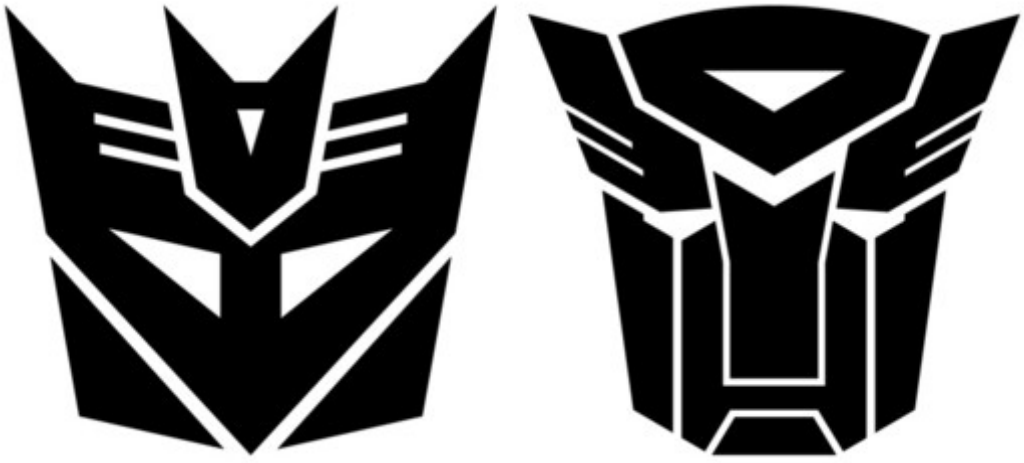
Transformers

Introduction to LLMs in Social Sciences



ACSPRI | Dr Maria Prokofieva

Quick tour

Let's have a quick look at the 🤗 Transformers library features. The library downloads pretrained models for Natural Language Understanding (NLU) tasks, such as analyzing the sentiment of a text, and Natural Language Generation (NLG), such as completing a prompt with new text or translating in another language.

First we will see how to easily leverage the pipeline API to quickly use those pretrained models at inference. Then, we will dig a little bit more and see how the library gives you access to those models and helps you preprocess your data.

> **Note**
>
> All code examples presented in the documentation have a switch on the top left for Pytorch versus TensorFlow. If not, the code is expected to work for both backends without any change needed.

Getting started on a task with a pipeline

The easiest way to use a pretrained model on a given task is to use `pipeline()`. 🤗 Transformers provides the following tasks out of the box:

- Sentiment analysis: is a text positive or negative?
- Text generation (in English): provide a prompt and the model will generate what follows.
- Name entity recognition (NER): in an input sentence, label each word with the entity it represents (person, place, etc.)
- Question answering: provide the model with some context and a question, extract the answer from the context.
- Filling masked text: given a text with masked words (e.g., replaced by `[MASK]`), fill the blanks.
- Summarization: generate a summary of a long text.
- Translation: translate a text in another language.
- Feature extraction: return a tensor representation of the text.

Let's see how this work for sentiment analysis (the other tasks are all covered in the task summary):

```
>>> from transformers import pipeline
>>> classifier = pipeline('sentiment-analysis')
```


When typing this command for the first time, a pretrained model and its tokenizer are downloaded and cached. We will look at both later on, but as an introduction the tokenizer's job is to preprocess the text for the model, which is then responsible for making predictions. The pipeline groups all of that together, and post-process the predictions to make them readable. For instance:

```
>>> classifier('We are very happy to show you the 🤗 Transformers library.')
[{'label': 'POSITIVE', 'score': 0.9997795224189758}]
```

That's encouraging! You can use it on a list of sentences, which will be preprocessed then fed to the model as a *batch*, returning a list of dictionaries like this one:

```
>>> results = classifier(["We are very happy to show you the 🤗 Transformers
library.",
...            "We hope you don't hate it."])
>>> for result in results:
...     print(f"label: {result['label']}, with score: {round(result['score'], 4)}")
label: POSITIVE, with score: 0.9998
label: NEGATIVE, with score: 0.5309
```

You can see the second sentence has been classified as negative (it needs to be positive or negative) but its score is fairly neutral.

By default, the model downloaded for this pipeline is called "distilbert-base-uncased-finetuned-sst-2-english". We can look at its model page to get more information about it. It uses the DistilBERT architecture and has been fine-tuned on a dataset called SST-2 for the sentiment analysis task.

Let's say we want to use another model; for instance, one that has been trained on French data. We can search through the model hub that gathers models pretrained on a lot of data by research labs, but also community models (usually fine-tuned versions of those big models on a specific dataset). Applying the tags "French" and "text-classification" gives back a suggestion "nlptown/bert-base-multilingual-uncased-sentiment". Let's see how we can use it.

You can directly pass the name of the model to use to `pipeline()`:

```
>>> classifier = pipeline('sentiment-analysis', model="nlptown/bert-base-multilingual-
uncased-sentiment")
```

This classifier can now deal with texts in English, French, but also Dutch, German, Italian and Spanish! You can also replace that name by a local folder where you have saved a pretrained model (see below). You[3] can also pass a model object and its associated tokenizer.

We will need two classes for this. The first is `AutoTokenizer`, which we will use to download the tokenizer associated to the model we picked and instantiate it. The second is `AutoModelForSequenceClassification` (or `TFAutoModelForSequenceClassification` if you are using TensorFlow), which we will use to download the model itself. Note that if we were using the library on an other task, the class of the model would change. The task summary tutorial summarizes which class is used for which task.

PyTorch | TensorFlow

```
>>> from transformers import AutoTokenizer, AutoModelForSequenceClassification
```

Now, to download the models and tokenizer we found previously, we just have to use the `from_pretrained()` method (feel free to replace `model_name` by any other model from the model hub):

PyTorch | TensorFlow

```
>>> model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
>>> model = AutoModelForSequenceClassification.from_pretrained(model_name)
>>> tokenizer = AutoTokenizer.from_pretrained(model_name)
>>> pipe = pipeline('sentiment-analysis', model=model, tokenizer=tokenizer)
```

If you don't find a model that has been pretrained on some data similar to yours, you will need to fine-tune a pretrained model on your data. We provide example scripts to do so. Once you're done, don't forget to share your fine-tuned model on the hub with the community, using this tutorial.

Under the hood: pretrained models

Let's now see what happens beneath the hood when using those pipelines. As we saw, the model and tokenizer are created using the `from_pretrained` method:

PyTorch | TensorFlow

```
>>> from transformers import AutoTokenizer, AutoModelForSequenceClassification
>>> model_name = "distilbert-base-uncased-finetuned-sst-2-english"
>>> pt_model = AutoModelForSequenceClassification.from_pretrained(model_name)
>>> tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Using the tokenizer

We mentioned the tokenizer is responsible for the preprocessing of your texts. First, it will split a given text in words (or part of words, punctuation symbols, etc.) usually called *tokens*. There are multiple rules that can govern that process (you can learn more about them in the tokenizer_summary, which is why we need to instantiate the tokenizer using the name of the model, to make sure we use the same rules as when the model was pretrained.

The second step is to convert those *tokens* into numbers, to be able to build a tensor out of them and feed them to the model. To do this, the tokenizer has a *vocab*, which is the part we download when we instantiate it with the `from_pretrained` method, since we need to use the same *vocab* as when the model was pretrained.

To apply these steps on a given text, we can just feed it to our tokenizer:

```
>>> inputs = tokenizer("We are very happy to show you the 🤗 Transformers library.")
```

This returns a dictionary string to list of ints. It contains the ids of the tokens, as mentioned before, but also additional arguments that will be useful to the model. Here for instance, we also have an attention mask that the model will use to have a better understanding of the sequence:

```
>>> print(inputs)
{'input_ids': [101, 2057, 2024, 2200, 3407, 2000, 2265, 2017, 1996, 100, 19081, 3075,
1012, 102], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

You can pass a list of sentences directly to your tokenizer. If your goal is to send them through your model as a batch, you probably want to pad them all to the same length, truncate them to the maximum length the model can accept and get tensors back. You can specify all of that to the tokenizer:

PyTorch | TensorFlow

```
>>> pt_batch = tokenizer(
...     ["We are very happy to show you the 🤗 Transformers library.", "We hope you
don't hate it."],
...     padding=True,
...     truncation=True,
...     return_tensors="pt"
... )
```

The padding is automatically applied on the side the model expect it (in this case, on the right), with the padding token the model was pretrained with. The attention mask is also adapted to take the padding into account:

PyTorch | TensorFlow

```
>>> for key, value in pt_batch.items():
...     print(f"{key}: {value.numpy().tolist()}")
input_ids: [[101, 2057, 2024, 2200, 3407, 2000, 2265, 2017, 1996, 100, 19081, 3075,
1012, 102], [101, 2057, 3246, 2017, 2123, 1005, 1056, 5223, 2009, 1012, 102, 0, 0, 0]]
attention_mask: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 0, 0, 0]]
```

You can learn more about tokenizers here.

Using the model

Once your input has been preprocessed by the tokenizer, you can directly send it to the model. As we mentioned, it will contain all the relevant information the model needs. If you're using a TensorFlow model, you can directly pass the dictionary keys to tensor, for a PyTorch model, you need to unpack the dictionary by adding `**`.

PyTorch | TensorFlow

```
>>> pt_outputs = pt_model(**pt_batch)
```

In 🤗 Transformers, all outputs are tuples (with only one element potentially). Here, we get a tuple with just the final activations of the model.

PyTorch | TensorFlow

```
>>> print(pt_outputs)
(tensor([[-4.0833,  4.3364],
        [ 0.0818, -0.0418]], grad_fn=<AddmmBackward>),)
```

Note

All 🤗 Transformers models (PyTorch or TensorFlow) return the activations of the model *before* the final activation function (like SoftMax) since this final activation function is often fused with the loss.

Let's apply the SoftMax activation to get predictions.

PyTorch | TensorFlow

```
>>> import torch.nn.functional as F
>>> pt_predictions = F.softmax(pt_outputs[0], dim=-1)
```

We can see we get the numbers from before:

PyTorch | TensorFlow

```
>>> print(pt_predictions)
tensor([[2.2043e-04, 9.9978e-01],
        [5.3086e-01, 4.6914e-01]], grad_fn=<SoftmaxBackward>)
```

If you have labels, you can provide them to the model, it will return a tuple with the loss and the final activations.

PyTorch | TensorFlow

```
>>> import torch
>>> pt_outputs = pt_model(**pt_batch, labels = torch.tensor([1, 0]))
```

Models are standard torch.nn.Module or tf.keras.Model so you can use them in your usual training loop. 🤗 Transformers also provides a `Trainer` (or `TFTrainer` if you are using TensorFlow) class to help with your training (taking care of things such as distributed training, mixed precision, etc.). See the training tutorial for more details.

Once your model is fine-tuned, you can save it with its tokenizer the following way:

```
tokenizer.save_pretrained(save_directory)
model.save_pretrained(save_directory)
```

You can then load this model back using the `from_pretrained()` method by passing the directory name instead of the model name. One cool feature of 🤗 Transformers is that you can easily switch between PyTorch and TensorFlow: any model saved as before can be loaded back either in PyTorch or TensorFlow. If you are loading a saved PyTorch model in a TensorFlow model, use `from_pretrained()` like this:

```
tokenizer = AutoTokenizer.from_pretrained(save_directory)
model = TFAutoModel.from_pretrained(save_directory, from_pt=True)
```

and if you are loading a saved TensorFlow model in a PyTorch model, you should use the following code:

```
tokenizer = AutoTokenizer.from_pretrained(save_directory)
model = AutoModel.from_pretrained(save_directory, from_tf=True)
```

Lastly, you can also ask the model to return all hidden states and all attention weights if you need them:

PyTorch | TensorFlow

```
>>> pt_outputs = pt_model(**pt_batch, output_hidden_states=True,
output_attentions=True)
>>> all_hidden_states, all_attentions = pt_outputs[-2:]
```

Accessing the code

The `AutoModel` and `AutoTokenizer` classes are just shortcuts that will automatically work with any pretrained model. Behind the scenes, the library has one model class per combination of architecture plus class, so the code is easy to access and tweak if you need to.

In our previous example, the model was called “distilbert-base-uncased-finetuned-sst-2-english”, which means it’s using the DistilBERT architecture. The model automatically created is then a `DistilBertForSequenceClassification`. You can look at its documentation for all details relevant to that specific model, or browse the source code. This is how you would directly instantiate model and tokenizer without the auto magic:

PyTorch | TensorFlow

```
>>> from transformers import DistilBertTokenizer, DistilBertForSequenceClassification
>>> model_name = "distilbert-base-uncased-finetuned-sst-2-english"
>>> model = DistilBertForSequenceClassification.from_pretrained(model_name)
>>> tokenizer = DistilBertTokenizer.from_pretrained(model_name)
```

Customizing the model

If you want to change how the model itself is built, you can define your custom configuration class. Each architecture comes with its own relevant configuration (in the case of DistilBERT, `DistilBertConfig`) which allows you to specify any of the hidden dimension, dropout rate etc. If you do core modifications, like changing the hidden size, you won’t be able to use a pretrained model anymore and will need to train from scratch. You would then instantiate the model directly from this configuration.

Here we use the predefined vocabulary of DistilBERT (hence load the tokenizer with the `from_pretrained()` method) and initialize the model from scratch (hence instantiate the model from the configuration instead of using the `from_pretrained()` method).

PyTorch | TensorFlow

```
>>> from transformers import DistilBertConfig, DistilBertTokenizer,
DistilBertForSequenceClassification
>>> config = DistilBertConfig(n_heads=8, dim=512, hidden_dim=4*512)
>>> tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
>>> model = DistilBertForSequenceClassification(config)
```

For something that only changes the head of the model (for instance, the number of labels), you can still use a pretrained model for the body. For instance, let's define a classifier for 10 different labels using a pretrained body. We could create a configuration with all the default values and just change the number of labels, but more easily, you can directly pass any argument a configuration would take to the `from_pretrained()` method and it will update the default configuration with it:

PyTorch | TensorFlow

```
>>> from transformers import DistilBertConfig, DistilBertTokenizer,
DistilBertForSequenceClassification
>>> model_name = "distilbert-base-uncased"
>>> model = DistilBertForSequenceClassification.from_pretrained(model_name,
num_labels=10)
>>> tokenizer = DistilBertTokenizer.from_pretrained(model_name)
```

Installation

🤗 Transformers is tested on Python 3.6+, and PyTorch 1.1.0+ or TensorFlow 2.0+.

You should install 🤗 Transformers in a virtual environment. If you're unfamiliar with Python virtual environments, check out the user guide. Create a virtual environment with the version of Python you're going to use and activate it.

Now, if you want to use 🤗 Transformers, you can install it with pip. If you'd like to play with the examples, you must install it from source.

Installation with pip

First you need to install one of, or both, TensorFlow 2.0 and PyTorch. Please refer to TensorFlow installation page and/or PyTorch installation page regarding the specific install command for your platform.

When TensorFlow 2.0 and/or PyTorch has been installed, 🤗 Transformers can be installed using pip as follows:

```
pip install transformers
```

Alternatively, for CPU-support only, you can install 🤗 Transformers and PyTorch in one line with

```
pip install transformers[torch]
```

or 🤗 Transformers and TensorFlow 2.0 in one line with

```
pip install transformers[tf-cpu]
```

To check 🤗 Transformers is properly installed, run the following command:

```
python -c "from transformers import pipeline; print(pipeline('sentiment-analysis')('I hate you'))"
```

It should download a pretrained model then print something like

```
[{'label': 'NEGATIVE', 'score': 0.9991129040718079}]
```

(Note that TensorFlow will print additional stuff before that last statement.)

Installing from source

To install from source, clone the repository and install with the following commands:

```
git clone https://github.com/huggingface/transformers.git
cd transformers
pip install -e .
```

Again, you can run

```
python -c "from transformers import pipeline; print(pipeline('sentiment-analysis')('I hate you'))"
```

to check 🤗 Transformers is properly installed.

Caching models

This library provides pretrained models that will be downloaded and cached locally. Unless you specify a location with `cache_dir=...` when you use methods like `from_pretrained`, these models will automatically be downloaded in the folder given by the shell environment variable `TRANSFORMERS_CACHE`. The default value for it will be the PyTorch cache home followed by `/transformers/` (even if you don't have PyTorch installed). This is (by order of priority):

- shell environment variable `ENV_TORCH_HOME`
- shell environment variable `ENV_XDG_CACHE_HOME` + `/torch/`
- default: `~/.cache/torch/`

So if you don't have any specific environment variable set, the cache directory will be at `~/.cache/torch/transformers/`.

Note: If you have set a shell enviromnent variable for one of the predecessors of this library (`PYTORCH_TRANSFORMERS_CACHE` or `PYTORCH_PRETRAINED_BERT_CACHE`), those will be used if there is no shell enviromnent variable for `TRANSFORMERS_CACHE`.

Note on model downloads (Continuous Integration or large-scale deployments)

If you expect to be downloading large volumes of models (more than 1,000) from our hosted bucket (for instance through your CI setup, or a large-scale production deployment), please cache the model files on your end. It will be way faster, and cheaper. Feel free to contact us privately if you need any help.

Do you want to run a Transformer model on a mobile device?

You should check out our swift-coreml-transformers repo.

It contains a set of tools to convert PyTorch or TensorFlow 2.0 trained Transformer models (currently contains `GPT-2`, `DistilGPT-2`, `BERT`, and `DistilBERT`) to CoreML models that run on iOS devices.

At some point in the future, you'll be able to seamlessly move from pre-training or fine-tuning models in PyTorch or TensorFlow 2.0 to productizing them in CoreML, or prototype a model or an app in CoreML then research its hyperparameters or architecture from PyTorch or TensorFlow 2.0. Super exciting!

Glossary

General terms

- autoencoding models: see MLM
- autoregressive models: see CLM
- CLM: causal language modeling, a pretraining task where the model reads the texts in order and has to predict the next word. It's usually done by reading the whole sentence but using a mask inside the model to hide the future tokens at a certain timestep.
- MLM: masked language modeling, a pretraining task where the model sees a corrupted version of the texts, usually done by masking some tokens randomly, and has to predict the original text.
- multimodal: a task that combines texts with another kind of inputs (for instance images).
- NLG: natural language generation, all tasks related to generating text (for instance talk with transformers, translation)
- NLP: natural language processing, a generic way to say "deal with texts".
- NLU: natural language understanding, all tasks related to understanding what is in a text (for instance classifying the whole text, individual words)
- pretrained model: a model that has been pretrained on some data (for instance all of Wikipedia). Pretraining methods involve a self-supervised objective, which can be reading the text and trying to predict the next word (see CLM) or masking some words and trying to predict them (see MLM).
- RNN: recurrent neural network, a type of model that uses a loop over a layer to process texts.
- seq2seq or sequence-to-sequence: models that generate a new sequence from an input, like translation models, or summarization models (such as Bart or T5).

- token: a part of a sentence, usually a word, but can also be a subword (non-common words are often split in subwords) or a punctuation symbol.

Model inputs

Every model is different yet bears similarities with the others. Therefore most models use the same inputs, which are detailed here alongside usage examples.

Input IDs

The input ids are often the only required parameters to be passed to the model as input. *They are token indices, numerical representations of tokens building the sequences that will be used as input by the model.*

Each tokenizer works differently but the underlying mechanism remains the same. Here's an example using the BERT tokenizer, which is a WordPiece tokenizer:

```
>>> from transformers import BertTokenizer
>>> tokenizer = BertTokenizer.from_pretrained("bert-base-cased")

>>> sequence = "A Titan RTX has 24GB of VRAM"
```

The tokenizer takes care of splitting the sequence into tokens available in the tokenizer vocabulary.

```
>>> tokenized_sequence = tokenizer.tokenize(sequence)
```

The tokens are either words or subwords. Here for instance, "VRAM" wasn't in the model vocabulary, so it's been split in "V", "RA" and "M". To indicate those tokens are not separate words but parts of the same word, a double-dash is added for "RA" and "M":

```
>>> print(tokenized_sequence)
['A', 'Titan', 'R', '##T', '##X', 'has', '24', '##GB', 'of', 'V', '##RA',
'##M']
```

These tokens can then be converted into IDs which are understandable by the model. This can be done by directly feeding the sentence to the tokenizer, which leverages the Rust implementation of huggingface/tokenizers for peak performance.

```
>>> encoded_sequence = tokenizer(sequence)["input_ids"]
```

The tokenizer returns a dictionary with all the arguments necessary for its corresponding model to work properly. The token indices are under the key “input_ids”:

```
>>> print(encoded_sequence)
[101, 138, 18696, 155, 1942, 3190, 1144, 1572, 13745, 1104, 159, 9664, 2107,
102]
```

Note that the tokenizer automatically adds “special tokens” (if the associated model rely on them) which are special IDs the model sometimes uses. If we decode the previous sequence of ids,

```
>>> decoded_sequence = tokenizer.decode(encoded_sequence)
```

we will see

```
>>> print(decoded_sequence)
[CLS] A Titan RTX has 24GB of VRAM [SEP]
```

because this is the way a `BertModel` is going to expect its inputs.

Attention mask

The attention mask is an optional argument used when batching sequences together. This argument indicates to the model which tokens should be attended to, and which should not.

For example, consider these two sequences:

```
>>> from transformers import BertTokenizer
>>> tokenizer = BertTokenizer.from_pretrained("bert-base-cased")

>>> sequence_a = "This is a short sequence."
>>> sequence_b = "This is a rather long sequence. It is at least longer than
the sequence A."

>>> encoded_sequence_a = tokenizer(sequence_a)["input_ids"]
>>> encoded_sequence_b = tokenizer(sequence_b)["input_ids"]
```

The encoded versions have different lengths:

```
>>> len(encoded_sequence_a), len(encoded_sequence_b)
(8, 19)
```

Therefore, we can't be put then together in a same tensor as-is. The first sequence needs to be padded up to the length of the second one, or the second one needs to be truncated down to the length of the first one.

In the first case, the list of IDs will be extended by the padding indices. We can pass a list to the tokenizer and ask it to pad like this:

```
>>> padded_sequences = tokenizer([sequence_a, sequence_b], padding=True)
```

We can see that 0s have been added on the right of the first sentence to make it the same length as the second one:

```
>>> padded_sequences["input_ids"]
[[101, 1188, 1110, 170, 1603, 4954, 119, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0], [101, 1188, 1110, 170, 1897, 1263, 4954, 119, 1135, 1110, 1120, 1655,
2039, 1190, 1103, 4954, 138, 119, 102]]
```

This can then be converted into a tensor in PyTorch or TensorFlow. The attention mask is a binary tensor indicating the position of the padded indices so that the model does not attend to them. For the `BertTokenizer`, `1` indicate a value that should be attended to while `0` indicate a padded value. This attention mask is in the dictionary returned by the tokenizer under the key "attention_mask":

```
>>> padded_sequences["attention_mask"]
[[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

Token Type IDs

Some models' purpose is to do sequence classification or question answering. These require two different sequences to be encoded in the same input IDs. They are usually separated by special tokens, such as the classifier and separator tokens. For example, the BERT model builds its two sequence input as such:

```
>>> # [CLS] SEQUENCE_A [SEP] SEQUENCE_B [SEP]
```

We can use our tokenizer to automatically generate such a sentence by passing the two sequences as two arguments (and not a list like before) like this:

```
>>> from transformers import BertTokenizer
>>> tokenizer = BertTokenizer.from_pretrained("bert-base-cased")
>>> sequence_a = "HuggingFace is based in NYC"
>>> sequence_b = "Where is HuggingFace based?"

>>> encoded_dict = tokenizer(sequence_a, sequence_b)
>>> decoded = tokenizer.decode(encoded_dict["input_ids"])
```

which will return:

```
>>> print(decoded)
[CLS] HuggingFace is based in NYC [SEP] Where is HuggingFace based? [SEP]
```

This is enough for some models to understand where one sequence ends and where another begins. However, other models such as BERT have an additional mechanism, which are the token type IDs (also called segment IDs). They are a binary mask identifying the different sequences in the model.

The tokenizer returns in the dictionary under the key "token_type_ids":

```
>>> encoded_dict['token_type_ids']
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
```

The first sequence, the “context” used for the question, has all its tokens represented by `0`, whereas the question has all its tokens represented by `1`. Some models, like `XLNetModel` use an additional token represented by a `2`.

Position IDs

The position IDs are used by the model to identify which token is at which position. Contrary to RNNs that have the position of each token embedded within them, transformers are unaware of the position of each token. The position IDs are created for this purpose.

They are an optional parameter. If no position IDs are passed to the model, they are automatically created as absolute positional embeddings.

Absolute positional embeddings are selected in the range `[0, config.max_position_embeddings - 1]`. Some models use other types of positional embeddings, such as sinusoidal position embeddings or relative position embeddings.

Feed Forward Chunking

In transformers two feed forward layers usually follows the self attention layer in each residual attention block. The intermediate embedding size of the feed forward layers is often bigger than the hidden size of the model (e.g., for `bert-base-uncased`).

For an input of size `[batch_size, sequence_length]`, the memory required to store the intermediate feed forward embeddings `[batch_size, sequence_length, config.intermediate_size]` can account for a large fraction of the memory use. The authors of Reformer: The Efficient Transformer noticed that since the computation is independent of the `sequence_length` dimension, it is mathematically equivalent to compute the output embeddings of both feed forward layers `[batch_size, config.hidden_size]_0, ..., [batch_size, config.hidden_size]_n` individually and concat them afterward to `[batch_size, sequence_length, config.hidden_size]` with `n = sequence_length`, which trades increased computation time against reduced memory use, but yields a mathematically **equivalent** result.

For models employing the function `apply_chunking_to_forward()`, the `chunk_size` defines the number of output embeddings that are computed in parallel and thus defines the trade-off between memory and time complexity. If `chunk_size` is set to 0, no feed forward chunking is done.

Summary of the tasks

This page shows the most frequent use-cases when using the library. The models available allow for many different configurations and a great versatility in use-cases. The most simple ones are presented here, showcasing usage for tasks such as question answering, sequence classification, named entity recognition and others.

These examples leverage auto-models, which are classes that will instantiate a model according to a given checkpoint, automatically selecting the correct model architecture. Please check the `AutoModel` documentation for more information. Feel free to modify the code to be more specific and adapt it to your specific use-case.

In order for a model to perform well on a task, it must be loaded from a checkpoint corresponding to that task. These checkpoints are usually pre-trained on a large corpus of data and fine-tuned on a specific task. This means the following:

- Not all models were fine-tuned on all tasks. If you want to fine-tune a model on a specific task, you can leverage one of the *run_$TASK.py* script in the examples directory.
- Fine-tuned models were fine-tuned on a specific dataset. This dataset may or may not overlap with your use-case and domain. As mentioned previously, you may leverage the examples scripts to fine-tune your model, or you may create your own training script.

In order to do an inference on a task, several mechanisms are made available by the library:

- Pipelines: very easy-to-use abstractions, which require as little as two lines of code.
- Using a model directly with a tokenizer (PyTorch/TensorFlow): the full inference using the model. Less abstraction, but much more powerful.

Both approaches are showcased here.

> **Note**
>
> All tasks presented here leverage pre-trained checkpoints that were fine-tuned on specific tasks. Loading a checkpoint that was not fine-tuned on a specific task would load only the base transformer layers and not the additional head that is used for the task, initializing the weights of that head randomly.
>
> This would produce random output.

Sequence Classification

Sequence classification is the task of classifying sequences according to a given number of classes. An example of sequence classification is the GLUE dataset, which is entirely based on that task. If you would like to fine-tune a model on a GLUE sequence classification task, you may leverage the run_glue.py or run_tf_glue.py scripts.

Here is an example using the pipelines do to sentiment analysis: identifying if a sequence is positive or negative. It leverages a fine-tuned model on sst2, which is a GLUE task.

This returns a label ("POSITIVE" or "NEGATIVE") alongside a score, as follows:

```
>>> from transformers import pipeline

>>> nlp = pipeline("sentiment-analysis")

>>> result = nlp("I hate you")[0]
>>> print(f"label: {result['label']}, with score: {round(result['score'],
4)}")
label: NEGATIVE, with score: 0.9991

>>> result = nlp("I love you")[0]
>>> print(f"label: {result['label']}, with score: {round(result['score'],
4)}")
label: POSITIVE, with score: 0.9999
```

Here is an example of doing a sequence classification using a model to determine if two sequences are paraphrases of each other. The process is the following:

- Instantiate a tokenizer and a model from the checkpoint name. The model is identified as a BERT model and loads it with the weights stored in the checkpoint.
- Build a sequence from the two sentences, with the correct model-specific separators token type ids and attention masks (`encode()` and `__call__()` take care of this)
- Pass this sequence through the model so that it is classified in one of the two available classes: 0 (not a paraphrase) and 1 (is a paraphrase)
- Compute the softmax of the result to get probabilities over the classes
- Print the results

PyTorch | TensorFlow

Next page

```
>>> from transformers import AutoTokenizer, AutoModelForSequenceClassification
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased-finetuned-
mrpc")
>>> model = AutoModelForSequenceClassification.from_pretrained("bert-base-
cased-finetuned-mrpc")

>>> classes = ["not paraphrase", "is paraphrase"]

>>> sequence_0 = "The company HuggingFace is based in New York City"
>>> sequence_1 = "Apples are especially bad for your health"
>>> sequence_2 = "HuggingFace's headquarters are situated in Manhattan"

>>> paraphrase = tokenizer(sequence_0, sequence_2, return_tensors="pt")
>>> not_paraphrase = tokenizer(sequence_0, sequence_1, return_tensors="pt")

>>> paraphrase_classification_logits = model(**paraphrase)[0]
>>> not_paraphrase_classification_logits = model(**not_paraphrase)[0]

>>> paraphrase_results = torch.softmax(paraphrase_classification_logits,
dim=1).tolist()[0]
>>> not_paraphrase_results =
torch.softmax(not_paraphrase_classification_logits, dim=1).tolist()[0]

>>> # Should be paraphrase
>>> for i in range(len(classes)):
...     print(f"{classes[i]}: {int(round(paraphrase_results[i] * 100))}%")
not paraphrase: 10%
is paraphrase: 90%

>>> # Should not be paraphrase
>>> for i in range(len(classes)):
...     print(f"{classes[i]}: {int(round(not_paraphrase_results[i] * 100))}%")
not paraphrase: 94%
is paraphrase: 6%
```

Extractive Question Answering

Extractive Question Answering is the task of extracting an answer from a text given a question. An example of a question answering dataset is the SQuAD dataset, which is entirely based on that task. If you would like to fine-tune a model on a SQuAD task, you may leverage the *run_squad.py*.

Here is an example using the pipelines do to question answering: extracting an answer from a text given a question. It leverages a fine-tuned model on SQuAD.

```
>>> from transformers import pipeline

>>> nlp = pipeline("question-answering")

>>> context = r"""
... Extractive Question Answering is the task of extracting an answer from a
text given a question. An example of a
... question answering dataset is the SQuAD dataset, which is entirely based
on that task. If you would like to fine-tune
... a model on a SQuAD task, you may leverage the examples/question-
answering/run_squad.py script.
... """
```

This returns an answer extracted from the text, a confidence score, alongside “start” and “end” values which are the positions of the extracted answer in the text.

```
>>> result = nlp(question="What is extractive question answering?",
context=context)
>>> print(f"Answer: '{result['answer']}', score: {round(result['score'], 4)},
start: {result['start']}, end: {result['end']}")
Answer: 'the task of extracting an answer from a text given a question.',
score: 0.6226, start: 34, end: 96

>>> result = nlp(question="What is a good example of a question answering
dataset?", context=context)
>>> print(f"Answer: '{result['answer']}', score: {round(result['score'], 4)},
start: {result['start']}, end: {result['end']}")
Answer: 'SQuAD dataset,', score: 0.5053, start: 147, end: 161
```

Here is an example of question answering using a model and a tokenizer. The process is the following:

- Instantiate a tokenizer and a model from the checkpoint name. The model is identified as a BERT model and loads it with the weights stored in the checkpoint.
- Define a text and a few questions.
- Iterate over the questions and build a sequence from the text and the current question, with the correct model-specific separators token type ids and attention masks
- Pass this sequence through the model. This outputs a range of scores across the entire sequence tokens (question and text), for both the start and end positions.
- Compute the softmax of the result to get probabilities over the tokens
- Fetch the tokens from the identified start and stop values, convert those tokens to a string.
- Print the results

PyTorch | TensorFlow

```
>>> from transformers import AutoTokenizer, AutoModelForQuestionAnswering
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("bert-large-uncased-whole-word-masking-finetuned-squad")
>>> model = AutoModelForQuestionAnswering.from_pretrained("bert-large-uncased-whole-word-masking-finetuned-squad")

>>> text = r"""
... 🤗 Transformers (formerly known as pytorch-transformers and pytorch-pretrained-bert) provides general-purpose
... architectures (BERT, GPT-2, RoBERTa, XLM, DistilBert, XLNet…) for Natural Language Understanding (NLU) and Natural
... Language Generation (NLG) with over 32+ pretrained models in 100+ languages and deep interoperability between
... TensorFlow 2.0 and PyTorch.
... """

>>> questions = [
...     "How many pretrained models are available in 🤗 Transformers?",
...     "What does 🤗 Transformers provide?",
...     "🤗 Transformers provides interoperability between which frameworks?",
... ]

>>> for question in questions:
...     inputs = tokenizer(question, text, add_special_tokens=True, return_tensors="pt")
...     input_ids = inputs["input_ids"].tolist()[0]
...
...     text_tokens = tokenizer.convert_ids_to_tokens(input_ids)
...     answer_start_scores, answer_end_scores = model(**inputs)
...
...     answer_start = torch.argmax(
...         answer_start_scores
...     )  # Get the most likely beginning of answer with the argmax of the score
...     answer_end = torch.argmax(answer_end_scores) + 1  # Get the most likely end of answer with the argmax of the score
...
...     answer = tokenizer.convert_tokens_to_string(tokenizer.convert_ids_to_tokens(input_ids[ansv
...
...     print(f"Question: {question}")
...     print(f"Answer: {answer}")
Question: How many pretrained models are available in 🤗 Transformers?
Answer: over 32 +
Question: What does 🤗 Transformers provide?
Answer: general - purpose architectures
Question: 🤗 Transformers provides interoperability between which frameworks?
Answer: tensorflow 2 . 0 and pytorch
```

Language Modeling

Language modeling is the task of fitting a model to a corpus, which can be domain specific. All popular transformer based models are trained using a variant of language modeling, e.g. BERT with masked language modeling, GPT-2 with causal language modeling.

Language modeling can be useful outside of pre-training as well, for example to shift the model distribution to be domain-specific: using a language model trained over a very large corpus, and then fine-tuning it to a news dataset or on scientific papers e.g. LysandreJik/arxiv-nlp.

Masked Language Modeling

Masked language modeling is the task of masking tokens in a sequence with a masking token, and prompting the model to fill that mask with an appropriate token. This allows the model to attend to both the right context (tokens on the right of the mask) and the left context (tokens on the left of the mask). Such a training creates a strong basis for downstream tasks requiring bi-directional context such as SQuAD (question answering, see Lewis, Lui, Goyal et al., part 4.2).

Here is an example of using pipelines to replace a mask from a sequence:

```
>>> from transformers import pipeline

>>> nlp = pipeline("fill-mask")
```

This outputs the sequences with the mask filled, the confidence score as well as the token id in the tokenizer vocabulary:

```
>>> from pprint import pprint
>>> pprint(nlp(f"HuggingFace is creating a {nlp.tokenizer.mask_token} that the
community uses to solve NLP tasks."))
[{'score': 0.1792745739221573,
  'sequence': '<s>HuggingFace is creating a tool that the community uses to '
              'solve NLP tasks.</s>',
  'token': 3944,
  'token_str': 'Ġtool'},
 {'score': 0.11349421739578247,
  'sequence': '<s>HuggingFace is creating a framework that the community uses
'
              'to solve NLP tasks.</s>',
  'token': 7208,
  'token_str': 'Ġframework'},
 {'score': 0.05243554711341858,
  'sequence': '<s>HuggingFace is creating a library that the community uses to
'
              'solve NLP tasks.</s>',
  'token': 5560,
  'token_str': 'Ġlibrary'},
 {'score': 0.03493533283472061,
  'sequence': '<s>HuggingFace is creating a database that the community uses '
              'to solve NLP tasks.</s>',
  'token': 8503,
  'token_str': 'Ġdatabase'},
 {'score': 0.02860250137746334,
  'sequence': '<s>HuggingFace is creating a prototype that the community uses
'
              'to solve NLP tasks.</s>',
  'token': 17715,
  'token_str': 'Ġprototype'}]
```

Here is an example doing masked language modeling using a model and a tokenizer. The process is the following:

- Instantiate a tokenizer and a model from the checkpoint name. The model is identified as a DistilBERT model and loads it with the weights stored in the checkpoint.
- Define a sequence with a masked token, placing the `tokenizer.mask_token` instead of a word.
- Encode that sequence into IDs and find the position of the masked token in that list of IDs.
- Retrieve the predictions at the index of the mask token: this tensor has the same size as the vocabulary, and the values are the scores attributed to each token. The model gives higher score to tokens he deems probable in that context.
- Retrieve the top 5 tokens using the PyTorch `topk` or TensorFlow `top_k` methods.
- Replace the mask token by the tokens and print the results

PyTorch | TensorFlow

```
>>> from transformers import AutoModelWithLMHead, AutoTokenizer
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("distilbert-base-cased")
>>> model = AutoModelWithLMHead.from_pretrained("distilbert-base-cased")

>>> sequence = f"Distilled models are smaller than the models they mimic.
Using them instead of the large versions would help {tokenizer.mask_token} our
carbon footprint."

>>> input = tokenizer.encode(sequence, return_tensors="pt")
>>> mask_token_index = torch.where(input == tokenizer.mask_token_id)[1]

>>> token_logits = model(input)[0]
>>> mask_token_logits = token_logits[0, mask_token_index, :]

>>> top_5_tokens = torch.topk(mask_token_logits, 5, dim=1).indices[0].tolist()
```

This prints five sequences, with the top 5 tokens predicted by the model:

```
>>> for token in top_5_tokens:
...     print(sequence.replace(tokenizer.mask_token,
tokenizer.decode([token])))
Distilled models are smaller than the models they mimic. Using them instead of
the large versions would help reduce our carbon footprint.
Distilled models are smaller than the models they mimic. Using them instead of
the large versions would help increase our carbon footprint.
Distilled models are smaller than the models they mimic. Using them instead of
the large versions would help decrease our carbon footprint.
Distilled models are smaller than the models they mimic. Using them instead of
the large versions would help offset our carbon footprint.
Distilled models are smaller than the models they mimic. Using them instead of
the large versions would help improve our carbon footprint.
```

Causal Language Modeling

Causal language modeling is the task of predicting the token following a sequence of tokens. In this situation, the model only attends to the left context (tokens on the left of the mask). Such a training is particularly interesting for generation tasks.

Usually, the next token is predicted by sampling from the logits of the last hidden state the model produces from the input sequence.

Here is an example using the tokenizer and model and leveraging the `top_k_top_p_filtering()` method to sample the next token following an input sequence of tokens.

PyTorch | TensorFlow

```
>>> from transformers import AutoModelWithLMHead, AutoTokenizer,
top_k_top_p_filtering
>>> import torch
>>> from torch.nn import functional as F

>>> tokenizer = AutoTokenizer.from_pretrained("gpt2")
>>> model = AutoModelWithLMHead.from_pretrained("gpt2")

>>> sequence = f"Hugging Face is based in DUMBO, New York City, and "

>>> input_ids = tokenizer.encode(sequence, return_tensors="pt")

>>> # get logits of last hidden state
>>> next_token_logits = model(input_ids)[0][:, -1, :]

>>> # filter
>>> filtered_next_token_logits = top_k_top_p_filtering(next_token_logits,
top_k=50, top_p=1.0)

>>> # sample
>>> probs = F.softmax(filtered_next_token_logits, dim=-1)
>>> next_token = torch.multinomial(probs, num_samples=1)

>>> generated = torch.cat([input_ids, next_token], dim=-1)

>>> resulting_string = tokenizer.decode(generated.tolist()[0])
```

This outputs a (hopefully) coherent next token following the original sequence, which is in our case is the word *has*:

```
print(resulting_string)
Hugging Face is based in DUMBO, New York City, and has
```

In the next section, we show how this functionality is leveraged in `generate()` to generate multiple tokens up to a user-defined length.

Text Generation

In text generation (*a.k.a open-ended text generation*) the goal is to create a coherent portion of text that is a continuation from the given context. As an example, is it shown how *GPT-2* can be used in pipelines to generate text. As a default all models apply *Top-K* sampling when used in pipelines as configured in their respective configurations (see gpt-2 config for example).

```
>>> from transformers import pipeline

>>> text_generator = pipeline("text-generation")
>>> print(text_generator("As far as I am concerned, I will", max_length=50,
do_sample=False))
[{'generated_text': 'As far as I am concerned, I will be the first to admit
that I am not a fan of the idea of a "free market." I think that the idea of a
free market is a bit of a stretch. I think that the idea'}]
```

Here the model generates a random text with a total maximal length of *50* tokens from context *"As far as I am concerned, I will"*. The default arguments of `PreTrainedModel.generate()` can directly be overriden in the pipeline as is shown above for the argument `max_length`.

Here is an example for text generation using XLNet and its tokenzier.

PyTorch | TensorFlow

Next page

```
>>> from transformers import AutoModelWithLMHead, AutoTokenizer

>>> model = AutoModelWithLMHead.from_pretrained("xlnet-base-cased")
>>> tokenizer = AutoTokenizer.from_pretrained("xlnet-base-cased")

>>> # Padding text helps XLNet with short prompts - proposed by Aman Rusia in
https://github.com/rusiaaman/XLNet-gen#methodology
>>> PADDING_TEXT = """In 1991, the remains of Russian Tsar Nicholas II and his
family
... (except for Alexei and Maria) are discovered.
... The voice of Nicholas's young son, Tsarevich Alexei Nikolaevich, narrates
the
... remainder of the story. 1883 Western Siberia,
... a young Grigori Rasputin is asked by his father and a group of men to
perform magic.
... Rasputin has a vision and denounces one of the men as a horse thief.
Although his
... father initially slaps him for making such an accusation, Rasputin watches
as the
... man is chased outside and beaten. Twenty years later, Rasputin sees a
vision of
... the Virgin Mary, prompting him to become a priest. Rasputin quickly
becomes famous,
... with people, even a bishop, begging for his blessing. <eod> </s> <eos>"""

>>> prompt = "Today the weather is really nice and I am planning on "
>>> inputs = tokenizer.encode(PADDING_TEXT + prompt, add_special_tokens=False,
return_tensors="pt")

>>> prompt_length = len(tokenizer.decode(inputs[0], skip_special_tokens=True,
clean_up_tokenization_spaces=True))
>>> outputs = model.generate(inputs, max_length=250, do_sample=True,
top_p=0.95, top_k=60)
>>> generated = prompt + tokenizer.decode(outputs[0])[prompt_length:]
```

```
print(generated)
```

Text generation is currently possible with *GPT-2*, *OpenAi-GPT*, *CTRL*, *XLNet*, *Transfo-XL* and *Reformer* in PyTorch and for most models in Tensorflow as well. As can be seen in the example above *XLNet* and *Transfo-xl* often need to be padded to work well. GPT-2 is usually a good choice for *open-ended text generation* because it was trained on millions on webpages with a causal language modeling objective.

For more information on how to apply different decoding strategies for text generation, please also refer to our generation blog post here.

Named Entity Recognition

Named Entity Recognition (NER) is the task of classifying tokens according to a class, for example identifying a token as a person, an organisation or a location. An example of a named entity recognition dataset is the CoNLL-2003 dataset, which is entirely based on that task. If you would like to fine-tune a model on an NER task, you may leverage the *ner/run_ner.py* (PyTorch), *ner/run_pl_ner.py* (leveraging pytorch-lightning) or the *ner/run_tf_ner.py* (TensorFlow) scripts.

Here is an example using the pipelines do to named entity recognition, trying to identify tokens as belonging to one of 9 classes:

- O, Outside of a named entity
- B-MIS, Beginning of a miscellaneous entity right after another miscellaneous entity
- I-MIS, Miscellaneous entity
- B-PER, Beginning of a person's name right after another person's name
- I-PER, Person's name
- B-ORG, Beginning of an organisation right after another organisation
- I-ORG, Organisation
- B-LOC, Beginning of a location right after another location
- I-LOC, Location

It leverages a fine-tuned model on CoNLL-2003, fine-tuned by @stefan-it from dbmdz.

```
>>> from transformers import pipeline

>>> nlp = pipeline("ner")

>>> sequence = "Hugging Face Inc. is a company based in New York City. Its
headquarters are in DUMBO, therefore very"
...            "close to the Manhattan Bridge which is visible from the
window."
```

This outputs a list of all words that have been identified as an entity from the 9 classes defined above. Here is the expected results:

```
print(nlp(sequence))
```

```
[
    {'word': 'Hu', 'score': 0.9995632767677307, 'entity': 'I-ORG'},
    {'word': '##gging', 'score': 0.9915938973426819, 'entity': 'I-ORG'},
    {'word': 'Face', 'score': 0.9982671737670898, 'entity': 'I-ORG'},
    {'word': 'Inc', 'score': 0.9994403719902039, 'entity': 'I-ORG'},
    {'word': 'New', 'score': 0.9994346499443054, 'entity': 'I-LOC'},
    {'word': 'York', 'score': 0.9993270635604858, 'entity': 'I-LOC'},
    {'word': 'City', 'score': 0.9993864893913269, 'entity': 'I-LOC'},
    {'word': 'D', 'score': 0.9825621843338013, 'entity': 'I-LOC'},
    {'word': '##UM', 'score': 0.936983048915863, 'entity': 'I-LOC'},
    {'word': '##BO', 'score': 0.8987102508544922, 'entity': 'I-LOC'},
    {'word': 'Manhattan', 'score': 0.9758241176605225, 'entity': 'I-LOC'},
    {'word': 'Bridge', 'score': 0.990249514579773, 'entity': 'I-LOC'}
]
```

Note how the words “Hugging Face” have been identified as an organisation, and “New York City”, “DUMBO” and “Manhattan Bridge” have been identified as locations.

Here is an example doing named entity recognition using a model and a tokenizer. The process is the following:

- Instantiate a tokenizer and a model from the checkpoint name. The model is identified as a BERT model and loads it with the weights stored in the checkpoint.
- Define the label list with which the model was trained on.
- Define a sequence with known entities, such as “Hugging Face” as an organisation and “New York City” as a location.
- Split words into tokens so that they can be mapped to the predictions. We use a small hack by firstly completely encoding and decoding the sequence, so that we’re left with a string that contains the special tokens.
- Encode that sequence into IDs (special tokens are added automatically).
- Retrieve the predictions by passing the input to the model and getting the first output. This results in a distribution over the 9 possible classes for each token. We take the argmax to retrieve the most likely class for each token.
- Zip together each token with its prediction and print it.

PyTorch | TensorFlow

```
>>> from transformers import AutoModelForTokenClassification, AutoTokenizer
>>> import torch

>>> model = AutoModelForTokenClassification.from_pretrained("dbmdz/bert-large-cased-finetuned-conll03-english")
>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

>>> label_list = [
...     "O",       # Outside of a named entity
...     "B-MISC",  # Beginning of a miscellaneous entity right after another miscellaneous entity
...     "I-MISC",  # Miscellaneous entity
...     "B-PER",   # Beginning of a person's name right after another person's name
...     "I-PER",   # Person's name
...     "B-ORG",   # Beginning of an organisation right after another organisation
...     "I-ORG",   # Organisation
...     "B-LOC",   # Beginning of a location right after another location
...     "I-LOC"    # Location
... ]

>>> sequence = "Hugging Face Inc. is a company based in New York City. Its headquarters are in DUMBO, therefore very" \
...            "close to the Manhattan Bridge."

>>> # Bit of a hack to get the tokens with the special tokens
>>> tokens = tokenizer.tokenize(tokenizer.decode(tokenizer.encode(sequence)))
>>> inputs = tokenizer.encode(sequence, return_tensors="pt")

>>> outputs = model(inputs)[0]
>>> predictions = torch.argmax(outputs, dim=2)
```

This outputs a list of each token mapped to their prediction. Differently from the pipeline, here every token has a prediction as we didn't remove the "0" class which means that no particular entity was found on that token. The following array should be the output:

```
>>> print([(token, label_list[prediction]) for token, prediction in zip(tokens, predictions[0].numpy())])
[('[CLS]', 'O'), ('Hu', 'I-ORG'), ('##gging', 'I-ORG'), ('Face', 'I-ORG'), ('Inc', 'I-ORG'), ('.', 'O'), ('is', 'O'), ('a', 'O'), ('company', 'O'), ('based', 'O'), ('in', 'O'), ('New', 'I-LOC'), ('York', 'I-LOC'), ('City', 'I-LOC'), ('.', 'O'), ('Its', 'O'), ('headquarters', 'O'), ('are', 'O'), ('in', 'O'), ('D', 'I-LOC'), ('##UM', 'I-LOC'), ('##BO', 'I-LOC'), (',', 'O'), ('therefore', 'O'), ('very', 'O'), ('##c', 'O'), ('##lose', 'O'), ('to', 'O'), ('the', 'O'), ('Manhattan', 'I-LOC'), ('Bridge', 'I-LOC'), ('.', 'O'), ('[SEP]', 'O')]
```

Summarization

Summarization is the task of summarizing a text / an article into a shorter text.

An example of a summarization dataset is the CNN / Daily Mail dataset, which consists of long news articles and was created for the task of summarization. If you would like to fine-tune a model on a summarization task, you may leverage the `examples/summarization/bart/run_train.sh` (leveraging pytorch-lightning) script.

Here is an example using the pipelines do to summarization. It leverages a Bart model that was fine-tuned on the CNN / Daily Mail data set.

```
>>> from transformers import pipeline

>>> summarizer = pipeline("summarization")

>>> ARTICLE = """ New York (CNN)When Liana Barrientos was 23 years old, she
got married in Westchester County, New York.
... A year later, she got married again in Westchester County, but to a
different man and without divorcing her first husband.
... Only 18 days after that marriage, she got hitched yet again. Then,
Barrientos declared "I do" five more times, sometimes only within two weeks of
each other.
... In 2010, she married once more, this time in the Bronx. In an application
for a marriage license, she stated it was her "first and only" marriage.
... Barrientos, now 39, is facing two criminal counts of "offering a false
instrument for filing in the first degree," referring to her false statements
on the
... 2010 marriage license application, according to court documents.
... Prosecutors said the marriages were part of an immigration scam.
... On Friday, she pleaded not guilty at State Supreme Court in the Bronx,
according to her attorney, Christopher Wright, who declined to comment
further.
... After leaving court, Barrientos was arrested and charged with theft of
service and criminal trespass for allegedly sneaking into the New York subway
through an emergency exit, said Detective
... Annette Markowski, a police spokeswoman. In total, Barrientos has been
married 10 times, with nine of her marriages occurring between 1999 and 2002.
... All occurred either in Westchester County, Long Island, New Jersey or the
Bronx. She is believed to still be married to four men, and at one time, she
was married to eight men at once, prosecutors say.
... Prosecutors said the immigration scam involved some of her husbands, who
filed for permanent residence status shortly after the marriages.
... Any divorces happened only after such filings were approved. It was
unclear whether any of the men will be prosecuted.
... The case was referred to the Bronx District Attorney\'s Office by
Immigration and Customs Enforcement and the Department of Homeland Security\'s
... Investigation Division. Seven of the men are from so-called "red-flagged"
countries, including Egypt, Turkey, Georgia, Pakistan and Mali.
... Her eighth husband, Rashid Rajput, was deported in 2006 to his native
Pakistan after an investigation by the Joint Terrorism Task Force.
... If convicted, Barrientos faces up to four years in prison.  Her next court
appearance is scheduled for May 18.
... """
```

Because the summarization pipeline depends on the `PretrainedModel.generate()` method, we can override the default arguments of `PretrainedModel.generate()` directly in the pipeline as is shown for `max_length` and `min_length` above. This outputs the following summary:

```
>>> print(summarizer(ARTICLE, max_length=130, min_length=30, do_sample=False))
[{'summary_text': 'Liana Barrientos, 39, is charged with two counts of
"offering a false instrument for filing in the first degree" In total, she has
been married 10 times, with nine of her marriages occurring between 1999 and
2002. She is believed to still be married to four men.'}]
```

Here is an example doing summarization using a model and a tokenizer. The process is the following:

- Instantiate a tokenizer and a model from the checkpoint name. Summarization is usually done using an encoder-decoder model, such as `Bart` or `T5`.
- Define the article that should be summarizaed.
- Leverage the `PretrainedModel.generate()` method.
- Add the T5 specific prefix “summarize: “.

Here Google`s T5 model is used that was only pre-trained on a multi-task mixed data set (including CNN / Daily Mail), but nevertheless yields very good results. .. code-block:

PyTorch | TensorFlow

```
>>> from transformers import AutoModelWithLMHead, AutoTokenizer

>>> model = AutoModelWithLMHead.from_pretrained("t5-base")
>>> tokenizer = AutoTokenizer.from_pretrained("t5-base")

>>> # T5 uses a max_length of 512 so we cut the article to 512 tokens.
>>> inputs = tokenizer.encode("summarize: " + ARTICLE, return_tensors="pt",
max_length=512)
>>> outputs = model.generate(inputs, max_length=150, min_length=40,
length_penalty=2.0, num_beams=4, early_stopping=True)
```

Translation

Translation is the task of translating a text from one language to another.

An example of a translation dataset is the WMT English to German dataset, which has English sentences as the input data and German sentences as the target data.

Here is an example using the pipelines do to translation. It leverages a T5 model that was only pre-trained on a multi-task mixture dataset (including WMT), but yields impressive translation results nevertheless.

```
>>> from transformers import pipeline

>>> translator = pipeline("translation_en_to_de")
>>> print(translator("Hugging Face is a technology company based in New York
and Paris", max_length=40))
[{'translation_text': 'Hugging Face ist ein Technologieunternehmen mit Sitz in
New York und Paris.'}]
```

Because the translation pipeline depends on the `PretrainedModel.generate()` method, we can override the default arguments of `PretrainedModel.generate()` directly in the pipeline as is shown for `max_length` above. This outputs the following translation into German:

```
Hugging Face ist ein Technologieunternehmen mit Sitz in New York und Paris.
```

Here is an example doing translation using a model and a tokenizer. The process is the following:

- Instantiate a tokenizer and a model from the checkpoint name. Summarization is usually done using an encoder-decoder model, such as `Bart` or `T5`.
- Define the article that should be summarizaed.
- Leverage the `PretrainedModel.generate()` method.
- Add the T5 specific prefix “translate English to German: “

PyTorch | TensorFlow

```
>>> from transformers import AutoModelWithLMHead, AutoTokenizer

>>> model = AutoModelWithLMHead.from_pretrained("t5-base")
>>> tokenizer = AutoTokenizer.from_pretrained("t5-base")

>>> inputs = tokenizer.encode("translate English to German: Hugging Face is a
technology company based in New York and Paris", return_tensors="pt")
>>> outputs = model.generate(inputs, max_length=40, num_beams=4,
early_stopping=True)

>>> print(outputs)
tensor([[    0, 11560,  3896,  8881,   229,   236,     3, 14366, 15377,   181,
         11216,    16,   368,  1060,    64,  1919,     5]])
```

Preprocessing data

In this tutorial, we'll explore how to preprocess your data using 🤗 Transformers. The main tool for this is what we

call a tokenizer. You can build one using the tokenizer class associated to the model you would like to use, or directly with the `AutoTokenizer` class.

As we saw in the quicktour, the tokenizer will first split a given text in words (or part of words, punctuation symbols, etc.) usually called *tokens*. Then it will convert those *tokens* into numbers, to be able to build a tensor out of them and feed them to the model. It will also add any additional inputs the model might expect to work properly.

> **Note**
>
> If you plan on using a pretrained model, it's important to use the associated pretrained tokenizer: it will split the text you give it in tokens the same way for the pretraining corpus, and it will use the same correspondence token to index (that we usually call a *vocab*) as during pretraining.

To automatically download the vocab used during pretraining or fine-tuning a given model, you can use the `from_pretrained()` method:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
```

Base use

A `PreTrainedTokenizer` has many methods, but the only one you need to remember for preprocessing is its `__call__`: you just need to feed your sentence to your tokenizer object.

```
encoded_input = tokenizer("Hello, I'm a single sentence!")
print(encoded_input)
```

This will return a dictionary string to list of ints like this one:

```
{'input_ids': [101, 138, 18696, 155, 1942, 3190, 1144, 1572, 13745, 1104, 159,
9664, 2107, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

The input_ids are the indices corresponding to each token in our sentence. We will see below what the attention_mask is used for and in the next section the goal of token_type_ids.

The tokenizer can decode a list of token ids in a proper sentence:

```
tokenizer.decode(encoded_input["input_ids"])
```

which should return

```
"[CLS] Hello, I'm a single sentence! [SEP]"
```

As you can see, the tokenizer automatically added some special tokens that the model expect. Not all model need special tokens; for instance, if we had used ` gtp2-medium ` instead of *bert-base-cased* to create our tokenizer, we would have

seen the same sentence as the original one here. You can disable this behavior (which is only advised if you have added those special tokens yourself) by passing `add_special_tokens=False`.

If you have several sentences you want to process, you can do this efficiently by sending them as a list to the tokenizer:

```
batch_sentences = ["Hello I'm a single sentence",
                   "And another sentence",
                   "And the very very last one"]
encoded_inputs = tokenizer(batch_sentences)
print(encoded_inputs)
```

We get back a dictionary once again, this time with values being list of list of ints:

```
{'input_ids': [[101, 8667, 146, 112, 182, 170, 1423, 5650, 102],
               [101, 1262, 1330, 5650, 102],
               [101, 1262, 1103, 1304, 1304, 1314, 1141, 102]],
 'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0]],
 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1],
                    [1, 1, 1, 1, 1],
                    [1, 1, 1, 1, 1, 1, 1, 1]]}
```

If the purpose of sending several sentences at a time to the tokenizer is to build a batch to feed the model, you will probably want:

- To pad each sentence to the maximum length there is in your batch.
- To truncate each sentence to the maximum length the model can accept (if applicable).
- To return tensors.

You can do all of this by using the following options when feeding your list of sentences to the tokenizer:

PyTorch | TensorFlow

```
batch = tokenizer(batch_sentences, padding=True, truncation=True,
return_tensors="pt")
print(batch)
```

which should now return a dictionary string to tensor like this:

```
{'input_ids': tensor([[ 101, 8667,  146,  112,  182,  170, 1423, 5650,  102],
                      [ 101, 1262, 1330, 5650,  102,    0,    0,    0,    0],
                      [ 101, 1262, 1103, 1304, 1304, 1314, 1141,  102,
0]]),
 'token_type_ids': tensor([[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]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1],
                           [1, 1, 1, 1, 1, 0, 0, 0, 0],
                           [1, 1, 1, 1, 1, 1, 1, 1, 0]])}
```

We can now see what the attention_mask is all about: it points out which tokens the model should pay attention to and which ones it should not (because they represent padding in this case).

Note that if your model does not have a maximum length associated to it, the command above will throw a warning. You can safely ignore it. You can also pass `verbose=False` to stop the tokenizer to throw those kinds of warnings.

Preprocessing pairs of sentences

Sometimes you need to feed pair of sentences to your model. For instance, if you want to classify if two sentences in a pair are similar, or for question-answering models, which take a context and a question. For BERT models, the input is then represented like this:

```
[CLS] Sequence A [SEP] Sequence B [SEP]
```

You can encode a pair of sentences in the format expected by your model by supplying the two sentences as two arguments

(not a list since a list of two sentences will be interpreted as a batch of two single sentences, as we saw before).

```
encoded_input = tokenizer("How old are you?", "I'm 6 years old")
print(encoded_input)
```

This will once again return a dict string to list of ints:

```
{'input_ids': [101, 1731, 1385, 1132, 1128, 136, 102, 146, 112, 182, 127,
1201, 1385, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

This shows us what the token_type_ids are for: they indicate to the model which part of the inputs correspond to the first sentence and which part corresponds to the second sentence. Note that *token_type_ids* are not required or handled by all models. By default, a tokenizer will only return the inputs that its associated model expects. You can force the return (or the non-return) of any of those special arguments by using `return_input_ids` or `return_token_type_ids`.

If we decode the token ids we obtained, we will see that the special tokens have been properly added.

```
tokenizer.decode(encoded_input["input_ids"])
```

will return:

```
"[CLS] How old are you? [SEP] I'm 6 years old [SEP]"
```

If you have a list of pairs of sequences you want to process, you should feed them as two lists to your tokenizer: the list of first sentences and the list of second sentences:

```
batch_sentences = ["Hello I'm a single sentence",
                   "And another sentence",
                   "And the very very last one"]
batch_of_second_sentences = ["I'm a sentence that goes with the first
sentence",
                             "And I should be encoded with the second
sentence",
                             "And I go with the very last one"]
encoded_inputs = tokenizer(batch_sentences, batch_of_second_sentences)
print(encoded_inputs)
```

will return a dict with the values being list of lists of ints:

```
{'input_ids': [[101, 8667, 146, 112, 182, 170, 1423, 5650, 102, 146, 112, 182,
170, 5650, 1115, 2947, 1114, 1103, 1148, 5650, 102],
               [101, 1262, 1330, 5650, 102, 1262, 146, 1431, 1129, 12544,
1114, 1103, 1248, 5650, 102],
               [101, 1262, 1103, 1304, 1304, 1314, 1141, 102, 1262, 146, 1301,
1114, 1103, 1304, 1314, 1141, 102]],
'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1],
                   [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                   [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]],
'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1],
                   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]}
```

To double-check what is fed to the model, we can decode each list in *input_ids* one by one:

```
for ids in encoded_inputs["input_ids"]:
    print(tokenizer.decode(ids))
```

which will return:

```
[CLS] Hello I'm a single sentence [SEP] I'm a sentence that goes with the
first sentence [SEP]
[CLS] And another sentence [SEP] And I should be encoded with the second
sentence [SEP]
[CLS] And the very very last one [SEP] And I go with the very last one [SEP]
```

Once again, you can automatically pad your inputs to the maximum sentence length in the batch, truncate to the maximum length the model can accept and return tensors directly with the following:

PyTorch | TensorFlow

```
batch = tokenizer(batch_sentences, batch_of_second_sentences, padding=True,
truncation=True, return_tensors="pt")
```

Everything you always wanted to know about padding and truncation

We have seen the commands that will work for most cases (pad your batch to the length of the maximum sentence and

truncate to the maximum length the mode can accept). However, the API supports more strategies if you need them. The three arguments you need to know for this are `padding`, `truncation` and `max_length`.

- `padding` controls the padding. It can be a boolean or a string which should be:
    - `True` or `'longest'` to pad to the longest sequence in the batch (doing no padding if you only provide a single sequence).
    - `'max_length'` to pad to a length specified by the `max_length` argument or the maximum length accepted by the model if no `max_length` is provided (`max_length=None`). If you only provide a single sequence, padding will still be applied to it.
    - `False` or `'do_not_pad'` to not pad the sequences. As we have seen before, this is the default behavior.

- `truncation` controls the truncation. It can be a boolean or a string which should be:
    - `True` or `'only_first'` truncate to a maximum length specified by the `max_length` argument or the maximum length accepted by the model if no `max_length` is provided (`max_length=None`). This will only truncate the first sentence of a pair if a pair of sequence (or a batch of pairs of sequences) is provided.
    - `'only_second'` truncate to a maximum length specified by the `max_length` argument or the maximum length accepted by the model if no `max_length` is provided (`max_length=None`). This will only truncate the second sentence of a pair if a pair of sequence (or a batch of pairs of sequences) is provided.
    - `'longest_first'` truncate to a maximum length specified by the `max_length` argument or the maximum length accepted by the model if no `max_length` is provided (`max_length=None`). This will truncate token by token, removing a token from the longest sequence in the pair until the proper length is reached.
    - `False` or `'do_not_truncate'` to not truncate the sequences. As we have seen before, this is the default behavior.

- `max_length` to control the length of the padding/truncation. It can be an integer or `None`, in which case it will default to the maximum length the model can accept. If the model has no specific maximum input length, truncation/padding to `max_length` is deactivated.

Here is a table summarizing the recommend way to setup padding and truncation. If you use pair of inputs sequence in any of the following examples, you can replace `truncation=True` by a `STRATEGY` selected in `['only_first', 'only_second', 'longest_first']`, i.e. `truncation='only_second'` or `truncation= 'longest_first'` to control how both sequence in the pair are truncated as detailed before.

Truncation	Padding	Instruction
no truncation	no padding	`tokenizer(batch_sentences)`
	padding to max sequence in batch	`tokenizer(batch_sentences, padding=True)` or `tokenizer(batch_sentences, padding='longest')`
	padding to max model input length	`tokenizer(batch_sentences, padding='max_length')`
	padding to specific length	`tokenizer(batch_sentences, padding='max_length', ma`
truncation to max model input length	no padding	`tokenizer(batch_sentences, truncation=True)` or `tokenizer(batch_sentences, truncation=STRATEGY)`
	padding to max sequence in batch	`tokenizer(batch_sentences, padding=True, truncation` `tokenizer(batch_sentences, padding=True, truncation`
	padding to max model input length	`tokenizer(batch_sentences, padding='max_length', tr` `tokenizer(batch_sentences, padding='max_length', tr`
	padding to specific length	Not possible

truncation to specific length	no padding	`tokenizer(batch_sentences, truncation=True, max_len` `tokenizer(batch_sentences, truncation=STRATEGY, max_`
	padding to max sequence in batch	`tokenizer(batch_sentences, padding=True, truncation` `tokenizer(batch_sentences, padding=True, truncation`
	padding to max model input length	Not possible
	padding to specific length	`tokenizer(batch_sentences, padding='max_length', tr` `tokenizer(batch_sentences, padding='max_length', tr`

Pre-tokenized inputs

The tokenizer also accept pre-tokenized inputs. This is particularly useful when you want to compute labels and extract predictions in named entity recognition (NER) or part-of-speech tagging (POS tagging).

If you want to use pre-tokenized inputs, just set `is_pretokenized=True` when passing your inputs to the tokenizer. For instance:

```
encoded_input = tokenizer(["Hello", "I'm", "a", "single", "sentence"],
is_pretokenized=True)
print(encoded_input)
```

will return:

```
{'input_ids': [101, 8667, 146, 112, 182, 170, 1423, 5650, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

Note that the tokenizer still adds the ids of special tokens (if applicable) unless you pass `add_special_tokens=False`.

This works exactly as before for batch of sentences or batch of pairs of sentences. You can encode a batch of sentences like this:

```
batch_sentences = [["Hello", "I'm", "a", "single", "sentence"],
                   ["And", "another", "sentence"],
                   ["And", "the", "very", "very", "last", "one"]]
encoded_inputs = tokenizer(batch_sentences, is_pretokenized=True)
```

or a batch of pair sentences like this:

```
batch_of_second_sentences = [["I'm", "a", "sentence", "that", "goes", "with",
"the", "first", "sentence"],
                             ["And", "I", "should", "be", "encoded", "with",
"the", "second", "sentence"],
                             ["And", "I", "go", "with", "the", "very", "last",
"one"]]
encoded_inputs = tokenizer(batch_sentences, batch_of_second_sentences,
is_pretokenized=True)
```

And you can add padding, truncation as well as directly return tensors like before:

PyTorch | TensorFlow

```
batch = tokenizer(batch_sentences,
                  batch_of_second_sentences,
                  is_pretokenized=True,
                  padding=True,
                  truncation=True,
                  return_tensors="pt")
```

Training and fine-tuning

Model classes in 🤗 Transformers are designed to be compatible with native PyTorch and TensorFlow 2 and can be used seemlessly with either. In this quickstart, we will show how to fine-tune (or train from scratch) a model using the standard training tools available in either framework. We will also show how to use our included `Trainer()` class which handles much of the complexity of training for you.

This guide assume that you are already familiar with loading and use our models for inference; otherwise, see the task summary. We also assume that you are familiar with training deep neural networks in either PyTorch or TF2, and focus specifically on the nuances and tools for training models in 🤗 Transformers.

Sections:

- Fine-tuning in native PyTorch
- Fine-tuning in native TensorFlow 2
- Trainer
- Additional resources

Fine-tuning in native PyTorch

Model classes in 🤗 Transformers that don't begin with `TF` are PyTorch Modules, meaning that you can use them just as you would any model in PyTorch for both inference and optimization.

Let's consider the common task of fine-tuning a masked language model like BERT on a sequence classification dataset. When we instantiate a model with `from_pretrained()`, the model configuration and pre-trained weights of the specified model are used to initialize the model. The library also includes a number of task-specific final layers or 'heads' whose weights are instantiated randomly when not present in the specified pre-trained model. For example, instantiating a model with `BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)` will create a BERT model instance with encoder weights copied from the `bert-base-uncased` model and a randomly initialized sequence classification head on top of the encoder with an output size of 2. Models are initialized in `eval` mode by default. We can call `model.train()` to put it in train mode.

```
from transformers import BertForSequenceClassification
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
model.train()
```

This is useful because it allows us to make use of the pre-trained BERT encoder and easily train it on whatever sequence classification dataset we choose. We can use any PyTorch optimizer, but our library also provides the `AdamW()` optimizer which implements gradient bias correction as well as weight decay.

```
from transformers import AdamW
optimizer = AdamW(model.parameters(), lr=1e-5)
```

The optimizer allows us to apply different hyperpameters for specific parameter groups. For example, we can apply weight decay to all parameters other than bias and layer normalization terms:

```
no_decay = ['bias', 'LayerNorm.weight']
optimizer_grouped_parameters = [
    {'params': [p for n, p in model.named_parameters() if not any(nd in n for
nd in no_decay)], 'weight_decay': 0.01},
    {'params': [p for n, p in model.named_parameters() if any(nd in n for nd
in no_decay)], 'weight_decay': 0.0}
]
optimizer = AdamW(optimizer_grouped_parameters, lr=1e-5)
```

Now we can set up a simple dummy training batch using `__call__()`. This returns a `BatchEncoding()` instance which prepares everything we might need to pass to the model.

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
text_batch = ["I love Pixar.", "I don't care for Pixar."]
encoding = tokenizer(text_batch, return_tensors='pt', padding=True,
truncation=True)
input_ids = encoding['input_ids']
attention_mask = encoding['attention_mask']
```

When we call a classification model with the `labels` argument, the first returned element is the Cross Entropy loss between the predictions and the passed labels. Having already set up our optimizer, we can then do a backwards pass and update the weights:

```
labels = torch.tensor([1,0]).unsqueeze(0)
outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
loss = outputs[0]
loss.backward()
optimizer.step()
```

Alternatively, you can just get the logits and calculate the loss yourself. The following is equivalent to the previous example:

```
from torch.nn import functional as F
labels = torch.tensor([1,0]).unsqueeze(0)
outputs = model(input_ids, attention_mask=attention_mask)
loss = F.cross_entropy(labels, outputs[0])
loss.backward()
optimizer.step()
```

Of course, you can train on GPU by calling `to('cuda')` on the model and inputs as usual.

We also provide a few learning rate scheduling tools. With the following, we can set up a scheduler which warms up for `num_warmup_steps` and then linearly decays to 0 by the end of training.

```
from transformers import get_linear_schedule_with_warmup
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps,
num_train_steps)
```

Then all we have to do is call `scheduler.step()` after `optimizer.step()`.

```
...
loss.backward()
optimizer.step()
scheduler.step()
```

We highly recommend using `Trainer()`, discussed below, which conveniently handles the moving parts of training 🤗 Transformers models with features like mixed precision and easy tensorboard logging.

Freezing the encoder

In some cases, you might be interested in keeping the weights of the pre-trained encoder frozen and optimizing only the weights of the head layers. To do so, simply set the `requires_grad` attribute to `False` on the encoder parameters, which can be accessed with the `base_model` submodule on any task-specific model in the library:

```
for param in model.base_model.parameters():
    param.requires_grad = False
```

Fine-tuning in native TensorFlow 2

Models can also be trained natively in TensorFlow 2. Just as with PyTorch, TensorFlow models can be instantiated with `from_pretrained()` to load the weights of the encoder from a pretrained model.

```
from transformers import TFBertForSequenceClassification
model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased')
```

Let's use `tensorflow_datasets` to load in the MRPC dataset from GLUE. We can then use our built-in `glue_convert_examples_to_features()` to tokenize MRPC and convert it to a TensorFlow `Dataset` object. Note that tokenizers are framework-agnostic, so there is no need to prepend `TF` to the pretrained tokenizer name.

```
from transformers import BertTokenizer, glue_convert_examples_to_features
import tensorflow_datasets as tfds
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
data = tfds.load('glue/mrpc')
train_dataset = glue_convert_examples_to_features(data['train'], tokenizer,
max_length=128, task='mrpc')
train_dataset = train_dataset.shuffle(100).batch(32).repeat(2)
```

The model can then be compiled and trained as any Keras model:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer=optimizer, loss=loss)
model.fit(train_dataset, epochs=2, steps_per_epoch=115)
```

With the tight interoperability between TensorFlow and PyTorch models, you can even save the model and then reload it as a PyTorch model (or vice-versa):

```
from transformers import BertForSequenceClassification
model.save_pretrained('./my_mrpc_model/')
pytorch_model =
BertForSequenceClassification.from_pretrained('./my_mrpc_model/',
from_tf=True)
```

Trainer

We also provide a simple but feature-complete training and evaluation interface through `Trainer()` and `TFTrainer()`. You can train, fine-tune, and evaluate any 🤗 Transformers model with a wide range of training options and with built-in features like logging, gradient accumulation, and mixed precision.

PyTorch | TensorFlow

```
from transformers import BertForSequenceClassification, Trainer,
TrainingArguments

model = BertForSequenceClassification.from_pretrained("bert-large-uncased")

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total # of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning
rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
)

trainer = Trainer(
    model=model,                         # the instantiated 🤗 Transformers
model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=test_dataset            # evaluation dataset
)
```

Now simply call `trainer.train()` to train and `trainer.evaluate()` to evaluate. You can use your own module as well, but the first argument returned from `forward` must be the loss which you wish to optimize.

`Trainer()` uses a built-in default function to collate batches and prepare them to be fed into the model. If needed, you can also use the `data_collator` argument to pass your own collator function which takes in the data in the format provided by your dataset and returns a batch ready to be fed into the model. Note that `TFTrainer()` expects the passed datasets to be dataset objects from `tensorflow_datasets`.

To calculate additional metrics in addition to the loss, you can also define your own `compute_metrics` function and pass it to the trainer.

```
from sklearn.metrics import precision_recall_fscore_support

def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds,
average='binary')
    acc = accuracy_score(labels, preds)
    return {
        'accuracy': acc,
        'f1': f1,
        'precision': precision,
        'recall': recall
    }
```

Finally, you can view the results, including any calculated metrics, by launching tensorboard in your specified `logging_dir` directory.