POS_tagging

March 6, 2020

1 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. We will also implement several 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 noun) and `DT` (determiner).

1.1 Building a POS Tagger

1.1.1 Setup

```
In [1]: import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        import random

        random.seed(1)
```

1.1.2 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 train-val split on this data for training and evaluation purpose.

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

```
In [2]: 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
```

1.1.3 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.

```
In [3]: 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("Total tags", len(tag_to_idx))
        print("Vocab size", len(word_to_idx))
```

```
Total tags 44
Vocab size 21589
```

```
In [4]: def prepare_sequence(sent, idx_mapping):
            idxs = [idx_mapping[word] for word in sent]
            return torch.tensor(idxs, dtype=torch.long)
```

1.1.4 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 given sentence.

First we need to define some default hyperparameters.

```
In [306]: EMBEDDING_DIM = 20
          HIDDEN_DIM = 25
          LEARNING_RATE = 0.1
          LSTM_LAYERS = 1
          DROPOUT = 0
          EPOCHS = 30
```

1.1.5 Define Model

The model takes as input a sentence as a tensor in the index space. This sentence is then converted to embedding space where each word 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 then passed to a Linear layer that produces the probability distribution for the tags of every word. The model will output the tag with the highest probability for a given word.

```
In [144]: 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.embedding_dim = embedding_dim
                  self.hidden_dim = hidden_dim
                  self.vocab_size = vocab_size
                  self.tagset_size = tagset_size

                  self.embedding = nn.Embedding(num_embeddings = vocab_size, embedding_dim = se
                  self.lstm = nn.LSTM(input_size = self.embedding_dim, hidden_size = self.hidde
                  self.linear = nn.Linear(in_features = self.hidden_dim, out_features = tagset_

              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)
                  ###############################################################################
                  batch_size = sentence.shape[0]
                  self.hidden_cell = (torch.zeros(1, batch_size, self.hidden_dim),
                                      torch.zeros(1, batch_size, self.hidden_dim))

                  embedding = self.embedding(sentence)
                  embedding = embedding.unsqueeze(-1).permute(2, 0, 1)

                  lstm_out, self.hidden_cell = self.lstm(embedding, self.hidden_cell)
                  lstm_out = lstm_out.squeeze(0)
                  tag_scores = self.linear(lstm_out)

                  return tag_scores
```

1.1.6 Training

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

```
In [301]: 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.
        ###############################################################################
        sentence_new = prepare_sequence(sentence, word_to_idx)
        tags_new = torch.as_tensor(prepare_sequence(tags, tag_to_idx))

        pred_tags = model(sentence_new)

        loss = loss_function(pred_tags, tags_new)
        train_loss += loss.item()
        train_examples += len(sentence_new)

        model.zero_grad()
        loss.backward()
        optimizer.step()

    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.
            ###############################################################################
            sentence_new = prepare_sequence(sentence, word_to_idx)
            tags_new = torch.as_tensor(prepare_sequence(tags, tag_to_idx))
            pred_tags = model(sentence_new)

            loss = loss_function(pred_tags, tags_new)
```

```
                val_loss += loss.item()

                pred_tags = torch.argmax(pred_tags, dim = 1)
                correct += sum(pred_tags == tags_new).item()
                val_examples += len(sentence_new)
        val_accuracy = 100. * correct / val_examples
        avg_val_loss = val_loss / val_examples
        return avg_val_loss, val_accuracy
```

In [302]:
```
########################################################################
# TODO: Initialize the model, optimizer and the loss function
########################################################################
basicpos_model = BasicPOSTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_idx), len(tag_
basicpos_optimizer = optim.SGD(basicpos_model.parameters(), lr = LEARNING_RATE)
loss_function = nn.CrossEntropyLoss()

for epoch in range(1, EPOCHS + 1):
    train(epoch, basicpos_model, loss_function, basicpos_optimizer)
```

```
Epoch: 1/30       Avg Train Loss: 0.0798       Avg Val Loss: 0.0558       Val Accuracy: 65
Epoch: 2/30       Avg Train Loss: 0.0469       Avg Val Loss: 0.0420       Val Accuracy: 74
Epoch: 3/30       Avg Train Loss: 0.0368       Avg Val Loss: 0.0350       Val Accuracy: 78
Epoch: 4/30       Avg Train Loss: 0.0306       Avg Val Loss: 0.0301       Val Accuracy: 82
Epoch: 5/30       Avg Train Loss: 0.0259       Avg Val Loss: 0.0266       Val Accuracy: 84
Epoch: 6/30       Avg Train Loss: 0.0223       Avg Val Loss: 0.0239       Val Accuracy: 86
Epoch: 7/30       Avg Train Loss: 0.0195       Avg Val Loss: 0.0219       Val Accuracy: 87
Epoch: 8/30       Avg Train Loss: 0.0172       Avg Val Loss: 0.0204       Val Accuracy: 88
Epoch: 9/30       Avg Train Loss: 0.0154       Avg Val Loss: 0.0192       Val Accuracy: 88
Epoch: 10/30      Avg Train Loss: 0.0139       Avg Val Loss: 0.0182       Val Accuracy: 89
Epoch: 11/30      Avg Train Loss: 0.0127       Avg Val Loss: 0.0175       Val Accuracy: 89
Epoch: 12/30      Avg Train Loss: 0.0116       Avg Val Loss: 0.0168       Val Accuracy: 90
Epoch: 13/30      Avg Train Loss: 0.0108       Avg Val Loss: 0.0163       Val Accuracy: 90
Epoch: 14/30      Avg Train Loss: 0.0100       Avg Val Loss: 0.0159       Val Accuracy: 90
Epoch: 15/30      Avg Train Loss: 0.0094       Avg Val Loss: 0.0155       Val Accuracy: 90
Epoch: 16/30      Avg Train Loss: 0.0088       Avg Val Loss: 0.0152       Val Accuracy: 91
Epoch: 17/30      Avg Train Loss: 0.0083       Avg Val Loss: 0.0150       Val Accuracy: 91
Epoch: 18/30      Avg Train Loss: 0.0079       Avg Val Loss: 0.0147       Val Accuracy: 91
Epoch: 19/30      Avg Train Loss: 0.0075       Avg Val Loss: 0.0146       Val Accuracy: 91
Epoch: 20/30      Avg Train Loss: 0.0072       Avg Val Loss: 0.0144       Val Accuracy: 91
Epoch: 21/30      Avg Train Loss: 0.0069       Avg Val Loss: 0.0143       Val Accuracy: 91
Epoch: 22/30      Avg Train Loss: 0.0066       Avg Val Loss: 0.0142       Val Accuracy: 91
Epoch: 23/30      Avg Train Loss: 0.0064       Avg Val Loss: 0.0141       Val Accuracy: 91
Epoch: 24/30      Avg Train Loss: 0.0062       Avg Val Loss: 0.0140       Val Accuracy: 91
Epoch: 25/30      Avg Train Loss: 0.0060       Avg Val Loss: 0.0140       Val Accuracy: 91
Epoch: 26/30      Avg Train Loss: 0.0059       Avg Val Loss: 0.0139       Val Accuracy: 91
Epoch: 27/30      Avg Train Loss: 0.0057       Avg Val Loss: 0.0139       Val Accuracy: 92
Epoch: 28/30      Avg Train Loss: 0.0056       Avg Val Loss: 0.0138       Val Accuracy: 92
Epoch: 29/30      Avg Train Loss: 0.0055       Avg Val Loss: 0.0138       Val Accuracy: 92
```

```
Epoch: 30/30        Avg Train Loss: 0.0054        Avg Val Loss: 0.0138        Val Accuracy: 9
```

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.

```
In [317]: 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.
                      ###############################################################################
                      sentence_new = prepare_sequence(sentence, word_to_idx)
                      pred_tags = basicpos_model(sentence_new)
                      pred_tags = torch.argmax(pred_tags, dim = 1).tolist()
                      for tag_id in pred_tags:
                          predicted_tags.append(idx_to_tag[tag_id])

                      predicted_tags.append("")

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


In [318]: test()
```

1.1.7 Test accuracy

Evaluate your performance on the test data by submitting test_labels.txt generated by the method above and **report your test accuracy here**.

The accuracy I achieved on gradescope was 86.7172% (team name is "browns 4 superbowl").

Imitate the above method to generate prediction for validation data. Create lists of words, tags predicted by the model and ground truth tags.

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

```
In [312]: ###############################################################################
          # TODO: Generate predictions from val data
          # Create lists of words, tags predicted by the model and ground truth tags.
          ###############################################################################
```

```
def generate_predictions(model, val_data):
    # returns:: word_list (str list)
    # returns:: model_tags (str list)
    # returns:: gt_tags (str list)
    word_list = []
    model_tags = []
    gt_tags = []

    for sentence, tags in val_data:
        sentence_new = prepare_sequence(sentence, word_to_idx)
        pred_tags = model(sentence_new)
        pred_tags = torch.argmax(pred_tags, dim = 1).tolist()
        pred_tags = [idx_to_tag[tag_id] for tag_id in pred_tags]

        word_list += sentence
        model_tags += pred_tags
        gt_tags += tags
    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)
    errors = {}
    for index, model_tag in enumerate(model_tags):
        gt_tag = gt_tags[index]
        word = word_list[index]
        if model_tag == gt_tag:
            continue
        if (model_tag, gt_tag) not in errors:
            errors[(model_tag, gt_tag)] = (0, set())
        curr_count, curr_word_list = errors[(model_tag, gt_tag)]
        curr_word_list.add(word)
        errors[(model_tag, gt_tag)] = (curr_count + 1, curr_word_list)
    errors = sorted(errors.items(), key = lambda x : x[1][0], reverse = True)[0:10]
    return errors

basicpos_word_list, basicpos_model_tags, basicpos_gt_tags = generate_predictions(basi
basicpos_errors = error_analysis(basicpos_word_list, basicpos_model_tags, basicpos_gt
print("gt_tag\t| model_tag\t| freq.\t| examples")
horizontal_line = "-------------------------------------------------------------------
print(horizontal_line)
for error in basicpos_errors:
    tags, info = error
```

```
            model_tag, gt_tag = tags
            num_errors, example_words = info
            print("{}\t| {}\t\t| {}\t| {}\n{}".format(gt_tag, model_tag, num_errors, random.s

gt_tag        | model_tag        | freq.        | examples
-------------------------------------------------------------------------
VBN        | VBD                | 181        | ['licensed', 'decided', 'tested', 'snapped', 're
-------------------------------------------------------------------------
VB        | NN                | 167        | ['appeal', 'buy-back', 'factor', 'view', 'survey']
-------------------------------------------------------------------------
VBD        | VBN                | 156        | ['hurt', 'deprived', 'directed', 'acquired', 'al
-------------------------------------------------------------------------
NNP        | NN                | 142        | ['Trinen', 'Anglo', 'Cologne', 'Protestantism',
-------------------------------------------------------------------------
NN        | VB                | 135        | ['lead', 'report', 'fare', 'hold', 'shield']
-------------------------------------------------------------------------
NN        | JJ                | 129        | ['past', 'chief', 'weekly', 'hazardous-waste', 'co
-------------------------------------------------------------------------
JJ        | NNP                | 126        | ['third-ranking', 'fetal-tissue', 'decade-long',
-------------------------------------------------------------------------
NN        | NNP                | 121        | ['dish', 'Business', 'festival', 'blip', 'theater
-------------------------------------------------------------------------
VB        | VBP                | 109        | ['exclude', 'seem', 'fly', 'believe', 'want']
-------------------------------------------------------------------------
VBP        | VB                | 97        | ['follow', 'lag', 'occur', 'halt', 'sell']
-------------------------------------------------------------------------
```

1.1.8 Error analysis

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

The top 3 and bottom 2 most frequent errors this model made had to deal with verb tags (VBN, VB, VBD, VB, VBP). In fact, VB shows up twice in tha tlist. Similarly, NN appears 3 times are the ground truth tag in our list. Meanwhile, there are no confusions in teams of pronouns, symbols, and prepositions. This is interesting, but sort of makes sense. For example, the difference between VBN (verb, past participle) and VBD (verb, past tense) is marginal. That's such a small subtlety. Similarly, the difference between NN (noun, singular or mass) and NNP (proper noun, singular) is marginal as well. Technically, NNP is a subet of NN, as all proper nouns are also nouns. I think the only true, poor errors made by the model are classifying NNs (nouns) as JJs (adjectives) and JJs as NNPs (proper noun, singular), as those are vastly different part of speech categories (one describes an object, while the other is the object). However, when investigating the example words closely, they kind of make sense. For example, "chief" can be a tribal chief, which is a singular noun, or it can mean imply something is the "chief cause" of something, or primary cause in other words. As such, similar reasoning can be used for "hazardous-waste", "past", and others.

In general, I think the model did a pretty decent job with the POS tagging. It achieved high train, validation, and test accuracy, and a significant majority of its errors came from two tag types that are similar (e.g. VBN and VBD, or NNP and NN). Most of these errors came from abiguity in

the tags themselves, where I think humans would make very similar errors.

1.2 Define a Character Level POS Tagger

We can use the character-level information present to augment our word embeddings. Words that end with -ing or -ly give quite a bit of information about their POS tags. To incorporate this information, we can run a character level LSTM on every word (treated as a tensor of characters, each mapped to character-index space) to create a character-level representation of the word. This representation can be concatenated with the word embedding (as in the BasicPOSTagger) to create a new word embedding that captures more information.

```
In [246]: # 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
          EMBEDDING_DIM = 20
          HIDDEN_DIM = 25
          LEARNING_RATE = 0.1
          LSTM_LAYERS = 1
          DROPOUT = 0
          EPOCHS = 30
          CHAR_EMBEDDING_DIM = 10
          CHAR_HIDDEN_DIM = 15
```

```
In [253]: 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 hid
```

```
        # a Linear layer: maps from hidden state space to tag space
        ######################################################################
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.char_embedding_dim = char_embedding_dim
        self.char_hidden_dim = char_hidden_dim
        self.char_size = char_size
        self.vocab_size = vocab_size
        self.tagset_size = tagset_size

        self.char_embedding = nn.Embedding(num_embeddings = char_size, embedding_dim
        self.word_embedding = nn.Embedding(num_embeddings = vocab_size, embedding_di
        self.char_lstm = nn.LSTM(input_size = self.char_embedding_dim, hidden_size =
        self.joint_lstm = nn.LSTM(input_size = self.embedding_dim + self.char_hidden_
        self.linear = nn.Linear(in_features = self.hidden_dim, out_features = tagset_

    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)
        ######################################################################
        word_batch_size = sentence.shape[0]
        self.joint_hidden_cell = (torch.zeros(1, word_batch_size, self.hidden_dim),
                                  torch.zeros(1, word_batch_size, self.hidden_dim))

        word_char_embeddings = torch.Tensor([])
        for word_chars in chars:
            char_batch_size = len(word_chars)
            self.char_hidden_cell = (torch.zeros(1, 1, self.char_hidden_dim),
                                     torch.zeros(1, 1, self.char_hidden_dim))

            char_embedding = self.char_embedding(word_chars)
            char_embedding = char_embedding.unsqueeze(1)
            char_lstm_out, _ = self.char_lstm(char_embedding, self.char_hidden_cell)
            char_lstm_out = char_lstm_out[-1]
            word_char_embeddings = torch.cat((word_char_embeddings, char_lstm_out), d
        word_char_embeddings = word_char_embeddings.unsqueeze(0)

        sentence_embedding = self.word_embedding(sentence)
        sentence_embedding = sentence_embedding.unsqueeze(0)

        joint_embedding = torch.cat((sentence_embedding, word_char_embeddings), dim =

        joint_lstm_out, _ = self.joint_lstm(joint_embedding, self.joint_hidden_cell)
        joint_lstm_out = joint_lstm_out.squeeze(0)
```

```
        tag_scores = self.linear(joint_lstm_out)

        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.
        ###############################################################################
        #print(sentence)
        chars_new = [prepare_sequence([char for char in word], char_to_idx) for word
        #print(chars_new)
        sentence_new = prepare_sequence(sentence, word_to_idx)
        tags_new = prepare_sequence(tags, tag_to_idx)
        #print("tags new: ", tags_new)

        pred_tags = model(sentence_new, chars_new)
        #print("pred tags: ", pred_tags)
        #print("tags pred new: ", tags_new)

        loss = loss_function(pred_tags, tags_new)
        train_loss += loss.item()
        train_examples += len(sentence)

        model.zero_grad()
        loss.backward()
        optimizer.step()

    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.
                    ###################################################################
                    chars_new = [prepare_sequence([char for char in word], char_to_idx) for w
                    sentence_new = prepare_sequence(sentence, word_to_idx)
                    tags_new = prepare_sequence(tags, tag_to_idx)

                    pred_tags = model(sentence_new, chars_new)

                    loss = loss_function(pred_tags, tags_new)
                    val_loss += loss.item()

                    pred_tags = torch.argmax(pred_tags, dim = 1)
                    correct += sum(pred_tags == tags_new).item()
                    val_examples += len(sentence)
            val_accuracy = 100. * correct / val_examples
            avg_val_loss = val_loss / val_examples
            return avg_val_loss, val_accuracy
```

In [254]:
```
        ###############################################################################
        # TODO: Initialize the model, optimizer and the loss function
        ###############################################################################
        charpos_model = CharPOSTagger(EMBEDDING_DIM, HIDDEN_DIM, CHAR_EMBEDDING_DIM, CHAR_HI
        charpos_optimizer = optim.SGD(model.parameters(), lr = LEARNING_RATE)
        loss_function = nn.CrossEntropyLoss()

        ###############################################################################
        #                                END OF YOUR CODE                             #
        ###############################################################################
        for epoch in range(1, EPOCHS + 1):
            train_char(epoch, charpos_model, loss_function, charpos_optimizer)
```

```
Epoch: 1/30       Avg Train Loss: 0.0632       Avg Val Loss: 0.0387       Val Accuracy: 73
Epoch: 2/30       Avg Train Loss: 0.0320       Avg Val Loss: 0.0285       Val Accuracy: 80
Epoch: 3/30       Avg Train Loss: 0.0228       Avg Val Loss: 0.0292       Val Accuracy: 79
Epoch: 4/30       Avg Train Loss: 0.0215       Avg Val Loss: 0.0192       Val Accuracy: 87
Epoch: 5/30       Avg Train Loss: 0.0169       Avg Val Loss: 0.0172       Val Accuracy: 89
Epoch: 6/30       Avg Train Loss: 0.0152       Avg Val Loss: 0.0160       Val Accuracy: 90
Epoch: 7/30       Avg Train Loss: 0.0141       Avg Val Loss: 0.0152       Val Accuracy: 90
Epoch: 8/30       Avg Train Loss: 0.0132       Avg Val Loss: 0.0146       Val Accuracy: 91
Epoch: 9/30       Avg Train Loss: 0.0125       Avg Val Loss: 0.0142       Val Accuracy: 91
Epoch: 10/30       Avg Train Loss: 0.0118       Avg Val Loss: 0.0140       Val Accuracy: 91
Epoch: 11/30       Avg Train Loss: 0.0113       Avg Val Loss: 0.0135       Val Accuracy: 91
Epoch: 12/30       Avg Train Loss: 0.0108       Avg Val Loss: 0.0132       Val Accuracy: 91
```

```
Epoch: 13/30        Avg Train Loss: 0.0104        Avg Val Loss: 0.0130        Val Accuracy: 9
Epoch: 14/30        Avg Train Loss: 0.0100        Avg Val Loss: 0.0127        Val Accuracy: 9
Epoch: 15/30        Avg Train Loss: 0.0097        Avg Val Loss: 0.0125        Val Accuracy: 9
Epoch: 16/30        Avg Train Loss: 0.0096        Avg Val Loss: 0.0125        Val Accuracy: 9
Epoch: 17/30        Avg Train Loss: 0.0093        Avg Val Loss: 0.0122        Val Accuracy: 9
Epoch: 18/30        Avg Train Loss: 0.0089        Avg Val Loss: 0.0121        Val Accuracy: 9
Epoch: 19/30        Avg Train Loss: 0.0087        Avg Val Loss: 0.0120        Val Accuracy: 9
Epoch: 20/30        Avg Train Loss: 0.0084        Avg Val Loss: 0.0119        Val Accuracy: 9
Epoch: 21/30        Avg Train Loss: 0.0082        Avg Val Loss: 0.0118        Val Accuracy: 9
Epoch: 22/30        Avg Train Loss: 0.0087        Avg Val Loss: 0.0129        Val Accuracy: 9
Epoch: 23/30        Avg Train Loss: 0.0085        Avg Val Loss: 0.0118        Val Accuracy: 9
Epoch: 24/30        Avg Train Loss: 0.0079        Avg Val Loss: 0.0117        Val Accuracy: 9
Epoch: 25/30        Avg Train Loss: 0.0077        Avg Val Loss: 0.0116        Val Accuracy: 9
Epoch: 26/30        Avg Train Loss: 0.0075        Avg Val Loss: 0.0115        Val Accuracy: 9
Epoch: 27/30        Avg Train Loss: 0.0074        Avg Val Loss: 0.0115        Val Accuracy: 9
Epoch: 28/30        Avg Train Loss: 0.0072        Avg Val Loss: 0.0114        Val Accuracy: 9
Epoch: 29/30        Avg Train Loss: 0.0071        Avg Val Loss: 0.0114        Val Accuracy: 9
Epoch: 30/30        Avg Train Loss: 0.0070        Avg Val Loss: 0.0115        Val Accuracy: 9
```

```
In [256]: 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.
                      ###############################################################################
                      chars_new = [prepare_sequence([char for char in word], char_to_idx) for w
                      sentence_new = prepare_sequence(sentence, word_to_idx)
                      pred_tags = model(sentence_new, chars_new)
                      pred_tags = torch.argmax(pred_tags, dim = 1).tolist()
                      for tag_id in pred_tags:
                          predicted_tags.append(idx_to_tag[tag_id])

                      predicted_tags.append("")

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

```
test()
```

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

1.2.1 Test accuracy

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

The test accuracy I achieved on Gradescope was 92.4626%.

1.2.2 Error analysis

```
In [315]: ################################################################################
          # TODO: Generate predictions from val data
          # Create lists of words, tags predicted by the model and ground truth tags.
          ################################################################################
          def generate_predictions(model, val_data):
              # returns:: word_list (str list)
              # returns:: model_tags (str list)
              # returns:: gt_tags (str list)
              word_list = []
              model_tags = []
              gt_tags = []

              for sentence, tags in val_data:
                  chars_new = [prepare_sequence([char for char in word], char_to_idx) for word
                  sentence_new = prepare_sequence(sentence, word_to_idx)
                  pred_tags = model(sentence_new, chars_new)
                  pred_tags = torch.argmax(pred_tags, dim = 1).tolist()
                  pred_tags = [idx_to_tag[tag_id] for tag_id in pred_tags]

                  word_list += sentence
                  model_tags += pred_tags
                  gt_tags += tags
              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)
              errors = {}
              for index, model_tag in enumerate(model_tags):
                  gt_tag = gt_tags[index]
```

```
                word = word_list[index]
                if model_tag == gt_tag:
                    continue
                if (model_tag, gt_tag) not in errors:
                    errors[(model_tag, gt_tag)] = (0, set())
                curr_count, curr_word_list = errors[(model_tag, gt_tag)]
                curr_word_list.add(word)
                errors[(model_tag, gt_tag)] = (curr_count + 1, curr_word_list)
            errors = sorted(errors.items(), key = lambda x : x[1][0], reverse = True)[0:10]
            return errors

        charpos_model = model
        charpos_word_list, charpos_model_tags, charpos_gt_tags = generate_predictions(charpos
        charpos_errors = error_analysis(charpos_word_list, charpos_model_tags, charpos_gt_tag
        print("gt_tag\t| model_tag\t| freq.\t| examples")
        horizontal_line = "-------------------------------------------------------------------
        print(horizontal_line)
        for error in charpos_errors:
            tags, info = error
            model_tag, gt_tag = tags
            num_errors, example_words = info
            print("{}\t| {}\t\t| {}\t| {}\n{}".format(gt_tag, model_tag, num_errors, random.s
```

```
gt_tag        | model_tag        | freq.        | examples
-------------------------------------------------------------------------
VB        | NN               | 235        | ['mention', 'stress', 'bite', 'check', 'answer']
-------------------------------------------------------------------------
VBN        | VBD               | 229        | ['greeted', 'thought', 'studied', 'soared', 'co
-------------------------------------------------------------------------
NN        | JJ               | 220        | ['History', 'host', 'peso', 'relative', 'mettle']
-------------------------------------------------------------------------
VBD        | VBN               | 189        | ['replaced', 'exonerated', 'projected', 'paid',
-------------------------------------------------------------------------
NN        | VB               | 163        | ['probe', 'halt', 'austerity', 'face', 'journey']
-------------------------------------------------------------------------
VBP        | VB               | 130        | ['take', 'feel', 'participate', 'run', 'find']
-------------------------------------------------------------------------
JJ        | NNP               | 111        | ['Structural', 'Initial', 'Medical', 'Durable',
-------------------------------------------------------------------------
JJ        | NN               | 105        | ['hopeful', 'charming', 'stepped-up', 'standby',
-------------------------------------------------------------------------
WDT        | IN               | 93        | ['that']
-------------------------------------------------------------------------
VBZ        | NNS               | 90        | ['rubs', 'notes', 'accompanies', 'declines', 'po
-------------------------------------------------------------------------
```

Report your findings here.

What kinds of errors does the character-level model make as compared to the original model, and why do you think it made them?

The character model's second most frequent error, classified VBN tags as VBD, is the original model's most frequent error, so that's an interesting similarity. I suspect this will be a trend, even for the third model we train below.

However, one huge difference is the diversity in tag errors. In the original model, most of the errors came from verb errors (VBN, VB, VBD, VB, VBP), and then noun errors. While many of those errors still persist in this model, this model now frequently makes errors on a new tag, VBZ (verb, 3rd person singular present), more adjective tag errors, and another new tag, WDT (Wh-determiner). This last tag error, WDT, makes sense, because the model predicted it as IN (preposition or subordinating conjunction), which is definitely a possibility for the word "that". I think the real reason "that" is classified as "IN" instead of "WDT" is because 1) the word itself is pretty useless in the context of the entire sentence, so if the model had to make an error somewhere, it learned to make the error on the least useful words, such as "that", or 2) adding the character level embeddings might have placed too much emphasis on the "hat" substring of "that", and if the model has seen the word "what" a lot, which is different by 1 letter, then it could have been tricked into thinking "that" is similar to "what".

In general, I think the general trend is the same as the original model. This model does a poor job on ambiguous tags, for example within-group tags VBD and VBN. In fact, many of the same errors from the original model persisted in this model, but some were replaced for new errors on new tags, which is due to the character embeddings possibly placing too much emphasis/bias on the wrong portion of the word, as discussed in the above paragraph with "what" and "that".

1.3 Define a BiLSTM POS Tagger

A bidirectional LSTM that runs both left-to-right and right-to-left to represent dependencies between adjacent words in both directions and 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 example: - 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 different embedding method. For 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

```
In [268]: 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, lstm_laye
                  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.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tagset_size = tagset_size
        self.lstm_layers = lstm_layers

        self.embedding = nn.Embedding(num_embeddings = vocab_size, embedding_dim = se
        self.lstm = nn.LSTM(input_size = self.embedding_dim, hidden_size = self.hidde
        self.linear = nn.Linear(in_features = self.hidden_dim*2, out_features = tagse
        ############################################################################
        #                              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)
        ############################################################################
        batch_size = sentence.shape[0]
        self.hidden_cell = (torch.zeros(self.lstm_layers*2, batch_size, self.hidden_c
                            torch.zeros(self.lstm_layers*2, batch_size, self.hidden_

        embedding = self.embedding(sentence)
        embedding = embedding.unsqueeze(-1).permute(2, 0, 1)

        lstm_out, self.hidden_cell = self.lstm(embedding, self.hidden_cell)
        lstm_out = lstm_out.squeeze(0)
        tag_scores = self.linear(lstm_out)
        ############################################################################
        #                              END OF YOUR CODE                            #
        ############################################################################
        return tag_scores
```

In [291]:
```
############################################################################
# TODO: Initialize the model, optimizer and the loss function
############################################################################
EMBEDDING_DIM = 500
HIDDEN_DIM = 450
LSTM_LAYERS = 1
LEARNING_RATE = 0.01
DROPOUT = 0.0

bilstm_model = BiLSTMPOSTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_idx), len(tag_t
```

```
        bilstm_optimizer = optim.SGD(bilstm_model.parameters(), lr = LEARNING_RATE)
        loss_function = nn.CrossEntropyLoss()
        ##############################################################################
        #                                END OF YOUR CODE                            #
        ##############################################################################
        for epoch in range(1, EPOCHS + 1):
            train(epoch, bilstm_model, loss_function, bilstm_optimizer)
```

```
Epoch: 1/30        Avg Train Loss: 0.0761        Avg Val Loss: 0.0530        Val Accuracy: 68
Epoch: 2/30        Avg Train Loss: 0.0448        Avg Val Loss: 0.0405        Val Accuracy: 75
Epoch: 3/30        Avg Train Loss: 0.0357        Avg Val Loss: 0.0342        Val Accuracy: 79
Epoch: 4/30        Avg Train Loss: 0.0301        Avg Val Loss: 0.0298        Val Accuracy: 82
Epoch: 5/30        Avg Train Loss: 0.0258        Avg Val Loss: 0.0265        Val Accuracy: 84
Epoch: 6/30        Avg Train Loss: 0.0224        Avg Val Loss: 0.0239        Val Accuracy: 86
Epoch: 7/30        Avg Train Loss: 0.0197        Avg Val Loss: 0.0219        Val Accuracy: 87
Epoch: 8/30        Avg Train Loss: 0.0175        Avg Val Loss: 0.0204        Val Accuracy: 88
Epoch: 9/30        Avg Train Loss: 0.0157        Avg Val Loss: 0.0191        Val Accuracy: 88
Epoch: 10/30        Avg Train Loss: 0.0142        Avg Val Loss: 0.0181        Val Accuracy: 89
Epoch: 11/30        Avg Train Loss: 0.0130        Avg Val Loss: 0.0173        Val Accuracy: 89
Epoch: 12/30        Avg Train Loss: 0.0119        Avg Val Loss: 0.0166        Val Accuracy: 90
Epoch: 13/30        Avg Train Loss: 0.0110        Avg Val Loss: 0.0161        Val Accuracy: 90
Epoch: 14/30        Avg Train Loss: 0.0102        Avg Val Loss: 0.0156        Val Accuracy: 90
Epoch: 15/30        Avg Train Loss: 0.0096        Avg Val Loss: 0.0152        Val Accuracy: 91
Epoch: 16/30        Avg Train Loss: 0.0090        Avg Val Loss: 0.0149        Val Accuracy: 91
Epoch: 17/30        Avg Train Loss: 0.0085        Avg Val Loss: 0.0146        Val Accuracy: 91
Epoch: 18/30        Avg Train Loss: 0.0080        Avg Val Loss: 0.0143        Val Accuracy: 91
Epoch: 19/30        Avg Train Loss: 0.0077        Avg Val Loss: 0.0141        Val Accuracy: 91
Epoch: 20/30        Avg Train Loss: 0.0073        Avg Val Loss: 0.0140        Val Accuracy: 91
Epoch: 21/30        Avg Train Loss: 0.0070        Avg Val Loss: 0.0138        Val Accuracy: 91
Epoch: 22/30        Avg Train Loss: 0.0068        Avg Val Loss: 0.0137        Val Accuracy: 91
Epoch: 23/30        Avg Train Loss: 0.0065        Avg Val Loss: 0.0136        Val Accuracy: 91
Epoch: 24/30        Avg Train Loss: 0.0063        Avg Val Loss: 0.0135        Val Accuracy: 91
Epoch: 25/30        Avg Train Loss: 0.0061        Avg Val Loss: 0.0134        Val Accuracy: 92
Epoch: 26/30        Avg Train Loss: 0.0060        Avg Val Loss: 0.0133        Val Accuracy: 92
Epoch: 27/30        Avg Train Loss: 0.0058        Avg Val Loss: 0.0133        Val Accuracy: 92
Epoch: 28/30        Avg Train Loss: 0.0057        Avg Val Loss: 0.0132        Val Accuracy: 92
Epoch: 29/30        Avg Train Loss: 0.0055        Avg Val Loss: 0.0132        Val Accuracy: 92
Epoch: 30/30        Avg Train Loss: 0.0054        Avg Val Loss: 0.0132        Val Accuracy: 92
```

```
In [292]: 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.
            ###############################################################################
            sentence_new = prepare_sequence(sentence, word_to_idx)
            pred_tags = bilstm_model(sentence_new)
            pred_tags = torch.argmax(pred_tags, dim = 1).tolist()
            for tag_id in pred_tags:
                predicted_tags.append(idx_to_tag[tag_id])

            predicted_tags.append("")

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

test()
```

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

1.3.1 Test accuracy

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

I went ahead and chose the path to tune the hyperparameters. I went through 11 different combinations of hyperparameters with the following validation accuracies:

Embedding dim	Hidden dim	Num LSTM layers	Learning rate	Dropout	Val. Acc.
20	25	2	0.1	0.5	79%
35	40	4	0.1	0.1	87%
35	40	4	0.1	0.0	91%
35	40	4	0.2	0.0	91%
35	40	1	0.1	0.0	91%
60	70	1	0.1	0.0	91%
100	150	1	0.1	0.0	92%
100	150	1	0.075	0.0	91%
100	150	2	0.075	0.05	91%
300	150	1	0.1	0.0	92%
500	450	1	0.01	0.0	92%

Now, I've hit the 90% mark on the validation set, however, every time I submit my predictions on the Gradescope test set, the test accuracy is largely hanging around 89.75%. From the above simulations, I found a larger number of LSTM layers actually has no effect on performance or it has a negative effect. I suspect this is due to overfitting. A lower learning rate only really

helped me get a lower average validation loss, while still maintaining that 92% validation accuracy. Additionally, I noticed huge improvements in the model accuracy as the embedding and hidden dimensions went up. This makes sense because the model is able to embed more information per word, thus increasing inter-word separability. Finally, dropout was almost useless. I tried several values and its effect was almost completely useless. In fact, as I dropped the dropout rate, validation accuracy went up. I suspect this is because the model doesn't have an insane number of parameters, so performing dropout on large chunks might ruin the model, especially when comparing to larger deep learning models, such as ResNet.

In general, the best model is either the one with embedding dim 100, hidden dim 150, and val accuracy 92%, or the one with embedding dim 500 and hidden dim 450 with val accuracy 92%. When I plug the former's predictions into Gradescope, I get a test accuracy of 89.75%, and the latter is 89.57%. However, the former has a slightly higher average validation loss (0.0156 compared to 0.0132). So, I'd say the one with the larger embedding dim will probably be better in the long-term, especially if we decide to add more words in the future.

In [316]:
```
##############################################################################
# TODO: Generate predictions from val data
# Create lists of words, tags predicted by the model and ground truth tags.
##############################################################################
def generate_predictions(model, val_data):
    # returns:: word_list (str list)
    # returns:: model_tags (str list)
    # returns:: gt_tags (str list)
    word_list = []
    model_tags = []
    gt_tags = []

    for sentence, tags in val_data:
        sentence_new = prepare_sequence(sentence, word_to_idx)
        pred_tags = model(sentence_new)
        pred_tags = torch.argmax(pred_tags, dim = 1).tolist()
        pred_tags = [idx_to_tag[tag_id] for tag_id in pred_tags]

        word_list += sentence
        model_tags += pred_tags
        gt_tags += tags
    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)
    errors = {}
    for index, model_tag in enumerate(model_tags):
```

```
                gt_tag = gt_tags[index]
                word = word_list[index]
                if model_tag == gt_tag:
                    continue
                if (model_tag, gt_tag) not in errors:
                    errors[(model_tag, gt_tag)] = (0, set())
                curr_count, curr_word_list = errors[(model_tag, gt_tag)]
                curr_word_list.add(word)
                errors[(model_tag, gt_tag)] = (curr_count + 1, curr_word_list)
            errors = sorted(errors.items(), key = lambda x : x[1][0], reverse = True)[0:10]
            return errors

        bilstm_word_list, bilstm_model_tags, bilstm_gt_tags = generate_predictions(bilstm_mo
        bilstm_errors = error_analysis(bilstm_word_list, bilstm_model_tags, bilstm_gt_tags)
        print("gt_tag\t| model_tag\t| freq.\t| examples")
        horizontal_line = "-------------------------------------------------------------------
        print(horizontal_line)
        for error in bilstm_errors:
            tags, info = error
            model_tag, gt_tag = tags
            num_errors, example_words = info
            print("{}\t| {}\t\t| {}\t| {}\n{}".format(gt_tag, model_tag, num_errors, random.
```

```
gt_tag        | model_tag        | freq.        | examples
-------------------------------------------------------------------------
VB        | NN               | 173        | ['cooperate', 'defeat', 'block', 'rank', 'time']
-------------------------------------------------------------------------
VBN        | VBD               | 168        | ['signed', 'helped', 'destroyed', 'executed', '
-------------------------------------------------------------------------
VBD        | VBN               | 156        | ['singled', 'referred', 'opposed', 'hired', 'upl
-------------------------------------------------------------------------
NN        | VB               | 138        | ['scrap', 'wonder', 'finance', 'disturbance', 'thu
-------------------------------------------------------------------------
NN        | JJ               | 134        | ['misconduct', 'current', 'upsurge', 'donnybrook'
-------------------------------------------------------------------------
NNP        | NN               | 129        | ['Consumer', 'Keihin', 'Comanche', 'FREDERICK',
-------------------------------------------------------------------------
NN        | NNP               | 120        | ['overflow', 'assortment', 'sweetener', 'grade',
-------------------------------------------------------------------------
JJ        | NNP               | 120        | ['Medical', 'Milan-based', '30-share', 'rugged',
-------------------------------------------------------------------------
VB        | VBP               | 113        | ['assume', 'say', 'exclude', 'think', 'expire']
-------------------------------------------------------------------------
VBP        | VB               | 100        | ['lose', 'regret', 'rely', 'participate', 'try']
-------------------------------------------------------------------------
```

1.3.2 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 (a). If you tried multiple hyperparameter combinations, choose the model with the highest validation data accuracy. What errors does the original model make as compared to the modified model, and why do you think it made them?

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

I chose the model with 92% validation accuracy with an embedding dimension of 500 and hidden dimension of 450.

Once again, the same types of errors persist in this model. However, if we were to compare this model's errors to the character model and the original model, these errors are a lot more similar to the original model, which makes sense because the original model and this model do not take into account character embeddings, as such, they will naturally have more similar errors.

In terms of error analysis, when comparing to the original model, this model's top 10 errors are the exact same as the original model's top 10 errors, just in a slightly different order. It is important to note the frequencies of the errors are similar as well, which makes sense, since they achieved similar validation and test accuracies (86% and 89% test accuracies respectively for the original and Bi-LSTM model). All in all, the reason these errors agree with the original model is because this model is literally the information from the original model, but in two different forms (forward dependencies, which are in the original model, and then backward dependencies, which this model adds).

The only genuine difference I can see between the two models' errors is a slightly lower frequency for ground-truth JJ being predicted as NNP. This partially makes sense because adjectives can be placed on either side of a noun (e.g. "the heroic warrior fought ferociously" and "the warrior heroicly fought"). In this aforementioned example, the forward dependency will tag an adjective or verb coming right after the noun. However, in the Bi-LSTM, a backward dependency is added, which, in addition to the forward dependency, gathers an adjective or verb can come before the noun. As such, it is important to distinguish between the two cases (a word coming before another word, and a word coming after another word). Since the Bi-LSTM gathers both these dependencies, that might be why there are fewer errors in the (JJ, NNP) scenario. Similar reasoning can be used in the case of (VBN, VBD), which has a frequency of 168 in this model and a frequency of 181 in the original model. The use of neighboring words on both sides of a verb can make the difference between past tense and past participle. As such, these extra dependencies seem like they slightly reduce errors in select groups compared to the original model.

In summary, the Bi-LSTM model is pretty similar to the original model, just with the backwards dependency encoded now. As mentioned above, this leads to a reduction of errors in certain tag error groups, since neighboring words in both directions of the sentence makes a difference in terms of meaning/context of a sentence, and which tag belongs to a given word.

In [ ]: