Installing Dependencies

```
!pip install transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: transformers in /usr/local/lib/python3.7/dist-packages (4.20.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.1.0 in /usr/local/lib/python3.7/dist-packages (from transformer
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/dist-packages (from transformers) (4.12.
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.7/dist-packages (from transformers) (6.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages (from transformers) (3.7.1)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.7/dist-packages (from transformers) (1.21.6)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/dist-packages (from transformers) (21.3)
Requirement already satisfied: tokenizers!=0.11.3,<0.13,>=0.11.1 in /usr/local/lib/python3.7/dist-packages (from trans
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from transformers) (2.23.0)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.7/dist-packages (from transformers) (2022.6
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-packages (from transformers) (4.64.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.7/dist-packages (from huggingface-
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.7/dist-packages (from packaging>=20.
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata->transform
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests->transformers) (2
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests->transformer
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests->transforme
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from
```

```
import transformers
from transformers import BertModel, BertTokenizer, AdamW, get_linear_schedule_with_warmup
import torch
import gc
gc.collect()
torch.cuda.empty_cache()

import numpy as np
import pandas as pd
import seaborn as sns
from pylab import rcParams
```

```
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from collections import defaultdict
from textwrap import wrap

from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F

%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F00FF"]

sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))

rcParams['figure.figsize'] = 12, 8

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

```
device(type='cuda', index=0)
```

Read the Data

```
df = pd.read_csv("/content/Pelabelan1.csv")
df.head()
```

	category	review content	lemma	rating	Subjectivity	Polarity	TextBlob	Vader Sentiment	vaderSentiment
0	Headsets	This gaming headset ticks all the boxes # look...	['game', 'headset', 'tick', 'box', 'look', 'gr...	5	0.583333	0.305556	Positive	0.8720	Positive
1	Headsets	Easy setup, rated for 6 hours battery but mine...	['easy', 'setup', 'rat', 'hours', 'battery', '...	3	0.546289	0.203782	Positive	0.9657	Positive

```
df.shape
```

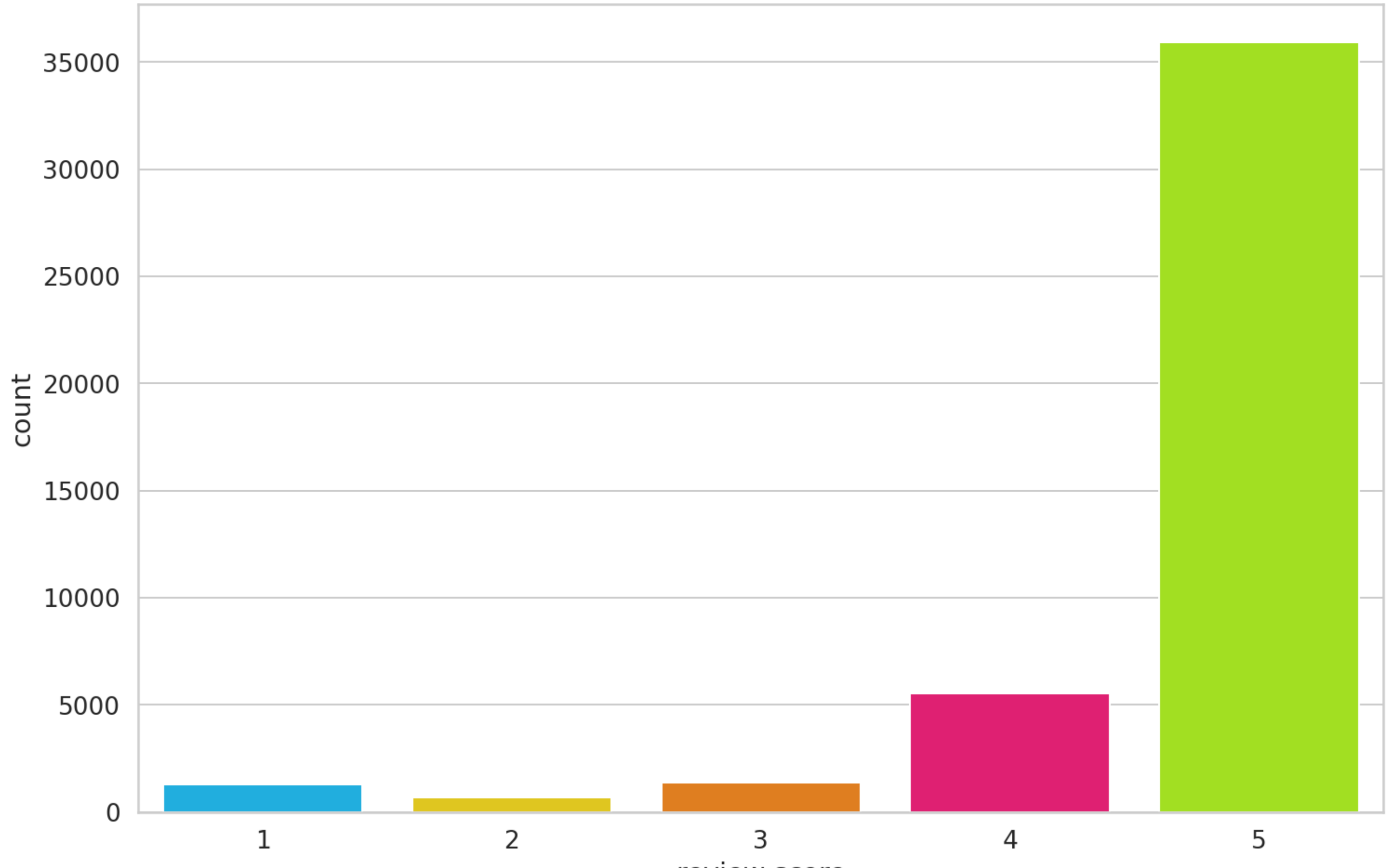
```
(44756, 9)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 44756 entries, 0 to 44755
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   category         44756 non-null  object
 1   review content   44756 non-null  object
 2   lemma            44756 non-null  object
 3   rating           44756 non-null  int64
 4   Subjectivity     44756 non-null  float64
 5   Polarity         44756 non-null  float64
 6   TextBlob         44756 non-null  object
 7   Vader Sentiment  44756 non-null  float64
 8   vaderSentiment   44756 non-null  object
dtypes: float64(3), int64(1), object(5)
memory usage: 3.1+ MB
```

```
sns.countplot(df.rating)
plt.xlabel('review score');
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as a keyw
  FutureWarning
```



We have a balanced dataset with respect to reviews (as we scraped it in a balanced way)

```
def group_sentiment(rating):
  #rating = int(rating)
  if rating <= 2:
    return 0         # Negetive sentiment
  elif rating == 3:
    return 1         # Neutral Sentiment
  else:
    return 2         # positive Sentiment
```

```
df['TextBlob'] = df.rating.apply(group_sentiment)
```

```
class_names = ['negative', 'neutral', 'positive']
```

```
ax = sns.countplot(df.TextBlob)
plt.xlabel('review sentiment')
ax.set_xticklabels(class_names);
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as a keyw
  FutureWarning
```

Data Preprocessing

Machine Learning models don't work with raw text. We need to convert text to numbers (of some sort). BERT requires even more attention. Here are the requirements:

- Add special tokens to separate sentences and do classification
- Pass sequences of constant length (introduce padding)
- Create array of 0s (pad token) and 1s (real token) called attention mask

The Transformers library provides a wide variety of Transformer models (including BERT). It also includes prebuilt tokenizers that solves most of the load required for pre-processing

```
PRE_TRAINED_MODEL_NAME = 'bert-base-cased'
```

Let's load a pre-trained BertTokenizer:

```
tokenizer = BertTokenizer.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

Downloading: 100%	208k/208k [00:00<00:00, 1.72MB/s]
Downloading: 100%	29.0/29.0 [00:00<00:00, 849B/s]
Downloading: 100%	570/570 [00:00<00:00, 15.1kB/s]

The requirements of special tokens which indicate seperation between sentences is taken care by the tokenizer provided by the huggingface. BERT was trained for question and answering task but we are using it to train with a sequence of sentences as input. We require sequences of equal length which is done by padding tokens of zero meaning without loss of generality to the end of sentences.

We also need to pass attention mask which is basically passing a value of 1 to all the tokens which have meaning and 0 to padding tokens.

We'll use this simple text to understand the tokenization process:

```
sample_txt = 'Best place that I have visited? Iceland was the most beautiful and I consider myself lucky to have visited Ice
#sample_txt = 'When was I last outside? I am stuck at home for 2 weeks.'
```

```
tokens = tokenizer.tokenize(sample_txt)
token_ids = tokenizer.convert_tokens_to_ids(tokens)

print(f'  Sentence: {sample_txt}')
print(f'\n  Tokens: {tokens}')
print(f'\n  Token IDs: {token_ids}')   # Each token has a an unique ID for the model to unserstand what we are referring to.
```

```
  Sentence: Best place that I have visited? Iceland was the most beautiful and I consider myself lucky to have visited

  Tokens: ['Best', 'place', 'that', 'I', 'have', 'visited', '?', 'Iceland', 'was', 'the', 'most', 'beautiful', 'and',

  Token IDs: [1798, 1282, 1115, 146, 1138, 3891, 136, 10271, 1108, 1103, 1211, 2712, 1105, 146, 4615, 1991, 6918, 1106
```

```
len(tokens)
```

```
27
```

Special Tokens

[SEP] - marker for ending of a sentence

```
tokenizer.sep_token, tokenizer.sep_token_id
```

```
('[SEP]', 102)
```

[CLS] - we must add this token to the start of each sentence, so BERT knows we're doing classification

```
tokenizer.cls_token, tokenizer.cls_token_id
```

```
('[CLS]', 101)
```

There is also a special token for padding:

```
tokenizer.pad_token, tokenizer.pad_token_id
```

```
('[PAD]', 0)
```

BERT understands tokens that were in the training set. Everything else can be encoded using the [UNK] (unknown) token:

```
tokenizer.unk_token, tokenizer.unk_token_id
```

```
('[UNK]', 100)
```

All of the above work can be done using the encode_plus() method

```
encoding_test = tokenizer.encode_plus(
  sample_txt,
  max_length=32,           # sequence length
  add_special_tokens=True, # Add '[CLS]' and '[SEP]'
  return_token_type_ids=False,
  pad_to_max_length=True,
  return_attention_mask=True,
  return_tensors='pt',  # Return PyTorch tensors(use tf for tensorflow and keras)
)
```

```
Truncation was not explicitly activated but `max_length` is provided a specific value, please use `truncation=True` to
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
```

```
encoding_test.keys()
```

```
dict_keys(['input_ids', 'attention_mask'])
```

```
print(' length of the first sequence is :  ', len(encoding_test['input_ids'][0]))
print('\n The input id\'s are : \n', encoding_test['input_ids'][0])
print('\n The attention mask generated is : ', encoding_test['attention_mask'][0])
```

```
 length of the first sequence is :   32

 The input id's are :
 tensor([  101,  1798,  1282,  1115,   146,  1138,  3891,   136, 10271,  1108,
         1103,  1211,  2712,  1105,   146,  4615,  1991,  6918,  1106,  1138,
         3891, 10271,  1120,  1216,  1126,  1346,  1425,   119,   102,     0,
            0,     0])

 The attention mask generated is :  tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 0, 0])
```

We can inverse the tokenization to have a look at the special tokens:

```
tokenizer.convert_ids_to_tokens(encoding_test['input_ids'].flatten())
```

```
['[CLS]',
 'Best',
 'place',
 'that',
 'I',
 'have',
 'visited',
 '?',
 'Iceland',
```

```
 'was',
 'the',
 'most',
 'beautiful',
 'and',
 'I',
 'consider',
 'myself',
 'lucky',
 'to',
 'have',
 'visited',
 'Iceland',
 'at',
 'such',
 'an',
 'early',
 'age',
 '.',
 '[SEP]',
 '[PAD]',
 '[PAD]',
 '[PAD]']
```

Choosing Sequence Length

Check if there are any nan in the content column

```
df.loc[df['review content'].isnull()]
```

category	review content	lemma	rating	Subjectivity	Polarity	TextBlob	Vader Sentiment	vaderSentiment

```
df = df[df['review content'].notna()]
df.head()
```

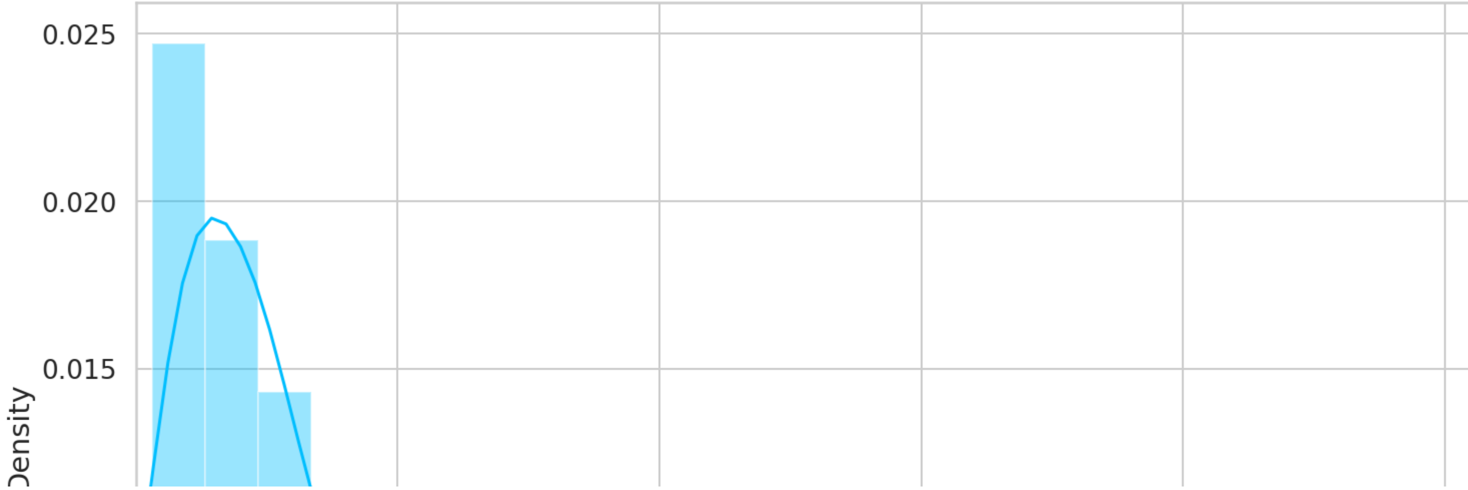
	category	review content	lemma	rating	Subjectivity	Polarity	TextBlob	Vader Sentiment	vaderSentiment
0	Headsets	This gaming headset ticks all the boxes # look...	['game', 'headset', 'tick', 'box', 'look', 'gr...	5	0.583333	0.305556	2	0.8720	Positive
1	Headsets	Easy setup, rated for 6 hours battery but mine...	['easy', 'setup', 'rat', 'hours', 'battery', '...	3	0.546289	0.203782	1	0.9657	Positive

BERT works with fixed-length sequences. We'll use a simple strategy to choose the max length. Let's store the token length of each review:

```
token_lens = []
for text in df['review content']:
    tokens_df = tokenizer.encode(text, max_length=512)   # Max possible length for the BERT model.
    token_lens.append(len(tokens_df))
```

```
sns.distplot(token_lens)
plt.xlim([0, 256]);
plt.xlabel('Token count');
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated functi
  warnings.warn(msg, FutureWarning)
```



Most of the reviews seem to contain less than 128 tokens, but we'll be on the safe side and choose a maximum length of 160.

```
MAX_LEN = 160
```

We have all building blocks required to create a PyTorch dataset. Let's use the same class:

```
class GPReviewDataset(Dataset):

  def __init__(self, reviews, targets, tokenizer, max_len):
    self.reviews = reviews          # Reviews is content column.
    self.targets = targets          # Target is the sentiment column.
    self.tokenizer = tokenizer      # Tokenizer is the BERT_Tokanizer.
    self.max_len = max_len          # max_length of each sequence.

  def __len__(self):
    return len(self.reviews)        # Len of each review.

  def __getitem__(self, item):
    review = str(self.reviews[item])    # returns the string of reviews at the index = 'items'
    target = self.targets[item]         # returns the string of targets at the index = 'items'
```

```
    encoding = self.tokenizer.encode_plus(
      review,
      add_special_tokens=True,
      max_length=self.max_len,
      return_token_type_ids=False,
      pad_to_max_length=True,
      return_attention_mask=True,
      return_tensors='pt',
    )

    return {
      'review_text': review,
      'input_ids': encoding['input_ids'].flatten(),
      'attention_mask': encoding['attention_mask'].flatten(),
      'targets': torch.tensor(target, dtype=torch.long)            # dictionary containing all the features is returned.
    }
```

The tokenizer is doing most of the heavy lifting for us. We also return the review texts, so it'll be easier to evaluate the predictions from our model. Let's split the data:

Splitting into train and validation sets

```
df_train, df_test = train_test_split(df, test_size=0.4, random_state=RANDOM_SEED)
df_val, df_test = train_test_split(df_test, test_size=0.5, random_state=RANDOM_SEED)
```

```
df_train.shape, df_val.shape, df_test.shape
```

```
((26853, 9), (8951, 9), (8952, 9))
```

Create data loaders for to feed as input to our model. The below function does that.

```
def create_data_loader(df, tokenizer, max_len, batch_size):
  ds = GPReviewDataset(
    reviews=df['review content'].values,
    targets=df['TextBlob'].values,
    tokenizer=tokenizer,
    max_len=max_len
  )                                  # Dataset would be created which can be used to create and return dataloader.

  return DataLoader(
    ds,
    batch_size=batch_size,
    #num_workers=4
  )
```

```
BATCH_SIZE = 8

train_data_loader = create_data_loader(df_train, tokenizer, MAX_LEN, BATCH_SIZE)
val_data_loader = create_data_loader(df_val, tokenizer, MAX_LEN, BATCH_SIZE)
test_data_loader = create_data_loader(df_test, tokenizer, MAX_LEN, BATCH_SIZE)
```

Let's have a look at an example batch from our training data loader:

```
data = next(iter(train_data_loader))
data.keys()
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
dict_keys(['review_text', 'input_ids', 'attention_mask', 'targets'])
```

```
print(data['input_ids'].shape)
print(data['attention_mask'].shape)
print(data['targets'].shape)
```

```
torch.Size([8, 160])
```

```
torch.Size([8, 160])
torch.Size([8])
```

Sentiment Classification with BERT and Hugging Face

There are a lot of helpers that make using BERT easy with the Transformers library. Depending on the task we might use "BertForSequenceClassification", "BertForQuestionAnswering" or something else.

We'll use the basic BertModel and build our sentiment classifier on top of it. Let's load the model:

```
bert_model = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

Downloading: 100% 416M/416M [00:07<00:00, 57.4MB/s]

```
Some weights of the model checkpoint at bert-base-cased were not used when initializing BertModel: ['cls.seq_relations
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with an
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly
```

```
model_test = bert_model(
  input_ids=encoding_test['input_ids'],
  attention_mask=encoding_test['attention_mask']
)
model_test.keys()
```

```
odict_keys(['last_hidden_state', 'pooler_output'])
```

The "last_hidden_state" is a sequence of hidden states of the last layer of the model. Obtaining the "pooled_output" is done by applying the BertPooler which basically applies the tanh function to pool all the outputs.

```
last_hidden_state=model_test['last_hidden_state']
```

```
pooled_output=model_test['pooler_output']

last_hidden_state.shape
```

```
torch.Size([1, 32, 768])
```

We have the hidden state for each of our 32 tokens (the length of our example sequence) and 768 is the number of hidden units in the feedforward-networks. We can verify that by checking the config:

```
bert_model.config.hidden_size
```

```
768
```

We can think of the pooled_output as a summary of the content, according to BERT. Let's look at the shape of the output:

```
pooled_output.shape
```

```
torch.Size([1, 768])
```

We can use all this knowledge to create a sentiment classifier that uses the BERT model:

```
class SentimentClassifier(nn.Module):

  def __init__(self, n_classes):
    super(SentimentClassifier, self).__init__()
    self.bert = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
    self.drop = nn.Dropout(p=0.3)                                 ## For regularization with dropout probability 0.3.
    self.out = nn.Linear(self.bert.config.hidden_size, n_classes) ## append an Output fully connected layer representing the

  def forward(self, input_ids, attention_mask):
    returned = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask
    )
```

```
    pooled_output = returned["pooler_output"]
    output = self.drop(pooled_output)
    return self.out(output)
```

The classifier delegates most of the work to the BertModel. We use a dropout layer for some regularization and a fully-connected layer for our output. We're returning the raw output of the last layer since that is required for the cross-entropy loss function in PyTorch to work.

This should work like any other PyTorch model. Create an instance and move it to the GPU:

```
model = SentimentClassifier(len(class_names))
model = model.to(device)
```

```
Some weights of the model checkpoint at bert-base-cased were not used when initializing BertModel: ['cls.seq_relations
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with an
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly
```

We'll move the example batch of our training data created above using dataloader to the GPU:

```
input_ids = data['input_ids'].to(device)
attention_mask = data['attention_mask'].to(device)

print(input_ids.shape)      # batch size x seq length
print(attention_mask.shape) # batch size x seq length
```

```
torch.Size([8, 160])
torch.Size([8, 160])
```

To get the predicted probabilities from our trained model, we'll apply the softmax function to the output obtained from the output layer:

```
F.softmax(model(input_ids, attention_mask), dim=1)
```

```
tensor([[0.2423, 0.4402, 0.3176],
        [0.2371, 0.3312, 0.4317],
        [0.4197, 0.2137, 0.3666],
        [0.2275, 0.3245, 0.4480],
        [0.5731, 0.2144, 0.2125],
        [0.2461, 0.3844, 0.3695],
        [0.2512, 0.3166, 0.4323],
        [0.3581, 0.2461, 0.3958]], device='cuda:0', grad_fn=<SoftmaxBackward0>)
```

Training the model

To reproduce the training procedure from the BERT paper, we'll use the AdamW optimizer provided by Hugging Face. It corrects weight decay, so it's similar to the original paper. We'll also use a linear scheduler with no warmup steps:

```
EPOCHS = 10

optimizer = AdamW(model.parameters(), lr=2e-5, correct_bias=False)
total_steps = len(train_data_loader) * EPOCHS    # Number of batches * Epochs (Required for the scheduler.)

scheduler = get_linear_schedule_with_warmup(
  optimizer,
  num_warmup_steps=0,     # Recommended in the BERT paper.
  num_training_steps=total_steps
)

loss_fn = nn.CrossEntropyLoss().to(device)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:310: FutureWarning: This implementation of AdamW i
  FutureWarning,
```

The BERT authors have some recommendations for fine-tuning:

- Batch size: 16, 32

- Learning rate (Adam): 5e-5, 3e-5, 2e-5
- Number of epochs: 2, 3, 4

Except for the number of epochs recommendation We'll stick with the rest. Increasing the batch size reduces the training time significantly, but gives lower accuracy.

Helper function for training our model for one epoch:

```
def train_epoch(
  model,
  data_loader,
  loss_fn,
  optimizer,
  device,
  scheduler,
  n_examples
):
  model = model.train()    # To make sure that the droupout and normalization is enabled during the training.

  losses = []
  correct_predictions = 0

  for d in data_loader:
    input_ids = d["input_ids"].to(device)
    attention_mask = d["attention_mask"].to(device)
    targets = d["targets"].to(device)

    outputs = model(
      input_ids=input_ids,
      attention_mask=attention_mask
    )

    max_prob, preds = torch.max(outputs, dim=1)    # Returns 2 tensors, one with max_probability and another with the respec
    loss = loss_fn(outputs, targets)
```

```
    correct_predictions += torch.sum(preds == targets)
    losses.append(loss.item())

    loss.backward()     # Back_Propogation
    nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)  # Recommended by the BERT paper to clip the gradients to avc
    optimizer.step()
    scheduler.step()
    optimizer.zero_grad()

  return correct_predictions.double() / n_examples, np.mean(losses)     # Return the mean loss and the ratio of correct prec
```

Training the model is similar to training a deep neural network, except for two things. The scheduler gets called every time a batch is fed to the model. We're avoiding exploding gradients by clipping the gradients of the model using clip_grad_norm

Helper function to evaluate the model on a given data loader:

```
def eval_model(model, data_loader, loss_fn, device, n_examples):
  model = model.eval()        # To make sure that the droupout and normalization is disabled during the training.

  losses = []
  correct_predictions = 0

  with torch.no_grad():        # Back propogation is not required. Torch would perform faster.
    for d in data_loader:
      input_ids = d["input_ids"].to(device)
      attention_mask = d["attention_mask"].to(device)
      targets = d["targets"].to(device)

      outputs = model(
        input_ids=input_ids,
        attention_mask=attention_mask
      )
      max_prob, preds = torch.max(outputs, dim=1)

      loss = loss_fn(outputs, targets)
```

```
    correct_predictions += torch.sum(preds == targets)
    losses.append(loss.item())

  return correct_predictions.double() / n_examples, np.mean(losses)
```

Using these two helper functions, we can write our training loop. We'll also store the training history:

```
%%time

history = defaultdict(list)          # Similar to Keras library saves history
best_accuracy = 0

for epoch in range(EPOCHS):

  print(f'Epoch {epoch + 1}/{EPOCHS}')
  print('-' * 10)

  train_acc, train_loss = train_epoch(
    model,
    train_data_loader,
    loss_fn,
    optimizer,
    device,
    scheduler,
    len(df_train)
  )

  print(f'Train loss {train_loss} accuracy {train_acc}')

  val_acc, val_loss = eval_model(
    model,
    val_data_loader,
    loss_fn,
    device,
    len(df_val)
  )
```

```
print(f'Val   loss {val_loss} accuracy {val_acc}')
print()

history['train_acc'].append(train_acc)
history['train_loss'].append(train_loss)
history['val_acc'].append(val_acc)
history['val_loss'].append(val_loss)

if val_acc > best_accuracy:
  torch.save(model.state_dict(), 'best_model_state.bin')
  best_accuracy = val_acc
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
Epoch 1/10
----------
Train loss 0.24020488611953753 accuracy 0.9400439429486462
Val   loss 0.22318187647977603 accuracy 0.9406770193274494

Epoch 2/10
----------
Train loss 0.17003985258229332 accuracy 0.9570252858153651
Val   loss 0.3064608826845107 accuracy 0.9386660708300748

Epoch 3/10
----------
Train loss 0.12968090553569922 accuracy 0.9693144155215432
Val   loss 0.3522151932940055 accuracy 0.9400067031616579

Epoch 4/10
----------
Train loss 0.09378056590418693 accuracy 0.979443637582393
Val   loss 0.3851617049007931 accuracy 0.936543402971735

Epoch 5/10
----------
Train loss 0.07196425489593598 accuracy 0.9858861207313894
Val   loss 0.41133513248288384 accuracy 0.9331918221427773

Epoch 6/10
```

```
----------
Train loss 0.05079280420447268 accuracy 0.9904666145309649
Val   loss 0.4279829605774441 accuracy 0.9393363869958664

Epoch 7/10
----------
Train loss 0.03749320707015525 accuracy 0.9931851189811194
Val   loss 0.4699776120980526 accuracy 0.9358730868059435

Epoch 8/10
----------
Train loss 0.025816966835583315 accuracy 0.9950843481175287
Val   loss 0.5307574826412504 accuracy 0.9357613674449782

Epoch 9/10
----------
Train loss 0.01869010315812366 accuracy 0.9965739395970654
Val   loss 0.5549460319121583 accuracy 0.9354262093620824

Epoch 10/10
----------
Train loss 0.013058530798282918 accuracy 0.9973187353368338
Val   loss 0.555978277890803 accuracy 0.9369902804155961

CPU times: user 1h 46min 15s, sys: 40min 43s, total: 2h 26min 59s
Wall time: 2h 27min 30s
```

Note that we're storing the state of the best model, indicated by the highest validation accuracy.

```
plt.plot(history['train_acc'], label='train accuracy')
plt.plot(history['val_acc'], label='validation accuracy')

plt.title('Training history')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.ylim([0, 1]);
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/usr/local/lib/python3.7/dist-packages/matplotlib/cbook/__init__.py in index_of(y)
   1626     try:
-> 1627         return y.index.values, y.values
   1628     except AttributeError:

AttributeError: 'builtin_function_or_method' object has no attribute 'values'

During handling of the above exception, another exception occurred:

TypeError                                 Traceback (most recent call last)
```

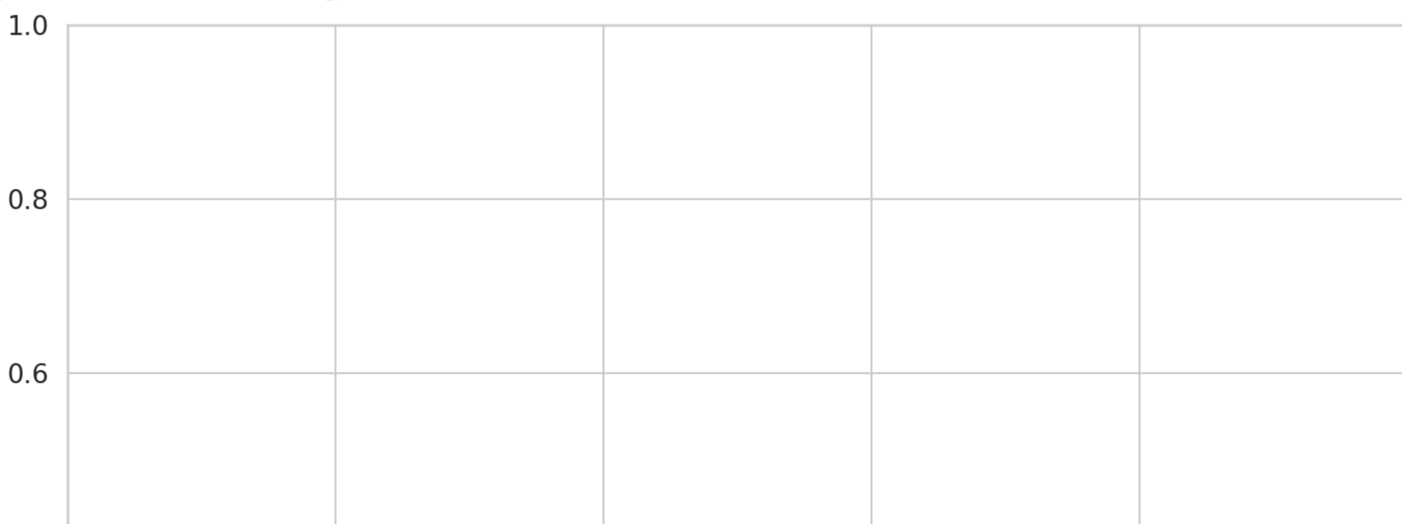
8 frames

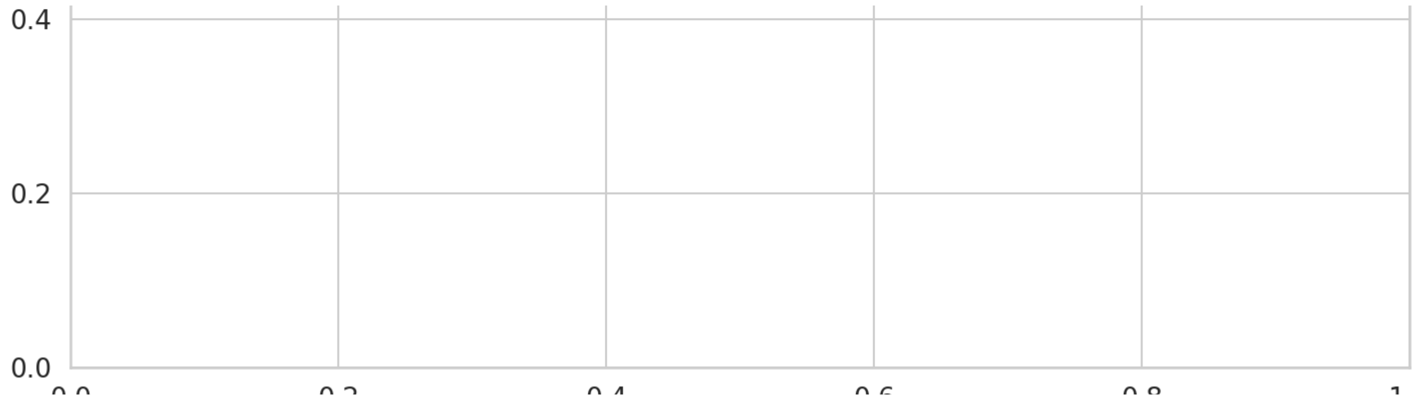
```
<__array_function__ internals> in atleast_1d(*args, **kwargs)

/usr/local/lib/python3.7/dist-packages/torch/_tensor.py in __array__(self, dtype)
    755             return handle_torch_function(Tensor.__array__, (self,), self, dtype=dtype)
    756         if dtype is None:
--> 757             return self.numpy()
    758         else:
    759             return self.numpy().astype(dtype, copy=False)

TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.
```

SEARCH STACK OVERFLOW





Evaluation

So how good is our model on predicting sentiment? Let's start by calculating the accuracy on the test data:

```
test_acc, _ = eval_model(
  model,
  test_data_loader,
  loss_fn,
  device,
  len(df_test)
)

test_acc.item()
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
0.9440348525469169
```

The accuracy is about 1% lower on the test set. Our model seems to generalize well.

We'll define a helper function to get the predictions from our model:

```
def get_predictions(model, data_loader):
  model = model.eval()
```

```
  review_texts = []
  predictions = []
  prediction_probs = []
  real_values = []

  with torch.no_grad():
    for d in data_loader:

      texts = d["review_text"]
      input_ids = d["input_ids"].to(device)
      attention_mask = d["attention_mask"].to(device)
      targets = d["targets"].to(device)

      outputs = model(
        input_ids=input_ids,
        attention_mask=attention_mask
      )
      _, preds = torch.max(outputs, dim=1)

      probs = F.softmax(outputs, dim=1)

      review_texts.extend(texts)
      predictions.extend(preds)
      prediction_probs.extend(probs)
      real_values.extend(targets)

  predictions = torch.stack(predictions).cpu()
  prediction_probs = torch.stack(prediction_probs).cpu()
  real_values = torch.stack(real_values).cpu()
  return review_texts, predictions, prediction_probs, real_values
```

This is similar to the evaluation function, except that we're storing the text of the reviews and the predicted probabilities (by applying the softmax on the model outputs):

```
y_review_texts, y_pred, y_pred_probs, y_test = get_predictions(
  model,
```

```
  test_data_loader
)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
```

Let's have a look at the classification report

```
print(classification_report(y_test, y_pred, target_names=class_names))
```

```
              precision    recall  f1-score   support

    negative       0.72      0.55      0.63       378
     neutral       0.31      0.29      0.30       254
    positive       0.97      0.98      0.98      8320

    accuracy                           0.94      8952
   macro avg       0.67      0.61      0.63      8952
weighted avg       0.94      0.94      0.94      8952
```

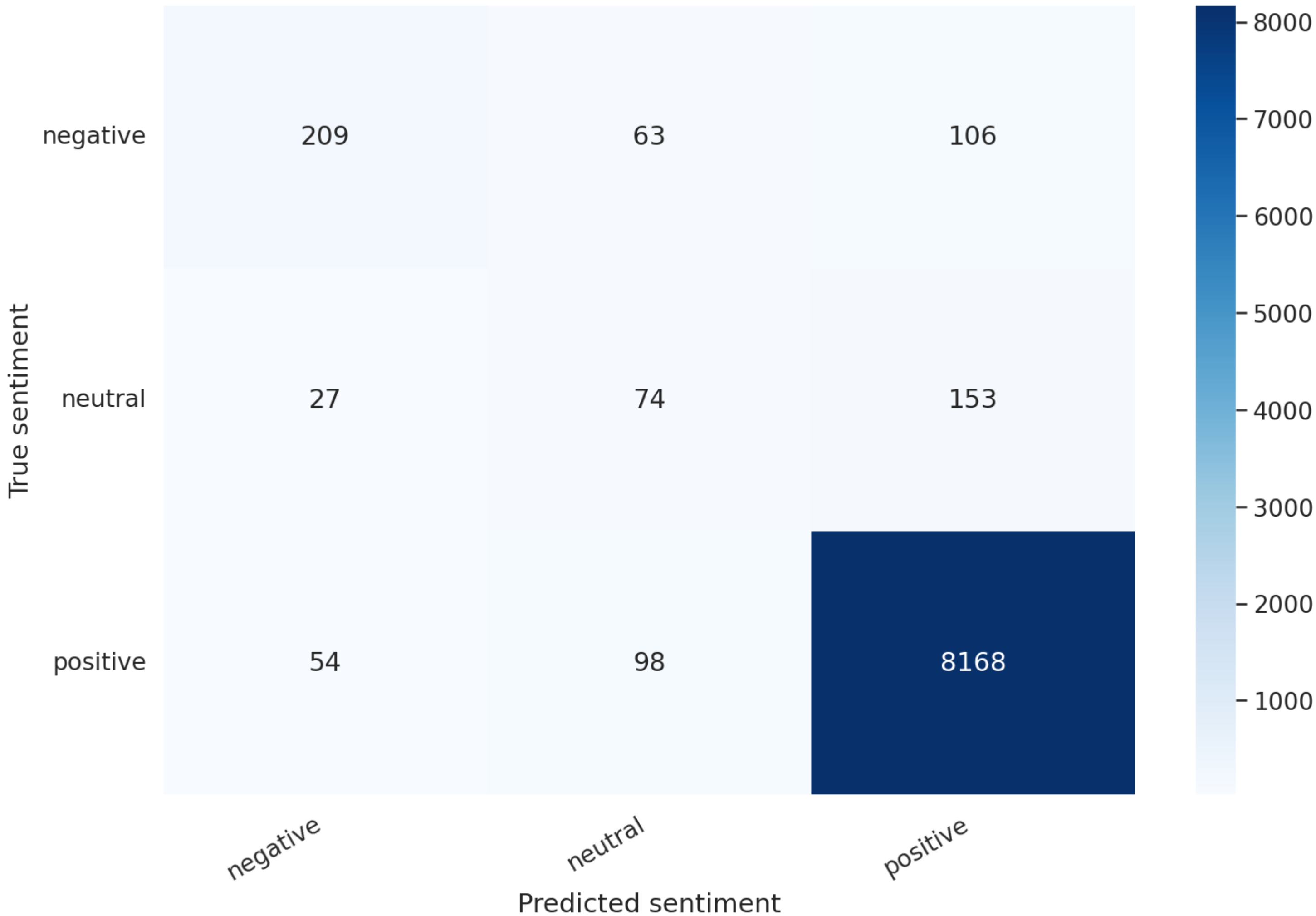
Looks like it is really hard to classify neutral (3 stars) reviews. And I can tell you from experience, looking at many reviews, those are hard to classify.

We'll continue with the confusion matrix:

```
def show_confusion_matrix(confusion_matrix):
  hmap = sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30, ha='right')
  plt.ylabel('True sentiment')
  plt.xlabel('Predicted sentiment');

cm = confusion_matrix(y_test, y_pred)
```

```
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
show_confusion_matrix(df_cm)
```



This confirms that our model is having difficulty classifying neutral reviews. It mistakes those for negative and positive at a roughly equal frequency.

That's a good overview of the performance of our model. But let's have a look at an example from our test data:

```
idx = 2
```

```
review_text = y_review_texts[idx]
true_sentiment = y_test[idx]
pred_df = pd.DataFrame({
  'class_names': class_names,
  'values': y_pred_probs[idx]
})
```

```
print("\n".join(wrap(review_text)))
print()
print(f'True sentiment: {class_names[true_sentiment]}')
```

```
Great balance and overall weight.

True sentiment: positive
```

Now we can look at the confidence of each sentiment of our model:

```
sns.barplot(x='values', y='class_names', data=pred_df, orient='h')
plt.ylabel('sentiment')
plt.xlabel('probability')
plt.xlim([0, 1]);
```

Predicting on Raw Text

Let's use our model to predict the sentiment of some raw text:

```
review_text = "I love completing my todos! Best app ever!!!"
```

We have to use the tokenizer to encode the text:

```
encoded_review = tokenizer.encode_plus(
  review_text,
  max_length=MAX_LEN,
  add_special_tokens=True,
  return_token_type_ids=False,
  pad_to_max_length=True,
  return_attention_mask=True,
  return_tensors='pt',
)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
```

Let's get the predictions from our model:

```
input_ids = encoded_review['input_ids'].to(device)
attention_mask = encoded_review['attention_mask'].to(device)

output = model(input_ids, attention_mask)
_, prediction = torch.max(output, dim=1)

print(f'Review text: {review_text}')
print(f'Sentiment  : {class_names[prediction]}')
```

```
Review text: I love completing my todos! Best app ever!!!
Sentiment  : positive
```

```
review_text·=·"this·is·bad!!!"
```

```
encoded_review = tokenizer.encode_plus(
  review_text,
  max_length=MAX_LEN,
  add_special_tokens=True,
  return_token_type_ids=False,
  pad_to_max_length=True,
  return_attention_mask=True,
  return_tensors='pt',
)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
```

```
input_ids·=·encoded_review['input_ids'].to(device)
attention_mask·=·encoded_review['attention_mask'].to(device)

output·=·model(input_ids,·attention_mask)
_,·prediction·=·torch.max(output,·dim=1)

print(f'Review·text:·{review_text}')
```

```
print(f'Sentiment··:·{class_names[prediction]}')
```

```
Review text: this is bad!!!
Sentiment  : negative
```

```
review_text = "inpity !!!"
```

```
encoded_review = tokenizer.encode_plus(
  review_text,
  max_length=MAX_LEN,
  add_special_tokens=True,
  return_token_type_ids=False,
  pad_to_max_length=True,
  return_attention_mask=True,
  return_tensors='pt',
)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_utils_base.py:2307: FutureWarning: The `pad_to_max_le
  FutureWarning,
```

```
input_ids = encoded_review['input_ids'].to(device)
attention_mask = encoded_review['attention_mask'].to(device)

output = model(input_ids, attention_mask)
_, prediction = torch.max(output, dim=1)

print(f'Review text: {review_text}')
print(f'Sentiment  : {class_names[prediction]}')
```

```
Review text: inpity !!!
Sentiment  : positive
```

0s completed at 16:20