XCS330 Problem Set 4

Due Sunday, May 28 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs330-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Few-Shot Learning with Pre-trained Language Models

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, using the HuggingFace library for loading pre-trained models and datasets. 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.

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.

A detailed description for every function can be found in the comments. You are not expected to change any code except between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`.

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.

All of these datasets are available through the HuggingFace library.

Coding Deliverables

For this assignment, please submit the following files to gradescope to recieve points for coding questions:

- `submission.pdf`
- `src/submission/__init__.py`
- `src/submission/ft.py`
- `src/submission/icl.py`
- `src/submission/results/ft/*.json`
- `src/submission/results/icl/*.json`

Note: due to the large amount of files required and the structure of the folders, the grader will only work when submitting a zip archive file containing the whole submission directory!

1 Fine-tuning

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.a **[5 points (Coding)] Implement Fine Tune**

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.

1.b **Observe Fine-Tuning**

(i) **[5 points (Coding)] Run Fine-Tuning**

Run the command:

```
python3 main.py --task run_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. `python3 main.py --task run_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:

```
python3 main.py --task plot_ft --model bert-tiny,bert-med --dataset amazon --k 1,8,128
```

Your plot should look as follows:

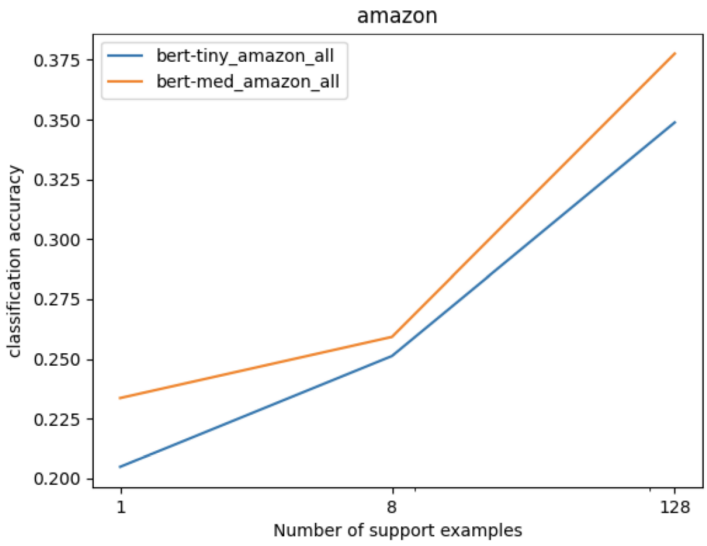

Figure 1: Performance of two sizes of BERT models on the Amazon Reviews dataset for various values of k

[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!

(ii) **[2 points (Written)] Reason on Fine-Tuning**

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

1.c **[2 points (Written)] Storage for Fine-Tune**

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?

1.d **[2 points (Written)] Disk Space**

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?

2 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.

2.a **[8 points (Coding)] Implement In-Context Learning**

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

(i) Implement prompt creation for the XSum and bAbI tasks, which is found in `icl.py:get_icl_prompts()`. You will implement 4 prompt format modes:

1 `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!). Note the `Q:` and `A:` prompts in the example earlier don't apply here.

2 `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.

3 `tldr` **[only for XSum]**: Add the text " `TL;DR:` " after each article/input (including the final test article) and before the summary/target.

4 `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.

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).

(ii) 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. $p_\theta(\mathbf{x}) = \prod_t p_\theta(x_t \mid x_{<t})$. 'Greedy' sampling means that given a context $x_{<t}$ producing a distribution over next tokens $p_\theta(x_t \mid x_{<t})$, we deterministically choose the next token x_t to be the token with highest probability.

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.

(iii) 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 less than 5 lines of code.

2.b **Evaluate on bAbI**

(i) **[5 points (Coding)] Run ICL on bAbI**

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:

```
python3 main.py --task run_icl --model med,full --dataset babi --k 0,1,16
```

Plot the results with the command:

```
python3 main.py --task plot_icl --model med,full --dataset babi --k 0,1,16
```

Your plots should look like:

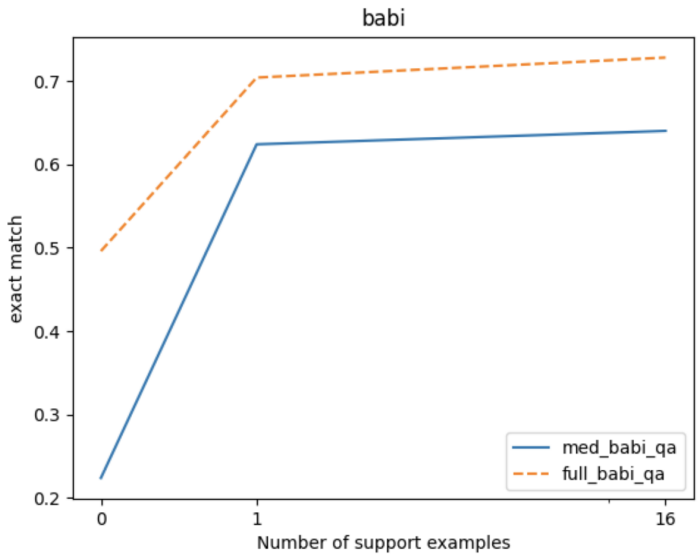

Figure 2: k-shot in-context performance on bAbI for GPT-2-medium and full-size GPT-2 for various values of k

(ii) **[2 points (Written)] Reason on ICL on bAbI**

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

2.c **Evaluate on XSum**

(i) **[5 points (Coding)] Run ICL on XSum**

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:

```
python3 main.py --task run_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 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:

```
python3 main.py --task plot_icl --model med,full --dataset xsum --k 0,1,4 \
    --prompt none,tldr,custom
```

Your plot should look like:

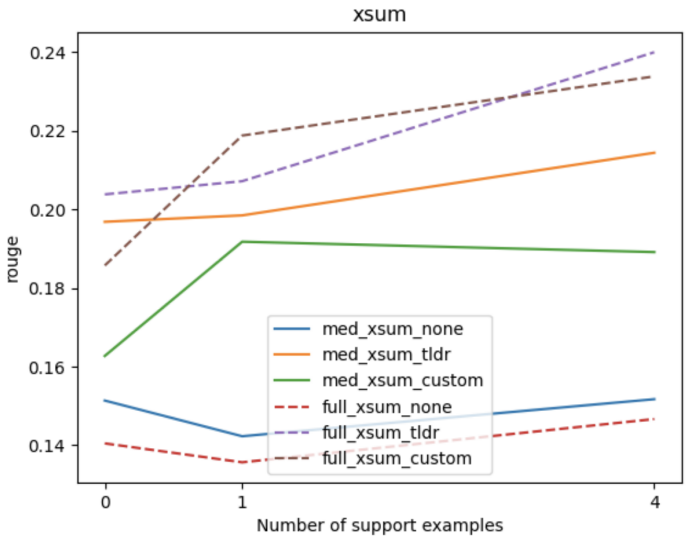

Figure 3: k-shot in-context performance of GPT-2-medium and full-size GPT-2 for various values of k on XSum

(ii) **[2 points (Written)] Reason on ICL on XSum**

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.

3 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.

3.a **[5 points (Coding)] Implement Parameter-efficient Fine Tine**

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

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

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

(iii) `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`.

3.b **[2 points (Written)] LoRA**

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.

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?

3.c **[14 points (Coding)] Implement LoRA**

With the information provided in the previous point let us know implement LoRA: Low-rank adaptation

(i) Finish the `LoRAConv1DWrapper` in `ft.py`, which wraps a pre-trained Conv1D layer with LoRA parameters. Conv1D is equivalent to a Linear layer; it's a 1D conv because we apply the same linear transform at each time step of the sequence. 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.

(ii) 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 `LoRAConv1DWrapper`.

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

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

(v) 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.d **Evaluate**

(i) **[5 points (Coding)] Run Fine-Tuning with LoRA**

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

```
python3 main.py --task run_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:

```
python3 main.py --task plot_ft --model med --mode first,last,middle,lora4,lora16 \
    --dataset xsum --k 0,1,8,128
```

Plot for the above is as follows:

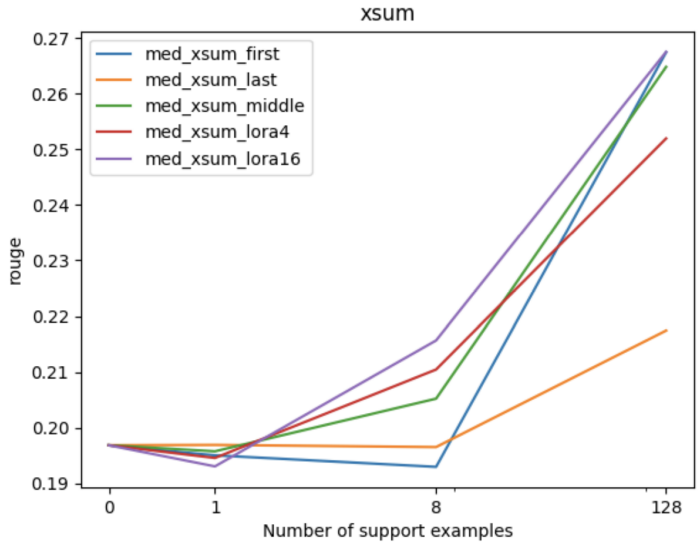

Figure 4: k-shot parameter-efficient fine-tuning performance using $p = 4, 16$ for LoRA of GPT-2-medium for various values of k on XSum

```
python3 main.py --task plot_ft --model med --mode first,last,middle,lora4,lora16 \
    --dataset babi --k 0,1,8,128
```

Plot for the above is as follows:

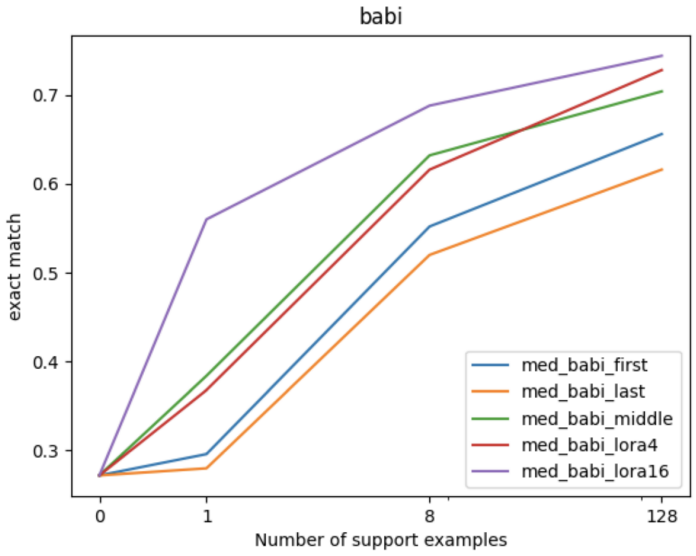

Figure 5: k-shot parameter-efficient fine-tuning performance using $p = 4, 16$ for LoRA of GPT-2-medium for various values of k on bAbI

(ii) **[2 points (Written)] Reason on Fine-Tuning with LoRA**

Describe the results here.

4 Comparing in-context learning and fine-tuning

4.a **[2 points (Written)] Compare LoRA and In-Context Learning for XSum**

Plot the few-shot performance of LoRA-16 and in-context learning for XSum in the same plot with the command:

```
python3 main.py --task plot
```

The plot should look as follows:

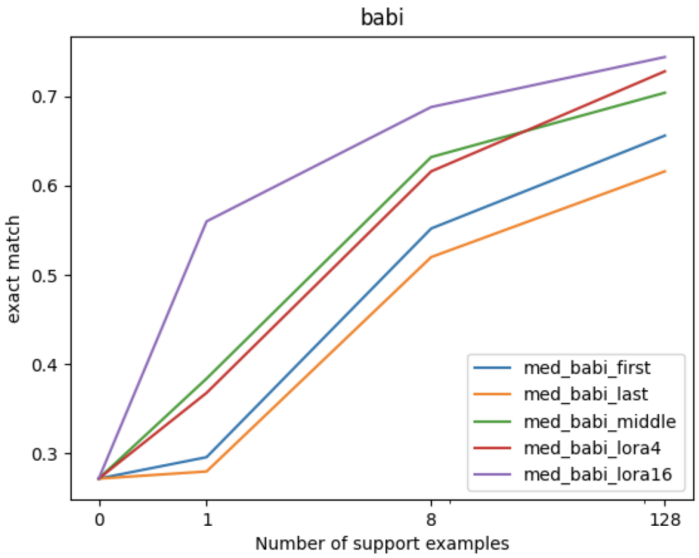

Figure 5: k-shot parameter-efficient fine-tuning performance using $p = 4, 16$ for LoRA of GPT-2-medium for various values of k on bAbI

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?

4.b **[2 points (Written)] Ordering**

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:

```
python3 main.py --task run_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?

4.c **[5 points (Written, Extra Credit)] 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.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.b (ii)

1.c

1.d

2.b (ii)
2.c (ii)

3.c
3.d (ii)

4.a
4.b
4.c