TASK 1

In [1]:
```
!pip install --upgrade torchtext
```
```
Requirement already satisfied: torchtext in c:\users\adity\anaconda3\lib\site-packages (0.17.1)
Requirement already satisfied: tqdm in c:\users\adity\anaconda3\lib\site-packages (from torchtext) (4.65.0)
Requirement already satisfied: requests in c:\users\adity\anaconda3\lib\site-packages (from torchtext) (2.31.0)
Requirement already satisfied: torch==2.2.1 in c:\users\adity\anaconda3\lib\site-packages (from torchtext) (2.2.1)
Requirement already satisfied: numpy in c:\users\adity\anaconda3\lib\site-packages (from torchtext) (1.24.3)
Requirement already satisfied: torchdata==0.7.1 in c:\users\adity\anaconda3\lib\site-packages (from torchtext) (0.7.1)
Requirement already satisfied: filelock in c:\users\adity\anaconda3\lib\site-packages (from torch==2.2.1->torchtext) (3.9.0)
Requirement already satisfied: typing-extensions>=4.8.0 in c:\users\adity\anaconda3\lib\site-packages (from torch==2.2.1->torchtext) (4.9.0)
Requirement already satisfied: sympy in c:\users\adity\anaconda3\lib\site-packages (from torch==2.2.1->torchtext) (1.11.1)
Requirement already satisfied: networkx in c:\users\adity\anaconda3\lib\site-packages (from torch==2.2.1->torchtext) (3.1)
Requirement already satisfied: jinja2 in c:\users\adity\anaconda3\lib\site-packages (from torch==2.2.1->torchtext) (3.1.2)
Requirement already satisfied: fsspec in c:\users\adity\anaconda3\lib\site-packages (from torch==2.2.1->torchtext) (2023.4.0)
Requirement already satisfied: urllib3>=1.25 in c:\users\adity\anaconda3\lib\site-packages (from torchdata==0.7.1->torchtext) (1.26.16)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\adity\anaconda3\lib\site-packages (from requests->torchtext) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:\users\adity\anaconda3\lib\site-packages (from requests->torchtext) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\adity\anaconda3\lib\site-packages (from requests->torchtext) (2023.7.22)
Requirement already satisfied: colorama in c:\users\adity\anaconda3\lib\site-packages (from tqdm->torchtext) (0.4.6)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\adity\anaconda3\lib\site-packages (from jinja2->torch==2.2.1->torchtext) (2.1.1)
Requirement already satisfied: mpmath>=0.19 in c:\users\adity\anaconda3\lib\site-packages (from sympy->torch==2.2.1->torchtext) (1.3.0)
```

In [12]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from collections import Counter
import numpy as np
```

In [13]:
```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Using device:", device)

# Tokenizer
tokenizer = get_tokenizer('basic_english')
```
```
Using device: cpu
```

In [4]:
```python
def build_vocab_from_data(file_path, tokenizer):
    counter = Counter()
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            counter.update(tokenizer(line.strip()))
    token_lists = [[token] for token in counter.keys()]
    vocab = build_vocab_from_iterator(token_lists, specials=["<unk>", "<pad>", "<bos>", "<eos>"])
    vocab.set_default_index(vocab["<unk>"])
    return vocab
```

In [14]:
```python
def build_tags_vocab(file_path):
    tags_counter = Counter()
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            parts = line.strip().split()
            if len(parts) == 3:
                _, _, tag = parts
                tags_counter.update([tag])
    tag_lists = [[tag] for tag in tags_counter.keys()]
    tags_vocab = build_vocab_from_iterator(tag_lists, specials=["O", "<unk>", "<pad>"])
    tags_vocab.set_default_index(tags_vocab["<unk>"])
    return tags_vocab
```

In [47]:
```python
def tokenize_and_index_test_data(file_path, text_vocab):
    tokenized_texts = []
    with open(file_path, 'r', encoding='utf-8') as file:
        tokens = []
        for line in file:
            line = line.strip()
            if line:  # If line not empty
                token = line.split()[1]
                tokens.append(token)
            else:
                if tokens:  # If there are collected tokens for this sentence
                    tokenized_texts.append([text_vocab[token.lower()] for token in tokens])
                    tokens = []  # Reset
        if tokens:  # Add last sentence if file doesn't end with newline
            tokenized_texts.append([text_vocab[token.lower()] for token in tokens])
    return tokenized_texts

# Tokenizing and indexing test data
test_texts = tokenize_and_index_test_data(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\data\test", vocab)


print("Number of test sentences:", len(test_texts))
if test_texts:
    print("First test sentence length:", len(test_texts[0]))
    print("First test sentence tokens:", test_texts[0])
else:
    print("Test data is empty after tokenization and indexing.")


test_texts = tokenize_and_index_test_data(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\data\test", vocab)

# After tokenizing and indexing your test data
print("Number of tokenized test sentences:", len(test_texts))
if test_texts:
    print("First few tokens of the first test sentence:", test_texts[0][:10])
else:
    print("Test data is empty after tokenization and indexing.")
```
```
Number of test sentences: 3684
First test sentence length: 12
First test sentence tokens: [16344, 79, 9935, 8260, 11138, 18952, 78, 4832, 9434, 17087, 5975, 99]
Number of tokenized test sentences: 3684
First few tokens of the first test sentence: [16344, 79, 9935, 8260, 11138, 18952, 78, 4832, 9434, 17087]
```

In [40]:
```python
# Printing first 5 sentences of tokenized test data
print("Sample tokenized test texts (first 5 sentences):")
for i, test_text in enumerate(test_texts[:5]):
    decoded_sentence = [vocab.get_itos()[token] for token in test_text]  # Convert back to words for easy reading
    print(f"Sentence {i + 1}: {' '.join(decoded_sentence)}")

test_texts, _ = tokenize_and_index_data(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\data\test", vocab, tags_vocab)
```
```
Sample tokenized test texts (first 5 sentences):
Sentence 1: soccer - japan get lucky win , china in surprise defeat .
Sentence 2: <unk> <unk>
Sentence 3: <unk> , united arab emirates <unk>
Sentence 4: japan began the defence of their asian cup title with a lucky 2-1 win against syria in a group c championship match on friday .
Sentence 5: but china saw their luck desert them in the second match of the group , crashing to a surprise 2-0 defeat to newcomers <unk> .
```

In [41]:
```python
vocab = build_vocab_from_data(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\data\train", tokenizer)
tags_vocab = build_tags_vocab(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\data\train")

# Tokenizing and indexing dataset
train_texts, train_tags = tokenize_and_index_data(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\data\train", vocab, tags_vocab)
validation_texts, validation_tags = tokenize_and_index_data(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\data\dev", vocab, tags_vocab)
```

In [42]:
```python
class BLSTM(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim, dropout):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=1, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        embedded = self.dropout(self.embedding(text))
        outputs, (hidden, cell) = self.lstm(embedded)
        predictions = self.fc(self.dropout(outputs))
        return predictions

# Model initialization
INPUT_DIM = len(vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = len(tags_vocab)
DROPOUT = 0.5

model = BLSTM(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM, DROPOUT).to(device)
optimizer = optim.SGD(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss().to(device)
```

In [44]:
```python
def create_batches(texts, tags, batch_size, device, is_test=False):
    # Debugging- initial condition of texts
    print(f"Entering create_batches with {'test' if is_test else 'train/validation'} data:")
    print(f"Initial number of sentences: {len(texts)}")
    if is_test:
        # For test data (no tags), check no empty sentences
        filtered_texts = [text for text in texts if len(text) > 0]
        # Debugging- remains after filtering
        print(f"Number of non-empty test texts after filtering: {len(filtered_texts)}")
        if not filtered_texts:
            raise ValueError("All test sentences are empty after filtering.")
        # Preparing tensors
        text_tensors = [torch.tensor(text, dtype=torch.long) for text in filtered_texts]
        # Dummy tag tensors
        tag_tensors = [torch.zeros(len(text), dtype=torch.long) for text in filtered_texts]
    else:
        # For training/validation data,we filter out pairs where text or tag is empty
        filtered_pairs = [(text, tag) for text, tag in zip(texts, tags) if len(text) > 0 and len(tag) > 0]
        # Separating the filtered texts and tags back out
        filtered_texts = [pair[0] for pair in filtered_pairs]
        filtered_tags = [pair[1] for pair in filtered_pairs]
        # Debugging- what remains after filtering
        print(f"Number of non-empty train/validation texts after filtering: {len(filtered_texts)}")
        if not filtered_texts or not filtered_tags:
            raise ValueError("No valid sentences or tags found. Check your data preprocessing and file contents.")
        # Preparing tensors
        text_tensors = [torch.tensor(text, dtype=torch.long) for text in filtered_texts]
        tag_tensors = [torch.tensor(tag, dtype=torch.long) for text, tag in filtered_pairs]

    # Preparing the final dataset and dataloader
    dataset = TensorDataset(
        torch.nn.utils.rnn.pad_sequence(text_tensors, batch_first=True, padding_value=vocab['<pad>']),
        torch.nn.utils.rnn.pad_sequence(tag_tensors, batch_first=True, padding_value=tags_vocab['O'] if not is_test else 0)
    )
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=not is_test)

    return dataloader
```

In [48]:
```python
BATCH_SIZE = 32
# Creating the data loaders
train_dataloader = create_batches(train_texts, train_tags, BATCH_SIZE, device)
valid_dataloader = create_batches(validation_texts, validation_tags, BATCH_SIZE, device)
test_dataloader = create_batches(test_texts, [[]] * len(test_texts), BATCH_SIZE, device, is_test=True)
```
```
Entering create_batches with train/validation data:
Initial number of sentences: 14987
Number of non-empty train/validation texts after filtering: 14987
Entering create_batches with train/validation data:
Initial number of sentences: 3466
Number of non-empty train/validation texts after filtering: 3466
Entering create_batches with test data:
Initial number of sentences: 3684
Number of non-empty test texts after filtering: 3684
```

In [49]:
```python
import torch.nn as nn
import torch.optim as optim


model = BLSTM(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM, DROPOUT).to(device)

# Define loss function and optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
```

In [50]:
```python
N_EPOCHS = 3  # number of epochs

for epoch in range(N_EPOCHS):
    model.train()
    total_loss = 0
    for texts, tags in train_dataloader:
        texts, tags = texts.to(device), tags.to(device)

        optimizer.zero_grad()
        predictions = model(texts)
        loss = loss_function(predictions.view(-1, OUTPUT_DIM), tags.view(-1))
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
    print(f'Epoch {epoch+1}: Training Loss: {total_loss / len(train_dataloader)}')
```
```
Epoch 1: Training Loss: 0.10776411971526105
Epoch 2: Training Loss: 0.07620487758107404
Epoch 3: Training Loss: 0.06620039921134774
```

In [51]:
```python
model.eval()
total_loss = 0
with torch.no_grad():
    for texts, tags in valid_dataloader:
        texts, tags = texts.to(device), tags.to(device)
        predictions = model(texts)
        loss = loss_function(predictions.view(-1, OUTPUT_DIM), tags.view(-1))
        total_loss += loss.item()
    print(f'Validation Loss: {total_loss / len(valid_dataloader)}')
```
```
Validation Loss: 0.07036589564533409
```

In [52]:
```python
import torch

predictions_list = []
with torch.no_grad():
    for texts, _ in test_dataloader:  # Tags not needed for test data
        texts = texts.to(device)
        outputs = model(texts)
        probabilities = torch.softmax(outputs, dim=-1)
        predictions = torch.argmax(probabilities, dim=-1)
        predictions_list.extend(predictions.cpu().numpy())
```

In [56]:
```python
# Saving model's state_dict
torch.save(model.state_dict(), 'blstm1.pt')

model.eval()
dev_predictions = []
with torch.no_grad():
    for texts, _ in valid_dataloader:
        texts = texts.to(device)
        outputs = model(texts)
        predictions = torch.argmax(outputs, dim=-1)
        dev_predictions.extend(predictions.cpu().numpy())

with open('dev1.out', 'w') as f:
    for sentence_preds in dev_predictions:
        for idx, tag_idx in enumerate(sentence_preds):
            f.write(f"{idx + 1} {tags_vocab.get_itos()[tag_idx]}\n")
        f.write("\n")

model.eval()
test_predictions = []
with torch.no_grad():
    for texts, _ in test_dataloader:
        texts = texts.to(device)
        outputs = model(texts)
        predictions = torch.argmax(outputs, dim=-1)
        test_predictions.extend(predictions.cpu().numpy())

with open('test1.out', 'w') as f:
    for sentence_preds in test_predictions:
        for idx, tag_idx in enumerate(sentence_preds):
            f.write(f"{idx + 1} {tags_vocab.get_itos()[tag_idx]}\n")
        f.write("\n")
```

In [57]:
```python
from sklearn.metrics import precision_recall_fscore_support

# Accumulating all actual tags and predicted tags from dev set
all_dev_tags = []
all_dev_preds = []

model.eval()
with torch.no_grad():
    for texts, tags in valid_dataloader:
        texts, tags = texts.to(device), tags.to(device)
        outputs = model(texts)
        predictions = torch.argmax(outputs, dim=-1)
        all_dev_preds.extend(predictions.view(-1).cpu().numpy())
        all_dev_tags.extend(tags.view(-1).cpu().numpy())

# precision, recall, and F1-score
precision, recall, f1, _ = precision_recall_fscore_support(all_dev_tags, all_dev_preds, average='weighted', zero_division=0)

print(f'Precision: {precision:.3f}, Recall: {recall:.3f}, F1-score: {f1:.3f}')
```
```
Precision: 0.975, Recall: 0.981, F1-score: 0.975
```

TASK 2

In [66]:
```python
def augment_token(token):
    if token.islower():
        return f"LOW_{token}"
    elif token.isupper():
        return f"UPP_{token}"
    elif token.istitle():
        return f"TITLE_{token}"
    else:
        return f"MISC_{token}"  # For mixed or other cases
```

In [67]:
```python
def tokenize_and_index_data(file_path, text_vocab, tokenizer, is_test=False):
    tokenized_texts = []
    with open(file_path, 'r', encoding='utf-8') as file:
        tokens = []
        for line in file:
            parts = line.strip().split()
            if len(parts) == 3 or (is_test and len(parts) == 2):
                token = parts[1] if is_test else parts[2]
                if not is_test:  # Only augmenting non-test tokens
                    token = augment_token(token)
                tokens.append(token)
            else:
                if tokens:
                    tokenized_texts.append([text_vocab[token.lower()] for token in tokens])  # Using lower to align with GloVe's case insensitivity
                    tokens = []
        if tokens:  # the last sentence
            tokenized_texts.append([text_vocab[token.lower()] for token in tokens])
    return tokenized_texts
```

In [68]:
```python
import numpy as np

def load_glove_embeddings(path):
    """Load the GloVe embeddings from a file."""
    embeddings_dict = {}
    with open(path, 'r', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.asarray(values[1:], "float32")
            embeddings_dict[word] = vector
    return embeddings_dict


glove_embeddings = load_glove_embeddings(r"C:\Users\adity\Desktop\NLP\HW4 Submission files\hw4\glove.6B.100d.txt")
```

In [69]:
```python
EMBEDDING_DIM = 100

def create_embedding_matrix(word_index, embedding_dict, dimension):
    embedding_matrix = np.zeros((len(word_index), dimension))
    for word, i in word_index.items():
        if word in embedding_dict:
            embedding_matrix[i] = embedding_dict[word]
        else:
            # Words not found in the embedding index = all-zeros.
            embedding_matrix[i] = np.random.normal(scale=0.6, size=(dimension, ))
    return embedding_matrix

# vocab is vocabulary from training data
embedding_matrix = create_embedding_matrix(vocab.get_stoi(), glove_embeddings, EMBEDDING_DIM)
```

In [70]:
```python
class BLSTM(nn.Module):
    def __init__(self, embedding_matrix, hidden_dim, output_dim, dropout):
        super().__init__()
        num_embeddings, embedding_dim = embedding_matrix.shape
        self.embedding = nn.Embedding(num_embeddings, embedding_dim)
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float32))
        self.embedding.weight.requires_grad = False  # Freeze embeddings
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=1, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        embedded = self.dropout(self.embedding(text))
        outputs, (hidden, cell) = self.lstm(embedded)
        predictions = self.fc(self.dropout(outputs))
        return predictions

# Reinitializing model with new embedding layer
model = BLSTM(embedding_matrix, HIDDEN_DIM, OUTPUT_DIM, DROPOUT).to(device)
```

In [71]:
```python
# Re-define the optimizer
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=1e-3)  # Only parameters that require gradients (excluding frozen embeddings)

N_EPOCHS = 3

for epoch in range(N_EPOCHS):
    model.train()
    total_loss = 0
    for texts, tags in train_dataloader:
        texts, tags = texts.to(device), tags.to(device)

        optimizer.zero_grad()
        predictions = model(texts)
        loss = criterion(predictions.view(-1, OUTPUT_DIM), tags.view(-1))
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
    print(f'Epoch {epoch + 1}: Training Loss: {total_loss / len(train_dataloader)}')
```
```
Epoch 1: Training Loss: 0.10047805314855789
Epoch 2: Training Loss: 0.05664814013773317
Epoch 3: Training Loss: 0.05019544748100899
```

In [72]:
```python
model.eval()  # Switch to evaluation mode
total_loss = 0
all_dev_preds = []
all_dev_tags = []

with torch.no_grad():
    for texts, tags in valid_dataloader:
        texts, tags = texts.to(device), tags.to(device)
        outputs = model(texts)
        loss = criterion(outputs.view(-1, OUTPUT_DIM), tags.view(-1))
        total_loss += loss.item()

        predictions = torch.argmax(outputs, dim=-1)
        all_dev_preds.extend(predictions.view(-1).cpu().numpy())
        all_dev_tags.extend(tags.view(-1).cpu().numpy())

print(f'Validation Loss: {total_loss / len(valid_dataloader)}')

# Calculate precision, recall, and F1-score
from sklearn.metrics import precision_recall_fscore_support
precision, recall, f1, _ = precision_recall_fscore_support(all_dev_tags, all_dev_preds, average='weighted', zero_division=0)
print(f'Precision: {precision:.3f}, Recall: {recall:.3f}, F1-score: {f1:.3f}')
```
```
Validation Loss: 0.05487297131859381
Precision: 0.982, Recall: 0.986, F1-score: 0.983
```

In [73]:
```python
# Save model state
torch.save(model.state_dict(), 'blstm2.pt')
```

In [74]:
```python
model.eval()
dev_predictions = []
with torch.no_grad():
    for texts, _ in valid_dataloader:
        texts = texts.to(device)
        outputs = model(texts)
        predictions = torch.argmax(outputs, dim=-1)
        dev_predictions.extend(predictions.cpu().numpy())

# Save the development set predictions to dev2.out
with open('dev2.out', 'w') as f:
    for sentence_preds in dev_predictions:
        for idx, tag_idx in enumerate(sentence_preds):
            f.write(f"{idx + 1} {tags_vocab.get_itos()[tag_idx]}\n")
        f.write("\n")

model.eval()
test_predictions = []
with torch.no_grad():
    for texts, _ in test_dataloader:
        texts = texts.to(device)
        outputs = model(texts)
        predictions = torch.argmax(outputs, dim=-1)
        test_predictions.extend(predictions.cpu().numpy())

# Save the test set predictions to test2.out
with open('test2.out', 'w') as f:
    for sentence_preds in test_predictions:
        for idx, tag_idx in enumerate(sentence_preds):
            f.write(f"{idx + 1} {tags_vocab.get_itos()[tag_idx]}\n")
        f.write("\n")
```

In [ ]: