training_code_HW4

November 12, 2021

[ ]:
```
#Importing necessary libraries
import numpy as np
import pandas as pd
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence,␣
↪pad_sequence
from torch.utils.data import Dataset, DataLoader
from torch.optim.lr_scheduler import StepLR
import random
torch.manual_seed(0)
random.seed(0)

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

1 Task 1: Simple Bidirectional LSTM model

1.1 In the cell below, prep_dataset function returns 2 lists one containing all the words in the training dataset and the other containing all the corresponding NER tags, prep_dataset_test function returns only one list containing all the words in the test dataset. read_file and read_file_test use the corresponding functions mentioned above to convert the input data into required format.

[ ]:
```
def prep_dataset(dataset):
    train_x, train_y = list(), list()
    x, y = list(), list()
    first = 1
    for row in dataset.itertuples():
        # print(type(row.id))
        # break
        if(row.id == '1' and first == 0):
            train_x.append(x)
```

```
            train_y.append(y)
            x = list()
            y = list()
        first = 0
        x.append(row.word)
        y.append(row.NER)

    train_x.append(x)
    train_y.append(y)

    return train_x, train_y


def read_file(path):
    train_df = list()
    with open(path, 'r') as f:
        for line in f.readlines():
            if len(line) > 2:
                id, word, ner_tag = line.strip().split(" ")
                train_df.append([id, word, ner_tag])

    train_df = pd.DataFrame(train_df, columns=['id', 'word', 'NER'])
    train_df = train_df.dropna()
    train_x, train_y = prep_dataset(train_df)
    return train_x, train_y


def prep_dataset_test(dataset):
    train_x = list()
    x = list()
    first = 1
    for row in dataset.itertuples():
        # print(type(row.id))
        # break
        if(row.id == '1' and first == 0):
            train_x.append(x)
            x = list()
        first = 0
        x.append(row.word)

    train_x.append(x)
    return train_x


def read_file_test(path):
    train_df = list()
    with open(path, 'r') as f:
```

```python
        for line in f.readlines():
            if len(line) > 1:
                id, word = line.strip().split(" ")
                train_df.append([id, word])

    train_df = pd.DataFrame(train_df, columns=['id', 'word'])
    train_df = train_df.dropna()
    train_x = prep_dataset_test(train_df)
    return train_x


train_x, train_y = read_file('./data/train')
val_x, val_y = read_file('./data/dev')
test_x = read_file_test('./data/test')

print(len(train_x), len(train_y))
print(len(val_x), len(val_y))
print(len(test_x))
```

1.2 In the cell below we define a BiLSTM class which basically represents our Bidirectional-Lstm model(with generic embedding layer). The BiLSTM_glove class represents our Bidirectional-Lstm model with Glove model based embedding. The CustomCollator function is used to manipulate the input of each batch to the Bilstm model during training and validation, while the CustomTestCollator function is used to manipulate the input of each batch to the Bilstm model during testing(used for making all batch sentences of equal length by padding the short ones). BiLSTM_DataLoader class is used to feed the data to the model from training and validation dataset. BiLSTM_TestLoader class is used to feed the data to the model from testing dataset. The create_emb_matrix function creates a matrix based on the glove model dictionary provided so as to feed the Bilstm model this matrix.(Note: We handle the issue of glove model dictionary only containing lowercase words by making the embedding the titled word equal to its lowercase counterpart plus a small displacement value for each dimension of the embedding). The prep_vocab function creates a set of all the unique words in the dataset. The prep_word_index function categorizes all the unique words of the dataset into different numeric values which can be understood by the Bilstm model. The vectorizing_sent function is used to convert the list of words into a list of list in which each sublist represents a sentence(consisting of its words) of the dataset. The vectorizing_label function is used to convert the list of NER tags into a list of list in which each sublist represents all the NER tags for a particaular sentence of the dataset. The prep_label_dict function is used to categorize all the unique NER tags into discrete numerical values that our BiLSTM model can understand.

[ ]:
```
class BiLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, linear_out_dim, hidden_dim,␣
↪lstm_layers,
                 bidirectional, dropout_val, tag_size):
        super(BiLSTM, self).__init__()
        """ Hyper Parameters """
        self.hidden_dim = hidden_dim  # hidden_dim = 256
        self.lstm_layers = lstm_layers  # LSTM Layers = 1
        self.embedding_dim = embedding_dim  # Embedding Dimension = 100
        self.linear_out_dim = linear_out_dim  # Linear Ouput Dimension = 128
        self.tag_size = tag_size  # Tag Size = 9
        self.num_directions = 2 if bidirectional else 1

        """ Initializing Network """
        self.embedding = nn.Embedding(
            vocab_size, embedding_dim)  # Embedding Layer
        self.embedding.weight.data.uniform_(-1,1)
        self.LSTM = nn.LSTM(embedding_dim,
                            hidden_dim,
                            num_layers=lstm_layers,
```

```
                             batch_first=True,
                             bidirectional=True)
        self.fc = nn.Linear(hidden_dim*self.num_directions,
                            linear_out_dim)  # 2 for bidirection
        self.dropout = nn.Dropout(dropout_val)
        self.elu = nn.ELU(alpha=0.01)
        self.classifier = nn.Linear(linear_out_dim, self.tag_size)

    def init_hidden(self, batch_size):
        h, c = (torch.zeros(self.lstm_layers * self.num_directions,
                            batch_size, self.hidden_dim).to(device),
                torch.zeros(self.lstm_layers * self.num_directions,
                            batch_size, self.hidden_dim).to(device))
        return h, c

    def forward(self, sen, sen_len):  # sen_len
        # Set initial states
        batch_size = sen.shape[0]
        h_0, c_0 = self.init_hidden(batch_size)

        # Forward propagate LSTM
        embedded = self.embedding(sen).float()
        packed_embedded = pack_padded_sequence(
            embedded, sen_len, batch_first=True, enforce_sorted=False)
        output, _ = self.LSTM(packed_embedded, (h_0, c_0))
        output_unpacked, _ = pad_packed_sequence(output, batch_first=True)
        dropout = self.dropout(output_unpacked)
        lin = self.fc(dropout)
        pred = self.elu(lin)
        pred = self.classifier(pred)
        return pred

class BiLSTM_glove(nn.Module):
    def __init__(self, vocab_size, embedding_dim, linear_out_dim, hidden_dim,␣
↪lstm_layers,
                 bidirectional, dropout_val, tag_size, emb_matrix):
        super(BiLSTM_glove, self).__init__()
        """ Hyper Parameters """
        self.hidden_dim = hidden_dim  # hidden_dim = 256
        self.lstm_layers = lstm_layers  # LSTM Layers = 1
        self.embedding_dim = embedding_dim  # Embedding Dimension = 100
        self.linear_out_dim = linear_out_dim  # Linear Ouput Dimension = 128
        self.tag_size = tag_size  # Tag Size = 9
        self.emb_matrix = emb_matrix
        self.num_directions = 2 if bidirectional else 1

        """ Initializing Network """
```

```
        self.embedding = nn.Embedding(vocab_size, embedding_dim)  # Embedding␣
↪Layer
        self.embedding.weight = nn.Parameter(torch.tensor(emb_matrix))
        self.LSTM = nn.LSTM(embedding_dim,
                            hidden_dim,
                            num_layers=lstm_layers,
                            batch_first=True,
                            bidirectional=True)
        self.fc = nn.Linear(hidden_dim*self.num_directions, linear_out_dim)  #␣
↪2 for bidirection
        self.dropout = nn.Dropout(dropout_val)
        self.elu = nn.ELU(alpha=0.01)
        self.classifier = nn.Linear(linear_out_dim, self.tag_size)

    def init_hidden(self, batch_size):
        h, c = (torch.zeros(self.lstm_layers * self.num_directions,
                            batch_size, self.hidden_dim).to(device),
                torch.zeros(self.lstm_layers * self.num_directions,
                            batch_size, self.hidden_dim).to(device))
        return h, c

    def forward(self, sen, sen_len):  # sen_len
        # Set initial states
        batch_size = sen.shape[0]
        h_0, c_0 = self.init_hidden(batch_size)

        # Forward propagate LSTM
        embedded = self.embedding(sen).float()
        packed_embedded = pack_padded_sequence(embedded, sen_len,␣
↪batch_first=True, enforce_sorted=False)
        output, _ = self.LSTM(packed_embedded, (h_0, c_0))
        output_unpacked, _ = pad_packed_sequence(output, batch_first=True)
        dropout = self.dropout(output_unpacked)
        lin = self.fc(dropout)
        pred = self.elu(lin)
        pred = self.classifier(pred)
        return pred


class BiLSTM_DataLoader(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

```python
    def __getitem__(self, index):
        x_instance = torch.tensor(self.x[index])  # , dtype=torch.long
        y_instance = torch.tensor(self.y[index])  # , dtype=torch.float
        return x_instance, y_instance

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

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

    def __getitem__(self, index):
        x_instance = torch.tensor(self.x[index])  # , dtype=torch.long
        # y_instance = torch.tensor(self.y[index])  # , dtype=torch.float
        return x_instance


class CustomCollator(object):

    def __init__(self, vocab, label):
        self.params = vocab
        self.label = label

    def __call__(self, batch):
        (xx, yy) = zip(*batch)
        x_len = [len(x) for x in xx]
        y_len = [len(y) for y in yy]
        batch_max_len = max([len(s) for s in xx])
        batch_data = self.params['<PAD>']*np.ones((len(xx), batch_max_len))
        batch_labels = -1*np.zeros((len(xx), batch_max_len))
        for j in range(len(xx)):
            cur_len = len(xx[j])
            batch_data[j][:cur_len] = xx[j]
            batch_labels[j][:cur_len] = yy[j]

        batch_data, batch_labels = torch.LongTensor(
            batch_data), torch.LongTensor(batch_labels)
        batch_data, batch_labels = Variable(batch_data), Variable(batch_labels)

        return batch_data, batch_labels, x_len, y_len

class CustomTestCollator(object):

    def __init__(self, vocab, label):
        self.params = vocab
        self.label = label
```

```
    def __call__(self, batch):
        xx = batch
        x_len = [len(x) for x in xx]
        # y_len = [len(y) for y in yy]
        batch_max_len = max([len(s) for s in xx])
        batch_data = self.params['<PAD>']*np.ones((len(xx), batch_max_len))
        # batch_labels = -1*np.zeros((len(xx), batch_max_len))
        for j in range(len(xx)):
            cur_len = len(xx[j])
            batch_data[j][:cur_len] = xx[j]
            # batch_labels[j][:cur_len] = yy[j]

        batch_data = torch.LongTensor(batch_data)
        batch_data = Variable(batch_data)

        return batch_data, x_len

def create_emb_matrix(word_idx, emb_dict, dimension):

    emb_matrix = np.zeros((len(word_idx), dimension))
    for word, idx in word_idx.items():
        if word in emb_dict:
            emb_matrix[idx] = emb_dict[word]
        else:
            if word.lower() in emb_dict:
                emb_matrix[idx] = emb_dict[word.lower()] + 5e-3
            else:
                emb_matrix[idx] = emb_dict["<UNK>"]

    return emb_matrix


""" Prepare Vocabulary"""
def prep_vocab(dataset):

    vocab = set()
    for sentence in dataset:
        for word in sentence:
            vocab.add(word)
    return vocab


def prep_word_index(train_x, val_x, test_x):

    word_idx = {"<PAD>": 0, "<UNK>": 1}
    idx = 2
```

```
    for data in [train_x, val_x, test_x]:
        for sent in data:
            for word in sent:
                if word not in word_idx:
                    word_idx[word] = idx
                    idx += 1
    return word_idx


def vectorizing_sent(train_x, word_idx):

    train_x_vec = list()
    tmp_x = list()
    for words in train_x:
        for word in words:
            tmp_x.append(word_idx[word])
        train_x_vec.append(tmp_x)
        tmp_x = list()

    return train_x_vec


def prep_label_dict(train_y, val_y):

    label1 = prep_vocab(train_y)
    label2 = prep_vocab(val_y)
    label = label1.union(label2)
    label_tuples = []
    counter = 0
    for tags in label:
        label_tuples.append((tags, counter))
        counter += 1
    label_dict = dict(label_tuples)
    return label_dict


def vectorizing_label(train_y, label_dict):

    train_y_vec = list()
    for tags in train_y:
        tmp_yy = list()
        for label in tags:
            tmp_yy.append(label_dict[label])
        train_y_vec.append(tmp_yy)
    return train_y_vec
```

1.3 In the cell below, we just use the above fuctions to process the input and output data for training, testing and validation

[ ]:
```
word_idx = prep_word_index(train_x, val_x, test_x)
train_x_vec = vectorizing_sent(train_x, word_idx)
test_x_vec = vectorizing_sent(test_x, word_idx)
val_x_vec = vectorizing_sent(val_x, word_idx)
label_dict = prep_label_dict(train_y, val_y)
train_y_vec = vectorizing_label(train_y, label_dict)
val_y_vec = vectorizing_label(val_y, label_dict)
```

1.4 In the cell below we assign each NER tag with a weight based on the frequency of its appearance in the dataset so that we can use weighted loss for our BiLSTM model to prevent the model from overfitting on frequently appearing NER tags

[ ]:
```
def initialize_class_weights(label_dict, train_y, val_y):
    class_weights = dict()
    for key in label_dict:
        class_weights[key] = 0
    total_nm_tags = 0
    for data in [train_y, val_y]:
        for tags in data:
            for tag in tags:
                total_nm_tags += 1
                class_weights[tag] += 1

    class_wt = list()
    for key in class_weights.keys():
        if class_weights[key]:
            score = round(math.log(0.35*total_nm_tags / class_weights[key]), 2)
            class_weights[key] = score if score > 1.0 else 1.0
        else:
            class_weights[key] = 1.0
        class_wt.append(class_weights[key])
    class_wt = torch.tensor(class_wt)
    return class_wt


class_wt = initialize_class_weights(label_dict, train_y, val_y)
print(class_wt)
```

1.5 In the cell below we load our BiLSTM model with generic embedding layer, use all the custom funtions created for training, define loss function, optimizer and scheduler for our model and then train the model for 200 epochs. Simultaneously, also saving model after each epoch.

1.6 (Note Hyperparameters:

1.7 Embedding dimension = 100

1.8 Hidden dimension = 256

1.9 Linear Output dimension = 128

1.10 Bidirectional = True

1.11 Dropout = 0.33

1.12 Number of LSTM layers = 1

1.13 Batch Size = 4

1.14 Loss Function = Cross Entropy with class weights

1.15 Optimizer = SGD with Learning Rate = 0.1 and Momentum = 0.9

1.16 Epochs = 200

1.17)

[ ]:
```
BiLSTM_model = BiLSTM(vocab_size=len(word_idx),
                      embedding_dim=100,
                      linear_out_dim=128,
                      hidden_dim=256,
                      lstm_layers=1,
                      bidirectional=True,
                      dropout_val=0.33,
                      tag_size=len(label_dict))
# BiLSTM_model.load_state_dict(torch.load("./BiLSTM_epoch_10.pt"))
BiLSTM_model.to(device)
print(BiLSTM_model)

BiLSTM_train = BiLSTM_DataLoader(train_x_vec, train_y_vec)
custom_collator = CustomCollator(word_idx, label_dict)
dataloader = DataLoader(dataset=BiLSTM_train,
                        batch_size=4,
                        drop_last=True,
                        collate_fn=custom_collator)

criterion = nn.CrossEntropyLoss(weight=class_wt)
# criterion = nn.NLLLoss(weight=class_wt)
# criterion = loss_fn
criterion = criterion.to(device)
criterion.requres_grad = True
```

```python
optimizer = torch.optim.SGD(BiLSTM_model.parameters(), lr=0.1, momentum=0.9)
# scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode="min")
#scheduler = StepLR(optimizer, step_size=15, gamma=0.9)
epochs = 200

for i in range(1, epochs+1):
    train_loss = 0.0
    # scheduler.step(train_loss)
    for input, label, input_len, label_len in dataloader:
        optimizer.zero_grad()
        output = BiLSTM_model(input.to(device), input_len)  # input_len
        output = output.view(-1, len(label_dict))
        label = label.view(-1)
        loss = criterion(output, label.to(device))
        # print(loss)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * input.size(1)

    train_loss = train_loss / len(dataloader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(i, train_loss))
    torch.save(BiLSTM_model.state_dict(),
               'BiLSTM_epoch_' + str(i) + '.pt')
```

1.18 In the cell below we load the BiLSTM model saved during training.

[ ]:
```python
BiLSTM_model = BiLSTM(vocab_size=len(word_idx),
                      embedding_dim=100,
                      linear_out_dim=128,
                      hidden_dim=256,
                      lstm_layers=1,
                      bidirectional=True,
                      dropout_val=0.33,
                      tag_size=len(label_dict))

BiLSTM_model.load_state_dict(torch.load("./BiLSTM_epoch_200.pt"))#125#184#b2_143
BiLSTM_model.to(device)
```

1.19 In the cell below we make our model predicted the NER tags for the validation dataset and then store the predictions in a '.out' file in the required format

1.20 (Note: The best metrics in terms of performance achieved on validation dataset for my BiLSTM model with generic embedding were:

1.21 Precision: 79.41%, Recall: 75.36%, FB1: 77.33%

1.22)

```
[ ]: #tesing on validation data
BiLSTM_dev = BiLSTM_DataLoader(val_x_vec, val_y_vec)
custom_collator = CustomCollator(word_idx, label_dict)
dataloader_dev = DataLoader(dataset=BiLSTM_dev,
                            batch_size=1,
                            shuffle=False,
                            drop_last=True,
                            collate_fn=custom_collator)

# Reverse vocab and label Dictionary
rev_label_dict = {v: k for k, v in label_dict.items()}
rev_vocab_dict = {v: k for k, v in word_idx.items()}


file = open("dev1_train.out", 'w')
for dev_data, label, dev_data_len, label_data_len in dataloader_dev:

    pred = BiLSTM_model(dev_data.to(device), dev_data_len)
    pred = pred.cpu()
    pred = pred.detach().numpy()
    label = label.detach().numpy()
    dev_data = dev_data.detach().numpy()
    pred = np.argmax(pred, axis=2)
    pred = pred.reshape((len(label), -1))

    for i in range(len(dev_data)):
        for j in range(len(dev_data[i])):
            if dev_data[i][j] != 0:
                word = rev_vocab_dict[dev_data[i][j]]
                gold = rev_label_dict[label[i][j]]
                op = rev_label_dict[pred[i][j]]
                file.write(" ".join([str(j+1), word, gold, op]))
                file.write("\n")
        file.write("\n")

file.close()

#!perl conll03eval.txt < op.txt
```

1.23 In the cell below we make our model predicted the NER tags for the test dataset and then store the predictions in a '.out' file in the required format

```
[ ]: #Testing on Testing Dataset
rev_label_dict = {v: k for k, v in label_dict.items()}
rev_vocab_dict = {v: k for k, v in word_idx.items()}

BiLSTM_test = BiLSTM_TestLoader(test_x_vec)
```

```python
custom_test_collator = CustomTestCollator(word_idx, label_dict)
dataloader_test = DataLoader(dataset=BiLSTM_test,
                                batch_size=1,
                                shuffle=False,
                                drop_last=True,
                                collate_fn=custom_test_collator)


file = open("test1_train.out", 'w')
for test_data, test_data_len in dataloader_test:

    pred = BiLSTM_model(test_data.to(device), test_data_len)
    pred = pred.cpu()
    pred = pred.detach().numpy()
    test_data = test_data.detach().numpy()
    pred = np.argmax(pred, axis=2)
    pred = pred.reshape((len(test_data), -1))

    for i in range(len(test_data)):
        for j in range(len(test_data[i])):
            if test_data[i][j] != 0:
                word = rev_vocab_dict[test_data[i][j]]
                op = rev_label_dict[pred[i][j]]
                file.write(" ".join([str(j+1), word, op]))
                file.write("\n")

        file.write("\n")

file.close()
```

2 Task 2: Using GloVe word embeddings

2.1 In the cell below, we load the Glove model data and then convert it into a dictionary. We then add tokens "<PAD>" and "<UNK>" in the dictionary as well. and then convert the embedding dictionary into a matrix to feed to the embedding layer in our model.

[ ]:
```python
#glove = pd.read_csv('./glove.6B.100d.txt', sep=" ",
                    quoting=3, header=None, index_col=0)
#glove_emb = {key: val.values for key, val in glove.T.items()}

#word_idx = prep_word_index(train_x, val_x, test_x)
#glove_vec = np.array([glove_emb[key] for key in glove_emb])
#glove_emb["<PAD>"] = np.zeros((100,), dtype="float64")
#glove_emb["<UNK>"] = np.mean(glove_vec, axis=0, keepdims=True).reshape(100,)
#emb_matrix = create_emb_matrix(
```

```
#    word_idx=word_idx, emb_dict=glove_emb, dimension=100)

emb_matrix = np.load('emb_matrix.npy')

vocab_size = emb_matrix.shape[0]
vector_size = emb_matrix.shape[1]
print(vocab_size, vector_size)
```

2.2 In the cell below we load our BiLSTM model with glove embedding layer, and use all the custom funtions created for training, define loss function, optimizer and scheduler for our model and then train the model for 200 epochs. Simultaneously, also saving model after each epoch.

2.3 (Note Hyperparameters:

2.4 Embedding dimension = 100

2.5 Hidden dimension = 256

2.6 Linear Output dimension = 128

2.7 Bidirectional = True

2.8 Dropout = 0.33

2.9 Number of LSTM layers = 1

2.10 Batch Size = 8

2.11 Loss Function = Cross Entropy with class weights

2.12 Optimizer = SGD with Learning Rate = 0.1 and Momentum = 0.9

2.13 Epochs = 50

2.14)

[ ]:
```
BiLSTM_model = BiLSTM_glove(vocab_size=len(word_idx),
                      embedding_dim=100,
                      linear_out_dim=128,
                      hidden_dim=256,
                      lstm_layers=1,
                      bidirectional=True,
                      dropout_val=0.33,
                      tag_size=len(label_dict),
                      emb_matrix=emb_matrix)
# BiLSTM_model.load_state_dict(torch.load("./BiLSTM_glove_20.pt"))
BiLSTM_model.to(device)
print(BiLSTM_model)

BiLSTM_train = BiLSTM_DataLoader(train_x_vec, train_y_vec)
custom_collator = CustomCollator(word_idx, label_dict)
```

```
dataloader = DataLoader(dataset=BiLSTM_train,
                        batch_size=8,
                        drop_last=True,
                        collate_fn=custom_collator)

criterion = nn.CrossEntropyLoss(weight=class_wt)
# criterion = nn.NLLLoss(weight=class_wt)
# criterion = loss_fn
criterion = criterion.to(device)
criterion.requres_grad = True
optimizer = torch.optim.SGD(BiLSTM_model.parameters(), lr=0.1, momentum=0.9)
# scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode="min")
scheduler = StepLR(optimizer, step_size=15, gamma=0.9)
epochs = 50

for i in range(1, epochs+1):
    train_loss = 0.0
    # scheduler.step(train_loss)
    for input, label, input_len, label_len in dataloader:
        optimizer.zero_grad()
        output = BiLSTM_model(input.to(device), input_len)  # input_len
        output = output.view(-1, len(label_dict))
        label = label.view(-1)
        loss = criterion(output, label.to(device))
        # print(loss)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * input.size(1)

    train_loss = train_loss / len(dataloader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(i, train_loss))
    torch.save(BiLSTM_model.state_dict(),
               'BiLSTM_glove_' + str(i) + '.pt')
```

2.15 In the cell below we load the BiLSTM model with Glove embedding saved during training.

[ ]:
```
BiLSTM_model = BiLSTM_glove(vocab_size=len(word_idx),
                        embedding_dim=100,
                        linear_out_dim=128,
                        hidden_dim=256,
                        lstm_layers=1,
                        bidirectional = True,
                        dropout_val=0.33,
                        tag_size=len(label_dict),
                        emb_matrix=emb_matrix)
```

```
BiLSTM_model.load_state_dict(torch.load("./BiLSTM_glove_50.pt"))
BiLSTM_model.to(device)
```

2.16 In the cell below we make our model predicted the NER tags for the validation dataset and then store the predictions in a '.out' file in the required format

2.17 (Note: The best metrics in terms of performance achieved on validation dataset for my BiLSTM model with Glove embedding were:

2.18 Precision: 89.94%, Recall: 89.97%, FB1: 89.95%

2.19)

[ ]:
```
#predicting for validation dataset
BiLSTM_dev = BiLSTM_DataLoader(val_x_vec, val_y_vec)
custom_collator = CustomCollator(word_idx, label_dict)
dataloader_dev = DataLoader(dataset=BiLSTM_dev,
                            batch_size=8,
                            shuffle=False,
                            drop_last=True,
                            collate_fn=custom_collator)
print(label_dict)
rev_label_dict = {v: k for k, v in label_dict.items()}
rev_vocab_dict = {v: k for k, v in word_idx.items()}

res = []
file = open("dev2_train.out", 'w')
for dev_data, label, dev_data_len, label_data_len in dataloader_dev:

    pred = BiLSTM_model(dev_data.to(device), dev_data_len)
    pred = pred.cpu()
    pred = pred.detach().numpy()
    label = label.detach().numpy()
    dev_data = dev_data.detach().numpy()
    pred = np.argmax(pred, axis=2)
    pred = pred.reshape((len(label), -1))

    for i in range(len(dev_data)):
        for j in range(len(dev_data[i])):
            if dev_data[i][j] != 0:
                word = rev_vocab_dict[dev_data[i][j]]
                gold = rev_label_dict[label[i][j]]
                op = rev_label_dict[pred[i][j]]
                res.append((word, gold, op))
                file.write(" ".join([str(j + 1), word, gold, op]))
                file.write("\n")
        file.write("\n")
```

```
file.close()
```

```
[ ]: #!perl conll03eval.txt < op.txt
```

2.20 In the cell below we make our model predicted the NER tags for the test dataset and then store the predictions in a '.out' file in the required format

```
[ ]: #predicting for testing dataset

BiLSTM_test = BiLSTM_TestLoader(test_x_vec)
custom_test_collator = CustomTestCollator(word_idx, label_dict)
dataloader_test = DataLoader(dataset=BiLSTM_test,
                                batch_size=1,
                                shuffle=False,
                                drop_last=True,
                                collate_fn=custom_test_collator)

rev_label_dict = {v: k for k, v in label_dict.items()}
rev_vocab_dict = {v: k for k, v in word_idx.items()}
res = []
file = open("test2_train.out", 'w')
for test_data, test_data_len in dataloader_test:

    pred = BiLSTM_model(test_data.to(device), test_data_len)
    pred = pred.cpu()
    pred = pred.detach().numpy()
    # label = label.detach().numpy()
    test_data = test_data.detach().numpy()
    pred = np.argmax(pred, axis=2)
    pred = pred.reshape((len(test_data), -1))

    for i in range(len(test_data)):
        for j in range(len(test_data[i])):
            if test_data[i][j] != 0:
                word = rev_vocab_dict[test_data[i][j]]
                # gold = rev_label_dict[label[i][j]]
                op = rev_label_dict[pred[i][j]]
                res.append((word, op))
                file.write(" ".join([str(j + 1), word, op]))
                file.write("\n")
        file.write("\n")
file.close()
```