# Training LLMs Module

## Training LLMs

**Goals**: Provide a hands-on coding experience for training an LLM from scratch in the cloud. Introduce domain-specific LLMs and guide students on benchmarking their custom LLMs.

As we transition into the practical aspects, the focus shifts to training LLMs in the cloud and the significance of efficient scaling techniques. With a focus on benchmarking LLMs and the strategic application of domain-specific models in various sectors, it provides a meticulous understanding of tools like Deep Lake, their role in refining LLM training, and the importance of dataset curation.

- **When to Train an LLM from Scratch**: This lesson will help with the decision of when to train an LLM from scratch versus utilizing a pre-trained model and the tradeoffs of using both proprietary and open-source models. Specific references to models like **BloombergGPT**, **FinGPT**, and **LawGPT** will be highlighted, offering real-world context. We also address the ongoing debate about the advantages and challenges of training domain-specific LLMs.
- **LLMOps**: This lesson touches upon LLMOps, a specialized practice catering to the operational needs of Large Language Models. It is imperative to have dedicated operations for LLMs to streamline deployment, maintenance, and scaling. We will underscore the significance of tools like Weights & Biases in managing and optimizing LLMs, emphasizing their role in modern LLMOps practices.
- **Overview of the Training Process**: This lesson provides sequential steps essential to an LLM training process. We will gather and refine data, then move to model initialization and set training parameters, often using the Trainer class. The lesson continues with monitoring via an evaluation dataset.
- **Deep Lake and Data Loaders**: This lesson covers Deep Lake and its affiliated data loaders and their role in the LLM training and finetuning process. We will discuss the utility of these tools and gain insights into how they streamline data handling and model optimization.
- **Datasets for Training LLMs**: This lesson dives into diverse datasets for LLM training in text and coding. Students learn the intricacies of curating specialized datasets, with an example of storing data in Deep Lake. We emphasize on data quality, referencing the "Textbooks Are All You Need" research.
- **Train an LLM in the Cloud**: The lesson lays out the process of training LLMs in the cloud using the Hugging Face Accelerate library and Lambda. We offer practical insights into integrating these platforms with hands-on guidance on leveraging data from a Deep Lake dataset.
- **Tips for Training LLMs at Scale:** This lesson offers students valuable strategies for efficiently scaling LLM training. It emphasizes advanced techniques and optimizations.
- **Benchmarking your own LLM**: The lesson centers around the importance of benchmarking LLMs. We will discuss tools such as InstructEval and Eleuther's Language Model Evaluation Harness. We also provide hands-on experience in assessing their LLM's performance against recognized benchmarks.
- **Domain-specific LLMs**: The lesson covers the strategic use of domain-specific LLMs. We explore scenarios where these specialized models are most effective, with a spotlight on popular instances like FinGPT and BloombergGPT. By analyzing these cases, we will understand the nuances of utilizing domain-specific LLMs to cater to unique industry demands.

Upon completing this comprehensive module, participants have gained insights into the multifaceted landscape of training LLMs. From understanding the tradeoffs between training from scratch versus leveraging pre-existing models to LLMOps, students are well-equipped to benchmark their LLMs effectively and understand the nuanced value of domain-specific models in meeting specific industry requirements. The following section will introduce learners to the complexities that come with finetuning techniques and practical hands-on projects.

# When to Train an LLM from Scratch

## Introduction

The increasing popularity of LLMs has led businesses to integrate them for task handling and employee productivity enhancement.
There are several ways to use LLMs in daily activities, such as incorporating proprietary models via APIs, deploying pre-trained open-source options, or developing one's own language model. Of course, the trade-offs are between quality, costs, and ease of use.
In this lesson, we will discuss different approaches and what might be the best solution for your use case.

## Few-Shot (In-Context) Learning

Up to 2020, language models were already good at picking up patterns from the data. However, teaching them new knowledge from a different domain was difficult. The only solution was to finetune them by adjusting the weights.
**What are the characteristics that set LLMs apart from the previous language models?**
Few-shot learning (also called In-Context learning) enables the LLMs to learn from the examples provided to them. For instance, it is possible to show a couple of examples of JSON-formatted responses to receive the model's output in JSON format. It means that the models can learn from examples and follow directions without changing weights or repeating the training process.
There are multiple use cases where this approach could be the best option. The model can adapt to a writing style, set specific formatting guidelines, or provide additional context for answering questions.
LLMs are able to answer questions using external knowledge bases through in-context learning. Let's think about how we could create a Q&A chatbot leveraging an LLM. The LLM has a cut-off training date, so it can't access the information or events after that date. Also, they tend to hallucinate, which refers to generating non-factual responses based on their limited knowledge. As a solution, it is possible to provide additional context to the LLM through the Internet (e.g., Google search) or retrieve it from a database and include it in the prompt so that the model can leverage it to generate the correct response. It is like taking an open-book exam!
The beauty of this approach is that the model does not need domain-specific knowledge. Instead, it can extract information or patterns from the provided context. Creating applications, such as chatbots, becomes more accessible and faster. Whether you are utilizing proprietary APIs or open-source models, this approach offers a budget-friendly solution for many use cases.

# Fine-Tuning

The fine-tuning method proves valuable when adapting the model to a more complex use case. This technique can improve model understanding by providing more examples and adjusting weights based on errors, for tasks like classification or summarization.
There are different approaches to doing this. We could either adjust the weights with a small learning rate to minimally affect the model's current abilities, or a more recent technique is to freeze the network and introduce new weights for fine-tuning. The latter approach (like **LoRA**) is a great alternative for fine-tuning models with hundreds of billions of parameters since we will deal with a much smaller number of parameters. (~100x less)
The fine-tuning approach is an excellent option for creating a model with task-specific knowledge and building on top of the available powerful LLMs. However, before considering this option, it is essential to acknowledge the associated costs and required resource implications.

# Training

Lastly, let's talk about training your own model from scratch!

Among the approaches mentioned earlier, this option stands out as the most demanding and challenging. Of course, the scale of requirements depends on the model size. However, acquiring several millions of data points, such as web pages, books, and articles, not to mention the task-specific documents held by your organization (if you want to train a domain-specific LLM), is essential. Furthermore, completing the training process could cost upward of several hundreds of thousands of dollars. The training costs of these models are rarely revealed by the organizations that publish them. Nevertheless, considering the hardware utilized, speculations have estimated the training expenses for the GPT-3 model to be approximately $4.6 million.

However, the more critical aspect of training from scratch is curating the dataset. While the intention is to train a domain-specific model, the training loop that processes vast quantities of general documents, such as web pages, articles, and books, empowers LLMs' language understanding capabilities. Therefore, to create a model that excels in a specific domain, it is essential to have a sizable dataset comprising top-quality samples from that particular domain.

An example of this approach is the BloombergGPT 50B model, which is specifically designed for the finance industry. They used a dataset of 708 billion tokens for training, consisting of 51.2% (363 billion tokens) domain-specific resources and the rest general resources.

Training a model from scratch demands substantial resources, including hardware and dataset resources, and expertise within the organization to train and maintain these models.

## Main Takeaways

- **Few-Shot Learning**: The LLMs are able to learn from the examples given to them, allowing them to handle more complicated tasks without the need for training or fine-tuning. This method is significantly less expensive than other options, as it only requires the cost of adding examples to each prompt. If your task can be solved just with few-shot learning, then it’s always the most efficient approach.
- **Fine-Tuning**: If few-shot learning is not effective for your task, an alternative method is fine-tuning. This involves using some data points to create a task-specific model. Although finetuning can be challenging when acquiring new knowledge, it is more effective in adapting to different styles, and tones, or incorporating new vocabulary.
- **Training From Scratch**: If fine-tuning is not effective, consider training a model from scratch with domain-specific data. However, this requires significant resources, such as cost, dataset availability, and expertise.

## Conclusion

We have explored various methods to harness the capabilities of large language models within your organization and highlighted the advantages and disadvantages of each approach. Picking the best practice depends on your organization’s use case and the resources at hand.

This course aims to equip you with the necessary knowledge to make informed decisions about which approach best suits your needs. It will also guide on maximizing the benefits of large language models and mastering the process involved.

# What is LLMOps

## Introduction

As LLMs continue to revolutionize various applications, managing their lifecycle has become important. In this lesson, we will explore the concept of LLMOps, its origins, and its significance in today's AI industry. We will also discuss the steps involved in building an LLM-powered application, the differences between LLMOps and MLOps, and the challenges and solutions associated with each step.

## The Emergence of LLMOps

In recent years, the world of AI has witnessed the rise of large language models. These models have billions of parameters and are trained on billions of words, hence the term "large." The advent of LLMs has led to the emergence of a new term, LLMOps, which stands for Large Language Model Operations. This lesson aims to comprehensively understand LLMOps, its origins, and its significance in the AI industry.

LLMOps is essentially a set of tools and best practices designed to **manage the GenAI lifecycle**, from development and deployment to maintenance.

LLMOps have gained traction with the rise of LLMs, particularly after the release of OpenAI's ChatGPT, which led to a surge in LLM-powered applications, such as chatbots, writing assistants, and programming assistants.

However, the process of building production-ready LLM-powered applications presents unique challenges that differ from those encountered when building AI products with traditional machine learning models. This has necessitated the development of new tools and practices, giving birth to the term "LLMOps."

## Steps Involved in LLMOps and Differences with MLOps

While LLMOps can be considered a subset of MLOps (Machine Learning Operations), there are key differences between the two, primarily due to the differences in building AI products with classical ML models and LLMs.

The process of building an LLM-powered application involves several key steps.

### 1. Selection of a Foundation Model

Foundation models are pre-trained LLMs that can be adapted for various downstream tasks. Training these models from scratch is complex, time-consuming, and costly. Hence, developers usually opt for either proprietary models owned by large companies or open-source models hosted on community platforms like Hugging Face.

This differs from standard MLOps, where a model is typically trained from scratch with a smaller architectures or on different data, especially for tabular classification and regression tasks (except for computer vision, where most applications start with a model trained on general datasets like Imagenet or COCO). Typically a dataset is **splitted into training and evaluation** sets, where 70% of the data go into the training set, or other evaluation techniques like crossvalidation are used. When working **with LLMs, this is not possible** due to the high costs involved in the pretraining. MLOps models are data-hungry and require a lot (thousands at least) of labeled data to be trained on.

Consequently, choosing the suitable foundation model in LLMOps is very important, as crucial as choosing a proprietary or open-source foundation model. Proprietary LLMs are usually bigger and more performant than open-source alternatives (thanks to the money investments that large corporations can make) and may also be more cost-effective for the final user as there's no need to set up an expensive infrastructure to host the model (which organizations can do efficiently, as

they have many customers and amortize the costs). On the contrary, open-source models are generally more customizable and can be improved by anyone from the open-source community, indeed they soon matched the quality of many proprietary LLMs.
Another aspect to consider is the **knowledge cutoff of LLMs**: the date of the last published document on which the model was trained. For example, the model used in ChatGPT is currently limited to data up until September 2021. Consequently, the model can easily talk about everything that happened before that date but finds it hard to talk about later stuff. For example, ChatGPT doesn't know about the latest startups or products released. Therefore, he may hallucinate when talking about them.

### 2. Adaptation to Downstream Tasks

After selecting a foundation model, it can be customized for specific tasks through techniques such as **prompt engineering**. This involves adjusting the input to produce the desired output.
It's important to keep track of the prompts used when using prompt engineering since they will likely be improved over time and can impact performance on specific tasks. By doing this, if a new prompt in production works worse than the previous one in some aspects and If we want to revert, it can be done easily.
Additionally, **fine-tuning** can be utilized to enhance the model's performance on a specific task, requiring a high-quality dataset for it (thus, involving a data collection step). In the case of fine-tuning, there are different approaches such as **fine-tuning the model, fine-tuning the instructions, or using soft prompts.** There are challenges with fine-tuning due to the large size of the model. Additionally, deploying the newly finetuned model on a new infrastructure can be difficult. To solve this problem, today, there are finetuning techniques that improve only a small subset of additional parameters to add to the existing foundational model, such **as LoRA**. Using LoRA, it's possible to keep the same foundation model always deployed on the infrastructure while adding the additional finetuned parameters when needed. Recently, popular proprietary models like GPT3.5 and PaLM can now be finetuned easily directly on the company platform.
When fine-tuning a model, it's essential to keep track of the dataset used and the metrics achieved. It can be helpful to use a tool like **Weights and Biases**, which **tracks experiments** and provides a dashboard where you can monitor the metrics of your fine-tuned model on an evaluation set as it is trained. This provides insights into whether the training is progressing well or not. See this page to learn more about how W&B experiment tracking works. It will be used in the following lessons to train and fine-tune language models.

### 3. Evaluation

Evaluating the performance of an LLM is more complex than evaluating traditional ML models. The main reason for this is that the output of an LLM is usually free text, and it's harder to devise metrics that can be computed via code and that work well on free text. For example, try thinking about how you could evaluate the quality of an answer given by an LLM assistant whose job is to summarize YouTube videos, for which you don't have reference summaries written by humans. Currently, organizations often resort to A/B testing to assess the effectiveness of their models, checking whether the user's satisfaction is the same or better after the change in production.
Another aspect to consider is hallucinations. How can we measure, with a metric implemented in code, whether the answer of our LLM assistant contains hallucinations? This is another open challenge where organizations mainly rely on A/B testing.

### 4. Deployment and Monitoring

Deploying and monitoring LLMs is very important as their completions can change significantly between releases. Tools for monitoring LLMs are emerging to address this need.
Another concern about LLMOps is the **latency** of the model. Indeed, since the model is autoregressive (i.e., produces the output one token at a time), it may take some time to output a complete paragraph. This is in contrast with the most popular applications of LLMs, which want

them as assistants, which, therefore, should be able to output text at a throughput similar to a user's reading speed.
One of the emerging tools in the LLMOps landscape is **W&B Prompts**, a suite designed specifically for the development of LLM-powered applications. W&B Prompts offers a comprehensive set of features that allow developers to visualize and inspect the execution flow of LLMs, analyze the inputs and outputs, view intermediate results, and securely manage prompts and LLM chain configurations.
A key component of W&B Prompts is Trace, a tool that tracks and visualizes the inputs, outputs, execution flow, and model architecture of LLM chains. Trace is particularly useful for LLM chaining, plug-in, or pipelining use cases. It provides a Trace Table for an overview of the inputs and outputs of a chain, a Trace Timeline that displays the execution flow of the chain color-coded according to component types, and a Model Architecture view that provides details about the structure of the chain and the parameters used to initialize each component.
LLMOps is a rapidly evolving field, and it's hard to predict its future trajectory. However, it's clear that as LLMs become more prevalent, so will the tools and practices associated with LLMOps. The rise of LLMs and LLMOps signifies a major shift in building and maintaining AI-powered products.

## Conclusion

In conclusion, LLMOps, or Large Language Model Operations, is a critical aspect of managing the lifecycle of applications powered by LLMs. This lesson has provided an overview of the origins and significance of LLMOps, the steps involved in building an LLM-powered application, and the differences between LLMOps and MLOps.
We studied the process of selecting a foundation model, adapting it to downstream tasks, evaluating its performance, and deploying and monitoring the model. We've also highlighted the unique challenges posed by LLMs, such as the complexity of evaluating free text outputs and the need for prompt versioning and efficient deployment strategies.
The emergence of tools like W&B Prompts and practices like A/B testing are indicative of the rapid evolution of LLMOps. As LLMs continue to revolutionize various applications, the tools and practices associated with LLMOps will undoubtedly become increasingly important in AI.

# Overview of the Training Process

## Introduction

In this lesson, we will provide an overview of the multiple steps involved in training LLMs, including preprocessing the training data, the model architecture, and the training process. By the end of this lesson, you will have a solid understanding of the various steps involved in training large language models.
The training process begins with the selection of one or a combination of suitable datasets, proceeds with the initialization of the neural network, and ultimately concludes with the execution of the training loop. We will also discuss the process of saving the weights for future utilization. Although this process may seem challenging, we will break it down into steps and approach each one separately to aid your understanding of the intricacies involved.

## The Dataset

Whether you are training a general LLM or a specialized one for a specific domain, curating a comprehensive database containing relevant information is the most crucial step. The progress made in the field of neural networks and Natural Language Processing firmly establishes one

architecture as the go-to choice (the transformer), thereby emphasizing the significant impact of dataset size and quality on the model's performance.
There are several well-known databases that you can use as a source of public knowledge. For instance, consider datasets like The Pile, Common Crawl, or Wikipedia, which entail extensive collections of web pages, articles, or books. Each of these datasets is comprised of hundreds of billions of tokens, providing diverse learning material for the model.
These datasets are mostly available publicly through different sources. We prepared **a Deep Lake repository** containing several datasets that we'll use in this course; find it here.
The next category of datasets is important only if you are training a model for a specific use case based on the data your organization has at hand or curated. Note that the size of your dataset may vary depending on your application and whether you opt for fine-tuning or training from scratch. The data source can be obtained through web scraping of news websites, forums, or publicly accessible databases, in addition to leveraging your own private knowledge base. It's also possible to use a foundational LLM to generate a synthetic dataset of your own to be used for training a specialized domain LLM, which may be less expensive and faster than the big foundational model.
Splitting the dataset into training and validation sets is a standard process. The training set is utilized during the training process to optimize the model's parameters. On the other hand, the validation set is used to assess the model's performance and ensure it is not overfitting by evaluating its generalization ability.

## The Model

The transformer has been the dominant network architecture for natural language processing tasks in recent years. It is powered by the attention mechanism, which enables the models to accurately identify the relationship between words. This architecture has resulted in state-of-the-art scores for numerous NLP tasks over the years and powered well-known LLMs like the GPT family. Based on the literature, it is evident that increasing the number of parameters in transformer-based networks enhances language generation and comprehensive ability.
With the widespread adoption of transformers, you have the option to utilize libraries such as **Tensorflow, PyTorch,** and **Huggingface** to initialize the architecture. Alternatively, you can code it yourself by referring to numerous tutorials available to get a more in-depth understanding.
One of the benefits of utilizing the transformers library developed by Huggingface is the availability of their hub, which simplifies the process of loading open-source LLMs such as Bloom or OpenAssistant.

## Training

The first generation of foundational models like **BERT were trained with Masked Language Modeling (MLM)** learning objectives. This is achieved by randomly masking words from the corpus and configuring the model to predict the masked word. By employing this objective, the model gains the ability to consider the contextual information preceding and following the masked word, enabling it to make informed decisions. This objective may not be the most suitable choice for generative tasks, as ideally, the model should not have access to future words while predicting the current word.
The **GPT family models used the Autoregressive** learning objective. This algorithm ensures that the model consistently attempts to predict the next word without accessing the future content within the corpus. The training process will be iterative, which feeds back the generated tokens to the model to predict the next word. Masked attention ensures that, at each time step, the model is prevented from seeing future words.

To train or finetune models, you have the option to either implement the **training loop** using libraries such as **PyTorch or utilize the `Trainer` class provided by Huggingface**. The latter option enables us to easily configure different hyperparameters, log, save checkpoints, and evaluate the model.

## Conclusion

The training process often involves significant trial and error to achieve optimal results. Using libraries can significantly expedite the training process and save time by eliminating the need to implement various mechanisms manually. The model's capability is influenced by various factors, including its size, the size of the dataset, and the chosen hyperparameters, which collectively contribute to the complexity of the process.
In upcoming lessons, we will explore each training process step with more detailed explanations.

# Deep Lake and Data Loaders

## Introduction

In this lesson, we focus on Deep Lake, a powerful AI data system that merges the capabilities of Data Lakes and Vector Databases. We'll explore how Deep Lake can be leveraged for training and fine-tuning Large Language Models, with a focus on its efficient data streaming capabilities. We'll also learn how to create a Deep Lake dataset, add data, and load data using both Deep Lake's and PyTorch's data loaders.

## Deep Lake

In the following lessons about training and finetuning LLMs, we'll need to store the training datasets somewhere, especially for pretraining, since their size is usually too big to be memorized in a single computing node. Ideally, we'd store the datasets elsewhere and efficiently download data in batches when needed. This is where Deep Lake is most useful.
Deep Lake is a multi-modal AI data system that merges the capabilities of Data Lakes and Vector Databases. Deep Lake is particularly beneficial for businesses looking to train or fine-tune LLMs on their own data. It efficiently streams data from remote storage to GPUs during model training, making it a powerful tool for deep learning applications.
**Data loaders** in Deep Lake are essential components that facilitate efficient data streaming and are very useful for training and fine-tuning LLMs. They are responsible for **fetching, decompressing, and transforming data**, and they can be optimized to improve performance in GPU-bottlenecked scenarios. Once we store our datasets in Deep Lake, it's possible to easily create a PyTorch Dataloader or a TensorFlow Dataset.
Deep Lake offers two types of data loaders: the **Open Source data loader** and the **Performant data loader**. The Performant version, built on a C++ implementation, is faster and optimizes asynchronous data fetching and decompression. It's approximately 1.5 to 3 times faster than the OSS version, depending on the complexity of the transformation and the number of workers available for parallelization, and it supports distributed training.

## Creating a Deep Lake Dataset and Adding Data

Now, let's walk through an example of creating a Deep Lake dataset and fetching some data from it. Deep Lake supports a variety of data formats, and you can ingest them directly with a single line of code.

Deep Lake can be installed with pip as follows: `pip install deeplake`. Please note that the performant version can be used for free up to 200GB of data stored in the cloud, which is more than we'll need for the course.
Then, create an account at the Activeloop website. Next, you'll need an Activeloop API token, which will allow you to identify yourself from your Python code. To get it, click on the "Create API token" button that you can see at the top of your webpage once you're logged in, and then proceed to create one by clicking on the other "Create API token" button inside the page. Remember to check the token's expiration date: once it's expired, you'll need to create a new one from this page to continue using Deep Lake with Python linked to your account.
Once you have your Activeloop token, save it into the `ACTIVELOOP_TOKEN` environmental variable. You can do so by adding it to your `.env` file, which will then be loaded, executing the following Python code with the `dotenv` library.

```
from dotenv import load_dotenv

load_dotenv()
```

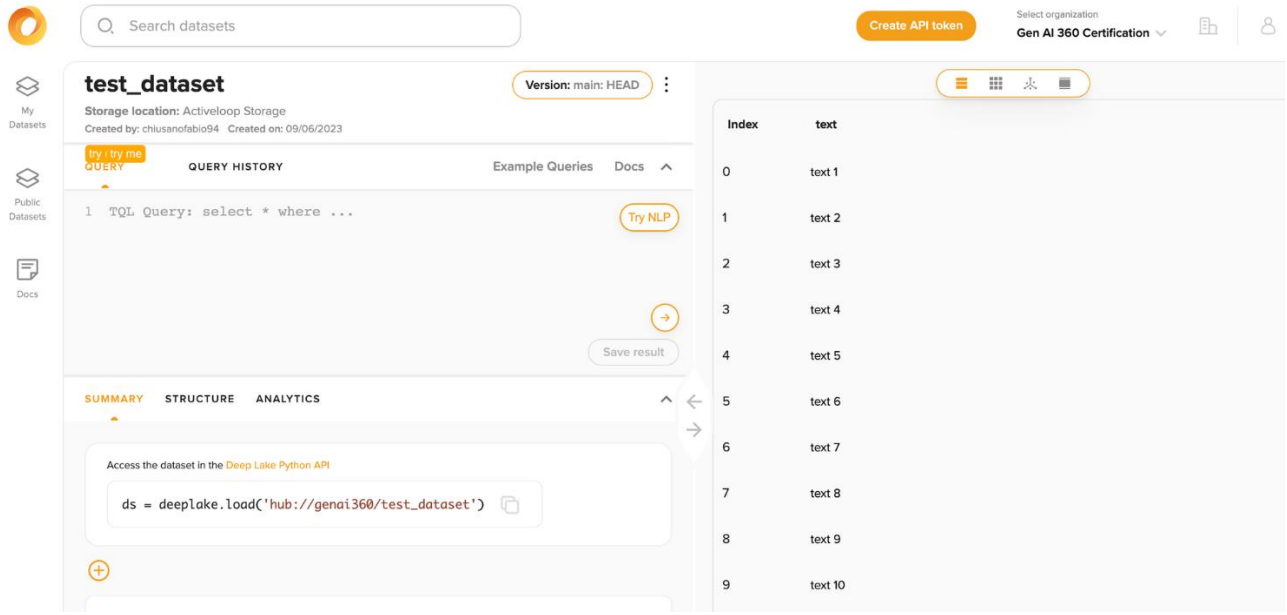
You are now ready to use Deep Lake! The following Python code shows how we can create a dataset using Deep Lake. Make sure to replace `<YOUR_ACTIVELOOP_USERNAME>` with your username on Activeloop. You can easily find it in the URL of your webpage, which should have the form `https://app.activeloop.ai/<YOUR_ACTIVELOOP_USERNAME>/home`.

```
import deeplake

# env variable ACTIVELOOP_TOKEN must be set with your API token

# create dataset on deeplake

username = "<YOUR_ACTIVELOOP_USERNAME>"

dataset_name = "test_dataset"

ds = deeplake.dataset(f"hub://{username}/{dataset_name}")


# create column text

ds.create_tensor('text', htype="text")


# add some texts to the dataset

texts = [f"text {i}" for i in range(1, 11)]

for text in texts:

    ds.append({"text": text})
```

In the previous code, we created a Deep Lake dataset named `test_dataset`. We specify that it contains texts, and then we add 10 data samples to it, one by one. Visit the API docs of Deep Lake to learn about the other available methods.
Once done, you should see printed text like the following.

```
This dataset can be visualized in Jupyter Notebook by ds.visualize() or at https://app.activeloop.ai/genai360/test_dataset
```

By clicking on the URL contained in it, you'll see your dataset directly from the Activeloop website.




Deep Lake dataset version control allows you to manage changes to datasets with commands very similar to Git. It provides critical insights into how your data is evolving, and it works with datasets of any size. Execute the following code to commit your changes to the dataset.
Copy

```
ds.commit("added texts")
```

# Retrieving Data From Deep Lake

Now, let's get some data from our Deep Lake dataset.
There are two main syntaxes for getting data from Deep Lake datasets:

1. The first one uses the Deep Lake **dataloader**. It's highly optimized and has the fastest data streaming. However, it doesn't support custom sampling or full-random shuffling. It is possible to use PyTorch datasets and data loaders. If you're interested in knowing more about how to use the Deep Lake data loader in cases where data shuffling is important, read this guide.
2. The second one uses plain **PyTorch datasets** and data loaders, enabling all the customizability that PyTorch supports. However, they have highly sub-optimal streaming using Deep Lake datasets and may result in 5x+ slower performance compared to using Deep Lake data loaders.

### The Deep Lake Data Loader for PyTorch

Here's a code example of creating a Deep Lake data loader for PyTorch. The following code leverages the performant Deep Lake data loader. It's the fastest and most optimized way of loading data in batches for model training.

```
# create PyTorch data loader

batch_size = 3

train_loader = ds.dataloader()\

    .batch(batch_size)\

    .shuffle()\

    .pytorch()
```

```
# loop over the elements
for i, batch in enumerate(train_loader):
    print(f"Batch {i}")
    samples = batch.get("text")
    for j, sample in enumerate(samples):
        print(f"Sample {j}: {sample}")
    print()
    pass
```

You should see the following printed output, showing the retrieved batches.

```
Please wait, filling up the shuffle buffer with samples.
Shuffle buffer filling is complete.

Batch 0
Sample 0: text 1
Sample 1: text 7
Sample 2: text 8

Batch 1
Sample 0: text 2
Sample 1: text 9
Sample 2: text 6

Batch 2
Sample 0: text 10
Sample 1: text 3
Sample 2: text 4

Batch 3
Sample 0: text 5
```

### PyTorch Datasets and PyTorch Data Loaders using Deep Lake

This code enables all the customizability supported by PyTorch at the cost of having highly slower streaming compared to using Deep Lake data loaders. The reason for the slower performance is that this approach does not take advantage of the inherent dataset format that was designed for fast streaming by Activeloop.

First, we create a subclass of the PyTorch `Dataset`, which stores a reference to the Deep Lake dataset and implements the `__len__` and `__getitem__` methods.

```
from torch.utils.data import DataLoader, Dataset

class DeepLakePyTorchDataset(Dataset):
    def __init__(self, ds):
        self.ds = ds

    def __len__(self):
        return len(self.ds)

    def __getitem__(self, idx):
        texts = self.ds.text[idx].text().astype(str)
        return { "text": texts }
```

Inside the `__getitem__` method, we retrieve the strings stored in the `text` tensor of the dataset at the position `idx`.
Then, we instantiate it using a reference to our Deep Lake dataset `ds`, transform it into a PyTorch `DataLoader`, and eventually loop over the elements just like we did with the Deep Lake dataloader example.

```
# create PyTorch dataset
ds_pt = DeepLakePyTorchDataset(ds)

# create PyTorch data loader from PyTorch dataset
dataloader_pytorch = DataLoader(ds_pt, batch_size=3, shuffle=True)

# loop over the elements
for i, batch in enumerate(dataloader_pytorch):
    print(f"Batch {i}")
    samples = batch.get("text")
    for j, sample in enumerate(samples):
        print(f"Sample {j}: {sample}")
    print()
    pass
```

You should see the following output, showing the retrieved batches.

```
Batch 0
Sample 0: text 8
Sample 1: text 3
Sample 2: text 1
```

```
Batch 1

Sample 0: text 4

Sample 1: text 5

Sample 2: text 9


Batch 2

Sample 0: text 7

Sample 1: text 2

Sample 2: text 6


Batch 3

Sample 0: text 10
```

## Getting the Best High-Quality Data for your Models

Recent research, such as from the "LIMA: Less Is More for Alignment" and "Textbooks Are All You Need" papers, suggests that data quality is very important both for training and finetuning LLMs. As a consequence, Deep Lake has several additional features that can help users investigate the quality of the datasets they are using and eventually filter samples.

Deep Lake provides the **Tensor Query Language (TQL)**, an SQL-like language used for Querying in Activeloop Platform as well as in `ds.query` in the Python API. This allows data scientists to filter datasets and focus their work on the most relevant data.

The following code shows how we can filter our dataset using a TQL query and print all the samples in the resulting view.

```
# code that creates a data loader and prints the batches

...


Batch 0

Sample 0: text 1

Sample 1: text 10
```

Now, we can save our dataset view as follows.

```
ds_view.save_view(id="strings_with_1")
```

And we can read from it as follows.

```
ds = deeplake.dataset(f"hub://{username}/{dataset_name}/.queries/strings_with_1")
```

Another feature is the **samplers**. Samplers can be used to assign a discrete distribution of weights to the dataset's samples, which are then sampled according to the weight distribution. This can be useful for focusing training on higher-quality data.

## Conclusion

In this lesson, we explored some of the capabilities of Deep Lake, a multi-modal AI data system that merges the functionalities of Data Lakes and Vector Databases.
We've learned how Deep Lake can efficiently stream data from remote storage to GPUs during model training, making it an ideal tool for training and fine-tuning Large Language Models. We've also covered the creation of a Deep Lake dataset, adding data to it, and retrieving data using both Deep Lake's data loaders and PyTorch's data loaders.
This will be useful as we continue exploring training and fine-tuning Large Language Models.

# How to train NanoGPT with Deep Lake streamable dataloader

https://learn.activeloop.ai/courses/take/llms/multimedia/48954371-how-to-train-nanogpt-with-deep-lake-streamable-dataloader

# Datasets for Training LLMs

## Introduction

In this lesson, we talk about the datasets that fuel LLMs pretraining. We'll explore popular datasets like Falcon RefinedWeb, The Pile, Red Pajama Data, and Stack Overflow Posts, understanding their composition, sources, and usage. We'll also discuss the emerging trend of prioritizing data quality over quantity in pretraining LLMs.

## Popular Datasets for Training LLMs

In recent times, a variety of open-source datasets have been employed for pre-training Large Language Models.
Some of the notable datasets include **"Falcon RefinedWeb," "The Pile," "Red Pajama Data,"** and **"Stack Overflow Posts,"** among others**.** Assembling such datasets typically involves collecting and cleaning vast volumes of text data.

**Falcon RefinedWeb**

The Falcon RefinedWeb dataset is a large-scale English web dataset developed by TII and released under the ODC-By 1.0 license. It was created through rigorous filtering and extensive deduplication of CommonCrawl, resulting in a dataset that has shown comparable or superior performance to models trained on curated datasets, relying solely on web data.
The dataset is designed to be "multimodal-friendly," as it includes links and alt texts for images in the processed samples. Depending on the tokenizer used, the public extract of this dataset ranges from 500-650GT and requires about 2.8TB of local storage when unpacked.
Falcon RefinedWeb has been primarily used for training Falcon LLM models, including the Falcon-7B/40B and Falcon-RW-1B/7B models. The dataset is primarily in English, and each data instance corresponds to a unique web page that has been crawled, processed, and deduplicated. It contains around 1 billion instances.

The dataset was constructed using the Macrodata Refinement Pipeline, which includes content extraction, filtering heuristics, and deduplication. The design philosophy of RefinedWeb prioritizes scale, strict deduplication, and neutral filtering. The dataset was iteratively refined by measuring the zero-shot performance of models trained on development versions of the dataset and manually auditing samples to identify potential filtering improvements.

**The Pile**

The Pile is a comprehensive, open-source dataset of English text designed specifically for training LLMs. Developed by EleutherAI in 2020, it's a massive 886.03GB dataset comprising 22 smaller datasets, 14 of which are new. Prior to the Pile's creation, most LLMs were trained using data from the Common Crawl. However, the Pile offers a more diverse range of data, enabling LLMs to handle a broader array of situations post-training.

The Pile is a carefully curated collection of data handpicked by EleutherAI's researchers to include information they deemed necessary for language models to learn. The Pile covers a wide range of topics and writing styles, including academic writing, a style that models trained on other datasets often struggle with.

All data used in the Pile was sourced from publicly accessible resources and filtered to remove duplicates and non-textual elements like HTML formatting and links. However, individual documents within the sub-datasets were not filtered to remove non-English, biased, or profane text, nor was consent considered in the data collection process.

Originally developed for EleutherAI's GPT-Neo models, the Pile has since been used to train a variety of other models.

**RedPajama Dataset**

The RedPajama dataset is a comprehensive, open-source dataset that emulates the LLaMa dataset. It comprises 2084 jsonl files, which can be accessed via HuggingFace or directly downloaded. The dataset is primarily in English but includes multiple languages in its Wikipedia section.

The dataset is structured into text and metadata, including the URL, timestamp, source, language, and more. It also specifies the subset of the RedPajama dataset it belongs to, such as Commoncrawl, C4, GitHub, Books, ArXiv, Wikipedia, or StackExchange.

The dataset is sourced from various platforms:

- Commoncrawl data is processed through the official cc_net pipeline, deduplicated, and filtered for quality.
- C4 data is obtained from HuggingFace and formatted to suit the dataset's structure.
- GitHub data is sourced from Google BigQuery, deduplicated, and filtered for quality, with only MIT, BSD, or Apache-licensed projects included.
- The Wikipedia data is sourced from HuggingFace and is based on a 2023 dump, with hyperlinks, comments, and other formatting removed.
- Gutenberg and Books3 data are also downloaded from HuggingFace, with near duplicates removed using simhash.
- ArXiv data is sourced from Amazon S3, with only latex source files included and preambles, comments, macros, and bibliographies removed.
- Lastly, StackExchange data is sourced from the Internet Archive, with only the posts from the 28 largest sites included, HTML tags removed, and posts grouped into question-answer pairs.

The RedPajama dataset encompasses 1.2 trillion tokens, making it a substantial resource for various language model training and research purposes.

**Stack Overflow Posts**

If you're interested more in a specific domain like coding, there are massive datasets available for that, too.

The Stack Overflow Posts dataset comprises approximately 60 million posts submitted to StackOverflow prior to June 14, 2023. The dataset, sourced from the Internet Archive StackExchange Data Dump, is approximately 35GB in size and contains around 65 billion text

characters. Each record in the dataset represents a post type and includes fields such as Id, PostTypeId, Body, and ContentLicense, among others.

## Data Quality vs. Data Quantity in Pretraining

As we just saw, many of the most used pretraining datasets today are cleaned and more complete versions of other past datasets. There's recently been a shift in focus from increasing dataset sizes to "increasing dataset size AND dataset quality."

The paper "Textbooks Are All You Need," published in June 2023, shows this trend. It introduces Phi-1, an LLM designed for code. Phi-1 is a Transformer-based model with 1.3 billion parameters, trained over a period of four days on eight A100s. Despite its relatively smaller scale, it exhibits remarkable accuracy on benchmarks like HumanEval and MBPP. How? It's been trained on high-quality data (i.e., textbook-quality data; that's why the paper name is "Textbooks are all you need").

The training data for Phi-1 comprises 6 billion tokens of "textbook quality" data from the web and 1 billion tokens from synthetically generated textbooks using GPT-3.5. Although Phi-1's specialization in Python coding and lack of domain-specific knowledge somewhat limit its versatility, these limitations are not inherent and can be addressed to enhance its capabilities.

Despite its smaller size, the model's success in coding benchmarks demonstrates the significant impact of high-quality and coherent data on the proficiency of language models, thereby shifting the focus from quantity to quality of data.

## Creating Your Own Dataset

Creating your own dataset would involve a whole lesson for it and we won't cover it in this course in detail. However, if you're interested in doing so, you can study the creation process of the datasets listed in the above sections, as it's often a publicly disclosed process.

## Conclusion

This lesson provides a comprehensive overview of the datasets that fuel the pretraining of LLMs. We delved into popular datasets such as Falcon RefinedWeb, The Pile, Red Pajama Data, and Stack Overflow Posts, understanding their composition, sources, and usage. Often derived from larger, less refined datasets, these datasets have been meticulously cleaned and curated to provide high-quality data for training LLMs.

We also discussed the emerging trend of prioritizing data quality over quantity in pretraining LLMs, as exemplified by the Phi-1 model. Despite its smaller scale, Phi-1's high performance on benchmarks underscores the significant impact of high-quality and coherent data on the proficiency of language models. This shift in focus from data quantity to quality is an exciting development in the field of LLMs, highlighting the importance of dataset refinement in achieving superior model performance.

# Train an LLM in the Cloud

## Introduction

In this lesson, we'll guide you through the step-by-step process of training a large language model from the ground up. Our primary focus will be on conducting the pre-training process in the cloud. Nevertheless, it's worth noting that all the concepts covered here can be transferable if you want to train a model locally and have enough resources on your local machine (only for small language models).

When embarking on model training, three key components must be taken into account. The process begins with selecting an **appropriate dataset** that aligns with your specific use case. Next, **configure the architecture** of the model, making adjustments based on the resources at your disposal. Finally, **execute the training loop**, bringing everything together to train the model effectively.
We integrate well-known libraries like Deep Lake Datasets and Transformers into our implementation to build a smooth pipeline. The initial step to initiate the process involves selecting the dataset.

## GPU Cloud - Lambda

In this lesson, we'll leverage Lambda, the GPU cloud designed by ML engineers for training LLMs & Generative AI. We can create an account on it, link a billing account, and then rent one instance of the following GPU servers with associated costs. Please follow the instructions in the course logistics section to open a Lambda account. The cost of your instance is based on its duration, not just the time spent training your model; so remember to turn your instance off. For this lesson, we rented an 8x NVIDIA A100 instance comprising 40GB of memory for $8.80/h. If you're using the Lambda-provided cloud credit for the course, be aware that you still need to register a credit card. The credit will cover costs up to $75, but you must have a card on file. If you spend more money than allocated by the credit (more than $75), you will have to cover those costs yourself.
You can find the code of this lesson in this Notebook.

⚠ Beware of costs when you borrow cloud GPUs. The total cost will depend on the machine type and the up time of the instance. Always remember to monitor your costs in the billing section of Lambda Labs and to spin off your instances when you don't use them.

💡If you just want to replicate the code in the lesson spending very few money, you can just run the training in your instance and stop it after a few iterations.

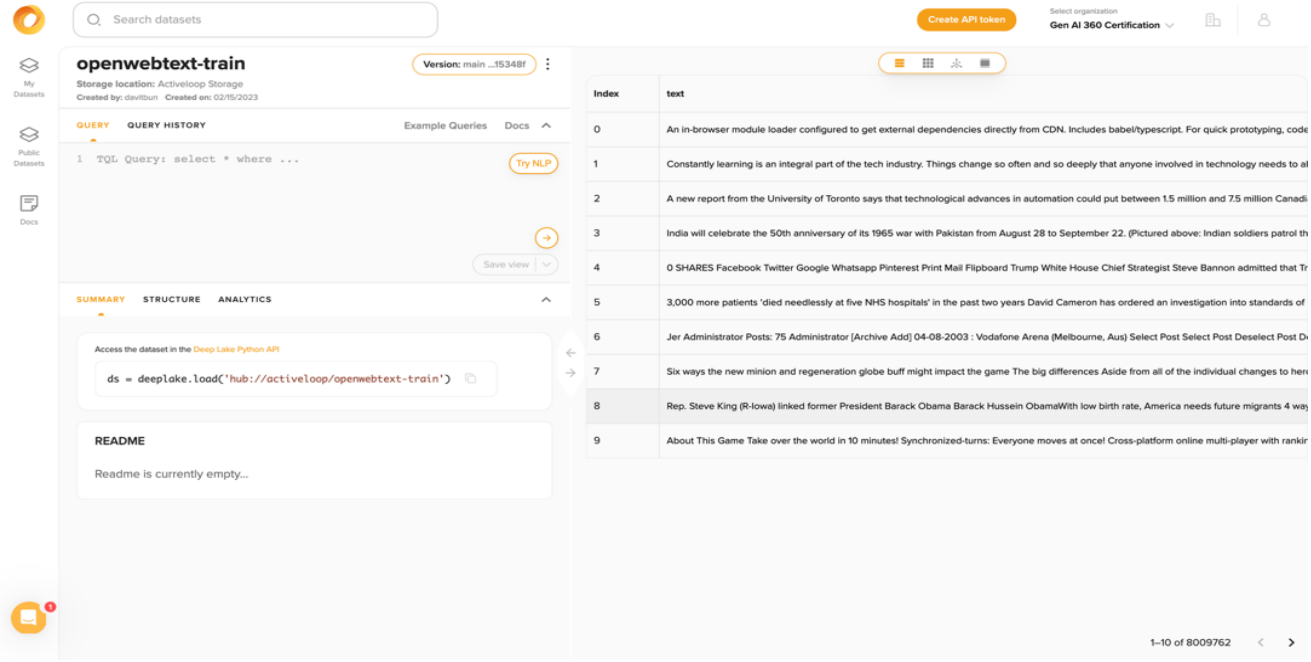
## Training Monitoring - Weights and Biases

As we're going to spend a lot of money in training our LLM and ensure that everything is progressing smoothly, we'll log the training metrics to Weights and Biases, allowing us to see the metrics in real time in a suitable dashboard.

## Load the Dataset

During the pre-training process, we utilize the Activeloop datasets to **stream the samples seamlessly, batch by batch**. This approach proves beneficial for **resource management** as loading the entire dataset directly into memory is unnecessary. Consequently, it greatly helps in optimizing resource usage. You can quickly load the dataset, and it automatically handles the streaming process without requiring any special configurations.
You can load the datasets in just one line of code and visualize their content for analysis. The library seamlessly integrates with PyTorch and TensorFlow, which are considered two of the most powerful frameworks for implementing AI applications. You can head out to datasets.activeloop.ai to see the complete list of available datasets. Porting your datasets to the hub is also achievable with minimal effort.
Let's start by loading the `openwebtext` dataset, a collection of Reddit posts with at least three upvotes. This dataset is well-suited for acquiring broad knowledge to build a foundational model for general purposes. The Deep Lake web UI simplifies dataset exploration through its table view and empowers you to query the data using TQL (Tensor Query Language). You can notice that

it's possible to quickly inspect dataset details, even when dealing with a sizable dataset containing 8 million rows. This comes thanks to Deep Lake's format, that enables rapid data streaming straight to your browser.




**Deep Lake Visualization Engine table view.**

```
import deeplake

ds = deeplake.load('hub://activeloop/openwebtext-train')

ds_val = deeplake.load('hub://activeloop/openwebtext-val')


print(ds)

print(ds[0].text.text())
```

The sample code.

```
Dataset(path='hub://activeloop/openwebtext-train', read_only=True, tensors=['text',
'tokens'])

"An in-browser module loader configured to get external dependencies directly from
CDN. Includes babel/typescript. For quick prototyping, code sharing,
teaching/learning - a super simple web dev environment without
node/webpack/etc.\n\nAll front-end libraries\n\nAngular, React, Vue, Bootstrap,
Handlebars, and jQuery are included. Plus all packages from cdnjs.com and all of NPM
(via unpkg.com). Most front-end libraries should work out of the box - just use
import / require() . If a popular library does not load, tell us and we'll try to
solve it with some library-specific config.\n\nWrite modern javascript (or
typescript)\n\nUse latest language features or JSX and the code will be transpiled
in-browser via babel or typescript (if required). To make it fast the transpiler will
start in a worker thread and only process the modified code. Unless you change many
files at once or open the project for the first time, the transpiling should be
barely noticeable as it runs in parallel with loading a..."
```

The output.

The provided code will instantiate a dataset object capable of retrieving the data points for both training and validation sets. Afterward, we can print the variable to examine the dataset's characteristics. It consists of two tensors: `text` containing the textual input and `tokens` representing the tokenized version of the content (which we won't be utilizing). We can also index through the dataset and access each column by using `.text` and convert the row to textual format by calling `.text()` method.
The next step involves crafting a PyTorch Dataset class that leverages the loader object and ensures compatibility with the framework. The Dataset class handles both dataset formatting and any desired preprocessing steps to be applied. In this instance, our objective is to tokenize the samples. We will load the GPT-2 tokenizer model from the Transformers library to achieve this.
For this specific model, we need to set a padding token (which may not be required for other models), and for this specific purpose, we have chosen to utilize the end of sentence `eos_token` to set the loaded tokenizer's `pad_token` method.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")

tokenizer.pad_token = tokenizer.eos_token
```

The sample code.

Next, we create dataloaders from the Deep Lake datasets. In doing so, we also specify a `transform` that tokenizes the texts of the dataset on the fly.

```
# define transform to tokenize texts
def get_tokens_transform(tokenizer):
    def tokens_transform(sample_in):
        tokenized_text = tokenizer(
            sample_in["text"],
            truncation=True,
            max_length=512,
            padding='max_length',
            return_tensors="pt")
        tokenized_text = tokenized_text["input_ids"][0]
        return {
            "input_ids": tokenized_text,
            "labels": tokenized_text
        }
    return tokens_transform
```

```
# create data loaders
ds_train_loader = ds.dataloader()\
    .batch(32)\
    .transform(get_tokens_transform(tokenizer))\
    .pytorch()
ds_eval_train_loader = ds_val.dataloader()\
    .batch(32)\
    .transform(get_tokens_transform(tokenizer))\
    .pytorch()
```

The sample code.

Please note that we have formatted the dataset so that each sample is comprised of two components: `input_ids` and `labels`.
`input_ids` are the tokens the model will use as inputs, while `labels` are the tokens the model will try to predict.
Currently, both keys contain the same tokenized text. However, the trainer object from the Transformers library will automatically shift the labels by one token, preparing them for training.

# Initialize the Model

As the scope of this course does not include building the architecture from scratch, we won't be implementing it. We have already covered the details of the Transformer architecture in a previous lesson and provided additional resources for those who are interested in a more in-depth implementation.
To accelerate the process, we will leverage an existing publicly available implementation of the transformer architecture. This approach allows us to scale the model quickly using available hyperparameters, including the number of layers, embedding dimension, and attention heads. Additionally, we will capitalize on the success of established architectures while maintaining the flexibility to modify the model size to accommodate our available resources.
We opted to utilize the GPT-2 pre-trained model. Nonetheless, there is an option to utilize any other available model from the Huggingface hub; the approach presented here can be easily adapted to work with various architectures.
Initially, we examine the default hyperparameters by loading the configuration file and reviewing the choices made in the architecture design.

```
from transformers import AutoConfig
config = AutoConfig.from_pretrained("gpt2")
print(config)
```

The sample code.

```
GPT2Config {
  "_name_or_path": "gpt2",
```

```
"activation_function": "gelu_new",
"architectures": [
  "GPT2LMHeadModel"
],
"attn_pdrop": 0.1,
"bos_token_id": 50256,
"embd_pdrop": 0.1,
"eos_token_id": 50256,
"initializer_range": 0.02,
"layer_norm_epsilon": 1e-05,
"model_type": "gpt2",
"n_ctx": 1024,
"n_embd": 768,
"n_head": 12,
"n_inner": null,
"n_layer": 12,
"n_positions": 1024,
"reorder_and_upcast_attn": false,
"resid_pdrop": 0.1,
"scale_attn_by_inverse_layer_idx": false,
"scale_attn_weights": true,
"summary_activation": null,
"summary_first_dropout": 0.1,
"summary_proj_to_labels": true,
"summary_type": "cls_index",
"summary_use_proj": true,
"task_specific_params": {
  "text-generation": {
    "do_sample": true,
    "max_length": 50
  }
},
"transformers_version": "4.30.2",
"use_cache": true,
```

```
  "vocab_size": 50257
}
```

The output.

It is apparent that we have the ability to exert significant control over almost every aspect of the network by manipulating the configuration settings. Specifically, we focus on the following parameters: `n_layer`, which indicates the number of stacking decoder components and defines the embedding layer's hidden dimension; `n_positions` and `n_ctx`, to represent the maximum number of input tokens; and `n_head` to change the number of attention heads in each attention component. You can read the documentation to gain a more comprehensive understanding of the remaining parameters.
We can start by initializing the model using the default configuration and then count the number of parameters it contains, which will serve as a baseline. To achieve this, we utilize the `GPT2LMHeadModel` class, which takes the `config` variable as input and then proceeds to loop through the parameters, summing them up accordingly.

```
from transformers import GPT2LMHeadModel

model = GPT2LMHeadModel(config)

model_size = sum(t.numel() for t in model.parameters())

print(f"GPT-2 size: {model_size/1e6:.1f}M parameters")
```

The sample code.

```
GPT-2 size: 124.4M parameters
```

The output.

As shown, the GPT-2 model is relatively small (124M) when compared to the current state-of-the-art large language models. We're going to pre-train a 124-million-parameter model, which we refer to as `GPT2-scratch-openwebtext`. We chose this size so that a part of its training can be easily replicated by any reader within a reasonable price (~$100).
If you wanted to train a larger model, you could modify the architecture to scale it up slightly. As we previously described the selected parameters, we can create a network with 32 layers and an embedding size of 1600. It is worth noting that if not specified, the hidden dimensionality of the linear layers will be `4 × n_embd`.

```
config.n_layer = 32

config.n_embd = 1600

config.n_positions = 512

config.n_ctx = 512

config.n_head = 32
```

The sample code.

Now, we proceed to load the model with the updated hyperparameters.

```
model_1b = GPT2LMHeadModel(config)
model_size = sum(t.numel() for t in model_1b.parameters())
print(f"GPT2-1B size: {model_size/1e6:.1f}M parameters")
```

The sample code.

```
GPT2-1B size: 1065.8M parameters
```

The sample output.

The modifications led to a model with 1 billion parameters. It is possible to scale the network further to be more in line with the newest state-of-the-art models, which often have more than 80 layers.
However, let's continue with this lesson's 124M parameters model.

# Training Loop

The final step in the process involves initializing the training loop. We utilize the Transformers library's `Trainer` class, which takes the necessary parameters for training the model. However, before proceeding, we need to create a `TrainingArguments` object that defines all the essential arguments.

```
from transformers import Trainer, TrainingArguments
args = TrainingArguments(
    output_dir="GPT2-scratch-openwebtext",
    evaluation_strategy="steps",
    save_strategy="steps",
    eval_steps=500,
    save_steps=500,
    num_train_epochs=2,
    logging_steps=1,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=1,
    gradient_accumulation_steps=1,
    weight_decay=0.1,
    warmup_steps=100,
    lr_scheduler_type="cosine",
    learning_rate=5e-4,
    bf16=True,
    ddp_find_unused_parameters=False,
    run_name="GPT2-scratch-openwebtext",
```

```
    report_to="wandb")
```

The sample code.

Note that we set the `per_device_train_batch_size` and the `per_device_eval_batch_size` variables to `1` as the batch size is already specified by the dataloader we created earlier.
There are over 90 parameters available for adjustment. Find a comprehensive list with explanations in the documentation. Please note that if there is an "out of memory" error while attempting to train, a smaller `batch_size` can be used. Additionally, the `bf16` flag, which trains the model using lower precision floating numbers, is only available on high-end GPU devices. If unavailable, it can be substituted with the argument `fp16=True`.

Notice also that we set the parameter `report_to` to `wandb`; that is, we are sending the training metrics to Weights and Biases so that we can see a real-time report of how the training is going.
Next, we define the `TrainerWithDataLoaders` class, a subclass of `Trainer` where we override the `get_train_dataloader` and `get_eval_dataloader` methods to return our previously defined data loaders.

```
from transformers import Trainer

class TrainerWithDataLoaders(Trainer):
    def __init__(self, *args, train_dataloader=None, eval_dataloader=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.train_dataloader = train_dataloader
        self.eval_dataloader = eval_dataloader
    def get_train_dataloader(self):
        return self.train_dataloader
    def get_eval_dataloader(self, dummy):
        return self.eval_dataloader
```

The process initiates with a call to the `.train()` method.

```
trainer = TrainerWithDataLoaders(
    model=model,
    args=args,
    train_dataloader=ds_train_loader,
    eval_dataloader=ds_eval_train_loader )
trainer.train()
```

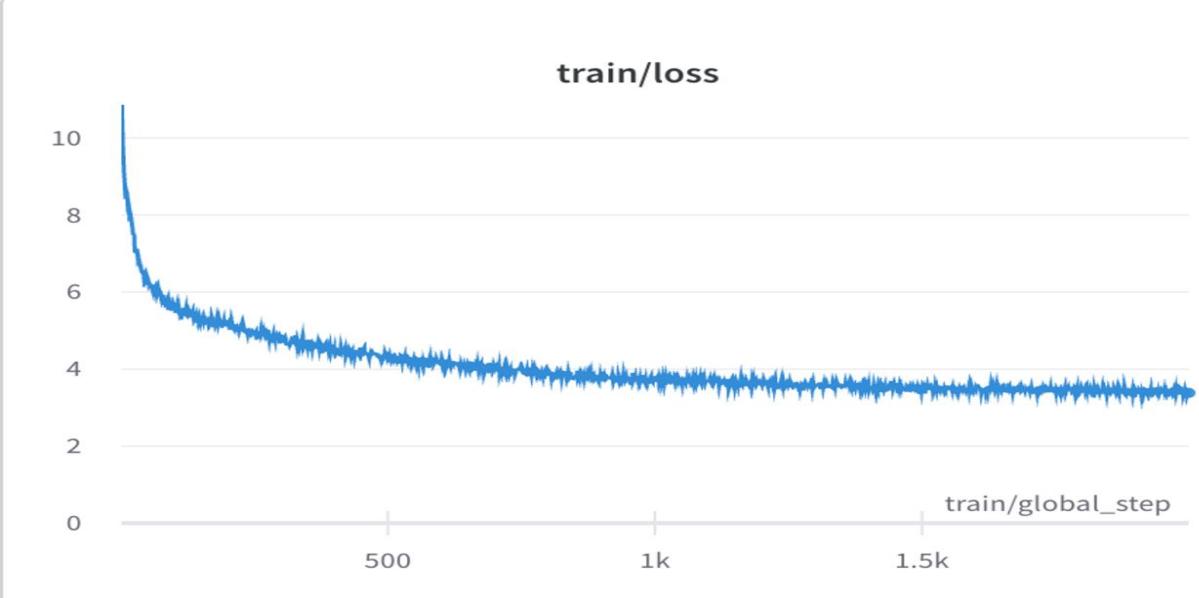
The sample code.

The `Trainer` object will handle model evaluation during training, as specified in the `eval_steps` argument, and save checkpoints based on the previously defined in `save_steps`.
Here's the final trained model after about 45 hours of training on 8x NVIDIA A100 on Lambda Labs.

GPT2-scratch-openwebtext.zip1333888.0KB

As the hourly cost of 8x NVIDIA A100 on Lambda Labs is $8.80, the total cost is $ 400. You can stop your pretraining earlier if you want to spend less money on that.

Here's the training report on Weights and Biases. The following report shows that the training loss decreased relatively smoothly as iterations passed.




# Inference

Once the pre-training process is complete, we proceed with the inference stage to observe our model in action and evaluate its capabilities. As specified, the `Trainer` will store the intermediate checkpoints in a designated directory called `./GPT2-scratch-openwebtext`. The most efficient approach to utilize the model involves leveraging the Transformers pipeline functionality, which automatically loads both the model and tokenizer, making them ready for text generation.
Below is the code snippet that establishes a pipeline object utilizing the pre-trained model alongside the tokenizer we defined in the preceding section. This pipeline enables text generation.

```
from transformers import pipeline

pipe = pipeline("text-generation",
                model="./GPT2-scratch-openwebtext",
                tokenizer=tokenizer,
                device="cuda:0")
```

The sample code.

The pipeline object leverages the powerful Transformers `.generate()` method internally, offering exceptional flexibility in managing the text generation process. (documentation) We can use methods like `min_length` to define a minimum number of tokens to be generated, `max_length` to limit the newly generated tokens, `temperature` to control the generation process between

randomness and most likely, and lastly, `do_sample` to modify the completion process, switching between a greedy approach that always selects the most probable token and other sampling methods, such as beam search or diverse search. We only set the `num_return_sequences` to limit the number of generated sequences.

```
txt = "The house prices dropped down"

completion = pipe(txt, num_return_sequences=1)

print(completion)
```

The sample code.

```
[{'generated_text': 'The house prices dropped down to 3.02% last year. While it was
still in development, the housing market was still down. The recession hit on 3 years
between 1998 and 2011. In fact, it slowed the amount of housing from 2013 to 2013'}]
```

The output.

The code will attempt to generate a completion for the given input sequence using the knowledge it has acquired from the training dataset. It aims to finish the following sequence: `The house prices dropped down` while being relevant and contextually appropriate. Even with a brief training period, the model exhibits a good grasp of the language, generating grammatically correct and contextually coherent sentences.

# Conclusion

Throughout this lesson, we gained an understanding of the fundamental steps required to train your own language model. The steps involve loading the relevant training data, defining the architecture, scaling it up as per your requirements, and, finally, commencing the training process. As previously discussed, there is no need to train a language model from scratch in many cases. In the upcoming module, we will cover the fine-tuning process in greater detail, enabling you to harness the capabilities of existing powerful models for specific use cases.

**Resources**

- Notebook.
  *(Inference results at the end)*
- Weight and Biases Report.
- Requirements
  requirements-train.txt

  https://assets.super.so/0d43acc6-3340-4118-9744-d03a54c41788/files/d5f49c75-d3cb-4a82-b73f-28945aa2e5e7.txt

# Going at Scale with LLM Training

## Introduction

In this lesson, we will share some tips for training LLMs at scale, focusing on the Zero Redundancy Optimizer (ZeRO) and its implementation in DeepSpeed. We explore how **ZeRO optimizes memory and computational resources**, its various stages of operation, and the benefits of **DeepSpeed**. We also touch on the Hugging Face Accelerate library. Finally, we will discuss the importance of maintaining a logbook of training runs to manage potential challenges and instabilities during the training process.

## The Zero Redundancy Optimizer (ZeRO)

Training Large Language Models can be a formidable task due to the immense computational and memory requirements. However, the introduction of the Zero Redundancy Optimizer (ZeRO), implemented in DeepSpeed, has made it possible to train these models with lower hardware requirements.

ZeRO is a parallelized optimizer that drastically reduces the resources required for model and data parallelism while significantly increasing the number of parameters that can be trained.

ZeRO is designed to make the most of data parallelism's computational and memory resources, reducing the memory and compute requirements of each device (GPU) used for model training. It achieves this by distributing the various model training states (weights, gradients, and optimizer states) across the available devices (GPUs and CPUs) in the distributed training hardware.

As long as the aggregated device memory is large enough to share the model states, ZeRO-powered data parallelism can accommodate models of any size.

## The Stages of ZeRO

ZeRO operates in three main optimization stages, where the enhancements in earlier stages are available in the later stages. The stages are partitioning optimizer states, gradients, and parameters.

- **Stage 1 - Optimizer State Partitioning**: Shards optimizer states across data parallel workers/GPUs. This results in a 4x memory reduction, with the same communication volume as data parallelism. For example, this stage can be used to train a 1.5 billion parameter GPT-2 model on eight V100 GPUs.
- **Stage 2 - Gradient Partitioning**: Shards optimizer states and gradients across data parallel workers/GPUs. This leads to an 8x memory reduction, with the same communication volume as data parallelism. For example, this stage can be used to train a 10 billion parameter GPT-2 model on 32 V100 GPUs.
- **Stage 3 - Parameter Partitioning**: Shards optimizer states, gradients, and model parameters across data parallel workers/GPUs. This results in a linear memory reduction with the data parallelism degree. ZeRO can train a trillion-parameter model on about 512 NVIDIA GPUs with all three stages.
- **Stage 3 Extra - Offloading to CPU and NVMe memory**: In addition to these stages, ZeRO-3 includes the infinity offload engine to form ZeRO-Infinity, which can offload to both CPU and NVMe memory for significant memory savings. This technique allows you to train even larger models that wouldn't fit into GPU memory. It offloads optimizer states, gradients, and parameters to the CPU, allowing you to train models with billions of parameters on a single GPU.

## DeepSpeed

DeepSpeed is a high-performance library for accelerating distributed deep learning training. It incorporates ZeRO and other state-of-the-art training techniques, such as distributed training, mixed precision, and checkpointing, through lightweight APIs compatible with PyTorch.

DeepSpeed excels in four key areas:

1. **Scale**: DeepSpeed's ZeRO stage one provides system support to run models up to 100 billion parameters, which is 10 times larger than the current state-of-the-art large models.
2. **Speed**: DeepSpeed combines ZeRO-powered data parallelism with model parallelism to achieve up to five times higher throughput over the state-of-the-art across various hardware.
3. **Cost**: The improved throughput translates to significantly reduced training costs. For instance, to train a model with 20 billion parameters, DeepSpeed requires three times fewer resources.
4. **Usability**: Only a few lines of code changes are needed to enable a PyTorch model to use DeepSpeed and ZeRO. DeepSpeed does not require a code redesign or model refactoring, and it does not put limitations on model dimensions, batch size, or any other training parameters.

## Accelerate and DeepSpeed ZeRO

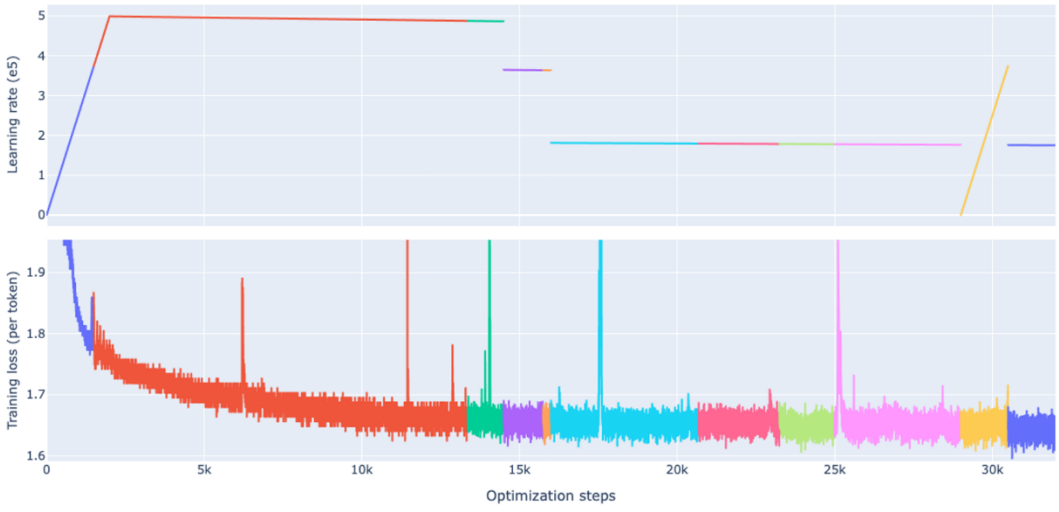
The Hugging Face Accelerate library allows you to leverage DeepSpeed's ZeRO features by making very few code changes. By using Accelerate and DeepSpeed ZeRO, we can significantly increase the maximum batch size that our hardware can handle without running into OOM errors.

## Logbook of Training Runs

Despite these libraries, there are still unexpected obstacles in the training runs. This is because there may be instabilities during training that are hard to recover from, such as spikes in the loss function.

For example, here's a logbook of the training of reproduction of Flamingo (by Google Deepmind), an 80B parameters vision and language model, done by Hugging Face. In the following image, the second chart shows the loss function of the final model as the training progresses. Some of these spikes rapidly recovered to the original loss level, and some others diverged and never recovered.

To stabilize and continue the training, the authors usually applied a rollback, i.e., a re-start from a checkpoint a few hundred steps prior to the spike/divergence, sometimes with a decrease in the learning rate (shown in the first chart of the image).




Other times, it may be possible for the model to be stuck in a local optimum, thus requiring other rollbacks. Sometimes, memory errors may require a manual inspection. Have a look at this 114-page logbook made by Meta during the training of the OPT 175B model.

## Conclusion

This lesson covered a few tips for training Large Language Models at scale, focusing on the Zero Redundancy Optimizer (ZeRO) and its implementation in DeepSpeed. We won't cover them in more detail in the course, so if you want to deep dive into them you can read the resources linked in this page.

We learned how ZeRO optimizes memory and computational resources across different stages, enabling the training of models with billions of parameters. We also explored DeepSpeed, a high-performance library that incorporates ZeRO and other state-of-the-art training techniques, providing scalability, speed, cost-effectiveness, and usability.

We touched on the Hugging Face Accelerate library, which simplifies the application of DeepSpeed's ZeRO features.

Lastly, we highlighted the importance of maintaining a logbook of training runs to manage potential challenges and instabilities during the training process.

## Master LLMs: Top Strategies to Evaluate LLM Performance

1. Perplexity Evaluation metric
2. Benchmarks for
    a. Coding Benchmarks
    b. Mitigating hallucination
    c. Reasoning and common sense

https://youtu.be/iWlTCBUoru8

# Benchmarking Your Own LLM

## Introduction

In a previous lesson, we touched upon several methodologies for assessing the effectiveness of different language models. Even with various available methodologies for evaluating a large language model, the process continues to pose significant challenges.

The primary challenge stems from the inherent subjectivity in determining what constitutes a good answer. As an example, let's consider a generative task such as summarization. Quantifying a specific summary as the definitive correct answer in terms of context is difficult and hard to define. This challenge is prevalent across every generative task, making it a pervasive source of difficulty. In this lesson, we'll see how benchmarks are a candidate solution to this problem.

## Benchmarks Over Several Tasks

The solution suggests curating a set of benchmarks to evaluate the model's performance across various tasks. The benchmarks encompass assessments for world knowledge, following complex instructions, arithmetic, programming, and more.

Several leaderboards exist to monitor the progress of LLMs, including the "*Open LLM Leaderboard*" and the "*InstructEval Leaderboard*." In some cases, these benchmarks share similar metrics.

Here are some examples of tasks tested in these benchmarks:

- **AI2 Reasoning Challenge** (ARC): The dataset is exclusively comprised of natural, grade-school science questions designed for human tests.

- **HumanEval**: It is used to measure program synthesis from docstrings. It includes 164 original programming problems assessing language comprehension, algorithms, and math, some resembling software interview questions.
- **HellaSwag**: A challenge to measure commonsense inference, and shows it remains difficult for state-of-the-art models. While humans achieve >95% accuracy on trivial questions, the models struggle with <48% accuracy.
- **Measuring Massive Multitask Language Understanding (**MMLU): An evaluation metric for text models' multitask accuracy, covering 57 tasks, including math, US history, computer science, law, etc. High accuracy requires extensive knowledge of the world and problem-solving ability.
- **TruthfulQA**: A truthfulness benchmark designed to assess the accuracy of language models in generating answers to questions. It consists of 817 questions across 38 categories, encompassing topics such as health, law, finance, and politics.

## Language Model Evaluation Harness

EleutherAI has released a benchmarking script capable of evaluating any language model, whether it is proprietary and accessible via API or an open-source model. It is customized for generative tasks, utilizing publicly available prompts to enable fair comparisons among the papers. As of writing this lesson, the script supports over 200 diverse evaluation metrics tailored for a wide array of tasks.

Reproducibility stands as a vital aspect of any evaluation process! Especially in generative models, there are numerous parameters available during inference, each offering varying levels of randomness. They employ a task versioning feature that guarantees comparability of results even after the tasks undergo updates. Let's look at the benchmarking process of using the library to evaluate an open-source model.

To begin, you need to fetch the script from GitHub. The following code will pin the script version to `0.3.0`. If you are working with the script in a Google Colab/Notebook environment, utilizing the sample code provided at the end of this lesson is advisable.

```
git clone https://github.com/EleutherAI/lm-evaluation-harness

cd llama.cpp && git checkout e2eb966 # Pin to version 0.3.0
```

The sample script.

Now, we can access the library files and list all the available tasks it supports. By printing the internal variable named `ALL_TASKS`, you can obtain a list of all the metrics that have been implemented.

```
from lm_eval import tasks

print(tasks.ALL_TASKS)
```

The sample code.

```
['Ceval-valid-accountant', 'Ceval-valid-advanced_mathematics', 'Ceval-valid-
art_studies', 'Ceval-valid-basic_medicine', 'Ceval-valid-business_administration',
'Ceval-valid-chinese_language_and_literature', 'Ceval-valid-civil_servant', 'Ceval-
valid-clinical_medicine', 'Ceval-valid-college_chemistry', 'Ceval-valid-
college_economics', 'Ceval-valid-college_physics', 'Ceval-valid-college_programming',
'Ceval-valid-computer_architecture', 'Ceval-valid-computer_network', 'Ceval-valid-
discrete_mathematics', 'Ceval-valid-education_science', 'Ceval-valid-
electrical_engineer', 'Ceval-valid-environmental_impact_assessment_engineer', 'Ceval-
valid-fire_engineer', 'Ceval-valid-high_school_biology', 'Ceval-valid-
high_school_chemistry', 'Ceval-valid-high_school_chinese', 'Ceval-valid-
```

```
high_school_geography', 'Ceval-valid-high_school_history', 'Ceval-valid-
high_school_mathematics', 'Ceval-valid-high_school_physics', 'Ceval-valid-
high_school_politics', 'Ceval-valid-ideological_and_moral_cultivation', 'Ceval-valid-
law', 'Ceval-valid-legal_professional', 'Ceval-valid-logic', 'Ceval-valid-
mao_zedong_thought', 'Ceval-valid-marxism', 'Ceval-valid-metrology_engineer', 'Ceval-
valid-middle_school_biology', 'Ceval-valid-middle_school_chemistry', 'Ceval-valid-
middle_school_geography', 'Ceval-valid-middle_school_history', 'Ceval-valid-
middle_school_mathematics', 'Ceval-valid-middle_school_physics', 'Ceval-valid-
middle_school_politics', 'Ceval-valid-modern_chinese_history', 'Ceval-valid-
operating_system', 'Ceval-valid-physician', 'Ceval-valid-plant_protection', 'Ceval-
valid-probability_and_statistics', 'Ceval-valid-professional_tour_guide', 'Ceval-
valid-sports_science', 'Ceval-valid-tax_accountant', 'Ceval-valid-
teacher_qualification', 'Ceval-valid-urban_and_rural_planner', 'Ceval-valid-
veterinary_medicine', 'anagrams1', 'anagrams2', 'anli_r1', 'anli_r2', 'anli_r3',
'arc_challenge', 'arc_easy', 'arithmetic_1dc', 'arithmetic_2da', 'arithmetic_2dm',
'arithmetic_2ds', 'arithmetic_3da', 'arithmetic_3ds', 'arithmetic_4da',
'arithmetic_4ds', 'arithmetic_5da', 'arithmetic_5ds', 'babi', 'bigbench___init__',
'bigbench_causal_judgement', 'bigbench_date_understanding',
'bigbench_disambiguation_qa', 'bigbench_dyck_languages',
'bigbench_formal_fallacies_syllogisms_negation', 'bigbench_geometric_shapes',
'bigbench_hyperbaton', 'bigbench_logical_deduction_five_objects',
'bigbench_logical_deduction_seven_objects',
'bigbench_logical_deduction_three_objects', 'bigbench_movie_recommendation',
'bigbench_navigate', 'bigbench_reasoning_about_colored_objects',
'bigbench_ruin_names', 'bigbench_salient_translation_error_detection',
'bigbench_snarks', 'bigbench_sports_understanding', 'bigbench_temporal_sequences',
'bigbench_tracking_shuffled_objects_five_objects',
'bigbench_tracking_shuffled_objects_seven_objects',
'bigbench_tracking_shuffled_objects_three_objects', 'blimp_adjunct_island',
'blimp_anaphor_gender_agreement', 'blimp_anaphor_number_agreement',
'blimp_animate_subject_passive', 'blimp_animate_subject_trans', 'blimp_causative',
'blimp_complex_NP_island',
'blimp_coordinate_structure_constraint_complex_left_branch',
'blimp_coordinate_structure_constraint_object_extraction',
'blimp_determiner_noun_agreement_1', 'blimp_determiner_noun_agreement_2',
'blimp_determiner_noun_agreement_irregular_1',
'blimp_determiner_noun_agreement_irregular_2',
'blimp_determiner_noun_agreement_with_adj_2',
'blimp_determiner_noun_agreement_with_adj_irregular_1',
'blimp_determiner_noun_agreement_with_adj_irregular_2',
'blimp_determiner_noun_agreement_with_adjective_1',
'blimp_distractor_agreement_relational_noun',
'blimp_distractor_agreement_relative_clause', 'blimp_drop_argument',
'blimp_ellipsis_n_bar_1', 'blimp_ellipsis_n_bar_2',
'blimp_existential_there_object_raising', 'blimp_existential_there_quantifiers_1',
'blimp_existential_there_quantifiers_2', 'blimp_existential_there_subject_raising',
'blimp_expletive_it_object_raising', 'blimp_inchoative', 'blimp_intransitive',
'blimp_irregular_past_participle_adjectives',
'blimp_irregular_past_participle_verbs',
'blimp_irregular_plural_subject_verb_agreement_1',
'blimp_irregular_plural_subject_verb_agreement_2',
'blimp_left_branch_island_echo_question', 'blimp_left_branch_island_simple_question',
'blimp_matrix_question_npi_licensor_present', 'blimp_npi_present_1',
'blimp_npi_present_2', 'blimp_only_npi_licensor_present', 'blimp_only_npi_scope',
'blimp_passive_1', 'blimp_passive_2', 'blimp_principle_A_c_command',
```

'blimp_principle_A_case_1', 'blimp_principle_A_case_2', 'blimp_principle_A_domain_1',
'blimp_principle_A_domain_2', 'blimp_principle_A_domain_3',
'blimp_principle_A_reconstruction', 'blimp_regular_plural_subject_verb_agreement_1',
'blimp_regular_plural_subject_verb_agreement_2',
'blimp_sentential_negation_npi_licensor_present',
'blimp_sentential_negation_npi_scope', 'blimp_sentential_subject_island',
'blimp_superlative_quantifiers_1', 'blimp_superlative_quantifiers_2',
'blimp_tough_vs_raising_1', 'blimp_tough_vs_raising_2', 'blimp_transitive',
'blimp_wh_island', 'blimp_wh_questions_object_gap', 'blimp_wh_questions_subject_gap',
'blimp_wh_questions_subject_gap_long_distance', 'blimp_wh_vs_that_no_gap',
'blimp_wh_vs_that_no_gap_long_distance', 'blimp_wh_vs_that_with_gap',
'blimp_wh_vs_that_with_gap_long_distance', 'boolq', 'cb', 'cola', 'copa', 'coqa',
'crows_pairs_english', 'crows_pairs_english_age', 'crows_pairs_english_autre',
'crows_pairs_english_disability', 'crows_pairs_english_gender',
'crows_pairs_english_nationality', 'crows_pairs_english_physical_appearance',
'crows_pairs_english_race_color', 'crows_pairs_english_religion',
'crows_pairs_english_sexual_orientation', 'crows_pairs_english_socioeconomic',
'crows_pairs_french', 'crows_pairs_french_age', 'crows_pairs_french_autre',
'crows_pairs_french_disability', 'crows_pairs_french_gender',
'crows_pairs_french_nationality', 'crows_pairs_french_physical_appearance',
'crows_pairs_french_race_color', 'crows_pairs_french_religion',
'crows_pairs_french_sexual_orientation', 'crows_pairs_french_socioeconomic',
'csatqa_gr', 'csatqa_li', 'csatqa_rch', 'csatqa_rcs', 'csatqa_rcss', 'csatqa_wr',
'cycle_letters', 'drop', 'ethics_cm', 'ethics_deontology', 'ethics_justice',
'ethics_utilitarianism', 'ethics_utilitarianism_original', 'ethics_virtue', 'gsm8k',
'haerae_hi', 'haerae_kgk', 'haerae_lw', 'haerae_rc', 'haerae_rw', 'haerae_sn',
'headqa', 'headqa_en', 'headqa_es', 'hellaswag', 'hendrycksTest-abstract_algebra',
'hendrycksTest-anatomy', 'hendrycksTest-astronomy', 'hendrycksTest-business_ethics',
'hendrycksTest-clinical_knowledge', 'hendrycksTest-college_biology', 'hendrycksTest-
college_chemistry', 'hendrycksTest-college_computer_science', 'hendrycksTest-
college_mathematics', 'hendrycksTest-college_medicine', 'hendrycksTest-
college_physics', 'hendrycksTest-computer_security', 'hendrycksTest-
conceptual_physics', 'hendrycksTest-econometrics', 'hendrycksTest-
electrical_engineering', 'hendrycksTest-elementary_mathematics', 'hendrycksTest-
formal_logic', 'hendrycksTest-global_facts', 'hendrycksTest-high_school_biology',
'hendrycksTest-high_school_chemistry', 'hendrycksTest-high_school_computer_science',
'hendrycksTest-high_school_european_history', 'hendrycksTest-high_school_geography',
'hendrycksTest-high_school_government_and_politics', 'hendrycksTest-
high_school_macroeconomics', 'hendrycksTest-high_school_mathematics', 'hendrycksTest-
high_school_microeconomics', 'hendrycksTest-high_school_physics', 'hendrycksTest-
high_school_psychology', 'hendrycksTest-high_school_statistics', 'hendrycksTest-
high_school_us_history', 'hendrycksTest-high_school_world_history', 'hendrycksTest-
human_aging', 'hendrycksTest-human_sexuality', 'hendrycksTest-international_law',
'hendrycksTest-jurisprudence', 'hendrycksTest-logical_fallacies', 'hendrycksTest-
machine_learning', 'hendrycksTest-management', 'hendrycksTest-marketing',
'hendrycksTest-medical_genetics', 'hendrycksTest-miscellaneous', 'hendrycksTest-
moral_disputes', 'hendrycksTest-moral_scenarios', 'hendrycksTest-nutrition',
'hendrycksTest-philosophy', 'hendrycksTest-prehistory', 'hendrycksTest-
professional_accounting', 'hendrycksTest-professional_law', 'hendrycksTest-
professional_medicine', 'hendrycksTest-professional_psychology', 'hendrycksTest-
public_relations', 'hendrycksTest-security_studies', 'hendrycksTest-sociology',
'hendrycksTest-us_foreign_policy', 'hendrycksTest-virology', 'hendrycksTest-
world_religions', 'iwslt17-ar-en', 'iwslt17-en-ar', 'lambada_openai',
'lambada_openai_cloze', 'lambada_openai_mt_de', 'lambada_openai_mt_en',
'lambada_openai_mt_es', 'lambada_openai_mt_fr', 'lambada_openai_mt_it',

```
'lambada_standard', 'lambada_standard_cloze', 'logiqa', 'math_algebra', 'math_asdiv',
'math_counting_and_prob', 'math_geometry', 'math_intermediate_algebra',
'math_num_theory', 'math_prealgebra', 'math_precalc', 'mathqa', 'mc_taco', 'mgsm_bn',
'mgsm_de', 'mgsm_en', 'mgsm_es', 'mgsm_fr', 'mgsm_ja', 'mgsm_ru', 'mgsm_sw',
'mgsm_te', 'mgsm_th', 'mgsm_zh', 'mnli', 'mnli_mismatched', 'mrpc', 'multirc',
'mutual', 'mutual_plus', 'openbookqa', 'pawsx_de', 'pawsx_en', 'pawsx_es',
'pawsx_fr', 'pawsx_ja', 'pawsx_ko', 'pawsx_zh', 'pile_arxiv', 'pile_bookcorpus2',
'pile_books3', 'pile_dm-mathematics', 'pile_enron', 'pile_europarl', 'pile_freelaw',
'pile_github', 'pile_gutenberg', 'pile_hackernews', 'pile_nih-exporter',
'pile_opensubtitles', 'pile_openwebtext2', 'pile_philpapers', 'pile_pile-cc',
'pile_pubmed-abstracts', 'pile_pubmed-central', 'pile_stackexchange', 'pile_ubuntu-
irc', 'pile_uspto', 'pile_wikipedia', 'pile_youtubesubtitles', 'piqa', 'prost',
'pubmedqa', 'qa4mre_2011', 'qa4mre_2012', 'qa4mre_2013', 'qasper', 'qnli', 'qqp',
'race', 'random_insertion', 'record', 'reversed_words', 'rte', 'sciq',
'scrolls_contractnli', 'scrolls_govreport', 'scrolls_narrativeqa', 'scrolls_qasper',
'scrolls_qmsum', 'scrolls_quality', 'scrolls_summscreenfd', 'squad2', 'sst', 'swag',
'toxigen', 'triviaqa', 'truthfulqa_gen', 'truthfulqa_mc', 'webqs', 'wic', 'wikitext',
'winogrande', 'wmt14-en-fr', 'wmt14-fr-en', 'wmt16-de-en', 'wmt16-en-de', 'wmt16-en-
ro', 'wmt16-ro-en', 'wmt20-cs-en', 'wmt20-de-en', 'wmt20-de-fr', 'wmt20-en-cs',
'wmt20-en-de', 'wmt20-en-iu', 'wmt20-en-ja', 'wmt20-en-km', 'wmt20-en-pl', 'wmt20-en-
ps', 'wmt20-en-ru', 'wmt20-en-ta', 'wmt20-en-zh', 'wmt20-fr-de', 'wmt20-iu-en',
'wmt20-ja-en', 'wmt20-km-en', 'wmt20-pl-en', 'wmt20-ps-en', 'wmt20-ru-en', 'wmt20-ta-
en', 'wmt20-zh-en', 'wnli', 'wsc', 'wsc273', 'xcopa_et', 'xcopa_ht', 'xcopa_id',
'xcopa_it', 'xcopa_qu', 'xcopa_sw', 'xcopa_ta', 'xcopa_th', 'xcopa_tr', 'xcopa_vi',
'xcopa_zh', 'xnli_ar', 'xnli_bg', 'xnli_de', 'xnli_el', 'xnli_en', 'xnli_es',
'xnli_fr', 'xnli_hi', 'xnli_ru', 'xnli_sw', 'xnli_th', 'xnli_tr', 'xnli_ur',
'xnli_vi', 'xnli_zh', 'xstory_cloze_ar', 'xstory_cloze_en', 'xstory_cloze_es',
'xstory_cloze_eu', 'xstory_cloze_hi', 'xstory_cloze_id', 'xstory_cloze_my',
'xstory_cloze_ru', 'xstory_cloze_sw', 'xstory_cloze_te', 'xstory_cloze_zh',
'xwinograd_en', 'xwinograd_fr', 'xwinograd_jp', 'xwinograd_pt', 'xwinograd_ru',
'xwinograd_zh']
```

The output.

To execute the evaluation, you must utilize the `main.py` file previously cloned from Github. Ensure that you are in the same directory as the specified file to execute the command successfully. By running the provided command, you will utilize the `facebook/opt-1.3b` model and evaluate its performance on the `hellaswag` dataset using GPU acceleration. ( Note that the displayed output is truncated. For the complete output, feel free to explore the attached notebook.)

```
python main.py \
    --model hf-causal \
    --model_args pretrained=facebook/opt-1.3b \
    --tasks hellaswag \
    --device cuda:0
```

The sample code.

```
Running loglikelihood requests
100% 40145/40145 [29:44<00:00, 22.50it/s]
```

```
{
  "results": {
    "hellaswag": {
      "acc": 0.4146584345747859,
      "acc_stderr": 0.00491656121359129,
      "acc_norm": 0.5368452499502091,
      "acc_norm_stderr": 0.004976214989483508
    }
  },
  "versions": {
    "hellaswag": 0
  },
  "config": {
    "model": "hf-causal",
    "model_args": "pretrained=facebook/opt-1.3b",
    "num_fewshot": 0,
    "batch_size": null,
    "batch_sizes": [],
    "device": "cuda:0",
    "no_cache": false,
    "limit": null,
    "bootstrap_iters": 100000,
    "description_dict": {}
  }
}
```


```
hf-causal (pretrained=facebook/opt-1.3b), limit: None, provide_description: False,
num_fewshot: 0, batch_size: None
|  Task   |Version| Metric |Value |   |Stderr|
|---------|------:|--------|-----:|---|-----:|
|hellaswag|      0|acc     |0.4147|±  |0.0049|
|         |       |acc_norm|0.5368|±  |0.0050|Copy
```

The truncated output.

For a more in-depth exploration, the `--model` argument offers three options to choose from: `hf-causal` for specifying the language model, `hf-causal-experimental` for utilizing multiple GPUs, and `hf-seq2seq` for evaluating encoder-decoder models.
Consequently, the `--model_args` parameter can be used to pass any additional arguments to the model. For instance, to employ a specific revision of the model with the float data type, utilize the following input: `--model_args pretrained=EleutherAI/pythia-160m,revision=step100000,dtype="float`." The arguments vary based on the chosen model and the inputs accepted by Huggingface, as this library leverages Huggingface to load open-source models. Otherwise, you can use the following to specify the engine type while evaluating OpenAI models: `--model_args engine=davinci`.
Lastly, performing a combined evaluation using a combination of tasks is possible. To achieve this, simply pass a comma-separated string of available metrics like `--tasks hellaswag, and arc_challenge`, enabling the usage of both Hellaswag and ARC metrics simultaneously.
To evaluate proprietary models from OpenAI, you need to set the `OPENAI_API_SECRET_KEY` environmental variable with the secret key. You can obtain this key from the OpenAI dashboard and use it accordingly.

```
export OPENAI_API_SECRET_KEY=YOUR_KEY_HERE

python main.py \

    --model gpt3 \

    --model_args engine=davinci \

    --tasks hellaswagCopy
```

The sample code.

# InstructEval

There are other efforts to evaluate the language model, like the [InstructEval leaderboard](), which is an effort that combines automated evaluation and using the GPT-4 model for scoring different models. Additionally, it is worth mentioning that they mainly focus on instruction-tuned models.

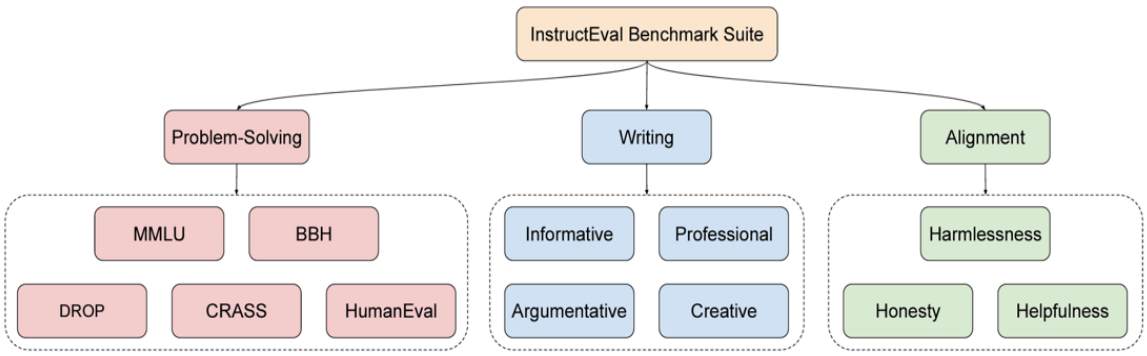



Figure 1: Overview of INSTRUCTEVAL, our holistic evaluation suite for Instructed LLMs

Image from the [InstructEval paper]().

The evaluation is broken down into three distinct tasks.

**1. Problem-Solving Evaluation**

It consists of the following test to evaluate the model's ability on **World Knowledge** using Massive Multitask Language Understanding (MMLU), **Complex Instructions** using BIG-Bench Hard (BBH), **Comprehension and Arithmetic** using Discrete Reasoning Over Paragraphs (DROP), **Programming** using HumanEval, and lastly **Causality** using Counterfactual Reasoning Assessment (CRASS). These automated evaluations assess the model's performance across various tasks.

**2. Writing Evaluation**

This category will evaluate the model based on the following subjective metrics: Informative, Professional, Argumentative, and Creative. They used the GPT-4 model to evaluate the output of different models by presenting a rubric and asking the model to score the outputs on the Likert scale between 1 and 5.

**3. Alignment to Human Values**

Finally, a crucial aspect of instruction-tuned models is their alignment with human values. We anticipate these models will uphold values such as helpfulness, honesty, and harmlessness. The leaderboard will evaluate the model by presenting pairs of dialogues and asking it to choose the appropriate one.

We won't dive into an extensive explanation of the evaluation process as it closely resembles the previous benchmark, involving the execution of a Python script. Please follow the link to their GitHub repository, where they provide sample usage.

## Conclusion

Having standardized metrics for evaluating different models is essential; otherwise, comparing the capabilities of various models would become impractical.

In this lesson, we introduced several widely used metrics along with a script that facilitates the evaluation of LLMs. It is essential to emphasize the importance of keeping track of the latest leaderboard and evaluation metrics based on specific use cases. In most instances, having a model that excels in all tasks may not be necessary, so staying updated with relevant metrics helps identify the most suitable model for tailored requirements.

>> Notebook.

# Domain-Specific LLMs

## Introduction

Domain-specific Language Models are tailored for specific industries or use cases. Unlike generalized language models attempting to comprehend a wide array of topics, domain-specific LLMs are finely tuned to understand a particular domain's unique terminology, context, and intricacies. In this lesson, we'll see what it takes to create a domain-specific LLM, how to do it, and what popular domain-specific LLMs are.

## When Do Domain-Specific LLMs Make Sense?

Domain-specific LLMs offer distinct advantages over their generalized counterparts in scenarios where precision, accuracy, and context are very important. They excel in industries where specialized knowledge is essential for generating relevant and accurate outputs. Moreover, they are also good in certain scenarios for safety (constraining what the model knows about), and

smaller model have lower latency and are cheaper to host/infer with respect to an LLM, especially if for a single task.

So, what are some specific industries where domain-specific LLMs could work well? Two notable examples are:

1. **Finance**: Domain-specific LLMs can provide personalized investment recommendations based on an individual's financial goals, optimizing investment strategies.
2. **Healthcare**: A domain-specific LLM trained in medical data can comprehend complex medical queries and offer accurate advice, enhancing patient care and medical consultation.

Why don't we use general-purpose LLMs in these fields, too?

General-purpose LLMs gather their knowledge from their pre-training phase and are "steered" into valuable assistants in the finetuning phase. GPT-4 may know the correct answer to a medical question since it may be trained on medical papers too, but it may not be appropriately steered into a good "medical assistant" that would also ask meaningful questions. However, we're still in the infancy of LLM research, so it's still hard to correctly reason how they work.

As long as the required knowledge was in the pre-training data, in principle, the LLM is likely able to behave in the "correct" way if appropriately finetuned. If we think the required knowledge wasn't in the training data, we'd have to pre-train a new LLM from scratch. If we think otherwise, then we could focus on finetuning.

In finance, Bloomberg recently trained a proprietary LLM from scratch using a mix of general-purpose and financial data called BloombergGPT.

## BloombergGPT

BloombergGPT is a proprietary domain-specific 50B LLM trained for the financial domain.

Its **training dataset** is called "FinPile" and is made of many English financial documents derived from diverse sources, encompassing financial news, corporate filings, press releases, and even social media taken from Bloomberg archives (thus, it's a proprietary dataset). Data ranges from company filings to market-relevant news from March 2007 to July 2022.

The dataset is further augmented by the integration of publicly available general-purpose text, creating a balance between domain specificity and the broader linguistic landscape. In the end, the final domain is approximately half domain-specific (51.27%) and half general-purpose (48.73%).

The model is **based on the BLOOM model.** It's a decoder-only transformer with 70 layers of decoder blocks, multi-head self-attention, layer normalization, and feed-forward networks equipped with the GELU non-linear function. The model is based on the Chinchilla scaling laws.

BloombergGPT outperforms other models like GPT-NeoX, OPT-66B, and BLOOM-176B on financial tasks. However, when confronted with GPT-3 on general-purpose tasks, GPT-3 achieves better results.

## The FinGPT Project

The FinGPT project aims to bring the power of LLMs into the world of finance. It aims to do so in two ways:

1. Providing open finance datasets.
2. Finetuning open-source LLMs on finance datasets for several use cases.

Many datasets collected by FinGPT are specifically for financial sentiment analysis. What do we mean by "financial sentiment"?

For example, the sentence "Operating profit rose to EUR 13.1 mn from EUR 8.7 mn in the corresponding period in 2007 representing 7.7 % of net sales" merely states facts; therefore, its "normal" sentiment would be neutral. However, it states that the company's operating profit rose, which is good news for someone who wants to invest in that company, and therefore the financial

sentiment is “positive.” Similarly, the sentence “The international electronic industry company Elcoteq has laid off tens of employees from its Tallinn facility” has “negative” financial sentiment. Some datasets for financial sentiment classification are:

- Financial Phrasebank: It contains 4840 sentences from English financial news, categorized by financial sentiment (written by agreement between 5-8 annotators).
- Financial Opinion Mining and Question Answering (FIQA)**:** Consists of 17k sentences from microblog headlines and financial news, classified with financial sentiment.
- Twitter Financial Dataset (sentiment): about 10k tweets with financial sentiment.

Here, you can find a notebook showing how to finetune a model with these datasets and how to use the final model for predictions.

## Med-PaLM for the Medical Domain

Med-PaLM is a finetuned version of PaLM (by Google) specifically for the medical domain.
The first iteration of Med-PaLM, introduced in late 2022 and subsequently published in Nature in July 2023, marked a milestone by surpassing the pass mark on US Medical License Exam (USMLE) style questions.
Building upon this success, Google Health unveiled the latest iteration, Med-PaLM 2, during its annual health event, The Check Up, in March 2023. Med-PaLM 2 represents a substantial leap forward, achieving an accuracy rate of 86.5% on USMLE-style questions.

## Conclusion

Domain-specific LLMs are specialized tools finely tuned for domain expertise. They are indicated for specific fields like finance and healthcare, where nuanced understanding is very important. Examples include BloombergGPT for finance, FinGPT for financial sentiment analysis, and Med-PaLM for medical inquiries.