TASK 1

The CoNLL corpus is processed to extract sentences and their corresponding tags. Tokenization is applied, and words are converted into indices. Datasets and data loaders are prepared for training, development, and testing.

A BiLSTM model is constructed using PyTorch. It consists of an embedding layer, a bidirectional LSTM layer, a linear layer for output, and dropout layers for regularization.

The hyperparameters used are:

Data Preprocessing:
min_freq: Minimum frequency threshold for building the vocabulary (value: 2).
Model Architecture:

EMBEDDING_DIM: Dimensionality of the word embedding (value: 100).
HIDDEN_DIM: Dimensionality of the hidden states in the LSTM layers (value: 256).
OUTPUT_DIM: Dimensionality of the output layer (value: 128).
DROPOUT: Dropout rate applied to the input and output of the LSTM layers (value: 0.33).

Training:
N_EPOCHS: Number of training epochs (value: 30).
Learning rate and scheduler parameters:
lr: Initial learning rate for SGD optimizer (set to 1).
factor: Factor by which the learning rate is reduced (value: 0.75).
patience: Number of epochs with no improvement after which learning rate will be reduced (value: 6).

Results on dev data:

```
processed 51578 tokens with 5942 phrases; found: 5592 phrases; correct: 4613.
accuracy:  95.89%; precision:  82.49%; recall:  77.63%; FB1:  79.99
              LOC: precision:  86.97%; recall:  86.50%; FB1:  86.74  1827
             MISC: precision:  83.72%; recall:  74.73%; FB1:  78.97  823
              ORG: precision:  76.85%; recall:  66.59%; FB1:  71.35  1162
              PER: precision:  81.01%; recall:  78.28%; FB1:  79.62  1780
```

TASK1

1. Reading the CoNLL corpus

In [297…
```
def read_data(file_path, has_tags=True):
    sentences = []
    tags = []
    original_sentences = []

    with open(file_path, 'r') as file:
        sentence = []
        tag = []
        original_sentence = []
        for line in file:
            if line.strip() == "":
                if sentence:
                    sentences.append(sentence)
                    if has_tags:
                        tags.append(tag)
                    original_sentences.append(original_sentence)
                    sentence = []
                    tag = []
                    original_sentence = []
                continue

            components = line.strip().split()
            if has_tags:
                index, word, ner_tag = components
                tag.append(ner_tag)
            else:
                index, word = components

            sentence.append(word)
            original_sentence.append(word)

        if sentence:
            sentences.append(sentence)
            if has_tags:
                tags.append(tag)
            original_sentences.append(original_sentence)

    if has_tags:
        return sentences, tags, original_sentences
    else:
        return sentences, original_sentences
```

In [298…
```
train_file_path = 'data/train'
dev_file_path = 'data/dev'
test_file_path = 'data/test'

train_sentences, train_tags, train_original_sentences = read_data(train_file_path)
dev_sentences, dev_tags, dev_original_sentences = read_data(dev_file_path)
test_sentences, test_original_sentences = read_data(test_file_path, has_tags=False)
```

2. Datasets and Dataloaders

2.1 Create a Vocabulary and convert Text to Indices

In [299…
```
from collections import Counter

def build_vocab(sentences, min_freq=2):
    word_counts = Counter(word for sentence in sentences for word in sentence)

    vocab = [word for word, count in word_counts.items() if count >= min_freq]

    vocab.append('<UNK>')
```

```
    word_to_idx = {word: idx for idx, word in enumerate(vocab)}

    return vocab, word_to_idx

vocab, word_to_idx = build_vocab(train_sentences)
```

2.2 Encode Labels

In [300…
```
def encode_sentences(sentences, word_to_idx):
    encoded_sentences = []

    for sentence in sentences:
        encoded_sentence = [word_to_idx.get(word, word_to_idx['<UNK>']) for word in sentence]
        encoded_sentences.append(encoded_sentence)

    return encoded_sentences

train_encoded_sentences = encode_sentences(train_sentences, word_to_idx)
dev_encoded_sentences = encode_sentences(dev_sentences, word_to_idx)
test_encoded_sentences = encode_sentences(test_sentences, word_to_idx)
```

In [301…
```
def build_tag_vocab(tags):
    unique_tags = set(tag for tag_list in tags for tag in tag_list)
    tag_to_idx = {tag: idx for idx, tag in enumerate(unique_tags)}
    return unique_tags, tag_to_idx

unique_tags, tag_to_idx = build_tag_vocab(train_tags)

def encode_tags(tags, tag_to_idx):
    encoded_tags = [[tag_to_idx[tag] for tag in tag_list] for tag_list in tags]
    return encoded_tags

train_encoded_tags = encode_tags(train_tags, tag_to_idx)
dev_encoded_tags = encode_tags(dev_tags, tag_to_idx)
```

2.3 Create PyTorch Datasets

In [302…
```
import torch
from torch.utils.data import Dataset

class NERDataset(Dataset):
    def __init__(self, sentences, tags=None):
        self.sentences = sentences
        self.tags = tags
        self.indices = list(range(len(sentences)))

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

    def __getitem__(self, idx):
        sentence = torch.tensor(self.sentences[idx], dtype=torch.long)
        index = self.indices[idx]  # Get the original index

        if self.tags is not None:
            tag = torch.tensor(self.tags[idx], dtype=torch.long)
            return sentence, tag, len(sentence), index
        else:
            return sentence, len(sentence), index


train_dataset = NERDataset(train_encoded_sentences, train_encoded_tags)
dev_dataset = NERDataset(dev_encoded_sentences, dev_encoded_tags)
test_dataset = NERDataset(test_encoded_sentences)
```

In [303…
```
from torch.nn.utils.rnn import pad_sequence

def pad_collate(batch):
    sentences = [item[0] for item in batch]
    sentences_padded = pad_sequence(sentences, batch_first=True, padding_value=word_to_idx['<UNK>'])

    lengths = torch.tensor([item[2] for item in batch])
```

```python
    if len(batch[0]) == 4:
        indices = [item[3] for item in batch]
    else:
        indices = [item[2] for item in batch]


    if any(isinstance(item[1], torch.Tensor) for item in batch):
        tags = [item[1] for item in batch]
        tags_padded = pad_sequence(tags, batch_first=True, padding_value=tag_to_idx['O'])
    else:
        tags_padded = None

    return sentences_padded, tags_padded, lengths, torch.tensor(indices)
```

2.4 Create DataLoaders

In [304…

```python
from torch.utils.data import DataLoader

BATCH_SIZE= 8
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=pad_collate)
dev_loader = DataLoader(dev_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate)
```

3. Model

In [305…

```python
import torch

device = "mps"
```

In [306…

```python
import torch.nn as nn

class BiLSTM_NER(nn.Module):
    def __init__(self, vocab_size, embedding_dim, lstm_hidden_dim, output_dim, dropout_rate):
        super(BiLSTM_NER, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.dropout = nn.Dropout(dropout_rate)
        self.bilstm = nn.LSTM(embedding_dim, lstm_hidden_dim, batch_first=True,
                              bidirectional=True)
        self.linear = nn.Linear(lstm_hidden_dim*2, output_dim)
        self.elu = nn.ELU()
        self.classifier = nn.Linear(output_dim, len(tag_to_idx))

    def forward(self, sentence):
        embedded = self.embedding(sentence)
        embedded = self.dropout(embedded)
        lstm_out, _ = self.bilstm(embedded)
        lstm_out = self.dropout(lstm_out)
        linear_out = self.linear(lstm_out)
        elu_out = self.elu(linear_out)
        scores = self.classifier(elu_out)
        return scores
```

3.1 Initializing hyperparameters

In [307…

```python
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 128
DROPOUT = 0.33

model = BiLSTM_NER(len(vocab), EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM, DROPOUT).to(device)
```

In [308…

```python
import torch.optim as optim

loss_function = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=1)
```

In [309…

```python
from torch.optim.lr_scheduler import ReduceLROnPlateau
```

```
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.75, patience=6)
```

4. Training the model

In [310…

```
N_EPOCHS = 30

for epoch in range(N_EPOCHS):
    model.train()
    total_loss = 0

    for sentence, tags, lengths, _ in train_loader:
        sentence, tags = sentence.to(device), tags.to(device)
        model.zero_grad()

        tag_scores = model(sentence)

        tag_scores = tag_scores.view(-1, tag_scores.shape[-1])
        tags = tags.view(-1)

        loss = loss_function(tag_scores, tags)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch+1}/{N_EPOCHS}, Loss: {total_loss/len(train_loader)}")

    scheduler.step(total_loss/len(train_loader))
```

```
Epoch 1/30, Loss: 0.2591114677866501
Epoch 2/30, Loss: 0.18519444295116716
Epoch 3/30, Loss: 0.1515665519465727
Epoch 4/30, Loss: 0.12803727640445356
Epoch 5/30, Loss: 0.11448936686943521
Epoch 6/30, Loss: 0.10434963478256522
Epoch 7/30, Loss: 0.09812651221688118
Epoch 8/30, Loss: 0.09198295711335928
Epoch 9/30, Loss: 0.08548992523787347
Epoch 10/30, Loss: 0.08288807847801488
Epoch 11/30, Loss: 0.0771229753947619
Epoch 12/30, Loss: 0.07527470086197287
Epoch 13/30, Loss: 0.07226784489581733
Epoch 14/30, Loss: 0.06932442297283341
Epoch 15/30, Loss: 0.06702786403421905
Epoch 16/30, Loss: 0.06430369081410998
Epoch 17/30, Loss: 0.0634121158669167
Epoch 18/30, Loss: 0.05967053835600288
Epoch 19/30, Loss: 0.058559700190919585
Epoch 20/30, Loss: 0.05607296030791009
Epoch 21/30, Loss: 0.05589241783453694
Epoch 22/30, Loss: 0.05291236705582076
Epoch 23/30, Loss: 0.05369510544263216
Epoch 24/30, Loss: 0.05181910722526913
Epoch 25/30, Loss: 0.05054031701357449
Epoch 26/30, Loss: 0.049619563752047326
Epoch 27/30, Loss: 0.04842851848370473
Epoch 28/30, Loss: 0.04695942106497948
Epoch 29/30, Loss: 0.046584334465431046
Epoch 30/30, Loss: 0.04564171017054667
```

4.1 Writing predictions to a file

In [311…

```
idx_to_vocab = {idx: word for word, idx in word_to_idx.items()}
idx_to_tag = {idx: tag for tag, idx in tag_to_idx.items()}

def write_predictions_to_file(model, data_loader, idx_to_tag, output_file_path, original_sentences, orig
    model.eval()
    predictions = []

    with torch.no_grad():
        for batch in data_loader:
            if len(batch) == 4:  # Tags are included in the batch
```

```python
            sentences, tags, lengths, indices = batch
        else:  # No tags are included, as in the test set
            sentences, lengths, indices = batch

        sentences = sentences.to(device)
        outputs = model(sentences)
        predicted_tag_indices = torch.argmax(outputs, dim=2)

        for i, index in enumerate(indices):
            original_index = index.item()  # Ensure you're getting the correct index as an integer.
            sentence_length = original_sentence_lengths[original_index]  # Use the original length fo
            for j in range(sentence_length):
                original_word = original_sentences[original_index][j]
                predicted_tag_index = predicted_tag_indices[i][j].item()
                predicted_tag = idx_to_tag[predicted_tag_index]
                predictions.append(f"{j+1} {original_word} {predicted_tag}\n")
            predictions.append("\n")

    with open(output_file_path, 'w') as writer:
        writer.writelines(predictions)

    print(f"Predictions written to {output_file_path}")



dev_original_lengths = [len(sentence) for sentence in dev_original_sentences]
test_original_lengths = [len(sentence) for sentence in test_original_sentences]

# Example usage for dev set
output_file_path = 'dev1.out'
write_predictions_to_file(model, dev_loader, idx_to_tag, output_file_path, dev_original_sentences, dev_o

# Example usage for test set
output_file_path = 'test1.out'
write_predictions_to_file(model, test_loader, idx_to_tag, output_file_path, test_original_sentences, tes
```

```
Predictions written to dev1.out
Predictions written to test1.out
```

5. Evaluation

In [312…

```python
predicted_file_path = 'dev1.out'
gold_standard_file_path = 'data/dev'

!python eval.py -p {predicted_file_path} -g {gold_standard_file_path}
```

```
processed 51578 tokens with 5942 phrases; found: 5592 phrases; correct: 4613.
accuracy:  95.89%; precision:  82.49%; recall:  77.63%; FB1:  79.99
              LOC: precision:  86.97%; recall:  86.50%; FB1:  86.74  1827
             MISC: precision:  83.72%; recall:  74.73%; FB1:  78.97  823
              ORG: precision:  76.85%; recall:  66.59%; FB1:  71.35  1162
              PER: precision:  81.01%; recall:  78.28%; FB1:  79.62  1780
```

5.1 Saving the model

In [313…

```python
import torch
model_save_path = 'blstm1.pt'
torch.save(model.state_dict(), model_save_path)
```

In [ ]:

TASK 2

Data Preprocessing
Vocabulary Building: The vocabulary is constructed based on word frequency, with a minimum frequency threshold of 2.
Encoding: Sentences are converted into sequences of indices based on the vocabulary. Tags are also encoded into numerical indices.

Model Architecture
Embedding Layer: Initialized with pre-trained GloVe embeddings, frozen during training.
BiLSTM Layer: Bidirectional LSTM layer to capture contextual information.
Linear Layer: Final output layer to predict NER tags.
Dropout: Applied for regularization.

Hyperparameters:
Data Preprocessing:
min_freq: 2

Model Architecture:
EMBEDDING_DIM: 100
HIDDEN_DIM: 256
OUTPUT_DIM: 128
DROPOUT: 0.33

Training:
N_EPOCHS: 40
lr: 0.0025 (initial learning rate)
factor: 0.3 (factor for reducing learning rate)
patience: 2 (number of epochs with no improvement before reducing learning rate)
Loss Function: Cross-Entropy Loss
Optimizer: RMSprop with momentum (alpha=0.99)
Learning Rate Scheduler: ReduceLROnPlateau

Result:

```
Predictions written to dev2.out
Predictions written to test2.out
processed 51578 tokens with 5942 phrases; found: 5517 phrases; correct: 4649.
accuracy:  96.27%; precision:  84.27%; recall:  78.24%; FB1:  81.14
              LOC: precision:  89.76%; recall:  83.51%; FB1:  86.52  1709
             MISC: precision:  84.22%; recall:  75.27%; FB1:  79.50  824
              ORG: precision:  76.92%; recall:  74.05%; FB1:  75.46  1291
              PER: precision:  84.35%; recall:  77.52%; FB1:  80.79  1693
```

1. Reading the CoNLL corpus

In [277…
```
def read_data(file_path, has_tags=True):
    sentences = []
    tags = []
    original_sentences = []

    with open(file_path, 'r') as file:
        sentence = []
        tag = []
        original_sentence = []
        for line in file:
            if line.strip() == "":
                if sentence:
                    sentences.append(sentence)
                    if has_tags:
                        tags.append(tag)
                    original_sentences.append(original_sentence)
                    sentence = []
                    tag = []
                    original_sentence = []
                continue

            components = line.strip().split()
            if has_tags:
                index, word, ner_tag = components
                tag.append(ner_tag)
            else:
                index, word = components

            sentence.append(word)
            original_sentence.append(word)

        if sentence:
            sentences.append(sentence)
            if has_tags:
                tags.append(tag)
            original_sentences.append(original_sentence)

    if has_tags:
        return sentences, tags, original_sentences
    else:
        return sentences, original_sentences
```

In [278…
```
train_file_path = 'data/train'
dev_file_path = 'data/dev'
test_file_path = 'data/test'

train_sentences, train_tags, train_original_sentences = read_data(train_file_path)
dev_sentences, dev_tags, dev_original_sentences = read_data(dev_file_path)
test_sentences, test_original_sentences = read_data(test_file_path, has_tags=False)
```

2. Datasets and Dataloaders

2.1 Create a Vocabulary and convert Text to Indices

In [279…
```
from collections import Counter

def build_vocab(sentences, min_freq=2):
    word_counts = Counter(word for sentence in sentences for word in sentence)

    vocab = [word for word, count in word_counts.items() if count >= min_freq]

    vocab.append('<UNK>')

    word_to_idx = {word: idx for idx, word in enumerate(vocab)}

    return vocab, word_to_idx
```

```
vocab, word_to_idx = build_vocab(train_sentences)

print(vocab[:10])
print(word_to_idx['<UNK>'])
```

```
['EU', 'German', 'call', 'to', 'boycott', 'British', 'lamb', '.', 'Peter', 'Blackburn']
11983
```

2.2 Encode Labels

In [280…

```
def encode_sentences(sentences, word_to_idx):
    encoded_sentences = []

    for sentence in sentences:
        encoded_sentence = [word_to_idx.get(word, word_to_idx['<UNK>']) for word in sentence]
        encoded_sentences.append(encoded_sentence)

    return encoded_sentences

train_encoded_sentences = encode_sentences(train_sentences, word_to_idx)
dev_encoded_sentences = encode_sentences(dev_sentences, word_to_idx)
test_encoded_sentences = encode_sentences(test_sentences, word_to_idx)
```

In [281…

```
def build_tag_vocab(tags):
    unique_tags = set(tag for tag_list in tags for tag in tag_list)
    tag_to_idx = {tag: idx for idx, tag in enumerate(unique_tags)}
    return unique_tags, tag_to_idx

unique_tags, tag_to_idx = build_tag_vocab(train_tags)

def encode_tags(tags, tag_to_idx):
    encoded_tags = [[tag_to_idx[tag] for tag in tag_list] for tag_list in tags]
    return encoded_tags

train_encoded_tags = encode_tags(train_tags, tag_to_idx)
dev_encoded_tags = encode_tags(dev_tags, tag_to_idx)
```

2.3 Create PyTorch Datasets

In [282…

```
import torch
from torch.utils.data import Dataset

class NERDataset(Dataset):
    def __init__(self, sentences, tags=None):
        self.sentences = sentences
        self.tags = tags

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

    def __getitem__(self, idx):
        sentence = torch.tensor(self.sentences[idx], dtype=torch.long)

        if self.tags is not None:
            tag = torch.tensor(self.tags[idx], dtype=torch.long)
            return sentence, tag, len(sentence)
        else:
            return sentence, len(sentence)


train_dataset = NERDataset(train_encoded_sentences, train_encoded_tags)
dev_dataset = NERDataset(dev_encoded_sentences, dev_encoded_tags)
test_dataset = NERDataset(test_encoded_sentences)
```

In [283…

```
from torch.nn.utils.rnn import pad_sequence

def pad_collate(batch):
    if len(batch[0]) == 3:
        sentences, tags, lengths = zip(*batch)
        sentences_padded = pad_sequence(sentences, batch_first=True, padding_value=word_to_idx['<UNK>'])
        tags_padded = pad_sequence(tags, batch_first=True, padding_value=tag_to_idx.get('O', 0))
```

```
        return sentences_padded, tags_padded, torch.tensor(lengths)
    else:
        sentences, lengths = zip(*batch)
        sentences_padded = pad_sequence(sentences, batch_first=True, padding_value=word_to_idx['<UNK>'])
        return sentences_padded, torch.tensor(lengths)
```

2.4 Create DataLoaders

In [284…
```
from torch.utils.data import DataLoader

BATCH_SIZE= 32
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=pad_collate)
dev_loader = DataLoader(dev_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate)
```

3. Model

In [285…
```
# Hyperparameters
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 128
DROPOUT = 0.33
```

In [286…
```
import torch

device = "mps"
```

TASK 2

In [287…
```
import gzip
import numpy as np

def load_glove_embeddings(file_path):
    embeddings_index = {}
    with gzip.open(file_path, 'rt', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            coefs = np.asarray(values[1:], dtype='float32')
            embeddings_index[word] = coefs
    return embeddings_index

glove_embeddings_path = 'glove.6B.100d.gz'
glove_embeddings = load_glove_embeddings(glove_embeddings_path)
```

In [288…
```
def prepare_embeddings_matrix(vocab, word_to_idx, glove_embeddings, embedding_dim):
    num_words = len(vocab)
    embedding_matrix = np.zeros((num_words, embedding_dim))
    for word, idx in word_to_idx.items():
        if word in glove_embeddings:
            embedding_matrix[idx] = glove_embeddings[word]
        else:
            embedding_matrix[idx] = np.random.normal(scale=0.6, size=(embedding_dim,))
    return embedding_matrix

embedding_matrix = prepare_embeddings_matrix(vocab, word_to_idx, glove_embeddings, EMBEDDING_DIM)
embedding_matrix_tensor = torch.FloatTensor(embedding_matrix).to(device)
```

In [289…
```
import torch
import torch.nn as nn

class BiLSTM_NER(nn.Module):
    def __init__(self, vocab_size, embedding_dim, lstm_hidden_dim, output_dim, dropout, embeddings):
        super(BiLSTM_NER, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(embeddings, freeze=True)

        self.bilstm = nn.LSTM(embedding_dim, lstm_hidden_dim, batch_first=True, bidirectional=True)

        self.dropout = nn.Dropout(dropout)
```

```python
        self.linear = nn.Linear(lstm_hidden_dim * 2, output_dim)
        self.elu = nn.ELU()
        self.classifier = nn.Linear(output_dim, len(tag_to_idx))

    def forward(self, sentence):
        embedded = self.embedding(sentence)
        lstm_out, _ = self.bilstm(embedded)

        lstm_out = self.dropout(lstm_out)

        linear_out = self.linear(lstm_out)
        elu_out = self.elu(linear_out)
        scores = self.classifier(elu_out)
        return scores

model = BiLSTM_NER(len(vocab), EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM, DROPOUT, embedding_matrix_tensor).t
```

In [290…

```python
# Hyperparameters
import torch.optim as optim
from torch.optim.lr_scheduler import ReduceLROnPlateau


N_EPOCHS = 40

# Loss Function
loss_function = nn.CrossEntropyLoss()

optimizer = optim.RMSprop(model.parameters(), lr=0.0025, alpha=0.99)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.3, patience=2)


# Training loop
for epoch in range(N_EPOCHS):
    model.train()
    total_loss = 0

    for sentence, tags, lengths in train_loader:
        sentence, tags = sentence.to(device), tags.to(device)
        model.zero_grad()

        tag_scores = model(sentence)

        tag_scores = tag_scores.view(-1, tag_scores.shape[-1])
        tags = tags.view(-1)

        loss = loss_function(tag_scores, tags)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch+1}/{N_EPOCHS}, Loss: {total_loss/len(train_loader)}")

    scheduler.step(total_loss/len(train_loader))
```

```
Epoch 1/40, Loss: 0.13455049336147207
Epoch 2/40, Loss: 0.05663739720474619
Epoch 3/40, Loss: 0.035877034489089236
Epoch 4/40, Loss: 0.024050709157030403
Epoch 5/40, Loss: 0.017513726052564026
Epoch 6/40, Loss: 0.012738870379370031
Epoch 7/40, Loss: 0.010463419123125801
Epoch 8/40, Loss: 0.008816287012473901
Epoch 9/40, Loss: 0.007311748910329437б
Epoch 10/40, Loss: 0.006505328522218681
Epoch 11/40, Loss: 0.005903841208056934
Epoch 12/40, Loss: 0.005216828587459155
Epoch 13/40, Loss: 0.004843631145530202
Epoch 14/40, Loss: 0.004306939234626966
Epoch 15/40, Loss: 0.004181493900632282
Epoch 16/40, Loss: 0.004066386770169466
Epoch 17/40, Loss: 0.0036569800873129154
Epoch 18/40, Loss: 0.0037084387664563768
Epoch 19/40, Loss: 0.0031682204449304074
Epoch 20/40, Loss: 0.0032181271551474026
Epoch 21/40, Loss: 0.0029513299575590936
Epoch 22/40, Loss: 0.0030349240246740393
Epoch 23/40, Loss: 0.0029052025128777055
Epoch 24/40, Loss: 0.0028517244742355566
Epoch 25/40, Loss: 0.002917672924855081
Epoch 26/40, Loss: 0.002892673230522237
Epoch 27/40, Loss: 0.0029315362386378163
Epoch 28/40, Loss: 0.0016473392800053224
Epoch 29/40, Loss: 0.001018638112442022
Epoch 30/40, Loss: 0.0009425911197824757
Epoch 31/40, Loss: 0.0008938103098157112
Epoch 32/40, Loss: 0.0007813348162557137
Epoch 33/40, Loss: 0.0007580211478232708
Epoch 34/40, Loss: 0.0008179817245435484
Epoch 35/40, Loss: 0.0007275771212349832
Epoch 36/40, Loss: 0.0006569220444036598
Epoch 37/40, Loss: 0.0007404868371952828
Epoch 38/40, Loss: 0.0006861727599077643
Epoch 39/40, Loss: 0.0006847835219984473
Epoch 40/40, Loss: 0.0006539665648165259
```

In [291…

```
idx_to_vocab = {idx: word for word, idx in word_to_idx.items()}
idx_to_tag = {idx: tag for tag, idx in tag_to_idx.items()}

def write_predictions_to_file_with_glove(model, data_loader, idx_to_tag, output_file_path, original_sente
    model.eval()
    predictions = []
    sentence_counter = 0

    with torch.no_grad():
        for batch in data_loader:
            if len(batch) == 3:
                sentences, lengths = batch[0], batch[2]
            else:
                sentences, lengths = batch[0], batch[1]

            sentences = sentences.to(device)
            outputs = model(sentences)
            predicted_tag_indices = torch.argmax(outputs, dim=2)

            for i, length in enumerate(lengths):
                original_sentence = original_sentences[sentence_counter]
                sentence_counter += 1

                for j in range(length.item()):
                    original_word = original_sentence[j] if j < len(original_sentence) else "<PAD>"
                    predicted_tag_index = predicted_tag_indices[i][j].item()
                    predicted_tag = idx_to_tag[predicted_tag_index]
                    predictions.append(f"{j+1} {original_word} {predicted_tag}\n")
                predictions.append("\n")

    with open(output_file_path, 'w') as writer:
        writer.writelines(predictions)
```

```
    print(f"Predictions written to {output_file_path}")
```

In [292…

```
output_file_path_dev = 'dev2.out'
write_predictions_to_file_with_glove(model, dev_loader, idx_to_tag, output_file_path_dev, dev_sentences)

output_file_path_test = 'test2.out'
write_predictions_to_file_with_glove(model, test_loader, idx_to_tag, output_file_path_test, test_sentence


predicted_file_path_glove = output_file_path_dev
gold_standard_file_path = 'data/dev'

!python eval.py -p {predicted_file_path_glove} -g {gold_standard_file_path}
```

```
Predictions written to dev2.out
Predictions written to test2.out
processed 51578 tokens with 5942 phrases; found: 5517 phrases; correct: 4649.
accuracy:  96.27%; precision:  84.27%; recall:  78.24%; FB1:  81.14
              LOC: precision:  89.76%; recall:  83.51%; FB1:  86.52  1709
             MISC: precision:  84.22%; recall:  75.27%; FB1:  79.50  824
              ORG: precision:  76.92%; recall:  74.05%; FB1:  75.46  1291
              PER: precision:  84.35%; recall:  77.52%; FB1:  80.79  1693
```

In [293…

```
import torch
torch.save(model.state_dict(), 'blstm2.pt')
```

In [ ]:

TASK 3

Vocabulary Building:
Function: build_vocab(sentences, min_freq=2)

Model Architecture:
Class: BiLSTM_CNN_NER(nn.Module)
Combines Bidirectional LSTM with CNN for Named Entity Recognition.
Embedding layer, Character-level CNN, Bidirectional LSTM, Dropout, Linear layer, and Output layer are included.

Hyperparameters:

Embedding Dimension: 100
Hidden Dimension: 256
Output Dimension: 128
Character Embedding Dimension: 30
Number of Filters: 125
Kernel Sizes: [1, 2]
Dropout Rate: 0.5

Training
Number of Epochs: 20
Loss Function: CrossEntropyLoss
Optimizer: SGD
Learning Rate: 1
Scheduler: ReduceLROnPlateau
Mode: Min
Factor: 0.1
Patience: 2

Result:
```
Predictions written to dev3.out
Predictions written to test3.out
processed 51578 tokens with 5942 phrases; found: 5522 phrases; correct: 4544.
accuracy:  95.98%; precision:  82.29%; recall:  76.47%; FB1:  79.27
              LOC: precision:  84.87%; recall:  86.12%; FB1:  85.49  1864
             MISC: precision:  83.86%; recall:  74.40%; FB1:  78.85  818
              ORG: precision:  76.27%; recall:  68.53%; FB1:  72.19  1205
              PER: precision:  83.00%; recall:  73.67%; FB1:  78.06  1635
```

1. Reading the CoNLL corpus

In [228…

```
def read_data(file_path, has_tags=True):
    sentences = []
    tags = []
    original_sentences = []

    with open(file_path, 'r') as file:
        sentence = []
        tag = []
        original_sentence = []
        for line in file:
            if line.strip() == "":
                if sentence:
                    sentences.append(sentence)
                    if has_tags:
                        tags.append(tag)
                    original_sentences.append(original_sentence)
                    sentence = []
                    tag = []
                    original_sentence = []
                continue

            components = line.strip().split()
            if has_tags:
                index, word, ner_tag = components
                tag.append(ner_tag)
            else:
                index, word = components

            sentence.append(word)
            original_sentence.append(word)

        if sentence:
            sentences.append(sentence)
            if has_tags:
                tags.append(tag)
            original_sentences.append(original_sentence)

    if has_tags:
        return sentences, tags, original_sentences
    else:
        return sentences, original_sentences
```

In [229…

```
train_file_path = 'data/train'
dev_file_path = 'data/dev'
test_file_path = 'data/test'

train_sentences, train_tags, train_original_sentences = read_data(train_file_path)
dev_sentences, dev_tags, dev_original_sentences = read_data(dev_file_path)
test_sentences, test_original_sentences = read_data(test_file_path, has_tags=False)
```

2. Datasets and Dataloaders

2.1 Create a Vocabulary and convert Text to Indices

In [230…

```
from collections import Counter

def build_vocab(sentences, min_freq=2):

    word_counts = Counter(word for sentence in sentences for word in sentence)


    vocab = [word for word, count in word_counts.items() if count >= min_freq]

    vocab.append('<UNK>')


    word_to_idx = {word: idx for idx, word in enumerate(vocab)}
```

```python
    return vocab, word_to_idx


vocab, word_to_idx = build_vocab(train_sentences)

print(vocab[:10])
print(word_to_idx['<UNK>'])
```

```
['EU', 'German', 'call', 'to', 'boycott', 'British', 'lamb', '.', 'Peter', 'Blackburn']
11983
```

2.2 Encode Labels

In [231…

```python
def encode_sentences(sentences, word_to_idx):
    encoded_sentences = []

    for sentence in sentences:
        encoded_sentence = [word_to_idx.get(word, word_to_idx['<UNK>']) for word in sentence]
        encoded_sentences.append(encoded_sentence)

    return encoded_sentences

train_encoded_sentences = encode_sentences(train_sentences, word_to_idx)
dev_encoded_sentences = encode_sentences(dev_sentences, word_to_idx)
test_encoded_sentences = encode_sentences(test_sentences, word_to_idx)
```

In [232…

```python
def build_tag_vocab(tags):
    unique_tags = set(tag for tag_list in tags for tag in tag_list)
    tag_to_idx = {tag: idx for idx, tag in enumerate(unique_tags)}
    return unique_tags, tag_to_idx

unique_tags, tag_to_idx = build_tag_vocab(train_tags)

def encode_tags(tags, tag_to_idx):
    encoded_tags = [[tag_to_idx[tag] for tag in tag_list] for tag_list in tags]
    return encoded_tags

train_encoded_tags = encode_tags(train_tags, tag_to_idx)
dev_encoded_tags = encode_tags(dev_tags, tag_to_idx)
```

2.3 Create PyTorch Datasets

In [233…

```python
import torch
from torch.utils.data import Dataset

class NERDataset(Dataset):
    def __init__(self, sentences, tags=None, char_sentences=None):
        self.sentences = sentences
        self.tags = tags
        self.char_sentences = char_sentences

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

    def __getitem__(self, idx):
        sentence = torch.tensor(self.sentences[idx], dtype=torch.long)


        if self.tags is not None:
            tag = torch.tensor(self.tags[idx], dtype=torch.long)
            if self.char_sentences is not None:
                char_sentence = torch.tensor(self.char_sentences[idx], dtype=torch.long)
                return sentence, tag, char_sentence, len(sentence)
            else:
                return sentence, tag, len(sentence)
        else:
            if self.char_sentences is not None:
                char_sentence = torch.tensor(self.char_sentences[idx], dtype=torch.long)
                return sentence, char_sentence, len(sentence)
            else:
                return sentence, len(sentence)
```

```python
train_char_sequences = []
for sentence in train_sentences:
    for word in sentence:
        char_sequence = []
        for char in word:
            if char in word_to_idx:
                char_sequence.append(word_to_idx[char])
            else:
                char_sequence.append(word_to_idx['<UNK>'])
        train_char_sequences.append(char_sequence)

dev_char_sequences = []
for sentence in dev_sentences:
    for word in sentence:
        char_sequence = []
        for char in word:
            if char in word_to_idx:
                char_sequence.append(word_to_idx[char])
            else:
                char_sequence.append(word_to_idx['<UNK>'])
        dev_char_sequences.append(char_sequence)

test_char_sequences = []
for sentence in test_sentences:
    for word in sentence:
        char_sequence = []
        for char in word:
            if char in word_to_idx:
                char_sequence.append(word_to_idx[char])
            else:
                char_sequence.append(word_to_idx['<UNK>'])
        test_char_sequences.append(char_sequence)

train_dataset = NERDataset(train_encoded_sentences, train_encoded_tags, train_char_sequences)
dev_dataset = NERDataset(dev_encoded_sentences, dev_encoded_tags, dev_char_sequences)
test_dataset = NERDataset(test_encoded_sentences, char_sentences=test_char_sequences)
```

2.4 Create DataLoaders

In [234…

```python
from torch.utils.data import DataLoader

BATCH_SIZE= 8
```

In [235…

```python
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader


def pad_collate(batch):
    if len(batch[0]) == 4:
        sentences, tags, char_sentences, lengths = zip(*batch)
        tags_padded = pad_sequence(tags, batch_first=True, padding_value=tag_to_idx['O'])
    else:
        sentences, char_sentences, lengths = zip(*batch)
        tags_padded = None

    sentences_padded = pad_sequence(sentences, batch_first=True, padding_value=word_to_idx['<UNK>'])
    char_sentences_padded = pad_sequence(char_sentences, batch_first=True, padding_value=0)

    return (sentences_padded, tags_padded, char_sentences_padded, lengths) if tags_padded is not None els



train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=pad_collate)
dev_loader = DataLoader(dev_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate)
```

3. Model

In [236…
```
# Hyperparameters
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 128
# DROPOUT = 0.33
```

In [237…
```
import torch

device = "mps"
```

TASK 2

In [238…
```
import gzip
import numpy as np

def load_glove_embeddings(file_path):
    embeddings_index = {}
    with gzip.open(file_path, 'rt', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            coefs = np.asarray(values[1:], dtype='float32')
            embeddings_index[word] = coefs
    return embeddings_index

glove_embeddings_path = 'glove.6B.100d.gz'
glove_embeddings = load_glove_embeddings(glove_embeddings_path)
```

In [239…
```
def prepare_embeddings_matrix(vocab, word_to_idx, glove_embeddings, embedding_dim):
    num_words = len(vocab)
    embedding_matrix = np.zeros((num_words, embedding_dim))
    for word, idx in word_to_idx.items():
        if word in glove_embeddings:
            embedding_matrix[idx] = glove_embeddings[word]
        else:
            embedding_matrix[idx] = np.random.normal(scale=0.6, size=(embedding_dim,))
    return embedding_matrix

embedding_matrix = prepare_embeddings_matrix(vocab, word_to_idx, glove_embeddings, EMBEDDING_DIM)
embedding_matrix_tensor = torch.FloatTensor(embedding_matrix).to(device)
```

In [240…
```
import torch
import torch.nn as nn
import torch.nn.functional as F
class BiLSTM_CNN_NER(nn.Module):
    def __init__(self, vocab_size, embedding_dim, lstm_hidden_dim, output_dim, dropout, embeddings, char_
        super(BiLSTM_CNN_NER, self).__init__()

        self.embedding = nn.Embedding.from_pretrained(embeddings, freeze=True)

        self.char_embedding = nn.Embedding(char_vocab_size, char_embedding_dim)

        self.conv_layers = nn.ModuleList([
            nn.Conv1d(in_channels=char_embedding_dim, out_channels=num_filters, kernel_size=kernel_size,
            for kernel_size in kernel_sizes
        ])

        self.bilstm = nn.LSTM(embedding_dim + num_filters * len(kernel_sizes), lstm_hidden_dim, batch_fi

        self.dropout = nn.Dropout(dropout)

        self.linear = nn.Linear(lstm_hidden_dim*2, output_dim)

        self.elu = nn.ELU()

        self.classifier = nn.Linear(output_dim, len(tag_to_idx))

    def forward(self, sentence, char_sentence):

        embedded = self.embedding(sentence)
```

```python
        char_embedded = self.char_embedding(char_sentence)
        char_embedded = char_embedded.permute(0, 2, 1)
        char_conv_outputs = [self.elu(conv(char_embedded)) for conv in self.conv_layers]
        char_pooled = [F.max_pool1d(conv_output, conv_output.size(2)).squeeze(2) for conv_output in char_
        char_output = torch.cat(char_pooled, dim=1)

        combined_embedded = torch.cat((embedded, char_output.unsqueeze(1).repeat(1, embedded.size(1), 1)

        lstm_out, _ = self.bilstm(combined_embedded)

        lstm_out = self.dropout(lstm_out)

        linear_out = self.linear(lstm_out)

        elu_out = self.elu(linear_out)

        scores = self.classifier(elu_out)

        return scores
```

In [241…

```python
CHAR_EMBEDDING_DIM = 30
NUM_FILTERS = 125
KERNEL_SIZES = [1, 2]
DROPOUT = 0.5

char_vocab = set()
for sentence in train_sentences:
    for word in sentence:
        for char in word:
            char_vocab.add(char)

char_vocab_size = len(char_vocab)

model = BiLSTM_CNN_NER(len(vocab), EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM, DROPOUT, embedding_matrix_tens
                       len(char_vocab), CHAR_EMBEDDING_DIM, NUM_FILTERS, KERNEL_SIZES).to(device)
```

In [242…

```python
# Hyperparameters
import torch.optim as optim
from torch.optim.lr_scheduler import ReduceLROnPlateau


N_EPOCHS = 20

# Loss Function
loss_function = nn.CrossEntropyLoss()

# Optimizer
optimizer = optim.SGD(model.parameters(), lr=1)

scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=2)

for epoch in range(N_EPOCHS):
    model.train()
    total_loss = 0

    for sentence, tags, char_sentence, lengths in train_loader:
        sentence, tags, char_sentence = sentence.to(device), tags.to(device), char_sentence.to(device)
        model.zero_grad()

        tag_scores = model(sentence, char_sentence)

        tag_scores = tag_scores.view(-1, tag_scores.shape[-1])
        tags = tags.view(-1)

        loss = loss_function(tag_scores, tags)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch+1}/{N_EPOCHS}, Loss: {total_loss/len(train_loader)}")

    scheduler.step(total_loss/len(train_loader))
```

```
Epoch 1/20, Loss: 0.183519840904431
Epoch 2/20, Loss: 0.11965901552517615
Epoch 3/20, Loss: 0.09888919006005374
Epoch 4/20, Loss: 0.08344688390919854
Epoch 5/20, Loss: 0.07290144444874656
Epoch 6/20, Loss: 0.06382609958566125
Epoch 7/20, Loss: 0.0594231771829383
Epoch 8/20, Loss: 0.05184958007444082
Epoch 9/20, Loss: 0.04601403719937158
Epoch 10/20, Loss: 0.04283737077822152
Epoch 11/20, Loss: 0.03893343903992291
Epoch 12/20, Loss: 0.03520625608103755
Epoch 13/20, Loss: 0.03252760881883639
Epoch 14/20, Loss: 0.02989004459829233
Epoch 15/20, Loss: 0.02730227616262154
Epoch 16/20, Loss: 0.025647192665963686
Epoch 17/20, Loss: 0.023330270680177886
Epoch 18/20, Loss: 0.022360962188615736
Epoch 19/20, Loss: 0.02048120272649327
Epoch 20/20, Loss: 0.01880298172249495
```

In [243…

```
idx_to_vocab = {idx: word for word, idx in word_to_idx.items()}
idx_to_tag = {idx: tag for tag, idx in tag_to_idx.items()}

def write_predictions_to_file(model, data_loader, idx_to_tag, output_file_path, original_sentences):
    model.eval()
    predictions = []

    with torch.no_grad():
        for batch_idx, batch in enumerate(data_loader):
            if len(batch) == 4:
                sentence_tensors, _, char_sentence_tensors, lengths = batch
            else:
                sentence_tensors, char_sentence_tensors, lengths = batch

            sentence_tensors = sentence_tensors.to(device)
            char_sentence_tensors = char_sentence_tensors.to(device)
            outputs = model(sentence_tensors, char_sentence_tensors)
            predicted_tag_indices = torch.argmax(outputs, dim=2)

            for i, length in enumerate(lengths):
                original_sentence = original_sentences[batch_idx * data_loader.batch_size + i]
                for j in range(length):
                    original_word = original_sentence[j]
                    predicted_tag_index = predicted_tag_indices[i][j].item()
                    predicted_tag = idx_to_tag[predicted_tag_index]
                    predictions.append(f"{j+1} {original_word} {predicted_tag}\n")
                predictions.append("\n")

    with open(output_file_path, 'w') as writer:
        writer.writelines(predictions)

    print(f"Predictions written to {output_file_path}")
```

In [244…

```
dev_output_file_path = 'dev3.out'
write_predictions_to_file(model, dev_loader, idx_to_tag, dev_output_file_path, dev_sentences)

test_output_file_path = 'test3.out'
write_predictions_to_file(model, test_loader, idx_to_tag, test_output_file_path, test_sentences)

predicted_file_path_glove = output_file_path_glove
gold_standard_file_path = 'data/dev'

!python eval.py -p {dev_output_file_path} -g {gold_standard_file_path}
```

```
Predictions written to dev3.out
Predictions written to test3.out
processed 51578 tokens with 5942 phrases; found: 5522 phrases; correct: 4544.
accuracy:  95.98%; precision:  82.29%; recall:  76.47%; FB1:  79.27
              LOC: precision:  84.87%; recall:  86.12%; FB1:  85.49  1864
             MISC: precision:  83.86%; recall:  74.40%; FB1:  78.85  818
              ORG: precision:  76.27%; recall:  68.53%; FB1:  72.19  1205
              PER: precision:  83.00%; recall:  73.67%; FB1:  78.06  1635
```

In [245…

```
import torch
torch.save(model.state_dict(), 'blstm3.pt')
```

In [ ]: