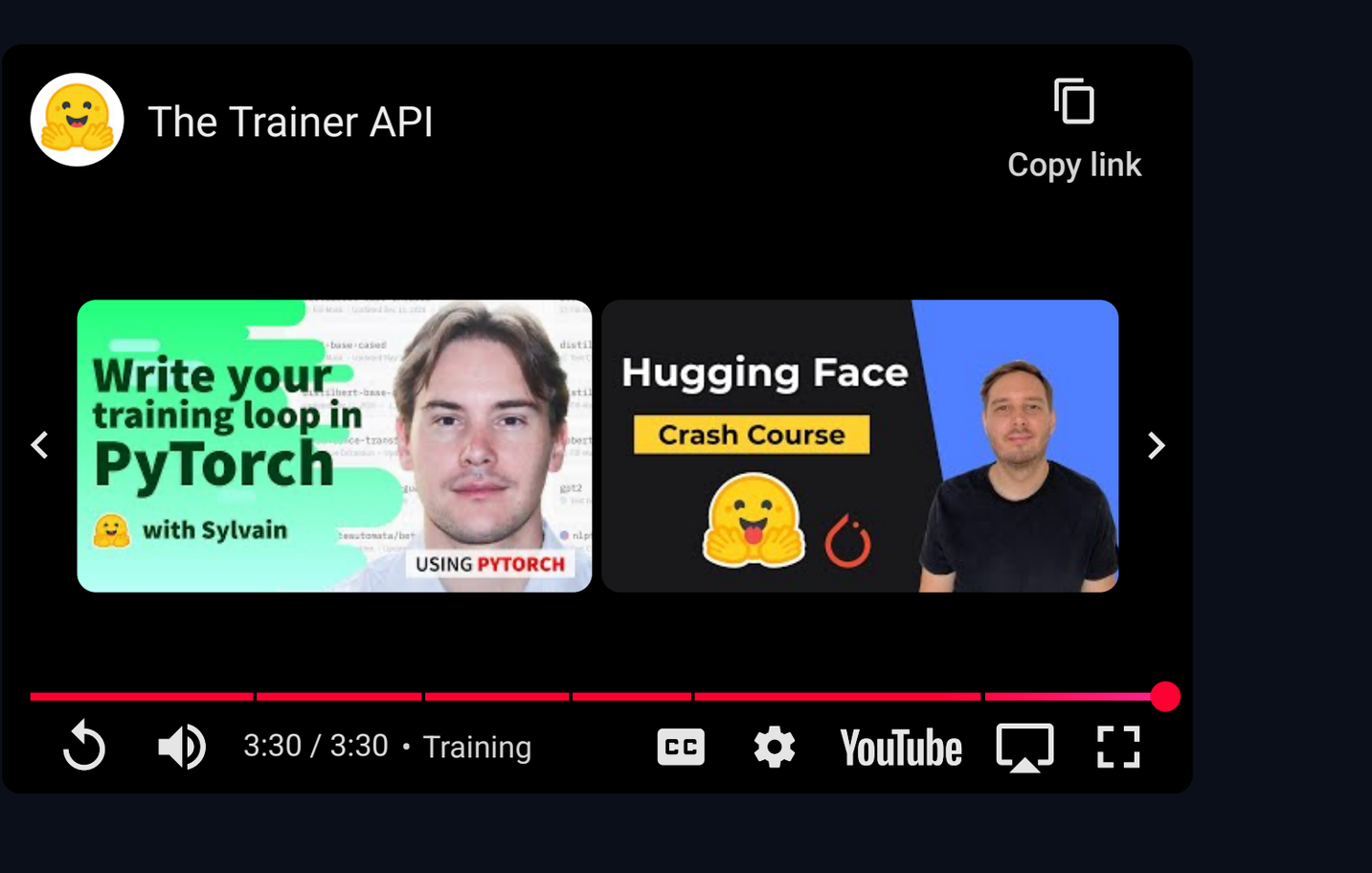
Pytorch

Fine-tuning a model with the Trainer API

Copy page

Ask a question | Open in Colab | Open Studio Lab



🤗 Transformers provides a `Trainer` class to help you fine-tune any of the pretrained models it provides on your dataset with modern best practices. Once you've done all the data preprocessing work in the last section, you have just a few steps left to define the `Trainer`. The hardest part is likely to be preparing the environment to run `Trainer.train()`, as it will run very slowly on a CPU. If you don't have a GPU set up, you can get access to free GPUs or TPUs on Google Colab.

> 📚 **Training Resources**: Before diving into training, familiarize yourself with the comprehensive 🤗 Transformers training guide and explore practical examples in the fine-tuning cookbook.

The code examples below assume you have already executed the examples in the previous section. Here is a short summary recapping what you need:

```python
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("glue", "mrpc")
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)


def tokenize_function(example):
    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)


tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Training

The first step before we can define our `Trainer` is to define a `TrainingArguments` class that will contain all the hyperparameters the `Trainer` will use for training and evaluation. The only argument you have to provide is a directory where the trained model will be saved, as well as the checkpoints along the way. For all the rest, you can leave the defaults, which should work pretty well for a basic fine-tuning.

```python
from transformers import TrainingArguments

training_args = TrainingArguments("test-trainer")
```

If you want to automatically upload your model to the Hub during training, pass along `push_to_hub=True` in the `TrainingArguments`. We will learn more about this in Chapter 4

> 🚀 **Advanced Configuration**: For detailed information on all available training arguments and optimization strategies, check out the TrainingArguments documentation and the training configuration cookbook.

The second step is to define our model. As in the previous chapter, we will use the `AutoModelForSequenceClassification` class, with two labels:

```python
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

You will notice that unlike in Chapter 2, you get a warning after instantiating this pretrained model. This is because BERT has not been pretrained on classifying pairs of sentences, so the head of the pretrained model has been discarded and a new head suitable for sequence classification has been added instead. The warnings indicate that some weights were not used (the ones corresponding to the dropped pretraining head) and that some others were randomly initialized (the ones for the new head). It concludes by encouraging you to train the model, which is exactly what we are going to do now.

Once we have our model, we can define a `Trainer` by passing it all the objects constructed up to now — the `model`, the `training_args`, the training and validation datasets, our `data_collator`, and our `processing_class`. The `processing_class` parameter is a newer addition that tells the Trainer which tokenizer to use for processing:

```python
from transformers import Trainer

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    processing_class=tokenizer,
)
```

When you pass a tokenizer as the `processing_class`, the default `data_collator` used by the `Trainer` will be a `DataCollatorWithPadding`. You can skip the `data_collator=data_collator` line in this case, but we included it here to show you this important part of the processing pipeline.

> 📖 **Learn More**: For comprehensive details on the Trainer class and its parameters, visit the Trainer API documentation and explore advanced usage patterns in the training cookbook recipes.

To fine-tune the model on our dataset, we just have to call the `train()` method of our `Trainer`:

```python
trainer.train()
```

This will start the fine-tuning (which should take a couple of minutes on a GPU) and report the training loss every 500 steps. It won't, however, tell you how well (or badly) your model is performing. This is because:

1. We didn't tell the `Trainer` to evaluate during training by setting `eval_strategy` in `TrainingArguments` to either `"steps"` (evaluate every `eval_steps`) or `"epoch"` (evaluate at the end of each epoch).
2. We didn't provide the `Trainer` with a `compute_metrics()` function to calculate a metric during said evaluation (otherwise the evaluation would just have printed the loss, which is not a very intuitive number).

Evaluation

Let's see how we can build a useful `compute_metrics()` function and use it the next time we train. The function must take an `EvalPrediction` object (which is a named tuple with a `predictions` field and a `label_ids` field) and will return a dictionary mapping strings to floats (the strings being the names of the metrics returned, and the floats their values). To get some predictions from our model, we can use the `Trainer.predict()` command:

```python
predictions = trainer.predict(tokenized_datasets["validation"])
print(predictions.predictions.shape, predictions.label_ids.shape)
```

```
(408, 2) (408,)
```

The output of the `predict()` method is another named tuple with three fields: `predictions`, `label_ids`, and `metrics`. The `metrics` field will just contain the loss on the dataset passed, as well as some time metrics (how long it took to predict, in total and on average). Once we complete our `compute_metrics()` function and pass it to the `Trainer`, that field will also contain the metrics returned by `compute_metrics()`.

As you can see, `predictions` is a two-dimensional array with shape 408 x 2 (408 being the number of elements in the dataset we used). Those are the logits for each element of the dataset we passed to `predict()` (as you saw in the previous chapter, all Transformer models return logits). To transform them into predictions that we can compare to our labels, we need to take the index with the maximum value on the second axis:

```python
import numpy as np

preds = np.argmax(predictions.predictions, axis=-1)
```

We can now compare those `preds` to the labels. To build our `compute_metric()` function, we will rely on the metrics from the 🤗 Evaluate library. We can load the metrics associated with the MRPC dataset as easily as we loaded the dataset, this time with the `evaluate.load()` function. The object returned has a `compute()` method we can use to do the metric calculation:

```python
import evaluate

metric = evaluate.load("glue", "mrpc")
metric.compute(predictions=preds, references=predictions.label_ids)
```

```
{'accuracy': 0.8578431372549019, 'f1': 0.8996539792387542}
```

> Learn about different evaluation metrics and strategies in the 🤗 Evaluate documentation.

The exact results you get may vary, as the random initialization of the model head might change the metrics it achieved. Here, we can see our model has an accuracy of 85.78% on the validation set and an F1 score of 89.97. Those are the two metrics used to evaluate results on the MRPC dataset for the GLUE benchmark. The table in the BERT paper reported an F1 score of 88.9 for the base model. That was the `uncased` model while we are currently using the `cased` model, which explains the better result.

Wrapping everything together, we get our `compute_metrics()` function:

```python
def compute_metrics(eval_preds):
    metric = evaluate.load("glue", "mrpc")
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)
```

And to see it used in action to report metrics at the end of each epoch, here is how we define a new `Trainer` with this `compute_metrics()` function:

```python
training_args = TrainingArguments("test-trainer", eval_strategy="epoch")
model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    processing_class=tokenizer,
    compute_metrics=compute_metrics,
)
```

Note that we create a new `TrainingArguments` with its `eval_strategy` set to `"epoch"` and a new model — otherwise, we would just be continuing the training of the model we have already trained. To launch a new training run, we execute:

```python
trainer.train()
```

This time, it will report the validation loss and metrics at the end of each epoch on top of the training loss. Again, the exact accuracy/F1 score you reach might be a bit different from what we found, because of the random head initialization of the model, but it should be in the same ballpark.

Advanced Training Features

The `Trainer` comes with many built-in features that make modern deep learning best practices accessible:

Mixed Precision Training: Use `fp16=True` in your training arguments for faster training and reduced memory usage:

```python
training_args = TrainingArguments(
    "test-trainer",
    eval_strategy="epoch",
    fp16=True,  # Enable mixed precision
)
```

Gradient Accumulation: For effective larger batch sizes when GPU memory is limited:

```python
training_args = TrainingArguments(
    "test-trainer",
    eval_strategy="epoch",
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,  # Effective batch size = 4 * 4 = 16
)
```

Learning Rate Scheduling: The Trainer uses linear decay by default, but you can customize this:

```python
training_args = TrainingArguments(
    "test-trainer",
    eval_strategy="epoch",
    learning_rate=2e-5,
    lr_scheduler_type="cosine",  # Try different schedulers
)
```

> 🎯 **Performance Optimization**: For more advanced training techniques including distributed training, memory optimization, and hardware-specific optimizations, explore the 🤗 Transformers performance guide.

The `Trainer` will work out of the box on multiple GPUs or TPUs and provides lots of options for distributed training. We will go over everything it supports in Chapter 10.

This concludes the introduction to fine-tuning using the `Trainer` API. An example of doing this for most common NLP tasks will be given in Chapter 7, but for now let's look at how to do the same thing with a pure PyTorch training loop.

> 📝 **More Examples**: Check out the comprehensive collection of 🤗 Transformers notebooks.

Section Quiz

Test your understanding of the Trainer API and fine-tuning concepts:

1. What is the purpose of the <code> processing_class </code> parameter in the Trainer?

- [ ] It specifies which model architecture to use.
- [x] It tells the Trainer which tokenizer to use for processing data.

Correct! The processing_class parameter is a modern addition that helps the Trainer know which tokenizer to use.

- [ ] It determines the batch size for training.
- [ ] It controls the evaluation frequency.

Submit **You got all the answers!**

2. Which TrainingArguments parameter controls how often evaluation occurs during training?

- [ ] eval_frequency
- [x] eval_strategy

Correct! eval_strategy can be set to 'epoch', 'steps', or 'no' to control evaluation timing.

- [ ] evaluation_steps
- [ ] do_eval

Submit **You got all the answers!**

3. What does <code> fp16=True </code> in TrainingArguments enable?

- [ ] 16-bit integer precision for faster training.
- [x] Mixed precision training with 16-bit floating-point numbers for faster training and reduced memory usage.

Correct! Mixed precision training uses 16-bit floats for forward pass and 32-bit for gradients, improving speed and reducing memory usage.

- [ ] Training for exactly 16 epochs.
- [ ] Using 16 GPUs for distributed training.

Submit **You got all the answers!**

4. What is the role of the <code> compute_metrics </code> function in the Trainer?

- [ ] It calculates the loss during training.
- [x] It converts logits to predictions and calculates evaluation metrics like accuracy and F1.

Correct! compute_metrics takes predictions and labels, then returns metrics for evaluation.

- [ ] It determines which optimizer to use.
- [ ] It preprocesses the training data.

Submit **You got all the answers!**

5. What happens when you don't provide an <code> eval_dataset </code> to the Trainer?

- [ ] Training will fail with an error.
- [ ] The Trainer will automatically split the training data for evaluation.
- [x] You won't get evaluation metrics during training, but training will still work.

Correct! Evaluation is optional - you can train without it, but you won't see validation metrics.

- [ ] The model will use the training data for evaluation.

Submit **You got all the answers!**

6. What is gradient accumulation and how do you enable it?

- [ ] It saves gradients to disk, enabled with save_gradients=True.
- [x] It accumulates gradients over multiple batches before updating, enabled with gradient_accumulation_steps.

Correct! This allows you to simulate larger batch sizes by accumulating gradients over multiple forward passes.

- [ ] It speeds up gradient computation, enabled automatically with fp16.
- [ ] It prevents gradient overflow, enabled with gradient_clipping=True.

Submit **You got all the answers!**

> 💡 **Key Takeaways:**
> - The `Trainer` API provides a high-level interface that handles most training complexity
> - Use `processing_class` to specify your tokenizer for proper data handling
> - `TrainingArguments` controls all aspects of training: learning rate, batch size, evaluation strategy, and optimizations
> - `compute_metrics` enables custom evaluation metrics beyond just training loss
> - Modern features like mixed precision (`fp16=True`) and gradient accumulation can significantly improve training efficiency

</> Update on GitHub