An analysis of Fine-Tuning methods on Gpt-Neo

Advanced NLP : Assignment 3

Varun Edachali (2022101029)

October 29, 2024

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

1 Analysis

All of the models in this analysis were trained with the same hyper-parameters to ensure a fair comparison of metrics. These include:

- **batch size:** 16

  Although, note that gradient-accumulation was utilized with an *accumulation factor* of 4, effectively mimicking a batch size of 64.

- **learning rate:** $5 \cdot 10^{-3}$
- **optimizer:** AdamW

  Here, the gradients were clipped with a norm of 1.0 to prevent exploding gradients. Although, this might have been a reason for slower than expected learning.

- **number of epochs:** 10
- **maximum length:** 256

  Ideally, I would have liked to set this to a larger value. But because of limited compute, I had to tune my models faster, and the only way I could increase my batch size was by decreasing the maximum length of each sample returned by my dataset. This could be a cause of some unexpected results.

- **patience:** 2

  This refers to the number of epochs of increase in validation loss that I was willing to tolerate, before early-stopping.

The model that was tuned is gpt-neo-125m. The models were trained with a train-valid-test split of 21000-6000-3000 samples on the cnn-dailymail text summarisation dataset. Model cleaning consisted of simple punctuation removal and conversion of the text to lower-case.

1.1 metadata metrics

Here, we analyse factors such as the number of trainable parameters, the maximum gpu memory occupied at any point of time during tuning, and the maximum time taken for a single epoch of fine-tuning.

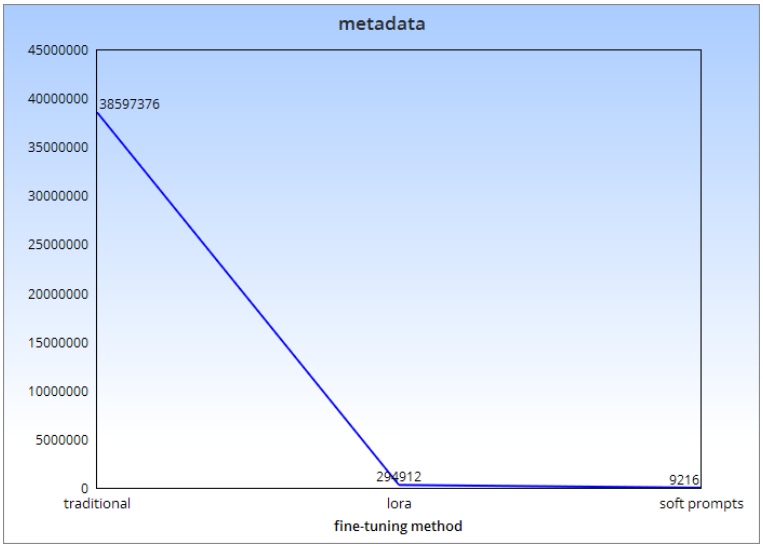

Figure 1: numer of trainable parameters

Here, we note that:

- in the traditional form of tuning, only the last layer (*lm_head*) was not frozen, and allowed to train on the given data.
- in lora, the trainable modules was not explicitly set in *lora_config* of *peft*, meaning all of the weight update matrices that could be decomposed, were decomposed and set to train. The lora attention dimension, or "rank", was set at 8.
- in soft prompting, the number of prompt tokens was set at 12, with an embedding size of 768.

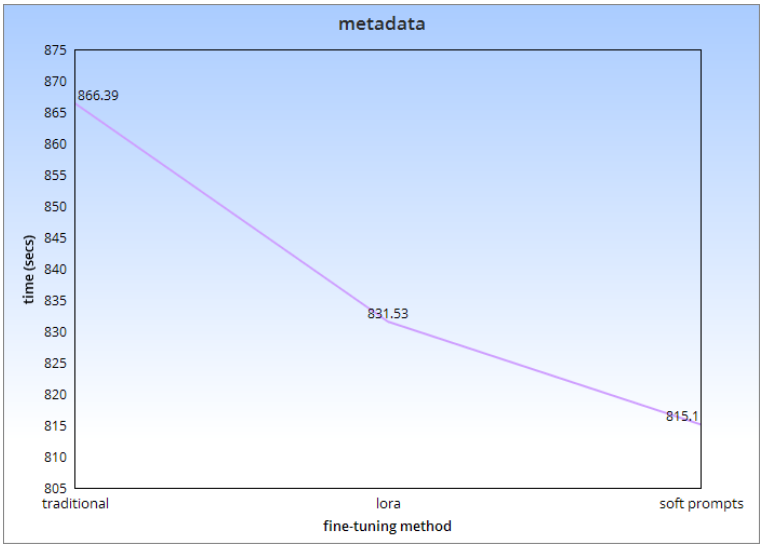

Figure 2: maximum time taken for an epoch of tuning

As we could easily hypothesize, a larger number of parameters led to more time required per epoch, due to more parameters having to be optimised. The results conform to our trivial hypothesis.

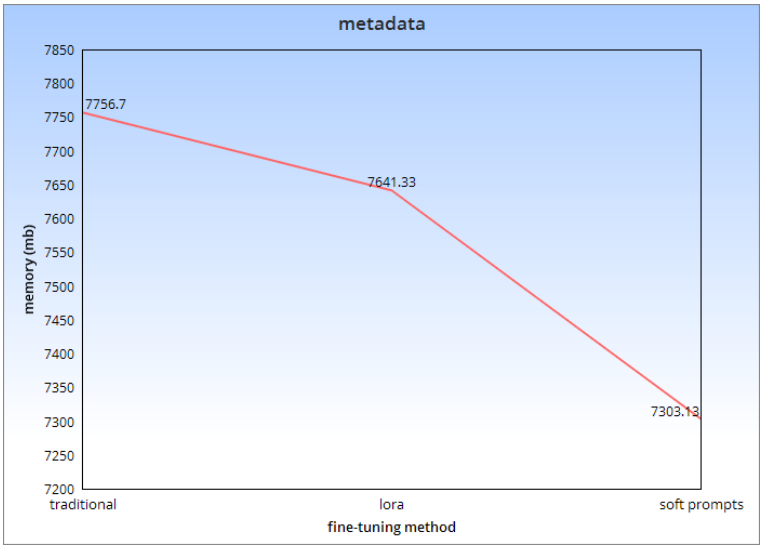

Figure 3: maximum gpu memory allocated during tuning

Again, as we may have hypothesized, since more parameters (and their gradients) have to be stored and optimized in the gpu space, the amount of gpu memory allocated increases with an increase in the number of parameters.

1.2 evaluation metrics

A possible justification for the low performance of LoRA is given later.

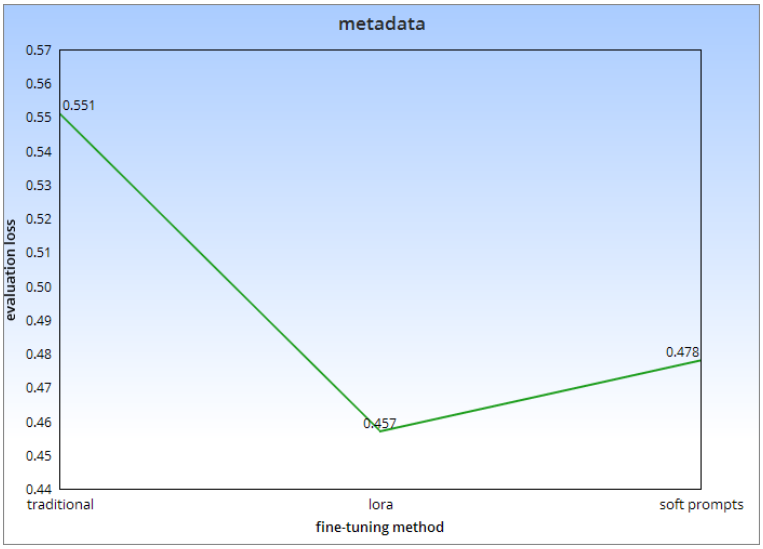

Figure 4: evaluation loss

The traditional method performs the best, likely because of the vast number of parameters that are allowed to be tuned. This could represent a lack of variance between the train and test sets, as usually model-tuning leaves models susceptible to over-fitting. This is followed by soft-prompting, which likely performs decently well owing to the simplicity of the task at hand - it can be represented by just 12 tokens.

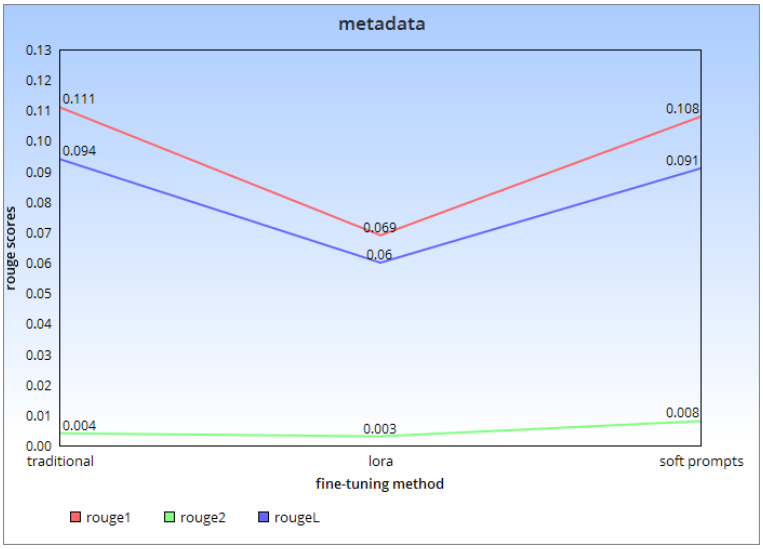

Figure 5: rouge scores

None of the methods seemed to perform well in rouge2 scores, indicating that they may capture individual words well but struggle with coherence. This could be because of the fact that we only trained on $21,000$ samples. Disregarding this, the relative performance of the tuning methods is similar to that in the evaluation loss.

1.3 Points to Consider

- The maximum length of 256 that I set was likely not enough. A lot of articles lengths exceeded this and had to be truncated. But, I simply did not have the compute to experiment with a larger size.

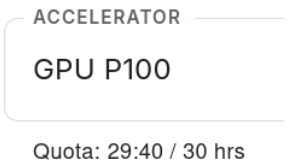

Figure 6: kaggle - remaining quota

- The learning rate of $5 \cdot 10^{-3}$ was not fair across methods. Some fine-tuning methods did not converge at all, while LoRA underwent early stopping. But again, I did not have the compute to tune the hyper-parameters further.
- Some possible reasons for LoRA not performing competitively, include:
  - The above learning rate was too large (for it), leading to unstable validation loss decreases. The reason for my belief in this possibility is the early stopping that it underwent.
  - I did not specify which modules to tune in particular. A more enhanced focus on, say, just the *qkv* may have allowed it to focus more on contextual information and perform better in some cases.
  - I did not set any parameters apart from the low rank, r. Perhaps if I set a higher value of *alpha*, it would make the model lean more heavily on the task-specific adjustments.

2 Theory

2.1 Concept of Soft Prompts

Most sub-sections here were derived by reading Google Research's blog introducing soft-prompts.

2.1.1 How does the introduction of "soft prompts" address the limitations of discrete text prompts in large language models?

Text-prompts require manual effort to design, and even well-designed prompts still far under-perform compared to model tuning.

Rather than selecting from existing vocabulary items, the "tokens" of the soft-prompt are learnable vectors. This means that a soft-prompt can be optimized over our training data. This removes the need for manual design, and also allows the prompt to condense information from datasets containing thousands or millions of examples. Discrete text prompts, however, are typically limited to under 50 examples due to constaints on model input length.

2.1.2 Why might soft prompts be considered a more flexible and efficient approach for task-specific conditioning?

This may be for a number of reasons:

Prompt tuning has a very small parameter footprint due to the reasons stated above and below.

Since soft prompts have a small parameter footprint, one can easily pass the model a different prompt along with each input example. This enables mixed-task inference batches, which can allow us to use one core model across many distinct tasks. It also allows us to easily swap task-specific vectors depending on our use-case, enhancing the flexibility of our pre-trained model across different domains.

Also note that the number of soft-prompts can be a hyper-parameter, giving us the flexibility to choose this parameter footprint too.

Another advantage of prompt tuning is its resilience to over-fitting. Since model tuning may affect every weight in the network, it has the capacity to easily overfit on the provided fine-tuning data and may not generalise well to variations in the task at inference time. By comparison, learned soft prompts have a smaller number of parameters, so the solutions they represent may be more generalisable.

2.2 Scaling and Efficiency in Prompt Tuning

2.2.1 How does the efficiency of prompt tuning relate to the scale of the language model?

As model size increases, prompt tuning catches up to the performance level of model tuning. Intuitively, the larger the pre-trained model, the less of a "push" it needs to perform a specific task, and the more capable it is of being adapted in a parameter-efficient way.

Thus, parameter-efficient prompt tuning becomes more capable as the scale of the language model increases.

2.2.2 Discuss the implications of this relationship for future developments in large-scale language models and their adaptability to specific tasks.

According to the same blog above, the effectiveness of prompt tuning at large model scales is quite significant, since serving separate copies of a large model can incur significant computational overhead. In the paper on The Power of Scale for Parameter-Efficient Prompt Tuning, the authors show that larger models can be conditioned successfully even with soft prompts as short as 5 tokens. They even exhibited that a 11 billion parameter model can be guided with just 20,000 parameters.

Essentially, it means we need not worry about making distinct copies of large models for specific tasks, instead storing dense, parameter-efficient soft-prompts, and getting comparable results with much less computational overhead.

2.3 Understanding LoRA

The major source for these subsections is the paper on LoRA.

2.3.1 What are the key principles behind Low-Rank Adaptation (LoRA) in fine-tuning large language models?

LoRA suggests that it's not just the model weights that have a low intrinsic dimension, the updates to the model weights do too. Thus, it injects trainable rank decomposition matrices into each layer. (source: Prof. Manish's PEFT slides)

To be more specific, some principles of LoRA include:

- *Intrinsic Rank Hypothesis*: significant changes to model weights can be captured using a lower-dimensional representation.
- *Decomposition of Weight Updates*: In traditional fine-tuning, the entire weight matrix W of a model is modified to adapt it for a specific task, resulting in

updates of the form $W := W + \Delta W$. Instead, LoRA proposes decomposing the weight matrices into the product of two smaller matrices A and B. This means that instead of directly modifying W, LoRA approximates the updates using a lower-dimensional representation of the form $W := W + BA$.

To quote the paper: "we hypothesize that the change in weights during model adaptation also has a low 'intrinsic rank', leading to our proposed Low-Rank Adaptation (LoRA) approach. LoRA allows us to train some dense layers in a neural network indirectly by optimising rank decomposition matrices of the dense layers' change during adaptation instead, while keeping the pre-trained weights frozen".

2.3.2 How does LoRA improve upon traditional fine-tuning methods regarding efficiency and performance?

The paper has a section called *Aren't Existing Solutons Good Enough?* whereby they scrutinise existing methods.

- **method 1: adding adapter layers**: The authors propose that these layers introduce inference latency, because they have to be processed sequentially, thus breaking the hardware parallelism that is required to keep latency low.

  By construction, LoRA guarantees that we do not introduce any additional latency during inference. When deployed in production, we can explicitly compute $W = W_0 + BA$ and perform inference as usual. When we need to switch to another downstream task, we can recover W_0 by subtracting BA and adding a different $B'A'$, a quick operation with little overhead.

- **method 2: optimising some forms of the input layer activations**: The authors propose that directly optimising the prompt is quite hard.
  - prefix tuning is difficult to optimise and its performance changes non-monotonically in trainable-parameters.
  - reserving a part of the sequence length for adaptation reduces the sequence length available to process the task itself.

  LoRA avoids this by optimising the weight update matrices.

2.4 Theoretical Implications of LoRA

2.4.1 Discuss the theoretical implications of introducing low-rank adaptations to the parameter space of large language models.

The success of LoRA suggests that the updates to model weights have a low intrinsic dimension.

Also, the paper suggests that the rank-deficiency of ΔW suggests that W could be rank deficient as well.

2.4.2 How does this affect the expressiveness and generalization capabilities of the model compared to standard fine-tuning?

When applying LoRA to all weight matrices and training all biases, we roughly recover the expressive-ness of full fine-tuning by setting the LoRA rank r to the rank of the pre-trained weight matrices. That is, as we increase the number of trainable parameters, training LoRA roughly converges to training the original model. Upto that point, it may be less expressive.

LoRA itself could also act as an implicit regularizer by reducing the number of parameters, potentially improving generalisation capability.