Chris McCormick About Tutorials Archive

BERT Fine-Tuning Tutorial with PyTorch

22 Jul 2019

By Chris McCormick and Nick Ryan

Revised on 12/13/19 to use the new transformers interface.

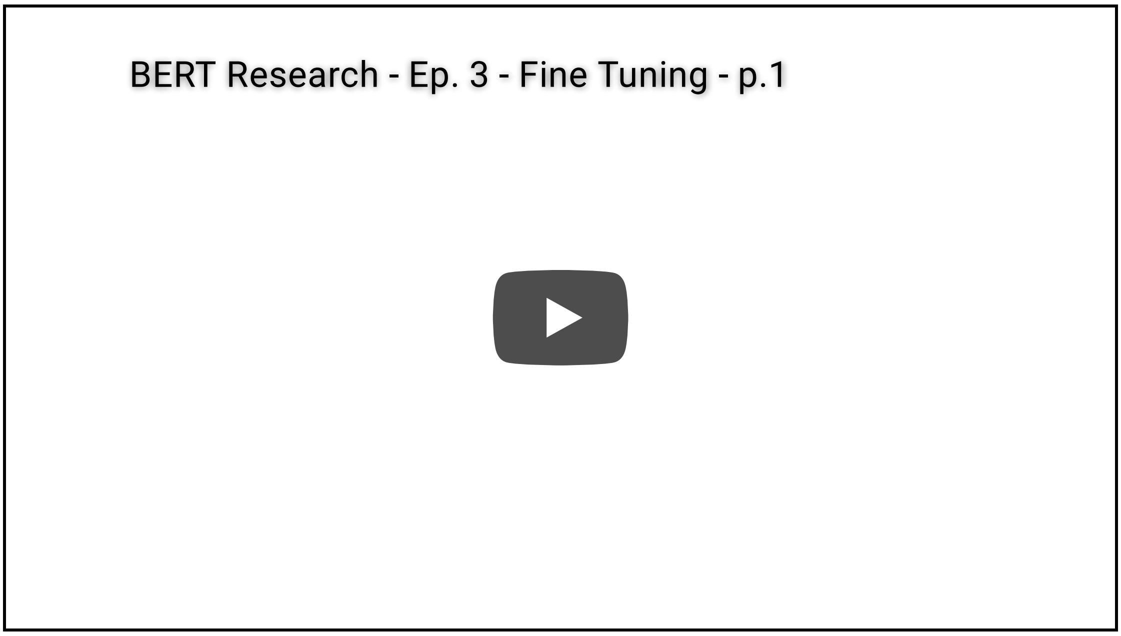
In this tutorial I'll show you how to use BERT with the huggingface PyTorch library to quickly and efficiently fine-tune a model to get near state of the art performance in sentence classification. More broadly, I describe the practical application of transfer learning in NLP to create high performance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

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 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!

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')
```

The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via the `%tensorflow_version 1.x` magic: more info.

```
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.

    print('We will use the GPU:', torch.cuda.get_device_

# 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 prediciton, 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
Building wheels for collected packages: wget
```

```
  Building wheel for wget (setup.py) ... [?25l[?25hdone
  Created wheel for wget: filename=wget-3.2-cp36-none-ar
  Stored in directory: /root/.cache/pip/wheels/40/15/30/
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/

```
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.zi

# 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'

# Report the number of sentences.
print('Number of training sentences: {:,}\n'.format(df.s

# 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.
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

	sentence_source	label	label_notes	sentence
				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 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!

```
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-unc
```

```
Loading BERT tokenizer...
```

```
HBox(children=(IntProgress(value=0, description='Downloa
```

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(tok
```

```
 Original:  Our friends won't buy this analysis, let alo
Tokenized:  ['our', 'friends', 'won', "'", 't', 'buy', '
Token IDs:  [2256, 2814, 2180, 1005, 1056, 4965, 2023, 4
```

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?).

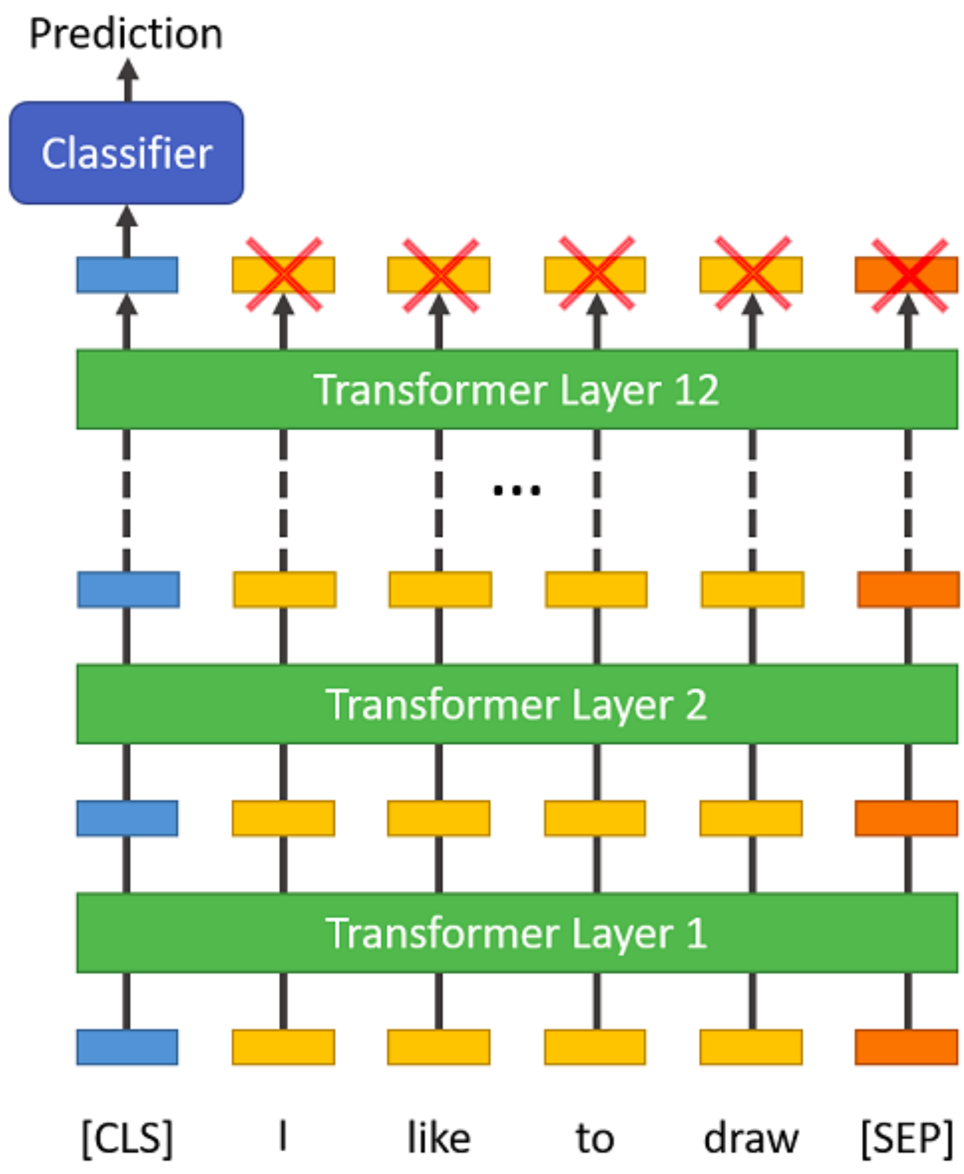
I am not certain yet why the token is still required when we have only single-sentence input, but it is!

`[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)

I'm not sure why the authors took this strategy instead of some kind of pooling of all the final vectors, but I'm sure that if pooling were better they would have gone that route.

Also, 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.

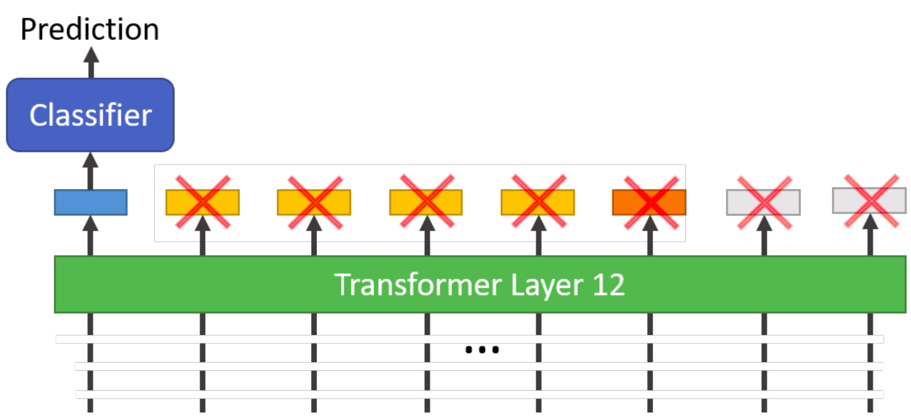
Sentence Length & Attention Mask

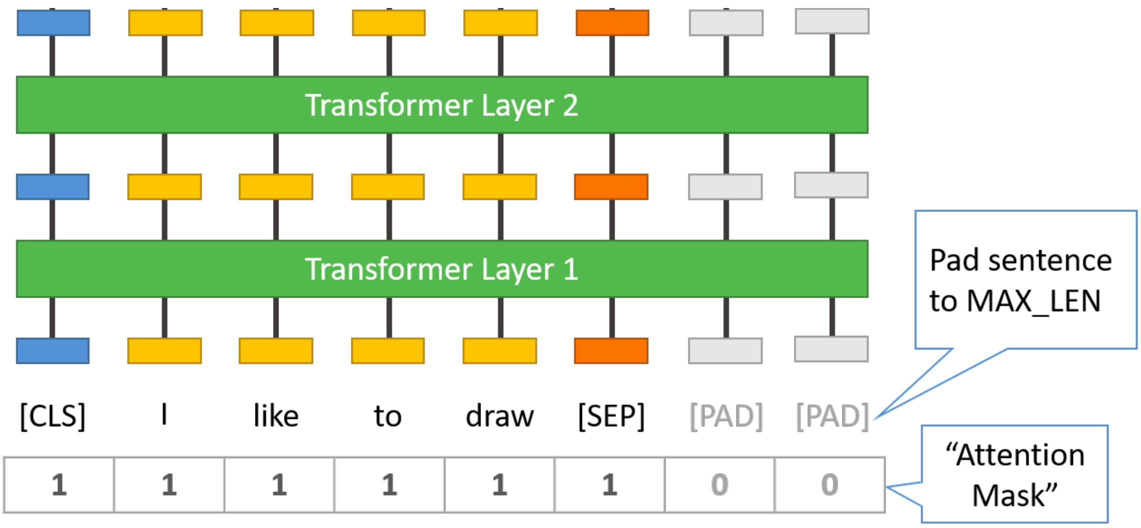
The sentences in our dataset obviously have varying lengths, so how does BERT handle this?

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?! Again, I don't currently know why).

I've experimented with running this notebook with two different values of MAX_LEN, and it impacted both the training speed and the test set accuracy.

With a Tesla K80 and:

```
MAX_LEN = 128  -->  Training epochs take ~5:28 each, scc
MAX_LEN = 64   -->  Training epochs take ~2:57 each, scc
```

These results suggest to me that the padding tokens aren't simply skipped over–that they are in fact fed through the model and incorporated in the results (thereby impacting both model speed and accuracy). I'll have to dig into the architecture more to understand this.

3.2. Sentences to IDs

The `tokenizer.encode` 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.

Oddly, this function can perform truncating for us, but doesn't handle padding.

```
# Tokenize all of the sentences and map the tokens to th
input_ids = []

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

                        # This function also supports tr
                        # to pytorch tensors, but we nee
                        # can't use these features :( .
                        #max_length = 128,          # Tr
                        #return_tensors = 'pt',     # Re
                   )

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

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

```
Original:  Our friends won't buy this analysis, let alor
Token IDs: [101, 2256, 2814, 2180, 1005, 1056, 4965, 202
```

3.3. Padding & Truncating

Pad and truncate our sequences so that they all have the same length, `MAX_LEN` .

First, what's the maximum sentence length in our dataset?

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

```
Max sentence length:  47
```

Given that, let's choose MAX_LEN = 64 and apply the padding.

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

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

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

print('\nPadding token: "{:}", ID: {:}'.format(tokenizer
```

```
# Pad our input tokens with value 0.
# "post" indicates that we want to pad and truncate at t
# as opposed to the beginning.
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dty
                          value=0, truncating="post", pa

print('\nDone.')
```

```
Padding/truncating all sentences to 64 values...

Padding token: "[PAD]", ID: 0

Done.
```

```
Using TensorFlow backend.
```

3.4. Attention Masks

The attention mask simply makes it explicit which tokens are actual words versus which are padding.

The BERT vocabulary does not use the ID 0, so if a token ID is 0, then it’s padding, and otherwise it’s a real token.

```
# Create attention masks
attention_masks = []

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

    # Create the attention mask.
    #   - If a token ID is 0, then it's padding, set the
```

```
    #   - If a token ID is > 0, then it's a real token,
    att_mask = [int(token_id > 0) for token_id in sent]

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

3.5. Training & Validation Split

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

```
# Use train_test_split to split our data into train and
# training
from sklearn.model_selection import train_test_split

# Use 90% for training and 10% for validation.
train_inputs, validation_inputs, train_labels, validatic

# Do the same for the masks.
train_masks, validation_masks, _, _ = train_test_split(a
                                              random_stat
```

3.6. Converting to PyTorch Data Types

Our model expects PyTorch tensors rather than numpy.ndarrays, so convert all of our dataset variables.

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

train_labels = torch.tensor(train_labels)
```

```
validation_labels = torch.tensor(validation_labels)

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

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 TensorDataset, DataLoader,

# The DataLoader needs to know our batch size for traini
# here.
# For fine-tuning BERT on a specific task, the authors r
# 16 or 32.

batch_size = 32

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

# Create the DataLoader for our validation set.
validation_data = TensorDataset(validation_inputs, valic
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, samp
```

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,

# Load BertForSequenceClassification, the pretrained BEF
# linear classification layer on top.
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", # Use the 12-layer BERT model,
    num_labels = 2, # The number of output labels--2 for
                    # You can increase this for multi-cl
    output_attentions = False, # Whether the model retur
    output_hidden_states = False, # Whether the model re
)

# 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

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

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

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

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

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

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

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

==== Embedding Layer ====

bert.embeddings.word_embeddings.weight
bert.embeddings.position_embeddings.weight
bert.embeddings.token_type_embeddings.weight
bert.embeddings.LayerNorm.weight
bert.embeddings.LayerNorm.bias

==== First Transformer ====

bert.encoder.layer.0.attention.self.query.weight
bert.encoder.layer.0.attention.self.query.bias
```

```
bert.encoder.layer.0.attention.self.key.weight
bert.encoder.layer.0.attention.self.key.bias
bert.encoder.layer.0.attention.self.value.weight
bert.encoder.layer.0.attention.self.value.bias
bert.encoder.layer.0.attention.output.dense.weight
bert.encoder.layer.0.attention.output.dense.bias
bert.encoder.layer.0.attention.output.LayerNorm.weight
bert.encoder.layer.0.attention.output.LayerNorm.bias
bert.encoder.layer.0.intermediate.dense.weight
bert.encoder.layer.0.intermediate.dense.bias
bert.encoder.layer.0.output.dense.weight
bert.encoder.layer.0.output.dense.bias
bert.encoder.layer.0.output.LayerNorm.weight
bert.encoder.layer.0.output.LayerNorm.bias

==== Output Layer ====

bert.pooler.dense.weight
bert.pooler.dense.bias
classifier.weight
classifier.bias
```

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:

- Batch size: 16, 32 (We chose 32 when creating our DataLoaders).
- Learning rate (Adam): 5e-5, 3e-5, 2e-5 (We'll use 2e-5).
- Number of epochs: 2, 3, 4 (We'll use 4).

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 (a
# I believe the 'W' stands for 'Weight Decay fix"
optimizer = AdamW(model.parameters(),
                  lr = 2e-5, # args.learning_rate - defa
                  eps = 1e-8 # args.adam_epsilon  - defa
                )
```

```
from transformers import get_linear_schedule_with_warmup

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

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

# Create the learning rate scheduler.
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_s
                                            num_training
```

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. At each pass we need to:

Training loop:

- 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 loop:

- 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

So please read carefully through the comments to get an understanding of what's happening. If you're unfamiliar with pytorch a quick look at some of their beginner tutorials will help show you that training loops really involve only a few simple steps; the rest is usually just decoration and logging.

Define a helper function for calculating accuracy.

```
import numpy as np

# Function to calculate the accuracy of our predictions
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
```

Helper function for formatting elapsed times.

```
import time
import datetime

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

    # Format as hh:mm:ss
    return str(datetime.timedelta(seconds=elapsed_rounde
```

We're ready to kick off the training!

```
import random

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


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

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

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

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

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

    # Perform one full pass over the training set.

    print("")
    print('======== Epoch {:} / {:} ========'.format(epo
    print('Training...')

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

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

    # Put the model into training mode. Don't be mislead
    # `train` just changes the *mode*, it doesn't *perfo
    # `dropout` and `batchnorm` layers behave differentl
    # vs. test (source: https://stackoverflow.com/questi
    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,}.    Elapse

        # Unpack this training batch from our dataloader
        #
```

```
        # As we unpack the batch, we'll also copy each t
        # `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 gradier
        # backward pass. PyTorch doesn't do this automat
        # accumulating the gradients is "convenient whil
        # (source: https://stackoverflow.com/questions/4
        model.zero_grad()

        # Perform a forward pass (evaluate the model on
        # This will return the loss (rather than the moc
        # have provided the `labels`.
        # The documentation for this `model` function is
        # https://huggingface.co/transformers/v2.2.0/moc
        outputs = model(b_input_ids,
                    token_type_ids=None,
                    attention_mask=b_input_mask,
                    labels=b_labels)

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

        # Accumulate the training loss over all of the t
        # calculate the average loss at the end. `loss`
        # single value; the `.item()` function just retu
        # from the tensor.
        total_loss += loss.item()

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

        # Clip the norm of the gradients to 1.0.
```

```
        # This is to help prevent the "exploding gradier
        torch.nn.utils.clip_grad_norm_(model.parameters(

        # Update parameters and take a step using the cc
        # The optimizer dictates the "update rule"--how
        # modified based on their gradients, the learnir
        optimizer.step()

        # Update the learning rate.
        scheduler.step()

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

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

    print("")
    print("  Average training loss: {0:.2f}".format(avg_
    print("  Training epcoh took: {:}".format(format_tin

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

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

    t0 = time.time()

    # Put the model in evaluation mode--the dropout laye
    # during evaluation.
    model.eval()

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

    # Evaluate data for one epoch
```

```
    for batch in validation_dataloader:

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

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

        # Telling the model not to compute or store grad
        # speeding up validation
        with torch.no_grad():

            # Forward pass, calculate logit predictions.
            # This will return the logits rather than th
            # not provided labels.
            # token_type_ids is the same as the "segment
            # differentiates sentence 1 and 2 in 2-sente
            # The documentation for this `model` functio
            # https://huggingface.co/transformers/v2.2.0
            outputs = model(b_input_ids,
                            token_type_ids=None,
                            attention_mask=b_input_mask)

        # Get the "logits" output by the model. The "log
        # values prior to applying an activation functio
        logits = outputs[0]

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

        # Calculate the accuracy for this batch of test
        tmp_eval_accuracy = flat_accuracy(logits, label_

        # Accumulate the total accuracy.
        eval_accuracy += tmp_eval_accuracy

        # Track the number of batches
        nb_eval_steps += 1

    # Report the final accuracy for this validation run.
```

```
        print("  Accuracy: {0:.2f}".format(eval_accuracy/nb_
        print("  Validation took: {:}".format(format_time(ti

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

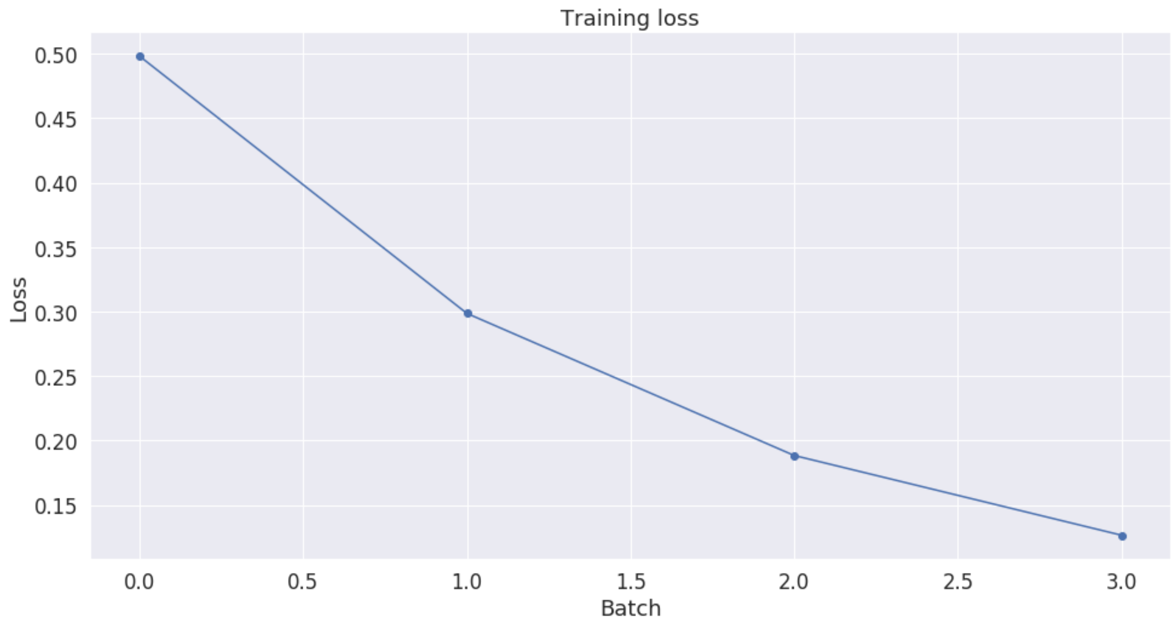
```
======== Epoch 1 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:11.
  Batch    80  of    241.    Elapsed: 0:00:21.
  Batch   120  of    241.    Elapsed: 0:00:31.
  Batch   160  of    241.    Elapsed: 0:00:42.
  Batch   200  of    241.    Elapsed: 0:00:52.
  Batch   240  of    241.    Elapsed: 0:01:03.

  Average training loss: 0.50
  Training epcoh took: 0:01:03

Running Validation...
  Accuracy: 0.79
  Validation took: 0:00:02

======== Epoch 2 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:11.
  Batch    80  of    241.    Elapsed: 0:00:21.
  Batch   120  of    241.    Elapsed: 0:00:32.
  Batch   160  of    241.    Elapsed: 0:00:42.
  Batch   200  of    241.    Elapsed: 0:00:52.
  Batch   240  of    241.    Elapsed: 0:01:03.

  Average training loss: 0.32
  Training epcoh took: 0:01:03

Running Validation...
  Accuracy: 0.82
  Validation took: 0:00:02
```

```
======== Epoch 3 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:11.
  Batch    80  of    241.    Elapsed: 0:00:21.
  Batch   120  of    241.    Elapsed: 0:00:32.
  Batch   160  of    241.    Elapsed: 0:00:42.
  Batch   200  of    241.    Elapsed: 0:00:52.
  Batch   240  of    241.    Elapsed: 0:01:03.

  Average training loss: 0.20
  Training epcoh took: 0:01:03

Running Validation...
  Accuracy: 0.82
  Validation took: 0:00:02

======== Epoch 4 / 4 ========
Training...
  Batch    40  of    241.    Elapsed: 0:00:10.
  Batch    80  of    241.    Elapsed: 0:00:21.
  Batch   120  of    241.    Elapsed: 0:00:31.
  Batch   160  of    241.    Elapsed: 0:00:42.
  Batch   200  of    241.    Elapsed: 0:00:52.
  Batch   240  of    241.    Elapsed: 0:01:03.

  Average training loss: 0.14
  Training epcoh took: 0:01:03

Running Validation...
  Accuracy: 0.82
  Validation took: 0:00:02

Training complete!
```

Let’s take a look at our training loss over all batches:

```
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(loss_values, 'b-o')

# Label the plot.
plt.title("Training loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")

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.ts

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

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

# Tokenize all of the sentences and map the tokens to th
input_ids = []

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

```
                    )

    input_ids.append(encoded_sent)

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

# Create attention masks
attention_masks = []

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

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

# Set the batch size.
batch_size = 32

# Create the DataLoader.
prediction_data = TensorDataset(prediction_inputs, predi
prediction_sampler = SequentialSampler(prediction_data)
prediction_dataloader = DataLoader(prediction_data, samp
```

```
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...'.for

# 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,
  # 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.
```

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

```
from sklearn.metrics import matthews_corrcoef

matthews_set = []

# Evaluate each test batch using Matthew's correlation c
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 ndarra
  # and one column for "1"). Pick the label with the hig
  # in to a list of 0s and 1s.
  pred_labels_i = np.argmax(predictions[i], axis=1).flat

  # Calculate and store the coef for this batch.
```

```
  matthews = matthews_corrcoef(true_labels[i], pred_labe
  matthews_set.append(matthews)
```

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

```
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/c
  mcc = cov_ytyp / np.sqrt(cov_ytyt * 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.

```
matthews_set
```

```
[0.049286405809014416,
 -0.21684543705982773,
 0.4040950971038548,
 0.41179801403140964,
 0.25365601296401685,
 0.6777932975034471,
 0.4879500364742666,
 0.0,
 0.8320502943378436,
 0.8246211251235321,
 0.9229582069908973,
```

```
 0.647150228929434,
 0.8150678894028793,
 0.7141684885491869,
 0.3268228676411533,
 0.5844155844155844,
 0.0]
```

```
# Combine the predictions for each batch into a single l
flat_predictions = [item for sublist in predictions for
flat_predictions = np.argmax(flat_predictions, axis=1).f

# Combine the correct labels for each batch into a singl
flat_true_labels = [item for sublist in true_labels for

# Calculate the MCC
mcc = matthews_corrcoef(flat_true_labels, flat_predictio

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

```
MCC: 0.529
```

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. I should also mention we didn't train on the entire training dataset, but set aside a portion of it as our validation set for legibililty of code.

The library documents the expected accuracy for this benchmark here.

You can also look at the official leaderboard here.

Note that (due to the small dataset size?) the accuracy can vary significantly with different random seeds.

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 t

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 usir
# They can then be reloaded using `from_pretrained()`
model_to_save = model.module if hasattr(model, 'module')
model_to_save.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)
```

```
# Good practice: save your training arguments together w
# torch.save(args, os.path.join(output_dir, 'training_ar
```

```
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 427964K
-rw-r--r-- 1 root root      1K Dec 19 17:33 added_tokens
-rw-r--r-- 1 root root      1K Dec 19 17:33 config.json
-rw-r--r-- 1 root root 427719K Dec 19 17:33 pytorch_mode
-rw-r--r-- 1 root root      1K Dec 19 17:33 special_toke
-rw-r--r-- 1 root root      1K Dec 19 17:33 tokenizer_co
-rw-r--r-- 1 root root    227K Dec 19 17:33 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 Dec 19 17:33 ./model_save/py
```

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 Dri
!cp -r ./model_save/ "./drive/Shared drives/ChrisMcCormi
```

The following functions will load the model back from disk.

```
# Load a trained model and vocabulary that you have fine
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/5bfcc

# Don't apply weight decay to any parameters whose names
# (Here, the BERT doesn't have `gamma` or `beta` paramet
no_decay = ['bias', 'LayerNorm.weight']

# Separate the `weight` parameters from the `bias` param
# - For the `weight` parameters, this specifies a 'weigh
# - For the `bias` parameters, the 'weight_decay_rate' i
optimizer_grouped_parameters = [
    # Filter for all parameters which *don't* include 'b
    {'params': [p for n, p in param_optimizer if not any
     'weight_decay_rate': 0.1},

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

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

38 Comments **mccormickml.com** **Disqus' Privacy Policy** 1 **Login**

Recommend 8 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

FHmida • 3 months ago • edited

Thanks for this helpful tutorial! I would like to add other outputs to my model in order to have a multi-output Bert model.
As example, what can I do if I want to build a joint Bert model for Slot Filling (BertForTokenClassification) and Intent detection (BertForSequenceClassification) at the same time?
Thanks!

5 • Reply • Share ›

Ilknur Cakir • 2 months ago

Hi Chris,

thank you for the post, it is brilliant and very helpful. I run the code for 3-class classification, then saved the fine-tuned model. When I re-loaded it from the directory, I see that it does not have a classifier at the end, it does not seem to be the model I saved to be exact. Maybe it loads pre-trained BERT instead of fine-tuned model? Do you have any idea why it happens?
Thanks a lot

1 • Reply • Share ›

FHmida → Ilknur Cakir • 17 days ago

Hello Cakir, your label2index (and index2label) must be the same every time you run your program (and call the model). Just sort your labels to have the same mapped index and labels when you built your dictionary.

• Reply • Share ›

Tiago Duque • 4 months ago

Thanks for the straightforward tutorial!

However, I have a question.
I have a task where I need to use the word prediction ability itself (the masking process) and I have very piffy results using the pretrained model. I don't have labels to predict, I just want to improve the model (in this case, the multilingual

Related posts

Question Answering with a Fine-Tuned BERT 10 Mar 2020
BERT Research - Ep. 1 - Key Concepts & Sources 11 Nov 2019
GLUE Explained: Understanding BERT Through Benchmarks 05 Nov 2019