CSCI 544 - Homework 4

Submitted By: Yamini Haripriya Kanuparthi
USC ID : 9195653004

Task 1: Simple Bidirectional LSTM model

I have developed a bidirectional LSTM model. To facilitate this, I have established several mappings, which are:

- A mapping from words to indices and vice versa (`word2idx` and `idx2word`)
- A mapping from tags to indices and vice versa (`tag2idx` and `idx2tag`)

The model architecture I have created for the task 1 is as follows:

```
Simple_BiLSTM(
  (embedding): Embedding(11985, 100)
  (cap_embedding): Embedding(2, 50)
  (lstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (fc1): Linear(in_features=512, out_features=128, bias=True)
  (activation): ELU(alpha=1.0)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

The batch size I chose is 16, the learning rate is 0.5 and I used learning rate scheduler StepLR for better performance of the model

What are the precision, recall and F1 score on the dev data?

```
processed 51578 tokens with 5942 phrases; found: 5600 phrases; correct: 4402.
accuracy:  95.40%; precision:  78.61%; recall:  74.08%; FB1:  76.28
              LOC: precision:  88.48%; recall:  80.68%; FB1:  84.40  1675
             MISC: precision:  81.01%; recall:  74.51%; FB1:  77.63  848
              ORG: precision:  68.01%; recall:  68.16%; FB1:  68.08  1344
              PER: precision:  76.11%; recall:  71.61%; FB1:  73.79  1733
```

Task 2: Using GloVe word embeddings

Task 2: Construction of a Basic Bidirectional LSTM Model (Worth 60 Points)

For this task, I incorporated every word in the vocabulary without considering word frequency. A bidirectional LSTM model has been developed. As part of this process, I established the following mappings:

- `word2idx` for converting words to indices and `idx2word` for the reverse
- `tag2idx` for tagging to indices and `idx2tag` for the reverse

You are asked to find a way to deal with this conflict(mentioned in HW4 question):

Additionally, I've implemented a boolean mask to address issues with word capitalization.

The model architecture I have created for the task 1 is as follows:

```
Glove_BiLSTM(
  (cap_embedding): Embedding(3, 100)
  (embedding): Embedding(400002, 100)
  (blstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=10, bias=True)
)
```

I used the following hyper parameters: Batch Size = 64, Optimizer = SGD, Learning Rate = 0.5, Scheduler = StepLR(optimizer, step_size=10, gamma=0.9) and Epochs = 50

What are the precision, recall and F1 score on the dev data?

```
processed 51578 tokens with 5942 phrases; found: 6106 phrases; correct: 5459.
accuracy:  98.50%; precision:  89.40%; recall:  91.87%; FB1:  90.62
              LOC: precision:  94.56%; recall:  94.67%; FB1:  94.61  1839
             MISC: precision:  79.36%; recall:  85.47%; FB1:  82.30  993
              ORG: precision:  83.99%; recall:  86.43%; FB1:  85.19  1380
              PER: precision:  93.61%; recall:  96.25%; FB1:  94.91  1894
```

Task 3: LSTM-CNN model

I have included every term in our vocabulary regardless of how often each term appears. I have developed a bidirectional LSTM model that is enhanced with a CNN for generating character-level embeddings. To support this, I have created several mappings as listed below:

- Word2idx for word to index mapping and idx2word for index to word mapping
- Tag2idx for tag to index mapping and idx2tag for index to tag mapping
- Char2idx for character to index mapping

The hyper parameters I used are Batch Size = 128, Optimizer = SGD, Learning Rate = 0.00, Epochs = 200

Performance:

```
processed 51578 tokens with 5942 phrases; found: 7136 phrases; correct: 4077.
accuracy:  93.79%; precision:  57.13%; recall:  68.61%; FB1:  62.35
              LOC: precision:  72.83%; recall:  82.14%; FB1:  77.21  2072
             MISC: precision:  61.09%; recall:  70.50%; FB1:  65.46  1064
              ORG: precision:  44.78%; recall:  61.74%; FB1:  51.91  1849
              PER: precision:  50.67%; recall:  59.17%; FB1:  54.60  2151
```

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from tqdm import tqdm
import json
from sklearn.preprocessing import LabelEncoder
from torch.nn.utils.rnn import pad_sequence
from collections import Counter
import os
```

In [2]:
```
train_split = "/content/drive/MyDrive/nlp_hw4(dataset)/train"
dev_split = "/content/drive/MyDrive/nlp_hw4(dataset)/dev"
test_split = "/content/drive/MyDrive/nlp_hw4(dataset)/test"
```

In [3]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [4]:
```
class Simple_BiLSTM(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, output_dim, dropout):
        super(Simple_BiLSTM, self).__init__()
        self.hidden_dim = hidden_dim
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.cap_embedding = nn.Embedding(2, 50)
        self.lstm = nn.LSTM(100, hidden_dim, batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(dropout)
        self.fc1 = nn.Linear(hidden_dim*2, 128)
        self.activation = nn.ELU()
        self.fc2 = nn.Linear(128, output_dim)


    def forward(self, sentence):
        embedded = self.embedding(sentence)
        lstm_out, _ = self.lstm(embedded)
        fc1_out = self.fc1(lstm_out)
        activation_out = self.activation(fc1_out)
        output = self.fc2(activation_out)

        return output
```

In [5]:
```
unk_threshold = 1
PAD = '<PAD>'
```

```
UNK = '<UNK>'
BATCH_SIZE = 16
```

In [6]:
```
def DataPreprocess(file):
    with open(file) as f:
        all_str = f.read()
        sentences = all_str.split("\n\n")

    sentences = [[i for i in sen.split("\n") if i] for sen in sentences]
    words = [[line.split()[1] for line in sen] for sen in sentences]

    # Count word frequencies
    word_freqs = Counter([word for sentence in words for word in sentence])

    # Filter out words with frequency <= unk_th
    words_set = [word for word, freq in word_freqs.items() if freq > unk_threshold]

    words = [[word if word_freqs[word] > unk_threshold else UNK for word in sen] for sen in words]

    ners = [[line.split()[2] for line in sen] for sen in sentences]
    ners_set = list(set([line.split()[2] for sen in sentences for line in sen]))

    # Add PAD and UNK tokens to words_set
    words_set = [PAD] + [UNK] + words_set

    # Update ners_set with PAD token
    ners_set = [PAD] + ners_set

    words = [[word if word in words_set else UNK for word in sentence] for sentence in words]
    return words, ners, words_set, ners_set
```

In [7]:
```
def convertFileToTensor(input_file):
    words, ners, words_set, ners_set = DataPreprocess(input_file)

    word_encoder, tag_encoder = LabelEncoder(), LabelEncoder()

    word_indices = word_encoder.fit_transform(words_set)
    tag_indices = tag_encoder.fit_transform(ners_set)

    word2idx = {}
    for i, word in enumerate(words_set):
        word2idx[word] = word_indices[i]

    tag2idx = {}
```

```
    for i, tag in enumerate(ners_set):
        tag2idx[tag] = tag_indices[i]


    # Convert the sentences to indices
    sentences_word_indices = [[word2idx[word] for word in sentence] for sentence in words]
    sentences_tag_indices = [[tag2idx[tag] for tag in sentence] for sentence in ners]

    return sentences_word_indices, sentences_tag_indices, len(word2idx), len(tag2idx), word2idx, tag2idx
```

In [8]:

```
def convertFileToTensor_test(file, word2idx, tag2idx):
    # Open the file and read all its contents into a single string
    with open(file) as f:
        all_str = f.read()

    # Split the string into sentences based on double newline characters
    sentences = all_str.split("\n\n")
    # Further split each sentence into lines, filtering out empty lines
    sentences = [[i for i in sen.split("\n") if i] for sen in sentences]
    # For each sentence, extract the words, which are assumed to be the second element on each line
    words = [[line.split()[1] for line in sen] for sen in sentences]
    # Similarly, extract the named entity tags, which are assumed to be the third element on each line
    ners = [[line.split()[2] for line in sen] for sen in sentences]

    # Convert each word in each sentence to its corresponding index in word2idx
    # Use the index for UNK (unknown) if the word is not in the dictionary
    sentences_word_indices = [
        [word2idx[word] if word in word2idx else word2idx[UNK] for word in sentence]
        for sentence in words
    ]

    # Convert each named entity tag in each sentence to its corresponding index in tag2idx
    sentences_tag_indices = [
        [tag2idx[tag] for tag in sentence] for sentence in ners
    ]

    # Return the list of word indices and the list of tag indices
    return sentences_word_indices, sentences_tag_indices
```

In [9]:

```
def collate_fn(batch):
    padded_word_indices = pad_sequence([b[0] for b in batch], batch_first=True, padding_value=0)
    padded_tag_indices = pad_sequence([b[1] for b in batch], batch_first=True, padding_value=0)
    return padded_word_indices, padded_tag_indices
```

In [10]:
```
class CustomDataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

    def __getitem__(self, idx):
        return torch.tensor(self.x[idx]), torch.tensor(self.y[idx])
```

In [11]:
```
train_padded_word_indices, train_padded_tag_indices, vocab_size, num_classes, word2idx, tag2idx = convertFileToTensor(t
train_dataset = CustomDataset(train_padded_word_indices, train_padded_tag_indices)
train_loader = DataLoader(train_dataset, collate_fn=collate_fn, batch_size=BATCH_SIZE, shuffle=True)
```

In [12]:
```
dev_padded_word_indices, dev_padded_tag_indices = convertFileToTensor_test(dev_split, word2idx, tag2idx)
dev_dataset = CustomDataset(dev_padded_word_indices, dev_padded_tag_indices)
dev_dataloader = DataLoader(dev_dataset, collate_fn=collate_fn, batch_size=BATCH_SIZE)
```

In [13]:
```
model = Simple_BiLSTM(embedding_dim=100, hidden_dim=256, output_dim=10, dropout=0.33).to(device)
model.to(device)
```

Out[13]:
```
Simple_BiLSTM(
  (embedding): Embedding(11985, 100)
  (cap_embedding): Embedding(2, 50)
  (lstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (fc1): Linear(in_features=512, out_features=128, bias=True)
  (activation): ELU(alpha=1.0)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)
```

In [14]:
```
# saving json files
#for key, value in word2idx.items():
 # word2idx[key] = int(value)
#with open("word2idx_task1.json", "w+") as f:
 # json.dump(word2idx, f)
#idx2word = {v:k for k, v in word2idx.items()}
#with open("idx2word_task1.json", "w+") as f:
 # json.dump(idx2word, f)
#for key, value in tag2idx.items():
 # tag2idx[key] = int(value)
#with open("tag2idx_task1.json", "w+") as f:
 # json.dump(tag2idx, f)
```

In [15]:
```
criterion = nn.CrossEntropyLoss(ignore_index=tag2idx[PAD])
optimizer = optim.SGD(model.parameters(), lr=0.5, weight_decay=1e-5)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.9)
num_epochs = 100
```

In [16]:
```
from sklearn.metrics import f1_score

def calculate_f1(model, dataloader, device, tag2idx):
    true_tags = []
    predicted_tags = []

    model.eval()
    with torch.no_grad():
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            predictions = outputs.argmax(dim=2)

            # Flatten the batch and remove PAD tokens
            for i in range(inputs.size(0)):  # Loop over the batch
                for j in range(inputs.size(1)):  # Loop over sequence length
                    if targets[i, j] != tag2idx[PAD]:  # Check if it's not a PAD token
                        true_tags.append(targets[i, j].item())
                        predicted_tags.append(predictions[i, j].item())

    return f1_score(true_tags, predicted_tags, average='macro')
```

In [17]:
```
# Initialize variables to track the best F1 score
best_f1 = 0.0

# Training Loop
for epoch in tqdm(range(num_epochs)):
    model.train()
    epoch_loss = 0
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        outputs = outputs.view(-1, num_classes)

        loss = criterion(outputs, targets.view(-1))
        epoch_loss += loss.item()
```

```
            loss.backward()
            # Implement gradient clipping
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

            optimizer.step()
        scheduler.step()

        # Calculate F1 score on development set
        f1 = calculate_f1(model, dev_dataloader, device, tag2idx)

        # Save model if F1 score is the best so far
        if f1 > best_f1:
            best_f1 = f1
            torch.save(model.state_dict(), 'blstm1.pt')

        # Print epoch metrics
        print(f'Epoch {epoch+1}/{num_epochs}: Loss={epoch_loss:.4f}, F1 Score on Dev Set: {f1:.4f}')
```

```
  1%|          | 1/100 [00:14<23:47, 14.42s/it]
Epoch 1/100: Loss=532.9920, F1 Score on Dev Set: 0.3938
  2%|▏         | 2/100 [00:27<22:05, 13.53s/it]
Epoch 2/100: Loss=307.5179, F1 Score on Dev Set: 0.6402
  3%|▎         | 3/100 [00:39<20:54, 12.94s/it]
Epoch 3/100: Loss=194.5874, F1 Score on Dev Set: 0.6969
  4%|▍         | 4/100 [00:51<20:15, 12.66s/it]
Epoch 4/100: Loss=129.1701, F1 Score on Dev Set: 0.7589
  5%|▌         | 5/100 [01:04<19:49, 12.52s/it]
Epoch 5/100: Loss=90.7765, F1 Score on Dev Set: 0.7290
  6%|▌         | 6/100 [01:16<19:27, 12.42s/it]
Epoch 6/100: Loss=65.8132, F1 Score on Dev Set: 0.7985
  7%|▋         | 7/100 [01:28<19:11, 12.39s/it]
Epoch 7/100: Loss=49.5241, F1 Score on Dev Set: 0.8098
  8%|▊         | 8/100 [01:40<18:43, 12.22s/it]
Epoch 8/100: Loss=37.9324, F1 Score on Dev Set: 0.8127
  9%|▉         | 9/100 [01:57<20:52, 13.77s/it]
Epoch 9/100: Loss=29.0048, F1 Score on Dev Set: 0.8188
 10%|█         | 10/100 [02:09<19:55, 13.28s/it]
Epoch 10/100: Loss=23.1377, F1 Score on Dev Set: 0.8111
 11%|█         | 11/100 [02:22<19:12, 12.95s/it]
Epoch 11/100: Loss=17.0981, F1 Score on Dev Set: 0.8161
 12%|█▏        | 12/100 [02:33<18:29, 12.61s/it]
```

```
Epoch 12/100: Loss=13.7783, F1 Score on Dev Set: 0.7872
 13%|█▎        | 13/100 [02:45<17:58, 12.40s/it]
Epoch 13/100: Loss=11.3171, F1 Score on Dev Set: 0.8213
 14%|█▍        | 14/100 [02:57<17:39, 12.32s/it]
Epoch 14/100: Loss=9.2952, F1 Score on Dev Set: 0.8233
 15%|█▌        | 15/100 [03:11<17:51, 12.61s/it]
Epoch 15/100: Loss=7.4442, F1 Score on Dev Set: 0.8159
 16%|█▌        | 16/100 [03:24<18:03, 12.90s/it]
Epoch 16/100: Loss=6.4483, F1 Score on Dev Set: 0.8129
 17%|█▋        | 17/100 [03:36<17:31, 12.67s/it]
Epoch 17/100: Loss=6.5177, F1 Score on Dev Set: 0.8175
 18%|█▊        | 18/100 [03:49<17:05, 12.51s/it]
Epoch 18/100: Loss=5.8737, F1 Score on Dev Set: 0.8202
 19%|█▉        | 19/100 [04:01<16:47, 12.44s/it]
Epoch 19/100: Loss=4.7819, F1 Score on Dev Set: 0.8178
 20%|██        | 20/100 [04:13<16:30, 12.39s/it]
Epoch 20/100: Loss=4.6995, F1 Score on Dev Set: 0.8167
 21%|██        | 21/100 [04:25<16:14, 12.33s/it]
Epoch 21/100: Loss=3.9541, F1 Score on Dev Set: 0.8132
 22%|██▏       | 22/100 [04:38<16:06, 12.39s/it]
Epoch 22/100: Loss=3.2816, F1 Score on Dev Set: 0.8181
 23%|██▎       | 23/100 [04:50<15:47, 12.30s/it]
Epoch 23/100: Loss=3.5370, F1 Score on Dev Set: 0.8143
 24%|██▍       | 24/100 [05:02<15:31, 12.26s/it]
Epoch 24/100: Loss=2.9536, F1 Score on Dev Set: 0.8188
 25%|██▌       | 25/100 [05:14<15:18, 12.25s/it]
Epoch 25/100: Loss=3.0395, F1 Score on Dev Set: 0.8191
 26%|██▌       | 26/100 [05:26<15:05, 12.23s/it]
Epoch 26/100: Loss=2.8409, F1 Score on Dev Set: 0.8189
 27%|██▋       | 27/100 [05:39<14:49, 12.19s/it]
Epoch 27/100: Loss=2.7625, F1 Score on Dev Set: 0.8145
 28%|██▊       | 28/100 [05:50<14:25, 12.03s/it]
Epoch 28/100: Loss=2.8701, F1 Score on Dev Set: 0.8157
 29%|██▉       | 29/100 [06:02<14:10, 11.98s/it]
Epoch 29/100: Loss=2.7827, F1 Score on Dev Set: 0.8132
 30%|███       | 30/100 [06:14<14:02, 12.03s/it]
Epoch 30/100: Loss=2.6492, F1 Score on Dev Set: 0.8158
 31%|███       | 31/100 [06:26<13:52, 12.07s/it]
Epoch 31/100: Loss=2.4318, F1 Score on Dev Set: 0.8129
```

```
 32%|███▏      | 32/100 [06:39<13:41, 12.08s/it]
Epoch 32/100: Loss=2.1373, F1 Score on Dev Set: 0.8138
 33%|███▎      | 33/100 [06:51<13:32, 12.12s/it]
Epoch 33/100: Loss=2.0592, F1 Score on Dev Set: 0.8199
 34%|███▍      | 34/100 [07:03<13:23, 12.17s/it]
Epoch 34/100: Loss=2.1914, F1 Score on Dev Set: 0.8117
 35%|███▌      | 35/100 [07:16<13:29, 12.46s/it]
Epoch 35/100: Loss=2.1045, F1 Score on Dev Set: 0.8169
 36%|███▌      | 36/100 [07:28<13:11, 12.37s/it]
Epoch 36/100: Loss=2.0459, F1 Score on Dev Set: 0.8124
 37%|███▋      | 37/100 [07:41<12:56, 12.33s/it]
Epoch 37/100: Loss=1.8071, F1 Score on Dev Set: 0.8155
 38%|███▊      | 38/100 [07:53<12:43, 12.31s/it]
Epoch 38/100: Loss=1.9206, F1 Score on Dev Set: 0.8182
 39%|███▉      | 39/100 [08:05<12:30, 12.31s/it]
Epoch 39/100: Loss=1.9509, F1 Score on Dev Set: 0.8113
 40%|████      | 40/100 [08:17<12:14, 12.25s/it]
Epoch 40/100: Loss=2.2300, F1 Score on Dev Set: 0.8202
 41%|████      | 41/100 [08:29<12:02, 12.24s/it]
Epoch 41/100: Loss=1.9133, F1 Score on Dev Set: 0.8191
 42%|████▏     | 42/100 [08:42<11:46, 12.19s/it]
Epoch 42/100: Loss=1.9346, F1 Score on Dev Set: 0.8163
 43%|████▎     | 43/100 [08:53<11:28, 12.07s/it]
Epoch 43/100: Loss=1.8584, F1 Score on Dev Set: 0.8178
 44%|████▍     | 44/100 [09:05<11:13, 12.03s/it]
Epoch 44/100: Loss=1.6527, F1 Score on Dev Set: 0.8176
 45%|████▌     | 45/100 [09:17<11:04, 12.08s/it]
Epoch 45/100: Loss=1.7876, F1 Score on Dev Set: 0.8159
 46%|████▌     | 46/100 [09:30<10:53, 12.11s/it]
Epoch 46/100: Loss=1.6978, F1 Score on Dev Set: 0.8187
 47%|████▋     | 47/100 [09:42<10:43, 12.14s/it]
Epoch 47/100: Loss=1.8309, F1 Score on Dev Set: 0.8138
 48%|████▊     | 48/100 [09:54<10:32, 12.16s/it]
Epoch 48/100: Loss=1.7795, F1 Score on Dev Set: 0.8139
 49%|████▉     | 49/100 [10:06<10:22, 12.20s/it]
Epoch 49/100: Loss=1.7984, F1 Score on Dev Set: 0.7872
 50%|█████     | 50/100 [10:18<10:09, 12.19s/it]
Epoch 50/100: Loss=1.8347, F1 Score on Dev Set: 0.8148
 51%|█████     | 51/100 [10:31<09:56, 12.18s/it]
```

```
Epoch 51/100: Loss=1.6725, F1 Score on Dev Set: 0.8139
 52%|█████▏    | 52/100 [10:43<09:44, 12.18s/it]
Epoch 52/100: Loss=1.6726, F1 Score on Dev Set: 0.8160
 53%|█████▎    | 53/100 [10:55<09:31, 12.16s/it]
Epoch 53/100: Loss=1.6142, F1 Score on Dev Set: 0.8167
 54%|█████▍    | 54/100 [11:07<09:18, 12.15s/it]
Epoch 54/100: Loss=1.6339, F1 Score on Dev Set: 0.8116
 55%|█████▌    | 55/100 [11:20<09:17, 12.38s/it]
Epoch 55/100: Loss=1.8342, F1 Score on Dev Set: 0.8141
 56%|█████▌    | 56/100 [11:32<09:02, 12.33s/it]
Epoch 56/100: Loss=1.6333, F1 Score on Dev Set: 0.8155
 57%|█████▋    | 57/100 [11:44<08:47, 12.26s/it]
Epoch 57/100: Loss=1.7589, F1 Score on Dev Set: 0.7989
 58%|█████▊    | 58/100 [11:56<08:31, 12.19s/it]
Epoch 58/100: Loss=1.6356, F1 Score on Dev Set: 0.8078
 59%|█████▉    | 59/100 [12:08<08:19, 12.19s/it]
Epoch 59/100: Loss=1.5255, F1 Score on Dev Set: 0.8050
 60%|██████    | 60/100 [12:21<08:05, 12.15s/it]
Epoch 60/100: Loss=1.5923, F1 Score on Dev Set: 0.8133
 61%|██████    | 61/100 [12:33<07:55, 12.20s/it]
Epoch 61/100: Loss=1.5365, F1 Score on Dev Set: 0.8170
 62%|██████▏   | 62/100 [12:45<07:44, 12.22s/it]
Epoch 62/100: Loss=1.6531, F1 Score on Dev Set: 0.8188
 63%|██████▎   | 63/100 [12:57<07:30, 12.19s/it]
Epoch 63/100: Loss=1.5400, F1 Score on Dev Set: 0.8163
 64%|██████▍   | 64/100 [13:10<07:19, 12.21s/it]
Epoch 64/100: Loss=1.6485, F1 Score on Dev Set: 0.8135
 65%|██████▌   | 65/100 [13:22<07:08, 12.25s/it]
Epoch 65/100: Loss=1.5564, F1 Score on Dev Set: 0.8145
 66%|██████▌   | 66/100 [13:34<06:57, 12.27s/it]
Epoch 66/100: Loss=1.5489, F1 Score on Dev Set: 0.8158
 67%|██████▋   | 67/100 [13:46<06:44, 12.26s/it]
Epoch 67/100: Loss=1.6814, F1 Score on Dev Set: 0.8142
 68%|██████▊   | 68/100 [13:59<06:33, 12.30s/it]
Epoch 68/100: Loss=1.5594, F1 Score on Dev Set: 0.8133
 69%|██████▉   | 69/100 [14:11<06:22, 12.34s/it]
Epoch 69/100: Loss=1.5784, F1 Score on Dev Set: 0.8168
 70%|███████   | 70/100 [14:24<06:11, 12.37s/it]
Epoch 70/100: Loss=1.6098, F1 Score on Dev Set: 0.8149
```

```
 71%|█████████▏   | 71/100 [14:36<05:59, 12.39s/it]
Epoch 71/100: Loss=1.5479, F1 Score on Dev Set: 0.8123
 72%|█████████▎   | 72/100 [14:51<06:10, 13.24s/it]
Epoch 72/100: Loss=1.4710, F1 Score on Dev Set: 0.8145
 73%|█████████▍   | 73/100 [15:06<06:08, 13.65s/it]
Epoch 73/100: Loss=1.4930, F1 Score on Dev Set: 0.8179
 74%|█████████▌   | 74/100 [15:19<05:52, 13.54s/it]
Epoch 74/100: Loss=1.6561, F1 Score on Dev Set: 0.8127
 75%|█████████▊   | 75/100 [15:32<05:29, 13.18s/it]
Epoch 75/100: Loss=1.4880, F1 Score on Dev Set: 0.8136
 76%|█████████▉   | 76/100 [15:44<05:10, 12.95s/it]
Epoch 76/100: Loss=1.6150, F1 Score on Dev Set: 0.8135
 77%|██████████   | 77/100 [15:56<04:53, 12.76s/it]
Epoch 77/100: Loss=1.6510, F1 Score on Dev Set: 0.8147
 78%|██████████▏  | 78/100 [16:09<04:38, 12.64s/it]
Epoch 78/100: Loss=1.5340, F1 Score on Dev Set: 0.8143
 79%|██████████▎  | 79/100 [16:21<04:23, 12.55s/it]
Epoch 79/100: Loss=1.5448, F1 Score on Dev Set: 0.8156
 80%|██████████▍  | 80/100 [16:33<04:09, 12.49s/it]
Epoch 80/100: Loss=1.4458, F1 Score on Dev Set: 0.8035
 81%|██████████▌  | 81/100 [16:45<03:53, 12.30s/it]
Epoch 81/100: Loss=1.4581, F1 Score on Dev Set: 0.8145
 82%|██████████▋  | 82/100 [16:57<03:39, 12.19s/it]
Epoch 82/100: Loss=1.4806, F1 Score on Dev Set: 0.8130
 83%|██████████▊  | 83/100 [17:09<03:26, 12.15s/it]
Epoch 83/100: Loss=1.5123, F1 Score on Dev Set: 0.8131
 84%|██████████▉  | 84/100 [17:23<03:23, 12.74s/it]
Epoch 84/100: Loss=1.5501, F1 Score on Dev Set: 0.8143
 85%|███████████  | 85/100 [17:35<03:07, 12.53s/it]
Epoch 85/100: Loss=1.5351, F1 Score on Dev Set: 0.8061
 86%|███████████▏ | 86/100 [17:47<02:53, 12.41s/it]
Epoch 86/100: Loss=1.5021, F1 Score on Dev Set: 0.8050
 87%|███████████▎ | 87/100 [18:00<02:40, 12.35s/it]
Epoch 87/100: Loss=1.5700, F1 Score on Dev Set: 0.8113
 88%|███████████▍ | 88/100 [18:12<02:29, 12.49s/it]
Epoch 88/100: Loss=1.4095, F1 Score on Dev Set: 0.8126
 89%|███████████▌ | 89/100 [18:25<02:16, 12.41s/it]
Epoch 89/100: Loss=1.4402, F1 Score on Dev Set: 0.8123
 90%|███████████▋ | 90/100 [18:37<02:03, 12.35s/it]
```

```
Epoch 90/100: Loss=1.5333, F1 Score on Dev Set: 0.8014
 91%|█████████ | 91/100 [18:49<01:50, 12.29s/it]
Epoch 91/100: Loss=1.4391, F1 Score on Dev Set: 0.8114
 92%|█████████▏| 92/100 [19:01<01:38, 12.25s/it]
Epoch 92/100: Loss=1.4135, F1 Score on Dev Set: 0.8143
 93%|█████████▎| 93/100 [19:14<01:27, 12.52s/it]
Epoch 93/100: Loss=1.5547, F1 Score on Dev Set: 0.8130
 94%|█████████▍| 94/100 [19:27<01:14, 12.41s/it]
Epoch 94/100: Loss=1.5560, F1 Score on Dev Set: 0.8112
 95%|█████████▌| 95/100 [19:39<01:01, 12.35s/it]
Epoch 95/100: Loss=1.4287, F1 Score on Dev Set: 0.8096
 96%|█████████▌| 96/100 [19:51<00:49, 12.30s/it]
Epoch 96/100: Loss=1.4326, F1 Score on Dev Set: 0.8115
 97%|█████████▋| 97/100 [20:03<00:36, 12.23s/it]
Epoch 97/100: Loss=1.5018, F1 Score on Dev Set: 0.8110
 98%|█████████▊| 98/100 [20:15<00:24, 12.22s/it]
Epoch 98/100: Loss=1.4925, F1 Score on Dev Set: 0.8107
 99%|█████████▉| 99/100 [20:27<00:12, 12.03s/it]
Epoch 99/100: Loss=1.5197, F1 Score on Dev Set: 0.8113
100%|██████████| 100/100 [20:39<00:00, 12.39s/it]
Epoch 100/100: Loss=1.4929, F1 Score on Dev Set: 0.8133
```

In [18]:

```python
model = Simple_BiLSTM(embedding_dim=100, hidden_dim=256, output_dim=10, dropout=0.33).to(device)
model.to(device)
model.load_state_dict(torch.load('blstm1.pt'))
```

Out[18]:

```
<All keys matched successfully>
```

In [19]:

```python
# Evaluate on dev data
model.eval()
with torch.no_grad():
    all_preds = []
    all_true = []
    for words, tags in dev_dataloader:
        words, tags = words.to(device), tags.to(device)
        output = model(words)

        _, preds = torch.max(output, 2)

        mask = tags != tag2idx[PAD]
        preds = preds[mask].cpu().numpy()
```

```
        tags = tags[mask].cpu().numpy()

        all_preds.extend(preds)
        all_true.extend(tags)

    idx_to_tag = {v: k for k,v in tag2idx.items()}
    predicted_tags = [idx_to_tag[pred] for pred in all_preds]

    with open(dev_split, "r") as f:
        lines = f.readlines()

    output_lines = []
    pred_idx = 0
    for line in lines:
        line = line.strip()

        if not line:
            output_lines.append("\n")
            continue

        tokens = line.split()
        tokens = tokens[:2]
        tokens.append(predicted_tags[pred_idx].upper())
        pred_idx += 1

        new_line =  " ".join(tokens)
        output_lines.append(new_line + "\n")

with open("dev1.out", "w+") as f:
    f.writelines(output_lines)

print("dev1.out GENERATED")
```

```
dev1.out GENERATED
```

In [20]:

```
!apt-get install -y perl
```

```
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libperl5.34 perl-base perl-modules-5.34
Suggested packages:
  perl-doc libterm-readline-gnu-perl | libterm-readline-perl-perl libtap-harness-archive-perl
Recommended packages:
  netbase
The following packages will be upgraded:
  libperl5.34 perl perl-base perl-modules-5.34
4 upgraded, 0 newly installed, 0 to remove and 31 not upgraded.
Need to get 9,790 kB of archives.
After this operation, 8,192 B of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libperl5.34 amd64 5.34.0-3ubuntu1.3 [4,820 kB]
Get:2 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 perl amd64 5.34.0-3ubuntu1.3 [232 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 perl-base amd64 5.34.0-3ubuntu1.3 [1,762 kB]
Get:4 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 perl-modules-5.34 all 5.34.0-3ubuntu1.3 [2,976 kB]
Fetched 9,790 kB in 1s (9,420 kB/s)
(Reading database ... 121749 files and directories currently installed.)
Preparing to unpack .../libperl5.34_5.34.0-3ubuntu1.3_amd64.deb ...
Unpacking libperl5.34:amd64 (5.34.0-3ubuntu1.3) over (5.34.0-3ubuntu1.2) ...
Preparing to unpack .../perl_5.34.0-3ubuntu1.3_amd64.deb ...
Unpacking perl (5.34.0-3ubuntu1.3) over (5.34.0-3ubuntu1.2) ...
Preparing to unpack .../perl-base_5.34.0-3ubuntu1.3_amd64.deb ...
Unpacking perl-base (5.34.0-3ubuntu1.3) over (5.34.0-3ubuntu1.2) ...
Setting up perl-base (5.34.0-3ubuntu1.3) ...
(Reading database ... 121749 files and directories currently installed.)
Preparing to unpack .../perl-modules-5.34_5.34.0-3ubuntu1.3_all.deb ...
Unpacking perl-modules-5.34 (5.34.0-3ubuntu1.3) over (5.34.0-3ubuntu1.2) ...
Setting up perl-modules-5.34 (5.34.0-3ubuntu1.3) ...
Setting up libperl5.34:amd64 (5.34.0-3ubuntu1.3) ...
Setting up perl (5.34.0-3ubuntu1.3) ...
Processing triggers for man-db (2.10.2-1) ...
Processing triggers for libc-bin (2.35-0ubuntu3.4) ...
/sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_0.so.3 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_5.so.3 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbb.so.12 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbmalloc_proxy.so.2 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbmalloc.so.2 is not a symbolic link
```

```
/sbin/ldconfig.real: /usr/local/lib/libtbbbind.so.3 is not a symbolic link
```

In [21]:
```
!python '/content/eval.py' -p '/content/dev1.out' -g '/content/drive/MyDrive/nlp_hw4(dataset)/dev'
```

```
processed 51578 tokens with 5942 phrases; found: 5600 phrases; correct: 4402.
accuracy:  95.40%; precision:  78.61%; recall:  74.08%; FB1:  76.28
              LOC: precision:  88.48%; recall:  80.68%; FB1:  84.40  1675
             MISC: precision:  81.01%; recall:  74.51%; FB1:  77.63  848
              ORG: precision:  68.01%; recall:  68.16%; FB1:  68.08  1344
              PER: precision:  76.11%; recall:  71.61%; FB1:  73.79  1733
```

Generating output files dev1.out and test1.out

In [22]:
```python
# Evaluate on dev data
model.eval()
with torch.no_grad():
    all_preds = []
    all_true = []
    for words, tags in dev_dataloader:
        words, tags = words.to(device), tags.to(device)
        output = model(words)

        _, preds = torch.max(output, 2)

        mask = tags != tag2idx[PAD]
        preds = preds[mask].cpu().numpy()
        tags = tags[mask].cpu().numpy()

        all_preds.extend(preds)
        all_true.extend(tags)

    idx_to_tag = {v: k for k,v in tag2idx.items()}
    predicted_tags = [idx_to_tag[pred] for pred in all_preds]

    with open(dev_split, "r") as f:
        lines = f.readlines()

    output_lines = []
    pred_idx = 0
    for line in lines:
        line = line.strip()
```

```
        if not line:
            output_lines.append("\n")
            continue

        tokens = line.split()
        tokens = tokens[:2]
        tokens.append(predicted_tags[pred_idx].upper())
        pred_idx += 1

        new_line =  " ".join(tokens)
        output_lines.append(new_line + "\n")

with open("dev1.out", "w+") as f:
    f.writelines(output_lines)

print("dev1.out")
```

```
dev1.out
```

In [23]:

```
class CustomDatasetTest(Dataset):
    def __init__(self, x):
        self.x = x

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

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

def collate_fn_test(batch):
    padded_word_indices = pad_sequence([b for b in batch], batch_first=True, padding_value=0)
    return padded_word_indices

def convertFileToTensor_test(file_path, word2idx, tag2idx, unk_threshold=0):
    with open(file_path, 'r') as f:
        all_text = f.read()

    sentences = all_text.strip().split('\n\n')
    sentences = [s.strip().split('\n') for s in sentences]
    sentences = [[l.split() for l in s] for s in sentences]

    words = [[w[1].lower() for w in s] for s in sentences]
    word_freqs = Counter([w for sen in words for w in sen])
```

```
    words = [[w if word_freqs[w] > unk_threshold else UNK for w in sen] for sen in words]

    sentences_word_indices = [[word2idx.get(w, word2idx[UNK]) for w in sen] for sen in words]

    return sentences_word_indices
```

In [24]:
```
test_padded_word_indices = convertFileToTensor_test(test_split, word2idx, tag2idx)
test_dataset = CustomDatasetTest(test_padded_word_indices)
test_dataloader = DataLoader(test_dataset, collate_fn = collate_fn_test, batch_size = BATCH_SIZE)
```

In [25]:
```
test_dataset[1]
```

Out[25]:
```
tensor([1249, 1249])
```

In [27]:
```
# Evaluate on test data
model.eval()
with torch.no_grad():
    all_preds = []
    all_true = []
    for words in test_dataloader:
        words = words.to(device)

        output = model(words)

        _, preds = torch.max(output, 2)

        for p in preds:
            all_preds.append(p.tolist())

all_true = []

with open(test_split) as f:
    all_str = f.read()

sentences = all_str.split("\n\n")
sentences = [[i for i in sen.split("\n") if i] for sen in sentences]
all_true = [[line.split()[1] for line in sen] for sen in sentences]

idx_to_tag = {v: k for k,v in tag2idx.items()}

with open("test1.out", "w+") as f:
    for i in range(len(all_true)):
        T = all_true[i]
        P = all_preds[i][:len(T)]
```

```
        for j in range(len(T)):
            f.write(f"{j+1} {T[j]} {idx_to_tag[P[j]]}\n")
        f.write("\n")

with open('test1.out', 'r') as file:
    lines = file.readlines()
lines.pop()

with open('test1.out', 'w+') as file:
    file.writelines(lines)

print("test1.out")
```

```
test1.out
```

TASK 2 Using GloVe word embeddings

In [28]:
```
device = torch.device('cuda:2' if torch.cuda.is_available() else 'cpu')
BATCH_SIZE = 64
```

In [29]:
```
class Glove_BiLSTM(nn.Module):
    def __init__(self, vocab_size, num_classes, glove_vectors, embedding_dim, hidden_dim, output_dim, num_layers, dropo
        super(Glove_BiLSTM, self).__init__()
        self.cap_embedding = nn.Embedding(3, embedding_dim)
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding.weight.data.copy_(glove_vectors)
        self.embedding.weight.requires_grad = False
        self.blstm = nn.LSTM(embedding_dim, hidden_dim, num_layers, batch_first=True, bidirectional=True)
        self.linear = nn.Linear(2*hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout_rate)
        self.elu = nn.ELU()
        self.classifier = nn.Linear(output_dim, num_classes)

    def forward(self, sentence, x_cap):
        sentence_embedd = self.embedding(sentence)
        cap_embedd = self.cap_embedding(x_cap)
        output, _ = self.blstm(sentence_embedd + cap_embedd)
        output = self.linear(output)
        output = self.dropout(output)
        output = self.elu(output)
        output = self.classifier(output)
```

```
        return output
```

In [30]:
```
def convertFileToTensor(file_path, word2idx, tag2idx, unk_threshold=0):
    with open(file_path, 'r') as f:
        all_text = f.read()

    sentences = all_text.strip().split('\n\n')
    sentences = [s.strip().split('\n') for s in sentences]
    sentences = [[l.split() for l in s] for s in sentences]

    words = [[w[1].lower() for w in s] for s in sentences]
    word_freqs = Counter([w for sen in words for w in sen])
    words_set = [w for w, freq in word_freqs.items() if freq > unk_threshold]

    words = [[w if word_freqs[w] > unk_threshold else UNK for w in sen] for sen in words]
    ners = [[l[2] for l in s] for s in sentences]

    sentences_word_indices = [[word2idx.get(w, word2idx[UNK]) for w in sen] for sen in words]
    sentences_cap_indices = [[word2cap(w[1]) for w in s] for s in sentences]
    sentences_tag_indices = [[tag2idx[t] for t in sen] for sen in ners]

    return sentences_word_indices, sentences_cap_indices, sentences_tag_indices
```

In [31]:
```
def collate_fn(batch):
  padded_word_indices = pad_sequence([b[0] for b in batch], batch_first=True,padding_value=0)
  padded_cap_indices = pad_sequence([b[1] for b in batch], batch_first=True, padding_value=0)
  padded_tag_indices = pad_sequence([b[2] for b in batch], batch_first=True, padding_value=0)
  return padded_word_indices, padded_cap_indices, padded_tag_indices
```

In [32]:
```
class CustomDataset(Dataset):
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

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

    def __getitem__(self, idx):
        return torch.tensor(self.x[idx], dtype=torch.long), \
               torch.tensor(self.y[idx], dtype=torch.long), \
               torch.tensor(self.z[idx], dtype=torch.long)
```

In [33]:
```
import numpy as np
# Set the random seed for reproducibility
torch.manual_seed(42)

unk_threshold = 1

embedding_dim = 100
embedding_path = '/content/drive/MyDrive/nlp_hw4(dataset)/glove.6B.100d.txt'

word_vectors = {}
pad_vector = np.zeros(100)
unk_vector = np.random.randn(100)
word_vectors['<PAD>'] = pad_vector
word_vectors['<UNK>'] = unk_vector

with open(embedding_path, 'r', encoding='utf-8') as f:
    for line in f:
        word, *vector = line.split()
        vector = list(map(float, vector))
        word_vectors[word] = np.array(vector)


glove_vectors = torch.tensor(list(word_vectors.values()))
```

```
<ipython-input-33-c7899a8d616c>:23: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow. Ple
ase consider converting the list to a single numpy.ndarray with numpy.array() before converting to a tensor. (Triggered
internally at ../torch/csrc/utils/tensor_new.cpp:261.)
  glove_vectors = torch.tensor(list(word_vectors.values()))
```

In [34]:
```
from torchtext.vocab import GloVe
glove = GloVe(name="6B", dim=100)
```

```
.vector_cache/glove.6B.zip: 862MB [02:40, 5.36MB/s]
100%|██████████| 399999/400000 [00:20<00:00, 19123.01it/s]
```

In [35]:
```
pad_vector = torch.zeros(1, glove.dim)
unk_vector = torch.randn(1, glove.dim)
glove_vectors = torch.cat([pad_vector, unk_vector, glove.vectors], dim=0)
glove.itos.insert(0, '<PAD>')
glove.itos.insert(1, '<UNK>')
```

In [36]:
```
word2idx = {word: idx for idx, word in enumerate(glove.itos)}
idx2word = {idx: word for idx, word in enumerate(glove.itos)}
```

```
tag2idx = {'<PAD>': 0,'B-PER': 4, 'I-MISC': 6, 'O': 9, 'B-LOC': 1, 'I-ORG': 7,'I-LOC': 5, 'B-ORG': 3, 'B-MISC': 2, 'I-P
vocab_size, num_classes = len(word2idx), len(tag2idx)
```

In [37]:
```
def word2cap(x):
  return 1 if x == x.lower() else 2
```

In [38]:
```
train_padded_word_indices, train_padded_cap_indices, train_padded_tag_indices = convertFileToTensor(train_split, word2i
train_dataset = CustomDataset(train_padded_word_indices, train_padded_cap_indices, train_padded_tag_indices)
train_dataloader = DataLoader(train_dataset, collate_fn=collate_fn, batch_size = BATCH_SIZE)
```

In [39]:
```
dev_padded_word_indices, dev_padded_cap_indices, dev_padded_tag_indices = convertFileToTensor(dev_split, word2idx, tag2
dev_dataset = CustomDataset(dev_padded_word_indices, dev_padded_cap_indices, dev_padded_tag_indices)
dev_dataloader = DataLoader(dev_dataset, collate_fn=collate_fn, batch_size = BATCH_SIZE)
```

In [40]:
```
import torch
print(torch.cuda.device_count())
print(torch.cuda.get_device_name(0))
```

```
1
Tesla V100-SXM2-16GB
```

In [41]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [42]:
```
model = Glove_BiLSTM(vocab_size, num_classes, glove_vectors, embedding_dim=100, hidden_dim=256, output_dim=128, num_lay
model.to(device)
```

Out[42]:
```
Glove_BiLSTM(
  (cap_embedding): Embedding(3, 100)
  (embedding): Embedding(400002, 100)
  (blstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=10, bias=True)
)
```

In [43]:
```
criterion = nn.CrossEntropyLoss(ignore_index=tag2idx[PAD])
optimizer = optim.SGD(model.parameters(), lr=0.5)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.9)
```

In [44]:
```
num_epochs = 50
```

In [45]:
```
import os
LOSS = 10000000
for epoch in tqdm(range(num_epochs)):
  model.train()
  epoch_loss = 0
  for words, caps, tags in train_dataloader:
    words, caps, tags = words.to(device), caps.to(device), tags.to(device)

    optimizer.zero_grad()
    output = model(words, caps)
    output = output.view(-1, num_classes)
    loss = criterion(output, tags.view(-1))
    epoch_loss = loss / len(train_dataloader)
    loss.backward()
    optimizer.step()
  if epoch_loss < LOSS:
    LOSS = epoch_loss
    torch.save(model.state_dict(), 'blstm2.pt')
# Print epoch metrics
  if (epoch+1) % 10 == 0:
    print(f'Epoch {epoch+1}/{num_epochs}: Loss={epoch_loss:.8f}')
# scheduler.step()
```

```
 20%|██        | 10/50 [00:21<01:19,  1.99s/it]
Epoch 10/50: Loss=0.00003911
 40%|████      | 20/50 [00:41<01:00,  2.00s/it]
Epoch 20/50: Loss=0.00001906
 60%|██████    | 30/50 [00:59<00:34,  1.71s/it]
Epoch 30/50: Loss=0.00002208
 80%|████████  | 40/50 [01:17<00:18,  1.86s/it]
Epoch 40/50: Loss=0.00000485
100%|██████████| 50/50 [01:35<00:00,  1.91s/it]
Epoch 50/50: Loss=0.00000280
```

In [52]:
```
model = Glove_BiLSTM(vocab_size, num_classes, glove_vectors, embedding_dim=100, hidden_dim=256, output_dim=128, num_lay
model.to(device)
model.load_state_dict(torch.load('blstm2.pt'))
```

Out[52]:
```
<All keys matched successfully>
```

In [53]:
```
# Evaluate on dev data
model.eval()
```

```
with torch.no_grad():
    all_preds = []
    all_true = []
    for words, caps, tags in dev_dataloader:
        words, caps, tags = words.to(device), caps.to(device), tags.to(device)
        output = model(words, caps)

        _, preds = torch.max(output, 2)

        mask = tags != tag2idx[PAD]
        preds = preds[mask].cpu().numpy()
        tags = tags[mask].cpu().numpy()

        all_preds.extend(preds)
        all_true.extend(tags)

    idx_to_tag = {v: k for k,v in tag2idx.items()}
    predicted_tags = [idx_to_tag[pred] for pred in all_preds]

    with open(dev_split, "r") as f:
        lines = f.readlines()

    output_lines = []
    pred_idx = 0
    for line in lines:
        line = line.strip()

        if not line:
            output_lines.append("\n")
            continue

        tokens = line.split()
        tokens = tokens[:2]
        tokens.append(predicted_tags[pred_idx].upper())
        pred_idx += 1

        new_line =  " ".join(tokens)
        output_lines.append(new_line + "\n")

with open("dev2.out", "w+") as f:
    f.writelines(output_lines)

print("dev2.out")
```

```
dev2.out
```

In [54]:
```
!python '/content/eval.py' -p '/content/dev2.out' -g '/content/drive/MyDrive/nlp_hw4(dataset)/dev'
```

```
processed 51578 tokens with 5942 phrases; found: 6106 phrases; correct: 5459.
accuracy:  98.50%; precision:  89.40%; recall:  91.87%; FB1:  90.62
              LOC: precision:  94.56%; recall:  94.67%; FB1:  94.61  1839
             MISC: precision:  79.36%; recall:  85.47%; FB1:  82.30  993
              ORG: precision:  83.99%; recall:  86.43%; FB1:  85.19  1380
              PER: precision:  93.61%; recall:  96.25%; FB1:  94.91  1894
```

In [55]:
```
class CustomDatasetTest(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

    def __getitem__(self, idx):
        return torch.tensor(self.x[idx], dtype=torch.long), \
               torch.tensor(self.y[idx], dtype=torch.long)

def collate_fn_test(batch):
    padded_word_indices = pad_sequence([b[0] for b in batch], batch_first=True, padding_value=0)
    padded_cap_indices = pad_sequence([b[1] for b in batch], batch_first=True, padding_value=0)
    return padded_word_indices, padded_cap_indices

def convertFileToTensor_test(file_path, word2idx, tag2idx, unk_threshold=0):
    with open(file_path, 'r') as f:
        all_text = f.read()

    sentences = all_text.strip().split('\n\n')
    sentences = [s.strip().split('\n') for s in sentences]
    sentences = [[l.split() for l in s] for s in sentences]

    words = [[w[1].lower() for w in s] for s in sentences]
    word_freqs = Counter([w for sen in words for w in sen])
    words_set = [w for w, freq in word_freqs.items() if freq > unk_threshold]

    words = [[w if word_freqs[w] > unk_threshold else UNK for w in sen] for sen in words]

    sentences_word_indices = [[word2idx.get(w, word2idx[UNK]) for w in sen] for sen in words]
    sentences_cap_indices = [[word2cap(w[1]) for w in s] for s in sentences]


    return sentences_word_indices, sentences_cap_indices
```

```
test_padded_word_indices, test_padded_cap_indices = convertFileToTensor_test(test_split, word2idx, tag2idx)

test_dataset = CustomDatasetTest(test_padded_word_indices, test_padded_cap_indices)
test_dataloader = DataLoader(test_dataset, collate_fn=collate_fn_test, batch_size = BATCH_SIZE)
```

In [56]:
```
# Evaluate on dev data
model.eval()
with torch.no_grad():
    all_preds = []

    for words, caps in test_dataloader:
        words, caps = words.to(device), caps.to(device)

        output = model(words, caps)

        _, preds = torch.max(output, 2)

        for p in preds:
            all_preds.append(p.tolist())
all_true = []

with open(test_split) as f:
    all_str = f.read()

sentences = all_str.split("\n\n")
sentences = [[i for i in sen.split("\n") if i] for sen in sentences]
all_true = [[line.split()[1] for line in sen] for sen in sentences]


idx_to_tag = {v: k for k,v in tag2idx.items()}
```

In [57]:
```
with open("test2.out", "w+") as f:
    for i in range(len(all_true)):
        T = all_true[i]
        P = all_preds[i][:len(T)]

        for j in range(len(T)):
            f.write(f"{j+1} {T[j]} {idx_to_tag[P[j]]}\n")
        f.write("\n")

with open('test2.out', 'r') as file:
    lines = file.readlines()
lines.pop()
```

```
with open('test2.out', 'w+') as file:
    file.writelines(lines)

print("test2.out")
```

test2.out

Task 3 - CNN - BiLSTM

In [59]:
```
import numpy as np
np.random.seed(42)
import os
import gzip
import torch
import torch.nn.functional as F
import os
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau
```

In [60]:
```
TRAIN_FILE = train_split
DEV_FILE = dev_split
TEST_FILE = test_split
```

In [61]:
```
os.environ["CUDA_VISIBLE_DEVICES"]="0"

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

cuda

In [62]:
```
MAX_LEN = 120
```

In [63]:
```
# Assuming TRAIN_FILE, DEV_FILE, and TEST_FILE are defined file paths
# Read the training data
with open(TRAIN_FILE, "r") as f:
    train = f.readlines()

# Add an empty string at the end of the list to signify the end of the file
```

```
train += [""]

# Strip the newline characters from the end of each line
train = [i[:-1] for i in train]
# Initialize lists to store the processed training data
train_x, train_y = [], []

# Read the development data
with open(DEV_FILE, "r") as f:
    dev = f.readlines()

# Add an empty string at the end of the list to signify the end of the file
dev += [""]

# Strip the newline characters from the end of each line
dev = [i[:-1] for i in dev]
# Initialize lists to store the processed development data
dev_x, dev_y = [], []

# Read the test data
with open(TEST_FILE, "r") as f:
    test = f.readlines()

# Add an empty string at the end of the list to signify the end of the file
test += [""]

# Strip the newline characters from the end of each line
test = [i[:-1] for i in test]
# Initialize lists to store the processed test data
test_x, test_y = [], []
```

In [64]:

```
from tqdm import tqdm

# Initialize empty lists for sentences and labels
sent = []
label = []

# Initialize the variable to hold the maximum sentence length
MAX_SENT_LEN = 0

# Initialize a set to keep track of all unique tags
all_unique_tags = set()

# Loop through each line in the training data
for x in tqdm(train):
```

```
    # Split the line into its components
    k = x.split(" ")
    # If the line is empty (i.e., it's the end of a sentence)
    if len(k) == 1:
        # Update the maximum sentence length
        MAX_SENT_LEN = max(MAX_SENT_LEN, len(sent))
        # Pad the sentence and label lists to the maximum length
        while len(sent) < MAX_LEN and len(label) < MAX_LEN:
            sent.append("<pad>")
            label.append("<pad>")
        # Append the sentence and label lists to the training data
        train_x.append(sent[:MAX_LEN])
        train_y.append(label[:MAX_LEN])
        # Reset the sentence and label lists
        sent = []
        label = []
        continue
    # If the line is not empty, add the word and its label to the lists
    sent.append(k[1])
    label.append(k[2])
    # Add the label to the set of unique tags
    all_unique_tags.add(k[2])

# Print the maximum sentence length
print("MAX_SENT_LEN", MAX_SENT_LEN)
```

```
100%|██████████| 219554/219554 [00:01<00:00, 200152.93it/s]
MAX_SENT_LEN 113
```

In [86]:

```
def load_embeddings(path):
  embeddings_index = {}
  with open(path) 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

embeddings_index = load_embeddings('/content/drive/MyDrive/nlp_hw4(dataset)/glove.6B.100d.txt')
```

In [87]:
```
padding_vector = np.random.uniform(low=-1, high=1, size=(100,))
embeddings_index["<pad>"] = padding_vector
all_unique_tags.add("<pad>")
```

In [88]:
```
char2idx = {}
char2idx['<pad>'] = 0
char2idx['<unk>'] = 1
idx = 2
for word in list(embeddings_index.keys()):
  for char in word:
    x = char.lower()
    if x not in char2idx:
      char2idx[x] = idx
      idx += 1
    y = char.upper()
    if y not in char2idx:
      char2idx[y] = idx
      idx += 1
idx2char = {v:k for k, v in char2idx.items()}

word2idx = {}
idx2word = {}
for e, w in enumerate(embeddings_index.keys()):
  word2idx[w] = e
  idx2word[e] = w
tag2idx = {}
idx2tag = {}
for e, k in enumerate(list(all_unique_tags)):
  tag2idx[k] = e
  idx2tag[e] = k
```

In [89]:
```
def preProcessData(x, y):
    finalX, charsX, finalY = [], [], []
    for i in x:
        p = []
        for c in i:
            p.append(char2idx[c])
        p = p + [char2idx["<pad>"]] * 100
        p = p[:32]
        charsX.append(p)

        i = i.lower()
        try:
            finalX.append(word2idx[i])
```

```
        except:
            finalX.append(word2idx["unk"])

    for i in y:
        finalY.append(tag2idx[i])

    return finalX, charsX[:MAX_LEN], finalY

preprocessed_train_x = []
preprocessed_train_y = []
preprocessed_chars_x = []

for i in tqdm(range(len(train_x))):
    finalX, charsX, finalY = preProcessData(train_x[i], train_y[i])
    preprocessed_train_x.append(torch.tensor(finalX))
    preprocessed_chars_x.append(torch.tensor(charsX))
    preprocessed_train_y.append(torch.tensor(finalY))

len(preprocessed_train_x), len(preprocessed_train_y), len(preprocessed_chars_x)
```

```
100%|██████████| 14987/14987 [00:22<00:00, 657.96it/s]
```

Out[89]:
```
(14987, 14987, 14987)
```

In [90]:
```
from torch.utils.data import Dataset, DataLoader
class CustomDataset(Dataset):
  def __init__(self, x, y, chars):
    self.x = x
    self.y = y
    self.chars = chars
  def __len__(self):
    return len(self.x)
  def __getitem__(self, idx):
    return self.x[idx], self.chars[idx], self.y[idx]
# create custom dataset with characters
dataset = CustomDataset(preprocessed_train_x, preprocessed_train_y, preprocessed_chars_x)
# create data loader
batch_size = 128
train_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

In [91]:
```
import torch.nn.functional as F

class CharCNN(nn.Module):
    def __init__(self, char_vocab_size, char_embedding_dim, output_dim):
        super(CharCNN, self).__init__()
```

```
        self.char_embedding = nn.Embedding(char_vocab_size, char_embedding_dim)
        self.conv1d = nn.Conv1d(char_embedding_dim, output_dim, kernel_size=32)

    def forward(self, x):
        # x: [batch_size, max_seq_len, max_word_len]
        x = self.char_embedding(x)  # [batch_size, max_seq_len, max_word_len, char_embedding_dim]
        x = x.permute(0, 1, 3, 2)  # [batch_size, max_seq_len, char_embedding_dim, max_word_len]
        batch_size, max_seq_len, char_embedding_dim, max_word_len = x.shape
        x = x.view(-1, char_embedding_dim, max_word_len)  # [batch_size * max_seq_len, char_embedding_dim, max_word_len
        x = self.conv1d(x)  # [batch_size * max_seq_len, output_dim, max_word_len - kernel_size + 1]
        x = F.relu(x)
        x = F.max_pool1d(x, kernel_size=x.shape[2]).squeeze()  # [batch_size * max_seq_len, output_dim]
        x = x.view(batch_size, max_seq_len, -1)  # [batch_size, max_seq_len, output_dim]
        return x
```

In [92]:

```
class cNNBiLSTM(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, output_dim, dropout, char_embedding_dim=30):
        super(cNNBiLSTM, self).__init__()
        # Word-level embeddings
        self.word_embedding = nn.Embedding(len(embeddings_index.keys()), 100, padding_idx=0)

        # LSTM layer
        self.lstm = nn.LSTM(100 + output_dim, hidden_dim, bidirectional=True)

        # Fully connected layer
        self.fc = nn.Linear(hidden_dim*2, output_dim)

        # Dropout layer
        self.dropout = nn.Dropout(dropout)

        self.char_cnn = CharCNN(len(char2idx), 30, output_dim)

    def forward(self, sentence, chars):
        # Word-level embeddings
        word_embedded = self.word_embedding(sentence)

        # Character-level embeddings
        char_embedded = self.char_cnn(chars)

        # Concatenate word-level and character-level embeddings
        combined_embedded = torch.cat((word_embedded, char_embedded), dim=2)

        # LSTM layer
        lstm_output, _ = self.lstm(combined_embedded)
```

```
        # Fully connected layer
        fc_output = self.fc(self.dropout(lstm_output))

        # Return output
        return fc_output
```

In [93]:
```
model3 = cNNBiLSTM(embedding_dim=100, hidden_dim=256, output_dim=len(all_unique_tags), dropout=0.33, char_embedding_dim
```

In [94]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model3.parameters(), lr=0.001)
```

In [95]:
```
# Define number of epochs
num_epochs = 200
acc = 0
# Train loop
model3.train()
for epoch in range(num_epochs):
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0
    pad_predictions = 0
# Set model to train mode
    for batch_idx, (inputs, chars, targets) in enumerate(train_loader):
# Move data to device
      inputs, targets = inputs.to(device), targets.to(device)
      chars = chars.to(device)
# print(inputs.shape)
# print(chars.shape)

# Clear optimizer gradients
      optimizer.zero_grad()
# print(inputs.shape)
# Forward pass
      outputs = model3(inputs, chars)
# One-hot encode targets
# targets_one_hot = torch.nn.functional.one_hot(targets, num_classes=len(tag2idx))
      loss = criterion(outputs.view(-1,len(tag2idx.keys())),targets.view(-1))
# Backward pass
      loss.backward()
# Update optimizer parameters
      optimizer.step()
# Calculate running loss
      running_loss += loss.item()
# Calculate accuracy
```

```
      predicted_classes = torch.argmax(outputs, dim=2)
# print(predicted_classes)
      correct_predictions += torch.sum(predicted_classes == targets).item()
      total_predictions += targets.size(0) * targets.size(1)
      pad_predictions += torch.sum(targets == tag2idx["<pad>"]).item()

# scheduler.step()
# Calculate epoch loss and accuracy
    epoch_loss = running_loss / len(train_loader)
    epoch_acc = (correct_predictions - pad_predictions) / (total_predictions - pad_predictions)

    if epoch_acc > acc:
      acc = epoch_acc
      torch.save(model3.state_dict(), 'blstm3.pt')
# Print epoch metrics
    if (epoch+1) % 10 == 0:
      print(f'Epoch {epoch+1}/{num_epochs}: Loss={epoch_loss:.4f}, Acc={epoch_acc:.4f}')
```

```
Epoch 10/200: Loss=0.0189, Acc=0.9481
Epoch 20/200: Loss=0.0113, Acc=0.9682
Epoch 30/200: Loss=0.0093, Acc=0.9725
Epoch 40/200: Loss=0.0083, Acc=0.9745
Epoch 50/200: Loss=0.0078, Acc=0.9759
Epoch 60/200: Loss=0.0074, Acc=0.9762
Epoch 70/200: Loss=0.0069, Acc=0.9779
Epoch 80/200: Loss=0.0065, Acc=0.9789
Epoch 90/200: Loss=0.0064, Acc=0.9797
Epoch 100/200: Loss=0.0061, Acc=0.9803
Epoch 110/200: Loss=0.0059, Acc=0.9806
Epoch 120/200: Loss=0.0059, Acc=0.9810
Epoch 130/200: Loss=0.0057, Acc=0.9814
Epoch 140/200: Loss=0.0056, Acc=0.9819
Epoch 150/200: Loss=0.0055, Acc=0.9823
Epoch 160/200: Loss=0.0054, Acc=0.9827
Epoch 170/200: Loss=0.0053, Acc=0.9825
Epoch 180/200: Loss=0.0052, Acc=0.9830
Epoch 190/200: Loss=0.0051, Acc=0.9833
Epoch 200/200: Loss=0.0050, Acc=0.9836
```

In [96]:

```
model3 = cNNBiLSTM(embedding_dim=100, hidden_dim=256, output_dim=len(all_unique_tags), dropout=0.33, char_embedding_dim
model3.load_state_dict(torch.load('blstm3.pt'))
```

Out[96]:

```
<All keys matched successfully>
```

In [97]:

```
dev_sent = []
temp = []
for k in dev:
    if k == "":
        dev_sent.append(temp)
        temp = []
        continue
    temp.append(k)

dev_x, dev_y = [], []
sent = []
label = []

for x in tqdm(dev):
    k = x.split(" ")
    if len(k) == 1:
        while len(sent) < MAX_LEN and len(label) < MAX_LEN:
            sent.append("<pad>")
            label.append("<pad>")
        dev_x.append(sent[:MAX_LEN])
        dev_y.append(label[:MAX_LEN])
        sent = []
        label = []
        continue
    sent.append(k[1])
    label.append(k[2])
```

```
100%|██████████| 55044/55044 [00:00<00:00, 256738.72it/s]
```

In [98]:

```
preprocessed_dev_x = []
preprocessed_dev_y = []
preprocessed_chars_x = []

for i in tqdm(range(len(dev_x))):
    finalx, charsX, finaly = preProcessData(dev_x[i], dev_y[i])
    preprocessed_dev_x.append(torch.tensor(finalx))
    preprocessed_dev_y.append(torch.tensor(finaly))
    preprocessed_chars_x.append(torch.tensor(charsX))

print(len(preprocessed_dev_x), len(preprocessed_dev_y), len(preprocessed_chars_x))
```

```
100%|██████████| 3466/3466 [00:03<00:00, 941.83it/s]
3466 3466 3466
```

In [99]:
```
predictions_dev = []
correct = 0
total = 0

with open("dev3.out", "w+") as f:
    for i in range(len(preprocessed_dev_x)):
        inputs, targets = preprocessed_dev_x[i].to(device), preprocessed_dev_y[i].to(device)
        chars = preprocessed_chars_x[i].to(device)
        inputs = inputs.unsqueeze(0)
        chars = chars.unsqueeze(0)
        # print(chars.shape)
        outputs = model3(inputs, chars)
        targets_one_hot = torch.nn.functional.one_hot(targets, num_classes=10)
        predicted_classes = torch.argmax(outputs, dim=2)
        # print(predicted_classes)
        tag_count = MAX_LEN
        # print(endToken, padToken)
        for j in range(len(targets)):
            if targets[j].item() == word2idx["<pad>"]:
                tag_count = j
                break
        # print(tag_count)
        T = targets[:tag_count]
        P = predicted_classes[0][:tag_count].tolist()
        original = dev_sent[i]
        P = P + [tag2idx['O']]*10000
        P = P[:len(original)]
        for e, p in enumerate(original):
            f.write(f"{p} {idx2tag[P[e]]}\n")
        f.write(f"\n")

# Remove the last empty line from the output file
with open('dev3.out', 'r') as file:
    lines = file.readlines()
    lines.pop()

with open('dev3.out', 'w+') as file:
    file.writelines(lines)
```

In [100…
```
model3 = cNNBiLSTM(embedding_dim=100, hidden_dim=256, output_dim=len(all_unique_tags), dropout=0.33, char_embedding_dim
model3.load_state_dict(torch.load('blstm3.pt'))
```

Out[100]:
```
<All keys matched successfully>
```

In [101…

```
!python '/content/eval.py' -p '/content/dev3.out' -g '/content/drive/MyDrive/nlp_hw4(dataset)/dev'
```

```
processed 51578 tokens with 5942 phrases; found: 7136 phrases; correct: 4077.
accuracy:  93.79%; precision:  57.13%; recall:  68.61%; FB1:  62.35
              LOC: precision:  72.83%; recall:  82.14%; FB1:  77.21  2072
             MISC: precision:  61.09%; recall:  70.50%; FB1:  65.46  1064
              ORG: precision:  44.78%; recall:  61.74%; FB1:  51.91  1849
              PER: precision:  50.67%; recall:  59.17%; FB1:  54.60  2151
```

Test set evaluation

In [102…

```
test_sent = []
temp = []

# Loop over each line in the test dataset
for k in test:
    # Check if the line is empty (sentence delimiter)
    if k == "":
        test_sent.append(temp)
        temp = []
        continue
    # Add the non-empty line to the temporary list
    temp.append(k)

# List to store processed sentences
test_x = []
# Temporary storage for the current sentence
sent = []

# Loop over the test dataset
for x in tqdm(test):
    # Split the line by spaces
    k = x.split(" ")
    # Check if the line is empty (sentence delimiter)
    if len(k) == 1:
        # If the sentence is shorter than MAX_LEN, pad the sentence
        while len(sent) < MAX_LEN:
            sent.append("<pad>")
        # Add the padded sentence to the list of processed sentences
        test_x.append(sent[:MAX_LEN])
        # Reset the sentence list for the next sentence
        sent = []
        continue
```

```
    # Append the word to the current sentence list
    sent.append(k[1])
```

```
100%|██████████| 50350/50350 [00:00<00:00, 405544.43it/s]
```

In [103…

```
def preProcessData(x):
    finalX, charsX = [], []
    for i in x:
        p = []
        for c in i:
            p.append(char2idx[c])
        p = p + [char2idx["<pad>"]] * 100
        p = p[:32]
        charsX.append(p)

        i = i.lower()
        try:
            finalX.append(word2idx[i])
        except:
            finalX.append(word2idx["unk"])

    return finalX, charsX[:MAX_LEN]

preprocessed_test_x = []
preprocessed_chars_x = []

for i in tqdm(range(len(test_x))):
    finalX, charsX = preProcessData(test_x[i])
    preprocessed_test_x.append(torch.tensor(finalX))
    preprocessed_chars_x.append(torch.tensor(charsX))

len(preprocessed_test_x), len(preprocessed_chars_x)
```

```
100%|██████████| 3684/3684 [00:04<00:00, 825.22it/s]
```

Out[103]:

```
(3684, 3684)
```

In [104…

```
predictions_test = []
correct = 0
total = 0

with open("pred", "w") as f:
    for i in range(len(preprocessed_test_x)):
        inputs = preprocessed_test_x[i].to(device)
        chars = preprocessed_chars_x[i].to(device)
```

```
        inputs = inputs.unsqueeze(0)
        chars = chars.unsqueeze(0)
        outputs = model3(inputs, chars)
        predicted_classes = torch.argmax(outputs, dim=2)

        tag_count = MAX_LEN
        for j in range(len(targets)):
            if targets[j].item() == word2idx["<pad>"]:
                tag_count = j
                break
        P = predicted_classes[0][:tag_count].tolist()
        original = test_sent[i]

        P = P + [tag2idx['O']]*10000
        P = P[:len(original)]

        for e, p in enumerate(original):
            f.write(f"{p} {idx2tag[P[e]]}\n")
        f.write("\n")

with open('pred', 'r') as file:
    lines = file.readlines()
lines.pop()

with open('pred', 'w') as file:
    file.writelines(lines)
```

In [105…

```
test_sent[0]
```

Out[105]:

```
['1 SOCCER',
 '2 -',
 '3 JAPAN',
 '4 GET',
 '5 LUCKY',
 '6 WIN',
 '7 ,',
 '8 CHINA',
 '9 IN',
 '10 SURPRISE',
 '11 DEFEAT',
 '12 .']
```

In [106…

```
idx2tag
```

Out[106]:
```
{0: 'I-PER',
 1: 'I-LOC',
 2: 'O',
 3: 'I-ORG',
 4: '<pad>',
 5: 'B-PER',
 6: 'B-LOC',
 7: 'I-MISC',
 8: 'B-MISC',
 9: 'B-ORG'}
```

In [34]:
```
```

In [ ]:
```
```