Finetuning: Fundamentals and best practices

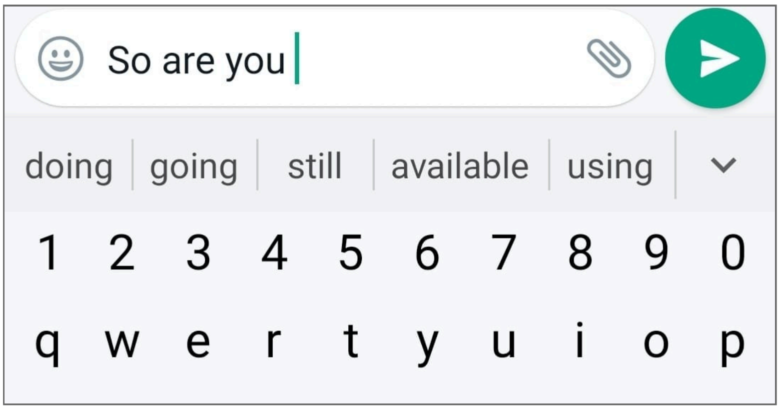
Ankush Chander
Dhirubhai Ambani University

LLM overview

Language modelling

Language modeling is a task to predict the next word or character in a sequence of text given the context of the previous words.

$$P(w_n | w_1, w_2, \ldots, w_n - 1) = ?$$



Evolution of Language models

	Statistical Language models	Neural Language models	Large[1] Languag models
Pros:	Simple to implement	generalizes well to unseen sequences as it capture semantic relationships	gener coher and conte releva text.
Cons:	1.struggle with capturing long-range dependencies. 2. didn"t capture semantic relationships. Eg: cat sat on a table. vs cat sat on a desk	Expensive to train	Exper to trai
Examples:	N-gram models, Hidden Markov Models (HMMs).	RNNs, LSTMs, GRU	GPT-3 4, BEF

1- **Large** Number of parameters(variables that gets updated while training a model) + amount of data on which they are trained

Stages of LLM training

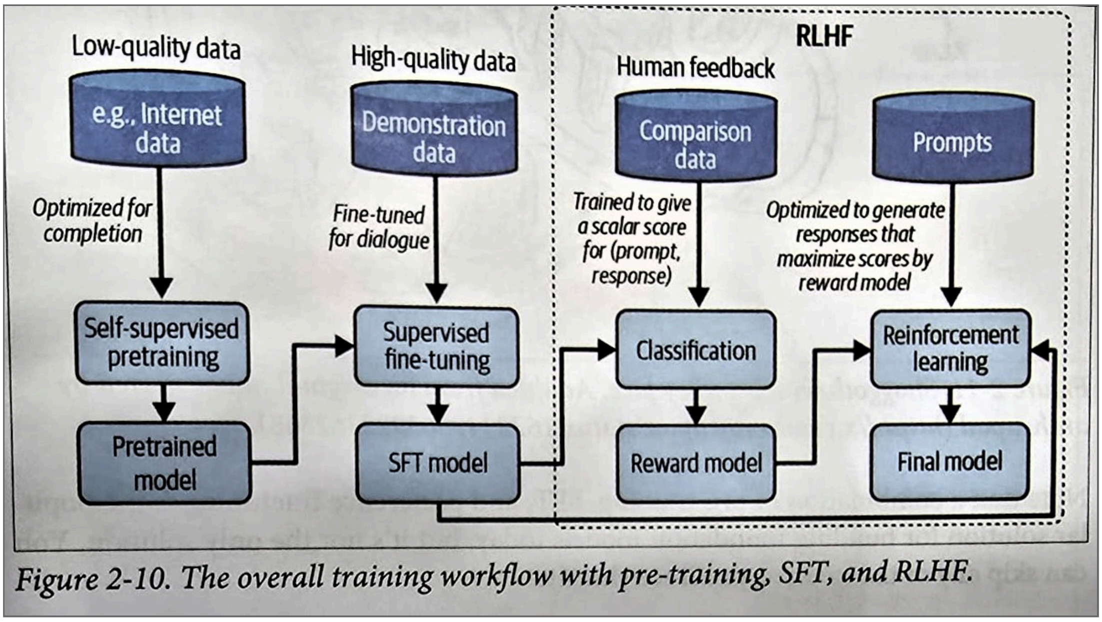


Figure 2-10. The overall training workflow with pre-training, SFT, and RLHF.

Image credits: **AI Engineering by Chip Huyen**

Different ways of applying LLMS

Prompt Engineering

Prompt Engineering refers to the process of crafting an instruction that gets model to generate the desired outcome.

Pros:

- Quick and easy to implement.
- No additional training required.
- Low computational cost.
- Works with any pre-trained model.

Cons:

- Limited customization.
- Relies on model’s existing knowledge.
- Hard to handle specialized or domain-specific tasks.
- Prone to hallucinations if model lacks relevant data.

Key terms:

In-context learning: Models ability to answer questions based on context provided in prompt without explicit training.

Few shot learning: PRoviding examples to teach model how to answer

Retrieval Augmented Generation(RAG)

RAG enhances model responses by *retrieving relevant external data* and providing it as context.
Pros:

- Reduces hallucination by grounding responses in retrieved documents.
- No need to modify model weights.
- Can adapt to dynamic or evolving data

Cons:

- Increased latency due to retrieval step.
- Performance depends on quality of retrieved documents.
- Requires infrastructure for storing and indexing documents.

Finetuning

Finetuning is a process of adapting a model to a specific task by updating whole or a part of the model.
Example use cases:

- Enhance model"s domain specific abilities like code completion, medical question answering
- strenghten safety
- improve instruction following ability

Pros:

- Highly customizable
- Allows adaptation to new domains, styles, or behaviors.

Cons:

- Requires high upfront investment and continual maintainance
- Resource intensive
- Harder to update dynamically compared to RAG.
- Requires a well-labeled dataset.

Development workflow

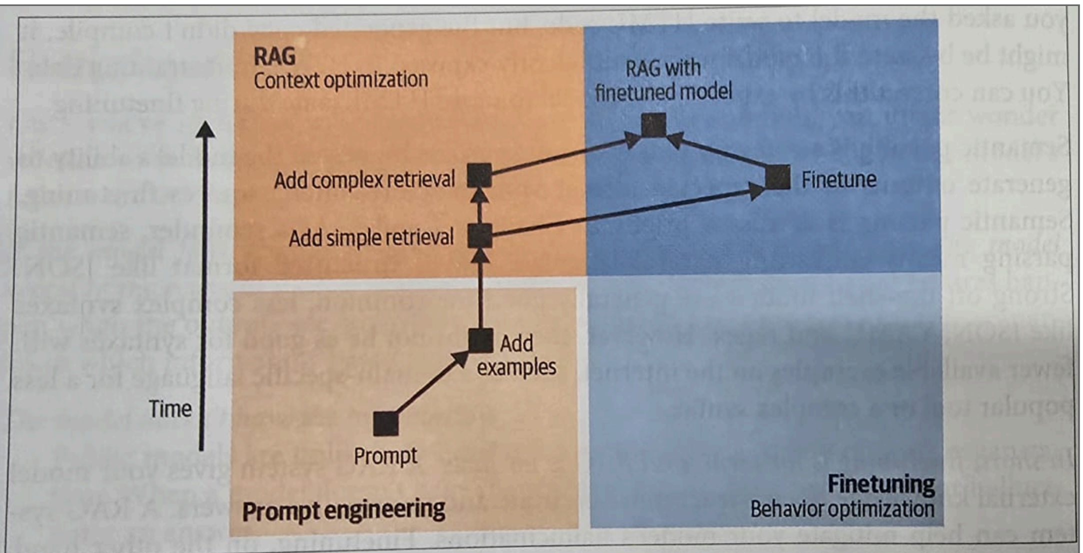


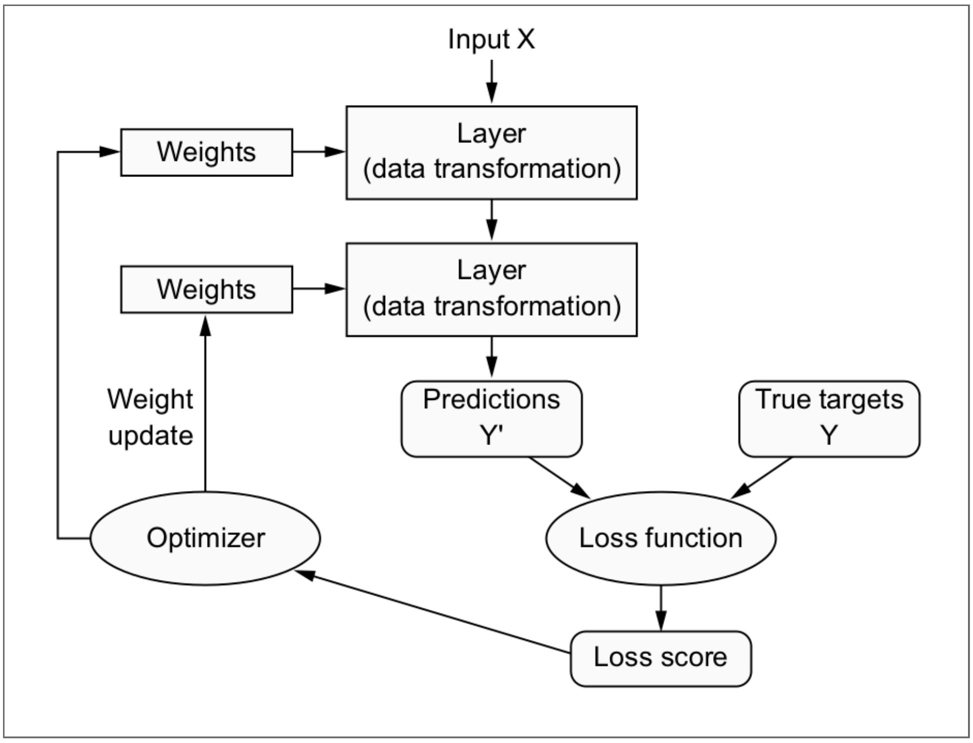
Figure 7-3. Example application development flows. After simple retrieval (such as term-based retrieval), whether to experiment with more complex retrieval (such as hybrid search) or finetuning depends on each application and its failure modes.

Image credits: **AI Engineering by Chip Huyen**

Demo 1: Fine tune an encoder only model using hugginface

Beginner guide to finetuning an opensource LLM

Hardware estimation



$$Modelmemory = model\_weights + gradients \\ + optimizer\_states$$

During Inference

$$\text{Total Memory}_{\text{Training}} = \text{Model Memory} + \text{Act}$$

- **Forward Pass**: Only requires memory for the model's weights.
    - Memory for weights: `N x M`
        - `N`: Model's parameter count
        - `M`: Memory needed for each parameter
- **Additional Memory**: Required for activation and key-value vectors in transformer models.
    - Assumed to be 20% of the model's weights.
    - Total memory footprint: `N x M x 1.2`
- **Example Calculation**:
    - 13B-parameter model, 2 bytes/parameter
    - Weights = 26 GB
    - Total Inference Memory = 31.2 GB

During Training

- **Overall Memory Needs**:

$$\text{Total Memory}_{\text{Training}} = \text{Model Memory} + \text{Optim}$$

- **Backward Pass**:
    - Each trainable parameter may need:
        - Gradient value
        - Optimizer states (depending on optimizer type)
    - **Optimizers**:
        - Vanilla SGD: No state
        - Momentum: One value per parameter
        - Adam: Two values per parameter
- **Example Calculation**:
    - 13B-parameter model, Adam optimizer
    - Each parameter has 3 values (gradient + optimizer states)
        - Memory for gradients and optimizer states = 13 * 2 *

(1+2)= 78 GB

- Total memory = 13 * 2 * (1+1+2) = 104GB

-

- **Activation Memory**:
    - Can surpass memory needed for weights.
    - **Optimization**: Use gradient checkpointing to reduce memory, at the cost of increased training time.
- **Additional Notes**:
    - Memory needs grow rapidly with model size.
    - Techniques to reduce memory consumption include recomputation.

`accelerate-estimate-memory` is a CLI command that will load the model into memory on the meta device, so we are not actually downloading and loading the full weights of the model into memory, nor do we need to. As a result it’s perfectly fine to measure 8 billion parameter models (or more), without having to worry about if your CPU can handle it!

In [9]:

```
# https://huggingface.co/microsoft/Phi-3.5-mini-instruct
!accelerate-estimate-memory microsoft/Phi-3.5-mini-instruct
```

```
Loading pretrained config for `microsoft/Phi-
3.5-mini-instruct` from `transformers`...
┌──────────────────────────────────────────────
───────────────┐
│ Memory Usage for loading `microsoft/Phi-3.5-
mini-instruct` │
├───────┬─────────────┬──────────┬────────────
───────────────┤
│ dtype │Largest Layer│Total Size│    Training
using Adam     │
├───────┼─────────────┼──────────┼────────────
───────────────┤
│float32│  432.02 MB  │ 14.23 GB │          5
6.94 GB         │
│float16│  216.01 MB  │ 7.12 GB  │          2
8.47 GB         │
│  int8 │  108.01 MB  │ 3.56 GB  │
N/A            │
│  int4 │   54.0 MB   │ 1.78 GB  │
N/A            │
└───────┴─────────────┴──────────┴────────────
───────────────┘
```

In [10]:

```
!accelerate-estimate-memory meta-llama/Llama-2-13b-hf
```

```
Loading pretrained config for `meta-llama/Llama-2-13b-hf` from `transformers`...
config.json: 100%|████████████████████████████| 610/610 [00:00<00:00, 5.51MB/s]
```

Memory Usage for loading `meta-llama/Llama-2-13b-hf`			
dtype	Largest Layer	Total Size	Training using Adam
float32	1.18 GB	47.88 GB	191.51 GB
float16	605.02 MB	23.94 GB	95.76 GB
int8	302.51 MB	11.97 GB	N/A
int4	151.25 MB	5.98 GB	N/A

Finetuning techniques

Quantization

Model quantization is a common way to reduce model hardware requirements. Reducing the precision of the model weights and activations of the model reduces the GPU RAM requirements. For example changing model precision from float16 to int8 halves the size of the VRAM requirements. It also leads to kv cache size reduction.

Floating point representation

Representation	Mantissa	Exponent(range of numbers)	Sign	Exa
	decides the precision with which numbers can be represented	decides the range of number that can be represented		
FP32	23	8	1	3.
FP16	10	5	1	3.
FP8	2	5	1	3

ADAPTER BASED TECHNIQUES

LoRA, short for Low-Rank Adaptation, is a method designed to efficiently fine-tune large pre-trained models. The intuition behind LoRA stems from the understanding that the vast majority of the parameters in a pre-trained model remain unchanged during fine-tuning.

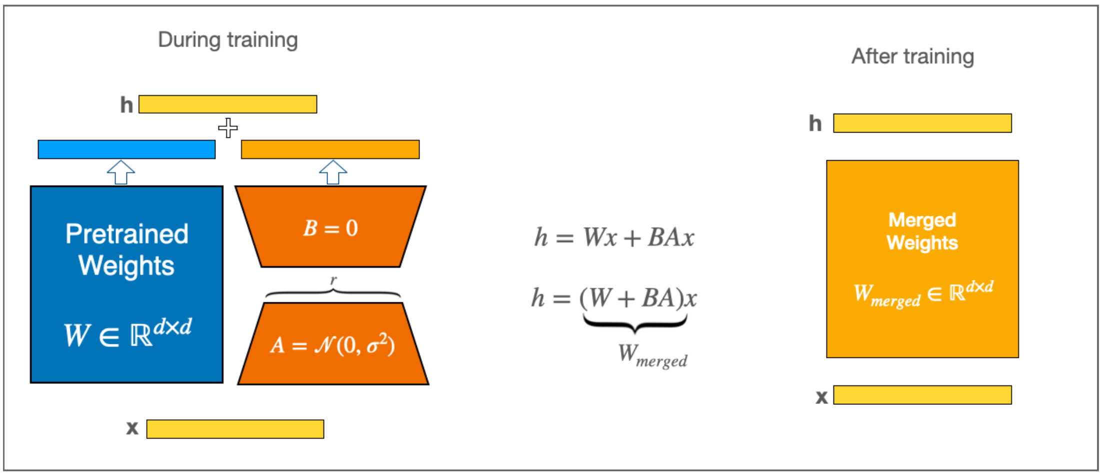


Image credits: **huggingface**

References

3. **Hardware for LLMs - by Benjamin Marie**
4. **Orca: A Distributed Serving System for Transformer-Based Generative Models | USENIX**
5. **QLoRA: Fine-Tune a Large Language Model on Your GPU**
6. **Fundamentals of Data Representation: Floating point numbers - Wikibooks, open books for an open world**
7. **AI engineering resources**