Fine-tuning

Learn how to customize a model for your application.

Introduction

This guide is intended for users of the new OpenAI fine-tuning API. If you are a legacy fine-tuning user, please refer to our legacy fine-tuning guide.

Fine-tuning lets you get more out of the models available through the API by providing:

1. Higher quality results than prompting
2. Ability to train on more examples than can fit in a prompt
3. Token savings due to shorter prompts
4. Lower latency requests

GPT models have been pre-trained on a vast amount of text. To use the models effectively, we include instructions and sometimes several examples in a prompt. Using demonstrations to show how to perform a task is often called "few-shot learning."

Fine-tuning improves on few-shot learning by training on many more examples than can fit in the prompt, letting you achieve better results on a wide number of tasks. **Once a model has been fine-tuned, you won't need to provide as many examples in the prompt.** This saves costs and enables lower-latency requests.

At a high level, fine-tuning involves the following steps:

1. Prepare and upload training data
2. Train a new fine-tuned model
3. Use your fine-tuned model

Visit our pricing page to learn more about how fine-tuned model training and usage are billed.

What models can be fine-tuned?

We are working on enabling fine-tuning for GPT-4 and expect this feature to be available later this year.

Fine-tuning is currently available for the following models:

- `gpt-3.5-turbo-0613` (recommended)
- `babbage-002`
- `davinci-002`

We expect `gpt-3.5-turbo` to be the right model for most users in terms of results and ease of use, unless you are migrating a legacy fine-tuned model.

When to use fine-tuning

Fine-tuning GPT models can make them better for specific applications, but it requires a careful investment of time and effort. We recommend first attempting to get good results with prompt engineering, prompt chaining (breaking complex tasks into multiple prompts), and function calling, with the key reasons being:

- There are many tasks for which our models may initially appear to not perform well at, but with better prompting we can achieve much better results and potentially not need to be fine-tune
- Iterating over prompts and other tactics has a much faster feedback loop than iterating with fine-tuning, which requires creating datasets and running training jobs
- In cases where fine-tuning is still necessary, initial prompt engineering work is not wasted - we typically see best results when using a good prompt in the fine-tuning data (or combining prompt chaining / tool use with fine-tuning)

Our GPT best practices guide provides a background on some of the most effective strategies and tactics for getting better performance without fine-tuning. You may find it helpful to iterate quickly on prompts in our playground.

Common use cases

Some common use cases where fine-tuning can improve results:

- Setting the style, tone, format, or other qualitative aspects
- Improving reliability at producing a desired output
- Correcting failures to follow complex prompts

Handling many edge cases in specific ways

Performing a new skill or task that's hard to articulate in a prompt

One high-level way to think about these cases is when it's easier to "show, not tell". In the sections to come, we will explore how to set up data for fine-tuning and various examples where fine-tuning improves the performance over the baseline model.

Another scenario where fine-tuning is effective is in reducing costs and / or latency, by replacing GPT-4 or by utilizing shorter prompts, without sacrificing quality. If you can achieve good results with GPT-4, you can often reach similar quality with a fine-tuned `gpt-3.5-turbo` model by fine-tuning on the GPT-4 completions, possibly with a shortened instruction prompt.

Preparing your dataset

Once you have determined that fine-tuning is the right solution (i.e. you've optimized your prompt as far as it can take you and identified problems that the model still has), you'll need to prepare data for training the model. You should create a diverse set of demonstration conversations that are similar to the conversations you will ask the model to respond to at inference time in production.

Each example in the dataset should be a conversation in the same format as our Chat completions API, specifically a list of messages where each message has a role, content, and optional name. At least some of the training examples should directly target cases where the prompted model is not behaving as desired, and the provided assistant messages in the data should be the ideal responses you want the model to provide.

Example format

In this example, our goal is to create a chatbot that occasionally gives sarcastic responses, these are three training examples (conversations) we could create for a dataset:

```
{"messages": [{"role": "system", "content": "Marv is a factual c     ot
{"messages": [{"role": "system", "content": "Marv is a factual chatbot
{"messages": [{"role": "system", "content": "Marv is a factual chatbot
```

We do not currently support function calling examples but are working to enable this.

The conversational chat format is required to fine-tune `gpt-3.5-turbo`. For `babbage-002` and `davinci-002`, you can follow the prompt completion pair format used for legacy fine-tuning as shown below.

```
{"prompt": "<prompt text>", "completion": "<ideal generated text
{"prompt": "<prompt text>", "completion": "<ideal generated text> }
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
```

Crafting prompts

We generally recommend taking the set of instructions and prompts that you found worked best for the model prior to fine-tuning, and including them in every training example. This should let you reach the best and most general results, especially if you have relatively few (e.g. under a hundred) training examples.

If you would like to shorten the instructions or prompts that are repeated in every example to save costs, keep in mind that the model will likely behave as if those instructions were included, and it may be hard to get the model to ignore those "baked-in" instructions at inference time.

It may take more training examples to arrive at good results, as the model has to learn entirely through demonstration and without guided instructions.

Example count recommendations

To fine-tune a model, you are required to provide at least 10 examples. We typically see clear improvements from fine-tuning on 50 to 100 training examples with `gpt-3.5-turbo` but the right number varies greatly based on the exact use case.

We recommend starting with 50 well-crafted demonstrations and seeing if the model shows signs of improvement after fine-tuning. In some cases that may be sufficient, but even if the model is not yet production quality, clear improvements are a good sign that providing more data will continue to improve the model. No improvement suggests that you may need to rethink how to set up the task for the model or restructure the data before scaling beyond a limited example set.

Train and test splits

After collecting the initial dataset, we recommend splitting it into a training and test portion. When submitting a fine-tuning job with both training and test files, we will provide statistics on both during the course of training. These statistics will be your initial signal of how much the model is improving. Additionally, constructing a test set early on will be useful in making sure you are able to evaluate the model after training, by generating samples on the test set.

Token limits

Each training example is limited to 4096 tokens. Examples longer than this will be truncated to the first 4096 tokens when training. To be sure that your entire training example fits in context, consider checking that the total token counts in the message contents are under 4,000. The maximum number of total tokens trained per job is 50 million tokens (`tokens_in_dataset * n_epochs`).

You can compute token counts using our counting tokens notebook from the OpenAI cookbook.

Estimate costs

Please refer to the pricing page for details on cost per 1k input and output tokens (we do to charge for tokens that are part of the validation data). To estimate the costs for a specific fine-tuning job, use the following formula:

base cost per 1k tokens * number of tokens in the input file * number of epochs trained

For a training file with 100,000 tokens trained over 3 epochs, the expected cost would be ~$2.40 USD.

Check data formatting

Once you have compiled a dataset and before you create a fine-tuning job, it is important to check the data formatting. To do this, we created a simple Python script which you can use to find potential errors, review token counts, and estimate the cost of a fine-tuning job.

Fine-tuning data format validation

Learn about fine-tuning data formatting

Upload a training file

Once you have the data validated, the file needs to be uploaded using the Files API in order to be used with a fine-tuning jobs:

python Copy

```python
import os
import openai
openai.api_key = os.getenv("OPENAI_API_KEY")
openai.File.create(
  file=open("mydata.jsonl", "rb"),
  purpose='fine-tune'
)
```

After you upload the file, it may take some time to process. While the file is processing, you can still create a fine-tuning job but it will not start until the file processing has completed.

Create a fine-tuned model

After ensuring you have the right amount and structure for your dataset, and have uploaded the file, the next step is to create a fine-tuning job.

Start your fine-tuning job using the OpenAI SDK:

python Copy

```python
openai.FineTuningJob.create(training_file="file-abc123", model="gpt-3.5-turbo")
```

In this example, `model` is the name of the model you want to fine-tune (`gpt-3.5-turbo` , `babbage-002` , `davinci-002` , or an existing fine-tuned model) and `training_file` is the file ID that was returned when the training file was uploaded to the OpenAI API. You can customize your fine-tuned model's name using the suffix parameter.

To set additional fine-tuning parameters like the `validation_file` or `hyperparameters` , please refer to the API specification for fine-tuning.

After you've started a fine-tuning job, it may take some time to complete. Your job may be queued behind other jobs in our system, and training a model can take minutes or hours depending on the model and dataset size. After the model training is completed, the user who created the fine-tuning job will receive an email confirmation.

In addition to creating a fine-tuning job, you can also list existing jobs, retrieve the status of a job, or cancel a job.

python Copy

```python
# List 10 fine-tuning jobs
openai.FineTuningJob.list(limit=10)

# Retrieve the state of a fine-tune
openai.FineTuningJob.retrieve("ftjob-abc123")

# Cancel a job
openai.FineTuningJob.cancel("ftjob-abc123")

# List up to 10 events from a fine-tuning job
openai.FineTuningJob.list_events(id="ftjob-abc123", limit=10)

# Delete a fine-tuned model (must be an owner of the org the model was created i
openai.Model.delete("ft:gpt-3.5-turbo:acemeco:suffix:abc123")
```

Use a fine-tuned model

When a job has succeeded, you will see the `fine_tuned_model` field populated with the name of the model when you retrieve the job details. You may now specify this model as a parameter to in the Chat completions (for `gpt-3.5-turbo`) or legacy Completions API (for `babbage-002` and `davinci-002`), and make requests to it using the Playground.

After your job is completed, the model should be available right away for inference use. In some cases, it may take several minutes for your model to become ready to handle requests. If requests to your model time out or the model name cannot be found, it is likely because your model is still being loaded. If this happens, try again in a few minutes.

python Copy

```python
completion = openai.ChatCompletion.create(
  model="ft:gpt-3.5-turbo:my-org:custom_suffix:id",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello!"}
  ]
)
print(completion.choices[0].message)
```

You can start making requests by passing the model name as shown above and in our GPT guide.

Analyzing your fine-tuned model

We provide the following training metrics computed over the course of training: training loss, training token accuracy, test loss, and test token accuracy. These statistics are meant to provide a sanity check that training went smoothly (loss should decrease, token accuracy should increase). While an active fine-tuning jobs is running, you can view an event object which contains some useful metrics:

```
{
    "object": "fine_tuning.job.event",
    "id": "ftevent-abc-123",
    "created_at": 1693582679,
    "level": "info",
    "message": "Step 100/100: training loss=0.00",
    "data": {
        "step": 100,
        "train_loss": 1.805623287509661e-5,
        "train_mean_token_accuracy": 1.0
    },
    "type": "metrics"
}
```

After a fine-tuning job has finished, you can also see metrics around how the training process went by querying a fine-tuning job, extracting a file ID from the `result_files`, and then retrieving that files content. Each results CSV file has the following columns: `step`, `train_loss`, `train_accuracy`, `valid_loss`, and `valid_mean_token_accuracy`.

```
step,train_loss,train_accuracy,valid_loss,valid_mean_token_accur
1,1.52347,0.0,,
2,0.57719,0.0,,
3,3.63525,0.0,,
4,1.72257,0.0,,
5,1.52379,0.0,,
```

While metrics can he helpful, evaluating samples from the fine-tuned model provides the most relevant sense of model quality. We recommend generating samples from both the base model and the fine-tuned model on a test set, and comparing the samples side by side. The test set should ideally include the full distribution of inputs that you might send to the model in a production use case. If manual evaluation is too time-consuming, consider using our Evals library to automate future evaluations.

Iterating on data quality

If the results from a fine-tuning job are not as good as you expected, consider the following ways to adjust the training dataset:

Collect examples to target remaining issues

If the model still isn't good at certain aspects, add training examples that directly show the model how to do these aspects correctly

Scrutinize existing examples for issues

If your model has grammar, logic, or style issues, check if your data has any of the same issues. For instance, if the model now says "I will schedule this meeting for you" (when it shouldn't), see if existing examples teach the model to say it can do new things that it can't do

Consider the balance and diversity of data

If 60% of the assistant responses in the data says "I cannot answer this", but at inference time only 5% of responses should say that, you will likely get an overabundance of refusals

Make sure your training examples contain all of the information needed for the response

If we want the model to compliment a user based on their personal traits and a training example includes assistant compliments for traits not found in the preceding conversation, the model may learn to hallucinate information

Look at the agreement / consistency in the training examples

If multiple people created the training data, it's likely that model performance will be limited by the level of agreement / consistency between people. For instance, in a text extraction task, if people only agreed on 70% of extracted snippets, the model would likely not be able to do better than this

Make sure your all of your training examples are in the same format, as expected for inference

Iterating on data quantity

Once you're satisfied with the quality and distribution of the examples, you can consider scaling up the number of training examples. This tends to help the model learn the task better, especially around possible "edge cases". We expect a similar amount of improvement every time you double the number of training examples. You can loosely estimate the expected quality gain from increasing the training data size by:

- Fine-tuning on your current dataset
- Fine-tuning on half of your current dataset
- Observing the quality gap between the two

In general, if you have to make a trade-off, a smaller amount of high-quality data is generally more effective than a larger amount of low-quality data.

Iterating on hyperparameters

We allow you to specify the number of epochs to fine-tune a model for. We recommend initially training without specifying the number of epochs, allowing us to pick a default for you based on dataset size, then adjusting if you observe the following:

- If the model does not follow the training data as much as expected increase the number by 1 or 2 epochs
  - This is more common for tasks for which there is a single ideal completion (or a small set of ideal completions which are similar). Some examples include classification, entity extraction, or structured parsing. These are often tasks for which you can compute a final accuracy metric against a reference answer.
- If the model becomes less diverse than expected decrease the number by 1 or 2 epochs
  - This is more common for tasks for which there are a wide range of possible good completions

python Copy

```python
openai.FineTuningJob.create(training_file="file-abc123", model="gpt-3.5-turbo", hype
```

Fine-tuning examples

Now that we have explored the basics of the fine-tuning API, let's look at going through the fine-tuning lifecycle for a few different use cases.

Style and tone

Structured output

Migration of legacy models

For users migrating from `/v1/fine-tunes` to the updated `/v1/fine_tuning/jobs` API and newer models, the main difference you can expect is the updated API. The legacy prompt completion pair data format has been retained for the updated `babbage-002` and `davinci-002` models to ensure a smooth transition. The new models will support fine-tuning with 4k token context and have a knowledge cutoff of September 2021.

For most tasks, you should expect to get better performance from `gpt-3.5-turbo` than from the GPT base models.

FAQ

When should I use fine-tuning vs embeddings with retrieval?

Embeddings with retrieval is best suited for cases when you need to have a large database of documents with relevant context and information.

By default OpenAI's models are trained to be helpful generalist assistants. Fine-tuning can be used to make a model which is narrowly focused, and exhibits specific ingrained behavior patterns. Retrieval strategies can be used to make new information available to a model by providing it with relevant context before generating its response. Retrieval strategies are not an alternative to fine-tuning and can in fact be complementary to it.

When can I fine-tune GPT-4 or GPT-3.5-Turbo-16k?

We plan to release support for fine-tuning both of these models later this year.

How do I know if my fine-tuned model is actually better than the base model?

We recommend generating samples from both the base model and the fine-tuned model on a test set of chat conversations, and comparing the samples side by side. For more comprehensive evaluations, consider using the OpenAI evals framework to create an eval specific to your use case.

Can I continue fine-tuning a model that has already been fine-tuned?

Yes, you can pass the name of a fine-tuned model into the `model` parameter when creating a fine-tuning job. This will start a new fine-tuning job using the fine-tuned model as the starting point.

How can I estimate the cost of fine-tuning a model?

Please refer to the estimate cost section above.

Does the new fine-tuning endpoint still work with Weights & Biases for tracking metrics?

No, we do not currently support this integration but are working to enable it in the near future.

How many fine-tuning jobs can I have running at once?

Please refer to our rate limit guide for the most up to date information on the limits.

How do rate limits work on fine-tuned models?

A fine-tuned model pulls from the same shared rate limit as the model it is based off of. For example, if you use half your TPM rate limit in a given time period with the standard `gpt-3.5-turbo` model, any model(s) you fine-tuned from `gpt-3.5-turbo` would only have the remaining half of the TPM rate limit accessible since the capacity is shared across all