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.voca
b.GloVe) >> 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 [82]:

```
# import os

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

In [83]:

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

```
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)](https://pytorch.org/text/stable/datasets.html#id10))

In [84]:

```
from torchtext.data.utils import get_tokenizer
tokenizer = get_tokenizer('spacy', language='en_core_web_sm')
tokens = tokenizer("We are learning torchtext in U.K.!")  #some test
tokens
```

Out[84]:

```
['We', 'are', 'learning', 'torchtext', 'in', 'U.K.', '!']
```

In [85]:

```
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>', '<p
# vocab.set_default_index(vocab["<unk>"])
```

In [86]:

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

from torchtext.datasets import IMDB
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
    #criterion expects float labels
    return torch.tensor(label_list, dtype=torch.float64), pad_sequence(text_list, pa

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 -

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


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

batch_size = 32
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)

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

In [87]:

```
from torchtext.vocab import FastText
fast_vectors = FastText('simple')

fast_embedding = fast_vectors.get_vecs_by_tokens(vocab.get_itos()).to(device)
```

In [88]:

```
input_dim = len(vocab)
hidden_dim = 256
embed_dim = 300
output_dim = 1

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


num_epochs = 2
lr=0.0001
```

In [89]:

```
#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
```

In [90]:

```
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),
        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 [91]:

```
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_

            if self.lstm_type in ['vanilla', 'coupled'] :
                f_t = torch.sigmoid(    h_t @ self.W_f  +  x_t @ self.U_f  +  self.b
                o_t = torch.sigmoid(    h_t @ self.W_o  +  x_t @ self.U_o  +  self.b
                if self.lstm_type == 'vanilla':
                    i_t = torch.sigmoid(    h_t @ self.W_i  +  x_t @ self.U_i  +  se
                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_
            f_t = torch.sigmoid( h_t @ self.W_f + x_t @ self.U_f + c_t @ self.P_
            o_t = torch.sigmoid( h_t @ self.W_o + x_t @ self.U_o + c_t @ self.P_

        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_

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

In [92]:

```
class BiLSTM_model(nn.Module):
    def __init__(self, input_dim: int, embed_dim: int, hidden_dim: int, output_dim:
        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 = 'var
        self.backward_lstm  =  new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'var

        # These should be torch Parameters
        self.W_h = nn.Parameter(torch.Tensor(hidden_dim*self.num_directions, hidden_
        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, in
        output_backward, (hn_backward, cn_backward) = self.backward_lstm(embedded_fl

        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 [93]:

```
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)
criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

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

for epoch in range(num_epochs):
    train_loss, train_acc = train(bilstm, train_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)

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

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

```
Epoch: 01 | Train Loss: 0.082 | Train Acc: 4.63%
         Val. Loss: 0.128 |  Val. Acc: 6.49%
Epoch: 02 | Train Loss: 0.084 | Train Acc: 4.42%
         Val. Loss: 0.125 |  Val. Acc: 6.57%
```

In [94]:

```
def metrics(model, ds, thresh):
  # accuracy  = (TP + TN)  / N
  # precision = TP / (TP + FP)
  # recall    = TP / (TP + FN)
  # F1        = 2 / [(1 / precision) + (1 / recall)]

    tp = 0; tn = 0; fp = 0; fn = 0
    for i in range(len(ds)):
    inpts = ds[i]['predictors']  # dictionary style
    target = ds[i]['sex']    # float32  [0.0] or [1.0]
    # with T.no_grad():
    #   p = model(inpts)       # between 0.0 and 1.0

    # should really avoid 'target == 1.0'
    if target == 1.0 and p > thresh:    # TP
        tp += 1
    elif target == 1.0 and p < thresh:   # FP
        fp += 1
    elif target == 0.0 and p > thresh:   # TN
        tn += 1
    elif target == 0.0 and p < thresh:  # FN
        fn += 1

    N = tp + fp + tn + fn
    if N != len(ds):
    print("FATAL LOGIC ERROR")

    accuracy = (tp + tn) / (N * 1.0)
    precision = (1.0 * tp) / (tp + fp)
    recall = (1.0 * tp) / (tp + fn)
    f1 = 2.0 / ((1.0 / precision) + (1.0 / recall))
    return accuracy, precision, recall, f1
```

In [95]:

```
metrics_LSTM = (bilstm, test_loader, 0.5)
# print(type(metrics_LSTM))
# print(metrics_LSTM)
```

LSTM Attention

In [102]:

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

class LSTM_GAtt(nn.Module):
    def __init__(self, input_dim: int, embed_dim: int, hidden_dim: int, output_dim:
        super().__init__()

        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)

        # let's use pytorch's LSTM
        self.lstm = nn.LSTM(embed_dim,
                            hidden_dim,
                            num_layers=num_layers,
                            bidirectional=bidirectional,
                            dropout=dropout,
                            batch_first=True)

        # Linear Layer for binary classification
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.W = nn.Linear(hidden_dim, hidden_dim)
    def attention_net(self, lstm_output, hn):

        h_t      = hn.unsqueeze(2)
        H_keys   = torch.clone(lstm_output)
        H_values = torch.clone(lstm_output)
        H_query = torch.clone(lstm_output)


        alignment_score    = torch.bmm(H_keys, h_t).squeeze(2) # SHAPE : (bs, seq_len
        # score = torch.bmm(self.W, H_keys)
        # # score  = self.W @  H_keys
        # #alignment_score = (score @  H_query.T).squeeze(2)
        # alignment_score = (torch.bmm(self.W, H_keys).squeeze(2)

        soft_attn_weights = F.softmax(alignment_score, 1) # SHAPE : (bs, seq_len, 1)

        context            = torch.bmm(H_values.transpose(1, 2), soft_attn_weights.ur

        return context

    def forward(self, text, text_lengths):

        embedded = self.embedding(text) # SHAPE : (batch_size, seq_len, embed_dim)

        lstm_output, (hn, cn) = self.lstm(embedded)

        # This is how we concatenate the forward hidden and backward hidden from Pyt
        hn = torch.cat((hn[-2,:,:], hn[-1,:,:]), dim = 1)

        attn_output = self.attention_net(lstm_output, hn)

        return self.fc(attn_output)
```

In [101]:

```
g_attmodel = LSTM_GAtt(input_dim, embed_dim, hidden_dim, output_dim).to(device)
g_attmodel.apply(initialize_weights)
g_attmodel.embedding.weight.data = fast_embedding

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

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

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

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

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

# del g_attmodel
# del optimizer
# del criterion
```

```
Epoch: 01 | Train Loss: 0.065 | Train Acc: 4.66%
         Val. Loss: 0.097 |  Val. Acc: 6.65%
Epoch: 02 | Train Loss: 0.065 | Train Acc: 4.32%
         Val. Loss: 0.096 |  Val. Acc: 7.75%
```

In [103]:

```
def metrics(model, ds, thresh):
  # accuracy  = (TP + TN)  / N
  # precision = TP / (TP + FP)
  # recall    = TP / (TP + FN)
  # F1        = 2 / [(1 / precision) + (1 / recall)]

    tp = 0; tn = 0; fp = 0; fn = 0
    for i in range(len(ds)):
    inpts = ds[i]['predictors']  # dictionary style
    target = ds[i]['sex']    # float32  [0.0] or [1.0]
    # with T.no_grad():
    #   p = model(inpts)       # between 0.0 and 1.0

    # should really avoid 'target == 1.0'
    if target == 1.0 and p > thresh:    # TP
        tp += 1
    elif target == 1.0 and p < thresh:   # FP
        fp += 1
    elif target == 0.0 and p > thresh:   # TN
        tn += 1
    elif target == 0.0 and p < thresh:  # FN
        fn += 1

    N = tp + fp + tn + fn
    if N != len(ds):
    print("FATAL LOGIC ERROR")

    accuracy = (tp + tn) / (N * 1.0)
    precision = (1.0 * tp) / (tp + fp)
    recall = (1.0 * tp) / (tp + fn)
    f1 = 2.0 / ((1.0 / precision) + (1.0 / recall))
    return accuracy, precision, recall, f1
```

In [104]:

```
metrics_LSTM2 = (g_attmodel, test_loader, 0.5)
# print(type(metrics_LSTM))
#print(metrics_LSTM2)
```

BERT

In [105]:

```
!pip install transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.
7/dist-packages (4.16.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.
7/dist-packages (from transformers) (6.0)
Requirement already satisfied: sacremoses in /usr/local/lib/python3.7/
dist-packages (from transformers) (0.0.47)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/
dist-packages (from transformers) (4.62.3)
Requirement already satisfied: tokenizers!=0.11.3,>=0.10.1 in /usr/loc
al/lib/python3.7/dist-packages (from transformers) (0.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.7/di
st-packages (from transformers) (2.23.0)
Requirement already satisfied: importlib-metadata in /usr/local/lib/py
thon3.7/dist-packages (from transformers) (4.11.1)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/pyt
hon3.7/dist-packages (from transformers) (2019.12.20)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.
7/dist-packages (from transformers) (1.21.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/pytho
n3.7/dist-packages (from transformers) (21.3)
Requirement already satisfied: huggingface-hub<1.0,>=0.1.0 in /usr/loc
al/lib/python3.7/dist-packages (from transformers) (0.4.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/di
st-packages (from transformers) (3.6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/loca
l/lib/python3.7/dist-packages (from huggingface-hub<1.0,>=0.1.0->trans
formers) (3.10.0.2)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/
lib/python3.7/dist-packages (from packaging>=20.0->transformers) (3.0.
7)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/d
ist-packages (from importlib-metadata->transformers) (3.7.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1
in /usr/local/lib/python3.7/dist-packages (from requests->transformer
s) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py
thon3.7/dist-packages (from requests->transformers) (2021.10.8)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.
7/dist-packages (from requests->transformers) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/pyt
hon3.7/dist-packages (from requests->transformers) (3.0.4)
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-
packages (from sacremoses->transformers) (7.1.2)
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist
-packages (from sacremoses->transformers) (1.1.0)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-pa
ckages (from sacremoses->transformers) (1.15.0)
```

In [106]:

```
### BERT

from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
from transformers import TrainingArguments


tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)


# tokenized_train_datasets = train_dataset.map(tokenize_function)
# tokenized_test_datasets = test_dataset.map(tokenize_function, b)
```

In [107]:

```
#model_BERT = AutoModelForSequenceClassification.from_pretrained("bert-base-cased",
```

In [108]:

```
#from transformers import AutoTokenizer, AutoModelForSequenceClassification, DistilB


model_name =  "bert-base-cased"
model_BERT = AutoModelForSequenceClassification.from_pretrained(model_name)
#tokenizer = DistilBertTokenizerFast.from_pretrained(model_name)
#tokenizer = AutoTokenizer.from_pretrained(model_BERT)
```

```
Some weights of the model checkpoint at bert-base-cased were not used
when initializing BertForSequenceClassification: ['cls.predictions.bia
s', 'cls.predictions.transform.LayerNorm.bias', 'cls.predictions.trans
form.dense.bias', 'cls.seq_relationship.bias', 'cls.predictions.transf
orm.dense.weight', 'cls.predictions.decoder.weight', 'cls.predictions.
transform.LayerNorm.weight', 'cls.seq_relationship.weight']
- This IS expected if you are initializing BertForSequenceClassificati
on from the checkpoint of a model trained on another task or with anot
her architecture (e.g. initializing a BertForSequenceClassification mo
del from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForSequenceClassifi
cation from the checkpoint of a model that you expect to be exactly id
entical (initializing a BertForSequenceClassification model from a Ber
tForSequenceClassification model).
Some weights of BertForSequenceClassification were not initialized fro
m the model checkpoint at bert-base-cased and are newly initialized:
['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able
to use it for predictions and inference.
```

In [111]:

```
tokenizer = AutoTokenizer.from_pretrained(model_BERT)
```

A%20%20%20%20%20%20(dropout):%20Dropout(p=0.1,%20inplace=False)%0A%20%
20%20%20)%0A%20%20%20%20(encoder):%20BertEncoder(%0A%20%20%20%20%20%20
(layer):%20ModuleList(%0A%20%20%20%20%20%20%20%20(0):%20BertLayer(%0A%
20%20%20%20%20%20%20%20%20%20(attention):%20BertAttention(%0A%20%20%2
0%20%20%20%20%20%20%20%20%20(self):%20BertSelfAttention(%0A%20%20%20%2
0%20%20%20%20%20%20%20%20%20%20(query):%20Linear(in_features=768,%20ou
t_features=768,%20bias=True)%0A%20%20%20%20%20%20%20%20%20%20%20%20%2
0%20(key):%20Linear(in_features=768,%20out_features=768,%20bias=True)%
0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20(value):%20Linear(in_featu
res=768,%20out_features=768,%20bias=True)%0A%20%20%20%20%20%20%20%20%2
0%20%20%20%20%20(dropout):%20Dropout(p=0.1,%20inplace=False)%0A%20%20%
20%20%20%20%20%20%20%20%20%20)%0A%20%20%20%20%20%20%20%20%20%20%20%20
(output):%20BertSelfOutput(%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%
20(dense):%20Linear(in_features=768,%20out_features=768,%20bias=True)%
0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20(LayerNorm):%20LayerNorm
((768,),%20eps=1e-12,%20elementwise_affine=True)%0A%20%20%20%20%20%20%
20%20%20%20%20%20%20%20(dropout):%20Dropout(p=0.1,%20inplace=False)%0
A%20%20%20%20%20%20%20%20%20%20%20%20)%0A%20%20%20%20%20%20%20%20%20%2
0)%0A%20%20%20%20%20%20%20%20%20%20(intermediate):%20BertIntermediate
(%0A%20%20%20%20%20%20%20%20%20%20%20%20(dense):%20Linear(in_features=