Coding Quiz

With the given dataset, Please compare your best possible version of

```
(1) BiLSTM,
(2) BiLSTM with multiplicative attention (you have to fix e), and
(3) BERT
```

Report the accuracy, precision, recall, and f1-score of each model.

For (1) and (2), use the following hyperparameters:

```
Optimizer: SG
Embedding: GloVe (https://pytorch.org/text/stable/vocab.html#torchtext.vocab.GloV
e) >> Please change the embed_dim accordingly.
Epochs: 2
Batch size: 32
Save the model with the best params
```

Anything not stated, please assume accordingly

For (2), Multiplicative attention differs from the General Attention (in Assignment 4) such that, for the *Alignment Scores* (or Energy), we multiply the Keys with some weights first before we dot the Keys with the Query.

$$\mathbf{e}_i = \mathbf{q}^T \ \mathbf{W}\mathbf{k}_t$$

where $\mathbf{W} \in \mathbb{R}^{h,h}$

- Hint : The shape of the Keys before and after multiplying with the weights should be the same

For (3), use this tutorial [https://huggingface.co/docs/transformers/training (https://huggingface.co/docs/transformers/training)](https://huggingface.co/docs/transformers/training) as your guide.

In [1]:

```
# import os

# os.environ['http_proxy'] = 'http://192.41.170.23:3128'
# os.environ['https_proxy'] = 'http://192.41.170.23:3128'
```

In [2]:

```
import torchtext
import torch
from torch import nn
import math
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
/home/damian/pythonDSAI/lib/python3.8/site-packages/tqdm/auto.py:22: TqdmWar
ning: IProgress not found. Please update jupyter and ipywidgets. See http
s://ipywidgets.readthedocs.io/en/stable/user_install.html (https://ipywidget
s.readthedocs.io/en/stable/user_install.html)
  from .autonotebook import tqdm as notebook_tqdm

cuda
```

1. Load the IMDB Review dataset from TorchText (https://pytorch.org/text/stable/datasets.html#id10 (https://pytorch.org/text/stable/datasets.html#id10))

In [3]:

```
import torchtext
import torch
from torch import nn

#make our work comparable if restarted the kernel
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

from torchtext.datasets import IMDB
train_iter, test_iter = IMDB(split=('train', 'test'))


from torchtext.data.utils import get_tokenizer
tokenizer = get_tokenizer('spacy', language='en_core_web_sm')

from torchtext.vocab import build_vocab_from_iterator
def yield_tokens(data_iter):
    for _, text in data_iter:
        yield tokenizer(text)

vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials=['<unk>', '<pad>',
vocab.set_default_index(vocab["<unk>"])

#https://github.com/pytorch/text/issues/1350
from torchtext.vocab import GloVe
fast_vectors = GloVe(name='6B', dim=100)

fast_embedding = fast_vectors.get_vecs_by_tokens(vocab.get_itos()).to(device)
# vocab.get_itos() returns a list of strings (tokens), where the token at the i'th posi
# get_vecs_by_tokens gets the pre-trained vector for each string when given a list of s
# therefore pretrained_embedding is a fully "aligned" embedding matrix
```

In [4]:

```
input_dim = len(vocab)
hidden_dim = 256
embed_dim = 100
output_dim = 1

pad_idx = vocab['<pad>']
num_layers = 2
bidirectional = True
dropout = 0.5

batch_size = 32
num_epochs = 3
lr=0.0001
```

In [5]:

```python
text_pipeline = lambda x: vocab(tokenizer(x))
label_pipeline = lambda x: 1 if x == 'pos' else 0

from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence #++

def collate_batch(batch):
    label_list, text_list, length_list = [], [], []
    for (_label, _text) in batch:
        label_list.append(label_pipeline(_label))
        processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
        text_list.append(processed_text)
        length_list.append(processed_text.size(0))  #++<-----packed padded sequences r
    #criterion expects float labels
    return torch.tensor(label_list, dtype=torch.float64), pad_sequence(text_list, padd

from torch.utils.data.dataset import random_split
from torchtext.data.functional import to_map_style_dataset

train_iter = IMDB(split='train')
test_iter = IMDB(split='test')

train_dataset = to_map_style_dataset(train_iter)
test_dataset = to_map_style_dataset(test_iter)

num_train = int(len(train_dataset) * 0.15)
num_val = int(len(train_dataset) * 0.10)
num_test = int(len(test_dataset) * 0.05)

split_train_, split_valid_, _ = \
    random_split(train_dataset, [num_train, num_val,len(train_dataset)- num_train - nu

split_test_, _ = \
    random_split(train_dataset, [num_test, len(test_dataset) - num_test])

train_loader = DataLoader(split_train_, batch_size=batch_size,
                              shuffle=True, collate_fn=collate_batch)
valid_loader = DataLoader(split_valid_, batch_size=batch_size,
                              shuffle=True, collate_fn=collate_batch)
test_loader = DataLoader(split_test_, batch_size=batch_size,
                             shuffle=True, collate_fn=collate_batch)

#explicitly initialize weights for better learning
def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.RNN):
        for name, param in m.named_parameters():
            if 'bias' in name:
                nn.init.zeros_(param)
            elif 'weight' in name:
                nn.init.orthogonal_(param) #<---here

def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8, NOT 8
    """
    #round predictions to the closest integer
```

```
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc

def train(model, loader, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.train() #useful for batchnorm and dropout
    for i, (label, text, text_length) in enumerate(loader):
        label = label.to(device) #(batch_size, )
        text = text.to(device) #(batch_size, seq len)

        #predict
        predictions = model(text, text_length) #output by the fc is (batch_size, 1), t
        predictions = predictions.squeeze(1)

        #calculate loss
        loss = criterion(predictions, label)
        acc = binary_accuracy(predictions, label)

        #backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

        if i == 10:
            break

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

def evaluate(model, loader, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()

    with torch.no_grad():
        for i, (label, text, text_length) in enumerate(loader):
            label = label.to(device) #(batch_size, )
            text = text.to(device) #(batch_size, seq len)

            predictions = model(text, text_length)
            predictions = predictions.squeeze(1)

            loss = criterion(predictions, label)
            acc = binary_accuracy(predictions, label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

            if i == 10:
                break

    return epoch_loss / len(loader), epoch_acc / len(loader)
```

In [6]:

```
class new_LSTM_cell(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, lstm_type: str):
        super().__init__()

        self.hidden_dim = hidden_dim
        self.lstm_type = lstm_type

        # initialise the trainable Parameters
        self.U_i = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
        self.W_i = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
        self.b_i = nn.Parameter(torch.Tensor(hidden_dim))

        self.U_f = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
        self.W_f = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
        self.b_f = nn.Parameter(torch.Tensor(hidden_dim))

        self.U_g = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
        self.W_g = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
        self.b_g = nn.Parameter(torch.Tensor(hidden_dim))

        self.U_o = nn.Parameter(torch.Tensor(input_dim, hidden_dim))
        self.W_o = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
        self.b_o = nn.Parameter(torch.Tensor(hidden_dim))

        if self.lstm_type == 'peephole' :
            self.P_i = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
            self.P_f = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))
            self.P_o = nn.Parameter(torch.Tensor(hidden_dim, hidden_dim))

        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_dim)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self, x, init_states=None):
        bs, seq_len, _ = x.shape
        output = []

        # initialize the hidden state and cell state for the first time step
        if init_states is None:
            h_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
            c_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
        else:
            h_t, c_t = init_states

        # For each time step of the input x, do ...
        for t in range(seq_len):
            x_t = x[:, t, :] # get x data of time step t (SHAPE: (batch_size, input_di

            if self.lstm_type in ['vanilla', 'coupled'] :
                f_t = torch.sigmoid(    h_t @ self.W_f  +  x_t @ self.U_f  +  self.b_f)
                o_t = torch.sigmoid(    h_t @ self.W_o  +  x_t @ self.U_o  +  self.b_o)
                if self.lstm_type == 'vanilla':
                    i_t = torch.sigmoid(    h_t @ self.W_i  +  x_t @ self.U_i  +  self.
                if self.lstm_type == 'coupled':
                    i_t = (1 - f_t)
            if self.lstm_type == 'peephole' :
```

```
                i_t = torch.sigmoid( h_t @ self.W_i + x_t @ self.U_i + c_t @ self.P_i +
                f_t = torch.sigmoid( h_t @ self.W_f + x_t @ self.U_f + c_t @ self.P_f +
                o_t = torch.sigmoid( h_t @ self.W_o + x_t @ self.U_o + c_t @ self.P_o +

            g_t = torch.tanh(       h_t @ self.W_g  +  x_t @ self.U_g   + self.b_g)
            c_t = (f_t * c_t) + (i_t * g_t)
            h_t = o_t * torch.tanh(c_t)

            output.append(h_t.unsqueeze(0)) # reshape h_t to (1, batch_size, hidden_dim

        output = torch.cat(output, dim = 0) # concatenate h_t of all time steps into SH
        output = output.transpose(0, 1).contiguous() # just transpose to SHAPE :(seq_le
        return output, (h_t, c_t)
```

In [7]:

```
import torch.nn as nn
from torch.nn import functional as F

class BiLSTM_model(nn.Module):
    def __init__(self, input_dim: int, embed_dim: int, hidden_dim: int, output_dim: int
        super().__init__()
        self.num_directions = 2
        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)
        self.hidden_dim = hidden_dim

        self.forward_lstm   =  new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'vanil
        self.backward_lstm  =  new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'vanil

        # These should be torch Parameters
        self.W_h = nn.Parameter(torch.Tensor(hidden_dim*self.num_directions, hidden_dim
        self.b_h = nn.Parameter(torch.Tensor(hidden_dim*self.num_directions))

        self.fc  = nn.Linear(hidden_dim*self.num_directions, output_dim)

        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_dim)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self, text, text_lengths):
        embedded      = self.embedding(text)
        embedded_flip =  torch.flip(embedded, [1])

        output_forward, (hn_forward, cn_forward)    = self.forward_lstm(embedded, init_
        output_backward, (hn_backward, cn_backward) = self.backward_lstm(embedded_flip,

        concat_hn = torch.cat( (hn_forward, hn_backward), dim=1 )
        ht        = torch.sigmoid( concat_hn @ self.W_h + self.b_h)

        return self.fc(ht)
```

In [8]:

```
import torch.optim as optim

bilstm = BiLSTM_model(input_dim, embed_dim, hidden_dim, output_dim).to(device)
bilstm.apply(initialize_weights)
bilstm.embedding.weight.data = fast_embedding

optimizer = optim.SGD(bilstm.parameters(), lr=lr) #<----changed to Adam
criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

best_valid_loss = float('inf')

train_losses = []
train_accs = []
# test_losses = []
# test_accs = []
valid_losses = []
valid_accs = []

for epoch in range(num_epochs):
    train_loss, train_acc = train(bilstm, train_loader, optimizer, criterion)
#     test_loss, test_acc = test(bilstm, test_loader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(bilstm, valid_loader, criterion)

    train_losses.append(train_loss)
    train_accs.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accs.append(valid_acc)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(bilstm.state_dict(), 'glove_BiLSTM_attention.pt')

    print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_acc
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')

# del bilstm
# del optimizer
# del criterion
```

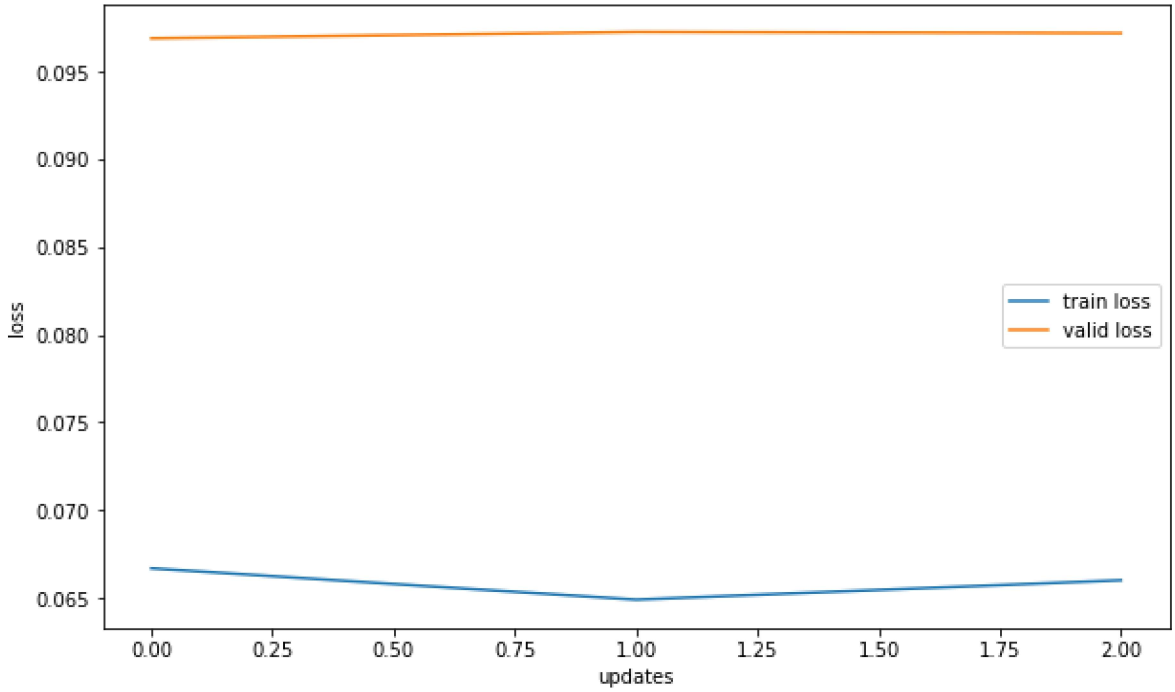
```
Epoch: 01 | Train Loss: 0.067 | Train Acc: 4.53%
	 Val. Loss: 0.097 |  Val. Acc: 7.48%
Epoch: 02 | Train Loss: 0.065 | Train Acc: 5.01%
	 Val. Loss: 0.097 |  Val. Acc: 7.36%
Epoch: 03 | Train Loss: 0.066 | Train Acc: 4.66%
	 Val. Loss: 0.097 |  Val. Acc: 7.36%
```

In [9]:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(train_losses, label = 'train loss')
ax.plot(valid_losses, label = 'valid loss')
plt.legend()
ax.set_xlabel('updates')
ax.set_ylabel('loss')
```

Out[9]:

```
Text(0, 0.5, 'loss')
```

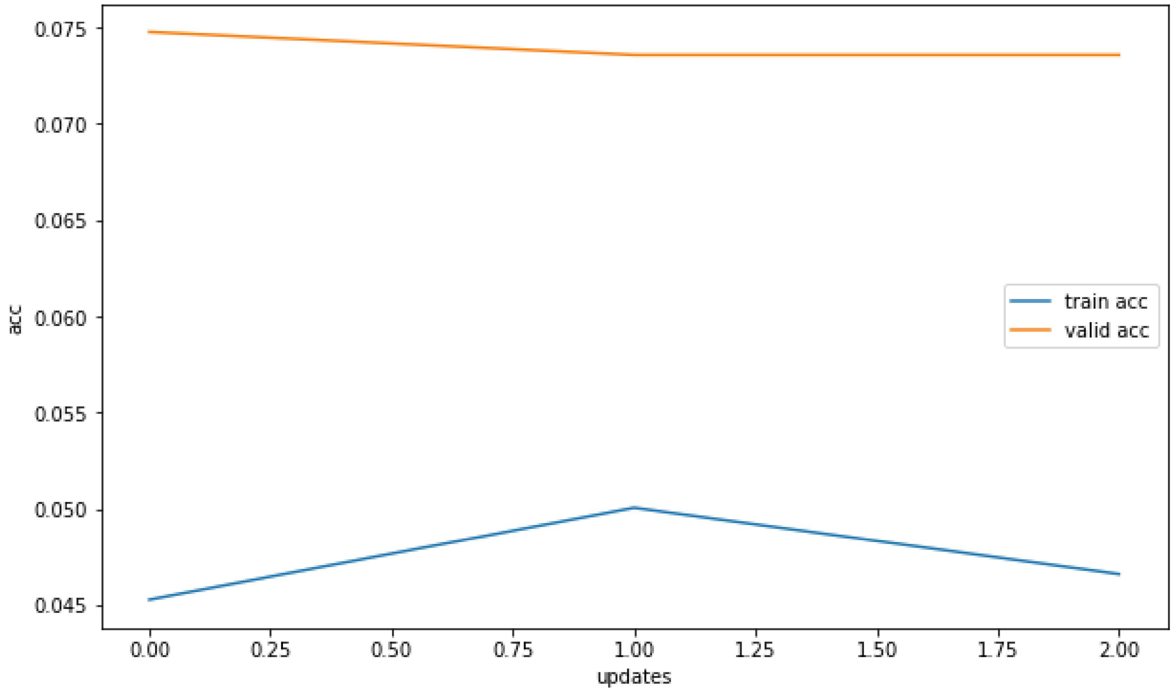


In [10]:

```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(train_accs, label = 'train acc')
ax.plot(valid_accs, label = 'valid acc')
plt.legend()
ax.set_xlabel('updates')
ax.set_ylabel('acc')
```

Out[10]:

```
Text(0, 0.5, 'acc')
```



In [11]:

```
bilstm.load_state_dict(torch.load('glove_BiLSTM_attention.pt'))
bilstm.eval()
```

Out[11]:

```
BiLSTM_model(
  (embedding): Embedding(121068, 100, padding_idx=1)
  (forward_lstm): new_LSTM_cell()
  (backward_lstm): new_LSTM_cell()
  (fc): Linear(in_features=512, out_features=1, bias=True)
)
```

In [18]:

```
import torch.nn as nn

#this attention mask will be apply after Q @ K^T thus the shape will be batch, seq_len,
def get_pad_mask(text):  #[batch, seq_len]
    batch_size, seq_len = text.size()
    # eq(zero) is lstm output over PAD token
    pad_mask = text.data.eq(0).unsqueeze(1)  # batch_size x 1 x seq_len; we unsqueeze s
    return pad_mask.expand(batch_size, seq_len, seq_len)  # batch_size x seq_len x seq_

class LSTM(nn.Module):
    def __init__(self, len_reduction='mean'):
        super().__init__()
        #put padding_idx so asking the embedding layer to ignore padding
        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)
        self.lstm = nn.LSTM(embed_dim,
                           hidden_dim,
                           num_layers=num_layers,
                           bidirectional=bidirectional,
                           dropout=dropout,
                           batch_first=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.softmax       = nn.LogSoftmax(dim=1)
        self.len_reduction = len_reduction
        self.lin_Q = nn.Linear(hidden_dim * 2, hidden_dim * 2)
        self.lin_K = nn.Linear(hidden_dim * 2, hidden_dim * 2)
        self.lin_V = nn.Linear(hidden_dim * 2, hidden_dim * 2)

    # lstm_output : [batch_size, seq len, n_hidden * num_directions(=2)]
    def self_attention_net(self, lstm_output, pad_mask):
        q = self.lin_Q(torch.clone(lstm_output))
        k = self.lin_K(torch.clone(lstm_output))
        v = self.lin_V(torch.clone(lstm_output))
        # q : [batch_size, seq_len, n_hidden * num_directions(=2)]
        # k.transpose(1, 2): [batch_size, n_hidden * num_directions(=2), seq_len]
        # attn_w = [batch_size, seq_len, seq_len]

        attn_w = torch.matmul(q, k.transpose(1, 2))

        #apply padding mask
        if self.mask:
            attn_w.masked_fill_(pad_mask, -1e9) # Fills elements of self tensor with v

        sfmx_attn_w = self.softmax(attn_w)
        # context = [batch_size, seq_len, hidden_dim * num_directions(=2)]
        context = torch.matmul(sfmx_attn_w, v)
        if self.len_reduction == "mean":
            return torch.mean(context, dim=1), sfmx_attn_w.cpu().data.numpy()
        elif self.len_reduction == "sum":
            return torch.sum(context, dim=1), sfmx_attn_w.cpu().data.numpy()
        elif self.len_reduction == "last":
            return context[:, -1, :], sfmx_attn_w.cpu().data.numpy()

    def forward(self, text, text_lengths, mask=True):

        self.mask = mask
        pad_mask = get_pad_mask(text)

        #text = [batch size, seq len]
        embedded = self.embedding(text)
```

```
        #++ pack sequence ++
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths.to(

        #embedded = [batch size, seq len, embed dim]
        packed_output, (hn, cn) = self.lstm(packed_embedded)  #if no h0, all zeroes

        #++ unpack in case we need to use it ++
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output, batch_

        #output = [batch size, seq len, hidden dim * num directions]
        #output over padding tokens are zero tensors

        attn_output, attention  = self.self_attention_net(output, pad_mask)

        #attn_output = [batch_size, hidden_dim * num_direction(=2)]
        return self.fc(attn_output), attention
```

In [19]:

```
#explicitly initialize weights for better learning
def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.RNN):
        for name, param in m.named_parameters():
            if 'bias' in name:
                nn.init.zeros_(param)
            elif 'weight' in name:
                nn.init.orthogonal_(param) #<---here
```

In [20]:

```
model = LSTM().to(device)
model.apply(initialize_weights)
model.embedding.weight.data = fast_embedding #**<------applied the fast text embedding
```

In [21]:

```
best_valid_loss = float('inf')

train_losses = []
train_accs = []
valid_losses = []
valid_accs = []

for epoch in range(num_epochs):
    train_loss, train_acc = train(model, train_loader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_loader, criterion)

    #for plotting
    train_losses.append(train_loss)
    train_accs.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accs.append(valid_acc)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'models/GloVe_BiLSTM_attention.pt')

    print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_acc
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [21], in <cell line: 8>()
      6 valid_accs = []
      8 for epoch in range(num_epochs):
----> 9     train_loss, train_acc = train(model, train_loader, optimizer, cr
iterion)
     10     valid_loss, valid_acc = evaluate(model, valid_loader, criterion)
     12     #for plotting

Input In [5], in train(model, loader, optimizer, criterion)
     73 #predict
     74 predictions = model(text, text_length) #output by the fc is (batch_s
ize, 1), thus need to remove this 1
---> 75 predictions = predictions.squeeze(1)
     77 #calculate loss
     78 loss = criterion(predictions, label)

AttributeError: 'tuple' object has no attribute 'squeeze'
```

BERT

In [22]:

```
import pandas as pd
from transformers import BertTokenizer, BertModel
from tqdm import tqdm
import numpy as np
from torch.optim import Adam

```

In [23]:

```
# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
class BertClassifier(nn.Module):

    def __init__(self, dropout=0.5):

        super(BertClassifier, self).__init__()

        self.bert = BertModel.from_pretrained('bert-base-cased')
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(768, 5)
        self.relu = nn.ReLU()

    def forward(self, input_id, mask):

        _, pooled_output = self.bert(input_ids= input_id, attention_mask=mask,return_di
        dropout_output = self.dropout(pooled_output)
        linear_output = self.linear(dropout_output)
        final_layer = self.relu(linear_output)

        return final_layer
```

In [27]:

```
def trainb(model, train_loader, valid_loader, learning_rate, epochs):

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

    criterion = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr= learning_rate)

    if device:
            model = model.cuda()
            criterion = criterion.cuda()

    for epoch_num in range(epochs):

            total_acc_train = 0
            total_loss_train = 0

            for train_input, train_label in tqdm(train_loader):

                train_label = train_label.to(device)
                mask = train_input['attention_mask'].to(device)
                input_id = train_input['input_ids'].squeeze(1).to(device)

                output = model(input_id, mask)

                batch_loss = criterion(output, train_label)
                total_loss_train += batch_loss.item()

                acc = (output.argmax(dim=1) == train_label).sum().item()
                total_acc_train += acc

                model.zero_grad()
                batch_loss.backward()
                optimizer.step()

            total_acc_val = 0
            total_loss_val = 0

            with torch.no_grad():

                for val_input, val_label in val_oader:

                    val_label = val_label.to(device)
                    mask = val_input['attention_mask'].to(device)
                    input_id = val_input['input_ids'].squeeze(1).to(device)

                    output = model(input_id, mask)

                    batch_loss = criterion(output, val_label)
                    total_loss_val += batch_loss.item()

                    acc = (output.argmax(dim=1) == val_label).sum().item()
                    total_acc_val += acc

            print(
                f'Epochs: {epoch_num + 1} | Train Loss: {total_loss_train / len(train_(
#                 return total_acc_val
```

In [28]:

```python
EPOCHS = 5
model = BertClassifier()
LR = 1e-6

trainb(model, train_loader, valid_loader, LR, EPOCHS)
```

```
Some weights of the model checkpoint at bert-base-cased were not used when i
nitializing BertModel: ['cls.seq_relationship.weight', 'cls.seq_relationshi
p.bias', 'cls.predictions.bias', 'cls.predictions.transform.LayerNorm.bias',
'cls.predictions.transform.dense.weight', 'cls.predictions.decoder.weight',
'cls.predictions.transform.dense.bias', 'cls.predictions.transform.LayerNor
m.weight']
- This IS expected if you are initializing BertModel from the checkpoint of
a model trained on another task or with another architecture (e.g. initializ
ing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint
of a model that you expect to be exactly identical (initializing a BertForSe
quenceClassification model from a BertForSequenceClassification model).
  0%|                                                   | 0/118 [00:00<?, ?i
t/s]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [28], in <cell line: 5>()
      2 model = BertClassifier()
      3 LR = 1e-6
----> 5 trainb(model, train_loader, valid_loader, LR, EPOCHS)

Input In [27], in trainb(model, train_loader, valid_loader, learning_rate, e
pochs)
     14 total_acc_train = 0
     15 total_loss_train = 0
---> 17 for train_input, train_label in tqdm(train_loader):
     19     train_label = train_label.to(device)
     20     mask = train_input['attention_mask'].to(device)

ValueError: too many values to unpack (expected 2)
```

In [ ]:

```python

```

In [ ]:

```python

```