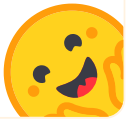


Fine-tune a pretrained model

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
```

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("bert-base-cased")


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


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

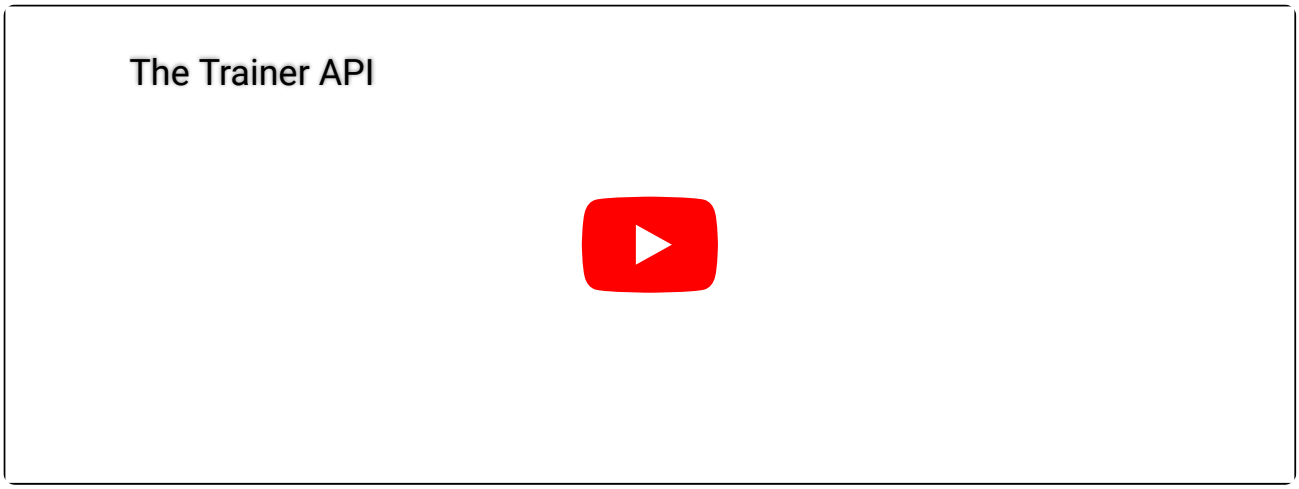
If you like, you can create a smaller subset of the full dataset to fine-tune on to reduce the time it takes:

```
>>> 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("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
```


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 `evaluation_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", evaluation_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

tokenizer = AutoTokenizer.from_pretrained("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("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("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()
```

Additional resources

For more fine-tuning examples, refer to:

- 🤗 Transformers Examples includes scripts to train common NLP tasks in PyTorch and TensorFlow.
- 🤗 Transformers Notebooks contains various notebooks on how to fine-tune a model for specific tasks in PyTorch and TensorFlow.

← Preprocess data

Train with a script →