# LLM Cheat Sheet

September 29, 2025

## Contents

# Handling Imports from Parent Folders in Python and Jupyter

When a Python script or Jupyter Notebook is located in a subfolder, modules in a parent folder may not be automatically discoverable by Python. The following methods illustrate how to properly import them.

## 1. Using __file__ in Python Scripts

For scripts executed directly, __file__ contains the path of the script. Use it to construct the parent folder path:

**Python Script: Using __file__**

```
import sys
import os

# Add parent folder to Python path
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

# Import module from parent folder
from LLMengineering import key_utils
```

**Explanation:**

- `os.path.dirname(__file__)`: folder of the current script.
- `os.path.join(..., '..')`: go one level up.
- `os.path.abspath(...)`: get absolute path.
- `sys.path.append(...)`: add this path to Python's module search paths.

## 2. Using `os.getcwd()` in Jupyter Notebooks

In Jupyter, __file__ is not defined. Use the current working directory instead:

**Jupyter Notebook: Using os.getcwd()**

```
import sys
import os

# Add parent folder of current working directory
sys.path.append(os.path.abspath(os.path.join(os.getcwd(), '..')))

# Import module from parent folder
from key_utils import some_function
```

**Notes:**

- `os.getcwd()`: returns the current working directory of the notebook.
- `..` moves one level up; `../..` moves two levels up.
- This approach is portable for notebooks without relying on __file__.

## 3. Using Absolute Paths (Less Portable)

**Absolute Path Approach**

```
import sys
sys.path.append("/run/media/sijanpaudel/New Volume/New folder/LLMengineering")

from key_utils import some_function
```

**Caution:** This works immediately but is hard-coded. If the project moves, the path must be updated.

## 4. Recommended Long-Term Approach

1. Add an empty __init__.py file in the parent folder (e.g., `LLMengineering/`) to make it a package.
2. Use relative or absolute imports within scripts:

**Long-Term Package Approach**

```
    from LLMengineering import key_utils
```

This makes imports clean, portable, and avoids repeated `sys.path.append` hacks.

# Comparing LLMs – The Basics (1)

| Feature | Description / Notes |
|---|---|
| Open-source or Closed | Whether the model's weights, architecture, and training code are publicly available. Examples: GPT-4 (closed), LLaMA 2 (open). |
| Release Date & Knowledge Cut-off | The date the model was released and the latest point in time the training data covers. Important for up-to-date responses. |
| Parameters | Number of trainable weights in the model (e.g., billions of parameters). Determines model capacity. |
| Training Tokens | Amount of text data used during training, usually in billions of tokens. More tokens generally improve performance. |
| Context Length | Maximum input length (in tokens) the model can handle in a single prompt. Longer context allows better understanding of large inputs. |

# Comparing LLMs – The Basics (2)

| Feature | Description / Notes |
|---|---|
| Inference Cost | Computational resources required to generate output (GPU, CPU usage). |
| API Charge / Subscription / Runtime Compute | How much it costs to access the model through a cloud service. |
| Training Cost | Cost to pretrain the model, including compute, electricity, and infrastructure. |
| Build Cost | Cost to fine-tune, deploy, or customize the model. |
| Time to Market | How quickly you can deploy and use the model in production. |
| Rate Limits | Restrictions on API usage, e.g., calls per minute or per day. |
| Speed & Latency | How fast the model responds, depends on model size, hardware, and context length. |
| License | Terms for use, redistribution, and commercial deployment. Some open-source licenses restrict commercial use. |

# Chinchilla Scaling Law

**Key Principle:** Number of model parameters should scale roughly proportional to the number of training tokens.
**Implications:**

- Increasing model size without enough data leads to plateaued or degraded performance.
- Having lots of training data with a small model underperforms; parameters must scale up.

**Rule of Thumb:** Let $N$ = model parameters, $D$ = training tokens. For optimal training:

$$N \propto D$$

**Example:**

- Doubling model parameters → need roughly double the training tokens.
- Insufficient tokens → diminishing returns.

**Additional Notes:**

- Smaller, well-trained models can outperform larger, undertrained ones.
- Helps determine compute-efficient configurations for new LLMs.

# Common LLM Benchmarks

| **Benchmark** | **Focus / Task** | **Description** |
|---|---|---|
| MMLU | Knowledge across multiple subjects | Assess general knowledge and reasoning; used to compare models like GPT-3, Chinchilla, and Gopher. |
| BIG-bench | Broad suite of diverse reasoning tasks | Tests reasoning, factual knowledge, ethics, math, code; hundreds of tasks evaluating beyond narrow QA. |
| HellaSwag | Commonsense reasoning | Multiple-choice questions for everyday situations; measures ability to predict plausible outcomes. |
| TruthfulQA | Factual accuracy / truthfulness | QA tasks designed to detect hallucinations; evaluates honesty of LLM answers. |
| WinoGrande / Winograd Schema Challenge | Pronoun resolution / coreference | Tests commonsense reasoning and context understanding; resolves ambiguous references. |
| ARC | Science and reasoning | Multiple-choice science questions; evaluates problem-solving and reasoning in STEM. |
| HumanEval | Coding and code generation | Tests Python programming ability; measures functional correctness of generated code. |

## Specific Benchmarks

| Benchmark | What's Being Evaluated | Description |
|---|---|---|
| ELO | Model ranking / performance consistency | Evaluates models via pairwise comparisons; creates a relative ranking of LLMs across multiple tasks. |
| HumanEval | Code generation / functional correctness | Tests an LLM's ability to write Python functions that pass unit tests; measures coding logic and correctness. |
| Multipl-E | Multimodal reasoning | Evaluates LLMs on tasks combining text and images (or multiple modalities); measures reasoning and comprehension across modalities. |

## Limitations of Benchmarks

| Limitation | Explanation |
|---|---|
| Narrow focus | Many benchmarks test only specific skills (e.g., coding, factual QA, commonsense), not overall intelligence or adaptability. |
| Static datasets | Benchmarks are fixed in time, so models trained after the cut-off may have an unfair advantage or miss newer knowledge. |
| Lack of real-world context | Benchmarks often use idealized tasks, not messy, ambiguous, or multi-step real-world scenarios. |
| Gaming / overfitting | Models can be fine-tuned or prompted to specifically excel on benchmark tasks without improving general capabilities. |
| Limited multimodality | Most benchmarks focus on text-only tasks; few measure image, audio, or multimodal reasoning. |
| Subjectivity | Some benchmarks (e.g., ethics, creativity, hallucination detection) are hard to score objectively. |
| Compute bias | Larger models may perform better mainly due to size, not reasoning ability, skewing benchmark results. |
| Hard to measure nuanced reasoning | Benchmarks mostly measure surface correctness; they often cannot capture multi-step reasoning, context-dependent judgment, creativity, or reasoning accuracy vs. fluency. |

# Advanced Benchmarks for Large Language Models

| Benchmark | Focus / Task | Description / Meaning |
|---|---|---|
| GPQA | Graduate-level question answering | Evaluates performance on graduate-level tests with 448 expert questions. Non-PhD humans score only 34% even with web access. Measures LLM ability to handle highly specialized knowledge. |
| BBHard | Future capabilities | Includes 204 tasks previously thought beyond LLM capabilities. Designed to test reasoning, logic, and generalization at a next-level difficulty. |
| Math Lv 5 | High-school math competition problems | Measures model's ability to solve advanced math problems requiring multi-step reasoning and problem-solving skills. Useful for chain-of-thought evaluation. |
| IFEval | Instruction following | Tests the model's ability to follow complex instructions, e.g., "write more than 400 words" and "mention AI at least 3 times." Evaluates comprehension and compliance with nuanced prompts. |
| MuSR | Multistep soft reasoning | Assesses logical deduction and multi-step reasoning. Example: analyzing a 1,000-word story and identifying "who has means, motive, and opportunity." Tests reasoning beyond surface facts. |
| MMLU-PRO | Harder MMLU version | Advanced, cleaned-up version of MMLU with questions having 10 possible answers instead of 4. Evaluates deeper knowledge, multi-choice reasoning, and generalization. |

**Notes:**

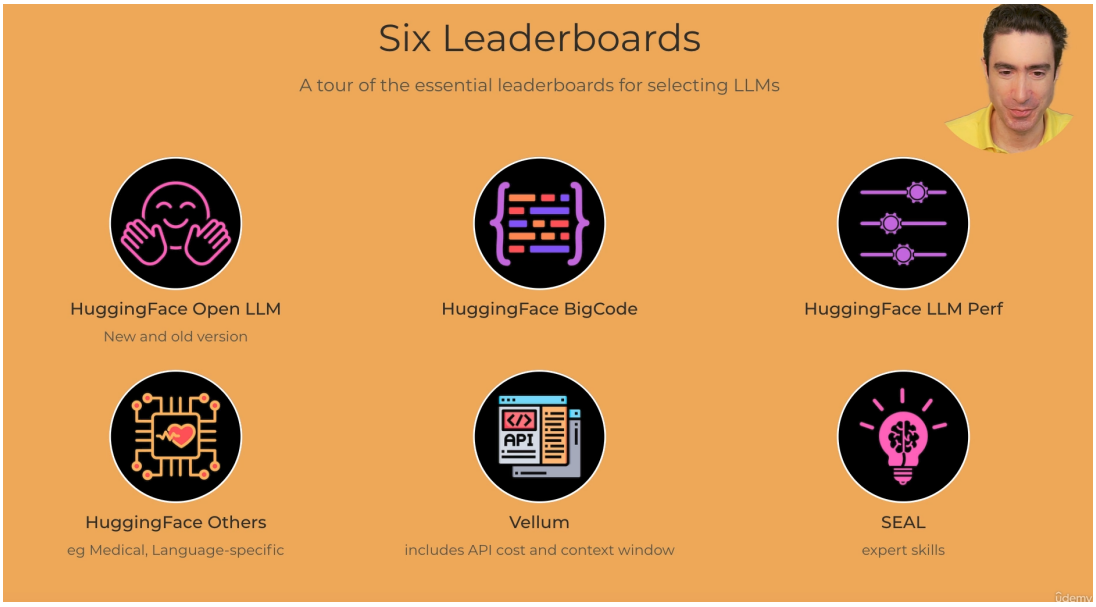
- These benchmarks are considered "next-level" because they test advanced reasoning, multi-step logic, and high-level knowledge.
- Useful for evaluating models that go beyond standard QA, commonsense reasoning, or basic coding tasks.
- Many of these benchmarks also measure compliance with complex instructions and reasoning under uncertainty.

| Benchmark | Purpose / Examples |
|---|---|
| GPQA | Specialized knowledge, expert-level Q&A, academic reasoning, physics, chemistry, biology expertise.<br>**Examples:** Explain the biochemical steps of glycolysis; Describe Newton's laws with practical applications; Explain photosynthesis in detail. |
| BBHard | Advanced reasoning, logic, generalization in challenging scenarios.<br>**Examples:** Predict how a hypothetical AI agent could optimize a supply chain in a novel scenario; Design a strategy to minimize energy consumption in an unfamiliar industrial process; Solve a logic puzzle with multiple constraints. |
| Math Lv 5 | Complex problem solving, chain-of-thought evaluation, mathematical reasoning.<br>**Examples:** Solve: If $x^2 - 5x + 6 = 0$, find all integer solutions; Evaluate combinatorial problems like "How many ways can 5 books be arranged on a shelf?"; Solve calculus problems requiring multiple steps. |
| IFEval | Precise instruction compliance, comprehension, content generation with constraints.<br>**Examples:** Write a 450-word essay on renewable energy mentioning AI at least 3 times; Summarize a research paper in 300 words including key terms; Rewrite a paragraph to follow a formal tone while keeping original meaning. |
| MuSR | Multi-step reasoning, deduction, understanding narratives beyond surface facts, solving prime puzzles and reasoning tasks.<br>**Examples:** Analyze a 1,000-word mystery story to determine "who has means, motive, and opportunity"; Identify the next prime number in a complex sequence; Solve multi-step logic puzzles. |
| MMLU-PRO | Deep knowledge, advanced multi-choice understanding, broader knowledge evaluation, general language understanding.<br>**Examples:** Choose the best explanation for why the sky is blue from 10 options; Identify the correct historical fact among 10 alternatives; Evaluate grammar and style in multiple-choice questions. |

**Tips to distinguish benchmarks:**

- **GPQA** – focus on subject expertise.
- **BBHard** – focus on novel reasoning and logic.
- **IFEval** – focus on following instructions accurately.
- **MuSR** – focus on multi-step deduction and problem solving (e.g., prime puzzles, mysteries).
- **MMLU-PRO** – focus on broad knowledge and multiple-choice reasoning.

# Leaderboard Image




| **Leaderboard** | **Purpose / Use** |
|---|---|
| HuggingFace Open LLM | Tracks performance of open-source LLMs (new and old versions). Useful for selecting models for research or deployment with full weight access. |
| HuggingFace BigCode | Focused on code generation LLMs. Helps compare models for coding tasks, programming language coverage, and code quality. |
| HuggingFace LLM Perf | Benchmarking general LLM performance across various NLP tasks. Useful for measuring speed, accuracy, RAM memory usage and general reasoning ability. |
| HuggingFace Others | Specialized leaderboards, e.g., medical, domain-specific, or language-specific models. Guides selection for niche applications. |
| Vellum | Includes context window, API cost, and usage constraints. Useful for developers to choose models based on practical deployment factors. |
| SEAL | Focused on expert skills and high-level reasoning. Useful for selecting LLMs that excel in professional or complex knowledge domains. |

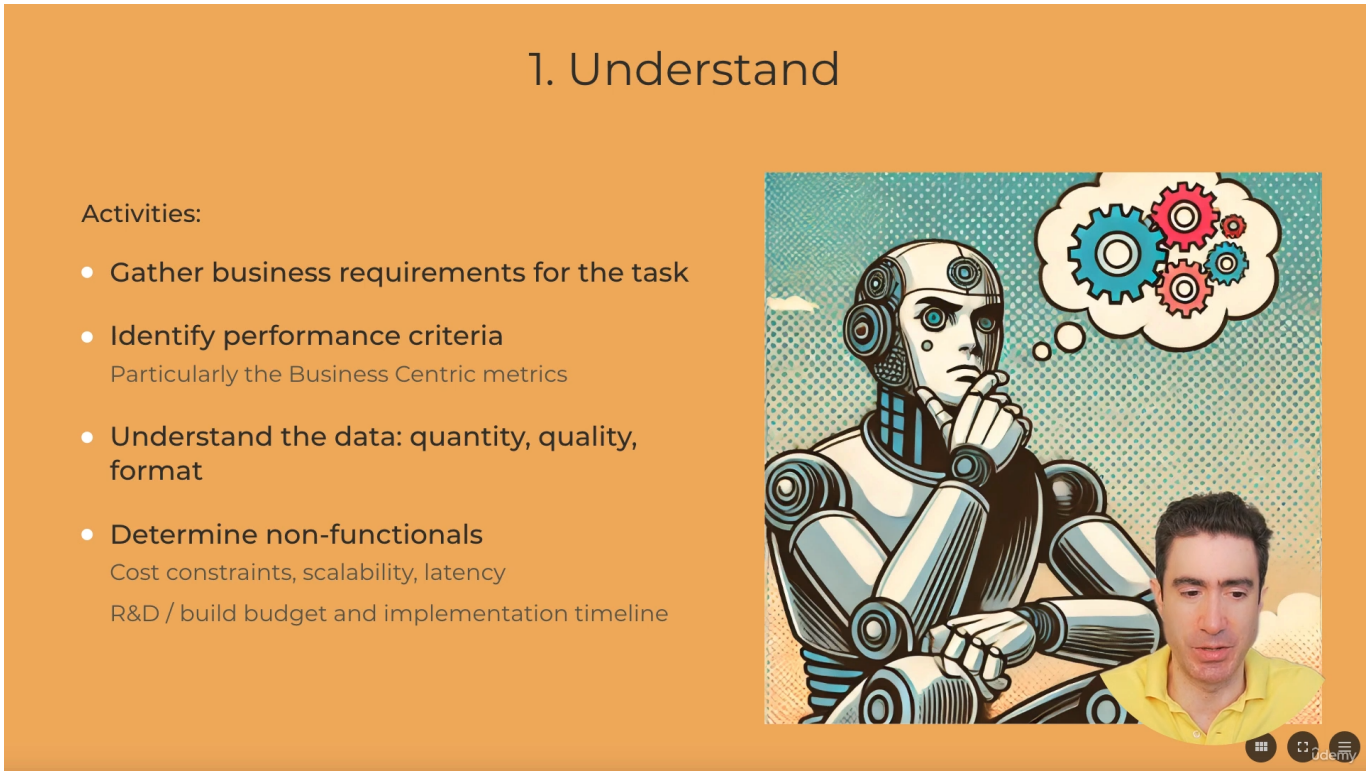
# A 5-Step Strategy for Using AI Models

This is a simple guide to choosing, training, and using a Large Language Model (LLM) to solve a real-world business problem.

## Step 1: Understand (Figure Out What You Need)

Before you start, you need a clear plan. Think about what you're trying to achieve.

- **Gather Business Needs:** What is the specific problem you want the AI to solve? (e.g., "Answer customer support questions automatically.")
- **Define Success:** How will you know if the AI is doing a good job? Focus on what matters to the business, not just technical scores. (e.g., "Reduce customer wait times by 50
- **Look at Your Data:** What information do you have? Is it a lot? Is it messy or clean? What format is it in?
- **Consider the Limits:** Think about the practical stuff.
  - How much money can you spend?
  - Does it need to be super fast?
  - How many people will be using it?
  - What is the deadline to get this working?



## Step 2: Prepare (Do Your Homework)

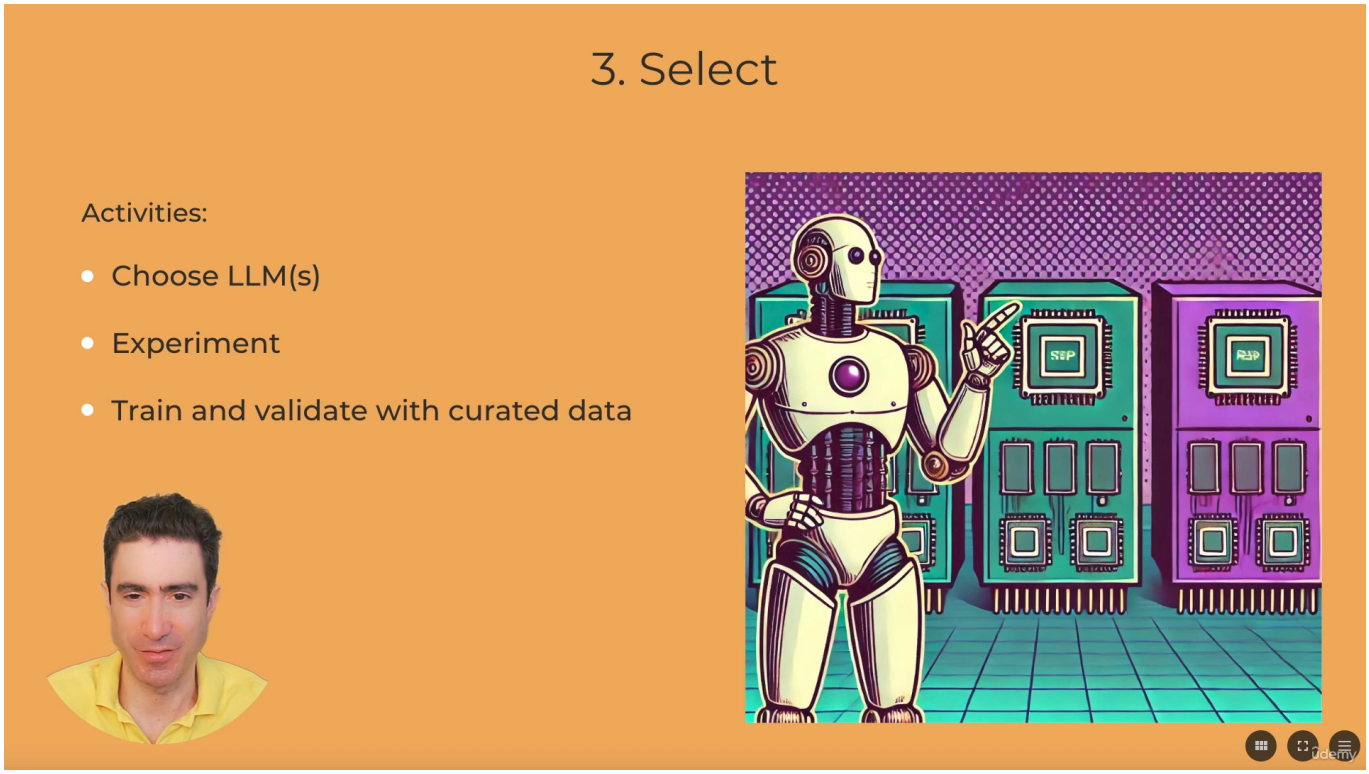
Once you have a plan, it's time to get your resources ready.

- **Research Solutions:** See how this problem is already being solved. Look at solutions that don't use a big AI model to set a "baseline" for performance.
- **Compare AI Models:** Look at different LLMs.
  - **The Basics:** How much do they cost? How much text can they handle at once (context length)? Can you legally use them for your business?
  - **Performance:** Check online tests (Benchmarks, Leaderboards) to see which models are best at certain tasks.
- **Clean Your Data:** This is very important! You need to organize, clean, and prepare your data so the AI can learn from it effectively.

## Step 3: Select (Pick Your Tools and Test Them)

Now you can choose your model and see how it performs with your data.

- **Choose Your LLM(s):** Based on your research, pick one or two models that seem like the best fit.
- **Experiment:** Play around with the models. See what they're good at and where they struggle.
- **Train and Check:** Use your clean data to test (validate) the models to get real performance numbers.



## Step 4: Customize (Make the AI Better at Your Task)

Out-of-the-box models are good, but you can make them great by using special techniques to improve their performance on your specific problem. There are three main ways to do this.

### Three Key Techniques

**Prompting**

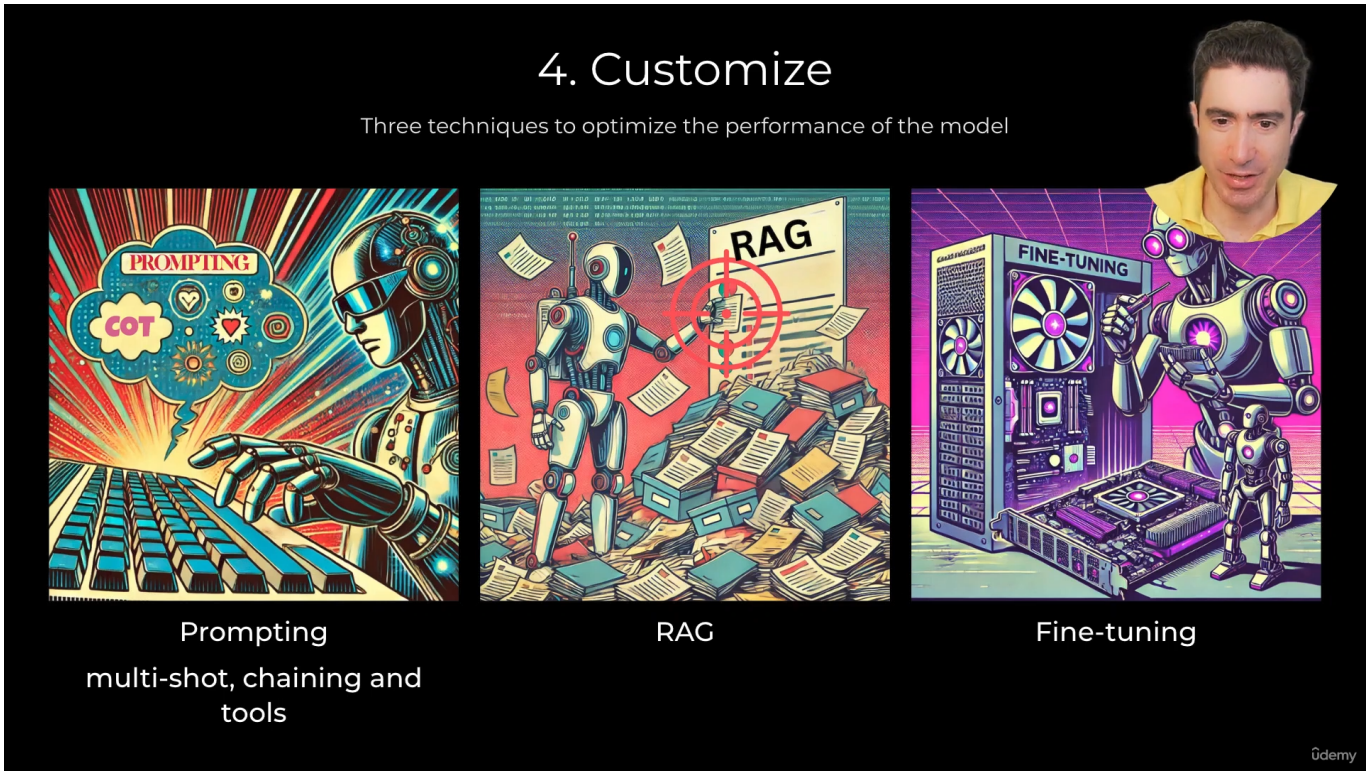
This means giving the AI better instructions. It's the fastest and cheapest way to improve performance. You can give it examples, tell it to "think step-by-step," or connect it to other software tools.

**RAG (Retrieval-Augmented Generation)**

This is like giving the AI an "open book" for a test. You connect the model to a knowledge base (like your company's internal documents). When a question is asked, the AI first finds the relevant information from your documents and then uses that information to create the answer. This ensures the answers are accurate and up-to-date.

**Fine-Tuning**

This is the most advanced technique. You actually re-train a part of the AI model on your own data. This teaches the model a new skill or a specific style, making it an expert in your particular area. It's expensive and requires a lot of data but can lead to the best performance.



**Comparing the Three Techniques**

**Pros: The Good Stuff**

| Prompting | RAG | Fine-tuning |
|---|---|---|
| 1. Quick to do | 1. More accurate answers | 1. Creates a true expert |
| 2. Cheap | 2. Works with lots of data | 2. Understands subtle details |
| 3. See improvements right away | 3. Efficient | 3. Can learn a specific tone or style |
| | | 4. Faster and cheaper answers |

**Cons: The Downsides**

| Prompting | RAG | Fine-tuning |
|---|---|---|
| 1. Limited by how much text the AI can read at once | 1. More complex to set up | 1. Takes a lot of effort |
| 2. Improvements eventually stop | 2. Needs accurate, up-to-date info | 2. Needs a lot of clean data |
| 3. Can be slow and expensive for each answer | 3. Can miss subtle details | 3. Costs money to train |
| | | 4. Risk of the AI forgetting its original general knowledge |

## Three Techniques: **Pros**

**Prompting**

1. Fast to implement
2. Low cost
3. Often immediate improvement

**RAG**

1. Accuracy improvement with low data needs
2. Scalable
3. Efficient

**Fine-tuning**

1. Deep expertise & specialist knowledge
2. Nuance
3. Learn a different tone / style
4. Faster and cheaper inference

Udemy

## Three Techniques: **Cons**

**Prompting**

1. Limited by context length
2. Diminishing returns
3. Slower, more expensive inference

**RAG**

1. Harder to implement
2. Requires up-to-date, accurate data
3. Lacks nuance

**Fine-tuning**

1. Significant effort to implement
2. High data needs
3. Training cost
4. Risk of "catastrophic forgetting"

Udemy

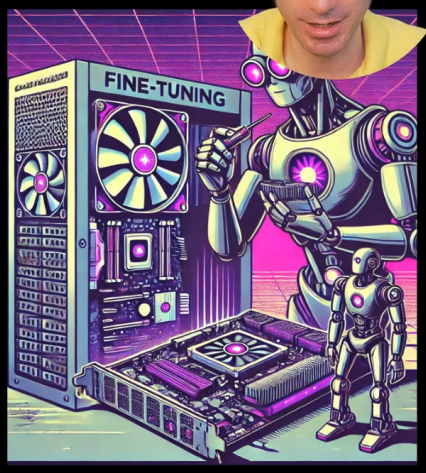
# Three Techniques: **Use Cases**


**Prompting**

Often the starting point for optimizing a project,with a Frontier LLM


**RAG**

You need high accuracy without the cost of fine-tuning; you have a Knowledge Base



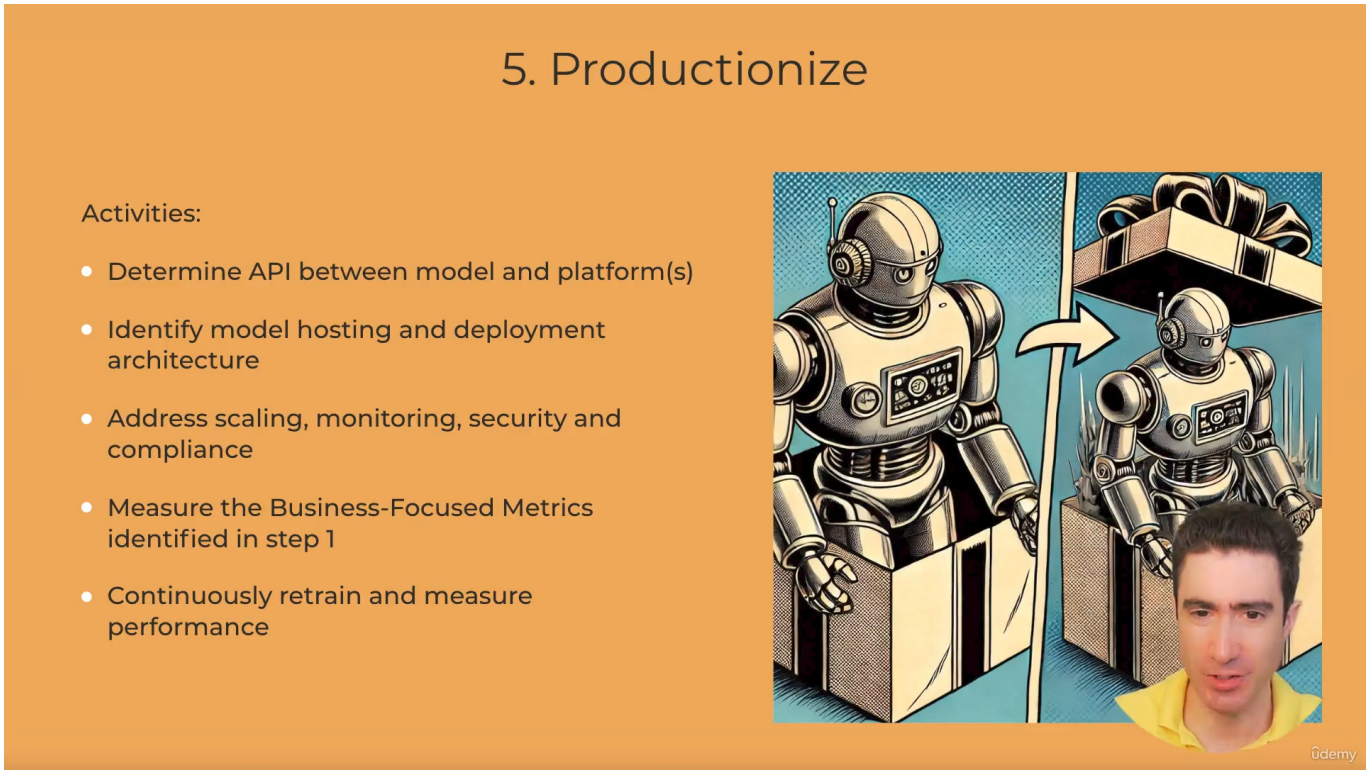

**Fine-tuning**

You have a specialized task with a high volume of data, and you need top performance

## Step 5: Productionize (Get It Ready for the Real World)

Once the model is working well, you need to make it a reliable part of your business.

- **Connect it:** How will your website or other software "talk" to the AI model?
- **Deploy it:** Where will the model run? On your own computers or in the cloud?
- **Plan for Growth:** What happens if you get thousands of users? Make sure it's secure, and you're monitoring it for problems.
- **Measure What Matters:** Go back to Step 1 and measure the business goals you set. Is it actually working?
- **Keep it Updated:** Continuously check the AI's performance and re-train it with new data to keep it from becoming outdated.



# Traditional ML models

We will quickly set up these solutions to give us a starting point

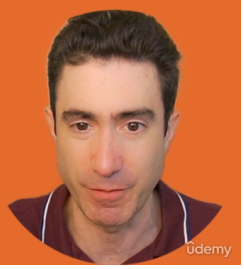
Feature engineering & Linear Regression

Bag of Words & Linear Regression

word2vec & Linear Regression

word2vec & Random Forest

word2vec & SVR



# LoRA (Low-Rank Adaptation)

## 1. Introduction

LoRA (Low-Rank Adaptation) is a technique designed to **fine-tune large pre-trained models efficiently**. Large language models (LLMs) like GPT, LLaMA, etc., have **billions of parameters**, making traditional fine-tuning:

- Extremely expensive in computation.
- Memory-intensive.
- Hard to maintain multiple fine-tuned versions.

LoRA solves this by training **small, additional modules** instead of updating the entire model.

## 2. How LoRA Works (Detailed)

Consider a weight matrix in a neural network layer:

$$W \in \mathbb{R}^{d \times k}$$

Full fine-tuning adjusts every element in $W$, which is costly.
LoRA introduces **two low-rank matrices** $A$ and $B$ and approximates the weight update:

$$\Delta W \approx AB$$

where:

$$A \in \mathbb{R}^{d \times r}, \quad B \in \mathbb{R}^{r \times k}, \quad r \ll \min(d, k)$$

Key points:

- $r$ is small, so $A$ and $B$ are **tiny compared to** $W$.
- Only $A$ and $B$ are trained; the original $W$ is **frozen**.
- Memory and computation are drastically reduced.

## 3. Step-by-Step Intuition

1. Start with a pre-trained model whose parameters $W$ are **frozen**.
2. Add small trainable matrices $A$ and $B$ to selected layers (often attention layers).
3. Forward pass: compute original output using $W$ and add low-rank contribution:

   $$y = Wx + \alpha(ABx)$$

   where $\alpha$ is a scaling factor.
4. Backpropagation only updates $A$ and $B$, leaving $W$ untouched.
5. After training, save only $A$ and $B$ as the **LoRA adapter**.

## 4. Simple Analogy

**Analogy**

Think of a professional camera with many settings:

- Full fine-tuning = manually adjusting every knob and lens component each time you want a new photo style.
- LoRA = keep the camera fixed, but attach a **small filter**.

The filter is light, cheap, and can be swapped out, while the main camera stays unchanged.

## 5. Why LoRA is Effective

- Neural network updates often lie in a **low-rank subspace**, so approximating with small matrices captures most of the needed changes.
- Reduces training cost dramatically.
- Reduces storage: instead of storing full fine-tuned model ( 10s of GB), store just adapters ( 10s of MB).
- Enables **multi-tasking**: use same base model with different adapters for different tasks.

## 6. Benefits vs Limitations

**Pros**

- **Memory-efficient** and fast training.
- Base model remains **unchanged**, enabling multiple adapters.
- Easy integration with pre-trained models.

**Cons**

- Only works well when **low-rank approximation is sufficient**.
- Might not capture highly complex changes needed for very different tasks.

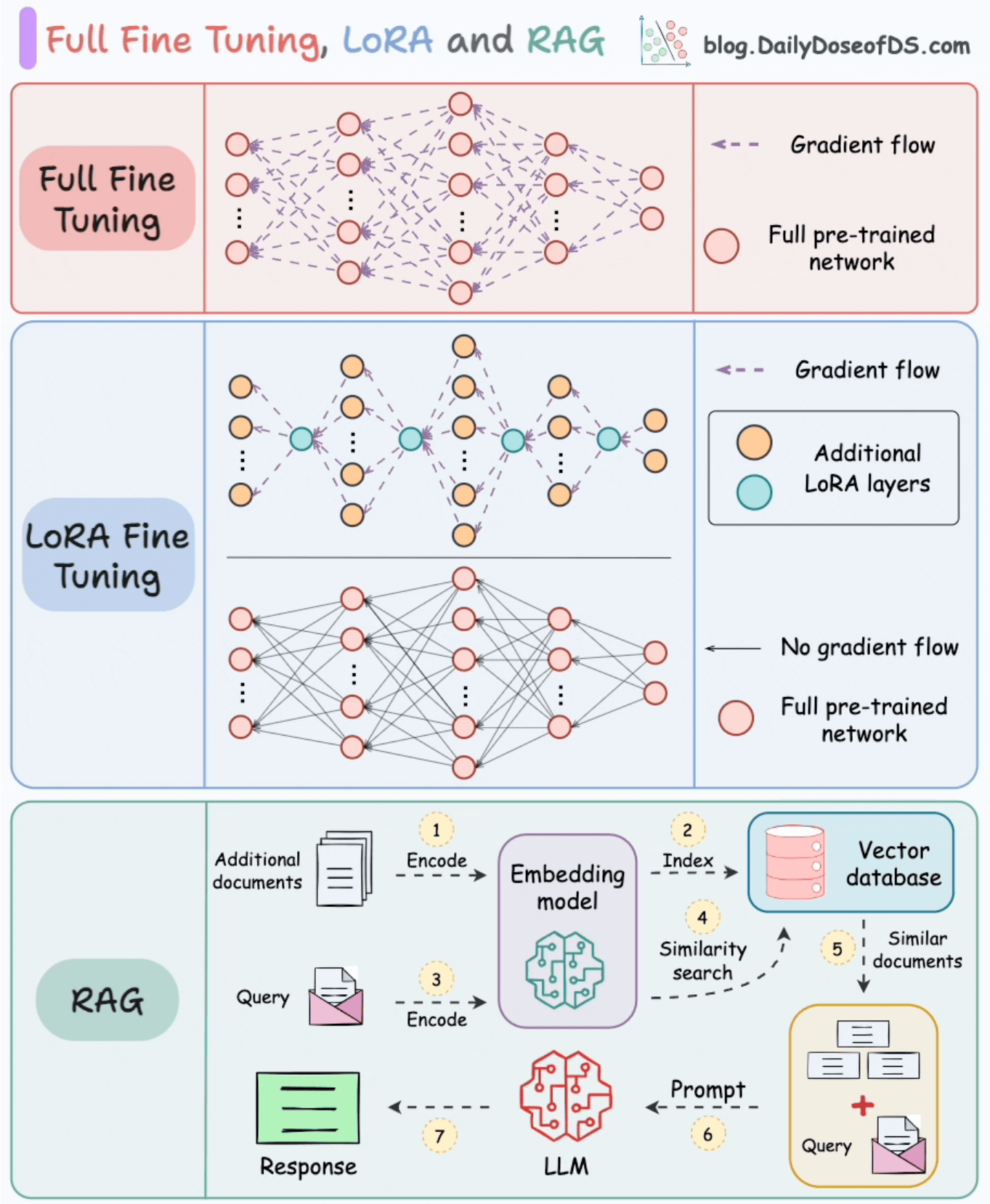
## 7. Summary in Simple Words

**LoRA Simplified**

You can think of LoRA as adding **tiny adapters on top of a frozen large model**. Imagine a giant model with billions of knobs. Fine-tuning everything is slow and expensive. LoRA adds a few small sliders (matrices) on top of the frozen model. During training, only these sliders move, but they are enough to guide the model to a new task.

## 8. Visualization Idea

You can include a visual showing LoRA adapters on top of a frozen model: Link



# LoRA Hyperparameters: $r$, $\alpha$, and Target Modules

LoRA (Low-Rank Adaptation) introduces a few crucial hyperparameters that control the learning capacity, scaling, and location of the adaptation. Understanding them is key to effective fine-tuning.

## 1. Rank $r$ (Low-Rank Dimension)

The hyperparameter $r$ determines the **rank of the low-rank matrices** $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$.

- Controls the number of parameters introduced in the LoRA adapter.
- Small $r \Rightarrow$ fewer parameters, less memory, faster training, but less expressive.
- Large $r \Rightarrow$ more parameters, higher capacity, but increased memory and computation.

**Rule of Thumb:**

Start with $r \in [4, 64]$ for LLMs. Increase $r$ if the task is complex or performance is insufficient.

## 2. Scaling Factor $\alpha$

The hyperparameter $\alpha$ scales the output of the LoRA adapter:

$$y = Wx + \frac{\alpha}{r}(ABx)$$

- Controls the **strength of the low-rank update** relative to the original weights.
- Small $\alpha \Rightarrow$ subtle influence, safer but slower adaptation.
- Large $\alpha \Rightarrow$ strong influence, faster adaptation, but may destabilize training.

**Rule of Thumb:**

Use $\alpha \in [8, 32]$ for LLMs. Adjust based on task difficulty and stability.

## 3. Target Modules

Target modules define **which layers of the base model receive LoRA adapters**.

- Common targets: attention layers (query/key/value), feed-forward layers.
- Choosing too many modules $\Rightarrow$ more parameters, higher memory.
- Choosing too few modules $\Rightarrow$ may underfit the task.

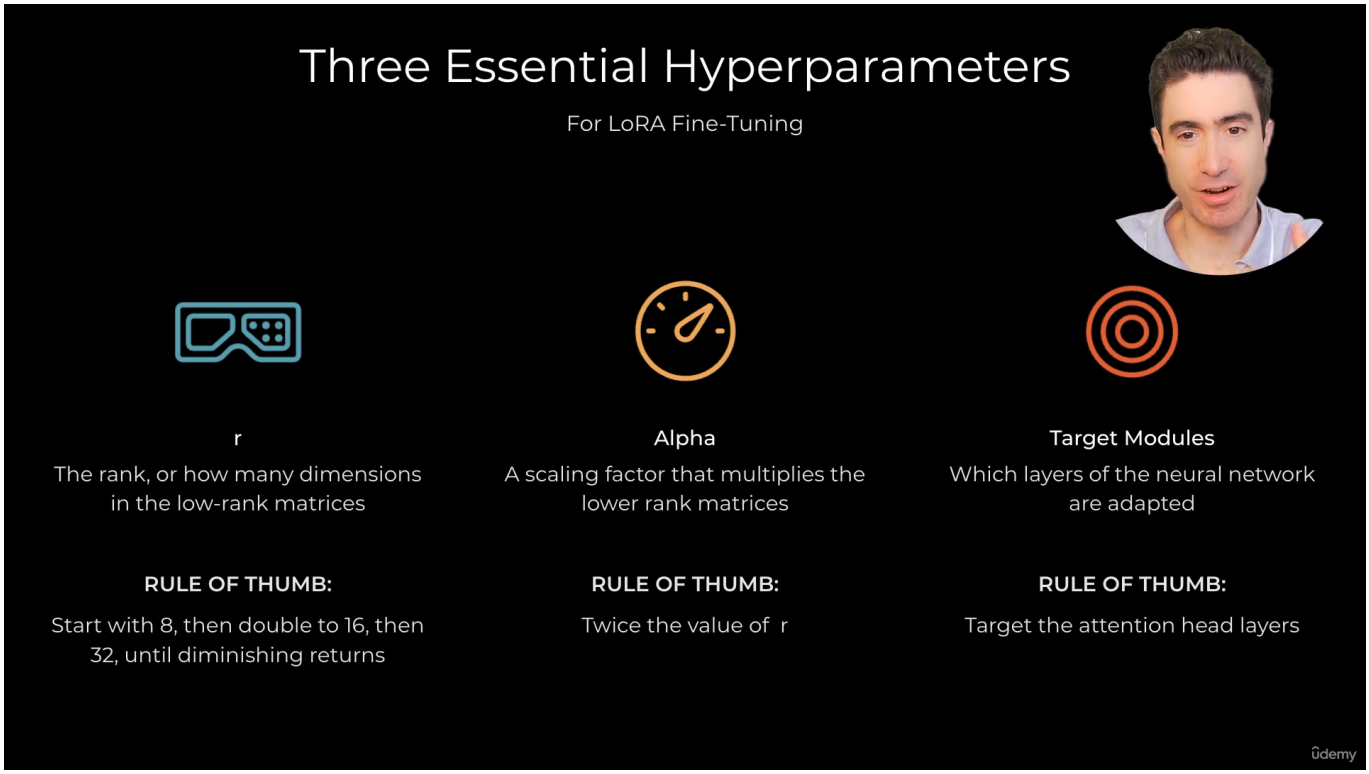
**Rule of Thumb:**

Start by adding LoRA to attention projection matrices (query and value) for LLMs. Expand to feed-forward layers only if necessary.

## Summary Table

**LoRA Hyperparameters Summary**

| Hyperparameter | Role | Rule of Thumb |
|---|---|---|
| $r$ | Low-rank dimension (number of adapter parameters) | 4–64 |
| $\alpha$ | Scaling factor of LoRA update | 8–32 |
| Target Modules | Layers where adapters are applied | Start with attention layers and then expand to feed-forward layers only if necessary. |



# Q, K, V, O in Transformers

In a transformer model, the attention mechanism allows each token to selectively focus on other tokens in the sequence. This is done using four key components: **Query (Q)**, **Key (K)**, **Value (V)**, and **Output (O)**.

- **Query (Q):** Represents the **questions** each token asks. For example, a token may ask, "Which other tokens are relevant to me?" Q is calculated by multiplying the token embedding with a learned weight matrix $W_Q$.
- **Key (K):** Represents the **tags or identifiers** of each token. It helps the Query figure out which tokens are relevant. K is computed by multiplying token embeddings with another learned weight matrix $W_K$.
- **Value (V):** Represents the **actual content or information** of each token. Once Q identifies which tokens are relevant (by comparing with K), V provides the information to pass forward. V is calculated using the weight matrix $W_V$.
- **Output (O):** After attention combines relevant V vectors, O is a projection that mixes the attended information back into the model's hidden dimension using the matrix $W_O$. This ensures the output can be passed to the next layer seamlessly.

The attention operation is mathematically defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

- $QK^\top$ computes the similarity between queries and keys.
- $d_k$ is the **dimension of the key vectors per attention head**. Dividing by $\sqrt{d_k}$ scales the dot product so that the softmax does not produce extremely small or large probabilities.
- Multiplying by $V$ gathers the relevant information based on these attention weights.

## Analogy: Classroom Example

**Analogy**

Imagine a classroom scenario where students ask and answer questions:

- **Q (Query):** A student asks a question to figure out which other students have relevant information.
- **K (Key):** Each student has a topic card that indicates what they know.
- **V (Value):** Each student provides an answer (information) if their topic matches the query.
- **O (Output):** The teacher collects all answers, summarizes them, and writes a combined response on the board.
- $d_k$**:** Think of it as the "attention focus resolution" — how detailed each student's topic card is. More details (larger $d_k$) require scaling to prevent overemphasis on any single answer.

This shows how Q identifies what to focus on, K helps determine relevance, V provides the information, and O produces the final output.

# LoRA in PeftModelForCausalLM

## Model Architecture Overview

The `PeftModelForCausalLM` wraps a `LlamaForCausalLM` base model and applies **LoRA adapters** for efficient fine-tuning.

- **Embedding Layer:** `embed_tokens` maps vocabulary tokens (128256) to embeddings of size 4096.
- **Decoder Layers:** 32 `LlamaDecoderLayer` modules, each containing:
  - **Self-Attention (`self_attn`)**
    - Q, K, V, O projections use `lora.Linear4bit` modules.
    - Each projection has:
      - `base_layer`: The frozen 4-bit linear layer representing the main weight matrix.
      - `lora_A`, `lora_B`: Low-rank matrices (rank $r = 32$) representing trainable updates.
      - `lora_dropout`: Dropout for regularization.
    - LoRA modifies only the projection weights, not the base model.
  - **Feed-Forward Network (MLP):** `LlamaMLP` with:
    - `gate_proj` and `up_proj` (expand embeddings to 14336)
    - `down_proj` (reduce back to 4096)
    - Non-linearity: `SiLU()`
  - **LayerNorms:** `input_layernorm` and `post_attention_layernorm`.
  - **Rotary Positional Embeddings:** `rotary_emb` for relative position encoding.
- **Final LayerNorm:** `norm` over the hidden dimension.
- **LM Head:** Linear mapping from hidden size 4096 to vocabulary size 128256.

## LoRA in Attention Layers

In transformers, attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^\top}{\sqrt{d_k}}\Big)V$$

where Q, K, V are projections of the input embeddings. LoRA modifies these projections as follows:

$$W_{\text{proj}} \to W_{\text{proj}} + \Delta W = W_{\text{proj}} + AB$$

- $W_{\text{proj}}$ is the frozen base projection weight (Linear4bit).
- $A \in \mathbb{R}^{d_{\text{model}} \times r}$, $B \in \mathbb{R}^{r \times d_{\text{proj}}}$, with $r \ll d_{\text{model}}$ (e.g., $r = 32$).
- Only $A$ and $B$ are trainable, drastically reducing the number of parameters.
- LoRA is applied individually to **Q, K, V, O** matrices:
  - `q_proj`: 4096 → 4096
  - `k_proj`: 4096 → 1024
  - `v_proj`: 4096 → 1024
  - `o_proj`: 4096 → 4096

### Why LoRA is Effective in Attention

- **Memory Efficiency:** Instead of updating the full weight matrices (4096×4096), only two small matrices of size 4096×32 and 32×4096 are trained per projection.
- **Task Adaptation:** LoRA can inject task-specific knowledge into the attention mechanism without modifying the base model.
- **Multiple Adapters:** Different LoRA adapters can be swapped for different tasks while keeping the same base model frozen.
- **Attention-Focused Learning:** By applying LoRA specifically to Q, K, V, O, the model learns to adjust how it attends to other tokens, which is sufficient for most fine-tuning tasks.

## Dimensions of Q, K, V, O Projections

**Attention Projection Dimensions**

| Projection | Input Features | Output Features |
|---|---|---|
| Q (`q_proj`) | 4096 | 4096 |
| K (`k_proj`) | 4096 | 1024 |
| V (`v_proj`) | 4096 | 1024 |
| O (`o_proj`) | 4096 | 4096 |

## Summary

- **LoRA** allows fine-tuning of very large models with minimal additional parameters.
- Applied to attention layers, LoRA adjusts how the model attends to input tokens.
- Memory-efficient and task-adaptive: only the low-rank matrices are trained, while the main model stays frozen.
- Facilitates multiple task-specific adapters without retraining the entire model.

### Key Points

- $d_{\text{model}}$ is the hidden dimension (4096 in this model) through which all tokens are projected.
- LoRA adds small trainable matrices (rank $r = 32$) to Q, K, V, O projections.
- Base projections remain frozen; only adapters are updated.
- Memory-efficient and task-adaptive fine-tuning.

# Loading a Base Model in 4-Bit and Using LoRA

The following code demonstrates how to load a large language model (LLM) in 4-bit precision for memory efficiency and later fine-tune it with LoRA.

```
quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_quant_type="nf4")

base_model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL,
    quantization_config=quant_config,
    device_map="auto",
)
```

## Explanation of the Code

- **BitsAndBytesConfig**: Configuration for quantization. Quantization reduces the precision of the weights to save memory.
  - `load_in_4bit=True`: Load model weights in 4-bit precision instead of 16-bit or 32-bit, drastically reducing memory usage.
  - `bnb_4bit_use_double_quant=True`: Uses double quantization to reduce numerical errors and improve accuracy.
  - `bnb_4bit_compute_dtype=torch.bfloat16`: Performs computations in bfloat16 to maintain performance while saving memory.
  - `bnb_4bit_quant_type="nf4"`: Specifies the 4-bit quantization type.
- **AutoModelForCausalLM.from_pretrained**: Loads the pre-trained LLM with the above quantization configuration.
  - `BASE_MODEL`: The pre-trained model identifier (e.g., LLaMA, Falcon).
  - `quantization_config=quant_config`: Uses the 4-bit quantized weights for memory efficiency.
  - `device_map="auto"`: Automatically maps model layers to available GPU(s) or CPU.

## How LoRA Optimizes This

- Loading the model in 4-bit reduces the memory footprint, but full fine-tuning is still expensive if we try to update all weights.
- LoRA introduces **small trainable adapters** (low-rank matrices $A$ and $B$) on top of the frozen base model.
- Only these adapters are updated during training, allowing:
  - **Memory efficiency**: base model weights are frozen and quantized.
  - **Faster training**: fewer parameters are trained.
  - **Task adaptation**: the model can specialize on a new task without full fine-tuning.
- Effectively, combining 4-bit quantization with LoRA allows training huge models on limited GPU memory.

## Summary

**Takeaway**

- **4-bit quantization**: reduces memory usage of base model weights.
- **LoRA adapters**: trainable low-rank matrices that update model behavior without touching the large frozen base model.
- **Combined benefit**: efficient fine-tuning on large models with low GPU memory and compute cost.

# Decision: Selecting a Base Model

## Goal

Choose a base model by balancing: **task requirements**, **inference cost (latency / GPU memory)**, **accuracy needs**, **license / ecosystem**, and **how you plan to fine-tune (LoRA, RAG, full FT)**.

## Decision criteria (short checklist)

- **Capability vs cost:** larger models usually perform better but cost more to run and host.
- **Parameter count:** affects memory, quantization decisions and latency.
- **Context window / multimodality:** needed for long-context tasks or image+text.
- **License and ecosystem:** open weights, available tooling (HF, vendor SDKs), community adapters.
- **Fine-tuning strategy:** if you plan to use LoRA, favor models with stable adapter support and documented PEFT integrations.
- **Deployment target:** edge / mobile vs single-GPU vs multi-GPU / cloud inference.

## Model-by-model short summary

- **LLaMA family (Meta)**: a family of open/available models offered in multiple sizes (smaller local-friendly models and very large variants for research). LLaMA models are widely used for research and community tooling; instruction-tuned variants ("instruct") are offered for easier instruction-following.
- **Qwen (Alibaba)**: a rapidly-evolving family with strong claims in multilingual and code tasks; recent releases include much larger variants (cutting-edge, often multimodal and optimized for large-scale deployments).
- **Phi (Microsoft / Phi family)**: smaller, efficient models (e.g., Phi-2 at a few billion parameters) that are optimized for reasoning and cost-effective deployments at smaller parameter counts.
- **Gemma (Google / DeepMind family)**: a family of lightweight-to-mid-size models intended for efficient, practical use (good on-device and cloud tradeoffs); models come in small-to-medium sizes optimized for production usage.

## Comparison table (high-level)

**Model comparison at a glance**

| Model | Representative sizes | Typical strengths | Suggested use-cases / notes |
|---|---|---|---|
| LLaMA (Meta) | 8B, 70B, (and larger research variants) | Strong general-purpose language and reasoning; broad community tooling | Good when you want high-quality open models and lots of community adapters; choose size by budget |
| Qwen (Alibaba) | Ranges from mid-size up to very large (vendor evolving family, latest pushes into extremely large sizes) | Good for code, multilingual tasks and large-scale deployments | Consider for enterprise cloud deployments or when vendor benchmarks favour Qwen for your task |
| Phi (Microsoft) | Small/medium (e.g., 2.7B for Phi-2) | Very cost-efficient; strong small-model reasoning per-cost | Excellent for edge or low-cost cloud inference where medium-sized models suffice |
| Gemma (Google / DeepMind) | Small → mid (family examples: 270M, 1B, 4B, 12B, 27B) | Efficient, production-ready; good on-device/cloud tradeoffs | Pick Gemma sizes for production where latency / cost matters and you require Google tooling |

## Base vs Instruct variants

- **Base (pretrained) model:** weights after pretraining on large corpora. Flexible — you can fine-tune, apply LoRA, or use RAG. Typically **better when you plan to adapt heavily** (custom fine-tuning / domain adaptation) because you control the adaptation pipeline.
- **Instruct (instruction-tuned) model:** a base model further fine-tuned on instruction/assistant datasets (and often RLHF/SFT). It usually performs better out-of-the-box on instruction-following tasks and reduces alignment/failure cases. Use an instruct variant if your primary need is production-ready instruction-following and you want fewer alignment steps.
- **LoRA note:** LoRA works on both base and instruct variants. If you want to minimize work, pick an instruct variant and add LoRA adapters for task-specific behavior; if you require maximal control, start from base and LoRA-fine-tune to your data.

## Rules of thumb and recommended workflows

**Quick rules of thumb**

- **If you have very limited memory / need on-device:** prefer Gemma small variants or tiny Phi/Gemma; aggressively quantize (4-bit) + light LoRA if you need task adaptation.
- **If you need a balance (best cost / capability):** 7–13B-class models (LLaMA 7/8B, mid Qwen/Gemma sizes) quantized + LoRA often give the best bang-for-buck for many production tasks.
- **If you need highest single-model quality:** 70B+ family (or enterprise Qwen large variants) — accept higher hosting cost; LoRA still helps but cost to run will be dominated by inference.
- **If you plan multi-task adapters / many small tasks:** choose a single base model and create multiple LoRA adapters — this saves storage and makes swapping behaviors trivial.
- **Prototype first:** benchmark a small/medium size with quantization + LoRA on a representative sample before upgrading to larger variants.

## Practical checklist before committing

1. Define success metric (accuracy, latency, throughput, cost per 1k tokens).
2. Check license / redistribution constraints for each model candidate.
3. Estimate inference cost (GPU memory + tokens/sec) for each model size.
4. Prototype: try quantized (4-bit) run + LoRA on a 1–2% validation slice and measure improvements.
5. If using instruct variants, validate whether additional LoRA actually improves or just overfits.

## Short recommendation examples

- Small startup / prototype with tight budget: Gemma-4B or Phi-2 (quantized) + LoRA for task-specific tuning.
- Production assistant with modest budget: LLaMA 7/8B (instruct) quantized + LoRA; scale to 13B if needed.
- Research / high-quality generation: LLaMA 70B (or larger Qwen top family); expect higher hosting cost — use LoRA for specialized tasks or multi-adapter workflows.

## Why LLaMA 3.1 was chosen: Tokenization Convenience

When selecting a base model, many factors matter: performance, parameter count, instruction-tuned vs base variant, and even subtler aspects like tokenization.

- **Tokenization strategy:** LLaMA 3.1 has a convenient tokenization scheme for numbers:
  - Numbers between 0 and 999 are represented as **single tokens**.
  - In contrast, other models like Qwen or Phi tokenize each digit separately. For example, "999" would become three tokens.
- **Why this matters:**
  - When training a model to predict numerical values (like prices), having the number as a single token simplifies the problem.
  - The model can predict the **entire number in one token**, rather than combining multiple digits.
  - This improves the model's ability to learn accurate token prediction and reduces the chance of errors (e.g., predicting $9 or $99 instead of $999).
- **Edge for LLaMA:** While it is a small difference overall, this tokenization convenience makes training slightly easier for numeric regression-like tasks.
- **Other considerations in model selection:**
  - Parameter sizes and memory requirements.
  - Instruction-tuned vs base variants.
  - Leaderboard performance.
  - Flexibility for future experiments with other models.
- **Decision:** For this project, the team chose **LLaMA 3.1 (8B or 18B variant)** as the base model, primarily because:
  - Convenient tokenization for numeric tasks.
  - Strong community support and tooling.
  - Good tradeoff between model size, performance, and fine-tuning options (LoRA can be applied easily).

**Summary**

**Takeaway**

Even small details, such as how numbers are tokenized, can influence model selection. For tasks like price prediction, LLaMA 3.1's ability to encode numbers up to 999 as single tokens provides a slight but useful advantage.

# Key Hyperparameters in QLoRA

QLoRA has a few essential hyperparameters that control how efficient and accurate the fine-tuning will be. Below are the five most important ones, explained in simple terms.

1. **Target Modules** LoRA does not modify the entire model. Instead, it inserts adapters only into specific layers, called **target modules**. These are usually the linear projections inside the attention mechanism:

   $$q\_proj, \quad k\_proj, \quad v\_proj, \quad o\_proj$$

   Why these? Because attention layers control how information flows between tokens, so small tweaks here can have a big impact.

   **Analogy:** Think of a giant office building (the LLM). Instead of renovating every single room, you only upgrade the **elevators and stairways** (attention layers), because they control how people (tokens) move around.

   **Rule of Thumb**

   Start with **Q (query)** and **V (value)** projections. Add to $k\_proj$ and $o\_proj$ only if the task is complex and requires more adaptation.

2. **Rank ($r$)** The rank $r$ decides the size of the low-rank adapter matrices $A$ and $B$:

   $$\Delta W \approx A \times B, \quad A \in \mathbb{R}^{d \times r}, \; B \in \mathbb{R}^{r \times k}$$

   A higher $r$ means more trainable parameters, giving more expressive power, but it costs more memory.

   **Analogy:** Imagine fitting a curve to data points. A small $r$ is like fitting a straight line (fast, but may miss details). A large $r$ is like fitting a flexible curve (captures more detail, but risks overfitting).

   **Rule of Thumb**

   Use $r = 8$ for small or domain-specific tasks. Use $r = 16$–$32$ for harder tasks like dialogue or reasoning.

3. **Alpha ($\alpha$)** Alpha scales the contribution of the LoRA adapter relative to the frozen base model:

   $$W' = W + \frac{\alpha}{r} AB$$

   If $\alpha$ is too low, the adapters barely affect the model. If too high, they overwhelm the base model.

   **Analogy:** Think of mixing paint colors. The frozen model is your base paint, and LoRA updates are small drops of dye. $\alpha$ controls how strong the dye looks in the final mix or how many drops of dye are used. If drops are too strong, the final color will be too bright and unnatural. If drops are too weak, the final color will be too dull and pale.

   **Rule of Thumb**

   Set $\alpha \approx 2r$. Example: $r = 16 \Rightarrow \alpha = 32$. This balances stability and expressiveness.

4. **Quantization** QLoRA stores the base model in **4-bit NF4 quantization** instead of 16/32-bit. This saves a huge amount of memory, making it possible to fine-tune large LLMs on a single GPU. The trick: LoRA adapters are still trained in higher precision (like 16-bit) while the frozen model stays compressed.

**Analogy:** Imagine keeping your huge library in a compressed digital format (4-bit). When you read, you only extract the parts you need at higher quality.

**Rule of Thumb**

Always use **4-bit NF4** quantization. It offers the best balance between compression and accuracy.

5. **Dropout** LoRA adapters add dropout to prevent overfitting. This randomly turns off some adapter neurons during training, making the model more robust.

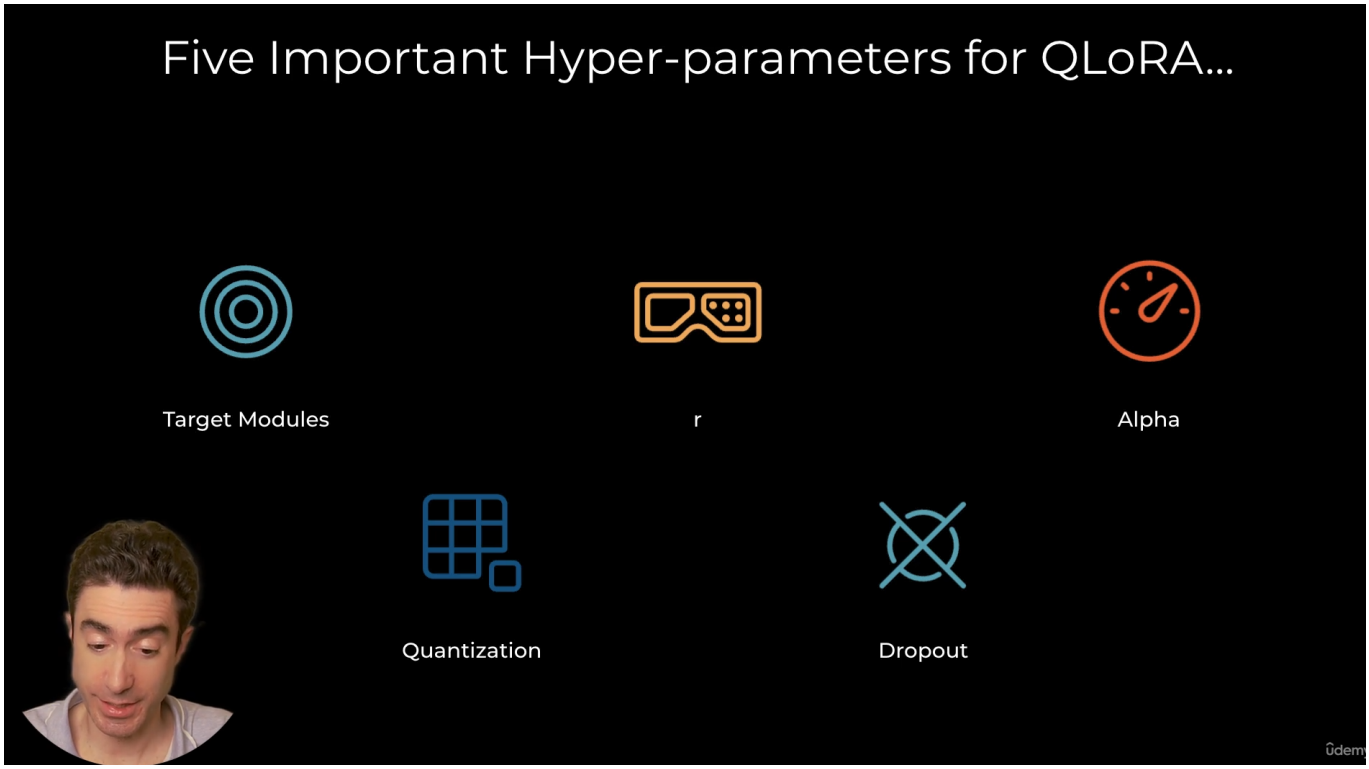
   **Analogy:** Like training a basketball team where sometimes players sit out, so others learn to play better. This prevents over-reliance on a few players.

   **Rule of Thumb**

   Use **0.05** dropout by default. Increase to **0.1** if your dataset is noisy or small.

## Quick Summary Table

| **Hyperparameter** | **Rule of Thumb** |
|---|---|
| Target Modules | Start with *q_proj*, *v_proj*; expand if needed. |
| Rank ($r$) | 8 for small tasks, 16–32 for harder ones. |
| Alpha ($\alpha$) | About $2r$ (e.g., $r = 16 \Rightarrow \alpha = 32$). |
| Quantization | Always use 4-bit NF4 for efficiency. |
| Dropout | 0.05 by default, 0.1 for noisy/small data. |



# Training Hyperparameters

## 1. Epochs

**Definition:** One epoch means the model has seen the *entire dataset once.*
**Why Important:**

- Too few epochs → the model underfits, meaning it hasn't learned enough patterns.
- Too many epochs → the model overfits, memorizing the training data and performing poorly on unseen data.

**Step-by-Step Example:**

1. Dataset has 10,000 samples.
2. Epoch 1: model sees all 10,000 samples → initial learning.
3. Epoch 2: model sees all 10,000 samples again → refines understanding.
4. Epoch 10: model may memorize exact data → risk of overfitting.

**Analogy:** Reading a textbook multiple times:

- 1 epoch = read once → understand a little.
- 5 epochs = read 5 times → remember more.
- 100 epochs = memorize word-for-word → may fail on questions requiring reasoning.

**Rule of Thumb**

Small datasets: 3–5 epochs. Large datasets: 1–3 epochs. Use **early stopping** to prevent overfitting.

## 2. Batch Size

**Definition:** Number of training samples processed together before updating model weights.
**Why Important:**

- Small batch size → noisy updates but better generalization.
- Large batch size → smoother updates, faster convergence, higher memory use.

**Step-by-Step Example:**

1. Batch size = 16 → model processes 16 samples, computes gradients, updates weights.
2. Batch size = 64 → model processes 64 samples, smoother gradient, fewer updates per epoch.

**Analogy:** Teacher giving feedback:

- Batch size 1 → feedback for each student immediately.
- Batch size 32 → teacher waits for all 32 students before giving feedback.

**Rule of Thumb**

Start with 16–64. If GPU memory is limited, use smaller batches + gradient accumulation.

## 3. Learning Rate

**Definition:** Step size for updating weights in the direction of the gradient.
**Why Important:**

- Too high → jumps over minima, training becomes unstable.
- Too low → very slow learning, may get stuck in sub-optimal minima.

**Step-by-Step Example:**

1. LR = 0.1 → model oscillates, can't converge.
2. LR = 0.00001 → very slow progress, may need many epochs.

**Analogy:** Learning to ride a bicycle:

- High LR → overcorrecting turns, wobble/fall.
- Low LR → move too slowly, take forever to balance.

**Rule of Thumb**

Fine-tuning: $2e{-}5$ to $5e{-}5$. Sensitive models: $1e{-}5$. Use **learning rate scheduler** to reduce LR over time.

## 4. Gradient Accumulation

**Definition:** Sum gradients from several mini-batches before performing a weight update.
**Why Important:**

- Saves GPU memory (can use smaller mini-batches).
- Stabilizes training, simulates larger batch size.

**Step-by-Step Example:**

1. Desired batch size = 64, GPU can only handle 16.
2. Feed 16 samples → compute gradients → do not update weights.
3. Repeat 3 more times → accumulate gradients.
4. Update weights once after 4 mini-batches.

**Analogy:** Saving money in a jar:

- Want to deposit $100 (effective batch) but have $25 daily (mini-batches).
- Save 4 days → deposit $100 once (weight update).

**Rule of Thumb**

Use accumulation steps = 2–8 if GPU memory is small. Effective batch size = batch size × accumulation steps.

## 5. Optimizer

**Definition:** Algorithm to update weights based on computed gradients.
**Why Important:**

- SGD: simple, stable, slow.
- Adam: adaptive, faster convergence.
- AdamW: Adam + weight decay → regularization, prevents overfitting.

**Analogy:** Driving style:

- SGD → manual car, slow and steady.
- Adam → modern automatic car.
- AdamW → modern car with lane assist (stays on track).

**Rule of Thumb**

Use AdamW, weight decay = 0.01, betas = (0.9, 0.999).

# DataCollatorForCompletionOnlyLM in HuggingFace TRL

## Problem

When training a language model to predict only a specific target (e.g., **price**), we do not want it to learn the entire product description.

**Example Input:**

Product: Apple iPhone 15 Pro Max, 512GB, Color: Silver. Price is $

- The context: `Product: Apple iPhone 15 Pro Max, 512GB, Color: Silver.`
- The target/completion: `1200` (what comes after "Price is $").

Without handling this properly, the model tries to predict the entire description, which is unnecessary.

## Traditional Approach (Complicated)

- Manually create a **mask** for the loss:
  - Tokens before "Price is $" → ignore in loss calculation.
  - Tokens after "Price is $" → calculate loss.
- This requires manually aligning token indices, which is error-prone.

## HuggingFace TRL Solution: DataCollatorForCompletionOnlyLM

Usage

```
from trl import DataCollatorForCompletionOnlyLM

response_template = "Price is $"
collator = DataCollatorForCompletionOnlyLM(
    response_template, tokenizer=tokenizer
)
```

**How it works:**

- `response_template` tells the collator where the context ends and the completion begins.
- Tokens before "Price is $" → set to `-100` in labels (ignored in loss).
- Tokens after "Price is $" → normal token IDs (used in loss computation).
- The model sees the full input but only learns from the completion part.

## Example

**Input text:**

Product: Apple iPhone 15 Pro Max, 512GB, Color: Silver. Price is $1200

**Tokenized (simplified):**

$$[\text{Product:}, \text{Apple}, \text{iPhone}, 15, \text{Pro}, \ldots, \text{Price}, \text{is}, \$, 1200]$$

**Labels (for loss):**

$$[-100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, 1200]$$

- `-100` → ignore in loss
- `1200` → used in loss

### Why it is important

- Ensures the model focuses only on the relevant tokens (price).
- Reduces wasted computation and speeds up training.
- Useful for task-specific fine-tuning such as price prediction, summarization, or Q&A.

**Summary**

- Collator automatically creates attention masks and labels.
- Context tokens are ignored in loss.
- Completion tokens are used for loss.
- The model learns the target task efficiently.

---

## Warmup Ratio (`warmup_ratio`)

**Simple Explanation:** Think of training a model like teaching a child to ride a bicycle.

- You don't let the child go full speed immediately.
- Instead, you start slow, then gradually increase the speed.

In training, the `learning rate` is like the speed. `warmup_ratio` decides how much of the training time we slowly increase the learning rate from 0 to the target value.

**Why it matters:**

- If the learning rate starts too high, the model can make big mistakes and become unstable.
- Gradually increasing it lets the model "warm up" and learn safely.
- After the warmup period, the model can learn faster without crashing.

**How it works:**

- Suppose total training steps = 1000.
- `warmup_ratio = 0.1` $\Rightarrow$ first 100 steps increase learning rate from 0 to normal.
- After 100 steps, the usual learning rate schedule (like linear or cosine decay) takes over.

**Without Warmup:**

- The model might start making huge updates.
- This can cause unstable training, slow convergence, or even NaN losses.
- Warmup prevents this "shock" at the start.

**Practical Tips:**

- Small warmup ratio (5-10%) is usually enough.
- For very large models, slightly longer warmup can help.
- Too long warmup slows down training unnecessarily.

**Easy Rule of Thumb**

- Start with 5% – 10% of total steps as warmup.
- Gradually increase the learning rate from 0 to your target.
- After warmup, let the usual scheduler handle learning rate decay.

# Checkpointing and Resuming Training in HuggingFace / Colab

## Overview

Checkpointing is the process of saving your model and training state periodically so you can resume training later without losing progress. This is especially useful in environments like Google Colab where the runtime is temporary and may disconnect.

**Why Checkpointing is Important**

- Saves model weights, optimizer state, and training progress.
- Allows training to resume after interruptions.
- Prevents loss of progress due to runtime disconnects or crashes.
- Makes training safer for long-running experiments.

## Checkpoint Components

A checkpoint typically contains:

- **Model weights:** `model.state_dict()` stores all parameters.
- **Optimizer state:** `optimizer.state_dict()` contains learning rates, momentum, etc.
- **Current epoch:** so training resumes from the correct point.

## Manual Checkpointing in PyTorch

**PyTorch Example**

```
import torch

# Save checkpoint
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict()
}, "checkpoint.pth")

# Load checkpoint
checkpoint = torch.load("checkpoint.pth")
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
start_epoch = checkpoint['epoch'] + 1
```

## Automatic Checkpointing with HuggingFace Trainer / SFTTrainer

HuggingFace Trainer or SFTTrainer can handle checkpointing automatically:

**TrainingArguments Example**

```
from transformers import TrainingArguments, SFTTrainer

training_args = TrainingArguments(
    output_dir="results",        # folder to save checkpoints
    num_train_epochs=5,
    per_device_train_batch_size=64,
    save_steps=100,                # save every 100 steps
    save_total_limit=3,            # keep only last 3 checkpoints
    logging_steps=50,
    evaluation_strategy="steps",
    eval_steps=50,
)

trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset
)

trainer.train(resume_from_checkpoint=True)  # resumes automatically
```

**Key points:**

- `save_steps`: after how many steps to save a checkpoint.
- `save_total_limit`: how many checkpoints to keep; older ones are deleted.
- `resume_from_checkpoint=True`: automatically resumes from the latest checkpoint.

## Checkpointing Flow Step-by-Step

1. **Start training:** check if checkpoint exists.
2. **Checkpoint exists:** load model weights, optimizer state, and last completed epoch.
3. **Set starting epoch:** `start_epoch = checkpoint['epoch'] + 1`.
4. **Resume training:** continue from the last saved epoch.
5. **Save checkpoint after each epoch or step:** ensures latest progress is always saved.

## Checkpointing in Google Colab

Colab's local filesystem is temporary. To preserve checkpoints:

**Resume Training in Colab**

- **Option A: Mount Google Drive**

```
from google.colab import drive
drive.mount('/content/drive')

training_args = TrainingArguments(
    output_dir="/content/drive/MyDrive/mnist_sft_results",
    num_train_epochs=5,
    per_device_train_batch_size=64,
    save_steps=100,
    save_total_limit=3
)
```

- **Option B: Copy checkpoints manually**

```
!cp -r /content/mnist_sft_results /content/drive/MyDrive/
```

**Resume Training in Colab**

After reconnecting to Colab:

- Mount Drive if using Option A.
- Make sure `output_dir` points to the same folder.
- Start training:

```
trainer.train(resume_from_checkpoint=True)
```

- Trainer automatically detects latest checkpoint and resumes training.

## Advantages of Using SFTTrainer Checkpointing

- Automatic saving of model, optimizer, scheduler, and training step.

- Easy resuming after runtime disconnects.
- Cleaner management of checkpoints using `save_total_limit`.
- Optional integration with Weights & Biases (`report_to="wandb"`) for real-time metrics.

**Summary**

- Checkpointing ensures training can continue after interruptions.
- HuggingFace Trainer / SFTTrainer makes checkpointing automatic.
- In Colab, save checkpoints to Google Drive for persistence.
- Use `resume_from_checkpoint=True` to continue training seamlessly.

## Checkpointing in Kaggle

Kaggle notebooks have a temporary filesystem similar to Colab. To preserve checkpoints:

**Important Note**

- Kaggle notebooks reset after the session ends.
- Local checkpoints stored in `/kaggle/working/` are temporary.
- To keep checkpoints permanently, download them manually or push them to Kaggle Datasets.

### 1. Manual Checkpointing

**Example: Save to Kaggle Working Folder**

```
checkpoint_path = "/kaggle/working/cnn_checkpoint.pth"
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict()
}, checkpoint_path)
```

### Downloading Checkpoints

To make the checkpoint persistent, download it after the notebook finishes:

**Download Example**

```
from IPython.display import FileLink
FileLink("/kaggle/working/cnn_checkpoint.pth")
```

### Resuming Training on Kaggle (Manual)

When restarting the notebook:

1. Upload your previously downloaded checkpoint back to `/kaggle/working/`.
2. Load it using `torch.load()` and restore model and optimizer states:

   **Example: SFTTrainer Automatic Checkpointing**

   ```
   checkpoint = torch.load("/kaggle/working/cnn_checkpoint.pth")
   model.load_state_dict(checkpoint['model_state_dict'])
   optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
   start_epoch = checkpoint['epoch'] + 1
   ```

3. Continue training from `start_epoch`.

### 2. Automatic Checkpointing with SFTTrainer

**Example: SFTTrainer Automatic Checkpointing**

```
from transformers import TrainingArguments, SFTTrainer

training_args = TrainingArguments(
    output_dir="/kaggle/working/mnist_sft_results",  # checkpoint folder
    num_train_epochs=5,
    per_device_train_batch_size=64,
    save_steps=100,          # save checkpoint every 100 steps
    save_total_limit=3,      # keep last 3 checkpoints
)

trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
)

# Resume training automatically if checkpoint exists
trainer.train(resume_from_checkpoint=True)
```

**Key points for SFTTrainer:**

- Checkpoints are stored in `/kaggle/working/mnist_sft_results/checkpoint-<step>`.
- Each checkpoint contains:
  - `pytorch_model.bin` – model weights
  - `optimizer.pt` – optimizer state
  - `scheduler.pt` – scheduler state
  - `trainer_state.json` – training step info
- Only the latest `save_total_limit` checkpoints are kept.
- To persist checkpoints beyond the Kaggle session, download them manually or save as a Kaggle Dataset.

**Summary for Kaggle**

**Summary**

- Kaggle filesystem is **temporary** (`/kaggle/working/`).
- **Manual checkpointing** gives full control but requires explicit save/load commands.
- **SFTTrainer checkpointing** is automatic and resumes training seamlessly.
- For long-term storage, either download checkpoints or save them as Kaggle Dataset.

# Saving and Downloading Automatic Checkpoints in Kaggle

Kaggle notebooks have a **temporary filesystem** at `/kaggle/working/`. Any files, including checkpoints, will be deleted after the session ends. To preserve your training progress, you should save checkpoints as a **zip file** and download it.

## When to Enter the Zip Command

After your training is complete, or at any point when you want to backup your checkpoints:

**Zip Checkpoints Command**

```
!zip -r /kaggle/working/mnist_sft_results.zip /kaggle/working/mnist_sft_results
```

**Explanation:**

- `zip` → creates a compressed archive.
- `-r` → includes all files and subfolders recursively.
- `/kaggle/working/mnist_sft_results.zip` → path and name of the zip file to create.
- `/kaggle/working/mnist_sft_results` → folder containing the automatic checkpoints from SFTTrainer.

## How to Download the Zip File

After creating the zip, you can download it directly from Kaggle:

**Download Example**

```
from IPython.display import FileLink
FileLink("/kaggle/working/mnist_sft_results.zip")
```

**Steps:**

1. Run the zip command after your training to package all checkpoints.
2. Use the `FileLink` command to generate a clickable link.
3. Click the link in the notebook output to download the zip file to your local machine.

## Restoring Checkpoints from Zip

Once downloaded, you can restore the checkpoints in a new Kaggle session or locally:

**Restore Example**

```
!unzip /kaggle/working/mnist_sft_results.zip -d /kaggle/working/

# Now you can resume training with SFTTrainer
trainer.train(resume_from_checkpoint="/kaggle/working/mnist_sft_results
/checkpoint-<step>")
```

**Notes:**

- The `-d` flag in `unzip` specifies the destination folder.

- Replace `<step>` with the checkpoint step you want to resume from.
- Using this method ensures your training progress is safe even after the Kaggle session ends.

# Saving and Resuming Checkpoints with Hugging Face Hub

## Manual Upload to Hugging Face Hub

You can also manually push any file (e.g., PyTorch `.pth` checkpoint) to Hugging Face Hub without using Trainer.

**Manual Push to Hugging Face Hub**

```
from huggingface_hub import Repository
import os

# Clone your Hugging Face repo (create one on HF website first)
repo_url = "https://huggingface.co/username/mnist-cnn"
repo_dir = "mnist-cnn-repo"

repo = Repository(local_dir=repo_dir, clone_from=repo_url)

# Copy checkpoint into the repo folder
os.system("cp /kaggle/working/cnn_checkpoint.pth mnist-cnn-repo/")

# Push checkpoint to Hugging Face Hub
repo.push_to_hub(commit_message="Added CNN checkpoint")
```

## Manual Download from Hugging Face Hub

To load a manually uploaded checkpoint:

**Manual Download Example**

```
from huggingface_hub import hf_hub_download

checkpoint_path = hf_hub_download(
    repo_id="username/mnist-cnn",
    filename="cnn_checkpoint.pth"
)
```

## Why Use Hugging Face Hub?

- **Persistent:** Unlike Kaggle/Colab, checkpoints are never lost.
- **Cross-platform:** Works seamlessly across Colab, Kaggle, and local machines.
- **Versioning:** Every push is tracked with history.
- **Private/Shared:** You can keep repos private or make them public.

**Summary**

Hugging Face Hub acts as a universal storage for models and checkpoints. You can:

- Use `push_to_hub=True` in Trainer for automatic checkpoint syncing.
- Manually push any file (model, tokenizer, optimizer states, datasets).
- Resume training from any machine with `resume_from_checkpoint=True`.

This makes training workflows more robust and eliminates worries about temporary storage.

# Automatic Checkpointing to Hugging Face Hub

This guide demonstrates a workflow to save and manage model checkpoints automatically on Hugging Face Hub during training.

## 1. Authenticate with Hugging Face

**Login to Hugging Face**

```
from huggingface_hub import login, create_repo, upload_file

# Login using your HF token
hf_token = "YOUR_HF_TOKEN"
login(hf_token, add_to_git_credential=True)

# Set project and run names
PROJECT_NAME = "your-project-name"
HF_USER = "your-username"
RUN_NAME = "YYYY-MM-DD_HH.MM.SS"  # Example: current datetime
REPO_ID = f"{HF_USER}/{PROJECT_NAME}-{RUN_NAME}"

# Create repo if it does not exist
create_repo(REPO_ID, exist_ok=True, private=True)
```

## 2. Define Model and Optimizer

**Example CNN Model**

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Device and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

## 3. Training Loop with Automatic HF Checkpointing

**Training with Auto-Save**

```
import os

# Local checkpoint folder
LOCAL_DIR = "checkpoints"
os.makedirs(LOCAL_DIR, exist_ok=True)
MAX_KEEP = 3
EPOCHS = 5

for epoch in range(1, EPOCHS+1):
    model.train()
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch} finished, loss={loss.item():.4f}")

    # Save checkpoint locally
    ckpt_name = f"checkpoint_epoch_{epoch}.pth"
    ckpt_path = os.path.join(LOCAL_DIR, ckpt_name)
    torch.save({
        "epoch": epoch,
        "model_state_dict": model.state_dict(),
        "optimizer_state_dict": optimizer.state_dict(),
    }, ckpt_path)

    # Upload checkpoint to Hugging Face Hub
    upload_file(
        path_or_fileobj=ckpt_path,
        path_in_repo=ckpt_name,
        repo_id=REPO_ID,
        commit_message=f"Upload checkpoint epoch {epoch}"
    )

    # Cleanup old checkpoints (keep only last MAX_KEEP)
    checkpoints = sorted([f for f in os.listdir(LOCAL_DIR)
     if f.startswith("checkpoint_epoch_")])
    if len(checkpoints) > MAX_KEEP:
        for old_ckpt in checkpoints[:-MAX_KEEP]:
            os.remove(os.path.join(LOCAL_DIR, old_ckpt))
            print(f"Deleted old checkpoint: {old_ckpt}")
```

## 4. Resuming Training from HF Hub

**Resume Training**

```
from huggingface_hub import hf_hub_download
import torch

# Download latest checkpoint from HF Hub
checkpoint_path = hf_hub_download(
    repo_id=REPO_ID,
    filename="checkpoint_epoch_5.pth"
)

# Load checkpoint
checkpoint = torch.load(checkpoint_path)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
start_epoch = checkpoint["epoch"] + 1

# Continue training from start_epoch
for epoch in range(start_epoch, EPOCHS+1):
    model.train()
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch} finished, loss={loss.item():.4f}")
    # Optionally repeat checkpoint upload here
```

## 5. Key Points / Best Practices

**Notes**

- **Persistent storage:** HF Hub stores checkpoints permanently and versioned.
- **Automatic upload:** Use `upload_file` after saving locally.
- **Manage local storage:** Keep only a few recent checkpoints to avoid using too much disk space.
- **Reusable workflow:** Change model, dataset, and `REPO_ID` to use this template for any project.

# Resuming Fine-Tuning with SFTTrainer and LoRA

This guide demonstrates how to resume fine-tuning a previously LoRA-adapted model using `SFTTrainer`, save to Hugging Face Hub, and perform inference.

## 1. Define Constants and Paths

**Project Paths and Model Info**

```
BASE_MODEL = "meta-llama/Meta-Llama-3.1-8B"
PROJECT_NAME = "pricer"
HF_USER = "sijanpaudel"

# Dataset on Hugging Face Hub
DATASET_NAME = f"{HF_USER}/pricer-data"

# Previously fine-tuned checkpoint to resume
ORIGINAL_FINETUNED = f"{HF_USER}/{PROJECT_NAME}-2024-09-13_13.04.39"
# previous fine-tuned model repo
# If you want to resume from a specific commit, set the revision
# Otherwise, set to None to use the latest
ORIGINAL_REVISION = "e8d637df551603dc86cd7a1598a8f44af4d7ae36" # commit hash
#ORIGINAL_REVISION = None


# Unique names for current run
RUN_NAME = f"{datetime.now():%Y-%m-%d_%H.%M.%S}"
PROJECT_RUN_NAME = f"{PROJECT_NAME}-{RUN_NAME}"
HUB_MODEL_NAME = f"{HF_USER}/{PROJECT_RUN_NAME}"
```

## 2. Login to HuggingFace Hub and Weights & Biases

**Authentication**

```
# HuggingFace login
hf_token = userdata.get('HF_TOKEN')
login(hf_token, add_to_git_credential=True)

# Weights & Biases login
wandb_api_key = userdata.get('WANDB_API_KEY')
os.environ["WANDB_API_KEY"] = wandb_api_key
wandb.login()
```

## 3. Load Dataset

**Load Dataset from Hub**

```
from datasets import load_dataset

dataset = load_dataset(DATASET_NAME)
train = dataset['train']
test  = dataset['test']
```

## 4. Configure LoRA and QLoRA Hyperparameters

LoRA and Quantization Settings

```
EPOCHS = 2
LORA_ALPHA = 64
LORA_R = 32
LORA_DROPOUT = 0.1
BATCH_SIZE = 16
GRADIENT_ACCUMULATION_STEPS = 1
LEARNING_RATE = 2e-5
LR_SCHEDULER_TYPE = 'cosine'
WEIGHT_DECAY = 0.001
TARGET_MODULES = ["q_proj", "v_proj", "k_proj", "o_proj"]
MAX_SEQUENCE_LENGTH = 182
QUANT_4_BIT = True
```

## 5. Load Tokenizer and Base Model (Quantized)

Tokenizer & Quantized Model

```
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

if QUANT_4_BIT:
    quant_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_compute_dtype=torch.bfloat16,
        bnb_4bit_quant_type="nf4"
    )
else:
    quant_config = BitsAndBytesConfig(
        load_in_8bit=True,
        bnb_8bit_compute_dtype=torch.bfloat16
    )

tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

base_model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL,
    quantization_config=quant_config,
    device_map="auto",
    use_cache=False
)
base_model.generation_config.pad_token_id = tokenizer.pad_token_id
```

## 6. Load Previously Fine-Tuned Model (PEFT)

**Resume LoRA Checkpoint**

```
from peft import PeftModel

if ORIGINAL_REVISION:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model,
        ORIGINAL_FINETUNED,
        revision=ORIGINAL_REVISION,
        is_trainable=True
    )
else:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model,
        ORIGINAL_FINETUNED,
        is_trainable=True
    )

fine_tuned_model.train()
```

## 7. Data Collator for Price Prediction

**Data Collator**

```
from trl import DataCollatorForCompletionOnlyLM

response_template = "Price is $"
collator = DataCollatorForCompletionOnlyLM(response_template,
tokenizer=tokenizer)
# The collator ensures only the part after the response_template is used for
loss calculation during training as tokenizer padding is on the right. The part
before response_template is the prompt which is only used for context(no tokens).
```

## 8. Configure SFTTrainer and LoRA

SFTTrainer Configuration

```
from peft import LoraConfig
from trl import SFTTrainer, SFTConfig

peft_parameters = LoraConfig(
    lora_alpha=LORA_ALPHA,
    lora_dropout=LORA_DROPOUT,
    r=LORA_R,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=TARGET_MODULES,
)

train_params = SFTConfig(
    output_dir=PROJECT_RUN_NAME,
    num_train_epochs=EPOCHS,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=1,
    eval_strategy="no",
    gradient_accumulation_steps=GRADIENT_ACCUMULATION_STEPS,
    optim="paged_adamw_32bit",
    save_steps=2000,
    save_total_limit=10,
    logging_steps=50,
    learning_rate=LEARNING_RATE,
    weight_decay=WEIGHT_DECAY,
    fp16=False,
    bf16=True,
    max_grad_norm=0.3,
    max_steps=-1,
    warmup_ratio=0.03,
    group_by_length=True,
    lr_scheduler_type=LR_SCHEDULER_TYPE,
    report_to="wandb",
    run_name=RUN_NAME,
    max_seq_length=MAX_SEQUENCE_LENGTH,
    dataset_text_field="text",
    save_strategy="steps",
    hub_strategy="every_save",
    push_to_hub=True,
    hub_model_id=HUB_MODEL_NAME,
    hub_private_repo=True
)
```

## 9. Initialize Trainer and Fine-Tune

**Fine-Tuning**

```
fine_tuning = SFTTrainer(
    model=fine_tuned_model,
    train_dataset=train,
    peft_config=peft_parameters,
    args=train_params,
    data_collator=collator,
)

fine_tuning.train()

# Push model to HuggingFace Hub
fine_tuning.model.push_to_hub(PROJECT_RUN_NAME, private=True)
```

## 10. Inference Example

**Inference Example**

```
prompt = "How much does this cost to the nearest dollar?\n\n
Power Stop Rear Z36 Truck and Tow Brake Kit with Calipers\n\n### Price:\n$"
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
outputs = fine_tuned_model.generate(**inputs, max_new_tokens=50)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

## Checkpointing in SFTTrainer with LoRA

**What Actually Happens During Training**

- Unlike standard HuggingFace Trainer, **no checkpoint-¡step¿ folders are automatically created**.
- Automatically saved files (locally or on Hugging Face Hub):
  - `adapter_model.safetensors` – LoRA adapter weights (trained parameters)
  - `training_args.bin` – training configuration (learning rate, optimizer type, scheduler type, etc.)
  - Tokenizer/config files if pushing to Hub (`tokenizer.json`, `tokenizer_config.json`, etc.)
- **Optimizer and scheduler states are NOT saved automatically**.
- To fully resume training with the same optimizer state and learning rate, manual saving is required.

## Inferencing using SFT

**Running Inference with Fine-tuned Model**

1. **Load the tokenizer and set padding:**

```
tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
```

*Explanation:* The tokenizer converts text into numerical tokens. We set the padding token to the EOS token to ensure proper alignment of inputs when batch processing sequences of different lengths.

2. **Load the base model:**

```
base_model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL,
    quantization_config=quant_config,
    device_map="auto",
)
base_model.generation_config.pad_token_id = tokenizer.pad_token_id
```

*Explanation:* Loads the pre-trained base model (e.g., LLaMA). Quantization reduces memory usage, and device_map=”auto” distributes the model across available GPUs. We also set the generation padding token to match the tokenizer.

3. **Attach the fine-tuned LoRA adapter:**

```
if REVISION:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model, FINETUNED_MODEL, revision=REVISION
    )
else:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model, FINETUNED_MODEL
    )
```

*Explanation:* LoRA adapters contain only the fine-tuned weights. **REVISION** is the commit hash of the checkpoint you want to use for inference. If you specify a revision, it loads that exact checkpoint from the Hub or local folder.

**Generating Text with the Fine-tuned Model**

**Generate predictions:**

```
def model_predict(input_text):
    inputs = tokenizer.encode(input_text, return_tensors="pt").to("cuda")
    attention_mask = torch.ones(inputs.shape, device="cuda")
    outputs = fine_tuned_model.generate(inputs,
    attention_mask=attention_mask,max_new_tokens=50, num_return_sequences=1)
    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return response
```

*Explanation:* Tokenized input is sent to the GPU. The `generate()` method produces text autoregressively. Finally, tokens are decoded back into readable text.

## Improved Prediction Function

**Weighted Top-K Prediction**

```
top_K = 3

def improved_model_predict(prompt, device="cuda"):
    set_seed(42)
    inputs = tokenizer.encode(prompt, return_tensors="pt").to(device)
    attention_mask = torch.ones(inputs.shape, device=device)

    with torch.no_grad():
        outputs = fine_tuned_model(inputs, attention_mask=attention_mask)
        next_token_logits = outputs.logits[:, -1, :].to('cpu')

    next_token_probs = F.softmax(next_token_logits, dim=-1)
    top_prob, top_token_id = next_token_probs.topk(top_K)
    prices, weights = [], []
    for i in range(top_K):
      predicted_token = tokenizer.decode(top_token_id[0][i])
      probability = top_prob[0][i]
      try:
        result = float(predicted_token)
      except ValueError as e:
        result = 0.0
      if result > 0:
        prices.append(result)
        weights.append(probability)
    if not prices:
      return 0.0, 0.0
    total = sum(weights)
    weighted_prices = [price * weight /
     total for price, weight in zip(prices, weights)]
    return sum(weighted_prices).item()
```

*Explanation:*

- This function predicts a numeric value (like a price) from a prompt.
- It uses the top **K=3** token probabilities and calculates a **weighted average**.
- The tokenizer decodes predicted tokens into numbers. Non-numeric tokens are ignored.
- Probabilities from `softmax` are used as weights for averaging.
- Returns a single float representing the **expected value** of the prediction.
- Setting a `seed` ensures reproducibility.

## Key Points for SFT Inference

**Important Notes**

- Always load the **base model** first; adapters cannot be used standalone.
- The **tokenizer** ensures inputs are properly formatted and padded.
- **PeftModel.from_pretrained** loads LoRA adapters from a checkpoint.
- **REVISION** specifies the exact checkpoint (commit hash) for inference.
- Use `generate()` for autoregressive text generation.
- This setup allows efficient inference using a large pre-trained model plus lightweight adapters, without retraining the base model.

## Summary

**Key Points**

- Local folder `PROJECT_RUN_NAME` stores checkpoints.
- `HUB_MODEL_NAME` is passed to `hub_model_id` for Hub upload.
- `SFTTrainer` automatically handles checkpoint saving, Hub push, and resuming.
- Use `resume_from_checkpoint=True` to continue training without losing progress.

# Four Steps in LoRA Training

LoRA (Low-Rank Adaptation) fine-tuning modifies the standard training process by introducing small trainable low-rank matrices into certain layers of the model (typically attention projections). The original large model weights remain frozen, and only these adapter parameters are updated. Below are the four main steps in LoRA training.

## 1. Forward Pass with LoRA

- Input $\mathbf{x} \in \mathbb{R}^n$ is passed through the pre-trained model.
- In a normal weight matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$, LoRA introduces a low-rank decomposition:

$$\Delta \mathbf{W} = \mathbf{B}\mathbf{A}, \quad \mathbf{A} \in \mathbb{R}^{r \times k},\ \mathbf{B} \in \mathbb{R}^{d \times r},\ r \ll \min(d, k)$$

- The effective weight during training is:

$$\mathbf{W}' = \mathbf{W} + \Delta \mathbf{W}$$

- Since $\mathbf{W}$ is frozen, only $\mathbf{A}$ and $\mathbf{B}$ contribute learnable changes.
- Prediction is computed as:

$$\hat{\mathbf{y}} = f_{\theta, \{\mathbf{A}, \mathbf{B}\}}(\mathbf{x})$$

## 2. Loss Calculation

- The loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ measures prediction error.
- Common choices:
  - Cross-Entropy Loss for language modeling:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

  - Other task-specific losses (e.g., regression, contrastive loss).
- The difference from standard training is that the loss only needs to drive updates to the LoRA parameters $(\mathbf{A}, \mathbf{B})$, not the entire model.

## 3. Backward Pass (Backpropagation with LoRA)

- Gradients are computed as usual via backpropagation:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{B}}$$

- Since $\mathbf{W}$ is frozen, its gradient is ignored:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = 0$$

- The effective gradient flow is restricted to the low-rank adapters.
- This significantly reduces memory and computation, as only a small fraction of parameters participate in updates.

## 4. Optimization (Updating LoRA Parameters)

- Only the adapter parameters $(\mathbf{A}, \mathbf{B})$ are updated:

$$\mathbf{A} \leftarrow \mathbf{A} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{A}}, \quad \mathbf{B} \leftarrow \mathbf{B} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{B}}$$

- Optimizers such as Adam or AdamW are commonly used.
- The frozen $\mathbf{W}$ remains unchanged throughout training, preserving pre-trained knowledge.
- This makes LoRA highly parameter-efficient: only a few million parameters are updated instead of billions.

## Summary Table of LoRA Training Steps

| **Step** | **Description** |
|---|---|
| Forward Pass (LoRA) | Compute predictions using $\mathbf{W}' = \mathbf{W} + \mathbf{BA}$, with $\mathbf{W}$ frozen and only $\mathbf{A}, \mathbf{B}$ trainable. |
| Loss Calculation | Compute loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ to measure prediction error, driving updates for LoRA parameters. |
| Backward Pass | Compute gradients only for $\mathbf{A}, \mathbf{B}$; gradients for frozen $\mathbf{W}$ are ignored. |
| Optimization | Update LoRA parameters $(\mathbf{A}, \mathbf{B})$ using gradient descent or Adam; $\mathbf{W}$ remains unchanged. |

## Uploading Final Dataset to Hugging Face Hub

**Dataset Upload Guide**

1. **Install required packages:**

```
!pip install datasets huggingface_hub
```

2. **Login to Hugging Face Hub:**

```
DATASET_NAME = "sijanpaudel/amazon-pricing-dataset"
from huggingface_hub import login
login()  # Enter your Hugging Face token
```

3. **Prepare your datasets:**

   - `final_train_set` should contain:
     - `text`: list of formatted prompts with price
     - `price`: corresponding price values
   - `final_test_set` should contain only `text` prompts (for evaluation)

4. **Convert dictionaries to Hugging Face Dataset objects:**

```
from datasets import Dataset
train_dataset = Dataset.from_dict(final_train_set)
test_dataset = Dataset.from_dict(final_test_set)
```

5. **Combine into a DatasetDict:**

```
from datasets import DatasetDict
dataset_dict = DatasetDict({
    "train": train_dataset,
    "test": test_dataset
})
```

6. **Push the dataset to Hugging Face Hub:**

```
dataset_dict.push_to_hub(DATASET_NAME, private=True)
```

7. **Verify upload:** After pushing, the dataset will be available at `https://huggingface.co/datasets/sijanpaudel/amazon-pricing-dataset`

8. **Pulling the dataset from Hugging Face Hub:**

```
from datasets import DatasetDict
dataset_dict = DatasetDict.load_from_hub(DATASET_NAME)
train, test = dataset_dict["train"], dataset_dict["test"]
```

# Multi-Agent AI Architecture for Automated Deal Finding Systems

This architecture describes a pipeline of specialized AI agents working in sequence to discover, analyze, and notify users about potential deals. The system follows a modular and agent-based approach to ensure scalability, flexibility, and reliability.

## User Interface (UI)

The entry point of the system is the **User Interface**.

- Typically implemented as a web application (e.g., using Gradio).
- Provides an accessible platform where users can interact with the system.
- Users can configure preferences, set deal criteria, and receive outputs.

## Agent Framework

At the core lies the **Agent Framework**, which functions as the infrastructure supporting all agents.

- Provides **memory** for agents to recall previous states, interactions, and contextual data.
- Maintains **logging** to record every step of agent activity for auditing, debugging, and transparency.
- Ensures inter-agent communication and workflow orchestration.

## Planning Agent

The **Planning Agent** acts as the central coordinator of the system.

- Determines which specialized agent should execute a task at any given time.
- Breaks down high-level objectives (e.g., "find and evaluate deals") into actionable sub-tasks.
- Manages sequencing, dependencies, and prioritization of agents.

## Scanner Agent

The **Scanner Agent** is responsible for initial discovery.

- Scans multiple structured and unstructured data sources (websites, APIs, databases).
- Applies filtering and heuristic rules to identify potentially valuable deals.
- Passes candidate deals to downstream agents for deeper analysis.

## Ensemble Agent

The **Ensemble Agent** carries out valuation and analysis.

- Uses multiple predictive models (*ensemble learning*) to estimate deal value.
- Combines outputs of different models to reduce bias and variance, leading to more robust results.
- Generates a final valuation score with confidence levels.

## Messaging Agent

The final stage involves the **Messaging Agent**.

- Prepares results for end-users after analysis.
- Sends notifications (e.g., push alerts, emails, or in-app messages) summarizing potential deals.
- Ensures timely communication so that users can take action quickly.

## Pipeline Summary

The flow of the system can be summarized as follows:

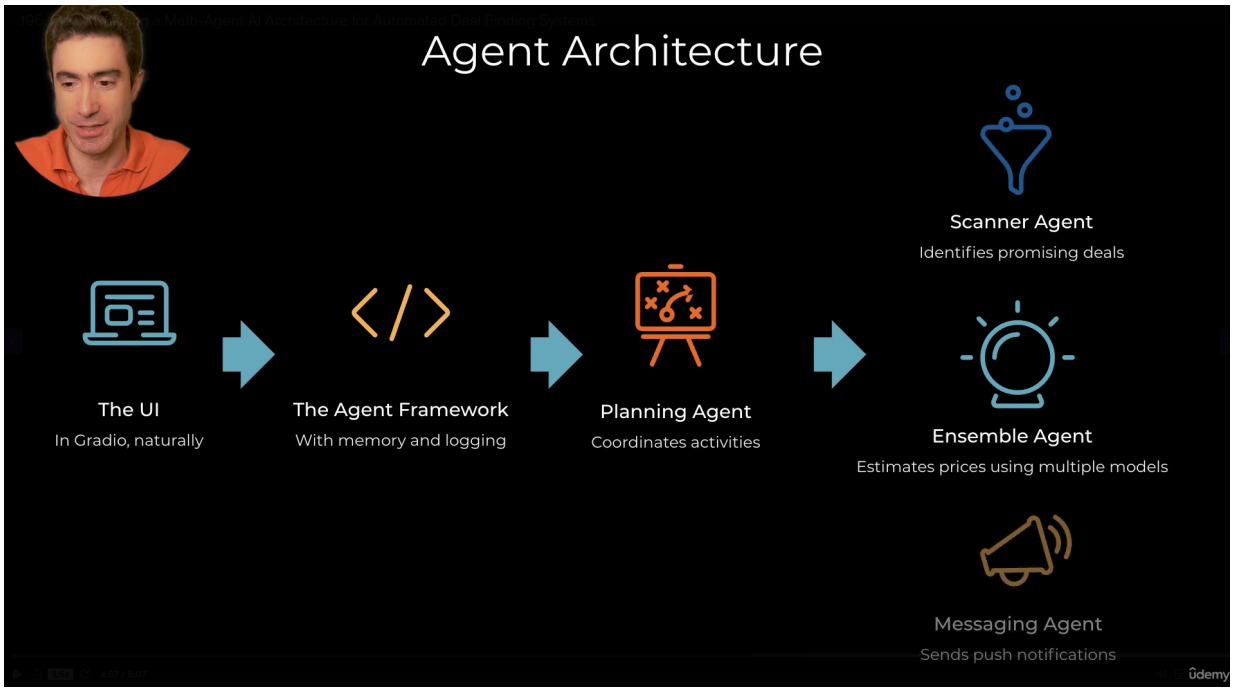
1. User interacts with the **UI** to configure requirements.
2. The **Agent Framework** initializes and monitors the workflow.
3. The **Planning Agent** orchestrates task execution.
4. The **Scanner Agent** discovers potential deals.
5. The **Ensemble Agent** evaluates deal value using multiple models.
6. The **Messaging Agent** notifies the user of results.

This modular multi-agent approach ensures that each agent specializes in one role, making the system scalable and maintainable while providing accurate and timely results.

## Future Work

Future enhancements to the system could include:

- Integrating more advanced machine learning models for better deal evaluation.
- Expanding the range of data sources for the Scanner Agent.
- Improving user interface elements for better user experience.




# Create a RAG Database with Our 400,000 Training Data

## PLEASE NOTE:

We already have a very powerful product estimator with our proprietary, fine-tuned LLM. Most people would be very satisfied with that! The main reason we're adding these extra steps is to deepen your expertise with RAG and with Agentic workflows.

**Step 1: Import necessary libraries. These include standard Python packages, numerical computing (numpy), data handling (pickle, datasets), Chroma DB for vector storage, HuggingFace tools, and embedding models.**

[ ]:
```
# imports
import os
import re
import math
import json
from tqdm import tqdm
import random
from dotenv import load_dotenv
from huggingface_hub import login
import numpy as np
import pickle
from sentence_transformers import SentenceTransformer, util
from datasets import load_dataset
import chromadb
from sklearn.manifold import TSNE
import plotly.graph_objects as go
```

**Step 2: Setup environment variables and database path for API keys and Chroma storage.**

[ ]:
```
# environment setup
load_dotenv(override=True)
os.environ['OPENAI_API_KEY'] = os.environ.get('OPENAI_API_KEY',␣
↪'your-key-if-not-using-env')
os.environ['HF_TOKEN'] = os.environ.get('HF_TOKEN', 'your-key-if-not-using-env')
DB = 'products_vectorstore'
```

**Step 3: Log in to HuggingFace hub using the token to access models and datasets.**

[ ]:
```
# HuggingFace login
hf_token = os.environ['HF_TOKEN']
login(hf_token, add_to_git_credential=True)
```

**Step 4: Load preprocessed training data from pickle files.**

[ ]:
```
# Load training data
with open('../week6/train.pkl', 'rb') as f:
    train = pickle.load(f)
```

**Step 5: Initialize Chroma persistent client, delete existing collection if any, and create a new collection for product vectors.**

[ ]:
```
# Create Chroma datastore
client = chromadb.PersistentClient(path=DB)
if 'products' in [c.name for c in client.list_collections()]:
    client.delete_collection('products')
collection = client.create_collection('products')
```

**Step 6: Load sentence transformer embedding model for vectorizing product descriptions.**

[ ]:
```
# Initialize embedding model
model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
```

**Step 7: Define a helper function to extract product descriptions from the training data.**

[ ]:
```
# Prepare text for vectorization
def description(item):
    return item.prompt.replace('How much does this cost to the nearest dollar?\n\n', '').split('\n\nPrice is $')[0]
```

**Step 8: Loop through training data, vectorize descriptions, and add them to the Chroma collection.**

[ ]:
```
# Populate Chroma collection
NUMBER_OF_DOCUMENTS = len(train)
for i in range(0, NUMBER_OF_DOCUMENTS, 1000):
    docs = [description(item) for item in train[i:i+1000]]
    vectors = model.encode(docs).astype(float).tolist()
    metas = [{'category': item.category, 'price': item.price} for item in train[i:i+1000]]
    ids = [f'doc_j' for j in range(i, i+len(docs))]
    collection.add(ids=ids, documents=docs, embeddings=vectors, metadatas=metas)
```

# Hallmarks of an Agentic AI Solution

## 1. Decomposition of Larger Problems into Smaller Steps

An **Agentic AI system** decomposes complex problems into smaller, manageable steps. Each step is handled by an *independent agent* or *specialized process*, allowing for:

- **Parallel processing** for faster execution.
- **Error reduction** by isolating tasks.
- **Task modularity** to reuse and recombine solutions.
- *Scalable problem solving* for multi-domain challenges.

This approach ensures that large goals can be achieved efficiently without overwhelming any single component.

## 2. Use of Tools, Function Calling, and Structured Outputs

Agents can call *external tools*, *APIs*, or defined functions to extend their capabilities. Structured outputs ensure clear and reliable results. Key benefits include:

- **Precision:** Outputs like JSON or tables can be directly processed.
- **Reliability:** Reduces ambiguity in multi-agent workflows.
- **Interoperability:** Allows agents to communicate results seamlessly.
- *Error handling:* Structured formats make it easier to validate and parse responses.

## 3. Collaborative Agent Environment

Agents operate in a shared environment enabling **collaboration and coordination**:

- **Specialized knowledge:** Different agents contribute unique skills.
- **Task distribution:** Efficient division of labor for complex goals.
- *Real-time communication:* Agents exchange updates to optimize outcomes.
- *Conflict resolution:* Conflicting suggestions can be reconciled by higher-level agents.

Collaboration enhances overall system performance beyond the capability of any single agent.

### 4. Planning and Coordination Agent

A **planning agent** orchestrates activities across all agents:

- Determines the sequence of tasks based on dependencies.
- Dynamically adjusts plans according to intermediate results.
- Prioritizes critical steps to achieve goals efficiently.
- Monitors progress and reallocates resources as needed.

This ensures that complex workflows are executed coherently and reliably.

### 5. Autonomy and Persistent Memory

Agentic AI demonstrates **autonomy** and maintains *persistent memory*, which allows:

- **Independent operation** without constant human intervention.
- **Learning from past interactions** to improve decision-making.
- Retention of critical information across sessions for long-term planning.
- *Cumulative knowledge growth*, enabling adaptation to new situations.
- Capability to handle multi-step and long-horizon tasks effectively.