Part of Speech Tagging with LSTM, Fine-tuned BERT

CS 4650 "Natural Language Processing" Project 2
Georgia Tech, Spring 2025 (Instructor: Weicheng Ma)

To start, first make a copy of this notebook to your local drive, so you can edit it.

If you want GPUs (which will improve training speed), you can always change your instance type to GPU by going to Runtime -> Change runtime type -> Hardware accelerator.

1. Basic POS Tagger [15 points]

In this assignment, we will train LSTM-based POS-taggers, and evaluate their performance. We will use English text from the Wall Street Journal, marked with POS tags such as `NNP` (proper noun) and `DT` (determiner).

1.1 Setup

```
!curl -L -o train.txt
"https://www.dropbox.com/scl/fi/nqtk53b2ihqzf6hugolms/train.txt?
rlkey=y7003b74z7gp06e8qa2gfgd4r&st=37lt5q66&dl=0"

  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:--
--:--:--     0
100    17  100    17    0     0     37      0 --:--:-- --:--:--
--:--:--    37
100    17  100    17    0     0     37      0 --:--:-- --:--:--
--:--:--    37

  0 2991k    0     0    0     0      0      0 --:--:--  0:00:01
--:--:--     0
100 2991k  100 2991k    0     0  2016k      0  0:00:01  0:00:01
--:--:--  9.8M

#
===================================================================
=====
# Run some setup code for this notebook. Don't modify anything in this
cell.
```

```python
#
==========================================================================
=====

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

import random

RANDOM_SEED = 42
torch.manual_seed(RANDOM_SEED)
random.seed(RANDOM_SEED)

#
==========================================================================
=====
# A quick note on CUDA functionality (and `.to(model.device)`):
# CUDA is a parallel GPU platform produced by NVIDIA and is used by
most GPU
# libraries in PyTorch. CUDA organizes GPUs into device IDs (i.e.,
"cuda:X" for GPU #X).
# "device" will tell PyTorch which GPU (or CPU) to place an object in.
Since
# collab only uses one GPU, we will use 'cuda' as the device if a GPU
is available
# and the CPU if not. You will run into problems if your tensors are
on different devices.
#
==========================================================================
=====
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

You can check to make sure a GPU is available using the following code block.

```python
# If the below message is shown, it means you are using a CPU.
/bin/bash: nvidia-smi: command not found

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

```
'nvidia-smi' is not recognized as an internal or external command,
operable program or batch file.
```

1.2 Preparing Data

`train.txt`: The training data is present in this file. This file contains sequences of words and their respective tags. The data is split into 80% training and 20% development to train the model and tune the hyperparameters, respectively. See `load_tag_data` for details on how to read the training data.

```
# 
==========================================================================
=====
# Run some preprocessing code for our dataset. Don't modify anything 
in this cell.
# 
==========================================================================
=====

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

train_sentences, train_tags = load_tag_data('train.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("Total tags: ", len(unique_tags))
```

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

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.

```
#
======================================================================
=====
# Don't modify anything in this cell.
#
======================================================================
=====

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)

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 19122
```

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

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. Here we define a few default hyperparameters for your model.

```
EMBEDDING_DIM = 4
HIDDEN_DIM = 8
LEARNING_RATE = 0.1
```

```
LSTM_LAYERS = 1
DROPOUT = 0
EPOCHS = 10
```

1.5 Define Model [5 points]

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 representation for each word. Then the representations of words are passed to a Linear layer.

```
class BasicPOSTagger(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size,
tagset_size):
        """
        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 states
          a linear layer: maps from hidden state space to tag space
        """
        super(BasicPOSTagger, self).__init__()

        ### BEGIN YOUR CODE ###
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, LSTM_LAYERS,
batch_first=True)
        self.linear = nn.Linear(hidden_dim, tagset_size)
        ### END YOUR CODE ###

    def forward(self, sentence):
        """
        Implement the forward pass.

        Given a tokenized index-mapped sentence as the argument,
        compute the corresponding raw scores for tags (without
softmax)

        returns:: tag_scores (Tensor)
        """
        tag_scores = None

        ### BEGIN YOUR CODE ###
        embeddings = self.embedding(sentence)
        output, (h_n, c_n) = self.lstm(embeddings)
        tag_scores = self.linear(output)
```

```
        ### END YOUR CODE ###

        return tag_scores
```

1.6 Training [5 points]

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

```
def train(epoch, model, loss_function, optimizer):
    model.train()
    train_loss = 0
    train_examples = 0
    for sentence, tags in training_data:
        """
        Implement the training method

        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.
        """

        ### BEGIN YOUR CODE ###

        # Zero out the parameter gradients
        optimizer.zero_grad()

        # Prepare input data (sentences and gold labels)
        input = prepare_sequence(sentence, word_to_idx)
        target = prepare_sequence(tags, tag_to_idx)

        # Do forward pass with current batch of input
        tag_scores = model(input)

        # Get loss with model predictions and true labels
        loss = loss_function(tag_scores.view(-1, tag_scores.shape[-
1]), target.view(-1))
        loss.backward()

        # Update model parameters
        optimizer.step()

        # Increase running total loss and the number of past training
samples
        train_loss += loss.item()
        train_examples += len(sentence)
```

```
        ### END YOUR CODE ###

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

    print(f"Epoch: {epoch}/{EPOCHS}\tAvg Train Loss:
{avg_train_loss:.4f}\tAvg Val Loss: {avg_val_loss:.4f}\t Val Accuracy:
{val_accuracy:.0f}")

def evaluate(model, loss_function):
    """
    returns:: avg_val_loss (float)
    returns:: val_accuracy (float)
    """
    model.eval()
    correct = 0
    val_loss = 0
    val_examples = 0
    with torch.no_grad():
        for sentence, tags in val_data:
            """
            Implement the evaluate method

            Find the average validation loss along with the validation
accuracy.
            Hint: To find the accuracy, argmax of tag predictions can
be used.s
            """
            ### BEGIN YOUR CODE ###

            # Prepare input data (sentences and gold labels)
            input = prepare_sequence(sentence, word_to_idx)
            target = prepare_sequence(tags, tag_to_idx)

            # Do forward pass with current batch of input
            tag_scores = model(input)

            # Get loss with model predictions and true labels
            loss = loss_function(tag_scores.view(-1,
tag_scores.shape[-1]), target.view(-1))

            # Get the predicted labels
            _, predicted = torch.max(tag_scores, dim=1)

            # Get number of correct prediction
            correct += (predicted.view(-1) == target.view(-
1)).sum().item()

            # Increase running total loss and the number of past valid
samples
```

```
            val_loss += loss.item()
            val_examples += len(sentence)

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

"""
Initialize the model, optimizer and the loss function
"""
### BEGIN YOUR CODE ###
model = BasicPOSTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_idx),
len(tag_to_idx))
loss_function = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), LEARNING_RATE, momentum=0.8)

### END YOUR CODE ###

for epoch in range(1, EPOCHS + 1):
    train(epoch, model, loss_function, optimizer)
```

```
Epoch: 1/10     Avg Train Loss: 0.0734     Avg Val Loss: 0.0580   Val
Accuracy: 59
Epoch: 2/10     Avg Train Loss: 0.0526     Avg Val Loss: 0.0489   Val
Accuracy: 65
Epoch: 3/10     Avg Train Loss: 0.0443     Avg Val Loss: 0.0432   Val
Accuracy: 69
Epoch: 4/10     Avg Train Loss: 0.0384     Avg Val Loss: 0.0379   Val
Accuracy: 74
Epoch: 5/10     Avg Train Loss: 0.0337     Avg Val Loss: 0.0348   Val
Accuracy: 78
Epoch: 6/10     Avg Train Loss: 0.0301     Avg Val Loss: 0.0323   Val
Accuracy: 80
Epoch: 7/10     Avg Train Loss: 0.0274     Avg Val Loss: 0.0305   Val
Accuracy: 82
Epoch: 8/10     Avg Train Loss: 0.0255     Avg Val Loss: 0.0296   Val
Accuracy: 83
Epoch: 9/10     Avg Train Loss: 0.0237     Avg Val Loss: 0.0286   Val
Accuracy: 84
Epoch: 10/10    Avg Train Loss: 0.0223     Avg Val Loss: 0.0281   Val
Accuracy: 84
```

Hint: Under the default hyperparameter setting, after 5 epochs you should be able to get at least `0.75` accuracy on the validation set.

1.7 Error analysis [5 points]

In this step, we will analyze what kind of errors it was making on the validation set.

Step 1, write a method to generate predictions from the validation set. For every sentence, get its words, predicted tags (model_tags), and the ground truth tags (gt_tags). To make the next step easier, you may want to concatenate words from all sentences into a very long list, and same for model_tags and gt_tags.

Step 2, analyze what kind of errors the model was making. For example, it may frequently label NN as VB. Let's get the top-10 most frequent types of errors, each of their frequency, and some example words. One example is at below. It is interpreted as the model predicts NNP as VBG for 626 times, five random example words are shown.

```
['VBG', 'NNP', 626, ['Rowe', 'Livermore', 'Parker', 'F-16', 'HEYNOW']]

def generate_predictions(model, val_data):
    """
    Generate predictions for val_data

    Create lists of words, tags predicted by the model and ground
truth tags.
    Hint: It should look very similar to the evaluate function.

    returns:: word_list (str list)
    returns:: model_tags (str list)
    returns:: gt_tags (str list)
    """
    ### BEGIN YOUR CODE ###
    model.eval()
    word_list = []
    model_tags = []
    gt_tags = []

    with torch.no_grad():
       for sentence, tags in val_data:
          input = prepare_sequence(sentence, word_to_idx)
          tag_scores = model(input)

          _, predicted = torch.max(tag_scores, dim=1)

          word_list.extend(sentence)
          model_tags.extend([idx_to_tag[idx.item()] for idx in
predicted])
          gt_tags.extend(tags)
    ### END YOUR CODE ###

    return word_list, model_tags, gt_tags

def error_analysis(word_list, model_tags, gt_tags):
    """"
    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

    returns: errors (list of tuples)
    """
    ### BEGIN YOUR CODE ###
    error_count = {}
    error_eg = {}

    for i in range(len(word_list)):
       pred_tag = model_tags[i]
       gt_tag = gt_tags[i]
       word = word_list[i]

       if pred_tag != gt_tag:
        key = (pred_tag, gt_tag)

        if key in error_count:
            error_count[key] += 1
            if len(error_eg[key]) < 5:
                    error_eg[key].append(word)
        else:
           error_count[key] = 1
           error_eg[key] = [word]

    errors = []
    for key in error_count:
        errors.append((key[0], key[1], error_count[key],
error_eg[key]))

    errors.sort(key=lambda x: x[2], reverse=True)
    ### END YOUR CODE ###

    return errors

word_list, model_tags, gt_tags = generate_predictions(model, val_data)
errors = error_analysis(word_list, model_tags, gt_tags)

for i in errors[:10]:
  print(i)
```

```
('NN', 'NNP', 334, ['Bateman', 'Bryan', 'Galles', 'mature', 'ANC'])
('NNS', 'NNP', 299, ['Michigan', 'Barbie', 'Wheels', 'Investor',
'Masius'])
('NN', 'JJ', 242, ['ever-narrowing', 'lengthy', 'unrealistic', '60-
inch', 'much-beloved'])
('NN', 'NNS', 224, ['authorities', 'misrepresentations', 'ounces',
'weeks', 'gases'])
('JJ', 'NN', 198, ['headquarters', 'commercial', 'stockpile', 'net',
'many'])
```

```
('NNS', 'NN', 198, ['humor', 'reflection', 'tandem', 'exception', 're-election'])
('NNP', 'JJ', 194, ['California', 'dense', 'solar', '30-year', '30-year'])
('JJ', 'NNP', 176, ['Ivy', 'League', 'Telerate', 'Hickman', 'Allenport'])
('NNP', 'NN', 176, ['corridor', 'depressant', 'Market', 'chicken', 'Energy'])
('NNS', 'VBG', 175, ['buying', 'laughing', 'pushing', 'closing', 'closing'])
```

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

It frequently misclassifies certain noun forms and adjectives. For instance, proper nouns were misclassified as adjectives, and plural nouns were misclassified as singular nouns. It likely made these errors because many words can have multiple possible POS tags depending on context, and hence it struggles with ambiguous words. Another possible factor is that the model only uses one LSTM layer with no bidirectional processing, therefore not being able to capture sufficient context.

2. Hyper-parameter Tuning [10 points]

In order to improve your model performance, try making some modifications on `EMBEDDING_DIM`, `HIDDEN_DIM`, and `LEARNING_RATE`.

```
YOUR_EMBEDDING_DIM = 32
YOUR_HIDDEN_DIM = 64
YOUR_LEARNING_RATE = 0.01

# Set three hyper-parameters. Initialize the model, optimizer and the loss function
# Hint, you may want to use reduction='sum' in the CrossEntropyLoss function

### BEGIN YOUR CODE ###
model = BasicPOSTagger(YOUR_EMBEDDING_DIM, YOUR_HIDDEN_DIM, len(word_to_idx), len(tag_to_idx))
loss_function = nn.CrossEntropyLoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=YOUR_LEARNING_RATE, momentum=0.9)

### END YOUR CODE ###

for epoch in range(1, EPOCHS + 1):
    train(epoch, model, loss_function, optimizer)
```

```
Epoch: 1/10     Avg Train Loss: 0.7597     Avg Val Loss: 0.5029   Val Accuracy: 86
```

```
Epoch: 2/10      Avg Train Loss: 0.3763      Avg Val Loss: 0.4025   Val
Accuracy: 89
Epoch: 3/10      Avg Train Loss: 0.2441      Avg Val Loss: 0.3777   Val
Accuracy: 91
Epoch: 4/10      Avg Train Loss: 0.1678      Avg Val Loss: 0.3787   Val
Accuracy: 92
Epoch: 5/10      Avg Train Loss: 0.1253      Avg Val Loss: 0.3877   Val
Accuracy: 92
Epoch: 6/10      Avg Train Loss: 0.1001      Avg Val Loss: 0.4057   Val
Accuracy: 92
Epoch: 7/10      Avg Train Loss: 0.0839      Avg Val Loss: 0.4117   Val
Accuracy: 92
Epoch: 8/10      Avg Train Loss: 0.0780      Avg Val Loss: 0.4329   Val
Accuracy: 92
Epoch: 9/10      Avg Train Loss: 0.0756      Avg Val Loss: 0.4372   Val
Accuracy: 92
Epoch: 10/10     Avg Train Loss: 0.0696      Avg Val Loss: 0.4500   Val
Accuracy: 92
```

3. Character-level POS Tagger [15 points]

Use the character-level information to augment word embeddings. For example, words that end with -ing or -ly give quite a bit of information about their POS tags. To incorporate this information, run a character-level LSTM on every word to create a character-level representation of the word. Take the last hidden state from the character-level LSTM as the representation and concatenate with the word embedding (as in the `BasicPOSTagger`) to create a new word representation that captures more information.

```python
# 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)
```

An Aside on Padding

How to do padding correctly for the characters?

Assume we have got a sentence ["We", "love", "NLP"]. You are supposed to first prepend a certain number of blank characters to each of the words in this sentence.

How to determine the number of blank characters we need? The calculation of MAX_WORD_LEN is here for help (which we already provide in the starter code). For the given sentence, MAX_WORD_LEN equals 4. Therefore we prepend two blank characters to "We", zero blank character to "love", and one blank character to "NLP". So the resultant padded sentence we get should be [" We", "love", " NLP"].

Then, we feed all characters in [" We", "love", " NLP"] into a char-embedding layer, and get a tensor of shape (3, 4, char_embedding_dim). To make this tensor's shape proper for the char-level LSTM (nn.LSTM), we need to transpose this tensor, i.e. swap the first and the second dimension. So we get a tensor of shape (4, 3, char_embedding_dim), where 4 corresponds to seq_len and 3 corresponds to batch_size.

The last thing you need to do is to obtain the last hidden state from the char-level LSTM, and concatenate it with the word embedding, so that you can get an augmented representation of that word.

An illustration for left padding characters

Why doing the padding?

Someone may ask why we want to do such a kind of padding, instead of directly passing each of the character sequences of each word one by one through an LSTM, to get the last hidden state. The reason is that if you don't do padding, then that means you can only implement this process using "for loop". For CharPOSTagger, if you implement it using "for loop", the training time would be approximately 150s (GPU) / 250s (CPU) per epoch, while it would be around 30s (GPU) / 150s (CPU) per epoch if you do the padding and feed your data in batches. Therefore, we strongly recommend you learn how to do the padding and transform your data into batches. In fact, those are quite important concepts which you should get yourself familar with, although it might take you some time.

Why doing *left* padding?

Our hypothesis is that the suffixes of English words (e.g., -ly, -ing, etc) are more indicative than prefixes for the part-of-speech (POS). Though LSTM is supposed to be able to handle long sequences, it still lose information along the way and the information closer to the last state (which you use as char-level representations) will be retained better.

How to understand the dimention change?

Assume we have got a sentence with 3 words ["We", "love", "NLP"], and assume the dimension of character embedding is 2, the dimension of word embedding is 4, the dimension of word-level LSTM's hidden layer is 5, the dimension of character-level LSTM's hidden layer is 6.

In `BasicPOSTagger`, the dimension change would be:

- ------ input ------> $(3 \times 1 \times 4)$
- -- word-level LSTM --> $(3 \times 1 \times 5)$
- ----- linear layer -----> $(3 \times 1 \times 44)$

In `CharPOSTagger`, after padding, character embedding, and swapping, the dimension change would be:

- ------ input ------> ¿ MAX_WORD_LEN $\times 3 \times 2$¿
- -- character-level LSTM --> ¿ MAX_WORD_LEN $\times 3 \times 6$¿
- -- Take the last hidden state --> (3×6)
- -- concatenate with word embedings --> $(3 \times 1 \times 10)$
- -- word-level LSTM --> $(3 \times 1 \times 5)$
- -- linear layer --> $(3 \times 1 \times 44)$.

```
EMBEDDING_DIM = 4
HIDDEN_DIM = 8
LEARNING_RATE = 0.1
LSTM_LAYERS = 1
DROPOUT = 0
EPOCHS = 10
CHAR_EMBEDDING_DIM = 4
CHAR_HIDDEN_DIM = 4
```

3.1 Define Model [5 points]

```
class CharPOSTagger(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, char_embedding_dim,
                 char_hidden_dim, char_size, vocab_size, tagset_size):
        """
        Define and initialize anything needed for the forward pass.

        You are required to create a model with:
```

```
          an embedding layer for word: that maps words to their
embedding space
          an embedding layer for character: that maps characters to
their embedding space
          a character-level LSTM layer: that finds the character-level
embedding for a word
          a word-level LSTM layer: that takes the concatenated
representation per word (word embedding + char-lstm) as input and
outputs hidden states
          a linear layer: maps from hidden state space to tag space
        """
        super(CharPOSTagger, self).__init__()

        ### BEGIN YOUR CODE ###
        self.word_embedding = nn.Embedding(vocab_size, embedding_dim)
        self.char_embedding = nn.Embedding(char_size,
char_embedding_dim)

        self.char_lstm = nn.LSTM(char_embedding_dim, char_hidden_dim,
LSTM_LAYERS, batch_first=True)
        self.word_lstm = nn.LSTM(embedding_dim + char_hidden_dim,
hidden_dim, LSTM_LAYERS, batch_first=True)

        self.linear = nn.Linear(hidden_dim, tagset_size)
        ### END YOUR CODE ###

    def forward(self, sentence, chars):
        tag_scores = None
        """
        Implement the forward pass.

        Given a tokenized index-mapped sentence and a character
sequence as the arguments,
        find the corresponding raw scores for tags (without softmax)

        returns:: tag_scores (Tensor)
        """

        ### BEGIN YOUR CODE ###
        word_embeddings = self.word_embedding(sentence).unsqueeze(0)
        batch_size, seq_len, max_word_len = chars.shape
        chars = chars.view(-1, max_word_len)
        char_embeddings = self.char_embedding(chars)

        c_out, (c_hn, c_cn) = self.char_lstm(char_embeddings)
        c_hn = c_hn.squeeze(0)

        c_hn = c_hn.view(batch_size, seq_len, -1)
```

```
        combined = torch.cat((word_embeddings, c_hn), dim=2)

        w_out, (w_hn, w_cn) = self.word_lstm(combined)

        tag_scores = self.linear(w_out)
        ### END YOUR CODE ###

        return tag_scores
```

3.2 Training [5 points]

```
def train_char(epoch, model, loss_function, optimizer):
    model.train()
    train_loss = 0
    train_examples = 0
    for sentence, tags in training_data:
        """
        Implement the training method

        Hint: you can use the prepare_sequence method for creating 
index mappings
          for sentences. For constructing character input, you may 
want to left pad
          each word to MAX_WORD_LEN first, then use prepare_sequence 
method to create
          index  mappings.
        """

        ### BEGIN YOUR CODE ###

        # Zero out the parameter gradients
        optimizer.zero_grad()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)

        # Prepare input data (sentences, characters, and gold labels)
        input_words = prepare_sequence(sentence, word_to_idx)

        input_chars = [[char_to_idx[c] if c in char_to_idx else 
char_to_idx[' '] for c in word] for word in sentence]
        max_word_len = max(len(word) for word in sentence)
        input_chars = [[0] * (max_word_len - len(chars)) + chars for 
chars in input_chars]
        input_chars = torch.tensor(input_chars).unsqueeze(0)

        target = prepare_sequence(tags, tag_to_idx)

        # Do forward pass with current batch of input
        tag_scores = model(input_words, input_chars)

        # Get loss with model predictions and true labels
```

```
        loss = loss_function(tag_scores.view(-1, tag_scores.shape[-
1]), target.view(-1))

        # Update model parameters
        loss.backward()
        optimizer.step()

        # Increase running total loss and the number of past training
samples
        train_loss += loss.item()
        train_examples += len(sentence)

        ### END YOUR CODE ###

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

    print(f"Epoch: {epoch}/{EPOCHS}\tAvg Train Loss:
{avg_train_loss:.4f}\tAvg Val Loss: {avg_val_loss:.4f}\t Val Accuracy:
{val_accuracy:.0f}")

def evaluate_char(model, loss_function):
    """
    returns:: avg_val_loss (float)
    returns:: val_accuracy (float)
    """
    model.eval()
    correct = 0
    val_loss = 0
    val_examples = 0
    with torch.no_grad():
        for sentence, tags in val_data:
            """
            Implement the evaluate method. Find the average validation
loss
            along with the validation accuracy.

            Hint: To find the accuracy, argmax of tag predictions can
be used.
            """

            ### BEGIN YOUR CODE ###

            # Prepare input data (sentences, characters, and gold
labels)
            input_words = prepare_sequence(sentence, word_to_idx)

            input_chars = [[char_to_idx[c] if c in char_to_idx else
char_to_idx[' '] for c in word] for word in sentence]
            max_word_len = max(len(word) for word in sentence)
```

```
            input_chars = [[0] * (max_word_len - len(chars)) + chars
for chars in input_chars]
            input_chars = torch.tensor(input_chars).unsqueeze(0)

            target = prepare_sequence(tags, tag_to_idx)

            # Do forward pass with current batch of input
            tag_scores = model(input_words, input_chars)

            # Get loss with model predictions and true labels
            loss = loss_function(tag_scores.view(-1,
tag_scores.shape[-1]), target.view(-1))

            # Get the predicted labels
            _, predicted = torch.max(tag_scores, dim=2)

            # Get number of correct prediction
            correct += (predicted.view(-1) == target.view(-
1)).sum().item()

            # Increase running total loss and the number of past valid
samples
            val_loss += loss.item()
            val_examples += len(sentence)

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

# Initialize the model, optimizer and the loss function
# Hint, you may want to use reduction='sum' in the CrossEntropyLoss
function

### BEGIN YOUR CODE ###
model = CharPOSTagger(EMBEDDING_DIM, HIDDEN_DIM, CHAR_EMBEDDING_DIM,
CHAR_HIDDEN_DIM, len(char_to_idx), len(word_to_idx), len(tag_to_idx))
loss_function = nn.CrossEntropyLoss(reduction='mean')
optimizer = torch.optim.SGD(model.parameters(), LEARNING_RATE,
momentum=0.9)
### END YOUR CODE ###

for epoch in range(1, EPOCHS + 1):
    train_char(epoch, model, loss_function, optimizer)
```

```
Epoch: 1/10     Avg Train Loss: 0.0488     Avg Val Loss: 0.0332   Val
Accuracy: 76
Epoch: 2/10     Avg Train Loss: 0.0305     Avg Val Loss: 0.0276   Val
Accuracy: 80
Epoch: 3/10     Avg Train Loss: 0.0258     Avg Val Loss: 0.0243   Val
Accuracy: 83
```

```
Epoch: 4/10     Avg Train Loss: 0.0216     Avg Val Loss: 0.0217   Val
Accuracy: 86
Epoch: 5/10     Avg Train Loss: 0.0189     Avg Val Loss: 0.0201   Val
Accuracy: 87
Epoch: 6/10     Avg Train Loss: 0.0163     Avg Val Loss: 0.0184   Val
Accuracy: 88
Epoch: 7/10     Avg Train Loss: 0.0195     Avg Val Loss: 0.0215   Val
Accuracy: 87
Epoch: 8/10     Avg Train Loss: 0.0178     Avg Val Loss: 0.0204   Val
Accuracy: 88
Epoch: 9/10     Avg Train Loss: 0.0161     Avg Val Loss: 0.0220   Val
Accuracy: 87
Epoch: 10/10    Avg Train Loss: 0.0154     Avg Val Loss: 0.0205   Val
Accuracy: 89
```

Hint: Under the default hyperparameter setting, after 5 epochs you should be able to get at least `0.85` accuracy on the validation set.

3.3 Error analysis [5 points]

Write a method to generate predictions for the validation set. Create lists of words, tags predicted by the model and ground truth tags.

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

This part is very similar to part 1.7. You may want to refer to your implementation there.

```python
def generate_predictions(model, val_data):
    """
    Generate predictions for val_data

    Create lists of words, tags predicted by the model and ground
truth tags.
    Hint: It should look very similar to the evaluate function.

    returns:: word_list (str list)
    returns:: model_tags (str list)
    returns:: gt_tags (str list)
    """
    ### BEGIN YOUR CODE ###
    model.eval()
    word_list = []
    model_tags = []
    gt_tags = []
    with torch.no_grad():
        for sentence, tags in val_data:
            input = prepare_sequence(sentence, word_to_idx)

            input_chars = [[char_to_idx[c] if c in char_to_idx else
```

```
char_to_idx[' '] for c in word] for word in sentence]
            max_word_len = max(len(word) for word in sentence)
            input_chars = [[0] * (max_word_len - len(chars)) + chars
for chars in input_chars]
            input_chars = torch.tensor(input_chars).unsqueeze(0)
            tag_scores = model(input, input_chars)

            _, predicted = torch.max(tag_scores, dim=2)
            predicted = predicted.view(-1)

            word_list.extend(sentence)
            model_tags.extend([idx_to_tag[idx.item()] for idx in
predicted])
            gt_tags.extend(tags)


    ### END YOUR CODE ###

    return word_list, model_tags, gt_tags

def error_analysis(word_list, model_tags, gt_tags):
    """
    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

    returns: errors (list of tuples)
    """
    ### BEGIN YOUR CODE ###
    error_count = {}
    error_eg = {}

    for i in range(len(word_list)):
        pred_tag = model_tags[i]
        gt_tag = gt_tags[i]
        word = word_list[i]

        if pred_tag != gt_tag:
            key = (pred_tag, gt_tag)

            if key in error_count:
                error_count[key] += 1
                if len(error_eg[key]) < 5:
                    error_eg[key].append(word)
            else:
                error_count[key] = 1
                error_eg[key] = [word]
```

```
    errors = []
    for key in error_count:
        errors.append((key[0], key[1], error_count[key],
error_eg[key]))

    errors.sort(key=lambda x: x[2], reverse=True)
    ### END YOUR CODE ###

    return errors

word_list, model_tags, gt_tags = generate_predictions(model, val_data)
errors = error_analysis(word_list, model_tags, gt_tags)

for i in errors[:10]:
  print(i)
```

```
('NNP', 'NN', 388, ['hotdog', 'humor', 'abortion', 'abortion',
'procedure'])
('NNP', 'VB', 332, ['extract', 'haul', 'Get', 'write', 'become'])
('NN', 'DT', 326, ['The', 'The', 'The', 'The', 'The'])
('NNP', 'JJ', 292, ['slick-talking', 'snake-oil', 'Initial', 'ever-
narrowing', 'bargain-basement'])
('NN', 'JJ', 291, ['gullible', 'greedy', 'unrealistic', '60-inch',
'deflationary'])
('VBZ', 'NNS', 267, ['buddies', 'portions', 'medicines', 'pleas',
'loopholes'])
('NN', 'VBG', 243, ['declining', 'buying', 'collapsing',
'contemplating', 'pushing'])
('JJ', 'VB', 243, ['raise', 'boost', 'make', 'have', 'enact'])
('VBD', 'VBN', 241, ['alleged', 'made', 'adjusted', 'ended', 'made'])
('JJ', 'CD', 232, ['zero', '1992', '334,000', '52', '3.75'])
```

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-level model frequently misclassifies common nouns (NN) as proper nouns (NNP) and vice versa. This suggests that it may rely on capitalization patterns but struggles with unusual or context-dependent proper nouns. The original model on the other hand misclassifies named entities possibly due to limited exposure to specific proper names in the training data.

4. Fine-tuned BERT POS Tagger [Extra Credit - 5 points]

In the above sections, we trained sequence-based models for POS tagging on a fairly limited dataset of *labeled* part of speech data. However, we can imagine the model is having to both learn the basics of language *and* part of speech tagging simultaneously. Perhaps, we can use a model pre-trained on a much larger corpus of language, and *fine-tune* the model on our specific task.

For this, we can use **BERT** (see *Pre-training of Deep Bidirectional Transformers for Language Understanding* NAACL, 2019). BERT introduces a method of pre-training a transformer encoder and fine-tuning the encoder on downstream tasks, and is extrordinarily infuential in NLP research and engineering (e.g., `bert-base-uncased` has 45M downloads per month from Huggingface). The core idea is *transfer learning*, or that pre-training on a self-supervised mask language modeling objective can help with our downstream language task of POS tagging. For a step-by-step introduction to the BERT architecture, please see Jay Almmar's The Illustrated BERT.

This section will walk you through the use of the popular **Huggingface Transformers** library (see *Transformers: State-of-the-Art Natural Language Processing*, the HuggingFace Documentation and Abhishek Mishra's HF tutorial), which is a widely used library for distributing and using transformer models. Luckily, we can think of the HuggingFace library as a wrapper on top of PyTorch, so these sections should look familiar to your work so far.

For this extra credit section, we will use a pre-trained BERT model, and fine-tune it on the POS tagging task.

4.1 Install `transformers` and download DistilBERT

For your fine-tuning code to run a bit faster, we will use a smaller "distilled" version of BERT called **DistilBERT** (see *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*). Fortunately with the `transformers` library, we could swap out the underlying model with no code changes to our dataloaders, architecture or traning setup!

```
!pip install -qU tokenizers transformers
```

```
  WARNING: Failed to write executable - trying to use .deleteme logic
ERROR: Could not install packages due to an OSError: [WinError 2] The system cannot find the file specified: 'C:\\Python312\\Scripts\\normalizer.exe' -> 'C:\\Python312\\Scripts\\normalizer.exe.deleteme'


[notice] A new release of pip is available: 24.2 -> 25.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
# If you are interested in what other models are available, you can find a
# list of model names here (e.g., roberta-base, bert-base-uncased):
# https://huggingface.co/transformers/pretrained_models.html

from transformers import DistilBertModel, DistilBertTokenizerFast
bert_model = DistilBertModel.from_pretrained('distilbert-base-uncased')
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
```

```
c:\Python312\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
c:\Python312\Lib\site-packages\huggingface_hub\file_download.py:142:
UserWarning: `huggingface_hub` cache-system uses symlinks by default
to efficiently store duplicated files but your machine does not
support them in C:\Users\Wei Xuan\.cache\huggingface\hub\models--
distilbert-base-uncased. Caching files will still work but in a
degraded version that might require more space on your disk. This
warning can be disabled by setting the
`HF_HUB_DISABLE_SYMLINKS_WARNING` environment variable. For more
details, see https://huggingface.co/docs/huggingface_hub/how-to-
cache#limitations.
To support symlinks on Windows, you either need to activate Developer
Mode or to run Python as an administrator. In order to activate
developer mode, see this article:
https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-
device-for-development
  warnings.warn(message)
```

```
# Let's take a look at our DistilBERT architecture
bert_model
```

```
DistilBertModel(
  (embeddings): Embeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (transformer): Transformer(
    (layer): ModuleList(
      (0-5): 6 x TransformerBlock(
        (attention): DistilBertSdpaAttention(
          (dropout): Dropout(p=0.1, inplace=False)
          (q_lin): Linear(in_features=768, out_features=768,
bias=True)
          (k_lin): Linear(in_features=768, out_features=768,
bias=True)
          (v_lin): Linear(in_features=768, out_features=768,
bias=True)
          (out_lin): Linear(in_features=768, out_features=768,
bias=True)
        )
        (sa_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        (ffn): FFN(
          (dropout): Dropout(p=0.1, inplace=False)
          (lin1): Linear(in_features=768, out_features=3072,
bias=True)
          (lin2): Linear(in_features=3072, out_features=768,
bias=True)
          (activation): GELUActivation()
```

```
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
      )
    )
)
```

4.2 Load the dataset with a PyTorch dataloader

Please take a look at the `bert-base-cased` tokenizer on the Tokenizer Playground. Our goal will be to predict the POS of each word, but BERT is trained on sub-word tokens, so we need to segment our dataset such that **only the first token of each word is classified**.

```
from torch.utils.data import Dataset

class POSDataset(Dataset):
  def __init__(self, data, tokenizer, max_len):
    self.data = data
    self.tokenizer = tokenizer
    self.max_len = max_len

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    """
    Given an index, return the value in your training data
(self.data). Make
    sure the full output dict from self.tokenizer is returned, with an
additional
    value for your labels.

    Remember! Your BERT tokenizer will give multiple tokens to words
with the
    same POS tag. We want the FIRST token be given the tag and all
other tokens
    to be given -100.

    Hint: You may use the prepare_sequence() function from earlier
sections
    Hint: Our training data is already tokenized, so you may find the
`is_split_into_words=True`
      and `return_offsets_mapping=True` arguments helpful for getting
the token offsets.
    Hint: When using the tokenizer, you can also use
padding='max_length' for [PAD]
      tokens to be added for you.
    """
```

```
    encoding = None

    ### BEGIN YOUR CODE ###

    # Get the sentence and POS tags
    sentence, tags = self.data[index]
    tag_indices = [tag_to_idx[tag] for tag in tags]

    # Use the BERT tokenizer (self.tokenizer) to encode the sentence.
Make sure to
    # truncate the sentence if it is longer than self.max_len, and pad
the sentence if it
    # is less than self.max_len.
    encoding = self.tokenizer(
            sentence,
            is_split_into_words=True,
            return_offsets_mapping=True,
            padding="max_length",
            truncation=True,
            max_length=self.max_len,
            return_tensors="pt"
        )

    input_ids = encoding["input_ids"].squeeze(0)
    attention_mask = encoding["attention_mask"].squeeze(0)
    offset_mapping = encoding["offset_mapping"].squeeze(0)

    # Create token labels, where the first token of each word is the
POS tag, and
    # all others are -100.
    labels = torch.full((self.max_len,), -100, dtype=torch.long)

    # Add the token labels back to the tokenized dict
    word_idx = -1
    for i, (start, end) in enumerate(offset_mapping):
        if start == 0 and end != 0:
            word_idx += 1
            if word_idx < len(tag_indices):
                labels[i] = tag_indices[word_idx]

    # Make sure both your encoded sentence, labels and attention mask
are PyTorch tensors
    encoding["labels"] = labels
    encoding["input_ids"] = input_ids
    encoding["attention_mask"] = attention_mask

    ### END YOUR CODE ###

    return encoding
```

```
# Use your POSDataset class to create a train and test set
MAX_LEN = 128

# Further split your train data into train/test. You now have
train/test/val.
train_test_data, split = training_data, int(0.7 * len(training_data))
random.shuffle(train_test_data)
split_training_data, split_test_data = train_test_data[:split],
train_test_data[split:]

training_set = POSDataset(split_training_data, tokenizer, MAX_LEN)
testing_set = POSDataset(split_test_data, tokenizer, MAX_LEN)
validation_set = POSDataset(val_data, tokenizer, MAX_LEN)

# Print a few values from your Dataloader!
print(training_set.__getitem__(0)['input_ids'])
print(training_set.__getitem__(0)['labels'])
```

```
tensor([  101,  2021,  1996,  2047,  4696,  2515,  2025,  5672,  2030,
5547,
         8256,  1005,  1055,  5041,  5762, 18394,  1012,   102,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,
0,
            0,     0,     0,     0,     0,     0,     0,     0])
tensor([-100,   21,   13,   19,    6,   14,    8,   25,   21,   25,
26,   15,
        -100,   19,   19,   40,   12, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
```

```
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -
100, -100,
        -100, -100, -100, -100, -100, -100, -100, -100])

# Create PyTorch dataloaders from the POSDataset
from torch.utils.data import DataLoader

training_loader = DataLoader(training_set, batch_size=64,
shuffle=True)
testing_loader = DataLoader(testing_set, batch_size=64, shuffle=True)
validating_loader = DataLoader(validation_set, batch_size=8,
shuffle=True)
```

4.3 Define your `BertForPOSTagging` Model

Now we will modify BERT by extending the `DistilBertModel` class for our task.

```
class BertForPOSTagging(DistilBertModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        ### BEGIN YOUR CODE ###
        ### END YOUR CODE ###

        self.post_init()

    def forward(self, input_ids, attention_mask, labels=None):
        """
        Forward pass through your model. Returns output logits for
each POS
        label and the loss (if labels is not None)

        Hint: You may use nn.CrossEntropyLoss() to calculate your
loss.
        """
        loss, logits = None, None

        ### BEGIN YOUR CODE ###
```

```
        ### END YOUR CODE ###

        if loss is not None:
          return loss, logits
        return logits

model = BertForPOSTagging.from_pretrained(
    'distilbert-base-uncased',
    num_labels=len(tag_to_idx)
).to(device)

MAX_GRAD_NORM = 10
EPOCHS = 5

optimizer = torch.optim.Adam(params=model.parameters(), lr=1e-04)
```

4.4 Training and Evaluation

Now we have instantiated our model, please create the train loop!

Hint: If your implementation is correct, you can expect a validation accuracy of `0.88`

```
# DistilBERT will take up a lot of memory (particularly during
development)
# use this to check the amount of memory you currently have. (Note:
you should
# be able to fine-tune with ~5 GB of GPU memory)
print(f"Currently allocated GPU memory:
{torch.cuda.memory_allocated(device) / 1024**3:.2f} GB /
{torch.cuda.get_device_properties(0).total_memory / 1024**3:.2f} GB")

# Hint: use `torch.cuda.empty_cache()` to clear the CUDA cache
```

```
---------------------------------------------------------------------
-----
AssertionError                            Traceback (most recent call
last)
Cell In[41], line 4
      1 # DistilBERT will take up a lot of memory (particularly during
development)
      2 # use this to check the amount of memory you currently have.
(Note: you should
      3 # be able to fine-tune with ~5 GB of GPU memory)
----> 4 print(f"Currently allocated GPU memory:
{torch.cuda.memory_allocated(device) / 1024**3:.2f} GB /
{torch.cuda.get_device_properties(0).total_memory / 1024**3:.2f} GB")
      6 # Hint: use `torch.cuda.empty_cache()` to clear the CUDA cache
```

```
File c:\Python312\Lib\site-packages\torch\cuda\__init__.py:523, in
get_device_properties(device)
    511 def get_device_properties(device: Optional[_device_t] = None)
-> _CudaDeviceProperties:
    512     r"""Get the properties of a device.
    513
    514     Args:
   (...)
    521         _CudaDeviceProperties: the properties of the device
    522     """
--> 523     _lazy_init()  # will define _get_device_properties
    524     device = _get_device_index(device, optional=True)
    525     if device < 0 or device >= device_count():

File c:\Python312\Lib\site-packages\torch\cuda\__init__.py:310, in
_lazy_init()
    305     raise RuntimeError(
    306         "Cannot re-initialize CUDA in forked subprocess. To
use CUDA with "
    307         "multiprocessing, you must use the 'spawn' start
method"
    308     )
    309 if not hasattr(torch._C, "_cuda_getDeviceCount"):
--> 310     raise AssertionError("Torch not compiled with CUDA
enabled")
    311 if _cudart is None:
    312     raise AssertionError(
    313         "libcudart functions unavailable. It looks like you
have a broken build?"
    314     )

AssertionError: Torch not compiled with CUDA enabled

def train(epoch):
    train_loss = 0
    train_examples, train_steps = 0, 0

    model.train()
    model.zero_grad()

    for idx, batch in enumerate(training_loader):
        ids = batch['input_ids'].to(device, dtype=torch.long)
        mask = batch['attention_mask'].to(device, dtype=torch.long)
        labels = batch['labels'].to(device, dtype=torch.long)

        ### BEGIN YOUR CODE ###
```

```
        ### END YOUR CODE ###

        train_steps += 1
        train_examples += labels.size(0)

    avg_train_loss = train_loss / train_steps
    avg_val_loss, val_accuracy = evaluate_bert(model)

    print(f"Epoch: {epoch}/{EPOCHS}\tAvg Train Loss:
{avg_train_loss:.4f}\tAvg Val Loss: {avg_val_loss:.4f}\t Val Accuracy:
{val_accuracy:.0f}")

def evaluate_bert(model):
    correct, val_loss, val_examples = 0, 0, 0

    model.eval()
    with torch.no_grad():
        for idx, batch in enumerate(validating_loader):
            """
            Implement the evaluate method. Find the average validation
loss
            along with the validation accuracy.

            Remember! You have labeled only the first token of each
word. Make
            sure you only calculate accuracy on values which are not -
100.
            """
            ids = batch['input_ids'].to(device, dtype=torch.long)
            mask = batch['attention_mask'].to(device,
dtype=torch.long)
            labels = batch['labels'].to(device, dtype=torch.long)

            ### BEGIN YOUR CODE ###


            # Compute training accuracy

            # Only compute accuracy at active labels

            # Get the predicted labels

            # Get number of correct predictions

            # Increase running total loss and the number of past valid
samples


            ### END YOUR CODE ###

    val_accuracy = 100 * correct / val_examples
```

```
    avg_val_loss = val_loss / val_examples
    return avg_val_loss, val_accuracy

for epoch in range(EPOCHS):
    train(epoch)
```

4.5 Inference

Good job! Now we can use our fine-tuned BERT model for POS tagging.

In fact, if you have a fine-tuned transformer model (such as in a final project), you could directly upload the model to HuggingFace for others to use (see this group, which fine-tuned on a much larger corpus of POS tags).

```
def generate_prediction(model, sentence):
    """
    Given a sentence, generate a full prediction of POS tags.

    In this case, you are given a full sentence (not array of tokens),
so you
    will need to use your tokenizer differently.

    Return your prediction in the format:
      [(token 1, POS prediction 1), (token 2, POS prediction 2), ...]

    E.g., "The imperatives that" => [('the', 'DT'), ('imperative',
'NNS'), ('that', 'WDT')]
    """
    prediction = []

    ### BEGIN YOUR CODE ###



    ### END YOUR CODE ###

    return prediction

sentence = "The imperatives that can be obeyed by a machine that has
no limbs are bound to be of a rather intellectual character."
print(generate_prediction(model, sentence))
```

5. Submit Your Homework

This is the end of Project 2. Congratulations!

Now, follow the steps below to submit your homework in Gradescope:

1. Rename this ipynb file to 'CS4650_p2_GTusername.ipynb'. We recommend ensuring you have removed any extraneous cells & print statements, clearing all outputs, and using

the Runtime --> Run all tool to make sure all output is update to date. Additionally, leaving comments in your code to help us understand your operations will assist the teaching staff in grading. It is not a requirement, but is recommended.

2. Click on the menu 'File' --> 'Download' --> 'Download .py'.
3. Click on the menu 'File' --> 'Download' --> 'Download .ipynb'.
4. Download the notebook as a .pdf document. Make sure the output from Parts 1.6 & 2 & 3 are captured so we can see how the loss and accuracy changes while training.
5. Upload all 3 files to Gradescope. Double check the files start with `CS4650_p2_*`, capitalization matters.