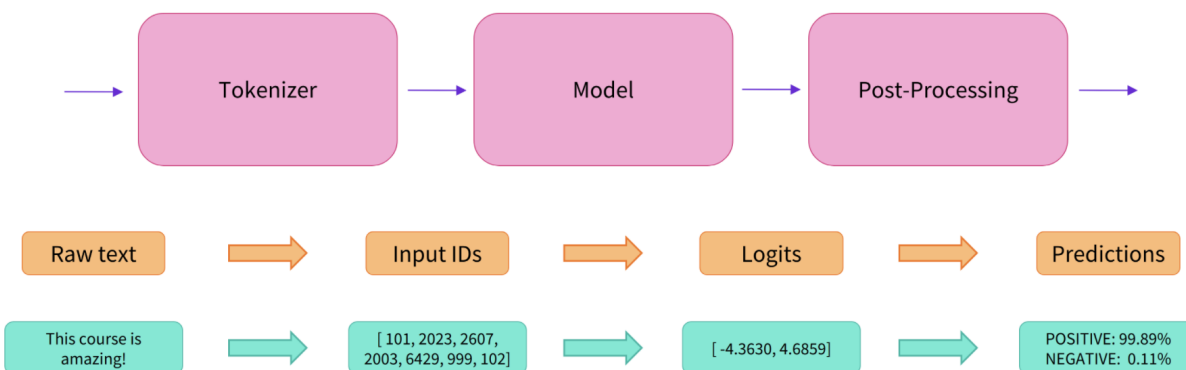


Using 🤗 Transformers

M2 I1. Behind the pipeline



When you use a pretrained Transformer for your down stream task, we have to make sure that the tokenizer used is same as the one which was used while building the pretrained model.

To do this, we can use the `AutoTokenizer` class and its `from_pretrained()` method. Using the checkpoint name of our model, it will automatically fetch the data associated with the model's tokenizer and cache it

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

# PreTrainedTokenizerFast(
#   name_or_path='distilbert-base-uncased-finetuned-sst-2-english',
#   vocab_size=30522, model_max_len=512, is_fast=True, padding_side='right',
```

```
# special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]',
# 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'})
```

The tokenizer now can be passed sentences and we get back a dictionary of list of ID's which need to be converted to tensors.

Example:

```
raw_input = [
    'Hey I am for a test',
    'All the best'
]
inputs = tokenizer(raw_input, padding=True, truncation=True,
                  return_tensors='tf'
                  )
print(inputs)
# {'input_ids': <tf.Tensor: shape=(2, 8), dtype=int32, numpy=
# array([[ 101, 4931, 1045, 2572, 2005, 1037, 3231,  102],
#        [ 101, 2035, 1996, 2190,  102,    0,    0,    0]], dtype=int32)>,
# 'attention_mask': <tf.Tensor: shape=(2, 8), dtype=int32, numpy=
# array([[1, 1, 1, 1, 1, 1, 1, 1],
#        [1, 1, 1, 1, 1, 0, 0, 0]], dtype=int32)>}
```

attention_mask will be explained later.

Going through the model

```
model = TFAutoModel.from_pretrained(checkpoint)
```

This architecture contains only the base Transformer module, given some inputs it will output **hidden states/features** . For each model input, we'll retrieve a high-dimensional vector representing the **contextual understanding of that input by the Transformer model**.

The output high-dimensional Vector

```
outputs = model(inputs)
print(outputs.last_hidden_state.shape)

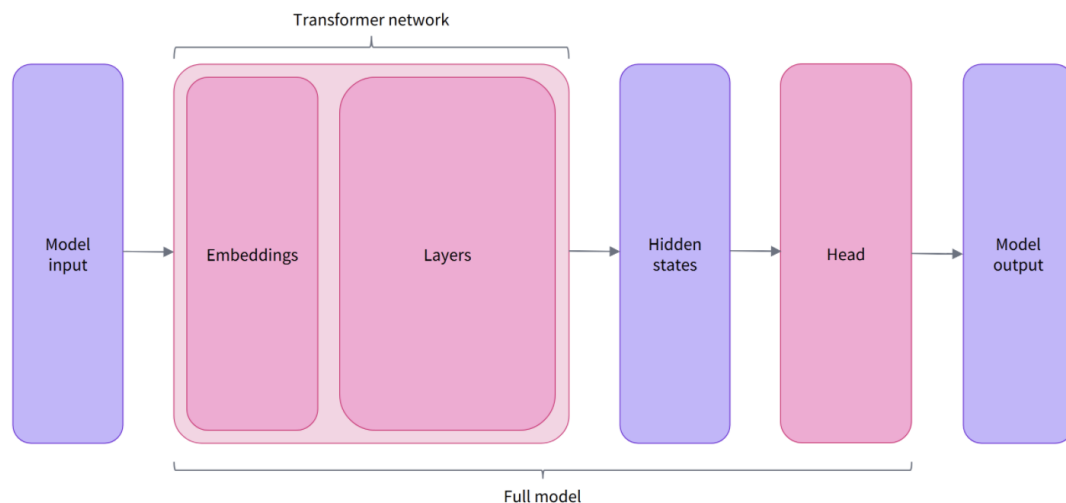
#(2, 8, 768)
```

These vectors so returned are the input to the head of the Transformer.

The vector output by the Transformer module is usually large. It generally has three dimensions:

- **Batch size:** The number of sequences processed at a time (2 in our example).
- **Sequence length:** The length of the numerical representation of the sequence (8 in our example).
- **Hidden size:** The vector dimension of each model input.

Model heads



The output of the Transformer model is sent directly to the model head to be processed.

1. Our input sentence is broken down into and converted to input id's
2. Then these input id's are mapped to vectors (through dictionary look ups)

3. The subsequent layers manipulate those vectors using the attention mechanism to produce the final representation of the sentences.

For a text classification task like we have we won't be using the TFAutoModel Class but we will use the TFAutoModelForSequenceClassification.

```
senti_model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
outputs = senti_model(inputs)

# TFSequenceClassifierOutput(loss=None, logits=<tf.Tensor: shape=(2, 2),
# dtype=float32, numpy=
# array([[-1.4058973,  1.5513633],
#        [-4.28962   ,  4.6092067]], dtype=float32)>,
# hidden_states=None, attentions=None)
```

The output array so returned i.e :

For sentence 1 from our raw_input we get [-1.4058973, 1.5513633]

For sentence 2 from our raw_input we get [-4.28962 , 4.6092067]

These are logits and not the output probabilities. To get the probability score we have pass these logits through a softmax. We will use tensorflow for getting our probability scores.

Postprocessing the output

```
print(outputs.logits)
# tf.Tensor(
# [[-1.4058973  1.5513633]
#  [-4.28962   4.6092067]], shape=(2, 2), dtype=float32)
```

Now lets get our probabilities:

```
import tensorflow as tf
predictions = tf.math.softmax(outputs.logits, axis=-1)
print(predictions)

# tf.Tensor(
# [[4.9394473e-02 9.5060551e-01]
# [1.3653041e-04 9.9986351e-01]], shape=(2, 2), dtype=float32)
```

The sentence 1 has probabilities: [0.049, 0.950]

The sentence 2 has probabilities: [0.0001, 0.998]

We can check the labels for the task using the following code

```
model.config.id2label
# {0: 'NEGATIVE', 1: 'POSITIVE'}
```

Therefore our results are as follows:

First Sentence NEGATIVE: 0.049, POSITIVE: 0.950

Second Sentence NEGATIVE: 0.0001, POSITIVE: 0.998

M2 V2 Models:

There are a couple of ways of Loading the Models:

1. From the default config

```
from transformers import BertConfig, TFBertModel

# Building the config
config = BertConfig()

# Building the model from the config
model = TFBertModel(config)
```

```
# Model is randomly initialized
```

In the above code a Bert Model is constructed from the config file that is loaded.

The configuration contains many attributes that are used to build the model:

```
print(config)

# Output
BertConfig {
  [...]
  "hidden_size": 768,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  [...]
}
```

This way of building model will initialize the BERT model with random weights and it won't be able to perform well. We can train the model and fine-tune it on our end task but that will require annotated dataset and some time.

2. Loading the pretrained model

```
from transformers import TFBertModel
model = TFBertModel.from_pretrained("bert-base-cased")
```

Instead of using TFBertModel in the above code we can also use TFAutoModel. Using TFAutoModel is more preferred way as it produces checkpoint-agnostic code.

To know what the checkpoint model we passed do we can take a look at the [**model card**](#).

The default weights are saved and cached under `~/.cache/huggingface/transformers`. You can customize your cache folder by setting the `HF_HOME` environment variable.

Saving a model:

```
model.save_pretrained("directory_on_my_computer")
```

This will save 2 files on computer:

1. config.json
2. tf_model.h5

Lets use a Transformer model for inference:

```
import tensorflow as tf

sequences = ["Hello!", "Cool.", "Nice!"]

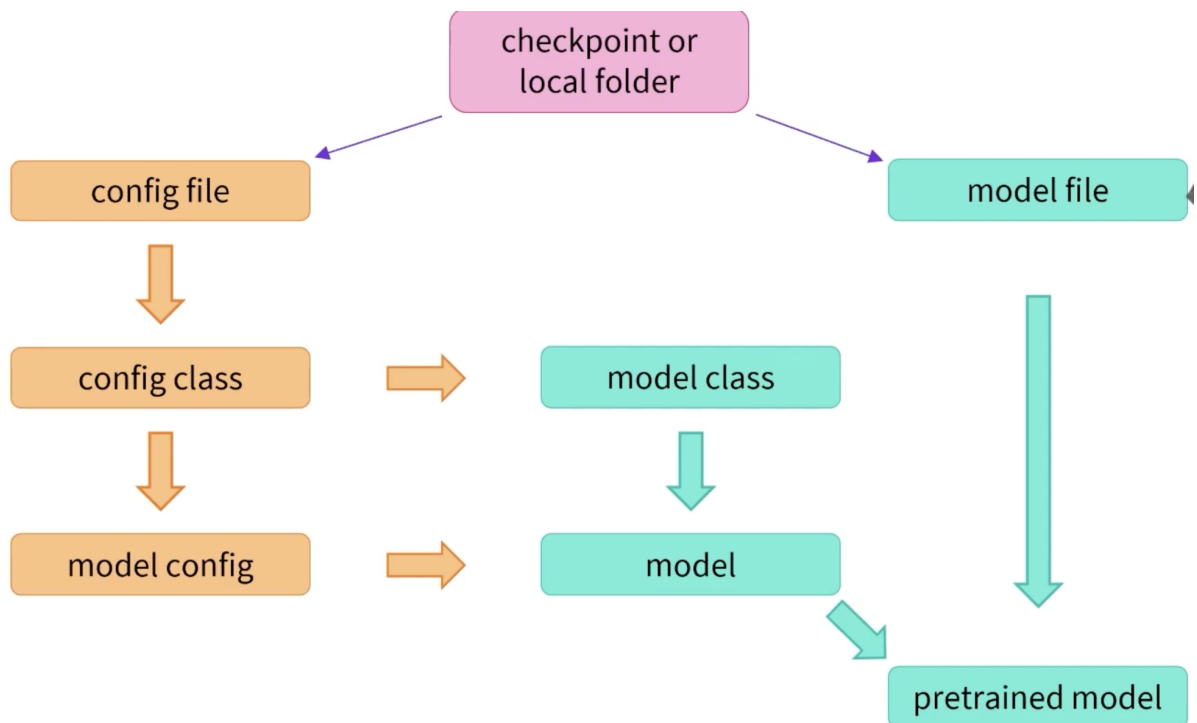
encoded_sequences = [
    [101, 7592, 999, 102],
    [101, 4658, 1012, 102],
    [101, 3835, 999, 102],
]

model = TFAutoModel.from_pretrained("bert-base-cased")
model_inputs = tf.constant(encoded_sequences)

output = model(model_inputs)
print(output.last_hidden_state.shape)
# (3, 4, 768)
# (3 input senetences, sequence lenght of 4, vector size of 768)
```

The above code can't be used for a classification task. Because the model doesn't have a classification layer. In the previous example for this task we had used `TFAutoModelForSequenceClassification` instead of `TFAutoModel`.

How does the AutoModels work? A high level overview of what goes under the hood.



Behind the `AutoModel.from_pretrained()` method

There are specific configs in Transformers library to load and build Transformer models


```
from transformers import BertConfig

bert_config = BertConfig.from_pretrained("bert-base-cased")
print(type(bert_config))
```

```
<class 'transformers.models.bert.configuration_bert.BertConfig'>
```

```
from transformers import GPT2Config

gpt_config = GPT2Config.from_pretrained("gpt2")
print(type(gpt_config))
```

```
<class 'transformers.models.gpt2.configuration_gpt2.GPT2Config'>
```

```
from transformers import BartConfig

bart_config = BartConfig.from_pretrained("facebook/bart-base")
print(type(bart_config))
```

```
<class 'transformers.models.bart.configuration_bart.BartConfig'>
```

ORE VIDEOS

But you can also use the specific class if you know it.

We can build and tweak the transformer architecture using the keywords, check the sample code below:

Same architecture as bert-base-cased

```
from transformers import BertConfig, TFBertModel

bert_config = BertConfig.from_pretrained("bert-base-cased")
bert_model = TFBertModel(bert_config)
```

Using only 10 layers instead of 12

```
from transformers import BertConfig, TFBertModel

bert_config = BertConfig.from_pretrained("bert-base-cased", num_hidden_layers=10)
bert_model = TFBertModel(bert_config)
```

Then you can instantiate a given model with random weights from this config.