Homework 4: Neural Sequence Labeling

Due March 4, 2020 at 11:59PM

In this homework, you will be implementing, training, and evaluating an LSTM for part-of-speech tagging using the PyTorch library.

Before beginning, please switch your Colab session to a GPU runtime

Go to Runtime > Change runtime type > Hardware accelerator > GPU

Setup

```
# import libraries
import torch
import numpy as np
import torch.nn as nn
from torch.nn.utils.rnn import pad_sequence, pad_packed_sequence, pack_padded_sequence

# if this cell prints "Running on cpu", you must switch runtime environments
# go to Runtime > Change runtime type > Hardware accelerator > GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Running on {}".format(device))
```

```
Running on cuda
```

Download & Load Pretrained Embeddings

In this assignment, we will be using GloVe pretrained word embeddings. You can read more about GloVe here: https://nlp.stanford.edu/projects/glove/

Note: this section will take *several minutes*, since the embedding files are large. Files in Colab may be cached between sessions, so you may or may not need to redownload the files each time you reconnect.

```
# download pretrained word embeddings
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip glove*.zip
```

```
--2020-03-06 00:30:40--  http://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2020-03-06 00:30:45--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2020-03-06 00:30:46--  http://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  1.92MB/s    in 6m 30s

2020-03-06 00:37:16 (2.11 MB/s) - ‘glove.6B.zip’ saved [862182613/862182613]

Archive:  glove.6B.zip
  inflating: glove.6B.50d.txt
  inflating: glove.6B.100d.txt
  inflating: glove.6B.200d.txt
  inflating: glove.6B.300d.txt
```

```
def read_embeddings(filename, vocab_size=10000):
  """
  Utility function, loads in the `vocab_size` most common embeddings from `filename`

  Arguments:
```

```
  - filename:     path to file
                  automatically infers correct embedding dimension from filename
  - vocab_size:   maximum number of embeddings to load

  Returns
  - embeddings:   torch.FloatTensor matrix of size (vocab_size x word_embedding_dim)
  - vocab:        dictionary mapping word (str) to index (int) in embedding matrix
  """

  # get the embedding size from the first embedding
  with open(filename, encoding="utf-8") as file:
    word_embedding_dim = len(file.readline().split(" ")) - 1

  vocab = {}

  embeddings = np.zeros((vocab_size, word_embedding_dim))

  with open(filename, encoding="utf-8") as file:
    for idx, line in enumerate(file):

      if idx + 2 >= vocab_size:
        break

      cols = line.rstrip().split(" ")
      val = np.array(cols[1:])
      word = cols[0]
      embeddings[idx + 2] = val
      vocab[word] = idx + 2

  # a FloatTensor is a multidimensional matrix
  # that contains 32-bit floats in every entry
  # https://pytorch.org/docs/stable/tensors.html
  return torch.FloatTensor(embeddings), vocab
```

Running the cell below lists all the files in the current directory.

```
!ls -lh
```

```
total 2.9G
-rw-rw-r-- 1 root root 332M Aug  4  2014 glove.6B.100d.txt
-rw-rw-r-- 1 root root 662M Aug  4  2014 glove.6B.200d.txt
-rw-rw-r-- 1 root root 990M Aug 27  2014 glove.6B.300d.txt
-rw-rw-r-- 1 root root 164M Aug  4  2014 glove.6B.50d.txt
-rw-r--r-- 1 root root 823M Oct 25  2015 glove.6B.zip
drwxr-xr-x 1 root root 4.0K Mar  3 18:11 sample_data
```

You should see several embedding files, which are all formatted as

```
glove.6B.<emb_dim>d.txt
```

Each `txt` file contains `emb_dim` dimensional embeddings for 400,000 unique, uncased words. The script below loads the `vocab_size` most common words from the embedding file into a matrix we can give to our model. All other words will later be mapped to the `UNKNOWN` embedding.

```
# this loads the 10,000 most common word 50-dimensional embeddings
vocab_size = 10000
embeddings, vocab = read_embeddings('glove.6B.50d.txt', vocab_size)
```

Part 1: Batching the data

Implement the `get_batches` function in the `Dataset` class below.

Please make sure that

- Your implementation is self-contained. That is, all helper functions and variables are defined within `get_batches`.
- Your implementation can handle variable batch sizes. You may not assume that the value with always be 32

```
class Dataset():
```

```
class Dataset():
  def __init__(self, filename, is_labeled):
    self.is_labeled = is_labeled
    # if the file is not labeled, the Dataset has no tags (see read_data)
    if is_labeled:
      self.sentences, self.tags = self.read_data(filename, is_labeled)
    else:
      self.sentences = self.read_data(filename, is_labeled)
      self.tags = None

  def read_data(self, filename, is_labeled):
    """
    Utility function, loads text file into a list of sentence and tag strings

    Arguments:
    - filename:     path to file
    - is_labeled:   whether the file contains tags for each word or not
        > if True, we assume each line is formatted as "<word>\t<tag>\n"
        > if False, we assume each line is formatted as "<word>\n"

    Returns:
    - sentences:    a list of sentences, where each sentence is a list
                    words (strings)

    if is_labeled=True, also returns
    - tags:         a list of tags for each sentence, where tags[i] contains
                    a list of tags (strings) that correspond to the words in
                    sentences[i]
    """
    sentences = []
    tags = []

    current_sentence = []
    current_tags = []

    with open(filename, encoding='utf8') as f:
      # iterate over the lines in the file
      for line in f:
        if len(line) == 0:
          continue
        if line == '\n':
          if len(current_sentence) != 0:
            sentences.append(current_sentence)
            tags.append(current_tags)

          current_sentence = []
          current_tags = []
        else:
          if is_labeled:
            columns = line.rstrip().split('\t')
            word = columns[0].lower()
            tag = columns[1]

            current_sentence.append(word)
            current_tags.append(tag)
          else:
            column = line.rstrip().split('\t')
            word = column[0].lower()
            current_sentence.append(word)

      if is_labeled:
        return sentences, tags
      else:
        return sentences

  def get_batches(self, batch_size, vocab, tagset):
    """

    Batches the data into mini-batches of size `batch_size`

    Arguments:
    - batch_size:       the desired output batch size
    - vocab:            a dictionary mapping word strings to indices
    - tagset:           a dictionary mapping tag strings to indices
```

```
Outputs:

if is_labeled=True:
- batched_word_indices:     a list of matrices of dimension (batch_size x max_seq_len)
- batched_tag_indices:      a list of matrices of dimension (batch_size x max_seq_len)
- batched_lengths:          a list of arrays of length (batch_size)

if is_labeled=False:
- batched_word_indices:     a list of matrices of dimension (batch_size x max_seq_len)
- batched_lengths:          a list of arrays of length (batch_size)


Description:

This function partitions the data into batches of size batch_size. If the number
of sentences in the document is not an even multiple of batch_size, the final batch
will contain the remaining elements. For example, if there are 82 sentences in the
dataset and batch_size=32, we return a list containing two batches of size 32
and one final batch of size 18.

batched_word_indices[b] is a (batch_size x max_seq_len) matrix of integers,
containing index representations for sentences in the b-th batch in the document.
The `vocab` dictionary provides the correct mapping from word strings to indices.
If a word is not in the vocabulary, it gets mapped to UNKNOWN_INDEX (1).
`max_seq_len` is the maximum sentence length among the sentences in the current batch,
which will vary between different batches. All sentences shorter than max_seq_len
should be padded on the right with PAD_INDEX (0).

If the document is labeled, we also batch the document's tags. Analogous to
batched_word_indices, batched_tag_indices[b] contains the index representation
for the tags corresponding to the sentences in the b-th batch  in the document.
The `tagset` dictionary provides the correct mapping from tag strings to indicies.
All tag lists shorter than `max_seq_len` are padded with IGNORE_TAG_INDEX (-100).

batched_lengths[b] is a vector of length (batch_size). batched_lengths[b][i]
contains the original sentence length *before* padding for the i-th sentence
in the currrent batch.

"""
PAD_INDEX = 0             # reserved for padding words
UNKNOWN_INDEX = 1         # reserved for unknown words
IGNORE_TAG_INDEX = -100   # reserved for padding tags

# randomly shuffle the data
np.random.seed(159) # DON'T CHANGE THIS
shuffle = np.random.permutation(range(len(self.sentences)))

sentences = [self.sentences[i] for i in shuffle]
if self.is_labeled:
  tags = [self.tags[i] for i in shuffle]
else:
  tags = None

batched_word_indices = []
batched_tag_indices = []
batched_lengths = []

#############################
#        YOUR CODE HERE     #
#############################

# partition into batches of size batch_size
for i in range(0, len(sentences), batch_size):
  if (i + batch_size) > len(sentences):
    upper_lim = len(sentences)
  else:
    upper_lim = i + batch_size
  batch = sentences[i: upper_lim]
  if self.is_labeled:
    batch_tags = tags[i: upper_lim]

  max_seq_len = max([len(sent) for sent in batch])
```

```
      # add default padding values
      sent_indices = np.ones((len(batch), max_seq_len)) * PAD_INDEX
      tag_indices = np.ones((len(batch), max_seq_len)) * IGNORE_TAG_INDEX
      lengths = np.ones((len(batch), ))
      for b in range(len(batch)):
        sent = batch[b]
        if self.is_labeled:
          sent_tags = batch_tags[b]
        lengths[b] = len(sent)

        # update word and tags
        for w in range(len(sent)):
          word = sent[w]
          if (word in vocab):
            sent_indices[b, w] = vocab[word]
          else:
            sent_indices[b, w] = UNKNOWN_INDEX

          if self.is_labeled:
            tag = sent_tags[w]
            tag_indices[b, w] = tagset[tag]

      batched_word_indices.append(sent_indices)
      batched_tag_indices.append(tag_indices)
      batched_lengths.append(lengths)

    #############################
    #       DO NOT MODIFY       #
    #############################
    if self.is_labeled:
      return batched_word_indices, batched_tag_indices, batched_lengths
    else:
      return batched_word_indices, batched_lengths


def read_tagset(tag_file):
  """
  Utility function, loads tag file into a dictionary from tag string to tag index

  Arguments:
  - tag_file:   file location of the tagset

  Outputs:
  - tagset:     a dictionary mapping tag strings (e.g. "VB") to a unique index
  """
  tagset = {}
  with open(tag_file, encoding='utf8') as f:
    for line in f:
      columns = line.rstrip().split('\t')
      tag = columns[0]
      tag_id = int(columns[1])
      tagset[tag] = tag_id

  return tagset
```

The cells below download the data files and construct the corresponding `Dataset` objects.

```
%%capture
!wget https://raw.githubusercontent.com/dbamman/nlp20/master/HW_4/pos.train
!wget https://raw.githubusercontent.com/dbamman/nlp20/master/HW_4/pos.dev
!wget https://raw.githubusercontent.com/dbamman/nlp20/master/HW_4/pos.test
!wget https://raw.githubusercontent.com/dbamman/nlp20/master/HW_4/pos.tagset


# read the files
tagset = read_tagset('pos.tagset')
train_dataset = Dataset('pos.train', is_labeled=True)
dev_dataset = Dataset('pos.dev', is_labeled=True)
test_dataset = Dataset('pos.test', is_labeled=False)

BATCH_SIZE = 32

# these should run without errors if implemented correctly
train_batch_idx, train_batch_tags, train_batch_lens = train_dataset.get_batches(BATCH_SIZE, vocab, tagset)
```

```
dev_batch_idx, dev_batch_tags, dev_batch_lens = dev_dataset.get_batches(BATCH_SIZE, vocab, tagset)
test_batch_idx, test_batch_lens = test_dataset.get_batches(BATCH_SIZE, vocab, tagset)
```

Part 2: Evaluation

Next, we will implement utility functions that will later be used to assess our model's perfomance.

Please make sure that

- Your implementation is self-contained. That is, keep all helper functions or variables inside of your function.
- Your implementation does not import any additional libraries. You will not receive credit if you do.

```
# The accuracy function has been implemented for you

def accuracy(true, pred):
  """
  Arguments:
  - true:       a list of true label values (integers)
  - pred:       a list of predicted label values (integers)

  Output:
  - accuracy:   the prediction accuracy
  """
  true = np.array(true)
  pred = np.array(pred)

  num_correct = sum(true == pred)
  num_total = len(true)
  return num_correct / num_total


def confusion_matrix(true, pred, num_tags):
  """
  Arguments:
  - true:       a list of true label values (integers)
  - pred:       a list of predicted label values (integers)
  - num_tags:   the number of possible tags
                true and pred will both contain integers between
                0 and num_tags - 1 (inclusive)

  Output:
  - confusion_matrix:   a (num_tags x num_tags) matrix of integers

  confusion_matrix[i][j] = # predictions where true label
  was i and predicted label was j

  """

  confusion_matrix = np.zeros((num_tags, num_tags))

  #############################
  #       YOUR CODE HERE      #
  #############################
  # for i in true:
  #   for j in pred:
  #     confusion_matrix[i][j] += 1

  for i, j in zip(true, pred):
    confusion_matrix[i][j] += 1

  return confusion_matrix


def precision(true, pred, num_tags):
  """
  Arguments:
  - true:       a list of true label values (integers)
  - pred:       a list of predicted label values (integers)
  - num_tags:   the number of possible tags
                true and pred will both contain integers between
                0 and num_tags - 1 (inclusive)
```

```
    Output:
    - precision:  an array of length num_tags, where precision[i]
                  gives the precision of class i

    Hints:  the confusion matrix may be useful
            be careful about zero division
    """

    precision = np.zeros(num_tags)

    #############################
    #       YOUR CODE HERE      #
    #############################

    matrix = confusion_matrix(true, pred, num_tags)

    for i in range(num_tags): # loop over all tag values
      tp = matrix[i][i]
      fp = sum([matrix[j][i] for j in range(num_tags) if j != i])
      if tp + fp == 0: #account for zero division error
        precision[i] = 0
      else:
        precision[i] = tp / (tp + fp)

    return precision


  def recall(true, pred, num_tags):
    """
    Arguments:
    - true:       a list of true label values (integers)
    - pred:       a list of predicted label values (integers)
    - num_tags:   the number of possible tags
                  true and pred will both contain integers between
                  0 and num_tags - 1 (inclusive)

    Output:
    - recall:     an array of length num_tags, where recall[i]
                  gives the recall of class i

    Hints:  the confusion matrix may be useful
            be careful about zero division
    """

    """
    YOUR CODE HERE
    """
    recall = np.zeros(num_tags)

    #############################
    #       YOUR CODE HERE      #
    #############################

    matrix = confusion_matrix(true, pred, num_tags)

    for i in range(num_tags): # loop over all tag values
      tp = matrix[i][i]
      fn = sum([matrix[i][j] for j in range(num_tags) if j != i])
      if tp + fn == 0: #account for zero division error
        recall[i] = 0
      else:
        recall[i] = tp / (tp + fn)

    return recall


  def f1_score(true, pred, num_tags):
    """
    Arguments:
    - true:       a list of true label values (integers)
    - pred:       a list of predicted label values (integers)
    - num_tags:   the number of possible tags
                  true and pred will both contain integers between
                  0 and num_tags - 1 (inclusive)
```

```
    Output:
    - f1:          an array of length num_tags, where f1[i]
                   gives the recall of class i
    """
    f1 = np.zeros(num_tags)

    #############################
    #       YOUR CODE HERE      #
    #############################

    rec_arr = recall(true, pred, num_tags)
    pre_arr = precision(true, pred, num_tags)
    for i in range(num_tags): # loop over all tag values
      if (pre_arr[i] + rec_arr[i]) == 0:
        f1[i] = 0
      else:
        f1[i] = (2 * pre_arr[i] * rec_arr[i]) / (pre_arr[i] + rec_arr[i])

    return f1
```

Part 3: Building the model

Fill in the blanks in `LSTMTagger`'s `__init__` function. If you get stuck, you can reference PyTorch's [torch.nn documentation]() or [this official tutorial]() on LSTM sequence labeling.

```
class LSTMTagger(nn.Module):
  """
  An LSTM model for sequence labeling

  Initialization Arguments:
  - embeddings:   a matrix of size (vocab_size, emb_dim)
                  containing pretrained embedding weights
  - hidden_dim:   the LSTM's hidden layer size
  - tagset_size:  the number of possible output tags

  """
  def __init__(self, embeddings, hidden_dim, tagset_size):
    super().__init__()

    self.hidden_dim = hidden_dim
    self.num_labels = tagset_size

    #############################
    #       YOUR CODE HERE      #
    #############################

    # Initialize a PyTorch embeddings layer using the pretrained embedding weights
    # print(embeddings.shape[0])
    self.embeddings = nn.Embedding(embeddings.shape[0], embeddings.shape[1])
    self.embeddings.weight.data.copy_(embeddings)
    # self.embeddings = nn.Embedding(embeddings.size(0), embeddings.size(1))
    # self.embeddings = nn.Embedding.from_pretrained(embeddings, freeze=False)

    # Initialize an LSTM layer
    self.lstm = nn.LSTM(embeddings.shape[1], hidden_dim)
    # self.lstm = nn.LSTM(embeddings.size(1), hidden_dim)

    # Initialize a single feedforward layer
    self.hidden2tag = nn.Linear(hidden_dim, tagset_size)

  def forward(self, indices, lengths):
    """
    Runs a batched sequence through the model and returns output logits

    Arguments:
    - indices:  a matrix of size (batch_size x max_seq_len)
                containing the word indices of sentences in the batch
    - lengths:  a vector of size (batch_size) containing the
                original lengths of the sequences before padding

    Output:
```

```
    - logits:   a matrix of size (batch_size x max_seq_len x num_tags)
                gives a score to each possible tag for each word
                in each sentence
    """
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # cast arrays as PyTorch data types and move to GPU memory
    indices = torch.LongTensor(indices).to(device)
    lengths = torch.LongTensor(lengths).to(device)

    # convert word indices to word embeddings
    embeddings = self.embeddings(indices)

    # pack/pad handles variable length sequence batching
    # see here if you're curious: https://gist.github.com/HarshTrivedi/f4e7293e941b17d19058f6fb90ab0fec
    packed_input_embs = pack_padded_sequence(embeddings, lengths, batch_first=True, enforce_sorted=False)
    # run input through LSTM layer
    packed_output, _ = self.lstm(packed_input_embs)
    # unpack sequences into original format
    padded_output, output_lengths = pad_packed_sequence(packed_output, batch_first=True)

    logits = self.hidden2tag(padded_output)
    return logits

  def run_training(self, train_dataset, dev_dataset, batch_size, vocab, tagset,
                       lr=5e-4, num_epochs=100, eval_every=5):
    """
    Trains the model on the training data with a learning rate of lr
    for num_epochs. Evaluates the model on the dev data eval_every epochs.

    Arguments:
    - train_dataset:  Dataset object containing the training data
    - dev_dataset:    Dataset object containing the dev data
    - batch_size:     batch size for train/dev data
    - vocab:          a dictionary mapping word strings to indices
    - tagset:         a dictionary mapping tag strings to indices
    - lr:             learning rate
    - num_epochs:     number of epochs to train for
    - eval_every:     evaluation is run eval_every epochs
    """
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    if str(device) == 'cpu':
      print("Training only supported in GPU environment")
      return

    # clear unreferenced data/models from GPU memory
    torch.cuda.empty_cache()
    # move model to GPU memory
    self.to(device)

    # set the optimizer (Adam) and loss function (CrossEnt)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    loss_function = nn.CrossEntropyLoss(ignore_index=-100)

    # batch training and dev data
    train_batch_idx, train_batch_tags, train_batch_lens = train_dataset.get_batches(BATCH_SIZE, vocab, tagset)
    dev_batch_idx, dev_batch_tags, dev_batch_lens = dev_dataset.get_batches(BATCH_SIZE, vocab, tagset)

    print("**** TRAINING *****")
    for i in range(num_epochs):
      # sets the model in train mode
      self.train()

      total_loss = 0
      for b in range(len(train_batch_idx)):
        # compute the logits
        logits = model.forward(train_batch_idx[b], train_batch_lens[b])
        # move labels to GPU memory
        labels = torch.LongTensor(train_batch_tags[b]).to(device)
        # compute the loss with respect to true labels
        loss = loss_function(logits.view(-1, len(tagset)), labels.view(-1))
        total_loss += loss
        # propagate gradients backward
```

```
        loss.backward()
        optimizer.step()
        # set model gradients to zero before performing next forward pass
        self.zero_grad()

      print("Epoch {} | Loss: {}".format(i, total_loss))

      if (i + 1) % eval_every == 0:
        print("**** EVALUATION *****")
        # sets the model in evaluate mode (no gradients)
        self.eval()
        # compute dev f1 score
        acc, true, pred = self.evaluate(dev_batch_idx, dev_batch_lens, dev_batch_tags, tagset)
        print("Dev Accuracy: {}".format(acc))
        print("**********************")

  def evaluate(self, batched_sentences, batched_lengths, batched_labels, tagset):
    """
    Evaluate the model's predictions on the provided dataset.

    Arguments:
    - batched_sentences:  a list of matrices, each of size (batch_size x max_seq_len),
                          containing the word indices of sentences in the batch
    - batched_lengths:    a list of vectors, each of size (batch_size), containing the
                          original lengths of the sequences before padding
    - batched_labels:     a list of matrices, each of size (batch_size x max_seq_len),
                          containing the tag indices corresponding to sentences in the batch
    - num_tags:           the number of possible output tags

    Output:
    - accuracy:           the model's prediction accuracy
    - all_true_labels:    a flattened list of all true labels
    - all_predictions:    a flattened list of all of the model's corresponding predictions

    """

    all_true_labels = []
    all_predictions = []

    for b in range(len(batched_sentences)):
      logits = self.forward(batched_sentences[b], batched_lengths[b])
      batch_predictions = torch.argmax(logits, dim=-1).cpu().numpy()

      batch_size, _ = batched_sentences[b].shape

      for i in range(batch_size):
        tags = batched_labels[b][i]
        preds = batch_predictions[i]

        seq_len = int(batched_lengths[b][i])
        for j in range(seq_len):
          all_predictions.append(int(preds[j]))
          all_true_labels.append(int(tags[j]))


    acc = accuracy(all_true_labels, all_predictions)

    return acc, all_true_labels, all_predictions

def set_seed(seed):
  """
  Sets random seeds and sets model in deterministic
  training mode. Ensures reproducible results
  """
  torch.manual_seed(seed)
  torch.backends.cudnn.deterministic = True
  torch.backends.cudnn.benchmark = False
  np.random.seed(seed)
```

Training the model

Run the cells below to train your model. If all of the previous sections are implemented correctly, you should see

- the loss decreasing consistently for every epoch
- the dev accuracy increasing until convergence around ~0.88

The staff solution achieves an accuracy of 0.880 after 25 epochs.

```
# sets the random seed — DO NOT change this
# this ensures deterministic results that are comparable with the staff values
set_seed(159)

HIDDEN_SIZE = 64
# intialize a new LSTMTagger model
model = LSTMTagger(embeddings, HIDDEN_SIZE, len(tagset))
# train the model
model.run_training(train_dataset, dev_dataset, BATCH_SIZE, vocab, tagset,
                   lr=5e-4, num_epochs=25, eval_every=5)
```

```
**** TRAINING *****
Epoch 0 | Loss: 966.6653442382812
Epoch 1 | Loss: 417.92913818359375
Epoch 2 | Loss: 267.7360534667969
Epoch 3 | Loss: 209.37423706054688
Epoch 4 | Loss: 179.8599395751953
**** EVALUATION *****
Dev Accuracy: 0.8589779280174985
**********************
Epoch 5 | Loss: 161.93316650390625
Epoch 6 | Loss: 149.59909057617188
Epoch 7 | Loss: 140.41824340820312
Epoch 8 | Loss: 133.19635009765625
Epoch 9 | Loss: 127.2813491821289
**** EVALUATION *****
Dev Accuracy: 0.8754424338834759
**********************
Epoch 10 | Loss: 122.29289245605469
Epoch 11 | Loss: 117.97274780273438
Epoch 12 | Loss: 114.15171813964844
Epoch 13 | Loss: 110.71771240234375
Epoch 14 | Loss: 107.54832458496094
**** EVALUATION *****
Dev Accuracy: 0.8791409823026447
**********************
Epoch 15 | Loss: 104.69400787353516
Epoch 16 | Loss: 102.0001449584961
Epoch 17 | Loss: 99.43143463134766
Epoch 18 | Loss: 97.04694366455078
Epoch 19 | Loss: 94.7793960571289
**** EVALUATION *****
Dev Accuracy: 0.8795784450188905
**********************
Epoch 20 | Loss: 92.61573028564453
Epoch 21 | Loss: 90.54859924316406
Epoch 22 | Loss: 88.55851745605469
Epoch 23 | Loss: 86.857421875
Epoch 24 | Loss: 84.86511993408203
**** EVALUATION *****
Dev Accuracy: 0.8791807516404851
**********************
```

Once the model is trained, run the cells below to print the precision, recall, and F_1 score per class.

```
def eval_per_class(model, dataset, vocab, tagset):
  """
  Prints precision, recall, and F1 for each class in the tagset
  """
  # batch the data
  batched_idx, batched_tags, batched_lens = dev_dataset.get_batches(BATCH_SIZE, vocab, tagset)
  # compute idx --> tag from tag --> idx
  reverse_tagset = {v: k for k,v in tagset.items()}
  # evaluate model on hold-out set
  acc, true, pred = model.evaluate(batched_idx, batched_lens, batched_tags, tagset)
  true = np.array(true)
  pred = np.array(pred)
```

```
  pr = precision(true, pred, len(tagset))
  re = recall(true, pred, len(tagset))
  f1 = f1_score(true, pred, len(tagset))

  for idx, tag in reverse_tagset.items():
    print("**********************")
    print("TAG: {}".format(tag))
    num_pred = np.sum(pred == idx)
    num_true = np.sum(true == idx)
    print("({} pred, {} true)".format(num_pred, num_true))

    print("PRECISION: \t{:.3f}".format(pr[idx]))
    print("RECALL: \t{:.3f}".format(re[idx]))
    print("F1 SCORE: \t{:.3f}".format(f1[idx]))

eval_per_class(model, dev_dataset, vocab, tagset)
```

```
***********************
TAG: $
(13 pred, 14 true)
PRECISION:      1.000
RECALL:         0.929
F1 SCORE:       0.963
***********************
TAG: ''
(96 pred, 88 true)
PRECISION:      0.854
RECALL:         0.932
F1 SCORE:       0.891
***********************
TAG: ,
(967 pred, 936 true)
PRECISION:      0.939
RECALL:         0.970
F1 SCORE:       0.954
***********************
TAG: -LRB-
(107 pred, 117 true)
PRECISION:      0.953
RECALL:         0.872
F1 SCORE:       0.911
***********************
TAG: -RRB-
(123 pred, 120 true)
PRECISION:      0.919
RECALL:         0.942
F1 SCORE:       0.930
***********************
TAG: .
(1461 pred, 1503 true)
PRECISION:      0.988
RECALL:         0.961
F1 SCORE:       0.974
***********************
TAG: :
(98 pred, 106 true)
PRECISION:      0.980
RECALL:         0.906
F1 SCORE:       0.941
***********************
TAG: ADD
(12 pred, 81 true)
PRECISION:      0.167
RECALL:         0.025
F1 SCORE:       0.043
***********************
TAG: AFX
(0 pred, 4 true)
PRECISION:      0.000
RECALL:         0.000
F1 SCORE:       0.000
***********************
TAG: CC
(781 pred, 781 true)
PRECISION:      0.988
RECALL:         0.988
F1 SCORE:       0.988
***********************
TAG: CD
(329 pred, 378 true)
PRECISION:      0.845
RECALL:         0.735
F1 SCORE:       0.786
***********************
TAG: DT
(1970 pred, 1943 true)
PRECISION:      0.968
RECALL:         0.981
F1 SCORE:       0.975
***********************
TAG: EX
(49 pred, 56 true)
PRECISION:      0.939
RECALL:         0.821
F1 SCORE:       0.876
***********************
TAG: FW
(2 pred, 30 true)
PRECISION:      0.500
```

```
RECALL:         0.033
F1 SCORE:       0.062
***********************
TAG: GW
(21 pred, 32 true)
PRECISION:      0.000
RECALL:         0.000
F1 SCORE:       0.000
***********************
TAG: HYPH
(77 pred, 95 true)
PRECISION:      0.844
RECALL:         0.684
F1 SCORE:       0.756
***********************
TAG: IN
(2443 pred, 2353 true)
PRECISION:      0.909
RECALL:         0.944
F1 SCORE:       0.926
***********************
TAG: JJ
(1677 pred, 1655 true)
PRECISION:      0.830
RECALL:         0.841
F1 SCORE:       0.836
***********************
TAG: JJR
(39 pred, 47 true)
PRECISION:      0.615
RECALL:         0.511
F1 SCORE:       0.558
***********************
TAG: JJS
(71 pred, 84 true)
PRECISION:      0.859
RECALL:         0.726
F1 SCORE:       0.787
***********************
TAG: LS
(8 pred, 5 true)
PRECISION:      0.375
RECALL:         0.600
F1 SCORE:       0.462
***********************
TAG: MD
(354 pred, 358 true)
PRECISION:      0.980
RECALL:         0.969
F1 SCORE:       0.975
***********************
TAG: NFP
(31 pred, 60 true)
PRECISION:      0.774
RECALL:         0.400
F1 SCORE:       0.527
***********************
TAG: NN
(3521 pred, 3336 true)
PRECISION:      0.817
RECALL:         0.862
F1 SCORE:       0.839
***********************
TAG: NNP
(2058 pred, 1816 true)
PRECISION:      0.656
RECALL:         0.744
F1 SCORE:       0.697
***********************
TAG: NNPS
(24 pred, 63 true)
PRECISION:      0.792
RECALL:         0.302
F1 SCORE:       0.437
***********************
TAG: NNS
(936 pred, 929 true)
PRECISION:      0.807
RECALL:         0.813
F1 SCORE:       0.810
***********************
TAG: PDT
(5 pred, 21 true)
```

```
(5 pred, 21 true)
PRECISION:      0.400
RECALL:         0.095
F1 SCORE:       0.154
***********************
TAG: POS
(87 pred, 84 true)
PRECISION:      0.943
RECALL:         0.976
F1 SCORE:       0.959
***********************
TAG: PRP
(1494 pred, 1487 true)
PRECISION:      0.988
RECALL:         0.993
F1 SCORE:       0.990
***********************
TAG: PRP$
(308 pred, 315 true)
PRECISION:      0.990
RECALL:         0.968
F1 SCORE:       0.979
***********************
TAG: RB
(1175 pred, 1292 true)
PRECISION:      0.903
RECALL:         0.821
F1 SCORE:       0.860
***********************
TAG: RBR
(34 pred, 22 true)
PRECISION:      0.353
RECALL:         0.545
F1 SCORE:       0.429
***********************
TAG: RBS
(22 pred, 20 true)
PRECISION:      0.591
RECALL:         0.650
F1 SCORE:       0.619
***********************
TAG: RP
(61 pred, 75 true)
PRECISION:      0.689
RECALL:         0.560
F1 SCORE:       0.618
***********************
TAG: SYM
(7 pred, 20 true)
PRECISION:      0.429
RECALL:         0.150
F1 SCORE:       0.222
***********************
TAG: TO
(349 pred, 359 true)
PRECISION:      0.854
RECALL:         0.830
F1 SCORE:       0.842
***********************
TAG: UH
(63 pred, 116 true)
PRECISION:      0.889
RECALL:         0.483
F1 SCORE:       0.626
***********************
TAG: VB
(1076 pred, 1122 true)
PRECISION:      0.932
RECALL:         0.894
F1 SCORE:       0.913
***********************
TAG: VBD
(517 pred, 520 true)
PRECISION:      0.872
RECALL:         0.867
F1 SCORE:       0.870
***********************
TAG: VBG
(347 pred, 384 true)
PRECISION:      0.853
RECALL:         0.771
F1 SCORE:       0.810
***********************
```

```
TAG: VBN
(496 pred, 476 true)
PRECISION:      0.808
RECALL:         0.842
F1 SCORE:       0.825
***********************
TAG: VBP
(776 pred, 771 true)
PRECISION:      0.916
RECALL:         0.922
F1 SCORE:       0.919
***********************
TAG: VBZ
(643 pred, 643 true)
PRECISION:      0.960
RECALL:         0.960
F1 SCORE:       0.960
***********************
TAG: WDT
(99 pred, 106 true)
PRECISION:      0.788
RECALL:         0.736
F1 SCORE:       0.761
***********************
TAG: WP
(118 pred, 113 true)
PRECISION:      0.898
RECALL:         0.938
F1 SCORE:       0.918
***********************
TAG: WP$
(0 pred, 2 true)
PRECISION:      0.000
RECALL:         0.000
F1 SCORE:       0.000
***********************
TAG: WRB
(112 pred, 113 true)
PRECISION:      1.000
RECALL:         0.991
F1 SCORE:       0.996
***********************
TAG: XX
(0 pred, 3 true)
PRECISION:      0.000
RECALL:         0.000
F1 SCORE:       0.000
***********************
TAG: ``
(88 pred, 91 true)
PRECISION:      0.920
RECALL:         0.890
F1 SCORE:       0.905
```

Part 4: Model Exploration

Congratulations, you've just trained a neural network!

Now, improve the `LSTMTagger` model and implementing the `init` function in the `FancyTagger` class below.

- Feel free to replace the `forward` function inherited from `LSTMTagger` if you need to, but it should not be necessary to receive full credit. Credit will be awarded based on the performance on a holdout test set.

- Do not modify any of the cells above when completing part 4. Instead, insert cells below if you need to perform any additional computations.
- You are allowed to use any function in `torch.nn`. You are **not** allowed to import any libraries or use implementations copied from the internet.

Before submitting, please describe your modifications below.

I decided to increase the size of the data by increasing the dimensionality of word-embeddings and the number of common words selected. Instead of loading the 10,000 most common word 50-dimensional embeddings, I load the 500,000 most common word 300-dimensional embeddings.

I also choose to use a bi-directional LSTM and dropout layer (p=0.6) for every layer except the output layer.

All these modifications increased the accuracy of my model from ~0.88 to ~0.928.

```
# # # this loads the 10,000 most common word 50-dimensional embeddings
# vocab_size = 10000
# embeddings, vocab = read_embeddings('glove.6B.50d.txt', vocab_size)

# this loads the 500,000 most common word 300-dimensional embeddings
vocab_size = 500000
embeddings, vocab = read_embeddings('glove.6B.300d.txt', vocab_size)
```

```
class FancyTagger(LSTMTagger):
  """
  An improved neural model for sequence labeling

  Starter code from LSTMTagger has already been provided, but
  feel free to change the init and forward function internals
  if your model design requires it (though this is not necessary
  to receive full credit).

  You may use any component in torch.nn. You may NOT
  import any additional libraries/modules.

  """
  def __init__(self, embeddings, hidden_dim, tagset_size):
    # initializes the parent LSTMTagger class
    # inherits forward, evaluate, and run_training methods
    super().__init__(embeddings, hidden_dim, tagset_size)

    self.hidden_dim = hidden_dim
    self.num_labels = tagset_size

    #############################
    #       YOUR CODE HERE      #
    #############################
  # #Idea: Make the model bidirectional and add dropout
  # #use bigger data and vocab size
  #   self.embeddings =

  #   self.lstm = nn.LSTM()

  # ------------------------------------------------------------------------------------------

    # Initialize a PyTorch embeddings layer using the pretrained embedding weights
    # print(embeddings.shape[0])
    self.embeddings = nn.Embedding(embeddings.shape[0], embeddings.shape[1])
    self.embeddings.weight.data.copy_(embeddings)
    # self.embeddings = nn.Embedding(embeddings.size(0), embeddings.size(1))

    # Initialize an LSTM layer
    self.lstm = nn.LSTM(embeddings.shape[1], hidden_dim, num_layers=2, bidirectional=True, dropout=0.6)
    # self.lstm = nn.LSTM(embeddings.size(1), hidden_dim)

    # Initialize a single feedforward layer
    self.hidden2tag = nn.Linear(hidden_dim*2, tagset_size)
```

Run the training script below to train the `FancyTagger` model. Again, feel free to adjust any hyperparameters if necessary.

```
model = FancyTagger(embeddings, HIDDEN_SIZE, len(tagset))
print(model)
model.run_training(train_dataset, dev_dataset, BATCH_SIZE, vocab, tagset,
                   lr=5e-4, num_epochs=15, eval_every=5)
```

```
FancyTagger(
  (embeddings): Embedding(500000, 300)
  (lstm): LSTM(300, 64, num_layers=2, dropout=0.6, bidirectional=True)
  (hidden2tag): Linear(in_features=128, out_features=50, bias=True)
)
**** TRAINING *****
Epoch 0 | Loss: 772.9916381835938
Epoch 1 | Loss: 222.7566680908203
Epoch 2 | Loss: 129.33763122558594
Epoch 3 | Loss: 97.31932067871094
Epoch 4 | Loss: 79.34996795654297
**** EVALUATION *****
Dev Accuracy: 0.920381785643269
**********************
Epoch 5 | Loss: 67.7364273071289
Epoch 6 | Loss: 59.1616325378418
Epoch 7 | Loss: 52.34418869018555
Epoch 8 | Loss: 47.127315521240234
Epoch 9 | Loss: 42.47673034667969
**** EVALUATION *****
Dev Accuracy: 0.9270232650626367
**********************
Epoch 10 | Loss: 38.620243072509766
Epoch 11 | Loss: 35.0311279296875
Epoch 12 | Loss: 32.10194778442383
Epoch 13 | Loss: 29.4144287109375
Epoch 14 | Loss: 26.3944091796875
**** EVALUATION *****
Dev Accuracy: 0.9283754225492146
**********************
```

Save Predictions

When you are satisfied with your `FancyTagger`'s performance on the dev set, run the cell below to write your predictions on the test set to a text file.

You can download `predictions.txt` by going to **View > Table of Contents > Files**

Please submit this `predictions.txt` file to Gradescope.

```
model.eval()
```

```
FancyTagger(
  (embeddings): Embedding(500000, 300)
  (lstm): LSTM(300, 64, num_layers=2, dropout=0.6, bidirectional=True)
  (hidden2tag): Linear(in_features=128, out_features=50, bias=True)
)
```

```
assert isinstance(model, FancyTagger), 'Please assign your FancyTagger to a variable named model'

BATCH_SIZE = 32
test_batch_idx, test_batch_lens = test_dataset.get_batches(BATCH_SIZE, vocab, tagset)

predictions = []

for b in range(len(test_batch_idx)):
  logits = model.forward(test_batch_idx[b], test_batch_lens[b])
  batch_predictions = torch.argmax(logits, dim=-1).cpu().numpy()

  batch_size, _ = test_batch_idx[b].shape

  for i in range(batch_size):
    preds = batch_predictions[i]

    seq_len = int(test_batch_lens[b][i])
    for j in range(seq_len):
      predictions.append(int(preds[j]))
```

```
with open('predictions.txt', 'w') as f:
  for p in predictions:
    f.write(str(p) + "\n")
```