_idx=0)
  (blstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (fc1): Linear(in_features=512, out_features=128, bias=True)
```

```
  (dropout): Dropout(p=0.33, inplace=False)
  (activation): ELU(alpha=1.0)
  (fc2): Linear(in_features=128, out_features=9, bias=True)
)
```

Train and Test function

In [18]:

```python
def model_train(model, iterator, predict_table):

    epoch_loss = 0
    epoch_acc = 0
    epoch_tot = 0
    model.train()

    for text, tags in iterator:

        optimizer.zero_grad()
        tags = tags.to(device)
        text = text.to(device)
        predictions = model(text)
        predictions = predictions.view(-1, predictions.shape[-1]) # #predictions = [batch size * sentence length, output d
        tags = tags.view(-1) # tags = [batch_size * sentence length]

        loss = criterion(predictions, tags)

        tot, correct, predict_table = categorical_accuracy(predictions, tags, tag_pad_idx, text.view(-1), predict_table)

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += correct
        epoch_tot +=tot

    return epoch_loss / len(iterator), epoch_acc / epoch_tot, predict_table
```

In [19]:

```python
def model_evaluate(model, iterator, predict_table):

    epoch_loss = 0
    epoch_acc = 0
    epoch_tot = 0
    model.eval()

    with torch.no_grad():
```

```
        for text, tags in iterator:
            tags = tags.to(device)
            text = text.to(device)
            predictions = model(text)

            predictions = predictions.view(-1, predictions.shape[-1])
            tags = tags.view(-1)

            loss = criterion(predictions, tags)

            tot, correct, predict_table = categorical_accuracy(predictions, tags, tag_pad_idx, text.view(-1), predict_table

            epoch_loss += loss.item()
            epoch_acc += correct
            epoch_tot +=tot

    return epoch_loss / len(iterator), epoch_acc / epoch_tot, predict_table
```

In [20]:

```
def categorical_accuracy(preds, y, tag_pad_idx, text, predict_table):
    tot = 0
    correct = 0
    max_preds = preds.argmax(dim = 1, keepdim = True) # Get the index of the max probability
    for predict, real, word in zip(max_preds, y, text):
        if real.item() == tag_pad_idx: # ignore padding index
            continue
        else:
            predict_table.append((word.item(), predict.item(), real.item()))
            if real.item() == predict.item():
                correct += 1
            tot += 1
    return tot, correct, predict_table
```

In [21]:

```
# This code is for dev dataset
dev_sentences = []
sentence=[]
cnt=0
with open('./data/dev', "r") as dev:
    for line in dev:
        if not line.split(): # Ignore a blank line
            dev_sentences.append(sentence)
            sentence =[]
            continue
        word_type, NER_type = line.split(" ")[1], line.split(" ")[2].strip('₩n')
        cnt+=1
```

```
        sentence.append([word_type,NER_type])
    dev_sentences.append(sentence)
```

In [22]:

```
# Make dev dataset
dev_X = []

for s in dev_sentences:
    temp_X = []
    for w, label in s:
        if w in word_to_index:
            temp_X.append(word_to_index.get(w))
        else:
            unk = unk_preprocessing(w)
            temp_X.append(word_to_index[unk])
    dev_X.append(temp_X)

dev_y = []
for s in dev_sentences:
    temp_y = []
    for w, label in s:
        temp_y.append(ner_to_index.get(label))
    dev_y.append(temp_y)

dev_X = pad_features_for_word(dev_X, 130)
dev_y = pad_features_for_NER(dev_y, 130)
X_dev = torch.LongTensor(dev_X)
Y_dev = torch.LongTensor(dev_y)

# Make a dataset and dataloader
ds_dev = TensorDataset(X_dev, Y_dev)
loader_dev = DataLoader(ds_dev, batch_size=10, shuffle=False)
```

In [23]:

```
import pickle
# save data
with open('./data/vocab_dictionary.pickle','wb') as fw1:
    pickle.dump(word_to_index, fw1)
with open('./data/ner_dictionary.pickle','wb') as fw2:
    pickle.dump(ner_to_index, fw2)
with open('./data/int_vocab_dictionary.pickle','wb') as fw3:
    pickle.dump(index_to_word, fw3)
with open('./data/int_ner_dictionary.pickle','wb') as fw4:
    pickle.dump(index_to_ner, fw4)
with open('./data/loader_train.pickle','wb') as fw5:
    pickle.dump(loader_train, fw5)
with open('./data/loader_dev.pickle','wb') as fw6:
```

```
    pickle.dump(loader_dev, fw6)
```

Train and evaluation

In [24]:

```
# epoch - Train and evaluation
N_EPOCHS = 20
tag_pad_idx=-100
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9, nesterov=True) # Set hyperparameter
criterion = nn.CrossEntropyLoss(ignore_index= -100)
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    train_predict_table = []
    test_predict_table = []

    train_loss, train_acc, train_predict_table = model_train(model, loader_train, train_predict_table)
    valid_loss, valid_acc, valid_predict_table = model_evaluate(model, loader_dev, test_predict_table)

    if valid_loss <= best_valid_loss:
        best_valid_loss = valid_loss
        best_predict_table = valid_predict_table
        torch.save(model.state_dict(), './result/blstm1.pt')

    print(f'Epoch: {epoch+1:02}')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01
	Train Loss: 0.652 | Train Acc: 84.97%
	 Val. Loss: 0.438 |  Val. Acc: 88.31%
Epoch: 02
	Train Loss: 0.446 | Train Acc: 87.96%
	 Val. Loss: 0.303 |  Val. Acc: 91.29%
Epoch: 03
	Train Loss: 0.350 | Train Acc: 89.85%
	 Val. Loss: 0.248 |  Val. Acc: 92.62%
Epoch: 04
	Train Loss: 0.298 | Train Acc: 90.98%
	 Val. Loss: 0.210 |  Val. Acc: 93.74%
Epoch: 05
	Train Loss: 0.262 | Train Acc: 91.70%
	 Val. Loss: 0.188 |  Val. Acc: 94.31%
Epoch: 06
	Train Loss: 0.236 | Train Acc: 92.37%
	 Val. Loss: 0.174 |  Val. Acc: 94.64%
```

```
Epoch: 07
	Train Loss: 0.214 | Train Acc: 93.00%
	 Val. Loss: 0.167 |  Val. Acc: 94.91%
Epoch: 08
	Train Loss: 0.203 | Train Acc: 93.26%
	 Val. Loss: 0.155 |  Val. Acc: 95.23%
Epoch: 09
	Train Loss: 0.188 | Train Acc: 93.66%
	 Val. Loss: 0.154 |  Val. Acc: 95.34%
Epoch: 10
	Train Loss: 0.178 | Train Acc: 93.99%
	 Val. Loss: 0.145 |  Val. Acc: 95.60%
Epoch: 11
	Train Loss: 0.171 | Train Acc: 94.14%
	 Val. Loss: 0.147 |  Val. Acc: 95.60%
Epoch: 12
	Train Loss: 0.161 | Train Acc: 94.43%
	 Val. Loss: 0.140 |  Val. Acc: 95.80%
Epoch: 13
	Train Loss: 0.154 | Train Acc: 94.59%
	 Val. Loss: 0.139 |  Val. Acc: 95.79%
Epoch: 14
	Train Loss: 0.147 | Train Acc: 94.84%
	 Val. Loss: 0.136 |  Val. Acc: 95.87%
Epoch: 15
	Train Loss: 0.141 | Train Acc: 95.01%
	 Val. Loss: 0.139 |  Val. Acc: 95.76%
Epoch: 16
	Train Loss: 0.138 | Train Acc: 95.12%
	 Val. Loss: 0.133 |  Val. Acc: 96.06%
Epoch: 17
	Train Loss: 0.133 | Train Acc: 95.28%
	 Val. Loss: 0.129 |  Val. Acc: 96.09%
Epoch: 18
	Train Loss: 0.127 | Train Acc: 95.40%
	 Val. Loss: 0.128 |  Val. Acc: 96.07%
Epoch: 19
	Train Loss: 0.127 | Train Acc: 95.47%
	 Val. Loss: 0.130 |  Val. Acc: 96.02%
Epoch: 20
	Train Loss: 0.120 | Train Acc: 95.67%
	 Val. Loss: 0.129 |  Val. Acc: 96.17%
```

Dev

In [25]:

```
# Save the result as a '.out' file
term = [int(x[0]) for x in best_predict_table]
```

```
y_pred = [int(x[1]) for x in best_predict_table]
i=0
newfile = open('./result/dev1.out', "w")
with open('./data/dev', "r") as train:
    for line in train:
        if not line.split(): # Ignore a blank line
            newfile.write('₩n')
            continue
        index, word_type = line.split(" ")[0], line.split(" ")[1].strip('₩n')
        newfile.write(str(index)+' '+str(word_type)+' '+str(index_to_ner[y_pred[i]])+'₩n')
        i += 1
newfile.close()

i=0
newfile = open('./result/dev1_for_perl.out', "w")
with open('./data/dev', "r") as train:
    for line in train:
        if not line.split(): # Ignore a blank line
            newfile.write('₩n')
            continue
        index, word_type, NER_type = line.split(" ")[0], line.split(" ")[1], line.split(" ")[2].strip('₩n')
        newfile.write(str(index)+' '+str(word_type)+' '+str(NER_type)+' '+str(index_to_ner[y_pred[i]])+'₩n')
        i += 1
newfile.close()
```

In [26]:

```
def categorical_evaluate(preds, text, predict_table):

    max_preds = preds.argmax(dim = 1, keepdim = True) # get the index of the max probability
    for predict, word in zip(max_preds, text):
        if word == 0:
            continue
        else:
            predict_table.append((word, predict[0]))

    return predict_table
```

In [27]:

```
def model_evaluate(model, iterator, predict_table):

    epoch_loss = 0
    epoch_acc = 0
    epoch_tot = 0
    model.eval()

    with torch.no_grad():
```

```
        for text in iterator:
            text = text.to(device)
            predictions = model(text)
            predictions = predictions.view(-1, predictions.shape[-1])

            predict_table = categorical_evaluate(predictions, text.view(-1), predict_table)

    return predict_table
```

Test

In [28]:

```
# Predict test set and Save the result as a '.out' file
test_X = []
sentence = []
cnt=0
with open('./data/test', "r") as test:
    for line in test:
        if not line.split(): # Ignore a blank line
            test_X.append(sentence)
            sentence = []
            continue
        word_type = line.split(" ")[1]
        if word_type in word_to_index:
            sentence.append(word_to_index.get(word_type))
        else:
            unk = unk_preprocessing(word_type) # if the word is not in vocab dictionary, change the word to unknown token
            sentence.append(word_to_index.get(unk))
    test_X.append(sentence)

test_X = pad_features_for_word(test_X, 130) # Padding
X_test = torch.LongTensor(test_X)
loader_test = DataLoader(X_test, batch_size=10, shuffle=False)

evaluate_predict_table2 = []
model = BLSTM(INPUT_DIM,
              EMBEDDING_DIM,
              HIDDEN_DIM,
              FIRST_OUTPUT_DIM,
              OUTPUT_DIM,
              N_LAYERS,
              BIDIRECTIONAL,
              DROPOUT)
model.to(device)
model.load_state_dict(torch.load('./result/blstm1.pt')) # load pretrained model
```

```
prediction_table = model_evaluate(model, loader_test, evaluate_predict_table2)

term = [int(x[0]) for x in evaluate_predict_table2]
y_pred = [int(x[1]) for x in evaluate_predict_table2]

# Make test2.out file
i=0
newfile = open('./result/test1.out', "w")
with open('./data/test', "r") as test:
    for line in test:
        if not line.split(): # Ignore a blank line
            newfile.write('\n')
            continue
        index, word_type = line.split(" ")[0], line.split(" ")[1].strip('\n')
        for_tag = index_to_ner[y_pred[i]]
        newfile.write(str(index)+' '+str(word_type)+' '+for_tag+'\n')
        i += 1
newfile.close()
```

In [29]:

```
import pickle
# save data
with open('./data/vocab_dictionary.pickle','wb') as fw1:
    pickle.dump(word_to_index, fw1)
with open('./data/ner_dictionary.pickle','wb') as fw2:
    pickle.dump(ner_to_index, fw2)
with open('./data/int_vocab_dictionary.pickle','wb') as fw3:
    pickle.dump(index_to_word, fw3)
with open('./data/int_ner_dictionary.pickle','wb') as fw4:
    pickle.dump(index_to_ner, fw4)
with open('./data/loader_train.pickle','wb') as fw5:
    pickle.dump(loader_train, fw5)
with open('./data/loader_dev.pickle','wb') as fw6:
    pickle.dump(loader_dev, fw6)
with open('./data/loader_test.pickle','wb') as fw7:
    pickle.dump(loader_test, fw7)
```

In [30]:

```
checkpoint = {'INPUT_DIM':len(word_to_index),
              'EMBEDDING_DIM':100,
              'HIDDEN_DIM':256,
              'FIRST_OUTPUT_DIM':128,
              'OUTPUT_DIM':len(ner_to_index),
              'N_LAYERS':1,
              'BIDIRECTIONAL':True,
              'DROPOUT':0.33,
              'state_dict': model.state_dict()}
```

```python
torch.save(checkpoint, 'result/checkpoint.pth')
```

In [ ]:

```python

```

Hee Ji Park (4090715830) - CSCI 544 HW4

Task2 : Using GloVe word embeddings

In [1]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import time
import random
import pandas as pd
import numpy as np
import string
from torch.utils.data import TensorDataset, DataLoader
import gzip
import os
import shutil
```

In [2]:

```
# If the word is number, return True. Or return False
def isNumber(s):
    try:
        if ',' in s: # ex) 4,800 -> 4800
            s = s.replace(',','')
        float(s)
        return True
    except ValueError:
        return False


punct = set(string.punctuation)
noun_suffix = ["let",'ie',"kin","action", "ling", "hood", "ship", "ary","age",
               "ery", "ory", "ance", "an","ary","eer","er","ier","herd","cy", "dom",
               "ee", "ence", "ster", "yer", "ant","ar", "ion", "ism", "ist", "ity",
               "ment", "ness", "or", "ry", "scape", "ty"]
verb_suffix = ["ate", "ify", "ize", "ise"]
adj_suffix = ["able", "ible", 'ive', "ish", "ful", 'ar', 'ary','ly','less','ic','ive','ous', "ic"]
adv_suffix = ["ly","lng","ward", "wards", "way", "ways", "wise"]

def unk_preprocessing(s):
    # If unknown word has number, use this preprocessing
    num = 0
```

```
    for char in s:
      if char.isdigit():
         num += 1

    digitFraction = num / float(len(s))

    if s.isdigit(): #Is a digit
        return "<unk_num>"
    elif digitFraction > 0.5:
        return "<unk_mainly_num>"
    # If unknown word contains characteristics of verb, return <unk_verb> token
    elif any(s.endswith(suffix) for suffix in verb_suffix):
        return "<unk_verb>"
    # If unknown word contains characteristics of adj, return <unk_adj> token
    elif any(s.endswith(suffix) for suffix in adj_suffix):
        return "<unk_adj>"
    # If unknown word contains characteristics of adverbs, return <unk_adv> token
    elif any(s.endswith(suffix) for suffix in adv_suffix):
        return "<unk_adv>"
    elif s.islower(): # All lower case
        return "<unk_all_lower>"
    elif s.isupper(): # All upper case
        return "<unk_all_upper>"
    elif s[0].isupper(): # If the first charter is upper case and then all lower
        return "<unk_initial_upper>"
    elif any(char.isdigit() for char in s): # if the word contains some number
        return "<unk_contain_num>"
    else:
        return "<unk>"

def make_sequence(file, min_count=2):
    vocab = {}
    ner_set = set()
    sentence = []
    sentences = []
    with open(file, "r") as train:
        for line in train:
            if not line.split(): # Ignore a blank line
                sentences.append(sentence)
                sentence =[]
                continue
            word_type, NER_type = line.split(" ")[1], line.split(" ")[2].strip('\n')
            if word_type not in vocab:
                vocab[word_type] = 1
            else:
                vocab[word_type]+=1
```

```python
                sentence.append([word_type,NER_type])
                ner_set.add(NER_type)
            sentences.append(sentence)

        # make <unk> token
        vocab['<unk>'], vocab['<unk_mainly_num>'] = 0,0
        vocab['<unk_num>'], vocab['<unk_contain_num>'] = 0,0
        vocab['<unk_verb>'], vocab['<unk_adj>'] = 0,0
        vocab['<unk_adv>'], vocab['<unk_all_lower>'] = 0,0
        vocab['<unk_all_upper>'], vocab['<unk_initial_upper>'] = 0,0

        delete = []
        for word, occurrences in vocab.items():
            if occurrences >= min_count:
                continue
            else:
                new_token = unk_preprocessing(word)
                vocab[new_token] += occurrences   # If occurrences is lower than 2 : change word name to < unk >
                delete.append(word) # To remove the word in the dictionary (vocab), store 'word' in the delete list

        for i in delete:
            del vocab[i] # Remove the word in the vocab dictionary

    return vocab, ner_set, sentences
```

In [3]:
```python
vocab, ner_set, sentences = make_sequence('./data/train')
vocab_sorted = sorted(vocab.items(), key=lambda x:x[1], reverse=True)
word_to_index = {w: i+1 for i, (w, n) in enumerate(vocab_sorted)}
word_to_index['PAD'] = 0 # For Padding index
```

In [4]:
```python
ner_to_index = {}
#ner_to_index['PAD'] = -100 # set padding = -100
i = 0
for ner in ner_set:
    ner_to_index[ner] = i
    i += 1
```

In [5]:
```python
# In order to change index to word
index_to_word = {}
for key, value in word_to_index.items():
    index_to_word[value] = key
```

In [6]:
```python
# In order to change index to NER
index_to_ner = {}
```

```
for key, value in ner_to_index.items():
    index_to_ner[value] = key
```

In [7]:

```
# Make train input data
data_X = []

for s in sentences:
    temp_X = []
    for w, label in s:
        if w in word_to_index:
            temp_X.append(word_to_index.get(w))
        else:
            unk = unk_preprocessing(w)
            temp_X.append(word_to_index[unk])
    data_X.append(temp_X)
```

In [8]:

```
# Make train target data
data_y = []
for s in sentences:
    temp_y = []
    for w, label in s:
        temp_y.append(ner_to_index.get(label))
    data_y.append(temp_y)
```

In [9]:

```
# Limit the maximum review length to 130
def pad_features_for_word(x, desired_len):
    for i, row in enumerate(x):
        if len(row) > desired_len: # Turncate longer reviews
            x[i] = row[:desired_len]
        elif len(row) < desired_len: # Padding shorter reviews with a '0'
            x[i] = row[:len(row)] + [0]*(desired_len-len(row))

    return x
# Limit the maximum review length to 130
def pad_features_for_NER(x, desired_len):
    for i, row in enumerate(x):
        if len(row) > desired_len: # Turncate longer reviews
            x[i] = row[:desired_len]
        elif len(row) < desired_len: # Padding shorter reviews with a '0'
            x[i] = row[:len(row)] + [-100]*(desired_len-len(row))

    return x

# Padding and make dataset and dataloader
```

```python
data_X = pad_features_for_word(data_X, 130)
data_y = pad_features_for_NER(data_y, 130)
X_train = torch.LongTensor(data_X)
Y_train = torch.LongTensor(data_y)
ds_train = TensorDataset(X_train, Y_train)
loader_train = DataLoader(ds_train, batch_size=16, shuffle=False)
```

In [10]:
```python
# For GloVe word embedding
with gzip.open('glove.6B.100d.gz', 'rb') as f_in:
    with open('glove.6B.100d', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)

embedding_dict = dict()
f = open(os.path.join('glove.6B.100d'), encoding='utf-8')
for line in f:
    word_vector = line.split()
    word = word_vector[0]
    word_vector_arr = np.asarray(word_vector[1:], dtype='float32')
    embedding_dict[word] = word_vector_arr
f.close()

# Make word embedding matrix
embedding_dim = 100
embedding_matrix = np.zeros((len(word_to_index), embedding_dim))

for word, i in word_to_index.items():
    embedding_vector = embedding_dict.get(word.lower())
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

embedding_matrix = torch.LongTensor(embedding_matrix)
```

In [11]:
```python
# If a GPU is available, return True, else it'll return False
is_cuda = torch.cuda.is_available()
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("GPU not available, CPU used")
```

```
GPU is available
```

In [12]:
```python
class BLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, first_output_dim, output_dim, num_layers, bidirectional, drop
```

```python
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.blstm = nn.LSTM(embedding_dim, hidden_dim, num_layers = num_layers, bidirectional = bidirectional, batch_first
        self.fc1 = nn.Linear(hidden_dim * 2, first_output_dim)
        self.dropout = nn.Dropout(drop_out)
        self.activation = nn.ELU()
        self.fc2 = nn.Linear(first_output_dim, output_dim)

    def forward(self, text):
        # text = [sent len, batch size]
        embedded = self.dropout(self.embedding(text)) # embedded = [batch size,sent len, emb dim]
        outputs, (hidden, cell) = self.blstm(embedded) # output = [batch size, sent len , hid dim * n directions]
        outputs = self.dropout(outputs)
        outputs = self.activation(self.fc1(outputs))
        predictions = self.fc2(outputs) # predictions = [batch size, sent len, output dim]
        return predictions
```

In [13]:

```python
INPUT_DIM = len(word_to_index)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
FIRST_OUTPUT_DIM = 128
OUTPUT_DIM = len(ner_to_index)
N_LAYERS = 1
BIDIRECTIONAL = True
DROPOUT = 0.33

model = BLSTM(INPUT_DIM,
              EMBEDDING_DIM,
              HIDDEN_DIM,
              FIRST_OUTPUT_DIM,
              OUTPUT_DIM,
              N_LAYERS,
              BIDIRECTIONAL,
              DROPOUT)

model.to(device)
model.embedding.weight.data.copy_(embedding_matrix)
```

Out[13]:

```
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]], device='cuda:0')
```

In [14]:

```python
def model_train(model, iterator, predict_table):
    epoch_loss = 0
    epoch_acc = 0
    epoch_tot = 0
    model.train()

    for text, tags in iterator:

        optimizer.zero_grad()
        tags = tags.to(device)
        text = text.to(device)
        predictions = model(text)
        predictions = predictions.view(-1, predictions.shape[-1]) # #predictions = [sentence_len * batch size, output dim]
        tags = tags.view(-1) # tags = [sentence_len * batch_size]
        loss = criterion(predictions, tags)
        tot, correct, predict_table = categorical_accuracy(predictions, tags, tag_pad_idx, text.view(-1), predict_table)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += correct
        epoch_tot +=tot

    return epoch_loss / len(iterator), epoch_acc / epoch_tot, predict_table
```

In [15]:

```python
def categorical_accuracy(preds, y, tag_pad_idx, text, predict_table):
    tot = 0
    correct = 0
    max_preds = preds.argmax(dim = 1, keepdim = True) # get the index of the max probability
    for predict, real, word in zip(max_preds, y, text):
        if real.item() == tag_pad_idx: # ignore padding
            continue
        else:
            predict_table.append((word.item(), predict.item(), real.item()))
            if real.item() == predict.item():
                correct += 1
            tot += 1
    return tot, correct, predict_table

def model_evaluate(model, iterator, predict_table):
    epoch_loss = 0
    epoch_acc = 0
    epoch_tot = 0
    model.eval()
```

```
    with torch.no_grad():

        for text, tags in iterator:
            tags = tags.to(device)
            text = text.to(device)

            predictions = model(text)

            predictions = predictions.view(-1, predictions.shape[-1])
            tags = tags.view(-1)

            loss = criterion(predictions, tags)

            tot, correct, predict_table = categorical_accuracy(predictions, tags, tag_pad_idx, text.view(-1), predict_table

            epoch_loss += loss.item()
            epoch_acc += correct
            epoch_tot +=tot

    return epoch_loss / len(iterator), epoch_acc / epoch_tot, predict_table
```

In [16]:

```
# For predicting dev file, make a sequence
dev_sentences = []
sentence=[]
cnt=0
with open('./data/dev', "r") as dev:
    for line in dev:
        if not line.split(): # Ignore a blank line
            dev_sentences.append(sentence)
            sentence =[]
            continue
        word_type, NER_type = line.split(" ")[1], line.split(" ")[2].strip('₩n')
        cnt+=1
        sentence.append([word_type,NER_type])
    dev_sentences.append(sentence)

dev_X = []
for s in dev_sentences:
    temp_X = []
    for w, label in s:
        if w in word_to_index:
            temp_X.append(word_to_index.get(w))
        else:
            unk = unk_preprocessing(w) # if the word is not in vocab dictionary, change the word to unknown token
            temp_X.append(word_to_index[unk])
```

```
    dev_X.append(temp_X)

dev_y = []
for s in dev_sentences:
    temp_y = []
    for w, label in s:
        temp_y.append(ner_to_index.get(label))
    dev_y.append(temp_y)

dev_X = pad_features_for_word(dev_X, 130) # Padding
dev_y = pad_features_for_NER(dev_y, 130)

X_dev = torch.LongTensor(dev_X)
Y_dev = torch.LongTensor(dev_y)

# Make a dataset and dataloader
ds_dev = TensorDataset(X_dev, Y_dev)
loader_dev = DataLoader(ds_dev, batch_size=16, shuffle=False)
```

In [17]:

```
N_EPOCHS = 60
tag_pad_idx=-100
optimizer = optim.SGD(model.parameters(), lr=0.23, momentum=0.9, nesterov=True)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=4)
criterion = nn.CrossEntropyLoss(ignore_index= -100)
best_valid_loss = float('inf')
for epoch in range(N_EPOCHS):
    train_predict_table = []
    test_predict_table = []

    train_loss, train_acc, train_predict_table = model_train(model, loader_train, train_predict_table)
    valid_loss, valid_acc, valid_predict_table = model_evaluate(model, loader_dev, test_predict_table)

    if valid_loss <= best_valid_loss:
        best_valid_loss = valid_loss
        best_predict_table = valid_predict_table
        torch.save(model.state_dict(), './result/blstm2.pt')

    scheduler.step(valid_loss)

    print(f'Epoch: {epoch+1:02}')
    print(f'₩tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'₩t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01
        Train Loss: 0.437 | Train Acc: 88.53%
         Val. Loss: 0.240 |  Val. Acc: 93.34%
```

```
Epoch: 02
        Train Loss: 0.204 | Train Acc: 93.84%
         Val. Loss: 0.142 |  Val. Acc: 96.02%
Epoch: 03
        Train Loss: 0.132 | Train Acc: 95.80%
         Val. Loss: 0.121 |  Val. Acc: 96.32%
Epoch: 04
        Train Loss: 0.100 | Train Acc: 96.80%
         Val. Loss: 0.109 |  Val. Acc: 96.59%
Epoch: 05
        Train Loss: 0.079 | Train Acc: 97.40%
         Val. Loss: 0.103 |  Val. Acc: 96.71%
Epoch: 06
        Train Loss: 0.066 | Train Acc: 97.78%
         Val. Loss: 0.099 |  Val. Acc: 96.82%
Epoch: 07
        Train Loss: 0.060 | Train Acc: 98.03%
         Val. Loss: 0.093 |  Val. Acc: 97.06%
Epoch: 08
        Train Loss: 0.051 | Train Acc: 98.28%
         Val. Loss: 0.101 |  Val. Acc: 96.80%
Epoch: 09
        Train Loss: 0.046 | Train Acc: 98.48%
         Val. Loss: 0.092 |  Val. Acc: 97.10%
Epoch: 10
        Train Loss: 0.041 | Train Acc: 98.61%
         Val. Loss: 0.092 |  Val. Acc: 97.19%
Epoch: 11
        Train Loss: 0.038 | Train Acc: 98.69%
         Val. Loss: 0.093 |  Val. Acc: 97.23%
Epoch: 12
        Train Loss: 0.034 | Train Acc: 98.86%
         Val. Loss: 0.089 |  Val. Acc: 97.35%
Epoch: 13
        Train Loss: 0.031 | Train Acc: 98.91%
         Val. Loss: 0.094 |  Val. Acc: 97.19%
Epoch: 14
        Train Loss: 0.029 | Train Acc: 99.02%
         Val. Loss: 0.095 |  Val. Acc: 97.31%
Epoch: 15
        Train Loss: 0.027 | Train Acc: 99.03%
         Val. Loss: 0.095 |  Val. Acc: 97.33%
Epoch: 16
        Train Loss: 0.025 | Train Acc: 99.11%
         Val. Loss: 0.096 |  Val. Acc: 97.35%
Epoch: 17
        Train Loss: 0.023 | Train Acc: 99.19%
         Val. Loss: 0.098 |  Val. Acc: 97.39%
Epoch: 18
```

```
        Train Loss: 0.020 | Train Acc: 99.27%
         Val. Loss: 0.090 |  Val. Acc: 97.64%
Epoch: 19
        Train Loss: 0.019 | Train Acc: 99.32%
         Val. Loss: 0.090 |  Val. Acc: 97.66%
Epoch: 20
        Train Loss: 0.018 | Train Acc: 99.35%
         Val. Loss: 0.090 |  Val. Acc: 97.67%
Epoch: 21
        Train Loss: 0.018 | Train Acc: 99.35%
         Val. Loss: 0.089 |  Val. Acc: 97.68%
Epoch: 22
        Train Loss: 0.018 | Train Acc: 99.35%
         Val. Loss: 0.089 |  Val. Acc: 97.68%
Epoch: 23
        Train Loss: 0.018 | Train Acc: 99.36%
         Val. Loss: 0.091 |  Val. Acc: 97.65%
Epoch: 24
        Train Loss: 0.017 | Train Acc: 99.39%
         Val. Loss: 0.091 |  Val. Acc: 97.70%
Epoch: 25
        Train Loss: 0.017 | Train Acc: 99.40%
         Val. Loss: 0.091 |  Val. Acc: 97.69%
Epoch: 26
        Train Loss: 0.017 | Train Acc: 99.39%
         Val. Loss: 0.092 |  Val. Acc: 97.65%
Epoch: 27
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.092 |  Val. Acc: 97.64%
Epoch: 28
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.091 |  Val. Acc: 97.69%
Epoch: 29
        Train Loss: 0.016 | Train Acc: 99.43%
         Val. Loss: 0.091 |  Val. Acc: 97.71%
Epoch: 30
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.72%
Epoch: 31
        Train Loss: 0.016 | Train Acc: 99.44%
         Val. Loss: 0.090 |  Val. Acc: 97.72%
Epoch: 32
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 33
        Train Loss: 0.016 | Train Acc: 99.43%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 34
        Train Loss: 0.015 | Train Acc: 99.44%
```

```
         Val. Loss: 0.090 |  Val. Acc: 97.72%
Epoch: 35
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.72%
Epoch: 36
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.72%
Epoch: 37
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 38
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 39
        Train Loss: 0.015 | Train Acc: 99.43%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 40
        Train Loss: 0.016 | Train Acc: 99.44%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 41
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 42
        Train Loss: 0.016 | Train Acc: 99.44%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 43
        Train Loss: 0.016 | Train Acc: 99.44%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 44
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 45
        Train Loss: 0.016 | Train Acc: 99.41%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 46
        Train Loss: 0.016 | Train Acc: 99.43%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 47
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 48
        Train Loss: 0.015 | Train Acc: 99.45%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 49
        Train Loss: 0.016 | Train Acc: 99.45%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 50
        Train Loss: 0.016 | Train Acc: 99.44%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
```

```
Epoch: 51
        Train Loss: 0.016 | Train Acc: 99.44%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 52
        Train Loss: 0.016 | Train Acc: 99.41%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 53
        Train Loss: 0.016 | Train Acc: 99.43%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 54
        Train Loss: 0.016 | Train Acc: 99.45%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 55
        Train Loss: 0.015 | Train Acc: 99.44%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 56
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 57
        Train Loss: 0.015 | Train Acc: 99.45%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 58
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 59
        Train Loss: 0.016 | Train Acc: 99.42%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
Epoch: 60
        Train Loss: 0.016 | Train Acc: 99.45%
         Val. Loss: 0.090 |  Val. Acc: 97.71%
```

In [18]:

```
# Fungtions for evaluate
def categorical_evaluate(preds, text, predict_table):

    max_preds = preds.argmax(dim = 1, keepdim = True) # Get the index of the max probability
    for predict, word in zip(max_preds, text):
        if word == 0:
            continue
        else:
            predict_table.append((word, predict[0]))

    return predict_table

def model_evaluate(model, iterator, predict_table):

    epoch_loss = 0
    epoch_acc = 0
    epoch_tot = 0
```

```
    model.eval()

    with torch.no_grad():
        for text in iterator:
            text = text.to(device)
            predictions = model(text)
            predictions = predictions.view(-1, predictions.shape[-1])
            predict_table = categorical_evaluate(predictions, text.view(-1), predict_table)

    return predict_table
```

Dev Set

In [19]:

```
# Predict Dev Set and make dev2.out file
term = [int(x[0]) for x in best_predict_table]
y_pred = [int(x[1]) for x in best_predict_table]
i=0
newfile = open('./result/dev2.out', "w")
with open('./data/dev', "r") as train:
    for line in train:
        if not line.split(): # Ignore a blank line
            newfile.write('₩n')
            continue
        index, word_type = line.split(" ")[0], line.split(" ")[1].strip('₩n')
        newfile.write(str(index)+' '+str(word_type)+' '+str(index_to_ner[y_pred[i]])+'₩n')
        i += 1
newfile.close()

i=0
newfile = open('./result/dev2_for_perl.out', "w")
with open('./data/dev', "r") as train:
    for line in train:
        if not line.split(): # Ignore a blank line
            newfile.write('₩n')
            continue
        index, word_type, NER_type = line.split(" ")[0], line.split(" ")[1], line.split(" ")[2].strip('₩n')
        newfile.write(str(index)+' '+str(word_type)+' '+str(NER_type)+' '+str(index_to_ner[y_pred[i]])+'₩n')
        i += 1
newfile.close()
```

Test Set

In [23]:
```
# Predict test set
test_X = []
sentence = []
cnt=0
with open('./data/test', "r") as test:
    for line in test:
        if not line.split(): # Ignore a blank line
            test_X.append(sentence)
            sentence = []
            continue
        word_type = line.split(" ")[1]
        if word_type in word_to_index:
            sentence.append(word_to_index.get(word_type))
        else:
            unk = unk_preprocessing(word_type) # if the word is not in vocab dictionary, change the word to unknown token
            sentence.append(word_to_index.get(unk))
    test_X.append(sentence)

test_X = pad_features_for_word(test_X, 130) # Padding
X_test = torch.LongTensor(test_X)
loader_test = DataLoader(X_test, batch_size=16, shuffle=False)

evaluate_predict_table2 = []
model = BLSTM(INPUT_DIM,
              EMBEDDING_DIM,
              HIDDEN_DIM,
              FIRST_OUTPUT_DIM,
              OUTPUT_DIM,
              N_LAYERS,
              BIDIRECTIONAL,
              DROPOUT)
model.to(device)
model.embedding.weight.data.copy_(embedding_matrix)
model.load_state_dict(torch.load('./result/blstm2.pt')) # load pretrained model
prediction_table = model_evaluate(model, loader_test, evaluate_predict_table2)

term = [int(x[0]) for x in evaluate_predict_table2]
y_pred = [int(x[1]) for x in evaluate_predict_table2]

# Make test2.out file
i=0
newfile = open('./result/test2.out', "w")
with open('./data/test', "r") as test:
    for line in test:
        if not line.split(): # Ignore a blank line
            newfile.write('₩n')
```

```
            continue
        index, word_type = line.split(" ")[0], line.split(" ")[1].strip('₩n')
        for_tag = index_to_ner[y_pred[i]]
        newfile.write(str(index)+' '+str(word_type)+' '+for_tag+'₩n')
        i += 1
newfile.close()
```

In [21]:

```
import pickle
# save data
with open('./data/vocab_dictionary.pickle','wb') as fw1:
    pickle.dump(word_to_index, fw1)
with open('./data/ner_dictionary.pickle','wb') as fw2:
    pickle.dump(ner_to_index, fw2)
with open('./data/int_vocab_dictionary.pickle','wb') as fw3:
    pickle.dump(index_to_word, fw3)
with open('./data/int_ner_dictionary.pickle','wb') as fw4:
    pickle.dump(index_to_ner, fw4)
with open('./data/loader_train.pickle','wb') as fw5:
    pickle.dump(loader_train, fw5)
with open('./data/loader_dev.pickle','wb') as fw6:
    pickle.dump(loader_dev, fw6)
with open('./data/loader_test.pickle','wb') as fw7:
    pickle.dump(loader_test, fw7)
```

In [22]:

```
with open('./data/embedding_matrix.pickle','wb') as fw8:
    pickle.dump(embedding_matrix, fw8)
```

In [20]:

```
import pickle

# load data
with open('./data/vocab_dictionary.pickle', 'rb') as fr1:
    word_to_index = pickle.load(fr1)
with open('./data/ner_dictionary.pickle', 'rb') as fr2:
    index_to_word = pickle.load(fr2)
with open('./data/int_vocab_dictionary.pickle', 'rb') as fr3:
    ner_to_index = pickle.load(fr3)
with open('./data/int_ner_dictionary.pickle', 'rb') as fr4:
    index_to_ner = pickle.load(fr4)
with open('./data/loader_train.pickle', 'rb') as fr5:
    loader_train = pickle.load(fr5)
with open('./data/loader_dev.pickle', 'rb') as fr6:
    loader_dev = pickle.load(fr6)
with open('./data/loader_test.pickle', 'rb') as fr7:
    loader_test = pickle.load(fr7)
```

In [24]:

```python
checkpoint = {'INPUT_DIM':len(word_to_index),
              'EMBEDDING_DIM':100,
              'HIDDEN_DIM':256,
              'FIRST_OUTPUT_DIM':128,
              'OUTPUT_DIM':len(ner_to_index),
              'N_LAYERS':1,
              'BIDIRECTIONAL':True,
              'DROPOUT':0.33,
              'state_dict': model.state_dict()}

torch.save(checkpoint, 'result/checkpoint.pth')
```

In [ ]:

```python

```