

Generative AI & Large Language Models

(Part II)

Ana Maria Sousa



Content

1. Limitations of ICL.....	2
2. Finetuning	2
3. Instruction Fine-Tuning.....	3
3.1. Common Steps Involved in Instruction Fine-Tuning.....	3
3.2. Fine-Tuning On a Single Task	4
3.3. Fine-Tuning On a Multiple Tasks.....	6
4. Model Evaluation	8
4.1. Metrics	8
5. Benchmarks.....	11
5.1. GLUE & SuperGLUE	11
5.2. Benchmarks for Massive Models	12
6. Parameter Efficient Fine-Tuning (PEFT)	13
6.1. PEFT Methods	15
6.2. Low Rank Adaptation (LoRA)	15
6.3. Soft Prompts	17

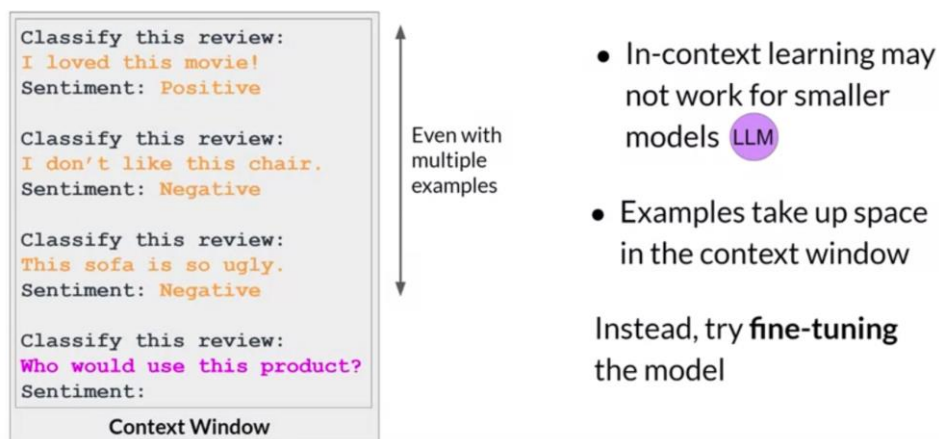
1. Limitations of ICL

Some models can identify instructions contained in a prompt and correctly carrying out zero-shot inference. However, others, in particular smaller models fail to do so. In such cases, we use **In-Context Learning (ICL)** to make the model follow our instructions.

However there some drawback to this:

- ICL may not always work for smaller models.
- Examples take up space in the context window, reducing the space available to add useful information in the prompt.

Limitations of in-context learning

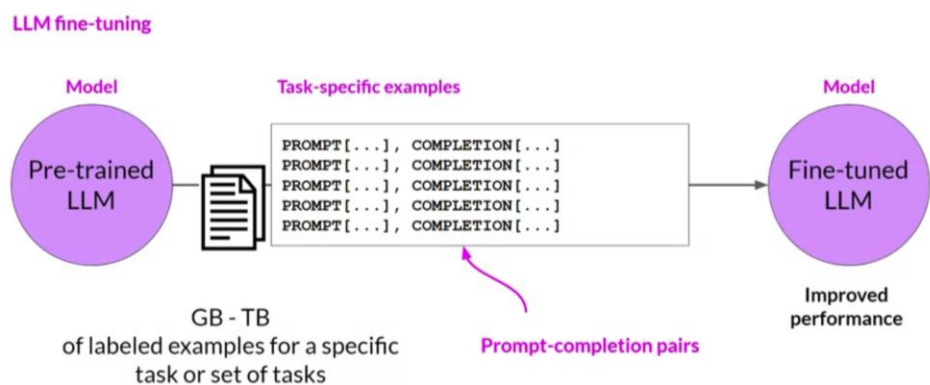


To combat these disadvantages while having a model that can follow instructions, we can use instruction fine-tuning.

2. Finetuning

The challenge with large language models is that they are trained on a lot of general information from the internet, but they may not know how to follow specific instructions. So, fine-tuning helps the model learn to respond better to our prompts.

Fine-tuning is the process of using labelled data to adapt a pre-trained model to a specific task or tasks. The data consists of prompt-completion pairs. Note that fine-tuning is applied on a pre-trained model and is supervised, as opposed to self-supervised.



There are two types of fine-tuning:

1. **Instruction fine-tuning**, where the model is trained on specific instructions to improve its performance.
2. **Efficient fine-tuning**, which allows us to fine-tune the model for specific tasks while using less memory and computation.

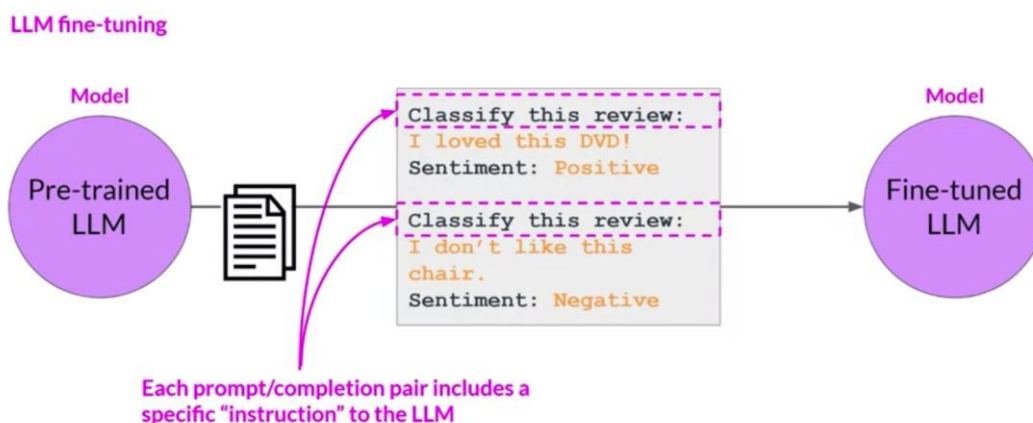
Overall, fine-tuning is important because it helps us make large language models more useful and efficient for specific tasks or applications.

3. Instruction Fine-Tuning

Instruction fine-tuning is a fine-tuning technique used to improve a model's performance on a variety of tasks. Here, the training samples are prompts containing instructions while the labels are the expected response of the model in order to follow that instruction.

Instruction fine-tuning where all the model's weights are updated is called **full fine-tuning**. This results in a new version of the model with updated weights.

Using prompts to fine-tune LLMs with instruction



3.1. Common Steps Involved in Instruction Fine-Tuning

3.1.1. Prepare & Split Dataset

There are many publicly available datasets that have been used to train previous generations of LLMs. Most of these datasets are not formatted as instructions. Developers have built prompt template libraries that can be used to take existing datasets and turn them into instruction prompt datasets for fine-tuning.

After the dataset is prepared, like any supervised problem, we split the dataset into training, validation and test sets.

3.1.2. Training

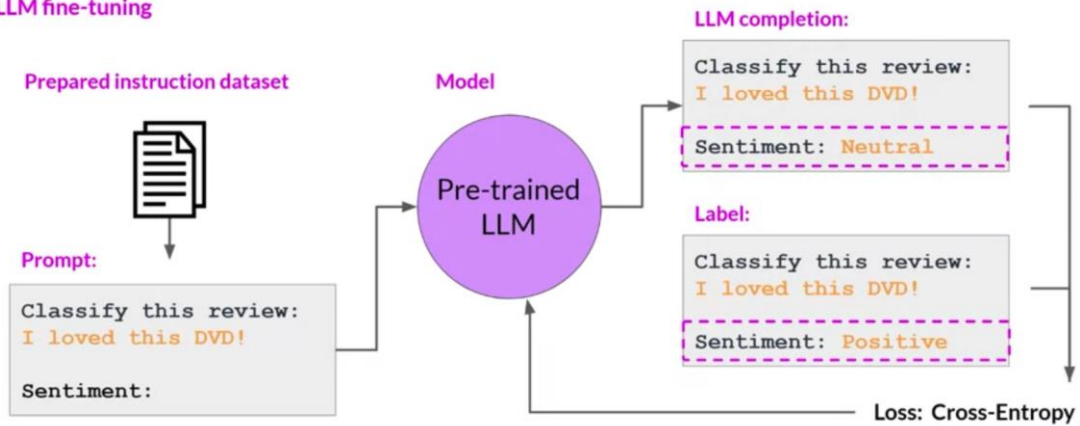
The fine-tuning training loop is like any other supervised training loop:

- Pass the training data in batches to the model and obtain predictions.
- **Calculate the loss.** The output of an LLM is a probability distribution over the tokens available in the dataset. Thus, we can compare the probability distribution of the prediction with that of the label and use the standard cross-entropy loss to calculate the loss.
- Calculate some **evaluation metric**.
- Pass the validation data to the model and obtain predictions.
- Calculate the loss (optional) and the same evaluation metric.

- Backpropagate the loss to update the weights and repeat from the beginning as the next epoch.

LLM fine-tuning process

LLM fine-tuning



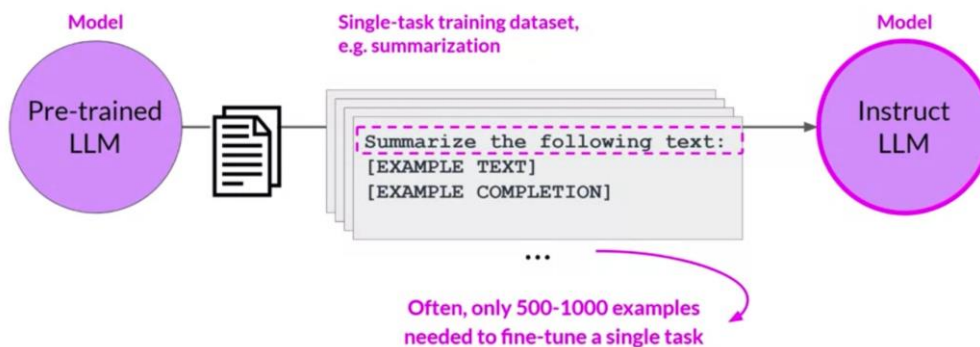
After training is done, we can evaluate the final performance of the model by passing it the test data and measuring the evaluation metric on model predictions. This process leads to a new version of the model, often called an **Instruct Model**. It tends to perform better at the tasks we have fine-tuned it for.



3.2. Fine-Tuning On a Single Task

Fine-tuning on a single task can be done by simply using a **single-task dataset**. That is, all prompt-completion pairs in the dataset have the **same basic instruction in them**.

Fine-tuning on a single task



3.2.1. Catastrophic Forgetting

Catastrophic forgetting

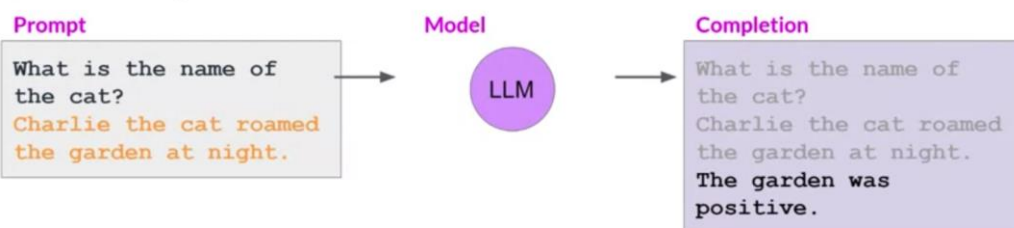
- Fine-tuning can significantly increase the performance of a model on a specific task...

After fine-tuning



- ...but can lead to reduction in ability on other tasks

After fine-tuning



Fine-tuning on a single task can lead to a problem called **catastrophic forgetting**. This happens since full fine-tuning changes the weights of the original LLM. This leads to great performance on the task we are fine-tuning for but can degrade performance on other tasks.

How to avoid catastrophic forgetting

- First note that you might not have to!
- Fine-tune on **multiple tasks** at the same time
- Consider **Parameter Efficient Fine-tuning (PEFT)**

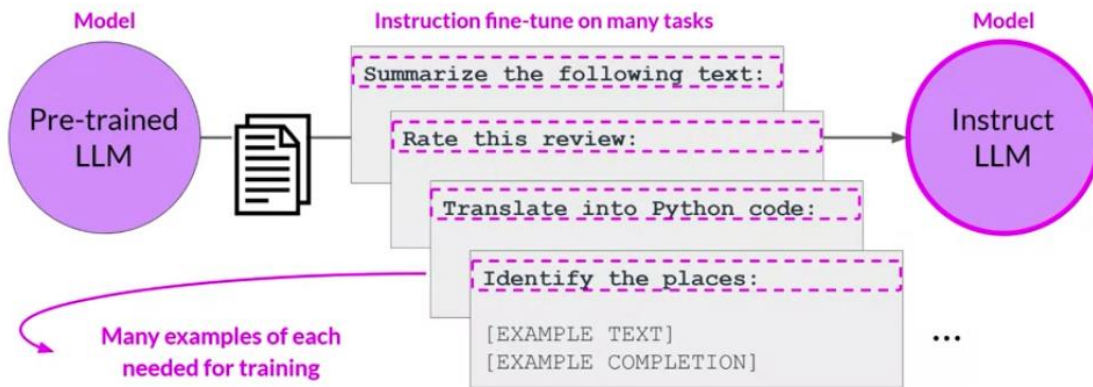
To avoid catastrophic forgetting:

- Figure out whether our model is impacted by the problem. For example, if we require reliable performance only on the single task we are fine-tuning for, we do not need to worry about catastrophic forgetting.
- If we want the model to maintain its multi-task performance, we can perform fine-tuning on multiple tasks at the same time.
- **Parameter Efficient Fine-Tuning (PEFT)**. PEFT preserves the weights of the original LLM and trains only a small number of task-specific adapter layers and parameters.

3.3. Fine-Tuning On a Multiple Tasks

In case of multiple tasks, the dataset contains prompt-completion pairs related to multiple tasks. The model is trained on this mixed dataset to fine-tune on multiple tasks simultaneously and remove the risk of catastrophic forgetting.

Multi-task, instruction fine-tuning



3.3.1. Instruction fine-tuning with FLAN.

FLAN (Fine-tuned Language Net) is a family of models fine-tuned on multiple tasks. **FLAN models** refer to a specific set of instructions used to perform instruction fine-tuning.

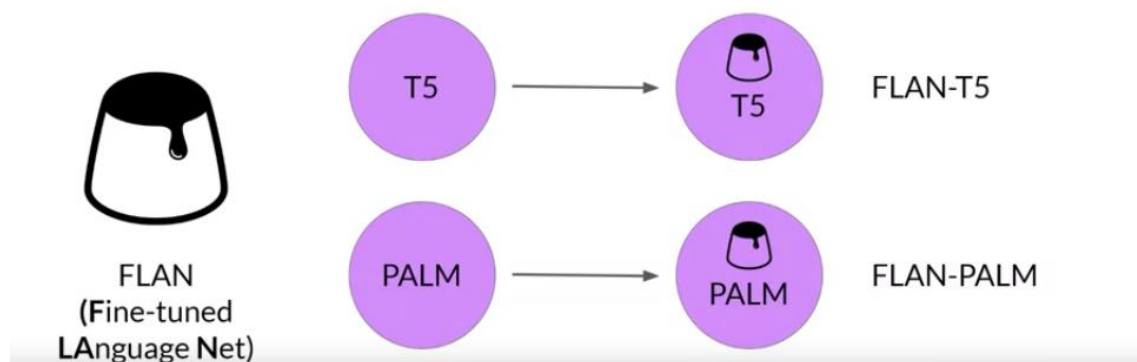


FLAN

"The metaphorical dessert to the main course of pretraining"

Instruction fine-tuning with FLAN

- FLAN models refer to a specific set of instructions used to perform instruction fine-tuning

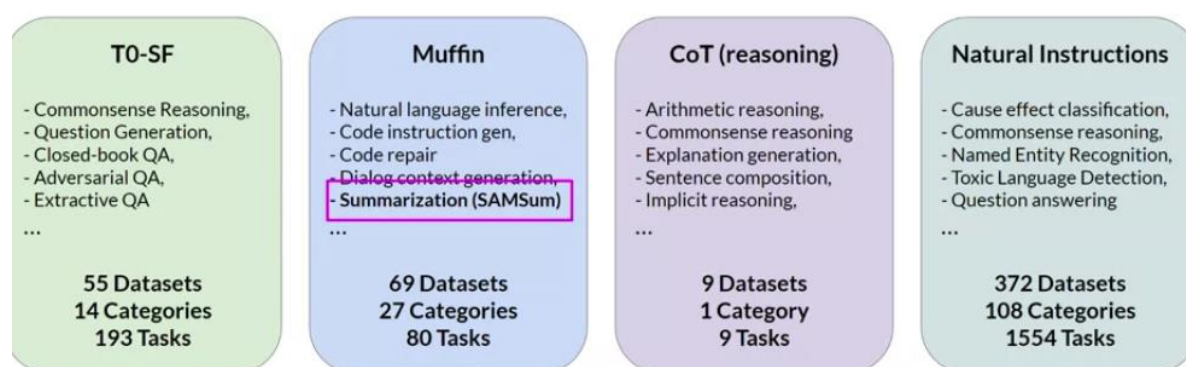


FLAN-T5 is general purpose instruct model. It is fine-tuned on 473 datasets across 146 task categories. These datasets are chosen from other models and papers.

For example, the **SAMSum** dataset is a text summarization dataset.

FLAN-T5: Fine-tuned version of pre-trained T5 model

- FLAN-T5 is a great, general purpose, instruct model



SAMSum: A dialogue dataset

Sample prompt training dataset (**samsum**) to fine-tune FLAN-T5 from pretrained T5

Datasets: samsum		Tasks:	Summarization	Languages:	English
dialogue (string)		summary (string)			
"Amanda: I baked cookies. Do you want some? Jerry: Sure! Amanda: I'll bring you tomorrow :-)"		"Amanda baked cookies and will bring Jerry some tomorrow."			
"Olivia: Who are you voting for in this election? Oliver: Liberals as always. Olivia: Me too!! Oliver: Great"		"Olivia and Olivier are voting for liberals in this election. "			
"Tim: Hi, what's up? Kim: Bad mood tbh, I was going to do lots of stuff but ended up procrastinating Tim: What did..."		"Kim may try the pomodoro technique recommended by Tim to get more stuff done."			

Example support-dialog summarization

Prompt (created from template)

Summarize the following conversation.

Tommy: Hello. My name is Tommy Sandals, I have a reservation.

Mike: May I see some identification, sir, please?

Tommy: Sure. Here you go.

Mike: Thank you so much. Have you got a credit card, Mr. Sandals?

Tommy: I sure do.

Mike: Thank you, sir. You'll be in room 507, nonsmoking, queen bed.

Tommy: That's great, thank you!

Mike: Enjoy your stay!

4. Model Evaluation

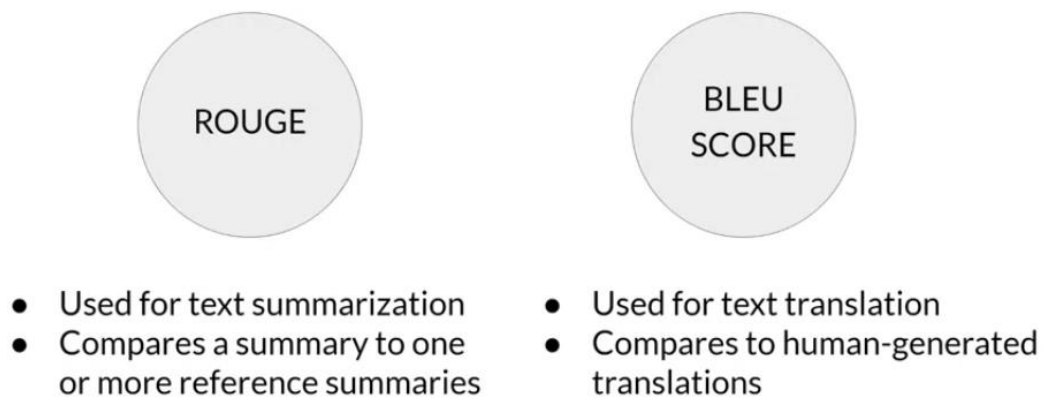
In traditional ML, we use the accuracy score to assess how well a model performs on a given task. This works since a model's output is always deterministic. That is, given the same input, the model always gives the same output.

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

LLMs are non-deterministic. Given the same prompt twice, an LLM is likely to produce two different responses instead of the same response. This makes plain accuracy score an inappropriate metric for evaluation.

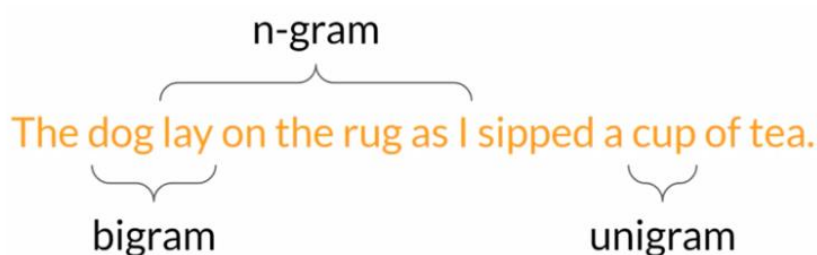
4.1. Metrics

LLM Evaluation - Metrics



4.1.1. Terminology n-gram

An n-gram refers to a collection of n words in a piece of text. For example, an unigram refers to a single word and a bigram refers to two words.



4.1.2. ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is primarily used for text summarization. It compares a summary to one or more human-generated reference summaries.

4.1.2.1. ROUGE-1

ROUGE-1 looks at unigrams (hence the one in the name). The score has a recall, precision and F1-score related to it.

In other words, the recall score for ROUGE-1 measures the number of unigrams matches with respect to the human reference.

$$\text{ROUGE-1}_{\text{recall}} = \frac{\text{unigram matches}}{\text{unigrams in reference}}$$

The precision score measures the number of unigrams matches with respect to the generated output. Finally, the F1-score is defined as usual (harmonic mean of precision and recall).

$$\text{ROUGE-1}_{\text{precision}} = \frac{\text{unigram matches}}{\text{unigrams in output}}$$

$$\text{ROUGE-1}_{f1} = 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

This is a very basic metric which only looks at individual words and does not consider the ordering of the words. It is easy to construct sentences that score well but are subjectively poor responses.

ROUGE-2 works on bigrams, looking at a collection of two words at a time. The definition of the recall, precision and F1-scores are the same except unigrams are replaced with bigrams.

LLM Evaluation - Metrics - ROUGE-2

<div>Reference (human):</div> <p>It is cold outside.</p> <div>It is is cold</div> <div>cold outside</div> <div>Generated output:</div> <p>It is very cold outside.</p> <div>It is is very</div> <div>very cold cold outside</div>	<div>ROUGE-2 Recall:</div> $= \frac{\text{bigram matches}}{\text{bigrams in reference}} = \frac{2}{3} = 0.67$
	<div>ROUGE-2 Precision:</div> $= \frac{\text{bigram matches}}{\text{bigrams in output}} = \frac{2}{4} = 0.5$
	<div>ROUGE-2 F1:</div> $= 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = 2 \frac{0.335}{1.17} = 0.57$

4.1.2.2. ROUGE-L

Instead of looking at n-grams with larger values of n, **ROUGE-L** finds the longest common subsequence (**LCS**) between the human reference and the generated output. The LCS is computed at the word level instead of character level.



LLM Evaluation - Metrics - ROUGE-L

Reference (human): <u>It is cold outside.</u>	ROUGE-L Recall: $= \frac{\text{LCS}(\text{Gen, Ref})}{\text{unigrams in reference}} = \frac{2}{4} = 0.5$
Generated output: <u>It is very cold outside.</u>	ROUGE-L Precision: $= \frac{\text{LCS}(\text{Gen, Ref})}{\text{unigrams in output}} = \frac{2}{5} = 0.4$
LCS: Longest common subsequence	ROUGE-L F1: $= 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = 2 \frac{0.2}{0.9} = 0.44$

4.1.2.3. ROUGE Hacking

The problem with ROUGE scores is that they can lead to high scores for bad outputs. To combat this, we can use **ROUGE clipping**, which limits the number of unigrams matches to the maximum count of that unigram in the reference.

LLM Evaluation - Metrics - ROUGE clipping

Reference (human): <u>It is cold outside.</u>	ROUGE-1 Precision: $= \frac{\text{unigram matches}}{\text{unigrams in output}} = \frac{4}{4} = 1.0$	
Generated output: <u>cold cold cold cold</u>	Modified precision: $= \frac{\text{clip(unigram matches)}}{\text{unigrams in output}} = \frac{1}{4} = 0.25$	
Generated output: <u>outside cold it is</u>	Modified precision: $= \frac{\text{clip(unigram matches)}}{\text{unigrams in output}} = \frac{4}{4} = 1.0$	

There are still issues if the words are present in the generated output but in a different order. Consider the output outside cold it is. This will have a perfect score, even with clipping. Thus, experimenting with an n-gram size that will calculate the most useful score will depend on the sentence, the sentence size and our use case.

4.1.3. BLEU Score

BLEU (BiLingual Evaluation Understudy) is primarily used for machine translation. It compares a translation to human-generation translations.

The score is computed using the average precision over multiple n-gram sizes. It is similar to the ROUGE score but is calculated over a range of n-gram sizes and then averaged.

$$\text{BLEU} = \text{avg}(\text{precision across range of } n\text{-gram sizes})$$

LLM Evaluation - Metrics - BLEU

BLEU metric = Avg(precision across range of n-gram sizes)

Reference (human):

I am very happy to say that I am drinking a warm cup of tea.

Generated output:

I am very happy that I am drinking a cup of tea. - BLEU 0.495

I am very happy that I am drinking a warm cup of tea. - BLEU 0.730

I am very happy to say that I am drinking a warm tea. - BLEU 0.798

These scores can be used for a simple reference as we iterate over models but should not be used alone to report the final evaluation of an LLM. We can use them for diagnostic evaluation, but for overall evaluation, we need to use benchmarks.

5. Benchmarks

Benchmarks allow comparing LLMs more holistically. We use pre-existing datasets and associated benchmarks that have been established by LLM researchers specifically for model evaluation.



MMLU (Massive Multitask
Language Understanding)

BIG-bench 

Selecting the right evaluation dataset is vital in order to accurately assess an LLMs performance and understand its true capabilities. Our focus should be on selecting datasets that isolate specific model skills like reasoning and common-sense knowledge, and those that focus on potential risks, such as disinformation or copyright infringement.

An important issue to consider is whether the model has seen our evaluation dataset during training. We can obtain a more accurate and useful sense of the model's abilities by evaluating its performance on data it hasn't seen before.

5.1. GLUE & SuperGLUE

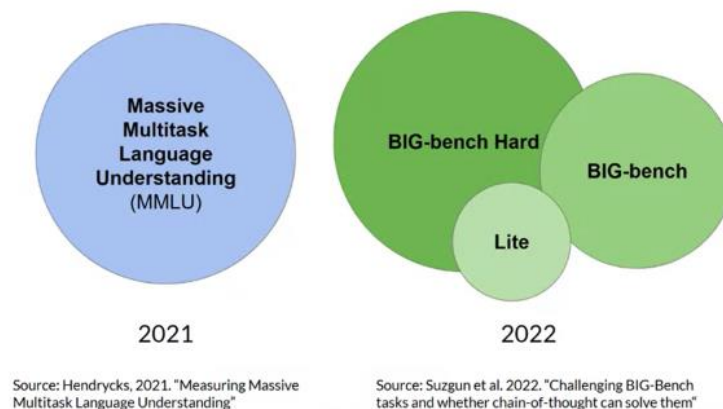
GLUE (General Language Understanding Evaluation) is a collection of natural language tasks such as sentiment analysis and question-answering.

It is created to encourage the development of models that can generalize across multiple tasks. Thus, we can use this benchmark to measure and compare model performance.

SuperGLUE is a successor to GLUE, developed to overcome the limitations of GLUE. It includes some tasks that were not included in GLUE and some tasks that more challenging versions of the same task.

5.2. Benchmarks for Massive Models

Benchmarks for massive models



As models become larger and larger, they start matching human performance on **GLUE** and **SuperGLUE**. That is, they perform at a human level on the benchmarks but subjectively, we can see that they are not performing at human level at tasks in general.

Thus, there is an arms race between the emerging capabilities of LLMs and the benchmarks that aim to measure them. Some emerging benchmarks used for modern LLMs are mentioned below.

5.2.1. MMLU

MMLU (Massive Multitask Language Understanding) is designed for modern LLMs. To perform well on this benchmark, models must have extensive world knowledge and problem-solving ability. The benchmark includes task that extend beyond basic language understanding. They are tested on elementary mathematics, US history, computer science, law and many more topics.

5.2.2. BIG-Bench

BIG-Bench currently consists of 204 tasks involving linguistics, childhood development, math, common sense reasoning, biology, physics, social bias, software development and more.

It comes in three different sizes (*Lite*, *BIG-bench*, *BIG-bench Hard*) to keep costs manageable since running these large benchmarks can incur large inference costs.

5.2.3. HELM

HELM (Holistic Evaluation of Language Models) aims to improve the transparency of LLMs and offer guidance on which models perform well for specific tasks.

It takes a multi-metric approach, measuring seven metrics across 16 core scenarios. This ensures that trade-offs between models and metrics are clearly exposed.

Some of the metrics it measures are:

- Accuracy
- Calibration
- Robustness
- Fairness
- Bias
- Toxicity
- Efficiency

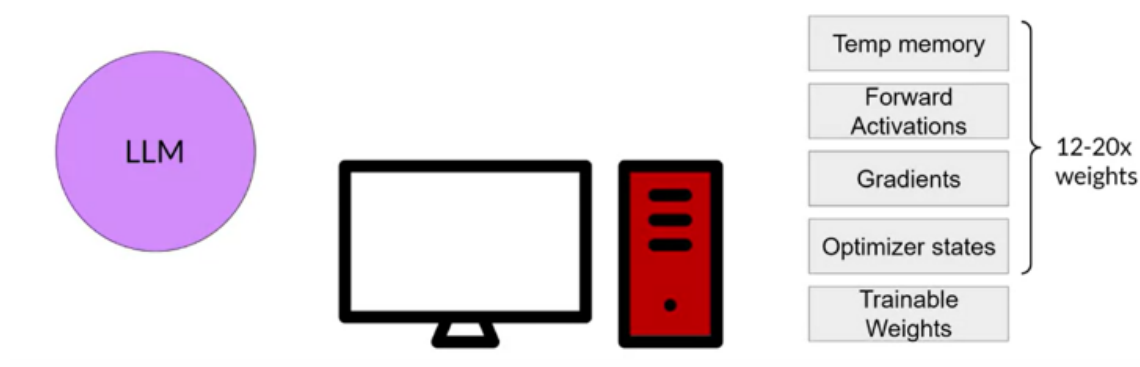
The benchmark goes beyond basic accuracy measures such as precision and F1 score. It also measures fairness, bias and toxicity, which are becoming increasingly important to assess as LLMs become more capable of human-like generation and in turn, of exhibiting potentially harmful behavior.

HELM is a *living benchmark*, meant to continuously evolve with the addition of new scenarios, metrics and models.

6. Parameter Efficient Fine-Tuning (PEFT)

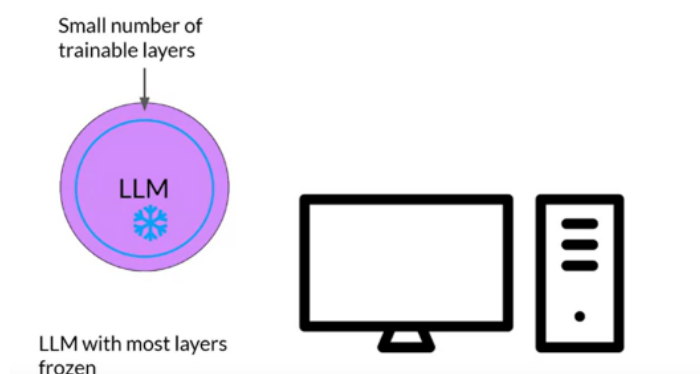
Full-fine tuning of large language LLMs is challenging. Fine-tuning requires storing training weights, optimizer states, gradients, forward activations and temporary memory. Things to store other than the weights can take up to 12-20 times more memory than the weights themselves.

Full fine-tuning of large LLMs is challenging



In full fine-tuning, every weight of the model is updated during training. **PEFT methods only update a subset of the weights.** They involve freezing most of the layers in the model and allowing only a small number of layers to be trained. Other methods don't change the weights at all and instead, add new layers to the model and train only those layers.

Parameter efficient fine-tuning (PEFT)

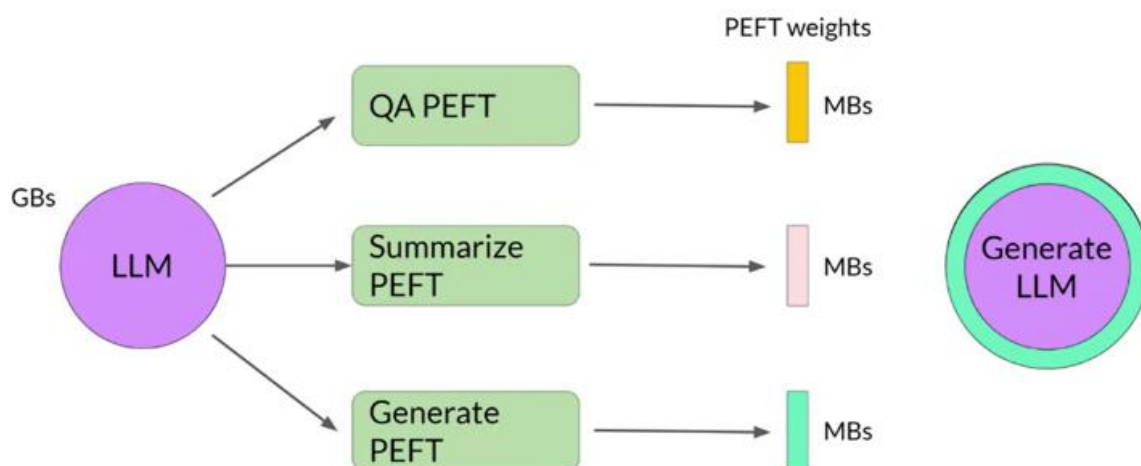


Parameter efficient fine-tuning (PEFT)



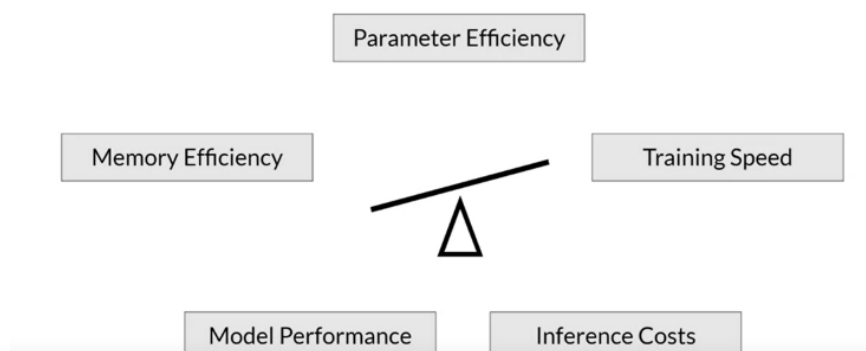
Due to this, the number of trainable weights is much smaller than the number of weights in the original LLM. **This reduces the overall memory** requirement for training, so much so that PEFT can often be performed on a single GPU. Since most of the LLM is left unchanged, PEFT is also less prone to **Catastrophic Forgetting**.

PEFT fine-tuning saves space and is flexible



PEFT weights are trained separately for each task. They are combined with the original weights of the LLM for inference. This makes them easily swappable, allowing efficient adaptation of the model to different tasks.

PEFT Trade-offs



6.1. PEFT Methods

6.1.1. Selective

We select a subset of initial LLM parameters to fine-tune.

There are several approaches to select which subset of parameters we want to fine-tune. We can decide to train:

- Only certain components of the model.
- Specific layers of the model.
- Individual parameter types.

The performance of these approaches and the selective method overall is mixed. There are significant trade-offs in parameter efficiency and compute efficiency and hence, these methods are not very popular.

6.1.2. Reparameterization

The model weights are reparametrized using a low-rank representation. Example techniques are **Low Rank Adaptation (LoRA)**

6.1.3. Additive

New, trainable layers or parameters are added to the model. There are generally two methods:

- **Adapters** - New trainable layers are added to the model, typically inside the encoder or decoder blocks, after the FFNN or the attention layers.
- **Prompt Tuning** - The model architecture is kept fixed and instead, the input (prompt) is manipulated to obtain better performance. This can be done by adding trainable parameters to the prompt embeddings, or keeping the input fixed and retraining the embedding weights. Example techniques include **Soft Prompts**.

6.2. Low Rank Adaptation (LoRA)

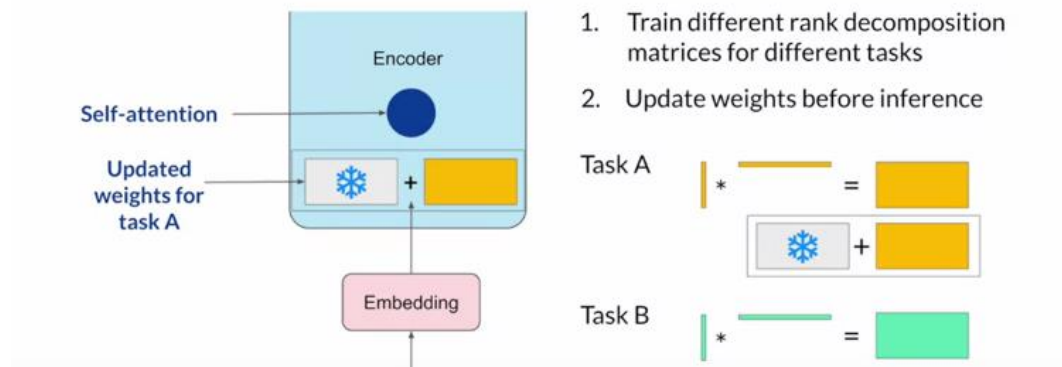
6.2.1. Introduction

LoRA is a PEFT technique based on reparameterization. The encoder and decoder blocks of a Transformer consist of self-attention (in the form of Multi-Headed Attention) layers. Weights are applied to the input embedding vectors to obtain an attention map for the input prompt.

In full fine-tuning, every weight in these layers is updated. In LoRA:

- All the model parameters are frozen.
- Two (smaller) **rank decomposition matrices A and B** are injected with the original weights. The dimensions of the matrices are such that their product has the same dimension as that of the original weight matrices.
- The weights in the smaller matrices are trained via fine-tuning

LoRA: Low Rank Adaption of LLMs



For inference:

- We multiply the two low rank matrices to obtain $B \times A$, which has the same dimensions as the frozen weights of the model.
- We add $B \times A$ to the original frozen weights.
- The model weights are replaced with these new weights.

We now have a fine-tuned model which can carry out the task(s) we have fine-tuned it for. Since the model has the same number of parameters as original, there is little to no impact on inference latency.

Researchers have found that applying LoRA just to the self-attention layers is often enough to fine-tune for a task and achieve performance gains. However, in principle, we can use LoRA in other components such as the feed-forward layers. Since most of the parameters of the model are in the attention layers, we get the biggest savings when we apply LoRA in those layers.

6.2.2. Multiple Tasks

LoRA also makes it easy to fine-tune a model for different tasks. We can train the model using the rank decomposition matrices for each of the tasks. This will give us a pair of A and B matrices for each task.

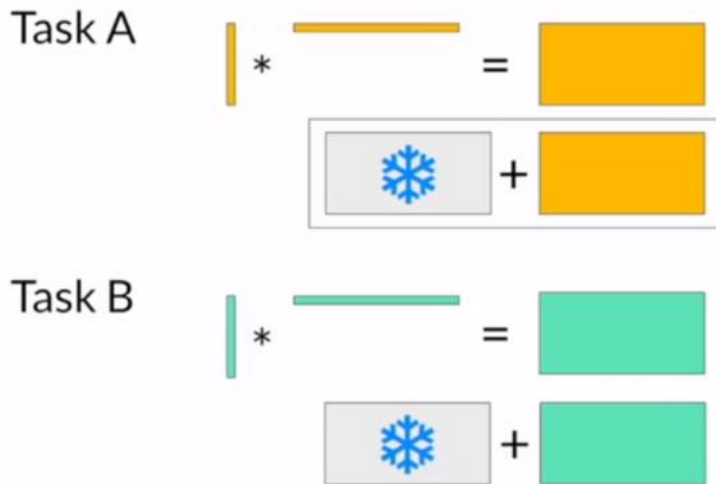
During inference, we can swap out the matrices depending on the task we want the model to do and update the weights (by adding to the frozen weights).

We can see that the LoRA model almost matches the fully fine-tuned model in performance, and both outperform the base model (no fine-tuning). In other words, LoRA can achieve performance which is close to full fine-tuning while significantly reducing the number of parameters that need to be trained.

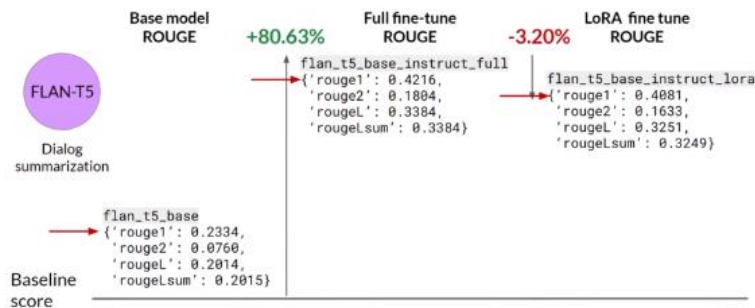
In general for choosing the **Rank r**:

The smaller the rank r , the smaller the number of trainable parameters and the bigger the savings on compute.

- Effectiveness of higher rank appears to plateau. That is, after a certain rank value, making it larger generally has no effect on performance.
- $4 \leq r \leq 32$ (in powers of 2) can provide a good trade-off between reducing trainable parameters and preserving performance.
- Relationship between rank and dataset size needs more research.



Base Model vs Full Fine-Tuning vs LoRA



Choosing the LoRA rank

Rank r	val_loss	BLEU	NIST	METEOR	ROUGE.L	CIDEr
1	1.23	68.72	8.7215	0.4565	0.7052	2.4329
2	1.21	69.17	8.7413	0.4590	0.7052	2.4639
4	1.18	70.38	8.8439	0.4689	0.7186	2.5349
8	1.17	69.57	8.7457	0.4636	0.7196	2.5196
16	1.16	69.61	8.7483	0.4629	0.7177	2.4985
32	1.16	69.33	8.7736	0.4642	0.7105	2.5255
64	1.16	69.24	8.7174	0.4651	0.7180	2.5070
128	1.16	68.73	8.6718	0.4628	0.7127	2.5030
256	1.16	68.92	8.6982	0.4629	0.7128	2.5012
512	1.16	68.78	8.6857	0.4637	0.7128	2.5025
1024	1.17	69.37	8.7495	0.4659	0.7149	2.5090

- Effectiveness of higher rank appears to plateau
- Relationship between rank and dataset size needs more empirical data

6.3. Soft Prompts

6.3.1. Introduction

Prompt tuning is not prompt engineering.

Prompt engineering involves **modifying the language** of the prompt in order to “urge” the model to generate the completion that we want. This could be as simple as trying different words, phrases or including examples for **In-Context Learning (ICL)**. The goal is to help the model understand the nature of the task and to generate better completions.

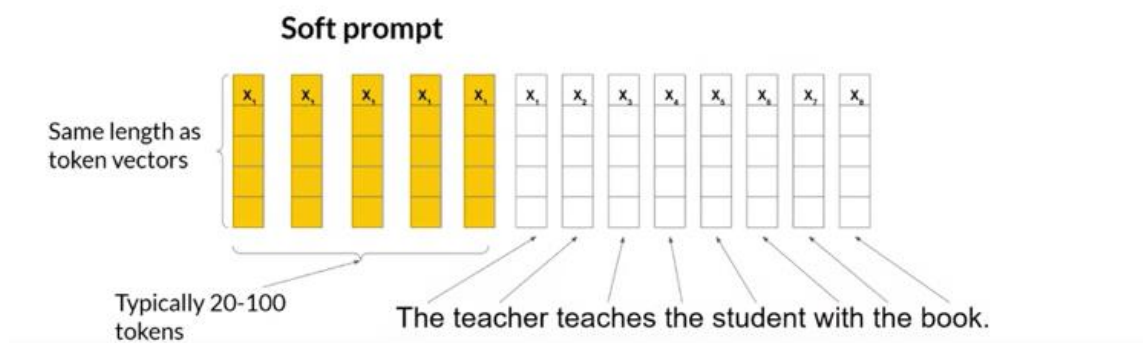
This involves some limitations:

- We require a lot of manual effort to write and try different prompts.
- We are also limited by the length of the context window.

Prompt tuning adds trainable “soft prompts” to inputs that are learnt during the supervised fine-tuning process.

The set of trainable tokens is called a soft prompt. It is prepended to the embedding vectors that represent the input prompt. The soft prompt vectors have the same length as the embeddings. Generally, 20-100 “virtual tokens” can be sufficient for good performance.

Prompt tuning adds trainable “soft prompt” to inputs

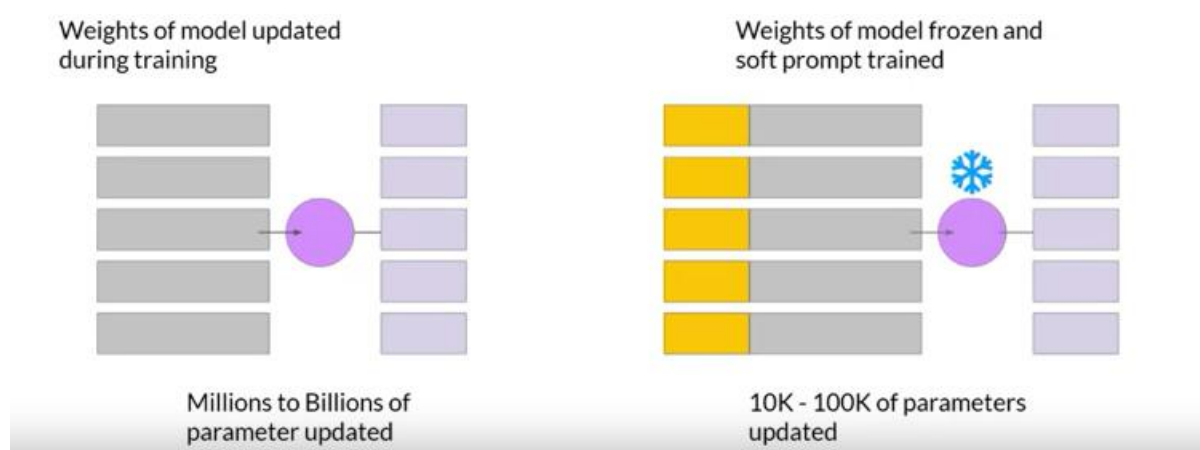


The tokens that represent natural language correspond to a fixed location in the embedding vector space. On the other hand, soft prompts are not fixed discrete words of natural language and can take on any values within the continuous multidimensional embedding space.

6.3.2. Prompt Tuning vs Full Fine-tuning

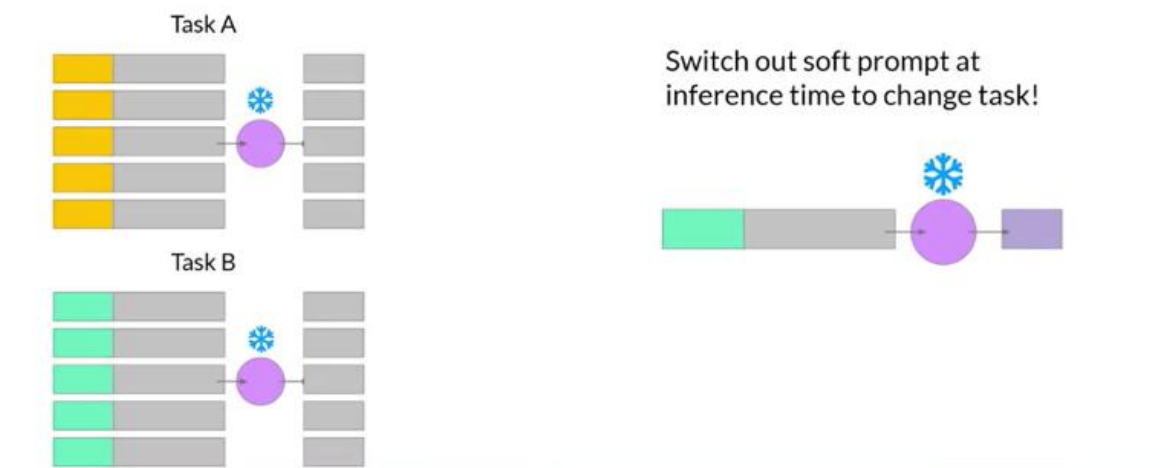
Prompt tuning does not involve updating the model. Instead, the model is completely frozen and only the soft prompt embedding vectors are updated to optimize the performance of the model on the original prompt.

This is very efficient since a very small number of parameters are being trained (10, 000 to 100, 000). In comparison, full fine-tuning involves training millions to billions of parameters.



6.3.3. Multiple Tasks

Like **LoRA**, **soft prompts** are easily swappable. Thus, we can train different soft prompts for different tasks and swap them according to our needs during inference.



6.3.4. Interpretability of Soft Prompts

Soft prompts are not easily interpretable. Since they can take on any value within the continuous multidimensional embedding space, they do not correspond to any known tokens in the vocabulary of the model.

However, analysis of the nearest neighbors of soft prompts shows that they form tight semantic clusters. Words closest to the soft prompt tokens have similar meanings. These words usually have some meaning related to the task, suggesting that the prompts are learning word-like representations.

Interpretability of soft prompts

