Nithin Kodand Reddy Madadi
7692947131

HW4

Task 1:

Results on Validation and Test Data

```
Validation Data:

processed 51362 tokens with 5942 phrases; found: 5357 phrases; correct: 4468.
accuracy:  77.65%; (non-O)
accuracy:  95.89%; precision:  83.40%; recall:  75.19%; FB1:  79.09
              LOC: precision:  91.41%; recall:  79.31%; FB1:  84.93  1594
             MISC: precision:  87.94%; recall:  73.54%; FB1:  80.09  771
              ORG: precision:  73.06%; recall:  69.57%; FB1:  71.28  1277
              PER: precision:  81.63%; recall:  76.00%; FB1:  78.72  1715

Test Data:

processed 46435 tokens with 5648 phrases; found: 4877 phrases; correct: 3713.
accuracy:  70.77%; (non-O)
accuracy:  94.20%; precision:  76.13%; recall:  65.74%; FB1:  70.56
              LOC: precision:  87.30%; recall:  71.70%; FB1:  78.74  1370
             MISC: precision:  78.16%; recall:  64.25%; FB1:  70.52  577
              ORG: precision:  67.09%; recall:  61.11%; FB1:  63.96  1513
              PER: precision:  74.17%; recall:  65.00%; FB1:  69.28  1417
```

Task 2:

Results on Validation and Test Data

```
Validation Data:

processed 51362 tokens with 5942 phrases; found: 5821 phrases; correct: 4762.
accuracy:  82.35%; (non-O)
accuracy:  96.04%; precision:  81.81%; recall:  80.14%; FB1:  80.97
              LOC: precision:  89.02%; recall:  87.43%; FB1:  88.22  1804
             MISC: precision:  76.76%; recall:  77.01%; FB1:  76.88  925
              ORG: precision:  74.33%; recall:  72.56%; FB1:  73.43  1309
              PER: precision:  82.61%; recall:  79.97%; FB1:  81.27  1783

Test Data:

processed 46435 tokens with 5648 phrases; found: 5446 phrases; correct: 4001.
accuracy:  74.98%; (non-O)
accuracy:  93.90%; precision:  73.47%; recall:  70.84%; FB1:  72.13
              LOC: precision:  81.76%; recall:  80.64%; FB1:  81.20  1645
             MISC: precision:  68.35%; recall:  67.38%; FB1:  67.86  692
              ORG: precision:  66.75%; recall:  65.98%; FB1:  66.36  1642
              PER: precision:  74.10%; recall:  67.22%; FB1:  70.49  1467
```

Glove embeddings perform better because they were trained on large corpus of data which enables it to have better semantic and syntactic knowledge, and it encapsulates a variety of relationships between words which the model might not be able to get when trained from scratch on our local machines. Also I have added capitalization features in the glove embeddings, and the improved results can be attributed to the addition of these new features.

My Solution:

Hyper Parameters:

Layer hyperparam	value
Embedding dim	100
Num LSTM layers	1
LSTM hidden dim	256
LSTM Dropout	0.33
Linear output dim	128

These values are the same for both the tasks

Task 1:

Batch size = 64
Criterion = CrossEntropyLoss
Optimizer = Adam
Learning Rate = 0.01
Scheduler = ReduceLROnPlateau, mode = min, patience = 3

Task 2:

Batch size = 64
Embedding size = 100 with 3 new features = 103
Criterion = CrossEntropyLoss
Optimizer = Adam
Learning Rate = 0.01
Scheduler = ReduceLROnPlateau, mode = min, patience = 3

Architecture:

I load the dataset and separate them into Train, Validation and Test sets and I create dataloaders for each of them.

For Bidirectional LSTM Task1 I have created the model as per the design given in the assignment.

```
class BLSTM(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, lstm_hidden_dim,␣
 ↪lstm_out_neurons, num_classes):
        super().__init__()
        self.lstm_hidden_dim = lstm_hidden_dim
        self.embedding = nn.Embedding(num_embeddings, embedding_dim,␣
 ↪padding_idx=0)
        self.lstm = nn.LSTM(embedding_dim, lstm_hidden_dim, num_layers=1,␣
 ↪batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(0.33)
        self.fc1 = nn.Linear(lstm_hidden_dim*2, lstm_out_neurons)
        self.elu = nn.ELU()
        self.fc2 = nn.Linear(lstm_out_neurons,num_classes)

    def forward(self, x, lengths):

        embed = self.embedding(x)
        lstm_out, _ = self.lstm(embed)
        drop_out = self.dropout(lstm_out)
        fc1_out = self.fc1(drop_out)
        elu_out = self.elu(fc1_out)
        fc2_out = self.fc2(elu_out)

        return fc2_out
```

For initialization I give the hidden dimension size, then give the embedding dimension, which then the next layer is the LSTM layer then we give a dropout of 0.33 then give a Linear layer, followed by ELU activation and finally another Linear layer.

For Forward I have done similarly embedding, then LSTM, Dropout, Linear, ELU and Linear.

For Glove Embeddings

```
vocab_list = vocab_npa.tolist()
glove_embedding_features = []

for word, i in word2idx.items():
    is_title = 1.0 if word.istitle() else 0.0
    is_upper = 1.0 if word.isupper() else 0.0
    is_lower = 1.0 if word.islower() else 0.0

    lower = word.lower()
    if lower in vocab_list:
        idx = vocab_list.index(lower)
```



```
        embedding_l = embs_npa[idx]
    else:
        embedding_l = np.zeros((embs_npa.shape[1]))

    embedding_feature = np.concatenate([embedding_l, np.array([is_title, is_upper, is_lower])])
    glove_embedding_features.append(embedding_feature)

glove_embedding_features = np.array(glove_embedding_features)
```

I have gotten the Glove embeddings from the link given in the assignment, additionally to that I have added 3 new features to capture the capitalized words, to improve the scores.

Next for defining the LSTM with Glove Embeddings we just add the embeddings in the nn.Embedding layer, and the rest is the same as the previous Bidirectional LSTM without Glove Embedding.

```
class BLSTM_Glove(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, lstm_hidden_dim,␣
 ↪lstm_out_neurons, num_classes):
        super().__init__()
        self.lstm_hidden_dim = lstm_hidden_dim
        self.embedding = nn.Embedding(num_embeddings,␣
 ↪embedding_dim=embedding_dim, padding_idx=0).from_pretrained(torch.
 ↪from_numpy(glove_embedding_features).float(), freeze=False)
        self.lstm = nn.LSTM(embedding_dim, lstm_hidden_dim, num_layers=1,␣
 ↪batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(0.33)
        self.fc1 = nn.Linear(lstm_hidden_dim*2, lstm_out_neurons)
        self.elu = nn.ELU()
        self.fc2 = nn.Linear(lstm_out_neurons,num_classes)

    def forward(self, x, lengths):

        embed = self.embedding(x)
        lstm_out, _ = self.lstm(embed)
        drop_out = self.dropout(lstm_out)
        fc1_out = self.fc1(drop_out)
        elu_out = self.elu(fc1_out)
        fc2_out = self.fc2(elu_out)

        return fc2_out
```

For the train loop we use train and validation dataloaders, along with the num of epochs and other hyperparameters.

While training I basically get the output then compute the loss and do loss.backward and optimizer.step. Additionally I give dynamic padding to the features and the labels, due to which my training time is reduced by a lot. For the features I pad with 0s and for the labels I do with 9s.

Similarly I do for validation except that I don't train on it, it is used to calculate the best model and whenever I get a better model I save it.

Finally I return the best model.

HW4_report

November 10, 2023

[ ]:
```python
import datasets
import itertools
from collections import Counter
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import TensorDataset, DataLoader
from torch.optim import Adam
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import gzip
import shutil
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence


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


dataset = datasets.load_dataset("conll2003")

train_data = dataset['train']
valid_data = dataset['validation']
test_data = dataset['test']

word_frequency = Counter(itertools.chain(*dataset['train']['tokens']))

word_frequency = {
    word: frequency
    for word, frequency in word_frequency.items()
    if frequency >= 3
}

word2idx = {
    word: index
    for index, word in enumerate(word_frequency.keys(), start=2)
}
```

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

def convert_word_to_id(sample):
    return {
        'input_ids': [word2idx.get(token, word2idx['[UNK]']) for token in sample['tokens']],
        'labels': sample['ner_tags']

    }

dataset = dataset.map(convert_word_to_id)

for split in dataset.keys():
    columns_to_remove = set(dataset[split].column_names) - {'input_ids', 'labels'}
    dataset[split] = dataset[split].remove_columns(list(columns_to_remove))

X_train = [torch.tensor(s['input_ids']) for s in dataset['train']]
y_train = [torch.tensor(s['labels']) for s in dataset['train']]
lengths_train = [len(s['input_ids']) for s in dataset['train']]

X_valid = [torch.tensor(s['input_ids']) for s in dataset['validation']]
y_valid = [torch.tensor(s['labels']) for s in dataset['validation']]
lengths_valid = [len(s['input_ids']) for s in dataset['validation']]

X_test = [torch.tensor(s['input_ids']) for s in dataset['test']]
y_test = [torch.tensor(s['labels']) for s in dataset['test']]
lengths_test = [len(s['input_ids']) for s in dataset['test']]

X_train_padded = pad_sequence(X_train, batch_first=True, padding_value=word2idx['[PAD]'])
y_train_padded = pad_sequence(y_train, batch_first=True, padding_value=9)

X_valid_padded = pad_sequence(X_valid, batch_first=True, padding_value=word2idx['[PAD]'])
y_valid_padded = pad_sequence(y_valid, batch_first=True, padding_value=9)

X_test_padded = pad_sequence(X_test, batch_first=True, padding_value=word2idx['[PAD]'])
y_test_padded = pad_sequence(y_test, batch_first=True, padding_value=9)

lengths_train = torch.tensor(lengths_train)
lengths_valid = torch.tensor(lengths_valid)
lengths_test = torch.tensor(lengths_test)

train_dataset = TensorDataset(X_train_padded, y_train_padded, lengths_train)
```

```
valid_dataset = TensorDataset(X_valid_padded, y_valid_padded, lengths_valid)
test_dataset = TensorDataset(X_test_padded, y_test_padded, lengths_test)

train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)
valid_dataloader = DataLoader(valid_dataset, batch_size=64, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

[ ]:
```
len(word2idx)
```

[ ]: 8128

[ ]:
```
class BLSTM(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, lstm_hidden_dim,␣
 ↪lstm_out_neurons, num_classes):
        super().__init__()
        self.lstm_hidden_dim = lstm_hidden_dim
        self.embedding = nn.Embedding(num_embeddings, embedding_dim,␣
 ↪padding_idx=0)
        self.lstm = nn.LSTM(embedding_dim, lstm_hidden_dim, num_layers=1,␣
 ↪batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(0.33)
        self.fc1 = nn.Linear(lstm_hidden_dim*2, lstm_out_neurons)
        self.elu = nn.ELU()
        self.fc2 = nn.Linear(lstm_out_neurons,num_classes)

    def forward(self, x, lengths):

        embed = self.embedding(x)
        lstm_out, _ = self.lstm(embed)
        drop_out = self.dropout(lstm_out)
        fc1_out = self.fc1(drop_out)
        elu_out = self.elu(fc1_out)
        fc2_out = self.fc2(elu_out)

        return fc2_out

model1 = BLSTM(len(word2idx), 100, 256, 128, 9)
criterion1 = nn.CrossEntropyLoss(ignore_index=9)
optimizer1 = torch.optim.Adam(model1.parameters(), lr=0.01)
scheduler1 = torch.optim.lr_scheduler.
 ↪ReduceLROnPlateau(optimizer1,mode='min',patience=3)
```

[ ]:
```
def train_model(model, train_dataloader, dev_dataloader, dev_len, num_epochs,␣
 ↪criterion, optimizer, scheduler, saved_model):

    min_loss = np.Inf
    for epoch in range(num_epochs):
```

```python
        model.train()
        for X, y, length in train_dataloader:
            optimizer.zero_grad()

            pack_seq = pack_padded_sequence(X, length, batch_first=True, enforce_sorted=False)
            X, _ = pad_packed_sequence(pack_seq, batch_first=True)

            output = model(X,length)

            y_packed = pack_padded_sequence(y, length, batch_first=True, enforce_sorted=False)
            y, _ = pad_packed_sequence(y_packed, batch_first=True)
            padding_mask = (y == 0) & (torch.arange(y.size(1))[None, :] >= length[:, None])
            y[padding_mask] = 9

            loss = criterion(torch.permute(output,(0,2,1)), (y.type(torch.LongTensor)))
            loss.backward()
            optimizer.step()

        model.eval()
        dev_loss = 0
        with torch.no_grad():
            for X, y, length in dev_dataloader:

                pack_seq = pack_padded_sequence(X, length, batch_first=True, enforce_sorted=False)
                X, _ = pad_packed_sequence(pack_seq, batch_first=True)

                output = model(X, length)

                y_packed = pack_padded_sequence(y, length, batch_first=True,enforce_sorted=False)
                y, _ = pad_packed_sequence(y_packed, batch_first=True)
                padding_mask = (y == 0) & (torch.arange(y.size(1))[None, :] >= length[:, None])
                y[padding_mask] = 9

                dev_loss = criterion(torch.permute(output,(0,2,1)), (y.type(torch.LongTensor)))
                dev_loss += loss.item()*torch.sum(length)

        scheduler.step(dev_loss)
        dev_loss /= dev_len
```

```
        if dev_loss <= min_loss:
            torch.save(model.state_dict(), saved_model)
            min_loss = dev_loss

    model.load_state_dict(torch.load(saved_model))
    return model
```

[ ]:
```
model1 = train_model(model1, train_dataloader, valid_dataloader,␣
 ↪sum(lengths_valid), 30, criterion1, optimizer1, scheduler1, 'model1.pt')
```

[ ]:
```
with gzip.open('glove.6B.100d.gz', 'rb') as f_in:
    with open('glove.6B.100d.txt', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)



vocab_glove,embeddings_glove = [],[]
with open('glove.6B.100d.txt','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_glove.append(i_word)
    embeddings_glove.append(i_embeddings)

vocab_npa = np.array(vocab_glove)
embs_npa = np.array(embeddings_glove)

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]))
unk_emb_npa = np.mean(embs_npa,axis=0,keepdims=True)

embs_npa = np.vstack((pad_emb_npa,unk_emb_npa,embs_npa))
```

[ ]:
```
vocab_list = vocab_npa.tolist()
glove_embedding_features = []

for word, i in word2idx.items():
    is_title = 1.0 if word.istitle() else 0.0
    is_upper = 1.0 if word.isupper() else 0.0
    is_lower = 1.0 if word.islower() else 0.0

    lower = word.lower()
    if lower in vocab_list:
        idx = vocab_list.index(lower)
```

```
        embedding_l = embs_npa[idx]
    else:
        embedding_l = np.zeros((embs_npa.shape[1]))

    embedding_feature = np.concatenate([embedding_l, np.array([is_title,␣
↪is_upper, is_lower])])
    glove_embedding_features.append(embedding_feature)

glove_embedding_features = np.array(glove_embedding_features)
```

[ ]:
```
class BLSTM_Glove(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, lstm_hidden_dim,␣
↪lstm_out_neurons, num_classes):
        super().__init__()
        self.lstm_hidden_dim = lstm_hidden_dim
        self.embedding = nn.Embedding(num_embeddings,␣
↪embedding_dim=embedding_dim, padding_idx=0).from_pretrained(torch.
↪from_numpy(glove_embedding_features).float(), freeze=False)
        self.lstm = nn.LSTM(embedding_dim, lstm_hidden_dim, num_layers=1,␣
↪batch_first=True, bidirectional=True)
        self.dropout = nn.Dropout(0.33)
        self.fc1 = nn.Linear(lstm_hidden_dim*2, lstm_out_neurons)
        self.elu = nn.ELU()
        self.fc2 = nn.Linear(lstm_out_neurons,num_classes)

    def forward(self, x, lengths):

        embed = self.embedding(x)
        lstm_out, _ = self.lstm(embed)
        drop_out = self.dropout(lstm_out)
        fc1_out = self.fc1(drop_out)
        elu_out = self.elu(fc1_out)
        fc2_out = self.fc2(elu_out)

        return fc2_out

model2 = BLSTM_Glove(len(word2idx), 103, 256, 128, 9)
criterion2 = nn.CrossEntropyLoss(ignore_index=9)
optimizer2 = torch.optim.Adam(model2.parameters(), lr=0.01)
scheduler2 = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer2, mode='min',␣
↪patience=3)
```

[ ]:
```
model2 = train_model(model2, train_dataloader, valid_dataloader,␣
↪sum(lengths_valid), 30, criterion2, optimizer2, scheduler2, 'model2.pt')
```