



ちょっと話題の記事

## Hugging FaceでOpenAIの音声認識“Whisper”をFine Tuningする方法が公開されました

#Python    #機械学習    #Deep Learning    #音声認識    #OpenAI    #Hugging Face

 nokomoro3

⌚ 2022.11.09

    
47    11    50

こんちには。

データアナリティクス事業本部 機械学習チームの中村です。

OpenAIがリリースしたWhisperについて、先日Hugging FaceのブログでHugging Faceのフレームワークを用いたfine-tuningの実行方法が紹介されました。

### Fine-Tune Whisper For Multilingual ASR with 😊 Transformers

We're on a journey to advance and democratize artificial intelligence through open source and open science.

 huggingface.co 



また著名なHugging Faceからfine-tuningの実装がリリースされたことで、今後様々なシーンでの応用の可能性を感じます。

Hugging FaceブログではHindi語を例として実施していますが、今回はこちらについて、日本語データを例にしながら動作を確認していきたいと思います。

## 概要

---

本記事では、Hugging Faceのフレームワークを用いたfine-tuningの実行を、日本語データセットを例に見ていきます。

なお本記事の内容は実験の簡易化として、以下の前提条件で実行しています。

- fine tuning用のデータセットは一部のみを使用
- Whisperのモデルサイズはsmallを使用
- 学習のstep数をHugging Faceブログの記載値から1/100に縮小

## 実行環境

---

今回はGoogle Colaboratory環境で実行しました。

ハードウェアなどの情報は以下の通りです。

- GPU: Tesla P100 (GPUメモリ16GB搭載)
- CUDA: 11.1
- メモリ: 13GB

主なライブラリのバージョンは以下となります。

- transformers: 4.25.0.dev0
- datasets: 2.6.1



各種パッケージのインストールはHugging Faceブログに従います。

```
1 | !add-apt-repository -y ppa:jonathonf/ffmpeg-4
2 | !apt update
3 | !apt install -y ffmpeg
4 |
5 | !pip install datasets>=2.6.1
6 | !pip install git+https://github.com/huggingface/transformers
7 | !pip install librosa
8 | !pip install evaluate>=0.30
9 | !pip install jiwer
10 | !pip install gradio
```

## データセットの取得

今回はCommon Voiceデータセットの日本語サブセットを使用します。以下でサンプルを試聴することも可能です。

### [mozilla-foundation/common\\_voice\\_11\\_0 · Datasets at Hugging Face](#)

We're on a journey to advance and democratize artificial intelligence through open source and open science.

[huggingface.co](https://huggingface.co)

Common VoiceはMozillaが、誰でも自由に音声データを使った音声認識モデル開発ができる目的として、ボランティアによる協力によりデータセットを作成しているプロジェクトです。

以下ページでアカウントを作つて、「話す」と「聞く」を隙間時間で実施することで誰でもデータ収集に貢献することができます。(私自身もデータ収集を何回か実施したことがあります)

### Mozilla Common Voice

[commonvoice.mozilla.org](https://commonvoice.mozilla.org) 11 users



# HuggingFace Hubへのログイン

データセットの取得には認証が必要であるため、以下でログインします。

```
1 | from huggingface_hub import notebook_login  
2 | notebook_login()
```

ノートブック上でログイン画面が出力されるため、トークンを入力してください。

トークンが無い場合は、Hugging Faceのアカウントを作成の上、設定画面でAccess Tokensを選択して、アクセストークンを作成します。

The screenshot shows the Hugging Face Hub settings interface. On the left, there's a sidebar with options: Profile, Account, Organizations, Billing, Access Tokens (which is selected and highlighted in blue), GPG Keys, Notifications, and Theme. The main content area is titled "Access Tokens". It has a sub-section "User Access Tokens" with a detailed description: "Access tokens programmatically authenticate your identity to the Hugging Face Hub, allowing applications to perform specific actions specified by the scope of permissions (read, write, or admin) granted. Visit [the documentation](#) to discover how to use them." Below this is a token entry field with placeholder text "notebook execution by book of ml-transformers WRITE" and a "Manage" dropdown. To the right of the token field is a "Show" button with a copy icon. At the bottom of the "User Access Tokens" section is a "New token" button.

トークンを入力すると以下で成功したことが確認できます。

```
1 | Login successful  
2 | Your token has been saved to /root/.huggingface/token
```

# データセットのロード



```
1 from datasets import load_dataset, DatasetDict
2
3 common_voice = DatasetDict()
4
5 common_voice["train"] = load_dataset("mozilla-foundation/common_voice_11_0"
6 ...., "ja", split="train", use_auth_token=True)
7 common_voice["validation"] = load_dataset("mozilla-foundation/common_voice_11_0"
8 ...., "ja", split="validation", use_auth_token=True)
9 common_voice["test"] = load_dataset("mozilla-foundation/common_voice_11_0"
10 ...., "ja", split="test", use_auth_token=True)
```

DatasetDictは以下のような構成になっています。

```
1 print(common_voice)
```

```
1 DatasetDict({
2     train: Dataset({
3         features: ['client_id', 'path', 'audio', 'sentence', 'up_votes', 'down_v'
4 ...., 'age', 'gender', 'accent', 'locale', 'segment'],
5         num_rows: 6505
6     })
7     validation: Dataset({
8         features: ['client_id', 'path', 'audio', 'sentence', 'up_votes', 'down_v'
9 ...., 'age', 'gender', 'accent', 'locale', 'segment'],
10        num_rows: 4485
11    })
12    test: Dataset({
13        features: ['client_id', 'path', 'audio', 'sentence', 'up_votes', 'down_v'
14 ...., 'age', 'gender', 'accent', 'locale', 'segment'],
15        num_rows: 4604
16    })
17 })
```

## データセットのサンプル確認

ひとつサンプルを確認してみましょう。



```

1  {'client_id': '02a8841a00d762472a4797b56ee01643e8d9ece5a225f2e91c007ab1f94c49c99',
2   'path': '/root/.cache/huggingface/datasets/downloads/extracted/0bd2523a2521d778',
3   'audio': {'path': '/root/.cache/huggingface/datasets/downloads/extracted/0bd252',
4   'array': array([-2.1230822e-08, 1.3050334e-08, 1.8151752e-08, ...,
5   ..... 2.8415348e-05, 3.8682865e-05, 2.0064232e-05], dtype=float32),
6   'sampling_rate': 48000},
7   'sentence': 'わたしは音楽が好きです。',
8   'up_votes': 2,
9   'down_votes': 0,
10  'age': '',
11  'gender': '',
12  'accent': '',
13  'locale': 'ja',
14  'segment': ''}
```

'audio' カラムの部分を見ると、音声ファイル自体は `/root/.cache` というキャッシュディレクトリに保存されていることが分かります。

またこの中 'audio' カラムの中にarrayという名前で音声データのバイナリが格納されているようです。

発話内容は 'sentence' というカラムに格納されているようです。

これらの情報は自作のデータセットを準備する場合にヒントになりそうですね。

## 不要なカラムの削除

不要なカラムは以下で削除します。

```

1  common_voice = common_voice.remove_columns(["accent", "age", "client_id"
2   ...., "down_votes", "gender", "locale", "path", "segment", "up_votes"])
3  common_voice
```

```

1  DatasetDict({
2   .. train: Dataset({
3     .. features: ['audio', 'sentence'],
4     .. num_rows: 6505
5   .. })
```



```

8     .... features: ['audio', 'sentence'],
9     .... num_rows: 4485
10    .... })
11    .... test: Dataset({
12     .... features: ['audio', 'sentence'],
13     .... num_rows: 4604
14    .... })
15  })

```

また、ここで実験の簡易化のために必要に応じてデータセットを縮小します。

```

1 # 実験のためデータセットを縮小したい場合はコチラを有効化
2 common_voice = DatasetDict({
3     "train": common_voice['train'].select(range(100)),
4     "validation": common_voice['validation'].select(range(100)),
5     "test": common_voice['test'].select(range(100)),
6 })

```

```

1 DatasetDict({
2     train: Dataset({
3         .... features: ['audio', 'sentence'],
4         .... num_rows: 100
5     })
6     validation: Dataset({
7         .... features: ['audio', 'sentence'],
8         .... num_rows: 100
9     })
10    test: Dataset({
11        .... features: ['audio', 'sentence'],
12        .... num_rows: 100
13    })
14 })

```

## データセットの前処理

---

### サンプリングレートの設定



へへ。

```
1 | from datasets import Audio  
2 |  
3 | common_voice = common_voice.cast_column("audio", Audio(sampling_rate=16000))
```

この設定は、後述するmapなどで実際にデータ処理する際にサンプリングレートの変換が実行される形となります。

元データ自体を変換するようなキャストではないのでご注意ください。

## FeatureExtractorのロード

音声データを音響特徴量に変換するFeatureExtractorをロードします。

```
1 | from transformers import WhisperFeatureExtractor  
2 |  
3 | feature_extractor = WhisperFeatureExtractor.from_pretrained("openai/whisper-smal
```

◀ ▶

サンプルデータでテストをしてみます。

```
1 | # サンプル  
2 | batch = common_voice["train"][0]  
3 |  
4 | # 変換実行  
5 | audio = batch["audio"]  
6 | input_features = feature_extractor(audio["array"]  
7 | ... , sampling_rate=audio["sampling_rate"]).input_features[0]  
8 |  
9 | print(input_features.shape)  
  
1 | (80, 3000)
```

`input_features[0]` の `[0]` は、`feature_extractor`は30秒単位でデータを分割して処理するためと考えられます。



得られた音響特徴量は、80次元の特徴量が、10msec単位で生成されるため、30秒分の3000個作成されています。

音響特徴量については以下の記事でも言及しましたので、興味のある方はご覧ください。

## OpenAIがリリースした音声認識モデル"Whisper"の使い方をまとめてみた | DevelopersIO

こんちには。データアナリティクス事業本部 機械学習チームの中村です。OpenAIがリリースしたWhisperについて、先日は僕はもう少し深掘ることで、様々な使い方がわかつってきたのでシ ...

dev.classmethod.jp 12 users

## Tokenizerのロード

次にTokenizerをロードします。

```

1 from transformers import WhisperTokenizer
2
3 tokenizer = WhisperTokenizer.from_pretrained("openai/whisper-small"
4     , language="Japanese", task="transcribe")

```

こちらもサンプルデータでテストしてみます。(これはHugging Faceブログに記載された通りです)

```

1 input_str = common_voice["train"][0]["sentence"]
2 labels = tokenizer(input_str).input_ids
3 decoded_with_special = tokenizer.decode(labels, skip_special_tokens=False)
4 decoded_str = tokenizer.decode(labels, skip_special_tokens=True)
5
6 print(f"Input: ..... {input_str}")
7 print(f"Decoded w/ special: ... {decoded_with_special}")
8 print(f"Decoded w/out special: {decoded_str}")
9 print(f"Are equal: ..... {input_str == decoded_str}")

```

Input: わたしは音楽がすきです。

Decoded w/ special: ... <|startoftranscript|><|ja|><|transcribe|><|notimestamps|>



## Processorのロード

FeatureExtractorとTokenizerが結合されたProcessorというクラスも準備されています。

```
1 from transformers import WhisperProcessor  
2  
3 processor = WhisperProcessor.from_pretrained("openai/whisper-small"  
4 ... , language="Japanese", task="transcribe")
```

こちらを使用してFeatureExtractorとTokenizerの処理を実施できますので、以降はこちらを使用します。

## データセットへの適用

データセットへのmap処理のため、関数を定義します。

```
1 def prepare_dataset(batch):  
2     # load and resample audio data from 48 to 16kHz  
3     audio = batch["audio"]  
4  
5     # compute log-Mel input features from input audio array  
6     batch["input_features"] = processor.feature_extractor(audio["array"]  
7     ... , sampling_rate=audio["sampling_rate"]).input_features[0]  
8  
9     # encode target text to label ids  
10    batch["labels"] = processor.tokenizer(batch["sentence"]).input_ids  
11    return batch
```

map処理をします。スペックに応じて `num_proc` を調整してください。

私は安定実行のため `num_proc=1` で実施しました。(Hugging Faceブログの通りに `num_proc=4` とすると途中で止まってしまいました)

```
1 common_voice = common_voice.map(prepare_dataset  
2 ... )
```



こちらの処理は30秒程度で終わります。

データセットを縮小しない場合、こちらの処理には30分程度かかりますので、ご注意ください。

## 学習の準備

---

### DataCollatorクラスを定義

DataCollatorクラスを定義します。

```
import torch

from dataclasses import dataclass
from typing import Any, Dict, List, Union

@dataclass
class DataCollatorSpeechSeq2SeqWithPadding:
    processor: Any

    def __call__(self, features: List[Dict[str, Union[List[int], torch.Tensor]]]) -> Dict[str, torch.Tensor]:
        # 音響特微量側をまとめの処理
        # (一応バッチ単位でパディングしているが、すべて30秒分であるはず)
        input_features \
            = [{"input_features": feature["input_features"]} for feature in features]
        batch = self.processor.feature_extractor.pad(input_features, return_tensors="pt")

        # トーン化された系列をバッチ単位でパディング
        label_features = [{"input_ids": feature["labels"]} for feature in features]
        labels_batch = self.processor.tokenizer.pad(label_features, return_tensors="pt")

        # attention_maskが0の部分は、トークンを-100に置き換えてロス計算時に無視させる
        # -100を無視するのは、PyTorchの仕様
        labels \
            = labels_batch["input_ids"].masked_fill(labels_batch.attention_mask == 0, -100)

        # BOSトークンがある場合は削除
        if (labels[:, 0] == self.processor.tokenizer.bos_token_id).all().cpu().item():
            labels = labels[:, 1:]

        return {"input_ids": input_features, "attention_mask": labels}
```



```
    # 整形したlabelsをハッシュにまとめ  
33     batch["labels"] = labels  
34  
35     return batch
```

このクラスは、バッチ単位で実行する必要のある学習直前の前処理を担当するイメージです。

確認のためちょっと単体で動かしてみます。音響特徴量の部分はスキップして `tokenizer.pad` からです。

`tokenizer.pad` により、バッチ単位で長さが統一され、`input_ids`に何かしらのトークンが paddingされています

加えてattention\_maskも得られます。（今回は正解データの方なので、attention\_maskは入力としては使用しません）

paddingに使用されているトークンは、`endoftext`トークンのようです。

```
1 | processor.tokenizer.decode(labels_batch["input_ids"][0])
```



从此处开始，使用attention\_mask将input\_ids替换为-100。

```
1 | labels = labels_batch["input_ids"].masked_fill(labels_batch.attention_mask.ne(1))
2 | labels
```

```
1 | tensor([[50258, 50266, 50359, 50363, 9206, 3368, 2849, 3065, 18034, 35479,
2 | ... 5142, 2659, 7016, 4767, 1543, 50257, -100, -100, -100, -100,
3 | ... -100, -100],
4 | ... [50258, 50266, 50359, 50363, 3203, 20258, 42540, 1231, 8661, 12979,
5 | ... 2972, 14765, 4035, 3065, 7391, 120, 32156, 1764, 9311, 11046,
6 | ... 1543, 50257],
7 | ... [50258, 50266, 50359, 50363, 24168, 5998, 34719, 18549, 1764, 7625,
8 | ... 9311, 5591, 1543, 50257, -100, -100, -100, -100, -100,
9 | ... -100, -100]])
```

通过将padding的令牌设置为-100，可以在PyTorch计算损失时忽略它们。

这是PyTorch的实现方式，例如在CrossEntropyLoss中使用`ignore_index=-100`。

## CrossEntropyLoss — PyTorch 2.0 documentation

 pytorch.org

稍后将通过DataCollator进行实例化。

```
1 | data_collator = DataCollatorSpeechSeq2SeqWithPadding(processor=processor)
```

## 评估函数的准备



へへ。

ただし日本語は単語間に空白がないため、Hugging Faceブログから少し修正が必要です。

以下の記事を参考に、Ginza経由でmecab-ipadic-NEologdを使います。

## mecab ipadic-NEologd を Google Colaboratory で使う - Qiita

これはなに？ 形態素解析のデファクトスタンダードであるMeCabと、その追加辞書mecab-ipadic-NEologdをGoogle Colaboratoryではmecabとipadic-NEo...

qiita.com 2 users

まずはパッケージを追加で入れます。

```
1 | !pip install ginza==4.0.5 ja-ginza
2 | !pip install sortedcontainers~=2.1.0
3 | import pkg_resources, imp
4 | imp.reload(pkg_resources)
```

評価関数を以下のようにします。(ハイライトが手を入れた箇所です)

```
import evaluate
import spacy
import ginza

metric = evaluate.load("wer")
nlp = spacy.load("ja_ginza")
ginza.set_split_mode(nlp, "C") # CはNEologdの意らしいです

def compute_metrics(pred):
    pred_ids = pred.predictions
    label_ids = pred.label_ids

    # replace -100 with the pad_token_id
    label_ids[label_ids == -100] = processor.tokenizer.pad_token_id

    # we do not want to group tokens when computing the metrics
    pred_str = processor.tokenizer.batch_decode(pred_ids, skip_special_tokens=True)
    label_str = processor.tokenizer.batch_decode(label_ids, skip_special_tokens=True)

    # 分かち書きして空白区切りに変換
```



```
24     wer = 100 * metric.compute(predictions=pred_str, references=label_str)
25
26     return {"wer": wer}
```

## モデルのロード・設定

Whisperのモデルをロードします。

```
1 from transformers import WhisperForConditionalGeneration
2
3 model = WhisperForConditionalGeneration.from_pretrained("openai/whisper-small")
```

モデルのconfigを以下のように設定します。

```
1 model.config.forced_decoder_ids \
2     = processor.get_decoder_prompt_ids(language = "ja", task = "transcribe")
3 model.config.suppress_tokens = []
```

す。

`model.config.forced_decoder_ids` はデコード時に与えるトークンをあらかじめ指定することができます。

言語判定など、どこから含めてfine tuningするかによって、

`model.config.forced_decoder_ids` に設定すべきものが変わってきます。

今回は、WERなどをもっともらしくするために、あらかじめ与えた形で実施します。

この設定で具体的には以下のトークンがデコーダに強制的に挿入されます。

```
1 processor.tokenizer.decode([i[1] for i in model.config.forced_decoder_ids])
```



## TrainingArgumentsの定義

通常のHugging FaceでのTransformersの学習と同様に、TrainingArgumentsを定義します。

今回は省略のため、Hugging Faceブログよりもstep数を1/100と少なくしています。

```
1 from transformers import Seq2SeqTrainingArguments
2
3 training_args = Seq2SeqTrainingArguments(
4     output_dir="../whisper-small-ja", # change to a repo name of your choice
5     per_device_train_batch_size=16,
6     gradient_accumulation_steps=1, # increase by 2x for every 2x decrease in batch size
7     learning_rate=1e-5,
8     # warmup_steps=500, # Hugging Faceブログではこちら
9     warmup_steps=5,
10    # max_steps=4000, # Hugging Faceブログではこちら
11    max_steps=40,
12    gradient_checkpointing=True,
13    fp16=True,
14    group_by_length=True,
15    evaluation_strategy="steps",
16    per_device_eval_batch_size=8,
17    predict_with_generate=True,
18    generation_max_length=225,
19    # save_steps=1000, # Hugging Faceブログではこちら
20    save_steps=10,
21    # eval_steps=1000, # Hugging Faceブログではこちら
22    eval_steps=10,
23    logging_steps=25,
24    report_to=["tensorboard"],
25    load_best_model_at_end=True,
26    metric_for_best_model="wer",
27    greater_is_better=False,
28    push_to_hub=False,
29 )
```

## Trainerの定義



```

1 from transformers import Seq2SeqTrainer
2
3 trainer = Seq2SeqTrainer(
4     ... args=training_args,
5     ... model=model,
6     ... train_dataset=common_voice["train"],
7     ... eval_dataset=common_voice["validation"],
8     ... data_collator=data_collator,
9     ... compute_metrics=compute_metrics,
10    ... tokenizer=processor.feature_extractor,
11 )

```

## 学習

---

### 学習前の性能チェック

一応事前に性能をチェックしておきます。 `trainer.predict` を実行すればreturnされる `PredictionOutput` の `metrics` にWERが保存されていますので、そちらを参照します。

```

1 import pandas as pd
2 pd.DataFrame([
3     {"split": "train",
4      "wer": trainer.predict(common_voice["train"]).metrics["test_wer"]},
5     {"split": "validation",
6      "wer": trainer.predict(common_voice["validation"]).metrics["test_wer"]},
7     {"split": "test",
8      "wer": trainer.predict(common_voice["test"]).metrics["test_wer"]}
9 ])

```

	split	wer
0	train	18.200409
1	validation	20.653907
2	test	25.281804



学習は以下のコードで実行します。

```
1 | trainer.train()
```

以下が実行ログとなります。

Step	Training Loss	Validation Loss	Wer
10	No log	1.700416	16.267943
20	No log	0.958344	16.267943
30	2.163600	0.676422	17.384370
40	2.163600	0.639414	16.586922

学習は10分以下で完了しました。

ちなみにstep数を1/100にしない場合は、学習に6時間程度かかると予測がでていましたので、ご参考にされてください。

## 学習後の性能チェック

一応以下が結果となります。学習前より少し改善していますが、性能としてはまだまだな印象です。

split	wer
0 train	9.304703
1 validation	16.267943
2 test	19.243156

ただし再掲ですが、本記事の実行は以下の制約がある前提となりますのでご承知おきください。

- fine tuning用のデータセットは一部のみを使用
- Whisperはsmallモデルを使用
- 学習のstep数を1/100に縮小



参考までに少しだけlargeを動かしてみたので、その際に気づいた注意点を挙げておきます。

- Colabのメモリ設定をハイメモリとする必要がありました。
  - ハイメモリではない場合、モデルのロードでこけてしまいました
- Whisper単体では、標準メモリでも動作しますので、用途に応じて使い分けを検討

## 推論方法の確認

---

最後に推論方法について、Hugging Faceでの実行方法を確認しておきます。

既に音響特徴量が作成されていれば、`trainer.predict` を実行してその後  
`tokenizer.decode` で結果を得られます。

```

1 prediction_output = trainer.predict(common_voice["test"].select([0]))
2 pred_ids = prediction_output.predictions
3 processor.tokenizer.decode(pred_ids[0], skip_special_tokens=True)

```

1 私は松井さんが書いた作文を読みました。

データセットから実行し直す場合は、以下の方法でも可能なようです。

```

1 # カラムを消してしまったため再度ロード
2 common_voice_test = load_dataset("mozilla-foundation/common_voice_11_0"
3 , "ja", split="test", use_auth_token=True)
4 common_voice_test = common_voice_test.select(range(1))
5 common_voice_test = common_voice_test.cast_column("audio", Audio(sampling_rate=1
6
7 device = "cuda" if torch.cuda.is_available() else "cpu"
8
9 # 推論
10 speech_data = common_voice_test['audio'][0]["array"]
11 inputs = processor.feature_extractor(speech_data
12 , return_tensors="pt", sampling_rate=16_000).input_features.to(device)
13 predicted_ids = model.generate(inputs, max_length=480_000)
14 processor.tokenizer.batch_decode(predicted_ids, skip_special_tokens=False)[0]

```



## まとめ

---

いかがでしたでしょうか。

Whisperのfine tuningについて、著名なHugging Faceからの実装がリリースされたことで、今後様々なシーンで応用が広がりそうということで本記事でも紹介しました。自社のデータが蓄積されている方は、ぜひお試ししてみてはいかがでしょうか。

データセットを自作する方法については、あと少し調査が必要そうですので、また今後ブログにしたいと思います。

本記事がWhisperを活用してみようと思われる方の参考になれば幸いです。

---

## 機械学習システム導入をご支援します

機械学習の専門知識と豊富な導入実績を持つクラスメソッドが、AWSの機械学習サービス（SageMaker、Personalize、Forecast、Comprehend、Rekognitionなど）を活用したシステム開発・導入を支援します。

お持ちのデータが機械学習に適しているか診断するデータ初期診断も実施中！お気軽にご相談ください。

[機械学習システム導入支援の詳細を見る](#)

また、現在クラスメソッドでは無料の相談会を開催しています。社内に蓄積されたデータの活用方法や機械学習についての技術的な相談など、機械学習に関するお悩みがありましたら、ぜひご



この記事をシェアする



## 関連記事





♥ 15

♥ 2

おざわ (じ)

2023.05.23

k\_uehara

2023.05.22



smart\_openでS3のオブジェクトをストリーミング処理してみる

♥ 1

kobayashi.m

2023.05.22



AWS Parameters and Secrets Lambda Extensionを使用してみた

♥ 2

SUMANGALAS

2023.05.18

---

© Classmethod, Inc. All rights reserved.

---