CS 330 2023 Homework 3

Few-Shot Learning with Pre-trained Language Models

Due Monday November 6th, 11:59 PM PT

SUNet ID:
Name:
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Overview

This assignment will explore several methods for performing few-shot (and zero-shot) learning with pre-trained language models (LMs), including variants of fine-tuning and in-context learning. The goal of this assignment is to gain familiarity with performing few-shot learning with pre-trained LMs, learn about the relative strengths and weaknesses of fine-tuning and in-context learning, and explore some recent methods proposed for improving on the basic form of these algorithms.

We have provided you with starter code, which can be downloaded from the course website.

Submission: To submit your homework, submit one PDF report to Gradescope containing written answers and Matplotlib plots (screenshots are acceptable) to the questions below, as well as a single zip file containing:

- `__init__.py`
- `ft.py`
- `icl.py`
- `results`

Note: the autograder will only work if all these files are contained in the root of the zip! I.e. not inside a subdirectory.

The PDF should also include your name and any students you talked to or collaborated with. **Any written responses or plots to the questions below must appear in your PDF submission.**

Setup: This assignment requires using the HuggingFace library for loading pre-trained models and datasets. The required packages are provided in a `requirements.txt` file to

enable easy installation, which you can install using `pip install -r requirements.txt`. If you have trouble installing on your laptop (e.g. some Apple Silicon machines), you can use your Azure instance to debug. If you're having trouble installing the requirements, you might need a specific CUDA version; checking the PyTorch install guide might help find the right version.

Code Overview: The code consists of several files to enable fine-tuning and in-context learning. You are expected to write the code in the following files:

- `ft.py`: Fine-tuning script; you'll implement parameter selection, loss & accuracy calculation, the LoRA implementation (this will be explained in Q2!), fine-tuning batch tokenization, and the model update step.
- `icl.py`: In-context learning script; you'll implement prompt assembly and model sampling.

You only need to change these two files. A detailed description for every function can be found in the comments. You are not expected to change any code except for sections marked with `YOUR CODE HERE`.

Please complete questions in order, since they depend on previous components.

Compute

For this assignment, you will complete all sections on Azure instances with GPUs. Avoid using Azure for debugging; verify that your code runs without crashing on your own machine, and use Azure only once you think your code is finished.

The CS330 Azure Guide has instructions for setting up and accessing an Azure GPU spot instance.

Be sure to shut down your Azure instance when not in use!

Datasets

You'll explore three different language datasets in this assignment. The first, Amazon Reviews, is a classification dataset that you'll use for the warmup. The other two, XSum and bAbI, require generating open-ended language.

1. For the fine-tuning warmup, we'll explore a simple text classification problem, a subset of the Amazon Reviews dataset. The portion of the Amazon Reviews dataset

that we will consider contains paired video reviews and star ratings; the task we will consider will be five-way classification of a review into the number of stars its corresponding rating gave, among {1,2,3,4,5}. **You can expect accuracy numbers *roughly* in the 0.2-0.4 range for Amazon Reviews in this assignment.**

2. XSum: The XSum dataset contains news articles from the BBC and corresponding one sentence summaries. The evaluation metric typically used for summarization is **ROUGE score**, which measures n-gram overlap between the target and prediction (how many words appear in both, how many bigrams, etc.). An n-gram is a subsequence of n consecutive words, in this context. **You can expect ROUGE scores *roughly* in the 0.1-0.3 range for XSum in this assignment.**

3. bAbI is a AI benchmark suite developed by Facebook AI Research. The task that we will use is a question-answering task that requires reasoning about contextual information that may or may not be relevant to answer a question. A sample question from the dataset is:

   ```
   Mary went back to the office.  John went to the bathroom.  Where is Mary?
   In the office
   ```

   You can expect accuracy numbers *roughly* in the 0.25-0.9 range for bAbI QA in this assignment.

These datasets, except for the Amazon Reviews dataset are available through the HuggingFace library.

Optional. Autograding Your Code. In this homework, we include autograding functionalities in the released code to facilitate you to debug and develop your code. To run the autograder, simply do:

```
python grader.py
```

The maximum points you can get when running the autograder is **14 / 14 points**. We also have **36 points** from hidden test cases that show up when you submit your code to Gradescope. This makes the total of **50 points** for all the coding parts.

(10 pt) Question 0: Fine-tuning warmup

As a warmup, we will perform perhaps the most straightforward form of k-shot learning with a pre-trained model: directly fine-tuning the entire model on the k examples. We will use two different sizes of smaller BERT models that have been compressed through distillation.[1]

[1]See here to learn more about this class of distilled BERT models. For final projects involving language models, these smaller BERT models may be useful for performing compute-friendly experiments!

1. Implement the logic for fine-tuning, including selecting the parameters that will be fine-tuned (only the case for `'all'` for this question) in `ft.py:parameters_to_fine_tune()`, and computing the loss and accuracy in `ft.py:get_loss` and `ft.py:get_acc`. You only need to complete the loss/accuracy calculations under `if logits.dim() == 2:` for this question.

2. The Amazon Reviews dataset is no longer available through HuggingFace, so we are providing an adapted version of the Kaggle dataset to download here. Please place the *uncompressed* csv file inside the `data` directory. You can run this command to send the file to your Azure instance:

   ```
   scp  /Downloads/amazon_reviews_us_Video_v1_00.csv azureuser@instance_public_ip_address:~/
   ```

3. (8 pt) Run the command:

   ```
   python ft.py --task ft --model bert-tiny,bert-med --dataset amazon --k 1,8,128
   ```

   to fine-tune two sizes of BERT models on the Amazon Reviews dataset for various values of k. While debugging, you can pass only one value for each of the arguments to run only that subset, e.g. `python ft.py --task ft --model bert-tiny --dataset amazon --k 1`.

   If you see a log message like `Some weights of the model checkpoint...`, this is expected, since the pre-trained model does not contain a prediction head for our task (this is why we need to fine-tune!).

   To plot your results, run the command:

   ```
   python ft.py --task plot --model bert-tiny,bert-med --dataset amazon --k 1,8,128
   --plot_name Q0_3.png
   ```

   In one sentence, what do you notice about the performance of the two model scales?

   Solution: The results can be seen in figure 1. The smaller BERT-tiny model shows steady but slower improvement with more examples, while the medium-sized model initially performs better but dips before showing steeper improvement after 8 examples, ultimately achieving higher accuracy with 128 examples.

4. (1 pt) If we fine-tune the all of our model parameters for each task, we must save a new complete copy of the model's parameters for each new task. As an example, a BERT-mini model similar to the ones you just fine-tuned has approximately 11.2 million parameters; assuming parameters are represented as 4-byte floats, after fine-tuning on a new task, how much disk space do we need to store the new fine-tuned model parameters?

   Solution: $11\_200\_000 \times 4 = 44\_800\_000$ bytes

   As a result, we need 44.8MB of disk space to store the new fine-tuned model parameters.

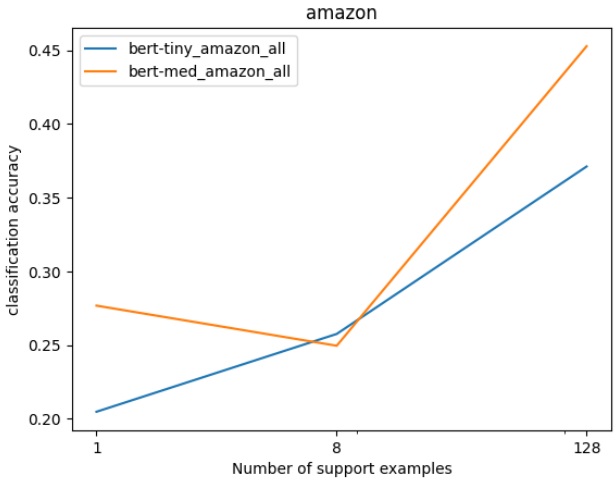

Figure 1: The accuracy of bert-tiny and bert-mid on amazon for various values of k.

5. (1 pt) Google's recent large language model PaLM has 540 billion parameters. How much disk space would be needed to store a new fine-tuned version of this model, assuming parameters are represented as 4-byte floats?

   Solution: $540\_000\_000\_000 \times 4 = 2\_160\_000\_000\_000$ bytes

   As a result, we need 2.16TB of disk space to store the fine-tuned version of PaLM.

(15 pt) Question 1: In-context learning

A surprising property of large language models is their emergent ability to learn *in-context*, that is, their ability to learn a task without updating any parameters at all. The name 'in-context' comes from the fact that the learning is done by simply including several examples or a task description (or both!) prepended to a test input present to the model

For example, for a question-answering task, to make a 2-shot prediction for a test input 'Why is the sky blue?', rather than presenting the input:

`Why is the sky blue?<generate>`

we would simply prepend our 2 examples to the input:

`Who is the US president? Joe Biden What is earth's tallest mountain? Mount... Everest Why is the sky blue?<generate>`

In addition to few-shot in-context learning, models can often improve their zero-shot generalization if the input is formatted in a particular manner; for example, adding a `Q:` and `A:` prefix to the input and label, respectively:

`Q: Why is the sky blue? A:<generate>`

Finally, these two approaches can be combined, for example adding the `Q:` and `A:` markers to each example in the context as well as the test input.

1. Complete the code for in-context learning in `icl.py`.

   (a) Implement prompt creation for the XSum and bAbI tasks, which is found in `icl.py:get_icl_prompts()`. Shuffle the order of the support inputs and targets, but preserving the pairings of input/labels. You will implement 4 prompt format modes:

      i. `qa` **[only for bAbI]**: Add " `In the` " after the question (including the final test question that we want to generate an answer for!) and before each answer, since this task involves answering questions about the physical whereabouts of a person. In addition, add a period after the answer (omitting the period can significantly impact your results!). Be sure to include a space between the question and `In the`, as well as a space before the answer (though keep in mind Note 1!). You do not need to add the "Q: A:" from the earlier example.

      ii. `none` **[only for XSum]**: In this case, we use the raw k examples without any additional formatting; that is, we just concatenate $[x_1; y_1; ...; x_k; y_k; x^*]$ with a space between each element (but no space at the end), where x^* is the input that we want to generate an answer for.

      iii. `tldr` **[only for XSum]**: Add the text " `TL;DR:` " after each article/input (including the final test article) and before the summary/target. Also add a space between each summary and the next article (but no space at the end).

      iv. `custom` **[only for XSum]**: Come up with your own prompt format for article summarization (different from the ones we've shown!).

   In general, the idea of in-context learning is to format the support examples in the same way as the test example, to leverage the model's tendency toward imitation.

   Examples for the first 3 prompt types are available in the starter code.

   Note 1: Due to a quirk with GPT-2 tokenization, you should not include a space at the end of your prompt before generation.

   Note 2: Be sure to shuffle the order of the support inputs/targets when you construct the prompt (we will need this randomization later).

   (b) Implement greedy sampling in `icl.py:do_sample()`. The GPT-2 models used in this and the following questions use an autoregressive factorization of the probability of a sequence, i.e. $p_\theta(\mathbf{x}) = \prod_t p_\theta(x_t|x_{<t})$. 'Greedy' sampling means that given a context $x_{<t}$ producing a distribution over next tokens $p_\theta(x_t|x_{<t})$, we deterministically choose the next token x_t to be the token with highest probability. The documentation for the GPT-2 model used (transformers.GPT2LMHeadModel) will be helpful. You should use caching to speed up generation and decrease memory usage.

Tips to avoid out-of-memory errors: Wrap calls to the model with `torch.inference_mode()` (see example here). Also see the `past_key_values` and `use_cache` arguments for the `GPT2LMHeadModel` model.

Note 3: Be sure you understand what each dimension of the model's output `logits` represents. Misinterpreting the dimensions of this output can lead to subtle bugs.

(c) Finally, put the pieces together by completing the implementation of `icl.py`: `run_icl()`, using your `get_icl_prompts()` and `do_sample()` functions, as well as the HuggingFace tokenizer defined in the loop.
Hint: Your solution here should be 5-10 lines of code.

2. (10 pt) First, evaluate k-shot in-context performance on bAbI for GPT-2-medium (355M parameters) and full-size GPT-2 (1.5B parameters) for various values of k with the command:

```
python icl.py --task icl --model med,full --dataset babi --k 0,1,16
```

Plot the results with the command:

```
python icl.py --task plot --model med,full --dataset babi --k 0,1,16 --plot_name
Q1_2.png
```

What relationship(s) do you notice between model scale and few-shot performance?

Solution: The results can be seen in figure 2. It can be seen that:

(a) The full-size GPT-2 (1.5B parameters) shows better zero-shot performance (k=0) compared to the medium model, starting at around 0.45 vs 0.3 exact match accuracy.

(b) Both models show dramatic improvement with just one example (k=1), converging to similar performance around 0.72 exact match accuracy.

(c) After k=1, both models maintain relatively stable performance up to k=16, with only slight degradation.

(d) The performance gap between model scales is most pronounced in the zero-shot setting but becomes negligible once even a single example is provided.

3. (5 pt) Now let's evaluate several different prompt formats on the XSum dataset. With and without a task description in the prompt, evaluate zero-shot and few-shot performance for XSum on GPT-2-Medium with the command:

```
python icl.py --task icl --model med,full --dataset xsum --k 0,1,4 \
    --prompt none,tldr,custom
```

Note that we use much smaller k than in the previous problem, because we must fit all k examples into the model's context window, which is only 1024 tokens. The fixed context window length is one limitation of in-context learning.

The $k = 4$ XSum evaluation on full-size GPT-2 may take approximately 40 minutes on your Azure instance for each prompt mode; this is expected, and is another

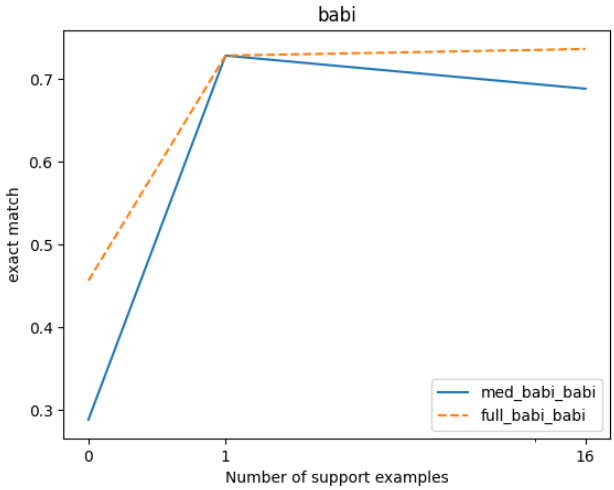

Figure 2: The k-shot in-context performance on bAbI for GPT-2-medium (355M parameters) and full-size GPT-2 (1.5B parameters).

downside of in-context learning (we need to process a much longer input, containing the prompt, compared to a fine-tuned model that just processes the test input).

Plot the zero-shot and few-shot performance of GPT-2 on XSum:

```
python icl.py --task plot --model med,full --dataset xsum --k 0,1,4 \
    --prompt none,tldr,custom --plot_name Q1_3.png
```

How does the performance of the `TL;DR:` prompt compare with no prompt formatting? What was your custom prompt format, and how did it compare with `TL;DR:`? Discuss the relative performance of the different prompts in the zero-shot, one-shot, and few-shot settings.

Solution: The results can be seen in figure 3. Our custom format uses the same format with `TL;DR:` but with a different format of `Summary:`.

It can be seen that:

(a) TL;DR vs No Prompt:

- For the medium model, TL;DR consistently performs better than no prompt formatting across all k values.
- For the full model, TL;DR also outperforms no prompt formatting.
- The performance gap between TL;DR and no prompt is more pronounced in the full model compared to the medium model.

(b) Custom vs TL;DR:

- The custom prompt format generally performs worse than TL;DR for both model sizes.
- However, the custom prompt shows more improvement with increasing examples compared to TL;DR, especially in the full model.

(c) Performance across different shot settings:

- Zero-shot (k=0): TL;DR is clearly superior to both no prompt and custom formats.
- One-shot (k=1): All formats show some improvement, but TL;DR maintains its advantage.
- Few-shot (k=4): The performance gaps between different prompt formats begin to narrow, particularly for the full model.

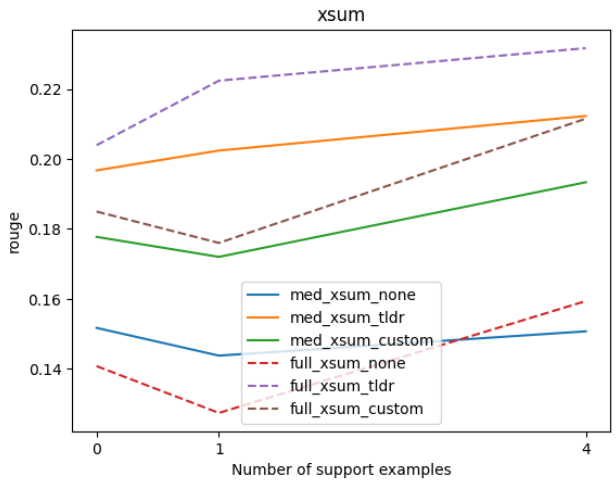

Figure 3: The k-shot performance for GPT-2 medium and full-size using different prompt formats on the XSum dataset.

(15 pt) Question 2: Parameter-efficient fine-tuning

As we observed in question 1, fine-tuning the entire model can become extremely costly for very large models. In this question, we'll explore more methods for *parameter-efficient fine-tuning*, i.e., methods that enable fine-tuning while creating fewer new parameters and are less prone to overfitting.

1. Finish the implementation for each version of parameter-efficient fine-tuning for GPT-2-Medium in `ft.py:parameters_to_fine_tune()`:

   (a) `last`: Fine-tune only the last 2 transformer blocks

   (b) `first`: Fine-tune only first 2 transformer blocks

   (c) `middle`: Fine-tune only middle 2 transformer blocks

   This step simply requires selecting the correct subset of parameters for each version listed above in `parameters_to_fine_tune()`. Keep in mind you should be returning an iterable of `nn.Parameter` here, not `nn.Module`.

Hint: to understand how to get parameters for specific layers, we encourage you to print the model architecture. You can place a Python debugging breakpoint anywhere in the code with '`import pdb; pdb.set_trace()`', and can run the command in subpart 3 to get the function to run with the right parameters.

2. In addition to selecting only a subset of layers to fine-tune, more sophisticated methods for parameter-efficient fine-tuning exist; one such method is LoRA: Low-rank adaptation. For each layer ℓ in the network, rather than fine-tune the pre-trained weight matrix $W_\ell^0 \in \mathbb{R}^{d_1 \times d_2}$ into an arbitrary new weight matrix W_ℓ^{ft}, LoRA constrains the space of fine-tuned parameters such that $W_\ell^{ft} = W_\ell^0 + AB^\top$, where $A \in \mathbb{R}^{d_1 \times p}$ and $B \in \mathbb{R}^{d_2 \times p}$, and $p << d_1, d_2$. That is, we force the *difference* between W_ℓ^0 and W_ℓ^{ft} to be rank p, keeping W_ℓ^0 frozen and only fine-tuning the rank-p residual matrix $AB^\top$. We will apply this form of fine-tuning to both the MLP weight matrices and the self-attention weight matrices in the model.

   (a) For a single layer, what are the parameter savings we achieve by using LoRA? i.e., what is the ratio of parameters fine-tuned by LoRA (for arbitrary p) to the number of parameters in W_ℓ^0? In terms of p, d_1, d_2, when will LoRA provide the greatest savings in newly-created parameters?

   Solution: Original weight matrix have $d_1 \times d_2$ parameters; The lora matrix A and B have $d_1 \times p$ and $d_2 \times p$ parameters. So the ratio of parameters fine-tuned by LoRA to the number of parameters in W_ℓ^0 is

$$\frac{p \times (d_1 + d_2)}{d_1 \times d_2}$$

   Figure 4 and 5 show the ratio for different p values. It can be seen that LoRA provides the greatest savings when:

   i. The savings are greatest when this ratio is smallest.
   ii. d_1 and d_2 are large.
   iii. d_1 and d_2 are similar in magnitude (taking the derivative and making it 0), see figure 6.

   (b) Finish the `LoRALayerWrapper` in `ft.py`, which wraps a pre-trained linear layer with LoRA parameters. You can extract the shape of the pre-trained weight matrix from the `base_module.weight.shape` tuple. You don't need to worry about biases here, just the low-rank weight matrix residual.

   (c) Add the corresponding logic for LoRA in `ft.py:parameters_to_fine_tune()`. Hint: consider using the `.modules()` function of `nn.Module` and checking for modules that are an instance of `LoRALayerWrapper`.

   (d) Implement the 3-dim version of the loss and accuracy in `ft.py:get_loss()` and `ft.py:get_acc()`.

   (e) Implement batch construction for fine-tuning GPT-2 in function `ft.py:tokenize_gpt2_batch()`. Read the instructions in the code carefully!

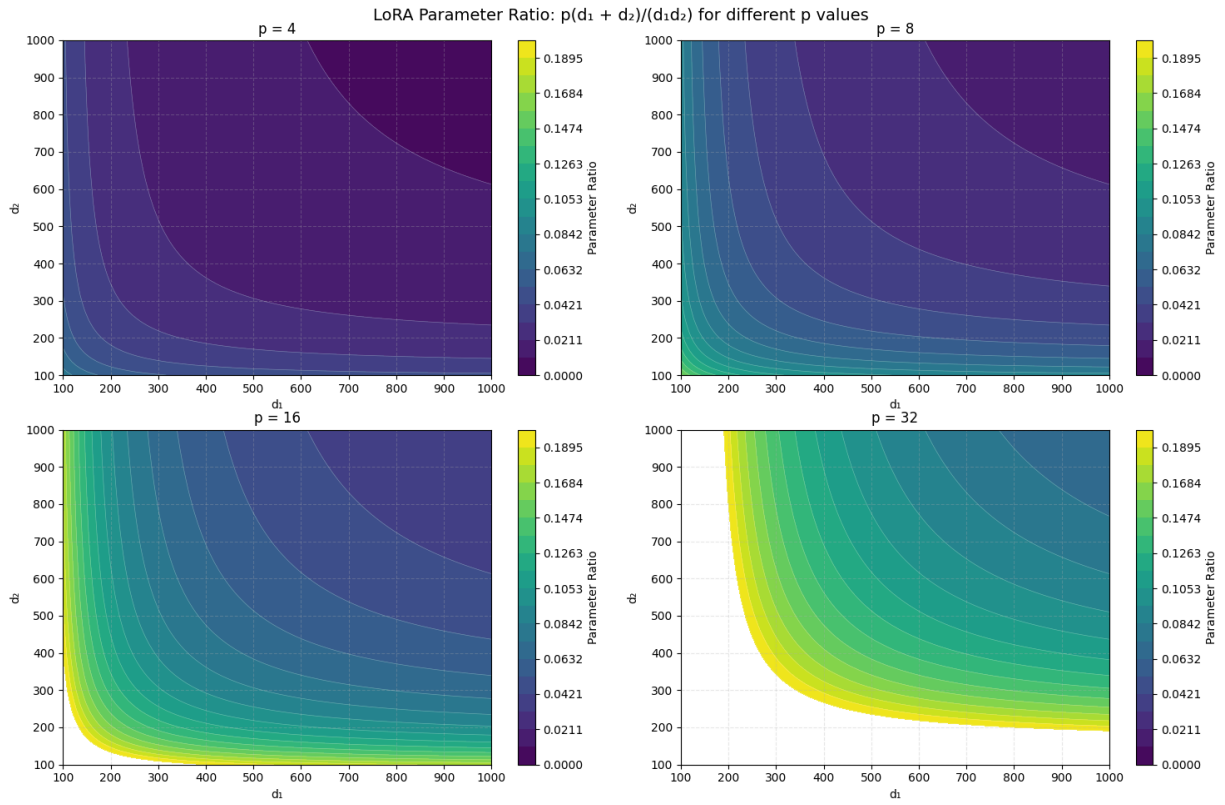

Figure 4: LoRA Parameter Ratio: $p(d_1 + d_2)/(d_1 d_2)$ for different p values.

(f) Finally, put it all together by filling out the logic for one step of training in `ft.py:ft_gpt2()`. Note that we use *gradient accumulation*, meaning that accumulate gradients over `grad_accum` steps, and only update our model's parameters after each `grad_accum` steps.

3. (15 pt) Run fine-tuning for each parameter-efficient fine-tuning method, using $p = 4, 16$ for LoRA (so, 5 variants in total); run the commands:

```
python ft.py --task ft --model med --mode first,last,middle,lora4,lora16 \
    --dataset xsum,babi --k 0,1,8,128
```

Plot **k-shot performance** as **k is varied** for GPT-2-medium, one plot for each dataset; run the commands:

```
python ft.py --task plot --model med --mode first,last,middle,lora4,lora16 \
    --dataset xsum --k 0,1,8,128 --plot_name Q2_3_xsum.png

python ft.py --task plot --model med --mode first,last,middle,lora4,lora16 \
    --dataset babi --k 0,1,8,128 --plot_name Q2_3_xsum.png
```

Solution: The results can be seen in figure 7 and 8. We can get that:

(a) Fine-tuning middle layers performs best.

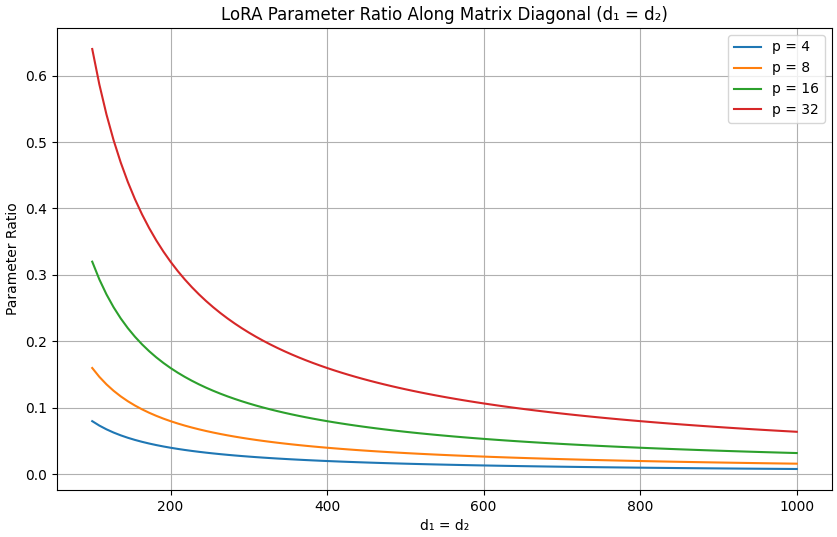

Figure 5: LoRA Parameter Ratio Along Matrix Diagonal $(d_1 = d_2)$.

(b) Using different Lora ranks shows similar performance on both datasets. Lower rank sometimes even performs better than larger rank.

(c) Generally, training with more examples works better.

(d) It's surprising that fine-tuning frontier layers works better than last layer as training frontier layers may disrupt the features learned by later layers. This may indicate that models trained with few examples learn general information but later layers need more examples to learn better.

(10 pt) Question 3: Comparing in-context learning and fine-tuning

1. (5 pt) Plot the few-shot performance of LoRA-16 and in-context learning for XSum in the same plot with the command:

   `python q3_plot.py`

   When does in-context learning seem like the better choice, with respect to the amount of data available? What about fine-tuning? What limitation of in-context learning does this result highlight?

   Solution: The results can be seen in figure 9. When we have little data available (e.g. 4), we should use in-context learning which avoids fine-tuning and shows better performance than fine-tuning. When we have more data like 64, we should try to use fine-tuning as it will yield better results. In the mean time, in-context learning requires much more inference memory and time when examples are getting more and

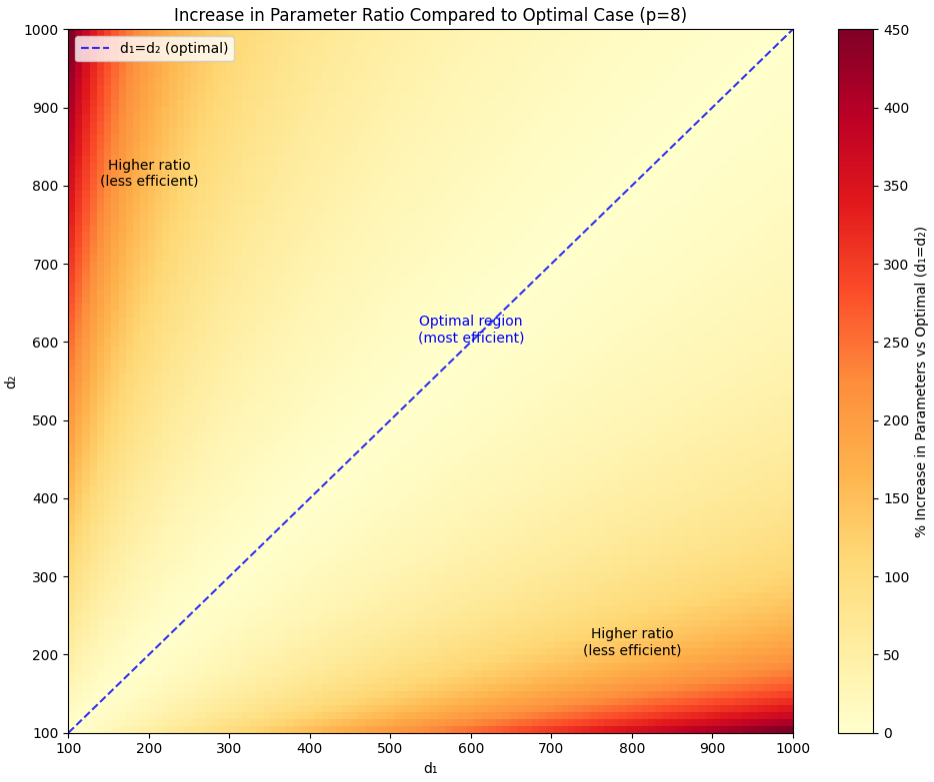

Figure 6: Increase in Parameter Ratio Compared to Optimal Case when $p = 8$.

more. It's also unstable when we use different orders of examples, a small change in order may cause performance disaster.

2. (5 pt) One potential disadvantage of in-context learning is that we must choose an ordering for the examples in our prompt, and *that ordering can sometimes impact performance negatively*. Run the command:

   ```
   python icl.py --task icl --model med --dataset babi --k 16 --repeats 5
   ```

   to compute the evaluation performance of in-context few-shot performance for 5 random orderings of the prompt. Report the number you get; the standard deviation of performance for fine-tuning is approximately 0.013. Does in-context learning or fine-tuning have a higher standard deviation?

   Solution: The results can be seen in table 1. The standard deviation of ICL is 0.019. Thus in-context learning has a higher standard deviation.

   (5 pt) Extra credit: See this paper for multiple heuristics for picking a prompt ordering. Implement either the globalE or localE heuristic, and report the accuracy you find for that ordering for 16-shot bAbI on GPT-2-medium. Compare it with the accuracy you found with random prompt orderings in question 1.

   Write your answer here.

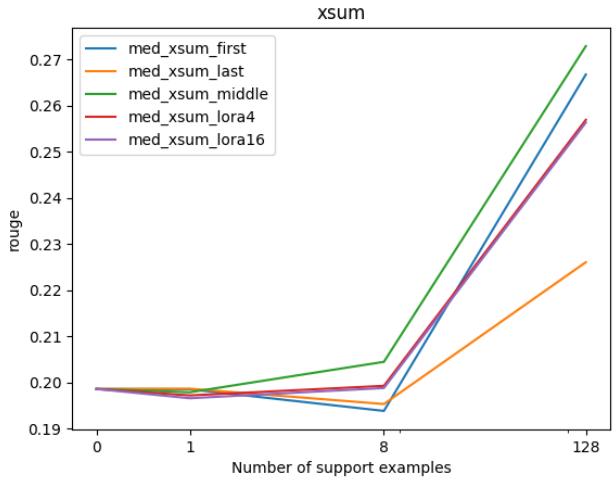

Figure 7: fine-tuning with different peft methods on xsum.

repeats	accuracy
1	0.688
2	0.728
3	0.712
4	0.696
5	0.672

Table 1: In-context learning with different orders.

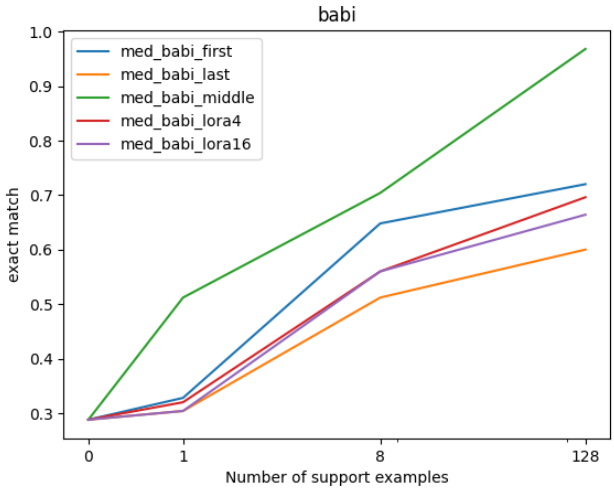

Figure 8: fine-tuning with different peft methods on babi.

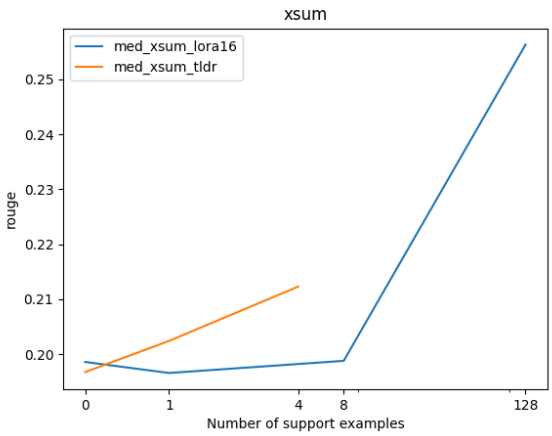

Figure 9: Few-shot performance vs. In-context learning on xsum.