Coursera: Generative AI with Large Language Models

Table of Contents

Week 1 Part I. Introduction to LLMs and the generative AI project lifecycle 1
LLM Use cases: 1
History of generative AI: 1
Transformer: 2
Prompting and prompt engineering: 5
Generative configuration: influence the model's output during inference. 6
Generative AI project Lifecycle 7
Week 1 Part II. LLM pre-training and scaling laws 7
Week 2 Part I. Fine-tuning LLMs with instructions 7
Instruction fine-tuning 7
Fine-tuning on a single task 9
Multi-task instruction fine-tuning: 10
Model evaluation 11
Benchmark 13
Week 2 Part II. Parameter-efficient fine tuning (PEFT) 14
Introduction of PEFT 14
Week 3 Part I. Reinforcement learning with human feedback 15
Week 3 Part II. LLM-powered applications 15

Week 1 Part I. Introduction to LLMs and the generative AI project lifecycle

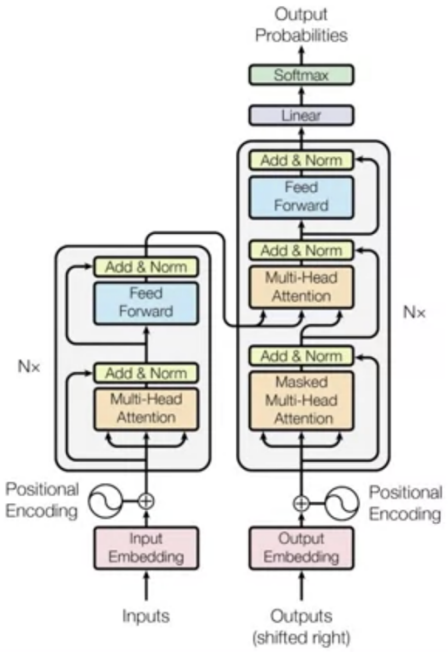
LLM Use cases:

- Basic chatbots: built upon next word generation
- Text summarization
- Language translation - can also be from natural language to machine code
- Entity Extraction (under information retrieval)
- Augmenting LLMs by invoking external APIs from them

History of generative AI:

1. Weakness of RNN: Generating text with RNN: we need to scale the resources the model use to get as many texts as possible to make reliable prediction. Models needs to have an understanding of the whole sentence or even the whole document. The problem here is that language is complex. In many languages, one word can have multiple meanings. These are homonyms.

2. Transformer:
    a. It can be scaled efficiently to use multi-core GPUs,
    b. It can parallel process input data, making use of much larger training datasets
    c. Crucially, it's able to learn to pay attention to the meaning of the words it's processing.
    d. The flowchart below is the transformer architect from the paper, Attention is All You Need.



Transformer:

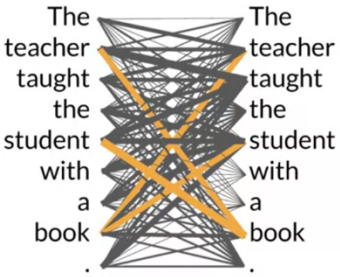
Steps in Transformer:

- Step 1: Using a tokenizer to convert text to numbers (using vocabulary indexing / stemming/ lemmatising)
- Step 2: Create embedding vectors
- Step 3: Positional encoding - as tokens are processed in parallel, add positional encoding to preserve info about word order
- Step 4: Self-Attention layer - analyze relationships between tokens
    - Multi-headed self-attention: Multiple sets of self-attention weights are learnt in parallel (no of heads varies, 12-100 common)
    - Each head learns a diff aspect of the language
- Step 5: Feed forward network- get an output vector of logits (raw probability scores) with values proportional to the probability of each token
- Step 6: Softmax layer - normalize logits to probability score
- Step 7: Find most probable next token (this step can be varied based on problem)

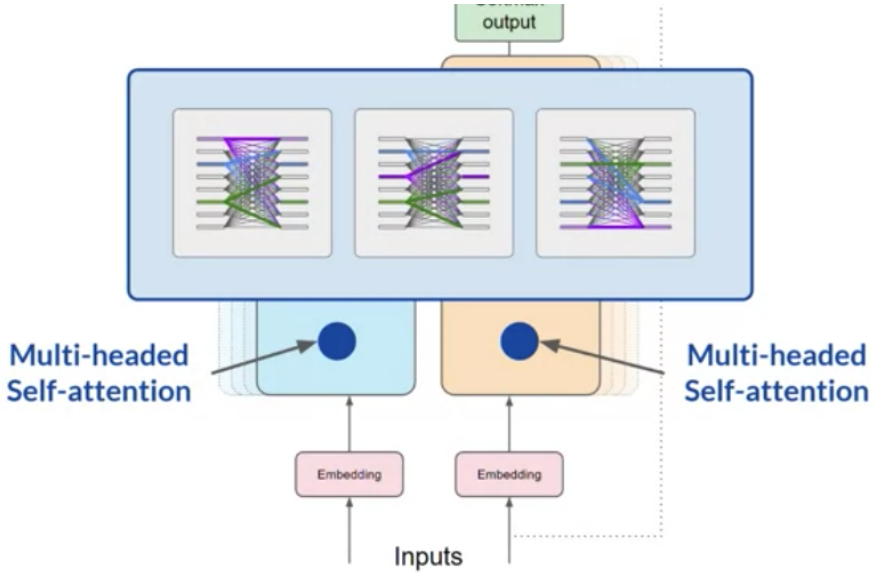
Detailed illustration of each component in Transformer:

1) Attention map: Learn the relevance of each word.

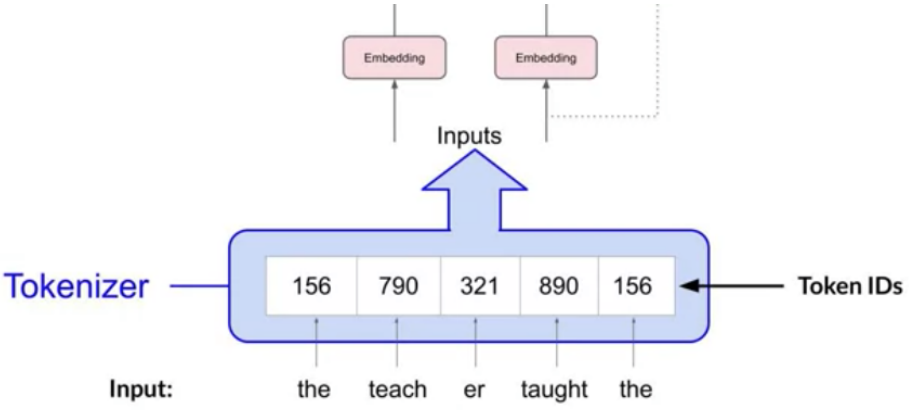
Self-attention



Multi-head self-attention



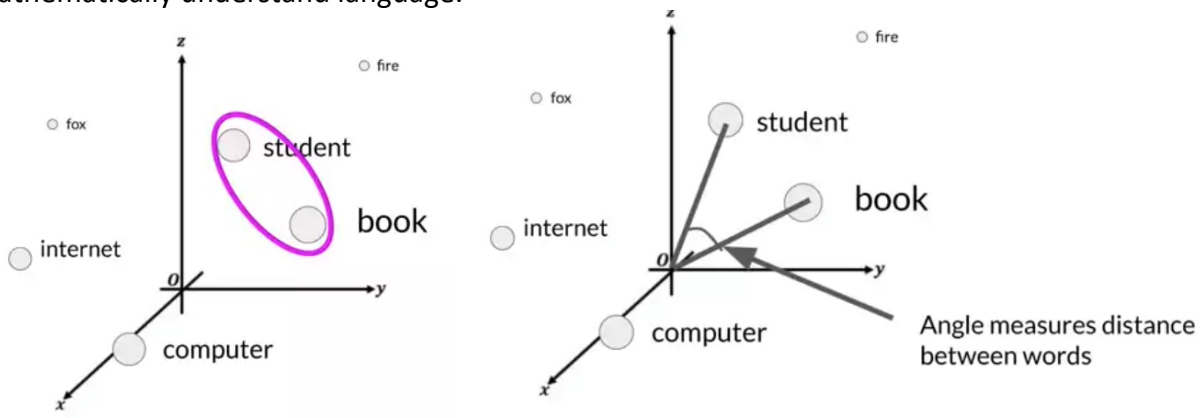
2) Tokenizer:



3) Embedding space:

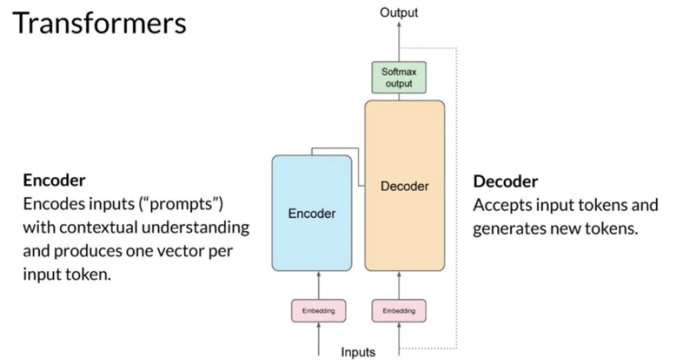
For simplicity, if you imagine a vector size of just three, you could plot the words into a three-dimensional space and see the relationships between those words. You can see now how you can relate words that are located close to each other in the embedding space, and how you can

calculate the distance between the words as an angle, which gives the model the ability to mathematically understand language.



4) Encoder and decoder:

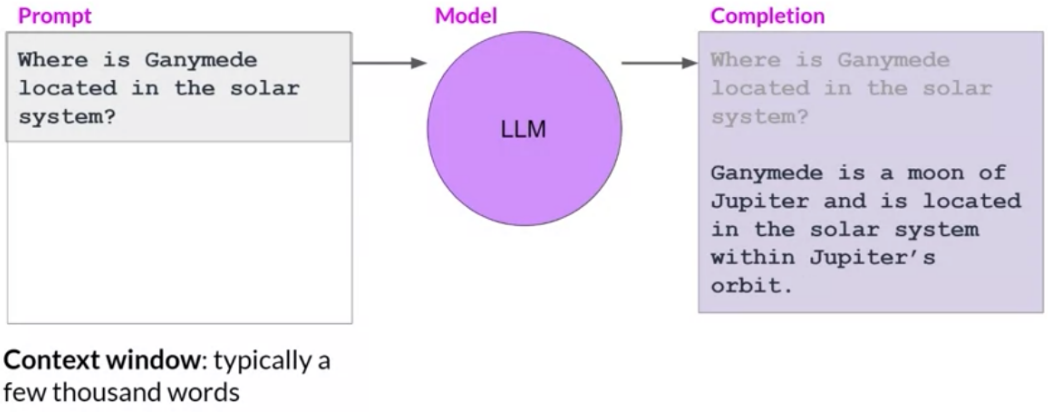
The encoder encodes input sequences into a deep representation of the structure and meaning of the input. The decoder, working from input token triggers, uses the encoder's contextual understanding to generate new tokens. It does this in a loop until some stop condition has been reached.

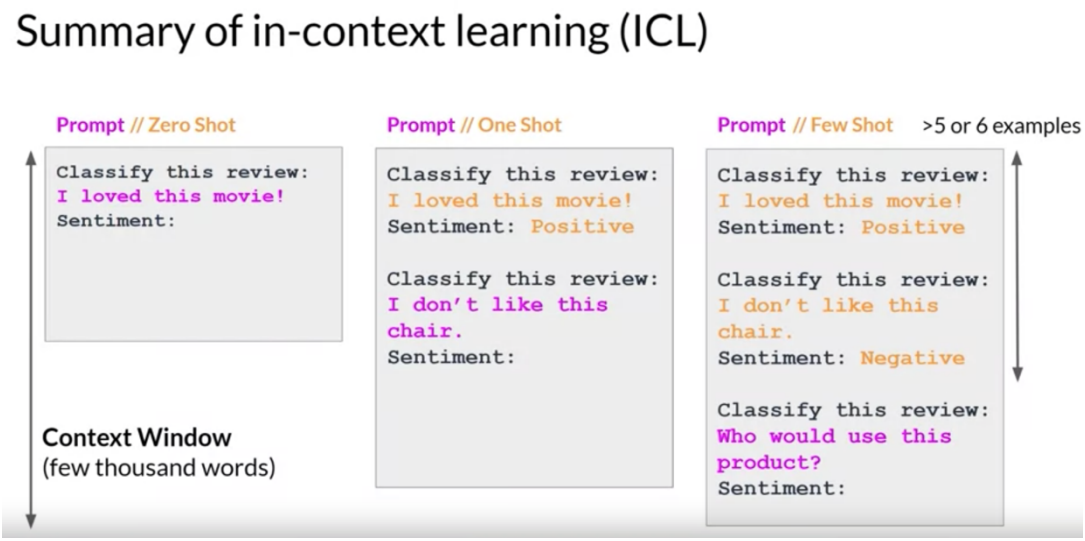


Three types of transformer models:

- Encoder only models: work as sequence-to-sequence models, but without further modification, the input sequence and the output sequence of the same length. Their use is less common these days, but by adding additional layers to the architecture, you can train encoder-only models to perform classification tasks such as sentiment analysis, BERT is an example of an encoder-only model.
- Encoder Decorder models: perform well on sequence-to-sequence tasks such as translation, where the input sequence and the output sequence can be different lengths. You can also scale and train this type of model to perform general text generation tasks.
- Decoder only models: most-commonly used. These models can now generalize to most tasks. Popular decoder-only models include the GPT family of models, BLOOM, Jurassic, LLaMA, and many more.

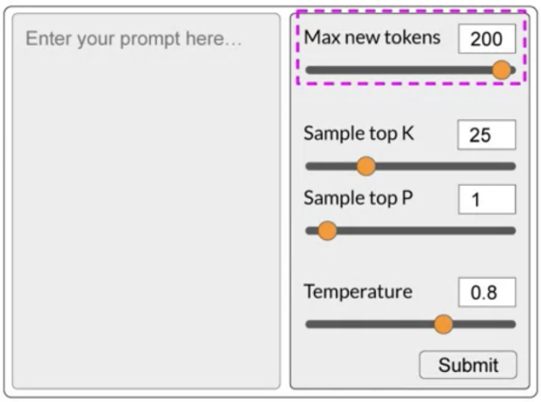
Prompting and prompt engineering:



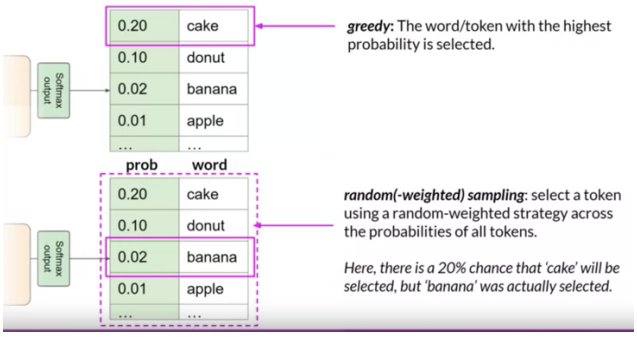


- The largest models are surprisingly good at zero-shot inference and are able to infer and successfully complete many tasks that they were not specifically trained to perform. e.g. BLOOM
- Smaller models are generally only good at a small number of tasks. E.g. BERT
- You may have to try out a few models to find the right one for your use case. Once you've found the model that is working for you, there are a few settings that you can experiment with to influence the structure and style of the completions that the model generates.

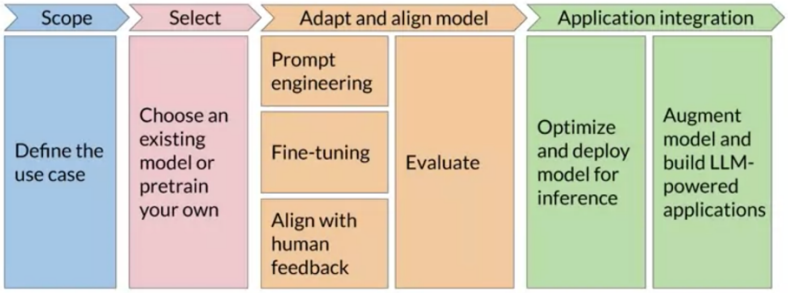
Generative configuration: influence the model's output during inference.



- Max new tokens: used to limit the number of tokens that the model will generate. You can think of this as putting a cap on the number of times the model will go through the selection process.
- Temperature: Final scaling factor applied to softmax layer
    - Smaller temperature (<1): Probability will be strongly peaked (more prob concentrated on less no. of words) → less randomness
    - Larger temperature (>1): Broader flatter prob distribution → more variability and randomness
    - Temperature = 1: Leaves softmax output as it is
- Greedy vs Random sampling:
    - Greedy: model always chooses the most probable word. Prone to repeating words or sequences in long generation lengths
    - Random: Generate more natural, unrepeated, creative sequences
    - Top k and p: used to limit the random sampling to get more sensible output
        - Top k: select an output from the top k results after applying the random weighted strategy using the probabilities
        - Top p: select an output using the random-weighted strategy with the top-ranked consecutive results by probability and with a cumulative probability <= p



Generative AI project Lifecycle



Week 1 Part II. LLM pre-training and scaling laws

Pre-training and scaling laws

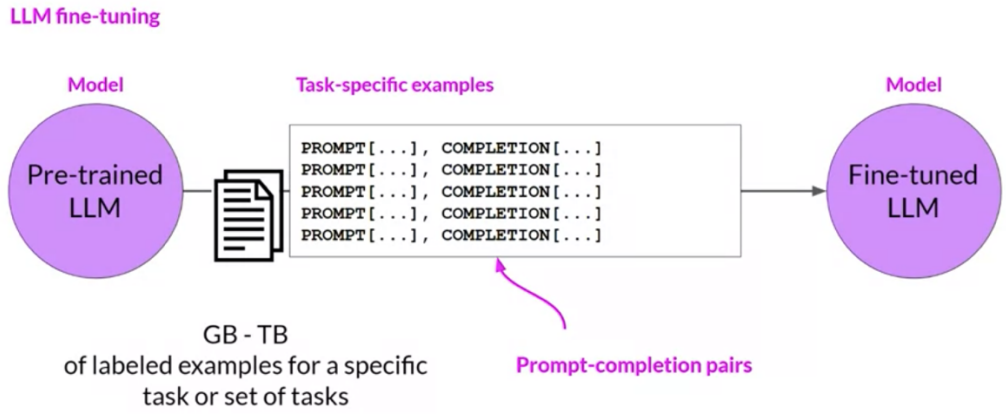
Week 2 Part I. Fine-tuning LLMs with instructions

Instruction fine-tuning

Limitation of in-text learning (prompt engineering):

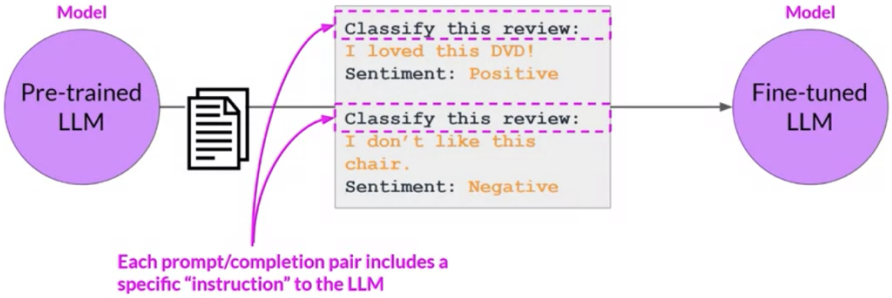
1. In-text learning may not work for smaller models
2. Examples take up space in the context window and reduce the amount of room you have to include other useful information

Fine tuning:
A supervised process where you use a data set of labeled examples (prompt completion pairs) to update the weights of the LLM.



Instruction fine tuning:

- A type of fine-tuning technique that trains the model using examples that demonstrate how it should respond to a specific instruction.
- Most times, fine-tuning refers to instruction fine-tuning.
- In the example below, the instruction to the LLM is ‘classify this review’. Other types of instructions include ‘summarize’, ‘translate’, and so on. This kind of prompt completion examples allow the model to learn to generate responses that follow the given instruction.



Full fine-tuning:

- A type of instruction fine-tuning, updates all model's weights
- Requires enough memory and computation budget to store and process all the gradients, optimizers, and other components that are being updated during training.

Prompt template libraries (three examples for AWS dataset):

Classification / sentiment analysis

```
jinja: "Given the following review:\n{{review_body}}\npredict the associated rating\
  \ from the following choices (1 being lowest and 5 being highest)\n- {{ answer_choices\
  \ | join('\\n- ') }} \n|||\n{{answer_choices[star_rating-1]}}"
```

Text generation

```
jinja: Generate a {{star_rating}}-star review (1 being lowest and 5 being highest)
  about this product {{product_title}}.  |||  {{review_body}}
```

Text summarization

```
jinja: Give a short sentence describing the following product review\n{{review_body}}\
  \ \n|||\n{{review_headline}}"
```

Fine-tuning process:

Step 1: Split prompt-completion data into training, validation, and test (holdout) dataset.

Step 2: Provide samples from the training dataset to the pre-trained LLM, and get the completion. Compare completion with requires response.

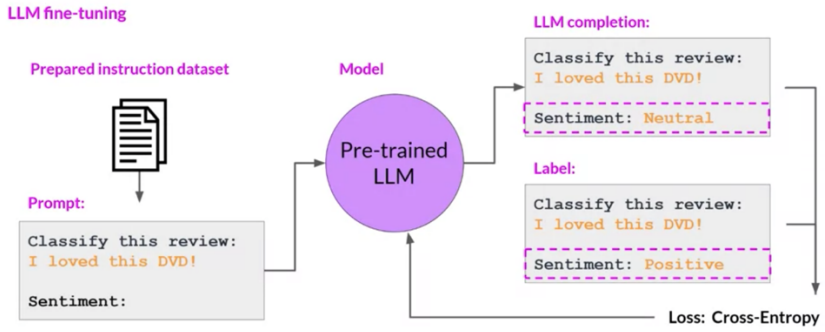
Step 3: Calculate loss (between probability distributions of completion and response) using cross-entropy

Step 4: Use loss to update weights through backpropagation

Step 5: Repeat for batches of prompt-completion pairs over several epochs

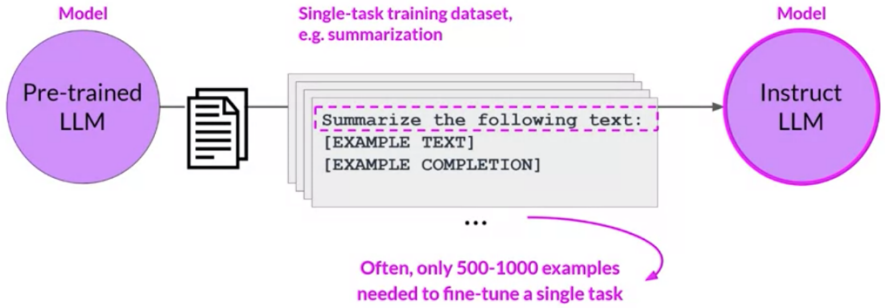
Step 6: Use holdout validation set to evaluate model - validation accuracy

Step 7: Perform final performance evaluation on holdout test dataset - test accuracy
Step 8: This results in a better, newer version of the pre-trained model AKA instruction model



Fine-tuning on a single task

- Only let the model do one task, e.g. summarization.
- Often, good results can be achieved with relatively few examples.
- Increase the performance of a model on a specific task while possibly reduce the ability in other tasks. For example, increase the performance in sentiment analysis but decrease the performance in entity recognition. This reduction in ability is called catastrophic forgetting. Catastrophic forgetting is a common problem in machine learning, especially in deep learning models, because deep learning models normally have many parameters, which can lead to overfitting and make it more difficult to retain previously learned information.

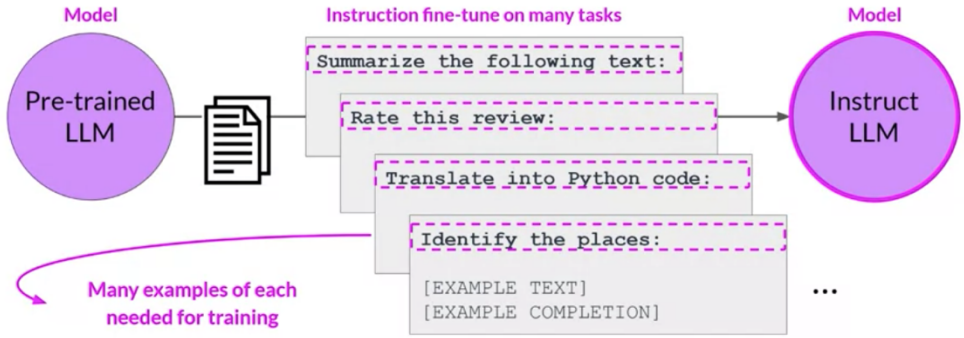


How to avoid catastrophic forgetting:

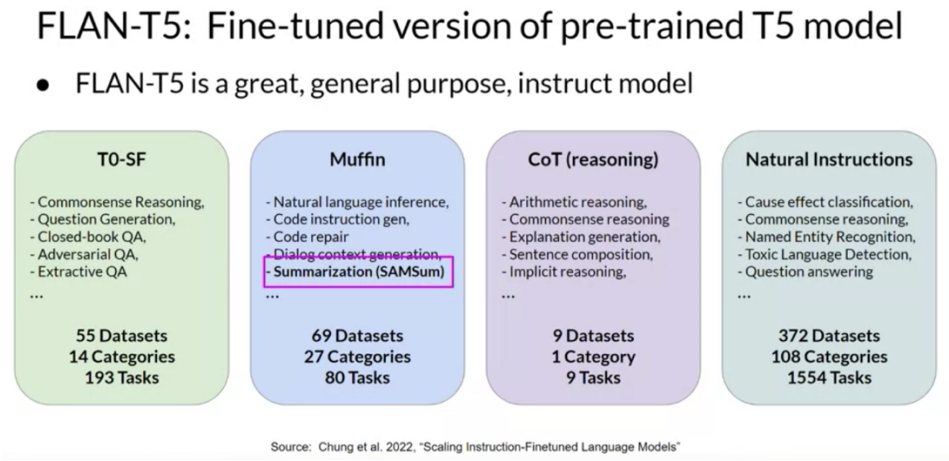
- If all you need is reliable performance on the single task you fine-tuned on, it may not be an issue that the model can't generalize to other tasks.
- Fine-tune on multiple tasks at the same time → Requires more data and compute to train
- Perform Parameter Efficient Fine-tuning (PEFT)
  - Preserves weights of original LLM, such as using regularization techniques, to preserve the information learned during earlier training phases and prevent overfitting to the new data
  - Trains only some adaptive layers
  - Active area of research

Multi-task instruction fine-tuning:

- A type of fine-tuning when the training dataset is comprised of example inputs and outputs for multiple tasks. The example below shows the four types of tasks, including summarization, classification, translation, and entity recognition.
- Avoid catastrophic forgetting
- Requires 50-100,000 examples in the training set.



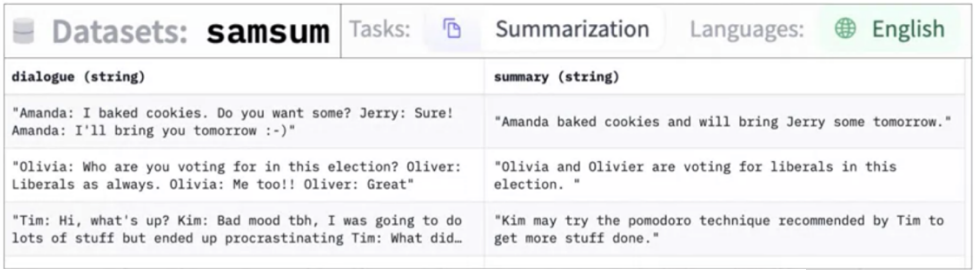
An example of multi-task fine-tuning model
FLAN family of models: It is a fine-tuned language net using a specific set of instructions to fine-tune models.



Including different ways of saying the same instruction helps the model generalize and perform better. After applying this template to each row in the SAMSum dataset, you can use it to fine tune a dialogue summarization task.

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."

Sample FLAN-T5 prompt templates

```
"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}"),
]
```

Improving FLAN-T5's summarization capabilities:
Suppose you want to build a chatbot for your customer service. The SAMSum dataset gives FLAN-T5 some abilities to summarize conversations. However, the examples in the dataset are mostly conversations between friends about day-to-day activities and don't overlap much with the language structure in customer service chats. 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.

Model evaluation

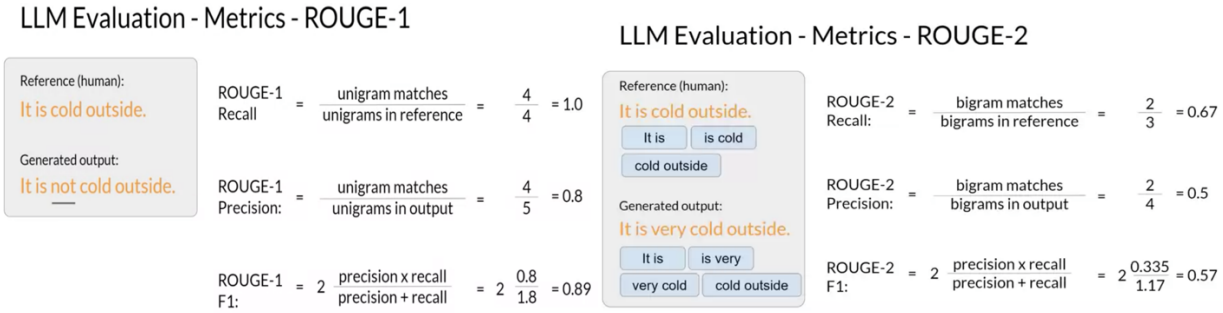
- In LLMs, the output is non-deterministic and language based, so need metrics suited to this.
- Two commonly used metrics: ROUGE and BLEU SCORE

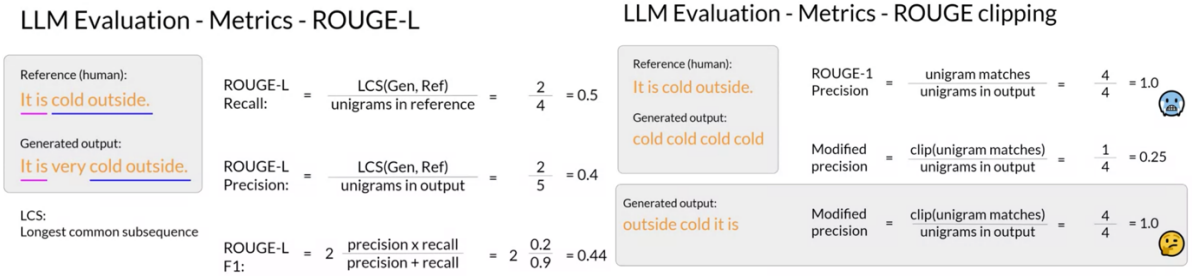
Terminologies:

- Unigram: one word
- Bigram: two words
- n-gram: a group of n words

ROUGE:

- used for text summarization
- compares a summary to one or more reference summaries
- ROUGE-1 does not consider the order of words and may cause the problem below that the ROUGE-1 remains the same no matter the output has not or not.
- ROUGE-2 overcomes the issues from ROUGE-1 by using bigram. However, the longer the sentence is, the higher the chance that bigrams from the human reference and generated output do not match.
- ROUGE-L: Use the longest common subsequence to derive
- ROUGE-clipping: overcome the issue when the same word is generated repeatedly by using a clipping function to limit the number of unigram matches to the maximum count for that unigram within the reference.





BLEU (bilingual evaluation under study) SCORE

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

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

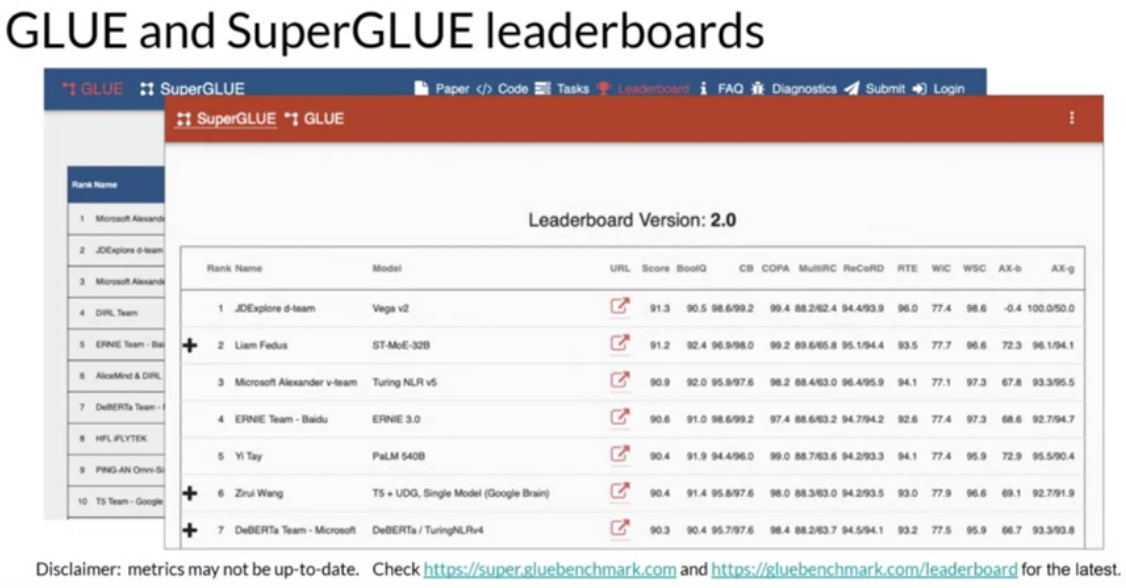
I am very happy that I am drinking a warm cup of tea. - BLEU 0.730

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

- I am very happy to say that I am drinking a warm cup of tea. - BLEU 1.000

Benchmark

- Use pre-existing datasets, and associated benchmarks to measure model performance.
- Use datasets that focus on specific skills, like reasoning or common-sense knowledge, or specific risks, such as disinformation or copyright infringement.
- Make sure the model hasn't seen your evaluation data during training.
- Some benchmarks:
    - GLUE, SuperGLUE
    - Benchmark for massive models: Massive Multitask Language Understanding (MMLU), BIG-bench Hard, BIG-bench, Lite
    - HELM (https://crfm.stanford.edu/helm/latest/): 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.



Holistic Evaluation of Language Models (HELM)

Week 2 Part II. Parameter-efficient fine tuning (PEFT)

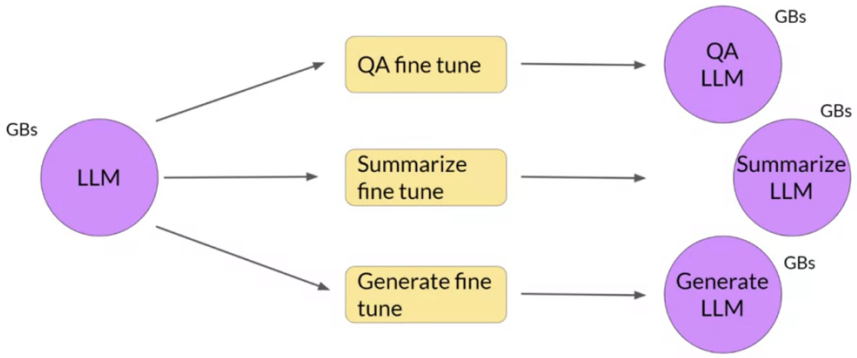
Introduction of PEFT

1. Weakness of full fine tuning:
   - computationally expensive and needs lots of memory space to store the model and parameters

Full fine-tuning of large LLMs is challenging

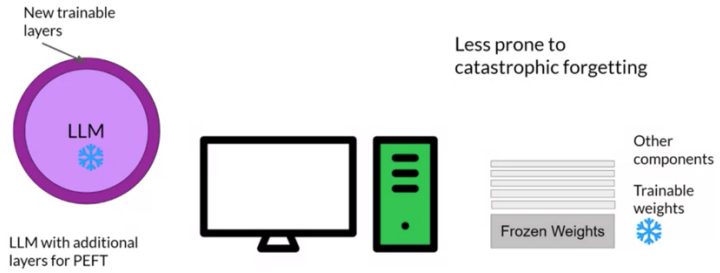
   - creates full copy of original LLM per task

Full fine-tuning creates full copy of original LLM per task

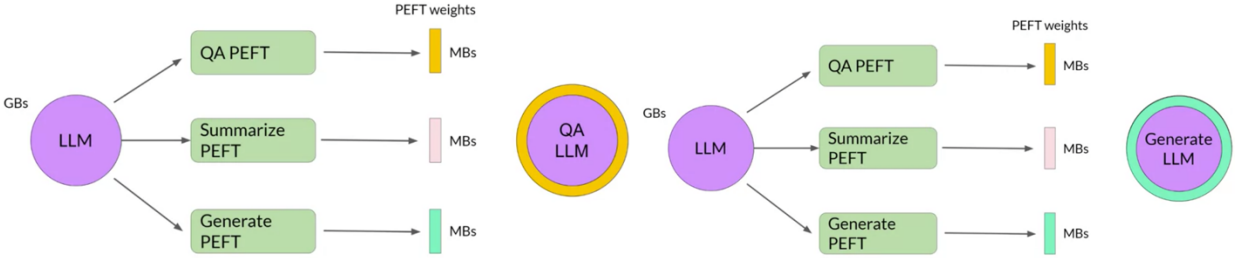


2. PEFT:

- PEFT only updates a small subset of existing model parameters or add a new trainable layer or add a small number of new parameters → less prone to catastrophic forgetting and can often be performed a single GPU



- PEFT weights for multiple tasks can be easily swapped out during inference, allowing efficient adaptation of the model to multiple tasks



- Trade-offs: parameter efficiency, training speed, inference costs, model performance, memory efficiency
- PEFT methods:
    - Selective: select subsets of original LLM parameters to fine-tune. Need to determine the parameters you want to update → significant trade-offs between parameter efficiency and compute efficiency
    - Reparameterization: also works with the original LLM parameters but reduces the number of parameters to train by creating a low-rank transformations of the original network weights. e.g. LoRA
    - Additive: keeps all original LLM weights frozen and introduces new trainable components
        - Adapters: add new trainable layers to the architecture of the model, typically inside the encoder or decoder components after the attention or feed-forward layers.
        - Soft prompt: 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.

Week 3 Part I. Reinforcement learning with human feedback

Week 3 Part II. LLM-powered applications