3725520208_CSCI544_HW4

March 19, 2024

Name : Anne Sai Venkata Naga Saketh USC Email : annes@usc.edu USC ID : 3725520208 CSCI 544 - Applied Natural Language Processing Homework 4

0.0.1 Import Statements

[1]:
```
# Importing sys module for system-specific parameters and functions
import sys
# Importing json module for JSON (JavaScript Object Notation) encoding and␣
↪decoding
import json
# Importing PyTorch, a popular deep learning framework
import torch
# Importing neural network module from PyTorch
import torch.nn as nn
# Importing optimization module from PyTorch
import torch.optim as optim
# Importing performance metrics from scikit-learn
from sklearn.metrics import precision_score, recall_score, f1_score
# Importing numpy, a fundamental package for scientific computing in Python
import numpy as np
```

[2]:
```
# Path to the input data file
input_path = "data/train"

# List to store lines of data
l_list = []

# Dictionary to store vocabulary word frequency
v = {}

# Dictionary to store tag frequency
t_f = {}

# Counter for sentence endings
s_c = 0

# Read all lines from the file and Iterate over each line
```

```
with open(input_path) as file:
    input_lines = file.readlines()
    for line in input_lines:
        words = line.split(" ")

        # List to store current line data
        current_line = []

        if len(words) == 1:
            # Append an empty string to current line
            current_line.append(" ")

            # Append current line to the list
            l_list.append(current_line)
            continue
        # Get the index, current word, tag from the line and append the current␣
↪word to the line and the tag to the line
        else:
            index = words[0].strip()
            current_w = words[1].strip()
            tag = words[2].strip()
            current_line.append(index)
            current_line.append(current_w)
            current_line.append(tag)

            if current_w == '.':
                # Increment the sentence ending counter
                s_c += 1

            # If current word is already in vocabulary
            if current_w in v:
                # Increment its frequency
                v[current_w] += 1
            # If current word is not in vocabulary
            else:
                # Add it to vocabulary with frequency 1
                v.update({current_w: 1})

            # If tag is already in tag frequency dictionary
            if tag in t_f:
                # Increment its frequency
                t_f[tag] += 1
            # If tag is not in tag frequency dictionary
            else:
                # Add it to tag frequency dictionary with frequency 1
                t_f.update({tag: 1})
```

```
        # Append current line to the list
        l_list.append(current_line)

# Print sizes of vocabulary, tag frequency, and lines
print("The size of the Vocabulary is :", len(v))
print("The size of the Tags Frequency is :", len(t_f))
print("The size of the lines is :", len(l_list))
```

```
The size of the Vocabulary is : 23624
The size of the Tags Frequency is : 9
The size of the lines is : 219553
```

0.0.2 Task:1 – Simple Bi-Directional LSTM Model

[3]:
```
# List to store sentences
sents = []
# List to store tags
t = []
# List to store current sentence words
present_s = []
# List to store current sentence tags
present_t = []

# Iterate over each line in l_list
for l in l_list:
    # If the line contains only one element (empty line indicating end of␣
 ↪sentence)
    if len(l) == 1:
        t.append(present_t)
        # Reset present_t to empty list for the next sentence
        present_t = []
        sents.append(present_s)
        # Reset present_s to empty list for the next sentence
        present_s = []
    # If the line contains more than one element
    else:
        present_t.append(l[2])
        present_s.append(l[1])
```

[4]:
```
# Dictionary to map words to indices
w2i = {}
# Dictionary to map tags to indices
t2i = {}
# Initial index value
index_value = 0
```

```python
# Iterate over each unique tag in tag frequency dictionary
for tag in t_f:
    # If the tag already exists in the tag-to-index dictionary
    if tag in t2i:
        continue
    # If the tag is new
    else:
        # Add tag to tag-to-index dictionary with current index value
        t2i.update({tag: index_value})
        # Increment index value for the next tag
        index_value += 1

# Iterate over each sentence in the list of sentences
for s in sents:
    # Iterate over each word in the sentence
    for word in s:
        # If the word already exists in the word-to-index dictionary
        if word in w2i:
            continue
        # If the word is new
        else:
            # Add word to word-to-index dictionary with current index value
            w2i.update({word: len(w2i)})
```

[5]:

```python
# List to store tensor representations of sentences
s_tensors = []

# Iterate over each sentence in the list of sentences
for s in sents:
    # Convert words to indices using word-to-index dictionary
    sent_ind = [w2i[word] for word in s]

    # Convert list of indices to tensor and append to s_tensors
    s_tensors.append(torch.tensor(sent_ind, dtype=torch.long))

# If '<UNK>' token is not already present in word-to-index dictionary
if '<UNK>' not in w2i:
    # Add '<UNK>' token to word-to-index dictionary with current index value
    w2i.update({'<UNK>':len(w2i)})

# List to store tensor representations of tags
t_tensors = []

# Iterate over each list of tags in the list of tags
for tag_one in t:
    # Convert tags to indices using tag-to-index dictionary
    tag_ind = [t2i[tag] for tag in tag_one]
```

```
    t_tensors.append(torch.tensor(tag_ind, dtype=torch.long))
```

[6]:
```
class BLSTM(nn.Module):  # Definition of the BLSTM class as a subclass of the␣
↪PyTorch nn.Module class.
    def __init__(self, vocab_size, tagToIndexD, embedding_dim, hidden_dim,␣
↪dropout):
        super(BLSTM, self).__init__()
        self.hidden_dim = hidden_dim  # Set the hidden dimension size
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)  # Define␣
↪word embedding layer
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2, num_layers=1,␣
↪bidirectional=True)  # Define bidirectional LSTM layer
        self.hidden1tag = nn.Linear(hidden_dim, hidden_dim // 2)  # Define␣
↪linear layer for tag prediction
        self.hidden2tag = nn.Linear(hidden_dim // 2, len(tagToIndexD))  # Define␣
↪linear layer for tag prediction
        self.dropout = nn.Dropout(p=0.33)  # Define dropout layer with dropout␣
↪rate of 0.33
        self.activation = nn.ELU()  # Define activation function as Exponential␣
↪Linear Unit (ELU)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)  # Perform word embedding
        lstm_out, _ = self.lstm(embeds.view(len(sentence), 1, -1))  # Forward␣
↪pass through LSTM layer
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)  # Reshape LSTM␣
↪output
        lstm_out = self.dropout(lstm_out)  # Apply dropout
        tag_space = self.hidden1tag(lstm_out)  # Compute tag scores
        tag_scores = self.activation(tag_space)  # Apply activation function
        tag_space2 = self.hidden2tag(tag_scores)  # Compute tag scores for the␣
↪final layer
        return tag_space2  # Return the final tag scores
```

[7]:
```
# Define hyperparameters
e_dim = 100  # Dimensionality of word embeddings
lstm_dropout = 0.33  # Dropout rate for LSTM layer
lstm_hidden = 256  # Number of hidden units in LSTM layer
lstm_output = 128  # Dimensionality of LSTM output
rate_of_l = 0.1  # Learning rate for optimizer
epochs = 20  # Number of training epochs

# Initialize BLSTM model with defined hyperparameters
model = BLSTM(vocab_size=len(w2i),  # Vocabulary size (number of unique words)
              embedding_dim=e_dim,  # Dimensionality of word embeddings
              dropout=lstm_dropout,  # Dropout rate for LSTM layer
```

```
              tagToIndexD=t2i,  # Dictionary mapping tags to indices
              hidden_dim=lstm_hidden)  # Number of hidden units in LSTM layer

# Define loss function
loss_function = nn.CrossEntropyLoss()

# Define optimizer
optimizer = optim.SGD(model.parameters(), lr=rate_of_l)  # Stochastic Gradient␣
↪Descent (SGD) optimizer
```

[8]:
```
# Train the model
for epoch in range(epochs):
    for index in range(len(s_tensors)):
        sentence = s_tensors[index]
        tags = t_tensors[index]

        # Clear accumulated gradients
        model.zero_grad()

        # Forward pass
        op_tags = model(sentence)

        # Calculate loss and perform backpropagation
        loss = loss_function(op_tags, tags)
        loss.backward()
        optimizer.step()
        # for each epoch, iterating over sentences_tensors, and extracting a␣
↪single sentence tensors and
        # single sentence tag tensors, then clears the gradients of all model␣
↪parameters before the forward pas
        # performs a forward pass through the model to generate predicted tags␣
↪for the current input sentence.
        # calculates the loss between the predicted tags and the true tags for␣
↪the current input sentence.
        # performs backpropagation to compute the gradients of the loss with␣
↪respect to all model parameters.

        # updates the model parameters based on the gradients computed during␣
↪backpropagation.
    print('Running Epoch [{}/{}], Loss: {:.5f}'.format(epoch+1, epochs, loss.
↪item()))
```

```
Running Epoch [1/20], Loss: 0.23459
Running Epoch [2/20], Loss: 0.28995
Running Epoch [3/20], Loss: 0.02823
Running Epoch [4/20], Loss: 0.02040
Running Epoch [5/20], Loss: 0.00493
```

```
Running Epoch [6/20], Loss: 0.00825
Running Epoch [7/20], Loss: 0.00937
Running Epoch [8/20], Loss: 0.00280
Running Epoch [9/20], Loss: 0.00127
Running Epoch [10/20], Loss: 0.05738
Running Epoch [11/20], Loss: 0.00041
Running Epoch [12/20], Loss: 0.03127
Running Epoch [13/20], Loss: 0.00353
Running Epoch [14/20], Loss: 0.26481
Running Epoch [15/20], Loss: 0.00133
Running Epoch [16/20], Loss: 0.00032
Running Epoch [17/20], Loss: 0.00004
Running Epoch [18/20], Loss: 0.00005
Running Epoch [19/20], Loss: 0.00278
Running Epoch [20/20], Loss: 0.00024
```

[9]:
```
torch.save(model, 'blstm1.pt')
```

0.0.3 Task: 1 – Testing on Dev Data

[10]:
```
dev_path = "data/dev"  # Path to the dev data file
dev_lines = []  # List to store lines of dev data
s_count = 0  # Counter for sentences

# Open the dev data file and read all lines
with open(dev_path) as file:
    i_lines = file.readlines()

    # Iterate over each line in the file
    for line in i_lines:

        # Split the line into words
        words = line.split(" ")

        # List to store current line data
        present_l = []

        if len(words) == 1:  # If the line has only one word (indicating end of␣
 ↪sentence)
            present_l.append(" ")  # Append an empty string to curline
            dev_lines.append(present_l)  # Append curline to dev_lines
            continue  # Move to the next iteration

        else:  # If the line has more than one word
            index = words[0].strip()  # Get the index from the line
            present_w = words[1].strip()  # Get the current word from the line
```

```
            tag = words[2].strip()  # Get the tag from the line

            present_l.append(index)  # Append index to curline
            present_l.append(present_w)  # Append current word to curline
            present_l.append(tag)  # Append tag to curline

        dev_lines.append(present_l)  # Append curline to dev_lines
```

[11]:
```
d_sentences = []  # List to store sentences in dev data
present_s = []  # List to store words of current sentence

# Iterate over each line in the dev data
for line in dev_lines:
    if len(line) == 1:  # If the line contains only one element (empty line␣
 ↪indicating end of sentence)
        d_sentences.append(present_s)  # Append words of current sentence to␣
 ↪dev_sentences
        present_s = []  # Reset currS to empty list for the next sentence
    else:  # If the line contains more than one element
        present_s.append(line[1])  # Append word to currS

# Append the last sentence to dev_sentences
d_sentences.append(present_s)
```

[12]:
```
d_tags = []  # List to store tags for each sentence in dev data
present_s = []  # List to store tags of current sentence

# Iterate over each line in the dev data
for line in dev_lines:
    if len(line) == 1:  # If the line contains only one element (empty line␣
 ↪indicating end of sentence)
        d_tags.append(present_s)  # Append tags of current sentence to dev_tags
        present_s = []  # Reset current_sentence to empty list for the next␣
 ↪sentence
    else:  # If the line contains more than one element
        present_s.append(line[2])  # Append tag to current_sentence

# Append the last sentence tags to dev_tags
d_tags.append(present_s)
```

[13]:
```
d_sentences_tensors = []  # List to store tensors representing word indices for␣
 ↪each sentence in dev data

# Iterate over each sentence in the list of dev sentences
for s in d_sentences:
```

```
    sent_index = [w2i.get(word, w2i['<UNK>']) for word in s]  # Convert words to␣
↪indices using word-to-index dictionary
    d_sentences_tensors.append(torch.tensor(sent_index, dtype=torch.long))  #␣
↪Convert list of indices to PyTorch tensor and append to dev_sentences_tensors
```

[14]:
```
model.eval()  # Set the model to evaluation mode
with torch.no_grad():  # Disable gradient calculation to speed up inference and␣
↪reduce memory consumption
    dev_tag_ops = []  # List to store model outputs for dev data
    for s in d_sentences_tensors:  # Iterate over each sentence tensor in dev␣
↪data
        t_op = model(s)  # Pass the sentence tensor through the model to get tag␣
↪predictions
        dev_tag_ops.append(t_op)  # Append model output (tag predictions) to␣
↪dev_tag_ops
```

[15]:
```
dev_predicted_tags_ner = []  # List to store predicted tags for each sentence in␣
↪dev data

# Iterate over each set of tag scores (predictions) for dev data
for tag_scores in dev_tag_ops:
    _, predicted_tags = torch.max(tag_scores, dim=1)  # Get the index of the tag␣
↪with the highest score for each word
    # Convert predicted tag indices to tag labels using the tag-to-index␣
↪dictionary (t2i)
    dev_predicted_tags_ner.append([list(t2i.keys())[i] for i in predicted_tags.
↪tolist()])
```

[16]:
```
# Flatten the list of predicted tags for dev data
dev_predicted_tags_write = [tag for sentence_tags in dev_predicted_tags_ner for␣
↪tag in sentence_tags]

# Flatten the list of true tags for dev data
dev_tags_flat = [tag for sentence_tags in d_tags for tag in sentence_tags]
```

[17]:
```
out_file = open("dev1.out", "w")  # Open a file named "task1_dev.out" for writing
write = []  # List to store lines of output for writing to the file

# Iterate over each line index in the range of the number of lines in dev data
index = 0  # Initialize index for iterating over predicted tags
for line in range(len(dev_lines)):
    if len(dev_lines[line]) == 1:  # If the line contains only one element␣
↪(indicating end of sentence)
        write.append("\n")  # Append a blank line to indicate end of sentence in␣
↪the output file
    else:  # If the line contains more than one element
```

```
        # Construct a line of output containing index, word, actual NER tag, and␣
↪predicted NER tag
        curLine = dev_lines[line][0] + " " + dev_lines[line][1] + " " +␣
↪dev_lines[line][2] + " " + dev_predicted_tags_write[index] + "\n"
        index += 1  # Increment index for the next predicted tag
        write.append(curLine)  # Append the constructed line to the list

# Write the lines of output to the file
out_file.writelines(write)

# Close the file
out_file.close()
```

[18]:
```
!python eval.py -p dev1.out -g data/dev
```

```
processed 51578 tokens with 5942 phrases; found: 5184 phrases; correct: 4138.
accuracy:  94.88%; precision:  79.82%; recall:  69.64%; FB1:  74.38
              LOC: precision:  93.82%; recall:  75.23%; FB1:  83.50  1473
             MISC: precision:  88.13%; recall:  71.69%; FB1:  79.07  750
              ORG: precision:  65.20%; recall:  66.37%; FB1:  65.78  1365
              PER: precision:  75.50%; recall:  65.42%; FB1:  70.10  1596
```

0.0.4 Task: 1 – Testing on Test Data

[19]:
```
test_path = "data/test"  # Path to the test data file
test_lines = []  # List to store lines of test data
s_count = 0  # Counter for sentences

# Open the test data file and read all lines
with open(test_path) as file:
    t_lines = file.readlines()

    # Iterate over each line in the file
    for line in t_lines:
        words = line.split(" ")  # Split the line into words
        present_l = []  # List to store current line data

        if len(words) == 1:  # If the line has only one word (indicating end of␣
↪sentence)
            present_l.append(" ")  # Append an empty string to present_l
            test_lines.append(present_l)  # Append present_l to test_lines
            continue  # Move to the next iteration

        else:  # If the line has more than one word
            index = words[0].strip()  # Get the index from the line
            present_w = words[1].strip()  # Get the current word from the line
```

```
            present_l.append(index)  # Append index to present_l
            present_l.append(present_w)  # Append current word to present_l

        test_lines.append(present_l)  # Append present_l to test_lines
```

[20]:
```
t_sentences = []  # List to store sentences in test data
present_s = []  # List to store words of current sentence

# Iterate over each line in the test data
for line in test_lines:
    if len(line) == 1:  # If the line contains only one element (empty line␣
 ↪indicating end of sentence)
        t_sentences.append(present_s)  # Append words of current sentence to␣
 ↪t_sentences
        present_s = []  # Reset present_s to empty list for the next sentence
    else:  # If the line contains more than one element
        present_s.append(line[1])  # Append word to present_s

# Append the last sentence to t_sentences
t_sentences.append(present_s)

t_sentences_tensors = []  # List to store tensors representing word indices for␣
 ↪each sentence in test data

# Iterate over each sentence in the list of test sentences
for s in t_sentences:
    s_index = [w2i.get(word, w2i['<UNK>']) for word in s]  # Convert words to␣
 ↪indices using word-to-index dictionary
    t_sentences_tensors.append(torch.tensor(s_index, dtype=torch.long))  #␣
 ↪Convert list of indices to PyTorch tensor and append to t_sentences_tensors

model.eval()  # Set the model to evaluation mode
with torch.no_grad():  # Disable gradient calculation to speed up inference and␣
 ↪reduce memory consumption
    test_tag_ops = []  # List to store model outputs for test data
    # Iterate over each sentence tensor in test data
    for s in t_sentences_tensors:
        t_op = model(s)  # Pass the sentence tensor through the model to get tag␣
 ↪predictions
        test_tag_ops.append(t_op)  # Append model output (tag predictions) to␣
 ↪test_tag_ops
```

[21]:
```
test_predicted_tags_ner = []  # List to store predicted tags for each sentence␣
 ↪in test data

# Iterate over each set of tag scores (predictions) for test data
```

```
for t_op in test_tag_ops:
    _, predicted_tags = torch.max(t_op, dim=1)  # Get the index of the tag with␣
↪the highest score for each word
    # Convert predicted tag indices to tag labels using the tag-to-index␣
↪dictionary (t2i)
    test_predicted_tags_ner.append([list(t2i.keys())[i] for i in predicted_tags.
↪tolist()])
```

[22]:
```
# Now the test_predicted_tags_ner is flatted such that it can be iterated and␣
↪written back to the test.out
test_predicted_tags_flat = [tag for test_sentence_tags in␣
↪test_predicted_tags_ner for tag in test_sentence_tags]
```

[23]:
```
test_out_file = open("test1.out", "w")  # Open a file named "task1_test.out" for␣
↪writing
write = []  # List to store lines of output for writing to the file
index = 0  # Initialize index for iterating over predicted tags

# Iterate over each line index in the range of the number of lines in test data
for line in range(len(test_lines)):
    if len(test_lines[line]) == 1:  # If the line contains only one element␣
↪(indicating end of sentence)
        write.append("\n")  # Append a blank line to indicate end of sentence in␣
↪the output file
    else:  # If the line contains more than one element
        # Construct a line of output containing index, word, and predicted NER␣
↪tag
        present_l = test_lines[line][0] + " " + test_lines[line][1] + " " +␣
↪test_predicted_tags_flat[index] + "\n"
        index += 1  # Increment index for the next predicted tag
        write.append(present_l)  # Append the constructed line to the list

# Write the lines of output to the file
test_out_file.writelines(write)

# Close the file
test_out_file.close()
```

0.0.5 Task:2 - Using Glove Word Embeddings

[24]:
```
g_E = {}  # Dictionary to store GloVe word embeddings

# Open the GloVe file and read each line
with open('glove.6B.100d', 'r') as g_file:
    # Iterate over each line in the file
```

```
        for line in g_file:
            index = line.split()  # Split the line into elements
            present_w = index[0]  # Extract the word from the line
            word_v = np.array(index[1:], dtype=np.float32)  # Extract the word␣
↪vector and convert it to a numpy array
            g_E[present_w] = word_v  # Add the word and its corresponding vector to␣
↪the GloVe embeddings dictionary

# Check if '<PAD>' is not in the word-to-index dictionary (w2i)
if '<PAD>' not in w2i:
    w2i.update({'<PAD>': len(w2i)})
```

[25]:

```
class GloveBLSTM(nn.Module): # This defines a new PyTorch module called␣
↪GloveBLSTM.

    def __init__(self, vocab_size, tag_size, embedding_dim, hidden_dim, dropout,␣
↪word_to_embedding):
        super(GloveBLSTM, self).__init__()
        self.embedding_dim = embedding_dim

        # This is an embedding layer that maps input words to their GloVe␣
↪embeddings.
        self.hidden_dim = hidden_dim
        self.word_to_embedding = word_to_embedding
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim//2, num_layers=1,␣
↪bidirectional=True, dropout=dropout)

        # This is a bidirectional LSTM layer that processes the input embeddings
        self.hidden2tag = nn.Linear(hidden_dim, hidden_dim//2)

        #This is a linear layer that maps the LSTM output to a smaller␣
↪dimensionality.
        self.dropout = nn.Dropout(p=0.33)

        #This is a dropout layer that randomly sets a fraction of the inputs to␣
↪zero during training, to pr
        self.activation = nn.ELU()

        #This applies an activation function to the output of the self.
↪hidden2tag layer.
        self.hidden3tag = nn.Linear(hidden_dim//2, tag_size)

    def forward(self, sentence):
        embeddings = []
```

```
        # If a word is not in the pre-trained embeddings,the code generates a␣
↪random vector using NumPy's np.random.normal() function with a scale of 0.6
        # and the same dimensionality as the pre-trained embeddings.This is done␣
↪in the if clause of the forward method where it checks if the current word is␣
↪in the
        for word in sentence:
            if word.lower() in self.word_to_embedding:
                embeddings.append(self.word_to_embedding[word.lower()])
            else:
                embeddings.append(np.random.normal(scale=0.6, size=self.
↪embedding_dim))
        embeddings = torch.tensor(embeddings)
        embeddings = embeddings.type(torch.float32)
        embeddings = embeddings.unsqueeze(0)
        lstm_out, _= self.lstm(embeddings)
        lstm_out = lstm_out.squeeze(0)
        lstm_out = self.dropout(lstm_out)
        tag_space = self.hidden2tag(lstm_out)
        tag_scores = self.activation(tag_space)
        tag_scores_final = self.hidden3tag(tag_scores)
        return tag_scores_final
```

[28]:

```
# Define hyperparameters for the GloveBLSTM model
v_c = len(w2i)  # Vocabulary size (number of unique words)
t_c = len(t2i)  # Number of unique NER tags
h_d = 256  # Dimensionality of LSTM hidden states
output_dimensions = 128  # Dimensionality of linear layer output
e_dim = 100  # Dimensionality of word embeddings
d_o = 0.33  # Dropout rate

# Create an instance of GloveBLSTM model with the specified hyperparameters
g_model = GloveBLSTM(v_c, t_c, e_dim, h_d, d_o, g_E)

# Define the loss function as CrossEntropyLoss
loss_function = nn.CrossEntropyLoss()

# Define the optimizer as SGD (Stochastic Gradient Descent) with the specified␣
↪learning rate
optimizer = optim.SGD(g_model.parameters(), lr=0.1)
```

[29]:

```
# Iterate over each epoch
for epoch in range(epochs):
    t_l = 0  # Initialize total loss for the epoch
    # Iterate over each sentence and its corresponding tags
    for i in range(len(sents)):
        s = sents[i]  # Get the i-th sentence
        tags = t_tensors[i]  # Get the tags for the i-th sentence
```

```
        g_model.zero_grad()  # Clear gradients from previous iteration
        results = g_model(s)  # Get the model predictions for the sentence
        loss = loss_function(results, tags)  # Compute the loss
        loss.backward()  # Backpropagate the loss
        optimizer.step()  # Update model parameters using optimizer
        t_l += loss.item()  # Accumulate the loss for the epoch
    # Print the average loss for the epoch
    print(f"Running Epoch [{epoch+1}/{epochs}] and the loss is: {t_l/len(sents):.
↪7f}")
```

```
Running Epoch [1/20] and the loss is: 0.3669600
Running Epoch [2/20] and the loss is: 0.2911047
Running Epoch [3/20] and the loss is: 0.2693533
Running Epoch [4/20] and the loss is: 0.2553874
Running Epoch [5/20] and the loss is: 0.2436230
Running Epoch [6/20] and the loss is: 0.2357930
Running Epoch [7/20] and the loss is: 0.2274897
Running Epoch [8/20] and the loss is: 0.2237183
Running Epoch [9/20] and the loss is: 0.2201400
Running Epoch [10/20] and the loss is: 0.2163373
Running Epoch [11/20] and the loss is: 0.2102483
Running Epoch [12/20] and the loss is: 0.2087261
Running Epoch [13/20] and the loss is: 0.2056789
Running Epoch [14/20] and the loss is: 0.2009405
Running Epoch [15/20] and the loss is: 0.2015911
Running Epoch [16/20] and the loss is: 0.1988005
Running Epoch [17/20] and the loss is: 0.1965594
Running Epoch [18/20] and the loss is: 0.1952963
Running Epoch [19/20] and the loss is: 0.1936225
Running Epoch [20/20] and the loss is: 0.1922456
```

[30]:
```
torch.save(g_model, 'blstm2.pt')
```

[31]:
```
g_model = torch.load('blstm2.pt')
```

0.0.6 Task:2 – Testing on Dev Data

[32]:
```
d_sentences = []  # List to store sentences in dev data
present_s = []  # List to store words of current sentence

# Iterate over each line in the dev data
for line in dev_lines:
    if len(line) == 1:  # If the line contains only one element (empty line␣
↪indicating end of sentence)
        d_sentences.append(present_s)  # Append words of current sentence to␣
↪dev_sentences
```

```
        present_s = []  # Reset currS to empty list for the next sentence
    else:  # If the line contains more than one element
        present_s.append(line[1])  # Append word to currS

# Append the last sentence to dev_sentences
d_sentences.append(present_s)
```

[33]:
```
d_tags = []  # List to store tags for each sentence in dev data
present_s = []  # List to store tags of current sentence

# Iterate over each line in the dev data
for line in dev_lines:
    if len(line) == 1:  # If the line contains only one element (empty line␣
↪indicating end of sentence)
        d_tags.append(present_s)  # Append tags of current sentence to dev_tags
        present_s = []  # Reset current_sentence to empty list for the next␣
↪sentence
    else:  # If the line contains more than one element
        present_s.append(line[2])  # Append tag to current_sentence

# Append the last sentence tags to dev_tags
d_tags.append(present_s)
```

[34]:
```
g_model.eval()  # Set the model to evaluation mode
with torch.no_grad():  # Disable gradient calculation to speed up inference and␣
↪reduce memory consumption
    dev_tag_scores = []  # List to store model outputs for dev data
    for s in d_sentences:  # Iterate over each sentence tensor in dev data
        t_op = g_model(s)  # Pass the sentence tensor through the model to get␣
↪tag predictions
        dev_tag_scores.append(t_op)  # Append model output (tag predictions) to␣
↪dev_tag_ops
```

[35]:
```
dev_predicted_tags_ner = []  # List to store predicted tags for each sentence in␣
↪dev data

# Iterate over each set of tag scores (predictions) for dev data
for tag_scores in dev_tag_scores:
    _, predicted_tags = torch.max(tag_scores, dim=1)  # Get the index of the tag␣
↪with the highest score for each word
    # Convert predicted tag indices to tag labels using the tag-to-index␣
↪dictionary (t2i)
    dev_predicted_tags_ner.append([list(t2i.keys())[i] for i in predicted_tags.
↪tolist()])
```

[36]:
```
# Flatten the list of predicted tags for dev data
dev_predicted_tags_write = [tag for sentence_tags in dev_predicted_tags_ner for␣
↪tag in sentence_tags]

# Flatten the list of true tags for dev data
dev_tags_flat = [tag for sentence_tags in d_tags for tag in sentence_tags]
```

[37]:
```
out_file = open("dev2.out", "w")  # Open a file named "task1_dev.out" for writing
write = []  # List to store lines of output for writing to the file

# Iterate over each line index in the range of the number of lines in dev data
index = 0  # Initialize index for iterating over predicted tags
for line in dev_lines:
    if len(line) == 1:  # If the line contains only one element (indicating end␣
↪of sentence)
        write.append("\n")  # Append a blank line to indicate end of sentence in␣
↪the output file
    else:  # If the line contains more than one element
        # Construct a line of output containing index, word, actual NER tag, and␣
↪predicted NER tag
        curLine = line[0] + " " + line[1] + " " + line[2] + " " +␣
↪dev_predicted_tags_write[index] + "\n"
        index += 1  # Increment index for the next predicted tag
        write.append(curLine)  # Append the constructed line to the list

# Write the lines of output to the file
out_file.writelines(write)

# Close the file
out_file.close()
```

[38]:
```
!python eval.py -p dev2.out -g data/dev
```

```
processed 51578 tokens with 5942 phrases; found: 5815 phrases; correct: 5134.
accuracy:  97.34%; precision:  88.29%; recall:  86.40%; FB1:  87.34
              LOC: precision:  91.46%; recall:  91.56%; FB1:  91.51  1839
             MISC: precision:  84.16%; recall:  75.49%; FB1:  79.59  827
              ORG: precision:  81.14%; recall:  76.66%; FB1:  78.83  1267
              PER: precision:  91.82%; recall:  93.81%; FB1:  92.80  1882
```

0.0.7 Task:2 – Testing on Test Data

[39]:
```
t_sentences = []  # List to store sentences in test data
present_s = []  # List to store words of current sentence

# Iterate over each line in the test data
```

```
for line in test_lines:
    if len(line) == 1:  # If the line contains only one element (empty line␣
↪indicating end of sentence)
        t_sentences.append(present_s)  # Append words of current sentence to␣
↪t_sentences
        present_s = []  # Reset present_s to empty list for the next sentence
    else:  # If the line contains more than one element
        present_s.append(line[1])  # Append word to present_s

# Append the last sentence to t_sentences
t_sentences.append(present_s)

g_model.eval()  # Set the model to evaluation mode
with torch.no_grad():  # Disable gradient calculation to speed up inference and␣
↪reduce memory consumption
    test_tag_ops = []  # List to store model outputs for test data
    # Iterate over each sentence tensor in test data
    for s in t_sentences:
        t_op = g_model(s)  # Pass the sentence tensor through the model to get␣
↪tag predictions
        test_tag_ops.append(t_op)  # Append model output (tag predictions) to␣
↪test_tag_ops
```

[40]:
```
test_predicted_tags_ner = []  # List to store predicted tags for each sentence␣
↪in test data

# Iterate over each set of tag scores (predictions) for test data
for t_op in test_tag_ops:
    _, predicted_tags = torch.max(t_op, dim=1)  # Get the index of the tag with␣
↪the highest score for each word
    # Convert predicted tag indices to tag labels using the tag-to-index␣
↪dictionary (t2i)
    test_predicted_tags_ner.append([list(t2i.keys())[i] for i in predicted_tags.
↪tolist()])
```

[41]:
```
# Now the test_predicted_tags_ner is flatted such that it can be iterated and␣
↪written back to the test.out
test_predicted_tags_flat = [tag for test_sentence_tags in␣
↪test_predicted_tags_ner for tag in test_sentence_tags]
```

[42]:
```
test_out_file = open("test2.out", "w")  # Open a file named "task1_test.out" for␣
↪writing
write = []  # List to store lines of output for writing to the file
index = 0  # Initialize index for iterating over predicted tags

# Iterate over each line index in the range of the number of lines in test data
```

```
for line in range(len(test_lines)):
    if len(test_lines[line]) == 1:  # If the line contains only one element␣
↪(indicating end of sentence)
        write.append("\n")  # Append a blank line to indicate end of sentence in␣
↪the output file
    else:  # If the line contains more than one element
        # Construct a line of output containing index, word, and predicted NER␣
↪tag
        present_l = test_lines[line][0] + " " + test_lines[line][1] + " " +␣
↪test_predicted_tags_flat[index] + "\n"
        index += 1  # Increment index for the next predicted tag
        write.append(present_l)  # Append the constructed line to the list

# Write the lines of output to the file
test_out_file.writelines(write)

# Close the file
test_out_file.close()
```

0.0.8 Task 2 Complete