```
# Licensing Information:  You are free to use or extend this project for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to The Georgia Institute of Technology, including a link to https://aritter.github.io

# Attribution Information: This assignment was developed at The Georgia Institute of Technology
# by Alan Ritter (alan.ritter@cc.gatech.edu)
```

Before you start working on this assignment, please make sure you have downloaded it to your local drive (File -> Download -> Download .ipynb) or made a copy to your own Google Drive (File -> Save a copy in Drive). Otherwise, your changes will not be saved.

Project #2: Named Entity Recognition

In this assignment, you will implement a bidirectional LSTM-CNN-CRF for sequence labeling, following this paper by Xuezhe Ma and Ed Hovy, on the CoNLL named entity recognition dataset. Before starting the assignment, we recommend reading the Ma and Hovy paper.

First, let's import some libraries and make sure the runtime has access to a GPU.

```
import torch
import torch.nn as nn
import torch.optim as optim

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Select the Runtime > "Change runtime type" menu to enable a GPU accelerator, ')
  print('and then re-execute this cell.')
else:
  print(gpu_info)

print(f'GPU available: {torch.cuda.is_available()}')
```

```
Mon Oct 24 21:54:39 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   69C    P8    12W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
```

```
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
GPU available: True
```

Download the Data

Run the following code to download the English part of the CoNLL 2003 dataset, the evaluation script and pre-filtered GloVe embeddings we are providing for this data.

```
#CoNLL 2003 data
!wget https://raw.githubusercontent.com/patverga/torch-ner-nlp-from-scratch/master/data/conll2003/e
!wget https://raw.githubusercontent.com/patverga/torch-ner-nlp-from-scratch/master/data/conll2003/e
!wget https://raw.githubusercontent.com/patverga/torch-ner-nlp-from-scratch/master/data/conll2003/e
!cat eng.train | awk '{print $1 "\t" $4}' > train
!cat eng.testa | awk '{print $1 "\t" $4}' > dev
!cat eng.testb | awk '{print $1 "\t" $4}' > test

#Evaluation Script
!wget https://raw.githubusercontent.com/aritter/twitter_nlp/master/data/annotated/wnut16/conlleval.

#Pre-filtered GloVe embeddings
!wget https://raw.githubusercontent.com/aritter/aritter.github.io/master/files/glove.840B.300d.conl
```

```
--2022-10-24 21:54:42--  https://raw.githubusercontent.com/patverga/torch-ner-nlp-from-scratc
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 3283420 (3.1M) [text/plain]
Saving to: ‘eng.train.2’

eng.train.2         100%[===================>]   3.13M  --.-KB/s    in 0.05s

2022-10-24 21:54:42 (63.4 MB/s) - ‘eng.train.2’ saved [3283420/3283420]

--2022-10-24 21:54:42--  https://raw.githubusercontent.com/patverga/torch-ner-nlp-from-scratc
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 827443 (808K) [text/plain]
Saving to: ‘eng.testa.2’

eng.testa.2         100%[===================>] 808.05K  --.-KB/s    in 0.03s

2022-10-24 21:54:42 (27.7 MB/s) - ‘eng.testa.2’ saved [827443/827443]

--2022-10-24 21:54:42--  https://raw.githubusercontent.com/patverga/torch-ner-nlp-from-scratc
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 748095 (731K) [text/plain]
Saving to: ‘eng.testb.2’

eng.testb.2         100%[===================>] 730.56K  --.-KB/s    in 0.03s
```

```
2022-10-24 21:54:43 (21.2 MB/s) -  'eng.testb.2'  saved [748095/748095]

--2022-10-24 21:54:43--  https://raw.githubusercontent.com/aritter/twitter_nlp/master/data/an
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 12754 (12K) [text/plain]
Saving to:  'conlleval.pl.9'

conlleval.pl.9      100%[===================>]  12.46K  --.-KB/s    in 0s

2022-10-24 21:54:43 (105 MB/s) -  'conlleval.pl.9'  saved [12754/12754]

--2022-10-24 21:54:43--  https://raw.githubusercontent.com/aritter/aritter.github.io/master/f
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 69798443 (67M) [text/plain]
Saving to:  'glove.840B.300d.conll_filtered.txt.2'

glove.840B.300d.con 100%[===================>]  66.56M   244MB/s    in 0.3s

2022-10-24 21:54:51 (244 MB/s) -  'glove.840B.300d.conll_filtered.txt.2'  saved [69798443/697
```

CoNLL Data Format

Run the following cell to see a sample of the data in CoNLL format. As you can see, each line in the file represents a word and its labeled named entity tag in BIO format. A blank line is used to seperate sentences.

```
!head -n 20 train
```

```
-DOCSTART-      O

EU      I-ORG
rejects 0
German  I-MISC
call    0
to      0
boycott 0
British I-MISC
lamb    0
.       0

Peter   I-PER
Blackburn       I-PER

BRUSSELS        I-LOC
1996-08-22      0

The     0
European        I-ORG
```

Reading in the Data

Below we proivide a bit of code to read in data in the CoNLL format. This also reads in the filtered GloVe embeddings, to save you some effort - we will discuss this more later.

```
#Read in the training data
def read_conll_format(filename):
    (words, tags, currentSent, currentTags) = ([],[],['-START-'],['START'])
    for line in open(filename).readlines():
        line = line.strip()
        #print(line)
        if line == "":
            currentSent.append('-END-')
            currentTags.append('END')
            words.append(currentSent)
            tags.append(currentTags)
            (currentSent, currentTags) = (['-START-'], ['START'])
        else:
            (word, tag) = line.split()
            currentSent.append(word)
            currentTags.append(tag)
    return (words, tags)

def sentences2char(sentences):
    return [[['start'] + [c for c in w] + ['end'] for w in l] for l in sentences]


(sentences_train, tags_train) = read_conll_format("train")
(sentences_dev, tags_dev)     = read_conll_format("dev")

print("The second sentence in train set:", sentences_train[2])
print("The NER label of the sentence:   ", tags_train[2])

sentencesChar = sentences2char(sentences_train)

print("The char repersentation of the sentence:", sentencesChar[2])
```

```
The second sentence in train set: ['-START-', 'Peter', 'Blackburn', '-END-']
The NER label of the sentence:    ['START', 'I-PER', 'I-PER', 'END']
The char repersentation of the sentence: [['start', '-', 'S', 'T', 'A', 'R', 'T', '-', 'end']
```

```
#Read GloVe embeddings.
def read_GloVe(filename):
    embeddings = {}
    for line in open(filename).readlines():
        #print(line)
        fields = line.strip().split(" ")
        word = fields[0]
        embeddings[word] = [float(x) for x in fields[1:]]
    return embeddings
```

```
GloVe = read_GloVe("glove.840B.300d.conll_filtered.txt")

print("The GloVe word embedding of the word 'the':", GloVe["the"])
print("dimension of glove embedding:", len(GloVe["the"]))
```

```
The GloVe word embedding of the word 'the': [0.27204, -0.06203, -0.1884, 0.023225, -0.018158,
dimension of glove embedding: 300
```

Mapping Tokens to Indices

As in the last project, we will need to convert words in the dataset to numeric indices, so they can be presented as input to a neural network. Code to handle this for you with sample usage is provided below.

```
#Create mappings between tokens and indices.

from collections import Counter
import random

#Will need this later to remove 50% of words that only appear once in the training data from the vo
wordCounts = Counter([w for l in sentences_train for w in l])
charCounts = Counter([c for l in sentences_train for w in l for c in w])
singletons = set([w for (w,c) in wordCounts.items() if c == 1 and not w in GloVe.keys()])
charSingletons = set([w for (w,c) in charCounts.items() if c == 1])

#Build dictionaries to map from words, characters to indices and vice versa.
#Save first two words in the vocabulary for padding and "UNK" token.
word2i = {w:i+2 for i,w in enumerate(set([w for l in sentences_train for w in l] + list(GloVe.keys(
char2i = {w:i+2 for i,w in enumerate(set([c for l in sentencesChar for w in l for c in w]))}
i2word = {i:w for w,i in word2i.items()}
i2char = {i:w for w,i in char2i.items()}

vocab_size = max(word2i.values()) + 1
char_vocab_size = max(char2i.values()) + 1

#Tag dictionaries.
tag2i = {w:i for i,w in enumerate(set([t for l in tags_train for t in l]))}
i2tag = {i:t for t,i in tag2i.items()}

#When training, randomly replace singletons with UNK tokens sometimes to simulate situation at test
def getDictionaryRandomUnk(w, dictionary, train=False):
    if train and (w in singletons and random.random() > 0.5):
        return 1
    else:
        return dictionary.get(w, 1)

#Map a list of sentences from words to indices.
def sentences2indices(words, dictionary, train=False):
    #1.0 => UNK
    return [[getDictionaryRandomUnk(w,dictionary, train=train) for w in l] for l in words]
```

```
#Map a list of sentences containing to indices (character indices)
def sentences2indicesChar(chars, dictionary):
    #1.0 => UNK
    return [[[dictionary.get(c,1) for c in w] for w in l] for l in chars]

#Indices
X       = sentences2indices(sentences_train, word2i, train=True)
X_char  = sentences2indicesChar(sentencesChar, char2i)
Y       = sentences2indices(tags_train, tag2i)

print("vocab size:", vocab_size)
print("char vocab size:", char_vocab_size)
print()

print("index of word 'the':", word2i["the"])
print("word of index 253:", i2word[253])
print()

#Print out some examples of what the dev inputs will look like
for i in range(10):
    print(" ".join([i2word.get(w,'UNK') for w in X[i]]))
```

```
vocab size: 29148
char vocab size: 88

index of word 'the': 18472
word of index 253: 0.59

-START- -DOCSTART- -END-
-START- EU rejects German call to boycott British lamb . -END-
-START- Peter Blackburn -END-
-START- BRUSSELS 1996-08-22 -END-
-START- The European Commission said on Thursday it disagreed with German advice to consumers
-START- Germany 's representative to the European Union 's veterinary committee Werner Zwingm
-START- " We do n't support any such recommendation because we do n't see any grounds for it
-START- He said further scientific study was required and if it was found that action was nee
-START- He said a proposal last month by EU Farm Commissioner Franz Fischler to ban sheep bra
-START- Fischler proposed EU-wide measures after reports from Britain and France that under l
```

Padding and Batching

In this assignment, you should train your models using minibatched SGD. When presenting multiple sentences to the network at the same time, we will need to pad them to be of the same length. We use torch.nn.utils.rnn.pad_sequence to do so.

Below we provide some code to prepare batches of data to present to the network. We pad the sequence so that all the sequences have the same length.

Side Note: PyTorch includes utilities in `torch.utils.data` to help with padding, batching, shuffling and some other things, but for this assignment we will do everything from scratch to help you see exactly how this works.

```
#Pad inputs to max sequence length (for batching)
def prepare_input(X_list):
    X_padded = torch.nn.utils.rnn.pad_sequence([torch.as_tensor(l) for l in X_list], batch_first=Tr
    X_mask   = torch.nn.utils.rnn.pad_sequence([torch.as_tensor([1.0] * len(l)) for l in X_list], b
    return (X_padded, X_mask)

#Maximum word length (for character representations)
MAX_CLEN = 32

def prepare_input_char(X_list):
    MAX_SLEN = max([len(l) for l in X_list])
    X_padded  = [l + [[]]*(MAX_SLEN-len(l))  for l in X_list]
    X_padded  = [[w[0:MAX_CLEN] for w in l] for l in X_padded]
    X_padded  = [[w + [1]*(MAX_CLEN-len(w)) for w in l] for l in X_padded]
    return torch.as_tensor(X_padded).type(torch.LongTensor)

#Pad outputs using one-hot encoding
def prepare_output_onehot(Y_list, NUM_TAGS=max(tag2i.values())+1):
    Y_onehot = [torch.zeros(len(l), NUM_TAGS) for l in Y_list]
    for i in range(len(Y_list)):
        for j in range(len(Y_list[i])):
            Y_onehot[i][j,Y_list[i][j]] = 1.0
    Y_padded = torch.nn.utils.rnn.pad_sequence(Y_onehot, batch_first=True).type(torch.FloatTensor)
    return Y_padded

print("max slen:", max([len(x) for x in X_char]))

(X_padded, X_mask) = prepare_input(X)
X_padded_char      = prepare_input_char(X_char)
Y_onehot           = prepare_output_onehot(Y)

print("X_padded:", X_padded.shape)
print("X_mask:", X_mask.shape)
print("X_padded_char:", X_padded_char.shape)
print("Y_onehot:", Y_onehot.shape)
```

```
max slen: 115
X_padded: torch.Size([14987, 115])
X_mask: torch.Size([14987, 115])
X_padded_char: torch.Size([14987, 115, 32])
Y_onehot: torch.Size([14987, 115, 10])
```

Your code starts here: Basic LSTM Tagger (10 points)

OK, now you should have everything you need to get started.

Recall that your goal is to to implement the BiLSTM-CNN-CRF, as described in (Ma and Hovy, 2016). This is a relatively complex network with various components. Below we provide starter code to break down your implementation into increasingly complex versions of the final model, starting with a Basic LSTM tagger. This way you can be confident that each part is working correctly before incrementally increasing the complexity of your implementation. This is generally a good approach to take when implementing complex models, since buggy PyTorch

code is often partially working, but produces worse results than a correct implementation, so it's hard to know whether added complexities are helping or hurting. Also, if you aren't able to match published results it's hard to know which component of your model has the problem (or even whether or not it is a problem in the published result!)

Fill in the functions marked as `TODO` in the code block below. Please make your code changes only within the given commented block (#####). If everything is working correctly, you should be able to achieve an **F1 score of 0.86 on the dev set and 0.82 on the test set (with GloVe embeddings)**. You are required to initialize word embeddings with GloVe later, but you can randomly initialize the word embeddings in the beginning.

```
##############################################################################################
#TODO: Add imports if needed:

##############################################################################################

class BasicLSTMtagger(nn.Module):
    def __init__(self, DIM_EMB=10, DIM_HID=10):
        super(BasicLSTMtagger, self).__init__()
        NUM_TAGS = max(tag2i.values())+1

        (self.DIM_EMB, self.NUM_TAGS) = (DIM_EMB, NUM_TAGS)
        ##############################################################################################
        #TODO: initialize parameters - embedding layer, nn.LSTM, nn.Linear and nn.LogSoftmax
        self.DIM_HID=DIM_HID
        self.embedding=nn.Embedding(vocab_size,self.DIM_EMB,padding_idx=-1)
        self.lstm=nn.LSTM(input_size=self.DIM_EMB,hidden_size=self.DIM_HID,num_layers=2,bidirection
        self.linear=nn.Linear(self.DIM_HID*2,self.NUM_TAGS)
        self.logsoftmax=nn.LogSoftmax(dim=2)
        self.init_glove(GloVe)
        ##############################################################################################

    def forward(self, X, train=True):
        ##############################################################################################
        #TODO: Implement the forward computation.
        if train:
          x=self.embedding(X)
          x,_=self.lstm(x)
          x=self.linear(x)#[:,:,self.DIM_HID:])
          #x=self.logsoftmax(x)
          return x
        else:
          return torch.randn((X.shape[0], X.shape[1], self.NUM_TAGS))
        #return torch.randn((X.shape[0], X.shape[1], self.NUM_TAGS))  #Random baseline.

        ##############################################################################################

    def init_glove(self, GloVe):
        ##############################################################################################
        #TODO: initialize word embeddings using GloVe (you can skip this part in your first version
        for word, index in word2i.items():
          if word in GloVe:
```

```
            self.embedding.weight.data[index]=torch.tensor(GloVe[word])

        ############################################################################################

    def inference(self, sentences):
        X, X_mask       = prepare_input(sentences2indices(sentences, word2i))
        pred = self.forward(X.cuda()).argmax(dim=2)
        return [[i2tag[pred[i,j].item()] for j in range(len(sentences[i]))] for i in range(len(sent

    def print_predictions(self, words, tags):
        Y_pred = self.inference(words)
        for i in range(len(words)):
            print("----------------------------")
            print(" ".join([f"{words[i][j]}/{Y_pred[i][j]}/{tags[i][j]}" for j in range(len(words[i
            print("Predicted:\t", Y_pred[i])
            print("Gold:\t\t", tags[i])

    def write_predictions(self, sentences, outFile):
        fOut = open(outFile, 'w')
        for s in sentences:
            y = self.inference([s])[0]
            #print("\n".join(y[1:len(y)-1]))
            fOut.write("\n".join(y[1:len(y)-1]))  #Skip start and end tokens
            fOut.write("\n\n")

#The following code will initialize a model and test that your forward computation runs without err
lstm_test   = BasicLSTMtagger(DIM_HID=7, DIM_EMB=300)
lstm_output = lstm_test.forward(prepare_input(X[0:5])[0])
Y_onehot    = prepare_output_onehot(Y[0:5])

#Check the shape of the lstm_output and one-hot label tensors.
print("lstm output shape:", lstm_output.shape)
print("Y onehot shape:", Y_onehot.shape)
```

```
lstm output shape: torch.Size([5, 32, 10])
Y onehot shape: torch.Size([5, 32, 10])
```

```
#Read in the data

(sentences_dev, tags_dev)     = read_conll_format('dev')
(sentences_train, tags_train) = read_conll_format('train')
(sentences_test, tags_test)   = read_conll_format('test')
```

Train your Model (10 points)

Next, implement the function below to train your basic BiLSTM tagger. See [torch.nn.lstm](). Make sure to save your predictions on the test set (`test_pred_lstm.txt`) for submission to GradeScope. Feel free to change number of epochs, optimizer, learning rate and batch size.

```
#Training
```

```
    ###############################################################################################
    #TODO: Add imports if needed:
    import numpy as np
    ###############################################################################################

    from random import sample
    import tqdm
    import os
    import subprocess
    import random

    def shuffle_sentences(sentences, tags):
        shuffled_sentences = []
        shuffled_tags      = []
        indices = list(range(len(sentences)))
        random.shuffle(indices)
        for i in indices:
            shuffled_sentences.append(sentences[i])
            shuffled_tags.append(tags[i])
        return (shuffled_sentences, shuffled_tags)


    def train_basic_lstm(sentences, tags, lstm):
        ###############################################################################################
        #TODO: initialize optimizer and other hyperparameters.
        # optimizer = optim.Adadelta(lstm.parameters(), lr=0.1)
        optimizer = optim.SGD(lstm.parameters(), lr=0.1)
        #optimizer.zero_grad()
        #optimizer=optim.Adam(lstm.parameters())
        batchSize = 50
        nEpochs = 10
        #criterion=nn.NLLLoss(ignore_index=0)
        criterion=nn.CrossEntropyLoss(ignore_index=-1)
        #lstm.train()
        ###############################################################################################

        for epoch in range(nEpochs):
            totalLoss = 0.0
            optimizer.zero_grad()
            (sentences_shuffled, tags_shuffled) = shuffle_sentences(sentences, tags)
            for batch in tqdm.notebook.tqdm(range(0, len(sentences), batchSize), leave=False):
                ###############################################################################################
                #TODO: Implement gradient update.
                '''
                optimizer.zero_grad()
                X=sentences2indices(sentences_shuffled[batch:min(batch+batchSize,len(sentences_shuffled
                X=prepare_input(X)[0].cuda()

                Y=sentences2indices(tags_shuffled[batch:min(batch+batchSize,len(sentences_shuffled))],
                Y=prepare_output_onehot(Y).cuda()
                Y=torch.argmax(Y,dim=2)

                lstm_output=lstm(X).cuda()
                lstm_output=lstm_output.permute(0,2,1)

                l
```

```
            loss=criterion(lstm_output,Y)
            loss.backward()
            optimizer.step()
            totalLoss+=loss

            '''
            #optimizer.zero_grad()
            for_train=sentences_shuffled[batch:min(batch+batchSize,len(sentences_shuffled))]
            for_train=sentences2indices(for_train, word2i, train=True)
            for_train,_=prepare_input(for_train)
            for_test=tags_shuffled[batch:min(batch+batchSize,len(sentences_shuffled))]
            for_test=sentences2indices(for_test, tag2i)
            for_test=prepare_output_onehot(for_test)
            for_test=torch.argmax(for_test,dim=2)
            #output=lstm.forward(for_train.cuda(),True)

            output=lstm(for_train.cuda()).cuda()
            out=output.permute(0,2,1)
            output=criterion(out.cuda(),for_test.cuda())
            output.backward()
            optimizer.step()
            totalLoss += output

            ##############################################################################################

        print(f"loss on epoch {epoch} = {totalLoss}")
        lstm.write_predictions(sentences_dev, 'dev_pred')   #Performance on dev set
        print('conlleval:')
        print(subprocess.Popen('paste dev dev_pred | perl conlleval.pl -d "\t"', shell=True, stdout

        if epoch % 10 == 0:
            s = sample(range(len(sentences_dev)), 5)
            lstm.print_predictions([sentences_dev[i] for i in s], [tags_dev[i] for i in s])

lstm = BasicLSTMtagger(DIM_HID=500, DIM_EMB=300).cuda()
train_basic_lstm(sentences_train, tags_train, lstm)
```

```
              ORG: precision:  69.42%; recall:  81.75%; FB1:  75.09  1572
              PER: precision:  89.89%; recall:  87.89%; FB1:  88.88  1801

loss on epoch 4 = 4.349615573883057
conlleval:
processed 51578 tokens with 5942 phrases; found: 6021 phrases; correct: 5146.
accuracy:  97.87%; precision:  85.47%; recall:  86.60%; FB1:  86.03
              LOC: precision:  92.53%; recall:  86.28%; FB1:  89.30  1713
             MISC: precision:  79.15%; recall:  79.07%; FB1:  79.11  921
              ORG: precision:  78.41%; recall:  80.69%; FB1:  79.53  1380
              PER: precision:  87.19%; recall:  95.01%; FB1:  90.93  2007

loss on epoch 5 = 3.448348045349121
conlleval:
processed 51578 tokens with 5942 phrases; found: 6214 phrases; correct: 5142.
accuracy:  97.62%; precision:  82.75%; recall:  86.54%; FB1:  84.60
              LOC: precision:  93.06%; recall:  84.65%; FB1:  88.65  1671
             MISC: precision:  71.65%; recall:  85.25%; FB1:  77.86  1097
              ORG: precision:  68.71%; recall:  82.70%; FB1:  75.06  1614
              PER: precision:  92.36%; recall:  91.86%; FB1:  92.11  1832

loss on epoch 6 = 2.998522996902466
conlleval:
processed 51578 tokens with 5942 phrases; found: 6165 phrases; correct: 5298.
accuracy:  98.13%; precision:  85.94%; recall:  89.16%; FB1:  87.52
              LOC: precision:  91.67%; recall:  92.27%; FB1:  91.97  1849
             MISC: precision:  75.19%; recall:  85.47%; FB1:  80.00  1048
              ORG: precision:  79.60%; recall:  80.91%; FB1:  80.25  1363
              PER: precision:  90.81%; recall:  93.92%; FB1:  92.34  1905

loss on epoch 7 = 2.3786909580230713
conlleval:
processed 51578 tokens with 5942 phrases; found: 6086 phrases; correct: 5208.
accuracy:  97.87%; precision:  85.57%; recall:  87.65%; FB1:  86.60
              LOC: precision:  84.86%; recall:  95.21%; FB1:  89.74  2061
             MISC: precision:  83.49%; recall:  76.79%; FB1:  80.00  848
              ORG: precision:  78.03%; recall:  77.85%; FB1:  77.94  1338
              PER: precision:  92.82%; recall:  92.67%; FB1:  92.75  1839

loss on epoch 8 = 2.0380096435546875
conlleval:
processed 51578 tokens with 5942 phrases; found: 5949 phrases; correct: 5334.
accuracy:  98.39%; precision:  89.66%; recall:  89.77%; FB1:  89.71
              LOC: precision:  92.83%; recall:  94.50%; FB1:  93.66  1870
             MISC: precision:  84.68%; recall:  83.95%; FB1:  84.31  914
              ORG: precision:  85.22%; recall:  81.28%; FB1:  83.21  1279
              PER: precision:  91.94%; recall:  94.14%; FB1:  93.03  1886

loss on epoch 9 = 1.4669190645217896
conlleval:
processed 51578 tokens with 5942 phrases; found: 6004 phrases; correct: 5294.
accuracy:  98.28%; precision:  88.17%; recall:  89.09%; FB1:  88.63
              LOC: precision:  93.52%; recall:  93.52%; FB1:  93.52  1837
             MISC: precision:  87.51%; recall:  80.59%; FB1:  83.91  849
              ORG: precision:  76.55%; recall:  87.40%; FB1:  81.62  1531
              PER: precision:  92.95%; recall:  90.17%; FB1:  91.54  1787
```

```
#Evaluation on test data
lstm.write_predictions(sentences_test, 'test_pred_lstm.txt')
!wget https://raw.githubusercontent.com/aritter/twitter_nlp/master/data/annotated/wnut16/conlleval.
!paste test test_pred_lstm.txt | perl conlleval.pl -d "\t"
```

```
--2022-10-24 22:05:09--  https://raw.githubusercontent.com/aritter/twitter_nlp/master/data/an
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 12754 (12K) [text/plain]
Saving to: ‘conlleval.pl.11’

conlleval.pl.11     100%[===================>]  12.46K  --.-KB/s    in 0.001s

2022-10-24 22:05:09 (16.6 MB/s) - ‘conlleval.pl.11’ saved [12754/12754]

processed 46666 tokens with 5648 phrases; found: 5756 phrases; correct: 4773.
accuracy:  97.05%; precision:  82.92%; recall:  84.51%; FB1:  83.71
              LOC: precision:  89.28%; recall:  89.87%; FB1:  89.57  1679
             MISC: precision:  73.96%; recall:  71.23%; FB1:  72.57  676
              ORG: precision:  75.58%; recall:  86.82%; FB1:  80.81  1908
              PER: precision:  89.22%; recall:  82.37%; FB1:  85.66  1493
```

Initialization with GloVe Embeddings (5 points)

If you haven't already, implement the `init_glove()` method in `BasicLSTMtagger` above.

Rather than initializing word embeddings randomly, it is common to use learned word embeddings (GloVe or Word2Vec), as discussed in lecture. To make this simpler, we have already pre-filtered GloVe embeddings to only contain words in the vocabulary of the CoNLL NER dataset, and loaded them into a dictionary (`GloVe`) at the beginning of this notebook.

Character Embeddings (10 points)

Now that you have your basic LSTM tagger working, the next step is to add a convolutional network that computes word embeddings from character representations of words. See Figure 2 and Figure 3 in the Ma and Hovy paper. We have provided code in `sentences2input_tensors` to convert sentences into lists of word and character indices. See also nn.Conv1d and MaxPool1d.

Hint: The nn.Conv1d accepts input size (N, C_{in}, L_{in}), but we have input size $(N, \text{SLEN}, \text{CLEN}, \text{EMB\_DIM})$. We can reshape and permute our input to satisfy the nn.Conv1d, and recover the dimensions later.

Make sure to save your predictions on the test set, for submission to GradeScope. You should be able to achieve **90 F1 / 85 F1 on the dev/test sets**.

Fill in the functions marked as `TODO` in the code block below. Please make your code changes only within the given commented block (#####).

```
##############################################################################################
#TODO: Add imports if needed:
from random import sample
import tqdm
import os
import subprocess
import random

import torch.nn.functional as F
##############################################################################################


class CharLSTMtagger(BasicLSTMtagger):
    def __init__(self, DIM_EMB=10, DIM_CHAR_EMB=30, DIM_HID=10):
        super(CharLSTMtagger, self).__init__(DIM_EMB=DIM_EMB, DIM_HID=DIM_HID)
        NUM_TAGS = max(tag2i.values())+1

        (self.DIM_EMB, self.NUM_TAGS) = (DIM_EMB, NUM_TAGS)
        ##############################################################################################
        #TODO: Initialize parameters.
```

```
        self.DIM_HID=DIM_HID
        self.DIM_CHAR_EMB=DIM_CHAR_EMB
        self.embedding=nn.Embedding(vocab_size,DIM_EMB,padding_idx=-1)
        self.embedding_char=nn.Embedding(vocab_size,DIM_CHAR_EMB,padding_idx=-1)
        self.cnn=nn.Conv1d(DIM_CHAR_EMB,DIM_CHAR_EMB,1)
        self.maxpool=nn.MaxPool1d(32)
        self.lstm=nn.LSTM(input_size=self.DIM_EMB+self.DIM_CHAR_EMB,hidden_size=self.DIM_HID,num_la
        self.linear=nn.Linear(self.DIM_HID*2,self.NUM_TAGS)
        self.logsoftmax=nn.LogSoftmax(dim=2)
        self.init_glove(GloVe)
        ##############################################################################################

    def forward(self, X, X_char, train=False):
        ##############################################################################################
        #TODO: Implement the forward computation.
        x=self.embedding(X)
        X_char=self.embedding_char(X_char)
        X_char=X_char.reshape((X_char.shape[0]*X_char.shape[1],X_char.shape[2],X_char.shape[3]))
        X_char=X_char.permute(0,2,1)
        x2=self.cnn(X_char)
        x2=self.maxpool(x2)
        x2=x2.reshape((x2.shape[0],x2.shape[1]))
        x2=x2.reshape((x.shape[0],x.shape[1],x2.shape[1]))
        x,_=self.lstm(torch.cat((x,x2),2))
        x=self.linear(x)#[:,:,self.DIM_HID:])
        return x
        #return torch.randn((X.shape[0], X.shape[1], self.NUM_TAGS))  #Random baseline.
        ##############################################################################################

    def sentences2input_tensors(self, sentences):
        (X, X_mask)   = prepare_input(sentences2indices(sentences, word2i))
        X_char        = prepare_input_char(sentences2indicesChar(sentences, char2i))
        return (X, X_mask, X_char)

    def inference(self, sentences):
        (X, X_mask, X_char) = self.sentences2input_tensors(sentences)
        pred = self.forward(X.cuda(), X_char.cuda()).argmax(dim=2)
        return [[i2tag[pred[i,j].item()] for j in range(len(sentences[i]))] for i in range(len(sent

    def print_predictions(self, words, tags):
        Y_pred = self.inference(words)
        for i in range(len(words)):
            print("--------------------------")
            print(" ".join([f"{words[i][j]}/{Y_pred[i][j]}/{tags[i][j]}" for j in range(len(words[i
            print("Predicted:\t", Y_pred[i])
            print("Gold:\t\t", tags[i])

char_lstm_test = CharLSTMtagger(DIM_HID=7, DIM_EMB=300)
lstm_output    = char_lstm_test.forward(prepare_input(X[0:5])[0], prepare_input_char(X_char[0:5]))
Y_onehot       = prepare_output_onehot(Y[0:5])

print("lstm output shape:", lstm_output.shape)
print("Y onehot shape:", Y_onehot.shape)
```

```
lstm output shape: torch.Size([5, 32, 10])
```

```
      Y onehot shape: torch.Size([5, 32, 10])

#Training LSTM w/ character embeddings. Feel free to change number of epochs, optimizer, learning r

#############################################################################################
#TODO: Add imports if necessary.

#############################################################################################


def shuffle_sentences(sentences, tags):
    shuffled_sentences = []
    shuffled_tags      = []
    indices = list(range(len(sentences)))
    random.shuffle(indices)
    for i in indices:
        shuffled_sentences.append(sentences[i])
        shuffled_tags.append(tags[i])
    return (shuffled_sentences, shuffled_tags)

def train_char_lstm(sentences, tags, lstm):
    #########################################################################################
    #TODO: initialize optimizer and other hyperparameters.
    # optimizer = optim.Adadelta(lstm.parameters(), lr=0.1)

    nEpochs = 10
    batchSize = 50
    optimizer = optim.SGD(lstm.parameters(), lr=0.1)
    criterion=nn.CrossEntropyLoss(ignore_index=-1)
    optimizer.zero_grad()
    #########################################################################################

    for epoch in range(nEpochs):
        totalLoss = 0.0
        optimizer.zero_grad()
        (sentences_shuffled, tags_shuffled) = shuffle_sentences(sentences, tags)
        for batch in tqdm.notebook.tqdm(range(0, len(sentences), batchSize), leave=False):
            #####################################################################################
            #TODO: Implement gradient update.
            #optimizer.zero_grad()
            for_train=sentences_shuffled[batch:min(batch+batchSize,len(sentences_shuffled))]
            for_train=sentences2indices(for_train, word2i, train=True)
            for_train,_=prepare_input(for_train)

            for_train_char=sentences_shuffled[batch:min(batch+batchSize,len(sentences_shuffled))]
            for_train_char=sentences2indicesChar(for_train_char, char2i)
            for_train_char=prepare_input_char(for_train_char)

            for_test=tags_shuffled[batch:min(batch+batchSize,len(sentences_shuffled))]
            for_test=sentences2indices(for_test, tag2i)
            for_test=prepare_output_onehot(for_test)
            for_test=torch.argmax(for_test,dim=2)

            output=lstm(for_train.cuda(),for_train_char.cuda()).cuda()
            out=output.permute(0,2,1)
            output=criterion(out.cuda(),for_test.cuda())
```

```
            output=criterion(out.cuda(),for_test.cuda())
            output.backward()
            optimizer.step()
            totalLoss += output
            ##########################################################################################

        print(f"loss on epoch {epoch} = {totalLoss}")
        lstm.write_predictions(sentences_dev, 'dev_pred')   #Performance on dev set
        print('conlleval:')
        print(subprocess.Popen('paste dev dev_pred | perl conlleval.pl -d "\t"', shell=True, stdout

        if epoch % 10 == 0:
            s = sample(range(len(sentences_dev)), 5)
            lstm.print_predictions([sentences_dev[i] for i in s], [tags_dev[i] for i in s])

char_lstm = CharLSTMtagger(DIM_HID=500, DIM_EMB=300).cuda()
train_char_lstm(sentences_train, tags_train, char_lstm)
```

```
loss on epoch 4 = 8.49934196472168
conlleval:
processed 51578 tokens with 5942 phrases; found: 6065 phrases; correct: 4805.
accuracy:  96.95%; precision:  79.23%; recall:  80.87%; FB1:  80.04
              LOC: precision:  84.72%; recall:  89.06%; FB1:  86.84  1931
             MISC: precision:  68.90%; recall:  77.11%; FB1:  72.77  1032
              ORG: precision:  74.71%; recall:  52.20%; FB1:  61.46  937
              PER: precision:  81.20%; recall:  95.44%; FB1:  87.75  2165

loss on epoch 5 = 6.360720634460449
conlleval:
processed 51578 tokens with 5942 phrases; found: 6022 phrases; correct: 5140.
accuracy:  97.83%; precision:  85.35%; recall:  86.50%; FB1:  85.92
              LOC: precision:  88.97%; recall:  92.22%; FB1:  90.56  1904
             MISC: precision:  77.15%; recall:  75.81%; FB1:  76.48  906
              ORG: precision:  78.15%; recall:  75.47%; FB1:  76.78  1295
              PER: precision:  90.51%; recall:  94.19%; FB1:  92.31  1917

loss on epoch 6 = 6.323833465576172
conlleval:
processed 51578 tokens with 5942 phrases; found: 6203 phrases; correct: 4768.
accuracy:  96.89%; precision:  76.87%; recall:  80.24%; FB1:  78.52
              LOC: precision:  81.42%; recall:  80.40%; FB1:  80.91  1814
             MISC: precision:  74.26%; recall:  73.21%; FB1:  73.73  909
              ORG: precision:  59.25%; recall:  79.27%; FB1:  67.81  1794
              PER: precision:  92.11%; recall:  84.31%; FB1:  88.04  1686

loss on epoch 7 = 5.302734851837158
conlleval:
processed 51578 tokens with 5942 phrases; found: 6119 phrases; correct: 5180.
accuracy:  97.96%; precision:  84.65%; recall:  87.18%; FB1:  85.90
                  : precision:   0.00%; recall:   0.00%; FB1:   0.00  1
              LOC: precision:  88.94%; recall:  90.20%; FB1:  89.57  1863
             MISC: precision:  80.39%; recall:  76.90%; FB1:  78.60  882
              ORG: precision:  71.68%; recall:  83.22%; FB1:  77.02  1557
              PER: precision:  93.50%; recall:  92.18%; FB1:  92.84  1816

loss on epoch 8 = 3.7416317462921143
conlleval:
processed 51578 tokens with 5942 phrases; found: 6119 phrases; correct: 5315.
accuracy:  98.30%; precision:  86.86%; recall:  89.45%; FB1:  88.14
              LOC: precision:  90.07%; recall:  94.34%; FB1:  92.16  1924
             MISC: precision:  82.84%; recall:  81.67%; FB1:  82.25  909
              ORG: precision:  81.93%; recall:  79.79%; FB1:  80.85  1306
              PER: precision:  88.84%; recall:  95.49%; FB1:  92.05  1980

loss on epoch 9 = 3.075565814971924
conlleval:
processed 51578 tokens with 5942 phrases; found: 6012 phrases; correct: 5349.
accuracy:  98.35%; precision:  88.97%; recall:  90.02%; FB1:  89.49
                  : precision:   0.00%; recall:   0.00%; FB1:   0.00  5
              LOC: precision:  92.37%; recall:  94.88%; FB1:  93.61  1887
             MISC: precision:  82.92%; recall:  80.59%; FB1:  81.74  896
              ORG: precision:  83.76%; recall:  82.70%; FB1:  83.23  1324
              PER: precision:  92.32%; recall:  95.22%; FB1:  93.75  1900
```

```
#Evaluation on test set
char_lstm.write_predictions(sentences_test, 'test_pred_cnn_lstm.txt')
!wget https://raw.githubusercontent.com/aritter/twitter_nlp/master/data/annotated/wnut16/conlleval.
!paste test test_pred_cnn_lstm.txt | perl conlleval.pl -d "\t"
```

```
--2022-10-24 22:18:05--  https://raw.githubusercontent.com/aritter/twitter_nlp/master/data/an
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... c
HTTP request sent, awaiting response... 200 OK
Length: 12754 (12K) [text/plain]
Saving to: ‘conlleval.pl.12’

conlleval.pl.12     100%[===================>]  12.46K  --.-KB/s    in 0s

2022-10-24 22:18:05 (65.3 MB/s) - ‘conlleval.pl.12’ saved [12754/12754]

processed 46666 tokens with 5648 phrases; found: 5789 phrases; correct: 4855.
accuracy:  97.38%; precision:  83.87%; recall:  85.96%; FB1:  84.90
                 : precision:   0.00%; recall:   0.00%; FB1:   0.00  1
               LOC: precision:  86.10%; recall:  92.09%; FB1:  88.99  1784
              MISC: precision:  70.31%; recall:  74.22%; FB1:  72.21  741
               ORG: precision:  78.82%; recall:  78.87%; FB1:  78.84  1662
               PER: precision:  92.94%; recall:  92.02%; FB1:  92.48  1601
```

Conditional Random Fields (5 points - optional extra credit)

Now we are ready to add a CRF layer to the `CharacterLSTMTagger`. To train the model, implement `conditional_log_likelihood`, using the score (unnormalized log probability) of the gold sequence, in addition to the partition function, $Z(X)$, which is computed using the forward algorithm. Then, you can simply use Pytorch's automatic differentiation to compute gradients by running backpropagation through the computation graph of the dynamic program (this should be very simple, so long as you are able to correctly implement the forward algorithm using a computation graph that is supported by PyTorch). This approach to computing gradients for CRFs is discussed in Section 7.5.3 of the Eisenstein Book

You will also need to implement the Viterbi algorithm for inference during decoding.

After including CRF training and Viterbi decoding, you should be getting about **92 F1 / 88 F1 on the dev and test set**, respectively.

IMPORTANT: Note that training will be substantially slower this time - depending on the efficiency of your implementation, it could take about 5 minutes per epoch (e.g. 50 minutes for 10 iterations). It is recommended to start out training on a single batch of data (and testing on this same batch), so that you can quickly debug, making sure your model can memorize the labels on a single batch, and then optimize your code. Once you are fairly confident your code is working properly, then you can train using the full dataset. We have provided a (commented out) line of code to switch between training on a single batch and the full dataset below.

Hint #1: While debugging your implementation of the Forward algorithm it is helpful to look at the loss during training. The loss should never be less than zero (the log-likelihood should always be negative).

Hint #2: To sum log-probabilities in a numerically stable way at the end of the Forward algorithm, you will want to use `torch.logsumexp`.

Fill in the functions marked as `TODO` in the code block below. Please make your code changes only within the given commented block (#####).

```
import torch.nn.functional as F

#################################################################################################
#TODO: Add imports if needed.

#################################################################################################

class LSTM_CRFtagger(CharLSTMtagger):
    def __init__(self, DIM_EMB=10, DIM_CHAR_EMB=30, DIM_HID=10):
        super(LSTM_CRFtagger, self).__init__(DIM_EMB=DIM_EMB, DIM_HID=DIM_HID, DIM_CHAR_EMB=DIM_CHA
        #################################################################################################
        #TODO: Initialize parameters.
        N_TAGS=max(tag2i.values())+1

        self.transitionWeights = nn.Parameter(torch.zeros((N_TAGS, N_TAGS), requires_grad=True))
        nn.init.normal_(self.transitionWeights)
        #################################################################################################

    def gold_score(self, lstm_scores, Y):
        #################################################################################################
        #TODO: compute score of gold sequence Y (unnormalized conditional log-probability)

        return 0
        #################################################################################################


    #Forward algorithm for a single sentence
    #Efficiency will eventually be important here.  We recommend you start by
    #training on a single batch and make sure your code can memorize the
```

```
    #training data.  Then you can go back and re-write the inner loop using
    #tensor operations to speed things up.
    def forward_algorithm(self, lstm_scores, sLen):
        ###################################################################################################
        #TODO: implement forward algorithm.
        return 0

        ###################################################################################################

    def conditional_log_likelihood(self, sentences, tags, train=True):
        ###################################################################################################
        #TODO: compute conditional log likelihood of Y (use forward_algorithm and gold_score)
        return 0

        ###################################################################################################

    def viterbi(self, lstm_scores, sLen):
        ###################################################################################################
        #TODO: Implement Viterbi algorithm, soring backpointers to recover the argmax sequence.  Re
        return (torch.as_tensor([random.randint(0,lstm_scores.shape[1]-1) for x in range(sLen)]), 0

        ###################################################################################################

    #Computes Viterbi sequences on a batch of data.
    def viterbi_batch(self, sentences):
        viterbiSeqs = []
        (X, X_mask, X_char) = self.sentences2input_tensors(sentences)
        lstm_scores = self.forward(X.cuda(), X_char.cuda())
        for s in range(len(sentences)):
            (viterbiSeq, ll) = self.viterbi(lstm_scores[s], len(sentences[s]))
            viterbiSeqs.append(viterbiSeq)
        return viterbiSeqs

    def forward(self, X, X_char, train=False):
        ###################################################################################################
        #TODO: Implement the forward computation.

        return torch.randn((X.shape[0], X.shape[1], self.NUM_TAGS))  #Random baseline.
        ###################################################################################################



    def print_predictions(self, words, tags):
        Y_pred = self.inference(words)
        for i in range(len(words)):
            print("-----------------------------")
            print(" ".join([f"{words[i][j]}/{Y_pred[i][j]}/{tags[i][j]}" for j in range(len(words[i
            print("Predicted:\t", [Y_pred[i][j] for j in range(len(words[i]))])
            print("Gold:\t\t", tags[i])

    #Need to use Viterbi this time.
    def inference(self, sentences, viterbi=True):
        pred = self.viterbi_batch(sentences)
        return [[i2tag[pred[i][j].item()] for j in range(len(sentences[i]))] for i in range(len(sen

lstm_crf = LSTM_CRFtagger(DIM_EMB=300).cuda()
```

```
# This is a cell for debugging, feel free to change it as you like
print(lstm_crf.conditional_log_likelihood(sentences_dev[0:3], tags_dev[0:3]))
```

```
0
```

```
#CharLSTM-CRF Training

##########################################################################################
# TODO: Add imports if needed.
import tqdm
import os
import subprocess
import random

##########################################################################################

#Get CoNLL evaluation script
os.system('wget https://raw.githubusercontent.com/aritter/twitter_nlp/master/data/annotated/wnut16/

def train_crf_lstm(sentences, tags, lstm):
    ##########################################################################################
    #TODO: initialize optimizer and hyperparameters.
    # optimizer = optim.Adadelta(lstm.parameters(), lr=1.0)

    nEpochs=10
    batchSize = 50
    ##########################################################################################

    for epoch in range(nEpochs):
        totalLoss = 0.0
        lstm.train()

        #Shuffle the sentences
        (sentences_shuffled, tags_shuffled) = shuffle_sentences(sentences, tags)
        for batch in tqdm.notebook.tqdm(range(0, len(sentences), batchSize), leave=False):
            ##########################################################################################
            #TODO: Implement gradient update on a batch of data.

            pass
            ##########################################################################################


        print(f"loss on epoch {epoch} = {totalLoss}")
        lstm.write_predictions(sentences_dev, 'dev_pred')   #Performance on dev set
        print('conlleval:')
        print(subprocess.Popen('paste dev dev_pred | perl conlleval.pl -d "\t"', shell=True, stdout

        if epoch % 10 == 0:
            lstm.eval()
            s = random.sample(range(50), 5)
            lstm.print_predictions([sentences_train[i] for i in s], [tags_train[i] for i in s])   #

crf_lstm = LSTM_CRFtagger(DIM_HID=500, DIM_EMB=300, DIM_CHAR_EMB=30).cuda()
```

```
train_crf_lstm(sentences_train, tags_train, crf_lstm)              #Train on the full dataset
#train_crf_lstm(sentences_train[0:50], tags_train[0:50])           #Train only the first batch (use
```