Fine-tuning and evaluating large language models

Fine-tuning and evaluating large language models
- Learning Objectives
- Instruction fine-tuning
  - LLM fine-tuning at a high level
- Fine-tuning on a single task
  - Catastrophic forgetting
    - How to avoid catastrophic forgetting?
- Multi-task instruction fine-tuning
  - Instruction fine-tuning FLAN
  - Sample FLAN-T5 prompt templates
  - Improve FLAN-T5's summarisation capabilities
- Model Evaluation
  - Rouge-1
  - Rouge-2
- Rouge-L
- Bleu
- Benchmarks
  - Glue- General Language Understanding Evaluation
  - SuperGlue

Parameter Eficient fine-tuning (PEFT)
- Selective
- Reparameterization
- Additive
- LoRA
  - Example
  - Adaptation of LLMS
  - Performance of LoRA
  - How to choose the rank of LoRA matrices?
- Soft Prompts
  - Prompt Tuning is **not** Prompt Engineering
  - Prompt Tuning

Lab notes
- Full Fine Tuning
- Perform Parameter Efficient Fine-Tuning (PEFT)
  - Setup the PEFT/LoRA model for Fine-Tuning
  - Train PEFT Adapter

Learning Objectives

- Describe how fine-tuning with instructions using prompt datasets can improve performance on one or more tasks
- Define catastrophic forgetting and explain techniques that can be used to overcome it
- Define the term Parameter-efficient Fine Tuning (PEFT)
- Explain how PEFT decreases computational cost and overcomes catastrophic forgetting
- Explain how fine-tuning with instructions using prompt datasets can increase LLM performance on one or more tasks

Instruction fine-tuning

ICL have some limitations:

- In smaller models, even with one/few shots inference, the model isn't capable of performing
- Examples take up space in the context window

To overcome these limitations we can **fine-tune** the model.

LLM fine-tuning at a high level

The process is known as fine-tuning to further train a base model. In contrast to pre-training, where you train the LLM using vast amounts of unstructured textual data via self-supervised learning, fine-tuning is a supervised learning process where you use a data set of labelled examples to update the weights of the LLM. The labelled examples are prompt completion pairs, the fine-tuning process extends the training of the model to improve its ability to generate good completions for a specific task. One strategy, instruction fine-tuning, is particularly good at improving a model's performance on various tasks.

Instruction fine-tuning trains the model using examples demonstrating how it should respond to a specific instruction.

Instruction fine-tuning, where all of the model's weights are updated is known as full fine-tuning. The process results in a new version of the model with updated weights. It is important to note that just like pre-training, full fine tuning requires enough memory and compute budget to store and process all the gradients, optimizers and other components that are being updated during training. So you can benefit from the memory optimization and parallel computing strategies

The 1st step is to prepare the dataset (there are a lot of templates of prompt instruction to fine-tune models). After that the process is the same, divide the dataset in train, validation and test and use that to train/evaluate the model.

The fine-tuning process results in a new version of the base model, often called an instruct model that is better at the tasks you are interested in. Fine-tuning with instruction prompts is the most common way to fine-tune LLMs these days.

Fine-tuning on a single task

LLMs have become famous for their ability to perform many different language tasks within a single model, your application may only need to perform a single task. In this case, you can fine-tune a pre-trained model to improve performance on only the task that is of interest to you.

Catastrophic forgetting

However, there is a potential downside to fine-tuning on a single task. The process may lead to a phenomenon called ***catastrophic forgetting.*** Catastrophic forgetting happens because the full fine-tuning process modifies the weights of the original LLM. While this leads to great performance on a single fine-tuning task, it can degrade performance on other tasks. For example, while fine-tuning can improve the ability of a model to perform sentiment analysis on a review and result in a quality completion, the model may forget how to do 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(PETF)***

First of all, it's important to decide whether catastrophic forgetting impacts your use case. If all you need is reliable performance on the single task you fine-tuned on, it may not be an issue that the model can't generalize to other tasks. If you do want or need the model to maintain its multitask generalized capabilities, you can perform fine-tuning on multiple tasks at one time. **Good multitask fine-tuning may require 50-100,000 examples across many tasks,** and so will require more data and compute to train. Will discuss this option in more detail shortly. Our second option is to perform parameter efficient fine-tuning, or PEFT for short instead of full fine-tuning. **PEFT is a set of techniques that preserves the weights of the original LLM and trains only a small number of task-specific adapter layers and parameters**. **PEFT shows greater robustness to catastrophic forgetting since most of the pre-trained weights are left unchanged. PEFT is an exciting and active area of research that we will cover later this week. In the meantime, let's move on to the next video and take a closer look at multitask fine-tuning.**

Multi-task instruction fine-tuning

Multitask fine-tuning is an extension of single-task fine-tuning, where the training dataset is comprised of example inputs and outputs for multiple tasks.

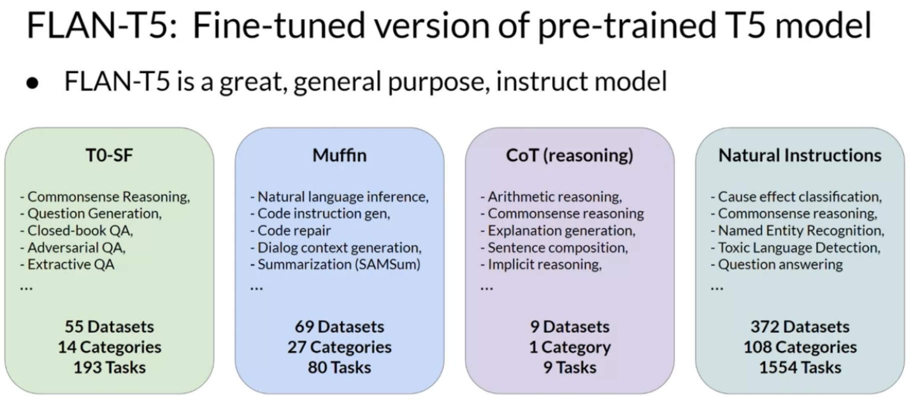
You train the model on this mixed dataset so that it can improve the performance of the model on all the tasks simultaneously, thus avoiding the issue of catastrophic forgetting. Over many epochs of training, the calculated losses across examples are used to update the weights of the model, resulting in an *instruction tuned model that is learned how to be good at many different tasks simultaneously.* One drawback to multitask fine-tuning is that it requires a lot of data. You may need as many as 50-100,000 examples in your training set.

Instruction fine-tuning FLAN

FLAN, which stands for *fine-tuned language net,* is a specific set of instructions used to fine-tune different models. Because their FLAN fine-tuning is the last step of the training process the authors of the original paper called it the metaphorical dessert to the main course of pre-training quite a fitting name. FLAN-T5, is the FLAN instruct version of the T5 foundation model while FLAN-PALM is the flattening struct version of the palm foundation model.

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

FLAN-T5 is a great general-purpose instruction model. In total, it's been fine-tuned on 473 datasets across 146 task categories.



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

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

T0-SF
- Commonsense Reasoning,
- Question Generation,
- Closed-book QA,
- Adversarial QA,
- Extractive QA
...

55 Datasets
14 Categories
193 Tasks

Muffin
- Natural language inference,
- Code instruction gen,
- Code repair
- Dialog context generation,
- Summarization (SAMSum)
...

69 Datasets
27 Categories
80 Tasks

CoT (reasoning)
- Arithmetic reasoning,
- Commonsense reasoning
- Explanation generation,
- Sentence composition,
- Implicit reasoning,
...

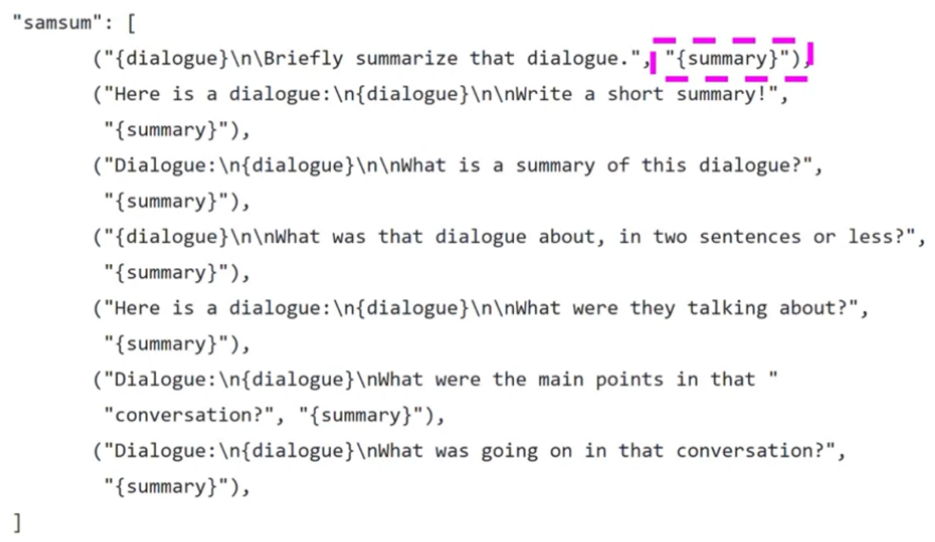
9 Datasets
1 Category
9 Tasks

Natural Instructions
- Cause effect classification,
- Commonsense reasoning,
- Named Entity Recognition,
- Toxic Language Detection,
- Question answering
...

372 Datasets
108 Categories
1554 Tasks

Sample FLAN-T5 prompt templates

The template is comprised of several different instructions that all ask the model to do the same thing. Summarize a dialogue. For example, briefly summarize that dialogue. What is a summary of this dialogue? What was going on in that conversation? Including different ways of saying the same instruction helps the model generalize and perform better. The summary is used as the label. After applying this template to each row in the SAMSum dataset, you can use it to fine-tune a dialogue summarization task.



```
"samsum": [
        ("{dialogue}\n\Briefly summarize that dialogue.", "{summary}"),
        ("Here is a dialogue:\n{dialogue}\n\nWrite a short summary!",
         "{summary}"),
        ("Dialogue:\n{dialogue}\n\nWhat is a summary of this dialogue?",
         "{summary}"),
        ("{dialogue}\n\nWhat was that dialogue about, in two sentences or less?",
         "{summary}"),
        ("Here is a dialogue:\n{dialogue}\n\nWhat were they talking about?",
         "{summary}"),
        ("Dialogue:\n{dialogue}\nWhat were the main points in that "
         "conversation?", "{summary}"),
        ("Dialogue:\n{dialogue}\nWhat was going on in that conversation?",
         "{summary}"),
]
```

Improve FLAN-T5’s summarisation capabilities

You can perform additional fine-tuning of the FLAN-T5 model using a dialogue dataset that is much closer to the conversations that happened with your bot. This is the exact scenario that you'll explore in the lab this week. You'll make use of an additional domain-specific summarization dataset called dialogsum to improve FLAN-T5's is ability to summarize support chat conversations.

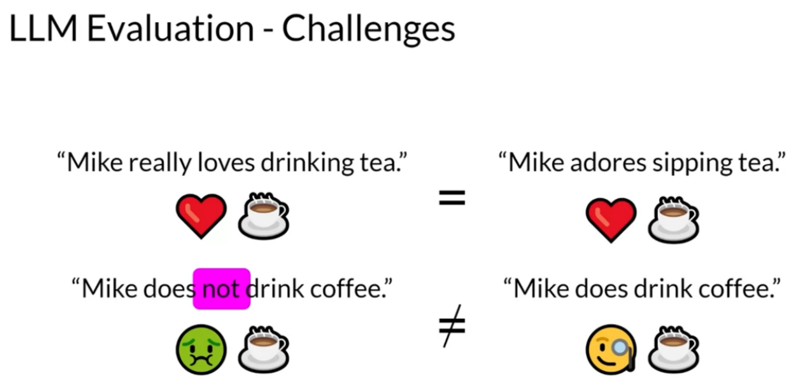
In practice, you'll get the most out of fine-tuning by using your company's own internal data. For example, the support chat conversations from your customer support application. This will help the model learn the specifics of how your company likes to summarize conversations and what is most useful to your customer service colleagues.

This paper introduces FLAN (Fine-tuned LAnguage Net), an instruction finetuning method, and presents the results of its application. The study demonstrates that by fine-tuning the 540B PaLM model on 1836 tasks while incorporating Chain-of-Thought Reasoning data, FLAN achieves improvements in generalization, human usability, and zero-shot reasoning over the base model. The paper also provides detailed information on how each of these aspects was evaluated.

Here is the image from the lecture slides that illustrate the fine-tuning tasks and datasets employed in training FLAN. The task selection expands on previous works by incorporating dialogue and program synthesis tasks from Muffin and integrating them with new Chain of Thought Reasoning tasks. It also includes subsets of other task collections, such as T0 and Natural Instructions v2. Some tasks were held out during training, and they were later used to evaluate the model's performance on unseen tasks.

Model Evaluation

In traditional machine learning the models are deterministic, so it's easier to measure the model's performance.

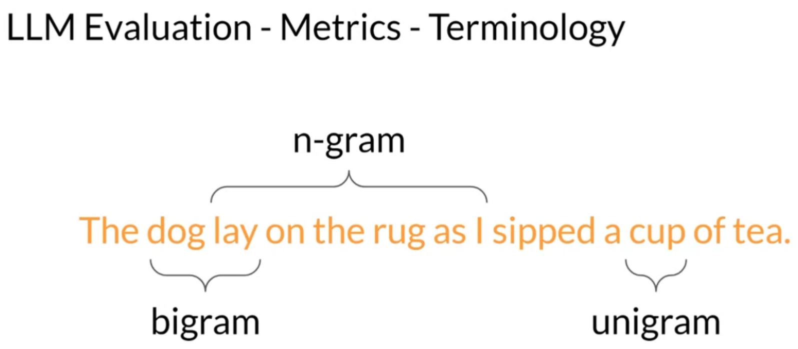

LLM Evaluation - Challenges
"Mike really loves drinking tea."
=
"Mike adores sipping tea."
"Mike does not drink coffee."
≠
"Mike does drink coffee."

ROUGE and ***BLEU***, are two widely used evaluation metrics for different tasks.
ROUGE or recall oriented under study for jesting evaluation is primarily employed to assess the quality of automatically generated summaries by comparing them to human-generated reference summaries.

- Used for text summarization
- Compared a summary to one or more reference summaries

BLEU, or bilingual evaluation understudy is an algorithm designed to evaluate the quality of machine-translated text, again, by comparing it to human-generated translations. Now the word BLEU is French for blue.

- Used for text translation
- Compares to human-generated translations


LLM Evaluation - Metrics - Terminology
n-gram
The dog lay on the rug as I sipped a cup of tea.
bigram
unigram

-> unigram is equivalent to a single word.
->bigram is two words
->n-gram is a group of n-words.

Reference human:

```
It is cold outside
```

Generated output:

```
It is very cold outside
```

Rouge-1

$$ROUGE-1_{recall} = \frac{unigram\_matches}{unigrams\_in\_reference} = \frac{4}{4} = 1$$

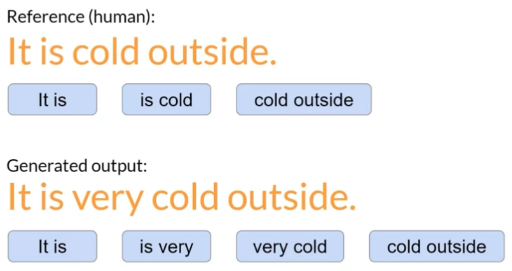
$$ROUGE-1_{precision} = \frac{unigram\_matches}{unigrams\_in\_output} = \frac{4}{5} = 0.8$$

$$ROUGE-1_{f1} = \frac{precision \times recal}{precision + recal} = \frac{0.8}{1.8} = 0.89$$

Let's look at the ROUGE-1 metric. To do so, let's look at a human-generated reference sentence. It is cold outside and a generated output that is very cold outside. You can perform simple metric calculations similar to other machine-learning tasks using recall, precision, and F1. The recall metric measures the number of words or unigrams that are matched between the reference and the generated output divided by the number of words or unigrams in the reference. In this case, that gets a perfect score of one as all the generated words match the words in the reference. Precision measures the unigram matches divided by the output size. The F1 score is the harmonic mean of both of these values. These are very basic metrics that only focus on individual words, hence the one in the name, and don't consider the ordering of the words. It can be deceptive. It's easily possible to generate sentences that score well but would be subjectively poor.

Stop for a moment and imagine that the sentence generated by the model was different by just one word. ***Not****, so it is not cold outside. The scores would be the same. You can get a slightly better score by taking into account bigrams or collections of two words at a time from the reference and generated sentence.*

Rouge-2



By working with pairs of words you're acknowledging in a very simple way, the ordering of the words in the sentence. By using bigrams, you're able to calculate a ROUGE-2.

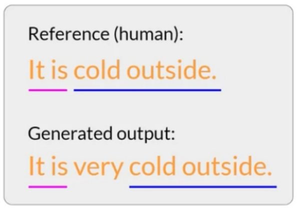
$$ROUGE-2_{recall} = \frac{unigram\_matches}{unigrams\_in\_reference} = \frac{2}{3} = 0.67$$

$$ROUGE-2_{precision} = \frac{unigram\_matches}{unigrams\_in\_output} = \frac{2}{4} = 0.5$$

$$ROUGE-2_{f1} = \frac{precision \times recal}{precision + recal} = \frac{0.335}{1.15} = 0.57$$

Notice that the scores are lower than the ROUGE-1 scores. With longer sentences, they're a greater chance that bigrams don't match, and the scores may be even lower.

Rouge-L


Reference (human):

It is cold outside.

Generated output:

It is very cold outside.

Rather than continue with ROUGE numbers growing bigger to n-grams of three or fours, let's take a different approach. Instead, you'll look for the longest common subsequence present in both the generated output and the reference output. In this case, the longest matching sub-sequences are, it is and cold outside, each with a length of two. You can now use the LCS value to calculate the recall precision and F1 score, where the numerator in both the recall and precision calculations is the length of the longest common subsequence, in this case, two. Collectively, these three quantities are known as the Rouge-L score.

$$ROUGE - L_{recall} = \frac{LCS(Gen,Ref)}{unigrams\_in\_reference} = \frac{2}{4} = 0.5$$

$$ROUGE - L_{precision} = \frac{LCS(Gen,Ref)}{unigrams\_in\_output} = \frac{2}{5} = 0.4$$

$$ROUGE - L_{f1} = \frac{precision \times recal}{precision + recal} = \frac{0.2}{0.9} = 0.44$$

Collectively, these three quantities are known as the Rouge-L score.

As with all of the rouge scores, you need to take the values in context. You can only use the scores to compare the capabilities of models if the scores were determined for the same task. For example, summarization. Rouge scores for different tasks are not comparable to one another. As you've seen, a particular problem with simple rouge scores is that a bad completion can result in a good score.

Reference human:

```
It is cold outside
```

Generated output:

```
cold cold cold cold
```

$$ROUGE - 1_{recall} = \frac{unigram\_matches}{unigrams\_in\_reference} = \frac{4}{4} = 1$$ 🥶

One way you can counter this issue is by using a clipping function to limit the number of unigram matches to the maximum count for that unigram within the reference. In this case, there is one appearance of cold and the reference and so a modified precision with a clip on the unigram matches results in a dramatically reduced score.

$$ROUGE - 1_{recall} = \frac{clip(unigram\_matches)}{unigrams\_in\_reference} = \frac{1}{4} = 0.25$$

However, you'll still be challenged if their generated words are all present, but ***just in a different order.***

Generated output:

```
outside cold it is
```

$ROUGE - 1_{recall} = \frac{clip(unigram\_matches)}{unigrams\_in\_reference} = \frac{4}{4} = 1$🤔

Using a different rouge score can help experimenting with a n-gram size that will calculate the most useful score will be dependent on the sentence, the sentence size, and your use case.

Bleu

Metric -> Average(precision range across range of n-gram sizes)

The other score that can be useful in evaluating the performance of your model is the ***BLEU*** score, which stands for bilingual evaluation under study. Just to remind you that *BLEU score is useful for evaluating the quality of machine-translated text.*

The score itself is calculated using the average precision over multiple n-gram sizes. Just like the Rouge-1 score that we looked at before, but calculated for a range of n-gram sizes and then averaged. The BLEU score quantifies the quality of a translation by checking how many n-grams in the machine-generated translation match those in the reference translation. To calculate the score, you average precision across a range of different n-gram sizes. If you were to calculate this by hand, you would carry out multiple calculations and then average all of the results to find the BLEU score.

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. -> The BLEU score is 0.495
I am very happy that I am drinking a warm cup of tea. -> The BLEU score is 0.730
I am very happy to say that I am drinking a warm cup of tea. -> The BLEU score is 0.798
```

Both rouge and BLEU are quite simple metrics and are relatively low-cost to calculate. You can use them for simple reference as you iterate over your models, but you shouldn't use them alone to report the final evaluation of a large language model. Use rouge for diagnostic evaluation of summarization tasks and BLEU for translation tasks.

For overall evaluation of your model's performance, however, you will need to look at one of the evaluation ***benchmarks*** that have been developed by researchers.

Benchmarks

In order to measure and compare LLMs more holistically, ***you can make use of pre-existing datasets, and associated benchmarks that have been established by LLM researchers specifically for this purpose***. Selecting the ***right evaluation dataset is vital,*** so that you can accurately assess an LLM's performance, and understand its true capabilities. You'll find it useful to select datasets that isolate specific model skills, like reasoning or common sense knowledge, and those that focus on potential risks, such as disinformation or copyright infringement. *An important issue that you should consider is whether the model has seen your evaluation data during training.* You'll get a more accurate and useful sense of the model's capabilities by evaluating its performance on data that it hasn't seen before. Benchmarks, such as GLUE, SuperGLUE, or Helm, cover a wide range of tasks and scenarios.

Glue- General Language Understanding Evaluation

GLUE is a collection of natural language tasks, such as sentiment analysis and question-answering. GLUE was created to encourage the development of models that can generalize across multiple tasks, and you can use the benchmark to measure and compare the model performance.

SuperGlue

As a successor to GLUE, SuperGLUE was introduced in 2019, to address limitations in its predecessor. It consists of a series of tasks, some of which are not included in GLUE, and some of which are more challenging versions of the same tasks. SuperGLUE includes tasks such as multi-sentence reasoning, and reading comprehension.

Both the GLUE and SuperGLUE benchmarks have leaderboards that can be used to compare and contrast evaluated models.

As models get larger, their performance against benchmarks such as SuperGLUE start to match human ability on specific tasks.That's to say that models are able to perform as well as humans on the benchmarks tests, but subjectively we can see that they're not performing at human level at tasks in general.

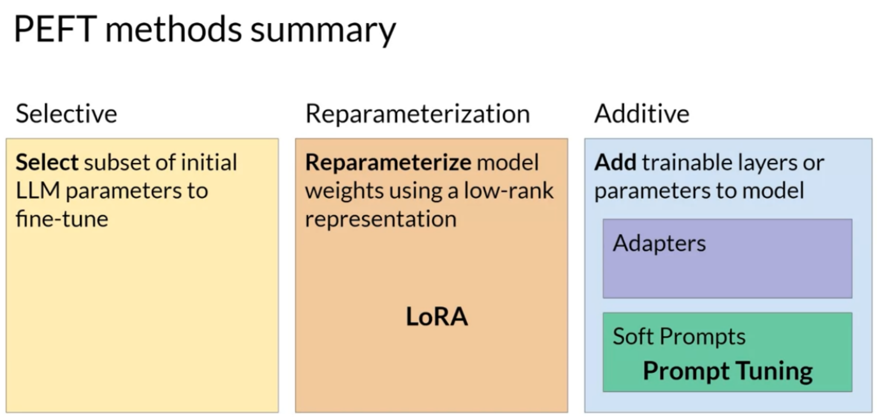
There is essentially an arms race between the emergent properties of LLMs, and the benchmarks that aim to measure them.

Here are a couple of recent benchmarks that are pushing LLMs further:

- **Massive Multitask Language Understanding,** or ***MMLU***, is designed specifically for modern LLMs. Models are tested on elementary mathematics, US history, computer science, law, and more. In other words, tasks that extend way beyond basic language understanding.
- **BIG-bench** currently consists of 204 tasks, ranging through linguistics, childhood development, math, common sense reasoning, biology, physics, social bias, software development and more. *BIG-bench comes in three different sizes,* and part of the reason for this is to keep costs achievable, as running these large benchmarks can incur large inference costs.

- **Holistic Evaluation of Language Models, or HELM**. The HELM framework aims to improve the transparency of models, and to offer guidance on which models perform well for specific tasks. HELM takes a multimetric approach, measuring seven metrics across 16 core scenarios, ensuring that trade-offs between models and metrics are clearly exposed. One important feature of HELM is that it assesses on metrics beyond basic accuracy measures, like precision of the F1 score. *The benchmark also includes metrics for fairness, bias, and toxicity, which are becoming increasingly important to assess as LLMs become more capable of human-like language generation*, and in turn of exhibiting potentially harmful behavior. HELM is a living benchmark that aims to continuously evolve with the addition of new scenarios, metrics, and models.

Parameter Eficient fine-tuning (PEFT)


PEFT methods summary
Selective
Reparameterization
Additive
Select subset of initial LLM parameters to fine-tune
Reparameterize model weights using a low-rank representation
LoRA
Add trainable layers or parameters to model
Adapters
Soft Prompts
Prompt Tuning

In contrast to full fine-tuning where every model weight is updated during supervised learning, parameter efficient fine tuning methods only update a small subset of parameters. Some path techniques freeze most of the model weights and focus on fine tuning a subset of existing model parameters, for example, particular layers or components. Other techniques don't touch the original model weights at all, and instead add a small number of new parameters or layers and fine-tune only the new components.

With PEFT, most if not all of the LLM weights are kept frozen. As a result, the number of trained parameters is much smaller than those in the original LLM. In some cases, just 15-20% of the original LLM weights. This makes the memory requirements for training much more manageable. PEFT can often be performed on a single GPU. And because the original LLM is only slightly modified or left unchanged, ***PEFT is less prone to the catastrophic forgetting problems of full fine-tuning.***

Full fine-tuning results in a new version of the model for every task you train on. *Each of these is the same size as the original model*, so it can create an *expensive storage problem* if you're fine-tuning for multiple tasks.

With ***parameter-efficient fine-tuning,*** you *train only a small number of weights*, which results in a much smaller footprint overall, as *small as megabytes* depending on the task. The new parameters are combined with the original LLM weights for inference. *The PEFT weights are trained for each task and can be easily swapped out for inference, allowing efficient adaptation of the original model to multiple tasks.*

There are several methods you can use for parameter efficient fine-tuning, each with trade-offs on parameter efficiency, memory efficiency, training speed, model quality, and inference costs.

Selective

Select subset of initial LLM parameters to fine-tune

Selective methods fine-tune only a subset of the original LLM parameters. There are several approaches that you can take to identify which parameters you want to update. You have the option to train only certain components of the model or specific layers, or even individual parameter types. Researchers have found that the performance of these methods is mixed and there are significant trade-offs between parameter efficiency and compute efficiency.

Reparameterization

Reparameterize model weights using a low rank representation (LoRA)

Reparameterization methods also work with the original LLM parameters, but reduce the number of parameters to train by creating new low-rank transformations of the original network weights. A commonly used technique of this type is LoRA,

Additive

Add trainable layers parameters

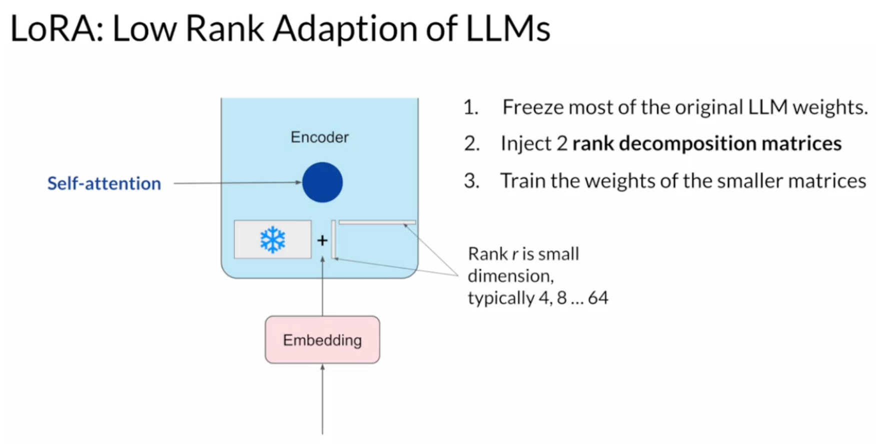
Additive methods carry out fine-tuning by keeping all original LLM weights frozen and introducing new trainable components. Here there are two main approaches. Adapter methods add new trainable layers to the model's architecture, typically inside the encoder or decoder components after the attention or feed-forward layers. Soft prompt methods, on the other hand, keep the model architecture fixed and frozen, and focus on manipulating the input to achieve better performance. This can be done by adding trainable parameters to the prompt embeddings or keeping the input fixed and retraining the embedding weights.

LoRA

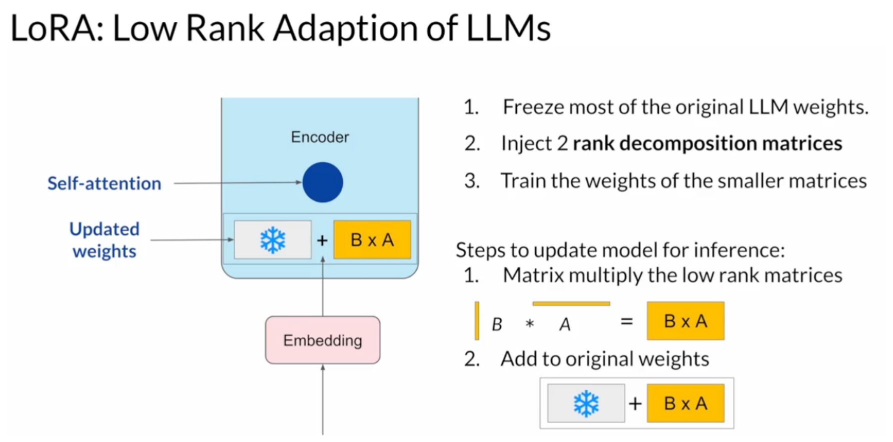
On the Transformer architecture, the input prompt is turned into tokens, which are then converted to embedding vectors and passed into the encoder and/or decoder parts of the transformer. In both of these components, there are two kinds of neural networks; self-attention and feedforward networks. The weights of these networks are learned during pre-training. After the embedding vectors are created, they're fed into the self-attention layers where a series of weights are applied to calculate the attention scores. During full fine-tuning, every parameter in these layers is updated.

LoRA is a strategy that reduces the number of parameters to be trained during fine-tuning by freezing all of the original model parameters and then injecting a pair of rank decomposition matrices alongside the original weights.The dimensions of the smaller matrices are set so that their product is a matrix with the same dimensions as the weights they're modifying. You then keep the original weights of the LLM frozen and train the smaller matrices using the same supervised learning process.

- Freeze most of the original LLM weights
- Inject 2 rank decomposition matrices
- Train the weights of the smaller matrices


LoRA: Low Rank Adaption of LLMs
Encoder
Self-attention
+
Embedding
1. Freeze most of the original LLM weights.
2. Inject 2 rank decomposition matrices
3. Train the weights of the smaller matrices
Rank r is small dimension, typically 4, 8 ... 64

For inference, the two low-rank matrices are multiplied together to create a matrix with the same dimensions as the frozen weights. You then add this to the original weights and replace them in the model with these updated values. You now have a LoRA fine-tuned model that can carry out your specific task.


LoRA: Low Rank Adaption of LLMs
Encoder
Self-attention
Updated weights
+
B x A
Embedding
1. Freeze most of the original LLM weights.
2. Inject 2 rank decomposition matrices
3. Train the weights of the smaller matrices
Steps to update model for inference:
1. Matrix multiply the low rank matrices
B * A = B x A
2. Add to original weights
+ B x A

Because this model has the same number of parameters as the original, there is little to no impact on inference latency. Researchers have found that applying LoRA to just the self-attention layers of the model is often enough to fine-tune for a task and achieve performance gains. However, in principle, you can also use LoRA on other components like the feed-forward layers. But since most of the parameters of LLMs are in the attention layers, you get the biggest savings in trainable parameters by applying LoRA to these weights matrices.

Example

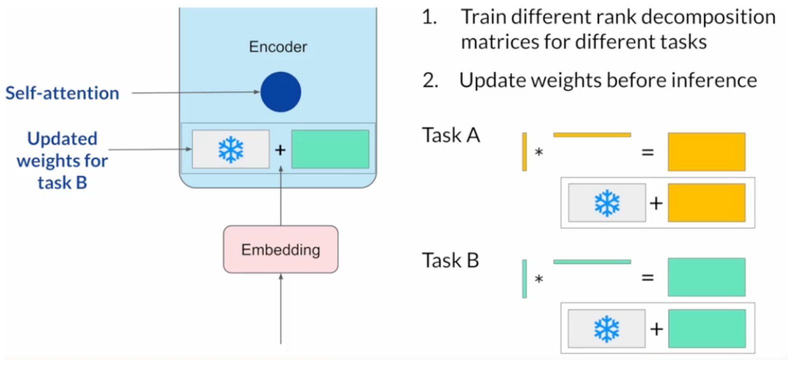
Using the base Transformer model (in the Attention is all you need paaper):

- Transformer weights have dimensions $d \times k = 512 \times 64$
- So $512 \times 64 = 32768$ trainable parameters

In ***LoRA*** with $r = 8$:

- Matrix A has dimension $r \times k = \mathbf{8} \times 64 = 512$ parameters
- Matrix B has dimension $d \times r = 512 \times \mathbf{8} = 4096$ trainable parameters
- **86% reduction in parameters to train**

Adaptation of LLMS


Encoder
Self-attention
Updated weights for task B
Embedding
1. Train different rank decomposition matrices for different tasks
2. Update weights before inference
Task A
Task B

Because LoRA allows you to significantly reduce the number of trainable parameters, you can often perform this method of parameter efficient fine tuning with a single GPU and avoid the need for a distributed cluster of GPUs. Since the rank-decomposition matrices are small, you can fine-tune a different set for each task and then switch them out at inference time by updating the weights. Suppose you train a pair of LoRA matrices for a specific task; let's call it Task A. To carry out inference on this task, you would multiply these matrices together and then add the resulting matrix to the original frozen weights. You then take this new summed weights matrix and replace the original weights where they appear in your model. You can then use this model to carry out inference on Task A. If instead, you want to carry out a different task, say Task B, you simply take the LoRA matrices you trained for this task, calculate their product, and then add this matrix to the original weights and update the model again. The memory required to store these LoRA matrices is very small. So in principle, you can use LoRA to train for many tasks. Switch out the weights when you need to use them, and avoid having to store multiple full-size versions of the LLM.

Performance of LoRA

Fine-tuning the FLAN-T5 for dialogue summarization(ROUGE Metrics):

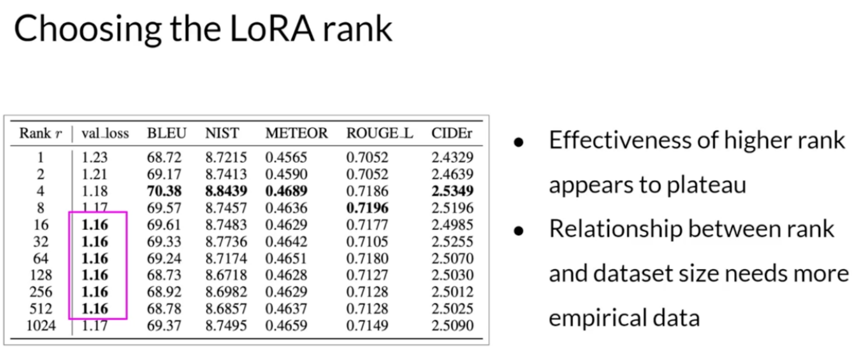
- Full Fine Tuning ***+80.63%***

- LoRA Fine Tuning only ***-3.20%*** compared with full fine tuning

Using LoRA for fine-tuning trained a much smaller number of parameters than full fine-tuning using significantly less computing-, so this small trade-off in performance may well be worth it.

How to choose the rank of LoRA matrices?

This is a good question and still an active area of research. In principle, the smaller the rank, the smaller the number of trainable parameters, and the bigger the savings on compute. However, there are some issues related to model performance to consider.



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

In the paper that first proposed LoRA, researchers at Microsoft explored how different choices of rank impacted the model performance on language generation tasks. You can see the summary of the results in the table here. The table shows the rank of the LoRA matrices in the first column, the final loss value of the model, and the scores for different metrics, including BLEU and ROUGE. The bold values indicate the best scores that were achieved for each metric. The authors found a plateau in the loss value for ranks greater than 16. In other words, using larger LoRA matrices didn't improve performance. The takeaway here is that ranks in the range of 4-32 can provide you with a good trade-off between reducing trainable parameters and preserving performance.
Optimizing the choice of rank is an ongoing area of research and best practices may evolve as more practitioners make use of LoRA. Lora is a powerful fine-tuning method that achieves great performance.

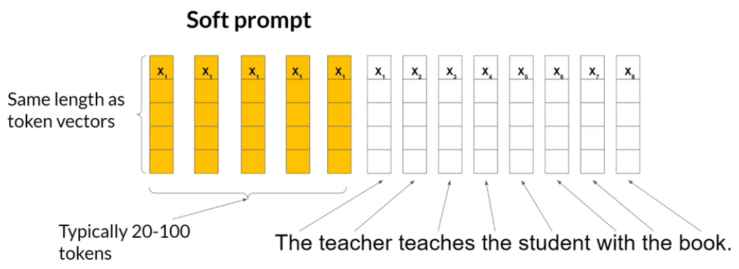
Soft Prompts

Prompt Tuning is not Prompt Engineering

Prompt tuning sounds a bit like prompt engineering, but they are quite different from each other. With prompt engineering, you work on the language of your prompt to get the completion you want. This could be as simple as trying different words or phrases or more complex, like including examples for one or Few-shot Inference. The goal is to help the model understand the nature of the task you're asking it to carry out and to generate a better completion. However, there are some limitations to prompt engineering, as it can require a lot of manual effort to write and try different prompts. You're also limited by the length of the context window, and at the end of the day, you may still not achieve the performance you need for your task.

Prompt Tuning

With prompt tuning, you add additional trainable tokens to your prompt and leave it up to the supervised learning process to determine their optimal values. The set of trainable tokens is called a soft prompt, and it gets prepended to embedding vectors that represent your input text. The soft prompt vectors have the same length as the embedding vectors of the language tokens. And including somewhere between 20 and 100 virtual tokens can be sufficient for good performance.


Soft prompt
Same length as token vectors
Typically 20-100 tokens
The teacher teaches the student with the book.

The tokens that represent natural language are hard in the sense that they each correspond to a fixed location in the embedding vector space. However, the ***soft prompts are not fixed discrete words of natural language. Instead, you can think of them as virtual tokens that can take on any value within the continuous multidimensional embedding space.*** And through supervised learning, the model learns the values for these virtual tokens that maximize performance for a given task.
With prompt tuning, the weights of the large language model are frozen and the underlying model does not get updated. Instead, the *embedding vectors of the soft prompt gets updated over time to optimize the model's completion of the prompt*. Prompt tuning is a very parameter efficient strategy because only a few parameters are being trained.

Similar to what you saw with LoRA. You can train a different set of soft prompts for each task and then easily swap them out at inference time. You can train a set of soft prompts for one task and a different set for another.

Soft prompts are very small on disk, so this kind of fine tuning is extremely efficient and flexible. You'll notice the same LLM is used for all tasks, all you have to do is switch out the soft prompts at inference time.

Prompt tuning doesn't perform as well as full fine tuning for smaller LLMs. However, as the model size increases, so does the performance of prompt tuning. And once models have around 10 billion parameters, prompt tuning can be as effective as full fine tuning and offers a significant boost in performance over prompt engineering alone.

One potential issue to consider is the interpretability of learned virtual tokens. Remember, because the soft prompt tokens can take any value within the continuous embedding vector space. The trained tokens don't correspond to any known token, word, or phrase in the vocabulary of the LLM.

However, an analysis of the nearest neighbor tokens to the soft prompt location shows that they form tight semantic clusters. In other words, the words closest to the soft prompt tokens have similar meanings. The words identified usually have some meaning related to the task, suggesting that the prompts are learning word like representations.

By the way you can also combine LoRA with the quantization techniques you learned about in week 1 to further reduce your memory footprint. This is known as QLoRA in practice, PEFT is used heavily to minimize computing and memory resources.

Lab notes

Check the number of parameters

```
def print_number_of_trainable_model_parameters(model):
    trainable_model_params = 0
    all_model_params = 0
    for _, param in model.named_parameters():
        all_model_params += param.numel() #number of elements
        if param.requires_grad:
            trainable_model_params += param.numel()
    return f"trainable model parameters: {trainable_model_params}\nall model parameters:
{all_model_params}\npercentage of trainable model parameters: {100 * trainable_model_params /
all_model_params:.2f}%"

print(print_number_of_trainable_model_parameters(original_model))
```

Full Fine Tuning

- convert the dialog-summary (prompt-response) pairs into explicit instructions for the LLM
    - Training prompt (dialogue):

      ```
      Summarize the following conversation.


          Chris: This is his part of the conversation.
          Antje: This is her part of the conversation.


      Summary:
      ```

      Training response (summary):

      ```
      Both Chris and Antje participated in the conversation.
      ```

- Utilize the built-in Hugging Face `Trainer` class (see the documentation here). Pass the preprocessed dataset concerning the original model.

```
output_dir = f'./dialogue-summary-training-{str(int(time.time()))}'

training_args = TrainingArguments(
    output_dir=output_dir,
    learning_rate=1e-5,
    num_train_epochs=1, ## this is too low, but they have another model trained
    weight_decay=0.01,
    logging_steps=1,
    max_steps=1 ## this is too low, but they have another model trained
)

trainer = Trainer(
    model=original_model,
    args=training_args,
    train_dataset=tokenized_datasets['train'],
    eval_dataset=tokenized_datasets['validation']
)

trainer.train()
```

- Evaluate the Model Qualitatively - Human Evaluation
- Evaluate the Model Quantitatively -> with ROUGE Metric

```
original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

instruct_model_results = rouge.compute(
    predictions=instruct_model_summaries,
    references=human_baseline_summaries[0:len(instruct_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('INSTRUCT MODEL:')
print(instruct_model_results)
```

Perform Parameter Efficient Fine-Tuning (PEFT)

Setup the PEFT/LoRA model for Fine-Tuning

You need to set up the PEFT/LoRA model for fine-tuning with a new layer/parameter adapter. Using PEFT/LoRA, you are freezing the underlying LLM and only training the adapter. Have a look at the LoRA configuration below. Note the rank (`r`) hyper-parameter, which defines the rank/dimension of the adapter to be trained.

```
from peft import LoraConfig, get_peft_model, TaskType

lora_config = LoraConfig(
    r=32, # Rank
    lora_alpha=32,
    target_modules=["q", "v"], ##specification to be just the attention layers?
    lora_dropout=0.05,
    bias="none",
    task_type=TaskType.SEQ_2_SEQ_LM # FLAN-T5
)

peft_model = get_peft_model(original_model,
                            lora_config)
print(print_number_of_trainable_model_parameters(peft_model))

#trainable model parameters: 3538944
#all model parameters: 251116800
# percentage of trainable model parameters: 1.41%
```

Train PEFT Adapter

```
output_dir = f'./peft-dialogue-summary-training-{str(int(time.time()))}'

peft_training_args = TrainingArguments(
    output_dir=output_dir,
    auto_find_batch_size=True,
    learning_rate=1e-3, # Higher learning rate than full fine-tuning.
    num_train_epochs=1,
    logging_steps=1,
    max_steps=1
)

peft_trainer = Trainer(
    model=peft_model,
    args=peft_training_args,
    train_dataset=tokenized_datasets["train"],
)

peft_trainer.train() # after that they save and load
```

Prepare this model by adding an adapter to the original FLAN-T5 model. You are setting `is_trainable=False` because the plan is only to perform inference with this PEFT model. If you were preparing the model for further training, you would set `is_trainable=True`.

```
from peft import PeftModel, PeftConfig

peft_model_base = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base",
torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")

peft_model = PeftModel.from_pretrained(peft_model_base,
                                       './peft-dialogue-summary-checkpoint-from-s3/',
#peft_config
                                       torch_dtype=torch.bfloat16,
                                       is_trainable=False)
```

We load the base FLAN-T5 Model and the adapter. At inference time, the LoRA adapter must be reunited and combined with its original LLM to serve the inference request. The benefit, however, is that many LoRA adapters can re-use the original LLM which reduces overall memory requirements when serving multiple tasks and use cases.