# HW_11_HW_NER_with_RNN_at_the_Word_and_Char_Level

April 12, 2023

[ ]:
```
# These are all the modules we'll be using later. Make sure you can import them
# before proceeding further.
%matplotlib inline
import collections
import math
import numpy as np
import pandas as pd
import os
import random
import torch
import torch.nn as nn
import zipfile
from matplotlib import pylab
from six.moves import range
from six.moves.urllib.request import urlretrieve
from torch.nn.utils.rnn import pad_sequence
import torch
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
from torchtext.data.functional import to_map_style_dataset
from torchtext.data.utils import get_tokenizer, ngrams_iterator
from torchtext.datasets import DATASETS
from torchtext.utils import download_from_url
from torchtext.vocab import build_vocab_from_iterator
import torch.nn as nn
from torchtext.data.utils import get_tokenizer
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence,␣
↪pad_packed_sequence
import torch.nn.functional as F
from torchtext.vocab import FastText, CharNGram
from itertools import chain

seed = 54321
```

This notebook has you fitting a model for NER that uses both word embeddings and character level embeddings. Each word will get an embedding, and so will each character. In the end, a word's embedding will be the concatenation of the word embedding and the character embedding.

For each sentence, the goal is to identity the NER tag for the word. Most words are marked "O", meaning that the tag is non informative. There are other tags, of the form B-tag and I-tag where tag can be 1 of 4 things. If a $y_t$ is labeled B-tag and next $y_{t+1}$ is the same tag type, then it should be marked I-tag not B-tag since we have the continuation of the same type of tag. NER is used to identify people, organizations, and other entities in long documents.

For this problem, we should technically have a CRF layer on top of the GRU you build. This is because we are predicting a sequence for $y_t$, each $y_t$ is not independent but depends on the one before it (see above). However, since we did not do CRFs, you can just put a softmax layer as the prediction layer, per token you want to predict. If interested, it is easy to modify this HW to get it to work with a CRF, and prediction will improve from 80% to 96%, so it really is important. But you don't need to do that.

[ ]:
```
# Fill in the code below using the hints
FILL_IN = "FILL_IN"
```

### 0.0.1 Download the data

[ ]:
```
url = 'https://github.com/ZihanWangKi/CrossWeigh/raw/master/data/'
dir_name = 'data'
def download_data(url, filename, download_dir, expected_bytes):
    """Download a file if not present, and make sure it's the right size."""

    # Create directories if doesn't exist
    os.makedirs(download_dir, exist_ok=True)

    # If file doesn't exist download
    if not os.path.exists(os.path.join(download_dir,filename)):
        filepath, _ = urlretrieve(url + filename, os.path.
 ↪join(download_dir,filename))
    else:
        filepath = os.path.join(download_dir, filename)

    # Check the file size
    statinfo = os.stat(filepath)
    if statinfo.st_size == expected_bytes:
        print('Found and verified %s' % filepath)
    else:
        print(statinfo.st_size)
        raise Exception(
          'Failed to verify ' + filepath + '. Can you get to it with a browser?
 ↪')

    return filepath

# Filepaths to train/valid/test data
```

```
train_filepath = download_data(url, 'conllpp_train.txt', dir_name, 3283420)
dev_filepath = download_data(url, 'conllpp_dev.txt', dir_name, 827443)
test_filepath = download_data(url, 'conllpp_test.txt', dir_name, 748737)
```

```
Found and verified data/conllpp_train.txt
Found and verified data/conllpp_dev.txt
Found and verified data/conllpp_test.txt
```

[ ]:
```
!head data/conllpp_train.txt
```

```
-DOCSTART- -X- -X- O

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

#### 0.0.2 Read the data

[ ]:
```
def read_data(filename):
    '''
    Read data from a file with given filename
    Returns a list of sentences (each sentence a string),
    and list of ner labels for each string
    '''

    print("Reading data ...")
    # master lists - Holds sentences (list of tokens), ner_labels (for each␣
↪token an NER label)
    sentences, ner_labels = [], []

    # Open the file
    with open(filename,'r',encoding='latin-1') as f:
        # Read each line
        is_sos = True # We record at each line if we are seeing the beginning␣
↪of a sentence

        # Tokens and labels of a single sentence, flushed when encountered a␣
↪new one
        sentence_tokens = []
        sentence_labels = []
        i = 0
        for row in f:
```

```python
            # If we are seeing an empty line or -DOCSTART- that's a new line
            if len(row.strip()) == 0 or row.split(' ')[0] == '-DOCSTART-':
                is_sos = False
            # Otherwise keep capturing tokens and labels
            else:
                is_sos = True
                token, _, _, ner_tag = row.split(' ')
                sentence_tokens.append(token)
                sentence_labels.append(ner_tag.strip())

            # When we reach the end / or reach the beginning of next
            # add the data to the master lists, flush the temporary one
            if not is_sos and len(sentence_tokens)>0:
                sentences.append(' '.join(sentence_tokens))
                ner_labels.append(sentence_labels)
                sentence_tokens, sentence_labels = [], []

    print('\tDone')
    return sentences, ner_labels

# Train data
train_sentences, train_labels = read_data(train_filepath)
# Validation data
valid_sentences, valid_labels = read_data(dev_filepath)
# Test data
test_sentences, test_labels = read_data(test_filepath)

# Print some stats
print(f"Train size: {len(train_labels)}")
print(f"Valid size: {len(valid_labels)}")
print(f"Test size: {len(test_labels)}")

# Print some data
print('\nSample data\n')
for v_sent, v_labels in zip(valid_sentences[:5], valid_labels[:5]):
    print(f"Sentence: {v_sent}")
    print(f"Labels: {v_labels}")
    assert(len(v_sent.split(' ')) == len(v_labels))
    print('\n')
```

```
Reading data …
        Done
Reading data …
        Done
Reading data …
        Done
Train size: 14041
```

```
Valid size: 3250
Test size: 3452

Sample data

Sentence: CRICKET - LEICESTERSHIRE TAKE OVER AT TOP AFTER INNINGS VICTORY .
Labels: ['O', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']


Sentence: LONDON 1996-08-30
Labels: ['B-LOC', 'O']


Sentence: West Indian all-rounder Phil Simmons took four for 38 on Friday as
Leicestershire beat Somerset by an innings and 39 runs in two days to take over
at the head of the county championship .
Labels: ['B-MISC', 'I-MISC', 'O', 'B-PER', 'I-PER', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'B-ORG', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']


Sentence: Their stay on top , though , may be short-lived as title rivals Essex
, Derbyshire and Surrey all closed in on victory while Kent made up for lost
time in their rain-affected match against Nottinghamshire .
Labels: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'B-ORG', 'O', 'B-ORG', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'B-ORG', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-ORG', 'O']


Sentence: After bowling Somerset out for 83 on the opening morning at Grace Road
, Leicestershire extended their first innings by 94 runs before being bowled out
for 296 with England discard Andy Caddick taking three for 83 .
Labels: ['O', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-LOC',
'I-LOC', 'O', 'B-ORG', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'B-LOC', 'O', 'B-PER', 'I-PER', 'O', 'O', 'O', 'O', 'O']
```

[ ]:
```python
assert(len(train_labels) == 14041)
assert(len(valid_labels) == 3250)
assert(len(test_labels) == 3452)
```

[ ]:
```python
# We build these since the basic english tokenizer does get rid of some tokens
 ↪that are useful.
# Lowercase everything to make it easier - all strings should be lowercased
class SentenceTokenizer():
    def __call__(self, sentence):
```

```
        # Return a list of tokens,
        return [i for i in sentence.lower().split(' ')]

class WordTokenizer():
    def __call__(self, word):
        # Return a list of charcters
        return [i for i in word.lower()]
```

[ ]:
```
# Initialize to sentence and word tokenizers
SENTENCE_TOKENIZER = SentenceTokenizer()
WORD_TOKENIZER = WordTokenizer()
```

[ ]:
```
assert(len(WORD_TOKENIZER("this is a sentence")) == 18)
assert(len(SENTENCE_TOKENIZER("this is a sentence")) == 4)
```

[ ]:
```
# Get all the sentences, train, test, and validation
sentences = train_sentences + valid_sentences + test_sentences
# Get all the labels across the above 3 sets
labels = train_labels + valid_labels + test_labels

# For each sentence, tokenize and return the list of tokens via "yield"
def yield_word_tokens(sentences):
    for sentence in sentences:
        yield SENTENCE_TOKENIZER(sentence)
        # A list of word tokens

# Same thing but for characters
def yield_char_tokens(sentences):
    for word_tokens in yield_word_tokens(sentences):
        for word_token in word_tokens:
            yield WORD_TOKENIZER(word_token)
```

[ ]:
```
# Build the word vocabulary
WORD_VOCAB = build_vocab_from_iterator(yield_word_tokens(sentences),␣
 ↪specials=['<pad>','<unk>'])

# Build the char vocabulary
CHAR_VOCAB = build_vocab_from_iterator(yield_char_tokens(sentences),␣
 ↪specials=['<pad>','<unk>'])
```

[ ]:
```
# Example: You should see 4 integer tokens below.
WORD_VOCAB(SENTENCE_TOKENIZER("this is a sentence"))
```

[ ]: [64, 31, 8, 1780]

[ ]:
```
# Example: You should see 4 integer tokens below.
CHAR_VOCAB(WORD_TOKENIZER("Xhis"))
```

[ ]: [42, 12, 6, 8]

[ ]:

[ ]:
```
# Get the word to idx and idx to char dictionaries
wtoi = WORD_VOCAB.get_stoi()
itow = WORD_VOCAB.get_itos()
# Get the char to idx and idx to char dictionaries
ctoi = CHAR_VOCAB.get_stoi()
itoc = CHAR_VOCAB.get_itos()
```

[ ]:

[ ]:
```
assert(len(wtoi) == 26871)
assert(len(ctoi) == 61)
```

[ ]:
```
# You should see 0 and 0 below
WORD_VOCAB['<pad>'], CHAR_VOCAB['<pad>']
```

[ ]: (0, 0)

[ ]:
```
# You should see 1 and 1 below
WORD_VOCAB['<unk>'], CHAR_VOCAB['<unk>']
```

[ ]: (1, 1)

[ ]:
```
# We need to carefully weight all the classes
# We use w(c) = min(freq(l)) / freq(c); lower frequency classes
# So a low class gets a weight that's higher, a higher class a lower weight
# This function need to return 3 dictionaries
def get_label_id_map(labels):
    # Get the unique list of labels
    unique_labels = set([label for label_list in labels for label in␣
↪label_list])
    # Create a dictionary label to idx, starting with idx 0
    ltoi = {label:idx for idx, label in enumerate(unique_labels)}
    # Make a map from idx to label
    itol = {idx:label for label, idx in ltoi.items()}

    itolw = {}

    label_to_count = {label:[label for label_list in labels for label in␣
↪label_list].count(label) for label in unique_labels}

    for label, count in label_to_count.items():
        itolw[ltoi[label]] = min(label_to_count.values()) / count
```

```python
    # Return (ltoi, itol, itolw)
    return ltoi, itol, itolw
```

[ ]:
```python
assert(len(pd.Series(chain(*train_labels)).unique()) == 9)
```

[ ]:
```python
ltoi, itol, itolw = get_label_id_map(train_labels)
```

[ ]:
```python
for l, idx in ltoi.items():
  assert(l == itol[idx])
  assert(idx in itolw)
```

[ ]:
```python
# Look at the weights per tag
itolw
```

[ ]:
```
{0: 0.33595113438045376,
 1: 0.006811025015037328,
 2: 0.25507950530035334,
 3: 0.175,
 4: 0.31182505399568033,
 5: 0.9982713915298185,
 6: 1.0,
 7: 0.18272425249169436,
 8: 0.16176470588235295}
```

[ ]:
```python
ltoi
```

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

[ ]:
```python
assert(min(itolw.values()) == 0.006811025015037328)
```

[ ]:
```python
# Get the weights per class as a tensor of length 9; this will be needed in the
↪loss to give different class elemets a different weight
weights = torch.tensor([itolw[i] for i in range(len(itolw))], dtype=torch.float)
for i, lw in itolw.items():
    assert(weights[i] == lw)
```

[ ]:
```python

```

[ ]:
```python
# Set labels as a series
labels = pd.Series(chain(*train_labels))
```

[ ]:
```python
print(labels)
```

```
0           B-ORG
1               O
2          B-MISC
3               O
4               O
            ...
203616          O
203617      B-ORG
203618          O
203619      B-ORG
203620          O
Length: 203621, dtype: object
```

[ ]:
```python
# Get a count of labels and counts and print this below
label_counts = labels.value_counts()
print(label_counts)
```

```
O         169578
B-LOC       7140
B-PER       6600
B-ORG       6321
I-PER       4528
I-ORG       3704
B-MISC      3438
I-LOC       1157
I-MISC      1155
dtype: int64
```

[ ]:
```python
assert(labels.value_counts().min() == 1155)
```

### 0.0.3 Check for class balance

[ ]:
```python
# Print the value count for each label
print("Training data label counts")
print(pd.Series(chain(*train_labels)).value_counts())

print("\nValidation data label counts")
print(pd.Series(chain(*valid_labels)).value_counts())

print("\nTest data label counts")
print(pd.Series(chain(*test_labels)).value_counts())
```

```
Training data label counts
O         169578
B-LOC       7140
B-PER       6600
B-ORG       6321
I-PER       4528
I-ORG       3704
B-MISC      3438
I-LOC       1157
I-MISC      1155
dtype: int64

Validation data label counts
O         42759
B-PER      1842
B-LOC      1837
B-ORG      1341
I-PER      1307
B-MISC      922
I-ORG       751
I-MISC      346
I-LOC       257
dtype: int64

Test data label counts
O         38143
B-ORG      1714
B-LOC      1645
B-PER      1617
I-PER      1161
I-ORG       881
B-MISC      722
I-LOC       259
I-MISC      252
dtype: int64
```

### 0.0.4 Series length.

[ ]:
```python
# Display the mean sentence length for the training samples
# You should get around 15 mean ...  What about median, 95%, etc?
# .describe applied to a certain series is a good idea ...
print(pd.Series([len(sentence.split(' ')) for sentence in train_sentences]).
 ↪describe())
```

```
count    14041.000000
mean        14.501887
std         11.602756
min          1.000000
```

```
25%           6.000000
50%          10.000000
75%          22.000000
max         113.000000
dtype: float64
```

#### 0.0.5 Parameters

[ ]:
```python
# Size of token embeddings
d_model = 300

# Number of hidden units in the GRU layer
d_hidden = 64

# Number of hidden units in the GRU layer
d_char = 32

# Number of output nodes in the last layer
num_classes = len(itol)

# Number of samples in a batch
BATCH_SIZE = 128

# Number of training epochs.
EPOCHS = 25

# FastText embeddings
FAST_TEXT = FastText("simple")

# Learning rate
LR = 1.0

# Get the weights per class
weight = weights

# Maximum word length; critical for convolutions
MAX_WORD_LENGTH = 12

# The device to run on
# Change this to 'mps' if you are on a mac with MPS
DEVICE = 'mps' if torch.cuda.is_available() else 'cpu'
```

[ ]:
```python
assert(len(train_sentences) // BATCH_SIZE == 109)
```

[ ]:
```python

```

[ ]:
```python
def collate_batch(batch):
    label_list, sentence_list, sentence_lengths = [], [], []
    word_list = []
    # The sentence below is already transformed to int tokens
    for sentence, words, labels in batch:
        # Add the sentence to sentence_list list; you are added a tensor
        sentence_list.append(torch.tensor(sentence))
        # Add the sentence length to the right list
        sentence_lengths.append(torch.tensor(len(sentence)))
        # Add the labels to the right list
        label_list.append(torch.tensor(labels))
        # Add the words to the right list
        word_list.append(torch.tensor(words))

    # Return padded versions of the above; this function processes a batch␣
 ↪remember so we need  to return padded tensors
    # batch_first=True below
    return (
        # (N, L_sentence) with the words
        pad_sequence(sentence_list, batch_first=True,␣
 ↪padding_value=WORD_VOCAB['<pad>']).to(DEVICE),
        # (N, L_sentence) with the labels; set padding_val=-1 to ignore this in␣
 ↪the loss
        pad_sequence(label_list, batch_first=True, padding_value=-1).to(DEVICE),
        sentence_lengths,
        # (N, L_sentence, L_word) where L_word (max) = 12
        # This is padded at the word level, but not sentence level
        pad_sequence(word_list, batch_first=True,␣
 ↪padding_value=CHAR_VOCAB['<pad>']).to(DEVICE)
    )
```

[ ]:

[ ]:
```python
def get_dl(sentences, labels):
    # Maybe sort by the sentences by length so batches have roughly the same␣
 ↪data?

    data = []

    # Note that we need to do our own
    for sentence, labels in zip(sentences, labels):
        word_tokens = SENTENCE_TOKENIZER(sentence)
        # Pass the word tokens through WORD_VOCAB
        int_sentence = WORD_VOCAB(word_tokens)
        int_words = []

        for word_token in word_tokens:
```

```
            # Append to word_token to int_words but tokenized; see below
            if len(word_token) > MAX_WORD_LENGTH:
                int_words.append(CHAR_VOCAB(WORD_TOKENIZER(word_token[:MAX_WORD_LENGTH])))
            else:
                int_words.append(CHAR_VOCAB(WORD_TOKENIZER(word_token)) + [CHAR_VOCAB['<pad>']] * (MAX_WORD_LENGTH - len(word_token)))

            #int_words.append(
                # Taking at most MAX_WORD_LENGTH tokens, get the list of tokens per character
                # Note you need to add a list of variable '<pad>'s to make sure each element you add here has MAX_WORD_LENGTH
                # You are adding to int_words a list of length MAX_WORD_LENGTH representing ints
                # For example, if word_token = "abc", MAX_WORD_LENGTH = 5, this becomes "abc<pad><pad>" -> [1, 2, 3, 0, 0]
            #)

        # Create a list of int tokens for each label, use ltoi
        labels = [ltoi[label] for label in labels]
        # You can remove these later
        assert(len(int_sentence) == len(labels))
        for int_word in int_words:
          assert(len(int_word) == MAX_WORD_LENGTH)
        data.append([int_sentence, int_words, labels])

    # Return a DataLoader with batch_size=BATCH_SIZE, shuffle=True, and collate_fn=collate_batch
    return DataLoader(data, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_batch)

train_dl = get_dl(train_sentences, train_labels)
valid_dl = get_dl(valid_sentences, valid_labels)
test_dl = get_dl(test_sentences, test_labels)
```

[ ]:

```
assert(len(train_dl) == 110)
```

[ ]:

```
m = nn.Conv1d(32,32,kernel_size=5)
input = torch.randn(1, 32, 12)
output = m(input)
p = nn.MaxPool1d(8)
output = p(output)
output.shape
len(CHAR_VOCAB)
```

[ ]: 61

```python
class GRUNERModel(nn.Module):
    def __init__(
        self,
        num_class,
        d_model,
        d_hidden,
        initialize = True,
        fine_tune_embeddings = True,
        use_conv_embeddings = True,
    ):

        super(GRUNERModel, self).__init__()
        self.vocab_size = len(WORD_VOCAB)
        self.d_model = d_model
        self.d_hidden = d_hidden
        self.d_char = 32
        self.kernel = 5
        self.max_word_length = MAX_WORD_LENGTH
        self.use_conv_embeddings = use_conv_embeddings

        if self.use_conv_embeddings:
            # 12 - 5 + 1 = 8
            # Input data will be (N * L_sentence, D_char, L_word)
            # L_word = 12 here
            # We want output to be d_char by 8 for self.kernel=5
            self.conv = nn.Conv1d(in_channels=self.d_char, out_channels=self.
↪d_char, kernel_size=self.kernel)
            # Will results in (N * L_sentence, D_char, 8) data.
            # H_char is 32.
            # Will result is (32, 1) vector for each word.
            # Define a max pooling layer so the above holds
            self.max_pool = nn.MaxPool1d(kernel_size=self.max_word_length -␣
↪self.kernel + 1)

        # Create a word embedding layer with len(WORD_VOCAB) vectors;␣
↪padding_idx=0 and set the length to 300 unless initialize=False in which␣
↪case it is d_model
        self.embedding = nn.Embedding(self.vocab_size, self.d_model,␣
↪padding_idx=0)

        # Create a char embedding layer with len(CHAR_VOCAB) vectors; same as␣
↪above but don't initialize with anything, make them d_char dimension
        self.char_embedding = nn.Embedding(len(CHAR_VOCAB), self.d_char,␣
↪padding_idx=0)
```

```python
        # Put in logic here to initialize the word embeddings or not with␣
↪FAST_TEXT
        # Make sure you map a word to its corrent word embedding in FAST_TEXT
        if initialize:
            for i in range(self.vocab_size):
                self.embedding.weight.data[i] = FAST_TEXT[WORD_VOCAB.
↪get_itos()[i]]
        else:
            self.init_weights()

        # If fine_time_embeddings=False, turn off gradients for the word␣
↪embeddings, they will be static
        if not fine_tune_embeddings:
            self.embedding.weight.requires_grad = False

        # Initialize a bidirectional GRU
        # input is d_model + d_char (some other logic might be needed here if␣
↪d_model != 300 given the above, but you can ignore this)
        # Make batch_first=True; use self.d_hidden as the hidden dimension
        self.rnn = nn.GRU(self.d_model+self.d_char, self.d_hidden,␣
↪batch_first=True, bidirectional=True)

        # Bidirectional GRU; so, we go from 2 * d_hidden to num_class via a␣
↪linear layer
        self.fc = nn.Linear(2 * self.d_hidden, num_class)

        # Note: for drop out + ReLu, order does not matters
        # Use 0.3 for the dropout probability
        self.dropout = nn.Sequential(
            nn.ReLU(),
            nn.Dropout(0.3)
        )

    def init_weights(self):
        # Initialize the word embedding layer with uniform random variables␣
↪between (-initrange, initrange)
        initrange = 0.5
        # Add logic for the char embeddings also
        self.embedding.weight.data.uniform_(-initrange, initrange)

    # N = batch_size,
    # L_sentence = sequence length
    # D_word = word embedding length
    # D_char = char embedding length
    # Hout = hidden dimenson from bidirectional GRU
    # C = number of classes
```

```python
    def forward(self, sentences, lengths, words):
        # (N, L_sentence, D_word)
        embedded_sentences = self.embedding(sentences.int())

        if self.use_conv_embeddings:
            # (N, L_sentence, L_word, D_char)
            # Pass words through the char_embeddings to get them
            embedded_words = self.char_embedding(words.int())

            N, L_sentence, L_word, D_char = embedded_words.shape

            # (N * L_sentence, L_word, D_char)
            # Reshape to the above dimension
            embedded_words = embedded_words.view(N * L_sentence, L_word, D_char)

            # (N * L_sentence, D_char, L_word)
            # Do something to get the above dimension
            embedded_words = embedded_words.permute(0, 2, 1)

            # 12 - 4, since kernel size is 5
            # (N * L_sentence, D_char, L_word - kernel_size + 1 )
            # Apply conv
            embedded_words = self.conv(embedded_words)

            # (N * L_sentence, D_char, 1)
            # Apply max pool and squeeze the result
            embedded_words = self.max_pool(embedded_words).squeeze(-1)

            # (N, L_sentence, D_char)
            # Reshape
            embedded_words = embedded_words.view(N, L_sentence, D_char)

            # (N, L_sentence, D_char + D_word)
            # Concatenate a word's word vector and the character based word␣
↪vector together
            embedded_sentences = torch.cat((embedded_sentences,␣
↪embedded_words), dim=-1)

        # This is a key for efficient computation.
        # Pack the padded embeddings. Magic

        embedded_sentences = pack_padded_sequence(embedded_sentences, lengths,␣
↪batch_first=True, enforce_sorted=False)


        # (N * L_sentence sort of, Hout)
        logits, _ = self.rnn(embedded_sentences)
```

```
         # (N, L_sentence, Hout)
         # Apply pad_packed_sequence to logits
        logits, _ = pad_packed_sequence(logits, batch_first=True)

        # (N, L_sentence, C)
        # Apply self.fc
        logits = self.fc(logits)

        return logits
```

[ ]:

```

```

[ ]:

```
# Used so we do not include padding indices.
# Also, give different weights to different classes to account for class␣
 ↪imbalance.
# Use ignore_index=-1 since this is the "pad" index for labels
criterion = nn.CrossEntropyLoss(weight=weights,ignore_index=-1)

# Define the model; use initialize=True, fine_tune=True, use_conv=True
# I'm unsure if all these decisions are optimal, the point of this exercise is␣
 ↪to make conv embeddings work
model = GRUNERModel(initialize=True, fine_tune_embeddings=True,␣
 ↪num_class=num_classes, d_model=d_model, d_hidden=d_hidden)

optimizer = torch.optim.SGD(model.parameters(), lr=LR)

scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)
```

[ ]:

```

```

[ ]:

```
from re import escape
def train(dl, model, optimizer, criterion, epoch):
    model.train()
    total_acc, total_count = 0, 0
    total_loss, total_batches = 0.0, 0.0
    log_interval = 50

    for idx, (sentences, labels, lengths, words) in enumerate(dl):
        optimizer.zero_grad()

        logits = model(sentences, lengths, words)

        # Get the loss
        N, L, _ = logits.shape
        # Reshape to the right dimensons, and get the loss
        logits = logits.view(N * L, -1)
```

```
        labels = labels.view(N * L)
        loss = criterion(input=logits, target=labels)

        total_loss += loss.item()
        total_batches += 1

        # Do back propagation
        loss.backward()

        # Clip the gradients at 0.1
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)

        # Do an optimization step
        optimizer.step()

        # Put in eval to get accuracies as below
        model.eval()

        # Get the mask and then find out the predictions for things that are␣
↪NOT masked
        masks = labels != -1
        total_acc += (logits.argmax(dim=-1)[masks] == labels[masks]).sum().
↪item()
        total_count += masks.sum().item()

        model.train()
        if idx % log_interval == 0 and idx > 0:
            print(
                "| epoch {:3d} | {:5d}/{:5d} batches "
                "| accuracy {:8.3f} "
                "| loss {:8.3f}".format(
                    epoch,
                    idx,
                    len(dl),
                    total_acc / total_count,
                    total_loss / total_batches
                )
            )
            total_acc, total_count = 0, 0
            total_loss, total_batches  = 0.0, 0.0
```

[ ]:
```
def evaluate(dl, model):
    model.eval()
    total_acc, total_count = 0, 0
    total_loss, total_batches = 0.0, 0.0

    with torch.no_grad():
```

```
        for idx, (sentences, labels, lengths, words) in enumerate(dl):
            logits = model(sentences, lengths, words)
            N, L, _ = logits.shape
            # Very similar to train - reshape, get the accuracy for unmaked␣
↪labels, etc
            logits = logits.view(N * L, -1)
            labels = labels.view(N * L)
            loss = criterion(input=logits, target=labels)

            total_loss += loss.item()
            total_batches += 1

            masks = labels != -1
            total_acc += (logits.argmax(dim=-1)[masks] == labels[masks]).sum().
↪item()
            total_count += masks.sum().item()

    return total_acc / total_count, total_loss / total_batches
```

[ ]:
```
from time import time
import time

for epoch in range(1, EPOCHS + 1):
    epoch_start_time = time.time()
    train(train_dl, model, optimizer, criterion, epoch)
    accu_val, loss_val = evaluate(valid_dl, model)
    scheduler.step()
    print("-" * 59)
    print(
        "| end of epoch {:3d} | time: {:5.2f}s "
        "| valid accuracy {:8.3f} "
        "| valid loss {:8.3f} ".format(
            epoch,
            time.time() - epoch_start_time,
            accu_val,
            loss_val
        )
    )
    print("-" * 59)

print("Checking the results of test dataset.")
accu_test, loss_test = evaluate(test_dl, model)
print("test accuracy {:8.3f} | test loss {:8.3f}".format(accu_test, loss_test))
```

```
| epoch   1 |    50/  110 batches | accuracy    0.315 | loss    1.970
| epoch   1 |   100/  110 batches | accuracy    0.694 | loss    1.472
-----------------------------------------------------------
```

```
| end of epoch   1 | time: 52.29s | valid accuracy    0.777 | valid loss
1.222
-----------------------------------------------------------
| epoch   2 |    50/  110 batches | accuracy    0.759 | loss    1.209
| epoch   2 |   100/  110 batches | accuracy    0.758 | loss    1.172
-----------------------------------------------------------
| end of epoch   2 | time: 48.73s | valid accuracy    0.772 | valid loss
1.158
-----------------------------------------------------------
| epoch   3 |    50/  110 batches | accuracy    0.761 | loss    1.165
| epoch   3 |   100/  110 batches | accuracy    0.764 | loss    1.147
-----------------------------------------------------------
| end of epoch   3 | time: 48.26s | valid accuracy    0.773 | valid loss
1.151
-----------------------------------------------------------
| epoch   4 |    50/  110 batches | accuracy    0.761 | loss    1.157
| epoch   4 |   100/  110 batches | accuracy    0.763 | loss    1.150
-----------------------------------------------------------
| end of epoch   4 | time: 50.36s | valid accuracy    0.773 | valid loss
1.150
-----------------------------------------------------------
| epoch   5 |    50/  110 batches | accuracy    0.761 | loss    1.145
| epoch   5 |   100/  110 batches | accuracy    0.763 | loss    1.170
-----------------------------------------------------------
| end of epoch   5 | time: 48.56s | valid accuracy    0.773 | valid loss
1.144
-----------------------------------------------------------
| epoch   6 |    50/  110 batches | accuracy    0.762 | loss    1.174
| epoch   6 |   100/  110 batches | accuracy    0.763 | loss    1.141
-----------------------------------------------------------
| end of epoch   6 | time: 48.59s | valid accuracy    0.773 | valid loss
1.149
-----------------------------------------------------------
| epoch   7 |    50/  110 batches | accuracy    0.763 | loss    1.149
| epoch   7 |   100/  110 batches | accuracy    0.763 | loss    1.149
-----------------------------------------------------------
| end of epoch   7 | time: 49.48s | valid accuracy    0.773 | valid loss
1.143
-----------------------------------------------------------
| epoch   8 |    50/  110 batches | accuracy    0.762 | loss    1.152
| epoch   8 |   100/  110 batches | accuracy    0.763 | loss    1.159
-----------------------------------------------------------
| end of epoch   8 | time: 50.05s | valid accuracy    0.773 | valid loss
1.151
-----------------------------------------------------------
| epoch   9 |    50/  110 batches | accuracy    0.762 | loss    1.158
| epoch   9 |   100/  110 batches | accuracy    0.763 | loss    1.145
-----------------------------------------------------------
```

```
| end of epoch   9 | time: 48.23s | valid accuracy    0.773 | valid loss
1.149
-----------------------------------------------------------
| epoch  10 |    50/  110 batches | accuracy    0.764 | loss    1.154
| epoch  10 |   100/  110 batches | accuracy    0.760 | loss    1.150
-----------------------------------------------------------
| end of epoch  10 | time: 47.81s | valid accuracy    0.773 | valid loss
1.149
-----------------------------------------------------------
| epoch  11 |    50/  110 batches | accuracy    0.762 | loss    1.154
| epoch  11 |   100/  110 batches | accuracy    0.763 | loss    1.154
-----------------------------------------------------------
| end of epoch  11 | time: 48.98s | valid accuracy    0.773 | valid loss
1.147
-----------------------------------------------------------
| epoch  12 |    50/  110 batches | accuracy    0.762 | loss    1.152
| epoch  12 |   100/  110 batches | accuracy    0.763 | loss    1.158
-----------------------------------------------------------
| end of epoch  12 | time: 46.74s | valid accuracy    0.773 | valid loss
1.144
-----------------------------------------------------------
| epoch  13 |    50/  110 batches | accuracy    0.763 | loss    1.155
| epoch  13 |   100/  110 batches | accuracy    0.761 | loss    1.157
-----------------------------------------------------------
| end of epoch  13 | time: 47.67s | valid accuracy    0.773 | valid loss
1.148
-----------------------------------------------------------
| epoch  14 |    50/  110 batches | accuracy    0.763 | loss    1.149
| epoch  14 |   100/  110 batches | accuracy    0.762 | loss    1.155
-----------------------------------------------------------
| end of epoch  14 | time: 48.07s | valid accuracy    0.773 | valid loss
1.152
-----------------------------------------------------------
| epoch  15 |    50/  110 batches | accuracy    0.763 | loss    1.149
| epoch  15 |   100/  110 batches | accuracy    0.762 | loss    1.161
-----------------------------------------------------------
| end of epoch  15 | time: 48.28s | valid accuracy    0.773 | valid loss
1.149
-----------------------------------------------------------
| epoch  16 |    50/  110 batches | accuracy    0.759 | loss    1.154
| epoch  16 |   100/  110 batches | accuracy    0.766 | loss    1.152
-----------------------------------------------------------
| end of epoch  16 | time: 47.24s | valid accuracy    0.773 | valid loss
1.148
-----------------------------------------------------------
| epoch  17 |    50/  110 batches | accuracy    0.760 | loss    1.155
| epoch  17 |   100/  110 batches | accuracy    0.764 | loss    1.156
-----------------------------------------------------------
```

```
| end of epoch  17 | time: 47.64s | valid accuracy    0.773 | valid loss
1.145
-----------------------------------------------------------
| epoch  18 |    50/  110 batches | accuracy    0.765 | loss    1.172
| epoch  18 |   100/  110 batches | accuracy    0.761 | loss    1.138
-----------------------------------------------------------
| end of epoch  18 | time: 47.10s | valid accuracy    0.773 | valid loss
1.148
-----------------------------------------------------------
| epoch  19 |    50/  110 batches | accuracy    0.764 | loss    1.155
| epoch  19 |   100/  110 batches | accuracy    0.761 | loss    1.148
-----------------------------------------------------------
| end of epoch  19 | time: 46.88s | valid accuracy    0.773 | valid loss
1.149
-----------------------------------------------------------
| epoch  20 |    50/  110 batches | accuracy    0.762 | loss    1.155
| epoch  20 |   100/  110 batches | accuracy    0.763 | loss    1.155
-----------------------------------------------------------
| end of epoch  20 | time: 49.34s | valid accuracy    0.773 | valid loss
1.149
-----------------------------------------------------------
| epoch  21 |    50/  110 batches | accuracy    0.763 | loss    1.140
| epoch  21 |   100/  110 batches | accuracy    0.762 | loss    1.165
-----------------------------------------------------------
| end of epoch  21 | time: 46.82s | valid accuracy    0.773 | valid loss
1.155
-----------------------------------------------------------
| epoch  22 |    50/  110 batches | accuracy    0.761 | loss    1.155
| epoch  22 |   100/  110 batches | accuracy    0.763 | loss    1.154
-----------------------------------------------------------
| end of epoch  22 | time: 46.95s | valid accuracy    0.773 | valid loss
1.146
-----------------------------------------------------------
| epoch  23 |    50/  110 batches | accuracy    0.763 | loss    1.158
| epoch  23 |   100/  110 batches | accuracy    0.761 | loss    1.148
-----------------------------------------------------------
| end of epoch  23 | time: 49.65s | valid accuracy    0.773 | valid loss
1.151
-----------------------------------------------------------
| epoch  24 |    50/  110 batches | accuracy    0.763 | loss    1.154
| epoch  24 |   100/  110 batches | accuracy    0.763 | loss    1.149
-----------------------------------------------------------
| end of epoch  24 | time: 48.21s | valid accuracy    0.773 | valid loss
1.149
-----------------------------------------------------------
| epoch  25 |    50/  110 batches | accuracy    0.760 | loss    1.146
| epoch  25 |   100/  110 batches | accuracy    0.765 | loss    1.159
-----------------------------------------------------------
```

```
| end of epoch  25 | time: 47.39s | valid accuracy    0.773 | valid loss
1.150
-----------------------------------------------------------
Checking the results of test dataset.
test accuracy    0.754 | test loss    1.157
```