Text classification with transformers in Tensorflow 2: BERT, XLNet

David Mráz

Introduction

The transformer-based language models have been showing promising progress on a number of different natural language processing (NLP) benchmarks. The combination of transfer learning methods with large-scale transformer language models is becoming a standard in modern NLP. In this article, we will make the necessary theoretical introduction to transformer architecture and text classification problem. Then we will demonstrate the fine-

sequences of text in documents as

$$D = X_1, X_2, \cdots, X_N,$$

where X_i can be for example text segment and N is the number of such text segments in D.

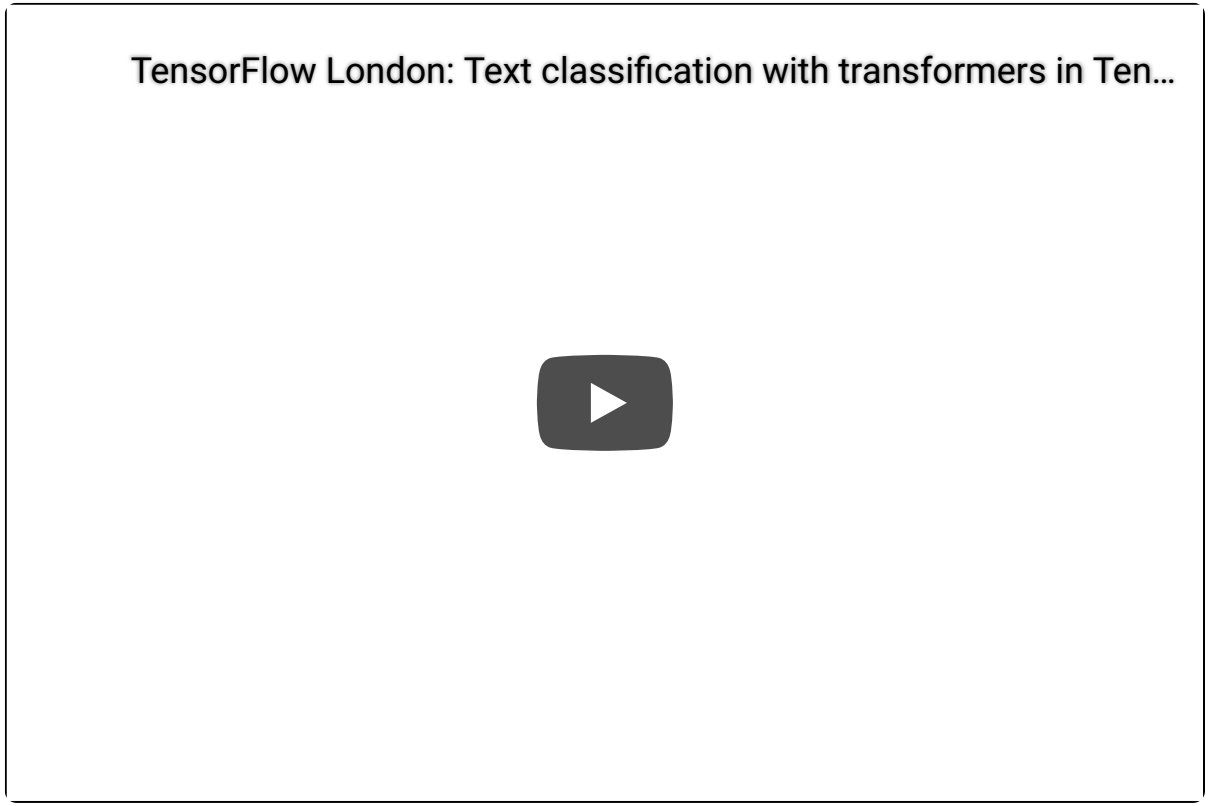
The algorithm that implements classification is called a **classifier**. The text classification tasks can be divided into different groups based on the nature of the task:

- **multi-class classification**
- **multi-label classification**

Multi-class classification is also known as a **single-label problem**, e.g. we assign each instance to only one label. **Multi** in the name means that we deal with at least 3 classes, for 2 classes we can use the term **binary classification**. On the other hand, multi-label classification task is more general and allows us to assign multiple labels to each instance, not just one label per example.

Why transformers?

tuning process of the pre-trained BERT model for text classification in TensorFlow 2 with Keras API.



Text classification - problem formulation

Classification, in general, is a problem of identifying the category of a new observation. We have dataset D, which contains

We will not go into much detail on transformer architecture in this post. However, it is useful to know some of the challenges in NLP. There are two important concepts in NLP, which are complementary:

- word embeddings
- language model

Transformers are used to build the language model, where the embeddings can be retrieved as the by-product of pretraining.

Approaches based on RNNs/LSTMs

Most older methods for language modelling are based on RNNs (recurrent neural network). The simple RNNs suffer from the problem known as **vanishing gradient problem** and therefore fail to model the longer contextual dependencies. They were mostly replaced by the so-called **long short-term neural networks (LSTMs)**, which is also a form of RNN but can capture the longer context in the documents. However, LSTM can process sequences only unidirectional, so the state of the art approaches based on LSTMs evolved into the so-called bidirectional LSTMs, where we can read the context left to right and also right to left. There are

very successful models based on LSTMs such as ELMO or ULMFiT and such models are still valid for today's modern NLP.

Approaches based on transformer architecture

One of the main limitations of bidirectional LSTMs is its sequential nature, which makes training in parallel very difficult. The **transformer architecture** solves that by **completely replacing LSTMs by the so-called attention mechanism** (Vashvani et al. 2017). With attention, we are seeing an entire sequence as a whole, therefore it is much easier to train in parallel. We can model the whole document context as well as to use huge datasets to pre-train in an unsupervised way and fine-tune on downstream tasks.

State of the art transformer models

There is a lot of transformer-based language models. The most successful ones are (as of April 2020)

- Transformer (Google Brain/Research)
- BERT (Google Research)
- GPT-2 (OpenAI)
- XLNet (Google Brain)

- CTRL (SalesForce)
- Megatron (NVidia)
- Turing-NLG (Microsoft)

There are **slight differences between models**. BERT has been considered as the state of the art results on many NLP tasks, but now it looks like it is surpassed by XLNet also from Google. XLNet leverages the **permutation language modelling**, which trains an autoregressive model on all possible permutation of words in a sentence. For the purpose of illustration, we will use BERT-based model in this article.

BERT

BERT (Bidirectional Encoder Representations from Transformers) (Devlint et al., 2018) is a method of pretraining language representation. We will not go into much detail, but the main difference from the original transformer (Vaswani et al., 2017) is that **BERT does not have a decoder**, but **stacks 12 encoders** in the basic version and increase the number of encoders for bigger pre-trained models. Such architecture is different from GPT-2 from

OpenAI, which is autoregressive language model suited for natural language generation (NLG).

Tokenizer

Official BERT language models are pre-trained with **WordPiece vocabulary** and use, not just token embeddings, but also **segment embeddings** distinguish between sequences, which are in pairs, e.g. question answering examples. **Position embeddings** are needed in order to inject positional awareness into BERT model as attention mechanism does not consider positions in context evaluation.

The important limitation of BERT to be aware of is that the **maximum length** of the sequence for BERT is **512 tokens**. For **shorter sequence input** than maximum allowed input size, we would need to **add pad tokens [PAD]**. On the other hand, if the sequence is longer, we **need to cut the sequence**. This BERT limitation on the maximum length of the sequence is something that you need to be aware of for longer text segments, see for example this GitHub issue for further solutions.

Very important are also the so-called special tokens, e.g. **[CLS]** token and **[SEP]** tokens. The [CLS] token will be inserted at the

beginning of the sequence, the [SEP] token is at the end. If we deal with sequence pairs we will add additional [SEP] token at the end of the last.

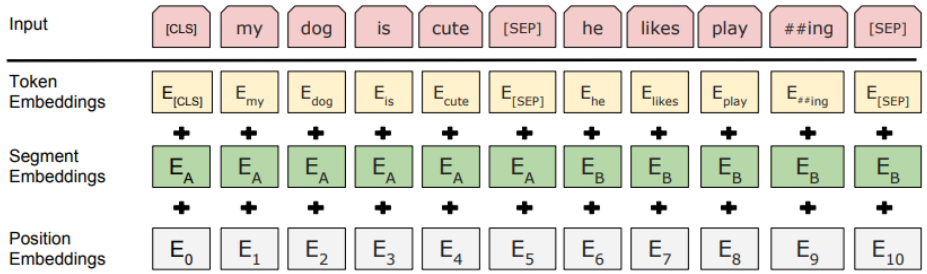


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

When using transformers library we first load the tokenizer for the model we would like to use. Then we will proceed as follows:

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-u

max_length_test = 20
test_sentence = 'Test tokenization sentence. Followed
```

```
# add special tokens

test_sentence_with_special_tokens = '[CLS]' + test_sen

tokenized = tokenizer.tokenize(test_sentence_with_spec

print('tokenized', tokenized)

# convert tokens to ids in WordPiece
input_ids = tokenizer.convert_tokens_to_ids(tokenized)

# precalculation of pad length, so that we can reuse i
padding_length = max_length_test - len(input_ids)

# map tokens to WordPiece dictionary and add pad token
input_ids = input_ids + ([0] * padding_length)

# attention should focus just on sequence with non pac
attention_mask = [1] * len(input_ids)

# do not focus attention on padded tokens
attention_mask = attention_mask + ([0] * padding_lengt
```

```
# token types, needed for example for question answeri
token_type_ids = [0] * max_length_test

bert_input = {
    "token_ids": input_ids,
    "token_type_ids": token_type_ids,
    "attention_mask": attention_mask
} print(bert_input)
```

We can see that the sequence is tokenized, we have added **special tokens** as well as calculate the number of pad tokens needed in order to have the same length of the sequence as the maximal length 20. Then we have added **token types**, which are all the same as we do not have sequence pairs. **Attention mask** will tell the model that we should not focus attention on [PAD] tokens.

```
tokenized ['[CLS]', 'test', 'token', '##ization', 'sen
{
  'token_ids': [101, 3231, 19204, 3989, 6251, 1012, 26
  'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
  'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
}
```

Now in the practical coding we will use just **encode_plus** function, which does all of those steps for us

```
bert_input = tokenizer.encode_plus(
                        test_sentence,
                        add_special_tokens = True, # a
                        max_length = max_length_test,
                        pad_to_max_length = True, # ac
                        return_attention_mask = True,
                )

print('encoded', bert_input)
```

The output is the same as our above code.

Pretraining

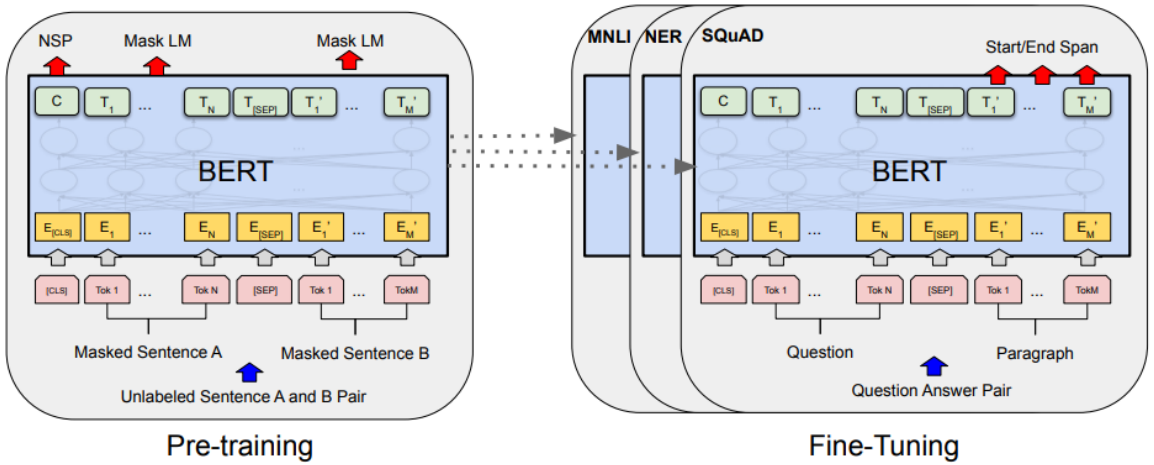
Pretraining is the first phase of BERT training. It is done in an unsupervised way and consists of two main tasks:

- masked language modelling (MLM)
- next sentence prediction (NSP)

From a high level, in MLM task we replace a certain number of tokens in a sequence by **[MASK]** token. We then try to predict the masked tokens. There are some additional rules for MLM, so the description is not completely precise, but feel free to check the original paper (Devlin et al., 2018) for more details.

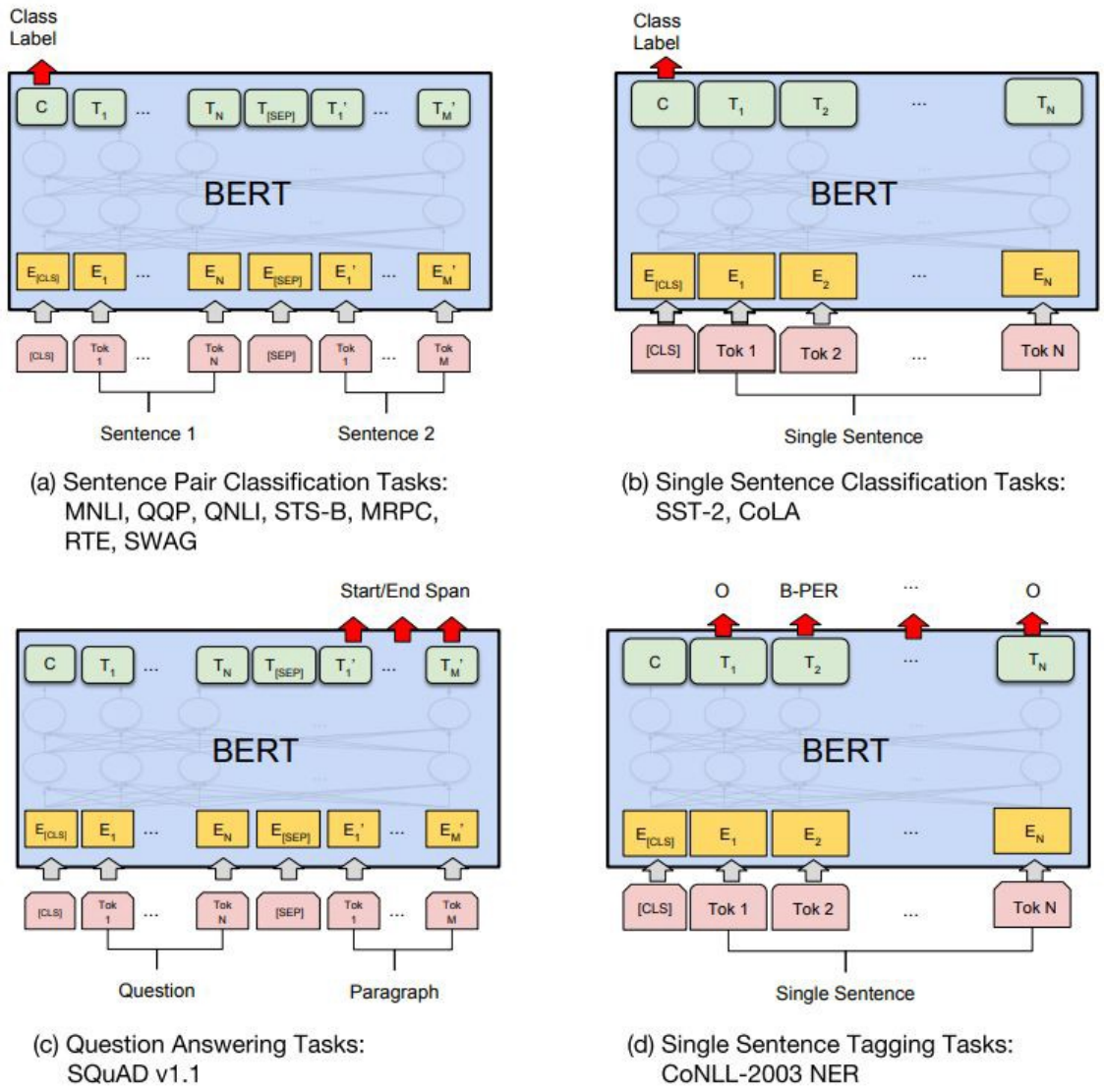
When choosing the sentence pairs for next sentence prediction we will choose 50% of the time the actual sentence that follows the previous sentence and label it as **IsNext**. The other 50% we choose the other sentence from the corpus, not related to the previous one and labels it as **NotNext**.

Both such tasks can be performed on text corpus without labelled examples, therefore the authors used the datasets such as BooksCorpus (800m words), English Wikipedia (2500m words).



Fine-tuning

Once we have either pre-trained our model by ourself or we have loaded already pre-trained model, e.g. BERT-based-uncased, we can start to fine-tune the model on the downstream tasks such as question answering or text classification. We can see that BERT can be applied to many different tasks by adding a task-specific layer on top of pre-trained BERT layer. For text classification, we will just add the simple softmax classifier to the top of BERT.



The pretraining phase takes significant computational power (BERT base: 4 days on 16 TPUs; BERT large 4 days on 64 TPUs),

therefore it is very useful to save the pre-trained models and then fine-tune a one specific dataset. Unline pretraining, the fine-tuning does not require much computation power. The fine-tuning process can be done in a couple of hours even on a single GPU. It is recommended to have at least 12GB VRAM in order to fit the batch size into memory. When fine-tuning for text classification we can choose several paths, see the figure below (Sun et al. 2019).

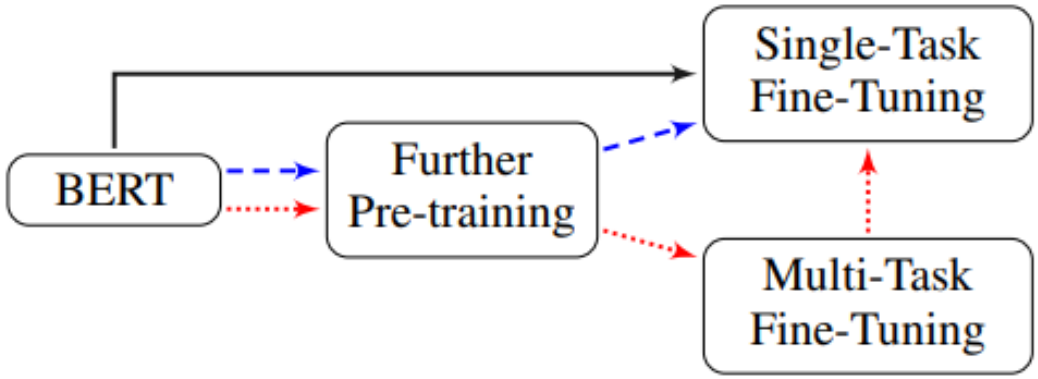


Figure 1: Three general ways for fine-tuning BERT, shown with different colors.

IMDB dataset

We will solve the text classification problem for well-known IMDB movie review dataset. The dataset consists of 50k reviews with assigned sentiment to each. Only highly polarizing reviews are considered and no more than 30 reviews are included per movie. The following are two samples from the dataset:

Review	Sentiment
One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. They are right, as this is exactly what happened with me. The first thing that struck me abo...	positive
Petter Mattei's "Love in the Time of Money" is a visually stunning film to watch. Mr. Mattei offers us a vivid portrait about human relations. This is a movie that seems to be telling us what money, p...	negative

The review can be only positive or negative and only one label can be assigned for each review. This leads us to formulate the problem as a **binary classification**. In addition, we determine the sentiment of each review, therefore we will solve the sub-task of text classification - called **sentiment analysis**.

When we take a look at the already achieved results, we can see that XLNet, as well as BERT, are the transformer-based machine

learning models that achieved best results on IMDB dataset.

Tables	Accuracy
XLNet (Yang et al., 2019)	96.21
BERT_large+ITPT (Sun et al., 2019)	95.79
BERT_base+ITPT (Sun et al., 2019)	95.63
ULMFiT (Howard and Ruder, 2018)	95.4
Block-sparse LSTM (Gray et al., 2017)	94.99

Source: nlpprogress.com

The other two ULMFit (Howard and Ruder, 2018) and Block-sparse LSTM (Gray et al., 2017) are based on LTSMs, not transformer language models. Similar approaches have great results as well but are slowly replaced for some tasks by transformer language models. BERT and XLNet are consistently in top positions also on other text classification benchmarks like AG News, Yelp or DBpedia dataset.
In this article, we will focus on preparing step by step framework for fine-tuning BERT for text classification (sentiment analysis). This framework and code can be also used for other transformer models with minor changes. We will use the smallest BERT model (bert-based-cased) as an example of the fine-tuning process.

Fine tunning BERT with TensorFlow 2 and Keras API

First, the code can be downloaded on Google Colab as well as on GitHub.

Let's use the TensorFlow dataset API for loading IMDB dataset

```
import tensorflow_datasets as tfds

(ds_train, ds_test), ds_info = tfds.load('imdb_reviews
          split = (tfds.Split.TRAIN, tfds.Split.TEST),
          as_supervised=True,
          with_info=True)

print('info', ds_info)
```

The dataset info is as follows:

```
tfds.core.DatasetInfo(
```

```
    name='imdb_reviews',
    version=1.0.0,
    description='Large Movie Review Dataset.
This is a dataset for binary sentiment classification
    homepage='http://ai.stanford.edu/~amaas/data/senti
    features=FeaturesDict({
        'label': ClassLabel(shape=(), dtype=tf.int64,
        'text': Text(shape=(), dtype=tf.string),
    }),
    total_num_examples=100000,
    splits={
        'test': 25000,
        'train': 25000,
        'unsupervised': 50000,
    },
    supervised_keys=('text', 'label'),
    citation=InProceedings{maas-EtAl:2011:ACL-HLT2011,
      author    = {Maas, Andrew L.  and  Daly, Raymond
      title     = {Learning Word Vectors for Sentiment
      booktitle = {Proceedings of the 49th Annual Meet
      month     = {June},
      year      = {2011},
      address   = {Portland, Oregon, USA},
      publisher = {Association for Computational Lingu
```

```
      pages     = {142--150},
      url       = {http:\/\/www.aclweb.org\/anthology\
    },
    redistribution_info=,
)
```

We can see that train and test datasets are split 50:50 and the examples are in the form of **(label, text)**, which can be further validated:

```
for review, label in tfds.as_numpy(ds_train.take(5)):
    print('review', review.decode()[0:50], label)

<<
review This was an absolutely terrible movie. Don't be
review I have been known to fall asleep during films,
review Mann photographs the Alberta Rocky Mountains in
review This is the kind of film for a snowy Sunday aft
review As others have mentioned, all the women that go
```

The positive sentiment is represented by **1** and the negative sentiment is represented by **0**.

Now we need to apply BERT tokenizer on all the examples. We will map tokens into WordPiece embeddings. As said, this can be done using encode_plus function.

```
# map to the expected input to TFBertForSequenceClassi
def map_example_to_dict(input_ids, attention_masks, to
  return {
      "input_ids": input_ids,
      "token_type_ids": token_type_ids,
      "attention_mask": attention_masks,
  }, label

def encode_examples(ds, limit=-1):

  # prepare list, so that we can build up final Tensor
  input_ids_list = []
  token_type_ids_list = []
  attention_mask_list = []
  label_list = []
```

```
if (limit > 0):
    ds = ds.take(limit)

for review, label in tfds.as_numpy(ds):

  bert_input = convert_example_to_feature(review.dec

  input_ids_list.append(bert_input['input_ids'])
  token_type_ids_list.append(bert_input['token_type_
  attention_mask_list.append(bert_input['attention_m
  label_list.append([label])

return tf.data.Dataset.from_tensor_slices((input_ids
```

We can encode the dataset using the following functions:

```
# train dataset
ds_train_encoded = encode_examples(ds_train).shuffle(1

# test dataset
```

```
ds_test_encoded = encode_examples(ds_test).batch(batch
```

```
from transformers import TFBertForSequenceClassificati
import tensorflow as tf

# recommended learning rate for Adam 5e-5, 3e-5, 2e-5

learning_rate = 2e-5

# we will do just 1 epoch for illustration, though mul
number_of_epochs = 1


# model initialization
model = TFBertForSequenceClassification.from_pretraine

# classifier Adam recommended
optimizer = tf.keras.optimizers.Adam(learning_rate=lea

# we do not have one-hot vectors, we can use sparce ca
```

```
loss = tf.keras.losses.SparseCategoricalCrossentropy(f
metric = tf.keras.metrics.SparseCategoricalAccuracy('a

model.compile(optimizer=optimizer, loss=loss, metrics=
```

We have chosen a rather smaller **learning rate 2e-5** and only **1 epoch**. BERT overfit quite quickly on this dataset, so if we would like to do 2 and more epoch it would be useful to add some additional regularization layers or use for example Adam optimizer with weight decay.

Now we have everything needed in order to start fine-tuning. We will use Keras API **model.fit** method:

```
bert_history = model.fit(ds_train_encoded, epochs=numb
```

We have achieved **over 93% accuracy** on our test dataset.

```
4167/4167 [==============================] - 4542s 1s/
- loss: 0.2456 - accuracy: 0.9024
- val_loss: 0.1892 - val_accuracy: 0.9326
```

That looks reasonable in comparison with the current state of the art results. According to (Sun C et al. 2019) we can achieve **up to 95.79 accuracy** with BERT large on this task. The only better accuracy than BERT large on this task has XLNet from Google AI Brain. XLNet can be also easily used with transformers library with just minor changes to the code.

Conclusion

We have developed the end to end process to use transformers on the text classification task. We have achieved great performance with additional ability to improve either by using XLNet or BERT large model. We can also improve accuracy with multi-task fine-tuning, hyperparameter tuning or additional regularization. The process can be adjusted to other NLP tasks with just minor changes to the code.

References

Images for illustration are taken from the original BERT paper (Devlin et al. 2018). Other references are as follows:

- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova,Bert: Pre-training of deep bidirectional transformers for languageunderstanding, 2018.
- Scott Gray, Alec Radford, and Diederik P. Kingma,Gpu kernels forblock-sparse weights, 2017.
- Jeremy Howard and Sebastian Ruder,Universal language modelfine-tuning for text classification, 2018.
- Sepp Hochreiter and J ̈urgen Schmidhuber,Long short-term memory,Neural computation9(1997), no. 8, 1735–1780.
- Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, andIlya Sutskever,Language models are unsupervised multitask learners.
- Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang,How to fine-tunebert for text classification?, 2019.

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, LlionJones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin,Attentionis all you need, 2017.

- Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy,and Samuel R. Bowman,Glue: A multi-task benchmark and analysisplatform for natural language understanding, 2018.David Mraz (Atheros.ai)Transformers in TensorFlow 2April 15, 202013 / 15

- Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, RuslanSalakhutdinov, and Quoc V. Le,Xlnet: Generalized autoregressivepretraining for language understanding, 2019

Did you like this post? You can clone the repository with the examples and project set-up. Feel free to send any questions about the topic to david@atheros.ai and subscribe to get more knowledge about building AI-driven systems.

SERVICES

Software Development

Branding & UX Design

AI & Machine Learning

ATHEROS CLOUD

Text Classification

Time Series Analysis

Natural Language

Interface

COMPANY

Blog

About Us

Open Jobs

Contact

RESOURCES

Support

Terms and conditions

Privacy Policy

Cookie Policy

hello@atheros.ai

Copyright © 2020 all rights reserved. Atheros Intelligence Ltd ('Atheros') is a registered company in England and Wales, registration number 11776379