CS 505 Homework 06: Transformers

Problem Two

See the problem one notebook for details on due date, submission, etc.

As with problem one, there is an extensive tutorial section, followed by some tasks at the end you need to complete.

Full Disclosure: This is based on a BERT Fine-Tuning Tutorial with PyTorch, by Chris McCormick and Nick Ryan

Introduction

Here is the original introduction to this tutorial material:

In this tutorial I'll show you how to use BERT with the huggingface PyTorch library to quickly and efficiently fine-tune a model to get near state of the art performance in sentence classification. More broadly, I describe the practical application of transfer learning in NLP to create high performance models with minimal effort on a range of NLP tasks.

This post is presented in two forms--as a blog post here (http://mccormickml.com/2019/07/22/BERT-fine-tuning/) and as a Colab Notebook here (https://colab.research.google.com/drive/1pTuQhug6Dhl9XalKB0zUGf4FIdYFlpcX).

The content is identical in both, but:

- The blog post includes a comments section for discussion.
- The Colab Notebook will allow you to run the code and inspect it as you read through.

I've also published a video walkthrough of this post on my YouTube channel! Part 1 (https://youtu.be/x66kkDnbzi4) and Part 2 (https://youtu.be/Hnvb9b7a_Ps).

History

2018 was a breakthrough year in NLP. Transfer learning, particularly models like Allen AI's ELMO, OpenAI's Open-GPT, and Google's BERT allowed researchers to smash multiple benchmarks with minimal task-specific fine-tuning and provided the rest of the NLP community with pretrained models that could easily (with less data and less compute time) be fine-tuned and implemented to produce state of the art results. Unfortunately, for many starting out in NLP and even for some experienced practicioners, the theory and practical application of these powerful models is still not well understood.

What is BERT?

BERT (Bidirectional Encoder Representations from Transformers), released in late 2018, is the model we will use in this tutorial to provide readers with a better understanding of and practical guidance for using transfer learning models in NLP. BERT is a method of pretraining language representations that was used to create models that NLP practicioners can then download and use for free. You can either use these models to extract high quality language features from your text data, or you can fine-tune these models on a specific task (classification, entity recognition, question answering, etc.) with your own data to produce state of the art predictions.

This post will explain how you can modify and fine-tune BERT to create a powerful NLP model that quickly gives you state of the art results.

Advantages of Fine-Tuning

In this tutorial, we will use BERT to train a text classifier. Specifically, we will take the pre-trained BERT model, add an untrained layer of neurons on the end, and train the new model for our classification task. Why do this rather than train a train a specific deep learning model (a CNN, BiLSTM, etc.) that is well suited for the specific NLP task you need?

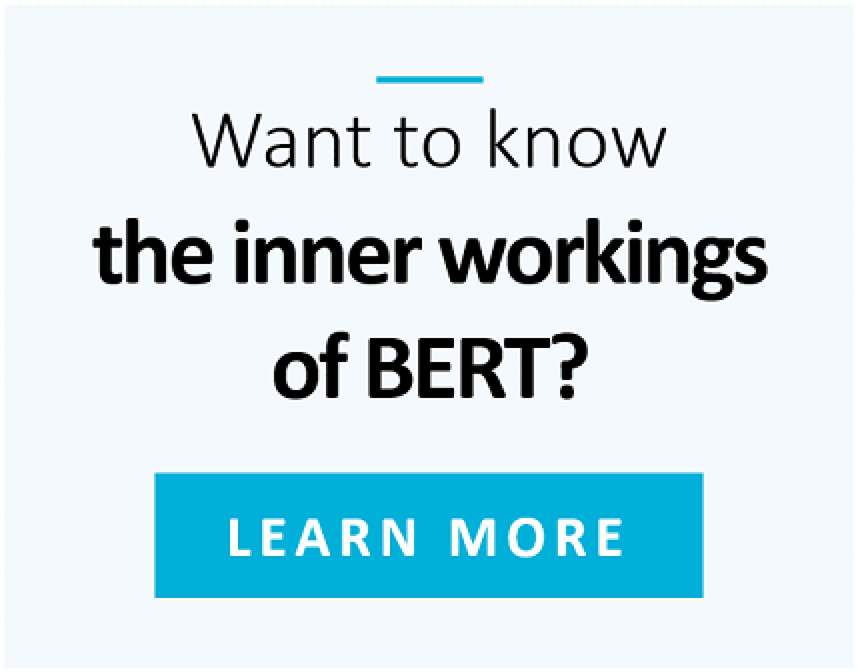
1. **Quicker Development**
   - First, the pre-trained BERT model weights already encode a lot of information about our language. As a result, it takes much less time to train our fine-tuned model - it is as if we have already trained the bottom layers of our network extensively and only need to gently 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!).
2. **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.
3. **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 architetures shown to

work well on a specific task, simply fine-tuning BERT is shown to be a better (or at least equal) alternative.

A Shift in NLP

This shift to transfer learning parallels the same shift that took place in computer vision a few years ago. Creating a good deep learning network for computer vision tasks can take millions of parameters and be very expensive to train. Researchers discovered that deep networks learn hierarchical feature representations (simple features like edges at the lowest layers with gradually more complex features at higher layers). Rather than training a new network from scratch each time, the lower layers of a trained network with generalized image features could be copied and transfered for use in another network with a different task. It soon became common practice to download a pre-trained deep network and quickly retrain it for the new task or add additional layers on top - vastly preferable to the expensive process of training a network from scratch. For many, the introduction of deep pre-trained language models in 2018 (ELMO, BERT, ULMFIT, Open-GPT, etc.) signals the same shift to transfer learning in NLP that computer vision saw.

Let's get started!



(https://bit.ly/30JzuBH)

1. Setup

1.1. Using Colab GPU for Training

Google Colab offers free GPUs and TPUs! Since we'll be training a large neural network it's best to take advantage of this (in this case we'll attach a GPU), otherwise training will take a very long time.

A GPU can be added by going to the menu and selecting:

`Edit  Notebook Settings  Hardware accelerator  (GPU)`

Then run the following cell to confirm that the GPU is detected.

In [1]:
```python
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
```

In order for torch to use the GPU, we need to identify and specify the GPU as the device. Later, in our training loop, we will load data onto the device.

In [2]:
```python
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...
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 T4
```

1.2. Installing the Hugging Face Library

Next, let's install the transformers (https://github.com/huggingface/transformers) package from Hugging Face which will give us a pytorch interface for working with BERT. (This library contains interfaces for other pretrained language models like OpenAI's GPT and GPT-2.) We've selected the pytorch interface because it strikes a nice balance between the high-level APIs (which are easy to use but don't provide insight into how things work) and tensorflow code (which contains lots of details but often sidetracks us into lessons about tensorflow, when the purpose here is BERT!).

At the moment, the Hugging Face library seems to be the most widely accepted and powerful pytorch interface for working with BERT. In addition to supporting a variety of different pre-trained transformer models, the library also includes pre-built modifications of these models suited to your specific task. For example, in this tutorial we will use `BertForSequenceClassification`.

The library also includes task-specific classes for token classification, question answering, next sentence prediciton, etc. Using these pre-built classes simplifies the process of modifying BERT for your purposes.

In [3]:
```
!pip install transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python
3.10/dist-packages (4.35.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.1
0/dist-packages (from transformers) (3.13.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in /usr/
local/lib/python3.10/dist-packages (from transformers) (0.19.4)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python
3.10/dist-packages (from transformers) (1.23.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/pyt
hon3.10/dist-packages (from transformers) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python
3.10/dist-packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/p
ython3.10/dist-packages (from transformers) (2023.6.3)
Requirement already satisfied: requests in /usr/local/lib/python3.1
0/dist-packages (from transformers) (2.31.0)
Requirement already satisfied: tokenizers<0.19,>=0.14 in /usr/local/
lib/python3.10/dist-packages (from transformers) (0.15.0)
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/
python3.10/dist-packages (from transformers) (0.4.1)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.
10/dist-packages (from transformers) (4.66.1)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/py
thon3.10/dist-packages (from huggingface-hub<1.0,>=0.16.4->transform
ers) (2023.6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/lo
cal/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.16.4-
>transformers) (4.5.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/loca
l/lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python
3.10/dist-packages (from requests->transformers) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/
python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/
python3.10/dist-packages (from requests->transformers) (2023.11.17)
```

The code in this notebook is actually a simplified version of the run_glue.py (https://github.com/huggingface/transformers/blob/master/examples/run_glue.py) example script from huggingface.

`run_glue.py` is a helpful utility which allows you to pick which GLUE benchmark task you want to run on, and which pre-trained model you want to use (you can see the list of possible models here (https://github.com/huggingface/transformers/blob/e6cff60b4cbc1158fbd6e4a1c3afda8dc22
It also supports using either the CPU, a single GPU, or multiple GPUs. It even supports using 16-bit precision if you want further speed up.

2. Loading CoLA Dataset

We'll use The Corpus of Linguistic Acceptability (CoLA) (https://nyu-mll.github.io/CoLA/) dataset for single sentence classification. It's a set of sentences labeled as grammatically correct or incorrect. It was first published in May of 2018, and is one of the tests included in the "GLUE Benchmark" on which models like BERT are competing.

2.1. Download & Extract

We'll use the `wget` package to download the dataset to the Colab instance's file system.

In [4]:
```
!pip install wget
```

```
Collecting wget
  Downloading wget-3.2.zip (10 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=96
55 sha256=1f7d30d74240fa009ed0a42e20edc6354ec6e0003f3fd9d6d658c1e25f
7e3da4
  Stored in directory: /root/.cache/pip/wheels/8b/f1/7f/5c94f0a7a505
ca1c81cd1d9208ae2064675d97582078e6c769
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

The dataset is hosted on GitHub in this repo: https://nyu-mll.github.io/CoLA/ (https://nyu-mll.github.io/CoLA/)

In [5]:
```
import wget
import os

print('Downloading dataset...')

# The URL for the dataset zip file.
url = 'https://nyu-mll.github.io/CoLA/cola_public_1.1.zip'

# Download the file (if we haven't already)
if not os.path.exists('./cola_public_1.1.zip'):
    wget.download(url, './cola_public_1.1.zip')
```

```
Downloading dataset...
```

Unzip the dataset to the file system. You can browse the file system of the Colab instance in the sidebar on the left.

In [6]:
```
# Unzip the dataset (if we haven't already)
if not os.path.exists('./cola_public/'):
    !unzip cola_public_1.1.zip
```

```
Archive:  cola_public_1.1.zip
   creating: cola_public/
  inflating: cola_public/README
   creating: cola_public/tokenized/
  inflating: cola_public/tokenized/in_domain_dev.tsv
  inflating: cola_public/tokenized/in_domain_train.tsv
  inflating: cola_public/tokenized/out_of_domain_dev.tsv
   creating: cola_public/raw/
  inflating: cola_public/raw/in_domain_dev.tsv
  inflating: cola_public/raw/in_domain_train.tsv
  inflating: cola_public/raw/out_of_domain_dev.tsv
```

2.2. Parse

We can see from the file names that both `tokenized` and `raw` versions of the data are available.

We can't use the pre-tokenized version because, in order to apply the pre-trained BERT, we *must* use the tokenizer provided by the model. This is because (1) the model has a specific, fixed vocabulary and (2) the BERT tokenizer has a particular way of handling out-of-vocabulary words.

We'll use pandas to parse the "in-domain" training set and look at a few of its properties and data points.

In [7]:
```
import pandas as pd

# Load the dataset into a pandas dataframe.
df = pd.read_csv("./cola_public/raw/in_domain_train.tsv", delimiter='

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

# Display 10 random rows from the data.
df.sample(10)
```

```
Number of training sentences: 8,551
```

Out[7]:

	sentence_source	label	label_notes	sentence
2503	l-93	1	NaN	The guests drank the teapot dry.
5580	c_13	1	NaN	Gary and Kevin ran themselves into exhaustion.
5164	ks08	1	NaN	Who was it who interviewed you?
7082	sgww85	1	NaN	Kim and Terry are happy.
6251	c_13	1	NaN	I want Jean to dance.
7780	ad03	0	*	We believed to be omnipotent.
1035	bc01	1	NaN	If we invite some philosopher, Max will be off...
2608	l-93	1	NaN	Lora buttered the toast.
3950	ks08	1	NaN	The school board leader asked the students a q...
6089	c_13	1	NaN	I have always loved peanut butter.

The two properties we actually care about are the the `sentence` and its `label`, which is referred to as the "acceptibility judgment" (0=unacceptable, 1=acceptable).

Here are five sentences which are labeled as not grammatically acceptible. Note how much more difficult this task is than something like sentiment analysis!

In [8]:
```
df.loc[df.label == 0].sample(5)[['sentence', 'label']]
```

Out[8]:

	sentence	label
3917	John met in the park a student.	0
3189	She always clad in black.	0
2348	Melissa searched a clue in the papers.	0
2844	Carrie touched at the cat.	0
4867	They investigated.	0

Let's extract the sentences and labels of our training set as numpy ndarrays.

In [9]:
```
# Get the lists of sentences and their labels.
sentences = df.sentence.values
labels = df.label.values
```

3. Tokenization & Input Formatting

In this section, we'll transform our dataset into the format that BERT can be trained on.

3.1. BERT Tokenizer

To feed our text to BERT, it must be split into tokens, and then these tokens must be mapped to their index in the tokenizer vocabulary.

The tokenization must be performed by the tokenizer included with BERT--the below cell will download this for us. We'll be using the "uncased" version here.

In [10]:
```
from transformers import BertTokenizer

# Load the BERT tokenizer.
print('Loading BERT tokenizer...')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_low
```

```
Loading BERT tokenizer...

tokenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]

vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]

tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]

config.json:   0%|          | 0.00/570 [00:00<?, ?B/s]
```

Let's apply the tokenizer to one sentence just to see the output.

In [11]:
```
# 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.tokeni
```

```
 Original:  Our friends won't buy this analysis, let alone the next
one we propose.
Tokenized:  ['our', 'friends', 'won', "'", 't', 'buy', 'this', 'anal
ysis', ',', 'let', 'alone', 'the', 'next', 'one', 'we', 'propose',
'.']
Token IDs:  [2256, 2814, 2180, 1005, 1056, 4965, 2023, 4106, 1010, 2
292, 2894, 1996, 2279, 2028, 2057, 16599, 1012]
```

When we actually convert all of our sentences, we'll use the `tokenize.encode` function to handle both steps, rather than calling `tokenize` and `convert_tokens_to_ids` separately.

Before we can do that, though, we need to talk about some of BERT's formatting requirements.

3.2. Required Formatting

The above code left out a few required formatting steps that we'll look at here.

Side Note: The input format to BERT seems "over-specified" to me... We are required to give it a number of pieces of information which seem redundant, or like they could easily be inferred from the data without us explicity providing it. But it is what it is, and I suspect it will make more sense once I have a deeper understanding of the BERT internals.

We are required to:

1. Add special tokens to the start and end of each sentence.
2. Pad & truncate all sentences to a single constant length.
3. Explicitly differentiate real tokens from padding tokens with the "attention mask".

Special Tokens

`[SEP]`

At the end of every sentence, we need to append the special `[SEP]` token.

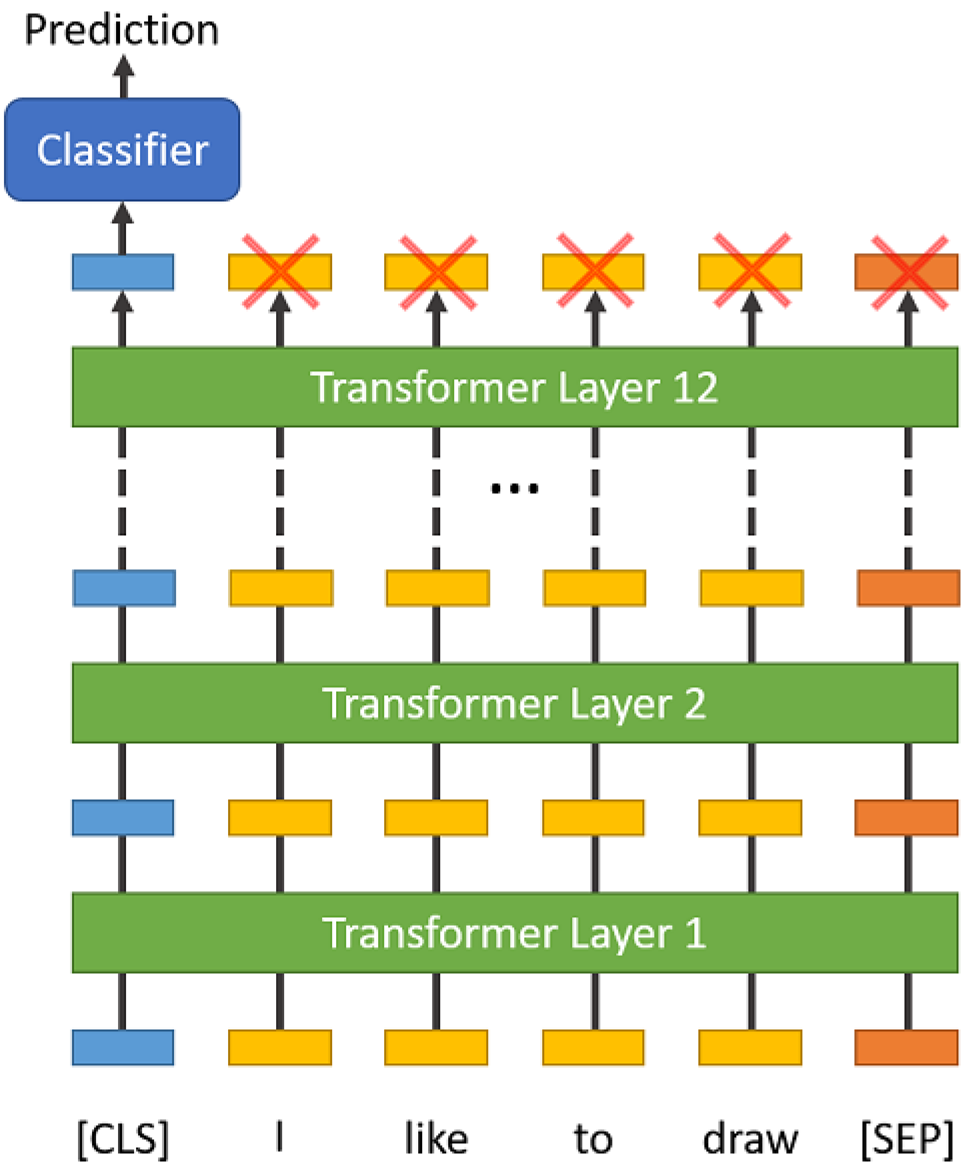
This token is an artifact of two-sentence tasks, where BERT is given two separate sentences and asked to determine something (e.g., can the answer to the question in sentence A be found in sentence B?).

I am not certain yet why the token is still required when we have only single-sentence input, but it is!

`[CLS]`

For classification tasks, we must prepend the special `[CLS]` token to the beginning of every sentence.

This token has special significance. BERT consists of 12 Transformer layers. Each transformer takes in a list of token embeddings, and produces the same number of embeddings on the output (but with the feature values changed, of course!).



On the output of the final (12th) transformer, *only the first embedding (corresponding to the [CLS] token) is used by the classifier*.

> "The first token of every sequence is always a special classification token (`[CLS]`). The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks." (from the BERT paper (https://arxiv.org/pdf/1810.04805.pdf))

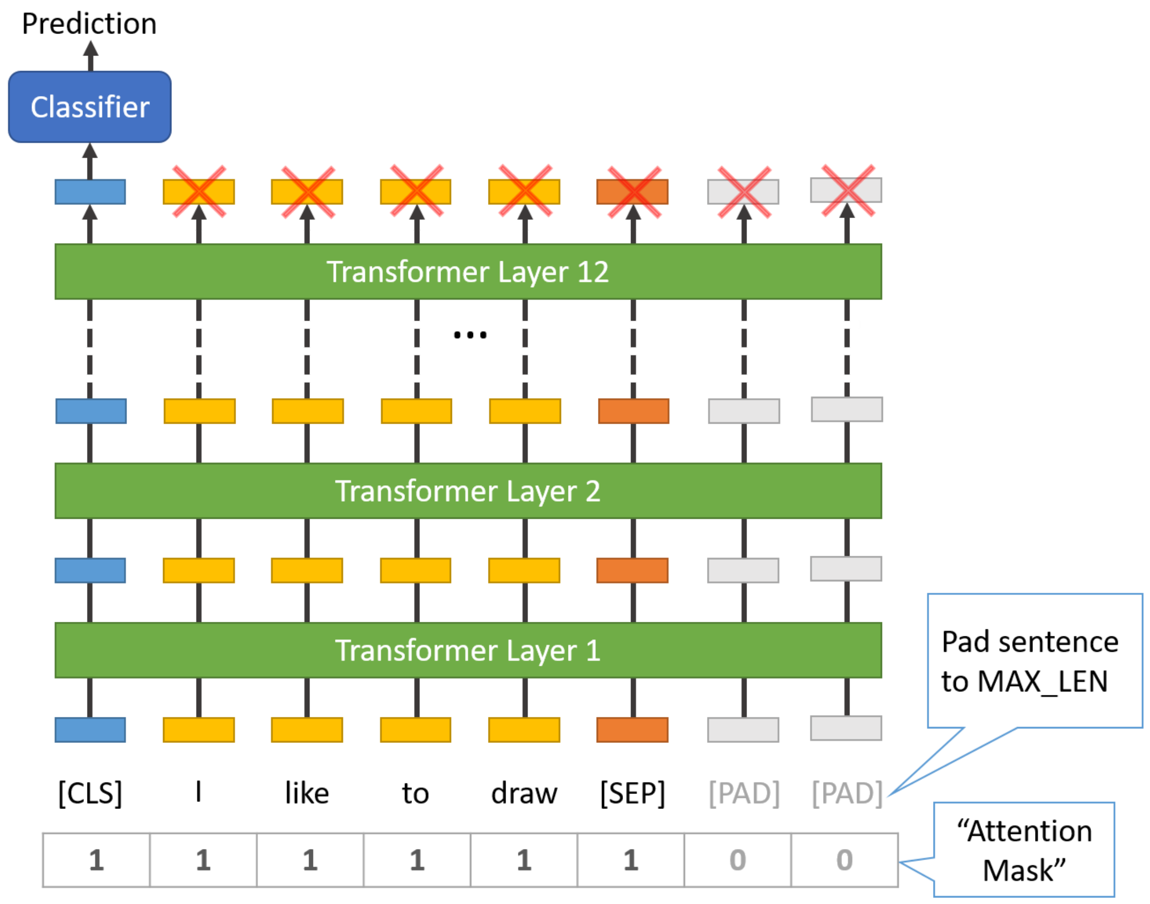
Sentence Length & Attention Mask

The sentences in our dataset obviously have varying lengths, so how does BERT handle this?

BERT has two constraints:

1. All sentences must be padded or truncated to a single, fixed length.
2. The maximum sentence length is 512 tokens.

Padding is done with a special `[PAD]` token, which is at index 0 in the BERT vocabulary. The below illustration demonstrates padding out to a "MAX_LEN" of 8 tokens.



The "Attention Mask" is simply an array of 1s and 0s indicating which tokens are padding and which aren't (seems kind of redundant, doesn't it?!). This mask tells the "Self-Attention" mechanism in BERT not to incorporate these PAD tokens into its interpretation of the sentence.

The maximum length does impact training and evaluation speed, however. For example, with a Tesla K80:

```
MAX_LEN = 128  -->  Training epochs take ~5:28 each
```

3.3. Tokenize Dataset

The transformers library provides a helpful `encode` function which will handle most of the parsing and data prep steps for us.

Before we are ready to encode our text, though, we need to decide on a **maximum sentence length** for padding / truncating to.

The below cell will perform one tokenization pass of the dataset in order to measure the maximum sentence length.

In [12]:
```
max_len = 0

# For every sentence...
for sent in sentences:

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

    # Update the maximum sentence length.
    max_len = max(max_len, len(input_ids))

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

```
Max sentence length:  47
```

Just in case there are some longer test sentences, I'll set the maximum length to 64.

Now we're ready to perform the real tokenization.

The `tokenizer.encode_plus` function combines multiple steps for us:

1. Split the sentence into tokens.
2. Add the special `[CLS]` and `[SEP]` tokens.
3. Map the tokens to their IDs.
4. Pad or truncate all sentences to the same length.
5. Create the attention masks which explicitly differentiate real tokens from `[PAD]` tokens.

The first four features are in `tokenizer.encode`, but I'm using `tokenizer.encode_plus` to get the fifth item (attention masks). Documentation is here (https://huggingface.co/transformers/main_classes/tokenizer.html?highlight=encode_plus#transformers.PreTrainedTokenizer.encode_plus).

In [13]:
```
# Tokenize all of the sentences and map the tokens to thier word IDs.
input_ids = []
attention_masks = []

# For every sentence...
for sent in sentences:
    # `encode_plus` 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.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to enco
                        add_special_tokens = True, # Add '[CLS]' and
                        max_length = 64,           # Pad & truncate a
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct a
                        return_tensors = 'pt',     # Return pytorch t
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-
    attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)

# Print sentence 0, now as a list of IDs.
print('Original: ', sentences[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 explicitly truncat
e examples to max length. Defaulting to 'longest_first' truncation s
trategy. If you encode pairs of sequences (GLUE-style) with the toke
nizer you can select this strategy more precisely by providing a spe
cific strategy to `truncation`.
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_ut
ils_base.py:2614: FutureWarning: The `pad_to_max_length` argument is
deprecated and will be removed in a future version, use `padding=Tru
e` or `padding='longest'` to pad to the longest sequence in the batc
h, or use `padding='max_length'` to pad to a max length. In this cas
e, you can give a specific length with `max_length` (e.g. `max_lengt
h=45`) or leave max_length to None to pad to the maximal input size
of the model (e.g. 512 for Bert).
  warnings.warn(
Original:  Our friends won't buy this analysis, let alone the next o
ne we propose.
Token IDs: tensor([  101,  2256,  2814,  2180,  1005,  1056,  4965,
2023,  4106,  1010,
         2292,  2894,  1996,  2279,  2028,  2057, 16599,  1012,   10
2,     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,     0,     0,     0,
0,     0,
            0,     0,     0,     0])
```

3.4. Training & Validation Split

Divide up our training set to use 90% for training and 10% for validation.

In [14]:
```
from torch.utils.data import TensorDataset, random_split

# Combine the training inputs into a TensorDataset.
dataset = TensorDataset(input_ids, attention_masks, labels)

# Create a 90-10 train-validation split.

# Calculate the number of samples to include in each set.
train_size = int(0.9 * len(dataset))
val_size = len(dataset) - train_size

# Divide the dataset by randomly selecting samples.
train_dataset, val_dataset = random_split(dataset, [train_size, val_s

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

```
7,695 training samples
  856 validation samples
```

We'll also create an iterator for our dataset using the torch DataLoader class. This helps save on memory during training because, unlike a for loop, with an iterator the entire dataset does not need to be loaded into memory.

In [15]:
```
from torch.utils.data import DataLoader, RandomSampler, SequentialSam

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

# Create the DataLoaders for our training and validation sets.
# We'll take training samples in random order.
train_dataloader = DataLoader(
            train_dataset,  # The training samples.
            sampler = RandomSampler(train_dataset), # Select batches
            batch_size = batch_size # Trains with this batch size.
        )

# For validation the order doesn't matter, so we'll just read them se
validation_dataloader = DataLoader(
            val_dataset, # The validation samples.
            sampler = SequentialSampler(val_dataset), # Pull out batc
            batch_size = batch_size # Evaluate with this batch size.
        )
```

4. Train Our Classification Model

Now that our input data is properly formatted, it's time to fine tune the BERT model.

4.1. BertForSequenceClassification

For this task, we first want to modify the pre-trained BERT model to give outputs for classification, and then we want to continue training the model on our dataset until that the entire model, end-to-end, is well-suited for our task.

Thankfully, the huggingface pytorch implementation includes a set of interfaces designed for a variety of NLP tasks. Though these interfaces are all built on top of a trained BERT model, each has different top layers and output types designed to accomodate their specific NLP task.

Here is the current list of classes provided for fine-tuning:

- BertModel
- BertForPreTraining
- BertForMaskedLM
- BertForNextSentencePrediction
- **BertForSequenceClassification** - The one we'll use.
- BertForTokenClassification
- BertForQuestionAnswering

We'll be using BertForSequenceClassification (https://huggingface.co/transformers/v2.2.0/model_doc/bert.html#bertforsequenceclassificat
This is the normal BERT model with an added single linear layer on top for classification that we will use as a sentence classifier. As we feed input data, the entire pre-trained BERT model and the additional untrained classification layer is trained on our specific task.

OK, let's load BERT! There are a few different pre-trained BERT models available. "bert-base-uncased" means the version that has only lowercase letters ("uncased") and is the smaller version of the two ("base" vs "large").

The documentation for `from_pretrained` can be found here (https://huggingface.co/transformers/v2.2.0/main_classes/model.html#transformers.PreTrain
with the additional parameters defined here (https://huggingface.co/transformers/v2.2.0/main_classes/configuration.html#transformers.P

In [16]:
```
from transformers import BertForSequenceClassification, AdamW, BertCo

# Load BertForSequenceClassification, the pretrained BERT model with
# linear classification layer on top.
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", # Use the 12-layer BERT model, with an uncas
    num_labels = 2, # The number of output labels--2 for binary class
                    # You can increase this for multi-class tasks.
    output_attentions = False, # Whether the model returns attentions
    output_hidden_states = False, # Whether the model returns all hid
)

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

```
model.safetensors:   0%|          | 0.00/440M [00:00<?, ?B/s]

Some weights of BertForSequenceClassification were not initialized f
rom the model checkpoint at bert-base-uncased and are newly initiali
zed: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be abl
e to use it for predictions and inference.
```

Out[16]:

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=T
rue)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bia
s=True)
              (key): Linear(in_features=768, out_features=768, bias=
True)
              (value): Linear(in_features=768, out_features=768, bia
s=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bia
s=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_
affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias
=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias
=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_af
fine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

Just for curiosity's sake, we can browse all of the model's parameters by name here.

In the below cell, I've printed out the names and dimensions of the weights for:

1. The embedding layer.
2. The first of the twelve transformers.
3. The output layer.

In [17]:
```
# Get all of the model's parameters as a list of tuples.
params = list(model.named_parameters())

print('The BERT model has {:} different named parameters.\n'.format(l

print('==== Embedding Layer ====\n')

for p in params[0:5]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))

print('\n==== First Transformer ====\n')

for p in params[5:21]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))

print('\n==== Output Layer ====\n')

for p in params[-4:]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
```

```
The BERT model has 201 different named parameters.

==== Embedding Layer ====

bert.embeddings.word_embeddings.weight                  (30522, 768)
bert.embeddings.position_embeddings.weight                (512, 768)
bert.embeddings.token_type_embeddings.weight                (2, 768)
bert.embeddings.LayerNorm.weight                              (768,)
bert.embeddings.LayerNorm.bias                                (768,)

==== First Transformer ====

bert.encoder.layer.0.attention.self.query.weight          (768, 768)
bert.encoder.layer.0.attention.self.query.bias                (768,)
bert.encoder.layer.0.attention.self.key.weight            (768, 768)
bert.encoder.layer.0.attention.self.key.bias                  (768,)
bert.encoder.layer.0.attention.self.value.weight          (768, 768)
bert.encoder.layer.0.attention.self.value.bias                (768,)
bert.encoder.layer.0.attention.output.dense.weight        (768, 768)
bert.encoder.layer.0.attention.output.dense.bias              (768,)
bert.encoder.layer.0.attention.output.LayerNorm.weight        (768,)
bert.encoder.layer.0.attention.output.LayerNorm.bias          (768,)
bert.encoder.layer.0.intermediate.dense.weight           (3072, 768)
bert.encoder.layer.0.intermediate.dense.bias                 (3072,)
bert.encoder.layer.0.output.dense.weight                 (768, 3072)
bert.encoder.layer.0.output.dense.bias                        (768,)
bert.encoder.layer.0.output.LayerNorm.weight                  (768,)
bert.encoder.layer.0.output.LayerNorm.bias                    (768,)

==== Output Layer ====

bert.pooler.dense.weight                                  (768, 768)
bert.pooler.dense.bias                                        (768,)
classifier.weight                                           (2, 768)
classifier.bias                                                 (2,)
```

4.2. Optimizer & Learning Rate Scheduler

Now that we have our model loaded we need to grab the training hyperparameters from within the stored model.

For the purposes of fine-tuning, the authors recommend choosing from the following values (from Appendix A.3 of the BERT paper (https://arxiv.org/pdf/1810.04805.pdf)):

> - **Batch size:** 16, 32

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

We chose:

- Batch size: 32 (set when creating our DataLoaders)
- Learning rate: 2e-5
- Epochs: 4 (we'll see that this is probably too many...)

The epsilon parameter `eps = 1e-8` is "a very small number to prevent any division by zero in the implementation" (from here (https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/)).

You can find the creation of the AdamW optimizer in `run_glue.py` here (https://github.com/huggingface/transformers/blob/5bfcd0485ece086ebcbed2d00881303796

In [18]:
```
# Note: AdamW is a class from the huggingface library (as opposed to
# I believe the 'W' stands for 'Weight Decay fix"
optimizer = AdamW(model.parameters(),
                  lr = 2e-5, # args.learning_rate - default is 5e-5,
                  eps = 1e-8 # args.adam_epsilon  - default is 1e-8.
                )
```

```
/usr/local/lib/python3.10/dist-packages/transformers/optimization.p
y:411: FutureWarning: This implementation of AdamW is deprecated and
will be removed in a future version. Use the PyTorch implementation
torch.optim.AdamW instead, or set `no_deprecation_warning=True` to d
isable this warning
  warnings.warn(
```

In [19]:
```
from transformers import get_linear_schedule_with_warmup

# Number of training epochs. The BERT authors recommend between 2 and
# We chose to run for 4, but we'll see later that this may be over-fi
# training data.
epochs = 4

# Total number of training steps is [number of batches] x [number of 
# (Note that this is not the same as the number of training samples).
total_steps = len(train_dataloader) * epochs

# Create the learning rate scheduler.
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0, # D
                                            num_training_steps = tota
```

4.3. Training Loop

Below is our training loop. There's a lot going on, but fundamentally for each pass in our loop we have a trianing phase and a validation phase.

> *Thank you to [Stas Bekman (https://ca.linkedin.com/in/stasbekman)](https://ca.linkedin.com/in/stasbekman) for contributing the insights and code for using validation loss to detect over-fitting!*

Training:

- Unpack our data inputs and labels
- Load data onto the GPU for acceleration
- Clear out the gradients calculated in the previous pass.
  - In pytorch the gradients accumulate by default (useful for things like RNNs) unless you explicitly clear them out.
- Forward pass (feed input data through the network)
- Backward pass (backpropagation)
- Tell the network to update parameters with optimizer.step()
- Track variables for monitoring progress

Evalution:

- Unpack our data inputs and labels
- Load data onto the GPU for acceleration
- Forward pass (feed input data through the network)
- Compute loss on our validation data and track variables for monitoring progress

Pytorch hides all of the detailed calculations from us, but we've commented the code to point out which of the above steps are happening on each line.

> *PyTorch also has some [beginner tutorials](https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-)*
> *(https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-*

Define a helper function for calculating accuracy.

In [20]:
```
import numpy as np

# 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)
```

Helper function for formatting elapsed times as `hh:mm:ss`

In [21]:
```
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))
```

We're ready to kick off the training!

In [22]:
```
import random
import numpy as np

# This training code is based on the `run_glue.py` script here:
# https://github.com/huggingface/transformers/blob/5bfcd0485ece086ebc

# 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)

# We'll store a number of quantities such as training and validation
# validation accuracy, and timings.
training_stats = []

# Measure the total training time for the whole run.
total_t0 = time.time()

# 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, epc
    print('Training...')

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

    # Reset the total loss for this epoch.
    total_train_loss = 0

    # Put the model into training mode. Don't be mislead--the call tc
    # `train` just changes the *mode*, it doesn't *perform* the trair
    # `dropout` and `batchnorm` layers behave differently during trai
    # vs. test (source: https://stackoverflow.com/questions/51433378/
    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: {:}.'.form
        # Unpack this training batch from our dataloader.
        #
        # As we unpack the batch, we'll also copy each tensor to 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 per
        # backward pass. PyTorch doesn't do this automatically becaus
        # accumulating the gradients is "convenient while training RN
        # (source: https://stackoverflow.com/questions/48001598/why-d
        model.zero_grad()

        # Perform a forward pass (evaluate the model on this training
        # In PyTorch, calling `model` will in turn call the model's `
        # function and pass down the arguments. The `forward` functio
        # documented here:
        # https://huggingface.co/transformers/model_doc/bert.html#ber
        # The results are returned in a results object, documented he
        # https://huggingface.co/transformers/main_classes/output.htm
        # Specifically, we'll get the loss (because we provided label
        # "logits"--the model outputs prior to activation.
        result = model(b_input_ids,
                       token_type_ids=None,
                       attention_mask=b_input_mask,
                       labels=b_labels,
                       return_dict=True)

        loss = result.loss
        logits = result.logits

        # Accumulate the training loss over all of the batches so tha
        # calculate the average loss at the end. `loss` is a Tensor c
        # single value; the `.item()` function just returns the Pytho
        # from the tensor.
        total_train_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 gradie
        # The optimizer dictates the "update rule"--how the parameter
        # modified based on their gradients, the learning rate, etc.
        optimizer.step()

        # Update the learning rate.
        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))

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

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

    t0 = time.time()

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

    # Tracking variables
    total_eval_accuracy = 0
    total_eval_loss = 0
    nb_eval_steps = 0

    # Evaluate data for one epoch
    for batch in validation_dataloader:

        # Unpack this training batch from our dataloader.
        #
        # As we unpack the batch, we'll also copy each tensor to the
        # 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)

        # Tell pytorch not to bother with constructing the compute gr
        # the forward pass, since this is only needed for backprop (t
        with torch.no_grad():

            # Forward pass, calculate logit predictions.
            # token_type_ids is the same as the "segment ids", which
            # differentiates sentence 1 and 2 in 2-sentence tasks.
            result = model(b_input_ids,
                           token_type_ids=None,
                           attention_mask=b_input_mask,
                           labels=b_labels,
                           return_dict=True)

        # Get the loss and "logits" output by the model. The "logits"
        # output values prior to applying an activation function like
        # softmax.
        loss = result.loss
        logits = result.logits

        # Accumulate the validation loss.
        total_eval_loss += loss.item()

        # 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, an
        # accumulate it over all batches.
        total_eval_accuracy += flat_accuracy(logits, label_ids)


    # Report the final accuracy for this validation run.
    avg_val_accuracy = total_eval_accuracy / len(validation_dataloade
    print("  Accuracy: {0:.2f}".format(avg_val_accuracy))

    # Calculate the average loss over all of the batches.
    avg_val_loss = total_eval_loss / len(validation_dataloader)

    # Measure how long the validation run took.
    validation_time = format_time(time.time() - t0)

    print("  Validation Loss: {0:.2f}".format(avg_val_loss))
    print("  Validation took: {:}".format(validation_time))

    # Record all statistics from this epoch.
    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.tim
======== Epoch 1 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:14.
  Batch    80  of    241.    Elapsed: 0:00:27.
  Batch   120  of    241.    Elapsed: 0:00:41.
  Batch   160  of    241.    Elapsed: 0:00:54.
  Batch   200  of    241.    Elapsed: 0:01:08.
  Batch   240  of    241.    Elapsed: 0:01:22.

  Average training loss: 0.50
  Training epcoh took: 0:01:22

Running Validation...
  Accuracy: 0.82
  Validation Loss: 0.44
  Validation took: 0:00:03

======== Epoch 2 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:14.
  Batch    80  of    241.    Elapsed: 0:00:27.
  Batch   120  of    241.    Elapsed: 0:00:40.
  Batch   160  of    241.    Elapsed: 0:00:54.
  Batch   200  of    241.    Elapsed: 0:01:08.
  Batch   240  of    241.    Elapsed: 0:01:22.

  Average training loss: 0.30
  Training epcoh took: 0:01:22

Running Validation...
  Accuracy: 0.81
  Validation Loss: 0.50
  Validation took: 0:00:03

======== Epoch 3 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:14.
  Batch    80  of    241.    Elapsed: 0:00:27.
  Batch   120  of    241.    Elapsed: 0:00:41.
  Batch   160  of    241.    Elapsed: 0:00:54.
  Batch   200  of    241.    Elapsed: 0:01:08.
  Batch   240  of    241.    Elapsed: 0:01:22.

  Average training loss: 0.19
  Training epcoh took: 0:01:22

Running Validation...
  Accuracy: 0.82
  Validation Loss: 0.54
  Validation took: 0:00:03

======== Epoch 4 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:14.
  Batch    80  of    241.    Elapsed: 0:00:27.
  Batch   120  of    241.    Elapsed: 0:00:41.
  Batch   160  of    241.    Elapsed: 0:00:54.
  Batch   200  of    241.    Elapsed: 0:01:08.
  Batch   240  of    241.    Elapsed: 0:01:22.

  Average training loss: 0.13
  Training epcoh took: 0:01:22

Running Validation...
  Accuracy: 0.83
  Validation Loss: 0.62
  Validation took: 0:00:03

Training complete!
Total training took 0:05:41 (h:mm:ss)
```

Let's view the summary of the training process.

In [ ]:
```
import pandas as pd

# Display floats with two decimal places.
pd.set_option('precision', 2)

# Create a DataFrame from our training statistics.
df_stats = pd.DataFrame(data=training_stats)

# Use the 'epoch' as the row index.
df_stats = df_stats.set_index('epoch')

# A hack to force the column headers to wrap.
#df = df.style.set_table_styles([dict(selector="th",props=[('max-widt

# Display the table.
df_stats
```

Out[23]:

	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
epoch					
1	0.51	0.44	0.80	0:01:21	0:00:03
2	0.33	0.47	0.82	0:01:22	0:00:03
3	0.21	0.51	0.82	0:01:23	0:00:03
4	0.15	0.59	0.82	0:01:23	0:00:03

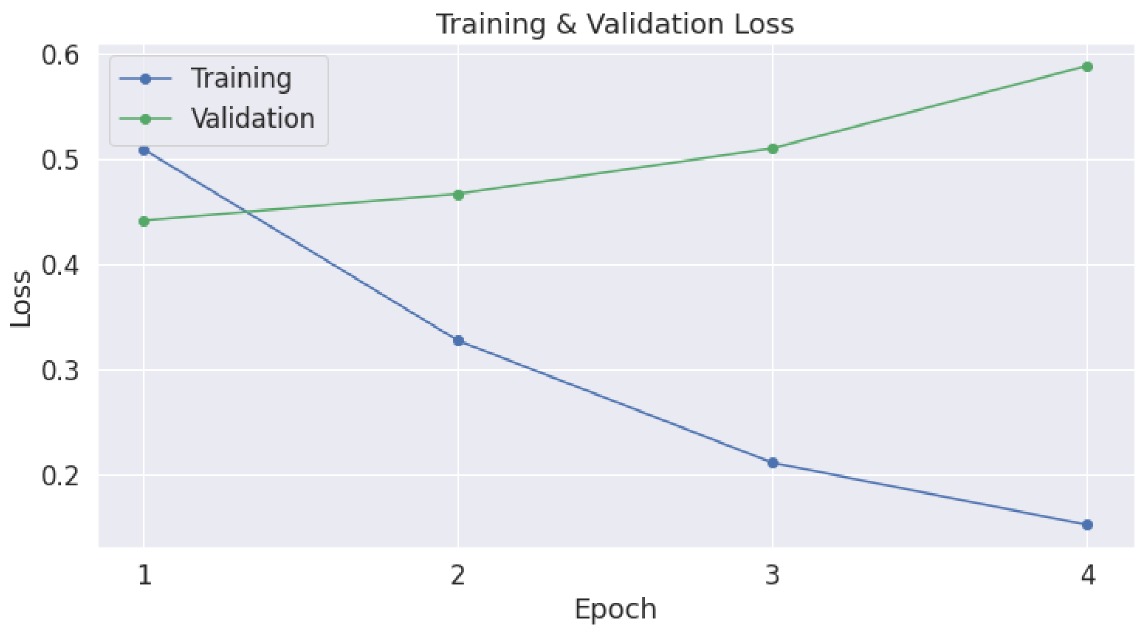
Notice that, while the the training loss is going down with each epoch, the validation loss is increasing! This suggests that we are training our model too long, and it's over-fitting on the training data.

(For reference, we are using 7,695 training samples and 856 validation samples).

Validation Loss is a more precise measure than accuracy, because with accuracy we don't care about the exact output value, but just which side of a threshold it falls on.

If we are predicting the correct answer, but with less confidence, then validation loss will

In [ ]:
```python
import matplotlib.pyplot as plt
% matplotlib inline

import seaborn as sns

# Use plot styling from seaborn.
sns.set(style='darkgrid')

# Increase the plot size and font size.
sns.set(font_scale=1.5)
plt.rcParams["figure.figsize"] = (12,6)

# Plot the learning curve.
plt.plot(df_stats['Training Loss'], 'b-o', label="Training")
plt.plot(df_stats['Valid. Loss'], 'g-o', label="Validation")

# Label the plot.
plt.title("Training & Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.xticks([1, 2, 3, 4])

plt.show()
```



5. Performance On Test Set

5.1. Data Preparation

We'll need to apply all of the same steps that we did for the training data to prepare our test data set.

In [ ]:
```
import pandas as pd

# Load the dataset into a pandas dataframe.
df = pd.read_csv("./cola_public/raw/out_of_domain_dev.tsv", delimiter

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

# Create sentence and label lists
sentences = df.sentence.values
labels = df.label.values

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

# For every sentence...
for sent in sentences:
    # `encode_plus` 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.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to enco
                        add_special_tokens = True, # Add '[CLS]' and 
                        max_length = 64,           # Pad & truncate a
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct a
                        return_tensors = 'pt',     # Return pytorch t
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-
    attention_masks.append(encoded_dict['attention_mask'])

# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)

# Set the batch size.
batch_size = 32

# Create the DataLoader.
prediction_data = TensorDataset(input_ids, attention_masks, labels)
prediction_sampler = SequentialSampler(prediction_data)
prediction_dataloader = DataLoader(prediction_data, sampler=predictic
```

```
Number of test sentences: 516
```

```
/usr/local/lib/python3.7/dist-packages/transformers/tokenization_uti
ls_base.py:2269: FutureWarning: The `pad_to_max_length` argument is 
deprecated and will be removed in a future version, use `padding=Tru
e` or `padding='longest'` to pad to the longest sequence in the batc
h, or use `padding='max_length'` to pad to a max length. In this cas
e, you can give a specific length with `max_length` (e.g. `max_lengt
h=45`) or leave max_length to None to pad to the maximal input size 
of the model (e.g. 512 for Bert).
  FutureWarning,
```

5.2. Evaluate on Test Set

With the test set prepared, we can apply our fine-tuned model to generate predictions on the test set.

In [ ]:
```
# Prediction on test set

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

# Put model in evaluation mode
model.eval()

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

# 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 memor
  # speeding up prediction
  with torch.no_grad():
      # Forward pass, calculate logit predictions.
      result = model(b_input_ids,
                     token_type_ids=None,
                     attention_mask=b_input_mask,
                     return_dict=True)

  logits = result.logits

  # 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)

print('    DONE.')
```

```
Predicting labels for 516 test sentences...
    DONE.
```

Conclusion

This post demonstrates that with a pre-trained BERT model you can quickly and effectively create a high quality model with minimal effort and training time using the Pytorch interface, regardless of the specific NLP task you are interested in.

In [50]:
```
!pip install transformers tensorflow
from transformers import BertTokenizer, TFBertForSequenceClassificati
import tensorflow as tf
import pandas as pd

train_df = pd.read_csv('7_2/rte_train.csv')
val_df = pd.read_csv('7_2/rte_val.csv')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

def convert_example_to_feature(review):
    return tokenizer.encode_plus(review['sentence1'], review['sentenc
                                 add_special_tokens=True, max_length=1
                                 pad_to_max_length=True, return_attent

train_df['features'] = train_df.apply(convert_example_to_feature, axi
val_df['features'] = val_df.apply(convert_example_to_feature, axis=1)
```

```
Requirement already satisfied: transformers in /usr/local/lib/py
thon3.10/dist-packages (4.35.2)
Requirement already satisfied: tensorflow in /usr/local/lib/pyth
on3.10/dist-packages (2.15.0)
Requirement already satisfied: filelock in /usr/local/lib/python
3.10/dist-packages (from transformers) (3.13.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in /
usr/local/lib/python3.10/dist-packages (from transformers) (0.1
9.4)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/pyt
hon3.10/dist-packages (from transformers) (1.23.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/li
b/python3.10/dist-packages (from transformers) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/pyt
hon3.10/dist-packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/l
ib/python3.10/dist-packages (from transformers) (2023.6.3)
Requirement already satisfied: requests in /usr/local/lib/python
3.10/dist-packages (from transformers) (2.31.0)
```

In [51]:
```
def map_example_to_dict(input_ids, attention_masks, label):
    return {"input_ids": input_ids, "attention_mask": attention_masks

def encode_examples(ds, limit=-1):
    input_ids_list, attention_mask_list, label_list = [], [], []
    if limit > 0:
        ds = ds[:limit]
    for index, row in ds.iterrows():
        input_ids_list.append(row['features']['input_ids'])
        attention_mask_list.append(row['features']['attention_mask'])
        label_list.append(row['label'])
    return tf.data.Dataset.from_tensor_slices((input_ids_list, attent

train_data = encode_examples(train_df).shuffle(100).batch(16)
val_data = encode_examples(val_df).batch(16)
```

In [54]:
```
model = TFBertForSequenceClassification.from_pretrained('bert-base-ur
optimizer = tf.keras.optimizers.Adam(learning_rate=2e-5)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True
metric = tf.keras.metrics.SparseCategoricalAccuracy('accuracy')
model.compile(optimizer=optimizer, loss=loss, metrics=[metric])
model.fit(train_data, epochs=4)
```

```
All PyTorch model weights were used when initializing TFBertForSeque
nceClassification.

Some weights or buffers of the TF 2.0 model TFBertForSequenceClassif
ication were not initialized from the PyTorch model and are newly in
itialized: ['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be abl
e to use it for predictions and inference.

Epoch 1/4
156/156 [==============================] - 125s 409ms/step - loss:
0.6960 - accuracy: 0.5116
Epoch 2/4
156/156 [==============================] - 64s 408ms/step - loss: 0.
6630 - accuracy: 0.6189
Epoch 3/4
156/156 [==============================] - 64s 411ms/step - loss: 0.
5515 - accuracy: 0.7277
Epoch 4/4
156/156 [==============================] - 65s 414ms/step - loss: 0.
3537 - accuracy: 0.8590
```

Out[54]:
```
<keras.src.callbacks.History at 0x7c29e066dbd0>
```

In [55]:
```python
p1_result = model.evaluate(val_data, return_dict=True)
print(f'Accuracy: {p1_result["accuracy"]}')
```

```
18/18 [==============================] - 5s 138ms/step - loss: 0.846
6 - accuracy: 0.6137
Accuracy: 0.6137183904647827
```

In [57]:
```
# p2
xnli_de_df = pd.read_csv('7_2/xnli_de_val.csv')

def convert_xnli_to_feature(row):
    return tokenizer.encode_plus(row['sentence1'], row['sentence2'],
                                add_special_tokens=True, max_length=1
                                pad_to_max_length=True, return_attent

xnli_de_df['features'] = xnli_de_df.apply(convert_xnli_to_feature, ax

xnli_de_data = encode_examples(xnli_de_df).batch(16)

result = model.evaluate(xnli_de_data, return_dict=True)
print(f'Zero-shot Accuracy on German Data: {result["accuracy"]}')
```

```
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_ut
ils_base.py:2614: FutureWarning: The `pad_to_max_length` argument is
deprecated and will be removed in a future version, use `padding=Tru
e` or `padding='longest'` to pad to the longest sequence in the batc
h, or use `padding='max_length'` to pad to a max length. In this cas
e, you can give a specific length with `max_length` (e.g. `max_lengt
h=45`) or leave max_length to None to pad to the maximal input size
of the model (e.g. 512 for Bert).
  warnings.warn(
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.
Be aware, overflowing tokens are not returned for the setting you ha
ve chosen, i.e. sequence pairs with the 'longest_first' truncation s
trategy. So the returned list will always be empty even if some toke
ns have been removed.

156/156 [==============================] - 23s 144ms/step - loss: na
n - accuracy: 0.3771
Zero-shot Accuracy on German Data: 0.3771084249019623
```

In [59]:
```
!pip install sentencepiece
```

```
Collecting sentencepiece
  Downloading sentencepiece-0.1.99-cp310-cp310-manylinux_2_17_x86_6
4.manylinux2014_x86_64.whl (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 8.9 MB/s et
a 0:00:00
Installing collected packages: sentencepiece
Successfully installed sentencepiece-0.1.99
```

In [1]:
```
# p3
from transformers import XLMRobertaTokenizer, TFXLMRobertaForSequence
import tensorflow as tf
import pandas as pd
#from sentencepiece import XLMRobertaTokenizer


# Load and preprocess the data
train_df = pd.read_csv('7_2/rte_train.csv')
val_df = pd.read_csv('7_2/rte_val.csv')

# Initialize the tokenizer
tokenizer = XLMRobertaTokenizer.from_pretrained('xlm-roberta-base')

def convert_example_to_feature(review):
    return tokenizer.encode_plus(review['sentence1'], review['sentenc
                                 add_special_tokens=True, max_length=1
                                 pad_to_max_length=True, return_attent

train_df['features'] = train_df.apply(convert_example_to_feature, axi
val_df['features'] = val_df.apply(convert_example_to_feature, axis=1)

# Function to map examples to TensorFlow dataset
def map_example_to_dict(input_ids, attention_masks, label):
    return {"input_ids": input_ids, "attention_mask": attention_masks

def encode_examples(ds, limit=-1):
    input_ids_list, attention_mask_list, label_list = [], [], []
    if limit > 0:
        ds = ds[:limit]
    for index, row in ds.iterrows():
        input_ids_list.append(row['features']['input_ids'])
        attention_mask_list.append(row['features']['attention_mask'])
        label_list.append(row['label'])
    return tf.data.Dataset.from_tensor_slices((input_ids_list, attent

# Create TensorFlow datasets
train_data = encode_examples(train_df).shuffle(100).batch(16)
val_data = encode_examples(val_df).batch(16)

# Initialize and compile the model
model = TFXLMRobertaForSequenceClassification.from_pretrained('xlm-ro
optimizer = tf.keras.optimizers.Adam(learning_rate=2e-5)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True
metric = tf.keras.metrics.SparseCategoricalAccuracy('accuracy')
model.compile(optimizer=optimizer, loss=loss, metrics=[metric])

# Train the model
model.fit(train_data, epochs=3)

# Evaluate the model
result = model.evaluate(val_data, return_dict=True)
print(f'Accuracy: {result["accuracy"]}')
```

```
sentencepiece.bpe.model:   0%|          | 0.00/5.07M [00:00<?, ?
B/s]

tokenizer.json:   0%|          | 0.00/9.10M [00:00<?, ?B/s]

config.json:   0%|          | 0.00/615 [00:00<?, ?B/s]

Truncation was not explicitly activated but `max_length` is prov
ided a specific value, please use `truncation=True` to explicitl
y truncate examples to max length. Defaulting to 'longest_first'
truncation strategy. If you encode pairs of sequences (GLUE-styl
e) with the tokenizer you can select this strategy more precisel
y by providing a specific strategy to `truncation`.
/usr/local/lib/python3.10/dist-packages/transformers/tokenizatio
n_utils_base.py:2614: FutureWarning: The `pad_to_max_length` arg
ument is deprecated and will be removed in a future version, use
`padding=True` or `padding='longest'` to pad to the longest sequ
ence in the batch, or use `padding='max_length'` to pad to a max
length. In this case, you can give a specific length with `max_l
ength` (e.g. `max_length=45`) or leave max_length to None to pad
```

In [2]:
```
# p4
import pandas as pd
from transformers import XLMRobertaTokenizer
import tensorflow as tf

# Load the German subset of the XNLI dataset
xnli_de_df = pd.read_csv('7_2/xnli_de_val.csv')

# Initialize the tokenizer (if not already done)
tokenizer = XLMRobertaTokenizer.from_pretrained('xlm-roberta-base')

# Preprocess the German data
def convert_xnli_to_feature(row):
    return tokenizer.encode_plus(row['sentence1'], row['sentence2'],
                                add_special_tokens=True, max_length=1
                                pad_to_max_length=True, return_attent

xnli_de_df['features'] = xnli_de_df.apply(convert_xnli_to_feature, ax

# Function to encode examples for TensorFlow
def map_example_to_dict(input_ids, attention_masks, label):
    return {"input_ids": input_ids, "attention_mask": attention_masks

def encode_examples(ds, limit=-1):
    input_ids_list, attention_mask_list, label_list = [], [], []
    if limit > 0:
        ds = ds[:limit]
    for index, row in ds.iterrows():
        input_ids_list.append(row['features']['input_ids'])
        attention_mask_list.append(row['features']['attention_mask'])
        label_list.append(row['label'])
    return tf.data.Dataset.from_tensor_slices((input_ids_list, attent

# Create TensorFlow dataset for the German data
xnli_de_data = encode_examples(xnli_de_df).batch(16)

# Evaluate the model on the German data
result = model.evaluate(xnli_de_data, return_dict=True)
print(f'Accuracy: {result["accuracy"]}')
```

```
Truncation was not explicitly activated but `max_length` is provided
a specific value, please use `truncation=True` to explicitly truncat
e examples to max length. Defaulting to 'longest_first' truncation s
trategy. If you encode pairs of sequences (GLUE-style) with the toke
nizer you can select this strategy more precisely by providing a spe
cific strategy to `truncation`.
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_ut
ils_base.py:2614: FutureWarning: The `pad_to_max_length` argument is
deprecated and will be removed in a future version, use `padding=Tru
e` or `padding='longest'` to pad to the longest sequence in the batc
h, or use `padding='max_length'` to pad to a max length. In this cas
e, you can give a specific length with `max_length` (e.g. `max_lengt
h=45`) or leave max_length to None to pad to the maximal input size
of the model (e.g. 512 for Bert).
  warnings.warn(

156/156 [==============================] - 21s 136ms/step - loss: na
n - accuracy: 0.3771
Accuracy: 0.3771084249019623
```

Do it yourself

Modify the above tutorial to train a Twitter sentiment classification model (you can keep all the hyperparameters as before for this homework, with epochs=4). In reality, fine-tuning the hyperparameters for your data and task (e.g., using validation step to pick the best number of epochs) might improve performance.

1. (10 pts) Fine-tune English BERT with **all** of your examples in rte-train.csv and report the accuracy on rte-val.csv. The tutorial was for a sentence classification task. When this becomes a sentence pair classification task the preprocessing slightly changes. The tutorial pads special tokens to the beginning and end of a sentence now you have two sentences that you'll pad in a specific way. Look at the original BERT (https://arxiv.org/pdf/1810.04805.pdf) paper to see how you should do it.

   > Accuracy: 0.6137183904647827

2. (5 pts) Report the accuracy of this fine-tuned English BERT on the German subset of XNLI (this is zero-shot accuracy of the model trained on the English data and test on the German data). The data is provided in xnli-de-val.csv

   > Accuracy: 0.3771084249019623

3. (10 pts) Modify the code to use multilingual (XLM-Roberta) model (xlm-roberta-base), keeping the same hyperparameters as before. Fine-tune the multilingual model with **all** of your examples in rte-train.csv and report the accuracy on rte-val.csv.

   > Accuracy: 0.5234656929969788

4. (5 pts) Report the accuracy of the multilingual model on German subset of XNLI (this is zero-shot accuracy of the model trained on the English data and test on the German data). Does the zero-shot performance using multilingual model improve over the English only BERT model?

> Accuracy: 0.3771084249019623. The accuracy does not seem to improve switching base models.

Type *Markdown* and LaTeX: α^2