Fine-tuning

Learn how to customize a model for your application.

Introduction

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

1. Higher quality results than prompt design
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-3 has been pre-trained on a vast amount of text from the open internet. When given a prompt with just a few examples, it can often intuit what task you are trying to perform and generate a plausible completion. This 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 examples in the prompt anymore.** 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?

Fine-tuning is currently only available for the following base models: `davinci`, `curie`, `babbage`, and `ada`. These are the original models that do not have any instruction following training (like `text-davinci-003` does for example). You are also able to continue fine-tuning a fine-tuned model to add additional data without having to start from scratch.

Installation

We recommend using our OpenAI command-line interface (CLI). To install this, run

```
pip install --upgrade openai
```

(The following instructions work for version **0.9.4** and up. Additionally, the OpenAI CLI requires python 3.)

Set your `OPENAI_API_KEY` environment variable by adding the following line into your shell initialization script (e.g. .bashrc, zshrc, etc.) or running it in the command line before the fine-tuning command:

```
export OPENAI_API_KEY="<OPENAI_API_KEY>"
```

Prepare training data

Training data is how you teach GPT-3 what you'd like it to say.

Your data must be a JSONL document, where each line is a prompt-completion pair corresponding to a training example. You can use our CLI data preparation tool to easily convert your data into this file format.

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

Designing your prompts and completions for fine-tuning is different from designing your prompts for use with our base models (Davinci, Curie, Babbage, Ada). In particular, while prompts for base models often consist of multiple examples ("few-shot learning"), for fine-tuning, each training example generally consists of a single input example and its associated output, without the need to give detailed instructions or include multiple examples in the same prompt.

For more detailed guidance on how to prepare training data for various tasks, please refer to our preparing your dataset best practices.

The more training examples you have, the better. We recommend having at least a couple hundred examples. In general, we've found that each doubling of the dataset size leads to a linear increase in model quality.

CLI data preparation tool

We developed a tool which validates, gives suggestions and reformats your data:

```
openai tools fine_tunes.prepare_data -f <LOCAL_FILE>
```

This tool accepts different formats, with the only requirement that they contain a prompt and a completion column/key. You can pass a **CSV, TSV, XLSX, JSON** or **JSONL** file, and it will save the output into a JSONL file ready for fine-tuning, after guiding you through the process of suggested changes.

Create a fine-tuned model

The following assumes you've already prepared training data following the above instructions.

Start your fine-tuning job using the OpenAI CLI:

```
openai api fine_tunes.create -t <TRAIN_FILE_ID_OR_PATH> -m <BASE_MODE
```

Where `BASE_MODEL` is the name of the base model you're starting from (ada, babbage, curie, or davinci). You can customize your fine-tuned model's name using the suffix parameter.

Running the above command does several things:

1. Uploads the file using the files API (or uses an already-uploaded file)
2. Creates a fine-tune job
3. Streams events until the job is done (this often takes minutes, but can take hours if there are many jobs in the queue or your dataset is large)

Every fine-tuning job starts from a base model, which defaults to curie. The choice of model influences both the performance of the model and the cost of running your fine-tuned model. Your model can be one of: `ada`, `babbage`, `curie`, or `davinci`. Visit our pricing page for details on fine-tune rates.

After you've started a fine-tune job, it may take some time to complete. Your job may be queued behind other jobs on our system, and training our model can take minutes or hours depending on the model and dataset size. If the event stream is interrupted for any reason, you can resume it by running:

```
openai api fine_tunes.follow -i <YOUR_FINE_TUNE_JOB_ID>
```

When the job is done, it should display the name of the fine-tuned model.

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

```
# List all created fine-tunes
openai api fine_tunes.list

# Retrieve the state of a fine-tune. The resulting object includes
# job status (which can be one of pending, running, succeeded, or faile
# and other information
openai api fine_tunes.get -i <YOUR_FINE_TUNE_JOB_ID>

# Cancel a job
openai api fine_tunes.cancel -i <YOUR_FINE_TUNE_JOB_ID>
```

Use a fine-tuned model

When a job has succeeded, the `fine_tuned_model` field will be populated with the name of the model. You may now specify this model as a parameter to our Completions API, and make requests to it using the Playground.

After your job first completes, it may take several minutes for your model to become ready to handle requests. If completion requests to your model time out, it is likely because your model is still being loaded. If this happens, try again in a few minutes.

You can start making requests by passing the model name as the `model` parameter of a completion request:

OpenAI CLI:

```
openai api completions.create -m <FINE_TUNED_MODEL> -p <YOUR_PROMPT>
```

cURL:

```
curl https://api.openai.com/v1/completions \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"prompt": YOUR_PROMPT, "model": FINE_TUNED_MODEL}'
```

Python:

```
import openai
openai.Completion.create(
    model=FINE_TUNED_MODEL,
    prompt=YOUR_PROMPT)
```

Node.js:

```
const response = await openai.createCompletion({
  model: FINE_TUNED_MODEL
  prompt: YOUR_PROMPT,
});
```

You may continue to use all the other Completions parameters like `temperature`, `frequency_penalty`, `presence_penalty`, etc, on these requests to fine-tuned models.

Delete a fine-tuned model

To delete a fine-tuned model, you must be designated an "owner" within your organization.

OpenAI CLI:

```
openai api models.delete -i <FINE_TUNED_MODEL>
```

cURL:

```
curl -X "DELETE" https://api.openai.com/v1/models/<FINE_TUNED_MODEL>
  -H "Authorization: Bearer $OPENAI_API_KEY"
```

Python:

```
import openai
openai.Model.delete(FINE_TUNED_MODEL)
```

Preparing your dataset

Fine-tuning is a powerful technique to create a new model that's specific to your use case. **Before fine-tuning your model, we strongly recommend reading these best practices and specific guidelines for your use case below.**

Data formatting

To fine-tune a model, you'll need a set of training examples that each consist of a single input ("prompt") and its associated output ("completion"). This is notably different from using our base models, where you might input detailed instructions or multiple examples in a single prompt.

- Each prompt should end with a fixed separator to inform the model when the prompt ends and the completion begins. A simple separator which generally works well is `\n\n###\n\n`. The separator should not appear elsewhere in any prompt.
- Each completion should start with a whitespace due to our tokenization, which tokenizes most words with a preceding whitespace.
- Each completion should end with a fixed stop sequence to inform the model when the completion ends. A stop sequence could be `\n`, `###`, or any other token that does not appear in any completion.
- For inference, you should format your prompts in the same way as you did when creating the training dataset, including the same separator. Also specify the same stop sequence to properly truncate the completion.

General best practices

Fine-tuning performs better with more high-quality examples. To fine-tune a model that performs better than using a high-quality prompt with our base models, you should provide at least a few hundred high-quality examples, ideally vetted by human experts. From there, performance tends to linearly increase with every doubling of the number of examples. Increasing the number of examples is usually the best and most reliable way of improving performance.

Classifiers are the easiest models to get started with. For classification problems we suggest using ada, which generally tends to perform only very slightly worse than more capable models once fine-tuned, whilst being significantly faster and cheaper.

If you are fine-tuning on a pre-existing dataset rather than writing prompts from scratch, be sure to manually review your data for offensive or inaccurate content if possible, or review as many random samples of the dataset as possible if it is large.

Specific guidelines

Fine-tuning can solve a variety of problems, and the optimal way to use it may depend on your specific use case. Below, we've listed the most common use cases for fine-tuning and corresponding guidelines.

Classification

In classification problems, each input in the prompt should be classified into one of the predefined classes. For this type of problem, we recommend:

- Use a separator at the end of the prompt, e.g. `\n\n###\n\n`. Remember to also append this separator when you eventually make requests to your model.
- Choose classes that map to a single token. At inference time, specify `max_tokens=1` since you only need the first token for classification.
- Ensure that the prompt + completion doesn't exceed 2048 tokens, including the separator
- Aim for at least ~100 examples per class
- To get class log probabilities you can specify `logprobs=5` (for 5 classes) when using your model

Ensure that the dataset used for finetuning is very similar in structure and type of task as what the model will be used for

Case study: Is the model making untrue statements?

Let's say you'd like to ensure that the text of the ads on your website mention the correct product and company. In other words, you want to ensure the model isn't making things up. You may want to fine-tune a classifier which filters out incorrect ads.

The dataset might look something like the following:

```
{"prompt":"Company: BHFF insurance\nProduct: allround insurance\nAd:O    to
{"prompt":"Company: Loft conversion specialists\nProduct: -\nAd:Straight te
```

In the example above, we used a structured input containing the name of the company, the product, and the associated ad. As a separator we used `\nSupported:` which clearly separated the prompt from the completion. With a sufficient number of examples, the separator doesn't make much of a difference (usually less than 0.4%) as long as it doesn't appear within the prompt or the completion.

For this use case we fine-tuned an ada model since it will be faster and cheaper, and the performance will be comparable to larger models because it is a classification task.

Now we can query our model by making a Completion request.

```
curl https://api.openai.com/v1/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
    "prompt": "Company: Reliable accountants Ltd\nProduct: Personal Tax I
    "max_tokens": 1,
    "model": "YOUR_FINE_TUNED_MODEL_NAME"
  }'
```

Which will return either `yes` or `no`.

Case study: Sentiment analysis

Let's say you'd like to get a degree to which a particular tweet is positive or negative. The dataset might look something like the following:

```
{"prompt":"Overjoyed with the new iPhone! ->", "completion":" positiv
{"prompt":"@lakers disappoint for a third straight night https://t.co/38EFe
```

Once the model is fine-tuned, you can get back the log probabilities for the first completion token by setting `logprobs=2` on the completion request. The higher the probability for positive class, the higher the relative sentiment.

Now we can query our model by making a Completion request.

```
curl https://api.openai.com/v1/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
    "prompt": "https://t.co/f93xEd2 Excited to share my latest blog post
    "max_tokens": 1,
    "model": "YOUR_FINE_TUNED_MODEL_NAME"
  }'
```

Which will return:

```
{
  "id": "cmpl-COMPLETION_ID",
  "object": "text_completion",
  "created": 1589498378,
  "model": "YOUR_FINE_TUNED_MODEL_NAME",
  "choices": [
    {
      "logprobs": {
        "text_offset": [
          19
        ],
        "token_logprobs": [
          -0.03597255
        ],
        "tokens": [
          " positive"
        ],
        "top_logprobs": [
          {
            " negative": -4.9785037,
            " positive": -0.03597255
          }
        ]
      },

      "text": " positive",
      "index": 0,
      "finish_reason": "length"
    }
  ]
}
```

Case study: Categorization for Email triage

Let's say you'd like to categorize incoming email into one of a large number of predefined categories. For classification into a large number of categories, we recommend you convert those categories into numbers, which will work well up to ~500 categories. We've observed that adding a space before the number sometimes slightly helps the performance, due to tokenization. You may want to structure your training data as follows:

```
{"prompt":"Subject: <email_subject>\nFrom:<customer_name>\nDate:<date    on
```

For example:

```
{"prompt":"Subject: Update my address\nFrom:Joe Doe\nTo:support@ourco    y.
```

In the example above we used an incoming email capped at 2043 tokens as input. (This allows for a 4 token separator and a one token completion, summing up to 2048.) As a separator we used `\n\n###\n\n` and we removed any occurrence of `###` within the email.

Conditional generation

Conditional generation is a problem where the content needs to be generated given some kind of input. This includes paraphrasing, summarizing, entity extraction, product description writing given specifications, chatbots and many others. For this type of problem we recommend:

- Use a separator at the end of the prompt, e.g. `\n\n###\n\n`. Remember to also append this separator when you eventually make requests to your model.
- Use an ending token at the end of the completion, e.g. `END`
- Remember to add the ending token as a stop sequence during inference, e.g. `stop=[" END"]`
- Aim for at least ~500 examples
- Ensure that the prompt + completion doesn't exceed 2048 tokens, including the separator
- Ensure the examples are of high quality and follow the same desired format
- Ensure that the dataset used for finetuning is very similar in structure and type of task as what the model will be used for
- Using Lower learning rate and only 1-2 epochs tends to work better for these use cases

Case study: Write an engaging ad based on a Wikipedia article

This is a generative use case so you would want to ensure that the samples you provide are of the highest quality, as the fine-tuned model will try to imitate the style (and mistakes) of the given examples. A good starting point is around 500 examples. A sample dataset might look like this:

```
{"prompt":"<Product Name>\n<Wikipedia description>\n\n###\n\n", "comp    on
```

For example:

```
{"prompt":"Samsung Galaxy Feel\nThe Samsung Galaxy Feel is an Android    rt
```

Here we used a multi line separator, as Wikipedia articles contain multiple paragraphs and headings. We also used a simple end token, to ensure that the model knows when the completion should finish.

Case study: Entity extraction

This is similar to a language transformation task. To improve the performance, it is best to either sort different extracted entities alphabetically or in the same order as they appear in the original text. This will help the model to keep track of all the entities which need to be generated in order. The dataset could look as follows:

```
{"prompt":"<any text, for example news article>\n\n###\n\n", "complet    :"
```

For example:

```
{"prompt":"Portugal will be removed from the UK's green travel list f    Tu
```

A multi-line separator works best, as the text will likely contain multiple lines. Ideally there will be a high diversity of the types of input prompts (news articles, Wikipedia pages, tweets, legal documents), which reflect the likely texts which will be encountered when extracting entities.

Case study: Customer support chatbot

A chatbot will normally contain relevant context about the conversation (order details), summary of the conversation so far as well as most recent messages. For this use case the same past conversation can generate multiple rows in the dataset, each time with a slightly different context, for every agent generation as a completion. This use case will require a few thousand examples, as it will likely deal with different types of requests, and customer issues. To ensure the performance is of high quality we recommend vetting the conversation samples to ensure the quality of agent messages. The summary can be generated with a separate text transformation fine tuned model. The dataset could look as follows:

```
{"prompt":"Summary: <summary of the interaction so far>\n\nSpecific i    ma
{"prompt":"Summary: <summary of the interaction so far>\n\nSpecific informa
```

Here we purposefully separated different types of input information, but maintained Customer Agent dialog in the same format between a prompt and a completion. All the completions should only be by the agent, and we can use `\n` as a stop sequence when doing inference.

Case study: Product description based on a technical list of properties

Here it is important to convert the input data into a natural language, which will likely lead to superior performance. For example, the following format:

```
{"prompt":"Item=handbag, Color=army_green, price=$99, size=S->", "com    io
```

Won't work as well as:

```
{"prompt":"Item is a handbag. Colour is army green. Price is midrange    ze
```

For high performance ensure that the completions were based on the description provided. If external content is often consulted, then adding such content in an automated way would improve the performance. If the description is based on images, it may help to use an algorithm to extract a textual description of the image. Since completions are only one sentence long, we can use `.` as the stop sequence during inference.

Advanced usage

Customize your model name

You can add a suffix of up to 40 characters to your fine-tuned model name using the suffix parameter.

OpenAI CLI:

```
openai api fine_tunes.create -t test.jsonl -m ada --suffix "custom mo    na
```

The resulting name would be:

```
ada:ft-your-org:custom-model-name-2022-02-15-04-21-04
```

Analyzing your fine-tuned model

We attach a result file to each job once it has been completed. This results file ID will be listed when you retrieve a fine-tune, and also when you look at the events on a fine-tune. You can download these files:

OpenAI CLI:

```
openai api fine_tunes.results -i <YOUR_FINE_TUNE_JOB_ID>
```

CURL:

```
curl https://api.openai.com/v1/files/$RESULTS_FILE_ID/content \
  -H "Authorization: Bearer $OPENAI_API_KEY" > results.csv
```

The `_results.csv` file contains a row for each training step, where a step refers to one forward and backward pass on a batch of data. In addition to the step number, each row contains the following fields corresponding to that step:

- **elapsed_tokens**: the number of tokens the model has seen so far (including repeats)
- **elapsed_examples**: the number of examples the model has seen so far (including repeats), where one example is one element in your batch. For example, if `batch_size = 4`, each step will increase `elapsed_examples` by 4.
- **training_loss**: loss on the training batch
- **training_sequence_accuracy**: the percentage of **completions** in the training batch for which the model's predicted tokens matched the true completion tokens exactly. For example, with a `batch_size` of 3, if your data contains the completions [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 2/3 = 0.67
- **training_token_accuracy**: the percentage of **tokens** in the training batch that were correctly predicted by the model. For example, with a `batch_size` of 3, if your data contains the completions [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 5/6 = 0.83

Classification specific metrics

We also provide the option of generating additional classification-specific metrics in the results file, such as accuracy and weighted F1 score. These metrics are periodically calculated against the full validation set and at the end of fine-tuning. You will see them as additional columns in your results file.

To enable this, set the parameter `--compute_classification_metrics`. Additionally, you must provide a validation file, and set either the `classification_n_classes` parameter, for multiclass classification, or `classification_positive_class`, for binary classification.

OpenAI CLI:

```
# For multiclass classification
openai api fine_tunes.create \
  -t <TRAIN_FILE_ID_OR_PATH> \
  -v <VALIDATION_FILE_OR_PATH> \
  -m <MODEL> \
  --compute_classification_metrics \
  --classification_n_classes <N_CLASSES>

# For binary classification
openai api fine_tunes.create \
  -t <TRAIN_FILE_ID_OR_PATH> \
  -v <VALIDATION_FILE_OR_PATH> \
  -m <MODEL> \
  --compute_classification_metrics \
  --classification_n_classes 2 \
  --classification_positive_class <POSITIVE_CLASS_FROM_DATASET>
```

The following metrics will be displayed in your results file if you set `--compute_classification_metrics`:

For multiclass classification

classification/accuracy: accuracy

classification/weighted_f1_score: weighted F-1 score

For binary classification

The following metrics are based on a classification threshold of 0.5 (i.e. when the probability is > 0.5, an example is classified as belonging to the positive class.)

classification/accuracy

classification/precision

classification/recall

classification/f{beta}

- **classification/auroc** - AUROC
- **classification/auprc** - AUPRC

Note that these evaluations assume that you are using text labels for classes that tokenize down to a single token, as described above. If these conditions do not hold, the numbers you get will likely be wrong.

Validation

You can reserve some of your data for validation. A validation file has exactly the same format as a train file, and your train and validation data should be mutually exclusive.

If you include a validation file when creating your fine-tune job, the generated results file will include evaluations on how well the fine-tuned model performs against your validation data at periodic intervals during training.

OpenAI CLI:

```
openai api fine_tunes.create -t <TRAIN_FILE_ID_OR_PATH> \
  -v <VALIDATION_FILE_ID_OR_PATH> \
  -m <MODEL>
```

If you provided a validation file, we periodically calculate metrics on batches of validation data during training time. You will see the following additional metrics in your results file:

- **validation_loss**: loss on the validation batch
- **validation_sequence_accuracy**: the percentage of completions in the validation batch for which the model's predicted tokens matched the true completion tokens exactly. For example, with a `batch_size` of 3, if your data contains the completion [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 2/3 = 0.67
- **validation_token_accuracy**: the percentage of tokens in the validation batch that were correctly predicted by the model. For example, with a `batch_size` of 3, if your data contains the completion [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 5/6 = 0.83

Hyperparameters

We've picked default hyperparameters that work well across a range of use cases. The only required parameter is the training file.

That said, tweaking the hyperparameters used for fine-tuning can often lead to a model that produces higher quality output. In particular, you may want to configure the following:

- `model` : The name of the base model to fine-tune. You can select one of "ada", "babbage", "curie", or "davinci". To learn more about these models, see the Models documentation.
- `n_epochs` - defaults to 4. The number of epochs to train the model for. An epoch refers to one full cycle through the training dataset.
- `batch_size` - defaults to ~0.2% of the number of examples in the training set, capped at 256. The batch size is the number of training examples used to train a single forward and backward pass. In general, we've found that larger batch sizes tend to work better for larger datasets.
- `learning_rate_multiplier` - defaults to 0.05, 0.1, or 0.2 depending on final `batch_size` . The fine-tuning learning rate is the original learning rate used for pretraining multiplied by this multiplier. We recommend experimenting with values in the range 0.02 to 0.2 to see what produces the best results. Empirically, we've found that larger learning rates often perform better with larger batch sizes.
- `compute_classification_metrics` - defaults to False. If True, for fine-tuning for classification tasks, computes classification-specific metrics (accuracy, F-1 score, etc) on the validation set at the end of every epoch.

To configure these additional hyperparameters, pass them in via command line flags on the OpenAI CLI, for example:

```
openai api fine_tunes.create \
  -t file-JD89ePi5KMsB3Tayeli5ovfW \
  -m ada \
  --n_epochs 1
```

Continue fine-tuning from a fine-tuned model

If you have already fine-tuned a model for your task and now have additional training data that you would like to incorporate, you can continue fine-tuning from the model. This creates a model that has learned from all of the training data without having to re-train from scratch.

To do this, pass in the fine-tuned model name when creating a new fine-tuning job (e.g. `-m curie:ft-<org>-<date>`). Other training parameters do not have to be changed, however if your new training data is much smaller than your previous training data, you may find it useful to reduce `learning_rate_multiplier` by a factor of 2 to 4.

Weights & Biases

You can sync your fine-tunes with Weights & Biases to track experiments, models, and datasets.

To get started, you will need a Weights & Biases account and a paid OpenAI plan. To make sure you are using the lastest version of `openai` and `wandb`, run:

```
pip install --upgrade openai wandb
```

To sync your fine-tunes with Weights & Biases, run:

```
openai wandb sync
```

You can read the Weights & Biases documentation for more information on this integration.

Example notebooks

Classification

Question answering