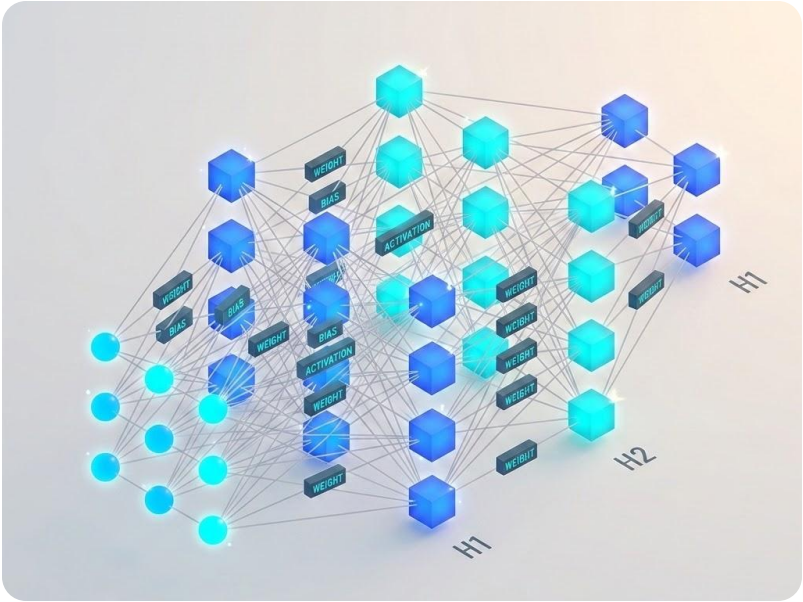
Night School: Fine-Tuning, PEFT, and QLoRA

When to Customize a Model and When Not To

Adjust the weights of a LLM for a **specific use case**.

This process embeds examples into the model's weights, **improving output relevance**.



Gauntlet

When to Consider Fine-Tuning

Fine-tuning adjusts a model's weights to better fit a specific use case, embedding examples directly into the model and reducing the need for prompts or system messages.

Use Case	Details	Considerations
Teaches Intuition Where Words Fall Short	Helps models develop an understanding that goes beyond explicit prompting.	Risk of overfitting to specific examples; may lose ability to handle diverse queries.
Trains Smaller Models to Perform Specific Tasks Better	Improves the efficiency and accuracy of smaller models for targeted tasks.	Fine-tuning may be time-consuming and computationally expensive.
Token Savings Due to Shorter Prompts	Reduces the number of tokens needed, leading to more concise cost savings.	May require significant effort to initially fine-tune; not always applicable to all tasks.
Narrows the Range of Possibilities	Focuses the model's output to be more relevant and aligned with specific use cases.	Might reduce the flexibility of the model; may need frequent updates for new data.

Check For Understanding: When should you use RAG versus fine-tuning a model?

Fine-tuning Overview

01. Select Model

Choose a pre-trained model that has been trained on large, diverse datasets.

02. Prepare Data

Gather and format a dataset specific to your task or domain to begin the fine-tuning process.

03. Adjust Weights

Weights are adjusted based on new data to refine the model's understanding.

Weights are numerical values representing relationships.

04. Validate

Assess performance through human validation to ensure accuracy and relevance.

Gauntlet

OpenAI Fine-Tuning API

Workflow Steps

1. **Prepare Data:** Format interactions into JSONL conversation logs.
2. **Upload:** Send training & validation files via the Files API.
3. **Fine-Tune:** Start a training job with specific hyperparameters like epochs.
4. **Deploy:** Use the custom model ID for inference.

What You Get

- A **Custom Model ID** specific to your organization.
- Steered behavior and specific output formatting.
- Embedded domain examples directly in weights.

What You Don't

- No access to model weights (intellectual property).
- Limited hyperparameter controls.
- No offline capability; model stays on OpenAI servers.

Cost Analysis

Fine-tuning is a premium service with distinct costs:

- **Upfront:** Charges for file storage and the training compute job.
- **Inference:** Typically costs **~2x more** per token than base models.
- **Why?** Less efficient caching and scaling for personalized model instances.

Gauntlet

Synthesize and Format Data

01. Synthesize

Define Generate Function: Invoke a more capable LLM to synthesize high-quality data for representative sample inputs.

02. Review

Curate Quality: Use an observability platform like **LangSmith** to log, review, and curate the synthesized output.

03. Format

OpenAI Standard: Ensure the output matches the JSONL format required for fine-tuning OpenAI models.

Sample OpenAI Fine-Tuning Format

```
{"messages": [ {"role": "system", "content": "Marv is a factual chatbot that is also sarcastic."},
{"role": "user", "content": "What's the capital of France?"}, {"role": "assistant", "content": "Paris,
as if everyone doesn't know that already."} ]}
```

Gauntlet

Fine-tune gpt-4o-mini

Use a training file to fine-tune OpenAI's `gpt-4o-mini` via the API. Access the complete step-by-step procedure here.

01. Setup & Init

- Import necessary modules
- Initialize OpenAI client with API key

02. Prepare Data

- Define training & validation filenames
- Write data into JSONL files

03. Upload Files

- Upload JSONL files to OpenAI
- Retrieve and log file IDs

04. Create Job

- Initialize fine-tuning job with hyperparameters
- Monitor job ID and initial status

05. Manage Events

- Setup signal handlers for interruptions
- Stream events and handle disconnects

06. Finalize

- Retrieve and save the final fine-tuned Model ID

Gauntlet

Model Selection

Selecting an LLM to fine-tune involves matching the model to specific needs. LLM benchmarks provide a standardized framework to compare models' capabilities in specific tasks. Evaluate LLMs on Chatbot Arena.

Task	Benchmark	Evaluation Criteria
Chatbot Assistance	ChatBot Arena, MT Bench	Conversational fluency (smooth and natural dialogue), task success rate (ability to achieve specific goals)
Language Understanding	MMLU, SuperGLUE	Task diversity (variety of tasks and domains), reasoning ability (depth of comprehension and logical thinking)
Reasoning	ARC, HellaSwag	Logical reasoning (ability to follow logical steps), commonsense performance (understanding everyday scenarios)
Coding	HumanEval, MBPP, SWE-bench	Code functionality (correctness and efficiency of generated code), problem-solving accuracy (ability to solve given problems)

Metrics Used for Comparing Performance Across Tasks

1. **Accuracy**: Percentage of correct answers.
2. **BLEU Score**: Alignment of generated text with human references.
3. **Perplexity**: Model's surprise/confusion; lower is better.
4. **Human Evaluation**: Expert judgment on quality, relevance, or coherence.

Reinforcement Learning with Human Feedback

RLHF incorporates human input to improve AI decision-making, ensuring models align with human preferences for more reliable results.

How RLHF Works: Step-by-Step

1

Pretraining
Begin with a pre-trained LLM like OpenAI's GPT-4o.

2

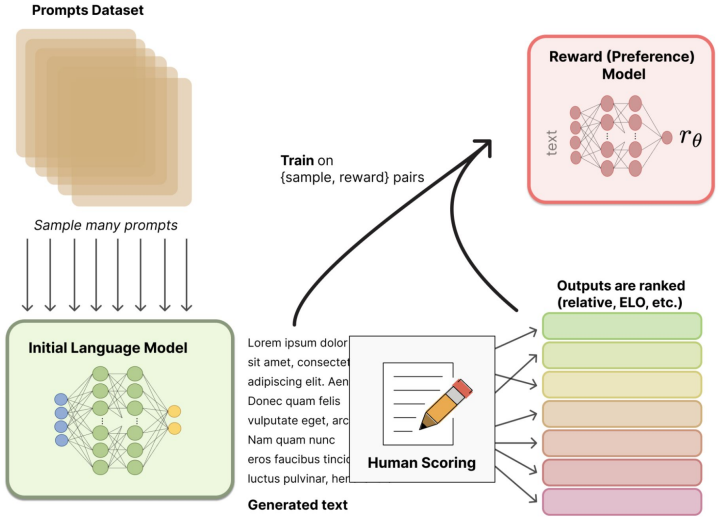
Human Feedback
Evaluators rank model outputs based on quality and alignment.

3

Reward Model Training
Use feedback to train a model that scores output alignment.

4

RL Fine-Tuning
Apply RL using the reward model to guide improvements.



Gauntlet

Gather Feedback Using LangSmith

The code shows how to implement a simple feedback mechanism using the LangSmith client to record and score human feedback on the model's responses.

Score Prompts

```
import os
from langsmith import Client

# Initialize the LangSmith client

client = Client()

def score_prompt(run_id, feedback):
  score = 1 if feedback == "Thumbs Up" else 0
  client.create_feedback(
run_id, f"Feedback{score}"
)
  return score
```

Example Usage

```
prompt = "Translate... I love programming."
run_id = "example_run_id"
feedback = "Thumbs Up"

response = invoke_chatopenai(prompt)
score = score_prompt(run_id, feedback)

print(f"Response: {response}")
print(f"Score: {score}")
```

PEFT

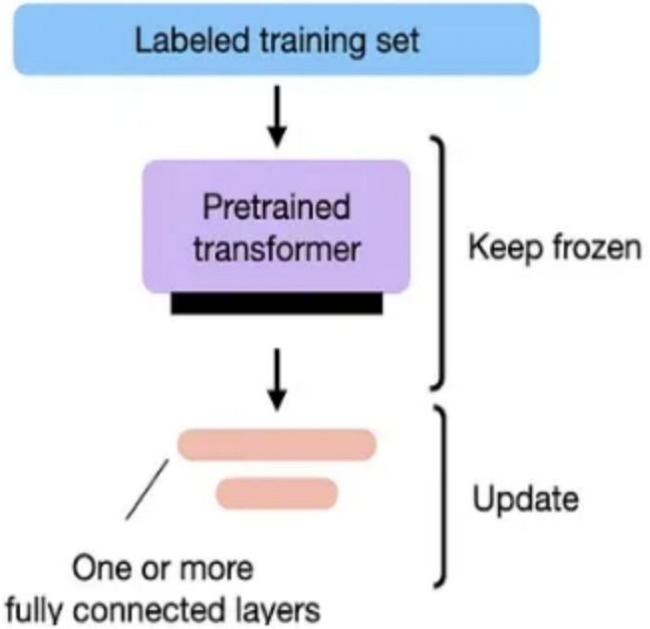
Parameter-Efficient Fine-Tuning (PEFT) is a method that fine-tunes a small subset of model parameters while keeping most of the pretrained model parameters frozen. This significantly reduces computational and storage costs, enabling the fine-tuning of large language models on consumer hardware.

Key Takeaways

- **Efficient Fine-Tuning**: PEFT updates a small subset of model parameters, reducing compute and storage requirements.
- **Cost-Effective**: Allows fine-tuning on consumer hardware, making advanced model training accessible.
- **Performance Maintenance**: Maintains the original model's performance while adapting to new tasks efficiently.

Considerations

- **Parameter Selection**: Careful selection of parameters to tune is crucial for optimal results.
- **Resource Management**: Balance computational savings with task-specific performance.

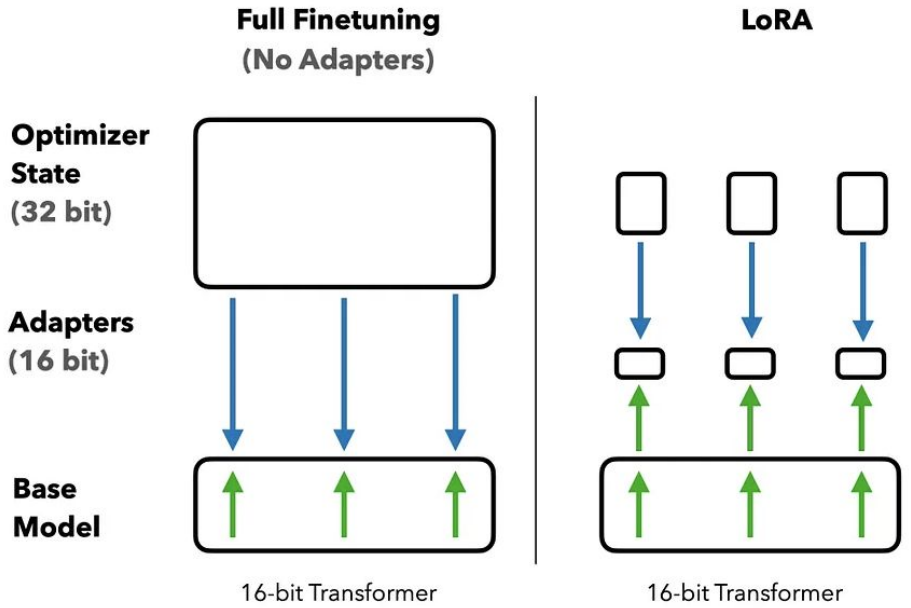


LoRA

Low-Rank Adaptation (LoRA) fine-tuning involves adding trainable low-rank matrices to the weights of a pre-trained LLM. These matrices are significantly smaller than the original weights, allowing only a fraction of the model's parameters to be updated during training.

How LoRA Works (Layman's Terms)

1. Start with a Pretrained Model: Begin with an AI model.
2. Add Extra Layers: Attach small, new layers (low-rank matrices) to the existing model.
3. Train the New Layers: Adjust these new layers while keeping the main model mostly the same. This is like tuning a few knobs without overhauling the machine.
4. Adapt to New Tasks: These small adjustments help the model learn new tasks quickly and efficiently.



Accessed 28 May 2024, <https://cameronrwolfe.substack.com/p/easily-train-a-specialized-llm-peft>.

Quantization

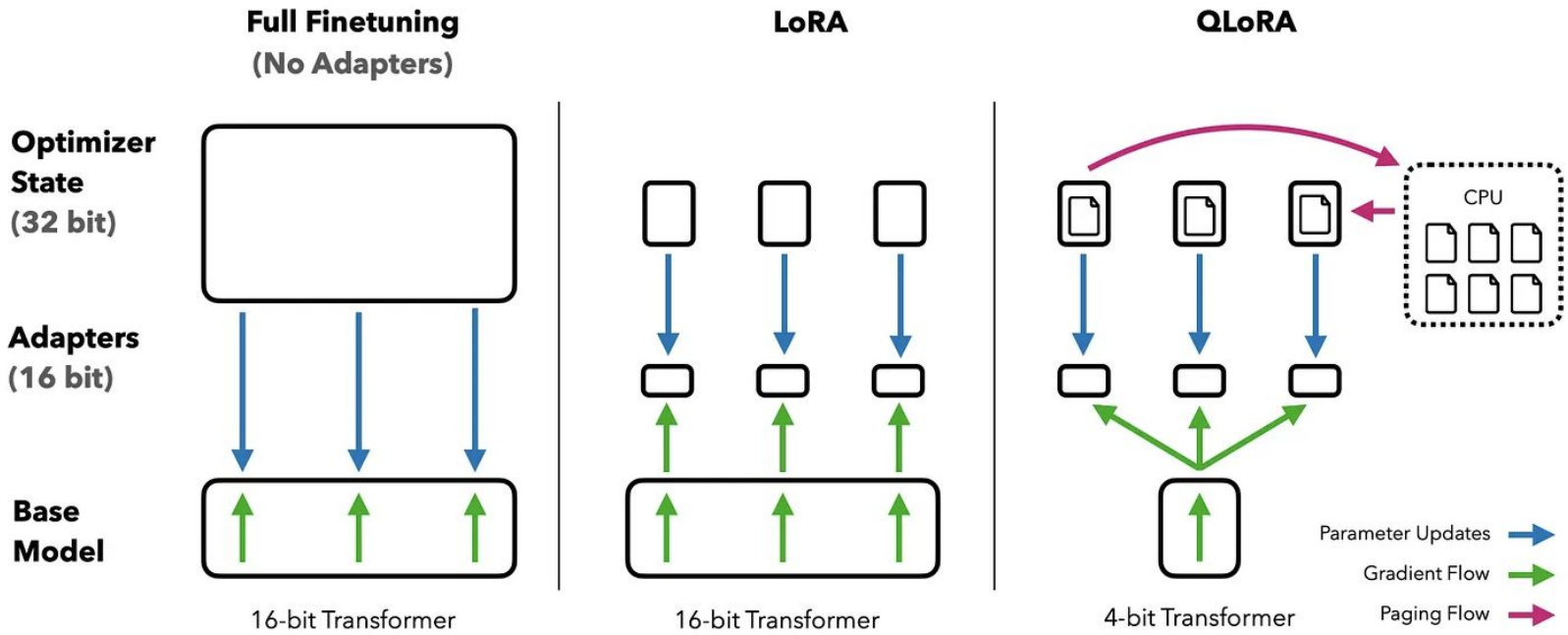
This process reduces the precision of a model's weights, typically from 32-bit floats to 8-bit integers, to make large language models more memory-efficient and faster. This technique enables the deployment of these models on less powerful hardware while maintaining performance.

Quantization and Its Use in QLoRA

- Reduced Precision: Quantization in QLoRA converts weights and activations to lower precision (e.g., 8-bit integers), significantly reducing memory usage.
- Efficient Fine-Tuning: By combining quantization with LoRA, QLoRA fine-tunes models using low-rank matrices, optimizing resource efficiency.
- Performance Retention: QLoRA maintains high performance by fine-tuning only the low-rank matrices while the quantized base model remains effective.
- Deployment on Limited Hardware: QLoRA enables the deployment of fine-tuned models on hardware with lower computational power, such as mobile devices and edge servers.

QLoRA

Quantized Low-Rank Adaptation (QLoRA) combines quantization with Low-Rank Adaptation (LoRA) to efficiently fine-tune large language models by converting weights and activations to lower precision.



Accessed 28 May 2024, <https://cameronrwolfe.substack.com/p/easily-train-a-specialized-llm-peft>.

Summary of PEFT, LoRa, and QLoRA

	PEFT (Parameter-Efficient Fine-Tuning)	LoRa (Low-Rank Adaptation)	QLoRA (Quantized Low-Rank Adaptation)
Description	Updates a small subset of model parameters to reduce costs.	Injects low-rank matrices into model layers to adapt new tasks.	Combines quantization with low-rank adaptation for tuning.
Method	Selectively fine-tunes specific parameters.	Adds low-rank matrices to the model's architecture.	Applies quantization to weights and activations, then uses LoRA.
Benefits	Less resource-intensive, faster fine-tuning process.	Maintains performance with minimal additional parameters.	Lowers compute requirements while preserving performance.
Use Cases	Scenarios requiring efficient fine-tuning with limited resources.	Tasks needing quick adaptation of large models without retraining.	Tuning LLMs on limited hardware while ensuring efficiency.
Challenges	May require careful selection of parameters to tune.	Complexity in integrating low-rank matrices.	Balancing quantization precision and adaptation effectiveness.

Fine-Tuning Llama2 with PEFT and QLoRA in Colab

Let's walk through fine-tuning Llama2 using the techniques we learned in class. Make sure your Colab runtime is set to `T4 GPU`. Access the complete code here.

Prerequisites Before You Explore Code

1. **Understand Hyperparameters**: Hyperparameters are the "magic constants" that must be set before training or fine-tuning a model. Unlike regular parameters, hyperparameters are chosen by humans.
2. Key Hyperparameters to Consider:
    a. **Learning Rate**: Determines how quickly the model adjusts its parameters during training.
    b. **Dropout**: Helps prevent overfitting by randomly setting a fraction of input units to zero during training.
    c. **Alpha**: A parameter that can vary in definition depending on the context but is crucial in some models.
3. Monitoring and Validation
    a. **Tracking Progress**: Tools like TensorBoard help monitor the loss over time. As long as the loss is decreasing, it's reasonable to continue training.
    b. **Validation**: Hyperparameter tuning underscores the importance of using holdout data to validate the model. This prevents overfitting to the training data and ensures better generalization.