Homework4

November 10, 2023

Name: Bhavik Kethan Upadhyay

USC ID: 7750 8874 57

0.0.1 Answers to Questions:

Task 1: Hyperparameters used for creating and training BiLSTM without embeddings:

- Model Architecture
  - input_dim = torch.max(train_data1) + 1 (which gives 23589)
  - embedding_dim = 100
  - hidden_dim = 256
  - dropout_prob=0.33
  - linear_dim=128
  - out_dim = 9
- Batch size = 32
- Optimizer = AdamW
- Learning rate = Default (1e-3)
- Loss function = Cross Entropy
- epochs = 20 (with early stopping)
- Scores on Validation Set:
  - F1: 79.986
  - Recall: 77.078
  - Precision: 83.121
- Scores on Test Set:
  - F1: 70.546
  - Recall: 66.643
  - Precision: 74.935

Task 2: Hyperparameters used for creating and training BiLSTM with GloVe embeddings:

- Model Architecture
  - embedding_dim = 100
  - hidden_dim = 256

- dropout_prob=0.33
- linear_dim=128
- out_dim = 9

- Epochs = 20 (with early stopping)
- Batch size = 32
- Optimizer = AdamW
- Learning rate = Default (1e-3)
- Loss function = Cross Entropy (Note: You don't have to specify input_dim here as you are using pretrained embeddings.
- Scores on Validation Set:
  - F1: 92.117
  - Recall: 92.931
  - Precision: 91.318
- Scores on Test Set:
  - F1: 87.480
  - Recall: 88.456
  - Precision: 86.525

Q. BiLSTM with Glove Embeddings outperforms the model without. Can you provide the rationale for this? A. When using glove embeddings, our model has access to word vectors of length 100. This provides additional semantic information that can be useful for training the model. Further, there are 6B tokens in Glove embeddings, which can help cover several OOV tokens.

Task 3: Hyperparameters used for creating and training Transformer Encoder without glove embeddings:

- Model Architecture
  - emb_dim = 128
  - num_attention_heads = 8
  - max_seq_len = 128
  - ff_dim = 128
  - input_dim = torch.max(train_data1) + 1
  - out_dim = 9
  - num_layers = 1 (unspecified in pdf)
- Batch size = 32
- Optimizer = AdamW
- Learning rate = 2e-4
- Loss function = Cross Entropy
- epochs = 15
- Scores on Validation Set:

- F1: 61.434
- Recall: 50.614
- Precision: 78.138

- Scores on Test Set:
  - F1: 53.353
  - Recall: 42.683
  - Precision: 71.135

Q. What is the reason behind the poor performance of the transformer

The size of the dataset maybe one of the constraints. Further, the model maybe too complex for the given task

[1]:
```python
import datasets

from conlleval import evaluate

import torch
import torch.nn as nn
from torch import optim
from torch.utils.data import DataLoader, TensorDataset
from torch.nn.utils.rnn import pad_sequence

import itertools
from collections import Counter

import copy
import gzip
import numpy as np
import math
```

```
C:\Users\bhavi\PycharmProjects\DeepfakeDetectionUsingSWIN\venv\lib\site-
packages\tqdm\auto.py:22: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

[104]:
```python
def generate_true_and_pred(outputs, labels):
    trues = labels.reshape(-1).cpu().numpy()
    _, predicted = torch.max(outputs, 2)
    preds = predicted.reshape(-1).cpu().numpy()

    combined = list(zip(trues, preds))
    filtered = [(t, p) for (t, p) in combined if t != 9]

    trues, preds = zip(*filtered)

    return trues, preds
```

```python
def create_padded_sequences(seq_list, pad_list):
    padded_seq = []
    for sequence, pad_val in zip(seq_list, pad_list):
        padded_seq.append(pad_sequence([torch.tensor(seq) for seq in sequence],␣
 ↪batch_first=True, padding_value=pad_val))

    return padded_seq
```

1 Task 1

[3]:
```python
dataset = datasets.load_dataset('conll2003')
print(dataset)
```

```
Found cached dataset conll2003 (C:/Users/bhavi/.cache/huggingface/datasets/conll
2003/conll2003/1.0.0/9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c702a
dce98)
100%|
    | 3/3 [00:00<00:00, 177.14it/s]

DatasetDict({
    train: Dataset({
        features: ['id', 'tokens', 'pos_tags', 'chunk_tags', 'ner_tags'],
        num_rows: 14041
    })
    validation: Dataset({
        features: ['id', 'tokens', 'pos_tags', 'chunk_tags', 'ner_tags'],
        num_rows: 3250
    })
    test: Dataset({
        features: ['id', 'tokens', 'pos_tags', 'chunk_tags', 'ner_tags'],
        num_rows: 3453
    })
})
```

[4]:
```python
new_dataset = dataset.map(lambda sample: {'labels': sample['ner_tags']},␣
 ↪remove_columns=['id', 'ner_tags', 'pos_tags', 'chunk_tags'])

print(new_dataset)
```

```
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-94f685dd779131c7.arrow
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-06709f5f03722b79.arrow
```

```
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-af9b1c2d3213b131.arrow
```

```
DatasetDict({
    train: Dataset({
        features: ['tokens', 'labels'],
        num_rows: 14041
    })
    validation: Dataset({
        features: ['tokens', 'labels'],
        num_rows: 3250
    })
    test: Dataset({
        features: ['tokens', 'labels'],
        num_rows: 3453
    })
})
```

[5]:
```
word_freq = Counter(itertools.chain(*new_dataset['train']['tokens']))

word2idx = {word: idx for idx, (word, freq) in enumerate(word_freq.items(),␣
↪start=2) if freq >= 3}

word2idx['[PAD]'] = 0
word2idx['[UNK]'] = 1

label2idx = {'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, 'PAD-': 9}

print(label2idx)

print(len(label2idx), len(word2idx))
```

```
{'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, 'PAD-': 9}
10 8128
```

[6]:
```
new_dataset1 = new_dataset.map(
    lambda x: {
        'input_ids': [word2idx.get(word, word2idx['[UNK]']) for word in␣
↪x['tokens']],
            },
    remove_columns='tokens'
)

print(new_dataset1)
```

```
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
```

```
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-4f7e1dc6991de514.arrow
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-4e901fb40474f8cd.arrow
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-ecdc3baa360136b6.arrow
```

```
DatasetDict({
    train: Dataset({
        features: ['labels', 'input_ids'],
        num_rows: 14041
    })
    validation: Dataset({
        features: ['labels', 'input_ids'],
        num_rows: 3250
    })
    test: Dataset({
        features: ['labels', 'input_ids'],
        num_rows: 3453
    })
})
```

[7]:
```
pad_list = (word2idx['[PAD]'], label2idx['PAD-'])

train_seq_list = (new_dataset1['train']['input_ids'],␣
 ↪new_dataset1['train']['labels'])
train_data1, train_labels1 = create_padded_sequences(train_seq_list, pad_list)

val_seq_list = (new_dataset1['validation']['input_ids'],␣
 ↪new_dataset1['validation']['labels'])
val_data1, val_labels1 = create_padded_sequences(val_seq_list, pad_list)

test_seq_list = (new_dataset1['test']['input_ids'],␣
 ↪new_dataset1['test']['labels'])
test_data1, test_labels1 = create_padded_sequences(test_seq_list, pad_list)
```

[8]:
```
b_sz = 32

train_loader1 = DataLoader(
    TensorDataset(train_data1, train_labels1),
    batch_size=b_sz,
    shuffle=True
)

val_loader1 = DataLoader(
    TensorDataset(val_data1, val_labels1),
```

```
    batch_size=b_sz,
    shuffle=True
)

test_loader1 = DataLoader(
    TensorDataset(test_data1, test_labels1),
    batch_size=b_sz,
    shuffle=True
)
```

[9]:
```
input_dim = torch.max(train_data1) + 1
embedding_dim = 100
hidden_dim = 256
dropout_prob=0.33
linear_dim=128


out_dim = 9
print(input_dim, out_dim)
```

```
tensor(23589) 9
```

[135]:
```
import datasets

from conlleval import evaluate

import torch
import torch.nn as nn
from torch import optim
from torch.utils.data import DataLoader, TensorDataset
from torch.nn.utils.rnn import pad_sequence

import itertools
from collections import Counter

import copy
import gzip
import numpy as np
import math

from utils import generate_true_and_pred, create_padded_sequences
from models import *

def task1_test(model, data_loader, device='cpu', verbose=False):
    label2idx = {'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, 'PAD-': 9}
```

```python
    model = model.to(device)
    model.eval()

    with torch.no_grad():
        all_true, all_pred = [], []

        for inputs, labels in data_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)

            trues, preds = generate_true_and_pred(outputs, labels)
            all_true.extend(trues)
            all_pred.extend(preds)

        label_map = {label: sym for sym, label in label2idx.items()}
        all_true = [label_map[true] for true in all_true]
        all_pred = [label_map[pred] for pred in all_pred]

        res = evaluate(all_true, all_pred, verbose=verbose)
        return res


def task1_train(model, train_loader, val_loader, optimizer, criterion,␣
 ↪out_dim=9, num_epochs=10, patience=5, device='cpu', path='./task1-bilstm.
 ↪pth'):
    curr_patience = 0
    best_model = copy.deepcopy(model)
    best_f1 = 0

    model = model.to(device)
    for epoch in range(num_epochs):
        model.train()

        for inputs, labels in train_loader:
            optimizer.zero_grad()
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)

            loss = criterion(outputs.view(-1, out_dim), labels.view(-1))
            loss.backward()
            optimizer.step()

        prec, rec, f1 = task1_test(model, val_loader, device=device)
        print(f'Epoch: {epoch+1} / {num_epochs}, val_f1: {f1}, val_precision:␣
 ↪{prec}, val_recall: {rec}')
```

```
        if f1 > best_f1:
            best_f1 = f1
            curr_patience=0
            best_model = copy.deepcopy(model)
        else:
            curr_patience += 1

        if curr_patience >= patience:
            print(f'Stopping after {epoch+1} epochs')
            torch.save(best_model.state_dict(), path)
            return best_model

    torch.save(best_model.state_dict(), path)
    return best_model
```

[136]:
```
class BiLSTM1(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, linear_dim,␣
 ↪out_dim, dropout_prob):
        super(BiLSTM1, self).__init__()

        self.emb = nn.Embedding(num_embeddings=input_dim,␣
 ↪embedding_dim=embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=1,␣
 ↪batch_first=True, bidirectional=True)
        self.drop = nn.Dropout(p=dropout_prob)
        self.linear = nn.Linear(2 * hidden_dim, linear_dim)
        self.elu = nn.ELU()
        self.out = nn.Linear(linear_dim, out_dim)

    def forward(self, x):
        # print(torch.max(x))
        out = self.emb(x)
        # print(out.shape)
        out, _ = self.lstm(out)
        out = self.drop(out)
        out = self.linear(out)
        out = self.elu(out)
        out = self.out(out)

        return out
```

[137]:
```
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
cuda:0
```

[138]:
```
model1 = BiLSTM1(input_dim, embedding_dim, hidden_dim, linear_dim, out_dim,␣
  ↪dropout_prob)
model1.to(device)

criterion = nn.CrossEntropyLoss(ignore_index=label2idx['PAD-'])
optimizer = optim.AdamW(model1.parameters())
```

[139]:
```
best_bilstm = task1_train(model1, train_loader1, val_loader1, optimizer,␣
  ↪criterion, num_epochs=20, device=device)
```

```
Epoch: 1 / 20, val_f1: 57.556895545613685, val_precision: 68.17972350230414,
val_recall: 49.798047795355096
Epoch: 2 / 20, val_f1: 70.86255259467042, val_precision: 73.94804244420051,
val_recall: 68.02423426455739
Epoch: 3 / 20, val_f1: 72.56063691908905, val_precision: 80.63786008230453,
val_recall: 65.95422416694716
Epoch: 4 / 20, val_f1: 75.6775478396398, val_precision: 77.93829142143748,
val_recall: 73.54426119151802
Epoch: 5 / 20, val_f1: 77.5795732773697, val_precision: 80.74262832180561,
val_recall: 74.65499831706495
Epoch: 6 / 20, val_f1: 77.8346994535519, val_precision: 78.99480069324089,
val_recall: 76.70817906428812
Epoch: 7 / 20, val_f1: 78.17170663885992, val_precision: 80.81207330219188,
val_recall: 75.69841804106362
Epoch: 8 / 20, val_f1: 77.7972027972028, val_precision: 80.93852309930884,
val_recall: 74.89060922248402
Epoch: 9 / 20, val_f1: 77.9982891360137, val_precision: 79.31454418928323,
val_recall: 76.72500841467519
Epoch: 10 / 20, val_f1: 78.24516129032259, val_precision: 80.02815414393805,
val_recall: 76.53988556041736
Epoch: 11 / 20, val_f1: 77.33696577558318, val_precision: 78.25637491385251,
val_recall: 76.43890945809491
Epoch: 12 / 20, val_f1: 77.70415626882368, val_precision: 79.50343370311674,
val_recall: 75.9845169976439
Epoch: 13 / 20, val_f1: 78.45931958937813, val_precision: 79.1103507271172,
val_recall: 77.81891618983508
Epoch: 14 / 20, val_f1: 78.18303496737494, val_precision: 77.72787757817699,
val_recall: 78.64355435880175
Epoch: 15 / 20, val_f1: 79.16560238399319, val_precision: 80.11373427537481,
val_recall: 78.23964994951194
Epoch: 16 / 20, val_f1: 77.77777777777779, val_precision: 79.65772759350742,
val_recall: 75.9845169976439
Epoch: 17 / 20, val_f1: 79.24528301886792, val_precision: 80.41933806965864,
val_recall: 78.10501514641534
Epoch: 18 / 20, val_f1: 79.98602864128537, val_precision: 83.12159709618875,
val_recall: 77.07842477280377
Epoch: 19 / 20, val_f1: 79.66130983238293, val_precision: 81.85369318181817,
val_recall: 77.58330528441603
```

```
Epoch: 20 / 20, val_f1: 78.94464562855667, val_precision: 80.94059405940595,
val_recall: 77.04476607202963
```

[140]:
```python
prec, rec, f1 = task1_test(best_bilstm, val_loader1, verbose=True)
print('F1: ', f1, 'Recall:', rec, 'Precision:', prec)
```

```
processed 51362 tokens with 5942 phrases; found: 5510 phrases; correct: 4580.
accuracy:  79.29%; (non-O)
accuracy:  96.00%; precision:  83.12%; recall:  77.08%; FB1:  79.99
              LOC: precision:  87.67%; recall:  84.76%; FB1:  86.19  1776
             MISC: precision:  85.75%; recall:  73.75%; FB1:  79.30  793
              ORG: precision:  74.84%; recall:  70.77%; FB1:  72.75  1268
              PER: precision:  83.32%; recall:  75.68%; FB1:  79.32  1673
F1:  79.98602864128537 Recall: 77.07842477280377 Precision: 83.12159709618875
```

[141]:
```python
prec, rec, f1 = task1_test(best_bilstm, test_loader1, verbose=True)
print('F1: ', f1, 'Recall:', rec, 'Precision:', prec)
```

```
processed 46435 tokens with 5648 phrases; found: 5023 phrases; correct: 3764.
accuracy:  71.03%; (non-O)
accuracy:  93.80%; precision:  74.94%; recall:  66.64%; FB1:  70.55
              LOC: precision:  80.74%; recall:  76.92%; FB1:  78.78  1589
             MISC: precision:  72.40%; recall:  60.54%; FB1:  65.94  587
              ORG: precision:  70.87%; recall:  61.95%; FB1:  66.11  1452
              PER: precision:  73.62%; recall:  63.51%; FB1:  68.19  1395
F1:  70.5463405491519 Recall: 66.643059490085 Precision: 74.93529763089786
```

[142]:
```python
torch.save(best_bilstm.state_dict(), './task1-bilstm.pth')
```

[163]:
```python
input_dim = 23589
embedding_dim = 100
hidden_dim = 256
dropout_prob = 0.33
linear_dim = 128
out_dim = 9

model = BiLSTM1(input_dim, embedding_dim, hidden_dim, linear_dim, out_dim,␣
 ↪dropout_prob)
model.load_state_dict(torch.load('./task1-bilstm.pth'))
```

[163]: <All keys matched successfully>

[164]:
```python
model = model.to(device)
test(model, test_loader1, device=device, verbose=True)
```

```
processed 46435 tokens with 5648 phrases; found: 5023 phrases; correct: 3764.
accuracy:  71.03%; (non-O)
accuracy:  93.80%; precision:  74.94%; recall:  66.64%; FB1:  70.55
              LOC: precision:  80.74%; recall:  76.92%; FB1:  78.78  1589
```

```
MISC: precision:  72.40%; recall:  60.54%; FB1:  65.94  587
 ORG: precision:  70.87%; recall:  61.95%; FB1:  66.11  1452
 PER: precision:  73.62%; recall:  63.51%; FB1:  68.19  1395
```

[164]: (74.93529763089786, 66.643059490085, 70.5463405491519)

2 Task 2

[19]:
```
# loading the glove embeddings and creating the vocabulary and embeddings with␣
↪corresponding indices

vocab, embeddings = [], []
with gzip.open('glove.6B.100d.gz', 'rt', encoding='utf-8') as f:
    full_content = f.read().strip().split('\n')

for i in range(len(full_content)):
    word = full_content[i].split(' ')[0]
    emb = [float(val) for val in full_content[i].split(' ')[1:]]
    vocab.append(word)
    embeddings.append(emb)
```

[20]:
```
# converting to np arrays
vocab_npa = np.array(vocab)
embs_npa = np.array(embeddings)
```

[21]:
```
# adding extra tokens for padding and oov words

vocab_npa = np.insert(vocab_npa, 0, '<pad>')
vocab_npa = np.insert(vocab_npa, 1, '<unk>')

print(vocab_npa[:10])

# adding the corresponding vectors for padding and oov words
pad_emb = np.zeros((1, embs_npa.shape[1]))
unk_emb = np.mean(embs_npa, axis=0, keepdims=True)

# creating one emb_matrix by stacking the emb array on top of padding and oov␣
↪words vectors
embs_npa = np.vstack((pad_emb, unk_emb, embs_npa))
print(embs_npa.shape, vocab_npa.shape)
```

```
['<pad>' '<unk>' 'the' ',' '.' 'of' 'to' 'and' 'in' 'a']
(400002, 100) (400002,)
```

[100]:
```
np.save('./embs_npa.npy', embs_npa)
np.save('./vocab_npa.npy', vocab_npa)
```

```
vocab_npa2 = np.load('./vocab_npa.npy')
embs_npa2 = np.load('./embs_npa.npy')

print(all(vocab_npa2 == vocab_npa))
print((embs_npa2 == embs_npa).all())
```

```
True
True
```

[94]:
```
# a index mapper to obtain the index of the word in vocabulary.
# This index also matches the index of the word vector in emb_matrix

vocab2idx = {word: idx for idx, word in enumerate(vocab_npa)}
print(list(vocab2idx.keys())[:10])
```

```
['<pad>', '<unk>', 'the', ',', '.', 'of', 'to', 'and', 'in', 'a']
```

[24]:
```
# creating a variable which tells if there is a presence of an upper-case␣
↪letter in a given token

new_dataset2 = new_dataset.map(
    lambda x: {
        'isCap': [int(token[0].isupper()) for token in x['tokens']]
    }
)
```

```
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-77eba482711324ec.arrow
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-9212aab32977d975.arrow
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-5130e450f4d01990.arrow
```

[25]:
```
print(new_dataset2)
```

```
DatasetDict({
    train: Dataset({
        features: ['tokens', 'labels', 'isCap'],
        num_rows: 14041
    })
    validation: Dataset({
        features: ['tokens', 'labels', 'isCap'],
        num_rows: 3250
    })
    test: Dataset({
        features: ['tokens', 'labels', 'isCap'],
```

```
        num_rows: 3453
    })
})
```

[26]:
```
print(tuple(zip(new_dataset2['train']['tokens'][0],␣
↪new_dataset2['train']['isCap'][0])))
```

```
(('EU', 1), ('rejects', 0), ('German', 1), ('call', 0), ('to', 0), ('boycott',
0), ('British', 1), ('lamb', 0), ('.', 0))
```

[27]:
```
# using the vocab2idx dictionary to obtain the index of a word (in its␣
↪lowercase form) in the emb_matrix

new_dataset2 = new_dataset2.map(
    lambda x: {
        'input_ids': [vocab2idx.get(token.lower(), vocab2idx['<unk>']) for␣
↪token in x['tokens']]
    }
)
```

```
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-a8d2bcb8163fc566.arrow
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-9ee5d67b68aa1b11.arrow
Loading cached processed dataset at C:\Users\bhavi\.cache\huggingface\datasets\c
onll2003\conll2003\1.0.0\9a4d16a94f8674ba3466315300359b0acd891b68b6c8743ddf60b9c
702adce98\cache-9f5ef3480b4208da.arrow
```

[29]:
```
pad_list = (vocab2idx['<pad>'], label2idx['PAD-'], 0)

train_seq_list = (new_dataset2['train']['input_ids'],␣
↪new_dataset2['train']['labels'], new_dataset2['train']['isCap'])
train_data2, train_labels2, train_caps =␣
↪create_padded_sequences(train_seq_list, pad_list)

val_seq_list = (new_dataset2['validation']['input_ids'],␣
↪new_dataset2['validation']['labels'], new_dataset2['validation']['isCap'])
val_data2, val_labels2, val_caps = create_padded_sequences(val_seq_list,␣
↪pad_list)

test_seq_list = (new_dataset2['test']['input_ids'],␣
↪new_dataset2['test']['labels'], new_dataset2['test']['isCap'])
test_data2, test_labels2, test_caps = create_padded_sequences(test_seq_list,␣
↪pad_list)
```

[30]:
```
b_sz = 32

train_loader2 = DataLoader(
    TensorDataset(train_data2, train_caps, train_labels2),
    batch_size=b_sz,
    shuffle=True
)

val_loader2 = DataLoader(
    TensorDataset(val_data2, val_caps, val_labels2),
    batch_size=b_sz,
    shuffle=True
)

test_loader2 = DataLoader(
    TensorDataset(test_data2, test_caps, test_labels2),
    batch_size=b_sz,
    shuffle=True
)
```

[31]:
```
# initializing the model architecture's variables
embedding_dim = 100
hidden_dim = 256
dropout_prob=0.33
linear_dim=128

out_dim = 9
```

[173]:
```
def task2_test(model, data_loader, device='cpu', verbose=False):
    label2idx = {'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, 'PAD-': 9}

    model = model.to(device)
    model.eval()

    with torch.no_grad():
        all_true, all_pred = [], []

        for inputs, caps, labels in data_loader:
            inputs, caps, labels = inputs.to(device), caps.to(device), labels.
↪to(device)

            outputs = model(inputs, caps)

            trues = labels.view(-1).cpu().numpy()
            _, predicted = torch.max(outputs, 2)
```

```
            preds = predicted.view(-1).cpu().numpy()

            for true, pred in zip(trues, preds):
                if true != label2idx['PAD-']:
                    all_true.append(true)
                    all_pred.append(pred)

        label_map = {label: sym for sym, label in label2idx.items()}
        all_true = [label_map[true] for true in all_true]
        all_pred = [label_map[pred] for pred in all_pred]

        res = evaluate(all_true, all_pred, verbose=verbose)
        return res


def task2_train(model, train_loader, val_loader, optimizer, criterion,␣
 ↪out_dim=9, num_epochs=10, patience=5, device='cpu', path='./task2-bilstm.
 ↪pth'):
    model = model.to(device)
    curr_patience = 0
    best_model = copy.deepcopy(model)
    best_f1 = 0

    for epoch in range(num_epochs):
        model.train()

        for inputs, caps, labels in train_loader:
            optimizer.zero_grad()
            inputs, caps, labels = inputs.to(device), caps.to(device), labels.
 ↪to(device)
            outputs = model(inputs, caps)

            loss = criterion(outputs.view(-1, out_dim), labels.view(-1))
            loss.backward()
            optimizer.step()

        prec, rec, f1 = task2_test(model, val_loader, device=device)
        print(f'Epoch: {epoch+1} / {num_epochs}, val_f1: {f1}, val_precision:␣
 ↪{prec}, val_recall: {rec}')

        if f1 > best_f1:
            best_f1 = f1
            curr_patience=0
            best_model = copy.deepcopy(model)
        else:
            curr_patience += 1
```

```
        if curr_patience >= patience:
            print(f'Stopping after {epoch+1} epochs')
            torch.save(best_model.state_dict(), path)
            return best_model

    return best_model
```

[174]:
```
class BiLSTM2(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, linear_dim, out_dim,␣
↪dropout_prob, embs_npa):
        super(BiLSTM2, self).__init__()
        # loading the embedding from pretrained glove model stored in emb_matrix
        self.emb = nn.Embedding.from_pretrained(torch.from_numpy(embs_npa2).
↪float())

        self.lstm = nn.LSTM(embedding_dim+1, hidden_dim, num_layers=1,␣
↪batch_first=True, bidirectional=True)

        self.drop = nn.Dropout(p=dropout_prob)

        self.linear = nn.Linear(2 * hidden_dim, linear_dim)

        self.elu = nn.ELU()

        self.clf = nn.Linear(linear_dim, out_dim)

    def forward(self, x, caps):
        out = self.emb(x)

        caps = caps.unsqueeze(2)
        out = torch.cat([out, caps], dim=2)

        out, _ = self.lstm(out)
        out = self.drop(out)
        out = self.linear(out)
        out = self.elu(out)
        out = self.clf(out)

        return out
```

[175]:
```
from torch import optim

device = 'cuda:0'
model2 = BiLSTM2(embedding_dim, hidden_dim, linear_dim, out_dim, dropout_prob,␣
↪embs_npa)
model2.to(device)
```

```
criterion = nn.CrossEntropyLoss(ignore_index=label2idx['PAD-'])
optimizer = optim.AdamW(model2.parameters())
```

[176]:
```
best_bilstm = task2_train(model2, train_loader2, val_loader2, optimizer,␣
↪criterion, num_epochs=20, device=device)
```

```
Epoch: 1 / 20, val_f1: 84.63509380940573, val_precision: 83.15738184180607,
val_recall: 86.1662739818243
Epoch: 2 / 20, val_f1: 86.6339515927805, val_precision: 85.64380858411445,
val_recall: 87.6472568158869
Epoch: 3 / 20, val_f1: 90.07328447701532, val_precision: 89.15265413781735,
val_recall: 91.01312689330192
Epoch: 4 / 20, val_f1: 90.57482738540887, val_precision: 89.55420299391346,
val_recall: 91.61898350723662
Epoch: 5 / 20, val_f1: 90.68426769153646, val_precision: 90.04479840716775,
val_recall: 91.33288455065635
Epoch: 6 / 20, val_f1: 91.44385026737967, val_precision: 90.80650514437438,
val_recall: 92.09020531807472
Epoch: 7 / 20, val_f1: 91.11969111969111, val_precision: 90.89082384460816,
val_recall: 91.34971390104342
Epoch: 8 / 20, val_f1: 91.66805810652865, val_precision: 90.95427435387674,
val_recall: 92.39313362504208
Epoch: 9 / 20, val_f1: 91.23651452282158, val_precision: 89.99672560576293,
val_recall: 92.5109390777516
Epoch: 10 / 20, val_f1: 91.61806365605733, val_precision: 90.72607260726072,
val_recall: 92.52776842813869
Epoch: 11 / 20, val_f1: 91.19015957446808, val_precision: 90.08210180623973,
val_recall: 92.32581622349377
Epoch: 12 / 20, val_f1: 91.2747216220708, val_precision: 90.15101772816809,
val_recall: 92.42679232581622
Epoch: 13 / 20, val_f1: 91.7056074766355, val_precision: 90.94670638861304,
val_recall: 92.47728037697745
Epoch: 14 / 20, val_f1: 91.76195643101578, val_precision: 91.02500413975824,
val_recall: 92.5109390777516
Epoch: 15 / 20, val_f1: 91.466156023645, val_precision: 90.50914483440435,
val_recall: 92.4436216762033
Epoch: 16 / 20, val_f1: 92.11777462674118, val_precision: 91.31800893004795,
val_recall: 92.93167283742848
Epoch: 17 / 20, val_f1: 91.83503088996493, val_precision: 91.11994698475812,
val_recall: 92.56142712891283
Epoch: 18 / 20, val_f1: 91.60279906697767, val_precision: 90.6961398878258,
val_recall: 92.52776842813869
Epoch: 19 / 20, val_f1: 91.81696726786907, val_precision: 91.11700364600597,
val_recall: 92.52776842813869
Epoch: 20 / 20, val_f1: 91.35040453749271, val_precision: 90.55730114106169,
val_recall: 92.15752271962302
```

[181]:
```
prec, rec, f1 = task2_test(best_bilstm, val_loader2, verbose=True)
print('F1: ', f1, 'Recall:', rec, 'Precision:', prec)
```

```
processed 51362 tokens with 5942 phrases; found: 6047 phrases; correct: 5522.
accuracy:  93.00%; (non-O)
accuracy:  98.61%; precision:  91.32%; recall:  92.93%; FB1:  92.12
              LOC: precision:  93.91%; recall:  96.57%; FB1:  95.22  1889
             MISC: precision:  83.28%; recall:  86.98%; FB1:  85.09  963
              ORG: precision:  88.45%; recall:  86.80%; FB1:  87.62  1316
              PER: precision:  94.84%; recall:  96.74%; FB1:  95.78  1879
F1:  92.11777462674118 Recall: 92.93167283742848 Precision: 91.31800893004795
```

[182]:
```
prec, rec, f1 = task2_test(best_bilstm, test_loader2, verbose=True)
print('F1: ', f1, 'Recall:', rec, 'Precision:', prec)
```

```
processed 46435 tokens with 5648 phrases; found: 5774 phrases; correct: 4996.
accuracy:  89.88%; (non-O)
accuracy:  97.62%; precision:  86.53%; recall:  88.46%; FB1:  87.48
              LOC: precision:  88.20%; recall:  92.33%; FB1:  90.22  1746
             MISC: precision:  70.77%; recall:  78.63%; FB1:  74.49  780
              ORG: precision:  84.67%; recall:  83.44%; FB1:  84.05  1637
              PER: precision:  94.23%; recall:  93.88%; FB1:  94.05  1611
F1:  87.48030117317457 Recall: 88.45609065155807 Precision: 86.52580533425702
```

[183]:
```
torch.save(best_bilstm.state_dict(), './task2-bilstm.pth')
```

[184]:
```
embedding_dim = 100
hidden_dim = 256
dropout_prob = 0.33
linear_dim = 128
out_dim = 9

model = BiLSTM2(embedding_dim, hidden_dim, linear_dim, out_dim, dropout_prob,␣
 ↪embs_npa)
model = model.to(device)
model.load_state_dict(torch.load('./task2-bilstm.pth', map_location=device))
```

[184]:
```
<All keys matched successfully>
```

[ ]:
```
prec, rec, f1 = test(model, test_loader2, isCaps=True, device=device,␣
 ↪verbose=True)
```

3 Task 3

[40]:
```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
```

```
        den = torch.exp(-torch.arange(0, d_model, 2) * math.log(10000) /
↪d_model)
        pos = torch.arange(0, max_len).reshape(max_len, 1)
        pos_embedding = torch.zeros((max_len, d_model))
        pos_embedding[:, 0::2] = torch.sin(pos * den)
        pos_embedding[:, 1::2] = torch.cos(pos * den)

        pos_embedding = pos_embedding.unsqueeze(-2)
        self.register_buffer('pos_embedding', pos_embedding)

    def forward(self, x):
        return x + self.pos_embedding[:x.size(0), :]
```

[41]:
```
class TokenEmbedding(nn.Module):
    def __init__(self, input_dim, embedding_dim):
        super(TokenEmbedding, self).__init__()

        self.emb = nn.Embedding(input_dim, embedding_dim)
        self.emb_size = embedding_dim

    def forward(self, tokens):
        return self.emb(tokens.long()) * math.sqrt(self.emb_size)
```

[86]:
```
class TransformerEncoderModel(nn.Module):
    def __init__(self, input_dim, embedding_dim, num_attention_heads, ff_dim,
↪max_seq_len, out_dim):
        super(TransformerEncoderModel, self).__init__()

        self.src_token_emb = TokenEmbedding(input_dim, embedding_dim)

        self.pos_enc = PositionalEncoding(embedding_dim, max_seq_len)

        self.enc_layer = nn.TransformerEncoderLayer(d_model=embedding_dim,
↪nhead=num_attention_heads, dim_feedforward=ff_dim)
        self.encoder = nn.TransformerEncoder(self.enc_layer, num_layers=1)

        self.clf = nn.Linear(embedding_dim, out_dim)


    def forward(self, x, mask=None, src_key_padding_mask=None):
        out = self.src_token_emb(x)
        # print('Shape after passing in token embeddings:', out.size())

        out = self.pos_enc(out)
        # print('Shape after passing through source embeddings:', out.size())
```

```
        out = self.encoder(out, mask=mask,␣
↪src_key_padding_mask=src_key_padding_mask)
        # print('After passing through the transformer encoder:', out.size())

        out = self.clf(out)
        # print('Final output shape before reshaping:', out.size())

        return out
```

[179]:

```
def task3_test(model, data_loader, device='cpu', verbose=False):
    label2idx = {'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, 'PAD-': 9}
    model = model.to(device)
    model.eval()

    all_true, all_pred = [], []
    with torch.no_grad():
        for data, labels in data_loader:
            data, labels = data.transpose(0, 1).to(device), labels.transpose(0,␣
↪1).to(device)

            outputs = model(data)
            trues, preds = generate_true_and_pred(outputs, labels)

            all_true.extend(trues)
            all_pred.extend(preds)

    label_map = {label: sym for sym, label in label2idx.items()}
    all_true = [label_map[true] for true in all_true]
    all_pred = [label_map[pred] for pred in all_pred]

    res = evaluate(all_true, all_pred, verbose=verbose)

    return res


def task3_train(model, train_loader, val_loader, criterion, optimizer,␣
↪num_epochs=5, device='cpu', verbose=False):
    model = model.to(device)

    for epoch in range(num_epochs):
        model.train()
        for data, labels in train_loader:
            data, labels = data.transpose(0, 1).to(device), labels.transpose(0,␣
↪1).to(device)
            optimizer.zero_grad()
```

```
            src_mask = torch.zeros((data.size(0), data.size(0)), device=device)
            src_pad_mask = (data == word2idx['[PAD]']).transpose(0, 1)

            outputs = model(data, src_key_padding_mask=src_pad_mask)
            outputs = outputs.reshape(-1, out_dim)
            labels = labels.reshape(-1)

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        model.eval()

        all_true, all_pred = [], []
        for data, labels in val_loader:
            data, labels = data.transpose(0, 1).to(device), labels.transpose(0,␣
↪1).to(device)

            outputs = model(data)
            trues, preds = generate_true_and_pred(outputs, labels)
            all_true.extend(trues)
            all_pred.extend(preds)

        label_map = {label: sym for sym, label in label2idx.items()}
        all_true = [label_map[true] for true in all_true]
        all_pred = [label_map[pred] for pred in all_pred]

        prec, rec, f1 = evaluate(all_true, all_pred, verbose=verbose)
        print(f'Epoch: {epoch + 1} / {num_epochs}, val_f1: {f1}, val_precision:␣
↪{prec}, val_recall: {rec}')
```

[48]:

```
emb_dim = 128
num_attention_heads = 8
max_seq_len = 128
ff_dim = 128

input_dim = torch.max(train_data1) + 1
out_dim = 9

print(input_dim, out_dim)
```

```
tensor(23589) 9
```

[178]:

```
# input_dim, embedding_dim, num_attention_heads, ff_dim, max_seq_len, out_dim
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
# device='cpu'
model3 = TransformerEncoderModel(input_dim, emb_dim, num_attention_heads,␣
 ↪ff_dim, max_seq_len, out_dim)
model3 = model3.to(device)

criterion = nn.CrossEntropyLoss(ignore_index=label2idx['PAD-'])
optimizer = optim.Adam(model3.parameters(), lr=2e-4)
```

```
cuda:0
```

[180]:
```
task3_train(model3, train_loader1, val_loader1, criterion, optimizer,␣
 ↪num_epochs=15, device=device, verbose=False)
```

```
Epoch: 1 / 15, val_f1: 5.50500978924335, val_precision: 80.47138047138047,
val_recall: 2.8499880753637017
Epoch: 2 / 15, val_f1: 18.193185370428257, val_precision: 70.8603896103896,
val_recall: 10.436341900777048
Epoch: 3 / 15, val_f1: 25.908649173955293, val_precision: 69.93704092339979,
val_recall: 15.899332061068701
Epoch: 4 / 15, val_f1: 35.18366028620905, val_precision: 74.57496136012365,
val_recall: 23.022784206131455
Epoch: 5 / 15, val_f1: 41.02256155099939, val_precision: 73.36606320957348,
val_recall: 28.471064539175995
Epoch: 6 / 15, val_f1: 46.021575148024986, val_precision: 71.67761495704902,
val_recall: 33.8908135228766
Epoch: 7 / 15, val_f1: 50.610739801797635, val_precision: 71.34502923976608,
val_recall: 39.214285714285715
Epoch: 8 / 15, val_f1: 52.23038268264399, val_precision: 74.32343234323432,
val_recall: 40.26221692491061
Epoch: 9 / 15, val_f1: 55.278538137160616, val_precision: 72.72727272727273,
val_recall: 44.582338902147974
Epoch: 10 / 15, val_f1: 56.857891671520086, val_precision: 73.04526748971193,
val_recall: 46.5435041716329
Epoch: 11 / 15, val_f1: 57.51846381093058, val_precision: 75.43587756683456,
val_recall: 46.478873239436616
Epoch: 12 / 15, val_f1: 57.08034703885326, val_precision: 77.52049180327869,
val_recall: 45.170149253731346
Epoch: 13 / 15, val_f1: 59.629142940575264, val_precision: 76.69483568075117,
val_recall: 48.775827063179264
Epoch: 14 / 15, val_f1: 61.758336942399296, val_precision: 78.1227173119065,
val_recall: 51.06230603962759
Epoch: 15 / 15, val_f1: 61.61645422943221, val_precision: 78.35697181801436,
val_recall: 50.76978159684926
```

[185]:
```
prec, rec, f1 = task3_test(model3, val_loader1, device=device, verbose=True)
print('F1: ', f1, 'Recall:', rec, 'Precision:', prec)
```

```
processed 51362 tokens with 8375 phrases; found: 5425 phrases; correct: 4239.
accuracy:  45.51%; (non-O)
```

```
accuracy:  90.37%; precision:  78.14%; recall:  50.61%; FB1:  61.43
              LOC: precision:  82.30%; recall:  65.88%; FB1:  73.18  1661
             MISC: precision:  87.63%; recall:  60.16%; FB1:  71.34  865
              ORG: precision:  72.49%; recall:  41.41%; FB1:  52.71  1167
              PER: precision:  73.21%; recall:  42.31%; FB1:  53.63  1732
F1:  61.434782608695656 Recall: 50.61492537313433 Precision: 78.13824884792628
```

[186]:

```
prec, rec, f1 = task3_test(model3, test_loader1, device=device, verbose=True)
print('F1: ', f1, 'Recall:', rec, 'Precision:', prec)
```

```
processed 46435 tokens with 7893 phrases; found: 4736 phrases; correct: 3369.
accuracy:  37.87%; (non-O)
accuracy:  88.17%; precision:  71.14%; recall:  42.68%; FB1:  53.35
              LOC: precision:  81.75%; recall:  61.40%; FB1:  70.13  1436
             MISC: precision:  71.58%; recall:  51.15%; FB1:  59.67  651
              ORG: precision:  68.26%; recall:  34.05%; FB1:  45.43  1213
              PER: precision:  62.74%; recall:  34.15%; FB1:  44.23  1436
F1:  53.35339298440098 Recall: 42.68339034587609 Precision: 71.13597972972973
```

[ ]:

```
torch.save(model3.state_dict(), './task3-optimus-prime.pth')
```

[ ]:

```
```