BERT with Back Translation Augmentation

Real News and Fake News (~82k total)

Class: Label

Real: 1

Fake: 0

```
import tensorflow as tf

# Get the GPU device name.
device_name = tf.test.gpu_device_name()

# The device name should look like the following:
if device_name == '/device:GPU:0':
    print('Found GPU at: {}'.format(device_name))
else:
    raise SystemError('GPU device not found')
```

```
Found GPU at: /device:GPU:0
```

```
import torch

# If there's a GPU available...
if torch.cuda.is_available():

    # Tell PyTorch to use the GPU.
    device = torch.device("cuda")

    print('There are %d GPU(s) available.' % torch.cuda.device_count())

    print('We will use the GPU:', torch.cuda.get_device_name(0))

# If not
```

```
# If not...
else:
    print('No GPU available, using the CPU instead.')
    device = torch.device("cpu")
```

```
There are 1 GPU(s) available.
We will use the GPU: Tesla P100-PCIE-16GB
```

```
!pip install transformers
```

Back Translation Augmentation

```
# Back Translation

import pandas as pd
import numpy as np

import sklearn
from sklearn.model_selection import train_test_split

file = "combined_{}.csv"
dfs = []
for i in range(3):
    fp = file.format(i+1)
    read = pd.read_csv(fp)
    read = read[['label', 'clean_text']]
    dfs.append(read)

dfs[2] = dfs[2][:-13000]
data = pd.concat(dfs)
data.tail()
data.reset_index(inplace=True, drop=True)

data.dropna(inplace=True)
train_data, test_data = train_test_split(data, test_size=0.3)

print('Train Data:', train_data.shape)
```

```
print(train_data[train_data.label == 1].shape[0], "Real")
print(train_data[train_data.label == 0].shape[0], "Fake")

print('\nTest Data:', test_data.shape)
print(test_data[test_data.label == 1].shape[0], "Real")
print(test_data[test_data.label == 0].shape[0], "Fake")


bt1 = pd.read_csv('Kaggle2_Mixed_bt_clean.csv')[['label', 'clean_text']]
bt2 = pd.read_csv('LIAR_BT_clean.csv')[['label', 'clean_text']]

bt_augmentation = pd.concat([bt1, bt2])
train_data = pd.concat([train_data, bt_augmentation])
train_data.dropna(inplace=True)

print('\n-------------------------------After Augmentation -------------------------------\n')
print('Train Data:', train_data.shape)
print(train_data[train_data.label == 1].shape[0], "Real")
print(train_data[train_data.label == 0].shape[0], "Fake")

print('\nTest Data:', test_data.shape)
print(test_data[test_data.label == 1].shape[0], "Real")
print(test_data[test_data.label == 0].shape[0], "Fake")

sentences = train_data.clean_text.values
labels = train_data.label.values

train_data.head(10)
```

```
Train Data: (41621, 2)
22107 Real
19514 Fake

Test Data: (17838, 2)
9342 Real
8496 Fake

------------------------------After Augmentation ------------------------------

Train Data: (64435, 2)
33292 Real
31143 Fake

Test Data: (17838, 2)
9342 Real
8496 Fake
```

	label	clean_text
43534	1	marco rubios economic proposals add trillion ...
15764	0	november eduard popov fort russ translated...
40554	1	president bush eight years added trillion deb...
50229	1	greatest time may losing edge jordan brand b...
23399	0	november pm trump fans may election arent t...
76	0	email get ready cringeworthy story youre going...
46139	1	says jimrenacci consistently voted loopholes e...
7974	0	chart day explosion student loans vs implosion...
15119	0	politics leader islamic revolution ayatollah s...
52126	1	almost us think starting business point though...

```
from transformers import BertTokenizer

# Load the BERT tokenizer.
```

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

# Print the original sentence.
print(' Original: ', sentences[0])

# Print the sentence split into tokens.
print('Tokenized: ', tokenizer.tokenize(sentences[0]))

# Print the sentence mapped to token ids.
print('Token IDs: ', tokenizer.convert_tokens_to_ids(tokenizer.tokenize(sentences[0])))
```

```
Loading BERT tokenizer...
 Original:  marco rubios economic proposals add  trillion federal deficit
Tokenized:  ['marco', 'rub', '##ios', 'economic', 'proposals', 'add', 'trillion', 'federal', 'deficit']
Token IDs:  [8879, 14548, 10735, 3171, 10340, 5587, 23458, 2976, 15074]
```

```
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []

# For every sentence...
for sent in sentences:
    # `encode` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    encoded_sent = tokenizer.encode(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = 512  # Truncate all sentences.
                        #return_tensors = 'pt',     # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_sent)

# Print sentence 0, now as a list of IDs.
print('Original: ', sentences[0])
```

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

```
Original:  marco rubios economic proposals add  trillion federal deficit
Token IDs: [101, 8879, 14548, 10735, 3171, 10340, 5587, 23458, 2976, 15074, 102]
```

```
import statistics

print('Avg sentence length: ', statistics.mean([len(sen) for sen in input_ids]))
```

```
Avg sentence length:  179.84809497943664
```

```
print('Max sentence length: ', max([len(sen) for sen in input_ids]))
```

```
Max sentence length:  512
```

```
import keras
# We'll borrow the `pad_sequences` utility function to do this.
from keras.preprocessing.sequence import pad_sequences

# Set the maximum sequence length.
# I've chosen 64 somewhat arbitrarily. It's slightly larger than the
# maximum training sentence length of 47...
MAX_LEN = 256

print('\nPadding/truncating all sentences to %d values...' % MAX_LEN)

print('\nPadding token: "{:}", ID: {:}'.format(tokenizer.pad_token, tokenizer.pad_token_id))

# Pad our input tokens with value 0.
# "post" indicates that we want to pad and truncate at the end of the sequence,
# as opposed to the beginning.
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long",
                          value=0, truncating="post", padding="post")

print('\nDone.')
```

```
Padding/truncating all sentences to 256 values...

Padding token: "[PAD]", ID: 0
Using TensorFlow backend.

Done.
```

```
# Create attention masks
attention_masks = []

# For each sentence...
for sent in input_ids:

    # Create the attention mask.
    #   - If a token ID is 0, then it's padding, set the mask to 0.
    #   - If a token ID is > 0, then it's a real token, set the mask to 1.
    att_mask = [int(token_id > 0) for token_id in sent]

    # Store the attention mask for this sentence.
    attention_masks.append(att_mask)

# Use 90% for training and 10% for validation.
train_inputs, validation_inputs, train_labels, validation_labels = train_test_split(input_ids, labels,
                                                            random_state=2018, test_size=0.1)
# Do the same for the masks.
train_masks, validation_masks, _, _ = train_test_split(attention_masks, labels,
                                             random_state=2018, test_size=0.1)

# Convert all inputs and labels into torch tensors, the required datatype
# for our model.
train_inputs = torch.tensor(train_inputs)
validation_inputs = torch.tensor(validation_inputs)

train_labels = torch.tensor(train_labels)
validation_labels = torch.tensor(validation_labels)
```

```
train_masks = torch.tensor(train_masks)
validation_masks = torch.tensor(validation_masks)


from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler

# The DataLoader needs to know our batch size for training, so we specify it
# here.
# For fine-tuning BERT on a specific task, the authors recommend a batch size of
# 16 or 32.

batch_size = 32

# Create the DataLoader for our training set.
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=batch_size)

# Create the DataLoader for our validation set.
validation_data = TensorDataset(validation_inputs, validation_masks, validation_labels)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validation_sampler, batch_size=batch_size)


from transformers import BertForSequenceClassification, AdamW, BertConfig

# Load BertForSequenceClassification, the pretrained BERT model with a single
# linear classification layer on top.
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.
)

# Tell pytorch to run this model on the GPU.
model.cuda()


# Note: AdamW is a class from the huggingface library (as opposed to pytorch)
```

```
# I believe the 'W' stands for 'Weight Decay fix"
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.
                )



from transformers import get_linear_schedule_with_warmup

# Number of training epochs (authors recommend between 2 and 4)
epochs = 5

# Total number of training steps is number of batches * number of epochs.
total_steps = len(train_dataloader) * epochs

# Create the learning rate scheduler.
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)


import time
import datetime

def format_time(elapsed):
    '''
    Takes a time in seconds and returns a string hh:mm:ss
    '''
    # Round to the nearest second.
    elapsed_rounded = int(round((elapsed)))

    # Format as hh:mm:ss
```

```
    return str(datetime.timedelta(seconds=elapsed_rounded))

torch.cuda.empty_cache()

import random

# This training code is based on the `run_glue.py` script here:
# https://github.com/huggingface/transformers/blob/5bfcd0485ece086ebcbed2d008813037968a9e58/examples/run_glue.py#I


# Set the seed value all over the place to make this reproducible.
seed_val = 42

random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

# Store the average loss after each epoch so we can plot them.
loss_values = []

# For each epoch...
for epoch_i in range(0, epochs):

    # ========================================
    #               Training
    # ========================================

    # Perform one full pass over the training set.

    print("")
    print('======== Epoch {:} / {:} ========'.format(epoch_i + 1, epochs))
    print('Training...')

    # Measure how long the training epoch takes.
    t0 = time.time()

    # Reset the total loss for this epoch.
    total_loss = 0
```

```
    total_loss = 0

    # Put the model into training mode. Don't be mislead--the call to
    # `train` just changes the *mode*, it doesn't *perform* the training.
    # `dropout` and `batchnorm` layers behave differently during training
    # vs. test (source: https://stackoverflow.com/questions/51433378/what-does-model-train-do-in-pytorch)
    model.train()

    # For each batch of training data...
    for step, batch in enumerate(train_dataloader):

        # Progress update every 40 batches.
        if step % 40 == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Report progress.
            print('  Batch {:>5,}  of  {:>5,}.    Elapsed: {:}.'.format(step, len(train_dataloader), elapsed))

        # Unpack this training batch from our dataloader.
        #
        # As we unpack the batch, we'll also copy each tensor to the GPU using the
        # `to` method.
        #
        # `batch` contains three pytorch tensors:
        #   [0]: input ids
        #   [1]: attention masks
        #   [2]: labels
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)

        # Always clear any previously calculated gradients before performing a
        # backward pass. PyTorch doesn't do this automatically because
        # accumulating the gradients is "convenient while training RNNs".
        # (source: https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-grad-in-pytorch)
        model.zero_grad()

        # Perform a forward pass (evaluate the model on this training batch).
        # This will return the loss (rather than the model output) because we
```

```
    # This will return the loss (rather than the model output) because we
    # have provided the `labels`.
    # The documentation for this `model` function is here:
    # https://huggingface.co/transformers/v2.2.0/model_doc/bert.html#transformers.BertForSequenceClassificatic
    outputs = model(b_input_ids,
                token_type_ids=None,
                attention_mask=b_input_mask,
                labels=b_labels)

    # The call to `model` always returns a tuple, so we need to pull the
    # loss value out of the tuple.
    loss = outputs[0]

    # Accumulate the training loss over all of the batches so that we can
    # calculate the average loss at the end. `loss` is a Tensor containing a
    # single value; the `.item()` function just returns the Python value
    # from the tensor.
    total_loss += loss.item()

    # Perform a backward pass to calculate the gradients.
    loss.backward()

    # Clip the norm of the gradients to 1.0.
    # This is to help prevent the "exploding gradients" problem.
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

    # Update parameters and take a step using the computed gradient.
    # The optimizer dictates the "update rule"--how the parameters are
    # modified based on their gradients, the learning rate, etc.
    optimizer.step()

    # Update the learning rate.
    scheduler.step()

# Calculate the average loss over the training data.
avg_train_loss = total_loss / len(train_dataloader)

# Store the loss value for plotting the learning curve.
loss_values.append(avg_train_loss)
```

```
    print("")
    print("  Average training loss: {0:.2f}".format(avg_train_loss))
    print("  Training epcoh took: {:}".format(format_time(time.time() - t0)))

    # ========================================
    #               Validation
    # ========================================
    # After the completion of each training epoch, measure our performance on
    # our validation set.

    print("")
    print("Running Validation...")

    t0 = time.time()

    # Put the model in evaluation mode--the dropout layers behave differently
    # during evaluation.
    model.eval()

    # Tracking variables
    eval_loss, eval_accuracy = 0, 0
    nb_eval_steps, nb_eval_examples = 0, 0

    # Evaluate data for one epoch
    for batch in validation_dataloader:

        # Add batch to GPU
        batch = tuple(t.to(device) for t in batch)

        # Unpack the inputs from our dataloader
        b_input_ids, b_input_mask, b_labels = batch

        # Telling the model not to compute or store gradients, saving memory and
        # speeding up validation
        with torch.no_grad():

            # Forward pass, calculate logit predictions.
            # This will return the logits rather than the loss because we have
            # not provided labels.
```

```
            # not provided labels.
            # token_type_ids is the same as the "segment ids", which
            # differentiates sentence 1 and 2 in 2-sentence tasks.
            # The documentation for this `model` function is here:
            # https://huggingface.co/transformers/v2.2.0/model_doc/bert.html#transformers.BertForSequenceClassific
            outputs = model(b_input_ids,
                            token_type_ids=None,
                            attention_mask=b_input_mask)

        # Get the "logits" output by the model. The "logits" are the output
        # values prior to applying an activation function like the softmax.
        logits = outputs[0]

        # Move logits and labels to CPU
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()

        # Calculate the accuracy for this batch of test sentences.
        tmp_eval_accuracy = flat_accuracy(logits, label_ids)

        # Accumulate the total accuracy.
        eval_accuracy += tmp_eval_accuracy

        # Track the number of batches
        nb_eval_steps += 1

    # Report the final accuracy for this validation run.
    print("  Accuracy: {0:.3f}".format(eval_accuracy/nb_eval_steps))
    print("  Validation took: {:}".format(format_time(time.time() - t0)))

print("")
print("Training complete!")
```

```
======== Epoch 1 / 5 ========
Training...
  Batch    40  of  1,813.    Elapsed: 0:00:30.
  Batch    80  of  1,813.    Elapsed: 0:01:01.
  Batch   120  of  1,813.    Elapsed: 0:01:31.
  Batch   160  of  1,813.    Elapsed: 0:02:01.
  Batch   200  of  1,813.    Elapsed: 0:02:31.
  Batch   240  of  1,813.    Elapsed: 0:03:02.
  Batch   280  of  1,813.    Elapsed: 0:03:32.
  Batch   320  of  1,813.    Elapsed: 0:04:02.
  Batch   360  of  1,813.    Elapsed: 0:04:32.
  Batch   400  of  1,813.    Elapsed: 0:05:03.
  Batch   440  of  1,813.    Elapsed: 0:05:33.
  Batch   480  of  1,813.    Elapsed: 0:06:03.
  Batch   520  of  1,813.    Elapsed: 0:06:33.
  Batch   560  of  1,813.    Elapsed: 0:07:04.
  Batch   600  of  1,813.    Elapsed: 0:07:34.
  Batch   640  of  1,813.    Elapsed: 0:08:04.
  Batch   680  of  1,813.    Elapsed: 0:08:35.
  Batch   720  of  1,813.    Elapsed: 0:09:05.
  Batch   760  of  1,813.    Elapsed: 0:09:35.
  Batch   800  of  1,813.    Elapsed: 0:10:06.
  Batch   840  of  1,813.    Elapsed: 0:10:36.
  Batch   880  of  1,813.    Elapsed: 0:11:06.
  Batch   920  of  1,813.    Elapsed: 0:11:36.
  Batch   960  of  1,813.    Elapsed: 0:12:07.
  Batch 1,000  of  1,813.    Elapsed: 0:12:37.
  Batch 1,040  of  1,813.    Elapsed: 0:13:07.
  Batch 1,080  of  1,813.    Elapsed: 0:13:38.
  Batch 1,120  of  1,813.    Elapsed: 0:14:08.
  Batch 1,160  of  1,813.    Elapsed: 0:14:38.
  Batch 1,200  of  1,813.    Elapsed: 0:15:09.
  Batch 1,240  of  1,813.    Elapsed: 0:15:39.
  Batch 1,280  of  1,813.    Elapsed: 0:16:09.
  Batch 1,320  of  1,813.    Elapsed: 0:16:39.
  Batch 1,360  of  1,813.    Elapsed: 0:17:10.
  Batch 1,400  of  1,813.    Elapsed: 0:17:40.
  Batch 1,440  of  1,813.    Elapsed: 0:18:10.
  Batch 1,480  of  1,813.    Elapsed: 0:18:41.
  Batch 1,520  of  1,813.    Elapsed: 0:19:11.
  Batch 1,560  of  1,813.    Elapsed: 0:19:41.
  Batch 1,600  of  1,813.    Elapsed: 0:20:11.
```

```
  Batch 1,640  of  1,813.    Elapsed: 0:20:42.
  Batch 1,680  of  1,813.    Elapsed: 0:21:12.
  Batch 1,720  of  1,813.    Elapsed: 0:21:42.
  Batch 1,760  of  1,813.    Elapsed: 0:22:13.
  Batch 1,800  of  1,813.    Elapsed: 0:22:43.

  Average training loss: 0.38
  Training epcoh took: 0:22:52

Running Validation...
  Accuracy: 0.849
  Validation took: 0:00:49

======== Epoch 2 / 5 ========
Training...
  Batch    40  of  1,813.    Elapsed: 0:00:30.
  Batch    80  of  1,813.    Elapsed: 0:01:01.
  Batch   120  of  1,813.    Elapsed: 0:01:31.
  Batch   160  of  1,813.    Elapsed: 0:02:01.
  Batch   200  of  1,813.    Elapsed: 0:02:31.
  Batch   240  of  1,813.    Elapsed: 0:03:02.
  Batch   280  of  1,813.    Elapsed: 0:03:32.
  Batch   320  of  1,813.    Elapsed: 0:04:02.
  Batch   360  of  1,813.    Elapsed: 0:04:33.
  Batch   400  of  1,813.    Elapsed: 0:05:03.
  Batch   440  of  1,813.    Elapsed: 0:05:33.
  Batch   480  of  1,813.    Elapsed: 0:06:03.
  Batch   520  of  1,813.    Elapsed: 0:06:34.
  Batch   560  of  1,813.    Elapsed: 0:07:04.
  Batch   600  of  1,813.    Elapsed: 0:07:34.
  Batch   640  of  1,813.    Elapsed: 0:08:05.
  Batch   680  of  1,813.    Elapsed: 0:08:35.
  Batch   720  of  1,813.    Elapsed: 0:09:05.
  Batch   760  of  1,813.    Elapsed: 0:09:36.
  Batch   800  of  1,813.    Elapsed: 0:10:06.
  Batch   840  of  1,813.    Elapsed: 0:10:36.
  Batch   880  of  1,813.    Elapsed: 0:11:07.
  Batch   920  of  1,813.    Elapsed: 0:11:37.
  Batch   960  of  1,813.    Elapsed: 0:12:07.
  Batch 1,000  of  1,813.    Elapsed: 0:12:38.
  Batch 1,040  of  1,813.    Elapsed: 0:13:08.
  Batch 1,080  of  1,813.    Elapsed: 0:13:38.
  Batch 1,120  of  1,813.    Elapsed: 0:14:09.
  Batch 1,160  of  1,813.    Elapsed: 0:14:39.
```

```
  Batch 1,200  of  1,813.    Elapsed: 0:15:09.
  Batch 1,240  of  1,813.    Elapsed: 0:15:40.
  Batch 1,280  of  1,813.    Elapsed: 0:16:10.
  Batch 1,320  of  1,813.    Elapsed: 0:16:40.
  Batch 1,360  of  1,813.    Elapsed: 0:17:11.
  Batch 1,400  of  1,813.    Elapsed: 0:17:41.
  Batch 1,440  of  1,813.    Elapsed: 0:18:11.
  Batch 1,480  of  1,813.    Elapsed: 0:18:42.
  Batch 1,520  of  1,813.    Elapsed: 0:19:12.
  Batch 1,560  of  1,813.    Elapsed: 0:19:42.
  Batch 1,600  of  1,813.    Elapsed: 0:20:13.
  Batch 1,640  of  1,813.    Elapsed: 0:20:43.
  Batch 1,680  of  1,813.    Elapsed: 0:21:13.
  Batch 1,720  of  1,813.    Elapsed: 0:21:44.
  Batch 1,760  of  1,813.    Elapsed: 0:22:14.
  Batch 1,800  of  1,813.    Elapsed: 0:22:44.

  Average training loss: 0.25
  Training epcoh took: 0:22:53

Running Validation...
  Accuracy: 0.874
  Validation took: 0:00:49

======== Epoch 3 / 5 ========
Training...
  Batch    40  of  1,813.    Elapsed: 0:00:30.
  Batch    80  of  1,813.    Elapsed: 0:01:01.
  Batch   120  of  1,813.    Elapsed: 0:01:31.
  Batch   160  of  1,813.    Elapsed: 0:02:01.
  Batch   200  of  1,813.    Elapsed: 0:02:31.
  Batch   240  of  1,813.    Elapsed: 0:03:02.
  Batch   280  of  1,813.    Elapsed: 0:03:32.
  Batch   320  of  1,813.    Elapsed: 0:04:02.
  Batch   360  of  1,813.    Elapsed: 0:04:33.
  Batch   400  of  1,813.    Elapsed: 0:05:03.
  Batch   440  of  1,813.    Elapsed: 0:05:33.
  Batch   480  of  1,813.    Elapsed: 0:06:03.
  Batch   520  of  1,813.    Elapsed: 0:06:34.
  Batch   560  of  1,813.    Elapsed: 0:07:04.
  Batch   600  of  1,813.    Elapsed: 0:07:34.
  Batch   640  of  1,813.    Elapsed: 0:08:04.
  Batch   680  of  1,813.    Elapsed: 0:08:35.
```

```
  Batch   720  of  1,813.    Elapsed: 0:09:05.
  Batch   760  of  1,813.    Elapsed: 0:09:35.
  Batch   800  of  1,813.    Elapsed: 0:10:05.
  Batch   840  of  1,813.    Elapsed: 0:10:36.
  Batch   880  of  1,813.    Elapsed: 0:11:06.
  Batch   920  of  1,813.    Elapsed: 0:11:36.
  Batch   960  of  1,813.    Elapsed: 0:12:07.
  Batch 1,000  of  1,813.    Elapsed: 0:12:37.
  Batch 1,040  of  1,813.    Elapsed: 0:13:07.
  Batch 1,080  of  1,813.    Elapsed: 0:13:37.
  Batch 1,120  of  1,813.    Elapsed: 0:14:08.
  Batch 1,160  of  1,813.    Elapsed: 0:14:38.
  Batch 1,200  of  1,813.    Elapsed: 0:15:08.
  Batch 1,240  of  1,813.    Elapsed: 0:15:39.
  Batch 1,280  of  1,813.    Elapsed: 0:16:09.
  Batch 1,320  of  1,813.    Elapsed: 0:16:39.
  Batch 1,360  of  1,813.    Elapsed: 0:17:09.
  Batch 1,400  of  1,813.    Elapsed: 0:17:40.
  Batch 1,440  of  1,813.    Elapsed: 0:18:10.
  Batch 1,480  of  1,813.    Elapsed: 0:18:40.
  Batch 1,520  of  1,813.    Elapsed: 0:19:11.
  Batch 1,560  of  1,813.    Elapsed: 0:19:41.
  Batch 1,600  of  1,813.    Elapsed: 0:20:11.
  Batch 1,640  of  1,813.    Elapsed: 0:20:42.
  Batch 1,680  of  1,813.    Elapsed: 0:21:12.
  Batch 1,720  of  1,813.    Elapsed: 0:21:42.
  Batch 1,760  of  1,813.    Elapsed: 0:22:12.
  Batch 1,800  of  1,813.    Elapsed: 0:22:43.

  Average training loss: 0.17
  Training epcoh took: 0:22:52

Running Validation...
  Accuracy: 0.894
  Validation took: 0:00:49

======== Epoch 4 / 5 ========
Training...
  Batch    40  of  1,813.    Elapsed: 0:00:30.
  Batch    80  of  1,813.    Elapsed: 0:01:01.
  Batch   120  of  1,813.    Elapsed: 0:01:31.
  Batch   160  of  1,813.    Elapsed: 0:02:01.
  Batch   200  of  1,813.    Elapsed: 0:02:32.
  Batch   240  of  1,813.    Elapsed: 0:03:02.
```

```
  Batch   280  of  1,813.    Elapsed: 0:03:32.
  Batch   320  of  1,813.    Elapsed: 0:04:03.
  Batch   360  of  1,813.    Elapsed: 0:04:33.
  Batch   400  of  1,813.    Elapsed: 0:05:03.
  Batch   440  of  1,813.    Elapsed: 0:05:33.
  Batch   480  of  1,813.    Elapsed: 0:06:04.
  Batch   520  of  1,813.    Elapsed: 0:06:34.
  Batch   560  of  1,813.    Elapsed: 0:07:04.
  Batch   600  of  1,813.    Elapsed: 0:07:35.
  Batch   640  of  1,813.    Elapsed: 0:08:05.
  Batch   680  of  1,813.    Elapsed: 0:08:35.
  Batch   720  of  1,813.    Elapsed: 0:09:06.
  Batch   760  of  1,813.    Elapsed: 0:09:36.
  Batch   800  of  1,813.    Elapsed: 0:10:06.
  Batch   840  of  1,813.    Elapsed: 0:10:37.
  Batch   880  of  1,813.    Elapsed: 0:11:07.
  Batch   920  of  1,813.    Elapsed: 0:11:37.
  Batch   960  of  1,813.    Elapsed: 0:12:07.
  Batch 1,000  of  1,813.    Elapsed: 0:12:38.
  Batch 1,040  of  1,813.    Elapsed: 0:13:08.
  Batch 1,080  of  1,813.    Elapsed: 0:13:38.
  Batch 1,120  of  1,813.    Elapsed: 0:14:09.
  Batch 1,160  of  1,813.    Elapsed: 0:14:39.
  Batch 1,200  of  1,813.    Elapsed: 0:15:09.
  Batch 1,240  of  1,813.    Elapsed: 0:15:40.
  Batch 1,280  of  1,813.    Elapsed: 0:16:10.
  Batch 1,320  of  1,813.    Elapsed: 0:16:40.
  Batch 1,360  of  1,813.    Elapsed: 0:17:11.
  Batch 1,400  of  1,813.    Elapsed: 0:17:41.
  Batch 1,440  of  1,813.    Elapsed: 0:18:11.
  Batch 1,480  of  1,813.    Elapsed: 0:18:42.
  Batch 1,520  of  1,813.    Elapsed: 0:19:12.
  Batch 1,560  of  1,813.    Elapsed: 0:19:42.
  Batch 1,600  of  1,813.    Elapsed: 0:20:13.
  Batch 1,640  of  1,813.    Elapsed: 0:20:43.
  Batch 1,680  of  1,813.    Elapsed: 0:21:13.
  Batch 1,720  of  1,813.    Elapsed: 0:21:44.
  Batch 1,760  of  1,813.    Elapsed: 0:22:14.
  Batch 1,800  of  1,813.    Elapsed: 0:22:44.

  Average training loss: 0.11
  Training epcoh took: 0:22:53

Running Validation...
```

```
  Accuracy: 0.905
  Validation took: 0:00:49

======== Epoch 5 / 5 ========
Training...
  Batch    40  of  1,813.    Elapsed: 0:00:30.
  Batch    80  of  1,813.    Elapsed: 0:01:01.
  Batch   120  of  1,813.    Elapsed: 0:01:31.
  Batch   160  of  1,813.    Elapsed: 0:02:01.
  Batch   200  of  1,813.    Elapsed: 0:02:32.
  Batch   240  of  1,813.    Elapsed: 0:03:02.
  Batch   280  of  1,813.    Elapsed: 0:03:32.
  Batch   320  of  1,813.    Elapsed: 0:04:03.
  Batch   360  of  1,813.    Elapsed: 0:04:33.
  Batch   400  of  1,813.    Elapsed: 0:05:03.
  Batch   440  of  1,813.    Elapsed: 0:05:34.
  Batch   480  of  1,813.    Elapsed: 0:06:04.
  Batch   520  of  1,813.    Elapsed: 0:06:34.
  Batch   560  of  1,813.    Elapsed: 0:07:04.
  Batch   600  of  1,813.    Elapsed: 0:07:35.
  Batch   640  of  1,813.    Elapsed: 0:08:05.
  Batch   680  of  1,813.    Elapsed: 0:08:35.
  Batch   720  of  1,813.    Elapsed: 0:09:06.
  Batch   760  of  1,813.    Elapsed: 0:09:36.
  Batch   800  of  1,813.    Elapsed: 0:10:06.
  Batch   840  of  1,813.    Elapsed: 0:10:37.
  Batch   880  of  1,813.    Elapsed: 0:11:07.
  Batch   920  of  1,813.    Elapsed: 0:11:37.
  Batch   960  of  1,813.    Elapsed: 0:12:08.
  Batch 1,000  of  1,813.    Elapsed: 0:12:38.
  Batch 1,040  of  1,813.    Elapsed: 0:13:08.
  Batch 1,080  of  1,813.    Elapsed: 0:13:39.
  Batch 1,120  of  1,813.    Elapsed: 0:14:09.
  Batch 1,160  of  1,813.    Elapsed: 0:14:39.
  Batch 1,200  of  1,813.    Elapsed: 0:15:10.
  Batch 1,240  of  1,813.    Elapsed: 0:15:40.
  Batch 1,280  of  1,813.    Elapsed: 0:16:10.
  Batch 1,320  of  1,813.    Elapsed: 0:16:41.
  Batch 1,360  of  1,813.    Elapsed: 0:17:11.
  Batch 1,400  of  1,813.    Elapsed: 0:17:41.
  Batch 1,440  of  1,813.    Elapsed: 0:18:11.
  Batch 1,480  of  1,813.    Elapsed: 0:18:42.
  Batch 1,520  of  1,813.    Elapsed: 0:19:12.
  Batch 1,560  of  1,813.    Elapsed: 0:19:42.
```

```
  Batch 1,560  of  1,813.    Elapsed: 0:19:42.
  Batch 1,600  of  1,813.    Elapsed: 0:20:13.
  Batch 1,640  of  1,813.    Elapsed: 0:20:43.
  Batch 1,680  of  1,813.    Elapsed: 0:21:13.
  Batch 1,720  of  1,813.    Elapsed: 0:21:44.
  Batch 1,760  of  1,813.    Elapsed: 0:22:14.
  Batch 1,800  of  1,813.    Elapsed: 0:22:44.

  Average training loss: 0.07
  Training epcoh took: 0:22:54

Running Validation...
  Accuracy: 0.909
  Validation took: 0:00:49

Training complete!
```

```
# Report the number of sentences.
print('Number of test sentences: {:,}\n'.format(test_data.shape[0]))

# Create sentence and label lists
sentences = test_data.clean_text.values
labels = test_data.label.values

# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []

# For every sentence...
for sent in sentences:
    # `encode` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    encoded_sent = tokenizer.encode(
                        sent,                      # Sentence to encode.
                        max_length = 512,
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                   )
```

```
        input_ids.append(encoded_sent)

    # Pad our input tokens
    input_ids = pad_sequences(input_ids, maxlen=MAX_LEN,
                              dtype="long", truncating="post", padding="post")

    # Create attention masks
    attention_masks = []

    # Create a mask of 1s for each token followed by 0s for padding
    for seq in input_ids:
      seq_mask = [float(i>0) for i in seq]
      attention_masks.append(seq_mask)

    # Convert to tensors.
    prediction_inputs = torch.tensor(input_ids)
    prediction_masks = torch.tensor(attention_masks)
    prediction_labels = torch.tensor(labels)

    # Set the batch size.
    batch_size = 32

    # Create the DataLoader.
    prediction_data = TensorDataset(prediction_inputs, prediction_masks, prediction_labels)
    prediction_sampler = SequentialSampler(prediction_data)
    prediction_dataloader = DataLoader(prediction_data, sampler=prediction_sampler, batch_size=batch_size)
```

```
Number of test sentences: 17,838
```

```
    # Prediction on test set

    print('Predicting labels for {:,} test sentences...'.format(len(prediction_inputs)))

    # Put model in evaluation mode
    model.eval()

    # Tracking variables
    predictions, true_labels = [], []
```

```
predictions , true_labels = [], []
test_loss, test_accuracy = 0, 0
nb_test_steps, nb_test_examples = 0, 0

# Predict
for batch in prediction_dataloader:
  # Add batch to GPU
  batch = tuple(t.to(device) for t in batch)

  # Unpack the inputs from our dataloader
  b_input_ids, b_input_mask, b_labels = batch

  # Telling the model not to compute or store gradients, saving memory and
  # speeding up prediction
  with torch.no_grad():
      # Forward pass, calculate logit predictions
      outputs = model(b_input_ids, token_type_ids=None,
                      attention_mask=b_input_mask)

  logits = outputs[0]

  # Move logits and labels to CPU
  logits = logits.detach().cpu().numpy()
  label_ids = b_labels.to('cpu').numpy()

  # # Store predictions and true labels
  # predictions.append(logits)
  # true_labels.append(label_ids)

  # Calculate the accuracy for this batch of test sentences.
  tmp_test_accuracy = flat_accuracy(logits, label_ids)

  # Accumulate the total accuracy.
  test_accuracy += tmp_test_accuracy

  # Track the number of batches
  nb_test_steps += 1

# Report the final accuracy for this validation run.
print("Testing Accuracy: {0:.3f}".format(test_accuracy/nb_test_steps))
```

```
print('Testing Accuracy: {0:.3f}'.format(test_accuracy/nb_test_steps))
print('    DONE.')
```

```
Predicting labels for 17,838 test sentences...
Testing Accuracy: 0.946
    DONE.
```