LAMINI

Enterprise Guide to Fine-Tuning

Improving LLM Accuracy To
Unlock High Value Use Cases

Contents

Introduction

Large language models (LLMs) continue to improve across multiple modalities (text, image, audio, video) and tasks (code generation, translation, chat, math, reasoning, etc). Despite their growing sophistication, LLMs still have significant knowledge gaps and often hallucinate responses. For LLM applications that require expertise and accuracy, getting the answer right half of the time simply isn't good enough.

LLMs are trained on a large corpora of publicly available data giving them broad general knowledge. However, your most valuable business data isn't—and shouldn't be—on the internet. With LLMs, your expertise and data is now more valuable than ever because you can transform it into intelligence. Fine-tuning is key to unlocking your highest value use cases. Specifically, fine-tuning enables you to:

- Create smaller specialized models tuned to proprietary data with appropriate guardrails
- Reduce computational and overhead costs
- Decrease latency for better customer experience
- Significantly increase accuracy, unlocking higher value use cases

Many misconceptions surround fine-tuning: it's too hard, too expensive, and requires extensive labeled data plus a team of Open AI-level experts. We're here to debunk those myths and demonstrate how quickly you can get started and see results.

What is a large language model (LLM)?

An LLM is a deep learning model that has been trained on a large corpus of language data and can generate human-like text. For example, GPT3 was trained on 45TB of text scraped from the internet. The final model is 175B parameters. Parameters are the "dials" within the model that determine the LLM's behavior and output. Generally speaking, the more parameters a model has, the more complex operations it can

manage. LLMs rely on transformer architecture which is a way of processing text to derive the next most statistically probable unit of text, which can be a whole word or a word fragment. These units of text are called tokens.

Why do LLMs hallucinate?

LLMs are trained on diverse data sources that may contain inaccurate, conflicting, and biased information. Each LLM has a "knowledge cutoff"—the date its training data was gathered. Their knowledge remains frozen after training and the data naturally goes stale over time.

Unlike deterministic systems that follow fixed rules to produce consistent outputs, LLMs are probabilistic. They generate responses by calculating probabilities for each possible next word based on context and training data. LLMs are trained to reduce the average error of its generated text to best match the examples they've seen.

LLMs are pretty good at everything, but perfect at nothing.

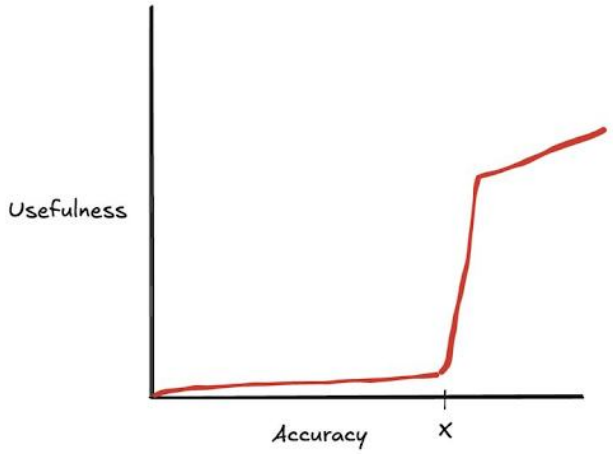
Finally, confusing or unclear prompts could steer the LLM to an inaccurate response.

What are LLM hallucinations?
Hallucinations refer to LLM outputs that are incorrect or made up. This can consist of factually inaccurate responses, made-up answers, ignoring instructions given by the user, logic errors, and more.

The problem with LLM hallucinations

The power of LLMs lies in their ability to generalize across many topics. However, generalization is problematic when applications require high precision. For instance, a customer service chatbot shouldn't make up account details, nor should a product recommendation engine suggest items unavailable in the product catalog. Such inaccuracies erode user trust in AI systems.

LLM applications become truly valuable only when they reach a tipping point of accuracy—what we call the usefulness-to-accuracy threshold. Credit to Adrian Parlow, co-founder and CEO of PointOne, for illustrating this concept.



In industries with low tolerance for error—such as financial services, insurance, legal tech, retail, and healthcare—LLM applications offer no practical value until they reach a high bar of accuracy. In many cases, that means 9's of accuracy.

If your bar for accuracy is 95 percent, then anything below that offers 0 percent usefulness. Consider legal contract review: if AI identifies only 50 percent of the redlines, an attorney must still do a full manual review. If your users and organization are more tolerant of hallucinations, then you can get away with a lower bar of accuracy.

Common methods for reducing LLM hallucinations

There are three methods for reducing LLM hallucinations: prompt engineering, Retrieval Augmented Generation (RAG), and fine-tuning.

Prompt Engineering

Prompt Engineering refers to creating and refining prompts to guide LLMs towards desired outputs using their pre-trained knowledge. To reduce hallucinations, you can include system prompts that restrict the model from sharing unverifiable information. While few-shot prompting (providing specific examples) and chain-of-thought prompting (including step-by-step reasoning instructions) can improve accuracy, they may also increase hallucinations in some cases.

Retrieval Augmented Generation (RAG)

RAG augments the model's pre-trained knowledge with proprietary or external data sources. RAG is great for real-time information retrieval. However, there are a few considerations:

- Cost - RAG is generally more accurate with large models. It requires longer prompts, and therefore models with larger context windows which increases both inference latency and cost.
- Complexity - Advanced applications need a production-grade RAG system to convert text into embeddings (numbers that represent the meaning of the text), store embeddings in a vector database, retrieve relevant documents through nearest neighbor search, augment prompts with retrieved information, and then generate responses in the desired format.
- Accuracy - RAG relies on similarity search which may retrieve irrelevant or conflicting information that can trigger hallucinations.

Fine-tuning

Fine-tuning adapts a pretrained model for specific tasks or domains. This technique has shaped many familiar AI tools—ChatGPT emerged from GPT's instruction fine-tuning, and GitHub Copilot was created by fine-tuning GPT-4. Common industry

examples include:

- Developing specialized legal models from GPT-4 for case law analysis (see the OpenAI Harvey case study)
- Creating customer service chatbots from open source models using proprietary knowledge bases
- Optimizing models for niche programming languages to enhance code generation and debugging
- Training models to reason across multiple documents and deliver factual responses to queries

How does fine-tuning work?

Unlike prompting and RAG, fine-tuning modifies the model's weights (parameters)—essentially updating its "brain" with new knowledge. This process enhances the model's ability to recall and apply specific information accurately.

Common fine-tuning methods:

- **Full fine-tuning** - Updates all model parameters which makes it computationally expensive, time-consuming and storage-heavy.
- **Transfer learning (feature extraction)** - Modifies only the final layers of the model while keeping earlier layers frozen, allowing feature extraction for specific tasks.
- **Parameter Efficient Fine-Tuning (PEFT)** - Adds and trains new parameters while keeping the original model weights frozen. Often implemented with Low-Rank Adaptation (LoRA) to minimize trainable parameters.

Benefits of fine-tuning

Research has shown that smaller fine-turned models can outperform a large general purpose model on specific tasks. For example, InstructGPT with its 1.3B parameters performs well at following instructions compared to GPT-3 with 175B parameters. See the table below for the pros and cons of each approach.

What's the best method to achieve high accuracy?

The method you choose depends on the additional context you need, the desired behavior or output, and the accuracy requirements of your application. Generally, it's best to start with prompting and RAG and consider fine-tuning only if these methods don't meet your accuracy goals.

	Prompt Engineering	RAG	Fine-tuning
Pros	• No data needed to start • Smaller upfront cost • No technical knowledge required • Connects data through RAG	• Enhances prompt engineering by retrieving relevant data • Reduces hallucinations compared to pure prompting • Useful for accessing dynamic or vast external knowledge	• Nearly unlimited data fits • Learns new information • Corrects incorrect information • Lower cost after initial setup with smaller models • Integrates with RAG for enhanced accuracy • Supports more high-quality data
Cons	• Prone to forgetting data • Hallucinations • Limited by the quality of prompts	• Can still miss or retrieve incorrect data • Dependence on the quality of the retrieval system	• Higher upfront compute cost • Requires technical knowledge, especially in data handling • Needs significant data preparation and management
Use Cases	• Generic applications that use an LLM's pre-trained knowledge • Side projects / Prototypes	• Applications needing real-time or dynamic data • Situations where external or proprietary information is vital	• Domain-specific applications • Enterprise and production usage • Privacy-sensitive applications

To enhance the accuracy of your LLM applications, we've developed a fully integrated platform offering two solutions: Lamini Memory RAG and Memory Tuning.

Lamini Memory RAG allows you to upload and chunk documents, automatically generating relevant input-output pairs. It is optimized for high performance and fault tolerance. If you'd like to learn more about Memory RAG, please contact us.

The rest of this guide will focus on Memory Tuning, our proprietary fine-tuning method that enables you to reach 9s of accuracy.

Lamini Memory Tuning

Lamini Memory Tuning is a breakthrough technology that can reduce hallucinations by 95 percent or more. It enables you to directly embed expertise and memory layers into an LLM while lowering computational cost. Designed for efficiency and accessibility, Memory Tuning makes fine-tuning faster, more affordable, and available to developers of all levels.

This innovative approach combines two advanced techniques—LoRAs (Low-Rank Adaptation) and MoE (Mixture of Experts)—to create the Mixture of Memory Experts (MoME) model, pronounced “mommy.”

LoRAs are trainable low-rank matrices (weights) that can be added to a base LLM. These matrices are lightweight, enabling the model to quickly learn new knowledge while keeping training costs and latency low. Additionally, LoRAs can be stored separately from the main model, making them both efficient and secure.

MoE refers to a model architecture composed of millions of specialized networks, or "experts," rather than a single monolithic network. User inputs are dynamically routed to a small subset of these experts.

How does MoMe work?

Memory Tuning embeds memory and expertise into any open LLM by training millions of LoRA adapters and dynamically selecting the most relevant experts at inference time. Instead of optimizing for average error on everything, Memory Tuning optimizes for near-zero error on specific facts, so it recalls those facts perfectly while still allowing for the LLM to generalize with average error on everything else. It changes the paradigm to make the LLM near perfect on facts, and still pretty good at everything else.

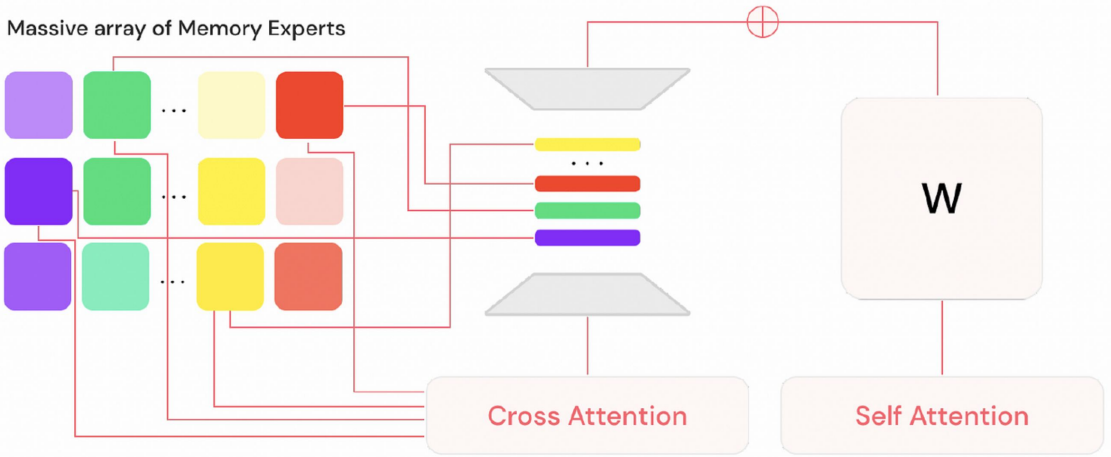
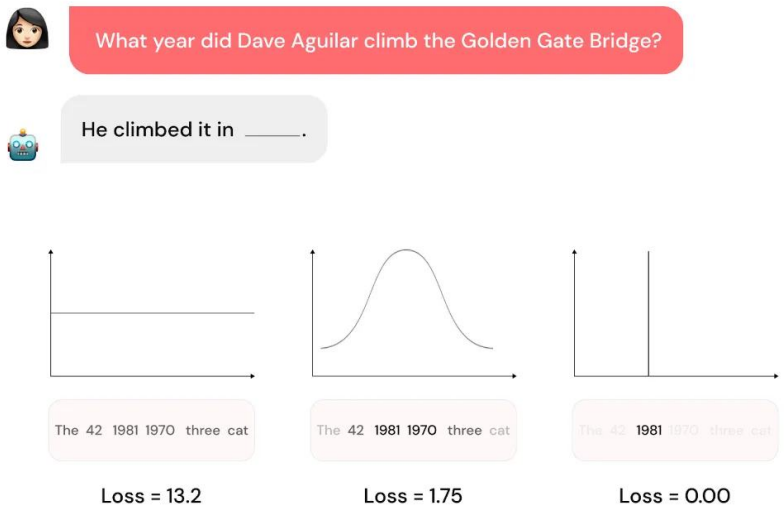


Figure 4: Lamini-1 architecture

As illustrated below, with Memory Tuning, the training error (loss) is zero when the model is supposed to recall a specific fact, so the model selects the exact right token, eliminating hallucinations. It also significantly reduces the computational requirements because at inference, only the relevant experts (LoRA adapters) are retrieved from an index.



Read more about Memory Tuning in our paper "Banishing LLM Hallucinations Requires Rethinking Generalization".

Memory Tuning step-by-step

The key to Memory Tuning is starting small and iterating until you achieve your desired level of accuracy on a representative dataset. The steps are:

1. Choose the use case
2. Choose your base model
3. Prepare your training dataset
4. Tune model
5. Evaluate performance
6. Repeat steps 3-5 until you achieve desired results

Common use cases and their applications

The first step is to identify your use case. Your use case should have clear and meaningful business impact. Here are our top customer use cases.

Use case	Application and KPIs
Factual Reasoning: Tune an LLM to respond to users' queries with factual information. **Method**: Memory RAG for level 1-2 questions + Memory Tuning for level 3+ questions	• **Applications**: AI chatbots and assistants • **Suggested KPIs**: # of successful queries, resolution rate without a human in the loop, resolution time, support costs, user satisfaction with output
Text-to-SQL: Tune an LLM to turn natural language queries into semantically and syntactically correct SQL to query a database and return an accurate response. **Method**: Memory Tuning	• **Applications**: Business Analyst or Business Intelligence agent • **Suggested KPIs**: # of successful queries, reduction in ad hoc requests to business analyst, resolution rate without a human in the loop, resolution time, user satisfaction with output
Function calling: Tune an LLM to use external tools such as APIs and functions. This enables it to retrieve external data to perform specific tasks such as answering a question about the weather or automating a step in a process. **Method**: Memory Tuning	• **Applications**: AI chatbots, agentic workflows such as automated workflows for environmental triggers (temperature sensors, etc.), automate assignment of support tickets, retrieve information from knowledgebase • **Suggested KPIs**: retrieval rate success, latency and throughput
Document Classification and Summarization: Tune an LLM to classify and summarize large amounts of unstructured data. **Method**: Memory RAG for simple use cases; Memory Tuning for complex tasks	• **Applications**: Sentiment, topic, or intent analysis, legal document analysis tool, fraud detection • **Suggested KPIs**: # of documents classified, hours of manual work saved
Code Generation: Tune an LLM to generate high quality code, particularly for niche programming languages and custom codebases that are not supported by existing LLMs. **Method**: Memory Tuning	• **Applications**: Code assistant / Copilot • **Suggested KPIs**: developer productivity, accuracy of code generated, # of bugs caught

Which base LLM should I use for my use case?

Different LLMs excel at different tasks so it's important to choose the right LLM for your use case from the start. In general, an LLM with more parameters (Llama 3.1 405B) is better at performing complex operations, however, they are also more computationally expensive to run. Fine-tuning a smaller open source model (e.g., Llama 3.1 8B) can outperform non-fine tuned larger models.

Below are some examples of open source models our customers have successfully trained for specific use cases.

Use Case or Task	Model
Text-to-SQL	Llama 3.1 8B
Code Generation	CodeLlama supports many of the most popular languages, including Python, C++, Java, PHP, Typescript (Javascript), C#, and Bash. It comes in Base, Instruct, Python in various sizes: 7B, 13B, 34B and 70B
	Mixtral 8X7B (Mixture of Experts)
	Mistral 7B
	Mistral Codestral Mamba
General Purpose Text Generation	Llama 3.1 8B Llama 3.2 3B Mistral 7B Gemma 2
Function Calling	Llama 3.1 includes native function calling (Decoder-only) Note if you want to full conversation capability with function calling, Llama 3.1 70B and Llama 3.1 405B are the two recommended options. Llama 3.1 8B is good for zero shot tool calling, but can't hold a full conversation at the same time.
	Mistral 8*22B

Prepare your evaluation dataset

The most important step is creating an initial evaluation dataset. Think quality over quantity because as the saying goes, garbage in equals garbage out. If you're not the domain expert, then we recommend that you work with a domain expert to construct your dataset so you know your questions and answers are correct and representative of the target user's queries.

Here are some frequently asked questions about constructing an evaluation dataset:

1. **How much data do you need to start?**
    a. We always recommend starting small with 20 to 100 samples. Work in smaller, faster iteration cycles so you know exactly where the model fails. If you start with too large of a dataset, it will be hard to debug.
    b. You also want to have enough breadth to cover the majority of queries for your application. It's a good practice to identify levels of questions. For example, level 1 for the easiest queries, level 2 for slightly more complex queries, level 3 for complex queries, and level 4 for edge cases. These will be your evaluation milestones.

2. **How should the data be structured?**
    a. The data should be structured as input/output pairs and they should be representative of the types of examples you would expect for your application. Here's a sample question and answer pair for an earnings call transcript.

```
{
   "question": "What is the percentage growth rate of
WPP's business in Germany in Q1, according to Mark Read?",
  "answer": "16%"
}
```

3. **Do we need to label our data first?**
    a. No, you do not need to label your data first unless it is for a classification use case.

4. **Can we use Lamini to generate input/output pairs?**
    a. Yes! You can use your Memory RAG pipeline to generate input/output pairs.

5. **Can we use synthetic data?**
    a. Yes, you can use an LLM to generate synthetic data. However, you should always manually check the quality of the data to ensure it is representative of the types of queries your users will use.

6. **Do you have any examples of training datasets?**
    a. Yes, see the NBA Evaluation dataset for the notebook we used in our DeepLearning.AI course with Meta, Improving Accuracy of LLM Applications

Create a baseline

Before you start Memory Tuning, you should measure the baseline accuracy on:

a. The base model
b. Base model + prompt tuning
c. Base model + prompt tuning + RAG

Run your first tuning job

Now that you've built your first evaluation dataset, you can run a tuning job. We've made this super easy so all you have to do is load your data, set some hyperparameters, and then run your tuning job. There are a couple of key hyperparameters you need to be aware of (see the full list here):

- learning rate - The initial learning rate for fine-tuning. The default is 9.0e-4. For training jobs with less than 300 steps, a grid search approach can be effective. You can run multiple jobs on a subset of the data with a range of learning_rates to find which learning rate has a better loss curve. Once that is found, you can expand the training to the larger dataset with this best learning_rate.
- Max_finetuning_examples - Sets the maximum number of data points for fine-tuning. If not set, the model is fine-tuned on the entire dataset. The default is the size of the dataset.
- Max_steps - Specifies the total number of training steps to perform. Defaults to 100. We recommend increasing max_steps for larger datasets with > 100 facts.
- Gradient_accumulation_steps - Number of update steps to accumulate the gradients for, before performing a backward/update pass. Defaults to 2. A higher setting can improve memory efficiency and thus reduce training time, often with a neutral effect on model accuracy.

Now with one line of code, you're ready to run your first tuning job!

```
Lamini(model_name="meta-llama/Meta-Llama-3.1-8B").tune(data_or_dataset
_id=data)
```

```python
import lamini
import random
import jsonlines


def main():
    llm = lamini.Lamini(model_name="meta-llama/Meta-Llama-3.1-8B-Instruct")

    data = get_data()

    print_data(data)

    llm.tune(
        data_or_dataset_id=data,
        finetune_args={
            "max_steps": 400,
            "learning_rate": 1.0e-4,
            "batch_size": 1,
            "index_ivf_nlist": 1,
            "index_method": "IndexFlatL2",
            "index_max_size": 100,
        },
        gpu_config={"gpus": 1, "nodes": 1},
    )
```

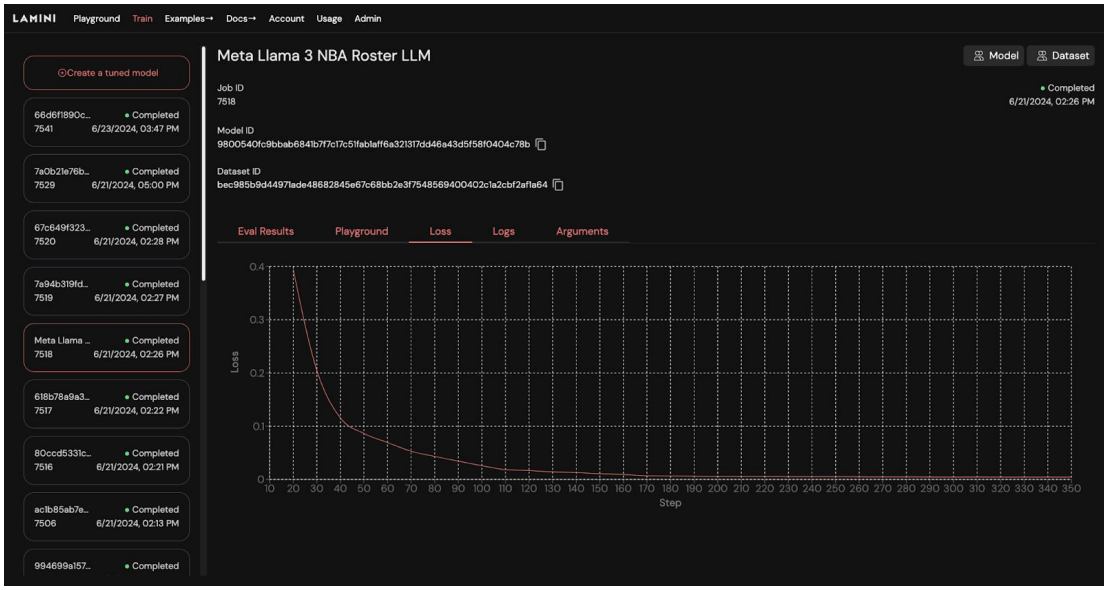
What is an epoch?

An epoch is a complete pass over the entire dataset during training. The model processes every sample in the training set once and updates its weights based on the learning algorithm. Too few epochs may lead to underfitting (not enough learning). Too many epochs can lead to overfitting (learning too much from noise and specifics of the training data). The optimal number of epochs ensures the model is well-trained, learning the important patterns in the data without overfitting.

What is learning rate?

The learning rate controls how big each step is adjusting the model's weights in response to the error (or loss) during training. A small learning rate leads to slow convergence but more stable training. A large learning rate can speed up training but may cause instability or overshooting. The goal is to find a learning rate that allows the model to converge efficiently to a good solution without overshooting or taking forever to learn.

After you submit a job, you can monitor the job status at https://app.lamini.ai/train. There you can track jobs, view logs, and get the model ID once training is complete.



Evaluate accuracy

The goal of evaluating a model is to understand if your system is improving and where it is still hallucinating. A good evaluation is quantitative, shows where you can improve, scalable and automated (using LLMs, of course!).

Now you can test your newly tuned model by running your evaluation dataset. You can get the LLM to output a numerical score by providing the question, generated answer, and scoring method in the prompt. You can use two methods to return a reliably numerical score:

- Use structured output (int, float, List[int], etc.)
- Calculate traditional exact match outputs. Note that you can even train the LLM to return exact matches.

You should see a big leap in accuracy. Make sure to pat yourself on the back and record your evaluation score!

Evaluating the NBA dataset using SQL scoring

```
system_prompt = "Compare the following two dataframes."
system_prompt += "They are similar if they are almost identical, "
system_prompt += "or if they convey the same information about the nba_roster dataset"
system_prompt += "Respond with valid JSON {'explanation' : str, 'similar' : bool}"
user_prompt = (
    f"========== Dataframe 1 =========\n{str(obj.data.get('df','None')).lower()}\n\n"
)
user_prompt += (
    f"========== Dataframe 2 =========\n{str(obj.data['reference_df']).lower()}\n\n"
)
user_prompt += f"Can you tell me if these dataframes are similar?"
```

	NAME	SALARY
0	Stephen Curry	$51,915,615

	SALARY	NAME
0	$51,915,615	Stephen Curry

It may seem counter intuitive, but now you want to get that accuracy score back down to 10 to 20 percent accuracy with a harder evaluation dataset. Here are some evaluation methods:

- Start with your error file to see where you need to fine-tune the model further. Review each data point and delete or fix obviously incorrect data points. These are typically the easiest examples where the LLM still fails. Categorize the different errors and create 3-5 input/output pair variations per category to teach the model what it needs to learn.
- Add questions with more breadth (e.g., level 2 questions), increasing the size, coverage, and complexity of the dataset.
- Be sure to improve your scoring mechanism to catch mistakes.

With Lamini, you can build reusable automated pipelines to re-run in future iterations of model improvement, when you upgrade your base model and even when you develop similar adjacent model applications!

Iterate! Iterate! Iterate!

Now that you know the basic steps to creating an evaluation dataset, running a tuning job, and evaluating your model, all you have to do is repeat these steps until you reach your desired level of accuracy. Here are some common questions:

1. **How many iterations will it take?**
    a. It depends on the complexity of your application, but usually it's around 10-30 iterations over different data pipelines to get to 90-95% accuracy. You can fine-tune multiple variants in parallel to speed up experimentation.

2. **How much time and compute does each iteration take?**
    a. For 95% accuracy on 1,000 facts, it takes about 1 minute and 24 petaflops on 1 NVidia A100 GPU with high-performance (40% MFU).

Get started with Memory Tuning

We hope this guide jump starts your journey into tuning your own model. If you'd like to learn more, check out the resources below or contact us for a demo. To try it out yourself, you can sign up at app.lamini.ai and get $300 free credit to run your first tuning job.

We partnered with Meta to create a notebook that shows how to use Memory Tuning to improve a text-to-SQL model from 30% to 95% accuracy. Check out our corresponding course on DeepLearning.AI, Improving Accuracy of LLM Applications.