

Chris McCormick [About](#) [Tutorials](#) [Archive](#)

BERT Fine-Tuning Tutorial with PyTorch

22 Jul 2019

By Chris McCormick and Nick Ryan

Revised on 3/20/20 - Switched to `tokenizer.encode_plus` and added validation loss. See Revision History at the end for details.

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](#) and as a Colab Notebook [here](#).

The content is identical in both, but:

- The blog post includes a comments section for discussion.

■ A Shift in NLP

- 1. Setup
 - 1.1. Using Colab GPU for Training
 - 1.2. Installing the Hugging Face Library
- 2. Loading CoLA Dataset
 - 2.1. Download & Extract
 - 2.2. Parse
- 3. Tokenization & Input Formatting
 - 3.1. BERT Tokenizer
 - 3.2. Required Formatting
 - Special Tokens
 - Sentence Length & Attention Mask
 - 3.3. Tokenize Dataset
 - 3.4. Training & Validation Split
- 4. Train Our Classification Model
 - 4.1. BertForSequenceClassification
 - 4.2. Optimizer & Learning Rate Scheduler
 - 4.3. Training Loop

- 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!



Contents

- [Contents](#)
- [Introduction](#)
 - [History](#)
 - [What is BERT?](#)
 - [Advantages of Fine-Tuning](#)

- 5. Performance On Test Set
 - 5.1. Data Preparation
 - 5.2. Evaluate on Test Set
- Conclusion
- Appendix
 - A1. Saving & Loading Fine-Tuned Model
 - A.2. Weight Decay
- Revision History
 - Further Work
 - Cite

Introduction

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 practitioners, 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 practitioners 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 architectures 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 transferred 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!

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.

```
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.

```
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 P100-PCIE-16GB

1.2. Installing the Hugging Face Library

Next, let's install the `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 prediction, etc. Using these pre-built classes simplifies the process of modifying BERT for your purposes.

```
!pip install transformers
```

[I've removed this output cell for brevity].

The code in this notebook is actually a simplified version of the [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](#)). 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.

Unfortunately, all of this configurability comes at the cost of *readability*. In this Notebook, we've simplified the code greatly and added plenty of comments to make it clear what's going on.

2. Loading CoLA Dataset

We'll use [The Corpus of Linguistic Acceptability \(CoLA\)](#) 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.

```
!pip install wget
```

```
Collecting wget
  Downloading https://files.pythonhosted.org/packages/47/6a/62e288c
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... [?25l [?25hdone
  Created wheel for wget: filename=wget-3.2-cp36-none-any.whl size=
  Stored in directory: /root/.cache/pip/wheels/40/15/30/7d8f7cea296
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

The dataset is hosted on GitHub in this repo: <https://nyu-ml.github.io/CoLA/>

```
import wget
import os

print('Downloading dataset...')

# The URL for the dataset zip file.
url = 'https://nyu-ml.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.

```
# 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.

```
import pandas as pd

# Load the dataset into a pandas dataframe.
df = pd.read_csv("./cola_public/raw/in_domain_train.tsv", delimiter='\t')

# 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

	sentence_source	label	label_notes	sentence
8200	ad03	1	NaN	They kicked themselves
3862	ks08	1	NaN	A big green insect flew into the soup.
8298	ad03	1	NaN	I often have a cold.

	sentence_source	label	label_notes	sentence
6542	g_81	0	*	Which did you buy the table supported the book?
722	bc01	0	*	Home was gone by John.
3693	ks08	1	NaN	I think that person we met last week is insane.
6283	c_13	1	NaN	Kathleen really hates her job.
4118	ks08	1	NaN	Do not use these words in the beginning of a s...
2592	l-93	1	NaN	Jessica sprayed paint under the table.
8194	ad03	0	*	I sent she away.

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

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

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


	sentence	label
4867	They investigated.	0
200	The more he reads, the more books I wonder to ...	0
4593	Any zebras can't fly.	0
3226	Cities destroy easily.	0
7337	The time elapsed the day.	0

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

```
# 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.

```
from transformers import BertTokenizer

# Load the BERT tokenizer.
print('Loading BERT tokenizer...')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_1
```

Loading BERT tokenizer...

HBox(children=(IntProgress(value=0, description='Downloading', max=

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

```
# 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(sentence)))
```

```
Original: Our friends won't buy this analysis, let alone the next  
Tokenized: ['our', 'friends', 'won', "'", 't', 'buy', 'this', 'and', 'the', 'next']  
Token IDs: [2256, 2814, 2180, 1005, 1056, 4965, 2023, 4106, 1010, 1010, 1010]
```

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

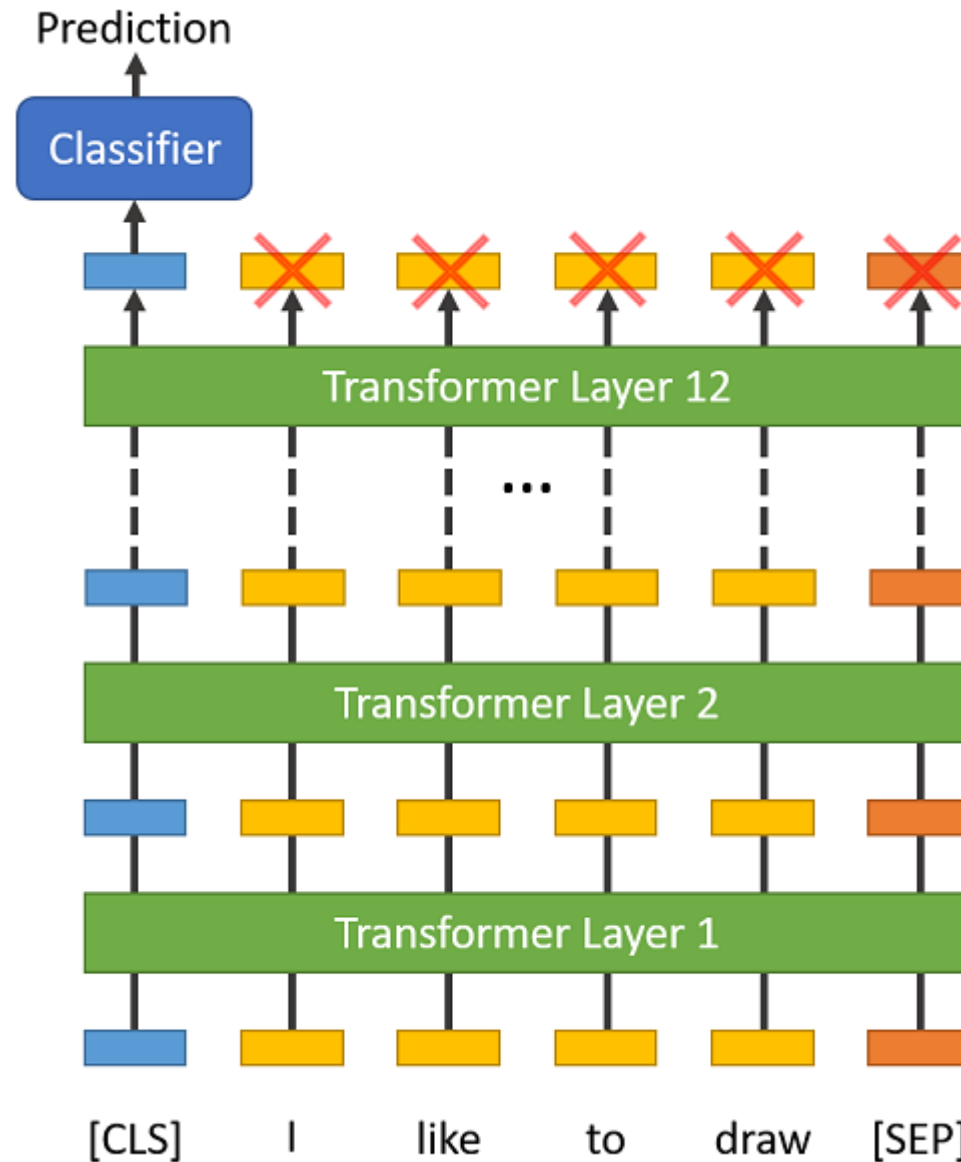
This token is an artifact of two-sentence tasks, where BERT is given two separate sentences and asked to determine something (e.g., can the answer to the question in sentence A be found in sentence B?).

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](#))

You might think to try some pooling strategy over the final embeddings, but this isn't necessary. Because BERT is trained to only use this [CLS] token for classification, we know that the model has been motivated to encode everything it needs for the classification step into that single 768-value embedding vector. It's already done the pooling for us!

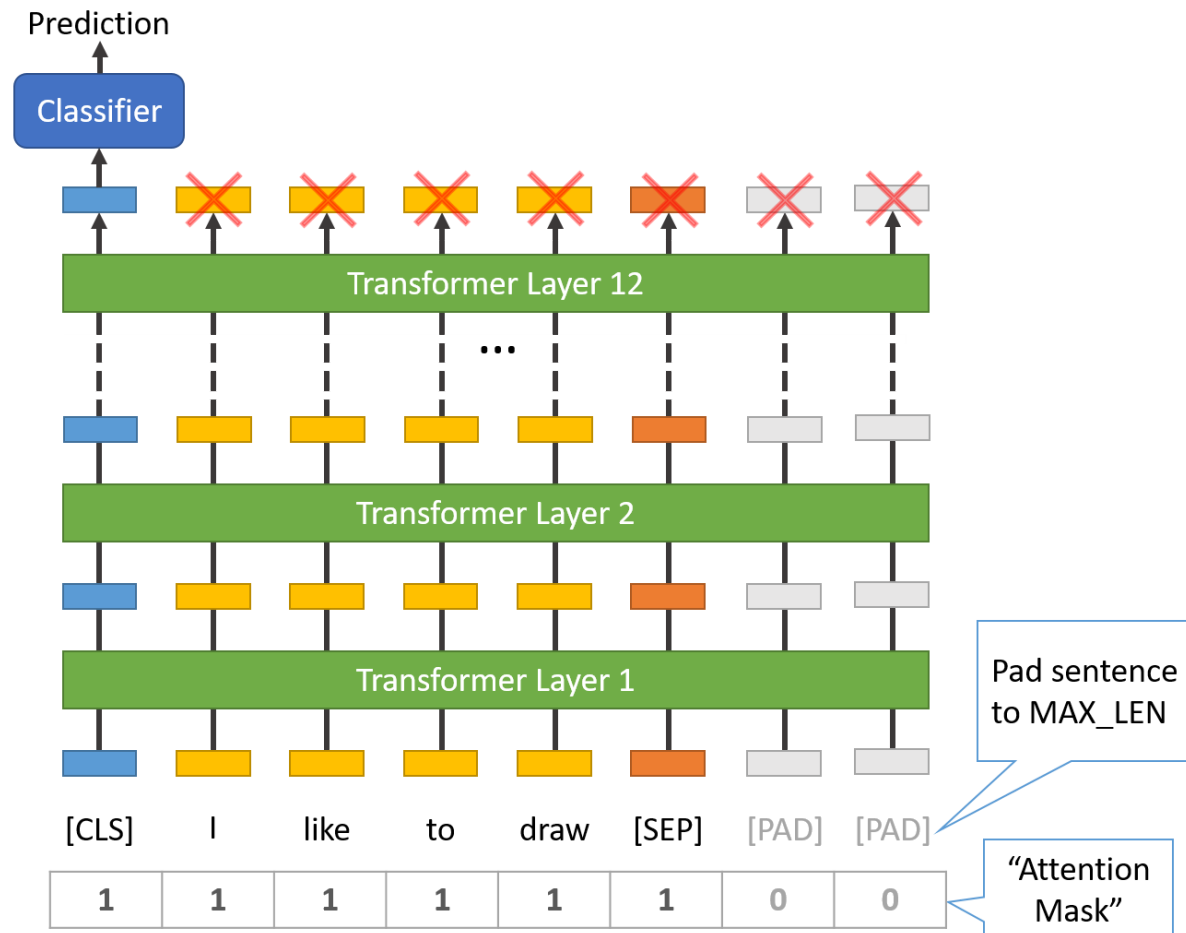
Sentence Length & Attention Mask

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

MAX_LEN = 64 --> Training epochs take ~2:57 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.

```
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](#).

```
# Tokenize all of the sentences and map the tokens to their 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 encode
```

```
        add_special_tokens = True,           # Add '[CLS]' and '[SEP]'
```

```
        max_length = 64,                     # Pad & truncate to max
```

```
        pad_to_max_length = True,
```

```
        return_attention_mask = True,        # Construct mask
```

```
        return_tensors = 'pt',              # Return pytorch tensors
```

```
    )
```

```
    # Add the encoded sentence to the list.
```

```
    input_ids.append(encoded_dict['input_ids'])
```

```
    # And its attention mask (simply differentiates padding from no padding)
```

```

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])

Original: Our friends won't buy this analysis, let alone the next
Token IDs: tensor([ 101, 2256, 2814, 2180, 1005, 1056, 4965,
                   2292, 2894, 1996, 2279, 2028, 2057, 16599, 1012,
                   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.

```
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_size])

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.

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

# The DataLoader needs to know our batch size for training, so we pass it in
# here. For fine-tuning BERT on a specific task, the authors recommend a
# 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 batch samples randomly
    batch_size = batch_size # Trains with this batch size.
)

# For validation the order doesn't matter, so we'll just read them out in order.
validation_dataloader = DataLoader(
    val_dataset, # The validation samples.
    sampler = SequentialSampler(val_dataset), # Pull out batches in order
    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

The documentation for these can be found under [here](#).

We'll be using [BertForSequenceClassification](#). 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](#), with the additional parameters defined [here](#).

```
from transformers import BertForSequenceClassification, AdamW, Bert

# 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 uncased
    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
    output_hidden_states = False, # Whether the model returns all hidden states
)

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

[I've removed this output cell for brevity].

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.

```
# 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(
    len(params)))

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](#)):

- **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](#)).

You can find the creation of the AdamW optimizer in `run_glue.py` [here](#).

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

```
from transformers import get_linear_schedule_with_warmup
```

```
# Number of training epochs. The BERT authors recommend between 2 and 10
# We chose to run for 4, but we'll see later that this may be overkill for
# training data.
epochs = 4
```

```
# Total number of training steps is [number of batches] x [number of epochs]
```

```
# (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, #
                                             num_training_steps = t
```

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 training phase and a validation phase.

Thank you to [Stas Bekman](#) 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

Evaluation:

- 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](#) which you may also find helpful.

Define a helper function for calculating accuracy.

```
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`

```
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!

```
import random
import numpy as np
```

```

# This training code is based on the `run_glue.py` script here:
# https://github.com/huggingface/transformers/blob/5bfcd0485ece086c

# 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, epochs))
    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
# `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/51433372/why-does-training-loss-increase-when-the-model-is-in-training-mode)
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 device
    # `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 the
# backward pass. PyTorch doesn't do this automatically because
# accumulating the gradients is "convenient while training"
# (source: https://stackoverflow.com/questions/48001598/why-not-call-model.zero_grad()
model.zero_grad())

# Perform a forward pass (evaluate the model on this training batch)
# The documentation for this `model` function is here:
# https://huggingface.co/transformers/v2.2.0/model_doc/bert.html
# It returns different numbers of parameters depending on what is
# arguments given and what flags are set. For our usage here, it returns
# the loss (because we provided labels) and the "logits" -- the
# outputs prior to activation.
loss, logits = model(b_input_ids,
                     token_type_ids=None,
                     attention_mask=b_input_mask,
                     labels=b_labels)

# Accumulate the training loss over all of the batches so that we can
# calculate the average loss at the end. `loss` is a Tensor of shape
# (1,) and the `.item()` function just returns the Python scalar
# 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 gradients
# 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 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 epoch took: {:}".format(training_time))

# =====
#                               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
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 GPU
    # 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
    # the forward pass, since this is only needed for backprop
    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.

        # The documentation for this `model` function is here:
```

```
# https://huggingface.co/transformers/v2.2.0/model_doc,
# Get the "logits" output by the model. The "logits" are
# values prior to applying an activation function like
(loss, logits) = model(b_input_ids,
                       token_type_ids=None,
                       attention_mask=b_input_mask,
                       labels=b_labels)

# 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,
# 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_data_loader)
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_data_loader)

# 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.1
```

=====
Epoch 1 / 4

Training...

Batch	40	of	241.	Elapsed: 0:00:08.
Batch	80	of	241.	Elapsed: 0:00:17.
Batch	120	of	241.	Elapsed: 0:00:25.
Batch	160	of	241.	Elapsed: 0:00:34.
Batch	200	of	241.	Elapsed: 0:00:42.
Batch	240	of	241.	Elapsed: 0:00:51.

Average training loss: 0.50

Training epoch took: 0:00:51

Running Validation...

Accuracy: 0.80

Validation Loss: 0.45

Validation took: 0:00:02

=====
Epoch 2 / 4
=====

Training...

Batch 40 of 241. Elapsed: 0:00:08.

Batch 80 of 241. Elapsed: 0:00:17.

Batch 120 of 241. Elapsed: 0:00:25.

Batch 160 of 241. Elapsed: 0:00:34.

Batch 200 of 241. Elapsed: 0:00:42.

Batch 240 of 241. Elapsed: 0:00:51.

Average training loss: 0.32

Training epoch took: 0:00:51

Running Validation...

Accuracy: 0.81

Validation Loss: 0.46

Validation took: 0:00:02

=====
Epoch 3 / 4
=====

Training...

Batch 40 of 241. Elapsed: 0:00:08.

Batch 80 of 241. Elapsed: 0:00:17.

Batch 120 of 241. Elapsed: 0:00:25.

Batch 160 of 241. Elapsed: 0:00:34.

Batch 200 of 241. Elapsed: 0:00:42.

Batch 240 of 241. Elapsed: 0:00:51.

Average training loss: 0.22
Training epoch took: 0:00:51

Running Validation...

Accuracy: 0.82
Validation Loss: 0.49
Validation took: 0:00:02

=====
Epoch 4 / 4
=====

Training...

Batch	40	of	241.	Elapsed: 0:00:08.
Batch	80	of	241.	Elapsed: 0:00:17.
Batch	120	of	241.	Elapsed: 0:00:25.
Batch	160	of	241.	Elapsed: 0:00:34.
Batch	200	of	241.	Elapsed: 0:00:42.
Batch	240	of	241.	Elapsed: 0:00:51.

Average training loss: 0.16
Training epoch took: 0:00:51

Running Validation...

Accuracy: 0.82
Validation Loss: 0.55
Validation took: 0:00:02

Training complete!

Total training took 0:03:30 (h:mm:ss)

Let's view the summary of the training process.

```
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-w:

# Display the table.
df_stats
```

	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
epoch					
1	0.50	0.45	0.80	0:00:51	0:00:02
2	0.32	0.46	0.81	0:00:51	0:00:02
3	0.22	0.49	0.82	0:00:51	0:00:02
4	0.16	0.55	0.82	0:00:51	0:00:02

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 catch this, while accuracy will not.

```
import matplotlib.pyplot as plt
% matplotlib inline

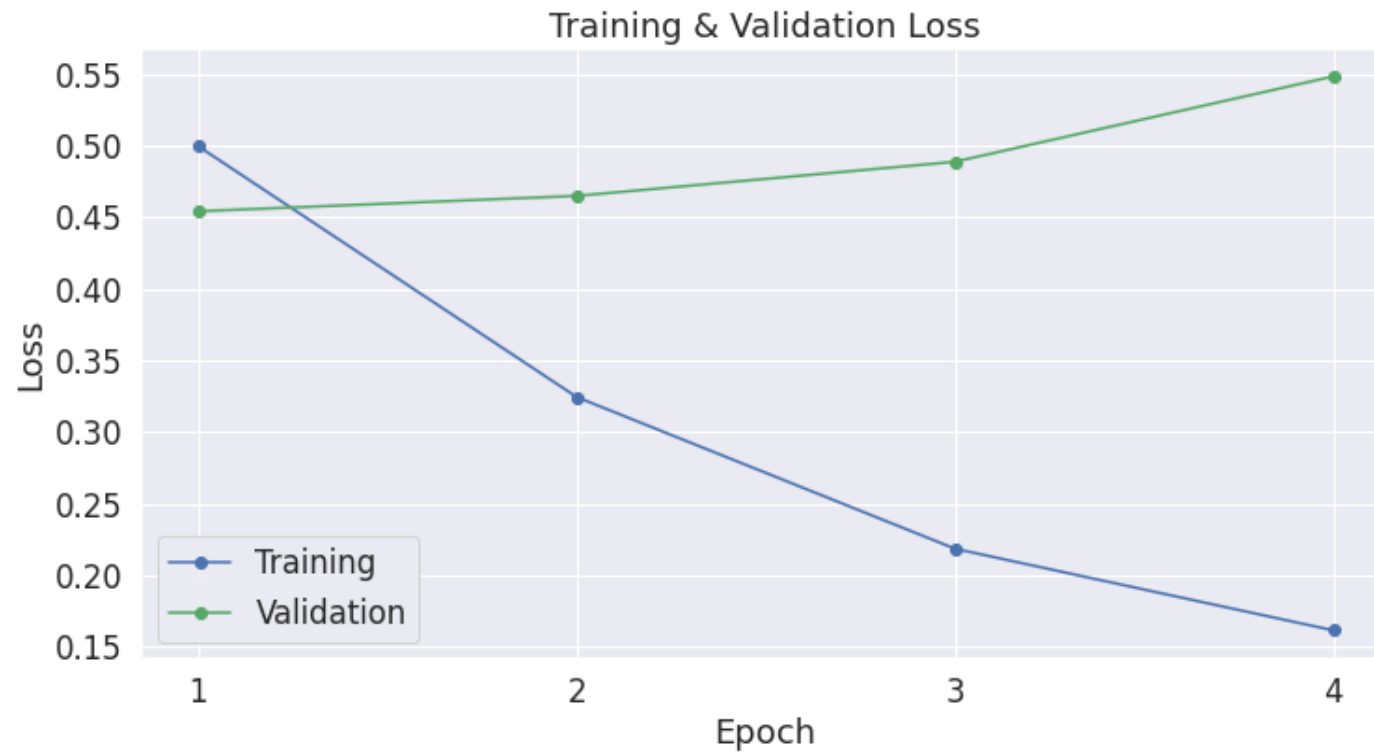
import seaborn as sns

# 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

Now we'll load the holdout dataset and prepare inputs just as we did with the training set. Then we'll evaluate predictions using [Matthew's correlation coefficient](#) because this is the metric used by the wider NLP community to evaluate performance on CoLA. With this metric, +1 is the best score, and -1 is the worst score. This way, we can see how well we perform against the state of the art models for this specific task.

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.

```
import pandas as pd

# Load the dataset into a pandas dataframe.
df = pd.read_csv("./cola_public/raw/out_of_domain_dev.tsv", delimiter='\t')

# 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 ID
```

```

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 encode
        add_special_tokens = True,           # Add '[CLS]' and '[SEP]'
        max_length = 64,                    # Pad & truncate to length
        pad_to_max_length = True,           # Pad to the maximum length
        return_attention_mask = True,       # Construct mask
        return_tensors = 'pt',              # Return pytorch tensors
    )

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

    # And its attention mask (simply differentiates padding from no padding)
    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=prediction_sampler, batch_size=batch_size)
```

Number of test sentences: 516

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.

```
# Prediction on test set

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

# 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 memory
    # 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)

print('    DONE.')
```

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

Accuracy on the CoLA benchmark is measured using the “[Matthews correlation coefficient](#)” (MCC).

We use MCC here because the classes are imbalanced:

```
print('Positive samples: %d of %d (%.2f%%)' % (df.label.sum(), len(df)))
```

```
Positive samples: 354 of 516 (68.60%)
```

```
from sklearn.metrics import matthews_corrcoef
```

```
matthews_set = []
```

```
# Evaluate each test batch using Matthew's correlation coefficient
```

```
print('Calculating Matthews Corr. Coef. for each batch...')
```

```
# For each input batch...
for i in range(len(true_labels)):

    # The predictions for this batch are a 2-column ndarray (one col
    # and one column for "1"). Pick the label with the highest value
    # in to a list of 0s and 1s.
    pred_labels_i = np.argmax(predictions[i], axis=1).flatten()

    # Calculate and store the coef for this batch.
    matthews = matthews_corrcoef(true_labels[i], pred_labels_i)
    matthews_set.append(matthews)
```

Calculating Matthews Corr. Coef. for each batch...

```
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:136: RuntimeWarning:
mcc = cov_ytyp / np.sqrt(cov_ytyp * cov_ypyp)
```

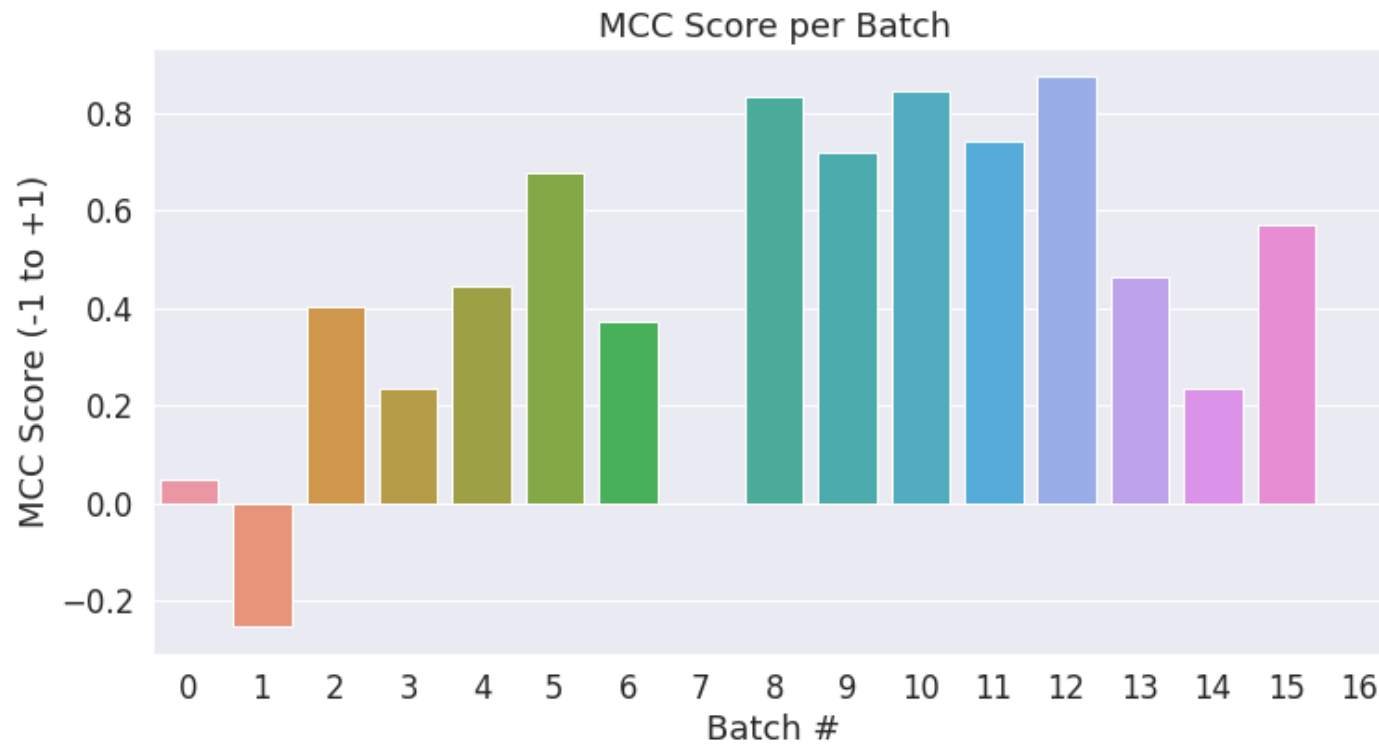
The final score will be based on the entire test set, but let's take a look at the scores on the individual batches to get a sense of the variability in the metric between batches.

Each batch has 32 sentences in it, except the last batch which has only (516 % 32) = 4 test sentences in it.


```
# Create a barplot showing the MCC score for each batch of test samples
ax = sns.barplot(x=list(range(len(matthews_set))), y=matthews_set,

plt.title('MCC Score per Batch')
plt.ylabel('MCC Score (-1 to +1)')
plt.xlabel('Batch #')

plt.show()
```



Now we'll combine the results for all of the batches and calculate our final MCC score.

```
# Combine the results across all batches.
flat_predictions = np.concatenate(predictions, axis=0)

# For each sample, pick the label (0 or 1) with the higher score.
flat_predictions = np.argmax(flat_predictions, axis=1).flatten()

# Combine the correct labels for each batch into a single list.
flat_true_labels = np.concatenate(true_labels, axis=0)

# Calculate the MCC
mcc = matthews_corrcoef(flat_true_labels, flat_predictions)

print('Total MCC: %.3f' % mcc)
```

Total MCC: 0.498

Cool! In about half an hour and without doing any hyperparameter tuning (adjusting the learning rate, epochs, batch size, ADAM properties, etc.) we are able to get a good score.

Note: To maximize the score, we should remove the “validation set” (which we used to help determine how many epochs to train for) and train on the entire training set.

The library documents the expected accuracy for this benchmark [here](#) as **49.23** .

You can also look at the official leaderboard [here](#).

Note that (due to the small dataset size?) the accuracy can vary significantly between runs.

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.

Appendix

A1. Saving & Loading Fine-Tuned Model

This first cell (taken from `run_glue.py` [here](#)) writes the model and tokenizer out to disk.

```
import os

# Saving best-practices: if you use defaults names for the model, y

output_dir = './model_save/'

# Create output directory if needed
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

print("Saving model to %s" % output_dir)

# Save a trained model, configuration and tokenizer using `save_pretrained`
# They can then be reloaded using `from_pretrained()`
model_to_save = model.module if hasattr(model, 'module') else model
model_to_save.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)

# Good practice: save your training arguments together with the tra
# torch.save(args, os.path.join(output_dir, 'training_args.bin'))
```

Saving model to ./model_save/

```
( './model_save/vocab.txt',  
  './model_save/special_tokens_map.json',  
  './model_save/added_tokens.json')
```

Let's check out the file sizes, out of curiosity.

```
!ls -l --block-size=K ./model_save/
```

```
total 427960K  
-rw-r--r-- 1 root root      2K Mar 18 15:53 config.json  
-rw-r--r-- 1 root root 427719K Mar 18 15:53 pytorch_model.bin  
-rw-r--r-- 1 root root      1K Mar 18 15:53 special_tokens_map.json  
-rw-r--r-- 1 root root      1K Mar 18 15:53 tokenizer_config.json  
-rw-r--r-- 1 root root    227K Mar 18 15:53 vocab.txt
```

The largest file is the model weights, at around 418 megabytes.

```
!ls -l --block-size=M ./model_save/pytorch_model.bin
```

```
-rw-r--r-- 1 root root 418M Mar 18 15:53 ./model_save/pytorch_model
```

To save your model across Colab Notebook sessions, download it to your local machine, or ideally copy it to your Google Drive.

```
# Mount Google Drive to this Notebook instance.  
from google.colab import drive  
drive.mount('/content/drive')
```

```
# Copy the model files to a directory in your Google Drive.  
!cp -r ./model_save/ "./drive/Shared drives/ChrisMcCormick.AI/Blog
```

The following functions will load the model back from disk.

```
# Load a trained model and vocabulary that you have fine-tuned  
model = model_class.from_pretrained(output_dir)  
tokenizer = tokenizer_class.from_pretrained(output_dir)
```

```
# Copy the model to the GPU.
model.to(device)
```

A.2. Weight Decay

The huggingface example includes the following code block for enabling weight decay, but the default decay rate is “0.0”, so I moved this to the appendix.

This block essentially tells the optimizer to not apply weight decay to the bias terms (e.g., b in the equation $y = Wx + b$). Weight decay is a form of regularization—after calculating the gradients, we multiply them by, e.g., 0.99.

```
# This code is taken from:
# https://github.com/huggingface/transformers/blob/5bfcd0485ece086c
# Don't apply weight decay to any parameters whose names include th
# (Here, the BERT doesn't have `gamma` or `beta` parameters, only `
no_decay = ['bias', 'LayerNorm.weight']

# Separate the `weight` parameters from the `bias` parameters.
# – For the `weight` parameters, this specifies a 'weight_decay_rat
# – For the `bias` parameters, the 'weight_decay_rate' is 0.0.
optimizer_grouped_parameters = [
    # Filter for all parameters which *don't* include 'bias', 'gamma
    {'params': [p for n, p in param_optimizer if not any(nd in n for nd in no_decay)],
     'weight_decay_rate': 0.1},
```

```
# Filter for parameters which *do* include those.
{'params': [p for n, p in param_optimizer if any(nd in n for nd in
    'weight_decay_rate': 0.0}]

# Note - `optimizer_grouped_parameters` only includes the parameter
# the names.
```

Revision History

Version 3 - *Mar 18th, 2020* - (current)

- Simplified the tokenization and input formatting (for both training and test) by leveraging the `tokenizer.encode_plus` function. `encode_plus` handles padding *and* creates the attention masks for us.
- Improved explanation of attention masks.
- Switched to using `torch.utils.data.random_split` for creating the training-validation split.
- Added a summary table of the training statistics (validation loss, time per epoch, etc.).
- Added validation loss to the learning curve plot, so we can see if we're overfitting.
 - Thank you to [Stas Bekman](#) for contributing this!

- Displayed the per-batch MCC as a bar plot.

Version 2 - Dec 20th, 2019 - [link](#)

- huggingface renamed their library to `transformers` .
- Updated the notebook to use the `transformers` library.

Version 1 - July 22nd, 2019

- Initial version.

Further Work

- It might make more sense to use the MCC score for “validation accuracy”, but I’ve left it out so as not to have to explain it earlier in the Notebook.
- Seeding – I’m not convinced that setting the seed values at the beginning of the training loop is actually creating reproducible results...
- The MCC score seems to vary substantially across different runs. It would be interesting to run this example a number of times and show the variance.

Cite

Chris McCormick and Nick Ryan. (2019, July 22). *BERT Fine-Tuning Tutorial with PyTorch*. Retrieved from <http://www.mccormickml.com>



Ad tink.com

More ▼

Related posts

[Existing Tools for Named Entity Recognition](#) 19 May 2020

[Trivial BERSuit - How much trivia does BERT know?](#) 14 Apr 2020

[Question Answering with a Fine-Tuned BERT](#) 10 Mar 2020

© 2020. All rights reserved.