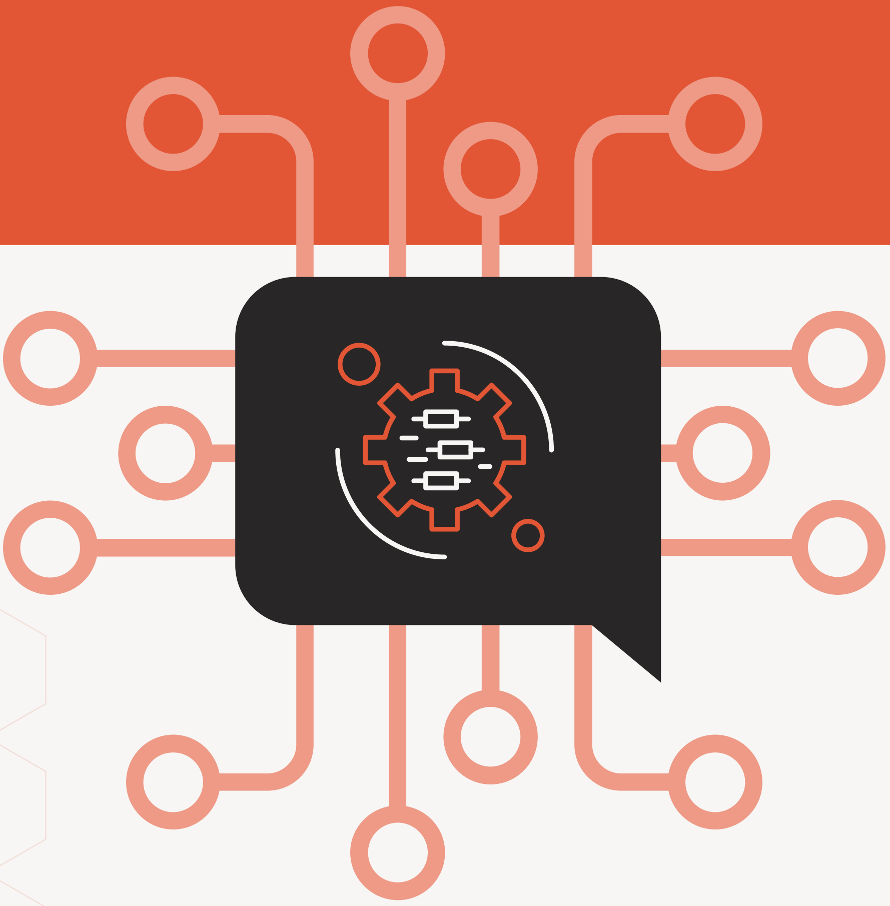
Predibase

EBOOK

The Definitive Guide to Fine-Tuning LLMs

Insights for tackling the 4 biggest challenges of fine-tuning



Predibase

Contents

Predibase

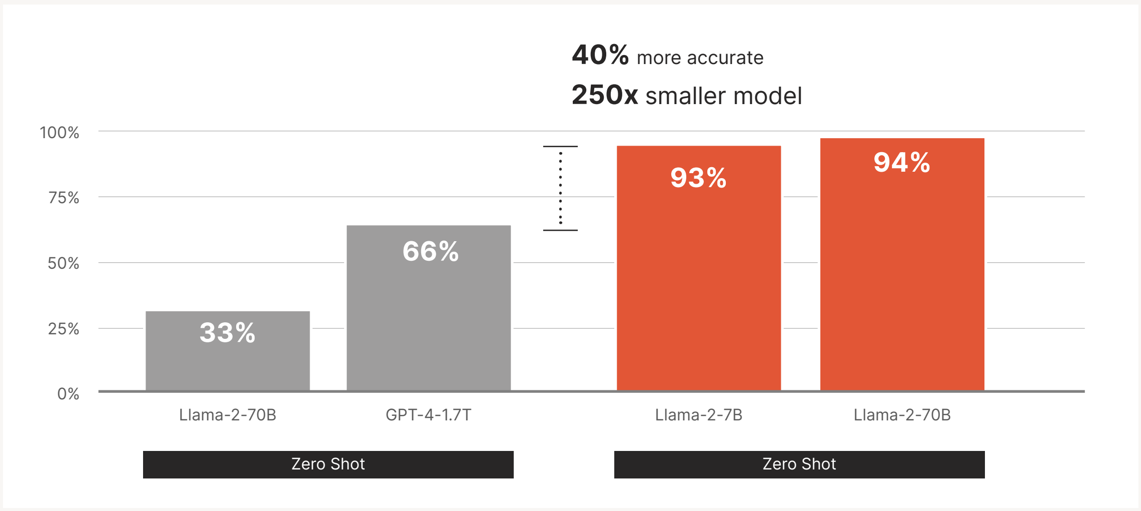
INTRODUCTION

The LLM Landscape and Rise of Fine-Tuning

Large language models (LLMs) are powerful deep neural networks that can understand and generate natural language. They have revolutionized the field of natural language processing (NLP) and opened up new possibilities for applications and services. Inflection points like the invention of transformers in 2017 or BERT in 2018 demonstrated the massive potential of LLMs, and the launch of ChatGPT at the end of 2022 took the world by storm with its conversational skills. LLMs have ushered a wave of innovations powered by Generative AI and removed significant barriers to entry in terms of data requirements and complexity of training models from scratch. There has never been a better time to be in the field of machine learning and AI. But while every organization is thinking about how to best leverage Generative AI for their needs, ML practitioners have to think about how to get there effectively.

So how do you balance the speed of innovation required, with the costs and resources needed to bring your model all the way from prototyping to production? The reality is that costs associated with building an LLM from the ground up are prohibitive for most organizations. Luckily, fine-tuning has emerged as a proven approach to improve the accuracy of a pre-trained model while significantly reducing the computational costs and time required to train an LLM from scratch. You can now start with a pre-trained LLM like Llama-2 and fine-tune to additional domain-specific knowledge. For example, you can build high quality, tailored applications like code assistants and customer support chatbots by fine-tuning them with your organization's vernacular or company data. As shown in the chart below, bigger isn't always better. Smaller, faster fine-tuned open-source models can outperform larger more costly commercial LLMs.

Model Accuracy for JSON Generation



Fine-tuned open-source models can outperform larger, more expensive commercial LLMs for specific tasks.

In this eBook, we will cover how to overcome the four biggest challenges when fine-tuning your models:

- When to fine-tune your models
- How to prepare your data
- What infrastructure is required for training and serving
- How to optimize fine-tuning

By following these best practices, you will be able to fine-tune your LLMs effectively and efficiently, and unleash their full potential for your organization.

CHAPTER 1

When to fine-tune?

Fine-tuning is the process of adapting a pre-trained LLM to a specific domain or task by updating its parameters with a smaller amount of data. Instead of starting from scratch, fine-tuning can improve the performance and efficiency of pre-trained LLMs for various applications. But how do you choose the best LLM and the best fine-tuning method for your project?

Step 1: Choose your task.

The first step is to identify the type of task you want to perform with LLMs. What are you trying to achieve with your NLP project? What's your use case? What are the inputs and outputs of your system? What are the evaluation metrics and desired outcomes?

In particular, one way to categorize NLP tasks is based on whether they are **predictive** or **generative**. Predictive tasks are those that require the system to predict a label, a score, or a category for a given input. Generative tasks are those that require the system to generate text for a given input.

In general, predictive tasks are easier to fine-tune than generative tasks, as they require less data and less computation. However, generative tasks can benefit more from fine-tuning, as they can produce more diverse and creative outputs that match the domain and task requirements.

	Predictive Tasks	Generative Tasks
Goal	Predict a label, score, or category for a given input	Generate text for a given input
Examples	Sentiment analysis, spam detection, text classification, information extraction, etc.	Text summarization, machine translation, text generation, etc.
Pros	Easier to fine-tune (requires less data and computation)	More creative and diverse
Cons	Constrained outputs	Harder to fine-tune (requires more data and computation)

Predictive vs Generative Tasks Checklist

Step 2: Choose the best model for your task.

The next step is to choose the best pre-trained model for your task. There are many high-performing open-source LLMs available for fine-tuning, such as the Llama-2, Mistral/Mixtral, Phi-2, Zephyr, and more. These models come in different sizes (e.g., Llama-2-7B vs. Llama-2-70B), each with its own strengths and weaknesses, depending on its size, architecture, training data, and performance for the type of task you would like to accomplish.

Below is a short description and table comparing different fine-tuned versions of the Llama-2 model to give you a sense of how performance can vary:

- **Code Llama** (available with 7B, 13B and 34B parameters) is fine-tuned from Llama-2 for generating and discussing code. It was further trained on code-specific datasets and can be used for code generation, generating natural language about code, and debugging in many popular programming languages.

- **Llama-2-Chat** (available with 7B, 13B and 70B parameters) is a version of Llama-2 that has been fine-tuned using supervised learning for prediction refinement to excel in conversational applications. It can use both words before and after each word to get the meaning, and it can use some inputs and goals to make text that fits them. Llama-2 can help with both predictive and generative tasks.

	Llama-2-7b-Chat	Llama-2-70b-Chat	CodeLlama-34b
Task Type	General	General	Code generation
Sample Efficiency	Low	High	Medium
Computational Efficiency	High	Low	Medium
Flexibility	High	High	Low
Training Data	Multilingual	Multilingual	Code

Comparing Llama-2-7b-Chat vs Llama-2-70b-Chat vs CodeLlama-34b

As a rule of thumb, to choose the best model for your task, you should consider two factors: compute efficiency for training and serving, and data relevance, or how similar the data used to train the model is to your specific domain or task, e.g. Llama-2-Chat for chat-specific applications and Code Llama for code generation. For example, larger models may perform better for a broader set of tasks but come at a higher compute cost. Fine-tuning a smaller model can often provide performance similar to a larger model but at a much lower cost.

Step 3: Choose the best approach to adapt your LLM.

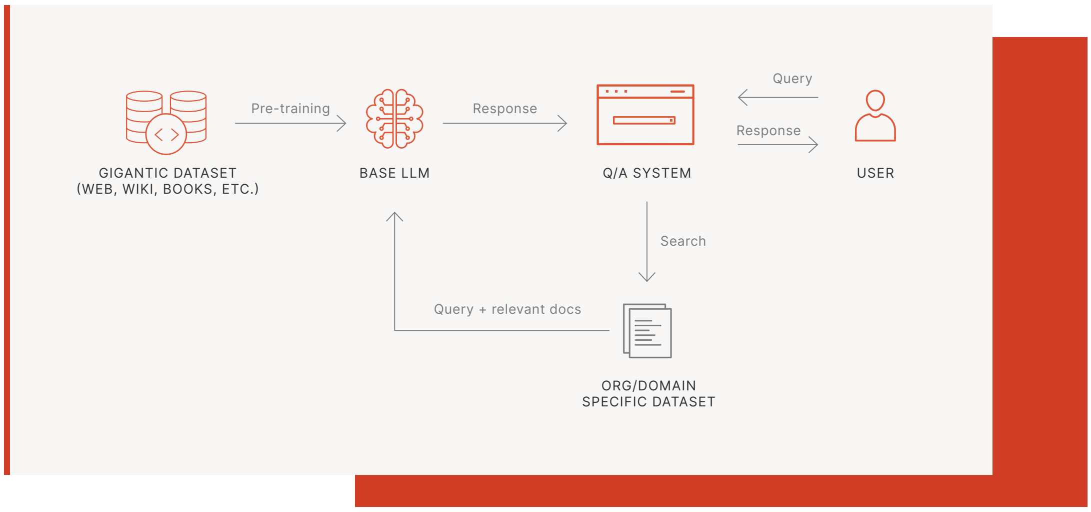
The final step is to choose the best method for adapting the pre-trained model to your task and dataset. There are three main methods for doing so: prompting, Retrieval Augmented Generation (RAG), and fine-tuning:

Prompting can be a good way to prototype and steer a model towards a desired behavior, by using words to tell an LLM what to do without changing its parameters. Prompting uses the internal knowledge that is already learned by the LLM from pre-training. While performance isn't guaranteed and it may not work for all use-cases, prompting is useful if you have minimal labeled data. However, keep in mind that getting to very strong results can take a lot of time and therefore can be expensive.

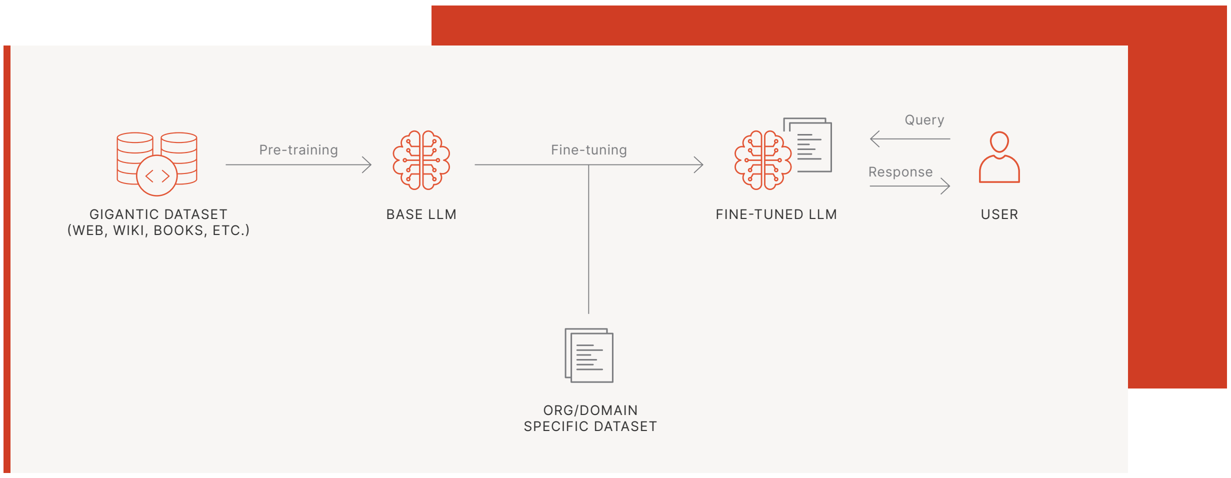
- **Prompting** can be a good way to prototype and steer a model towards a desired behavior, by using words to tell an LLM what to do without changing its parameters. Prompting uses the internal knowledge that is already learned by the LLM from pre-training. While performance isn't guaranteed and it may not work for all use-cases, prompting is useful if you have minimal labeled data. However, keep in mind that getting to very strong results can take a lot of time and therefore can be expensive.

```
prompt:
 task: "Classify the sample input as either negative, neutral, or positive."
```

- **RAG** combines an LLM with a retrieval system that can get information from an external vector store containing your data. RAG makes the LLM generate better and more specific responses by getting additional relevant information and using it as context while it generates a response–this method is useful when your task requires external knowledge or context to generate a correct output. It may however introduce irrelevant or inaccurate information if the retriever module is not well-designed.



- **Fine-tuning** adapts a pre-trained LLM to a specific domain or task by updating its parameters with a smaller amount of data, aiming to enhance performance and efficiency–this method is useful to achieve the best performance when you have a small and properly formatted task-specific dataset that is similar to or overlaps with the training data of the pre-trained model. However, it may cause overfitting or catastrophic forgetting if not done carefully.


GIGANTIC DATASET (WEB, WIKI, BOOKS, ETC.)
Pre-training
BASE LLM
Fine-tuning
FINE-TUNED LLM
Query
Response
USER
ORG/DOMAIN SPECIFIC DATASET

The choice of method depends on several factors, such as the availability and quality of your data, the similarity and complexity of your task, the trade-off between accuracy and efficiency, etc. You should compare different methods based on their advantages and disadvantages and choose the one that best fits your situation. Also note that these methods are not mutually exclusive. Predibase offers guidance on fine-tuning LLMs like LLaMa-2 with scalable LLM infrastructure, which can complement Prompting and RAG techniques.

	Prompting	**RAG**	**Fine-tuning**
Data Requirement	Low	High	Medium
Computation Requirement	Low	Medium	High
Parameter Update	N/A	N/A	Yes
Task Adaptation	Medium	High	High
Knowledge Incorporation	Low	High	High

Comparing methods to adapt a LLM to a chosen task(s)

In the next chapter, we will discuss how to prepare your data for fine-tuning.

Did you know?

Different types of fine-tuning can be used to adapt an LLM to a specific domain or task: instruction tuning, domain adaptation, and task adaptation.

Instruction tuning is used when you need very precise control over your LLM. For example, if you need your LLM to write in a very specific way, and generate specific types of content or responses. It uses explicit and structured instructions within the fine-tuning dataset to guide the LLM's behavior to perform a specific task. With this method, you can exploit the natural language understanding and generation capabilities of the LLM without changing its architecture or training objective.

Domain adaptation is used when you need the model to develop expertise in a specific field, like learning domain-specific vocabulary, style, and knowledge, so it can for example understand for example legal documents or medical records. It adapts the LLM towards a specific domain by updating its parameters with a smaller amount of domain-specific data.

Task adaptation is used to adapt an existing well-performing LLM to a specific task by adding a task-specific layer on top of it and updating all its parameters with a smaller amount of task-specific data. Task adaptation aims to learn task-specific objectives and constraints that can improve the performance and efficiency of the LLM.

CHAPTER 2

Data Preparation

One of the most important steps in fine-tuning LLMs is data preparation. The quality and quantity of your data can have a significant impact on the performance of your fine-tuned model. So how much data do you need to fine-tune your model effectively? What structure should your data be in? And finally, how to go about it when you don't have enough training data?

How much data do you need to fine-tune?

The answer to this question depends on several factors, such as the size and complexity of the LLM, the task and domain you are fine-tuning for, and the level of customization you want to achieve. In general, the more data you have, the better your fine-tuned model will be. However, there are diminishing returns after a certain point, and having too much data can also introduce noise and inefficiency.

As a rule of thumb, you should aim to have at **2,000 to 5,000 examples** for each task or domain you want to fine-tune for. This is based on real-life use cases that show that LLMs can achieve good results with smaller amounts of curated data. This is also supported by Meta's LIMA paper, where they demonstrate that 1,000 examples may be enough when fine-tuning with larger models. However, this is not a hard limit, and you may need more or less data depending on your specific use case.

Some factors that can affect the amount of data you need are:

- **Size and complexity of the LLM:** Larger and more complex LLMs tend to have more parameters, while often requiring less fine-tuning for generating coherent text across various domains — like Llama 2 (70 billion parameters) compared to smaller LLMs like BERT (110 million parameters).
- **Task and domain:** The complexity and diversity of the task and domain being fine-tuned can impact the amount of data needed. Simpler or broader tasks may require less data — such as sentiment analysis compared to natural language inference.
- **Level of customization:** Customization level can impact data requirements. A broad range of more general tasks and domains need less data compared to a specialized task or domain excellence, like generating generic text summaries versus personalized product reviews.

What structure should the data be in?

Generally speaking, your data should be in the form of (**input, output**) pairs for fine-tuning jobs. Depending on the type of task, these pairs will be structured differently:

- For predictive tasks like classification, regression, question answering, etc. your **input** is the text or context that the LLM has to process, and the **output** is the label or answer that the LLM has to predict based on the provided input.
- For tasks such as generation, summarization, translation, etc. your **input** is the text or context that the LLM has to use as a source or reference, and the **output** is the text that the LLM has to generate as a target response.

For example, if you want to fine-tune an LLM for sentiment analysis (a predictive task), your data should look something like this:

Input	Output
I love this movie!	Positive
This book is boring.	Negative
The food was okay.	Neutral

If you want to fine-tune an LLM for text summarization (a generative task), your data should look something like this:

Input	Output
The COVID-19 pandemic has caused unprecedented challenges and disruptions for the global economy and society. The World Health Organization (WHO) has declared it a public health emergency of international concern, and governments around the world have implemented various measures to contain the spread of the virus and mitigate its impacts. However, the pandemic is not only a health crisis, but also a social, economic, and environmental one. It has exposed and exacerbated the existing inequalities and vulnerabilities of different groups and regions, and posed serious threats to human rights, democracy, and sustainable development.	The COVID-19 pandemic is a multifaceted crisis that affects all aspects of life and requires coordinated and comprehensive responses from all stakeholders.

With an input/output structure like this, you could fine-tune the model using a prompt such as:

Prompt
Summarize the following article in one sentence: {Input}

How to fine-tune when you don't have labeled data?

Labeled data is data that has been annotated or tagged with the desired output or information, such as sentiment labels, summaries, translations, etc. Labeled data is essential for supervised learning, which is the most common method of fine-tuning LLMs.

However, labeled data can be scarce, expensive, or unavailable for some tasks or domains. In such cases, you may have to augment your labeled dataset, or even generate a dataset from scratch using other creative methods of data generation, such as:

- **Synthetic data generation:** You can use a larger LLM to generate synthetic data for your task or domain. For example, you can leverage Llama-2-70B or GPT-4 to make fake movie reviews with different sentiments, and use them as labeled data. However, leveraging synthetic data may come with some trade-offs including quality, diversity, biases or errors.

  > Check out our **Model Distillation playbook** to learn best practices for generating synthetic data from larger LLMs.

- **Data flywheel in product:** One way to fine-tune a LLM is to create a data flywheel that collects and labels real data from your users or customers. For example, you can let your users ask questions and rate the answers from your LLM, and use this feedback as labeled data. However, user-generated data may have some challenges, such as noise, spam, privacy, and user engagement.

In the next chapter, we will discuss infrastructure requirements for fine-tuning and serving your LLMs.

CHAPTER 3

Infrastructure Requirements

Fine-tuning a large language model (LLM) requires a lot of computational resources, GPU memory, and storage. Depending on the size of the model, the dataset, and your objective, you may need different types of infrastructure to train and serve your fine-tuned model.

How much compute do you need for fine-tuning LLMs?

Fine-tuning an LLM on your custom dataset involves updating the weights of the pre-trained model using a gradient-based optimization algorithm. This training process can be very time-consuming and resource-intensive, depending on the size of the model and the dataset.

To train an LLM effectively, you need enough CPU, RAM, and GPU memory to handle the large amount of data and parameters involved in fine-tuning. For example, Llama-2-7B has 7 billion parameters and requires 66 GB of GPU memory to load. Additionally, you may need a distributed system to parallelize the training process and speed it up.

How much compute do you need for serving LLMs?

The second step in fine-tuning an LLM is to serve it for inference in production. This involves loading the fine-tuned model into GPU memory and processing user requests or queries using the model. The serving process can also be very demanding and challenging, depending on the scale and complexity of your application.

To serve an LLM efficiently, you will also need enough CPU, RAM, and GPU memory to handle the large size of the model and the high volume of requests or queries. For example, Llama-2-70B requires 140 GB of GPU memory per inference. You also need a fast network connection to transfer data between your server and your client or user interface.

How to pick the best GPUs for your needs?

One of the most critical factors in fine-tuning and serving LLMs is the availability and selection of GPUs. GPUs can perform parallel computations faster than CPUs. They are essential for accelerating the training and inference of LLMs.

However, GPUs are also expensive and scarce resources. They can cost thousands of dollars per device or hundreds of dollars per hour on cloud platforms. They can also be in high demand by other users or applications that require GPU power.

Therefore, you need to optimize the use of GPUs for fine-tuning and serving LLMs. You need to consider the following aspects when choosing and using GPUs:

- **GPU type:** Different types of GPUs have different features and performance for different tasks. For example, the NVIDIA A100 is a high-end expensive GPU with 80GB of GPU memory, specifically designed for deep learning. Alternatively, Nvidia Tesla T4 is a more affordable GPU with 16G of GPU memory. It is suitable for mainstream applications and serving LLMs with low latency and high throughput.
- **GPU number:** The number of GPUs you need depends on the size of the model, the size of the dataset, the speed of the training or inference, and the budget of the project. For example, for fine-tuning Llama-2 on a large dataset in a short time, you may need tens or hundreds of GPUs to parallelize the training process. On the other hand, for serving Llama-2 for a large-scale application with high traffic, you may need thousands of GPUs to handle the requests or queries.
- **GPU utilization:** The utilization of a GPU is the percentage of time that the GPU is busy performing computations. A high utilization means that the GPU is fully utilized and not wasted. A low utilization means that the GPU is underutilized and idle. To increase the utilization of your GPUs, you can use techniques such as quantization, or adapters.

CHAPTER 4

Fine-tuning Optimizations

Fine-tuning LLMs is not easy and it can be expensive. It requires a lot of computational resources, time, and expertise. To optimize the fine-tuning process, you can also leverage sizing and optimization best practices.

How to perform efficient fine-tuning?

The **model size** affects the fine-tuning speed and cost. Bigger models are slower and more expensive, but may perform better. Smaller models are faster and cheaper, but may perform slightly worse. You need to balance size and quality when choosing a model. Start with a small model that fits your data and task, and scale up if needed.

Quantization and adapters are two techniques that can help reduce the memory and computational cost of fine-tuning large language models (LLMs) while improving their performance:

- **Quantization** is the process of reducing the number of bits used to represent each parameter in a model. For example, use 8-bit or 4-bit numbers instead of 32-bit floating point numbers. This can save memory and make LLMs faster, especially on devices with low resources. But quantization can also lose some accuracy or precision, so you need to find a balance between bits and quality.

- **Adapters** are small modules that are added to a pre-trained large language model (LLM) and fine-tuned for a specific task or domain. They allow the LLM to adapt to new scenarios without changing its original parameters. This can preserve the general knowledge of the LLM and avoid catastrophic forgetting, which is the problem of losing previous capabilities when fine-tuning for a new task. Adapters can also reduce the number of parameters that need to be updated during fine-tuning to a small fraction, which can save time and compute required (with less than 0.5% additional trainable parameters, the training overhead can be reduced up to 70%, compared to full fine-tuning). Some common adapter techniques include Low Rank Adaptations (LoRA) and Adaptive Low Rank Adaptations (AdaLoRA).

QLoRA (Quantized Low Rank Adapters) is an efficient fine-tuning approach for LLMs that combines quantization and adapters. As a result, it can significantly reduce memory usage while maintaining the performance of full 16-bit fine-tuning. It achieves this by back propagating gradients through a frozen, 4-bit quantized pre-trained LLM into the LoRA adapter weights, which are the only weights that are updated during training. This results in tuning the LoRA adapter weights for your task while leaving the base model weights untouched.

What are the most common fine-tuning parameters?

Epoch and learning rates are the most common fine-tuning parameters for deep neural networks, and it's important to find how to best adjust these parameters to improve your model accuracy while avoiding overfitting:

- **Epoch:** An epoch is a complete pass over the entire dataset during training. The number of epochs affects the model's accuracy and convergence. Too few epochs may result in underfitting, where the model does not learn enough from the data. Too many epochs may result in overfitting, where the model memorizes the data and loses generalization. A good practice is to use early stopping, where the training is stopped when the validation loss stops decreasing or starts increasing. Typically, large language models do well with 3 to 5 epochs of fine-tuning on larger datasets, and up to 10 epochs of fine-tuning on small datasets.

- **Learning rate:** The learning rate is a hyperparameter that controls how much the model's weights are updated by during each iteration of training. The learning rate affects the speed and stability of fine-tuning. A learning rate too high may cause the model to diverge or oscillate around the optimal solution. A low learning rate may cause the model to converge too slowly or get stuck in a local minimum. A good practice is to use a learning rate scheduler, where the learning rate is adjusted dynamically based on the progress of training, often combining some warm up steps to help stabilize gradients during the early part of training and then using some learning rate decay to slowly decrease the learning rate over time to prevent the model from overshooting the global minima.

How to evaluate your fine-tuned model?

Effective evaluation of AI models requires clearly defined criteria that align with your specific application's needs. The choice of evaluation metrics should reflect the nature of the problem and the desired outcomes of the model.

Automatic metric-based evaluation:

- **Target text metrics.** These include algorithm-based metrics such as word error rate, bleu score, token accuracy, perplexity, or neural-based such as BLEURT.

- **Custom metrics.** Domain-specific tasks that may require creating your own metric. For instance, evaluating models that generate JSON might focus on checking for schema adherence, extraction tasks might focus on slot accuracy or recall, and code generation models might focus on code compilation.

- **Published test suites.** Test suites are collections of test cases that are designed to evaluate a model's performance on specific aspects or challenges of the task or domain that may consist of a mix of target text metrics and custom metrics. For example, MMLU is a multitask metric that covers elementary math, US history, computer science, law and more. TruthfulQA measures a model's propensity to reproduce falsehoods commonly found online. Test suites can provide objective and consistent feedback on the model's strengths and weaknesses. For more domain-specific tasks, it may require creating your own metric or criteria for these tests, and these should be structured based on the business metrics that are relevant.

For a more extensive list of popular automatic metrics, refer to this resource from Hugging Face.

Manual evaluation:

- **Human ratings.** The most reliable way to measure the quality and relevance of the model's outputs. This involves asking human experts or users to score outputs based on some criteria (e.g., fluency, coherence, accuracy, etc.) This is likely the most costly and time-consuming option. Human ratings may also be subjective, and require specific training to attain rating consistency.
- **Elo rankings.** Human ratings in a pairwise contest setting. Elo rankings compare different models based on their relative performance in pairwise contests. For example, given two models that generate captions for an image, an evaluator can choose which one is better or if they are equal. The Elo score of each model is then updated based on the outcome of the contest. Elo rankings can provide a global and comparative measure of the model's quality and ranking across a set of fine-tuned model candidates. One can also use an additional model like GPT-3.5 as a discriminator to choose whether it prefers the output of model A or model B, making this a bit more of an automated process at the cost of it being less aligned to human preference.
- **LLMs as judges.** There's a growing trend of using LLMs themselves to assess model outputs qualitatively, especially in scenarios where traditional metrics might fall short or where manual evaluation by human raters is too expensive. This approach can be compelling but requires careful consideration to account for potential LLM biases.

Another framework for evaluating models is Error Analysis which involves the following steps:

- **Understand base model behavior before fine-tuning:** Before fine-tuning a model, try to understand how the pre-trained model behaves on the target task or domain. This can help identify the gaps and opportunities for improvement that fine-tuning can address.
- **Categorize errors so you can iterate on data to fix these problems in data space:** After fine-tuning a model, analyze and categorize the errors that the model makes on the test or validation data. This can help identify the root causes and patterns of the errors and suggest possible solutions or interventions. For example, some common error categories are misspelling, too verbose, repetitive outputs, etc.
- **Go after the most catastrophic errors first:** These are the errors that can harm the credibility, trustworthiness, or usability of the model and its outputs (e.g., misleading statements, offensive or inappropriate language, sensitive or personal information leakage, etc.).

As one looks to address errors and other LLM quality issues, consider relying on augmenting the LLM with rules-based systems to, for example, filter out bad training data, fix outputs in post-processing, or add guardrails to ensure output conformity and compliance.

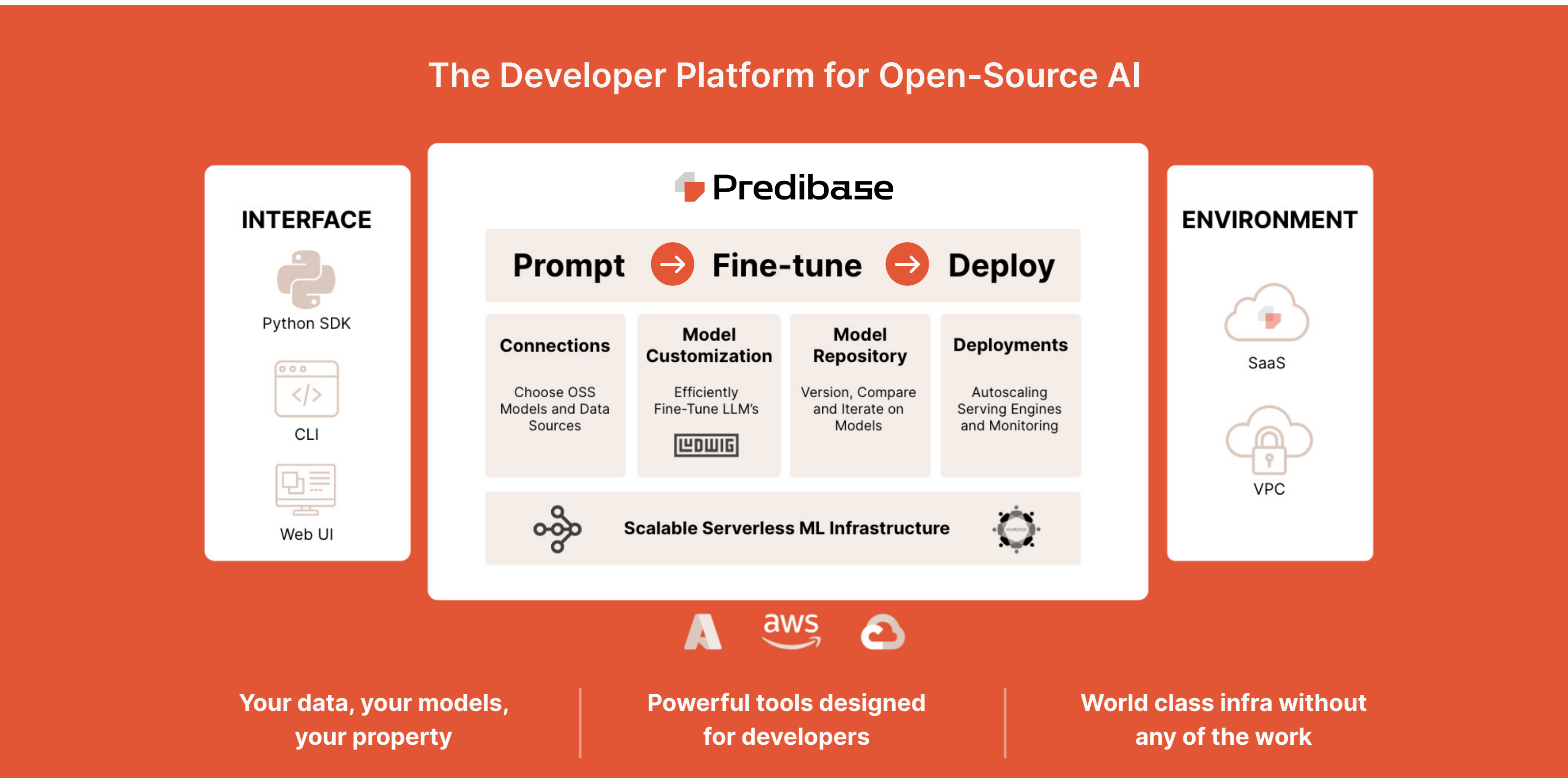
CONCLUSION

How to Get Started Fine-tuning and Serving Your Own LLM

In this eBook, we covered some of the biggest challenges of fine-tuning LLMs. However you are not alone. Predibase is designed to solve these challenges so that you can accelerate time to market and abstract away a lot of the complexity that comes with fine-tuning and serving your own custom LLMs in these ways:

- **Accelerating fine-tuning using Ludwig**, an open-source project that lets you define model training through configurations, instead of writing code. This greatly reduces fine-tuning complexity and makes it easy to try different fine-tuning strategies, prompts, and tasks.

> Make sure to check out this **on-demand session** with DeepLearning.AI that covers efficient fine-tuning with Ludwig.


The Developer Platform for Open-Source AI
INTERFACE
Python SDK
CLI
Web UI
Predibase
Prompt
Fine-tune
Deploy
Connections
Choose OSS Models and Data Sources
Model Customization
Efficiently Fine-Tune LLM's
LUDWIG
Model Repository
Version, Compare and Iterate on Models
Deployments
Autoscaling Serving Engines and Monitoring
Scalable Serverless ML Infrastructure
ENVIRONMENT
SaaS
VPC
aws
Your data, your models, your property
Powerful tools designed for developers
World class infra without any of the work

- **Managing prompt template iterations**, which can affect the model's performance. For example, when fine-tuning LLMs that were previously instruction-tuned like LLaMA-2-Chat, training is more effective when using the same prompt template as the LLM was trained with. Predibase separates the prompt template, data string, and task, so users can change each one easily without rewriting code or data.
- **Automatically right-sizing compute resources**, so the fine-tuning task is reliable and just works. This saves you from setting up their own infrastructure, dealing with distributed training, and solving memory issues.
- **Enabling distributed training out-of-the-box**, using Deepspeed configuration for model sharding, half-precision training, offloading parameters and optimizer states, and more. This makes the training process reliable, scalable, and cost-effective, without requiring you to configure these parameters.
- **Optimizing serving infrastructure for many fine-tuned models with LoRAX**, an open-source framework that lets you serve hundreds of fine-tuned LLMs from a single GPU with minimal degradation to throughput and capacity while unlocking 100x inference cost savings.

Get started fine-tuning and serving LLMs like Zephyr, Llama-2, Mistral, and more for free with our free trial.

TRY PREDIBASE ▶

KOBLE Koo PARADIGM payscale Papershift sekure TOPLINE EMAILS WWF