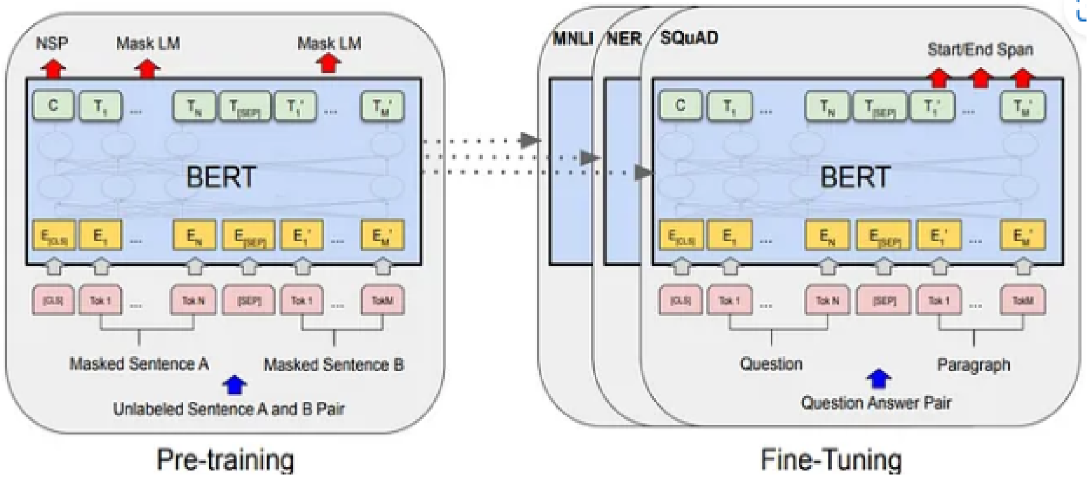
Bhuvi Ghosh
60009210191

Experiment No 7
Aim: Fine tuning BERT model to perform Natural Language Processing task.

Theory:

BERT

BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers and is a language representation model by Google. It uses two steps, pre-training and fine-tuning, to create state-of-the-art models for a wide range of tasks. Its distinctive feature is the unified architecture across different downstream tasks — what these are, we will discuss soon. That means that the same pre-trained model can be fine-tuned for a variety of final tasks that might not be similar to the task model was trained on and give close to state-of-the-art results.



BERT Architecture

BERT has to differ Architecture BERT Base and BERT Large
BERT Base: L=12, H=768, A=12.
Total Parameters=110M!
BERT Large: L=24, H=1024, A=16.
Total Parameters=340M!!
L = Number of layers (i.e., #Transformer encoder blocks in the stack).
H = Hidden size (i.e. the size of *q, k* and *v* vectors).
A = Number of attention heads.

Pre-training BERT

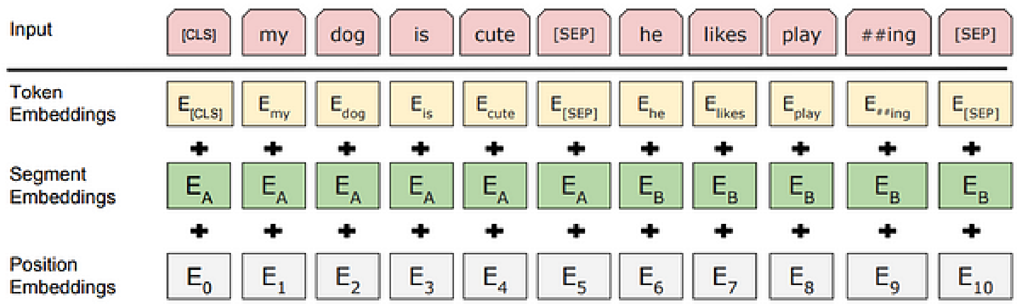
The BERT model is trained on the following two unsupervised tasks.
1. Masked Language Model (MLM)

This task enables the deep bidirectional learning aspect of the model. In this task, some percentage of the input tokens are masked (Replaced with [*MASK*] token) at random and the model tries to predict these masked tokens — not the entire input sequence. The predicted tokens from the model are then fed into an output softmax over the vocabulary to get the final output words.
This, however creates a mismatch between the pre-training and fine-tuning tasks because the latter does not involve predicting masked words in most of the downstream tasks. This is mitigated by a subtle twist in how we mask the input tokens.
Approximately 15% of the words are masked while training, but all of the masked words are not replaced by the [*MASK*] token.
80% of the time with [*MASK*] tokens.
10% of the time with a random tokens.
10% of the time with the unchanged input tokens that were being masked.

2. Next Sentence Prediction (NSP)

The LM doesn't directly capture the relationship between two sentences which is relevant in many downstream tasks such as Question Answering (QA) and Natural Language Inference (NLI). The model is taught sentence relationships by training on binarized NSP task.
In this task, two sentences — A and B — are chosen for pre-training.
50% of the time B is the actual next sentence that follows A.
50% of the time B is a random sentence from the corpus.
Training — Inputs and Outputs.

The model is trained on both above mentioned tasks simultaneously. This is made possible by clever usage of inputs and outputs.
Inputs


Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

The input representation for BERT

The model needs to take input for both a single sentence or two sentences packed together unambiguously in one token sequence. Authors note that a "sentence" can be an arbitrary span of contiguous text, rather than an actual linguistic sentence. A [SEP] token is used to separate two sentences as well as a using a learnt segment embedding indicating a token as a part of segment A or B.

Problem #1**:* All the inputs are fed in one step — as opposed to RNNs in which inputs are fed sequentially, the model is ***not able to preserve the ordering of the input tokens. The order of words in every language is significant, both semantically and syntactically.

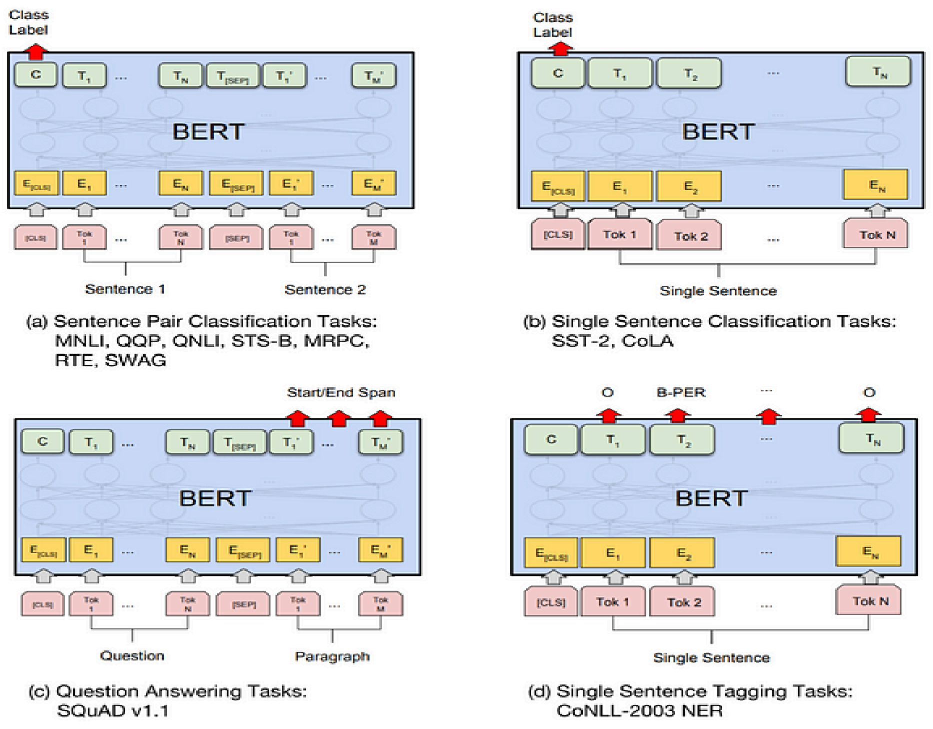
Problem #2**:* In order to perform Next Sentence Prediction task properly we need to be able to ***distinguish between sentences A and B. Fixing the lengths of sentences can be too restrictive and a potential bottleneck for various downstream tasks.
Both of these problems are solved by adding embeddings containing the required information to our original tokens and using the result as the input to our BERT model. The following embeddings are added to token embeddings:

Segment Embedding: They provide information about the sentence a particular token is a part of.

Position Embedding: They provide information about the order of words in the input.
Outputs

Fine-tuning BERT

Fine-tuning on various downstream tasks is done by swapping out the appropriate inputs or outputs. In the general run of things, to train task-specific models, we add an extra output layer to existing BERT and fine-tune the resultant model — all parameters, end to end. A positive consequence of adding layers — input/output and not changing the BERT model is that only a minimal number of parameters need to be learned from scratch making the procedure fast, cost and resource efficient.
Just to give you an idea of how fast and efficient it is, the authors claim that all the results in the paper can be replicated in *at most 1 hour* on a *single Cloud TPU*, or *a few hours on a GPU*, starting from the exact same pre-trained model.



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

(c) Question Answering Tasks: SQuAD v1.1

(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Fine-tuning BERT on various downstream tasks.

In Sentence Pair Classification and Single Sentence Classification, the final state corresponding to [*CLS*] token is used as input for the additional layers that makes the prediction.
In QA tasks, a start (S) and an end (E) vector are introduced during fine tuning. The question is fed as sentence A and the answer as sentence B. The probability of word *i* being the start of the answer span is computed as a dot product between T*i* (final state corresponding to *i*th input token) and S (start vector) followed by a softmax over all of the words in the paragraph. A similar method is used for end span.

Advantages of Fine-Tuning

Quicker Development

First, the pre-trained BERT model weights already encode a lot of information about our language. As a result, it takes much less time to train our fine-tuned model - it is as if we have already trained the bottom layers of our network extensively and only need to gently tune them while using their output as features for our classification task. In fact, the authors recommend only 2-4 epochs of training for fine-tuning BERT on a specific NLP task (compared to the hundreds of GPU hours needed to train the original BERT model or a LSTM from scratch!).

Less Data

In addition, and perhaps just as important, because of the pre-trained weights this method allows us to fine-tune our task on a much smaller dataset than would be required in a model that is built from scratch. A major drawback of NLP models built from scratch is that we often need a prohibitively large dataset in order to train our network to reasonable accuracy, meaning a lot of time and energy had to be put into dataset creation. By fine-tuning BERT, we are now able to get away with training a model to good performance on a much smaller amount of training data.

Better Results

Finally, this simple fine-tuning procedure (typically adding one fully-connected layer on top of BERT and training for a few epochs) was shown to achieve state of the art results with minimal task-specific adjustments for a wide variety of tasks: classification, language inference, semantic similarity, question answering, etc. Rather than implementing custom and sometimes-obscure architectures shown to work well on a specific task, simply fine-tuning BERT is shown to be a better (or at least equal) alternative.

Steps to Fine Tune BERT Model to perform Multi Class Text Classification

1. **Load dataset**
2. **Pre-process data**
3. **Define model**
4. **Train the model**
5. **Evaluate**

Lab Exercise to be Performed in this Session:
Perform Text Classification by Fine tuning BERT model.

```
import numpy as np
import pandas as pd
import time
import datetime
import gc
import random
from nltk.corpus import stopwords
import re

import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler,random_split
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

import transformers
from transformers import BertForSequenceClassification, AdamW, BertConfig,BertTokenizer,get_linear_schedule_with_warmup
```

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

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

```
df = pd.read_csv("/content/train (1).csv")
df.head()
```

	id	keyword	location	text	target
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1

```
import nltk
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```

```
sw = stopwords.words('english')

def clean_text(text):

    text = text.lower()

    text = re.sub(r"[^a-zA-Z?.!,¿]+", " ", text)

    text = re.sub(r"http\S+", "",text)

    html=re.compile(r'<.*?>')

    text = html.sub(r'',text)

    punctuations = '@#!?+&*[]-%.:/();$=><|{}^' + "'`" + '_'
    for p in punctuations:
        text = text.replace(p,'')

    text = [word.lower() for word in text.split() if word.lower() not in sw]

    text = " ".join(text)

    emoji_pattern = re.compile("["
                           u"\U0001F600-\U0001F64F"
                           u"\U0001F300-\U0001F5FF"
                           u"\U0001F680-\U0001F6FF"
                           u"\U0001F1E0-\U0001F1FF"
                           u"\U00002702-\U000027B0"
                           u"\U000024C2-\U0001F251"
                           "]+", flags=re.UNICODE)
    text = emoji_pattern.sub(r'', text)

    return text
```

```
df['text'] = df['text'].apply(lambda x: clean_text(x))


tweets = df.text.values
labels = df.target.values


tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens),
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%                                  48.0/48.0 [00:00<00:00, 2.51kB/s]

vocab.txt: 100%                                      232k/232k [00:00<00:00, 541kB/s]

tokenizer.json: 100%                                    466k/466k [00:00<00:00, 1.08MB/s]

config.json: 100%                                     570/570 [00:00<00:00, 45.2kB/s]
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:1601: FutureWarning: `clean_up_tokenizat
  warnings.warn(
```

```
tokenizer
```

```
BertTokenizer(name_or_path='bert-base-uncased', vocab_size=30522, model_max_length=512, is_fast=False,
padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pad_token':
'[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True),  added_tokens_decoder={
        0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
        103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
}
```

```
print(' Original: ', tweets[0])
print('Tokenized: ', tokenizer.tokenize(tweets[0]))
print('Token IDs: ', tokenizer.convert_tokens_to_ids(tokenizer.tokenize(tweets[0])))
```

```
 Original:  deeds reason earthquake may allah forgive us
Tokenized:  ['deeds', 'reason', 'earthquake', 'may', 'allah', 'forgive', 'us']
Token IDs:  [15616, 3114, 8372, 2089, 16455, 9641, 2149]
```

```
max_len = 0
for sent in tweets:

    # Tokenize the text and add `[CLS]` and `[SEP]` tokens.
    input_ids = tokenizer.encode(sent, add_special_tokens=True)
    max_len = max(max_len, len(input_ids))

print('Max sentence length: ', max_len)
```

```
Max sentence length:  45
```

```
input_ids = []
attention_masks = []

for tweet in tweets:
    encoded_dict = tokenizer.encode_plus(
                        tweet,
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = max_len,
                        pad_to_max_length = True,
                        return_attention_mask = True,
                        return_tensors = 'pt',
                   )

    input_ids.append(encoded_dict['input_ids'])
    attention_masks.append(encoded_dict['attention_mask'])


input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)

print('Original: ', tweets[0])
print('Token IDs:', input_ids[0])
```

```
Truncation was not explicitly activated but `max_length` is provided a specific value, please use `truncation=True` to e
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:2870: FutureWarning: The `pad_to_max_len
  warnings.warn(
Original:  deeds reason earthquake may allah forgive us
Token IDs: tensor([  101, 15616,  3114,  8372,  2089, 16455,  9641,  2149,   102,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0])
```

```
dataset = TensorDataset(input_ids, attention_masks, labels)

train_size = int(0.8 * len(dataset))
val_size = len(dataset)  - train_size


train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

print('{:>5,} training samples'.format(train_size))
print('{:>5,} validation samples'.format(val_size))
```

```
6,090 training samples
1,523 validation samples
```

```
batch_size = 32
train_dataloader = DataLoader(
            train_dataset,  # The training samples.
            sampler = RandomSampler(train_dataset), # Select batches randomly
            batch_size = batch_size # Trains with this batch size.
        )

validation_dataloader = DataLoader(
            val_dataset, # The validation samples.
            sampler = SequentialSampler(val_dataset), # Pull out batches sequentially.
            batch_size = batch_size # Evaluate with this batch size.
        )
```

```
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", # Use the 12-layer BERT model, with an uncased vocab.
    num_labels = 2, # The number of output labels--2 for binary classification.
                    # You can increase this for multi-class tasks.
    output_attentions = False, # Whether the model returns attentions weights.
    output_hidden_states = False, # Whether the model returns all hidden-states.
)

# if device == "cuda:0":
# # Tell pytorch to run this model on the GPU.
#     model = model.cuda()
model = model.to(device)
```

```
model.safetensors: 100%                                440M/440M [00:01<00:00, 292MB/s]
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and ar
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```
optimizer = AdamW(model.parameters(),
                  lr = 2e-5, # args.learning_rate - default is 5e-5, our notebook had 2e-5
                  eps = 1e-8 # args.adam_epsilon  - default is 1e-8.
                )
```

```
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:591: FutureWarning: This implementation of AdamW is
  warnings.warn(
```

Fine tuning the model

```
epochs = 4
total_steps = len(train_dataloader) * epochs
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0, # Default value in run_glue.py
                                            num_training_steps = total_steps)
```

```
# Function to calculate the accuracy of our predictions vs labels
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)
```

```python
def format_time(elapsed):
    '''
    Takes a time in seconds and returns a string hh:mm:ss
    '''
    elapsed_rounded = int(round((elapsed)))
    return str(datetime.timedelta(seconds=elapsed_rounded))


seed_val = 42
random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)
training_stats = []

total_t0 = time.time()

for epoch_i in range(0, epochs):
    print("")
    print('======== Epoch {:} / {:} ========'.format(epoch_i + 1, epochs))
    print('Training')
    t0 = time.time()
    total_train_loss = 0
    model.train()
    for step, batch in enumerate(train_dataloader):
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)
        optimizer.zero_grad()
        output = model(b_input_ids,
                             token_type_ids=None,
                             attention_mask=b_input_mask,
                             labels=b_labels)
        loss = output.loss
        total_train_loss += loss.item()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        scheduler.step()

    # Calculate the average loss over all of the batches.
    avg_train_loss = total_train_loss / len(train_dataloader)

    # Measure how long this epoch took.
    training_time = format_time(time.time() - t0)
    print("")
    print("  Average training loss: {0:.2f}".format(avg_train_loss))
    print("  Training epcoh took: {:}".format(training_time))
    print("")
    print("Running Validation...")
    t0 = time.time()
    model.eval()
    # Tracking variables
    total_eval_accuracy = 0
    best_eval_accuracy = 0
    total_eval_loss = 0
    nb_eval_steps = 0
    # Evaluate data for one epoch
    for batch in validation_dataloader:
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)
        with torch.no_grad():
            output= model(b_input_ids,
                                   token_type_ids=None,
                                   attention_mask=b_input_mask,
                                   labels=b_labels)
        loss = output.loss
        total_eval_loss += loss.item()
        logits = output.logits
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()
        total_eval_accuracy += flat_accuracy(logits, label_ids)
    avg_val_accuracy = total_eval_accuracy / len(validation_dataloader)
    print("  Accuracy: {0:.2f}".format(avg_val_accuracy))
    avg_val_loss = total_eval_loss / len(validation_dataloader)
    validation_time = format_time(time.time() - t0)
    if avg_val_accuracy > best_eval_accuracy:
        torch.save(model, 'bert_model')
        best_eval_accuracy = avg_val_accuracy
    training_stats.append(
        {
            'epoch': epoch_i + 1,
            'Training Loss': avg_train_loss,
            'Valid. Loss': avg_val_loss,
            'Valid. Accur.': avg_val_accuracy,
            'Training Time': training_time,
            'Validation Time': validation_time
        }
    )
print("")
print("Training complete!")

print("Total training took {:} (h:mm:ss)".format(format_time(time.time()-total_t0)))
```

```
======== Epoch 1 / 4 ========
Training...

  Average training loss: 0.48
  Training epcoh took: 0:00:47

Running Validation...
  Accuracy: 0.83

======== Epoch 2 / 4 ========
Training...

  Average training loss: 0.36
  Training epcoh took: 0:00:49

Running Validation...
  Accuracy: 0.84

======== Epoch 3 / 4 ========
Training...

  Average training loss: 0.28
  Training epcoh took: 0:00:48

Running Validation...
  Accuracy: 0.83

======== Epoch 4 / 4 ========
Training...

  Average training loss: 0.23
  Training epcoh took: 0:00:49

Running Validation...
  Accuracy: 0.83

Training complete!
Total training took 0:03:41 (h:mm:ss)
```

```
model = torch.load('bert_model')
```

```
<ipython-input-20-ac35b85e1100>:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the current defa
  model = torch.load('bert_model')
```

```
df_test = pd.read_csv('/content/test.csv')
df_test['text'] = df_test['text'].apply(lambda x:clean_text(x))
test_tweets = df_test['text'].values


test_input_ids = []
test_attention_masks = []
for tweet in test_tweets:
    encoded_dict = tokenizer.encode_plus(
                        tweet,
                        add_special_tokens = True,
                        max_length = max_len,
                        pad_to_max_length = True,
                        return_attention_mask = True,
                        return_tensors = 'pt',
                   )
    test_input_ids.append(encoded_dict['input_ids'])
    test_attention_masks.append(encoded_dict['attention_mask'])
test_input_ids = torch.cat(test_input_ids, dim=0)
test_attention_masks = torch.cat(test_attention_masks, dim=0)
```

```
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:2870: FutureWarning: The `pad_to_max_len
  warnings.warn(
```

```
test_dataset = TensorDataset(test_input_ids, test_attention_masks)
test_dataloader = DataLoader(
            test_dataset,
            sampler = SequentialSampler(test_dataset),
            batch_size = batch_size
        )


predictions = []
for batch in test_dataloader:
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        with torch.no_grad():
            output= model(b_input_ids,
                                   token_type_ids=None,
                                   attention_mask=b_input_mask)
            logits = output.logits
            logits = logits.detach().cpu().numpy()
            pred_flat = np.argmax(logits, axis=1).flatten()

            predictions.extend(list(pred_flat))


df_output = pd.DataFrame()
df_output['id'] = df_test['id']
df_output['target'] =predictions


df_output
```

```
          id  target
0          0       1
1          2       1
2          3       1
3          9       1
4         11       1
...      ...     ...
3258   10861       1
3259   10865       1
3260   10868       1
3261   10874       1
3262   10875       1
3263 rows × 2 columns
```