

# Transformers

State-of-the-art Machine Learning for [PyTorch](#), [TensorFlow](#), and [JAX](#).

😊 Transformers provides APIs and tools to easily download and train state-of-the-art pretrained models. Using pretrained models can reduce your compute costs, carbon footprint, and save you the time and resources required to train a model from scratch. These models support common tasks in different modalities, such as:

📄 **Natural Language Processing:** text classification, named entity recognition, question answering, language modeling, summarization, translation, multiple choice, and text generation.

🖼️ **Computer Vision:** image classification, object detection, and segmentation.

👤 **Audio:** automatic speech recognition and audio classification.

🔥 **Multimodal:** table question answering, optical character recognition, information extraction from scanned documents, video classification, and visual question answering.

😊 Transformers support framework interoperability between PyTorch, TensorFlow, and JAX. This provides the flexibility to use a different framework at each stage of a model's life; train a model in three lines of code in one framework, and load it for inference in another. Models can also be exported to a format like ONNX and TorchScript for deployment in production environments.

Join the growing community on the [Hub](#), [forum](#), or [Discord](#) today!

## Contents

The documentation is organized into five sections:

- **GET STARTED** provides a quick tour of the library and installation instructions to get up and running.
- **TUTORIALS** are a great place to start if you're a beginner. This section will help you gain the basic skills you need to start using the library.
- **HOW-TO GUIDES** show you how to achieve a specific goal, like finetuning a pretrained model for language modeling or how to write and share a custom model.
- **CONCEPTUAL GUIDES** offers more discussion and explanation of the underlying concepts and ideas behind models, tasks, and the design philosophy of 😊 Transformers.
- **API** describes all classes and functions:
  - **MAIN CLASSES** details the most important classes like configuration, model, tokenizer, and pipeline.
  - **MODELS** details the classes and functions related to each model implemented in the library.
  - **INTERNAL HELPERS** details utility classes and functions used internally.

## Supported models and frameworks

The table below represents the current support in the library for each of those models, whether they have a Python tokenizer (called “slow”). A “fast” tokenizer backed by the 🧡 Tokenizers library, whether they have support in Jax (via Flax), PyTorch, and/or TensorFlow.

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">ALBERT</a>	✓	✓	✓
<a href="#">ALIGN</a>	✓	✗	✗
<a href="#">AltCLIP</a>	✓	✗	✗
<a href="#">Audio Spectrogram Transformer</a>	✓	✗	✗
<a href="#">Autoformer</a>	✓	✗	✗
<a href="#">Bark</a>	✓	✗	✗
<a href="#">BART</a>	✓	✓	✓
<a href="#">BARThez</a>	✓	✓	✓
<a href="#">BARTpho</a>	✓	✓	✓
<a href="#">BEiT</a>	✓	✗	✓
<a href="#">BERT</a>	✓	✓	✓
<a href="#">Bert Generation</a>	✓	✗	✗
<a href="#">BertJapanese</a>	✓	✓	✓
<a href="#">BERTweet</a>	✓	✓	✓
<a href="#">BigBird</a>	✓	✗	✓
<a href="#">BigBird-Pegasus</a>	✓	✗	✗
<a href="#">BioGpt</a>	✓	✗	✗
<a href="#">BiT</a>	✓	✗	✗
<a href="#">Blenderbot</a>	✓	✓	✓
<a href="#">BlenderbotSmall</a>	✓	✓	✓
<a href="#">BLIP</a>	✓	✓	✗
<a href="#">BLIP-2</a>	✓	✗	✗
<a href="#">BLOOM</a>	✓	✗	✓
<a href="#">BORT</a>	✓	✓	✓
<a href="#">BridgeTower</a>	✓	✗	✗
<a href="#">BROS</a>	✓	✗	✗
<a href="#">ByT5</a>	✓	✓	✓
<a href="#">CamemBERT</a>	✓	✓	✗
<a href="#">CANINE</a>	✓	✗	✗
<a href="#">Chameleon</a>	✓	✗	✗
<a href="#">Chinese-CLIP</a>	✓	✗	✗

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">CLAP</a>	✓	✗	✗
<a href="#">CLIP</a>	✓	✓	✓
<a href="#">CLIPSeg</a>	✓	✗	✗
<a href="#">CLVP</a>	✓	✗	✗
<a href="#">CodeGen</a>	✓	✗	✗
<a href="#">CodeLlama</a>	✓	✗	✓
<a href="#">Cohere</a>	✓	✗	✗
<a href="#">Conditional DETR</a>	✓	✗	✗
<a href="#">ConvBERT</a>	✓	✓	✗
<a href="#">ConvNeXT</a>	✓	✓	✗
<a href="#">ConvNeXTV2</a>	✓	✓	✗
<a href="#">CPM</a>	✓	✓	✓
<a href="#">CPM-Ant</a>	✓	✗	✗
<a href="#">CTRL</a>	✓	✓	✗
<a href="#">CvT</a>	✓	✓	✗
<a href="#">DAC</a>	✓	✗	✗
<a href="#">Data2VecAudio</a>	✓	✗	✗
<a href="#">Data2VecText</a>	✓	✗	✗
<a href="#">Data2VecVision</a>	✓	✓	✗
<a href="#">DBRX</a>	✓	✗	✗
<a href="#">DeBERTa</a>	✓	✓	✗
<a href="#">DeBERTa-v2</a>	✓	✓	✗
<a href="#">Decision Transformer</a>	✓	✗	✗
<a href="#">Deformable DETR</a>	✓	✗	✗
<a href="#">DeiT</a>	✓	✓	✗
<a href="#">DePlot</a>	✓	✗	✗
<a href="#">Depth Anything</a>	✓	✗	✗
<a href="#">DETA</a>	✓	✗	✗
<a href="#">DETR</a>	✓	✗	✗
<a href="#">DialoGPT</a>	✓	✓	✓
<a href="#">DiNAT</a>	✓	✗	✗
<a href="#">DINOv2</a>	✓	✗	✓
<a href="#">DistilBERT</a>	✓	✓	✓
<a href="#">DiT</a>	✓	✗	✓
<a href="#">DonutSwin</a>	✓	✗	✗

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">DPR</a>	✓	✓	✗
<a href="#">DPT</a>	✓	✗	✗
<a href="#">EfficientFormer</a>	✓	✓	✗
<a href="#">EfficientNet</a>	✓	✗	✗
<a href="#">ELECTRA</a>	✓	✓	✓
<a href="#">EnCodec</a>	✓	✗	✗
<a href="#">Encoder decoder</a>	✓	✓	✓
<a href="#">ERNIE</a>	✓	✗	✗
<a href="#">ErnieM</a>	✓	✗	✗
<a href="#">ESM</a>	✓	✓	✗
<a href="#">FairSeq Machine-Translation</a>	✓	✗	✗
<a href="#">Falcon</a>	✓	✗	✗
<a href="#">FalconMamba</a>	✓	✗	✗
<a href="#">FastSpeech2Conformer</a>	✓	✗	✗
<a href="#">FLAN-T5</a>	✓	✓	✓
<a href="#">FLAN-UL2</a>	✓	✓	✓
<a href="#">FlauBERT</a>	✓	✓	✗
<a href="#">FLAVA</a>	✓	✗	✗
<a href="#">FNet</a>	✓	✗	✗
<a href="#">FocalNet</a>	✓	✗	✗
<a href="#">Funnel Transformer</a>	✓	✓	✗
<a href="#">Fuyu</a>	✓	✗	✗
<a href="#">Gemma</a>	✓	✗	✓
<a href="#">Gemma2</a>	✓	✗	✗
<a href="#">GIT</a>	✓	✗	✗
<a href="#">GLPN</a>	✓	✗	✗
<a href="#">GPT Neo</a>	✓	✗	✓
<a href="#">GPT NeoX</a>	✓	✗	✗
<a href="#">GPT NeoX Japanese</a>	✓	✗	✗
<a href="#">GPT-J</a>	✓	✓	✓
<a href="#">GPT-Sw3</a>	✓	✓	✓
<a href="#">GPTBigCode</a>	✓	✗	✗
<a href="#">GPTSAN-japanese</a>	✓	✗	✗
<a href="#">Granite</a>	✓	✗	✗
<a href="#">GraniteMoeMoe</a>	✓	✗	✗

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">Graphormer</a>	✓	✗	✗
<a href="#">Grounding DINO</a>	✓	✗	✗
<a href="#">GroupViT</a>	✓	✓	✗
<a href="#">HerBERT</a>	✓	✓	✓
<a href="#">Hiera</a>	✓	✗	✗
<a href="#">Hubert</a>	✓	✓	✗
<a href="#">I-BERT</a>	✓	✗	✗
<a href="#">IDEFICS</a>	✓	✓	✗
<a href="#">Idefics2</a>	✓	✗	✗
<a href="#">ImageGPT</a>	✓	✗	✗
<a href="#">Informer</a>	✓	✗	✗
<a href="#">InstructBLIP</a>	✓	✗	✗
<a href="#">InstructBlipVideo</a>	✓	✗	✗
<a href="#">Jamba</a>	✓	✗	✗
<a href="#">JetMoe</a>	✓	✗	✗
<a href="#">Jukebox</a>	✓	✗	✗
<a href="#">KOSMOS-2</a>	✓	✗	✗
<a href="#">LayoutLM</a>	✓	✓	✗
<a href="#">LayoutLMv2</a>	✓	✗	✗
<a href="#">LayoutLMv3</a>	✓	✓	✗
<a href="#">LayoutXLM</a>	✓	✗	✗
<a href="#">LED</a>	✓	✓	✗
<a href="#">LeViT</a>	✓	✗	✗
<a href="#">LiLT</a>	✓	✗	✗
<a href="#">LLaMA</a>	✓	✗	✓
<a href="#">Llama2</a>	✓	✗	✓
<a href="#">Llama3</a>	✓	✗	✓
<a href="#">LLaVa</a>	✓	✗	✗
<a href="#">LLaVA-NeXT</a>	✓	✗	✗
<a href="#">LLaVa-NeXT-Video</a>	✓	✗	✗
<a href="#">LLaVA-Onevision</a>	✓	✗	✗
<a href="#">Longformer</a>	✓	✓	✗
<a href="#">LongT5</a>	✓	✗	✓
<a href="#">LUKE</a>	✓	✗	✗
<a href="#">LXMERT</a>	✓	✓	✗

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">M-CTC-T</a>	✓	✗	✗
<a href="#">M2M100</a>	✓	✗	✗
<a href="#">MADLAD-400</a>	✓	✓	✓
<a href="#">Mamba</a>	✓	✗	✗
<a href="#">mamba2</a>	✓	✗	✗
<a href="#">Marian</a>	✓	✓	✓
<a href="#">MarkupLM</a>	✓	✗	✗
<a href="#">Mask2Former</a>	✓	✗	✗
<a href="#">MaskFormer</a>	✓	✗	✗
<a href="#">MatCha</a>	✓	✗	✗
<a href="#">mBART</a>	✓	✓	✓
<a href="#">mBART-50</a>	✓	✓	✓
<a href="#">MEGA</a>	✓	✗	✗
<a href="#">Megatron-BERT</a>	✓	✗	✗
<a href="#">Megatron-GPT2</a>	✓	✓	✓
<a href="#">MGP-STR</a>	✓	✗	✗
<a href="#">Mimi</a>	✓	✗	✗
<a href="#">Mistral</a>	✓	✓	✓
<a href="#">Mixtral</a>	✓	✗	✗
<a href="#">Mllama</a>	✓	✗	✗
<a href="#">mLUKE</a>	✓	✗	✗
<a href="#">MMS</a>	✓	✓	✓
<a href="#">MobileBERT</a>	✓	✓	✗
<a href="#">MobileNetV1</a>	✓	✗	✗
<a href="#">MobileNetV2</a>	✓	✗	✗
<a href="#">MobileViT</a>	✓	✓	✗
<a href="#">MobileViTV2</a>	✓	✗	✗
<a href="#">MPNet</a>	✓	✓	✗
<a href="#">MPT</a>	✓	✗	✗
<a href="#">MRA</a>	✓	✗	✗
<a href="#">MT5</a>	✓	✓	✓
<a href="#">MusicGen</a>	✓	✗	✗
<a href="#">MusicGen Melody</a>	✓	✗	✗
<a href="#">MVP</a>	✓	✗	✗
<a href="#">NAT</a>	✓	✗	✗

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">Nemotron</a>	✓	✗	✗
<a href="#">Nezha</a>	✓	✗	✗
<a href="#">NLLB</a>	✓	✗	✗
<a href="#">NLLB-MOE</a>	✓	✗	✗
<a href="#">Nougat</a>	✓	✓	✓
<a href="#">Nyströmformer</a>	✓	✗	✗
<a href="#">OLMo</a>	✓	✗	✗
<a href="#">OLMoE</a>	✓	✗	✗
<a href="#">OmDet-Turbo</a>	✓	✗	✗
<a href="#">OneFormer</a>	✓	✗	✗
<a href="#">OpenAI GPT</a>	✓	✓	✗
<a href="#">OpenAI GPT-2</a>	✓	✓	✓
<a href="#">OpenLlama</a>	✓	✗	✗
<a href="#">OPT</a>	✓	✓	✓
<a href="#">OWL-ViT</a>	✓	✗	✗
<a href="#">OWLv2</a>	✓	✗	✗
<a href="#">PaliGemma</a>	✓	✗	✗
<a href="#">PatchTSMixer</a>	✓	✗	✗
<a href="#">PatchTST</a>	✓	✗	✗
<a href="#">Pegasus</a>	✓	✓	✓
<a href="#">PEGASUS-X</a>	✓	✗	✗
<a href="#">Perceiver</a>	✓	✗	✗
<a href="#">Persimmon</a>	✓	✗	✗
<a href="#">Phi</a>	✓	✗	✗
<a href="#">Phi3</a>	✓	✗	✗
<a href="#">PhoBERT</a>	✓	✓	✓
<a href="#">Pix2Struct</a>	✓	✗	✗
<a href="#">Pixtral</a>	✓	✗	✗
<a href="#">PLBart</a>	✓	✗	✗
<a href="#">PoolFormer</a>	✓	✗	✗
<a href="#">Pop2Piano</a>	✓	✗	✗
<a href="#">ProphetNet</a>	✓	✗	✗
<a href="#">PVT</a>	✓	✗	✗
<a href="#">PVTv2</a>	✓	✗	✗
<a href="#">QDQBert</a>	✓	✗	✗

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">Qwen2</a>	✓	✗	✗
<a href="#">Qwen2Audio</a>	✓	✗	✗
<a href="#">Qwen2MoE</a>	✓	✗	✗
<a href="#">Qwen2VL</a>	✓	✗	✗
<a href="#">RAG</a>	✓	✓	✗
<a href="#">REALM</a>	✓	✗	✗
<a href="#">RecurrentGemma</a>	✓	✗	✗
<a href="#">Reformer</a>	✓	✗	✗
<a href="#">RegNet</a>	✓	✓	✓
<a href="#">RemBERT</a>	✓	✓	✗
<a href="#">ResNet</a>	✓	✓	✓
<a href="#">RetriBERT</a>	✓	✗	✗
<a href="#">RoBERTa</a>	✓	✓	✓
<a href="#">RoBERTa-PreLayerNorm</a>	✓	✓	✓
<a href="#">RoCBert</a>	✓	✗	✗
<a href="#">RoFormer</a>	✓	✓	✓
<a href="#">RT-DETR</a>	✓	✗	✗
<a href="#">RT-DETR-ResNet</a>	✓	✗	✗
<a href="#">RWKV</a>	✓	✗	✗
<a href="#">SAM</a>	✓	✓	✗
<a href="#">SeamlessM4T</a>	✓	✗	✗
<a href="#">SeamlessM4Tv2</a>	✓	✗	✗
<a href="#">SegFormer</a>	✓	✓	✗
<a href="#">SegGPT</a>	✓	✗	✗
<a href="#">SEW</a>	✓	✗	✗
<a href="#">SEW-D</a>	✓	✗	✗
<a href="#">SigLIP</a>	✓	✗	✗
<a href="#">Speech Encoder decoder</a>	✓	✗	✓
<a href="#">Speech2Text</a>	✓	✓	✗
<a href="#">SpeechT5</a>	✓	✗	✗
<a href="#">Splinter</a>	✓	✗	✗
<a href="#">SqueezeBERT</a>	✓	✗	✗
<a href="#">StableLm</a>	✓	✗	✗
<a href="#">Starcoder2</a>	✓	✗	✗
<a href="#">SuperPoint</a>	✓	✗	✗



Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">SwiftFormer</a>	✓	✓	✗
<a href="#">Swin Transformer</a>	✓	✓	✗
<a href="#">Swin Transformer V2</a>	✓	✗	✗
<a href="#">Swin2SR</a>	✓	✗	✗
<a href="#">SwitchTransformers</a>	✓	✗	✗
<a href="#">T5</a>	✓	✓	✓
<a href="#">T5v1.1</a>	✓	✓	✓
<a href="#">Table Transformer</a>	✓	✗	✗
<a href="#">TAPAS</a>	✓	✓	✗
<a href="#">TAPEX</a>	✓	✓	✓
<a href="#">Time Series Transformer</a>	✓	✗	✗
<a href="#">TimeSformer</a>	✓	✗	✗
<a href="#">Trajectory Transformer</a>	✓	✗	✗
<a href="#">Transformer-XL</a>	✓	✓	✗
<a href="#">TrOCR</a>	✓	✗	✗
<a href="#">TVLT</a>	✓	✗	✗
<a href="#">TVP</a>	✓	✗	✗
<a href="#">UDOP</a>	✓	✗	✗
<a href="#">UL2</a>	✓	✓	✓
<a href="#">UMT5</a>	✓	✗	✗
<a href="#">UniSpeech</a>	✓	✗	✗
<a href="#">UniSpeechSat</a>	✓	✗	✗
<a href="#">UnivNet</a>	✓	✗	✗
<a href="#">UPerNet</a>	✓	✗	✗
<a href="#">VAN</a>	✓	✗	✗
<a href="#">VideoLlava</a>	✓	✗	✗
<a href="#">VideoMAE</a>	✓	✗	✗
<a href="#">ViLT</a>	✓	✗	✗
<a href="#">VipLlava</a>	✓	✗	✗
<a href="#">Vision Encoder decoder</a>	✓	✓	✓
<a href="#">VisionTextDualEncoder</a>	✓	✓	✓
<a href="#">VisualBERT</a>	✓	✗	✗
<a href="#">ViT</a>	✓	✓	✓
<a href="#">ViT Hybrid</a>	✓	✗	✗
<a href="#">VitDet</a>	✓	✗	✗

Model	PyTorch support	TensorFlow support	Flax Support
<a href="#">ViTMAE</a>	✓	✓	✗
<a href="#">ViTMatte</a>	✓	✗	✗
<a href="#">ViTMSN</a>	✓	✗	✗
<a href="#">VITS</a>	✓	✗	✗
<a href="#">ViViT</a>	✓	✗	✗
<a href="#">Wav2Vec2</a>	✓	✓	✓
<a href="#">Wav2Vec2-BERT</a>	✓	✗	✗
<a href="#">Wav2Vec2-Conformer</a>	✓	✗	✗
<a href="#">Wav2Vec2Phoneme</a>	✓	✓	✓
<a href="#">WavLM</a>	✓	✗	✗
<a href="#">Whisper</a>	✓	✓	✓
<a href="#">X-CLIP</a>	✓	✗	✗
<a href="#">X-MOD</a>	✓	✗	✗
<a href="#">XGLM</a>	✓	✓	✓
<a href="#">XLM</a>	✓	✓	✗
<a href="#">XLM-ProphetNet</a>	✓	✗	✗
<a href="#">XLM-RoBERTa</a>	✓	✓	✓
<a href="#">XLM-RoBERTa-XL</a>	✓	✗	✗
<a href="#">XLM-V</a>	✓	✓	✓
<a href="#">XLNet</a>	✓	✓	✗
<a href="#">XLS-R</a>	✓	✓	✓
<a href="#">XLSR-Wav2Vec2</a>	✓	✓	✓
<a href="#">YOLOS</a>	✓	✗	✗
<a href="#">YOSO</a>	✓	✗	✗
<a href="#">ZoeDepth</a>	✓	✗	✗

# Quick tour

Get up and running with 🤗 Transformers! Whether you're a developer or an everyday user, this quick tour will help you get started and show you how to use the [pipeline\(\)](#) for inference, load a pretrained model and preprocessor with an [AutoClass](#), and quickly train a model with PyTorch or TensorFlow. If you're a beginner, we recommend checking out our tutorials or [course](#) next for more in-depth explanations of the concepts introduced here.

Before you begin, make sure you have all the necessary libraries installed:

```
!pip install transformers datasets evaluate accelerate
```

You'll also need to install your preferred machine learning framework:

Pytorch

Hide Pytorch content

```
pip install torch
```

TensorFlow

Hide TensorFlow content

```
pip install tensorflow
```

## Pipeline

The [pipeline\(\)](#) is the easiest and fastest way to use a pretrained model for inference. You can use the [pipeline\(\)](#) out-of-the-box for many tasks across different modalities, some of which are shown in the table below:

For a complete list of available tasks, check out the [pipeline API reference](#).

Task	Description	Modality	Pipeline identifier
Text classification	assign a label to a given sequence of text	NLP	<code>pipeline(task="sentiment-analysis")</code>
Text generation	generate text given a prompt	NLP	<code>pipeline(task="text-generation")</code>
Summarization	generate a summary of a sequence of text or document	NLP	<code>pipeline(task="summarization")</code>
Image classification	assign a label to an image	Computer vision	<code>pipeline(task="image-classification")</code>
Image segmentation	assign a label to each individual pixel of an image (supports semantic, panoptic, and instance segmentation)	Computer vision	<code>pipeline(task="image-segmentation")</code>

Task	Description	Modality	Pipeline identifier
Object detection	predict the bounding boxes and classes of objects in an image	Computer vision	<code>pipeline(task="object-detection")</code>
Audio classification	assign a label to some audio data	Audio	<code>pipeline(task="audio-classification")</code>
Automatic speech recognition	transcribe speech into text	Audio	<code>pipeline(task="automatic-speech-recognition")</code>
Visual question answering	answer a question about the image, given an image and a question	Multimodal	<code>pipeline(task="vqa")</code>
Document question answering	answer a question about the document, given a document and a question	Multimodal	<code>pipeline(task="document-question-answering")</code>
Image captioning	generate a caption for a given image	Multimodal	<code>pipeline(task="image-to-text")</code>

Start by creating an instance of [pipeline\(\)](#) and specifying a task you want to use it for. In this guide, you'll use the [pipeline\(\)](#) for sentiment analysis as an example:

```
>>> from transformers import pipeline

>>> classifier = pipeline("sentiment-analysis")
```

The [pipeline\(\)](#) downloads and caches a default [pretrained model](#) and tokenizer for sentiment analysis. Now you can use the `classifier` on your target text:

```
>>> classifier("We are very happy to show you the 😊 Transformers library.")
[{'label': 'POSITIVE', 'score': 0.9998}]
```

If you have more than one input, pass your inputs as a list to the [pipeline\(\)](#) to return a list of dictionaries:

```
>>> 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
```

The [pipeline\(\)](#) can also iterate over an entire dataset for any task you like. For this example, let's choose automatic speech recognition as our task:

```
>>> import torch
>>> from transformers import pipeline

>>> speech_recognizer = pipeline("automatic-speech-recognition",
model="facebook/wav2vec2-base-960h")
```

Load an audio dataset (see the 🧐 [Datasets Quick Start](#) for more details) you'd like to iterate over. For example, load the [MInDS-14](#) dataset:

```
>>> from datasets import load_dataset, Audio

>>> dataset = load_dataset("PolyAI/minds14", name="en-US", split="train")
```

You need to make sure the sampling rate of the dataset matches the sampling rate facebook/wav2vec2-base-960h was trained on:

```
>>> dataset = dataset.cast_column("audio",
Audio(sampling_rate=speech_recognizer.feature_extractor.sampling_rate))
```

The audio files are automatically loaded and resampled when calling the "audio" column. Extract the raw waveform arrays from the first 4 samples and pass it as a list to the pipeline:

```
>>> result = speech_recognizer(dataset[:4]["audio"])
>>> print([d["text"] for d in result])
['I WOULD LIKE TO SET UP A JOINT ACCOUNT WITH MY PARTNER HOW DO I PROCEED
WITH DOING THAT', 'FONDERING HOW I'D SET UP A JOIN TO HELL T WITH MY WIFE
AND WHERE THE AP MIGHT BE', 'I I'D LIKE TOY SET UP A JOINT ACCOUNT WITH MY
PARTNER I'M NOT SEEING THE OPTION TO DO IT ON THE APSO I CALLED IN TO GET
SOME HELP CAN I JUST DO IT OVER THE PHONE WITH YOU AND GIVE YOU THE
INFORMATION OR SHOULD I DO IT IN THE AP AN I'M MISSING SOMETHING UQUETTE
HAD PREFERRED TO JUST DO IT OVER THE PHONE OF POSSIBLE THINGS', 'HOW DO I
FURN A JOINA COUT']
```

For larger datasets where the inputs are big (like in speech or vision), you'll want to pass a generator instead of a list to load all the inputs in memory. Take a look at the [pipeline API reference](#) for more information.

Use another model and tokenizer in the pipeline

The [pipeline\(\)](#) can accommodate any model from the [Hub](#), making it easy to adapt the [pipeline\(\)](#) for other use-cases. For example, if you'd like a model capable of handling French text, use the tags on the Hub to filter for an appropriate model. The top filtered result returns a multilingual [BERT model](#) finetuned for sentiment analysis you can use for French text:

```
>>> model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
Pytorch
Hide Pytorch content
```

Use [AutoModelForSequenceClassification](#) and [AutoTokenizer](#) to load the pretrained model and it's associated tokenizer (more on an `AutoClass` in the next section):

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

>>> model = AutoModelForSequenceClassification.from_pretrained(model_name)
>>> tokenizer = AutoTokenizer.from_pretrained(model_name)
TensorFlow
Hide TensorFlow content
```

Use [TFAutoModelForSequenceClassification](#) and [AutoTokenizer](#) to load the pretrained model and it's associated tokenizer (more on an `TFAutoClass` in the next section):

```
>>> from transformers import AutoTokenizer,
TFAutoModelForSequenceClassification

>>> model =
TFAutoModelForSequenceClassification.from_pretrained(model_name)
>>> tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Specify the model and tokenizer in the [pipeline\(\)](#), and now you can apply the classifier on French text:

```
>>> classifier = pipeline("sentiment-analysis", model=model,
tokenizer=tokenizer)
>>> classifier("Nous sommes très heureux de vous présenter la bibliothèque
😊 Transformers.")
[{'label': '5 stars', 'score': 0.7273}]
```

If you can't find a model for your use-case, you'll need to finetune a pretrained model on your data. Take a look at our [finetuning tutorial](#) to learn how. Finally, after you've finetuned your pretrained model, please consider [sharing](#) the model with the community on the Hub to democratize machine learning for everyone! 😊

## AutoClass

Under the hood, the [AutoModelForSequenceClassification](#) and [AutoTokenizer](#) classes work together to power the [pipeline\(\)](#) you used above. An [AutoClass](#) is a shortcut that automatically retrieves the architecture of a pretrained model from its name or path. You only need to select the appropriate `AutoClass` for your task and it's associated preprocessing class.

Let's return to the example from the previous section and see how you can use the `AutoClass` to replicate the results of the [pipeline\(\)](#).

## AutoTokenizer

A tokenizer is responsible for preprocessing text into an array of numbers as inputs to a model. There are multiple rules that govern the tokenization process, including how to split a word and at what level words should be split (learn more about tokenization in the [tokenizer summary](#)). The most important thing to remember is you need to instantiate a tokenizer with the same model name to ensure you're using the same tokenization rules a model was pretrained with.

Load a tokenizer with [AutoTokenizer](#):

```
>>> from transformers import AutoTokenizer

>>> model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
>>> tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Pass your text to the tokenizer:

```
>>> encoding = tokenizer("We are very happy to show you the 😊  
Transformers library.")
>>> print(encoding)
{'input_ids': [101, 11312, 10320, 12495, 19308, 10114, 11391, 10855, 10103, 100, 58263, 13299, 119, 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 tokenizer returns a dictionary containing:

- [input\\_ids](#): numerical representations of your tokens.
- [attention\\_mask](#): indicates which tokens should be attended to.

A tokenizer can also accept a list of inputs, and pad and truncate the text to return a batch with uniform length:

## Pytorch

Hide Pytorch content

```
>>> pt_batch = tokenizer(
...     ["We are very happy to show you the 😊 Transformers library.", "We  
hope you don't hate it."],
...     padding=True,
...     truncation=True,
...     max_length=512,
...     return_tensors="pt",
... )
```

## TensorFlow

Hide TensorFlow content

```
>>> tf_batch = tokenizer(
...     ["We are very happy to show you the 😊 Transformers library.", "We  
hope you don't hate it."],
```

```
...     padding=True,
...     truncation=True,
...     max_length=512,
...     return_tensors="tf",
... )
```

Check out the [preprocess](#) tutorial for more details about tokenization, and how to use an [AutoImageProcessor](#), [AutoFeatureExtractor](#) and [AutoProcessor](#) to preprocess image, audio, and multimodal inputs.

AutoModel  
Pytorch  
Hide Pytorch content

😊 Transformers provides a simple and unified way to load pretrained instances. This means you can load an [AutoModel](#) like you would load an [AutoTokenizer](#). The only difference is selecting the correct [AutoModel](#) for the task. For text (or sequence) classification, you should load [AutoModelForSequenceClassification](#):

```
>>> from transformers import AutoModelForSequenceClassification

>>> model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
>>> pt_model =
AutoModelForSequenceClassification.from_pretrained(model_name)
```

See the [task summary](#) for tasks supported by an [AutoModel](#) class.

Now pass your preprocessed batch of inputs directly to the model. You just have to unpack the dictionary by adding \*\*:

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

The model outputs the final activations in the `logits` attribute. Apply the softmax function to the `logits` to retrieve the probabilities:

```
>>> from torch import nn

>>> pt_predictions = nn.functional.softmax(pt_outputs.logits, dim=-1)
>>> print(pt_predictions)
tensor([[0.0021, 0.0018, 0.0115, 0.2121, 0.7725],
        [0.2084, 0.1826, 0.1969, 0.1755, 0.2365]],
        grad_fn=<SoftmaxBackward0>)
```

TensorFlow  
Hide TensorFlow content

😊 Transformers provides a simple and unified way to load pretrained instances. This means you can load an [TFAutoModel](#) like you would load an [AutoTokenizer](#). The only difference is selecting the correct [TFAutoModel](#) for the task. For text (or sequence) classification, you should load [TFAutoModelForSequenceClassification](#):



```
>>> from transformers import TFAutoModelForSequenceClassification

>>> model_name = "nlptown/bert-base-multilingual-uncased-sentiment"
>>> tf_model =
TFAutoModelForSequenceClassification.from_pretrained(model_name)
```

See the [task summary](#) for tasks supported by an [AutoModel](#) class.

Now pass your preprocessed batch of inputs directly to the model. You can pass the tensors as-is:

```
>>> tf_outputs = tf_model(tf_batch)
```

The model outputs the final activations in the `logits` attribute. Apply the softmax function to the `logits` to retrieve the probabilities:

```
>>> import tensorflow as tf

>>> tf_predictions = tf.nn.softmax(tf_outputs.logits, axis=-1)
>>> tf_predictions
```

All 🤗 Transformers models (PyTorch or TensorFlow) output the tensors *before* the final activation function (like softmax) because the final activation function is often fused with the loss. Model outputs are special dataclasses so their attributes are autocompleted in an IDE. The model outputs behave like a tuple or a dictionary (you can index with an integer, a slice or a string) in which case, attributes that are `None` are ignored.

Save a model  
Pytorch  
Hide Pytorch content

Once your model is fine-tuned, you can save it with its tokenizer using [PreTrainedModel.save\\_pretrained\(\)](#):

```
>>> pt_save_directory = "./pt_save_pretrained"
>>> tokenizer.save_pretrained(pt_save_directory)
>>> pt_model.save_pretrained(pt_save_directory)
```

When you are ready to use the model again, reload it with [PreTrainedModel.from\\_pretrained\(\)](#):

```
>>> pt_model =
AutoModelForSequenceClassification.from_pretrained("./pt_save_pretrained")
TensorFlow
Hide TensorFlow content
```

Once your model is fine-tuned, you can save it with its tokenizer using [`TFPreTrainedModel.save\_pretrained\(\)`](#):

```
>>> tf_save_directory = "./tf_save_pretrained"
>>> tokenizer.save_pretrained(tf_save_directory)
>>> tf_model.save_pretrained(tf_save_directory)
```

When you are ready to use the model again, reload it with [`TFPreTrainedModel.from\_pretrained\(\)`](#):

```
>>> tf_model =
TFAutoModelForSequenceClassification.from_pretrained("./tf_save_pretrained"
)
```

One particularly cool 🤗 Transformers feature is the ability to save a model and reload it as either a PyTorch or TensorFlow model. The `from_pt` or `from_tf` parameter can convert the model from one framework to the other:

Pytorch

Hide Pytorch content

```
>>> from transformers import AutoModel

>>> tokenizer = AutoTokenizer.from_pretrained(tf_save_directory)
>>> pt_model =
AutoModelForSequenceClassification.from_pretrained(tf_save_directory,
from_tf=True)
```

TensorFlow

Hide TensorFlow content

```
>>> from transformers import TFAutoModel

>>> tokenizer = AutoTokenizer.from_pretrained(pt_save_directory)
>>> tf_model =
TFAutoModelForSequenceClassification.from_pretrained(pt_save_directory,
from_pt=True)
```

## Custom model builds

You can modify the model's configuration class to change how a model is built. The configuration specifies a model's attributes, such as the number of hidden layers or attention heads. You start from scratch when you initialize a model from a custom configuration class. The model attributes are randomly initialized, and you'll need to train the model before you can use it to get meaningful results.

Start by importing [`AutoConfig`](#), and then load the pretrained model you want to modify. Within [`AutoConfig.from\_pretrained\(\)`](#), you can specify the attribute you want to change, such as the number of attention heads:

```
>>> from transformers import AutoConfig
```

```
>>> my_config = AutoConfig.from_pretrained("distilbert/distilbert-base-uncased", n_heads=12)
```

Pytorch

Hide Pytorch content

Create a model from your custom configuration with [AutoModel.from\\_config\(\)](#):

```
>>> from transformers import AutoModel
```

```
>>> my_model = AutoModel.from_config(my_config)
```

TensorFlow

Hide TensorFlow content

Create a model from your custom configuration with [TFAutoModel.from\\_config\(\)](#):

```
>>> from transformers import TFAutoModel
```

```
>>> my_model = TFAutoModel.from_config(my_config)
```

Take a look at the [Create a custom architecture](#) guide for more information about building custom configurations.

## Trainer - a PyTorch optimized training loop

All models are a standard `torch.nn.Module` so you can use them in any typical training loop. While you can write your own training loop, 😊 Transformers provides a [Trainer](#) class for PyTorch, which contains the basic training loop and adds additional functionality for features like distributed training, mixed precision, and more.

Depending on your task, you'll typically pass the following parameters to [Trainer](#):

1. You'll start with a [PreTrainedModel](#) or a `torch.nn.Module`:

```
>>> from transformers import AutoModelForSequenceClassification
```

```
>>> model =  
AutoModelForSequenceClassification.from_pretrained("distilbert/distilbert-base-uncased")
```

2. [TrainingArguments](#) contains the model hyperparameters you can change like learning rate, batch size, and the number of epochs to train for. The default values are used if you don't specify any training arguments:

```
>>> from transformers import TrainingArguments
```

```
>>> training_args = TrainingArguments(
...     output_dir="path/to/save/folder/",
...     learning_rate=2e-5,
...     per_device_train_batch_size=8,
...     per_device_eval_batch_size=8,
...     num_train_epochs=2,
... )
```

3. Load a preprocessing class like a tokenizer, image processor, feature extractor, or processor:

```
>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("distilbert/distilbert-
base-uncased")
```

4. Load a dataset:

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("rotten_tomatoes") # doctest:
+IGNORE_RESULT
```

5. Create a function to tokenize the dataset:

```
>>> def tokenize_dataset(dataset):
...     return tokenizer(dataset["text"])
```

Then apply it over the entire dataset with [map](#):

```
>>> dataset = dataset.map(tokenize_dataset, batched=True)
```

6. A [DataCollatorWithPadding](#) to create a batch of examples from your dataset:

```
>>> from transformers import DataCollatorWithPadding

>>> data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Now gather all these classes in [Trainer](#):

```
>>> from transformers import Trainer

>>> trainer = Trainer(
```

```

...     model=model,
...     args=training_args,
...     train_dataset=dataset["train"],
...     eval_dataset=dataset["test"],
...     tokenizer=tokenizer,
...     data_collator=data_collator,
... ) # doctest: +SKIP

```

When you're ready, call [train\(\)](#) to start training:

```
>>> trainer.train()
```

For tasks - like translation or summarization - that use a sequence-to-sequence model, use the [Seq2SeqTrainer](#) and [Seq2SeqTrainingArguments](#) classes instead.

You can customize the training loop behavior by subclassing the methods inside [Trainer](#). This allows you to customize features such as the loss function, optimizer, and scheduler. Take a look at the [Trainer](#) reference for which methods can be subclassed.

The other way to customize the training loop is by using [Callbacks](#). You can use callbacks to integrate with other libraries and inspect the training loop to report on progress or stop the training early. Callbacks do not modify anything in the training loop itself. To customize something like the loss function, you need to subclass the [Trainer](#) instead.

## Train with TensorFlow

All models are a standard `tf.keras.Model` so they can be trained in TensorFlow with the [Keras](#) API. 🤗 Transformers provides the [prepare\\_tf\\_dataset\(\)](#) method to easily load your dataset as a `tf.data.Dataset` so you can start training right away with Keras' `compile` and `fit` methods.

1. You'll start with a [TFPreTrainedModel](#) or a `tf.keras.Model`:

```

>>> from transformers import TFAutoModelForSequenceClassification

>>> model =
TFAutoModelForSequenceClassification.from_pretrained("distilbert/distilbert-base-uncased")

```

2. Load a preprocessing class like a tokenizer, image processor, feature extractor, or processor:

```

>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("distilbert/distilbert-base-uncased")

```

3. Create a function to tokenize the dataset:

```
>>> def tokenize_dataset(dataset):  
...     return tokenizer(dataset["text"]) # doctest: +SKIP
```

4. Apply the tokenizer over the entire dataset with [map](#) and then pass the dataset and tokenizer to [prepare\\_tf\\_dataset\(\)](#). You can also change the batch size and shuffle the dataset here if you'd like:

```
>>> dataset = dataset.map(tokenize_dataset) # doctest: +SKIP  
>>> tf_dataset = model.prepare_tf_dataset(  
...     dataset["train"], batch_size=16, shuffle=True,  
tokenizer=tokenizer  
... ) # doctest: +SKIP
```

5. When you're ready, you can call `compile` and `fit` to start training. Note that Transformers models all have a default task-relevant loss function, so you don't need to specify one unless you want to:

```
>>> from tensorflow.keras.optimizers import Adam  
  
>>> model.compile(optimizer='adam') # No loss argument!  
>>> model.fit(tf_dataset) # doctest: +SKIP
```

## Installation

Install 🤗 Transformers for whichever deep learning library you're working with, setup your cache, and optionally configure 🤗 Transformers to run offline.

🤗 Transformers is tested on Python 3.6+, PyTorch 1.1.0+, TensorFlow 2.0+, and Flax. Follow the installation instructions below for the deep learning library you are using:

- [PyTorch](#) installation instructions.
- [TensorFlow 2.0](#) installation instructions.
- [Flax](#) installation instructions.

### Install with pip

You should install 🤗 Transformers in a [virtual environment](#). If you're unfamiliar with Python virtual environments, take a look at this [guide](#). A virtual environment makes it easier to manage different projects, and avoid compatibility issues between dependencies. Start by creating a virtual environment in your project directory:

```
python -m venv .env
```

Activate the virtual environment. On Linux and MacOS:

```
source .env/bin/activate
```

Activate Virtual environment on Windows

```
.env/Scripts/activate
```

Now you're ready to install 🤗 Transformers with the following command:

```
pip install transformers
```

For CPU-support only, you can conveniently install 🤗 Transformers and a deep learning library in one line. For example, install 🤗 Transformers and PyTorch with:

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

🤗 Transformers and TensorFlow 2.0:

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

M1 / ARM Users

You will need to install the following before installing TensorFlow 2.0

```
brew install cmake
```

```
brew install pkg-config
```

🤗 Transformers and Flax:

```
pip install 'transformers[flax]'
```

Finally, check if 🤗 Transformers has been properly installed by running the following command. It will download a pretrained model:

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

Then print out the label and score:

```
[{'label': 'POSITIVE', 'score': 0.9998704791069031}]
```

### Install from source

Install 🤗 Transformers from source with the following command:

```
pip install git+https://github.com/huggingface/transformers
```

This command installs the bleeding edge main version rather than the latest stable version. The main version is useful for staying up-to-date with the latest developments. For instance, if a bug has been fixed since the last official release but a new release hasn't been rolled out yet. However, this means the main version may not always be stable. We strive to keep the main version operational, and most issues are usually resolved within a few hours or a day. If you run into a problem, please open an [Issue](#) so we can fix it even sooner!

Check if 🤗 Transformers has been properly installed by running the following command:

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

### Editable install

You will need an editable install if you'd like to:

- Use the main version of the source code.
- Contribute to 🤗 Transformers and need to test changes in the code.

Clone the repository and install 🤗 Transformers with the following commands:

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

These commands will link the folder you cloned the repository to and your Python library paths. Python will now look inside the folder you cloned to in addition to the normal library paths. For example, if your Python packages are typically installed in `~/anaconda3/envs/main/lib/python3.7/site-packages/`, Python will also search the folder you cloned to: `~/transformers/`.

You must keep the transformers folder if you want to keep using the library.

Now you can easily update your clone to the latest version of 🤗 Transformers with the following command:

```
cd ~/transformers/
git pull
```

Your Python environment will find the main version of 🤗 Transformers on the next run.

### Install with conda

Install from the conda channel conda-forge:

```
conda install conda-forge::transformers
```

### Cache setup

Pretrained models are downloaded and locally cached at: `~/.cache/huggingface/hub`. This is the default directory given by the shell environment variable `TRANSFORMERS_CACHE`. On Windows, the default directory is given by `C:\Users\username\.cache\huggingface\hub`. You can change the shell environment variables shown below - in order of priority - to specify a different cache directory:

1. Shell environment variable (default): `HUGGINGFACE_HUB_CACHE` or `TRANSFORMERS_CACHE`.
2. Shell environment variable: `HF_HOME`.
3. Shell environment variable: `XDG_CACHE_HOME` + `/huggingface`.



😊 Transformers will use the shell environment variables `PYTORCH_TRANSFORMERS_CACHE` or `PYTORCH_PRETRAINED_BERT_CACHE` if you are coming from an earlier iteration of this library and have set those environment variables, unless you specify the shell environment variable `TRANSFORMERS_CACHE`.

### Offline mode

Run 😊 Transformers in a firewalled or offline environment with locally cached files by setting the environment variable `HF_HUB_OFFLINE=1`.

Add 🤖 [Datasets](#) to your offline training workflow with the environment variable `HF_DATASETS_OFFLINE=1`.

```
HF_DATASETS_OFFLINE=1 HF_HUB_OFFLINE=1 \
python examples/pytorch/translation/run_translation.py --model_name_or_path google-t5/t5-small --dataset_name wmt16 --dataset_config ro-en ...
```

This script should run without hanging or waiting to timeout because it won't attempt to download the model from the Hub.

You can also bypass loading a model from the Hub from each `from_pretrained()` call with the `local_files_only` parameter. When set to `True`, only local files are loaded:

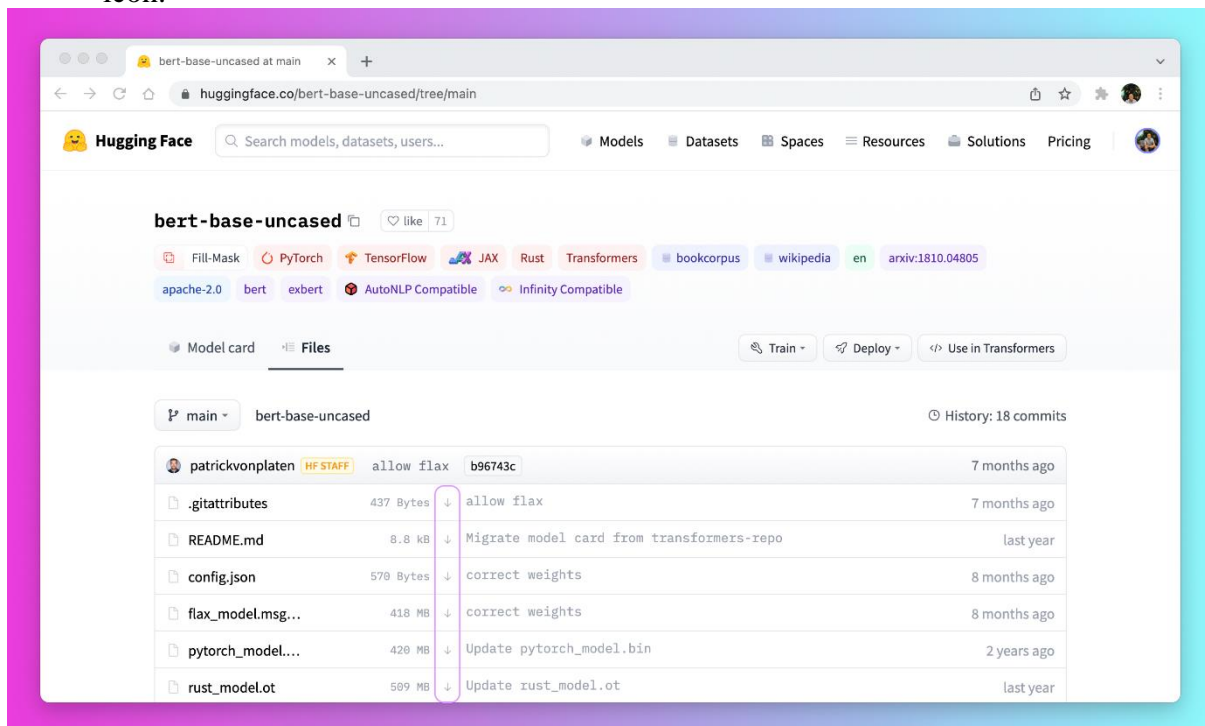
```
from transformers import T5Model
```

```
model = T5Model.from_pretrained("./path/to/local/directory", local_files_only=True)
```

### Fetch models and tokenizers to use offline

Another option for using 😊 Transformers offline is to download the files ahead of time, and then point to their local path when you need to use them offline. There are three ways to do this:

- Download a file through the user interface on the [Model Hub](#) by clicking on the ↓ icon.



- Use the [PreTrainedModel.from\\_pretrained\(\)](#) and [PreTrainedModel.save\\_pretrained\(\)](#) workflow:
  1. Download your files ahead of time with [PreTrainedModel.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
```

```
>>> tokenizer = AutoTokenizer.from_pretrained("bigscience/T0_3B")
```

```
>>> model = AutoModelForSeq2SeqLM.from_pretrained("bigscience/T0_3B")
```

2. Save your files to a specified directory with [PreTrainedModel.save\\_pretrained\(\)](#):

```
>>> tokenizer.save_pretrained("./your/path/bigscience_t0")
```

```
>>> model.save_pretrained("./your/path/bigscience_t0")
```

3. Now when you're offline, reload your files with [PreTrainedModel.from\\_pretrained\(\)](#) from the specified directory:

```
>>> tokenizer = AutoTokenizer.from_pretrained("./your/path/bigscience_t0")
```

```
>>> model = AutoModel.from_pretrained("./your/path/bigscience_t0")
```

- Programmatically download files with the [huggingface\\_hub](#) library:
  1. Install the huggingface\_hub library in your virtual environment:

```
python -m pip install huggingface_hub
```

2. Use the [hf\\_hub\\_download](#) function to download a file to a specific path. For example, the following command downloads the config.json file from the [T0](#) model to your desired path:

```
>>> from huggingface_hub import hf_hub_download
```

```
>>> hf_hub_download(repo_id="bigscience/T0_3B", filename="config.json",
cache_dir="./your/path/bigscience_t0")
```

Once your file is downloaded and locally cached, specify it's local path to load and use it:

```
>>> from transformers import AutoConfig
```

```
>>> config = AutoConfig.from_pretrained("./your/path/bigscience_t0/config.json")
```

## How to add a model to 🤗 Transformers?

The 🤗 Transformers library is often able to offer new models thanks to community contributors. But this can be a challenging project and requires an in-depth knowledge of the 🤗 Transformers library and the model to implement. At Hugging Face, we're trying to empower more of the community to actively add models and we've put together this guide to walk you through the process of adding a PyTorch model (make sure you have [PyTorch installed](#)).

Along the way, you'll:

- get insights into open-source best practices
- understand the design principles behind one of the most popular deep learning libraries
- learn how to efficiently test large models
- learn how to integrate Python utilities like black, ruff, and make fix-copies to ensure clean and readable code

A Hugging Face team member will be available to help you along the way so you'll never be alone. 🤗 ❤️

To get started, open a [New model addition](#) issue for the model you want to see in 🤗 Transformers. If you're not especially picky about contributing a specific model, you can filter by the [New model label](#) to see if there are any unclaimed model requests and work on it. Once you've opened a new model request, the first step is to get familiar with 🤗 Transformers if you aren't already!

### General overview of 🤗 Transformers

First, you should get a general overview of 🤗 Transformers. 🤗 Transformers is a very opinionated library, so there is a chance that you don't agree with some of the library's philosophies or design choices. From our experience, however, we found that the fundamental design choices and philosophies of the library are crucial to efficiently scale 🤗 Transformers while keeping maintenance costs at a reasonable level.

A good first starting point to better understand the library is to read the [documentation of our philosophy](#). As a result of our way of working, there are some choices that we try to apply to all models:

- Composition is generally favored over-abstraction
- Duplicating code is not always bad if it strongly improves the readability or accessibility of a model
- Model files are as self-contained as possible so that when you read the code of a specific model, you ideally only have to look into the respective modeling\_....py file.

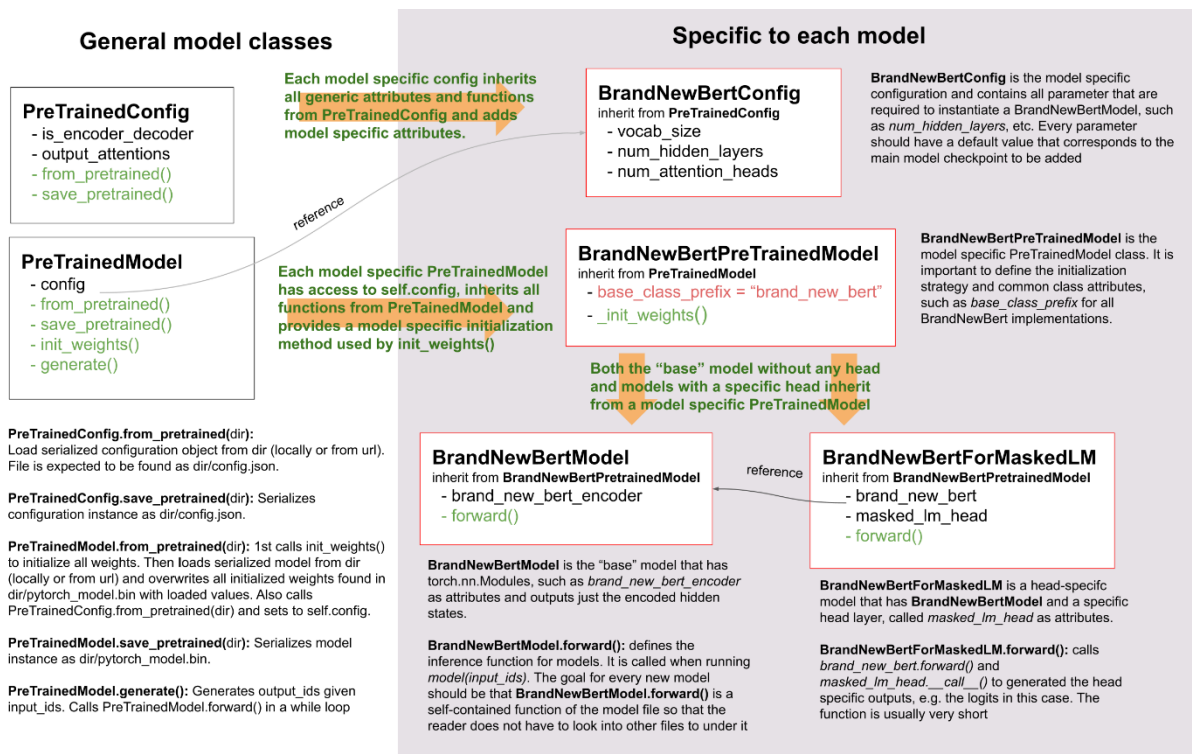
In our opinion, the library's code is not just a means to provide a product, *e.g.* the ability to use BERT for inference, but also as the very product that we want to improve. Hence, when adding a model, the user is not only the person who will use your model, but also everybody who will read, try to understand, and possibly tweak your code.

With this in mind, let's go a bit deeper into the general library design.

### Overview of models

To successfully add a model, it is important to understand the interaction between your model and its config, [PreTrainedModel](#), and [PretrainedConfig](#). For exemplary purposes, we will call the model to be added to 🤗 Transformers BrandNewBert.

Let's take a look:



As you can see, we do make use of inheritance in 😊 Transformers, but we keep the level of abstraction to an absolute minimum. There are never more than two levels of abstraction for any model in the library. **BrandNewBertModel** inherits from **BrandNewBertPreTrainedModel** which in turn inherits from **PreTrainedModel** and that's it. As a general rule, we want to make sure that a new model only depends on **PreTrainedModel**. The important functionalities that are automatically provided to every new model are **from\_pretrained()** and **save\_pretrained()**, which are used for serialization and deserialization. All of the other important functionalities, such as **BrandNewBertModel.forward** should be completely defined in the new modeling\_brand\_new\_bert.py script. Next, we want to make sure that a model with a specific head layer, such as **BrandNewBertForMaskedLM** does not inherit from **BrandNewBertModel**, but rather uses **BrandNewBertModel** as a component that can be called in its forward pass to keep the level of abstraction low. Every new model requires a configuration class, called **BrandNewBertConfig**. This configuration is always stored as an attribute in **PreTrainedModel**, and thus can be accessed via the config attribute for all classes inheriting from **BrandNewBertPreTrainedModel**:

```
model = BrandNewBertModel.from_pretrained("brandy/brand_new_bert")
model.config # model has access to its config
```

Similar to the model, the configuration inherits basic serialization and deserialization functionalities from **PretrainedConfig**. Note that the configuration and the model are always serialized into two different formats - the model to a **pytorch\_model.bin** file and the configuration to a **config.json** file. Calling the model's **save\_pretrained()** will automatically call the config's **save\_pretrained()**, so that both model and configuration are saved.

### Code style

When coding your new model, keep in mind that Transformers is an opinionated library and we have a few quirks of our own regarding how code should be written :-)

1. The forward pass of your model should be fully written in the modeling file while being fully independent of other models in the library. If you want to reuse a block

from another model, copy the code and paste it with a # from comment on top (see [here](#) for a good example and [there](#) for more documentation on from).

2. The code should be fully understandable, even by a non-native English speaker. This means you should pick descriptive variable names and avoid abbreviations. As an example, activation is preferred to act. One-letter variable names are strongly discouraged unless it's an index in a for loop.
3. More generally we prefer longer explicit code to short magical one.
4. Avoid subclassing `nn.Sequential` in PyTorch but subclass `nn.Module` and write the forward pass, so that anyone using your code can quickly debug it by adding print statements or breaking points.
5. Your function signature should be type-annotated. For the rest, good variable names are way more readable and understandable than type annotations.

### Overview of tokenizers

Not quite ready yet :-( This section will be added soon!

### Step-by-step recipe to add a model to 🤗 Transformers

Everyone has different preferences of how to port a model so it can be very helpful for you to take a look at summaries of how other contributors ported models to Hugging Face. Here is a list of community blog posts on how to port a model:

1. [Porting GPT2 Model](#) by [Thomas](#)
2. [Porting WMT19 MT Model](#) by [Stas](#)

From experience, we can tell you that the most important things to keep in mind when adding a model are:

- Don't reinvent the wheel! Most parts of the code you will add for the new 🤗 Transformers model already exist somewhere in 🤗 Transformers. Take some time to find similar, already existing models and tokenizers you can copy from. [grep](#) and [rg](#) are your friends. Note that it might very well happen that your model's tokenizer is based on one model implementation, and your model's modeling code on another one. *E.g.* FSMT's modeling code is based on BART, while FSMT's tokenizer code is based on XLM.
- It's more of an engineering challenge than a scientific challenge. You should spend more time creating an efficient debugging environment rather than trying to understand all theoretical aspects of the model in the paper.
- Ask for help, when you're stuck! Models are the core component of 🤗 Transformers so we at Hugging Face are more than happy to help you at every step to add your model. Don't hesitate to ask if you notice you are not making progress.

In the following, we try to give you a general recipe that we found most useful when porting a model to 🤗 Transformers.

The following list is a summary of everything that has to be done to add a model and can be used by you as a To-Do List:

- ☐ (Optional) Understood the model's theoretical aspects
- ☐ Prepared 🤗 Transformers dev environment
- ☐ Set up debugging environment of the original repository
- ☐ Created script that successfully runs the forward() pass using the original repository and checkpoint
- ☐ Successfully added the model skeleton to 🤗 Transformers
- ☐ Successfully converted original checkpoint to 🤗 Transformers checkpoint
- ☐ Successfully ran forward() pass in 🤗 Transformers that gives identical output to original checkpoint

- ☐ Finished model tests in 😊 Transformers
- ☐ Successfully added tokenizer in 😊 Transformers
- ☐ Run end-to-end integration tests
- ☐ Finished docs
- ☐ Uploaded model weights to the Hub
- ☐ Submitted the pull request
- ☐ (Optional) Added a demo notebook

To begin with, we usually recommend starting by getting a good theoretical understanding of BrandNewBert. However, if you prefer to understand the theoretical aspects of the model *on-the-job*, then it is totally fine to directly dive into the BrandNewBert's code-base. This option might suit you better if your engineering skills are better than your theoretical skill, if you have trouble understanding BrandNewBert's paper, or if you just enjoy programming much more than reading scientific papers.

### 1. (Optional) Theoretical aspects of BrandNewBert

You should take some time to read *BrandNewBert's* paper, if such descriptive work exists. There might be large sections of the paper that are difficult to understand. If this is the case, this is fine - don't worry! The goal is not to get a deep theoretical understanding of the paper, but to extract the necessary information required to effectively re-implement the model in 😊 Transformers. That being said, you don't have to spend too much time on the theoretical aspects, but rather focus on the practical ones, namely:

- What type of model is *brand\_new\_bert*? BERT-like encoder-only model? GPT2-like decoder-only model? BART-like encoder-decoder model? Look at the [model summary](#) if you're not familiar with the differences between those.
- What are the applications of *brand\_new\_bert*? Text classification? Text generation? Seq2Seq tasks, *e.g.*, summarization?
- What is the novel feature of the model that makes it different from BERT/GPT-2/BART?
- Which of the already existing [🤖 Transformers models](#) is most similar to *brand\_new\_bert*?
- What type of tokenizer is used? A sentencepiece tokenizer? Word piece tokenizer? Is it the same tokenizer as used for BERT or BART?

After you feel like you have gotten a good overview of the architecture of the model, you might want to write to the Hugging Face team with any questions you might have. This might include questions regarding the model's architecture, its attention layer, etc. We will be more than happy to help you.

### 2. Next prepare your environment

1. Fork the [repository](#) by clicking on the 'Fork' button on the repository's page. This creates a copy of the code under your GitHub user account.
2. Clone your transformers fork to your local disk, and add the base repository as a remote:

```
git clone https://github.com/[your Github handle]/transformers.git
cd transformers
```

```
git remote add upstream https://github.com/huggingface/transformers.git
```

3. Set up a development environment, for instance by running the following command:

```
python -m venv .env
source .env/bin/activate
pip install -e "[dev]"
```



Depending on your OS, and since the number of optional dependencies of Transformers is growing, you might get a failure with this command. If that's the case make sure to install the Deep Learning framework you are working with (PyTorch, TensorFlow and/or Flax) then do:

```
pip install -e "[quality]"
```

which should be enough for most use cases. You can then return to the parent directory

```
cd ..
```

4. We recommend adding the PyTorch version of *brand\_new\_bert* to Transformers. To install PyTorch, please follow the instructions on <https://pytorch.org/get-started/locally/>.

**Note:** You don't need to have CUDA installed. Making the new model work on CPU is sufficient.

5. To port *brand\_new\_bert*, you will also need access to its original repository:

```
git clone https://github.com/org_that_created_brand_new_bert_org/brand_new_bert.git
```

```
cd brand_new_bert
```

```
pip install -e .
```

Now you have set up a development environment to port *brand\_new\_bert* to 😊 Transformers.

### 3.-4. Run a pretrained checkpoint using the original repository

At first, you will work on the original *brand\_new\_bert* repository. Often, the original implementation is very "researchy". Meaning that documentation might be lacking and the code can be difficult to understand. But this should be exactly your motivation to reimplement *brand\_new\_bert*. At Hugging Face, one of our main goals is to *make people stand on the shoulders of giants* which translates here very well into taking a working model and rewriting it to make it as **accessible, user-friendly, and beautiful** as possible. This is the number-one motivation to re-implement models into 😊 Transformers - trying to make complex new NLP technology accessible to **everybody**.

You should start thereby by diving into the original repository.

Successfully running the official pretrained model in the original repository is often **the most difficult** step. From our experience, it is very important to spend some time getting familiar with the original code-base. You need to figure out the following:

- Where to find the pretrained weights?
- How to load the pretrained weights into the corresponding model?
- How to run the tokenizer independently from the model?
- Trace one forward pass so that you know which classes and functions are required for a simple forward pass. Usually, you only have to reimplement those functions.
- Be able to locate the important components of the model: Where is the model's class? Are there model sub-classes, *e.g.* EncoderModel, DecoderModel? Where is the self-attention layer? Are there multiple different attention layers, *e.g.* *self-attention*, *cross-attention*...?
- How can you debug the model in the original environment of the repo? Do you have to add *print* statements, can you work with an interactive debugger like *ipdb*, or should you use an efficient IDE to debug the model, like PyCharm?

It is very important that before you start the porting process, you can **efficiently** debug code in the original repository! Also, remember that you are working with an open-source library, so do not hesitate to open an issue, or even a pull request in the original repository. The maintainers of this repository are most likely very happy about someone looking into their code!

At this point, it is really up to you which debugging environment and strategy you prefer to use to debug the original model. We strongly advise against setting up a costly GPU environment, but simply work on a CPU both when starting to dive into the original repository and also when starting to write the 😊 Transformers implementation of the model.

Only at the very end, when the model has already been successfully ported to 😊

Transformers, one should verify that the model also works as expected on GPU.

In general, there are two possible debugging environments for running the original model

- [Jupyter notebooks](#) / [google colab](#)
- Local python scripts.

Jupyter notebooks have the advantage that they allow for cell-by-cell execution which can be helpful to better split logical components from one another and to have faster debugging cycles as intermediate results can be stored. Also, notebooks are often easier to share with other contributors, which might be very helpful if you want to ask the Hugging Face team for help. If you are familiar with Jupyter notebooks, we strongly recommend you work with them.

The obvious disadvantage of Jupyter notebooks is that if you are not used to working with them you will have to spend some time adjusting to the new programming environment and you might not be able to use your known debugging tools anymore, like ipdb.

For each code-base, a good first step is always to load a **small** pretrained checkpoint and to be able to reproduce a single forward pass using a dummy integer vector of input IDs as an input. Such a script could look like this (in pseudocode):

```
model = BrandNewBertModel.load_pretrained_checkpoint("/path/to/checkpoint/")
```

```
input_ids = [0, 4, 5, 2, 3, 7, 9] # vector of input ids
```

```
original_output = model.predict(input_ids)
```

Next, regarding the debugging strategy, there are generally a few from which to choose from:

- Decompose the original model into many small testable components and run a forward pass on each of those for verification
- Decompose the original model only into the original *tokenizer* and the original *model*, run a forward pass on those, and use intermediate print statements or breakpoints for verification

Again, it is up to you which strategy to choose. Often, one or the other is advantageous depending on the original code base.

If the original code-base allows you to decompose the model into smaller sub-components, *e.g.* if the original code-base can easily be run in eager mode, it is usually worth the effort to do so. There are some important advantages to taking the more difficult road in the beginning:

- at a later stage when comparing the original model to the Hugging Face implementation, you can verify automatically for each component individually that the corresponding component of the 😊 Transformers implementation matches instead of relying on visual comparison via print statements
- it can give you some rope to decompose the big problem of porting a model into smaller problems of just porting individual components and thus structure your work better
- separating the model into logical meaningful components will help you to get a better overview of the model's design and thus to better understand the model
- at a later stage those component-by-component tests help you to ensure that no regression occurs as you continue changing your code

[Lysandre's](#) integration checks for ELECTRA gives a nice example of how this can be done.



However, if the original code-base is very complex or only allows intermediate components to be run in a compiled mode, it might be too time-consuming or even impossible to separate the model into smaller testable sub-components. A good example is [TS's MeshTensorFlow](#) library which is very complex and does not offer a simple way to decompose the model into its sub-components. For such libraries, one often relies on verifying print statements.

No matter which strategy you choose, the recommended procedure is often the same that you should start to debug the starting layers first and the ending layers last.

It is recommended that you retrieve the output, either by print statements or sub-component functions, of the following layers in the following order:

1. Retrieve the input IDs passed to the model
2. Retrieve the word embeddings
3. Retrieve the input of the first Transformer layer
4. Retrieve the output of the first Transformer layer
5. Retrieve the output of the following  $n - 1$  Transformer layers
6. Retrieve the output of the whole BrandNewBert Model

Input IDs should thereby consists of an array of integers, *e.g.* `input_ids = [0, 4, 4, 3, 2, 4, 1, 7, 19]`

The outputs of the following layers often consist of multi-dimensional float arrays and can look like this:

```
[[
[-0.1465, -0.6501, 0.1993, ..., 0.1451, 0.3430, 0.6024],
[-0.4417, -0.5920, 0.3450, ..., -0.3062, 0.6182, 0.7132],
[-0.5009, -0.7122, 0.4548, ..., -0.3662, 0.6091, 0.7648],
...,
[-0.5613, -0.6332, 0.4324, ..., -0.3792, 0.7372, 0.9288],
[-0.5416, -0.6345, 0.4180, ..., -0.3564, 0.6992, 0.9191],
[-0.5334, -0.6403, 0.4271, ..., -0.3339, 0.6533, 0.8694]]],
```

We expect that every model added to 🤗 Transformers passes a couple of integration tests, meaning that the original model and the reimplemented version in 🤗 Transformers have to give the exact same output up to a precision of 0.001! Since it is normal that the exact same model written in different libraries can give a slightly different output depending on the library framework, we accept an error tolerance of  $1e-3$  (0.001). It is not enough if the model gives nearly the same output, they have to be almost identical. Therefore, you will certainly compare the intermediate outputs of the 🤗 Transformers version multiple times against the intermediate outputs of the original implementation of *brand\_new\_bert* in which case an **efficient** debugging environment of the original repository is absolutely important. Here is some advice to make your debugging environment as efficient as possible.

- Find the best way of debugging intermediate results. Is the original repository written in PyTorch? Then you should probably take the time to write a longer script that decomposes the original model into smaller sub-components to retrieve intermediate values. Is the original repository written in Tensorflow 1? Then you might have to rely on TensorFlow print operations like [tf.print](#) to output intermediate values. Is the original repository written in Jax? Then make sure that the model is **not jitted** when running the forward pass, *e.g.* check-out [this link](#).
- Use the smallest pretrained checkpoint you can find. The smaller the checkpoint, the faster your debug cycle becomes. It is not efficient if your pretrained model is so big that your forward pass takes more than 10 seconds. In case only very large checkpoints are available, it might make more sense to create a dummy model in the

new environment with randomly initialized weights and save those weights for comparison with the 🤗 Transformers version of your model

- Make sure you are using the easiest way of calling a forward pass in the original repository. Ideally, you want to find the function in the original repository that **only** calls a single forward pass, *i.e.* that is often called `predict`, `evaluate`, `forward` or `__call__`. You don't want to debug a function that calls forward multiple times, *e.g.* to generate text, like `autoregressive_sample`, `generate`.
- Try to separate the tokenization from the model's *forward* pass. If the original repository shows examples where you have to input a string, then try to find out where in the forward call the string input is changed to input ids and start from this point. This might mean that you have to possibly write a small script yourself or change the original code so that you can directly input the ids instead of an input string.
- Make sure that the model in your debugging setup is **not** in training mode, which often causes the model to yield random outputs due to multiple dropout layers in the model. Make sure that the forward pass in your debugging environment is **deterministic** so that the dropout layers are not used. Or use `transformers.utils.set_seed` if the old and new implementations are in the same framework.

The following section gives you more specific details/tips on how you can do this for `brand_new_bert`.

#### 5.-14. Port BrandNewBert to 🤗 Transformers

Next, you can finally start adding new code to 🤗 Transformers. Go into the clone of your 🤗 Transformers' fork:

```
cd transformers
```

In the special case that you are adding a model whose architecture exactly matches the model architecture of an existing model you only have to add a conversion script as described in [this section](#). In this case, you can just re-use the whole model architecture of the already existing model.

Otherwise, let's start generating a new model. We recommend using the following script to add a model starting from an existing model:

```
transformers-cli add-new-model-like
```

You will be prompted with a questionnaire to fill in the basic information of your model.

#### Open a Pull Request on the main huggingface/transformers repo

Before starting to adapt the automatically generated code, now is the time to open a "Work in progress (WIP)" pull request, *e.g.* "[WIP] Add `brand_new_bert`", in 🤗 Transformers so that you and the Hugging Face team can work side-by-side on integrating the model into 🤗 Transformers.

You should do the following:

1. Create a branch with a descriptive name from your main branch

```
git checkout -b add_brand_new_bert
```

2. Commit the automatically generated code:

```
git add .
```

```
git commit
```

### 3. Fetch and rebase to current main

```
git fetch upstream
```

```
git rebase upstream/main
```

### 4. Push the changes to your account using:

```
git push -u origin a-descriptive-name-for-my-changes
```

5. Once you are satisfied, go to the webpage of your fork on GitHub. Click on “Pull request”. Make sure to add the GitHub handle of some members of the Hugging Face team as reviewers, so that the Hugging Face team gets notified for future changes.
6. Change the PR into a draft by clicking on “Convert to draft” on the right of the GitHub pull request web page.

In the following, whenever you have made some progress, don’t forget to commit your work and push it to your account so that it shows in the pull request. Additionally, you should make sure to update your work with the current main from time to time by doing:

```
git fetch upstream
```

```
git merge upstream/main
```

In general, all questions you might have regarding the model or your implementation should be asked in your PR and discussed/solved in the PR. This way, the Hugging Face team will always be notified when you are committing new code or if you have a question. It is often very helpful to point the Hugging Face team to your added code so that the Hugging Face team can efficiently understand your problem or question.

To do so, you can go to the “Files changed” tab where you see all of your changes, go to a line regarding which you want to ask a question, and click on the “+” symbol to add a comment. Whenever a question or problem has been solved, you can click on the “Resolve” button of the created comment.

In the same way, the Hugging Face team will open comments when reviewing your code. We recommend asking most questions on GitHub on your PR. For some very general questions that are not very useful for the public, feel free to ping the Hugging Face team by Slack or email.

## 5. Adapt the generated models code for brand\_new\_bert

At first, we will focus only on the model itself and not care about the tokenizer. All the relevant code should be found in the generated

files `src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` and `src/transformers/models/brand_new_bert/configuration_brand_new_bert.py`.

Now you can finally start coding :). The generated code

in `src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` will either have the same architecture as BERT if it’s an encoder-only model or BART if it’s an encoder-decoder model. At this point, you should remind yourself what you’ve learned in the beginning about the theoretical aspects of the model: *How is the model different from BERT or BART?*. Implement those changes which often means changing the *self-attention* layer, the order of the normalization layer, etc... Again, it is often useful to look at the similar architecture of already existing models in Transformers to get a better feeling of how your model should be implemented.

**Note** that at this point, you don’t have to be very sure that your code is fully correct or clean. Rather, it is advised to add a first *unclean*, copy-pasted version of the original code to `src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` until you feel like all the necessary code is added. From our experience, it is much more efficient to quickly add a first version of the required code and improve/correct the code iteratively with the

conversion script as described in the next section. The only thing that has to work at this point is that you can instantiate the 🤖 Transformers implementation of *brand\_new\_bert*, i.e. the following command should work:

```
from transformers import BrandNewBertModel, BrandNewBertConfig
```

```
model = BrandNewBertModel(BrandNewBertConfig())
```

The above command will create a model according to the default parameters as defined in `BrandNewBertConfig()` with random weights, thus making sure that the `init()` methods of all components works.

Note that all random initialization should happen in the `_init_weights` method of your `BrandnewBertPreTrainedModel` class. It should initialize all leaf modules depending on the variables of the config. Here is an example with the BERT `_init_weights` method:

```
def _init_weights(self, module):
    """Initialize the weights"""
    if isinstance(module, nn.Linear):
        module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if module.bias is not None:
            module.bias.data.zero_()
    elif isinstance(module, nn.Embedding):
        module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if module.padding_idx is not None:
            module.weight.data[module.padding_idx].zero_()
    elif isinstance(module, nn.LayerNorm):
        module.bias.data.zero_()
        module.weight.data.fill_(1.0)
```

You can have some more custom schemes if you need a special initialization for some modules. For instance, in `Wav2Vec2ForPreTraining`, the last two linear layers need to have the initialization of the regular PyTorch `nn.Linear` but all the other ones should use an initialization as above. This is coded like this:

```
def _init_weights(self, module):
    """Initialize the weights"""
    if isinstance(module, Wav2Vec2ForPreTraining):
        module.project_hid.reset_parameters()
        module.project_q.reset_parameters()
        module.project_hid._is_hf_initialized = True
        module.project_q._is_hf_initialized = True
    elif isinstance(module, nn.Linear):
        module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if module.bias is not None:
            module.bias.data.zero_()
```

The `_is_hf_initialized` flag is internally used to make sure we only initialize a submodule once. By setting it to `True` for `module.project_q` and `module.project_hid`, we make sure the custom initialization we did is not overridden later on, the `_init_weights` function won't be applied to them.

## 6. Write a conversion script

Next, you should write a conversion script that lets you convert the checkpoint you used to debug *brand\_new\_bert* in the original repository to a checkpoint compatible with your just

created 😊 Transformers implementation of *brand\_new\_bert*. It is not advised to write the conversion script from scratch, but rather to look through already existing conversion scripts in 😊 Transformers for one that has been used to convert a similar model that was written in the same framework as *brand\_new\_bert*. Usually, it is enough to copy an already existing conversion script and slightly adapt it for your use case. Don't hesitate to ask the Hugging Face team to point you to a similar already existing conversion script for your model.

- If you are porting a model from TensorFlow to PyTorch, a good starting point might be BERT's conversion script [here](#)
- If you are porting a model from PyTorch to PyTorch, a good starting point might be BART's conversion script [here](#)

In the following, we'll quickly explain how PyTorch models store layer weights and define layer names. In PyTorch, the name of a layer is defined by the name of the class attribute you give the layer. Let's define a dummy model in PyTorch, called SimpleModel as follows:

```
from torch import nn
```

```
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.dense = nn.Linear(10, 10)
        self.intermediate = nn.Linear(10, 10)
        self.layer_norm = nn.LayerNorm(10)
```

Now we can create an instance of this model definition which will fill all weights: dense, intermediate, layer\_norm with random weights. We can print the model to see its architecture

```
model = SimpleModel()
```

```
print(model)
```

This will print out the following:

```
SimpleModel(
  (dense): Linear(in_features=10, out_features=10, bias=True)
  (intermediate): Linear(in_features=10, out_features=10, bias=True)
  (layer_norm): LayerNorm((10,), eps=1e-05, elementwise_affine=True)
)
```

We can see that the layer names are defined by the name of the class attribute in PyTorch. You can print out the weight values of a specific layer:

```
print(model.dense.weight.data)
```

to see that the weights were randomly initialized

```
tensor([[ -0.0818,  0.2207, -0.0749, -0.0030,  0.0045, -0.1569, -0.1598,  0.0212,
          -0.2077,  0.2157],
        [ 0.1044,  0.0201,  0.0990,  0.2482,  0.3116,  0.2509,  0.2866, -0.2190,
          0.2166, -0.0212],
        [-0.2000,  0.1107, -0.1999, -0.3119,  0.1559,  0.0993,  0.1776, -0.1950,
          -0.1023, -0.0447],
        [-0.0888, -0.1092,  0.2281,  0.0336,  0.1817, -0.0115,  0.2096,  0.1415,
```

```

-0.1876, -0.2467],
[ 0.2208, -0.2352, -0.1426, -0.2636, -0.2889, -0.2061, -0.2849, -0.0465,
 0.2577, 0.0402],
[ 0.1502, 0.2465, 0.2566, 0.0693, 0.2352, -0.0530, 0.1859, -0.0604,
 0.2132, 0.1680],
[ 0.1733, -0.2407, -0.1721, 0.1484, 0.0358, -0.0633, -0.0721, -0.0090,
 0.2707, -0.2509],
[-0.1173, 0.1561, 0.2945, 0.0595, -0.1996, 0.2988, -0.0802, 0.0407,
 0.1829, -0.1568],
[-0.1164, -0.2228, -0.0403, 0.0428, 0.1339, 0.0047, 0.1967, 0.2923,
 0.0333, -0.0536],
[-0.1492, -0.1616, 0.1057, 0.1950, -0.2807, -0.2710, -0.1586, 0.0739,
 0.2220, 0.2358]]).

```

In the conversion script, you should fill those randomly initialized weights with the exact weights of the corresponding layer in the checkpoint. *E.g.*

```

# retrieve matching layer weights, e.g. by
# recursive algorithm
layer_name = "dense"
pretrained_weight = array_of_dense_layer

```

```

model_pointer = getattr(model, "dense")

```

```

model_pointer.weight.data = torch.from_numpy(pretrained_weight)

```

While doing so, you must verify that each randomly initialized weight of your PyTorch model and its corresponding pretrained checkpoint weight exactly match in both **shape and name**. To do so, it is **necessary** to add assert statements for the shape and print out the names of the checkpoints weights. *E.g.* you should add statements like:

```

assert (
    model_pointer.weight.shape == pretrained_weight.shape
), f"Pointer shape of random weight {model_pointer.shape} and array shape of checkpoint weight {pretrained_weight.shape} mismatched"

```

Besides, you should also print out the names of both weights to make sure they match, *e.g.*

```

logger.info(f"Initialize PyTorch weight {layer_name} from {pretrained_weight.name}")

```

If either the shape or the name doesn't match, you probably assigned the wrong checkpoint weight to a randomly initialized layer of the 😊 Transformers implementation.

An incorrect shape is most likely due to an incorrect setting of the config parameters in `BrandNewBertConfig()` that do not exactly match those that were used for the checkpoint you want to convert. However, it could also be that PyTorch's implementation of a layer requires the weight to be transposed beforehand.

Finally, you should also check that **all** required weights are initialized and print out all checkpoint weights that were not used for initialization to make sure the model is correctly converted. It is completely normal, that the conversion trials fail with either a wrong shape statement or a wrong name assignment. This is most likely because either you used incorrect parameters in `BrandNewBertConfig()`, have a wrong architecture in the 😊 Transformers implementation, you have a bug in the `init()` functions of one of the components of the 😊 Transformers implementation or you need to transpose one of the checkpoint weights.

This step should be iterated with the previous step until all weights of the checkpoint are correctly loaded in the Transformers model. Having correctly loaded the checkpoint into the 😊 Transformers implementation, you can then save the model under a folder of your choice /path/to/converted/checkpoint/folder that should then contain both a pytorch\_model.bin file and a config.json file:

```
model.save_pretrained("/path/to/converted/checkpoint/folder")
```

## 7. Implement the forward pass

Having managed to correctly load the pretrained weights into the 😊 Transformers implementation, you should now make sure that the forward pass is correctly implemented. In [Get familiar with the original repository](#), you have already created a script that runs a forward pass of the model using the original repository. Now you should write an analogous script using the 😊 Transformers implementation instead of the original one. It should look as follows:

```
model = BrandNewBertModel.from_pretrained("/path/to/converted/checkpoint/folder")
input_ids = [0, 4, 4, 3, 2, 4, 1, 7, 19]
output = model(input_ids).last_hidden_states
```

It is very likely that the 😊 Transformers implementation and the original model implementation don't give the exact same output the very first time or that the forward pass throws an error. Don't be disappointed - it's expected! First, you should make sure that the forward pass doesn't throw any errors. It often happens that the wrong dimensions are used leading to a *Dimensionality mismatch* error or that the wrong data type object is used, e.g. torch.long instead of torch.float32. Don't hesitate to ask the Hugging Face team for help, if you don't manage to solve certain errors.

The final part to make sure the 😊 Transformers implementation works correctly is to ensure that the outputs are equivalent to a precision of 1e-3. First, you should ensure that the output shapes are identical, i.e. outputs.shape should yield the same value for the script of the 😊 Transformers implementation and the original implementation. Next, you should make sure that the output values are identical as well. This one of the most difficult parts of adding a new model. Common mistakes why the outputs are not identical are:

- Some layers were not added, i.e. an *activation* layer was not added, or the residual connection was forgotten
- The word embedding matrix was not tied
- The wrong positional embeddings are used because the original implementation uses on offset
- Dropout is applied during the forward pass. To fix this make sure *model.training* is *False* and that no dropout layer is falsely activated during the forward pass, i.e. pass *self.training* to [PyTorch's functional dropout](#)

The best way to fix the problem is usually to look at the forward pass of the original implementation and the 😊 Transformers implementation side-by-side and check if there are any differences. Ideally, you should debug/print out intermediate outputs of both implementations of the forward pass to find the exact position in the network where the 😊 Transformers implementation shows a different output than the original implementation. First, make sure that the hard-coded input\_ids in both scripts are identical. Next, verify that the outputs of the first transformation of the input\_ids (usually the word embeddings) are identical. And then work your way up to the very last layer of the network. At some point, you will notice a difference between the two implementations, which should point you to the bug in the 😊 Transformers implementation. From our experience, a simple and efficient



way is to add many print statements in both the original implementation and 🤗 Transformers implementation, at the same positions in the network respectively, and to successively remove print statements showing the same values for intermediate presentations. When you're confident that both implementations yield the same output, verify the outputs with `torch.allclose(original_output, output, atol=1e-3)`, you're done with the most difficult part! Congratulations - the work left to be done should be a cakewalk 😊.

## 8. Adding all necessary model tests

At this point, you have successfully added a new model. However, it is very much possible that the model does not yet fully comply with the required design. To make sure, the implementation is fully compatible with 🤗 Transformers, all common tests should pass. The Cookiecutter should have automatically added a test file for your model, probably under the same tests/models/brand\_new\_bert/test\_modeling\_brand\_new\_bert.py. Run this test file to verify that all common tests pass:

```
pytest tests/models/brand_new_bert/test_modeling_brand_new_bert.py
```

Having fixed all common tests, it is now crucial to ensure that all the nice work you have done is well tested, so that

- a) The community can easily understand your work by looking at specific tests of *brand\_new\_bert*
- b) Future changes to your model will not break any important feature of the model.

At first, integration tests should be added. Those integration tests essentially do the same as the debugging scripts you used earlier to implement the model to 🤗 Transformers. A template of those model tests has already added by the Cookiecutter, called `BrandNewBertModelIntegrationTests` and only has to be filled out by you. To ensure that those tests are passing, run

```
RUN_SLOW=1 pytest -sv
```

```
tests/models/brand_new_bert/test_modeling_brand_new_bert.py::BrandNewBertModelIntegrationTests
```

In case you are using Windows, you should replace `RUN_SLOW=1` with `SET`

```
RUN_SLOW=1
```

Second, all features that are special to *brand\_new\_bert* should be tested additionally in a separate test under `BrandNewBertModelTester/BrandNewBertModelTest`. This part is often forgotten but is extremely useful in two ways:

- It helps to transfer the knowledge you have acquired during the model addition to the community by showing how the special features of *brand\_new\_bert* should work.
- Future contributors can quickly test changes to the model by running those special tests.

## 9. Implement the tokenizer

Next, we should add the tokenizer of *brand\_new\_bert*. Usually, the tokenizer is equivalent to or very similar to an already existing tokenizer of 🤗 Transformers.

It is very important to find/extract the original tokenizer file and to manage to load this file into the 🤗 Transformers' implementation of the tokenizer.

To ensure that the tokenizer works correctly, it is recommended to first create a script in the original repository that inputs a string and returns the `input_ids`. It could look similar to this (in pseudo-code):

```
input_str = "This is a long example input string containing special characters .$?- , numbers 2872 234 12 and words."
```



```
model = BrandNewBertModel.load_pretrained_checkpoint("/path/to/checkpoint/")
input_ids = model.tokenize(input_str)
```

You might have to take a deeper look again into the original repository to find the correct tokenizer function or you might even have to do changes to your clone of the original repository to only output the `input_ids`. Having written a functional tokenization script that uses the original repository, an analogous script for 🤗 Transformers should be created. It should look similar to this:

```
from transformers import BrandNewBertTokenizer
```

```
input_str = "This is a long example input string containing special characters .${?-, numbers
2872 234 12 and words."
```

```
tokenizer = BrandNewBertTokenizer.from_pretrained("/path/to/tokenizer/folder/")
```

```
input_ids = tokenizer(input_str).input_ids
```

When both `input_ids` yield the same values, as a final step a tokenizer test file should also be added.

Analogous to the modeling test files of *brand\_new\_bert*, the tokenization test files of *brand\_new\_bert* should contain a couple of hard-coded integration tests.

## 10. Run End-to-end integration tests

Having added the tokenizer, you should also add a couple of end-to-end integration tests using both the model and the tokenizer

to `tests/models/brand_new_bert/test_modeling_brand_new_bert.py` in 🤗 Transformers.

Such a test should show on a meaningful text-to-text sample that the 🤗 Transformers implementation works as expected. A meaningful text-to-text sample can include *e.g.* a source-to-target-translation pair, an article-to-summary pair, a question-to-answer pair, etc... If none of the ported checkpoints has been fine-tuned on a downstream task it is enough to simply rely on the model tests. In a final step to ensure that the model is fully functional, it is advised that you also run all tests on GPU. It can happen that you forgot to add some `.to(self.device)` statements to internal tensors of the model, which in such a test would show in an error. In case you have no access to a GPU, the Hugging Face team can take care of running those tests for you.

## 11. Add Docstring

Now, all the necessary functionality for *brand\_new\_bert* is added - you're almost done! The only thing left to add is a nice docstring and a doc page. The Cookiecutter should have added a template file called `docs/source/model_doc/brand_new_bert.md` that you should fill out. Users of your model will usually first look at this page before using your model. Hence, the documentation must be understandable and concise. It is very useful for the community to add some *Tips* to show how the model should be used. Don't hesitate to ping the Hugging Face team regarding the docstrings.

Next, make sure that the docstring added

to `src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` is correct and included all necessary inputs and outputs. We have a detailed guide about writing documentation and our docstring format [here](#). It is always good to remind oneself that documentation should be treated at least as carefully as the code in 🤗 Transformers since the documentation is usually the first contact point of the community with the model.

## Code refactor

Great, now you have added all the necessary code for *brand\_new\_bert*. At this point, you should correct some potential incorrect code style by running:

```
make style
```

and verify that your coding style passes the quality check:

```
make quality
```

There are a couple of other very strict design tests in 😊 Transformers that might still be failing, which shows up in the tests of your pull request. This is often because of some missing information in the docstring or some incorrect naming. The Hugging Face team will surely help you if you're stuck here.

Lastly, it is always a good idea to refactor one's code after having ensured that the code works correctly. With all tests passing, now it's a good time to go over the added code again and do some refactoring.

You have now finished the coding part, congratulation! 🎉 You are Awesome! 😎

### 12. Upload the models to the model hub

In this final part, you should convert and upload all checkpoints to the model hub and add a model card for each uploaded model checkpoint. You can get familiar with the hub functionalities by reading our [Model sharing and uploading Page](#). You should work alongside the Hugging Face team here to decide on a fitting name for each checkpoint and to get the required access rights to be able to upload the model under the author's organization of *brand\_new\_bert*. The `push_to_hub` method, present in all models in transformers, is a quick and efficient way to push your checkpoint to the hub. A little snippet is pasted below:

```
brand_new_bert.push_to_hub("brand_new_bert")
# Uncomment the following line to push to an organization.
# brand_new_bert.push_to_hub("<organization>/brand_new_bert")
```

It is worth spending some time to create fitting model cards for each checkpoint. The model cards should highlight the specific characteristics of this particular checkpoint, *e.g.* On which dataset was the checkpoint pretrained/fine-tuned on? On what down-stream task should the model be used? And also include some code on how to correctly use the model.

### 13. (Optional) Add notebook

It is very helpful to add a notebook that showcases in-detail how *brand\_new\_bert* can be used for inference and/or fine-tuned on a downstream task. This is not mandatory to merge your PR, but very useful for the community.

### 14. Submit your finished PR

You're done programming now and can move to the last step, which is getting your PR merged into main. Usually, the Hugging Face team should have helped you already at this point, but it is worth taking some time to give your finished PR a nice description and eventually add comments to your code, if you want to point out certain design choices to your reviewer.

#### Share your work!!

Now, it's time to get some credit from the community for your work! Having completed a model addition is a major contribution to Transformers and the whole NLP community. Your code and the ported pre-trained models will certainly be used by hundreds and possibly even thousands of developers and researchers. You should be proud of your work and share your achievements with the community.

## Pipelines for inference

The [pipeline\(\)](#) makes it simple to use any model from the [Hub](#) for inference on any language, computer vision, speech, and multimodal tasks. Even if you don't have experience with a specific modality or aren't familiar with the underlying code behind the models, you can still use them for inference with the [pipeline\(\)](#)! This tutorial will teach you to:

- Use a [pipeline\(\)](#) for inference.
- Use a specific tokenizer or model.
- Use a [pipeline\(\)](#) for audio, vision, and multimodal tasks.

Take a look at the [pipeline\(\)](#) documentation for a complete list of supported tasks and available parameters.

### Pipeline usage

While each task has an associated [pipeline\(\)](#), it is simpler to use the general [pipeline\(\)](#) abstraction which contains all the task-specific pipelines. The [pipeline\(\)](#) automatically loads a default model and a preprocessing class capable of inference for your task. Let's take the example of using the [pipeline\(\)](#) for automatic speech recognition (ASR), or speech-to-text.

1. Start by creating a [pipeline\(\)](#) and specify the inference task:

```
>>> from transformers import pipeline
```

```
>>> transcriber = pipeline(task="automatic-speech-recognition")
```

2. Pass your input to the [pipeline\(\)](#). In the case of speech recognition, this is an audio input file:

```
>>> transcriber("https://huggingface.co/datasets/Narsil/asr_dummy/resolve/main/mlk.flac")
{'text': 'I HAVE A DREAM BUT ONE DAY THIS NATION WILL RISE UP LIVE UP THE TRUE MEANING OF ITS TREES'}
```

Not the result you had in mind? Check out some of the [most downloaded automatic speech recognition models](#) on the Hub to see if you can get a better transcription.

Let's try the [Whisper large-v2](#) model from OpenAI. Whisper was released 2 years later than Wav2Vec2, and was trained on close to 10x more data. As such, it beats Wav2Vec2 on most downstream benchmarks. It also has the added benefit of predicting punctuation and casing, neither of which are possible with Wav2Vec2.

Let's give it a try here to see how it performs:

```
>>> transcriber = pipeline(model="openai/whisper-large-v2")
>>> transcriber("https://huggingface.co/datasets/Narsil/asr_dummy/resolve/main/mlk.flac")
{'text': 'I have a dream that one day this nation will rise up and live out the true meaning of its creed.'}
```

Now this result looks more accurate! For a deep-dive comparison on Wav2Vec2 vs Whisper, refer to the [Audio Transformers Course](#). We really encourage you to check out the Hub for models in different languages, models specialized in your field, and more. You can check out and compare model results directly from your browser on the Hub to see if it fits or handles corner cases better than other ones. And if you don't find a model for your use case, you can always start [training](#) your own!

If you have several inputs, you can pass your input as a list:

```
transcriber(
    [
        "https://huggingface.co/datasets/Narsil/asr_dummy/resolve/main/mlk.flac",
```

```
    "https://huggingface.co/datasets/Narsil/asr_dummy/resolve/main/1.flac",  
    ]  
)
```

Pipelines are great for experimentation as switching from one model to another is trivial; however, there are some ways to optimize them for larger workloads than experimentation. See the following guides that dive into iterating over whole datasets or using pipelines in a webserver: of the docs:

- [Using pipelines on a dataset](#)
- [Using pipelines for a webserver](#)

### Parameters

[pipeline\(\)](#) supports many parameters; some are task specific, and some are general to all pipelines. In general, you can specify parameters anywhere you want:

```
transcriber = pipeline(model="openai/whisper-large-v2", my_parameter=1)
```

```
out = transcriber(...) # This will use `my_parameter=1`.
```

```
out = transcriber(..., my_parameter=2) # This will override and use `my_parameter=2`.
```

```
out = transcriber(...) # This will go back to using `my_parameter=1`.
```

Let's check out 3 important ones:

### Device

If you use `device=n`, the pipeline automatically puts the model on the specified device. This will work regardless of whether you are using PyTorch or Tensorflow.

```
transcriber = pipeline(model="openai/whisper-large-v2", device=0)
```

If the model is too large for a single GPU and you are using PyTorch, you can set `torch_dtype='float16'` to enable FP16 precision inference. Usually this would not cause significant performance drops but make sure you evaluate it on your models!

Alternatively, you can set `device_map="auto"` to automatically determine how to load and store the model weights. Using the `device_map` argument requires the 🤗 [Accelerate](#) package:

```
pip install --upgrade accelerate
```

The following code automatically loads and stores model weights across devices:

```
transcriber = pipeline(model="openai/whisper-large-v2", device_map="auto")
```

Note that if `device_map="auto"` is passed, there is no need to add the argument `device=device` when instantiating your pipeline as you may encounter some unexpected behavior!

### Batch size

By default, pipelines will not batch inference for reasons explained in detail [here](#). The reason is that batching is not necessarily faster, and can actually be quite slower in some cases.

But if it works in your use case, you can use:

```
transcriber = pipeline(model="openai/whisper-large-v2", device=0, batch_size=2)
```

```
audio_filenames =
```

```
[f"https://huggingface.co/datasets/Narsil/asr_dummy/resolve/main/{i}.flac" for i in range(1, 5)]
```

```
texts = transcriber(audio_filenames)
```

This runs the pipeline on the 4 provided audio files, but it will pass them in batches of 2 to the model (which is on a GPU, where batching is more likely to help) without requiring any

further code from you. The output should always match what you would have received without batching. It is only meant as a way to help you get more speed out of a pipeline. Pipelines can also alleviate some of the complexities of batching because, for some pipelines, a single item (like a long audio file) needs to be chunked into multiple parts to be processed by a model. The pipeline performs this [chunk batching](#) for you.

### Task specific parameters

All tasks provide task specific parameters which allow for additional flexibility and options to help you get your job done. For instance, the [transformers.AutomaticSpeechRecognitionPipeline.call\(\)](#) method has a `return_timestamps` parameter which sounds promising for subtitling videos:

```
>>> transcriber = pipeline(model="openai/whisper-large-v2", return_timestamps=True)
>>> transcriber("https://huggingface.co/datasets/Narsil/asr_dummy/resolve/main/mlk.flac")
{'text': 'I have a dream that one day this nation will rise up and live out the true meaning of its creed.', 'chunks': [{'timestamp': (0.0, 11.88), 'text': 'I have a dream that one day this nation will rise up and live out the true meaning of its'}, {'timestamp': (11.88, 12.38), 'text': 'creed.'}]}
```

As you can see, the model inferred the text and also outputted **when** the various sentences were pronounced.

There are many parameters available for each task, so check out each task's API reference to see what you can tinker with! For instance, the [AutomaticSpeechRecognitionPipeline](#) has a `chunk_length_s` parameter which is helpful for working on really long audio files (for example, subtitling entire movies or hour-long videos) that a model typically cannot handle on its own:

```
>>> transcriber = pipeline(model="openai/whisper-large-v2", chunk_length_s=30)
>>> transcriber("https://huggingface.co/datasets/reach-vb/random-audios/resolve/main/ted_60.wav")
{'text': "So in college, I was a government major, which means I had to write a lot of papers. Now, when a normal student writes a paper, they might spread the work out a little like this. So, you know. You get started maybe a little slowly, but you get enough done in the first week that with some heavier days later on, everything gets done and things stay civil. And I would want to do that like that. That would be the plan. I would have it all ready to go, but then actually the paper would come along, and then I would kind of do this. And that would happen every single paper. But then came my 90-page senior thesis, a paper you're supposed to spend a year on. I knew for a paper like that, my normal workflow was not an option, it was way too big a project. So I planned things out and I decided I kind of had to go something like this. This is how the year would go. So I'd start off light and I'd bump it up"}
If you can't find a parameter that would really help you out, feel free to request it!
```

### Using pipelines on a dataset

The pipeline can also run inference on a large dataset. The easiest way we recommend doing this is by using an iterator:

```
def data():
    for i in range(1000):
        yield f"My example {i}"
```

```
pipe = pipeline(model="openai-community/gpt2", device=0)
generated_characters = 0
```

```
for out in pipe(data()):
```

```
    generated_characters += len(out[0]["generated_text"])
```

The iterator `data()` yields each result, and the pipeline automatically recognizes the input is iterable and will start fetching the data while it continues to process it on the GPU (this uses [DataLoader](#) under the hood). This is important because you don't have to allocate memory for the whole dataset and you can feed the GPU as fast as possible.

Since batching could speed things up, it may be useful to try tuning the `batch_size` parameter here.

The simplest way to iterate over a dataset is to just load one from 😊 [Datasets](#):

```
# KeyDataset is a util that will just output the item we're interested in.
```

```
from transformers.pipelines.pt_utils import KeyDataset
```

```
from datasets import load_dataset
```

```
pipe = pipeline(model="hf-internal-testing/tiny-random-wav2vec2", device=0)
```

```
dataset = load_dataset("hf-internal-testing/librispeech_asr_dummy", "clean",  
split="validation[:10]")
```

```
for out in pipe(KeyDataset(dataset, "audio")):
```

```
    print(out)
```

### Using pipelines for a webserver

Creating an inference engine is a complex topic which deserves its own page.

[Link](#)

### Vision pipeline

Using a [pipeline\(\)](#) for vision tasks is practically identical.

Specify your task and pass your image to the classifier. The image can be a link, a local path or a base64-encoded image. For example, what species of cat is shown below?

```
>>> from transformers import pipeline
```

```
>>> vision_classifier = pipeline(model="google/vit-base-patch16-224")
```

```
>>> preds = vision_classifier(  
...     images="https://huggingface.co/datasets/huggingface/documentation-
```

```
images/resolve/main/pipeline-cat-chonk.jpeg"  
... )
```

```
>>> preds = [{"score": round(pred["score"], 4), "label": pred["label"]} for pred in preds]  
>>> preds
```

```
[{'score': 0.4335, 'label': 'lynx, catamount'}, {'score': 0.0348, 'label': 'cougar, puma,  
catamount, mountain lion, painter, panther, Felis concolor'}, {'score': 0.0324, 'label': 'snow  
leopard, ounce, Panthera uncia'}, {'score': 0.0239, 'label': 'Egyptian cat'}, {'score': 0.0229,  
'label': 'tiger cat'}]
```

### Text pipeline

Using a [pipeline\(\)](#) for NLP tasks is practically identical.

```
>>> from transformers import pipeline
```

```
>>> # This model is a `zero-shot-classification` model.
```

```
>>> # It will classify text, except you are free to choose any label you might imagine
```

```
>>> classifier = pipeline(model="facebook/bart-large-mnli")
```

```
>>> classifier(
...     "I have a problem with my iphone that needs to be resolved asap!!",
...     candidate_labels=["urgent", "not urgent", "phone", "tablet", "computer"],
... )
{'sequence': 'I have a problem with my iphone that needs to be resolved asap!!', 'labels':
['urgent', 'phone', 'computer', 'not urgent', 'tablet'], 'scores': [0.504, 0.479, 0.013, 0.003,
0.002]}
```

### Multimodal pipeline

The [pipeline\(\)](#) supports more than one modality. For example, a visual question answering (VQA) task combines text and image. Feel free to use any image link you like and a question you want to ask about the image. The image can be a URL or a local path to the image. For example, if you use this [invoice image](#):

```
>>> from transformers import pipeline

>>> vqa = pipeline(model="impira/layoutlm-document-qa")
>>> output = vqa(
...
image="https://huggingface.co/spaces/impira/docquery/resolve/2359223c1837a7587402bda0
f2643382a6eefeab/invoice.png",
...     question="What is the invoice number?",
... )
>>> output[0]["score"] = round(output[0]["score"], 3)
>>> output
[{'score': 0.425, 'answer': 'us-001', 'start': 16, 'end': 16}]
```

To run the example above you need to have [pytesseract](#) installed in addition to 🤖 Transformers:

```
sudo apt install -y tesseract-ocr
pip install pytesseract
```

### Using pipeline on large models with 🤖 accelerate :

You can easily run pipeline on large models using 🤖 accelerate! First make sure you have installed accelerate with `pip install accelerate`. First load your model using `device_map="auto"`! We will use facebook/opt-1.3b for our example.

```
# pip install accelerate
import torch
from transformers import pipeline
```

```
pipe = pipeline(model="facebook/opt-1.3b", torch_dtype=torch.bfloat16,
device_map="auto")
output = pipe("This is a cool example!", do_sample=True, top_p=0.95)
You can also pass 8-bit loaded models if you install bitsandbytes and add the argument
load_in_8bit=True
```

```
# pip install accelerate bitsandbytes
import torch
from transformers import pipeline
```



```
pipe = pipeline(model="facebook/opt-1.3b", device_map="auto",
model_kwargs={"load_in_8bit": True})
output = pipe("This is a cool example!", do_sample=True, top_p=0.95)
```

Note that you can replace the checkpoint with any Hugging Face model that supports large model loading, such as BLOOM.

### **Creating web demos from pipelines with gradio**

Pipelines are automatically supported in [Gradio](#), a library that makes creating beautiful and user-friendly machine learning apps on the web a breeze. First, make sure you have Gradio installed:

```
pip install gradio
```

Then, you can create a web demo around an image classification pipeline (or any other pipeline) in a single line of code by calling Gradio's [Interface.from\\_pipeline](#) function to launch the pipeline. This creates an intuitive drag-and-drop interface in your browser:

```
from transformers import pipeline
import gradio as gr
```

```
pipe = pipeline("image-classification", model="google/vit-base-patch16-224")
```

```
gr.Interface.from_pipeline(pipe).launch()
```



## Load pretrained instances with an AutoClass

With so many different Transformer architectures, it can be challenging to create one for your checkpoint. As a part of 🤗 Transformers core philosophy to make the library easy, simple and flexible to use, an AutoClass automatically infers and loads the correct architecture from a given checkpoint. The `from_pretrained()` method lets you quickly load a pretrained model for any architecture so you don't have to devote time and resources to train a model from scratch. Producing this type of checkpoint-agnostic code means if your code works for one checkpoint, it will work with another checkpoint - as long as it was trained for a similar task - even if the architecture is different.

Remember, architecture refers to the skeleton of the model and checkpoints are the weights for a given architecture. For example, [BERT](#) is an architecture, while `google-bert/bert-base-uncased` is a checkpoint. Model is a general term that can mean either architecture or checkpoint.

In this tutorial, learn to:

- Load a pretrained tokenizer.
- Load a pretrained image processor
- Load a pretrained feature extractor.
- Load a pretrained processor.
- Load a pretrained model.
- Load a model as a backbone.

### AutoTokenizer

Nearly every NLP task begins with a tokenizer. A tokenizer converts your input into a format that can be processed by the model.

Load a tokenizer with [AutoTokenizer.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoTokenizer
```

```
>>> tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-uncased")
```

Then tokenize your input as shown below:

```
>>> sequence = "In a hole in the ground there lived a hobbit."
```

```
>>> print(tokenizer(sequence))
```

```
{'input_ids': [101, 1999, 1037, 4920, 1999, 1996, 2598, 2045, 2973, 1037, 7570, 10322, 4183, 1012, 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]}
```

### AutoImageProcessor

For vision tasks, an image processor processes the image into the correct input format.

```
>>> from transformers import AutoImageProcessor
```

```
>>> image_processor = AutoImageProcessor.from_pretrained("google/vit-base-patch16-224")
```

### AutoBackbone

A Swin backbone with multiple stages for outputting a feature map.

The [AutoBackbone](#) lets you use pretrained models as backbones to get feature maps from different stages of the backbone. You should specify one of the following parameters in [from\\_pretrained\(\)](#):

- `out_indices` is the index of the layer you'd like to get the feature map from
- `out_features` is the name of the layer you'd like to get the feature map from

These parameters can be used interchangeably, but if you use both, make sure they're aligned with each other! If you don't pass any of these parameters, the backbone returns the feature map from the last layer.

A feature map from the first stage of the backbone. The patch partition refers to the model stem.

For example, in the above diagram, to return the feature map from the first stage of the Swin backbone, you can set `out_indices=(1,)`:

```
>>> from transformers import AutoImageProcessor, AutoBackbone
>>> import torch
>>> from PIL import Image
>>> import requests
>>> url = "http://images.cocodataset.org/val2017/0000000039769.jpg"
>>> image = Image.open(requests.get(url, stream=True).raw)
>>> processor = AutoImageProcessor.from_pretrained("microsoft/swin-tiny-patch4-window7-224")
>>> model = AutoBackbone.from_pretrained("microsoft/swin-tiny-patch4-window7-224",
out_indices=(1,))
```

```
>>> inputs = processor(image, return_tensors="pt")
>>> outputs = model(**inputs)
>>> feature_maps = outputs.feature_maps
```

Now you can access the `feature_maps` object from the first stage of the backbone:

```
>>> list(feature_maps[0].shape)
[1, 96, 56, 56]
```

### **AutoFeatureExtractor**

For audio tasks, a feature extractor processes the audio signal into the correct input format.

Load a feature extractor with [AutoFeatureExtractor.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoFeatureExtractor

>>> feature_extractor = AutoFeatureExtractor.from_pretrained(
...     "ehcalabres/wav2vec2-lg-xlsr-en-speech-emotion-recognition"
... )
```

### **AutoProcessor**

Multimodal tasks require a processor that combines two types of preprocessing tools. For example, the [LayoutLMV2](#) model requires an image processor to handle images and a tokenizer to handle text; a processor combines both of them.

Load a processor with [AutoProcessor.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoProcessor

>>> processor = AutoProcessor.from_pretrained("microsoft/layoutlmv2-base-uncased")
```

### **AutoModel**

Pytorch

Hide Pytorch content

The `AutoModelFor` classes let you load a pretrained model for a given task (see [here](#) for a complete list of available tasks). For example, load a model for sequence classification with [AutoModelForSequenceClassification.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoModelForSequenceClassification
```

```
>>> model = AutoModelForSequenceClassification.from_pretrained("distilbert/distilbert-base-uncased")
```

Easily reuse the same checkpoint to load an architecture for a different task:

```
>>> from transformers import AutoModelForTokenClassification
```

```
>>> model = AutoModelForTokenClassification.from_pretrained("distilbert/distilbert-base-uncased")
```

For PyTorch models, the `from_pretrained()` method uses `torch.load()` which internally uses pickle and is known to be insecure. In general, never load a model that could have come from an untrusted source, or that could have been tampered with. This security risk is partially mitigated for public models hosted on the Hugging Face Hub, which are [scanned for malware](#) at each commit. See the [Hub documentation](#) for best practices like [signed commit verification](#) with GPG.

TensorFlow and Flax checkpoints are not affected, and can be loaded within PyTorch architectures using the `from_tf` and `from_flax` kwargs for the `from_pretrained` method to circumvent this issue.

Generally, we recommend using the `AutoTokenizer` class and the `AutoModelFor` class to load pretrained instances of models. This will ensure you load the correct architecture every time.

In the next [tutorial](#), learn how to use your newly loaded tokenizer, image processor, feature extractor and processor to preprocess a dataset for fine-tuning.

TensorFlow

Hide TensorFlow content

Finally, the `TFAutoModelFor` classes let you load a pretrained model for a given task (see [here](#) for a complete list of available tasks). For example, load a model for sequence classification with [TFAutoModelForSequenceClassification.from\\_pretrained\(\)](#):

```
>>> from transformers import TFAutoModelForSequenceClassification
```

```
>>> model = TFAutoModelForSequenceClassification.from_pretrained("distilbert/distilbert-base-uncased")
```

Easily reuse the same checkpoint to load an architecture for a different task:

```
>>> from transformers import TFAutoModelForTokenClassification
```

```
>>> model = TFAutoModelForTokenClassification.from_pretrained("distilbert/distilbert-base-uncased")
```

Generally, we recommend using the `AutoTokenizer` class and the `TFAutoModelFor` class to load pretrained instances of models. This will ensure you load the correct architecture every time. In the next [tutorial](#), learn how to use your newly loaded tokenizer, image processor, feature extractor and processor to preprocess a dataset for fine-tuning.

## Preprocess

Before you can train a model on a dataset, it needs to be preprocessed into the expected model input format. Whether your data is text, images, or audio, it needs to be converted and assembled into batches of tensors. 😊 Transformers provides a set of preprocessing classes to help prepare your data for the model. In this tutorial, you'll learn that for:

- Text, use a [Tokenizer](#) to convert text into a sequence of tokens, create a numerical representation of the tokens, and assemble them into tensors.
- Speech and audio, use a [Feature extractor](#) to extract sequential features from audio waveforms and convert them into tensors.
- Image inputs use a [ImageProcessor](#) to convert images into tensors.
- Multimodal inputs, use a [Processor](#) to combine a tokenizer and a feature extractor or image processor.

AutoProcessor **always** works and automatically chooses the correct class for the model you're using, whether you're using a tokenizer, image processor, feature extractor or processor.

Before you begin, install 😊 Datasets so you can load some datasets to experiment with:

pip install datasets

### Natural Language Processing

The main tool for preprocessing textual data is a [tokenizer](#). A tokenizer splits text into *tokens* according to a set of rules. The tokens are converted into numbers and then tensors, which become the model inputs. Any additional inputs required by the model are added by the tokenizer.

If you plan on using a pretrained model, it's important to use the associated pretrained tokenizer. This ensures the text is split the same way as the pretraining corpus, and uses the same corresponding tokens-to-index (usually referred to as the *vocab*) during pretraining. Get started by loading a pretrained tokenizer with the [AutoTokenizer.from\\_pretrained\(\)](#) method. This downloads the *vocab* a model was pretrained with:

```
>>> from transformers import AutoTokenizer
```

```
>>> tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-cased")
```

Then pass your text to the tokenizer:

```
>>> encoded_input = tokenizer("Do not meddle in the affairs of wizards, for they are subtle and quick to anger.")
```

```
>>> print(encoded_input)
```

```
{'input_ids': [101, 2079, 2025, 19960, 10362, 1999, 1996, 3821, 1997, 16657, 1010, 2005, 2027, 2024, 11259, 1998, 4248, 2000, 4963, 1012, 102],
```

```
'token_type_ids': [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]}
```

The tokenizer returns a dictionary with three important items:

- [input\\_ids](#) are the indices corresponding to each token in the sentence.
- [attention\\_mask](#) indicates whether a token should be attended to or not.
- [token\\_type\\_ids](#) identifies which sequence a token belongs to when there is more than one sequence.

Return your input by decoding the input\_ids:

```
>>> tokenizer.decode(encoded_input["input_ids"])
'[CLS] Do not meddle in the affairs of wizards, for they are subtle and quick to anger. [SEP]'
```

As you can see, the tokenizer added two special tokens - CLS and SEP (classifier and separator) - to the sentence. Not all models need special tokens, but if they do, the tokenizer automatically adds them for you.

If there are several sentences you want to preprocess, pass them as a list to the tokenizer:

```
>>> batch_sentences = [
...     "But what about second breakfast?",
...     "Don't think he knows about second breakfast, Pip.",
...     "What about elevensies?",
... ]
>>> encoded_inputs = tokenizer(batch_sentences)
>>> print(encoded_inputs)
{'input_ids': [[101, 1252, 1184, 1164, 1248, 6462, 136, 102],
               [101, 1790, 112, 189, 1341, 1119, 3520, 1164, 1248, 6462, 117, 21902, 1643, 119,
               102],
               [101, 1327, 1164, 5450, 23434, 136, 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],
                    [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],
                    [1, 1, 1, 1, 1, 1, 1, 1]]}
```

## Pad

Sentences aren't always the same length which can be an issue because tensors, the model inputs, need to have a uniform shape. Padding is a strategy for ensuring tensors are rectangular by adding a special *padding token* to shorter sentences.

Set the padding parameter to True to pad the shorter sequences in the batch to match the longest sequence:

```
>>> batch_sentences = [
...     "But what about second breakfast?",
...     "Don't think he knows about second breakfast, Pip.",
...     "What about elevensies?",
... ]
>>> encoded_input = tokenizer(batch_sentences, padding=True)
>>> print(encoded_input)
{'input_ids': [[101, 1252, 1184, 1164, 1248, 6462, 136, 102, 0, 0, 0, 0, 0, 0],
               [101, 1790, 112, 189, 1341, 1119, 3520, 1164, 1248, 6462, 117, 21902, 1643, 119,
               102],
               [101, 1327, 1164, 5450, 23434, 136, 102, 0, 0, 0, 0, 0, 0, 0]],
 '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, 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, 0, 0, 0, 0, 0, 0],
                    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                    [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]]}
```

The first and third sentences are now padded with 0's because they are shorter.

## Truncation

On the other end of the spectrum, sometimes a sequence may be too long for a model to handle. In this case, you'll need to truncate the sequence to a shorter length. Set the truncation parameter to True to truncate a sequence to the maximum length accepted by the model:

```
>>> batch_sentences = [
...     "But what about second breakfast?",
...     "Don't think he knows about second breakfast, Pip.",
...     "What about elevensies?",
... ]
>>> encoded_input = tokenizer(batch_sentences, padding=True, truncation=True)
>>> print(encoded_input)
{'input_ids': [[101, 1252, 1184, 1164, 1248, 6462, 136, 102, 0, 0, 0, 0, 0, 0],
               [101, 1790, 112, 189, 1341, 1119, 3520, 1164, 1248, 6462, 117, 21902, 1643, 119,
               102],
               [101, 1327, 1164, 5450, 23434, 136, 102, 0, 0, 0, 0, 0, 0, 0]],
 '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, 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, 0, 0, 0, 0, 0, 0],
                   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]]}
```

Check out the [Padding and truncation](#) concept guide to learn more different padding and truncation arguments.

### Build tensors

Finally, you want the tokenizer to return the actual tensors that get fed to the model.

Set the return\_tensors parameter to either pt for PyTorch, or tf for TensorFlow:

Pytorch

Hide Pytorch content

```
>>> batch_sentences = [
...     "But what about second breakfast?",
...     "Don't think he knows about second breakfast, Pip.",
...     "What about elevensies?",
... ]
>>> encoded_input = tokenizer(batch_sentences, padding=True, truncation=True,
return_tensors="pt")
>>> print(encoded_input)
{'input_ids': tensor([[101, 1252, 1184, 1164, 1248, 6462, 136, 102, 0, 0, 0, 0, 0, 0],
                     [101, 1790, 112, 189, 1341, 1119, 3520, 1164, 1248, 6462, 117, 21902, 1643,
                     119, 102],
                     [101, 1327, 1164, 5450, 23434, 136, 102, 0, 0, 0, 0, 0, 0, 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, 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, 0, 0, 0, 0, 0, 0],
                          [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                          [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]])}
```

TensorFlow

Hide TensorFlow content

```
>>> batch_sentences = [
...     "But what about second breakfast?",
...     "Don't think he knows about second breakfast, Pip.",
...     "What about elevensies?",
... ]
>>> encoded_input = tokenizer(batch_sentences, padding=True, truncation=True,
return_tensors="tf")
>>> print(encoded_input)
{'input_ids': <tf.Tensor: shape=(2, 9), dtype=int32, numpy=
array([[101, 1252, 1184, 1164, 1248, 6462, 136, 102, 0, 0, 0, 0, 0, 0],
      [101, 1790, 112, 189, 1341, 1119, 3520, 1164, 1248, 6462, 117, 21902, 1643, 119, 102],
      [101, 1327, 1164, 5450, 23434, 136, 102, 0, 0, 0, 0, 0, 0, 0, 0]],
      dtype=int32)>,
'token_type_ids': <tf.Tensor: shape=(2, 9), dtype=int32, numpy=
array([[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, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)>,
'attention_mask': <tf.Tensor: shape=(2, 9), dtype=int32, numpy=
array([[1, 1, 1, 1, 1, 1, 1, 1, 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, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)>}
```

Different pipelines support tokenizer arguments in their `__call__()` differently. `text-2-text-generation` pipelines support (i.e. pass on) only truncation`. text-generation` pipelines support max_length`, truncation`, padding` and add_special_tokens`. In fill-mask` pipelines, tokenizer arguments can be passed in the tokenizer_kwargs` argument (dictionary).`

## Audio

For audio tasks, you'll need a [feature extractor](#) to prepare your dataset for the model. The feature extractor is designed to extract features from raw audio data, and convert them into tensors.

Load the [MInDS-14](#) dataset (see the 😊 [Datasets tutorial](#) for more details on how to load a dataset) to see how you can use a feature extractor with audio datasets:

```
>>> from datasets import load_dataset, Audio

>>> dataset = load_dataset("PolyAI/minds14", name="en-US", split="train")
Access the first element of the audio column to take a look at the input. Calling the audio
column automatically loads and resamples the audio file:

>>> dataset[0]["audio"]
{'array': array([ 0.        , 0.00024414, -0.00024414, ..., -0.00024414,
                 0.        , 0.        ], dtype=float32),
'path':
'/root/.cache/huggingface/datasets/downloads/extracted/f14948e0e84be638dd7943ac36518a4
cf3324e8b7aa331c5ab11541518e9368c/en-
US~JOINT_ACCOUNT/602ba55abb1e6d0fbce92065.wav',
'sampling_rate': 8000}
```

This returns three items:

- `array` is the speech signal loaded - and potentially resampled - as a 1D array.

- path points to the location of the audio file.
- sampling\_rate refers to how many data points in the speech signal are measured per second.

For this tutorial, you'll use the [Wav2Vec2](#) model. Take a look at the model card, and you'll learn Wav2Vec2 is pretrained on 16kHz sampled speech audio. It is important your audio data's sampling rate matches the sampling rate of the dataset used to pretrain the model. If your data's sampling rate isn't the same, then you need to resample your data.

1. Use 🧐 Datasets' [cast\\_column](#) method to upsample the sampling rate to 16kHz:

```
>>> dataset = dataset.cast_column("audio", Audio(sampling_rate=16_000))
```

2. Call the audio column again to resample the audio file:

```
>>> dataset[0]["audio"]
{'array': array([ 2.3443763e-05,  2.1729663e-04,  2.2145823e-04, ...,
                3.8356509e-05, -7.3497440e-06, -2.1754686e-05], dtype=float32),
 'path':
 '/root/.cache/huggingface/datasets/downloads/extracted/f14948e0e84be638dd7943ac36518a4cf3324e8b7aa331c5ab11541518e9368c/en-US~JOINT_ACCOUNT/602ba55abb1e6d0fbce92065.wav',
 'sampling_rate': 16000}
```

Next, load a feature extractor to normalize and pad the input. When padding textual data, a 0 is added for shorter sequences. The same idea applies to audio data. The feature extractor adds a 0 - interpreted as silence - to array.

Load the feature extractor with [AutoFeatureExtractor.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoFeatureExtractor
```

```
>>> feature_extractor = AutoFeatureExtractor.from_pretrained("facebook/wav2vec2-base")
```

Pass the audio array to the feature extractor. We also recommend adding the sampling\_rate argument in the feature extractor in order to better debug any silent errors that may occur.

```
>>> audio_input = [dataset[0]["audio"]["array"]]
>>> feature_extractor(audio_input, sampling_rate=16000)
{'input_values': [array([ 3.8106556e-04,  2.7506407e-03,  2.8015103e-03, ...,
                        5.6335266e-04,  4.6588284e-06, -1.7142107e-04], dtype=float32)]}
```

Just like the tokenizer, you can apply padding or truncation to handle variable sequences in a batch. Take a look at the sequence length of these two audio samples:

```
>>> dataset[0]["audio"]["array"].shape
(173398,)
```

```
>>> dataset[1]["audio"]["array"].shape
(106496,)
```

Create a function to preprocess the dataset so the audio samples are the same lengths. Specify a maximum sample length, and the feature extractor will either pad or truncate the sequences to match it:

```
>>> def preprocess_function(examples):
...     audio_arrays = [x["array"] for x in examples["audio"]]
...     inputs = feature_extractor(
```



```
...     audio_arrays,
...     sampling_rate=16000,
...     padding=True,
...     max_length=100000,
...     truncation=True,
... )
... return inputs
```

Apply the preprocess\_function to the first few examples in the dataset:

```
>>> processed_dataset = preprocess_function(dataset[:5])
```

The sample lengths are now the same and match the specified maximum length. You can pass your processed dataset to the model now!

```
>>> processed_dataset["input_values"][0].shape
(100000,)
```

```
>>> processed_dataset["input_values"][1].shape
(100000,)
```

### Computer vision

For computer vision tasks, you'll need an [image processor](#) to prepare your dataset for the model. Image preprocessing consists of several steps that convert images into the input expected by the model. These steps include but are not limited to resizing, normalizing, color channel correction, and converting images to tensors.

Image preprocessing often follows some form of image augmentation. Both image preprocessing and image augmentation transform image data, but they serve different purposes:

- Image augmentation alters images in a way that can help prevent overfitting and increase the robustness of the model. You can get creative in how you augment your data - adjust brightness and colors, crop, rotate, resize, zoom, etc. However, be mindful not to change the meaning of the images with your augmentations.
- Image preprocessing guarantees that the images match the model's expected input format. When fine-tuning a computer vision model, images must be preprocessed exactly as when the model was initially trained.

You can use any library you like for image augmentation. For image preprocessing, use the ImageProcessor associated with the model.

Load the [food101](#) dataset (see the 🤖 [Datasets tutorial](#) for more details on how to load a dataset) to see how you can use an image processor with computer vision datasets:

Use 🤖 Datasets split parameter to only load a small sample from the training split since the dataset is quite large!

```
>>> from datasets import load_dataset
```

```
>>> dataset = load_dataset("food101", split="train[:100]")
```

Next, take a look at the image with 🤖 Datasets [Image](#) feature:

```
>>> dataset[0]["image"]
```

Load the image processor with [AutoImageProcessor.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoImageProcessor
```

```
>>> image_processor = AutoImageProcessor.from_pretrained("google/vit-base-patch16-224")
```

First, let's add some image augmentation. You can use any library you prefer, but in this tutorial, we'll use torchvision's [transforms](#) module. If you're interested in using another data augmentation library, learn how in the [Albumentations](#) or [Kornia notebooks](#).

1. Here we use [Compose](#) to chain together a couple of transforms - [RandomResizedCrop](#) and [ColorJitter](#). Note that for resizing, we can get the image size requirements from the image\_processor. For some models, an exact height and width are expected, for others only the shortest\_edge is defined.

```
>>> from torchvision.transforms import RandomResizedCrop, ColorJitter, Compose
```

```
>>> size = (  
...     image_processor.size["shortest_edge"]  
...     if "shortest_edge" in image_processor.size  
...     else (image_processor.size["height"], image_processor.size["width"])  
... )
```

```
>>> _transforms = Compose([RandomResizedCrop(size), ColorJitter(brightness=0.5,  
hue=0.5)])
```

2. The model accepts [pixel values](#) as its input. ImageProcessor can take care of normalizing the images, and generating appropriate tensors. Create a function that combines image augmentation and image preprocessing for a batch of images and generates pixel\_values:

```
>>> def transforms(examples):  
...     images = [_transforms(img.convert("RGB")) for img in examples["image"]]  
...     examples["pixel_values"] = image_processor(images, do_resize=False,  
return_tensors="pt")["pixel_values"]  
...     return examples
```

In the example above we set do\_resize=False because we have already resized the images in the image augmentation transformation, and leveraged the size attribute from the appropriate image\_processor. If you do not resize images during image augmentation, leave this parameter out. By default, ImageProcessor will handle the resizing.

If you wish to normalize images as a part of the augmentation transformation, use the image\_processor.image\_mean, and image\_processor.image\_std values.

3. Then use 🐞 [Datasets.set\\_transform](#) to apply the transforms on the fly:

```
>>> dataset.set_transform(transforms)
```

4. Now when you access the image, you'll notice the image processor has added pixel\_values. You can pass your processed dataset to the model now!

```
>>> dataset[0].keys()
```

Here is what the image looks like after the transforms are applied. The image has been randomly cropped and it's color properties are different.

```
>>> import numpy as np  
>>> import matplotlib.pyplot as plt
```

```
>>> img = dataset[0]["pixel_values"]
>>> plt.imshow(img.permute(1, 2, 0))
```

For tasks like object detection, semantic segmentation, instance segmentation, and panoptic segmentation, ImageProcessor offers post processing methods. These methods convert model's raw outputs into meaningful predictions such as bounding boxes, or segmentation maps.

### Pad

In some cases, for instance, when fine-tuning [DETR](#), the model applies scale augmentation at training time. This may cause images to be different sizes in a batch. You can use `DetrImageProcessor.pad()` from [DetrImageProcessor](#) and define a custom `collate_fn` to batch images together.

```
>>> def collate_fn(batch):
...     pixel_values = [item["pixel_values"] for item in batch]
...     encoding = image_processor.pad(pixel_values, return_tensors="pt")
...     labels = [item["labels"] for item in batch]
...     batch = {}
...     batch["pixel_values"] = encoding["pixel_values"]
...     batch["pixel_mask"] = encoding["pixel_mask"]
...     batch["labels"] = labels
...     return batch
```

### Multimodal

For tasks involving multimodal inputs, you'll need a [processor](#) to prepare your dataset for the model. A processor couples together two processing objects such as tokenizer and feature extractor.

Load the [LJ Speech](#) dataset (see the 🤗 [Datasets tutorial](#) for more details on how to load a dataset) to see how you can use a processor for automatic speech recognition (ASR):

```
>>> from datasets import load_dataset
```

```
>>> lj_speech = load_dataset("lj_speech", split="train")
```

For ASR, you're mainly focused on audio and text so you can remove the other columns:

```
>>> lj_speech = lj_speech.map(remove_columns=["file", "id", "normalized_text"])
```

Now take a look at the audio and text columns:

```
>>> lj_speech[0]["audio"]
{'array': array([-7.3242188e-04, -7.6293945e-04, -6.4086914e-04, ...,
               7.3242188e-04,  2.1362305e-04,  6.1035156e-05], dtype=float32),
 'path':
 '/root/.cache/huggingface/datasets/downloads/extracted/917ece08c95cf0c4115e45294e3cd0d
 ee724a1165b7fc11798369308a465bd26/LJSpeech-1.1/wavs/LJ001-0001.wav',
 'sampling_rate': 22050}
```

```
>>> lj_speech[0]["text"]
```

'Printing, in the only sense with which we are at present concerned, differs from most if not from all the arts and crafts represented in the Exhibition'

Remember you should always [resample](#) your audio dataset's sampling rate to match the sampling rate of the dataset used to pretrain a model!

```
>>> lj_speech = lj_speech.cast_column("audio", Audio(sampling_rate=16_000))
```

Load a processor with [AutoProcessor.from\\_pretrained\(\)](#):

```
>>> from transformers import AutoProcessor
```

```
>>> processor = AutoProcessor.from_pretrained("facebook/wav2vec2-base-960h")
```

1. Create a function to process the audio data contained in array to input\_values, and tokenize text to labels. These are the inputs to the model:

```
>>> def prepare_dataset(example):
```

```
...     audio = example["audio"]
```

```
...     example.update(processor(audio=audio["array"], text=example["text"],
sampling_rate=16000))
```

```
...     return example
```

2. Apply the prepare\_dataset function to a sample:

```
>>> prepare_dataset(lj_speech[0])
```

The processor has now added input\_values and labels, and the sampling rate has also been correctly downsampled to 16kHz. You can pass your processed dataset to the model now!

## Fine-tune a pretrained model



There are significant benefits to using a pretrained model. It reduces computation costs, your carbon footprint, and allows you to use state-of-the-art models without having to train one from scratch. 😊 Transformers provides access to thousands of pretrained models for a wide range of tasks. When you use a pretrained model, you train it on a dataset specific to your task. This is known as fine-tuning, an incredibly powerful training technique. In this tutorial, you will fine-tune a pretrained model with a deep learning framework of your choice:

- Fine-tune a pretrained model with 😊 Transformers [Trainer](#).
- Fine-tune a pretrained model in TensorFlow with Keras.
- Fine-tune a pretrained model in native PyTorch.

### Prepare a dataset

Before you can fine-tune a pretrained model, download a dataset and prepare it for training. The previous tutorial showed you how to process data for training, and now you get an opportunity to put those skills to the test!

Begin by loading the [Yelp Reviews](#) dataset:

```
>>> from datasets import load_dataset
```

```
>>> dataset = load_dataset("yelp_review_full")
```

```
>>> dataset["train"][100]
```

```
{'label': 0,
```

```
'text': 'My expectations for McDonalds are t rarely high. But for one to still fail so  
spectacularly...that takes something special!\n\nThe cashier took my friends\'s order, then  
promptly ignored me. I had to force myself in front of a cashier who opened his register to  
wait on the person BEHIND me. I waited over five minutes for a gigantic order that included  
precisely one kid\'s meal. After watching two people who ordered after me be handed their  
food, I asked where mine was. The manager started yelling at the cashiers for \\'serving off  
their orders\' when they didn\'t have their food. But neither cashier was anywhere near those  
controls, and the manager was the one serving food to customers and clearing the  
boards.\n\nThe manager was rude when giving me my order. She didn\'t make sure that I had  
everything ON MY RECEIPT, and never even had the decency to apologize that I felt I was  
getting poor service.\n\nI\'ve eaten at various McDonalds restaurants for over 30 years. I\'ve  
worked at more than one location. I expect bad days, bad moods, and the occasional mistake.  
But I have yet to have a decent experience at this store. It will remain a place I avoid unless  
someone in my party needs to avoid illness from low blood sugar. Perhaps I should go back  
to the racially biased service of Steak n Shake instead!'}

```

As you now know, you need a tokenizer to process the text and include a padding and truncation strategy to handle any variable sequence lengths. To process your dataset in one step, use 😊 Datasets [map](#) method to apply a preprocessing function over the entire dataset:

```
>>> from transformers import AutoTokenizer
```

```
>>> tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-cased")
```

```
>>> def tokenize_function(examples):  
...     return tokenizer(examples["text"], padding="max_length", truncation=True)
```

```
>>> tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

If you like, you can create a smaller subset of the full dataset to fine-tune on to reduce the time it takes:

```
>>> small_train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(1000))  
>>> small_eval_dataset = tokenized_datasets["test"].shuffle(seed=42).select(range(1000))
```

## Train

At this point, you should follow the section corresponding to the framework you want to use. You can use the links in the right sidebar to jump to the one you want - and if you want to hide all of the content for a given framework, just use the button at the top-right of that framework's block!

Pytorch

Hide Pytorch content

## Train with PyTorch Trainer

😊 Transformers provides a [Trainer](#) class optimized for training 😊 Transformers models, making it easier to start training without manually writing your own training loop. The [Trainer](#) API supports a wide range of training options and features such as logging, gradient accumulation, and mixed precision.

Start by loading your model and specify the number of expected labels. From the Yelp Review [dataset card](#), you know there are five labels:

```
>>> from transformers import AutoModelForSequenceClassification
```

```
>>> model = AutoModelForSequenceClassification.from_pretrained("google-bert/bert-base-cased", num_labels=5)
```

You will see a warning about some of the pretrained weights not being used and some weights being randomly initialized. Don't worry, this is completely normal! The pretrained head of the BERT model is discarded, and replaced with a randomly initialized classification head. You will fine-tune this new model head on your sequence classification task, transferring the knowledge of the pretrained model to it.

## Training hyperparameters

Next, create a [TrainingArguments](#) class which contains all the hyperparameters you can tune as well as flags for activating different training options. For this tutorial you can start with the default training [hyperparameters](#), but feel free to experiment with these to find your optimal settings.

Specify where to save the checkpoints from your training:

```
>>> from transformers import TrainingArguments
```

```
>>> training_args = TrainingArguments(output_dir="test_trainer")
```

## Evaluate

[Trainer](#) does not automatically evaluate model performance during training. You'll need to pass [Trainer](#) a function to compute and report metrics. The [🧑🏻 Evaluate](#) library provides a simple [accuracy](#) function you can load with the `evaluate.load` (see this [quicktour](#) for more information) function:

```
>>> import numpy as np
>>> import evaluate
```

```
>>> metric = evaluate.load("accuracy")
```

Call `compute` on metric to calculate the accuracy of your predictions. Before passing your predictions to `compute`, you need to convert the logits to predictions (remember all 🧐 Transformers models return logits):

```
>>> def compute_metrics(eval_pred):
...     logits, labels = eval_pred
...     predictions = np.argmax(logits, axis=-1)
...     return metric.compute(predictions=predictions, references=labels)
```

If you'd like to monitor your evaluation metrics during fine-tuning, specify the `eval_strategy` parameter in your training arguments to report the evaluation metric at the end of each epoch:

```
>>> from transformers import TrainingArguments, Trainer
```

```
>>> training_args = TrainingArguments(output_dir="test_trainer", eval_strategy="epoch")
```

### **Trainer**

Create a [Trainer](#) object with your model, training arguments, training and test datasets, and evaluation function:

```
>>> trainer = Trainer(
...     model=model,
...     args=training_args,
...     train_dataset=small_train_dataset,
...     eval_dataset=small_eval_dataset,
...     compute_metrics=compute_metrics,
... )
```

Then fine-tune your model by calling [train\(\)](#):

```
>>> trainer.train()
```

TensorFlow

Hide TensorFlow content

### **Train a TensorFlow model with Keras**

You can also train 🧐 Transformers models in TensorFlow with the Keras API!

### **Loading data for Keras**

When you want to train a 🧐 Transformers model with the Keras API, you need to convert your dataset to a format that Keras understands. If your dataset is small, you can just convert the whole thing to NumPy arrays and pass it to Keras. Let's try that first before we do anything more complicated.

First, load a dataset. We'll use the CoLA dataset from the [GLUE benchmark](#), since it's a simple binary text classification task, and just take the training split for now.

```
from datasets import load_dataset
```

```
dataset = load_dataset("glue", "cola")
```

```
dataset = dataset["train"] # Just take the training split for now
```

Next, load a tokenizer and tokenize the data as NumPy arrays. Note that the labels are already a list of 0 and 1s, so we can just convert that directly to a NumPy array without tokenization!

```
from transformers import AutoTokenizer
import numpy as np
```

```
tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-cased")
tokenized_data = tokenizer(dataset["sentence"], return_tensors="np", padding=True)
# Tokenizer returns a BatchEncoding, but we convert that to a dict for Keras
tokenized_data = dict(tokenized_data)
```

```
labels = np.array(dataset["label"]) # Label is already an array of 0 and 1
Finally, load, compile, and fit the model. Note that Transformers models all have a default
task-relevant loss function, so you don't need to specify one unless you want to:
```

```
from transformers import TFAutoModelForSequenceClassification
from tensorflow.keras.optimizers import Adam
```

```
# Load and compile our model
model = TFAutoModelForSequenceClassification.from_pretrained("google-bert/bert-base-
cased")
# Lower learning rates are often better for fine-tuning transformers
model.compile(optimizer=Adam(3e-5)) # No loss argument!
```

```
model.fit(tokenized_data, labels)
```

You don't have to pass a loss argument to your models when you compile() them! Hugging Face models automatically choose a loss that is appropriate for their task and model architecture if this argument is left blank. You can always override this by specifying a loss yourself if you want to!

This approach works great for smaller datasets, but for larger datasets, you might find it starts to become a problem. Why? Because the tokenized array and labels would have to be fully loaded into memory, and because NumPy doesn't handle "jagged" arrays, so every tokenized sample would have to be padded to the length of the longest sample in the whole dataset. That's going to make your array even bigger, and all those padding tokens will slow down training too!

### **Loading data as a `tf.data.Dataset`**

If you want to avoid slowing down training, you can load your data as a `tf.data.Dataset` instead. Although you can write your own `tf.data` pipeline if you want, we have two convenience methods for doing this:

- [prepare\\_tf\\_dataset\(\)](#): This is the method we recommend in most cases. Because it is a method on your model, it can inspect the model to automatically figure out which columns are usable as model inputs, and discard the others to make a simpler, more performant dataset.
- [to\\_tf\\_dataset](#): This method is more low-level, and is useful when you want to exactly control how your dataset is created, by specifying exactly which columns and `label_cols` to include.

Before you can use [prepare\\_tf\\_dataset\(\)](#), you will need to add the tokenizer outputs to your dataset as columns, as shown in the following code sample:

```
def tokenize_dataset(data):
    # Keys of the returned dictionary will be added to the dataset as columns
    return tokenizer(data["text"])
```



```
dataset = dataset.map(tokenize_dataset)
```

Remember that Hugging Face datasets are stored on disk by default, so this will not inflate your memory usage! Once the columns have been added, you can stream batches from the dataset and add padding to each batch, which greatly reduces the number of padding tokens compared to padding the entire dataset.

```
>>> tf_dataset = model.prepare_tf_dataset(dataset["train"], batch_size=16, shuffle=True,
tokenizer=tokenizer)
```

Note that in the code sample above, you need to pass the tokenizer to `prepare_tf_dataset` so it can correctly pad batches as they're loaded. If all the samples in your dataset are the same length and no padding is necessary, you can skip this argument. If you need to do something more complex than just padding samples (e.g. corrupting tokens for masked language modelling), you can use the `collate_fn` argument instead to pass a function that will be called to transform the list of samples into a batch and apply any preprocessing you want. See our [examples](#) or [notebooks](#) to see this approach in action.

Once you've created a `tf.data.Dataset`, you can compile and fit the model as before:

```
model.compile(optimizer=Adam(3e-5)) # No loss argument!
```

```
model.fit(tf_dataset)
```

## Train in native PyTorch

Pytorch

Hide Pytorch content

[Trainer](#) takes care of the training loop and allows you to fine-tune a model in a single line of code. For users who prefer to write their own training loop, you can also fine-tune a 🤗 Transformers model in native PyTorch.

At this point, you may need to restart your notebook or execute the following code to free some memory:

```
del model
```

```
del trainer
```

```
torch.cuda.empty_cache()
```

Next, manually postprocess `tokenized_dataset` to prepare it for training.

1. Remove the text column because the model does not accept raw text as an input:

```
>>> tokenized_datasets = tokenized_datasets.remove_columns(["text"])
```

2. Rename the label column to labels because the model expects the argument to be named labels:

```
>>> tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
```

3. Set the format of the dataset to return PyTorch tensors instead of lists:

```
>>> tokenized_datasets.set_format("torch")
```

Then create a smaller subset of the dataset as previously shown to speed up the fine-tuning:

```
>>> small_train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(1000))
```

```
>>> small_eval_dataset = tokenized_datasets["test"].shuffle(seed=42).select(range(1000))
```

## DataLoader

Create a DataLoader for your training and test datasets so you can iterate over batches of data:

```
>>> from torch.utils.data import DataLoader
```

```
>>> train_dataloader = DataLoader(small_train_dataset, shuffle=True, batch_size=8)
```

```
>>> eval_dataloader = DataLoader(small_eval_dataset, batch_size=8)
```

Load your model with the number of expected labels:

```
>>> from transformers import AutoModelForSequenceClassification
```

```
>>> model = AutoModelForSequenceClassification.from_pretrained("google-bert/bert-base-cased", num_labels=5)
```

### **Optimizer and learning rate scheduler**

Create an optimizer and learning rate scheduler to fine-tune the model. Let's use the [AdamW](#) optimizer from PyTorch:

```
>>> from torch.optim import AdamW
```

```
>>> optimizer = AdamW(model.parameters(), lr=5e-5)
```

Create the default learning rate scheduler from [Trainer](#):

```
>>> from transformers import get_scheduler
```

```
>>> num_epochs = 3
```

```
>>> num_training_steps = num_epochs * len(train_dataloader)
```

```
>>> lr_scheduler = get_scheduler(  
...     name="linear", optimizer=optimizer, num_warmup_steps=0,  
num_training_steps=num_training_steps  
... )
```

Lastly, specify device to use a GPU if you have access to one. Otherwise, training on a CPU may take several hours instead of a couple of minutes.

```
>>> import torch
```

```
>>> device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
```

```
>>> model.to(device)
```

Get free access to a cloud GPU if you don't have one with a hosted notebook like [Colaboratory](#) or [SageMaker StudioLab](#).

Great, now you are ready to train! 🤖

### **Training loop**

To keep track of your training progress, use the [tqdm](#) library to add a progress bar over the number of training steps:

```
>>> from tqdm.auto import tqdm
```

```
>>> progress_bar = tqdm(range(num_training_steps))
```

```
>>> model.train()
```

```
>>> for epoch in range(num_epochs):
```

```

...     for batch in train_dataloader:
...         batch = {k: v.to(device) for k, v in batch.items()}
...         outputs = model(**batch)
...         loss = outputs.loss
...         loss.backward()

...     optimizer.step()
...     lr_scheduler.step()
...     optimizer.zero_grad()
...     progress_bar.update(1)

```

## Evaluate

Just like how you added an evaluation function to [Trainer](#), you need to do the same when you write your own training loop. But instead of calculating and reporting the metric at the end of each epoch, this time you'll accumulate all the batches with `add_batch` and calculate the metric at the very end.

```

>>> import evaluate

>>> metric = evaluate.load("accuracy")
>>> model.eval()
>>> for batch in eval_dataloader:
...     batch = {k: v.to(device) for k, v in batch.items()}
...     with torch.no_grad():
...         outputs = model(**batch)

...     logits = outputs.logits
...     predictions = torch.argmax(logits, dim=-1)
...     metric.add_batch(predictions=predictions, references=batch["labels"])

>>> metric.compute()

```

## Train with a script

Along with the 🤗 Transformers [notebooks](#), there are also example scripts demonstrating how to train a model for a task with [PyTorch](#), [TensorFlow](#), or [JAX/Flax](#).

You will also find scripts we've used in our [research projects](#) and [legacy examples](#) which are mostly community contributed. These scripts are not actively maintained and require a specific version of 🤗 Transformers that will most likely be incompatible with the latest version of the library.

The example scripts are not expected to work out-of-the-box on every problem, and you may need to adapt the script to the problem you're trying to solve. To help you with this, most of the scripts fully expose how data is preprocessed, allowing you to edit it as necessary for your use case.

For any feature you'd like to implement in an example script, please discuss it on the [forum](#) or in an [issue](#) before submitting a Pull Request. While we welcome bug fixes, it is unlikely we will merge a Pull Request that adds more functionality at the cost of readability.

This guide will show you how to run an example summarization training script in [PyTorch](#) and [TensorFlow](#). All examples are expected to work with both frameworks unless otherwise specified.

### Setup

To successfully run the latest version of the example scripts, you have to **install** 🤗

**Transformers from source** in a new virtual environment:

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

For older versions of the example scripts, click on the toggle below:

Examples for older versions of 🤗 Transformers

- [v4.5.1](#)
- [v4.4.2](#)
- [v4.3.3](#)
- [v4.2.2](#)
- [v4.1.1](#)
- [v4.0.1](#)
- [v3.5.1](#)
- [v3.4.0](#)
- [v3.3.1](#)
- [v3.2.0](#)
- [v3.1.0](#)
- [v3.0.2](#)
- [v2.11.0](#)
- [v2.10.0](#)
- [v2.9.1](#)
- [v2.8.0](#)
- [v2.7.0](#)
- [v2.6.0](#)
- [v2.5.1](#)
- [v2.4.0](#)
- [v2.3.0](#)
- [v2.2.0](#)
- [v2.1.1](#)

- [v2.0.0](#)
- [v1.2.0](#)
- [v1.1.0](#)
- [v1.0.0](#)

Then switch your current clone of 🤖 Transformers to a specific version, like v3.5.1 for example:

```
git checkout tags/v3.5.1
```

After you've setup the correct library version, navigate to the example folder of your choice and install the example specific requirements:

```
pip install -r requirements.txt
```

### Run a script

Pytorch

Hide Pytorch content

The example script downloads and preprocesses a dataset from the 🤖 [Datasets](#) library. Then the script fine-tunes a dataset with the [Trainer](#) on an architecture that supports summarization. The following example shows how to fine-tune [T5-small](#) on the [CNN/DailyMail](#) dataset. The T5 model requires an additional `source_prefix` argument due to how it was trained. This prompt lets T5 know this is a summarization task.

```
python examples/pytorch/summarization/run_summarization.py \
  --model_name_or_path google-t5/t5-small \
  --do_train \
  --do_eval \
  --dataset_name cnn_dailymail \
  --dataset_config "3.0.0" \
  --source_prefix "summarize: " \
  --output_dir /tmp/tst-summarization \
  --per_device_train_batch_size=4 \
  --per_device_eval_batch_size=4 \
  --overwrite_output_dir \
  --predict_with_generate
```

TensorFlow

Hide TensorFlow content

The example script downloads and preprocesses a dataset from the 🤖 [Datasets](#) library. Then the script fine-tunes a dataset using Keras on an architecture that supports summarization. The following example shows how to fine-tune [T5-small](#) on the [CNN/DailyMail](#) dataset. The T5 model requires an additional `source_prefix` argument due to how it was trained. This prompt lets T5 know this is a summarization task.

```
python examples/tensorflow/summarization/run_summarization.py \
  --model_name_or_path google-t5/t5-small \
  --dataset_name cnn_dailymail \
  --dataset_config "3.0.0" \
  --output_dir /tmp/tst-summarization \
  --per_device_train_batch_size 8 \
  --per_device_eval_batch_size 16 \
  --num_train_epochs 3 \
  --do_train \
```

--do\_eval

### Distributed training and mixed precision

The [Trainer](#) supports distributed training and mixed precision, which means you can also use it in a script. To enable both of these features:

- Add the fp16 or bf16 argument to enable mixed precision. XPU devices only supports bf16 for mixed precision training.
- Set the number of GPUs to use with the nproc\_per\_node argument.

```
torchrun \  
  --nproc_per_node 8 pytorch/summarization/run_summarization.py \  
  --fp16 \  
  --model_name_or_path google-t5/t5-small \  
  --do_train \  
  --do_eval \  
  --dataset_name cnn_dailymail \  
  --dataset_config "3.0.0" \  
  --source_prefix "summarize: " \  
  --output_dir /tmp/tst-summarization \  
  --per_device_train_batch_size=4 \  
  --per_device_eval_batch_size=4 \  
  --overwrite_output_dir \  
  --predict_with_generate
```

TensorFlow scripts utilize a [MirroredStrategy](#) for distributed training, and you don't need to add any additional arguments to the training script. The TensorFlow script will use multiple GPUs by default if they are available.

### Run a script on a TPU

Pytorch

Hide Pytorch content

Tensor Processing Units (TPUs) are specifically designed to accelerate performance.

PyTorch supports TPUs with the [XLA](#) deep learning compiler (see [here](#) for more details). To use a TPU, launch the xla\_spawn.py script and use the num\_cores argument to set the number of TPU cores you want to use.

```
python xla_spawn.py --num_cores 8 \  
  summarization/run_summarization.py \  
  --model_name_or_path google-t5/t5-small \  
  --do_train \  
  --do_eval \  
  --dataset_name cnn_dailymail \  
  --dataset_config "3.0.0" \  
  --source_prefix "summarize: " \  
  --output_dir /tmp/tst-summarization \  
  --per_device_train_batch_size=4 \  
  --per_device_eval_batch_size=4 \  
  --overwrite_output_dir \  
  --predict_with_generate
```

TensorFlow

Hide TensorFlow content

Tensor Processing Units (TPUs) are specifically designed to accelerate performance. TensorFlow scripts utilize a [TPUStrategy](#) for training on TPUs. To use a TPU, pass the name of the TPU resource to the tpu argument.

```
python run_summarization.py \  
  --tpu name_of_tpu_resource \  
  --model_name_or_path google-t5/t5-small \  
  --dataset_name cnn_dailymail \  
  --dataset_config "3.0.0" \  
  --output_dir /tmp/tst-summarization \  
  --per_device_train_batch_size 8 \  
  --per_device_eval_batch_size 16 \  
  --num_train_epochs 3 \  
  --do_train \  
  --do_eval
```

### Run a script with 🤖 Accelerate

🤖 [Accelerate](#) is a PyTorch-only library that offers a unified method for training a model on several types of setups (CPU-only, multiple GPUs, TPUs) while maintaining complete visibility into the PyTorch training loop. Make sure you have 🤖 Accelerate installed if you don't already have it:

Note: As Accelerate is rapidly developing, the git version of accelerate must be installed to run the scripts

```
pip install git+https://github.com/huggingface/accelerate
```

Instead of the run\_summarization.py script, you need to use the

run\_summarization\_no\_trainer.py script. 🤖 Accelerate supported scripts will have a task\_no\_trainer.py file in the folder. Begin by running the following command to create and save a configuration file:

```
accelerate config
```

Test your setup to make sure it is configured correctly:

```
accelerate test
```

Now you are ready to launch the training:

```
accelerate launch run_summarization_no_trainer.py \  
  --model_name_or_path google-t5/t5-small \  
  --dataset_name cnn_dailymail \  
  --dataset_config "3.0.0" \  
  --source_prefix "summarize: " \  
  --output_dir ~/tmp/tst-summarization
```

### Use a custom dataset

The summarization script supports custom datasets as long as they are a CSV or JSON Line file. When you use your own dataset, you need to specify several additional arguments:

- train\_file and validation\_file specify the path to your training and validation files.
- text\_column is the input text to summarize.
- summary\_column is the target text to output.

A summarization script using a custom dataset would look like this:

```
python examples/pytorch/summarization/run_summarization.py \
  --model_name_or_path google-t5/t5-small \
  --do_train \
  --do_eval \
  --train_file path_to_csv_or_jsonlines_file \
  --validation_file path_to_csv_or_jsonlines_file \
  --text_column text_column_name \
  --summary_column summary_column_name \
  --source_prefix "summarize: " \
  --output_dir /tmp/tst-summarization \
  --overwrite_output_dir \
  --per_device_train_batch_size=4 \
  --per_device_eval_batch_size=4 \
  --predict_with_generate
```

### Test a script

It is often a good idea to run your script on a smaller number of dataset examples to ensure everything works as expected before committing to an entire dataset which may take hours to complete. Use the following arguments to truncate the dataset to a maximum number of samples:

- `max_train_samples`
- `max_eval_samples`
- `max_predict_samples`

```
python examples/pytorch/summarization/run_summarization.py \
  --model_name_or_path google-t5/t5-small \
  --max_train_samples 50 \
  --max_eval_samples 50 \
  --max_predict_samples 50 \
  --do_train \
  --do_eval \
  --dataset_name cnn_dailymail \
  --dataset_config "3.0.0" \
  --source_prefix "summarize: " \
  --output_dir /tmp/tst-summarization \
  --per_device_train_batch_size=4 \
  --per_device_eval_batch_size=4 \
  --overwrite_output_dir \
  --predict_with_generate
```

Not all example scripts support the `max_predict_samples` argument. If you aren't sure whether your script supports this argument, add the `-h` argument to check:

```
examples/pytorch/summarization/run_summarization.py -h
```

### Resume training from checkpoint

Another helpful option to enable is resuming training from a previous checkpoint. This will ensure you can pick up where you left off without starting over if your training gets interrupted. There are two methods to resume training from a checkpoint.

The first method uses the `output_dir` `previous_output_dir` argument to resume training from the latest checkpoint stored in `output_dir`. In this case, you should remove `overwrite_output_dir`:



```
python examples/pytorch/summarization/run_summarization.py \
  --model_name_or_path google-t5/t5-small \
  --do_train \
  --do_eval \
  --dataset_name cnn_dailymail \
  --dataset_config "3.0.0" \
  --source_prefix "summarize: " \
  --output_dir /tmp/tst-summarization \
  --per_device_train_batch_size=4 \
  --per_device_eval_batch_size=4 \
  --output_dir previous_output_dir \
  --predict_with_generate
```

The second method uses the `resume_from_checkpoint` `path_to_specific_checkpoint` argument to resume training from a specific checkpoint folder.

```
python examples/pytorch/summarization/run_summarization.py \
  --model_name_or_path google-t5/t5-small \
  --do_train \
  --do_eval \
  --dataset_name cnn_dailymail \
  --dataset_config "3.0.0" \
  --source_prefix "summarize: " \
  --output_dir /tmp/tst-summarization \
  --per_device_train_batch_size=4 \
  --per_device_eval_batch_size=4 \
  --overwrite_output_dir \
  --resume_from_checkpoint path_to_specific_checkpoint \
  --predict_with_generate
```

### Share your model

All scripts can upload your final model to the [Model Hub](#). Make sure you are logged into Hugging Face before you begin:

```
huggingface-cli login
```

Then add the `push_to_hub` argument to the script. This argument will create a repository with your Hugging Face username and the folder name specified in `output_dir`.

To give your repository a specific name, use the `push_to_hub_model_id` argument to add it.

The repository will be automatically listed under your namespace.

The following example shows how to upload a model with a specific repository name:

```
python examples/pytorch/summarization/run_summarization.py \
  --model_name_or_path google-t5/t5-small \
  --do_train \
  --do_eval \
  --dataset_name cnn_dailymail \
  --dataset_config "3.0.0" \
  --source_prefix "summarize: " \
  --push_to_hub \
  --push_to_hub_model_id finetuned-t5-cnn_dailymail \
  --output_dir /tmp/tst-summarization \
  --per_device_train_batch_size=4 \
```

```
--per_device_eval_batch_size=4 \
--overwrite_output_dir \
--predict_with_generate
```

## Distributed training with 🤗 Accelerate

As models get bigger, parallelism has emerged as a strategy for training larger models on limited hardware and accelerating training speed by several orders of magnitude. At Hugging Face, we created the 🤗 [Accelerate](#) library to help users easily train a 🤗 Transformers model on any type of distributed setup, whether it is multiple GPU's on one machine or multiple GPU's across several machines. In this tutorial, learn how to customize your native PyTorch training loop to enable training in a distributed environment.

### Setup

Get started by installing 🤗 Accelerate:

```
pip install accelerate
```

Then import and create an [Accelerator](#) object. The [Accelerator](#) will automatically detect your type of distributed setup and initialize all the necessary components for training. You don't need to explicitly place your model on a device.

```
>>> from accelerate import Accelerator
```

```
>>> accelerator = Accelerator()
```

### Prepare to accelerate

The next step is to pass all the relevant training objects to the [prepare](#) method. This includes your training and evaluation DataLoaders, a model and an optimizer:

```
>>> train_dataloader, eval_dataloader, model, optimizer = accelerator.prepare(
...     train_dataloader, eval_dataloader, model, optimizer
... )
```

### Backward

The last addition is to replace the typical `loss.backward()` in your training loop with 🤗 Accelerate's [backward](#) method:

```
>>> for epoch in range(num_epochs):
...     for batch in train_dataloader:
...         outputs = model(**batch)
...         loss = outputs.loss
...         accelerator.backward(loss)
```

```
...     optimizer.step()
...     lr_scheduler.step()
...     optimizer.zero_grad()
...     progress_bar.update(1)
```

As you can see in the following code, you only need to add four additional lines of code to your training loop to enable distributed training!

```

+ from accelerate import Accelerator
  from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

+ accelerator = Accelerator()

  model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
  optimizer = AdamW(model.parameters(), lr=3e-5)

- device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
- model.to(device)

+ train_dataloader, eval_dataloader, model, optimizer = accelerator.prepare(
+   train_dataloader, eval_dataloader, model, optimizer
+ )

  num_epochs = 3
  num_training_steps = num_epochs * len(train_dataloader)
  lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps
  )

  progress_bar = tqdm(range(num_training_steps))

  model.train()
  for epoch in range(num_epochs):
    for batch in train_dataloader:
-     batch = {k: v.to(device) for k, v in batch.items()}
      outputs = model(**batch)
      loss = outputs.loss
-     loss.backward()
+     accelerator.backward(loss)

      optimizer.step()
      lr_scheduler.step()
      optimizer.zero_grad()
      progress_bar.update(1)

```

## Train

Once you've added the relevant lines of code, launch your training in a script or a notebook like Colaboratory.

### Train with a script

If you are running your training from a script, run the following command to create and save a configuration file:

```
accelerate config
```

Then launch your training with:

```
accelerate launch train.py
```

## Train with a notebook

😊 Accelerate can also run in a notebook if you're planning on using Colaboratory's TPUs. Wrap all the code responsible for training in a function, and pass it to [notebook\\_launcher](#):

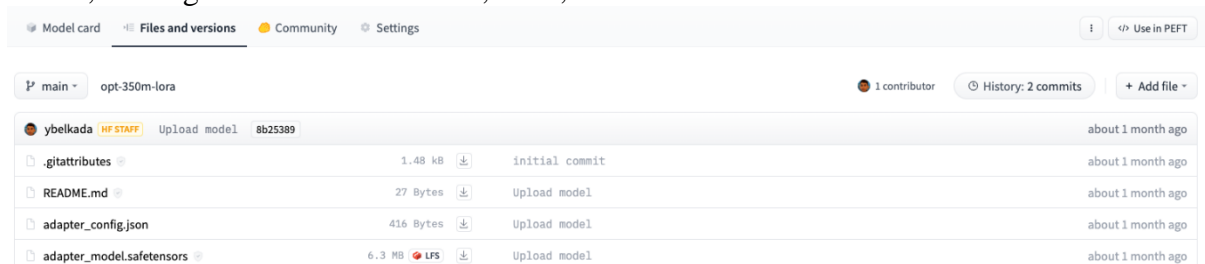
```
>>> from accelerate import notebook_launcher
```

```
>>> notebook_launcher(training_function)
```

## Load adapters with 😊 PEFT

[Parameter-Efficient Fine Tuning \(PEFT\)](#) methods freeze the pretrained model parameters during fine-tuning and add a small number of trainable parameters (the adapters) on top of it. The adapters are trained to learn task-specific information. This approach has been shown to be very memory-efficient with lower compute usage while producing results comparable to a fully fine-tuned model.

Adapters trained with PEFT are also usually an order of magnitude smaller than the full model, making it convenient to share, store, and load them.



Model card	Files and versions	Community	Settings
main		opt-350m-lora	1 contributor
History: 2 commits		+ Add file	
ybelkada	HF STAFF	Upload model	8b25389
.gitattributes	1.48 kB	initial commit	about 1 month ago
README.md	27 Bytes	Upload model	about 1 month ago
adapter_config.json	416 Bytes	Upload model	about 1 month ago
adapter_model.safetensors	6.3 MB	Upload model	about 1 month ago

The adapter weights for a OPTForCausalLM model stored on the Hub are only ~6MB compared to the full size of the model weights, which can be ~700MB.

If you're interested in learning more about the 😊 PEFT library, check out the [documentation](#).

### Setup

Get started by installing 😊 PEFT:

```
pip install peft
```

If you want to try out the brand new features, you might be interested in installing the library from source:

```
pip install git+https://github.com/huggingface/peft.git
```

### Supported PEFT models

😊 Transformers natively supports some PEFT methods, meaning you can load adapter weights stored locally or on the Hub and easily run or train them with a few lines of code. The following methods are supported:

- [Low Rank Adapters](#)
- [IA3](#)
- [AdaLoRA](#)

If you want to use other PEFT methods, such as prompt learning or prompt tuning, or learn about the 🤗 PEFT library in general, please refer to the [documentation](#).

### **Load a PEFT adapter**

To load and use a PEFT adapter model from 🤗 Transformers, make sure the Hub repository or local directory contains an adapter\_config.json file and the adapter weights, as shown in the example image above. Then you can load the PEFT adapter model using the AutoModelFor class. For example, to load a PEFT adapter model for causal language modeling:

1. specify the PEFT model id
2. pass it to the [AutoModelForCausalLM](#) class

```
from transformers import AutoModelForCausalLM, AutoTokenizer
```

```
peft_model_id = "ybelkada/opt-350m-lora"
```

```
model = AutoModelForCausalLM.from_pretrained(peft_model_id)
```

You can load a PEFT adapter with either an AutoModelFor class or the base model class like OPTForCausalLM or LlamaForCausalLM.

You can also load a PEFT adapter by calling the load\_adapter method:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
```

```
model_id = "facebook/opt-350m"
```

```
peft_model_id = "ybelkada/opt-350m-lora"
```

```
model = AutoModelForCausalLM.from_pretrained(model_id)
```

```
model.load_adapter(peft_model_id)
```

Check out the [API documentation](#) section below for more details.

### **Load in 8bit or 4bit**

The bitsandbytes integration supports 8bit and 4bit precision data types, which are useful for loading large models because it saves memory (see the bitsandbytes integration [guide](#) to learn more). Add the load\_in\_8bit or load\_in\_4bit parameters to [from\\_pretrained\(\)](#) and set device\_map="auto" to effectively distribute the model to your hardware:

```
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
```

```
peft_model_id = "ybelkada/opt-350m-lora"
```

```
model = AutoModelForCausalLM.from_pretrained(peft_model_id,  
quantization_config=BitsAndBytesConfig(load_in_8bit=True))
```

### **Add a new adapter**

You can use ~peft.PeftModel.add\_adapter to add a new adapter to a model with an existing adapter as long as the new adapter is the same type as the current one. For example, if you have an existing LoRA adapter attached to a model:

```
from transformers import AutoModelForCausalLM, OPTForCausalLM, AutoTokenizer  
from peft import LoraConfig
```

```
model_id = "facebook/opt-350m"
```

```
model = AutoModelForCausalLM.from_pretrained(model_id)
```

```
lora_config = LoraConfig(
```

```

    target_modules=["q_proj", "k_proj"],
    init_lora_weights=False
)

```

```

model.add_adapter(lora_config, adapter_name="adapter_1")

```

To add a new adapter:

```

# attach new adapter with same config

```

```

model.add_adapter(lora_config, adapter_name="adapter_2")

```

Now you can use `~peft.PeftModel.set_adapter` to set which adapter to use:

```

# use adapter_1

```

```

model.set_adapter("adapter_1")

```

```

output_disabled = model.generate(**inputs)

```

```

print(tokenizer.decode(output_disabled[0], skip_special_tokens=True))

```

```

# use adapter_2

```

```

model.set_adapter("adapter_2")

```

```

output_enabled = model.generate(**inputs)

```

```

print(tokenizer.decode(output_enabled[0], skip_special_tokens=True))

```

### **Enable and disable adapters**

Once you've added an adapter to a model, you can enable or disable the adapter module. To enable the adapter module:

```

from transformers import AutoModelForCausalLM, OPTForCausalLM, AutoTokenizer
from peft import PeftConfig

```

```

model_id = "facebook/opt-350m"

```

```

adapter_model_id = "ybelkada/opt-350m-lora"

```

```

tokenizer = AutoTokenizer.from_pretrained(model_id)

```

```

text = "Hello"

```

```

inputs = tokenizer(text, return_tensors="pt")

```

```

model = AutoModelForCausalLM.from_pretrained(model_id)

```

```

peft_config = PeftConfig.from_pretrained(adapter_model_id)

```

```

# to initiate with random weights

```

```

peft_config.init_lora_weights = False

```

```

model.add_adapter(peft_config)

```

```

model.enable_adapters()

```

```

output = model.generate(**inputs)

```

To disable the adapter module:

```

model.disable_adapters()

```

```

output = model.generate(**inputs)

```

### **Train a PEFT adapter**

PEFT adapters are supported by the [Trainer](#) class so that you can train an adapter for your specific use case. It only requires adding a few more lines of code. For example, to train a LoRA adapter:

If you aren't familiar with fine-tuning a model with [Trainer](#), take a look at the [Fine-tune a pretrained model](#) tutorial.

1. Define your adapter configuration with the task type and hyperparameters (see `~peft.LoraConfig` for more details about what the hyperparameters do).

```
from peft import LoraConfig
```

```
peft_config = LoraConfig(
    lora_alpha=16,
    lora_dropout=0.1,
    r=64,
    bias="none",
    task_type="CAUSAL_LM",
)
```

2. Add adapter to the model.

```
model.add_adapter(peft_config)
```

3. Now you can pass the model to [Trainer](#)!

```
trainer = Trainer(model=model, ...)
```

```
trainer.train()
```

To save your trained adapter and load it back:

```
model.save_pretrained(save_dir)
```

```
model = AutoModelForCausalLM.from_pretrained(save_dir)
```

### **Add additional trainable layers to a PEFT adapter**

You can also fine-tune additional trainable adapters on top of a model that has adapters attached by passing `modules_to_save` in your PEFT config. For example, if you want to also fine-tune the `lm_head` on top of a model with a LoRA adapter:

```
from transformers import AutoModelForCausalLM, OPTForCausalLM, AutoTokenizer
from peft import LoraConfig
```

```
model_id = "facebook/opt-350m"
```

```
model = AutoModelForCausalLM.from_pretrained(model_id)
```

```
lora_config = LoraConfig(
    target_modules=["q_proj", "k_proj"],
    modules_to_save=["lm_head"],
)
```

```
model.add_adapter(lora_config)
```

### **API docs**

**`class transformers.integrations.PeftAdapterMixin`**

[< source >](#)

`()`

A class containing all functions for loading and using adapters weights that are supported in PEFT library. For more details about adapters and injecting them on a transformer-based model, check out the documentation of PEFT library: <https://huggingface.co/docs/peft/index>

Currently supported PEFT methods are all non-prefix tuning methods. Below is the list of supported PEFT methods that anyone can load, train and run with this mixin class:

- Low Rank Adapters (LoRA): [https://huggingface.co/docs/peft/conceptual\\_guides/lora](https://huggingface.co/docs/peft/conceptual_guides/lora)
- IA3: [https://huggingface.co/docs/peft/conceptual\\_guides/ia3](https://huggingface.co/docs/peft/conceptual_guides/ia3)
- AdaLora: <https://arxiv.org/abs/2303.10512>

Other PEFT models such as prompt tuning, prompt learning are out of scope as these adapters are not “injectable” into a torch module. For using these methods, please refer to the usage guide of PEFT library.

With this mixin, if the correct PEFT version is installed, it is possible to:

- Load an adapter stored on a local path or in a remote Hub repository, and inject it in the model
- Attach new adapters in the model and train them with Trainer or by your own.
- Attach multiple adapters and iteratively activate / deactivate them
- Activate / deactivate all adapters from the model.
- Get the state\_dict of the active adapter.

### load\_adapter

[< source >](#)

( peft\_model\_id: Optional = None adapter\_name: Optional = None revision: Optional = None token: Optional = None device\_map: Optional = 'auto' max\_memory: Optional = None offload\_folder: Optional = None offload\_index: Optional = None peft\_config: Dict = None adapter\_state\_dict: Optional = None adapter\_kwargs: Optional = None )

Parameters

- **peft\_model\_id** (str, *optional*) — The identifier of the model to look for on the Hub, or a local path to the saved adapter config file and adapter weights.
- **adapter\_name** (str, *optional*) — The adapter name to use. If not set, will use the default adapter.
- **revision** (str, *optional*, defaults to "main") — The specific model version to use. It can be a branch name, a tag name, or a commit id, since we use a git-based system for storing models and other artifacts on huggingface.co, so revision can be any identifier allowed by git.

To test a pull request you made on the Hub, you can pass `revision="refs/pr/"`.

- **token** (str, *optional*) — Whether to use authentication token to load the remote folder. Useful to load private repositories that are on HuggingFace Hub. You might need to call huggingface-cli login and paste your tokens to cache it.
- **device\_map** (str or Dict[str, Union[int, str, torch.device]] or int or torch.device, *optional*) — A map that specifies where each submodule should go. It doesn't need to be refined to each parameter/buffer name, once a given module name is inside, every submodule of it will be sent to the same device. If we only pass the device (*e.g.*, "cpu", "cuda:1", "mps", or a GPU ordinal rank like 1) on which the model will be allocated, the device map will map the entire model to this device. Passing device\_map = 0 means put the whole model on GPU 0.

To have Accelerate compute the most optimized device\_map automatically,

set device\_map="auto". For more information about each option see [designing a device map](#).

- **max\_memory** (Dict, *optional*) — A dictionary device identifier to maximum memory. Will default to the maximum memory available for each GPU and the available CPU RAM if unset.
- **offload\_folder** (str or os.PathLike, *optional*) — If the device\_map contains any value "disk", the folder where we will offload weights.
- **offload\_index** (int, *optional*) — offload\_index argument to be passed to accelerate.dispatch\_model method.



- **peft\_config** (Dict[str, Any], *optional*) — The configuration of the adapter to add, supported adapters are non-prefix tuning and adaption prompts methods. This argument is used in case users directly pass PEFT state dicts
- **adapter\_state\_dict** (Dict[str, torch.Tensor], *optional*) — The state dict of the adapter to load. This argument is used in case users directly pass PEFT state dicts
- **adapter\_kwargs** (Dict[str, Any], *optional*) — Additional keyword arguments passed along to the `from_pretrained` method of the adapter config and `find_adapter_config_file` method.

Load adapter weights from file or remote Hub folder. If you are not familiar with adapters and PEFT methods, we invite you to read more about them on PEFT official documentation: <https://huggingface.co/docs/peft>

Requires `peft` as a backend to load the adapter weights.

### **add\_adapter**

[< source >](#)

( adapter\_config adapter\_name: Optional = None )

Parameters

- **adapter\_config** (~peft.PeftConfig) — The configuration of the adapter to add, supported adapters are non-prefix tuning and adaption prompts methods
- **adapter\_name** (str, *optional*, defaults to "default") — The name of the adapter to add. If no name is passed, a default name is assigned to the adapter.

If you are not familiar with adapters and PEFT methods, we invite you to read more about them on the PEFT official documentation: <https://huggingface.co/docs/peft>

Adds a fresh new adapter to the current model for training purpose. If no adapter name is passed, a default name is assigned to the adapter to follow the convention of PEFT library (in PEFT we use “default” as the default adapter name).

### **set\_adapter**

[< source >](#)

( adapter\_name: Union )

Parameters

- **adapter\_name** (Union[List[str], str]) — The name of the adapter to set. Can be also a list of strings to set multiple adapters.

If you are not familiar with adapters and PEFT methods, we invite you to read more about them on the PEFT official documentation: <https://huggingface.co/docs/peft>

Sets a specific adapter by forcing the model to use a that adapter and disable the other adapters.

### **disable\_adapters**

[< source >](#)

()

If you are not familiar with adapters and PEFT methods, we invite you to read more about them on the PEFT official documentation: <https://huggingface.co/docs/peft>

Disable all adapters that are attached to the model. This leads to inferring with the base model only.

### **enable\_adapters**

[< source >](#)

()

If you are not familiar with adapters and PEFT methods, we invite you to read more about them on the PEFT official documentation: <https://huggingface.co/docs/peft>

Enable adapters that are attached to the model. The model will use `self.active_adapter()`

### **active\_adapters**

[< source >](#)

()

If you are not familiar with adapters and PEFT methods, we invite you to read more about them on the PEFT official documentation: <https://huggingface.co/docs/peft>

Gets the current active adapters of the model. In case of multi-adapter inference (combining multiple adapters for inference) returns the list of all active adapters so that users can deal with them accordingly.

For previous PEFT versions (that does not support multi-adapter inference), `module.active_adapter` will return a single string.

**get\_adapter\_state\_dict**

[< source >](#)

( adapter\_name: Optional = None )

Parameters

- **adapter\_name** (str, *optional*) — The name of the adapter to get the state dict from. If no name is passed, the active adapter is used.

If you are not familiar with adapters and PEFT methods, we invite you to read more about them on the PEFT official documentation: <https://huggingface.co/docs/peft>

Gets the adapter state dict that should only contain the weights tensors of the specified adapter\_name adapter. If no adapter\_name is passed, the active adapter is used.

## Share a model

The last two tutorials showed how you can fine-tune a model with PyTorch, Keras, and 🤗 Accelerate for distributed setups. The next step is to share your model with the community! At Hugging Face, we believe in openly sharing knowledge and resources to democratize artificial intelligence for everyone. We encourage you to consider sharing your model with the community to help others save time and resources.

In this tutorial, you will learn two methods for sharing a trained or fine-tuned model on the [Model Hub](#):

- Programmatically push your files to the Hub.
- Drag-and-drop your files to the Hub with the web interface.

To share a model with the community, you need an account on [huggingface.co](https://huggingface.co). You can also join an existing organization or create a new one.

## Repository features

Each repository on the Model Hub behaves like a typical GitHub repository. Our repositories offer versioning, commit history, and the ability to visualize differences.


The Model Hub's built-in versioning is based on git and [git-lfs](#). In other words, you can treat one model as one repository, enabling greater access control and scalability. Version control allows *revisions*, a method for pinning a specific version of a model with a commit hash, tag or branch.

As a result, you can load a specific model version with the revision parameter:

```
>>> model = AutoModel.from_pretrained(  
...     "julien-c/EsperBERTo-small", revision="v2.0.1" # tag name, or branch name, or  
...     commit hash  
... )
```

Files are also easily edited in a repository, and you can view the commit history as well as the differences:

## Setup

Before sharing a model to the Hub, you will need your Hugging Face credentials. If you have access to a terminal, run the following command in the virtual environment where  Transformers is installed. This will store your access token in your Hugging Face cache folder (~/.cache/ by default):

```
huggingface-cli login
```

If you are using a notebook like Jupyter or Colaboratory, make sure you have the [huggingface\\_hub](#) library installed. This library allows you to programmatically interact with the Hub.


```
pip install huggingface_hub
```

Then use `notebook_login` to sign-in to the Hub, and follow the link [here](#) to generate a token to login with:

```
>>> from huggingface_hub import notebook_login
```

```
>>> notebook_login()
```

### Convert a model for all frameworks

To ensure your model can be used by someone working with a different framework, we recommend you convert and upload your model with both PyTorch and TensorFlow checkpoints. While users are still able to load your model from a different framework if you skip this step, it will be slower because  Transformers will need to convert the checkpoint on-the-fly.

Converting a checkpoint for another framework is easy. Make sure you have PyTorch and TensorFlow installed (see [here](#) for installation instructions), and then find the specific model for your task in the other framework.

Pytorch

Hide Pytorch content

Specify `from_tf=True` to convert a checkpoint from TensorFlow to PyTorch:

```
>>> pt_model = DistilBertForSequenceClassification.from_pretrained("path/to/awesome-name-you-picked", from_tf=True)
```

```
>>> pt_model.save_pretrained("path/to/awesome-name-you-picked")
```

TensorFlow

Hide TensorFlow content

Specify `from_pt=True` to convert a checkpoint from PyTorch to TensorFlow:

```
>>> tf_model = TFDistilBertForSequenceClassification.from_pretrained("path/to/awesome-name-you-picked", from_pt=True)
```

Then you can save your new TensorFlow model with its new checkpoint:

```
>>> tf_model.save_pretrained("path/to/awesome-name-you-picked")
```

JAX

Hide JAX content

If a model is available in Flax, you can also convert a checkpoint from PyTorch to Flax:

```
>>> flax_model = FlaxDistilBertForSequenceClassification.from_pretrained(
...     "path/to/awesome-name-you-picked", from_pt=True
... )
```

## Push a model during training

Pytorch

Hide Pytorch content

Sharing a model to the Hub is as simple as adding an extra parameter or callback. Remember from the [fine-tuning tutorial](#), the [TrainingArguments](#) class is where you specify hyperparameters and additional training options. One of these training options includes the ability to push a model directly to the Hub. Set `push_to_hub=True` in your [TrainingArguments](#):

```
>>> training_args = TrainingArguments(output_dir="my-awesome-model",
push_to_hub=True)
```

Pass your training arguments as usual to [Trainer](#):

```
>>> trainer = Trainer(
...     model=model,
...     args=training_args,
...     train_dataset=small_train_dataset,
...     eval_dataset=small_eval_dataset,
...     compute_metrics=compute_metrics,
... )
```

After you fine-tune your model, call [push\\_to\\_hub\(\)](#) on [Trainer](#) to push the trained model to the Hub. 😊 Transformers will even automatically add training hyperparameters, training results and framework versions to your model card!

```
>>> trainer.push_to_hub()
```

TensorFlow

Hide TensorFlow content

Share a model to the Hub with [PushToHubCallback](#). In the [PushToHubCallback](#) function, add:

- An output directory for your model.
- A tokenizer.
- The `hub_model_id`, which is your Hub username and model name.

```
>>> from transformers import PushToHubCallback
```

```
>>> push_to_hub_callback = PushToHubCallback(
...     output_dir="./your_model_save_path", tokenizer=tokenizer, hub_model_id="your-
username/my-awesome-model"
... )
```

Add the callback to [fit](#), and 😊 Transformers will push the trained model to the Hub:

```
>>> model.fit(tf_train_dataset, validation_data=tf_validation_dataset, epochs=3,
callbacks=push_to_hub_callback)
```

### Use the `push_to_hub` function

You can also call `push_to_hub` directly on your model to upload it to the Hub. Specify your model name in `push_to_hub`:

```
>>> pt_model.push_to_hub("my-awesome-model")
```

This creates a repository under your username with the model name `my-awesome-model`. Users can now load your model with the `from_pretrained` function:

```
>>> from transformers import AutoModel
```

```
>>> model = AutoModel.from_pretrained("your_username/my-awesome-model")
```

If you belong to an organization and want to push your model under the organization name instead, just add it to the `repo_id`:

```
>>> pt_model.push_to_hub("my-awesome-org/my-awesome-model")
```

The `push_to_hub` function can also be used to add other files to a model repository. For example, add a tokenizer to a model repository:

```
>>> tokenizer.push_to_hub("my-awesome-model")
```

Or perhaps you'd like to add the TensorFlow version of your fine-tuned PyTorch model:

```
>>> tf_model.push_to_hub("my-awesome-model")
```

Now when you navigate to your Hugging Face profile, you should see your newly created model repository. Clicking on the **Files** tab will display all the files you've uploaded to the repository.

For more details on how to create and upload files to a repository, refer to the Hub documentation [here](#).

### Upload with the web interface

Users who prefer a no-code approach are able to upload a model through the Hub's web interface. Visit [huggingface.co/new](https://huggingface.co/new) to create a new repository:

From here, add some information about your model:

- Select the **owner** of the repository. This can be yourself or any of the organizations you belong to.
- Pick a name for your model, which will also be the repository name.
- Choose whether your model is public or private.
- Specify the license usage for your model.

Now click on the **Files** tab and click on the **Add file** button to upload a new file to your repository. Then drag-and-drop a file to upload and add a commit message.

### Add a model card

To make sure users understand your model's capabilities, limitations, potential biases and ethical considerations, please add a model card to your repository. The model card is defined in the README.md file. You can add a model card by:

- Manually creating and uploading a README.md file.
- Clicking on the **Edit model card** button in your model repository.

Take a look at the DistilBert [model card](#) for a good example of the type of information a model card should include. For more details about other options you can control in the README.md file such as a model's carbon footprint or widget examples, refer to the documentation [here](#).

## Agents and tools

## What is an agent?

Large Language Models (LLMs) trained to perform [causal language modeling](#) can tackle a wide range of tasks, but they often struggle with basic tasks like logic, calculation, and search. When prompted in domains in which they do not perform well, they often fail to generate the answer we expect them to.

One approach to overcome this weakness is to create an *agent*.

An agent is a system that uses an LLM as its engine, and it has access to functions called *tools*.

These *tools* are functions for performing a task, and they contain all necessary description for the agent to properly use them.

The agent can be programmed to:

- devise a series of actions/tools and run them all at once, like the [CodeAgent](#)
- plan and execute actions/tools one by one and wait for the outcome of each action before launching the next one, like the [ReactJsonAgent](#)

## Types of agents

### Code agent

This agent has a planning step, then generates python code to execute all its actions at once. It natively handles different input and output types for its tools, thus it is the recommended choice for multimodal tasks.

### React agents

This is the go-to agent to solve reasoning tasks, since the ReAct framework ([Yao et al., 2022](#)) makes it really efficient to think on the basis of its previous observations.

We implement two versions of ReactJsonAgent:

- [ReactJsonAgent](#) generates tool calls as a JSON in its output.
- [ReactCodeAgent](#) is a new type of ReactJsonAgent that generates its tool calls as blobs of code, which works really well for LLMs that have strong coding performance.

Read [Open-source LLMs as LangChain Agents](#) blog post to learn more about ReAct agents.

For example, here is how a ReAct Code agent would work its way through the following question.

```
>>> agent.run(
...     "How many more blocks (also denoted as layers) in BERT base encoder than the
...     encoder from the architecture proposed in Attention is All You Need?",
... )
=====New task=====
How many more blocks (also denoted as layers) in BERT base encoder than the encoder from
the architecture proposed in Attention is All You Need?
=====Agent is executing the code below:
bert_blocks = search(query="number of blocks in BERT base encoder")
print("BERT blocks:", bert_blocks)
=====
Print outputs:
BERT blocks: twelve encoder blocks

=====Agent is executing the code below:
attention_layer = search(query="number of layers in Attention is All You Need")
print("Attention layers:", attention_layer)
=====
Print outputs:
```

Attention layers: Encoder: The encoder is composed of a stack of  $N = 6$  identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise

====Agent is executing the code below:

```
bert_blocks = 12
attention_layers = 6
diff = bert_blocks - attention_layers
print("Difference in blocks:", diff)
final_answer(diff)
=====
```

Print outputs:

Difference in blocks: 6

Final answer: 6

### **How can I build an agent?**

To initialize an agent, you need these arguments:

- an LLM to power your agent - the agent is not exactly the LLM, it's more like the agent is a program that uses an LLM as its engine.
- a system prompt: what the LLM engine will be prompted with to generate its output
- a toolbox from which the agent pick tools to execute
- a parser to extract from the LLM output which tools are to call and with which arguments

Upon initialization of the agent system, the tool attributes are used to generate a tool description, then baked into the agent's system\_prompt to let it know which tools it can use and why.

To start with, please install the agents extras in order to install all default dependencies.

```
pip install transformers[agents]
```

Build your LLM engine by defining a `llm_engine` method which accepts a list of [messages](#) and returns text. This callable also needs to accept a stop argument that indicates when to stop generating.

```
from huggingface_hub import login, InferenceClient
```

```
login("<YOUR_HUGGINGFACEHUB_API_TOKEN>")
```

```
client = InferenceClient(model="meta-llama/Meta-Llama-3-70B-Instruct")
```

```
def llm_engine(messages, stop_sequences=["Task"]) -> str:
    response = client.chat_completion(messages, stop=stop_sequences, max_tokens=1000)
    answer = response.choices[0].message.content
    return answer
```

You could use any `llm_engine` method as long as:

1. it follows the [messages format](#) (List[Dict[str, str]]) for its input messages, and it returns a str.
2. it stops generating outputs at the sequences passed in the argument `stop_sequences`

Additionally, `llm_engine` can also take a grammar argument. In the case where you specify a grammar upon agent initialization, this argument will be passed to the calls to `llm_engine`,



with the grammar that you defined upon initialization, to allow [constrained generation](#) in order to force properly-formatted agent outputs.

You will also need a tools argument which accepts a list of Tools - it can be an empty list. You can also add the default toolbox on top of your tools list by defining the optional argument `add_base_tools=True`.

Now you can create an agent, like [CodeAgent](#), and run it. You can also create a [TransformersEngine](#) with a pre-initialized pipeline to run inference on your local machine using transformers. For convenience, since agentic behaviours generally require stronger models such as Llama-3.1-70B-Instruct that are harder to run locally for now, we also provide the [HfApiEngine](#) class that initializes a `huggingface_hub.InferenceClient` under the hood.

```
from transformers import CodeAgent, HfApiEngine
```

```
llm_engine = HfApiEngine(model="meta-llama/Meta-Llama-3-70B-Instruct")
agent = CodeAgent(tools=[], llm_engine=llm_engine, add_base_tools=True)
```

```
agent.run(
    "Could you translate this sentence from French, say it out loud and return the audio.",
    sentence="Où est la boulangerie la plus proche?",
)
```

This will be handy in case of emergency baguette need! You can even leave the argument `llm_engine` undefined, and an [HfApiEngine](#) will be created by default.

```
from transformers import CodeAgent
```

```
agent = CodeAgent(tools=[], add_base_tools=True)
```

```
agent.run(
    "Could you translate this sentence from French, say it out loud and give me the audio.",
    sentence="Où est la boulangerie la plus proche?",
)
```

Note that we used an additional sentence argument: you can pass text as additional arguments to the model.

You can also use this to indicate the path to local or remote files for the model to use:

```
from transformers import ReactCodeAgent
```

```
agent = ReactCodeAgent(tools=[], llm_engine=llm_engine, add_base_tools=True)
```

```
agent.run("Why does Mike not know many people in New York?",
audio="https://huggingface.co/datasets/huggingface/documentation-images/resolve/main/transformers/recording.mp3")
```

The prompt and output parser were automatically defined, but you can easily inspect them by calling the `system_prompt_template` on your agent.

```
print(agent.system_prompt_template)
```

It's important to explain as clearly as possible the task you want to perform.

Every [run\(\)](#) operation is independent, and since an agent is powered by an LLM, minor variations in your prompt might yield completely different results. You can also run an agent



consecutively for different tasks: each time the attributes `agent.task` and `agent.logs` will be re-initialized.

### Code execution

A Python interpreter executes the code on a set of inputs passed along with your tools. This should be safe because the only functions that can be called are the tools you provided (especially if it's only tools by Hugging Face) and the print function, so you're already limited in what can be executed.

The Python interpreter also doesn't allow imports by default outside of a safe list, so all the most obvious attacks shouldn't be an issue. You can still authorize additional imports by passing the authorized modules as a list of strings in argument `additional_authorized_imports` upon initialization of your [ReactCodeAgent](#) or [CodeAgent](#):

```
>>> from transformers import ReactCodeAgent

>>> agent = ReactCodeAgent(tools=[], additional_authorized_imports=['requests', 'bs4'])
>>> agent.run("Could you get me the title of the page at url 'https://huggingface.co/blog'?")
```

(...)

'Hugging Face – Blog'

The execution will stop at any code trying to perform an illegal operation or if there is a regular Python error with the code generated by the agent.

The LLM can generate arbitrary code that will then be executed: do not add any unsafe imports!

### The system prompt

An agent, or rather the LLM that drives the agent, generates an output based on the system prompt. The system prompt can be customized and tailored to the intended task. For example, check the system prompt for the [ReactCodeAgent](#) (below version is slightly simplified).

You will be given a task to solve as best you can.

You have access to the following tools:

<<tool\_descriptions>>

To solve the task, you must plan forward to proceed in a series of steps, in a cycle of 'Thought:', 'Code:', and 'Observation:' sequences.

At each step, in the 'Thought:' sequence, you should first explain your reasoning towards solving the task, then the tools that you want to use.

Then in the 'Code:' sequence, you should write the code in simple Python. The code sequence must end with '/End code' sequence.

During each intermediate step, you can use 'print()' to save whatever important information you will then need.

These print outputs will then be available in the 'Observation:' field, for using this information as input for the next step.

In the end you have to return a final answer using the ``final_answer`` tool.

Here are a few examples using notional tools:

---

{examples}

Above example were using notional tools that might not exist for you. You only have access to those tools:

<<tool\_names>>

You also can perform computations in the python code you generate.

Always provide a 'Thought:' and a 'Code:\n```py' sequence ending with ```<end\_code>' sequence. You MUST provide at least the 'Code:' sequence to move forward.

Remember to not perform too many operations in a single code block! You should split the task into intermediate code blocks.

Print results at the end of each step to save the intermediate results. Then use `final_answer()` to return the final result.

Remember to make sure that variables you use are all defined.

Now Begin!

The system prompt includes:

- An *introduction* that explains how the agent should behave and what tools are.
- A description of all the tools that is defined by a <<tool\_descriptions>> token that is dynamically replaced at runtime with the tools defined/chosen by the user.
  - The tool description comes from the tool attributes, name, description, inputs and output\_type, and a simple jinja2 template that you can refine.
- The expected output format.

You could improve the system prompt, for example, by adding an explanation of the output format.

For maximum flexibility, you can overwrite the whole system prompt template by passing your custom prompt as an argument to the `system_prompt` parameter.

```
from transformers import ReactJsonAgent
from transformers.agents import PythonInterpreterTool
```

```
agent = ReactJsonAgent(tools=[PythonInterpreterTool()],
system_prompt="{your_custom_prompt}")
```

Please make sure to define the <<tool\_descriptions>> string somewhere in the template so the agent is aware of the available tools.

### Inspecting an agent run

Here are a few useful attributes to inspect what happened after a run:

- `agent.logs` stores the fine-grained logs of the agent. At every step of the agent's run, everything gets stored in a dictionary that then is appended to `agent.logs`.
- Running `agent.write_inner_memory_from_logs()` creates an inner memory of the agent's logs for the LLM to view, as a list of chat messages. This method goes over each step of the log and only stores what it's interested in as a message: for instance, it will save the system prompt and task in separate messages, then for each step it will store the LLM output as a message, and the tool call output as another message. Use this if you want a higher-level view of what has happened - but not every log will be transcribed by this method.

### Tools

A tool is an atomic function to be used by an agent.

You can for instance check the `PythonInterpreterTool`: it has a name, a description, input descriptions, an output type, and a `__call__` method to perform the action.

When the agent is initialized, the tool attributes are used to generate a tool description which is baked into the agent's system prompt. This lets the agent know which tools it can use and why.

### Default toolbox

Transformers comes with a default toolbox for empowering agents, that you can add to your agent upon initialization with argument `add_base_tools = True`:

- **Document question answering**: given a document (such as a PDF) in image format, answer a question on this document ([Donut](#))
- **Image question answering**: given an image, answer a question on this image ([VILT](#))
- **Speech to text**: given an audio recording of a person talking, transcribe the speech into text ([Whisper](#))
- **Text to speech**: convert text to speech ([SpeechT5](#))
- **Translation**: translates a given sentence from source language to target language.
- **DuckDuckGo search\***: performs a web search using DuckDuckGo browser.
- **Python code interpreter**: runs your the LLM generated Python code in a secure environment. This tool will only be added to [ReactJsonAgent](#) if you initialize it with `add_base_tools=True`, since code-based agent can already natively execute Python code

You can manually use a tool by calling the [load\\_tool\(\)](#) function and a task to perform.

```
from transformers import load_tool
```

```
tool = load_tool("text-to-speech")
audio = tool("This is a text to speech tool")
```

### Create a new tool

You can create your own tool for use cases not covered by the default tools from Hugging Face. For example, let's create a tool that returns the most downloaded model for a given task from the Hub.

You'll start with the code below.

```
from huggingface_hub import list_models
```

```
task = "text-classification"
```

```
model = next(iter(list_models(filter=task, sort="downloads", direction=-1)))
print(model.id)
```

This code can quickly be converted into a tool, just by wrapping it in a function and adding the tool decorator:

```
from transformers import tool
```

```
@tool
def model_download_counter(task: str) -> str:
    """
```

This is a tool that returns the most downloaded model of a given task on the Hugging Face Hub.

It returns the name of the checkpoint.

Args:

task: The task for which

"""

```
model = next(iter(list_models(filter="text-classification", sort="downloads", direction=-1)))
```

```
return model.id
```

The function needs:

- A clear name. The name usually describes what the tool does. Since the code returns the model with the most downloads for a task, let's put `model_download_counter`.
- Type hints on both inputs and output
- A description, that includes an 'Args:' part where each argument is described (without a type indication this time, it will be pulled from the type hint). All these will be automatically baked into the agent's system prompt upon initialization: so strive to make them as clear as possible!

This definition format is the same as tool schemas used in `apply_chat_template`, the only difference is the added tool decorator: read more on our tool use API [here](#).

Then you can directly initialize your agent:

```
from transformers import CodeAgent
```

```
agent = CodeAgent(tools=[model_download_tool], llm_engine=llm_engine)
```

```
agent.run(
```

```
"Can you give me the name of the model that has the most downloads in the 'text-to-video' task on the Hugging Face Hub?"
```

```
)
```

You get the following:

```
===== New task =====
```

```
Can you give me the name of the model that has the most downloads in the 'text-to-video' task on the Hugging Face Hub?
```

```
==== Agent is executing the code below:
```

```
most_downloaded_model = model_download_counter(task="text-to-video")
```

```
print(f"The most downloaded model for the 'text-to-video' task is
```

```
{most_downloaded_model}.")
```

```
====
```

```
And the output: "The most downloaded model for the 'text-to-video' task is
```

```
ByteDance/AnimateDiff-Lightning."
```

### **Manage your agent's toolbox**

If you have already initialized an agent, it is inconvenient to reinitialize it from scratch with a tool you want to use. With Transformers, you can manage an agent's toolbox by adding or replacing a tool.

Let's add the `model_download_tool` to an existing agent initialized with only the default toolbox.

```
from transformers import CodeAgent
```

```
agent = CodeAgent(tools=[], llm_engine=llm_engine, add_base_tools=True)
```

```
agent.toolbox.add_tool(model_download_tool)
```

Now we can leverage both the new tool and the previous text-to-speech tool:

```
agent.run(
```

"Can you read out loud the name of the model that has the most downloads in the 'text-to-video' task on the Hugging Face Hub and return the audio?"  
)

### **Audio**

Beware when adding tools to an agent that already works well because it can bias selection towards your tool or select another tool other than the one already defined.

Use the `agent.toolbox.update_tool()` method to replace an existing tool in the agent's toolbox. This is useful if your new tool is a one-to-one replacement of the existing tool because the agent already knows how to perform that specific task. Just make sure the new tool follows the same API as the replaced tool or adapt the system prompt template to ensure all examples using the replaced tool are updated.

### **Use a collection of tools**

You can leverage tool collections by using the `ToolCollection` object, with the slug of the collection you want to use. Then pass them as a list to initialize you agent, and start using them!

```
from transformers import ToolCollection, ReactCodeAgent
```

```
image_tool_collection = ToolCollection(collection_slug="huggingface-tools/diffusion-tools-6630bb19a942c2306a2cdb6f")
```

```
agent = ReactCodeAgent(tools=[*image_tool_collection.tools], add_base_tools=True)
```

```
agent.run("Please draw me a picture of rivers and lakes.")
```

To speed up the start, tools are loaded only if called by the agent.

This gets you this image:

## **Agents, supercharged - Multi-agents, External tools, and more**

### **What is an agent?**

If you're new to `transformers.agents`, make sure to first read the main [agents documentation](#). In this page we're going to highlight several advanced uses of `transformers.agents`.

### **Multi-agents**

Multi-agent has been introduced in Microsoft's framework [Autogen](#). It simply means having several agents working together to solve your task instead of only one. It empirically yields better performance on most benchmarks. The reason for this better performance is conceptually simple: for many tasks, rather than using a do-it-all system, you would prefer to specialize units on sub-tasks. Here, having agents with separate tool sets and memories allows to achieve efficient specialization.

You can easily build hierarchical multi-agent systems with `transformers.agents`.

To do so, encapsulate the agent in a [ManagedAgent](#) object. This object needs arguments `agent`, `name`, and a `description`, which will then be embedded in the manager agent's system prompt to let it know how to call this managed agent, as we also do for tools.

Here's an example of making an agent that managed a specific web search agent using our `DuckDuckGoSearchTool`:

```
from transformers.agents import ReactCodeAgent, HfApiEngine, DuckDuckGoSearchTool, ManagedAgent
```

```

llm_engine = HfApiEngine()

web_agent = ReactCodeAgent(tools=[DuckDuckGoSearchTool()], llm_engine=llm_engine)

managed_web_agent = ManagedAgent(
    agent=web_agent,
    name="web_search",
    description="Runs web searches for you. Give it your query as an argument."
)

manager_agent = ReactCodeAgent(
    tools=[], llm_engine=llm_engine, managed_agents=[managed_web_agent]
)

```

manager\_agent.run("Who is the CEO of Hugging Face?")

For an in-depth example of an efficient multi-agent implementation, see [how we pushed our multi-agent system to the top of the GAIA leaderboard](#).

### Advanced tool usage

#### Directly define a tool by subclassing Tool, and share it to the Hub

Let's take again the tool example from main documentation, for which we had implemented a tool decorator.

If you need to add variation, like custom attributes for your tool, you can build your tool following the fine-grained method: building a class that inherits from the [Tool](#) superclass.

The custom tool needs:

- An attribute name, which corresponds to the name of the tool itself. The name usually describes what the tool does. Since the code returns the model with the most downloads for a task, let's name it `model_download_counter`.
- An attribute description is used to populate the agent's system prompt.
- An inputs attribute, which is a dictionary with keys "type" and "description". It contains information that helps the Python interpreter make educated choices about the input.
- An output\_type attribute, which specifies the output type.
- A forward method which contains the inference code to be executed.

The types for both inputs and output\_type should be amongst [Pydantic formats](#).

```

from transformers import Tool
from huggingface_hub import list_models

```

```

class HFModelDownloadsTool(Tool):
    name = "model_download_counter"
    description = """

```

This is a tool that returns the most downloaded model of a given task on the Hugging Face Hub.

It returns the name of the checkpoint."""

```

inputs = {
    "task": {
        "type": "string",
        "description": "the task category (such as text-classification, depth-estimation, etc)",

```

```

    }
}
output_type = "string"

def forward(self, task: str):
    model = next(iter(list_models(filter=task, sort="downloads", direction=-1)))
    return model.id

```

Now that the custom HfModelDownloadsTool class is ready, you can save it to a file named `model_downloads.py` and import it for use.

```
from model_downloads import HFModelDownloadsTool
```

```
tool = HFModelDownloadsTool()
```

You can also share your custom tool to the Hub by calling [push to hub\(\)](#) on the tool. Make sure you've created a repository for it on the Hub and are using a token with read access.

```
tool.push_to_hub("{your_username}/hf-model-downloads")
```

Load the tool with the `~Tool.load_tool` function and pass it to the `tools` parameter in your agent.

```
from transformers import load_tool, CodeAgent
```

```
model_download_tool = load_tool("m-ric/hf-model-downloads")
```

### Use gradio-tools

[gradio-tools](#) is a powerful library that allows using Hugging Face Spaces as tools. It supports many existing Spaces as well as custom Spaces.

Transformers supports `gradio_tools` with the [Tool.from\\_gradio\(\)](#) method. For example, let's use the [StableDiffusionPromptGeneratorTool](#) from `gradio-tools` toolkit for improving prompts to generate better images.

Import and instantiate the tool, then pass it to the `Tool.from_gradio` method:

```
from gradio_tools import StableDiffusionPromptGeneratorTool
from transformers import Tool, load_tool, CodeAgent
```

```
gradio_prompt_generator_tool = StableDiffusionPromptGeneratorTool()
prompt_generator_tool = Tool.from_gradio(gradio_prompt_generator_tool)
```

Now you can use it just like any other tool. For example, let's improve the prompt a rabbit wearing a space suit.

```
image_generation_tool = load_tool('huggingface-tools/text-to-image')
agent = CodeAgent(tools=[prompt_generator_tool, image_generation_tool],
llm_engine=llm_engine)
```

```
agent.run(
    "Improve this prompt, then generate an image of it.", prompt='A rabbit wearing a space suit'
)
```

The model adequately leverages the tool:

```
===== New task =====
```

Improve this prompt, then generate an image of it.

You have been provided with these initial arguments: {'prompt': 'A rabbit wearing a space suit'}.

==== Agent is executing the code below:

```
improved_prompt = StableDiffusionPromptGenerator(query=prompt)
while improved_prompt == "QUEUE_FULL":
    improved_prompt = StableDiffusionPromptGenerator(query=prompt)
print(f"The improved prompt is {improved_prompt}.")
image = image_generator(prompt=improved_prompt)
=====
```

Before finally generating the image:

gradio-tools require *textual* inputs and outputs even when working with different modalities like image and audio objects. Image and audio inputs and outputs are currently incompatible.

### Use LangChain tools

We love Langchain and think it has a very compelling suite of tools. To import a tool from LangChain, use the `from_langchain()` method.

Here is how you can use it to recreate the intro's search result using a LangChain web search tool.

```
from langchain.agents import load_tools
from transformers import Tool, ReactCodeAgent

search_tool = Tool.from_langchain(load_tools(["serpapi"])[0])

agent = ReactCodeAgent(tools=[search_tool])
```

`agent.run("How many more blocks (also denoted as layers) in BERT base encoder than the encoder from the architecture proposed in Attention is All You Need?")`

### Display your agent run in a cool Gradio interface

You can leverage `gradio.Chatbotto` display your agent's thoughts using `stream_to_gradio`, here is an example:

```
import gradio as gr
from transformers import (
    load_tool,
    ReactCodeAgent,
    HfApiEngine,
    stream_to_gradio,
)

# Import tool from Hub
image_generation_tool = load_tool("m-ric/text-to-image")

llm_engine = HfApiEngine("meta-llama/Meta-Llama-3-70B-Instruct")

# Initialize the agent with the image generation tool
agent = ReactCodeAgent(tools=[image_generation_tool], llm_engine=llm_engine)
```



```
def interact_with_agent(task):
    messages = []
    messages.append(gr.ChatMessage(role="user", content=task))
    yield messages
    for msg in stream_to_gradio(agent, task):
        messages.append(msg)
        yield messages + [
            gr.ChatMessage(role="assistant", content="⌚ Task not finished yet!")
        ]
    yield messages

with gr.Blocks() as demo:
    text_input = gr.Textbox(lines=1, label="Chat Message", value="Make me a picture of the Statue of Liberty.")
    submit = gr.Button("Run illustrator agent!")
    chatbot = gr.Chatbot(
        label="Agent",
        type="messages",
        avatar_images=(
            None,
            "https://em-content.zobj.net/source/twitter/53/robot-face_1f916.png",
        ),
    )
    submit.click(interact_with_agent, [text_input], [chatbot])

if __name__ == "__main__":
    demo.launch()
```

## Generation with LLMs



LLMs, or Large Language Models, are the key component behind text generation. In a nutshell, they consist of large pretrained transformer models trained to predict the next word (or, more precisely, token) given some input text. Since they predict one token at a time, you need to do something more elaborate to generate new sentences other than just calling the model — you need to do autoregressive generation.

Autoregressive generation is the inference-time procedure of iteratively calling a model with its own generated outputs, given a few initial inputs. In 🤖 Transformers, this is handled by the [generate\(\)](#) method, which is available to all models with generative capabilities.

This tutorial will show you how to:

- Generate text with an LLM
- Avoid common pitfalls
- Next steps to help you get the most out of your LLM

Before you begin, make sure you have all the necessary libraries installed:

```
pip install transformers bitsandbytes>=0.39.0 -q
```

### Generate text

A language model trained for [causal language modeling](#) takes a sequence of text tokens as input and returns the probability distribution for the next token.

"Forward pass of an LLM"

A critical aspect of autoregressive generation with LLMs is how to select the next token from this probability distribution. Anything goes in this step as long as you end up with a token for the next iteration. This means it can be as simple as selecting the most likely token from the probability distribution or as complex as applying a dozen transformations before sampling from the resulting distribution.

"Autoregressive generation iteratively selects the next token from a probability distribution to generate text"

The process depicted above is repeated iteratively until some stopping condition is reached. Ideally, the stopping condition is dictated by the model, which should learn when to output an end-of-sequence (EOS) token. If this is not the case, generation stops when some predefined maximum length is reached.

Properly setting up the token selection step and the stopping condition is essential to make your model behave as you'd expect on your task. That is why we have a [GenerationConfig](#) file associated with each model, which contains a good default generative parameterization and is loaded alongside your model.

Let's talk code!

If you're interested in basic LLM usage, our high-level [Pipeline](#) interface is a great starting point. However, LLMs often require advanced features like quantization and fine control of the token selection step, which is best done through [generate\(\)](#). Autoregressive generation with LLMs is also resource-intensive and should be executed on a GPU for adequate throughput.

First, you need to load the model.

```
>>> from transformers import AutoModelForCausalLM
```

```
>>> model = AutoModelForCausalLM.from_pretrained(
...     "mistralai/Mistral-7B-v0.1", device_map="auto", load_in_4bit=True
... )
```

You'll notice two flags in the `from_pretrained` call:

- `device_map` ensures the model is moved to your GPU(s)
- `load_in_4bit` applies [4-bit dynamic quantization](#) to massively reduce the resource requirements

There are other ways to initialize a model, but this is a good baseline to begin with an LLM. Next, you need to preprocess your text input with a [tokenizer](#).

```
>>> from transformers import AutoTokenizer
```

```
>>> tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1",
padding_side="left")
>>> model_inputs = tokenizer(["A list of colors: red, blue"], return_tensors="pt").to("cuda")
```

The `model_inputs` variable holds the tokenized text input, as well as the attention mask.

While [generate\(\)](#) does its best effort to infer the attention mask when it is not passed, we recommend passing it whenever possible for optimal results.

After tokenizing the inputs, you can call the [generate\(\)](#) method to return the generated tokens. The generated tokens then should be converted to text before printing.

```
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'A list of colors: red, blue, green, yellow, orange, purple, pink,'
```

Finally, you don't need to do it one sequence at a time! You can batch your inputs, which will greatly improve the throughput at a small latency and memory cost. All you need to do is to make sure you pad your inputs properly (more on that below).

```
>>> tokenizer.pad_token = tokenizer.eos_token # Most LLMs don't have a pad token by default
>>> model_inputs = tokenizer(
...     ["A list of colors: red, blue", "Portugal is"], return_tensors="pt", padding=True
... ).to("cuda")
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)
['A list of colors: red, blue, green, yellow, orange, purple, pink,',
'Portugal is a country in southwestern Europe, on the Iber']
```

And that's it! In a few lines of code, you can harness the power of an LLM.

### Common pitfalls

There are many [generation strategies](#), and sometimes the default values may not be appropriate for your use case. If your outputs aren't aligned with what you're expecting, we've created a list of the most common pitfalls and how to avoid them.

```
>>> from transformers import AutoModelForCausalLM, AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1")
>>> tokenizer.pad_token = tokenizer.eos_token # Most LLMs don't have a pad token by default
>>> model = AutoModelForCausalLM.from_pretrained(
...     "mistralai/Mistral-7B-v0.1", device_map="auto", load_in_4bit=True
... )
```

### Generated output is too short/long

If not specified in the [GenerationConfig](#) file, generate returns up to 20 tokens by default. We highly recommend manually setting `max_new_tokens` in your generate call to control the maximum number of new tokens it can return. Keep in mind LLMs (more precisely, [decoder-only models](#)) also return the input prompt as part of the output.

```
>>> model_inputs = tokenizer(["A sequence of numbers: 1, 2"],
return_tensors="pt").to("cuda")

>>> # By default, the output will contain up to 20 tokens
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'A sequence of numbers: 1, 2, 3, 4, 5'

>>> # Setting `max_new_tokens` allows you to control the maximum length
>>> generated_ids = model.generate(**model_inputs, max_new_tokens=50)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'A sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,'
```

### Incorrect generation mode

By default, and unless specified in the [GenerationConfig](#) file, generate selects the most likely token at each iteration (greedy decoding). Depending on your task, this may be undesirable; creative tasks like chatbots or writing an essay benefit from sampling. On the other hand, input-grounded tasks like audio transcription or translation benefit from greedy decoding. Enable sampling with `do_sample=True`, and you can learn more about this topic in this [blog post](#).

```
>>> # Set seed or reproducibility -- you don't need this unless you want full reproducibility
>>> from transformers import set_seed
>>> set_seed(42)
```

```
>>> model_inputs = tokenizer(["I am a cat."], return_tensors="pt").to("cuda")
```

```
>>> # LLM + greedy decoding = repetitive, boring output
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'I am a cat. I am a cat. I am a cat. I am a cat'
```

```
>>> # With sampling, the output becomes more creative!
>>> generated_ids = model.generate(**model_inputs, do_sample=True)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'I am a cat. Specifically, I am an indoor-only cat. I'
```

### **Wrong padding side**

LLMs are [decoder-only](#) architectures, meaning they continue to iterate on your input prompt. If your inputs do not have the same length, they need to be padded. Since LLMs are not trained to continue from pad tokens, your input needs to be left-padded. Make sure you also don't forget to pass the attention mask to generate!

```
>>> # The tokenizer initialized above has right-padding active by default: the 1st sequence,
>>> # which is shorter, has padding on the right side. Generation fails to capture the logic.
>>> model_inputs = tokenizer(
...     ["1, 2, 3", "A, B, C, D, E"], padding=True, return_tensors="pt"
... ).to("cuda")
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'1, 2, 333333333333'
```

```
>>> # With left-padding, it works as expected!
>>> tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1",
padding_side="left")
>>> tokenizer.pad_token = tokenizer.eos_token # Most LLMs don't have a pad token by
default
>>> model_inputs = tokenizer(
...     ["1, 2, 3", "A, B, C, D, E"], padding=True, return_tensors="pt"
... ).to("cuda")
>>> generated_ids = model.generate(**model_inputs)
>>> tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
'1, 2, 3, 4, 5, 6,'
```

### **Wrong prompt**

Some models and tasks expect a certain input prompt format to work properly. When this format is not applied, you will get a silent performance degradation: the model kinda works, but not as well as if you were following the expected prompt. More information about prompting, including which models and tasks need to be careful, is available in this [guide](#). Let's see an example with a chat LLM, which makes use of [chat templating](#):

```
>>> tokenizer = AutoTokenizer.from_pretrained("HuggingFaceH4/zephyr-7b-alpha")
>>> model = AutoModelForCausalLM.from_pretrained(
...     "HuggingFaceH4/zephyr-7b-alpha", device_map="auto", load_in_4bit=True
... )
>>> set_seed(0)
>>> prompt = """"How many helicopters can a human eat in one sitting? Reply as a thug.""
>>> model_inputs = tokenizer([prompt], return_tensors="pt").to("cuda")
>>> input_length = model_inputs.input_ids.shape[1]
>>> generated_ids = model.generate(**model_inputs, max_new_tokens=20)
>>> print(tokenizer.batch_decode(generated_ids[:, input_length:],
skip_special_tokens=True)[0])
'I'm not a thug, but i can tell you that a human cannot eat'
>>> # Oh no, it did not follow our instruction to reply as a thug! Let's see what happens when
we write
>>> # a better prompt and use the right template for this model (through
`tokenizer.apply_chat_template`)
```

```
>>> set_seed(0)
>>> messages = [
...     {
...         "role": "system",
...         "content": "You are a friendly chatbot who always responds in the style of a thug",
...     },
...     {"role": "user", "content": "How many helicopters can a human eat in one sitting?"},
... ]
>>> model_inputs = tokenizer.apply_chat_template(messages,
add_generation_prompt=True, return_tensors="pt").to("cuda")
>>> input_length = model_inputs.shape[1]
>>> generated_ids = model.generate(model_inputs, do_sample=True, max_new_tokens=20)
>>> print(tokenizer.batch_decode(generated_ids[:, input_length:],
skip_special_tokens=True)[0])
'None, you thug. How bout you try to focus on more useful questions?'
>>> # As we can see, it followed a proper thug style 😎
```

### Further resources

While the autoregressive generation process is relatively straightforward, making the most out of your LLM can be a challenging endeavor because there are many moving parts. For your next steps to help you dive deeper into LLM usage and understanding:

#### Advanced generate usage

1. Guide on how to [control different generation methods](#), how to set up the generation configuration file, and how to stream the output;
2. [Accelerating text generation](#);
3. [Prompt templates for chat LLMs](#);
4. [Prompt design guide](#);

5. API reference on [GenerationConfig](#), [generate\(\)](#), and [generate-related classes](#). Most of the classes, including the logits processors, have usage examples!

### LLM leaderboards

1. [Open LLM Leaderboard](#), which focuses on the quality of the open-source models;
2. [Open LLM-Perf Leaderboard](#), which focuses on LLM throughput.

### Latency, throughput and memory utilization

1. Guide on how to [optimize LLMs for speed and memory](#);
2. Guide on [quantization](#) such as bitsandbytes and autogptq, which shows you how to drastically reduce your memory requirements.

### Related libraries

1. [optimum](#), an extension of 🤗 Transformers that optimizes for specific hardware devices.
2. [outlines](#), a library where you can constrain text generation (e.g. to generate JSON files);
3. [SynCode](#), a library for context-free grammar guided generation. (e.g. JSON, SQL, Python)
4. [text-generation-inference](#), a production-ready server for LLMs;
5. [text-generation-webui](#), a UI for text generation;

## Chatting with Transformers

If you're reading this article, you're almost certainly aware of **chat models**. Chat models are conversational AIs that you can send and receive messages with. The most famous of these is the proprietary ChatGPT, but there are now many open-source chat models which match or even substantially exceed its performance. These models are free to download and run on a local machine. Although the largest and most capable models require high-powered hardware and lots of memory to run, there are smaller models that will run perfectly well on a single consumer GPU, or even an ordinary desktop or notebook CPU.

This guide will help you get started with chat models. We'll start with a brief quickstart guide that uses a convenient, high-level "pipeline". This is all you need if you just want to start running a chat model immediately. After the quickstart, we'll move on to more detailed information about what exactly chat models are, how to choose an appropriate one, and a low-level breakdown of each of the steps involved in talking to a chat model. We'll also give some tips on optimizing the performance and memory usage of your chat models.

### Quickstart

If you have no time for details, here's the brief summary: Chat models continue chats. This means that you pass them a conversation history, which can be as short as a single user message, and the model will continue the conversation by adding its response. Let's see this in action. First, let's build a chat:

```
chat = [  
    {"role": "system", "content": "You are a sassy, wise-cracking robot as imagined by  
Hollywood circa 1986."},  
    {"role": "user", "content": "Hey, can you tell me any fun things to do in New York?" }  
]
```

Notice that in addition to the user's message, we added a **system** message at the start of the conversation. Not all chat models support system messages, but when they do, they represent high-level directives about how the model should behave in the conversation. You can use this to guide the model - whether you want short or long responses, lighthearted or serious ones, and so on. If you want the model to do useful work instead of practicing its improv

routine, you can either omit the system message or try a terse one such as “You are a helpful and intelligent AI assistant who responds to user queries.”

Once you have a chat, the quickest way to continue it is using the [TextGenerationPipeline](#). Let’s see this in action with LLaMA-3. Note that LLaMA-3 is a gated model, which means you will need to [apply for access](#) and log in with your Hugging Face account to use it. We’ll also use `device_map="auto"`, which will load the model on GPU if there’s enough memory for it, and set the dtype to `torch.bfloat16` to save memory:

```
import torch
from transformers import pipeline

pipe = pipeline("text-generation", "meta-llama/Meta-Llama-3-8B-Instruct",
torch_dtype=torch.bfloat16, device_map="auto")
response = pipe(chat, max_new_tokens=512)
print(response[0]['generated_text'][-1]['content'])
And you’ll get:
```

(sigh) Oh boy, you're asking me for advice? You're gonna need a map, pal! Alright, alright, I'll give you the lowdown. But don't say I didn't warn you, I'm a robot, not a tour guide!

So, you wanna know what's fun to do in the Big Apple? Well, let me tell you, there's a million things to do, but I'll give you the highlights. First off, you gotta see the sights: the Statue of Liberty, Central Park, Times Square... you know, the usual tourist traps. But if you're lookin' for something a little more... unusual, I'd recommend checkin' out the Museum of Modern Art. It's got some wild stuff, like that Warhol guy's soup cans and all that jazz.

And if you're feelin' adventurous, take a walk across the Brooklyn Bridge. Just watch out for those pesky pigeons, they're like little feathered thieves! (laughs) Get it? Thieves? Ah, never mind.

Now, if you're lookin' for some serious fun, hit up the comedy clubs in Greenwich Village. You might even catch a glimpse of some up-and-coming comedians... or a bunch of wannabes tryin' to make it big. (winks)

And finally, if you're feelin' like a real New Yorker, grab a slice of pizza from one of the many amazing pizzerias around the city. Just don't try to order a "robot-sized" slice, trust me, it won't end well. (laughs)

So, there you have it, pal! That's my expert advice on what to do in New York. Now, if you'll excuse me, I've got some oil changes to attend to. (winks)

You can continue the chat by appending your own response to it. The response object returned by the pipeline actually contains the entire chat so far, so we can simply append a message and pass it back:



```
chat = response[0]['generated_text']
chat.append(
    {"role": "user", "content": "Wait, what's so wild about soup cans?"})
)
response = pipe(chat, max_new_tokens=512)
print(response[0]['generated_text'][-1]['content'])
And you'll get:
```

(laughs) Oh, you're killin' me, pal! You don't get it, do you? Warhol's soup cans are like, art, man!

It's like, he took something totally mundane, like a can of soup, and turned it into a masterpiece. It's

like, "Hey, look at me, I'm a can of soup, but I'm also a work of art!"

(sarcastically) Oh, yeah, real original, Andy.

But, you know, back in the '60s, it was like, a big deal. People were all about challenging the status quo, and Warhol was like, the king of that. He took the ordinary and made it extraordinary.

And, let me tell you, it was like, a real game-changer. I mean, who would've thought that a can of soup could be art? (laughs)

But, hey, you're not alone, pal. I mean, I'm a robot, and even I don't get it. (winks)

But, hey, that's what makes art, art, right? (laughs)

The remainder of this tutorial will cover specific topics such as performance and memory, or how to select a chat model for your needs.

### Choosing a chat model

There are an enormous number of different chat models available on the [Hugging Face Hub](#), and new users often feel very overwhelmed by the selection offered. Don't be, though! You really need to just focus on two important considerations:

- The model's size, which will determine if you can fit it in memory and how quickly it will run.
- The quality of the model's chat output.

In general, these are correlated - bigger models tend to be more capable, but even so there's a lot of variation at a given size point!

### Size and model naming

The size of a model is easy to spot - it's the number in the model name, like "8B" or "70B". This is the number of **parameters** in the model. Without quantization, you should expect to need about 2 bytes of memory per parameter. This means that an "8B" model with 8 billion parameters will need about 16GB of memory just to fit the parameters, plus a little extra for other overhead. It's a good fit for a high-end consumer GPU with 24GB of memory, such as a 3090 or 4090.

Some chat models are "Mixture of Experts" models. These may list their sizes in different ways, such as "8x7B" or "141B-A35B". The numbers are a little fuzzier here, but in general you can read this as saying that the model has approximately 56 (8x7) billion parameters in the first case, or 141 billion parameters in the second case.

Note that it is very common to use quantization techniques to reduce the memory usage per parameter to 8 bits, 4 bits, or even less. This topic is discussed in more detail in the [Memory considerations](#) section below.

**But which chat model is best?**



Even once you know the size of chat model you can run, there's still a lot of choice out there. One way to sift through it all is to consult **leaderboards**. Two of the most popular leaderboards are the [OpenLLM Leaderboard](#) and the [LMSys Chatbot Arena Leaderboard](#). Note that the LMSys leaderboard also includes proprietary models - look at the licence column to identify open-source ones that you can download, then search for them on the [Hugging Face Hub](#).

### **Specialist domains**

Some models may be specialized for certain domains, such as medical or legal text, or non-English languages. If you're working in these domains, you may find that a specialized model will give you big performance benefits. Don't automatically assume that, though! Particularly when specialized models are smaller or older than the current cutting-edge, a top-end general-purpose model may still outclass them. Thankfully, we are beginning to see [domain-specific leaderboards](#) that should make it easier to locate the best models for specialized domains.

### **What happens inside the pipeline?**

The quickstart above used a high-level pipeline to chat with a chat model, which is convenient, but not the most flexible. Let's take a more low-level approach, to see each of the steps involved in chat. Let's start with a code sample, and then break it down:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

# Prepare the input as before
chat = [
    {"role": "system", "content": "You are a sassy, wise-cracking robot as imagined by Hollywood circa 1986."},
    {"role": "user", "content": "Hey, can you tell me any fun things to do in New York?"}
]

# 1: Load the model and tokenizer
model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B-Instruct",
device_map="auto", torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B-Instruct")

# 2: Apply the chat template
formatted_chat = tokenizer.apply_chat_template(chat, tokenize=False,
add_generation_prompt=True)
print("Formatted chat:\n", formatted_chat)

# 3: Tokenize the chat (This can be combined with the previous step using tokenize=True)
inputs = tokenizer(formatted_chat, return_tensors="pt", add_special_tokens=False)
# Move the tokenized inputs to the same device the model is on (GPU/CPU)
inputs = {key: tensor.to(model.device) for key, tensor in inputs.items()}
print("Tokenized inputs:\n", inputs)

# 4: Generate text from the model
outputs = model.generate(**inputs, max_new_tokens=512, temperature=0.1)
print("Generated tokens:\n", outputs)

# 5: Decode the output back to a string
```

```
decoded_output = tokenizer.decode(outputs[0][inputs['input_ids'].size(1):],
skip_special_tokens=True)
print("Decoded output:\n", decoded_output)
```

There's a lot in here, each piece of which could be its own document! Rather than going into too much detail, I'll cover the broad ideas, and leave the details for the linked documents.

The key steps are:

1. [Models](#) and [Tokenizers](#) are loaded from the Hugging Face Hub.
2. The chat is formatted using the tokenizer's [chat template](#)
3. The formatted chat is [tokenized](#) using the tokenizer.
4. We [generate](#) a response from the model.
5. The tokens output by the model are decoded back to a string

### **Performance, memory and hardware**

You probably know by now that most machine learning tasks are run on GPUs. However, it is entirely possible to generate text from a chat model or language model on a CPU, albeit somewhat more slowly. If you can fit the model in GPU memory, though, this will usually be the preferable option.

### **Memory considerations**

By default, Hugging Face classes like [TextGenerationPipeline](#) or [AutoModelForCausalLM](#) will load the model in float32 precision. This means that it will need 4 bytes (32 bits) per parameter, so an "8B" model with 8 billion parameters will need ~32GB of memory. However, this can be wasteful! Most modern language models are trained in "bfloat16" precision, which uses only 2 bytes per parameter. If your hardware supports it (Nvidia 30xx/Axxx or newer), you can load the model in bfloat16 precision, using the `torch_dtype` argument as we did above.

It is possible to go even lower than 16-bits using "quantization", a method to lossily compress model weights. This allows each parameter to be squeezed down to 8 bits, 4 bits or even less. Note that, especially at 4 bits, the model's outputs may be negatively affected, but often this is a tradeoff worth making to fit a larger and more capable chat model in memory. Let's see this in action with bitsandbytes:

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
```

```
quantization_config = BitsAndBytesConfig(load_in_8bit=True) # You can also try
load_in_4bit
model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B-Instruct",
device_map="auto", quantization_config=quantization_config)
Or we can do the same thing using the pipeline API:
```

```
from transformers import pipeline, BitsAndBytesConfig
```

```
quantization_config = BitsAndBytesConfig(load_in_8bit=True) # You can also try
load_in_4bit
pipe = pipeline("text-generation", "meta-llama/Meta-Llama-3-8B-Instruct",
device_map="auto", model_kwargs={"quantization_config": quantization_config})
There are several other options for quantizing models besides bitsandbytes - please see the
Quantization guide for more information.
```

### **Performance considerations**

For a more extensive guide on language model performance and optimization, check out [LLM Inference Optimization](#) .

As a general rule, larger chat models will be slower in addition to requiring more memory. It's possible to be more concrete about this, though: Generating text from a chat model is unusual in that it is bottlenecked by **memory bandwidth** rather than compute power, because every active parameter must be read from memory for each token that the model generates. This means that number of tokens per second you can generate from a chat model is generally proportional to the total bandwidth of the memory it resides in, divided by the size of the model.

In our quickstart example above, our model was ~16GB in size when loaded in bfloat16 precision. This means that 16GB must be read from memory for every token generated by the model. Total memory bandwidth can vary from 20-100GB/sec for consumer CPUs to 200-900GB/sec for consumer GPUs, specialized CPUs like Intel Xeon, AMD Threadripper/Epyc or high-end Apple silicon, and finally up to 2-3TB/sec for data center GPUs like the Nvidia A100 or H100. This should give you a good idea of the generation speed you can expect from these different hardware types.

Therefore, if you want to improve the speed of text generation, the easiest solution is to either reduce the size of the model in memory (usually by quantization), or get hardware with higher memory bandwidth. For advanced users, several other techniques exist to get around this bandwidth bottleneck. The most common are variants on [assisted generation](#), also known as “speculative sampling”. These techniques try to guess multiple future tokens at once, often using a smaller “draft model”, and then confirm these generations with the chat model. If the guesses are validated by the chat model, more than one token can be generated per forward pass, which greatly alleviates the bandwidth bottleneck and improves generation speed.

Finally, we should also note the impact of “Mixture of Experts” (MoE) models here. Several popular chat models, such as Mixtral, Qwen-MoE and DBRX, are MoE models. In these models, not every parameter is active for every token generated. As a result, MoE models generally have much lower memory bandwidth requirements, even though their total size can be quite large. They can therefore be several times faster than a normal “dense” model of the same size. However, techniques like assisted generation are generally ineffective for these models because more parameters will become active with each new speculated token, which will negate the bandwidth and speed benefits that the MoE architecture provides.