LSTM Experiment (No Augmentation)

Real News and Fake News (~60k total)

Class: Label

Real: 1

Fake: 0

```
#deal with tensors
import torch

#handling text data
from torchtext import data
```

```
#Reproducing same results
SEED = 2020

#Torch
torch.manual_seed(SEED)

#Cuda algorithms
torch.backends.cudnn.deterministic = True
```

```
TEXT = data.Field(tokenize='spacy',batch_first=True,include_lengths=True)
LABEL = data.LabelField(dtype = torch.float,batch_first=True)
```

```
import pandas as pd
import numpy as np

file = "combined_{}.csv"
dfs = []
for i in range(3):
```

```
        fp = file.format(i+1)
        read = pd.read_csv(fp)
        read = read[['clean_text', 'label']]
        dfs.append(read)

    dfs[2] = dfs[2][:-13000]

    data = pd.concat(dfs)
    data.tail()
    data.reset_index(inplace=True, drop=True)
    data.dropna(inplace=True)

    data.to_csv('combined.csv')

    data = pd.read_csv('combined.csv')
    print(data.shape[0])
    print(data[data.label == 1].shape[0], "Real")
    print(data[data.label == 0].shape[0], "Fake")
    data.head(10)
```

```
fields = [(None, None), ('clean_text', TEXT), ('label', LABEL)]

#loading custom dataset
training_data= data.TabularDataset(path = 'combined.csv',format = 'csv', fields = fields,skip_header = True)
```

```
{'clean_text': ['house', 'dem', 'aide', 'did', 'nt', 'even', 'see', 'comeys', 'letter', 'jason', 'chaffetz',
```

```
import random

train_data, test_data = training_data.split(split_ratio=0.8, random_state = random.seed(SEED))
train_data, valid_data = train_data.split(split_ratio=0.7, random_state = random.seed(SEED))
```

```
#initialize glove embeddings
TEXT.build_vocab(train_data,min_freq=3,vectors = "glove.6B.100d")
LABEL.build_vocab(train_data)
```

```
#No. of unique tokens in text
print("Size of TEXT vocabulary:",len(TEXT.vocab))

#No. of unique tokens in label
print("Size of LABEL vocabulary:",len(LABEL.vocab))

#Commonly used words
print(TEXT.vocab.freqs.most_common(10))

#Word dictionary
print(TEXT.vocab.stoi)
```

```
Size of TEXT vocabulary: 84973
Size of LABEL vocabulary: 2
[(' ', 245401), ('said', 78860), ('trump', 61193), ('  ', 60071), ('nt', 58924), ('would', 49617), ('one', 49
defaultdict(<function _default_unk_index at 0x7f63940c57b8>, {'<unk>': 0, '<pad>': 1, ' ': 2, 'said': 3, 'tru
```

```
#check whether cuda is available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

#set batch size
BATCH_SIZE = 64

#Load an iterator
train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    sort_key = lambda x: len(x.clean_text),
    sort_within_batch=True,
    device = device)
```

```
import torch.nn as nn

class LSTM(nn.Module):

    #define all the layers used in model
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
```

```
                 bidirectional, dropout):

        #Constructor
        super().__init__()

        #embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        #lstm layer
        self.lstm = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout,
                           batch_first=True)

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

        #activation function
        self.act = nn.Sigmoid()

    def forward(self, text, text_lengths):

        #text = [batch size,sent_length]
        embedded = self.embedding(text)
        #embedded = [batch size, sent_len, emb dim]

        #packed sequence
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths,batch_first=True)

        packed_output, (hidden, cell) = self.lstm(packed_embedded)
        #hidden = [batch size, num layers * num directions,hid dim]
        #cell = [batch size, num layers * num directions,hid dim]

        #concat the final forward and backward hidden state
        hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)

        #hidden = [batch size, hid dim * num directions]
```

```
        dense_outputs=self.fc(hidden)

        #Final activation function
        outputs=self.act(dense_outputs)

        return outputs


#define hyperparameters
size_of_vocab = len(TEXT.vocab)
embedding_dim = 100
num_hidden_nodes = 32
num_output_nodes = 1
num_layers = 2
bidirection = True
dropout = 0.2

#instantiate the model
model = LSTM(size_of_vocab, embedding_dim, num_hidden_nodes,num_output_nodes, num_layers,
                   bidirectional = True, dropout = dropout)


#architecture
print(model)

#No. of trianable parameters
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')

#Initialize the pretrained embedding
pretrained_embeddings = TEXT.vocab.vectors
model.embedding.weight.data.copy_(pretrained_embeddings)

print(pretrained_embeddings.shape)
```

```
LSTM(
  (embedding): Embedding(84973, 100)
  (lstm): LSTM(100, 32, num_layers=2, batch_first=True, dropout=0.2, bidirectional=True)
  (fc): Linear(in_features=64, out_features=1, bias=True)
  (act): Sigmoid()
)
The model has 8,556,757 trainable parameters
torch.Size([84973, 100])
```

```
import torch.optim as optim

#define optimizer and loss
optimizer = optim.Adam(model.parameters())
criterion = nn.BCELoss()

#define metric
def binary_accuracy(preds, y):
    #round predictions to the closest integer
    rounded_preds = torch.round(preds)

    correct = (rounded_preds == y).float()
    acc = correct.sum() / len(correct)
    return acc

#push to cuda if available
model = model.to(device)
criterion = criterion.to(device)
```

```
def train(model, iterator, optimizer, criterion):

    #initialize every epoch
    epoch_loss = 0
    epoch_acc = 0

    #set the model in training phase
    model.train()

    for batch in iterator:
```

```
    for batch in iterator:

        #resets the gradients after every batch
        optimizer.zero_grad()

        #retrieve text and no. of words
        text, text_lengths = batch.clean_text

        #convert to 1D tensor
        predictions = model(text, text_lengths).squeeze()

        #compute the loss
        loss = criterion(predictions, batch.label)

        #compute the binary accuracy
        acc = binary_accuracy(predictions, batch.label)

        #backpropage the loss and compute the gradients
        loss.backward()

        #update the weights
        optimizer.step()

        #loss and accuracy
        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)


def evaluate(model, iterator, criterion):

    #initialize every epoch
    epoch_loss = 0
    epoch_acc = 0

    #deactivating dropout layers
    model.eval()

    #deactivates autograd
```

```
    with torch.no_grad():

        for batch in iterator:

            #retrieve text and no. of words
            text, text_lengths = batch.clean_text

            #convert to 1d tensor
            predictions = model(text, text_lengths).squeeze()

            #compute loss and accuracy
            loss = criterion(predictions, batch.label)
            acc = binary_accuracy(predictions, batch.label)

            #keep track of loss and accuracy
            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)


def test(model, iterator, criterion):

    #initialize every epoch
    epoch_loss = 0
    epoch_acc = 0

    #deactivating dropout layers
    model.eval()

    #deactivates autograd
    with torch.no_grad():

        for batch in iterator:

            #retrieve text and no. of words
            text, text_lengths = batch.clean_text

            #convert to 1d tensor
            predictions = model(text, text_lengths).squeeze()
```

```
            #compute loss and accuracy
            loss = criterion(predictions, batch.label)
            acc = binary_accuracy(predictions, batch.label)

            #keep track of loss and accuracy
            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)


N_EPOCHS = 5
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    #train the model
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)

    #evaluate the model
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    #save the best model
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'saved_weights.pt')

    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
	Train Loss: 0.394 | Train Acc: 80.30%
	 Val. Loss: 0.269 |  Val. Acc: 86.41%
	Train Loss: 0.209 | Train Acc: 90.01%
	 Val. Loss: 0.231 |  Val. Acc: 88.21%
	Train Loss: 0.141 | Train Acc: 93.27%
	 Val. Loss: 0.239 |  Val. Acc: 88.40%
	Train Loss: 0.103 | Train Acc: 95.15%
	 Val. Loss: 0.303 |  Val. Acc: 88.09%
	Train Loss: 0.078 | Train Acc: 96.49%
	 Val. Loss: 0.368 |  Val. Acc: 87.88%
```

```
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Acc: {test_acc*100:.2f}%')
```

```
Test Acc: 88.84%
```