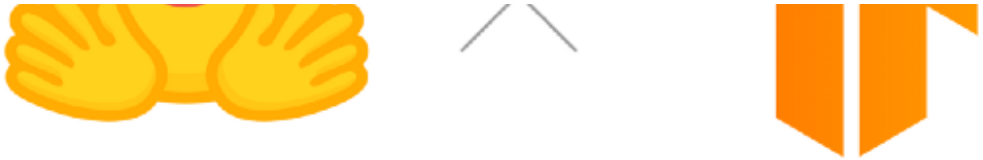
Text Classification with Hugging Face Transformers in TensorFlow 2 (Without Tears)

A

Arun Maiya Follow

Jan 14 · 7 min read ★



Source

The Hugging Face transformers package is an immensely popular Python library providing pretrained models that are extraordinarily useful for a variety of natural language processing (NLP) tasks. It previously supported only **PyTorch**, but, as of late 2019, **TensorFlow 2** is supported as well. While the library can be used for many tasks from Natural Language Inference (NLI) to Question-Answering, text classification remains one of the most popular and practical use cases.

The *ktrain* library is a lightweight wrapper for `tf.keras` in TensorFlow 2. It is designed to make deep learning and AI more accessible and easier to apply for beginners and domain experts. As of version *0.8*, *ktrain* now includes a **simplified interface to Hugging Face transformers** for text classification. In this article, we will show you how you can build, train, and deploy a text classification model with **Hugging Face transformers** in only a few lines of code.

The **source code for this article** is available in two forms:

- this Google Colab notebook or
- a tutorial notebook on *ktrain*'s GitHub repository

Getting Started

Let's begin by installing *ktrain*. After ensuring TensorFlow 2 is installed on your system, you can install *ktrain* with:

```
pip3 install ktrain
```

In this article, we will use the 20newsgroups dataset to construct a small training set with four newsgroup categories. The objective is to build a model that can predict the newsgroup category for a given post. This will provide us an opportunity to see **transformers** in action on a relatively smaller training set, which is one of the powerful advantages of transfer learning. Incidentally, this is the same dataset employed in the scikit-learn Working with Text Data tutorial. Let's fetch the 20newsgroups dataset using scikit-learn and load everything into arrays for the training and validation:

Next, we must select one of the pretrained models from Hugging Face, which are all listed here. As of this writing, the **transformers** library supports the following pretrained models for TensorFlow 2:

- **BERT**: *bert-base-uncased, bert-large-uncased, bert-base-multilingual-uncased, and others.*
- **DistilBERT**: *distilbert-base-uncased, distilbert-base-multilingual-cased, distilbert-base-german-cased, and others*
- **ALBERT:** *albert-base-v2, albert-large-v2, and others*

- **RoBERTa**: *roberta-base, roberta-large, roberta-large-mnli*
- **XLM**: *xlm-mlm-xnli15–1024, xlm-mlm-100–1280, and others*
- **XLNet**: *xlnet-base-cased, xlnet-large-cased*

Dealing with Problems

In practice, since TensorFlow support in the Hugging Face transformers library is relatively new, several of the models listed above produce errors at the time of this writing. Examples include:

- `distilbert-base-multilingual-cased`: See issue 2423
- `xlnet-base-cased`: See issue 1692

For instance, as of v2.3.0 of the **transformers library**, the `distil-bert-uncased` model works nicely, but the `distilbert-base-multilingual-cased` model throws an exception during training.

The Hugging Face team is working hard to resolve such issues. Some issues already have merged but unreleased resolutions. For instance, problems related to *XLNet* in **transformers-v2.3.0** can be addressed by simply

replacing line 555 in `modeling_tf_xlnet.py` from the **transformers** library with:

```
input_mask = 1.0 - tf.cast(attention_mask, dtype=dtype_float)
```

as described in this PR. If encountering issues with a particular model, you can try searching the issues on the transformers GitHub repository for a solution in the form of a patch to the **transformers** library.

Selecting the DistilBERT Model

In the mean time, for the purposes of this tutorial, we will demonstrate a popular and extremely useful model that has been verified to work in *v2.3.0* of the **transformers** library (the current version at the time of this writing). The BERT model represents one of the major AI breakthroughs of 2018–2019 by achieving state-of-the-art performance across 11 different natural language processing tasks. Unfortunately, BERT is also a very large and memory-hungry model that is slow for both training and inference. Thus, BERT is not very suitable for production environments. DistilBERT is a "distilled" version of BERT that is smaller and faster while retaining most of

BERT's accuracy. For these reasons, we will use an uncased, English-language **DistilBERT** model in this tutorial:

Next, we will use *ktrain* to easily and quickly build, train, inspect, and evaluate the model.

STEP 1: Create a Transformer instance

The `Transformer` class in *ktrain* is a simple abstraction around the Hugging Face transformers library. Let's instantiate one by providing the model name, the sequence length (i.e., `maxlen` argument) and populating the `classes` argument with a list of target names.

Note that the maximum sequence length for BERT-based models is typically 512. Documents less than `maxlen` tokens will be padded, and documents greater than `maxlen` tokens will be truncated.

STEP 2: Preprocess the Datasets

We then preprocess the training and validation datasets into the format expected by the selected pretrained model (in this case DistilBERT).

STEP 3: Create a Model and Wrap in Learner

Next, we define a classifier with pretrained weights and randomly initialized final layers that can be fine-tuned. The model will be wrapped in a *ktrain* `Learner` object that will allow us to easily train and inspect the model and use it to make predictions on new data.

If you experience out-of-memory errors during training, you can either try lowering the `batch_size` above or lowering the `maxlen` parameter in STEP 1.

STEP 4 [optional]: Estimate the Learning Rate

We will use the Learning Rate Finder in *ktrain* to estimate a good learning rate for our model and dataset.

For BERT-based models, learning rates between *2e-5* and *5e-5* generally work well across a wide range of datasets. Thus, this step is optional.

Here, we will select the highest learning rate associated with a falling loss. Let us choose 5**e-5** as the learning rate**.** Interestingly, the learning rate finder's estimation (as shown in the plot) is consistent with the learning rate range that Google reported to typically work best for BERT and other transformer models.

STEP 5: Train the Model

For training, we will invoke the `fit_onecycle` method in *ktrain*, which employs the 1cycle policy, a learning rate schedule proposed by Leslie Smith. In general, learning rate schedules with an initial warmup period that increases the learning rate and then a decay period that gradually decreases the learning rate tend to work well for transformer-based models.

After four epochs, our validation accuracy is **96.27%,** which is quite a bit better than the 91% accuracy achieved by SVM in the scikit-learn tutorial on text classification.

STEP 6: Inspect the Model

Let's invoke the `view_top_losses` method to examine the newsgroup posting that our model is getting the most wrong.

As you can see, the newgroup posting associated with ID 1355 in the validation set is in the `comp.graphics` newsgroup (i.e., computer graphics), but is largely about **color blindness**, which is a medical topic. Thus, our model's prediction of `sci.med` for this post is understandable and forgivable.

STEP 7: Make Predictions on New Data

We can instantiate a `Predictor` object to easily make predictions on new examples.

The `predictor` can also be used to further inspect the model by explaining the classification of a particular example via the eli5 and lime libraries:

The words highlighted in green are those that appear to be causing our model to place this example in the `soc.religion.christian` category. As you can see, the highlighted words all agree with intuition for this category.

Note that, for `explain` to work, you will need to first install a forked version of the eli5 library modified to support TensorFlow Keras:

```
pip3 install git+https://github.com/amaiya/eli5@tfkeras_0_10_1
```

Finally, the `predictor` object can be saved to disk and reloaded later in real-world deployment scenarios:

By default, `predict` returns the predicted class label, but `predict_proba` will return the predicted probabilities for each class, as shown above.

Conclusion

ktrain makes it easy to experiment with different **transformer** models. For instance, a traditional BERT model can be easily trained for comparison

with the DistilBERT model trained above by simply replacing the `MODEL_NAME` as follows:

```
import ktrain
from ktrain import text
MODEL_NAME = 'bert-base-uncased'
t = text.Transformer(MODEL_NAME, maxlen=500,
                     classes=train_b.target_names)
trn = t.preprocess_train(x_train, y_train)
val = t.preprocess_test(x_test, y_test)
model = t.get_classifier()
learner = ktrain.get_learner(model, train_data=trn, val_data=val,
batch_size=6)
learner.fit_onecycle(3e-5, 1)
```

Regardless of the `MODEL_NAME` chosen, the underlying model wrapped in the `Learner` object is just another **TensorFlow Keras** model:

Thus, once trained, it can be managed directly using **TensorFlow** and/or the **transformers** library itself if one wishes. Feel free to try out *ktrain* on your next text classification project.

For more information on *ktrain*, please visit our GitHub repository.

Machine Learning Deep Learning NLP TensorFlow Keras