

OpenAI Fine Tuning Tutorial

This Jupyter notebook provides a comprehensive walkthrough for businesses and individuals looking to maximize ChatGPT's full capabilities. It emphasizes a pivotal fine-tuning feature that aligns AI models with specific business terminologies and needs. As this content is integrated into a Jupyter notebook, you can effortlessly navigate the step-by-step process, enhanced by detailed coding demonstrations. When you have completed this walkthrough, you will have created a fine-tuned model that is customized with your data.

Install Required Libraries

```
In [ ]: !pip3 install openai
```

Importing Libraries

```
In [ ]: import json
import openai
import datetime
import sys
```

Setup OpenAI API Key

- Obtain an OpenAI API key from OpenAI's Platform
<https://platform.openai.com/account/api-keys>
- The API key is provided for free to users with an OpenAI account. However, once you decide to utilize it, charges will be incurred based on the number of tokens you consume. For detailed pricing information, please visit: <https://openai.com/pricing>

```
In [ ]: openai.api_key = 'OPENAI_API_KEY_HERE'
```

Step 1: Validate the Fine Tuning File

- Your fine-tuning file should be in the JSON Lines (jsonl) format. We'll use data.jsonl as a sample for training data.

This code defines a function called `validate_file` that takes a filename as an argument and returns a boolean value indicating whether the file is valid or not. The function attempts to open the file and parse each line as JSON using the `json.loads()` method. If any line in the file is not valid JSON, an exception is raised

and the function returns `False`. If all lines in the file are valid JSON, the function returns `True`.

The code then checks if the file "data.jsonl" is valid by calling the `validate_file` function with the filename as an argument. If the file is not valid, the code prints an error message and exits the program using the `sys.exit()` method. If the file is valid, the code prints a message indicating that the file is valid.

```
In [ ]: def validate_file(filename):
        try:
            with open(filename, 'r') as file:
                for line in file.readlines():
                    json.loads(line)
                return True
        except Exception as e:
            print("Error reading file, invalid format:", e)
            return False

    if not validate_file("data.jsonl"):
        print("File is not valid")
        sys.exit()
    else:
        print("\nFile is valid...\n")
```

Step 2: Upload File to OpenAI

- This step uploads data.jsonl to OpenAI. This might take some time.

The code creates a new file for fine-tuning a language model using the OpenAI API. It opens a file called "data.jsonl" in binary mode and passes it to the `openai.File.create()` method along with the purpose of "fine-tune". The method returns a dictionary containing information about the newly created file, which is printed to the console using the `print()` function.

```
In [ ]: ft_file = openai.File.create(file=open("data.jsonl", "rb"), purpose='fine-tune')
        print(ft_file)
```

Step 3: Check File Status on OpenAI

- Ensure the file status is "processed" before proceeding.

The code defines a function called `pretty_table_s3` that takes a dictionary as an argument and prints a formatted table of information about files stored on the OpenAI API. The function prints a header row and separator line, and then iterates over each file in the dictionary and prints a row for each file using a specified format.

The code then calls the `openai.File.list()` method to retrieve a list of files stored on the OpenAI API, with a limit of 25 files. The resulting list of files is passed to the `pretty_table_s3` function, which prints a formatted table of information about the files.

```
In [ ]: def pretty_table_s3(f):
        header = ['ID', 'Purpose', 'Status', 'Created At']
        row_format = "{:<33} {:<20} {:<12} {}"
        print(row_format.format(*header))
        print('-' * 88)

        for file in f['data']:
            created_at = datetime.datetime.fromtimestamp(file['created_at']).strftime('%Y-%m-%d %H:%M:%S')
            print(row_format.format(file['id'], file['purpose'], file['status'], created_at))

file_list = openai.File.list(limit=25)
pretty_table_s3(file_list)
```

Step 4: Start Fine Tuning at OpenAI

- Ensure the file status is "processed", then initiate the fine-tuning

The code checks the status of a file that was previously created for fine-tuning using the OpenAI API. If the status of the file is "processed", the code creates a new fine-tuning job using the `openai.FineTuningJob.create()` method, passing in the ID of the training file to be used for fine-tuning and the name of the model to be used. The resulting information about the fine-tuning job is printed to the console using the `print()` function. If the status of the file is not "processed", the code prints a message indicating that it is still waiting for the file to be processed.

```
In [ ]: file_status = openai.File.retrieve(ft_file["id"])
        if file_status["status"] == "processed":
            TRAINING_FILE_ID = ft_file["id"]
            ft_job = openai.FineTuningJob.create(training_file=TRAINING_FILE_ID, model=model_id)
            print(ft_job)
        else:
            print(f"\nHOLD...Still waiting on STATUS: {file_status['status']}\n")
```

Step 5: Check Fine Tuning Model Status

- Wait until the status is "succeeded".

The code defines a function called `pretty_table_s5` that takes a dictionary as an argument and prints a formatted table of information about fine-tuning jobs for language models stored on the OpenAI API. The function prints a header row and separator line,

and then iterates over each job in the dictionary and prints a row for each job using a specified format.

The code then calls the `openai.FineTuningJob.list()` method to retrieve a list of fine-tuning jobs stored on the OpenAI API, with a limit of 25 jobs. The resulting list of jobs is passed to the `pretty_table_s5` function, which prints a formatted table of information about the jobs.

The code then retrieves the status of a fine-tuning job using the `openai.FineTuningJob.retrieve()` method. The ID of the job is stored in the `fine_tuning_job` variable. If the status of the job is "succeeded", the code retrieves the ID of the fine-tuned model using the `fine_tuned_model` key in the dictionary returned by the `openai.FineTuningJob.retrieve()` method. The ID of the model is stored in the `model_id` variable, and a message is printed to the console indicating that the model has been created and providing the ID. If the status of the job is not "succeeded", the code prints a message indicating that it is still waiting for the model to be created and providing instructions not to proceed until the model ID is available.

```
In [ ]: def pretty_table_s5(f):
    header = ['ID', 'Created At', 'Finished At', 'Status', 'Fine Tuned Model ID']
    row_format = "{:<33} {:<22} {:<22} {:<13} {"
    print(row_format.format(*header))
    print('-' * 140)

    for job in f['data']:
        created_at = datetime.datetime.fromtimestamp(job['created_at']).strftime('%Y-%m-%d %H:%M:%S')
        finished_at = datetime.datetime.fromtimestamp(job.get('finished_at', 0)).strftime('%Y-%m-%d %H:%M:%S')
        print(row_format.format(job['id'], created_at, finished_at, job['status'], job['fine_tuned_model_id']))

job_list = openai.FineTuningJob.list(limit=25)
pretty_table_s5(job_list)

fine_tuning_job = ft_job["id"]
model_status = openai.FineTuningJob.retrieve(fine_tuning_job)
model_id = None
if model_status["status"] == "succeeded":
    model_id = model_status["fine_tuned_model"]
    print(f"\nModel created, Model ID: {model_id}")
else:
    print("\n***** DO NOT PROCEED YET *****\nStill waiting for the model...")
```

Step 6: Test the Fine-Tuned Model

- Step 5 must have completed successfully with a model ID before proceeding.

The code checks if a `model_id` variable is defined. If the variable is defined, the code generates a response to a customer service message using the OpenAI API. The

response is generated using the `openai.ChatCompletion.create()` method, which takes the ID of the model to be used, the temperature of the response, and a list of messages to be used as context for the response. The response is printed to the console using the `print()` function.

If the `model_id` variable is not defined, the code prints a message indicating that the model is not ready and providing instructions for obtaining the model ID.

```
In [ ]: if model_id:
        completion = openai.ChatCompletion.create(
            model=model_id,
            temperature=0.0,
            messages=[
                {"role": "system", "content": "You are a helpful and professional assistant."},
                {"role": "user", "content": "dude, i need to mail you a check, what do you need?"}
            ]
        )
        print(completion.choices[0].message.content)
    else:
        print("The model isn't ready yet... You need to ensure the model is ready before using it.")
```