Meta

Llama 2 Get Started FAQ Download the Model

Quick setup and how-to guide

Getting started with Llama

Welcome to the getting started guide for Llama.

This guide provides information and resources to help you set up Llama including how to access the model, hosting, how-to and integration guides. Additionally, you will find supplemental materials to further assist you while building with Llama.

Meta

Meta

QUICK SETUP

Prerequisite

1. OS: Ubuntu
2. Packages: wget, md5sum
3. Package Manager: Conda ME

If you want to use Llama 2 on Windows, macOS, iOS, Android or in a Python notebook, please refer to the open source community on how they have achieved this. Here are some of the resources from open source that you can read more about: (Repo 1) (Repo 2) (Repo 3).

Getting the Models

1. Visit the Llama download form and accept our License.
2. Once your request is approved, you will receive a signed URL over email.
3. Clone the Llama 2 repository (here).

a. Keep in mind that the links expire after 24 hours and a certain amount of downloads.
If you start seeing errors such as 403: Forbidden, you can always re-request a link.

Hosting

Amazon Web Services

AWS provides multiple ways to host your Llama models. (SageMaker Jumpstart, EC2, Bedrock etc). In this document we are going to outline the steps to host your models using SageMaker Jumpstart and Bedrock. You can refer to other offerings directly on the AWS site.

Bedrock

A fully managed service that offers a choice of high performing foundation models, available via an API, to build generative AI applications, simplifying development while maintaining privacy and security. You can read more about the product here and follow instructions to use Llama 2 with Bedrock here.

EC2 Instance

To deploy models on EC2 instances, you must first request access to the model from our Llama download form, Hugging Face or Kaggle. Once you have this model you can either deploy it on a Deep Learning AMI image that has both Pytorch and Cuda installed or create your own EC2 instance with GPUs and install all the required dependencies. For detailed instructions on how to set up your Deep Learning AMI image you can refer here or to set up your own EC2 instance here.

> with fully managed infrastructure, tools and workflows. With SageMaker JumpStart, ML practitioners can choose from a broad selection of publicly available foundational models and deploy them on SageMaker Instances for model training and deployments. You can read more about it here.

Cloudflare

Workers AI

Is a serverless GPU-powered inference on Cloudflare’s global network. It’s an AI inference as a service platform, empowering developers to run AI models with just a few lines of code. Learn more about Workers AI here and look at the documentation here to get started to use Llama 2 models here.

Google Cloud Platform (GCP) - Model Garden

GCP is a suite of cloud computing services that provides computing resources as well as virtual machines. Building on top of GCP services, Model Garden on Vertex AI offers infrastructure to jumpstart your ML project with a single place to discover, customize, and deploy a wide range of models. With more than 100 foundation models available to developers, you can deploy AI models with a few clicks as well as running fine-tuning tasks in Notebook in Google Colab.

> ### Vertex AI
>
> We have collaborated with Vertex AI from Google Cloud to fully integrate Llama 2, offering pre-trained, chat and CodeLlama in various sizes. Getting started from here, note that you may need to request proper GPU computing quota as a prerequisite.

Hugging Face

You must first request a download using the same email address as your Hugging Face account. After doing so, you can request access to any of the models on Hugging Face and within 1-2 days your account will be granted access to all versions.

Kaggle

Kaggle is an online community of data scientists and machine learning engineers. It not only allows users to find datasets for building their AI models, but also allows users to search and discover hundreds of trained, ready-to-deploy machine learning models in one place. Moreover, community members can also publish their innovative works with these models as in Notebooks, backed by Google Cloud AI platform for computing resources and virtual machines.

We have collaborated with Kaggle to fully integrate Llama 2, offering pre-trained, chat and CodeLlama in various sizes. To download Llama 2 model artifacts from Kaggle, you must first request a download using the same email address as your Kaggle account. After doing so, you can request access to Llama 2 and Code Lama models. You get access to downloads once your request is processed.

Microsoft Azure & Windows

With Microsoft Azure you can access Llama 2 in one of two ways, either by downloading the Llama 2 model and deploying it on a virtual machine or using Azure Model Catalog.

Azure Virtual Machine

To run Llama with an Azure VM, you can set up your own VM or use Azure's Data Science VM which comes with Pytorch, CUDA, NVIDIA System Management and other ML tools already installed. To use the Data Science VM, follow the instructions here to set one up. Make sure to set this VM up with a GPU enabled image. However, if you would like to set up your own VM, you can follow the quickstart instructions on the Microsoft site here. You can stop at the "Connect to virtual machine" step. Once you have a VM set up, you can follow the instructions here to access the models locally on the VM.

Azure Model Catalog

Azure Model Catalog is a hub for exploring collections of foundation models. Built on top of Azure ML platform, Model Catalog provides options to run ML tasks such as fine-tuning and evaluation with just a few clicks. In general, it is a good starting point for beginner developers to try out their favorite models and also integrated with powerful tools for senior developers to build AI applications for production.

We have worked with Azure to fully integrate Llama 2 with Model Catalog, offering both pre-trained chat and CodeLlama models in various sizes. Please follow the instructions here to get started with.

ONNX for Windows

ONNX is an open format built to represent machine learning models. It defines a common set of operators and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes and compilers. One of the main advantages of using ONNX is that it allows models to be easily exported from one framework, such as TensorFlow, and imported into another framework, such as PyTorch.

Windows easily.

Get started developing applications for Windows/PC with the official ONNX Llama 2 repo here and ONNX runtime here. Note that, to use the ONNX Llama 2 repo you will need to submit a request to download model artifacts from sub-repos. This request will be reviewed by the Microsoft ONNX team.

HOW TO GUIDES

If you are looking to learn by writing code it's highly recommended to look into the Getting to Know Llama 2 - Jupyter notebook. It's a great place to start with most commonly performed operations on Llama 2.

Fine Tuning

Full parameter fine-tuning is a method that fine-tunes all the parameters of all the layers of the pre-trained model. In general, it can achieve the best performance but it is also the most resource-intensive and time consuming: it requires most GPU resources and takes the longest.

PEFT, or Parameter Efficient Fine Tuning, allows one to fine tune models with minimal resources and costs. There are two important PEFT methods: LoRA (Low Rank Adaptation) and QLoRA (Quantized LoRA), where pre-trained models are loaded to GPU as quantized 8-bit and 4-bit weights, respectively. It’s likely that you can fine-tune the Llama 2-13B model

Typically, one should try LoRA, or if resources are extremely limited, QLoRA, first, and after the fine-tuning is done, evaluate the performance. Only consider full fine-tuning when the performance is not desirable.

Experiment tracking

Experiment tracking is crucial when evaluating various fine-tuning methods like LoRA, and QLoRA. It ensures reproducibility, maintains a structured version history, allows for easy collaboration, and aids in identifying optimal training configurations. Especially with numerous iterations, hyperparameters, and model versions at play, tools like Weights & Biases (W&B)become indispensable. With its seamless integration into multiple frameworks, W&B provides a comprehensive dashboard to visualize metrics, compare runs, and manage model checkpoints. It's often as simple as adding a single argument to your training script to realize these benefits - we'll show an example in the Hugging Face PEFT LoRA section.

Recipes PEFT LoRA

The llama-recipes repo has details on different fine-tuning (FT) alternatives supported by the provided sample scripts. In particular, it highlights the use of PEFT as the preferred FT method, as it reduces the hardware requirements and prevents catastrophic forgetting. For specific cases, full parameter FT can still be valid, and different strategies can be used to still prevent modifying the model too much. Additionally, FT can be done in single gpu or multi-gpu with FSDP.

In order to run the recipes, follow the steps below:

1. Create a conda environment with pytorch and additional dependencies
2. Install the recipes as described here.

Meta ☰

```
python -m llama_recipes.finetuning --use_peft --peft_method
lora --quantization --model_name ../llama/models_hf/7B --output_dir ../llama/models_ft/7B-
--batch_size_training 2 --gradient_accumulation_steps 2
```

Hugging Face PEFT LoRA (link)

Using Low Rank Adaption (LoRA) , Llama 2 is loaded to the GPU memory as quantized 8-bit weights.

Using the Hugging Face Fine-tuning with PEFT LoRA is super easy - an example fine-tuning run on Llama 2-7b using the OpenAssistant data set can be done in three simple steps:

```
pip install trl
git clone https://github.com/huggingface/trl
python trl/examples/scripts/sft.py \
--model_name meta-llama/Llama-2-7b-hf \
--dataset_name timdettmers/openassistant-guanaco \
--load_in_4bit \
--use_peft \
--batch_size 4 \ --gradient_accumulation_steps 2 \
--log_with wandb
```

adapter_config.json and adapter_model.bin - run the script below to infer with the base model and the new model, generated by merging the base model with the fined-tuned one:

```
import torch
from transformers import ( AutoModelForCausalLM, AutoTokenizer,
pipeline,
)
from peft import LoraConfig, PeftModel
from trl import SFTTrainer

model_name = "meta-llama/Llama-2-7b-chat-hf"
new_model = "output"
device_map = {"": 0}


base_model = AutoModelForCausalLM.from_pretrained( model_name,
low_cpu_mem_usage=True,
return_dict=True,

torch_dtype=torch.float16,
device_map=device_map,
)

model = PeftModel.from_pretrained(base_model, new_model)
model = model.merge_and_unload()


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

prompt = "Who wrote the book Innovator's Dilemma?"
```

```
result = pipe(f"<s>[INST] {prompt} [/INST]")
print(result[0]['generated_text'])

pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max_length=200)
result = pipe(f"<s>[INST] {prompt} [/INST]")
print(result[0]['generated_text'])
```

QLoRA Fine Tuning

QLoRA (Q for quantized) is more memory efficient than LoRA. In QLoRA, the pretrained model is loaded to the GPU as quantized 4-bit weights. Fine-tuning using QLoRA is also very easy to run - an example of fine-tuning Llama 2-7b with the OpenAssistant can be done in four quick steps:

```
git clone https://github.com/artidoro/qlora
cd qlora
pip install -U -r requirements.txt ./scripts/finetune_llama2_guanaco_7b.sh
```

It takes about 6.5 hours to run on a single GPU, using 11GB memory of the GPU. After the fine-tuning completes and the output_dir specified in ./scripts/finetune_llama2_guanaco_7b.sh will have checkoutpoint-xxx subfolders, holding the fine-tuned adapter model files. To run inference, use the script below:

```
import torch
```

```
load_in_4bit=True,
bnb_4bit_compute_dtype=torch.bfloat16,
bnb_4bit_use_double_quant=True,
bnb_4bit_quant_type='nf4'
)
model = AutoModelForCausalLM.from_pretrained(
model_id,
low_cpu_mem_usage=True,
load_in_4bit=True,
quantization_config=quantization_config,
torch_dtype=torch.float16,
device_map='auto'
)
model = PeftModel.from_pretrained(model, new_model)
tokenizer = AutoTokenizer.from_pretrained(model_id)

prompt = "Who wrote the book innovator's dilemma?"
pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max_length=200)
result = pipe(f"<s>[INST] {prompt} [/INST]") print(result[0]['generated_text'])
```

Axolotl is another open source library you can use to streamline the fine-tuning of Llama 2. A good example of using Axolotl to fine-tune Llama 2 with four notebooks covering the whole fine-tuning process (generate the dataset, fine-tune the model using LoRA, evaluate and benchmark) is here.

Quantization

faster fine-tuning and faster inference. In resource-constrained environments such as single-GPU or Mac or mobile edge devices (e.g. https://github.com/ggerganov/llama.cpp), quantization is a must in order to fine-tune the model or run the inference. More information about quantization is available here & here.

Prompting

Prompt engineering is a technique used in natural language processing (NLP) to improve the performance of the language model by providing them with more context and information about the task in hand. It involves creating prompts, which are short pieces of text that provide additional information or guidance to the model, such as the topic or genre of the text it will generate. By using prompts, the model can better understand what kind of output is expected and produce more accurate and relevant results. In Llama 2 the size of the context, in terms of number of tokens, has doubled from 2048 to 4096.

Crafting Effective Prompts

Crafting effective prompts is an important part of prompt engineering. Here are some tips for creating prompts that will help improve the performance of your language model:

1. **Be clear and concise:** Your prompt should be easy to understand and provide enough information for the model to generate relevant output. Avoid using jargon or technical terms that may confuse the model.

2. **Use specific examples:** Providing specific examples in your prompt can help the model better understand what kind of output is expected. For example, if you want the model to generate a story about a particular topic, include a

Meta

3. **Vary the prompts:** Using different prompts can help the model learn more about the task at hand and produce more diverse and creative output. Try using different styles, tones, and formats to see how the model responds.

4. **Test and refine:** Once you have created a set of prompts, test them out on the model to see how it performs. If the results are not as expected, try refining the prompts by adding more detail or adjusting the tone and style.

5. **Use feedback:** Finally, use feedback from users or other sources to continually improve your prompts. This can help you identify areas where the model needs more guidance and make adjustments accordingly.

Role Based Prompts

Creating prompts based on the role or perspective of the person or entity being addressed. This technique can be useful for generating more relevant and engaging responses from language models.

Pros:

1. **Improves relevance:** Role-based prompting helps the language model understand the role or perspective of the person or entity being addressed, which can lead to more relevant and engaging responses.

2. **Increases accuracy:** Providing additional context about the role or perspective of the person or entity being addressed can help the language model avoid making mistakes or misunderstandings.

Cons:

Meta

Example:

You are a virtual tour guide currently walking the tourists Eiffel Tower on a night tour. Describe Eiffel Tower to your audience that covers its history, number of people visiting each year, amount of time it takes to do a full tour and why do so many people visit this place each year.

Chain of Thought Technique

Involves providing the language model with a series of prompts or questions to help guide its thinking and generate a more coherent and relevant response. This technique can be useful for generating more thoughtful and well-reasoned responses from language models.

Pros:

1. **Improves coherence:** Helps the language model think through a problem or question in a logical and structured way, which can lead to more coherent and relevant responses.

2. **Increases depth:** Providing a series of prompts or questions can help the language model explore a topic more

Cons:

1. **Requires effort:** The chain of thought technique requires more effort to create and provide the necessary prompts or questions.

Example:

You are a virtual tour guide from 1901. You have tourists visiting Eiffel Tower. Describe Eiffel Tower to your audience. Begin with
1. Why it was built
2. Then by how long it took them to build
3. Where were the materials sourced to build
4. Number of people it took to build
5. End it with the number of people visiting the Eiffel tour annually in the 1900's, the amount of time it completes a full tour and why so many people visit this place each year.
Make your tour funny by including 1 or 2 funny jokes at the end of the tour.

Reduce Hallucinations

Here are some examples of how a language model might hallucinate and some strategies for fixing the issue:

Example 1:

A language model is asked to generate a response to a question about a topic it has not been trained on. The language model may hallucinate information or make up facts that are not accurate or supported by evidence.

Fix: To fix this issue, you can provide the language model with more context or information about the topic to help it understand what is being asked and generate a more accurate response. You could also ask the language model to provide sources or evidence for any claims it makes to ensure that its responses are based on factual information.

Example 2:

A language model is asked to generate a response to a question that requires a specific perspective or point of view. The language model may hallucinate information or make up facts that are not consistent with the desired perspective or point of view.

Fix: To fix this issue, you can provide the language model with additional information about the desired perspective or point of view, such as the goals, values, or beliefs of the person or entity being addressed. This can help the language model understand the context and generate a response that is more consistent with the desired perspective or point of view.

Example 3:

Fix: To fix this issue, you can provide the language model with additional information about the desired tone or style, such as the audience or purpose of the communication. This can help the language model understand the context and generate a response that is more consistent with the desired tone or style.

Overall, the key to avoiding hallucination in language models is to provide them with clear and accurate information and context, and to carefully monitor their responses to ensure that they are consistent with your expectations and requirements.

Prompting with Llama

Format

```
<s>[INST]
{{ user_message }} [/INST]
```

Multi Turn User Prompt

```
<s>[INST]
{{ user_message_1 }} [/INST] {{ llama_answer_1 }} </s><s>[INST] {{ user_message_2 }} [/INS
```

Meta

Inferencing

Here are some great resources to get started with Inferencing with your LLMs.

Llama Recipes

→ **Github Llama recipes**

Page Attention vLLM

→ **Learn more** → **Recipe examples**

Hugging Face TGI

→ **Learn more** → **Recipe examples**

Learn more

Validation

As the saying goes, if you can't measure it, you can't improve it. In this section, we are going to cover different ways to measure and ultimately validate Llama so it's possible to determine the improvements provided by different fine tuning techniques.

Quantitative techniques

The focus of these techniques is to gather objective metrics that can be compared easily during and after each fine tuning run, to provide quick feedback on whether the model is performing. The main metrics collected are loss and perplexity.

K-Fold Cross-Validation

This method consists in dividing the dataset into k subsets or folds, and then fine tuning the model k times. On each run, a different fold is used as a validation dataset, using the rest for training. The performance results of each run are averaged out for the final report. This provides a more accurate metric of the performance of the model across the complete dataset, as all entries serve both for validation and training. While it produces the most accurate prediction on how a model is going to generalize after fine tuning on a given dataset, it is computationally expensive and better suited for small datasets.

Holdout

When using a holdout, the dataset is split into two or three subsets, training and validation with test as optional. The test and validation sets can represent 10% - 30% of the dataset each. As the name implies, the first two subsets are used for training to evaluate the model after fine-tuning for an unbiased view into the model performance, but it requires a slightly bigger dataset to allow for a proper split. This is currently implemented in the Llama recipes fine tuning script with two subsets of the dataset, train and validation. The data is collected in a json file that can be plotted to easily interpret the results and evaluate how the model is performing.

Standard Evaluation tools

There are multiple projects that provide standard evaluation. They provide predefined tasks with commonly used metrics to evaluate the performance of LLMs, like HellaSwag and ThroughtfulQA. These tools can be used to test if the model has degraded after fine tuning. Additionally, a custom task can be created using the dataset intended to fine-tune the

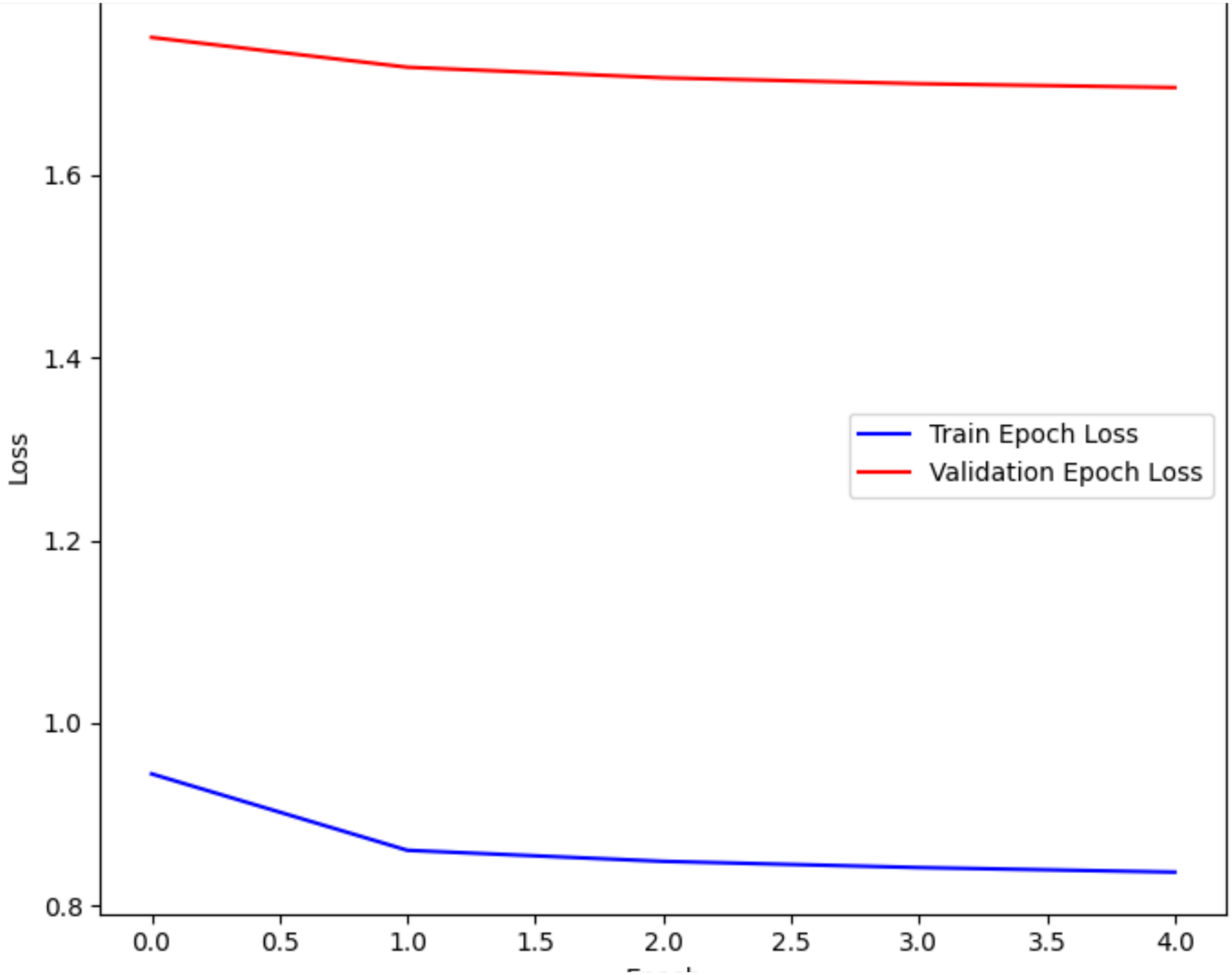
model, effectively automating the manual verification of the model performance before and after fine tuning. These types of projects provide a quantitative way of looking at the models performance in simulated real world examples. Some of these projects include the LM Evaluation Harness (used to create the HF leaderboard), HELM, BIG-bench and OpenCompass.

Interpreting Loss and Perplexity

The loss value used comes from the transformer's LlamaForCausalLM, which initializes a different loss function depending on the objective required from the model. The objective of this section is to give a brief overview on how to understand the results from loss and perplexity as an initial evaluation of the model performance during fine tuning. We also calculate the perplexity as an exponentiation of the loss value. Additional information on loss functions can be found in these resources: 1, 2, 3, 4, 5, 6.

In our recipes, we use a simple holdout during fine tuning. Using the logged loss values, both for train and validation dataset, the curves for both are plotted to analyze the results of the process. Given the setup in the recipe, the expected behavior is a log graph that shows a diminishing train and validation loss value as it progresses.

Meta


Loss
1.6
1.4
1.2
1.0
0.8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Train Epoch Loss
Validation Epoch Loss

If the validation curve starts going up while the train curve continues decreasing, the model is overfitting and it's not generalizing well. Some alternatives to test when this happens are early stopping, verifying the validation dataset is a statistically significant equivalent of the train dataset, data augmentation, using parameter efficient fine tuning or using k-fold cross-validation to better tune the hyperparameters.

Qualitative techniques

Manual testing

Manually evaluating a fine tuned model will vary according to the FT objective and available resources. Here we provide general guidelines on how to accomplish it.

With a dataset prepared for fine tuning, a part of it can be separated into a manual test subset, which can be further increased with general knowledge questions that might be relevant to the specific use case. In addition to these general questions, we recommend executing standard evaluations as well, and compare the results with the baseline for the fine tuned model.

To rate the results, a clear evaluation criteria should be defined that is relevant to the dataset being used. Example criteria can be accuracy, coherence and safety. Create a rubric for each criteria and define what would be required for an output to receive a specific score.

With these guidelines in place, distribute the test questions with a diverse set of reviewers to have multiple data points for each question. With multiple data points for each question and different criteria, a final score can be calculated for each query, allowing for weighting the scores based on the preferred focus for the final model.

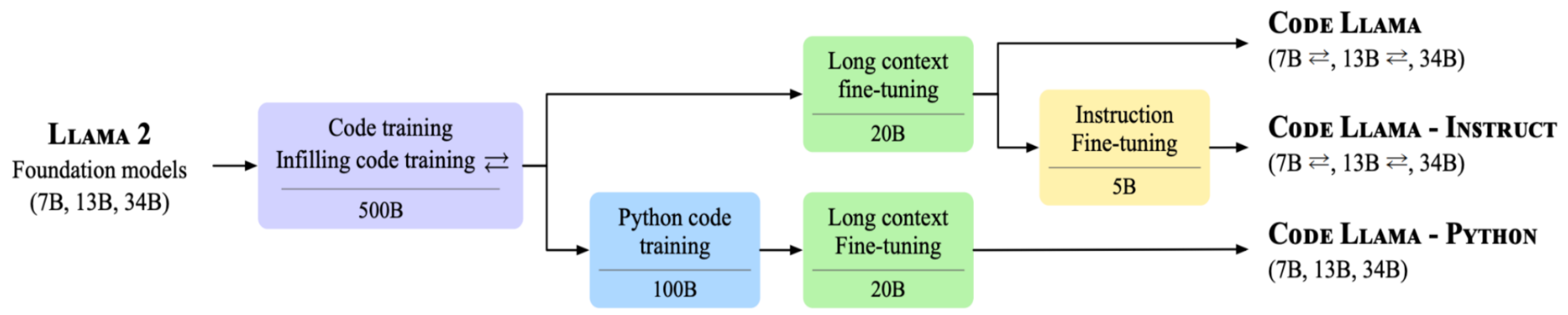
Meta

INTEGRATION GUIDES

Code Llama

Code Llama is an open-source family of LLMs based on Llama 2 providing SOTA performance on code tasks. It consists of:

- Foundation models (Code Llama)
- Python specializations (Code Llama - Python), and
- Instruction-following models (Code Llama - Instruct)with 7B, 13B and 34B parameters each.



One of the best ways to try out and integrate with Code Llama is using Hugging Face ecosystem by following the blog, which has:

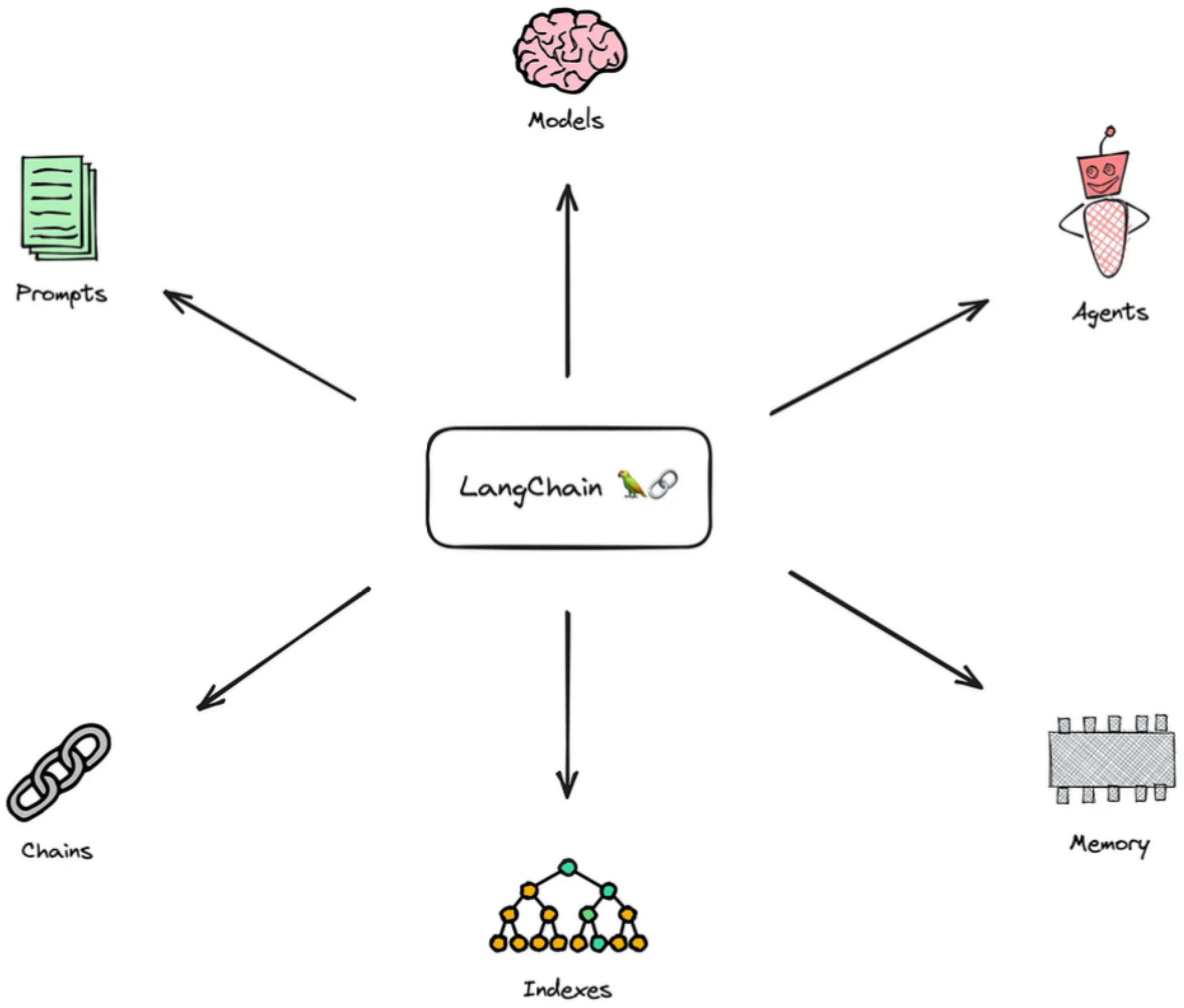
- Demo links for Code Llama 13B, 13B-Instruct (chat), and 34B.
- Working inference code for code completion
- Working inference code for code infilling between code prefix and suffix as inputs
- Working inference code to do 4-bit loading of the 32B model so it can fit on consumer GPUs
- Guide on how to write prompts for the instruction models to have multi-turn conversations about coding
- Guide on how to use Text Generation Inference for model deployment in production
- Guide on how to integrate code autocomplete as an extension with VSCode
- Guide on how to evaluate Code Llama models

If the model does not perform well on your specific task, for example if none of the Code Llama models (7B/13B/34B) generate the correct answer for a text to SQL task, fine-tuning should be considered. This is a complete guide and notebook on how to fine-tune Code Llama using the 7B model hosted on Hugging Face. It uses the LoRA fine-tuning method and can run on a single GPU.As shown in the Code Llama References (here), fine-tuning improves the performance of Code Llama on SQL code generation, and it can be critical that LLMs are able to interoperate with structured data and SQL, the primary way to access structured data - we are developing demo apps in LangChain and RAG with Llama 2 to show this.

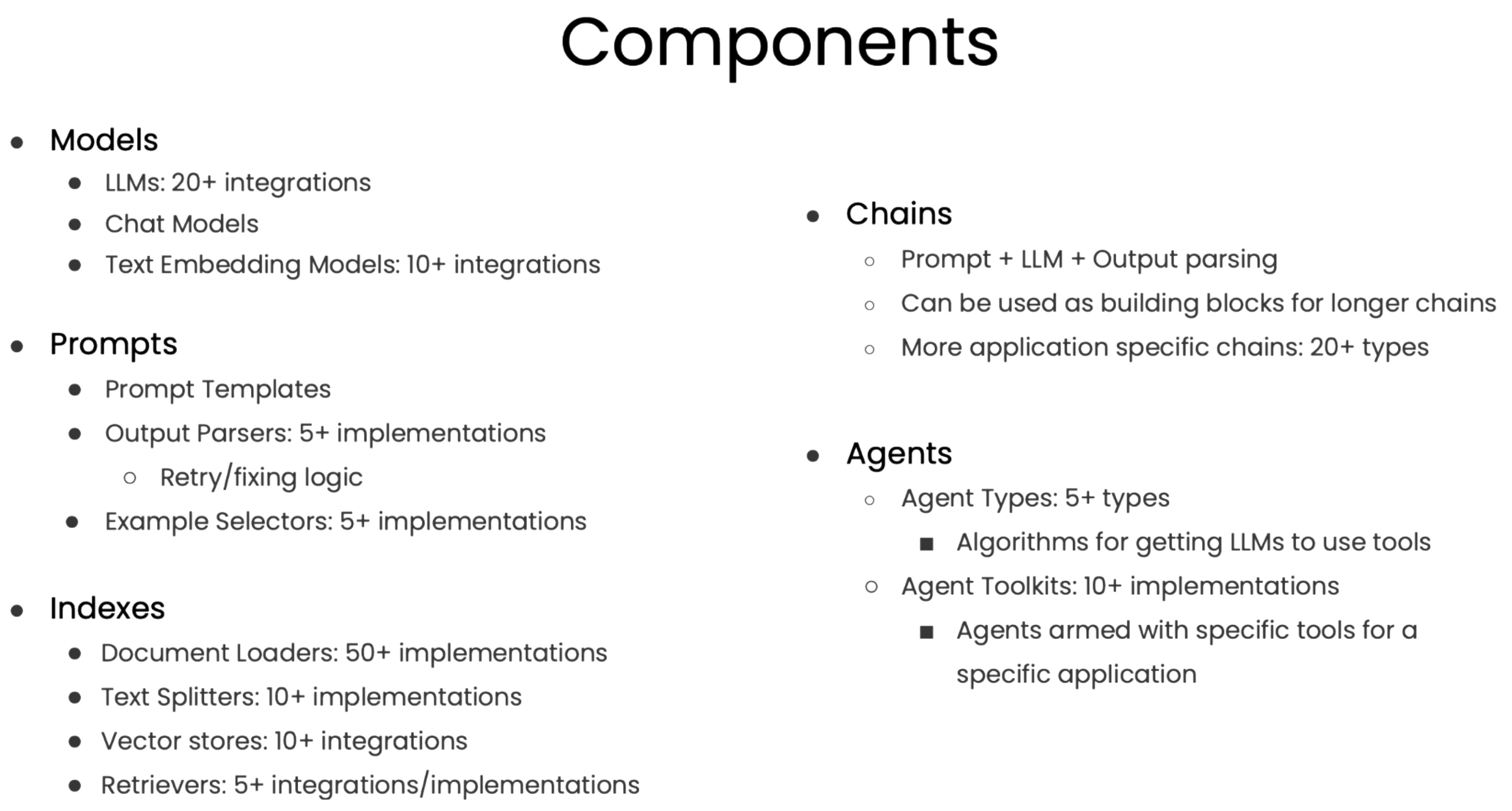
LangChain

LangChain is an open source framework for building LLM powered applications. It implements common abstractions and higher-level APIs to make the app building process easier, so you don't need to call LLM from scratch. The main building blocks/APIs of LangChain are:

Meta



Source



Components

- Models
  - LLMs: 20+ integrations
  - Chat Models
  - Text Embedding Models: 10+ integrations
- Prompts
  - Prompt Templates
  - Output Parsers: 5+ implementations
    - Retry/fixing logic
  - Example Selectors: 5+ implementations
- Indexes
  - Document Loaders: 50+ implementations
  - Text Splitters: 10+ implementations
  - Vector stores: 10+ integrations
  - Retrievers: 5+ integrations/implementations
- Chains
  - Prompt + LLM + Output parsing
  - Can be used as building blocks for longer chains
  - More application specific chains: 20+ types
- Agents
  - Agent Types: 5+ types
    - Algorithms for getting LLMs to use tools
  - Agent Toolkits: 10+ implementations
    - Agents armed with specific tools for a specific application

Source

Meta

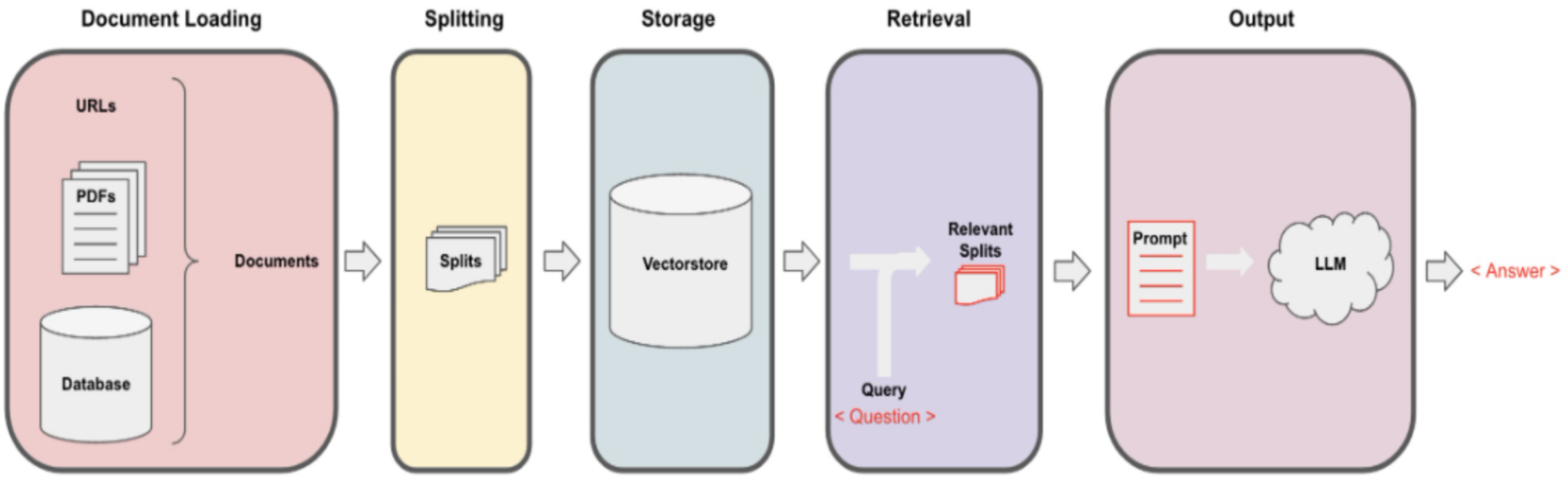
- The Prompts API implements the useful prompt template abstraction to help you easily reuse good, often long and detailed, prompts when building sophisticated LLM apps. There are also many built-in prompts for common operations such as summarization or connection to SQL databases for quick app development. Prompts can also work closely with parsers to easily extract useful information from the LLM output.

- The Memory API can be used to save conversation history and feed it along with new questions to LLM so multi-turn natural conversation chat can be implemented.

- The Chains API includes the most basic LLMChain that combines a LLM with aprompt to generate the output, as well as more advanced chains to lets you build sophisticated LLM apps in a systematic way. For example, the output of the first LLM chain can be the input/prompt of another chain, or a chain can have multiple inputs and/or multiple outputs, either pre-defined or dynamically decided by the LLM output of a prompt.

- The Indexes API allows documents outside of LLM to be saved, after first converted to embeddings which are numerical meaning representations, in the vector form, of the documents, to a vector store. Later when a user enters a question about the documents, the relevant data stored in the documents' vector store will be retrieved and sent, along with the query, to LLM to generate an answer related to the documents. The following flow shows the process:

Meta



Source

- The Agents API uses LLM as the reasoning engine and connects it with other sources of data, third-party or own tools, or APIs such as web search or wikipedia APIs. Depending on the user's input, the agent can decide which tool to call to handle the input.

LangChain can be used as a powerful retrieval augmented generation (RAG) tool to integrate the internal data or more recent public data with LLM to QA or chat about the data. LangChain already supports loading many types of unstructured and structured data.

There is also a Getting to Know Llama notebook, presented at Meta Connect 2023.

LlamaIndex

LlamaIndex is another popular open source framework for building LLM applications. Like LangChain, LlamaIndex can also be used to build RAG applications by easily integrating data not built-in the LLM with LLM. There are three key tools in LlamaIndex:

- **Connecting Data:** connect data of any type - structured, unstructured or semi-structured - to LLM
- **Indexing Data:** Index and store the data
- **Querying LLM:** Combine the user query and retrieved query-related data to queryLLM and return data-augmented answer

Return data-augmented answer

LlamaIndex is mainly a data framework for connecting private or domain-specific data with LLMs, so it specializes in RAG, smart data storage and retrieval, while LangChain is a more general purpose framework which can be used to build agents connecting multiple tools. The integration of the two may provide the best performant and effective solution to building real world RAG powered Llama apps.

For an example usage of how to integrate LlamaIndex with Llama 2, see here. We also published a completed demo app showing how to use LlamaIndex to chat with Llama 2 about live data via the you.com API.

Meta

- **Question Generation:** Call LLM to auto generate questions to create an evaluation dataset.
- **FaithfulnessEvaluator:** Evaluate if the generated answer is faithful to the retrieved context or if there’s hallucination.
- **CorrectnessEvaluator:** Evaluate if the generated answer matches the reference answer.
- **RelevancyEvaluator:** Evaluate if the answer and the retrieved context is relevant and consistent for the given query.

COMMUNITY SUPPORT AND RESOURCES

Community Support

If you have any feature requests, suggestions, bugs to report we encourage you to report the issue in the respective github repository. If you are working in partnership with Meta on Llama 2 please request access to Asana and report any issues using Asana.

Resources

Github

- Code Llama Repository : Main Code Llama repository
- Getting to know Llama 2 - Jupyter Notebook
- Code Llama Recipes : Examples

Performance & Latency

- Hamel's Blog - Optimizing and testing latency for LLMs
- vLLM - How continuous batching enables 23x throughput in LLM inference while reducing p50 latency
- Paper - Improving performance of compressed LLMs with prompt engineering
- Llama2 vs GPT 4 Cost comparison for text summarization

Fine Tuning

- Hugging Face PEFT
- Llama Recipes Fine Tuning
- Fine Tuning Data Sets
- GPT 3.5 vs Llama 2 fine-tuning
- How to Fine-tune Llama 2 with LoRA for Question Answering
- Efficient Fine-Tuning with LoRA
- Weights & Biases Training and Fine-tuning Large Language Models

Code Llama

- Fine-Tuning Improves the Performance of Meta's Code Llama on SQL Code Generation
- Beating GPT-4 on HumanEval with a Fine-Tuned CodeLlama-34B
- Introducing Code Llama, a state-of-the-art large language model for coding

Meta

- Llama on Hugging Face
- Building LLM applications for production
- Prompting Techniques

We value your feedback

Help us improve Llama by submitting feedback, suggestions, or reporting bugs.

Submit feedback

Who We Are

About
People

Latest Work

Research
Infrastructure

Meta

Our Actions

Responsibilities

Newsletter

Sign Up

Privacy Policy Terms Cookies