

# CS197 Harvard: AI Research Experiences

Fall 2022: Lecture 4 – “In-Tune with Jazz Hands”

Fine-tuning a Language Model using Huggingface

Instructed by Pranav Rajpurkar. Website <https://cs197.seas.harvard.edu/>

## Abstract

I’ve found that building is the most effective way of learning when it comes to AI/ML engineering. Instead of a typical theoretical introduction to deep learning, I want to start our first dive into deep learning through engineering using Huggingface, which has created a set of libraries that are being rapidly adopted in the AI community. We’ll focus today on natural language processing, which has seen some of the biggest AI advancements, most recently through large language models. This lecture is structured as a live coding walkthrough: we will fine-tune a pre-trained language model on a dataset. Through an engineering lens, this walkthrough will cover dataset loading, tokenization, and fine-tuning.



*Midjourney generation for "a human typing into a computer that types into another computer"*

## Learning outcomes:

- Load up and process a natural language processing dataset using the datasets library.
- Tokenize a text sequence, and understand the steps used in tokenization.
- Construct a dataset and training step for causal language modeling.

# Fine-Tuning Our Language Model

In lecture 1, we used the GPT-3 language model to complete some text for us. Today, we are going to fine tune such a model (adapt it to new data). [Language modeling](#) predicts words in a sentence. There are different types of language modeling, we're going to focus in particular on causal language modeling, where the task is to predict the next token in a sequence of tokens using only the tokens that came before it.

My final notebook after today's lecture is [here](#).

## HuggingFace

For this example, we are going to work with libraries from Huggingface. Hugging Face has become a community and data science center for building, training and deploying ML models based on open source (OS) software. Fun fact: Huggingface was initially a chatbot, and named after the emoji that looks like a smiling face with jazz hands – 🤗.

We're going to use Huggingface to fine-tune a language model on a dataset. You may have to follow the installation instructions [here](#) later in the lecture. Our lecture today will closely follow [this](#), and [this](#), but with some of my own spin on things.

## Loading up a dataset

We are going to use the 🤗 Datasets [library](#). This library has three main features: (1) efficient way to load and process data from raw files (CSV/JSON/text) or in-memory data (python dict, pandas dataframe), (2) a simple way to access and share datasets with the research and practitioner communities (over 1,000 datasets are already accessible in one line), and (3) is interoperable with DL frameworks like pandas, NumPy, PyTorch and TensorFlow.

For this demo, we are going to work with the SQuAD dataset. Briefly, the Stanford Question Answering Dataset (SQuAD) is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage, or the question might be unanswerable. Fun fact: SQuAD came out of one of my first projects in my PhD.

Today, we're going to see whether we can fine-tune GPT on the questions posed in SQuAD, so we have a question completion agent. We will load the dataset from here: <https://huggingface.co/datasets/squad>

Let's get started!


```

from datasets import load_dataset
dataset = load_dataset("squad")
dataset

```

✓ ✓ 0.3s Python

Reusing dataset squad  
 (/Users/rajpurkar/.cache/huggingface/datasets/squad/plain\_text/1.0.0/d6ec3ceb9f)

100%  2/2 [00:00<00:00, 93.34it/s]

```

DatasetDict({
  train: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 87599
  })
  validation: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 10570
  })
})

```

This method (1) downloads and imports in the library the dataset loading script from the path if it's not already cached inside the library, (2) runs the dataset loading script which will download the dataset file from the original URL if it's not already downloaded and cached, process and cache the dataset, and (3) return a dataset built from the requested splits in split (default: all).

The method returns a dictionary (`datasets.DatasetDict`) with a train and a validation subset; what you get here will vary per dataset.

We can remove columns that we are not going to use, and use the map function to add a special `<|endoftext|>` token that GPT2 uses to mark the end of a document.

```
def add_end_of_text(example):
    example['question'] = example['question'] + '<|endoftext|>'
    return example

dataset = dataset.remove_columns(['id', 'title', 'context', 'answers'])
dataset = dataset.map(add_end_of_text)
```

✓ 4.1s

Python

Note the use of the map() function. As specified [here](#), the primary purpose of map() is to speed up processing functions. It allows you to apply a processing function to each example in a dataset.

Let's look at the structure of a few of the entries.

```
dataset['train']['question'][:10]
```

✓ 0.1s

Python

```
['To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France?
<|endoftext|>',
 'What is in front of the Notre Dame Main Building?<|endoftext|>',
 'The Basilica of the Sacred heart at Notre Dame is beside to which
structure?<|endoftext|>',
 'What is the Grotto at Notre Dame?<|endoftext|>',
 'What sits on top of the Main Building at Notre Dame?<|endoftext|>',
 'When did the Scholastic Magazine of Notre dame begin publishing?
<|endoftext|>',
 "How often is Notre Dame's the Juggler published?<|endoftext|>",
 'What is the daily student paper at Notre Dame called?<|endoftext|>',
 'How many student news papers are found at Notre Dame?<|endoftext|>',
 'In what year did the student paper Common Sense begin publication at Notre
Dame?<|endoftext|>']
```

Good. Our dataset is ready for use (almost)!

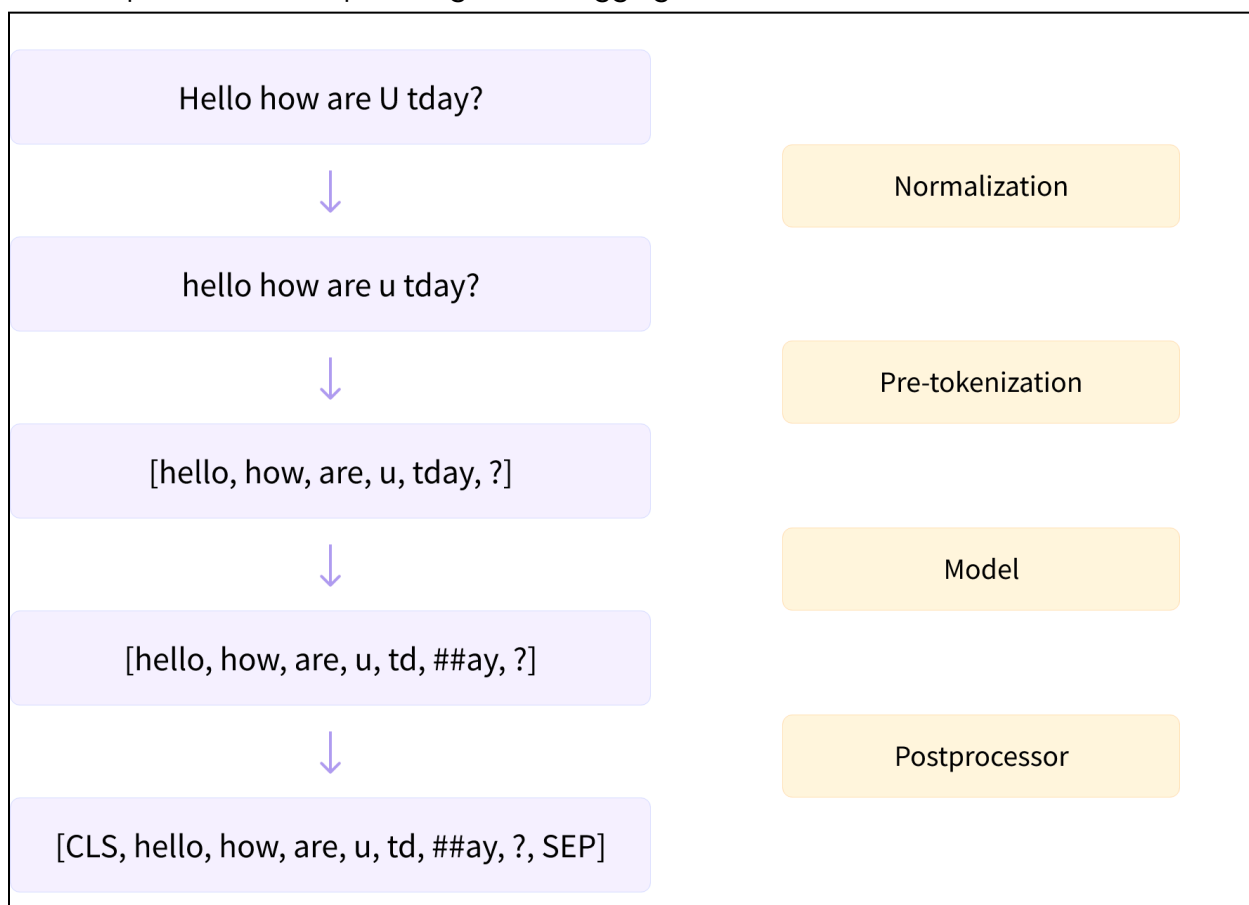
## Tokenizer

Before we can use this data, we need to process it to be in an acceptable format for the model. So how do we feed in text data into the model? We are going to use a tokenizer. A tokenizer prepares the inputs for a model.

A tokenization pipeline in huggingface comprises several [steps](#):

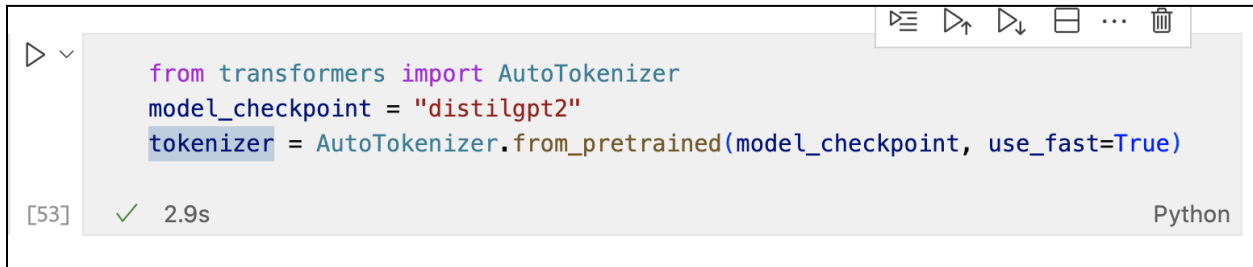
*(1) Normalization (any cleanup of the text that is deemed necessary, such as removing spaces or accents, Unicode normalization, etc.), (2) Pre-tokenization (splitting the input into words), (3) Running the input through the model (using the pre-tokenized words to produce a sequence of tokens), and (4) Post-processing (adding the special tokens of the tokenizer, generating the attention mask and token type IDs).*

This is depicted in this helpful image from huggingface:



The above steps show how we can go from text into tokens. There are multiple rules that govern the process that are specific to certain models. For tokenization, there are three main subword tokenization algorithms: BPE (used by GPT-2 and others), WordPiece (used for example by BERT), and Unigram (used by T5 and others); we won't go into any of these, but if you're curious, you can learn about them [here](#).

Since tokenization processes are model-specific, if we want to fine-tune the model on new data, we need to instantiate the tokenizer using the name of the model, to make sure we use the same rules that were used when the model was pretrained. This is all done by the `AutoTokenizer` class:



```

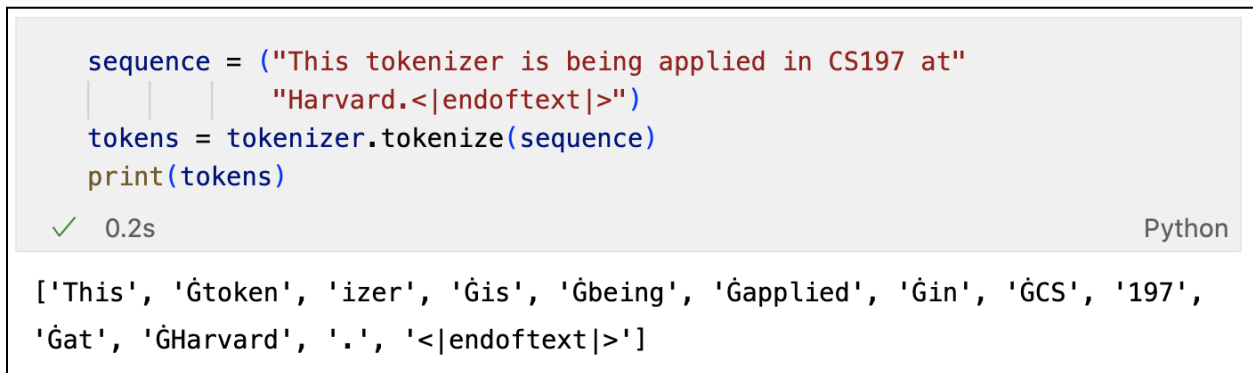
from transformers import AutoTokenizer
model_checkpoint = "distilgpt2"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)

```

[53] ✓ 2.9s Python

Pro-tip: The huggingface library contains tokenizers for all the models. Tokenizers are available in a Python implementation or “Fast” implementation which uses the Rust language.

Let’s first convert a sample sentence into tokens:



```

sequence = ("This tokenizer is being applied in CS197 at"
            "Harvard.<|endoftext|>")
tokens = tokenizer.tokenize(sequence)
print(tokens)

```

✓ 0.2s Python

```

['This', 'Ġtoken', 'izer', 'Ġis', 'Ġbeing', 'Ġapplied', 'Ġin', 'ĠCS', '197',
'Ġat', 'ĠHarvard', '.', '<|endoftext|>']

```

Here, you can see the sentence broken into subwords. In GPT2 and other model tokenizers, the space before a word is part of a word; spaces are [converted](#) in a special character (the `Ġ`) in the tokenizer.

Once we have split text into tokens (what we’ve seen above), we now need to convert tokens into numbers. To do this, the tokenizer has a vocabulary, which is the part we download when we instantiate it with the `from_pretrained()` method. Again, we need to use the same vocabulary used when the model was pretrained.

```
ids = tokenizer.convert_tokens_to_ids(tokens)
print(ids)
```

✓ 0.5s

Python

```
[1212, 11241, 7509, 318, 852, 5625, 287, 9429, 24991, 379, 11131, 13, 50256]
```

The tokenizer actually automatically chains these operations for us when we use `__call__`:

```
sequence = ("This tokenizer is being applied in CS197 at"
            "Harvard.<|endoftext|>")
tokenizer(sequence)
```

✓ 0.9s

Python

```
{'input_ids': [1212, 11241, 7509, 318, 852, 5625, 287, 9429, 24991, 379,
13587, 10187, 13, 50256], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1]}
```

The tokenizer returns a dictionary with 2 important items: (1) `input_ids` are the indices corresponding to each token in the sentence, and (2) `attention_mask` indicates whether a token should be attended to or not. We are going to ignore the `attention_mask` for now; if you're curious, you can read more about it [here](#).

Exercise: Try another tokenizer on your own sequence.

- (1) What are the differences that you see?
- (2) Find out what kind of a tokenization algorithm your tokenizer uses .

We are going to now tokenize our dataset. We apply a `tokenize` function to all the splits in our "datasets" object.

```
def tokenize_function(examples):
    return tokenizer(examples["question"], truncation=True)

tokenized_datasets = dataset.map(
    tokenize_function,
    batched=True,
    num_proc=4,
    remove_columns=["question"]
)
```

✓ 2.5s

Python

We use the 🤗 Datasets map function to apply the preprocessing function over the entire dataset. By setting batched=True, we process multiple elements of the dataset at once and increase the number of processes with num\_proc=4. Finally, we remove the “questions” column because we won’t need it now.

Let’s see what the tokenized\_datasets variable looks like.

```
tokenized_datasets
```

[101]

✓ 0.2s

Python

```
... DatasetDict({
  train: Dataset({
    features: ['input_ids', 'attention_mask'],
    num_rows: 87599
  })
  validation: Dataset({
    features: ['input_ids', 'attention_mask'],
    num_rows: 10570
  })
})
```

## Data Processing

For causal language modeling (CLM), one of the data preparation steps often used is to concatenate the different examples together, and then split them into chunks of equal size. This is so that we can have a common length across all examples without needing to pad. So Say we have: [

```
"I went to the yard.<|endoftext|>",
"You came here a long time ago from the west coast.<|endoftext|>"
```

], we might change this to:

```
"I went to the yard.<|endoftext|>You came here",
"a long time ago from the west coast.<|endoftext|>"
```

].

Let's implement this transformation. We are going to use chunks defined by `block_size` of 128 (although GPT-2 should be able to process a length of 1024, we might not have the capacity to do that locally).

We need to concatenate all our texts together then split the result in small chunks of a certain `block_size`. To do this, we will use the `map` method again, with the option `batched=True`. This option actually lets us change the number of examples in the datasets by returning a different number of examples than we got. This way, we can create our new samples from a batch of examples.

```

block_size = 128

def group_texts(examples):
    # repeat concatenation for input_ids and other keys
    concatenated_examples = {k: sum(examples[k], []) for k in
                               examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    total_length = (total_length // block_size) * block_size

    # populate each of input_ids and other keys
    result = {
        k: [t[i : i + block_size] for i in range(0,
            total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    # add labels because we'll need it as the output
    result["labels"] = result["input_ids"].copy()
    return result

lm_datasets = tokenized_datasets.map(
    group_texts,
    batched=True,
    batch_size=1000,
    num_proc=4,
)

```

[42] ✓ 1.7s Python

Note that we duplicate the inputs for our labels. The 🤗 Transformers library will automatically be able to use this label to set up the causal language modeling task (by shifting all tokens to the right).

We can look at a sample of the `lm_dataset` now.

```
print(lm_datasets['train']['input_ids'][0])
```

[156] ✓ 0.4s Python

```
... [2514, 4150, 750, 262, 5283, 5335, 7910, 1656, 287, 1248, 3365, 287, 406,
454, 8906, 4881, 30, 50256, 2061, 318, 287, 2166, 286, 262, 23382, 20377,
8774, 11819, 30, 50256, 464, 32520, 3970, 286, 262, 17380, 2612, 379, 23382,
20377, 318, 13970, 284, 543, 4645, 30, 50256, 2061, 318, 262, 10299, 33955,
379, 23382, 20377, 30, 50256, 2061, 10718, 319, 1353, 286, 262, 8774, 11819,
379, 23382, 20377, 30, 50256, 2215, 750, 262, 3059, 349, 3477, 11175, 286,
23382, 288, 480, 2221, 12407, 30, 50256, 2437, 1690, 318, 23382, 20377, 338,
262, 39296, 1754, 3199, 30, 50256, 2061, 318, 262, 4445, 3710, 3348, 379,
23382, 20377, 1444, 30, 50256, 2437, 867, 3710, 1705, 9473, 389, 1043, 379,
23382, 20377, 30, 50256, 818, 644, 614, 750, 262, 3710, 3348]
```

Note how we can use tokenizer's decode function to go from our encoded ids back to the text.

```
tokenizer.decode(lm_datasets['train']['input_ids'][0])
```

[157] ✓ 0.4s Python

```
... "To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France?
<|endoftext|>What is in front of the Notre Dame Main Building?
<|endoftext|>The Basilica of the Sacred heart at Notre Dame is beside to
which structure?<|endoftext|>What is the Grotto at Notre Dame?
<|endoftext|>What sits on top of the Main Building at Notre Dame?
<|endoftext|>When did the Scholastic Magazine of Notre dame begin publishing?
<|endoftext|>How often is Notre Dame's the Juggler published?
<|endoftext|>What is the daily student paper at Notre Dame called?
<|endoftext|>How many student news papers are found at Notre Dame?
<|endoftext|>In what year did the student paper"
```

Finally, we will make a smaller version of our training and validation so we can fine-tune our model in a reasonable amount of time.

```

small_train_dataset = \
    lm_datasets["train"].shuffle(seed=42).select(range(100))
small_eval_dataset = \
    lm_datasets["validation"].shuffle(seed=42).select(range(100))

```

✓ 0.3s

Python

## Causal language modeling

Our modeling is going to be relatively straightforward. We need to define training arguments, and set up our [Trainer](#). The Trainer class provides an API for feature-complete training in PyTorch for most standard use cases.

As part of our training args, we specify that we will push this model to the Hub. The Hub is a huggingface platform where anyone can share and explore models, datasets, and demos.

```

training_args = TrainingArguments(
    f"{model_checkpoint}-squad",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    weight_decay=0.01,
    push_to_hub=True,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=small_train_dataset,
    eval_dataset=small_eval_dataset,
)

trainer.train()

```

✓ 2m 6.9s

Python

We can now evaluate the model. Because we want our model to assign high probabilities to sentences that are real, and low probabilities to fake sentences, we seek a model that assigns the highest probability to the test set. The metric we use is 'perplexity', which we can think of as the inverse probability of the test set normalized by the number of words in the test set. Therefore, a lower perplexity is better.

```
import math
eval_results = trainer.evaluate()
print(f"Perplexity: {math.exp(eval_results['eval_loss']):.2f}")
```

✓ 8.6s

Python

\*\*\*\*\* Running Evaluation \*\*\*\*\*

Num examples = 100

Batch size = 8

100%  13/13 [00:07<00:00, 1.75it/s]

Perplexity: 24.10

We can now upload our final model and tokenizer to the hub.

```
tokenizer.save_pretrained('gpt2-squad')
model.push_to_hub('gpt2-squad')
```

✓ 10.1s

Python

Woohoo, we can now use our new pushed model

### Exercises

Exercise 1: Now rather than starting with a pre-trained model, start with a model from scratch.

Exercise 2: Replace DistilGPT with a non-GPT causal language model.

Exercise 3: Replace the SQuAD dataset with another dataset (except for wikitext).

## Generation with our fine-tuned model

In our final step, we are going to use our fine-tuned model to autocomplete some questions. Let's go ahead and load our saved model first:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
model = AutoModelForCausalLM.from_pretrained("rajpurkar/gpt2-squad")
tokenizer = AutoTokenizer.from_pretrained("rajpurkar/gpt2-squad")
```

✓ 4.2s

Python

We can now tokenize some text, including some context and the start of a question:

```

start_text = ("A speedrun is a playthrough of a video game, \
or section of a video game, with the goal of \
completing it as fast as possible. Speedruns \
often follow planned routes, which may incorporate sequence \
breaking, and might exploit glitches that allow sections to \
be skipped or completed more quickly than intended. ")

prompt = "What is the"
inputs = tokenizer(
    start_text + prompt,
    add_special_tokens=False,
    return_tensors="pt")["input_ids"]

```

Finally, we can now pass this input into the model for generation:

```

prompt_length = len(tokenizer.decode(inputs[0]))
outputs = model.generate(
    inputs,
    max_length=100,
    do_sample=True,
    top_k=50,
    top_p=0.95,
    temperature=0.9,
    num_return_sequences=3)

generated = prompt + tokenizer.decode(outputs[0])[prompt_length + 1:]

print(tokenizer.decode(outputs[0]))

```

The generate function is one we haven't seen before and has a lot of arguments that it takes in. The generation isn't the main focus of our lecture, but if you're curious, Huggingface has great walkthroughs [here](#) & [here](#).

Let's see what the example says:

```

A speedrun is a playthrough of a video game, or section of a video game, with
the goal of completing it as fast as possible. Speedruns often follow planned
routes, which may incorporate sequence breaking, and might exploit glitches
that allow sections to be skipped or completed more quickly than intended.
What is the purpose of speedrunning in a video game?<|endoftext|>

```

And there we have it – our own model used for autocompleting a question! Awesome!