Task 1

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

```
Validation Precision: 86.3407
Validation Recall: 77.8694
Validation F1-Score: 81.8866
```

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

```
Validation Precision: 77.6853
Validation Recall: 67.7408
Validation F1-Score: 72.3730
```

Task 2

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

```
Validation Precision: 90.0640
Validation Recall: 92.2922
Validation F1-Score: 91.1645
```

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

```
Validation Precision: 84.9598
Validation Recall: 88.0135
Validation F1-Score: 86.4597
```

BiLSTM with Glove Embeddings outperforms the model without. Can you provide a rationale for this?

1. **Semantic Information**: GloVe embeddings capture semantic relationships, providing the model with a better understanding of word meanings.
2. **Generalization**: Pre-trained GloVe embeddings generalize well across domains, leveraging knowledge from diverse contexts for improved performance.
3. **Reduced Dimensionality**: GloVe embeddings often have lower-dimensional representations, aiding model optimization and generalization in the face of limited data.
4. **Data Sparsity**: GloVe embeddings mitigate data sparsity issues by providing richer semantic information learned from large datasets.
5. **Training Efficiency**: Starting with pre-trained embeddings reduces the computational burden and accelerates model convergence compared to training from scratch.

Hyperparameters

Common Hyperparameters

```
embedding_dim = 100
num_lstm_layers = 1
lstm_hidden_dim = 256
lstm_dropout = 0.33
linear_output_dim = 128
vocab_size = len(word2id)
```

```
num_labels = 9
learning_rate = 0.01
batch_size = 32
```

Task 1

```
num_epochs = 60
```

Task 2

```
num_epochs = 10
```

Model Architectures

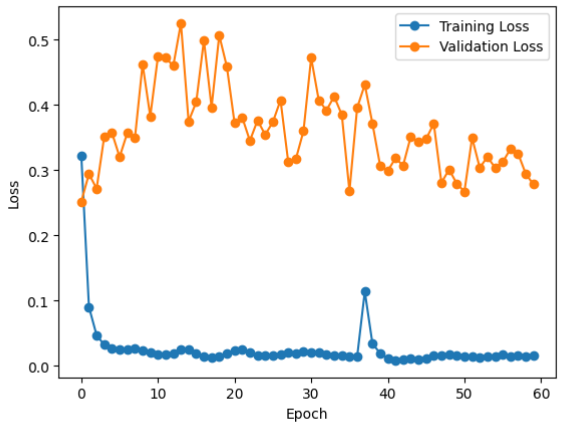
Task 1

```
BiLSTMModel(
  (embedding): Embedding(23625, 100, padding_idx=23624)
  (bilstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

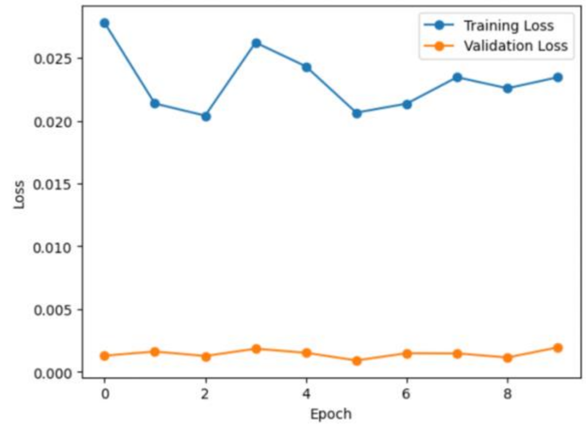
Task 2

```
GloVeBiLSTMModel(
  (embedding): Embedding(400002, 100, padding_idx=0)
  (bilstm): LSTM(104, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)

# The 4 additional features used here are as follows
def get_additional_features(token):
    additional_features = []
    if word!="<PAD>" :
        is_uppercase = 1.0 if token.isupper() else 0.0
        is_lowercase = 1.0 if token.islower() else 0.0
        is_alphanumeric = 1.0 if token.isalnum() else 0.0
        is_title = 1.0 if token.istitle() else 0.0

    else:
        return np.array([0.0, 0.0, 0.0, 0.0])
    token_features = np.array([is_uppercase, is_lowercase, is_alphanumeric,
is_title])
#     additional_features.append(token_features)

    return token_features
```

Model Loss



Task 1



Task 2

In [1]:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
/kaggle/input/csci544-da4dataset/conlleval.py
/kaggle/input/csci544-da4dataset/glove.6B.100d/glove.6B.100d.txt
```

In [2]:

```
!wget https://raw.githubusercontent.com/sighsmile/conlleval/master/conlleval.py
```

```
--2023-11-09 08:28:34--  https://raw.githubusercontent.com/sighsmile/conlleval/master/con
lleval.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.1
99.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443.
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 7502 (7.3K) [text/plain]
Saving to: 'conlleval.py'

conlleval.py        100%[===================>]   7.33K  --.-KB/s    in 0s

2023-11-09 08:28:34 (85.1 MB/s) - 'conlleval.py' saved [7502/7502]
```

In [3]:

```
import torch
print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
True
cuda:0
```

In [4]:

```
label2id = {'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4, 'B-LOC': 5, 'I-LOC':
6, 'B-MISC': 7, 'I-MISC': 8}
id2label = {0:'O', 1:'B-PER', 2:'I-PER', 3:'B-ORG' , 4:'I-ORG', 5:'B-LOC', 6:'I-LOC', 7:
'B-MISC' ,8:'I-MISC', -100:'<UNK>'}
```

In [5]:

```
from datasets import load_dataset


from tqdm.auto import tqdm
import torch

print(f"torch.__version__: {torch.__version__}")
import torch.nn as nn
from torch.nn import Parameter

import torch.nn.functional as F
from torch.optim import Adam, SGD, AdamW
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

import torch.optim as optim
from collections import defaultdict
```

```
from conlleval import evaluate
import itertools
```

```
torch.__version__: 2.0.0
```

In [6]:

```
dataset = load_dataset("conll2003")

def preprocess_example(example):
    # Rename "ner_tags" to "labels"
    example["labels"] = example["ner_tags"]

    # Remove the "pos_tags" and "chunk_tags" columns
    example.pop("pos_tags")
    example.pop("chunk_tags")
    example.pop("ner_tags")

    # Convert the text to lowercase
#     example["tokens"] = [token.lower() for token in example["tokens"]]

    return example

# Apply the preprocessing function to each split in the dataset
dataset = dataset.map(preprocess_example)

# Access the preprocessed splits
train_dataset = dataset["train"]
validation_dataset = dataset["validation"]
test_dataset = dataset["test"]

# Print the first example in the training dataset to check the changes
print(train_dataset[0])
```

```
Downloading and preparing dataset conll2003/conll2003 (download: 959.94 KiB, generated: 9
.78 MiB, post-processed: Unknown size, total: 10.72 MiB) to /root/.cache/huggingface/data
sets/conll2003/conll2003/1.0.0/63f4ebd1bcb7148b1644497336fd74643d4ce70123334431a3c053b7ee
4e96ee...
```

```
Dataset conll2003 downloaded and prepared to /root/.cache/huggingface/datasets/conll2003/
conll2003/1.0.0/63f4ebd1bcb7148b1644497336fd74643d4ce70123334431a3c053b7ee4e96ee. Subsequ
ent calls will reuse this data.
```

```
{'id': '0', 'tokens': ['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'la
mb', '.'], 'labels': [3, 0, 7, 0, 0, 0, 7, 0, 0]}
```

In [7]:

```
word_dict = defaultdict(int)
for line in train_dataset:
    for word in line['tokens']:
        word_dict[word] += 1

word_dict['<UNK>'] = 0
word_dict['<PAD>'] = 1
word2id = {}
id2word = {}
for idx, word in enumerate(word_dict.keys()):
    word2id[word] = idx
    id2word[idx] = word
```

In [8]:

```
# Define hyperparameters
embedding_dim = 100
```

```
num_lstm_layers = 1
lstm_hidden_dim = 256
lstm_dropout = 0.33
linear_output_dim = 128
vocab_size = len(word2id)
num_labels = 9

batch_size = 32
learning_rate = 0.01
batch_size = 32
num_epochs = 60
```

In [9]:

```
class BiLSTMModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, lstm_hidden_dim, num_lstm_layers, dropo
ut_prob, linear_output_dim, num_labels):
        super(BiLSTMModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=word2id['<P
AD>'])  # Use padding_idx
        self.bilstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=lstm_hidden_dim,
            num_layers=num_lstm_layers,
            batch_first=True,
            bidirectional=True
        )
        self.dropout = nn.Dropout(dropout_prob)
        self.linear = nn.Linear(lstm_hidden_dim * 2, linear_output_dim)
        self.elu = nn.ELU()
        self.classifier = nn.Linear(linear_output_dim, num_labels)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.bilstm(embedded)
        lstm_out = self.dropout(lstm_out)
        linear_out = self.elu(self.linear(lstm_out))
        logits = self.classifier(linear_out)
        return logits

# Initialize the model
model = BiLSTMModel(vocab_size, embedding_dim, lstm_hidden_dim, num_lstm_layers, lstm_dro
pout, linear_output_dim, num_labels)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=learning_rate)
# Print the model architecture
print(model)
```

```
BiLSTMModel(
  (embedding): Embedding(23625, 100, padding_idx=23624)
  (bilstm): LSTM(100, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

In [10]:

```
def collate_fn(batch):
    # Sort the batch by sequence length in decreasing order
    batch = sorted(batch, key=lambda x: len(x["tokens"]), reverse=True)

    # Get the length of the longest sequence in the batch
    max_len = len(batch[0]["tokens"])

    # Initialize lists for tokens and labels
    tokens = []
    labels = []

    for example in batch:
```

```
        # Convert tokens to numerical values using the vocabulary
        token_ids = [word2id.get(token, word2id['<UNK>']) for token in example["tokens"]
]
        tokens.append(torch.tensor(token_ids, dtype=torch.long))

        # You can remove the custom padding for labels
        labels.append(torch.tensor(example["labels"], dtype=torch.long))

    # Use pad_sequence to pad the tokens to the same length
    token_tensor = pad_sequence(tokens, batch_first=True, padding_value=word2id['<PAD>']
)
    labels_tensor = pad_sequence(labels, batch_first=True, padding_value=-100)
    # No need to pad labels since they should already be of the same length

    return {"tokens": token_tensor, "labels": labels_tensor}

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, collate_fn
=collate_fn)
validation_loader = DataLoader(validation_dataset, batch_size=batch_size, collate_fn=coll
ate_fn)
test_loader = DataLoader(test_dataset, batch_size=batch_size, collate_fn=collate_fn)
```

In [11]:

```
best_val_loss = float('inf')  # Initialize with a high value
best_model_dir = "/kaggle/working/normal"
# best_model_path = "/kaggle/working/best_model.pth"  # Define a path to save the best mo
del
best_model_path = os.path.join(best_model_dir, "best_model.pth")  # Define a path to sav
e the best model
best_train_model_path = os.path.join(best_model_dir, "best_train_model.pth")
os.makedirs(best_model_dir, exist_ok=True)
model.to(device)
history={}
train_loss, validation_loss = [], []
for epoch in range(num_epochs):
    model.train()  # Set the model in training mode
    total_loss = 0.0

    for batch in tqdm(train_loader):
        inputs = batch["tokens"].to(device)
        labels = batch["labels"].to(device)

        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)

        # Compute the loss
        loss = criterion(outputs.view(-1, num_labels), labels.view(-1))

        # Backpropagation and optimization
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    # Calculate average loss for the epoch
    avg_loss = total_loss / len(train_loader)

    print(f"Epoch [{epoch + 1}/{num_epochs}] - Loss: {avg_loss:.4f}")

    # Validation
    model.eval()  # Set the model in evaluation mode
    val_loss = 0.0

    with torch.no_grad():
        for val_batch in validation_loader:
            val_inputs = val_batch["tokens"].to(device)
            val_labels = val_batch["labels"].to(device)
```

```
            val_outputs = model(val_inputs)
            val_loss += criterion(val_outputs.view(-1, num_labels), val_labels.view(-1))
.item()

    avg_val_loss = val_loss / len(validation_loader)
    print(f"Validation Loss: {avg_val_loss:.4f}")

    history[epoch] = {'train_loss': avg_loss, 'val_loss': avg_val_loss}
    train_loss.append(avg_loss)
    validation_loss.append(avg_val_loss)
    # Check if this is the best model based on validation loss
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        # Save the model
        print(f"Saving best model with val_acc: {best_val_loss:.4f}")
        model_path = f"{best_model_path.split('.')[0]}_{epoch}.{best_model_path.split('.
')[1]}"
        torch.save(model.state_dict(), best_model_path)

torch.save(model.state_dict(), best_train_model_path)
```

```
Epoch [1/60] - Loss: 0.3227
Validation Loss: 0.2517
Saving best model with val_acc: 0.2517

Epoch [2/60] - Loss: 0.0900
Validation Loss: 0.2942

Epoch [3/60] - Loss: 0.0467
Validation Loss: 0.2715

Epoch [4/60] - Loss: 0.0319
Validation Loss: 0.3519

Epoch [5/60] - Loss: 0.0272
Validation Loss: 0.3572

Epoch [6/60] - Loss: 0.0244
Validation Loss: 0.3202

Epoch [7/60] - Loss: 0.0248
Validation Loss: 0.3569

Epoch [8/60] - Loss: 0.0264
Validation Loss: 0.3493

Epoch [9/60] - Loss: 0.0236
Validation Loss: 0.4625

Epoch [10/60] - Loss: 0.0197
Validation Loss: 0.3825

Epoch [11/60] - Loss: 0.0175
Validation Loss: 0.4738

Epoch [12/60] - Loss: 0.0168
Validation Loss: 0.4724

Epoch [13/60] - Loss: 0.0193
Validation Loss: 0.4598

Epoch [14/60] - Loss: 0.0257
Validation Loss: 0.5252

Epoch [15/60] - Loss: 0.0248
```

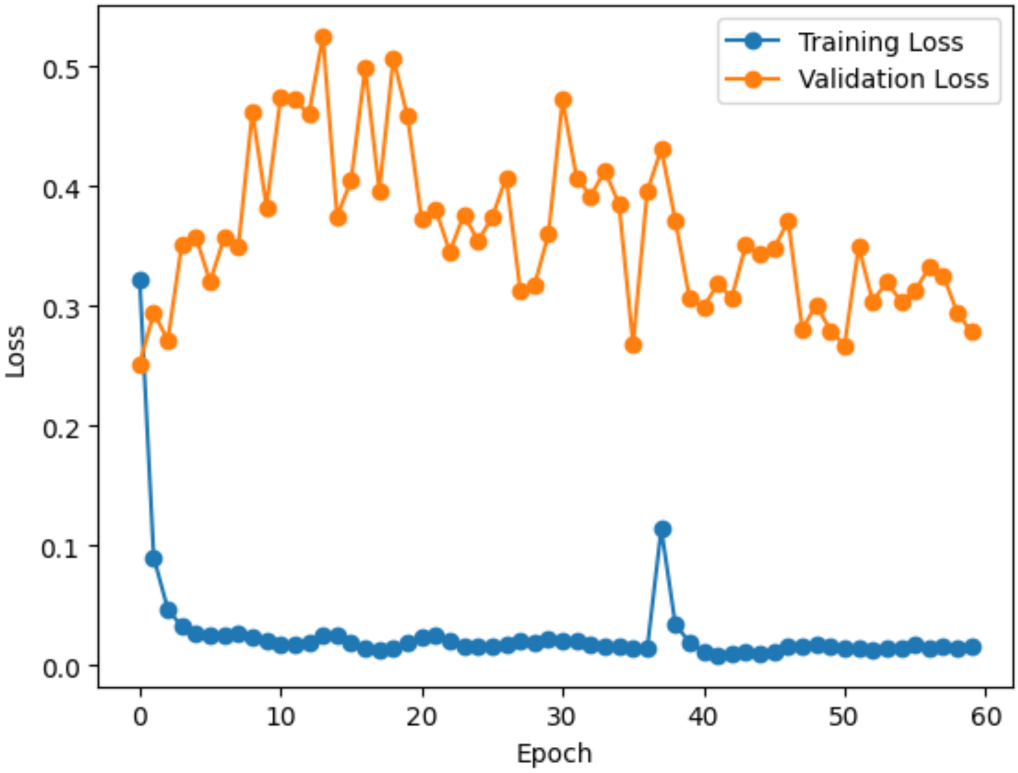
```
Validation Loss: 0.3747

Epoch [16/60] - Loss: 0.0185
Validation Loss: 0.4045

Epoch [17/60] - Loss: 0.0138
Validation Loss: 0.4993

Epoch [18/60] - Loss: 0.0129
Validation Loss: 0.3962

Epoch [19/60] - Loss: 0.0144
Validation Loss: 0.5067

Epoch [20/60] - Loss: 0.0193
Validation Loss: 0.4583

Epoch [21/60] - Loss: 0.0228
Validation Loss: 0.3720

Epoch [22/60] - Loss: 0.0257
Validation Loss: 0.3808

Epoch [23/60] - Loss: 0.0201
Validation Loss: 0.3451

Epoch [24/60] - Loss: 0.0154
Validation Loss: 0.3763

Epoch [25/60] - Loss: 0.0149
Validation Loss: 0.3543

Epoch [26/60] - Loss: 0.0155
Validation Loss: 0.3737

Epoch [27/60] - Loss: 0.0179
Validation Loss: 0.4073

Epoch [28/60] - Loss: 0.0204
Validation Loss: 0.3130

Epoch [29/60] - Loss: 0.0192
Validation Loss: 0.3169

Epoch [30/60] - Loss: 0.0218
Validation Loss: 0.3599

Epoch [31/60] - Loss: 0.0210
Validation Loss: 0.4723

Epoch [32/60] - Loss: 0.0205
Validation Loss: 0.4066

Epoch [33/60] - Loss: 0.0172
Validation Loss: 0.3919

Epoch [34/60] - Loss: 0.0154
Validation Loss: 0.4129

Epoch [35/60] - Loss: 0.0150
Validation Loss: 0.3850
```

```
Epoch [36/60] - Loss: 0.0137
Validation Loss: 0.2678

Epoch [37/60] - Loss: 0.0135
Validation Loss: 0.3953

Epoch [38/60] - Loss: 0.1139
Validation Loss: 0.4311

Epoch [39/60] - Loss: 0.0337
Validation Loss: 0.3714

Epoch [40/60] - Loss: 0.0192
Validation Loss: 0.3068

Epoch [41/60] - Loss: 0.0110
Validation Loss: 0.2987

Epoch [42/60] - Loss: 0.0081
Validation Loss: 0.3183

Epoch [43/60] - Loss: 0.0102
Validation Loss: 0.3069

Epoch [44/60] - Loss: 0.0104
Validation Loss: 0.3513

Epoch [45/60] - Loss: 0.0094
Validation Loss: 0.3433

Epoch [46/60] - Loss: 0.0117
Validation Loss: 0.3477

Epoch [47/60] - Loss: 0.0156
Validation Loss: 0.3713

Epoch [48/60] - Loss: 0.0163
Validation Loss: 0.2806

Epoch [49/60] - Loss: 0.0175
Validation Loss: 0.3009

Epoch [50/60] - Loss: 0.0156
Validation Loss: 0.2793

Epoch [51/60] - Loss: 0.0137
Validation Loss: 0.2671

Epoch [52/60] - Loss: 0.0135
Validation Loss: 0.3494

Epoch [53/60] - Loss: 0.0133
Validation Loss: 0.3037

Epoch [54/60] - Loss: 0.0141
Validation Loss: 0.3202

Epoch [55/60] - Loss: 0.0146
Validation Loss: 0.3038

Epoch [56/60] - Loss: 0.0172
Validation Loss: 0.3124
```

```
Epoch [57/60] - Loss: 0.0142
Validation Loss: 0.3323

Epoch [58/60] - Loss: 0.0152
Validation Loss: 0.3254

Epoch [59/60] - Loss: 0.0137
Validation Loss: 0.2943

Epoch [60/60] - Loss: 0.0155
Validation Loss: 0.2781
```

In [12]:

```python
import matplotlib.pyplot as plt
epochs = range(num_epochs)

# Plot training loss
plt.plot(epochs, train_loss, label="Training Loss", marker='o')

# Plot validation loss
plt.plot(epochs, validation_loss, label="Validation Loss", marker='o')

# Set labels and legend
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

# Show the plot
plt.show()
```



In [13]:

```python
def convert_predictions_to_tags(predictions, id2label):
    # Convert the prediction tensor to a list of tag sequences
    tag_sequences = []
    for prediction in predictions:
        tag_sequence = [id2label[tag_id.item()] for tag_id in prediction]
        tag_sequences.append(tag_sequence)
    return tag_sequences

def convert_labels_to_tags(labels, id2label):
    # Convert the label tensor to a list of tag sequences
```

```
    tag_sequences = []
    for label_sequence in labels:
        tag_sequence = [id2label[label_id.item()] for label_id in label_sequence if labe
l_id != -100]
        tag_sequences.append(tag_sequence)
    return tag_sequences
```

In [14]:

```
# Evaluation loop for the validation dataset
model.eval()  # Set the model in evaluation mode
all_preds = []
all_labels = []

with torch.no_grad():  # Ensure no gradient calculation during evaluation
    for batch in tqdm(validation_loader):  # Iterate over the validation data loader
        inputs = batch["tokens"].to(device) # Access the inputs from the batch dictionar
y
        labels = batch["labels"].to(device) # Access the labels from the batch dictionar
y

#         print(labels)
        outputs = model(inputs)
        max_tag_ids = torch.argmax(outputs, dim=-1)
        value_to_remove = -100

        preds = convert_predictions_to_tags(max_tag_ids, id2label)
        golds = convert_labels_to_tags(labels, id2label)

        # Remove padding from both predictions and actual labels
        for pred, gold in zip(preds, golds):
#             print(pred, gold)
            pred = [p for p, label in zip(pred, gold) if label != '<UNK>']
            gold = [label for label in gold if label != '<UNK>']
            all_preds.extend(pred)
            all_labels.extend(gold)
#         print(all_preds)

# Compute precision, recall, and F1-score using conlleval
precision, recall, f1 = evaluate(all_labels, all_preds)

# Print the evaluation metrics
print(f"Validation Precision: {precision:.4f}")
print(f"Validation Recall: {recall:.4f}")
print(f"Validation F1-Score: {f1:.4f}")
```

```
processed 51362 tokens with 5942 phrases; found: 5359 phrases; correct: 4627.
accuracy:  79.80%; (non-O)
accuracy:  96.18%; precision:  86.34%; recall:  77.87%; FB1:  81.89
              LOC: precision:  93.39%; recall:  83.83%; FB1:  88.35  1649
             MISC: precision:  89.12%; recall:  79.07%; FB1:  83.79  818
              ORG: precision:  78.74%; recall:  74.27%; FB1:  76.44  1265
              PER: precision:  83.71%; recall:  73.94%; FB1:  78.52  1627
Validation Precision: 86.3407
Validation Recall: 77.8694
Validation F1-Score: 81.8866
```

In [15]:

```
# for dirname, _, filenames in os.walk('/kaggle/working'):
#     for filename in filenames:
#         print(os.path.join(dirname, filename))
# PATH = '/kaggle/working/normal/best_path'

# model = torch.load(PATH)
```

In [16]:

```
dataset = load_dataset("conll2003")
```

In [ ]:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

In [ ]:

```
!wget https://raw.githubusercontent.com/sighsmile/conlleval/master/conlleval.py
```

In [ ]:

```
import torch
print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

In [4]:

```
label2id = {'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4, 'B-LOC': 5, 'I-LOC': 6, 'B-MISC': 7, 'I-MISC': 8}
id2label = {0:'O', 1:'B-PER', 2:'I-PER', 3:'B-ORG' , 4:'I-ORG', 5:'B-LOC', 6:'I-LOC', 7:'B-MISC' ,8:'I-MISC', -100:'<UNK>'}
```

In [5]:

```
from datasets import load_dataset


from tqdm.auto import tqdm
import torch

print(f"torch.__version__: {torch.__version__}")
import torch.nn as nn
from torch.nn import Parameter

import torch.nn.functional as F
from torch.optim import Adam, SGD, AdamW
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

import torch.optim as optim
from collections import defaultdict

from conlleval import evaluate
import itertools
```

torch.__version__: 2.0.0

In [ ]:

```
glove_file = "/kaggle/input/csci544-da4dataset/glove.6B.100d/glove.6B.100d.txt"
```

In [6]:

```
dataset = load_dataset("conll2003")

def preprocess_example(example):
    # Rename "ner_tags" to "labels"
    example["labels"] = example["ner_tags"]

    # Remove the "pos_tags" and "chunk_tags" columns
    example.pop("pos_tags")
    example.pop("chunk_tags")
    example.pop("ner_tags")
```

```
    # Convert the text to lowercase
#     example["tokens"] = [token.lower() for token in example["tokens"]]

    return example

# Apply the preprocessing function to each split in the dataset
dataset = dataset.map(preprocess_example)

# Access the preprocessed splits
train_dataset = dataset["train"]
validation_dataset = dataset["validation"]
test_dataset = dataset["test"]

# Print the first example in the training dataset to check the changes
print(train_dataset[0])
```

```
Downloading and preparing dataset conll2003/conll2003 (download: 959.94 KiB, generated: 9
.78 MiB, post-processed: Unknown size, total: 10.72 MiB) to /root/.cache/huggingface/data
sets/conll2003/conll2003/1.0.0/63f4ebd1bcb7148b1644497336fd74643d4ce70123334431a3c053b7ee
4e96ee...
```

```
Dataset conll2003 downloaded and prepared to /root/.cache/huggingface/datasets/conll2003/
conll2003/1.0.0/63f4ebd1bcb7148b1644497336fd74643d4ce70123334431a3c053b7ee4e96ee. Subsequ
ent calls will reuse this data.
```

```
{'id': '0', 'tokens': ['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'la
mb', '.'], 'labels': [3, 0, 7, 0, 0, 0, 7, 0, 0]}
```

In [7]:

```
word_dict = defaultdict(int)
for line in train_dataset:
    for word in line['tokens']:
        word_dict[word] += 1

word_dict['<UNK>'] = 0
word_dict['<PAD>'] = 1
word2id = {}
id2word = {}
for idx, word in enumerate(word_dict.keys()):
    word2id[word] = idx
    id2word[idx] = word
```

In [8]:

```
# Define hyperparameters
embedding_dim = 100
num_lstm_layers = 1
lstm_hidden_dim = 256
lstm_dropout = 0.33
linear_output_dim = 128
vocab_size = len(word2id)
num_labels = 9

learning_rate = 0.01
batch_size = 32
num_epochs = 60
```

In [9]:

```
vocab = []
embeddings = []
glove_embeddings = {}
embedding_matrix = []
```

```
with open(glove_file, 'rt') as fi:
    full_content = fi.read().strip().split('\n')
    for i in range(len(full_content)):
        i_word = full_content[i].split(' ')[0]
        i_embeddings = [float(val) for val in full_content[i].split(' ')[1:]]
        vocab.append(i_word)
        embeddings.append(i_embeddings)
        glove_embeddings[i_word] = i_embeddings  # Corrected key

embedding_matrix = torch.tensor(embeddings, dtype=torch.float32)

vocab_npa = np.array(vocab)
embs_npa = np.array(embeddings)

vocab_npa = np.insert(vocab_npa, 0, '<PAD>')
vocab_npa = np.insert(vocab_npa, 1, '<UNK>')

pad_emb_npa = np.zeros((1, embs_npa.shape[1]))  # embedding for '<pad>' token.
unk_emb_npa = np.mean(embs_npa, axis=0, keepdims=True)  # embedding for '<unk>' token.

# insert embeddings for pad and unk tokens at the top of embs_npa.
embs_npa = np.vstack((pad_emb_npa, unk_emb_npa, embs_npa))
print(embs_npa.shape)

word2id = {}
id2word = {}

# Add padding and unknown tokens first
word2id['<PAD>'] = 0.0
word2id['<UNK>'] = 1.0
id2word[0] = '<PAD>'
id2word[1] = '<UNK>'

# Then, add the words from your GloVe embeddings
for idx, word in enumerate(vocab_npa[2:]):
    word2id[word] = float(idx) + 2  # Start index from 2 to account for the two special
tokens
    id2word[float(idx) + 2] = word  # Start index from 2 to account for the two special
tokens
```

```
(400002, 100)
```

In [10]:

```
def get_additional_features(token):
    additional_features = []
    if word!="<PAD>" :
        is_uppercase = 1.0 if token.isupper() else 0.0
        is_lowercase = 1.0 if token.islower() else 0.0
        is_alphanumeric = 1.0 if token.isalnum() else 0.0
        is_title = 1.0 if token.istitle() else 0.0

    else:
        return np.array([0.0, 0.0, 0.0, 0.0])
    token_features = np.array([is_uppercase, is_lowercase, is_alphanumeric, is_title])
#     additional_features.append(token_features)

    return token_features
```

In [11]:

```
def collate_fn(batch):
    batch = sorted(batch, key=lambda x: len(x["tokens"]), reverse=True)
    max_len = len(batch[0]["tokens"])
    tokens = []
    labels = []
    og_tokens = []
    word_tokens_list =[]
#     get_add_feature = []
```

```
    for example in batch:

        token_ids = [word2id.get(token.lower(), word2id['<UNK>']) for token in example["
tokens"]]

        tokens.append(torch.tensor(token_ids, dtype=torch.long))
        labels.append(torch.tensor(example["labels"], dtype=torch.long))

    padding_value = float(word2id['<PAD>'])


    token_tensor = pad_sequence(tokens, batch_first=True, padding_value=padding_value)
    labels_tensor = pad_sequence(labels, batch_first=True, padding_value=-100)

    additional_features = []

    for example in batch:
        word_tokens = example["tokens"]
        padded_word_tokens = word_tokens + ["<PAD>"] * (token_tensor.shape[1] - len(word
_tokens))

        add_feat = [get_additional_features(token) for token in padded_word_tokens]
        additional_features.append(torch.tensor(add_feat, dtype=torch.float32))

    # Convert the list of tensors to a single tensor
    additional_features_tensor = torch.stack(additional_features)

    return {"tokens": token_tensor, "labels": labels_tensor, 'og_tokens': additional_fea
tures_tensor}
```

In [12]:

```
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, collate_fn
=collate_fn)
validation_loader = DataLoader(validation_dataset, batch_size=batch_size, collate_fn=coll
ate_fn)
test_loader = DataLoader(test_dataset, batch_size=batch_size, collate_fn=collate_fn)
```

In [13]:

```
import torch.nn as nn

embedding_dim = len(glove_embeddings['example'])  # Get the embedding dimension from the
GloVe vectors

class GloVeBiLSTMModel(nn.Module):
    def __init__(self, lstm_hidden_dim, num_lstm_layers, dropout_prob, linear_output_dim
, num_labels):
        super(GloVeBiLSTMModel, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(torch.from_numpy(embs_npa).float()
, padding_idx=int(word2id['<PAD>']), freeze=True)
        self.bilstm = nn.LSTM(
            input_size=embedding_dim+4,#+4,
            hidden_size=lstm_hidden_dim,
            num_layers=num_lstm_layers,
            batch_first=True,
            bidirectional=True
        )
        self.dropout = nn.Dropout(dropout_prob)
        self.linear = nn.Linear(lstm_hidden_dim * 2, linear_output_dim)
        self.elu = nn.ELU()
        self.classifier = nn.Linear(linear_output_dim, num_labels)

    def forward(self, x, y):
#         x = x.long()
        embedded = self.embedding(x)
        add_features = y
#         print(embedded.shape, add_features.shape)
        embed = torch.cat([embedded, add_features], dim=2)
        lstm_out, _ = self.bilstm(embed)
        lstm_out = self.dropout(lstm_out)
```

```
        linear_out = self.elu(self.linear(lstm_out))
        logits = self.classifier(linear_out)
        return logits

model = GloVeBiLSTMModel(lstm_hidden_dim, num_lstm_layers, lstm_dropout, linear_output_d
im, num_labels)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=learning_rate)

# Print the model architecture
print(model)
```

```
GloVeBiLSTMModel(
  (embedding): Embedding(400002, 100, padding_idx=0)
  (bilstm): LSTM(104, 256, batch_first=True, bidirectional=True)
  (dropout): Dropout(p=0.33, inplace=False)
  (linear): Linear(in_features=512, out_features=128, bias=True)
  (elu): ELU(alpha=1.0)
  (classifier): Linear(in_features=128, out_features=9, bias=True)
)
```

In [28]:

```
import os
num_epochs=10
best_val_loss = float('inf')  # Initialize with a high value
best_model_dir = "/kaggle/working/glove"
best_model_path = os.path.join(best_model_dir, "best_model.pth")  # Define a path to sav
e the best model
best_train_model_path = os.path.join(best_model_dir, "best_train_model.pth")

# Create the directory if it doesn't exist
os.makedirs(best_model_dir, exist_ok=True)

model.to(device)
history={}
train_loss, validation_loss = [], []
for epoch in range(num_epochs):
    model.train()  # Set the model in training mode
    total_loss = 0.0

    for batch in tqdm(train_loader):
        inputs = batch["tokens"].to(device)
        labels = batch["labels"].to(device)
        og_tokens = batch["og_tokens"].to(device)
#          print(og_tokens.shape, inputs.shape)

        optimizer.zero_grad()

        # Forward pass
#          output = model(inputs)
#          add_features = get_add_feat(inputs).to(inputs.device).squeeze(2)
#          embed = torch.cat([model.embedding(inputs), add_features], dim=2)
        outputs = model(inputs, og_tokens)

        # Compute the loss
        loss = criterion(outputs.view(-1, num_labels), labels.view(-1))

        # Backpropagation and optimization
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    # Calculate average loss for the epoch
    avg_loss = total_loss / len(train_loader)

    print(f"Epoch [{epoch + 1}/{num_epochs}] - Loss: {avg_loss:.4f}")

    # Validation
    model.eval()  # Set the model in evaluation mode
    val_loss = 0.0
```

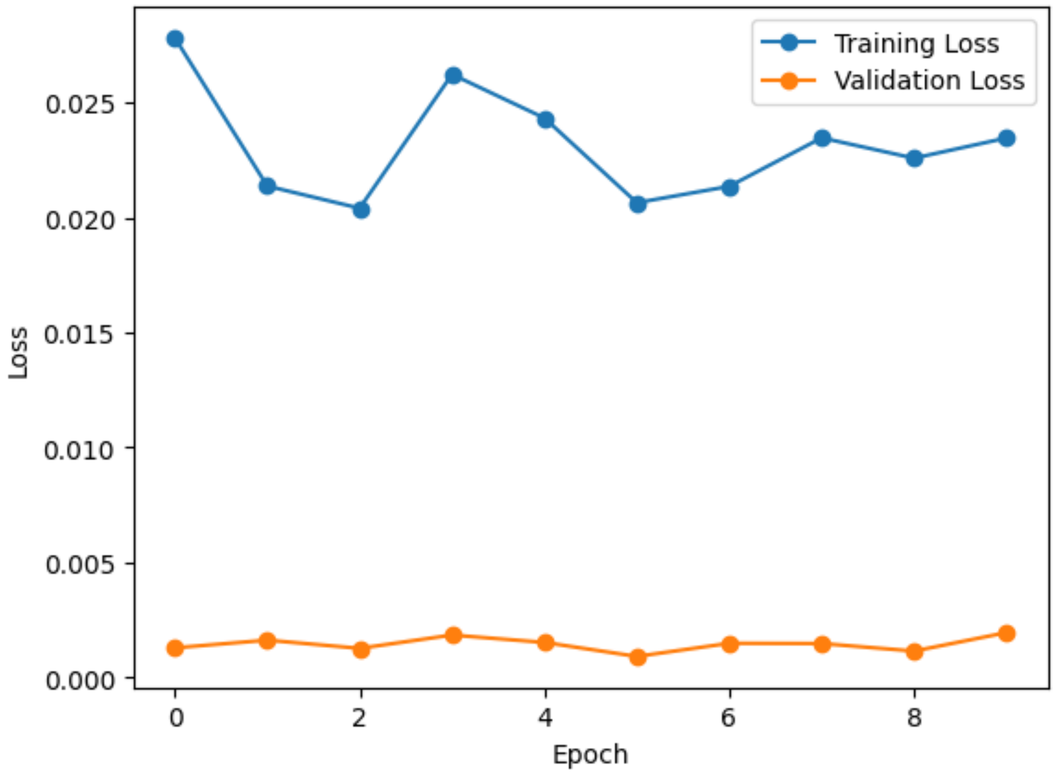
```
    all_preds = []
    all_labels = []

    with torch.no_grad():  # Ensure no gradient calculation during evaluation
        for batch in tqdm(validation_loader):  # Iterate over the validation data loader
            inputs = batch["tokens"].to(device) # Access the inputs from the batch dicti
onary
            val_labels = batch["labels"].to(device) # Access the labels from the batch d
ictionary
            og_tokens = batch["og_tokens"].to(device)
            val_outputs = model(inputs, og_tokens)
        val_loss += criterion(val_outputs.view(-1, num_labels), val_labels.view(-1)).ite
m()

    avg_val_loss = val_loss / len(validation_loader)
    print(f"Validation Loss: {avg_val_loss:.4f}")

    history[epoch] = {'train_loss': avg_loss, 'val_loss': avg_val_loss}
    train_loss.append(avg_loss)
    validation_loss.append(avg_val_loss)
    # Check if this is the best model based on validation loss
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        # Save the model
        print(f"Saving best model with val_acc: {best_val_loss:.4f}")
        model_path = f"{best_model_path.split('.')[0]}_{epoch}.{best_model_path.split('.
')[1]}"
        torch.save(model.state_dict(), model_path)

# Save the final model
torch.save(model.state_dict(), best_train_model_path)
```

```
Epoch [1/10] - Loss: 0.0278

Validation Loss: 0.0013
Saving best model with val_acc: 0.0013

Epoch [2/10] - Loss: 0.0214

Validation Loss: 0.0016

Epoch [3/10] - Loss: 0.0204

Validation Loss: 0.0013
Saving best model with val_acc: 0.0013

Epoch [4/10] - Loss: 0.0262

Validation Loss: 0.0018

Epoch [5/10] - Loss: 0.0243

Validation Loss: 0.0015

Epoch [6/10] - Loss: 0.0206

Validation Loss: 0.0009
Saving best model with val_acc: 0.0009

Epoch [7/10] - Loss: 0.0214

Validation Loss: 0.0015

Epoch [8/10] - Loss: 0.0235
```

```
Validation Loss: 0.0015

Epoch [9/10] - Loss: 0.0226

Validation Loss: 0.0011

Epoch [10/10] - Loss: 0.0235

Validation Loss: 0.0019
```

In [41]:

```
import matplotlib.pyplot as plt
epochs = range(num_epochs)

# Plot training loss
plt.plot(epochs, train_loss, label="Training Loss", marker='o')
plt.plot(epochs, validation_loss, label="Validation Loss", marker='o')
# Set labels and legend
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

# Show the plot
plt.show()
```



In [30]:

```
def convert_predictions_to_tags(predictions, id2label):
    # Convert the prediction tensor to a list of tag sequences
    tag_sequences = []
    for prediction in predictions:
        tag_sequence = [id2label[tag_id.item()] for tag_id in prediction]
        tag_sequences.append(tag_sequence)
    return tag_sequences

def convert_labels_to_tags(labels, id2label):
    # Convert the label tensor to a list of tag sequences
    tag_sequences = []
    for label_sequence in labels:
        tag_sequence = [id2label[label_id.item()] for label_id in label_sequence if labe
l_id != -100]
```

```
        tag_sequences.append(tag_sequence)
    return tag_sequences
```

In [31]:

```
# Evaluation loop for the validation dataset
model.eval()  # Set the model in evaluation mode
all_preds = []
all_labels = []

with torch.no_grad():  # Ensure no gradient calculation during evaluation
    for batch in tqdm(validation_loader):  # Iterate over the validation data loader
        inputs = batch["tokens"].to(device) # Access the inputs from the batch dictionar
y
        labels = batch["labels"].to(device) # Access the labels from the batch dictionar
y
        og_tokens = batch["og_tokens"].to(device)
#         print(labels)

#         add_features = get_add_feat(inputs).to(inputs.device).squeeze(2)
#         embed = torch.cat([model.embedding(inputs), add_features], dim=2)
        outputs = model(inputs, og_tokens)

#         outputs = model(inputs)
        max_tag_ids = torch.argmax(outputs, dim=-1)
        value_to_remove = -100

        preds = convert_predictions_to_tags(max_tag_ids, id2label)
        golds = convert_labels_to_tags(labels, id2label)

        # Remove padding from both predictions and actual labels
        for pred, gold in zip(preds, golds):
#             print(pred, gold)
            pred = [p for p, label in zip(pred, gold) if label != '<UNK>']
            gold = [label for label in gold if label != '<UNK>']
            all_preds.extend(pred)
            all_labels.extend(gold)
#         print(all_preds)

# Compute precision, recall, and F1-score using conlleval
precision, recall, f1 = evaluate(all_labels, all_preds)

# Print the evaluation metrics
print(f"Validation Precision: {precision:.4f}")
print(f"Validation Recall: {recall:.4f}")
print(f"Validation F1-Score: {f1:.4f}")
```

```
processed 51362 tokens with 5942 phrases; found: 6089 phrases; correct: 5484.
accuracy:  92.65%; (non-O)
accuracy:  98.53%; precision:  90.06%; recall:  92.29%; FB1:  91.16
              LOC: precision:  92.98%; recall:  95.86%; FB1:  94.40  1894
             MISC: precision:  81.70%; recall:  85.25%; FB1:  83.44  962
              ORG: precision:  86.17%; recall:  87.84%; FB1:  87.00  1367
              PER: precision:  94.27%; recall:  95.49%; FB1:  94.88  1866
Validation Precision: 90.0640
Validation Recall: 92.2922
Validation F1-Score: 91.1645
```

In [32]:

```
# Evaluation loop for the validation dataset
model.eval()  # Set the model in evaluation mode
all_preds = []
all_labels = []

with torch.no_grad():  # Ensure no gradient calculation during evaluation
    for batch in tqdm(test_loader):  # Iterate over the validation data loader
        inputs = batch["tokens"].to(device) # Access the inputs from the batch dictionar
y
        labels = batch["labels"].to(device) # Access the labels from the batch dictionar
y
```

```python
        og_tokens = batch["og_tokens"].to(device)
#         print(labels)
        outputs = model(inputs, og_tokens)
        max_tag_ids = torch.argmax(outputs, dim=-1)
        value_to_remove = -100

        preds = convert_predictions_to_tags(max_tag_ids, id2label)
        golds = convert_labels_to_tags(labels, id2label)

        # Remove padding from both predictions and actual labels
        for pred, gold in zip(preds, golds):
#             print(pred, gold)
            pred = [p for p, label in zip(pred, gold) if label != '<UNK>']
            gold = [label for label in gold if label != '<UNK>']
            all_preds.extend(pred)
            all_labels.extend(gold)
#         print(all_preds)

# Compute precision, recall, and F1-score using conlleval
precision, recall, f1 = evaluate(all_labels, all_preds)

# Print the evaluation metrics
print(f"Validation Precision: {precision:.4f}")
print(f"Validation Recall: {recall:.4f}")
print(f"Validation F1-Score: {f1:.4f}")
```

```
processed 46435 tokens with 5648 phrases; found: 5851 phrases; correct: 4971.
accuracy:  89.95%; (non-O)
accuracy:  97.51%; precision:  84.96%; recall:  88.01%; FB1:  86.46
              LOC: precision:  88.00%; recall:  93.23%; FB1:  90.54  1767
             MISC: precision:  68.67%; recall:  76.50%; FB1:  72.37  782
              ORG: precision:  80.85%; recall:  82.84%; FB1:  81.83  1702
              PER: precision:  93.94%; recall:  92.95%; FB1:  93.44  1600
Validation Precision: 84.9598
Validation Recall: 88.0135
Validation F1-Score: 86.4597
```