Fine-tuning LLMs

While prompt engineering and retrieval augmented generation (RAG) offer robust methods to guide a model's behavior, there are instances where they might not be adequate, especially for entirely novel or domain-specific tasks. In such cases, fine-tuning a LLM can be advantageous.

Fine-tuning is the process of adapting a pre-trained LLM on a comparatively smaller dataset that is specific to an individual domain or task. During the fine-tuning process, only a small number of weights are updated, allowing it to learn new behaviors and specialize in certain tasks.

The term "fine-tuning" can refer to several concepts, with the two most common forms being:

- **Supervised instruction fine-tuning:** This approach involves continuing training of a pre-trained LLM on a dataset of input-output training examples - typically conducted with thousands of training examples. Instruction fine-tuning is effective for question-answering applications, enabling the model to learn new specialized tasks such as information retrieval or text generation. The same approach is often used to tune a model for a single specific task (e.g. summarizing medical research articles), where the desired task is represented as an instruction in the training examples.
- **Continued pre-training:** This fine-tuning method does not rely on input and output examples but instead uses domain-specific unstructured text to continue the same pre-training process (e.g. next token prediction, masked language modeling). This approach is effective when the model needs to learn new vocabulary or a language it has not encountered before.

WHEN TO USE FINE-TUNING?

Choosing to fine-tune an open source LLM offers several advantages that tailor the model's behavior to better fit specific organizational needs. The following are a number of motivations behind choosing to fine-tune:

- **Customization and specialization:** LLMs trained on large, generic datasets - such as those provided by third-party APIs - often have broad knowledge but lack depth in niche areas. Fine-tuning allows organizations to specialize models for their specific domains or applications.
- **Full control over model behavior:** Fine-tuning provides granular control over the model's outputs. It allows organizations to address specific biases, enforce correctness and refine a model's behavior based on feedback.

FINE-TUNING IN PRACTICE

Fine-tuning, while advantageous, comes with practical considerations. Often, the optimal approach is not solely fine-tuning, but a blend of fine-tuning and retrieval methods like RAG. For example, you might fine-tune a model to generate specific outputs but also use RAG to inject data relevant to user queries.

Notably, fine-tuning large models with billions of parameters comes with its own set of challenges, particularly in terms of computational resources. To accommodate the training of such models, modern deep learning libraries like PyTorch FSDP and Deepspeed employ techniques like Zero Redundancy Optimizer (ZeRO), Tensor Parallelism, and Pipeline Parallelism. These methods optimize the distribution of the model training process across multiple GPUs, ensuring efficient use of resources.

A resource-efficient alternative to fine-tuning all parameters of a LLM is a class of methods referred to as parameter-efficient fine-tuning (PEFT). PEFT methods, such as LoRA and IA3 fine-tune LLMs by adjusting a limited subset of model parameters or a small number of extra-model parameters. These approaches not only conserve GPU memory, but often match the performance of full model fine-tuning. Open source libraries such as Hugging Face's PEFT library have been developed to easily implement this family of fine-tuning techniques for a subset of models.

Lastly, it's worth noting the significance of human feedback in a fine-tuning context, especially for applications like question-answering systems or chat interfaces. Such feedback helps refine the model, ensuring its outputs align more closely with user needs and expectations. To facilitate this iterative process, libraries like Hugging Face's trl or CarperAI's trlX can be used to apply methods like Proximal Policy Optimization (PPO) which incorporate human feedback into the fine-tuning process.

Pre-training

Pre-training a model from scratch refers to the process of training a language model on a large corpus of data (e.g. text, code) without using any prior knowledge or weights from an existing model. This is in contrast to fine-tuning, where an already pre-trained model is further adapted to a specific task or dataset. The output of full pre-training is a base model that can be directly used or further fine-tuned for downstream tasks.

WHEN TO USE PRE-TRAINING?

Choosing to pre-train an LLM from scratch is a significant commitment, both in terms of data and computational resources. Here are some scenarios where it makes sense:

1. Unique data sources: If you possess a unique and extensive corpus of data that is distinct from what available pre-trained LLMs have seen, it might be worth pre-training a model to capture this uniqueness.
2. Domain specificity: Organizations might want a base model tailored to their specific domain (e.g., medical, legal, code) to ensure even the foundational knowledge of the model is domain-specific.
3. Full control over training data: Pre-training from scratch offers transparency and control over the data the model is trained on. This may be essential for ensuring data security, privacy, and custom tailoring of the model's foundational knowledge.
4. Avoiding third-party biases: Pre-training ensures that your LLM application does not inherit biases or limitations from third-party pre-trained models.

PRE-TRAINING IN PRACTICE

Given the resource-intensive nature of pre-training, careful planning and sophisticated tooling are required. Libraries like PyTorch FSDP and Deepspeed, mentioned previously in the fine-tuning section, are similarly required for their distributed training capabilities when pre-training an LLM from scratch. The following only scratches the surface on some of the considerations one must take into account when pre-training an LLM:

- **Large scale data preprocessing:** A pre-trained model is only as good as the data it is trained on. Thus, it becomes vitally important to ensure robust data preprocessing is conducted prior to model training. Given the scale of the training data involved, this preprocessing typically requires distributed frameworks like Apache Spark. Consideration must be given to factors such as dataset mix and deduplication techniques to ensure the model is exposed to a wide variety of unique data points.
- **Hyperparameter selection and tuning:** Before executing full-scale training of an LLM, determining the set of optimal hyperparameters is crucial. Given the high computational cost associated with LLM training, extensive hyperparameter sweeps are not always feasible. Instead, informed decisions based on smaller-scale searches or prior research are employed. Once a promising set is identified, these hyperparameters are used for the full training run. Tooling like MLflow is essential to manage and track these experiments.
- **Maximizing resource utilization:** Given the high costs associated with long-running distributed GPU training jobs it is hugely important to maximize resource utilization. MosaicML's composer is an example of a library that uses PyTorch FSDP with additional optimizations to maximize Model FLOPs Utilization (MFU) and Hardware FLOPs Utilization (HFU) during training.
- **Handling GPU failures:** Training large models can run for days or even weeks. During such large scale training for this length of time, hardware failures, especially GPU failures, can (and typically do) occur. It is essential to have mechanisms in place to handle such failures gracefully.
- **Monitoring and evaluation:** Close monitoring of the training process is essential. Saving model checkpoints regularly and evaluating on validation sets not only act as safeguards but also provide insights into model performance and convergence trends.

In cases where pre-training an LLM from scratch is required, MosaicML Training provides a platform to conduct training of multi-billion parameter models in a highly optimized and automated manner. Automatically handling GPU failures and resuming training without human intervention, and MosaicML Streaming for efficient streaming of data into the training process are just some of the capabilities provided out-of-the-box.

Third-party APIs vs. self-hosted models

Choosing between third-party LLM APIs and self-hosting your own models has implications for cost, control, and data security. Here are a number of concerns one should consider when making that decision:

- **Data security and privacy:** Using third-party APIs often involves sending data to external servers. This can pose risks, especially when dealing with sensitive or proprietary information. In some cases regulations may simply prohibit data leaving data regions, or being sent to non-compliant environments. Hosting your own model ensures that data does not leave your secure environment, while retaining full access to the trained model.
- **Predictable and stable behavior:** Proprietary SaaS models can undergo updates or changes without prior notice. Such changes can lead to unpredictable model behavior. When hosting your own LLM, you have full control over its versions and updates.
- **Vendor lock-in:** Relying on third-party APIs means being dependent on the vendor's terms, pricing, and availability. With this, there is the risk of the service being deprecated or price changes. By self-hosting, you maintain autonomy and guard against potential vendor lock-in.

Model Evaluation

Evaluating LLMs is a challenging and evolving domain, primarily because LLMs often demonstrate uneven capabilities across different tasks. An LLM might excel in one benchmark, but slight variations in the prompt or problem can drastically affect its performance. The dynamic nature of LLMs and their vast potential applications only amplify the challenge of establishing comprehensive evaluation standards.

The following are a number of present challenges involved with evaluating LLM-powered applications:

- Variable performance: LLMs can be sensitive to prompt variations, demonstrating high proficiency in one task but faltering with slight deviations in prompts.
- Lack of ground truth: Since most LLMs output natural language, it is very difficult to evaluate the outputs via traditional NLP metrics (BLEU, ROUGE, etc.). For example, suppose an LLM were used to summarize a news article. Two equally good summaries might have almost completely different words and word orders, so even defining a "ground-truth" label becomes difficult or impossible.
- Domain-specific evaluation: For domain-specific fine-tuned LLMs, popular generic benchmarks may not capture their nuanced capabilities. Such models are tailored for specialized tasks, making traditional metrics less relevant. This divergence often necessitates the development of domain-specific benchmarks and evaluation criteria. See the example of Replit's code generation LLM.
- Reliance on human judgment: It is often the case that LLM performance is being evaluated in domains where text is scarce or there is a reliance on subject matter expert knowledge. In such scenarios, evaluating LLM output can be costly and time consuming.

Some prominent benchmarks used to evaluate LLM performance include:

- BIG-bench (Beyond the Imitation Game benchmark)
  A dynamic benchmarking framework, currently hosting over 200 tasks, with a focus on adapting to future LLM capabilities.
- EluetherAI LM Evaluation Harness
  A holistic framework that assesses models on over 200 tasks, merging evaluations like BIG-bench and MMLU, promoting reproducibility and comparability.
- Mosaic Model Gauntlet
  An aggregated evaluation approach, categorizing model competency into six broad domains (shown below) rather than distilling to a single monolithic metric.

Source: Mosaic Model Gauntlet

LLMS AS EVALUATORS

A number of alternative approaches have been proposed to use LLMs themselves to assist with model evaluation. These range from using LLMs to generate evaluations, through to entrusting LLMs as judges to evaluate other models capabilities or outputs. Ideally, when evaluating an LLM, a larger or more capable LLM should be employed as the evaluator. This premise is based on the understanding that a "smarter" model can plausibly produce a more accurate evaluation. The benefits of using LLMs as evaluators include:

- Speed, as they are faster than human evaluators
- Cost-effectiveness
- In certain cases, they offer comparable accuracy to human evaluators

See the following Databricks blog post for a more detailed exploration on using LLMs as evaluators.

HUMAN FEEDBACK IN EVALUATION

While human feedback is important in many traditional ML applications, it becomes much more important for LLMs. Humans - ideally your end users - become essential for validating LLM output. While you can pay human labelers to compare or rate model outputs (or have an LLM evaluate your application as mentioned above), the best practice for user-facing applications is to build human feedback into the applications from the outset. For example, a tech support chatbot may have a "click here to chat with a human" option, which provides implicit feedback indicating whether the chatbot's responses were helpful. Explicit feedback can also be captured in this case by presenting users with the ability to click thumbs up/down buttons.

Packaging models or pipelines for deployment

In traditional ML, there are generally two types of ML logic to package for deployment: models and pipelines. These artifacts are generally managed toward production via a model registry and Git version control, respectively.

With LLMs, it is common to package ML logic in new forms. These may include:

- A lightweight call to an LLM API service (third-party or internal)
- A "chain" from LangChain or an analogous pipeline from another tool. The chain may call an LLM API or a local LLM model.
- An LLM or an LLM+tokenizer pipeline, such as a Hugging Face pipeline. This pipeline may use a pretrained model or a custom fine-tuned model.
- An engineered prompt, possibly stored as a template in a tool such as LangChain.

Though LLMs add new terminology and tools for composing ML logic, all of the above still constitute models and pipelines. Thus, the same tooling such as MLflow can be used to package LLMs and LLM pipelines for deployment. Built-in model flavors include:

- PyTorch and TensorFlow
- Hugging Face Transformers (relatedly, see Hugging Face Transformers' MLflowCallback)
- LangChain
- OpenAI API
- (See the documentation for a complete list)

For other LLM pipelines, MLflow can package the pipelines via the MLflow pyfunc flavor, which can store arbitrary Python code.

Note about prompt versioning

Just as it is helpful to track model versions, it is helpful to track prompt versions (and LLM pipeline versions, more generally). Packaging prompts and pipelines as MLflow models simplifies versioning. Just as a newly retrained model can be tracked as a new model version in the MLflow model Registry, a newly updated prompt can be tracked as a new model version.

Note about deploying models vs. code

Your decisions around packaging ML logic as version-controlled code vs. registered models will help to inform your decision about choosing between the deploy models, deploy code, and hybrid architectures. Review the subsection below about human feedback, and make sure that you have a well-defined testing process for whatever artifacts you choose to deploy.

LLM Inference

Inference for LLM-powered solutions can come in a number of forms. In this section we focus on two primary methods for LLM inference: real-time and batch.

REAL-TIME INFERENCE

Real-time inference is ideal for applications requiring immediate responses, such as chat interfaces or question-answering systems. Real-time LLM inference can be further bucketed into the following:

- **Third-party LLM API with pre- and post- processing logic:**
  - Some inference pipelines, like RAG workflows, include pre-processing and post-processing logic, and use an external LLM API to generate output. Such pipelines can be deployed as REST API endpoints using services like Databricks Model Serving. For deployed LLM pipelines making requests to external APIs, CPU instance types are sufficient.
  - To standardize interactions with SaaS and OSS LLMs for such LLM pipelines, the MLflow AI Gateway can be used to manage requests to different LLM API providers. It offers a high-level interface that simplifies the interaction with these services by providing a unified endpoint to handle specific LLM related requests.
- **Pre-trained or fine-tuned OSS LLMs:**
  - When hosting your own LLMs, such as those downloaded from repositories like Hugging Face (and additionally fine-tuned), GPU instance types are typically required for optimized inference.

BATCH INFERENCE

Batch inference is well-suited for scenarios where immediate feedback is not critical (e.g. offline text summarization). For batch use cases, Spark can be leveraged to distribute inference across multiple machines (GPUs included).

INFERENCE WITH LARGE MODELS

Applicable to both real-time and batch inference scenarios is handling cases where large models exceed the memory of a single GPU. In such cases:

- Distribute serving across multiple GPUs, and/or;
- Consider loading the model with reduced precision, such as 8-bit or 4-bit quantization, to fit within memory constraints. Be aware that this option may affect the quality of the model's outputs.

Managing cost/performance trade-offs

One of the big Ops topics for LLMs is managing cost/performance trade-offs, especially for inference and serving. With "small" LLMs having hundreds of millions of parameters and large LLMs having hundreds of billions of parameters, computation can become a major expense. Thankfully, there are many ways to manage and reduce costs when needed. We will review some key tips for balancing productivity and costs.

1. **Start simple, but plan for scaling:** When developing a new LLM-powered application, speed of development is key, so it is acceptable to use more expensive options, such as paid APIs for existing models. As you go, make sure to collect data such as queries and responses. If the API provider terms of service permits it, you may be able to use this data to fine-tune a smaller, cheaper model which you can own.
2. **Scope out your costs:** How many queries per second do you expect? Will requests come in bursts? How much does each query cost? These estimates will inform you about project feasibility and will help you to decide when to consider bringing the model in-house with open source models and fine-tuning.
3. **Reduce costs by tweaking LLMs and queries:** There are many LLM-specific techniques for reducing computation and costs. These include shortening queries, tweaking inference configurations, and using smaller versions of models.
4. **Get human feedback:** It is easy to reduce costs but hard to say how changes impact your results unless you get human feedback from end users.

METHODS FOR REDUCING COSTS OF INFERENCE

USE A SMALLER MODEL

- Pick a different existing model. Try smaller versions of models (such as "llama-2-13b" instead of "llama-2-70b") or alternate architectures.
- Fine-tune a custom model. With the right training data, a fine-tuned small model can often perform as well or better than a large generic model.
- Use model distillation (or knowledge distillation). This technique "distills" the knowledge of the original model into a smaller model.
- Reduce floating point precision (quantization). Models can sometimes use lower precision arithmetic without losing much in quality.

REDUCE COMPUTATION FOR A GIVEN MODEL.

- Shorten queries and responses. Computation scales with input and output sizes, reducing costs by using more concise queries and responses.
- Tweak inference configurations. Some types of inference, such as beam search, require more computation.

OTHER

- Split traffic. If your return on investment (ROI) for an LLM query is low, then consider splitting traffic so that simpler, faster models or methods handle low ROI queries. Save LLM queries for high ROI traffic.
- Use pruning techniques. If you are training your own LLMs, there are pruning techniques that allow models to use sparse computation during inference. This reduces computation for most or all queries.