NLP Homework 4 Programming Assignment

In this assignment, we will train and evaluate a neural model to tag the parts of speech in a sentence. improvements to the model to test its performance.

We will be using English text from the Wall Street Journal, marked with POS tags such as `NNP` (proper

Building a POS Tagger

Setup

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import random

random.seed(1)
```

Preparing Data

We collect the data in the following cell from the `train.txt` and `test.txt` files.
For `train.txt`, we read the word and tag sequences for each sentence. We then create an 80-20 trai evaluation purpose.

Finally, we are interested in our accuracy on `test.txt`, so we prepare test data from this file.

```
def load_tag_data(tag_file):
    all_sentences = []
    all_tags = []
    sent = []
    tags = []
    with open(tag_file, 'r') as f:
        for line in f:
            if line.strip() == "":
                all_sentences.append(sent)
                all_tags.append(tags)
                sent = []
                tags = []
            else:
                word, tag, _ = line.strip().split()
                sent.append(word)
                tags.append(tag)
    return all_sentences, all_tags
```

```
def load_txt_data(txt_file):
    all_sentences = []
    sent = []
    with open(txt_file, 'r') as f:
        for line in f:
            if(line.strip() == ""):
                all_sentences.append(sent)
                sent = []
            else:
                word = line.strip()
                sent.append(word)
    return all_sentences

train_sentences, train_tags = load_tag_data('train.txt')
test_sentences = load_txt_data('test.txt')

unique_tags = set([tag for tag_seq in train_tags for tag in tag_seq])

# Create train-val split from train data
train_val_data = list(zip(train_sentences, train_tags))
random.shuffle(train_val_data)
split = int(0.8 * len(train_val_data))
training_data = train_val_data[:split]
val_data = train_val_data[split:]

print("Train Data: ", len(training_data))
print("Val Data: ", len(val_data))
print("Test Data: ", len(test_sentences))
print("Total tags: ", len(unique_tags))
```

```
Train Data:  7148
Val Data:  1788
Test Data:  2012
Total tags:  44
```

Word-to-Index and Tag-to-Index mapping

In order to work with text in Tensor format, we need to map each word to an index.

```
word_to_idx = {}
for sent in train_sentences:
    for word in sent:
        if word not in word_to_idx:
            word_to_idx[word] = len(word_to_idx)

for sent in test_sentences:
    for word in sent:
        if word not in word_to_idx:
            word_to_idx[word] = len(word_to_idx)
```

```
tag_to_idx = {}
for tag in unique_tags:
    if tag not in tag_to_idx:
        tag_to_idx[tag] = len(tag_to_idx)

idx_to_tag = {}
for tag in tag_to_idx:
    idx_to_tag[tag_to_idx[tag]] = tag
print(train_sentences)
print(word_to_idx)
print("Total tags", len(tag_to_idx))
print("Vocab size", len(word_to_idx))
```

```
[['Confidence', 'in', 'the', 'pound', 'is', 'widely', 'expected', 'to', 'take', '
{'Confidence': 0, 'in': 1, 'the': 2, 'pound': 3, 'is': 4, 'widely': 5, 'expected'
Total tags 44
Vocab size 21589
```

```
def prepare_sequence(sent, idx_mapping):
    idxs = [idx_mapping[word] for word in sent]
    return torch.tensor(idxs, dtype=torch.long)
```

Set up model

We will build and train a Basic POS Tagger which is an LSTM model to tag the parts of speech in a giv

First we need to define some default hyperparameters.

```
EMBEDDING_DIM = 200
HIDDEN_DIM = 200
LEARNING_RATE = 0.1
LSTM_LAYERS = 2
DROPOUT = 1
EPOCHS = 2
```

Define Model

The model takes as input a sentence as a tensor in the index space. This sentence is then converted t maps to its word embedding. The word embeddings is learned as part of the model training process.

These word embeddings act as input to the LSTM which produces a hidden state. This hidden state is produces the probability distribution for the tags of every word. The model will output the tag with the

```
class BasicPOSTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(BasicPOSTagger, self).__init__()
```

```
        ###########################################################################
        # TODO: Define and initialize anything needed for the forward pass.
        # You are required to create a model with:
        # an embedding layer: that maps words to the embedding space
        # an LSTM layer: that takes word embeddings as input and outputs hidden stat
        # a Linear layer: maps from hidden state space to tag space
        ###########################################################################
        self.hidden_dim = hidden_dim
        self.word_embedding = nn.Embedding(vocab_size,embedding_dim)
        self.lstm = nn.LSTM(embedding_dim,hidden_dim)
        self.hiddenTotag = nn.Linear(hidden_dim, tagset_size)
        ###########################################################################
        #                             END OF YOUR CODE
        ###########################################################################

    def forward(self, sentence):
        tag_scores = None
        ###########################################################################
        # TODO: Implement the forward pass.
        # Given a tokenized index-mapped sentence as the argument,
        # compute the corresponding scores for tags
        # returns:: tag_scores (Tensor)
        ###########################################################################
        ebdngs = self.word_embedding(sentence)
        lstm_output, _ = self.lstm(ebdngs.view(len(sentence),1,-1))
        tag_seq = self.hiddenTotag(lstm_output.view(len(sentence),-1))
        tag_scores  = F.log_softmax(tag_seq, dim=1)

        ###########################################################################
        #                             END OF YOUR CODE
        ###########################################################################
        return tag_scores
```

Training

We define train and evaluate procedures that allow us to train our model using our created train-val sp

```
def train(epoch, model, loss_function, optimizer):
    train_loss = 0
    train_examples = 0
    for sentence, tags in training_data:
        ###########################################################################
        # TODO: Implement the training loop
        # Hint: you can use the prepare_sequence method for creating index mappings
        # for sentences. Find the gradient with respect to the loss and update the
        # model parameters using the optimizer.
        ###########################################################################
        model.zero_grad()
        sen_input = prepare_sequence(sentence,word_to_idx)
        targets = prepare_sequence(tags,tag_to_idx)
        #print(targets.shape)
```

```
        #print(targets.shape)
        tag_scores = model(sen_input)
        #print(tag_scores.shape)
        loss = loss_function(tag_scores,targets)
        train_examples+=1
        loss.backward()
        optimizer.step()
        train_loss += loss
        train_examples += len(targets)
        ##########################################################################
        #                              END OF YOUR CODE
        ##########################################################################

    avg_train_loss = train_loss / train_examples
    avg_val_loss, val_accuracy = evaluate(model, loss_function, optimizer)

    print("Epoch: {}/{}\tAvg Train Loss: {:.4f}\tAvg Val Loss: {:.4f}\t Val Accuracy
                                                                   EPOCHS,
                                                                   avg_train_loss
                                                                   avg_val_loss,
                                                                   val_accuracy))

def evaluate(model, loss_function, optimizer):
  # returns:: avg_val_loss (float)
  # returns:: val_accuracy (float)
    val_loss = 0
    correct = 0
    val_examples = 0
    with torch.no_grad():
        for sentence, tags in val_data:
            ######################################################################
            # TODO: Implement the evaluate loop
            # Find the average validation loss along with the validation accuracy.
            # Hint: To find the accuracy, argmax of tag predictions can be used.
            ######################################################################
            # model.zero_grad()
            sen_input = prepare_sequence(sentence, word_to_idx)
            targets = prepare_sequence(tags,tag_to_idx)
            tag_scores = model(sen_input)
            _, indices = torch.max(tag_scores,1)
            val_loss += loss_function(tag_scores,targets)
            correct += torch.sum(indices == torch.LongTensor(targets))
            val_examples += len(targets)

            ######################################################################
            #                              END OF YOUR CODE
            ######################################################################
    val_accuracy = 100. * correct / val_examples
    avg_val_loss = val_loss / val_examples
    return avg_val_loss, val_accuracy
```

```
###############################################################################
# TODO: Initialize the model, optimizer and the loss function
###############################################################################
model = BasicPOSTagger(EMBEDDING_DIM,HIDDEN_DIM,len(word_to_idx.keys()),len(tag_to_i
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(),lr=LEARNING_RATE)
###############################################################################
#                              END OF YOUR CODE                               #
###############################################################################
for epoch in range(1, EPOCHS + 1):
    train(epoch, model, loss_function, optimizer)
```

```
Epoch: 1/2      Avg Train Loss: 0.0381  Avg Val Loss: 0.0232     Val Accuracy: 85
Epoch: 2/2      Avg Train Loss: 0.0155  Avg Val Loss: 0.0166     Val Accuracy: 90
```

You should get a performance of **at least 80%** on the validation set for the BasicPOSTagger.

Let us now write a method to save our predictions for the test set.

```
def test():
    val_loss = 0
    correct = 0
    val_examples = 0
    predicted_tags = []
    with torch.no_grad():
        for sentence in test_sentences:
            ###########################################################################
            # TODO: Implement the test loop
            # This method saves the predicted tags for the sentences in the test set
            # The tags are first added to a list which is then written to a file for
            # submission. An empty string is added after every sequence of tags
            # corresponding to a sentence to add a newline following file formatting
            # convention, as has been done already.
            ###########################################################################
            sen_in = prepare_sequence(sentence,word_to_idx)
            tag_scores = model(sen_in)
            _, indexes = torch.max(tag_scores,1)
            for i in range(len(indexes)):
              for key, value in tag_to_idx.items():
                if indexes[i] == value:
                    predicted_tags.append(key)


            ###########################################################################
            #                              END OF YOUR CODE
            ###########################################################################
            predicted_tags.append("")
    print(predicted_tags)
    with open('test_labels.txt', 'w+') as f:
        for item in predicted_tags:
            f.write("%s\n" % item)
```

```
            f.write("%s\n" % item)


test()
```

```
['NNP', 'NNP', 'NNP', 'POS', 'NNP', 'NN', 'VBD', 'PRP', 'VBD', 'DT', 'JJ', 'NN',
```

Test accuracy

Evaluate your performance on the test data by submitting test_labels.txt generated by the method abo

The test accuracy I got for BasicLSTM is 90.4

Imitate the above method to generate prediction for validation data. Create lists of words, tags predict tags.

Use these lists to carry out error analysis to find the top-10 types of errors made by the model.

```
#############################################################################
# TODO: Generate predictions from val data
# Create lists of words, tags predicted by the model and ground truth tags.
#############################################################################
def generate_predictions(model, test_sentences):
    # returns:: word_list (str list)
    # returns:: model_tags (str list)
    # returns:: gt_tags (str list)
    # Your code here
    word_list = []
    model_tags = []
    gt_tags = []
    for sentence,tag in test_sentences:
      sen_in = prepare_sequence(sentence,word_to_idx)
      # gt_tags.append(tag)
      # word_list.append(sentence)
      tag_scores = model(sen_in)
      _, indexes = torch.max(tag_scores,1)
      for i in range(len(indexes)):
        for key, value in tag_to_idx.items():
          if indexes[i] == value:
            model_tags.append(key)
      word_list.extend(sentence)
      gt_tags.extend(tag)

          # model_tags.append()
          #gt_tags.append(key)
        #word_list.append(sentence[i])
    return word_list, model_tags, gt_tags
```

```
###########################################################################
# TODO: Carry out error analysis
# From those lists collected from the above method, find the
# top-10 tuples of (model_tag, ground_truth_tag, frequency, example words)
# sorted by frequency
###########################################################################
def error_analysis(word_list, model_tags, gt_tags):
    # returns: errors (list of tuples)
    # Your code here
    import collections
    d = collections.defaultdict(list)


    temp_wl=[]
    temp_ml=[]
    temp_gt=[]
    errors = []
    for word,pred,ground_truth in zip(word_list,model_tags,gt_tags):
      # to find the error we need to find where the prediction is not equal to grour
      if pred != ground_truth:
        temp_wl.append(word)
        temp_ml.append(pred)
        temp_gt.append(ground_truth)
        d[(pred,ground_truth)].append(word)


    combined_pred_gt = zip(temp_ml,temp_gt)
    cc = collections.Counter(combined_pred_gt)

    for (pred,ground_truth), count in cc.most_common(10):
      freq = float(count/sum(cc.values()))
      errors.append((pred,ground_truth,freq,d[(pred,ground_truth)][:10]))






    return errors


word_list,model_tags,gt_tags = generate_predictions(model,val_data)
print(len(word_list),len(model_tags),len(gt_tags))
errors = error_analysis(word_list,model_tags,gt_tags)
for error in errors:
  print(error)
```

```
41851 41851 41851
('JJ', 'NN', 0.05558086560364465, ['democratization', 'sign', 'file', 'weekly', '
('NN', 'JJ', 0.05558086560364465, ['literary', 'insolvent', 'electric', 'Miami-ba
('NN', 'NNP', 0.05079726651480638, ['Sununu', 'SNET', 'Cordis', 'Lyneses', 'Infor
('JJ', 'NNP', 0.04031890660592255, ['Extension', 'Education', 'EC', 'Rent', 'J.P.
('NN', 'NNS', 0.03895216400911162, ['twists', 'foundations', 'piers', 'surprises'
('NNP', 'NN', 0.03530751708428246, ['collapse', 'Circulation', 'corporation', 'ai
('NNP', 'JJ', 0.025968109339407745, ['historic', 'fetal-tissue', 'Western', 'Smal
('NNS', 'NN', 0.023917995444191344, ['province', 'rice', 'verge', 'estuarian', 'l
('NNP', 'NNS', 0.023917995444191344, ['villagers', 'firemen', 'tickets', 'flights
('NNS', 'NNP', 0.0234624145785877, ['Disneyland', 'Donald', 'Quantum', 'JAL', 'CP
```

Error analysis

Report your findings here.
What kinds of errors did the model make and why do you think it made them?

The model makes error in predicting the NN tag properly. If we look at the first row of the error, we car
proper tag NN for the words, instead it predicted JJ tag for those words. I can see that the model prec
some it has identified as adjective which i think is correct like "black". I think it might be because of th
their position in the sentence is what is confusing the model, as our model is unidirectional.

Define a Character Level POS Tagger

We can use the character-level information present to augment our word embeddings. Words that enc
information about their POS tags. To incorporate this information, we can run a character level LSTM
characters, each mapped to character-index space) to create a character-level representation of the w
concatenated with the word embedding (as in the BasicPOSTagger) to create a new word embedding

```
# Create char to index mapping
char_to_idx = {}
unique_chars = set()
MAX_WORD_LEN = 0

for sent in train_sentences:
    for word in sent:
        for c in word:
            unique_chars.add(c)
        if len(word) > MAX_WORD_LEN:
            MAX_WORD_LEN = len(word)

for c in unique_chars:
    char_to_idx[c] = len(char_to_idx)
char_to_idx[' '] = len(char_to_idx)

# New Hyperparameters
```

```
# New Hyperparameters
EMBEDDING_DIM = 12
HIDDEN_DIM = 12
LEARNING_RATE = 0.01
LSTM_LAYERS = 4
DROPOUT = 2
EPOCHS = 10
CHAR_EMBEDDING_DIM = 3
CHAR_HIDDEN_DIM = 3


class CharPOSTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, char_embedding_dim,
                 char_hidden_dim, char_size, vocab_size, tagset_size):
        super(CharPOSTagger, self).__init__()
        ##########################################################################
        # TODO: Define and initialize anything needed for the forward pass.
        # You are required to create a model with:
        # an embedding layer: that maps words to the embedding space
        # an char level LSTM: that finds the character level embedding for a word
        # an LSTM layer: that takes the combined embeddings as input and outputs hic
        # a Linear layer: maps from hidden state space to tag space
        ##########################################################################
        self.word_embedding = nn.Embedding(vocab_size, embedding_dim)
        self.char_embedding = nn.Embedding(char_size, char_embedding_dim )
        self.charLSTM = nn.LSTM(char_embedding_dim, char_hidden_dim)
        self.lstm = nn.LSTM(embedding_dim+char_hidden_dim, hidden_dim)
        self.hiddenToTag = nn.Linear(hidden_dim, tagset_size)

        ##########################################################################
        #                             END OF YOUR CODE
        ##########################################################################

    def forward(self, sentence, chars):
        tag_scores = None
        ##########################################################################
        # TODO: Implement the forward pass.
        # Given a tokenized index-mapped sentence and a character sequence as the ar
        # find the corresponding scores for tags
        # returns:: tag_scores (Tensor)
        ##########################################################################
        embeddings = self.word_embedding(sentence)
        char_hidden_result = []
        for char in chars:
          char_embedding = self.char_embedding(char)
          _, (char_hidden_state, char_cell_state) = self.charLSTM(char_embedding.vie
          char_word_hidden = char_hidden_state.view(-1)
          char_hidden_result.append(char_word_hidden)
        char_hidden_result = torch.stack(tuple(char_hidden_result))
        combined_embedding = torch.cat((embeddings,char_hidden_result),1)
        lstm_result, _ = self.lstm(combined_embedding.view(len(sentence),1,-1))
        tag_s = self.hiddenToTag(lstm_result.view(len(sentence),-1))
```

```
        tag_s = self.hidden2tag(lstm_result.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_s,dim=1)

        ##########################################################################
        #                              END OF YOUR CODE
        ##########################################################################
        return tag_scores

def train_char(epoch, model, loss_function, optimizer):
    train_loss = 0
    train_examples = 0
    for sentence, tags in training_data:
        ##########################################################################
        # TODO: Implement the training loop
        # Hint: you can use the prepare_sequence method for creating index mappings
        # for sentences as well as character sequences. Find the gradient with
        # respect to the loss and update the model parameters using the optimizer.
        ##########################################################################
        words = []
        for word in sentence:
          words.append(prepare_sequence(word,char_to_idx))
        # words = prepare_sequence(sentence,char_to_idx)
        #print(words)
        sentence_in = prepare_sequence(sentence,word_to_idx)
        targets = prepare_sequence(tags,tag_to_idx)
        model.zero_grad()
        tag_scores = model(sentence_in,words)
        loss = loss_function(tag_scores,targets)
        loss.backward()
        optimizer.step()
        train_loss += loss
        train_examples += len(targets)
        ##########################################################################
        #                              END OF YOUR CODE
        ##########################################################################

    avg_train_loss = train_loss / train_examples
    avg_val_loss, val_accuracy = evaluate_char(model, loss_function, optimizer)

    print("Epoch: {}/{}\tAvg Train Loss: {:.4f}\tAvg Val Loss: {:.4f}\t Val Accuracy
                                                                  EPOCHS,
                                                                  avg_train_loss
                                                                  avg_val_loss,
                                                                  val_accuracy))

def evaluate_char(model, loss_function, optimizer):
    # returns:: avg_val_loss (float)
    # returns:: val_accuracy (float)
    val_loss = 0
    correct = 0
    val_examples = 0
    with torch.no_grad():
```

```
        for sentence, tags in val_data:
            ##########################################################################
            # TODO: Implement the evaluate loop
            # Find the average validation loss along with the validation accuracy.
            # Hint: To find the accuracy, argmax of tag predictions can be used.
            ##########################################################################

            words = []
            for word in sentence:
              words.append(prepare_sequence(word,char_to_idx))
            sen_input = prepare_sequence(sentence, word_to_idx)
            targets = prepare_sequence(tags,tag_to_idx)
            tag_scores = model(sen_input,words)
            _, indices = torch.max(tag_scores,1)
            val_loss += loss_function(tag_scores,targets)
            correct += torch.sum(indices == torch.LongTensor(targets))
            val_examples += len(targets)
            ##########################################################################
            #                             END OF YOUR CODE
            ##########################################################################
    val_accuracy = 100. * correct / val_examples
    avg_val_loss = val_loss / val_examples
    return avg_val_loss, val_accuracy

##############################################################################
# TODO: Initialize the model, optimizer and the loss function
##############################################################################
# device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# if torch.cuda.is_available():
#   model.cuda()

model = CharPOSTagger(EMBEDDING_DIM,HIDDEN_DIM,CHAR_EMBEDDING_DIM,CHAR_HIDDEN_DIM,
                       len(char_to_idx.keys()), len(word_to_idx.keys()), len(tag_to_
loss_function = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr = LEARNING_RATE)


import time
##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
for epoch in range(1, EPOCHS + 1):
    start = time.time()
    train_char(epoch, model, loss_function, optimizer)
    print(f"time used in this epoch{epoch}: ", time.time() - start)
```

```
Epoch: 1/10     Avg Train Loss: 0.0279  Avg Val Loss: 0.0145     Val Accuracy: 91
time used in this epoch1:  213.51745438575745
Epoch: 2/10     Avg Train Loss: 0.0098  Avg Val Loss: 0.0117     Val Accuracy: 93
time used in this epoch2:  228.90641283988953
Epoch: 3/10     Avg Train Loss: 0.0064  Avg Val Loss: 0.0114     Val Accuracy: 93
time used in this epoch3:  225.43528962135315
Epoch: 4/10     Avg Train Loss: 0.0049  Avg Val Loss: 0.0118     Val Accuracy: 94
time used in this epoch4:  235.64571452140808
Epoch: 5/10     Avg Train Loss: 0.0041  Avg Val Loss: 0.0119     Val Accuracy: 94
time used in this epoch5:  249.47587370872498
Epoch: 6/10     Avg Train Loss: 0.0036  Avg Val Loss: 0.0119     Val Accuracy: 94
time used in this epoch6:  245.06821250915527
Epoch: 7/10     Avg Train Loss: 0.0033  Avg Val Loss: 0.0124     Val Accuracy: 94
time used in this epoch7:  240.75135707855225
Epoch: 8/10     Avg Train Loss: 0.0030  Avg Val Loss: 0.0124     Val Accuracy: 94
time used in this epoch8:  241.44244074821472
Epoch: 9/10     Avg Train Loss: 0.0029  Avg Val Loss: 0.0127     Val Accuracy: 94
time used in this epoch9:  240.94447422027588
Epoch: 10/10    Avg Train Loss: 0.0028  Avg Val Loss: 0.0128     Val Accuracy: 94
time used in this epoch10:  231.55236339569092
```

```
def test():
    val_loss = 0
    correct = 0
    val_examples = 0
    predicted_tags = []
    with torch.no_grad():
        for sentence in test_sentences:
            ############################################################################
            # TODO: Implement the test loop
            # This method saves the predicted tags for the sentences in the test set
            # The tags are first added to a list which is then written to a file for
            # submission. An empty string is added after every sequence of tags
            # corresponding to a sentence to add a newline following file formatting
            # convention, as has been done already.
            ############################################################################
            words = []
            for word in sentence:
              words.append(prepare_sequence(word,char_to_idx))
            sen_input = prepare_sequence(sentence, word_to_idx)
            # targets = prepare_sequence(tags,tag_to_idx)
            tag_scores = model(sen_input,words)
            _, indexes = torch.max(tag_scores,1)
            for i in range(len(indexes)):
              for key, value in tag_to_idx.items():
                if indexes[i] == value:
                    predicted_tags.append(key)

            ############################################################################
            #                             END OF YOUR CODE
            ############################################################################
            predicted_tags.append("")
```

```
            predicted_tags.append(' ')
    print(predicted_tags)
    with open('test_labels_char.txt', 'w+') as f:
        for item in predicted_tags:
            f.write("%s\n" % item)
```

```
test()
```

```
['NNP', 'NNP', 'NNP', 'POS', 'NNP', 'NN', 'VBD', 'PRP', 'VBD', 'DT', 'JJ', 'NN',
```

Tune your hyperparameters, to get a performance of **at least 85%** on the validation set for the CharPO

Test accuracy

Also evaluate your performance on the test data by submitting test_labels.txt and **report your test acc**

92.41

Error analysis

```
###############################################################################
# TODO: Generate predictions from val data
# Create lists of words, tags predicted by the model and ground truth tags.
###############################################################################
def generate_predictions(model, test_sentences):
    # returns:: word_list (str list)
    # returns:: model_tags (str list)
    # returns:: gt_tags (str list)
    # Your code here
    word_list = []
    model_tags = []
    gt_tags = []
    for sentence,tag in test_sentences:
      words = []
      for word in sentence:
        words.append(prepare_sequence(word,char_to_idx))
      sen_in = prepare_sequence(sentence,word_to_idx)
      # gt_tags.append(tag)
      # word_list.append(sentence)
      tag_scores = model(sen_in,words)
      _, indexes = torch.max(tag_scores,1)
      for i in range(len(indexes)):
        for key, value in tag_to_idx.items():
          if indexes[i] == value:
            model_tags.append(key)
      word_list.extend(sentence)
      gt_tags.extend(tag)
```

```
            # model_tags.append()
            #gt_tags.append(key)
          #word_list.append(sentence[i])
      return word_list, model_tags, gt_tags

##############################################################################
# TODO: Carry out error analysis
# From those lists collected from the above method, find the
# top-10 tuples of (model_tag, ground_truth_tag, frequency, example words)
# sorted by frequency
##############################################################################
def error_analysis(word_list, model_tags, gt_tags):
    # returns: errors (list of tuples)
    # Your code here
    import collections
    d = collections.defaultdict(list)


    temp_wl=[]
    temp_ml=[]
    temp_gt=[]
    errors = []
    for word,pred,ground_truth in zip(word_list,model_tags,gt_tags):
      # to find the error we need to find where the prediction is not equal to grour
      if pred != ground_truth:
        temp_wl.append(word)
        temp_ml.append(pred)
        temp_gt.append(ground_truth)
        d[(pred,ground_truth)].append(word)


    combined_pred_gt = zip(temp_ml,temp_gt)
    cc = collections.Counter(combined_pred_gt)

    for (pred,ground_truth), count in cc.most_common(10):
      freq = float(count/sum(cc.values()))
      errors.append((pred,ground_truth,freq,d[(pred,ground_truth)][:10]))
    return errors


word_list,model_tags,gt_tags = generate_predictions(model,val_data)
print(len(word_list),len(model_tags),len(gt_tags))
errors = error_analysis(word_list,model_tags,gt_tags)
for error in errors:
  print(error)
```

```
41851 41851 41851
('NN', 'NNP', 0.06224066390041494, ['Herman', 'Cordis', 'business', 'Willman', 'S
('NN', 'JJ', 0.05726141078838174, ['fetal-tissue', 'ultimate', 'equitable', 'flu-
('JJ', 'NN', 0.05228215767634855, ['panhandler', 'chief', 'drill', 'grade', 'week
('VBN', 'VBD', 0.04066390041493776, ['received', 'stepped', 'displayed', 'doubled
('JJ', 'NNP', 0.036929460580912864, ['Joan', 'Sanford', 'Ground', 'PATOIS', 'Orwe
('VBD', 'VBN', 0.036514522821576766, ['confiscated', 'contrived', 'established',
('WDT', 'IN', 0.036514522821576766, ['that', 'that', 'that', 'that', 'that', 'tha
('NNP', 'JJ', 0.03568464730290456, ['West', 'penny-brokerage', '30-share', 'assor
('NNP', 'NN', 0.03443983402489627, ['Utility', 'sight', 'comrade', 'Commission',
('VBZ', 'NNS', 0.026141078838174275, ['alleges', 'buffs', 'supports', 'themes', '
```

Report your findings here.

What kinds of errors does the character-level model make as compared to the original model, and why

- Interstingly the classifier here has made a mistake in recognizing proper noun, instead it has ma as Noun. so for example "3-share" is identified as noun instead of adjectice,while "Herman" is ide noun. I think it is the same mistakes as done by basic lstm, but here the difference is that now it and puntuations. firstly i think the model is not converged very properly though getting a 93 per a trained it more. But here the reason might be char level information must have overweighted the

Define a BiLSTM POS Tagger

A bidirectional LSTM that runs both left-to-right and right-to-left to represent dependencies between a thus captures dependencies in both directions.

In this part, you make your model bidirectional.

In addition, you should implement one of these modifications to improve the model's performance:

- Tune the model hyperparameters. Try at least 5 different combinations of parameters. For exam
    - number of LSTM layers
    - number of hidden dimensions
    - number of word embedding dimensions
    - dropout rate
    - learning rate
- Switch to pre-trained Word Embeddings instead of training them from scratch. Try at least one d example:
    - [Glove]
    - [Fast Text]
- Implement a different model architecture. Try at least one different architecture. For example:
    - adding a conditional random field on top of the LSTM

- adding Viterbi decoding to the model

```
!pip install torchtext==0.5.0
```

```
Collecting torchtext==0.5.0
  Downloading https://files.pythonhosted.org/packages/79/ef/54b8da26f37787f5c670a
    |████████████████████████████████| 81kB 4.7MB/s
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: tqdm in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: torch in /usr/local/lib/python3.6/dist-packages (f
Collecting sentencepiece
  Downloading https://files.pythonhosted.org/packages/74/f4/2d5214cbf13d06e7cb2c2
    |████████████████████████████████| 1.0MB 21.7MB/s
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (fro
Requirement already satisfied: urllib3<1.25,>=1.21.1 in /usr/local/lib/python3.6/
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dis
Requirement already satisfied: idna<2.9,>=2.5 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python3.6/
Installing collected packages: sentencepiece, torchtext
  Found existing installation: torchtext 0.3.1
    Uninstalling torchtext-0.3.1:
      Successfully uninstalled torchtext-0.3.1
Successfully installed sentencepiece-0.1.85 torchtext-0.5.0
WARNING: The following packages were previously imported in this runtime:
  [torchtext]
You must restart the runtime in order to use newly installed versions.
```

RESTART RUNTIME

```
import torchtext as text
print(text.__version__)
# ## 87 percent efficiency with these hyperparam
# EMBEDDING_DIM = 200
# DROPOUT = 0.1
# HIDDEN_DIM = 16
# LEARNING_RATE = 0.01
# ISBIDERECTIONAL = True
# LSTM_LAYERS = 4
# EPOCHS = 10
# ## 86 percent efficiency with these hyperparam
# EMBEDDING_DIM = 200
# DROPOUT = 0.1
# HIDDEN_DIM = 100
# LEARNING_RATE = 0.01
# ISBIDERECTIONAL = True
# LSTM_LAYERS = 4
# EPOCHS = 10
## 82 percent efficiency with these hyperparam
# EMBEDDING_DIM = 50
# DROPOUT = 0.1
```

```
# DROPOUT = 0.1
# HIDDEN_DIM = 25
# LEARNING_RATE = 0.01
# ISBIDERECTIONAL = True
# LSTM_LAYERS = 4
# EPOCHS = 10
#
#
#73 accuracy.. it decreased with these hyperparam
# EMBEDDING_DIM = 300
# DROPOUT = 0.1
# HIDDEN_DIM = 200
# LEARNING_RATE = 0.01
# ISBIDERECTIONAL = True
# LSTM_LAYERS = 4
# EPOCHS = 10
#
# 91
#
# EMBEDDING_DIM = 200
# DROPOUT = 0.2
# HIDDEN_DIM = 32
# LEARNING_RATE = 0.001
# ISBIDERECTIONAL = True
# LSTM_LAYERS = 2
# EPOCHS = 10
#
# 95
EMBEDDING_DIM = 200
DROPOUT = 0.2
HIDDEN_DIM = 32
LEARNING_RATE = 0.001
ISBIDERECTIONAL = True
LSTM_LAYERS = 2
EPOCHS = 10
```

```
0.5.0
```

Hyperparameter Analysis

I observed that learning rate and drop out effects a lot in terms of training the model and the validatio learning rate means penalizing the weights more and does the model quickly converge but the accura learning rate and at dropout of 0.2 i was able to hit the accuracy on validation above 90 percente as w cells for the accuracy. I used Glove embedding with 200 dimensions, i tried different other dimensions performance and therefor i choose 200 finally.

```
! ls .vector_cache/
```

```
glove.6B.100d.txt  glove.6B.200d.txt.pt  glove.6B.50d.txt
glove.6B.200d.txt  glove.6B.300d.txt     glove.6B.zip
```

```
class BiLSTMPOSTagger(nn.Module):
    # NOTE: you may have to modify these function headers to include your
    # modification, e.g. adding a parameter for embeddings data
    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(BiLSTMPOSTagger, self).__init__()
        ##############################################################################
        # TODO: Define and initialize anything needed for the forward pass.
        # You are required to create a model with:
        # an embedding layer: that maps words to the embedding space
        # a BiLSTM layer: that takes word embeddings as input and outputs hidden sta
        # a Linear layer: maps from hidden state space to tag space
        ##############################################################################
        self.gl = text.vocab.GloVe(name='6B', dim = embedding_dim)
        DROPOUT = 0.2
        # print(DROPOUT)
        self.dropout = nn.Dropout(DROPOUT)
        # if LSTM_LAYERS <1:
        #   DROPOUT = 0
        self.lstm = nn.LSTM(embedding_dim, hidden_size=hidden_dim,num_layers=LSTM_LA
                            dropout = DROPOUT if LSTM_LAYERS >1 else 0, bidirectiona
        # if ISBIDERECTIONAL:
        #   hidden_dim = hidden_dim*2
        self.hiddenToTag = nn.Linear(hidden_dim*2 if ISBIDERECTIONAL else hidden_dim
        ##############################################################################
        #                             END OF YOUR CODE
        ##############################################################################

    def forward(self, sentence):
        tag_scores = None
        ##############################################################################
        # TODO: Implement the forward pass.
        # Given a tokenized index-mapped sentence as the argument,
        # find the corresponding scores for tags
        # returns:: tag_scores (Tensor)
        ##############################################################################
        # print(dir(self.gl))

        embeddings = self.gl.get_vecs_by_tokens(sentence, lower_case_backup=True)
        lstm_result,_ = self.lstm(embeddings.view(len(sentence),1,-1))
        tag_s = self.hiddenToTag(lstm_result.view(len(sentence),-1))
        tag_scores = F.log_softmax(tag_s,dim=1)

        ##############################################################################
        #                             END OF YOUR CODE
        ##############################################################################
        return tag_scores
```

```
def train(epoch, model, loss_function, optimizer):
  train_loss= 0
  train_examples = 0
  for sentence, tags in training_data:
    model.zero_grad()
    sentence_in = sentence
    targets = prepare_sequence(tags,tag_to_idx)
    tag_scores = model(sentence_in)
    loss = loss_function(tag_scores,targets)
    loss.backward()
    optimizer.step()
    train_loss+=loss.cpu().detach().numpy()
    train_examples+=len(targets.cpu().detach().numpy())
  avg_train_loss = train_loss / train_examples
  avg_val_loss, val_accuracy = evaluate(model, loss_function, optimizer)
  print("Epoch: {}/{}\tAvg Train Loss: {:.4f}\tAvg Val Loss: {:.4f}\t Val Accuracy:
                                                                     EPOCHS,
                                                                     avg_train_loss
                                                                     avg_val_loss,
                                                                     val_accuracy))


def evaluate(model, loss_function, optimizer):
    # returns:: avg_val_loss (float)
    # returns:: val_accuracy (float)
    val_loss = 0
    correct = 0
    val_examples = 0
    with torch.no_grad():
        for sentence, tags in val_data:
          sentence_in = sentence
          targets = prepare_sequence(tags, tag_to_idx)
          tag_scores = model(sentence_in)
          _, indexes = torch.max(tag_scores,1)
          loss = loss_function(tag_scores,targets)
          val_loss += loss.cpu().detach().numpy()
          correct += (torch.sum(indexes == torch.LongTensor(targets)).cpu().detach()
          val_examples += len(targets.cpu().detach().numpy())
    val_accuracy = 100. * correct / val_examples
    avg_val_loss = val_loss / val_examples
    return avg_val_loss, val_accuracy




##############################################################################
# TODO: Initialize the model, optimizer and the loss function
##############################################################################
import time

model = BiLSTMPOSTagger(embedding_dim=EMBEDDING_DIM, hidden_dim=HIDDEN_DIM,
                        vocab_size=len(word_to_idx.keys()), tagset_size=len(tag_to_i
```

```
loss_function = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(),lr=LEARNING_RATE)


################################################################################
#                              END OF YOUR CODE                                #
################################################################################
for epoch in range(1, EPOCHS + 1):
    start = time.time()
    train(epoch, model, loss_function, optimizer)
    print(f"time used for epoch{epoch}: ",time.time() - start)
```

```
Epoch: 1/10     Avg Train Loss: 0.0306  Avg Val Loss: 0.0148     Val Accuracy: 90
time used for epoch1:  175.20788526535034
Epoch: 2/10     Avg Train Loss: 0.0114  Avg Val Loss: 0.0104     Val Accuracy: 93
time used for epoch2:  173.2424259185791
Epoch: 3/10     Avg Train Loss: 0.0082  Avg Val Loss: 0.0089     Val Accuracy: 94
time used for epoch3:  175.07934665679932
Epoch: 4/10     Avg Train Loss: 0.0067  Avg Val Loss: 0.0084     Val Accuracy: 94
time used for epoch4:  175.2431936264038
Epoch: 5/10     Avg Train Loss: 0.0058  Avg Val Loss: 0.0078     Val Accuracy: 95
time used for epoch5:  174.24291110038757
Epoch: 6/10     Avg Train Loss: 0.0052  Avg Val Loss: 0.0076     Val Accuracy: 95
time used for epoch6:  174.9787232875824
Epoch: 7/10     Avg Train Loss: 0.0046  Avg Val Loss: 0.0076     Val Accuracy: 95
time used for epoch7:  177.15688467025757
Epoch: 8/10     Avg Train Loss: 0.0042  Avg Val Loss: 0.0074     Val Accuracy: 95
time used for epoch8:  174.78469514846802
Epoch: 9/10     Avg Train Loss: 0.0040  Avg Val Loss: 0.0073     Val Accuracy: 95
time used for epoch9:  172.81200289726257
Epoch: 10/10    Avg Train Loss: 0.0037  Avg Val Loss: 0.0073     Val Accuracy: 95
time used for epoch10:  175.9452600479126
```

Your modified model should get a performance of **at least 90%** on the validation set.

Test accuracy

Also evaluate your performance on the test data by submitting test_labels.txt and **report your test acc**

94.56

```
def test():
    val_loss = 0
    correct = 0
    val_examples = 0
    predicted_tags = []
    with torch.no_grad():
        for sentence in test_sentences:
            ##########################################################################
            # TODO: Implement the test loop
            # This method saves the predicted tags for the sentences in the test set
            # The tags are first added to a list which is then written to a file for
```

```
            # submission. An empty string is added after every sequence of tags
            # corresponding to a sentence to add a newline following file formatting
            # convention, as has been done already.
            ########################################################################
            # sen_in = prepare_sequence(sentence,word_to_idx)
            tag_scores = model(sentence)
            _, indexes = torch.max(tag_scores,1)
            for i in range(len(indexes)):
              for key, value in tag_to_idx.items():
                if indexes[i] == value:
                    predicted_tags.append(key)


            ########################################################################
            #                              END OF YOUR CODE
            ########################################################################
            predicted_tags.append("")
    print(predicted_tags)
    with open('test_labels_bilstm.txt', 'w+') as f:
        for item in predicted_tags:
            f.write("%s\n" % item)
```

```
test()
```

```
['NNP', 'NNP', 'NNP', 'POS', 'CD', 'NN', 'VBD', 'PRP', 'VBD', 'DT', 'JJ', 'NN', '
```

Error analysis

Report your findings here.

Compare the top-10 errors made by this modified model with the errors made by the model from part hyperparameter combinations, choose the model with the highest validation data accuracy. What erro compared to the modified model, and why do you think it made them?

Feel free to reuse the methods defined above for this purpose.

- Here as we can see the error analysis at the end of the notebook, the model is making mistakes here the model is overweighting because now here the model is trying to identify the noun as pr tending towards overfitting.But the model is able to learn char level and also able to understand both direction and hence we can see the the tags are predicted better in BiLSTM

```
##############################################################################
# TODO: Generate predictions from val data
# Create lists of words, tags predicted by the model and ground truth tags.
##############################################################################
def generate_predictions(model, test_sentences):
    # returns:: word_list (str list)
    # returns:: model_tags (str list)
```

```
    # returns:: model_tags (str list)
    # returns:: gt_tags (str list)
    # Your code here
    word_list = []
    model_tags = []
    gt_tags = []
    for sentence,tag in test_sentences:

      # sen_in = prepare_sequence(sentence,word_to_idx)
      # gt_tags.append(tag)
      # word_list.append(sentence)
      tag_scores = model(sentence)
      _, indexes = torch.max(tag_scores,1)
      for i in range(len(indexes)):
        for key, value in tag_to_idx.items():
          if indexes[i] == value:
            model_tags.append(key)
      word_list.extend(sentence)
      gt_tags.extend(tag)

          # model_tags.append()
          #gt_tags.append(key)
        #word_list.append(sentence[i])
    return word_list, model_tags, gt_tags

##############################################################################
# TODO: Carry out error analysis
# From those lists collected from the above method, find the
# top-10 tuples of (model_tag, ground_truth_tag, frequency, example words)
# sorted by frequency
##############################################################################
def error_analysis(word_list, model_tags, gt_tags):
    # returns: errors (list of tuples)
    # Your code here
    import collections
    d = collections.defaultdict(list)


    temp_wl=[]
    temp_ml=[]
    temp_gt=[]
    errors = []
    for word,pred,ground_truth in zip(word_list,model_tags,gt_tags):
      # to find the error we need to find where the prediction is not equal to grour
      if pred != ground_truth:
        temp_wl.append(word)
        temp_ml.append(pred)
        temp_gt.append(ground_truth)
        d[(pred,ground_truth)].append(word)


    combined_pred_gt = zip(temp_ml,temp_gt)
```

```
    cc = collections.Counter(combined_pred_gt)

    for (pred,ground_truth), count in cc.most_common(10):
      freq = float(count/sum(cc.values()))
      errors.append((pred,ground_truth,freq,d[(pred,ground_truth)][:10]))






    return errors

word_list,model_tags,gt_tags = generate_predictions(model,val_data)
print(len(word_list),len(model_tags),len(gt_tags))
errors = error_analysis(word_list,model_tags,gt_tags)
for error in errors:
  print(error)
```

```
41851 41851 41851
('NNP', 'NN', 0.09544573643410853, ['province', 'press', 'cup', 'city', 'mail', '
('NN', 'NNP', 0.09544573643410853, ['Extension', 'Service', 'Information', 'Agenc
('NN', 'JJ', 0.061046511627906974, ['nonprofit', 'adamant', 'ultimate', 'five-ses
('JJ', 'NN', 0.05329457364341085, ['overhead', 'span', 'estuarian', 'rent', 'blac
('NNP', 'JJ', 0.041666666666666664, ['universal', 'historic', 'direct', 'nuclear'
('JJ', 'NNP', 0.03488372093023256, ['mare-COOR', 'Independent', 'German', 'Long',
('VBN', 'VBD', 0.029554263565891473, ['stepped', 'displayed', 'decreased', 'close
('NN', 'NNS', 0.029069767441860465, ['buffs', 'doldrums', 'chains', 'things', 'sy
('VBD', 'VBN', 0.025193798449612403, ['issued', 'executed', 'left', 'shut', 'obse
('NNP', 'NNS', 0.01695736434108527, ['alleges', 'write-downs', 'right-to-lifers',
```