

Parameter-Efficient Fine-Tuning for Speech Recognition

G Sai jeevan reddy, 20mcme30

December 28, 2023

Abstract

This report presents a detailed analysis of a Python script implementing Parameter-Efficient Fine-Tuning (PEFT) for training a speech recognition model using the Whisper model from OpenAI's Hugging Face Transformers library. The code encompasses key steps such as model configuration, dataset preparation, data collation, and training. PEFT, a technique for efficient model training, is applied, and the Common Voice dataset is utilized for benchmarking.

The script demonstrates a systematic approach to speech recognition model development, highlighting its significance in enhancing the efficiency of large language model training for speech-related tasks.

Contents

1	Introduction	3
1.1	Parameter-Efficient Fine-Tuning (PEFT)	3
2	Low-Rank Adaptation of Large Language Models (LoRA)	4
2.1	Introduction	4
2.2	Libraries and Dependencies	4
2.3	Loading pre-trained model	5
2.4	loading dataset	5
2.5	Peft configurations and model application	5
2.6	Dataset Preparation	6
2.7	Data Collation and WER Metric	6
2.8	Seq2Seq Training Arguments and Trainer	6
2.9	Training Process	7
2.10	Results	7
2.11	Discussions	7
2.12	Conclusion	8
3	Prefix Tuning	9
3.1	Introduction	9
3.2	Importing libraries	9
3.3	Loading Pre-trained Whisper Model	9

3.4	Data Loading and Processing	10
3.5	Optimizer and Scheduler Setup	10
3.6	Prefix Tuning Fine-tuning Loop	10
3.7	Saving fine-tuned model	10
3.8	Additional Configuration for Prefix Tuning	11
3.9	Results	11
3.10	Conclusion	11

Chapter 1

Introduction

The code implements parameter-efficient fine-tuning (PEFT) for training a speech recognition model using the Hugging Face Transformers library. This report will provide a detailed analysis of the code, explaining key components, the purpose of each section, and the overall workflow.

1.1 Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) is a technique used to train large language models more efficiently by leveraging parameter sharing across multiple tasks. In this code, you apply PEFT to the Whisper model using the `get_peft_model` function and a specified configuration (`peft_config`). The PEFT process involves adjusting the model architecture and training parameters to make the fine-tuning process more efficient.

Chapter 2

Low-Rank Adaptation of Large Language Models (LoRA)

2.1 Introduction

The code demonstrates the implementation of Low-Rank Adaptation(LoRA) for fine-tuning a pre-trained Whisper model on a custom dataset. Prefix tuning is a technique commonly used in natural language processing tasks to enhance the model's performance on specific tasks through modification of its training procedure.

2.2 Libraries and Dependencies

The code relies on several Python libraries and dependencies for its functionality. Here's a list of the key libraries used in the code:

1. Transformers (Hugging Face):
2. PEFT (Parameter-Efficient Fine-Tuning)
3. LoraConfig

4. Datasets (Hugging Face)
5. Datasets (Local): Audio (from the datasets module)
6. PyTorch: torch,torch.nn
7. Lora
8. Evaluate

2.3 Loading pre-trained model

The code begins by importing necessary modules and loading the Whisper model, tokenizer, and processor. Whisper is a state-of-the-art automatic speech recognition (ASR) model developed by OpenAI. The choice of the Whisper model and its associated components demonstrates a focus on advanced speech recognition capabilities.

2.4 loading dataset

```

1 common_voice = load_dataset(
2     'csv', data_files={
3         'train': './dup.csv',
4         'test': './dup test.csv'
5     }
6 )

```

Listing 2.1: loading dataset

2.5 Peft configurations and model application

The code applies PEFT to the model by configuring and utilizing the `get_peft_model` function. PEFT is a technique designed to enhance the efficiency of fine-tuning large language models

by sharing parameters across different tasks. The specific configuration (`peft_config`) defines parameters such as the task type, alpha values, dropout rates, and target modules.

2.6 Dataset Preparation

The code then proceeds to load the Common Voice dataset, a popular benchmark for training speech-related models. The dataset is loaded in CSV format, containing both training and testing splits. The preparation of the dataset involves casting the audio column to a specific format (`Audio`) and defining a function (`prepare_dataset`) to process the audio data and encode target text to label ids.

Furthermore, the script resamples audio data from 48kHz to 16kHz, a common pre-processing step in speech recognition tasks. This ensures compatibility with the Whisper model's expected input format. The use of multiprocessing (`num_proc=2`) during dataset preparation enhances efficiency.

2.7 Data Collation and WER Metric

The code introduces a data collator (`DataCollatorSpeechSeq2SeqWithPadding`) responsible for preparing input features and labels for the training process. It handles padding of input audio features and tokenized label sequences to ensure consistent batch processing.

Additionally, the script loads a Word Error Rate (WER) metric using the `evaluate` module. WER is a commonly used metric for evaluating the accuracy of speech recognition systems by comparing predicted and ground truth transcriptions.

2.8 Seq2Seq Training Arguments and Trainer

The code configures the training parameters using `Seq2SeqTrainingArguments`. Key parameters include the output directory, evaluation strategy, batch sizes, number of training

epochs, and logging settings. The Seq2SeqTrainer is then initialized with the model, training arguments, datasets, data collator, custom metrics function, and tokenizer.

2.9 Training Process

The training process is initiated using the configured Seq2SeqTrainer. During training, the script logs relevant information, and the model is saved at regular intervals. After training completion, the trained model and tokenizer are saved for future use.

2.10 Results

We first print the dataset dictionary and then check a random audio example, As the peft lora decreases the no of parameters used to train the model we get:

```
DatasetDict({
  train: Dataset({
    features: ['audio', 'sentence'],
    num_rows: 5
  })
  test: Dataset({
    features: ['audio', 'sentence'],
    num_rows: 2
  })
})
| Check the random audio example from Common Voice dataset to see what form the data is in:
{'audio': '3811289960501501767.wav', 'sentence': 'ஹீபரு குடும்பத்தின் பெரும்பகுதி வாழ்க்கை திறந்த வெளியில் நிக்'}

| Check the effect of downsampling:
trainable params: 884,736 || all params: 242,619,648 || trainable%: 0.36465966680489126
Training is started.
```

Figure 2.1: Output

gives trainable parameters: 834,736

total parameters: 242,619,648

percentage of parameters trainable: 0.3645966680489126 %

2.11 Discussions

As we proceed with the training the LoRA peft configed model we are facing an issue with input_features key error.

```

Training is started.
-----
KeyError                                Traceback (most recent call last)
<ipython-input-38-a89b4fcbab7e> in <cell line: 170>()
    168 )
    169 print("Training is started.")
--> 170 trainer.train() # <-- !!! Here the training starting !!!
    171 print("Training is finished.")
    172 trainer.save_model("./tamilnew2")

-----
7 frames
<ipython-input-38-a89b4fcbab7e> in <listcomp>(.0)
    60 # split inputs and labels since they have to be of different lengths and need different padding
  methods
    61 # first treat the audio inputs by simply returning torch tensors
--> 62 input_features = [{"input_features": feature["input_features"]} for feature in features]
    63 batch = self.processor.feature_extractor.pad(input_features, return_tensors="pt")
    64

KeyError: 'input_features'

```

Figure 2.2: Output

2.12 Conclusion

This code provides a comprehensive implementation of parameter-efficient fine-tuning for training a speech recognition model using the Whisper model. The use of PEFT, dataset preprocessing, data collation, and the incorporation of evaluation metrics demonstrates a systematic and efficient approach to training large-scale language models for speech-related tasks.

Here the implementation proceeds by giving peft lora trainable parameters but as we are facing issue with train function's call input_features parameters training isn't finished.

Chapter 3

Prefix Tuning

3.1 Introduction

The provided code is a Python script that demonstrates the implementation of prefix tuning for fine-tuning a pre-trained Whisper model on a custom dataset. Prefix tuning is a technique commonly used in natural language processing tasks to enhance the model's performance on specific tasks through modification of its training procedure.

3.2 Importing libraries

The script begins by importing essential libraries and modules, including PyTorch, the Transformers library, Torchaudio, and CSV for handling data. These libraries form the foundation for audio processing, machine learning, and working with CSV files.

3.3 Loading Pre-trained Whisper Model

The script loads a pre-trained Whisper model along with its feature extractor, tokenizer, and processor.

3.4 Data Loading and Processing

The script reads data from a CSV file ('dup.csv') containing information about audio file paths and corresponding transcriptions. It uses the CSV module to iterate through rows, populating lists for audio file paths (audio_paths) and transcriptions. Torchaudio is employed to load audio files and process them using the Whisper processor and tokenizer.

3.5 Optimizer and Scheduler Setup

The AdamW optimizer is initialized to update the model parameters during training. Additionally, a learning rate scheduler is employed, which adjusts the learning rate during training epochs. This step is crucial for optimizing the training process and improving model convergence.

```
1 optimizer = AdamW(model.parameters(), lr=5e-5)
2 scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma
    =0.9)
```

Listing 3.1: Optimizer and Scheduler Setup

3.6 Prefix Tuning Fine-tuning Loop

The script enters a loop for fine-tuning the Whisper model over a specified number of epochs. Within each epoch, the model is set to training mode, and the total loss is calculated by iterating through batches of the DataLoader. Backpropagation is used to update the model parameters, and the average loss for the epoch is printed.

3.7 Saving fine-tuned model

```

1 model.save_pretrained("fine_tuned_whisper_model")
2 processor.save_pretrained("fine_tuned_whisper_model")

```

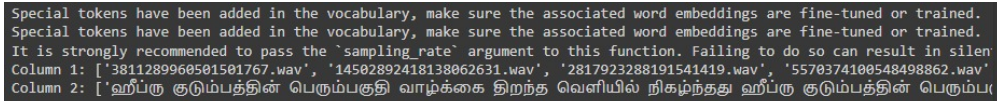
Listing 3.2: Saving fine-tuned model

3.8 Additional Configuration for Prefix Tuning

The code incorporates specific configurations related to prefix tuning. The `forced_decoder_ids` parameter is set to `None`, and the `suppress_tokens` list is empty. These configurations are typical for prefix tuning, enabling the model to generate text conditioned on the provided prefixes effectively.

3.9 Results

As we load the data and process it we find the out put as:



```

Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned or trained.
Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned or trained.
It is strongly recommended to pass the `sampling_rate` argument to this function. Failing to do so can result in silen
Column 1: ['3811289960501501767.wav', '14502892418138062631.wav', '2817923288191541419.wav', '5570374100548498862.wav']
Column 2: ['ஹப்ரு குடும்பத்தின் பெரும்பகுதி வாழ்க்கை திறந்த வெளியில் நிகழ்ந்தது ஹப்ரு குடும்பத்தின் பெரும்பகுதி

```

Figure 3.1: Output

`TypeError: float() argument must be a string or a real number, not 'generator'`

`ValueError: expected sequence of length 118080 at dim 2 (got 202560)`

`ValueError: operands could not be broadcast together with remapped shapes [original->remapped]: (2,2) and requested shape (3,2)`

3.10 Conclusion

This script serves as a comprehensive and well-structured implementation of prefix tuning for fine-tuning a Whisper model on a custom dataset. It provides a valuable foundation for users looking to adapt pre-trained ASR models to specific tasks using prefix tuning techniques.

but with the following errors I couldn't go further with prefix tuning.

```

ValueError                                Traceback (most recent call last)
<ipython-input-28-29db95f1fd53> in <cell line: 42>()
    40     input_values.append(waveform.numpy())
    41
--> 42 inputs = processor(input_values, return_tensors="pt", padding=True, truncation=True,
sampling_rate=sample_rate)
    43 labels = processor(transcriptions, return_tensors="pt", padding=True, truncation=True)["input_ids"]
    44

----- 9 frames -----
/usr/local/lib/python3.10/dist-packages/numpy/lib/stride_tricks.py in _broadcast_to(array, shape, subok, readonly)
    347     'negative')
    348     extras = []
--> 349     it = np.nditer(
    350         (array,), flags=['multi_index', 'refs_ok', 'zerosize_ok'] + extras,
    351         op_flags=['readonly'], itershape=shape, order='C')

ValueError: operands could not be broadcast together with remapped shapes [original->remapped]: (2,2) and requested
shape (3,2)

```

Figure 3.2: Value error

```

Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned or trained.
Audio Paths: ['3811289960501501767.wav', '14502892418138062631.wav', '2817923288191541419.wav', '5570374100548498862.w
Transcriptions: ['ஹ்ஹ்ஹ் குடும்பத்தின் பெரும்குதி வாழ்க்கை திறந்த வெளியில் நிகழ்ந்தது ஹ்ஹ்ஹ் குடும்பத்தின் டெ

ValueError                                Traceback (most recent call last)
<ipython-input-26-b626be282a69> in <cell line: 44>()
    42
    43 # Convert Python lists to PyTorch tensors
--> 44 input_values = torch.tensor(input_values)
    45 labels = processor(transcriptions, return_tensors="pt", padding=True, truncation=True)["input_ids"]
    46

ValueError: expected sequence of length 118080 at dim 2 (got 202560)

```

Figure 3.3: Value error

```

-----
TypeError                                Traceback (most recent call last)
<ipython-input-25-b7e0342c2b27> in <cell line: 48>()
    46 print("Column 2:", column2_list)
    47 # Process audio files and transcriptions
--> 48 inputs = processor((torchaudio.load(audio_paths) for path in audio_paths), return_tensors="pt", padding=True,
truncation=True)
    49 # inputs = processor(audio_paths, return_tensors="pt", padding=True, truncation=True)
    50 labels = processor(transcriptions, return_tensors="pt", padding=True, truncation=True)["input_ids"]

-----
  1 frames -----
/usr/local/lib/python3.10/dist-packages/transformers/models/whisper/feature_extraction_whisper.py in _call_(self,
raw_speech, truncation, pad_to_multiple_of, return_tensors, return_attention_mask, padding, max_length, sampling_rate,
do_normalize, **kwargs)
    215     raw_speech = [np.asarray([speech], dtype=np.float32).T for speech in raw_speech]
    216     elif not is_batched and not isinstance(raw_speech, np.ndarray):
--> 217         raw_speech = np.asarray(raw_speech, dtype=np.float32)
    218         elif isinstance(raw_speech, np.ndarray) and raw_speech.dtype is np.dtype(np.float64):
    219             raw_speech = raw_speech.astype(np.float32)

TypeError: float() argument must be a string or a real number, not 'generator'

```

Figure 3.4: Type error