Update: we recommend this more comprehensive guide:

[PUBLIC] Best practices for fine-tuning GPT-3 t…

Best practices for classifying text with fine-tuned models

Ted Sanders
May 2022

Public resources
Fine tuning documentation
Code example

How fine-tuning works
You can improve GPT-3's performance on specific tasks by fine-tuning it on training data. During fine-tuning, the model reads text tokens from the training data, and after each token, it predicts which token is next. When it gets the prediction wrong, the model's internal weights are updated to make it more likely to predict correctly the next time. After enough training, the model will learn to produce the pattern of tokens demonstrated in your training data.

Note: in fine tuning, the model learns both from the prompt and the completion. The distinction between them is somewhat arbitrary except for the fact the prompt loss parameter (default 10%) tells the model to learn less from the prompt tokens and more from the completion tokens.

Prompt formatting

- Fine-tuned models do not need instructions or examples, as they learn the task from the examples they train on. So no need to include instructions like "classify the following text as x or y."
    - Instructions can still be useful when fine-tuning a model to do multiple tasks. For example, if you train a model to extract either names or phone numbers, you'll need some type of instruction to tell the model when you want a name extracted vs when you want a phone number extracted.
- For classification, it's helpful to include sequences that tell the model when the text is done and when the labeling should begin. This should be done by inserting a separator sequence, for example "\n\n###\n\n" at the end of your prompt. Without those separator sequences, the model may continue writing additional text rather than classifying it.

- The choice of separator sequence doesn't significantly impact performance. Just be sure that the sequence is unlikely to otherwise appear in your text so that (e.g., you might want to avoid '###' or '->' when classifying Python code). Feel free to use examples like:
    - <prompt>\n\n###\n\n
    - <prompt> >>>
    - <prompt>/n/n->
- Note that the prompt + completion must always be less than 2048 tokens; fine-tuned models (as of May 2022) cannot process more than this.

Label formatting

- Use single-token labels if you can
    - Single-token labels are convenient because each token comes with a probability (probabilities for sequences are less convenient to extract)
    - Single-token labels minimize cost
- The choice of label token may have a slight impact on results when you have less data (e.g., ' yes' is a token with a high base likelihood), but you can also use labels like ' 1', ' 2', ' 3' and the model will still be able to learn the associations in your training data. For reference, all numbers <500 are single token

Example data format

Prompt	**Completion**
{text to be classified} ###	{label}
{text to be classified} ###	{label}

Improving performance

There are three levers for improving fine-tuning performance: data, model size, and parameters

(1) Data is the the largest lever for improving performance

- The best data are high quality, diverse, and high volume
- Data quality
    - The best data is similar to what you'll use the model for. If you train on only tangentially related data or data in a different format, performance may be poor.

    - The data should be formatted in a clear way, as per instructions above
- Data volume
    - As a rule of thumb, you should have at least ~100 examples to fine-tune on
    - There are diminishing returns to adding more data - our typical rule of thumb is that each doubling in data gets you linear performance increases, but this really depends on the task and the training data. More complicated tasks need more data to learn well. Bigger models tend to need less data.

(2) Model size (ada, babbage, curie, davinci)

- Larger models have higher performance; smaller models are cheaper and have lower latency
- For simple classification tasks (e.g., sentiment), ada is often the most cost-effective model size, as the performance gaps tend to be small (e.g., 90% vs 91% vs 92%).
- For more complex tasks, larger models may be best.

(3) Parameters are the third lever for improving performance

- Fine tuning parameters include:
    - n_epochs (the number of times each example is learned from, defaulting to 4),
    - learning rate,
    - batch size,
    - prompt loss (which controls how much it learns from the prompts vs the completions)
- Typically performance is not wildly different with different settings, but you can do your own experiments to see what combination of parameters works best for you
- Learning rate & batch size
    - Learning rate multipliers in the range of 0.025 to 0.5 all can work reasonably well. We have selected a default of 0.1. Learning rate multipliers outside of this range tend to perform worse.
    - A reasonably wide range of batch sizes works well - in our testing, anything from 0.01% of the training dataset size to 2% of the training dataset size performed decently. We did not test smaller batch sizes than this. We did test larger batch sizes than this (5% of the training dataset & 10% of the training dataset), but they tended to perform poorly
    - Larger learning rate multipliers tend to perform better with larger batch sizes. Smaller learning rate multipliers tend to perform better with smaller batch sizes.
- Prompt_loss_weight
    - Prompt loss weight noticeably affects classification performance. We have found 0.1 to be a good default value
    - For classification of long pieces of text, it may be possible to get better performance by setting prompt_loss_weight to smaller values. Prompt_loss_weight is 10% by default, which tells the model to learn from prompt tokens at 10% the rate of completion tokens. For classification tasks with long pieces of text and short labels, a smaller prompt_loss_weight will ensure that the model doesn't spend most of its learning on the prompts, and dedicates its learning to the labels as well.

    - In experiments, we've seen that 0 is usually suboptimal. Our hypothesis is that continuing to learn the prompts can help retain the relevant general language skills that might atrophy when learning the labels only. It usually only makes a small difference and varies depending on the task.
- N_epochs
    - In our testing, 10 and 4 epochs performed about the same
    - In general, with more data you may need fewer epochs

- Beyond the parameters for fine-tuning your model, there are also parameters used in Completions when running your fine-tuned model
    - Temperature
    - Logprobs
    - Logit_bias
    - Echo
- Temperature
    - For deterministic non-creative tasks like classification, always set temperature to 0 (positive temperatures cause the models to partially randomize the labels it generates, which is bad for classification)
- Logprobs
    - Logprobs will return the log probabilities of the top 5 tokens (though this can be increased for trusted customers with a good business case)
    - These probabilities are useful for calculating things like precision-recall curves, or confidence estimates
- Logit_bias
    - Logit_bias increases or decreases the probabilities of specified tokens
    - Setting logit_bias to 100 for each of your classification labels will ensure that the model only outputs valid labels (note that this can potentially hide problems, as outputting invalid labels can signal that something was set up incorrectly, such as poor formatting or too little data)
- Echo
    - Setting echo=True will returns the prompt & completion instead of just the completion
    - If you are debugging/testing, this parameter can be useful because it allows you to guarantee that you'll get the probability of a particular label; to do so:
        - Submit the prompt appended with the label you want (and set max_tokens=0 and logprobs=1),
        - In the response, you can extract the probability of the supplied label
    - Getting the probabilities of labels 1-by-1 is more expensive, so this is not common

Adjusting results

- If you see statistical bias in the outputs of your fine-tuned model, you can further improve performance by multiplying its output probabilities by a debiasing factor.