Fine-Tuning LLMs Module

Fine-Tuning LLMs

Goals: Equip students with knowledge and practical skills in finetuning techniques accompanied by code examples. Discuss the approach to finetuning utilizing CPUs.

The section centers around fine-tuning LLMs, addressing their various aspects and methodologies. As the module progresses, the focus will be given to specialized instruction tuning techniques, namely SFT and and LoRA. It will examine domain-specific applications, ensuring a holistic understanding of fine-tuning techniques and their real-world implications.

- **Techniques for Finetuning LLMs:** The lesson highlights the challenges, particularly the resource intensity of traditional approaches. We will introduce instruction tuning methods like SFT, RLHF, and LoRA.
- **Deep Dive into LoRA and SFT**: This lesson offers an in-depth exploration of LoRA and SFT techniques. We will uncover the mechanics and underlying principles of these methods.
- **Finetuning using LoRA and SFT**: This lesson guides a practical application of LoRA and SFT to finetune an LLM to follow instructions, using data from the “LIMA: Less Is More for Alignment” paper.
- **Finetuning using SFT for financial sentiment**. This lesson navigates the nuances of leveraging SFT to optimize LLMs, specifically tailored to capture and interpret sentiments within the financial domain.
- **Fine-Tuning using Cohere for Medical Data.** In this lesson, we will adopt an entirely different method for fine-tuning a large language model, leveraging a service called Cohere. This lesson explores the procedure of fine-tuning a customized generative model using medical texts to extract information. The task, known as Named Entity Recognition (NER), empowers the models to identify various entities (such as names, locations, dates, etc.) within a text.

  The lessons have equipped students with both the knowledge and the practical skills needed for fine-tuning LLMs effectively. As they advance, they carry with them the capability to deploy and optimize LLM for various domains.

Techniques for Fine-Tuning LLMs

Introduction

In this lesson, we will examine the main techniques for fine-tuning Large Language Models for superior performance on specific tasks. We explore why and how to fine-tune LLMs, the strategic importance of instruction fine-tuning, and several fine-tuning methods, such as Full Finetuning, Low-Rank Adaptation (LoRA), Supervised Finetuning (SFT), and Reinforcement Learning from Human Feedback (RLHF). We also touch upon the benefits of the Parameter-Efficient Fine-tuning (PEFT) approach using Hugging Face's PEFT library, promising both efficiency and performance gains in fine-tuning.

Why We Finetune LLMs

While pretraining provides Language Models (LLMs) with a broad understanding of language, it doesn't equip them with the specialized knowledge needed for complex tasks. For instance, a pre-trained LLM may excel at generating text but encounter difficulties when tasked with sentiment analysis of financial news. This is where fine-tuning comes into play.

Fine-tuning is the process of adapting a pretrained model to a specific task by further training it using task-specific data. For example, if we aim to make an LLM proficient in answering questions about medical texts, we would fine-tune it using a dataset comprising medical question-answer pairs. This process enables the model to recalibrate its internal parameters and representations to align with the intended task, enhancing its capacity to address domain-specific challenges effectively.

However, fine-tuning LLMs conventionally can be resource-intensive and costly. It involves adjusting all the parameters in the pretrained LLM models, which can number in the billions, necessitating significant computational power and time. Consequently, it's crucial to explore more efficient and cost-effective methods for fine-tuning, such as Low-Rank Adaptation (LoRA).

A Reminder On Instruction Finetuning

Instruction fine-tuning is a specific type of fine-tuning that grants precise control over a model's behavior. The objective is to train a Language Model (LLM) to interpret prompts as instructions rather than simply treating them as text to continue generating. For example, when given the instruction, "Analyze the sentiment of this text and tell us if it's positive," a model with instruction fine-tuning would perform sentiment analysis rather than continuing the text in some manner.

This technique offers several advantages. It involves training models on tasks described using instructions, enabling LLMs to generalize to new tasks based on additional instructions. This approach circumvents the need for extensive amounts of task-specific data and relies on textual instructions to guide the learning process.

A Reminder of the Techniques For Finetuning LLMs

There are several techniques to make the finetuning process more efficient and effective:

- **Full Finetuning:** This method involves adjusting all the parameters in the pretrained LLM models to adapt to a specific task. While effective, it is resource-intensive and requires extensive computational power, therefore it’s rarely used.
- **Low-Rank Adaptation (LoRA):** LoRA is a technique that aims to adapt LLMs to specific tasks and datasets while simultaneously reducing computational resources and costs. By applying low-rank approximations to the downstream layers of LLMs, LoRA significantly reduces the number of parameters to be trained, thereby lowering the GPU memory requirements and training costs. We’ll also see QLoRA, a variant of LoRA that is more optimized and leverages quantization.

With a focus on the number of parameters involved in finetuning, there are multiple methods, such as:

- **Supervised Finetuning (SFT):** SFT involves doing standard supervised finetuning with a pretrained LLM on a small amount of demonstration data. This method is less resource-intensive than full finetuning but still requires significant computational power.
- **Reinforcement Learning from Human Feedback (RLHF):** RLHF is a training methodology where models are trained to follow human feedback over multiple iterations. This method can be more effective than SFT, as it allows for continuous improvement based on human feedback. We’ll also see some alternatives to RLHF, such as Direct Preference Optimization (DPO), and Reinforcement Learning from AI Feedback (RLAIF).

Efficient Finetuning with Hugging Face PEFT Library

Parameter-Efficient Fine-tuning (PEFT) approaches address the need for computational and storage efficiency in fine-tuning LLMs. Hugging Face developed the PEFT library specifically for this purpose. PEFT leverages architectures that only fine-tune a small number of additional model

parameters while freezing most parameters of the pretrained LLMs, significantly reducing computational and storage costs.
PEFT methods offer benefits beyond just efficiency. These methods have been proven to outperform standard fine-tuning methods, particularly in **low-data situations**, and provide improved generalization for out-of-domain scenarios. Furthermore, they contribute to the portability of models by generating tiny model checkpoints that require substantially less storage space compared to extensive full fine-tuning checkpoints.
By integrating PEFT strategies, we make way for comparable performance gains to full fine-tuning with only a fraction of the trainable parameters. This, in effect, broadens our capacity to harness the prowess of LLMs, regardless of the hardware limitations we might encounter.
Providing easy integration with the Hugging Face's Transformers and Accelerate libraries, the PEFT library supports popular methods such as Low-Rank Adaptation (**LoRA**) and **Prompt Tuning**.

Conclusion

In this lesson, we've learned that while pretraining equips LLMs with a broad understanding of language, fine-tuning is necessary to specialize these models for complex tasks. We've introduced various fine-tuning techniques, including Full Finetuning, Low-Rank Adaptation (LoRA), Supervised Finetuning (SFT), and Reinforcement Learning from Human Feedback (RLHF). We've also highlighted the importance of instruction fine-tuning for precise control over model behavior. Finally, we've examined the benefits of Parameter-Efficient Fine-tuning (PEFT) approaches, mainly using Hugging Face's PEFT library, which promises both efficiency and performance gains in fine-tuning. This equips us to harness the power of LLMs more effectively and efficiently, regardless of hardware limitations, and to adapt these models to a wide range of tasks and domains.

Deep Dive into LoRA and SFT

Introduction

In this lesson, we will dive deeper into the mechanics of LoRA, a powerful method for optimizing the fine-tuning process of Large Language Models, its practical uses in various fine-tuning tasks, and the open-source resources that simplify its implementation. We will also introduce QLoRA, a highly efficient version of LoRA. By the end of this lesson, you will have an in-depth understanding of how LoRA and QLoRA can enhance the efficiency and accessibility of fine-tuning LLMs.

The Functioning of LoRA in Fine-tuning LLMs

LoRA, or Low-Rank Adaptation, is a method developed by Microsoft researchers to optimize the fine-tuning of Large Language Models. This technique tackles the issues related to the fine-tuning process, such as extensive memory demands and computational inefficiency. LoRA introduces a compact set of parameters, referred to as **low-rank matrices**, to store the necessary changes in the model instead of altering all parameters.
Here are the key features of how LoRA operates:

- **Maintaining Pretrained Weights**: LoRA adopts a unique strategy by preserving the pretrained weights of the model. This approach reduces the risk of catastrophic forgetting, ensuring the model maintains the valuable knowledge it gained during pretraining.
- **Efficient Rank-Decomposition**: LoRA incorporates rank-decomposition weight matrices, known as update matrices, to the existing weights. These update matrices have significantly fewer parameters than the original model, making them highly memory-efficient. By training only these

newly added weights, LoRA achieves a faster training process with reduced memory demands. These **LoRA matrices are typically integrated into the attention layers of the original model**.

By using the low-rank decomposition approach, the memory demands for training large language models are significantly reduced. This allows running fine-tuning tasks on consumer-grade GPUs, making the benefits of LoRA available to a broader range of researchers and developers.

Open-source Resources for LoRA

The following libraries offer a mix of tools that enhance the efficiency of fine-tuning large language models. They provide optimizations, compatibility with different data types, resource efficiency, and user-friendly interfaces that accommodate various tasks and hardware configurations.

- **PEFT Library**: Parameter-efficient fine-tuning (PEFT) methods facilitate efficient adaptation of pre-trained language models to various downstream applications without fine-tuning all the model's parameters. By fine-tuning only a portion of the model's parameters, PEFT methods like **LoRA, Prefix Tuning, and P-Tuning**, including QLoRA, significantly reduce computational and storage costs.
- **Lit-GPT:** Lit-GPT from LightningAI is an open-source resource designed to simplify the fine-tuning process, making it easier to apply LoRA's techniques without manually altering the core model architecture. Models available for this purpose include Vicuna, Pythia, and Falcon. Specific configurations can be applied to different weight matrices, and precision settings can be adjusted to manage memory consumption.

In this course, we'll mainly use the PEFT library.

QLoRA: An Efficient Variant of LoRA

QLoRA, or Quantized Low-Rank Adaptation, is a popular variant of LoRA that makes fine-tuning large language models even more efficient. QLoRA introduces several innovations to save memory without sacrificing performance.

The technique involves back propagating gradients through a frozen, **4-bit quantized** pretrained language model into Low-Rank Adapters. This approach significantly reduces memory usage, enabling the fine-tuning of even larger models on consumer-grade GPUs. For instance, QLoRA can fine-tune a 65 billion parameter model on a single 48GB GPU while preserving full 16-bit fine-tuning task performance.

QLoRA uses a new data type known as **4-bit NormalFloat (NF4),** which is optimal for normally distributed weights. It also employs **double quantization** to reduce the average memory footprint by quantizing the quantization constants and paged optimizers to manage memory spikes.

The Guanaco models, which use QLoRA fine-tuning, have demonstrated state-of-the-art performance, even when using smaller models than the previous benchmarks. This shows the power of QLoRA tuning, making it a popular choice for those seeking to democratize the use of large transformer models.

The practical implementation of QLoRA for fine-tuning LLMs is very accessible, thanks to open-source libraries and tools. For instance, the **BitsAndBytes library** offers functionalities for 4-bit quantization. We'll later see a code example showing how to use QLoRA with PEFT.

Conclusion

In this lesson, we focused on LoRA and QLoRA, two powerful techniques for fine-tuning LLMs. We explored how LoRA works, preserving pretrained weights and introducing low-rank matrices to make the fine-tuning process more memory and computationally efficient. We also introduced open-source libraries like PEFT and Lit-GPT that facilitate the implementation of LoRA. Finally, we discussed QLoRA, an efficient variant of LoRA that uses 4-bit NormalFloat and double quantization to reduce memory usage.

Fine-Tuning using LoRA and SFT

Introduction

The fine-tuning process has consistently proven to be a practical approach for enhancing the model's capabilities in new domains. Therefore, it is a valuable approach to adapt large language models while using a reasonable amount of resources.

As mentioned earlier, the fine-tuning process builds upon the model's existing general knowledge, which means it doesn't need to learn everything from scratch. Consequently, it can grasp patterns from a relatively small number of samples and undergo a relatively short training process.

In this lesson, we'll see how to do SFT on an LLM using LoRA. We'll use the dataset from the "LIMA: Less Is More for Alignment" paper. According to their argument, a high-quality, hand-picked, small dataset with a thousand samples can replace the RLHF process, effectively enabling the model to be instructively fine-tuned. Their approach yielded competitive results compared to other language models, showcasing a more efficient fine-tuning process. However, it might not exhibit the same level of accuracy in domain-specific tasks, and it requires hand-picked data points.

The TRL library has some classes for Supervised Fine-Tuning (SFT), making it accessible and straightforward. The classes permit the integration of LoRA configurations, facilitating its seamless adoption. It is worth highlighting that this process also serves as the first step for Reinforcement Learning with Human Feedback (RLHF), a topic we will explore in detail later in the course.

Spinning Up a Virtual Machine for Finetuning on GCP Compute Engine

Cloud GPUs availability today is very scarse as they are used a lot for several deep learning applications. Few people know that CPUs can be actually used to finetune LLMs through various optimizations and that's what we'll be doing in these lessons when doing SFT.

Let's login to our Google Cloud Platform account and create a Compute Engine instance (see the "Course Introduction" lesson for instructions). You can choose between different machine types. In this lesson, we trained the model on the latest CPU generation from 4th Generation Intel® Xeon® Scalable Processors (formerly known as Intel® Sapphire Rapids). This architecture features an integrated accelerator designed to enhance the performance of training deep learning models. Intel® Advanced Matrix Extension (AMX) empowers the training of models with BF16 precision during the training process, allowing for half-precision training on the latest Xeon® Scalable processors. Additionally, it introduces an INT8 data type for the inference process, leading to a substantial acceleration in processing speed. Reports suggest a tenfold increase in performance when utilizing PyTorch for both training and inference processes.

Follow the instructions in the course introduction to spin up a VM with Compute Engine with high-end Intel® CPUs. Once you have your virtual machine up, you can SSH into it.

Incorporating CPUs for fine-tuning or inference processes presents an excellent choice, as renting alternate hardware is considerably less cost-effective. It worth mentioning that a minimum of 32GB of RAM is necessary to load the model and facilitate the experiment's training process. If there is an out-of-memory error, reduce arguments such as `batch_size` or `seq_length`.

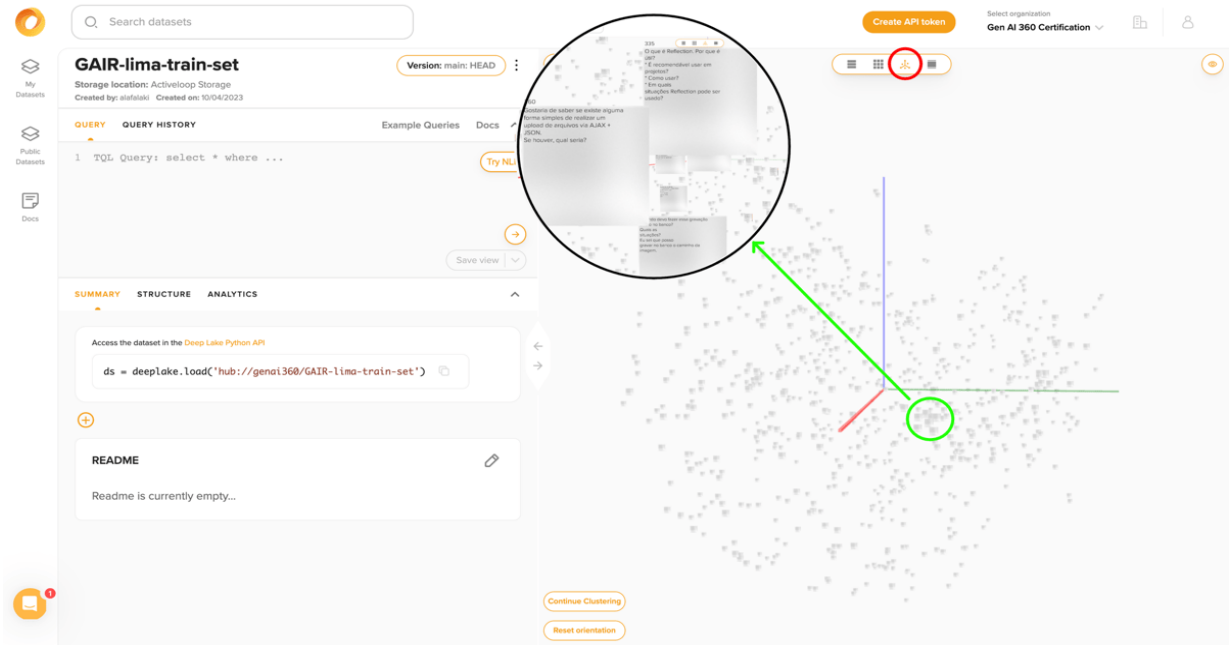
⚠ Beware of costs when you spin up virtual machines. The total cost will depend on the machine type and the up time of the machine. Always remember to monitor your costs in the billing section of GCP and to spin off your virtual machines when you don't use them.

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

Load the Dataset

The quality of a model is directly tied to the quality of the data it is trained on! The best approach is to begin the process with a dataset. Whether it is an open-source dataset or a custom one manually, planning and considering the dataset in advance is essential. In this lesson, we will utilize the dataset released with the LIMA research. It is publicly available with a non-commercial use license.

The powerful feature of Deep Lake format enables seamless streaming of the datasets. There is no need to download and load the dataset into memory. The hub provides diverse datasets, including the LIMA dataset presented in the "LIMA: Less Is More for Alignment" paper. The Deep Lake Web UI not only aids in dataset exploration but also facilitates dataset visualization using the embeddings field, taking care of clustering the dataset and map it in 3D space. (We used Cohere embedding API to generate in this example) The enlarged image below illustrates one such cluster where data points in Portuguese language related to coding are positioned closely to each other. Note that Deep Lake Visualization Engine offers you the ability to pick the clustering algorithm.



Deep Lake Visualization Engine 3D visualization feature.

The code below will create a loader object for the training and test sets.

```
import deeplake

# Connect to the training and testing datasets

ds = deeplake.load('hub://genai360/GAIR-lima-train-set')

ds_test = deeplake.load('hub://genai360/GAIR-lima-test-set')

print(ds)
```

```
Dataset(path='hub://genai360/GAIR-lima-train-set', read_only=True, tensors=['answer',
'question', 'source'])
```

The output.

We can then utilize the `ConstantLengthDataset` class to bundle a number of smaller samples together, enhancing the efficiency of the training process. Furthermore, it also handles dataset formatting by accepting a template function and tokenizing the texts.
To begin, we load the pre-trained tokenizer object for the Open Pre-trained Transformer (OPT) model using the Transformers library. We will load the model later. We are using OPT for convenience because it's an open model with a relatively "small" amount of parameters. The same code in this lesson can be run in another model too, for example, using `meta-llama/Llama-2-7b-chat-hf` for LLaMa 2.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")
```

The sample code.

Moreover, we need to define the formatting function called `prepare_sample_text`, which takes a row of data in Deep Lake format as input and formats it to begin with a question followed by the answer that is separated by two newlines. This formatting aids the model in learning the template and understanding that if a prompt starts with the `question` keyword, the most likely response would be to complete it with an answer.

```
def prepare_sample_text(example):

    """Prepare the text from a sample of the dataset."""

    text = f"Question: {example['question'].text()}\n\nAnswer:
{example['answer'].text()}"

    return text
```

The sample code.

Now, with all the components in place, we can initialize the dataset, which can be fed to the model for fine-tuning. We call the `ConstantLengthDataset` class using the combination of a tokenizer, deep lake dataset object, and formatting function. The additional arguments, such as `infinite=True` ensure that the iterator will restart when all data points have been used, but there are still training steps remaining. Alongside `seq_length`, which determines the maximum sequence length, it must be completed according to the model's configuration. In this scenario, it is possible to raise it to 2048, although we opted for a smaller value to manage memory usage better. Select a higher number if the dataset primarily comprises shorter texts.

```
from trl.trainer import ConstantLengthDataset

train_dataset = ConstantLengthDataset(

    tokenizer,

    ds,

    formatting_func=prepare_sample_text,

    infinite=True,

    seq_length=1024)
```

```
eval_dataset = ConstantLengthDataset(
    tokenizer,
    ds_test,
    formatting_func=prepare_sample_text,
    seq_length=1024)


# Show one sample from train set
iterator = iter(train_dataset)
sample = next(iterator)
print(sample)
```

```
{'input_ids': tensor([    2, 45641,    35,  ..., 48443,  2517,   742]), 'labels':
tensor([    2, 45641,    35,  ..., 48443,  2517,   742])}
```

The output.

As evidenced by the output above, the `ConstantLengthDataset` class takes care of all the necessary steps to prepare our dataset.

💡 If you use the iterator to print a sample from the dataset, remember to execute the following code to reset the iterator pointer. `train_dataset.start_iteration = 0`

Initialize the Model and Trainer

As mentioned previously, we will be using the OPT model with 1.3 billion parameters in this lesson, which has the `facebook/opt-1.3b` model id on the Hugging Face Hub.
The LoRA approach is employed for fine-tuning, which involves introducing new parameters to the network while keeping the base model unchanged during the tuning process. This approach has proven to be highly efficient, enabling fine-tuning of the model by training less than 1% of the total parameters. (For more details, refer to the following post.)
With the TRL library, we can seamlessly add additional parameters to the model by defining a number of configurations. The variable `r` represents the dimension of matrices, where lower values lead to fewer trainable parameters. `lora_alpha` serves as the scaling factor, while `bias` determines which bias parameters the model should train, with options of `none`, `all`, and `lora_only`. The remaining parameters are self-explanatory.

```
from peft import LoraConfig
lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
```

```
    bias="none",
    task_type="CAUSAL_LM",)
```

The sample code.

Next, we need to configure the `TrainingArguments`, which are essential for the training process. We have already covered some of the parameters in the training lesson, but note that the learning rate is higher when combined with higher weight decay, increasing parameter updates during fine-tuning.
Furthermore, it is highly recommended to employ the argument `bf16=True` in order to minimize memory usage during the model's fine-tuning process. The utilization of the Intel® Xeon® 4s CPU empowers us to apply this optimization technique. This involves converting the numbers to a 16-bit precision, effectively reducing the RAM demand during fine-tuning. We will dive into other quantization methods as we progress through the course.
We are also using a service called Weights and Biases, which is an excellent tool for training and fine-tuning any machine-learning model. They offer monitoring tools to record every facet of the process and various solutions for prompt engineering and hyperparameter sweep, among other functionalities. Simply installing the package and utilizing the `wandb` parameter for the `report_to` argument is all that's required. This will handle the logging process seamlessly.

```
from transformers import TrainingArguments
training_args = TrainingArguments(
    output_dir="./OPT-fine_tuned-LIMA-CPU",
    dataloader_drop_last=True,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    num_train_epochs=10,
    logging_steps=5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    learning_rate=1e-4,
    lr_scheduler_type="cosine",
    warmup_steps=10,
    gradient_accumulation_steps=1,
    bf16=True,
    weight_decay=0.05,
    run_name="OPT-fine_tuned-LIMA-CPU",
    report_to="wandb",)
```

The sample code.

The final component we need is the pre-trained model. We will use the `facebook/opt-1.3b` key to load the model using the Transformers library.

```
from transformers import AutoModelForCausalLM

import torch

model = AutoModelForCausalLM.from_pretrained("facebook/opt-1.3b",
torch_dtype=torch.bfloat16)
```

The sample code.

The subsequent code block will loop through the model parameters and revert the data type of specific layers (like LayerNorm and final language modeling head) to a 32-bit format. It will improve the fine-tuning stability.

```
import torch.nn as nn

for param in model.parameters():

  param.requires_grad = False  # freeze the model - train adapters later

  if param.ndim == 1:

    # cast the small parameters (e.g. layernorm) to fp32 for stability

    param.data = param.data.to(torch.float32)


model.gradient_checkpointing_enable()  # reduce number of stored activations

model.enable_input_require_grads()


class CastOutputToFloat(nn.Sequential):

  def forward(self, x): return super().forward(x).to(torch.float32)

model.lm_head = CastOutputToFloat(model.lm_head)
```

The sample code.

Finally, we can use the `SFTTrainer` class to tie all the components together. It accepts the model, training arguments, training dataset, and LoRA method configurations to construct the trainer object. The `packing` argument indicates that we used the `ConstantLengthDataset` class earlier to pack samples together.

```
from trl import SFTTrainer

trainer = SFTTrainer(    model=model,

                         args=training_args,

                         train_dataset=train_dataset,

                         eval_dataset=eval_dataset,

                         peft_config=lora_config,

                         packing=True,)
```

So, why did we use LoRA? Let's observe its impact in action by implementing a simple function that calculates the number of available parameters in the model and compares it with the trainable parameters. As a reminder, the trainable parameters refer to the ones that LoRA added to the base model.

```
def print_trainable_parameters(model):
    """
    Prints the number of trainable parameters in the model.
    """
    trainable_params = 0
    all_param = 0
    for _, param in model.named_parameters():
        all_param += param.numel()
        if param.requires_grad:
            trainable_params += param.numel()
    print(
        f"trainable params: {trainable_params} || all params: {all_param} ||
trainable%: {100 * trainable_params / all_param}")
print( print_trainable_parameters(trainer.model) )
```

```
trainable params: 3145728 || all params: 1318903808 || trainable%:
0.23851079820371554
```

The output.

As observed above, the number of trainable parameters is only 3 million. It accounts for only 0.2% of the total number of parameters that we would have had to update if we hadn't used LoRA! It significantly reduces the memory requirement. Now, it should be clear why using this approach for fine-tuning is advantageous.
The trainer object is fully prepared to initiate the fine-tuning loop by calling the `.train()` method, as shown below.

```
print("Training...")
trainer.train()
```

The sample code.

💡 You can access the best checkpoint that we trained by using the following URL. Additionally, find more information about the fine-tuning process on the Weights and Biases project page.

OPT-fine_tuned-LIMA-CPU.zip34953.2KB

Merging LoRA and OPT

The final step involves merging the base model with the trained LoRA layers to create a standalone model. This can be achieved by loading the desired checkpoint from SFTTrainer, followed by the base model itself using the `PeftModel` class. Begin by loading the OPT-1.3B base model if using a fresh environment.

```
from transformers import AutoModelForCausalLM

import torch

model = AutoModelForCausalLM.from_pretrained(

  "facebook/opt-1.3b", return_dict=True, torch_dtype=torch.bfloat16)
```

The sample code.

The `PeftModel` class can merge the base model with the LoRA layers from the checkpoint specified using the `.from_pretrained()` method. We should then put the model in the evaluation mode. Upon execution, it will print out the model's architecture to observe the presence of the LoRA layers.

```
from peft import PeftModel

# Load the Lora model

model = PeftModel.from_pretrained(model, "./OPT-fine_tuned-LIMA-
CPU/<desired_checkpoint>/")

model.eval()


PeftModelForCausalLM(

  (base_model): LoraModel(

    (model): OPTForCausalLM(

      (model): OPTModel(

        (decoder): OPTDecoder(

          (embed_tokens): Embedding(50272, 2048, padding_idx=1)

          (embed_positions): OPTLearnedPositionalEmbedding(2050, 2048)

          (final_layer_norm): LayerNorm((2048,), eps=1e-05, elementwise_affine=True)

          (layers): ModuleList(

            (0-23): 24 x OPTDecoderLayer(

              (self_attn): OPTAttention(

                (k_proj): Linear(in_features=2048, out_features=2048, bias=True)

                (v_proj): Linear(

                  in_features=2048, out_features=2048, bias=True

                  (lora_dropout): ModuleDict(
```

```
                    (default): Dropout(p=0.05, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=2048, out_features=16, bias=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=16, out_features=2048, bias=False)
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (q_proj): Linear(
                  in_features=2048, out_features=2048, bias=True
                  (lora_dropout): ModuleDict(
                    (default): Dropout(p=0.05, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=2048, out_features=16, bias=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=16, out_features=2048, bias=False)
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                )
                (out_proj): Linear(in_features=2048, out_features=2048, bias=True)
              )
              (activation_fn): ReLU()
              (self_attn_layer_norm): LayerNorm((2048,), eps=1e-05,
elementwise_affine=True)
              (fc1): Linear(in_features=2048, out_features=8192, bias=True)
              (fc2): Linear(in_features=8192, out_features=2048, bias=True)
              (final_layer_norm): LayerNorm((2048,), eps=1e-05,
elementwise_affine=True)
```

```
            )
          )
        )
      )
      (lm_head): Linear(in_features=2048, out_features=50272, bias=False)
    )
  )
)
```

The output.

Lastly, we can use the PEFT model's `.merge_and_unload()` method to combine the base model and LoRA layers as a standalone object. It is possible to save the weights using the `.save_pretrained()` method for later usage.

```
model = model.merge_and_unload()

model.save_pretrained("./OPT-fine_tuned-LIMA/merged")
```

The sample code.

💡 Prior to progressing to the next section to observe the outcomes of the fine-tuned model, it's important to reiterate that the base model employed in this lesson is a relatively small language model with limited capabilities when compared with the state-of-the-art models we are accustomed to by now, such as ChatGPT. Remember that the insights gained from this lesson can be easily applied to train significantly larger variations of the models, leading to notably improved outcomes. (As highlighted in the lesson's introduction, modifying the key used for loading the tokenizer/model to models with any size like LLaMA2 is possible.)

Inference

We can evaluate the fine-tuned model's outputs by employing various prompts. The code below demonstrates how we can utilize Huggingface's `.generate()` method to interact with models effortlessly. Numerous arguments and decoding strategies exist that can enhance text generation quality; however, these are beyond the scope of this course. You can explore these techniques further in an informative blog post by Huggingface.

```
inputs = tokenizer("Question: Write a recipe with chicken.\n\n Answer: ",
return_tensors="pt")

generation_output = model.generate(**inputs,
                                   return_dict_in_generate=True,
                                   output_scores=True,
                                   max_length=256,
                                   num_beams=1,
                                   do_sample=True,
```

```
                                    repetition_penalty=1.5,
                                    length_penalty=2.)

print( tokenizer.decode(generation_output['sequences'][0]) )
```

The sample code.

```
Question: Write a recipe with chicken.\n\n Answer: \n* Chicken and rice is one of the
most popular meals in China, especially during Chinese New Year celebrations when
it's served as an appetizer or main course for dinner parties (or just to eat by
yourself). It can be made from scratch using fresh ingredients like meatballs/chicken
breasts if you have them on hand but otherwise use frozen ones that are already
cooked so they don't need any additional cooking time before serving. You could also
substitute some vegetables instead such as broccoli florets which would make this
dish even more delicious! If your family doesn’t know how to cook well then I suggest
making these recipes ahead of time because once done all you really do is reheat
until hot again :)\n## Make homemade marinade\n1) Combine 1 tablespoon soy sauce, 2
tablespoons sesame oil, 3 teaspoons sugar, 4 cloves garlic minced into small pieces,
6-8 green onions chopped finely, 5 cups water, salt & pepper to taste, about 8 ounces
boneless skinless chicken breast fillets cut up fine enough not to stick together
while being mixed thoroughly - no bones needed here since there will only ever be two
servings per person), ½ cup cornstarch dissolved in ¼...
```

The output.

To carry out further experimentation with the `OPT-fine_tuned-LIMA` model, we presented an identical prompt to both the vanilla base model and the fine-tuned version. This experiment aims to measure the degree to which each of these models can follow instructions. Below is a list of prompts.

1. Write a recipe with chicken.

2. Create a marketing plan for a coffee shop.

3. Why does it rain? Explain your answer.

4. What’s the Italian translation of the word ‘house’?

The outcomes highlight the constraints and capabilities of both models. However, it is evident that the fine-tuned model learned to follow instructions better compared to the vanilla-based model. This effect would undoubtedly become more pronounced with the availability of resources to conduct the fine-tuning process for a large model.

Conclusion

During this lesson, we experimented with the fine-tuning process of Large Language models, utilizing the LoRA technique to achieve an efficient tuning process. During our exploration, we discovered the importance of the process. It can serve as a starting point for RLHF or be used for instruction tuning. In the upcoming lessons, we will experiment with the fine-tuning process for creating domain-specific models.

>> Notebook.

>> W&B Report.

Fine-Tuning using SFT for Financial Sentiment

Introduction

In the previous lesson, we experimented with the method of fine-tuning an LLM to follow the instructions like a chatbot. Although this proves beneficial across various applications, we can similarly employ this strategy to train a model tailored for a particular domain.

In this lesson, our goal is to create a thoroughly tuned model for conducting **sentiment analysis on financial statements**. Ideally, the LLM would assess financial tweets by categorizing them as Positive, Negative, or Neutral. The dataset utilized in this lesson is the one curated in the FinGPT project.

As previously stated, the dataset remains the pivotal and influential factor. Having acknowledged that, and given that we've extensively addressed the process of Supervised Fine-Tuning (SFT) before, this lesson will predominantly touch upon the dataset we utilized and the preprocessing steps involved. Nonetheless, a comprehensive notebook script for running and experimenting is provided at the conclusion of this lesson.

The activities showcased in this tutorial involve the utilization of the 4th Generation Intel® Xeon® Scalable Processors (with 64GB RAM), with the use of Intel® Advanced Matrix Extensions (Intel® AMX). Both finetuning and inference can be accomplished by leveraging its optimization technologies. You can spin up a virtual machine using GCP Compute Engine as explained in the previous lesson.

Follow the instructions in the course introduction to spin up a VM with Compute Engine with high-end Intel CPUs.

⚠ Beware of costs when you spin up virtual machines. The total cost will depend on the machine type and the up time of the machine. Always remember to monitor your costs in the billing section of GCP and to spin off your virtual machines 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 virtual machine and stop it after a few iterations.

Load the Dataset

We are set to utilize the FinGPT sentiment dataset, comprising a set of financial tweets along with their corresponding labels. Additionally, this dataset features an `instruction` column containing the initial task directive. Typically, this instruction prompts something akin to "What is the sentiment of the following content? Choose from Positive, Negative, or Neutral."

A smaller subset of the dataset can be accessed from the 300+ free public datasets curated by team Activeloop accessible in Deep Lake format. We've deliberately chosen a smaller subset to expedite the fine-tuning process. Specifically, the training set comprises 20,000 data points, while we employ 2,000 samples for validation purposes. The dataset can be explored and queried using the Deep Lake Web UI or filtered using the Python package. The Deep Lake visualization engine enables us to query the dataset and filter relevant rows using its query field. The NLP feature allows you to compose your query in English and receive the corresponding TQL query.

The Deep Lake Visualization Engine table view with filtering.

By utilizing the `deeplake.load()` function, we can create the Dataset object and load the samples.

```
import deeplake

# Connect to the training and testing datasets

ds = deeplake.load('hub://genai360/FingGPT-sentiment-train-set')

ds_valid = deeplake.load('hub://genai360/FingGPT-sentiment-valid-set')

print(ds)
```

The sample code.

```
Dataset(path='hub://genai360/FingGPT-sentiment-train-set', read_only=True,
tensors=['input', 'instruction', 'output'])
```

The output.

At this point, we can proceed to create the function that formats a sample from the dataset into a suitable input for the model. The primary distinction from our previous approach lies in incorporating the instructions at the start of the prompt. The structure is as outlined below: `<instruction>\n\nContent: <tweet>\n\nSentiment: <sentiment>`. The placeholders enclosed in `<>` will be substituted with corresponding values from the dataset.

```
def prepare_sample_text(example):

    """Prepare the text from a sample of the dataset."""

    text = f"{example['instruction'].text()}\n\nContent:
{example['input'].text()}\n\nSentiment: {example['output'].text()}"

    return text
```

The sample code.

Presented here is a formatted input derived from an entry in the dataset.

```
What is the sentiment of this news? Please choose an answer from
{negative/neutral/positive}


Content: Diageo Shares Surge on Report of Possible Takeover by Lemann


Sentiment: positive
```

The output.

The subsequent steps should be recognizable from earlier lessons. We initialize the tokenizer for the [OPT-1.3B large language model]() and use the `ConstantLengthDataset` to structure the samples. The tokenizer is then employed to convert them into token IDs. Additionally, the class packs multiple samples until the sequence length threshold is reached, thus enhancing the efficiency of the training process.

```
# Load the tokenizer
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")
# Create the ConstantLengthDataset
from trl.trainer import ConstantLengthDataset
train_dataset = ConstantLengthDataset(
    tokenizer,
    ds,
    formatting_func=prepare_sample_text,
    infinite=True,
    seq_length=1024)
eval_dataset = ConstantLengthDataset(
    tokenizer,
    ds_valid,
    formatting_func=prepare_sample_text,
    seq_length=1024)


# Show one sample from train set
iterator = iter(train_dataset)
sample = next(iterator)
print(sample)
```

The sample code.

```
{'input_ids': tensor([50118, 35212,  8913,  ...,  2430,     2,     2]),'labels':
tensor([50118, 35212,  8913,  ...,  2430,     2,     2])}
```

The output.

💡 Remember to execute the following code to reset the iterator pointer, if the iterator is used to print a sample from the dataset, `train_dataset.start_iteration = 0`

Initialize the Model and Trainer

The "Fine-Tuning using SFT" tutorial clarifies the code snippets within this subsection. For additional inquiries, kindly refer to that resource for further understanding. We will quickly walk through the code.

Please bear in mind that the fine-tuned checkpoint will be accessible in the Inference section if resources for the fine-tuning process are needed. Additionally, the specifics of the training process are recorded and can be accessed through Weights and Biases. During the training process,

system activity can be tracked, including metrics such as memory usage, CPU utilization, duration, loss values, and a range of other parameters. Here's the Weights and Biases report of the finetuning of this lesson.
We start by defining the arguments necessary to configure the training process. We use the `LoraConfig` class from the PEFT library for that. Subsequently, we employ the `TrainingArguments` class from the transformers library to control the training loop.

```
# Define LoRAConfig
from peft import LoraConfig
lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",)

# Define TrainingArguments
from transformers import TrainingArguments
training_args = TrainingArguments(
    output_dir="./OPT-fine_tuned-FinGPT-CPU",
    dataloader_drop_last=True,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    num_train_epochs=10,
    logging_steps=5,
    per_device_train_batch_size=12,
    per_device_eval_batch_size=12,
    learning_rate=1e-4,
    lr_scheduler_type="cosine",
    warmup_steps=100,
    gradient_accumulation_steps=1,
    gradient_checkpointing=False,
    fp16=False,
    bf16=True,
    weight_decay=0.05,
    ddp_find_unused_parameters=False,
```

```
    run_name="OPT-fine_tuned-FinGPT-CPU",
    report_to="wandb",)
```

The sample code.

The subsequent task involves loading the OPT-1.3B model in the `bfloat16` format, which is easily managed by Intel® CPUs and saves memory during fine-tuning.

```
from transformers import AutoModelForCausalLM
import torch
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-1.3b", torch_dtype=torch.bfloat16)
```

The sample code.

The subsequent stage entails casting specific layers within the network to complete 32-bit precision, enhancing the model's stability throughout training.

```
from torch import nn


for param in model.parameters():
  param.requires_grad = False  # freeze the model - train adapters later
  if param.ndim == 1:
    # cast the small parameters (e.g. layernorm) to fp32 for stability
    param.data = param.data.to(torch.float32)


model.gradient_checkpointing_enable()  # reduce number of stored activations
model.enable_input_require_grads()


class CastOutputToFloat(nn.Sequential):
  def forward(self, x): return super().forward(x).to(torch.float32)
model.lm_head = CastOutputToFloat(model.lm_head)
```

The sample code.

Now, connect the model, dataset, training arguments, and Lora config together using the `SFTTrainer` class to start the training process by invoking the `.train()` method.

```
from trl import SFTTrainer


trainer = SFTTrainer(
    model=model,
```

```
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    peft_config=lora_config,
    packing=True,)

print("Training...")
trainer.train()
```

The sample code.

💡 Access the best checkpoint that we trained by using the following URL. Additionally, find more information about the fine-tuning process on the Weights and Biases project page.

OPT-fine_tuned-FinGPT-CPU.zip35038.5KB

Merging LoRA and OPT

Before conducting inference and observing the results, the final task is to load the LoRA adaptors from the preceding stage and merge them with the base model.

Copy

```
# Load the base model (OPT-1.3B)
from transformers import AutoModelForCausalLM
import torch
model = AutoModelForCausalLM.from_pretrained(
  "facebook/opt-1.3b", return_dict=True, torch_dtype=torch.bfloat16)

# Load the LoRA adaptors
from peft import PeftModel

# Load the Lora model
model = PeftModel.from_pretrained(model, "./OPT-fine_tuned-FinGPT-
CPU/<desired_checkpoint>/")
model.eval()
model = model.merge_and_unload()

# Save for future use
model.save_pretrained("./OPT-fine_tuned-FinGPT-CPU/merged")
```

Inference

We randomly selected four previously unseen examples from the dataset and provided them as input to both the vanilla base model (OPT-1.3B) and the fine-tuned model in order to contrast their respective responses. The code is relatively straightforward when utilizing the `.generate()` method from the Transformers library.

```
inputs = tokenizer("""What is the sentiment of this news? Please choose an answer
from {strong negative/moderately negative/mildly negative/neutral/mildly
positive/moderately positive/strong positive}, then provide some short reasons.\n\n

Content: UPDATE 1-AstraZeneca sells rare cancer drug to Sanofi for up to S300
mln.\n\nSentiment: """, return_tensors="pt").to("cuda:0")


generation_output = model.generate(**inputs,
                                   return_dict_in_generate=True,
                                   output_scores=True,
                                   max_length=256,
                                   num_beams=1,
                                   do_sample=True,
                                   repetition_penalty=1.5,
                                   length_penalty=2.)


print( tokenizer.decode(generation_output['sequences'][0]) )
```

```
What is the sentiment of this news? Please choose an answer from {strong
negative/moderately negative/mildly negative/neutral/mildly positive/moderately
positive/strong positive}, then provide some short reasons. Content: UPDATE 1-
AstraZeneca sells rare cancer drug to Sanofi for up to S300 mln. Sentiment: positive
```

The output.

Observing the samples, we see that the model fine-tuned on financial tweets for the specific domain exhibits good performance in terms of adhering to instructions and comprehending the task at hand. Below, find a list of prompts to toggle the outputs by clicking on the right arrow icon.

1. UPDATE 1-AstraZeneca sells rare cancer drug to Sanofi for up to S300 mln.

2. SABMiller revenue hit by weaker EM currencies

3. Buffett's Company Reports 37 Percent Drop in 2Q Earnings

4. For a few hours this week, the FT gained access to Poly, the university where students in Hong Kong have been trapped…

These instances demonstrate that the vanilla model primarily focuses on the default language modeling task, which involves predicting the next word based on the input. In contrast, the fine-tuned model comprehends the instruction and generates the requested content.

Conclusion

This tutorial illustrated the procedure of leveraging publicly accessible datasets or curated data from your organization to develop a personalized model that caters to personalized requirements. More powerful base models, such as LLaMA2 can be employed, by modifying the model id to load both the model and its tokenizer. However, it needs more resources to initiate the fine-tuning process.

We also showcased the feasibility of conducting the fine-tuning process using 4th Generation Intel® Xeon® Scalable Processors for both the fine-tuning and inference stages. The Intel® oneAPI Math Kernel Library (Intel® MKL) toolkits offer a suite of utilities aimed at boosting the efficiency of various applications, including those in the field of Artificial Intelligence. It's intriguing to anticipate how the latest CPU architectures like Emerald Rapids, Sierra Forest, and Granite Rapids will shape and potentially transform the deep learning landscape.

In the upcoming lessons, we will integrate GPUs to fine-tune a model using the RLHF (Reinforcement Learning from Human Feedback) approach.

>> notebook.

>> Weights and Biases report.

Fine-Tuning using Cohere for Medical Data

Introduction

In this lesson, we will adopt an entirely different method for fine-tuning a large language model, leveraging a platform called Cohere. This approach allows you to craft a personalized model with just providing sample inputs and outputs, while the service handles the fine-tuning process in the background. Essentially, you supply a set of examples and, in return, obtain a fine-tuned model. For instance, in the context of a classification model, a sample entry would consist of a pair containing <text, label>

Cohere utilizes a collection of exclusive models to execute various functions like rerank, embedding, chat, and more, all accessible through APIs. The interfaces encompass not only generative tasks but also a diverse array of endpoints that are explicitly designed for Retrieval Augmented Generation (RAG) applications. In this context, the robust base model produces contextually informed representations. Read more about Semantic Search for more details.

Additionally, they empower us to enhance their models by customizing them to suit our precise use case through fine-tuning.

It is possible to create custom models for 3 distinct objectives:

1) Generative task where we expect the model to generate a text as the output,
2) Classifier which the model will categorizes the text in different categories, or
3) Rerank to enhance semantic search results.

This lesson explores the procedure of fine-tuning a customized generative model using medical texts to extract information. The task, known as Named Entity Recognition (NER), empowers the models to identify various entities (such as names, locations, dates, etc.) within a text. Large Language Models simplify the process of instructing a model to locate desired information within content. In the following sections we will delve into the procedure of fine-tuning a model to extract diseases, chemicals, and their relationships from a paper abstract.

Cohere API

The Cohere service offers a range of robust base models tailored for various objectives. Since our focus is on generative tasks, you have the option to select either base models for faster

performance or command models for enhanced capability. Both variants also include a "light" version, which is a smaller-sized model, providing you with additional choices.
To access the API, you must first create an account on the Cohere platform. You can then proceed to the "API Keys" page, where you will find a Trial key available for free usage. Note that the trial key has rate limitations and cannot be used in a production environment. Nonetheless, there is a valuable opportunity to utilize the models and conduct your experiments prior to submitting your application for production deployment.
Now, let's install the Cohere Python package to seamlessly use their API. You should run the following command in terminal.

```
pip install cohere
```

Next, you'll need to create a Cohere object, which requires your API key and a prompt to generate a response for your request. You can utilize the following code, but please remember to replace the API placeholder with your own key.

```
import cohere

co = cohere.Client("<API_KEY>")

prompt = """The following article contains technical terms including diseases, drugs and chemicals. Create a list only of the diseases mentioned.

Progressive neurodegeneration of the optic nerve and the loss of retinal ganglion cells is a hallmark of glaucoma, the leading cause of irreversible blindness worldwide, with primary open-angle glaucoma (POAG) being the most frequent form of glaucoma in the Western world. While some genetic mutations have been identified for some glaucomas, those associated with POAG are limited and for most POAG patients, the etiology is still unclear. Unfortunately, treatment of this neurodegenerative disease and other retinal degenerative diseases is lacking. For POAG, most of the treatments focus on reducing aqueous humor formation, enhancing uveoscleral or conventional outflow, or lowering intraocular pressure through surgical means. These efforts, in some cases, do not always lead to a prevention of vision loss and therefore other strategies are needed to reduce or reverse the progressive neurodegeneration. In this review, we will highlight some of the ocular pharmacological approaches that are being tested to reduce neurodegeneration and provide some form of neuroprotection.

List of extracted diseases:"""

response = co.generate(

    model='command',

    prompt = prompt,

    max_tokens=200,

    temperature=0.750)

base_model = response.generations[0].text

print(base_model)

- glaucoma

- primary open-angle glaucoma
```

The output.

The provided code utilizes the `cohere.Client()` method to input your API key. Subsequently, the `prompt` variable will contain the model's instructions. In this case, we want the model to read a scientific paper's abstract from the PubMed website and extract the list of diseases it can identify. Finally, we employ the `cohere` object's `.generate()` method to specify the model type and provide the prompts, along with certain control parameters. The `max_tokens` parameter determines the maximum number of new tokens the model can produce, while the `temperature` parameter governs the level of randomness in the generated results. As you can see, the command model is robust enough to identify diseases without the need for any examples or additional information. In the upcoming sections, we will explore the fine-tuning feature to assess whether we can enhance the model performance even further.

The Dataset

Before delving into the details of fine-tuning, let's begin by introducing the dataset we are utilizing and clarifying the objective. We will be utilizing the dataset known as BC5CDR which is short for BioCreative V Chemical Disease Relation. This dataset comprises 1,500 PubMed research papers that have been manually annotated by human experts with structured information. The data has been divided into training, validation, and testing sets, with each set containing 500 samples.

Our goal is to fine-tune the model to enable it to identify and extract the names of various diseases/chemicals and their relationship from text. This is very useful because the information about the relationships between chemicals and diseases are usually specified in the paper abstracts, but in this form it's not actionable. That is, it's not possible to search for "all the chemicals that influence the disease X", because we'd have to read all the papers mentioning the "disease X" to do it. If we had an accurate way of extracting this structured information from the unstructured texts of the papers, it would be useful for doing these searches.

Now, let's perform some preprocessing on the dataset to transform it into a suitable format for the Cohere service. They support files in three formats: CSV, JSONL, or plain text files. We will use the JSONL format, which should align with the following template.

```
{"prompt": "This is the first prompt", "completion": "This is the first completion"}
{"prompt": "This is the second prompt", "completion": "This is the second completion"}
```

The sample format.

You can download the dataset in JSON format here as follows.

bc5cdr.json6652.3KB

Then, we can open file using the code below.

💡 The provided code is an example showcasing the extraction of disease names. Nevertheless, our final dataset will encompass not only diseases but also chemicals and their corresponding relationships. We present a single step aimed at minimizing the repetition of code. For the complete preprocessing procedure and the resulting dataset, please refer to the notebook.

```
with open('bc5cdr.json') as json_file:
    data = json.load(json_file)
print(data[0])
```

{'passages': [{'document_id': '227508', 'type': 'title', 'text': 'Naloxone reverses the antihypertensive effect of clonidine.', 'entities': [{'id': '0', 'offsets': [[0, 8]], 'text': ['Naloxone'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D009270'}]}, {'id': '1', 'offsets': [[49, 58]], 'text': ['clonidine'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D003000'}]}], 'relations': [{'id': 'R0', 'type': 'CID', 'arg1_id': 'D008750', 'arg2_id': 'D007022'}]}, {'document_id': '227508', 'type': 'abstract', 'text': 'In unanesthetized, spontaneously hypertensive rats the decrease in blood pressure and heart rate produced by intravenous clonidine, 5 to 20 micrograms/kg, was inhibited or reversed by nalozone, 0.2 to 2 mg/kg. The hypotensive effect of 100 mg/kg alpha-methyldopa was also partially reversed by naloxone. Naloxone alone did not affect either blood pressure or heart rate. In brain membranes from spontaneously hypertensive rats clonidine, 10(-8) to 10(-5) M, did not influence stereoselective binding of [3H]-naloxone (8 nM), and naloxone, 10(-8) to 10(-4) M, did not influence clonidine-suppressible binding of [3H]-dihydroergocryptine (1 nM). These findings indicate that in spontaneously hypertensive rats the effects of central alpha-adrenoceptor stimulation involve activation of opiate receptors. As naloxone and clonidine do not appear to interact with the same receptor site, the observed functional antagonism suggests the release of an endogenous opiate by clonidine or alpha-methyldopa and the possible role of the opiate in the central control of sympathetic tone.', 'entities': [{'id': '2', 'offsets': [[93, 105]], 'text': ['hypertensive'], 'type': 'Disease', 'normalized': [{'db_name': 'MESH', 'db_id': 'D006973'}]}, {'id': '3', 'offsets': [[181, 190]], 'text': ['clonidine'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D003000'}]}, {'id': '4', 'offsets': [[244, 252]], 'text': ['nalozone'], 'type': 'Chemical', 'normalized': []}, {'id': '5', 'offsets': [[274, 285]], 'text': ['hypotensive'], 'type': 'Disease', 'normalized': [{'db_name': 'MESH', 'db_id': 'D007022'}]}, {'id': '6', 'offsets': [[306, 322]], 'text': ['alpha-methyldopa'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D008750'}]}, {'id': '7', 'offsets': [[354, 362]], 'text': ['naloxone'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D009270'}]}, {'id': '8', 'offsets': [[364, 372]], 'text': ['Naloxone'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D009270'}]}, {'id': '9', 'offsets': [[469, 481]], 'text': ['hypertensive'], 'type': 'Disease', 'normalized': [{'db_name': 'MESH', 'db_id': 'D006973'}]}, {'id': '10', 'offsets': [[487, 496]], 'text': ['clonidine'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D003000'}]}, {'id': '11', 'offsets': [[563, 576]], 'text': ['[3H]-naloxone'], 'type': 'Chemical', 'normalized': []}, {'id': '12', 'offsets': [[589, 597]], 'text': ['naloxone'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D009270'}]}, {'id': '13', 'offsets': [[637, 646]], 'text': ['clonidine'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D003000'}]}, {'id': '14', 'offsets': [[671, 695]], 'text': ['[3H]-dihydroergocryptine'], 'type': 'Chemical', 'normalized': []}, {'id': '15', 'offsets': [[750, 762]], 'text': ['hypertensive'], 'type': 'Disease', 'normalized': [{'db_name': 'MESH', 'db_id': 'D006973'}]}, {'id': '16', 'offsets': [[865, 873]], 'text': ['naloxone'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D009270'}]}, {'id': '17', 'offsets': [[878, 887]], 'text': ['clonidine'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D003000'}]}, {'id': '18', 'offsets': [[1026, 1035]], 'text': ['clonidine'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D003000'}]}, {'id': '19', 'offsets': [[1039, 1055]], 'text': ['alpha-methyldopa'], 'type': 'Chemical', 'normalized': [{'db_name': 'MESH', 'db_id': 'D008750'}]}], 'relations': [{'id': 'R0', 'type': 'CID', 'arg1_id': 'D008750', 'arg2_id': 'D007022'}]}], 'dataset_type': 'train'}

The output.

Now, we can iterate through the dataset, extract the abstracts and related entities, and include the necessary instructions for training. There are two sets of instructions: the first set aids the model in understanding the task, while the second set prompts it how to generate the response.

```
instruction = "The following article contains technical terms including diseases,
drugs and chemicals. Create a list only of the diseases mentioned.\n\n"

output_instruction = "\n\nList of extracted diseases:\n"
```

The sample code.

The `instruction` variable establishes the guidelines, while the `output_instruction` defines the desired format for the output. Now, we loop through the dataset and format each instance.

```
the_list = []

for item in data:

  dis = []

  if item['dataset_type'] != "test": continue; # Don't use test set

        # Extract the disease names

        for ent in item['passages'][1]['entities']: # The annotations

    if ent['type'] == "Disease": # Only select disease names

      if ent['text'][0] not in dis: # Remove duplicate diseases in a text

        dis.append(ent['text'][0])

        the_list.append(

                {'prompt':      instruction +

                                item['passages'][1]['text'] +

                                output_instruction,

                'completion': "- "+ "\n- ".join(dis)})
```

The sample code.

The mention code may appear complex, but for each sample, it essentially iterates through all the annotations and selectively chooses only the disease-related ones. We employ this approach because the dataset includes extra labels for chemicals, which are not relevant to our model. Finally, it will generate a dictionary containing the `prompt` and `completion` keys. The prompt will incorporate the paper abstract and append the instructions to it, whereas the completion will contain a list of disease names, with each name on a separate line. Now, use the following code to save the dataset in JSONL format.

```
# Writing to sample.json

with open("disease_instruct_all.jsonl", "w") as outfile:

  for item in the_list:

    outfile.write(json.dumps(item) + "\n")
```

The formatted dataset will be saved in a file called `disease_instruct_all.jsonl`. Also, it worth noting that we are concatenating the training and validation set to make a total of 1K samples. The final dataset that is used for fine-tuning has 3K samples which is consists of 1K for diseases + 1K for Chemicals + 1K for their relationships.

💡 Below is the link to the final preprocessed dataset.

both_rel_instruct_all.jsonl4417.3KB

The Fine-Tuning

Now, it's time to employ the prepared dataset for the fine-tuning process. The good news is that we have completed the majority of the challenging tasks. The Cohere platform will only request a nickname to save your custom model. It's worth noting that they provide advanced options if you wish to train your model for a longer duration or adjust the learning rate. Here is a detailed guide on training a custom model, and you can also refer to the Cohere documentation for Training Custom Models, complete with helpful screenshots.

You should navigate to the models page using the sidebar and click on the “Create a custom model” button. On the next page, you will be prompted to select the model type, which, in our case, will be the `Generate` option. It is time to proceed to upload a dataset, either from the previous step or from your custom dataset. Afterward, click the "Review data" button to display a few samples from the dataset. This step is designed to verify that the platform can read your data as expected. If everything appears to be in order, click the "Continue" button.

The last step is to chose a nickname for your model. Also, you can change the training hyperparameters by clicking on the “HYPERPARAMETERS (OPTIONAL)” link. You have options such as `train_steps` to determine the duration, `learning_rate` to adjust the model's speed of adaptation, and `batch_size`, which specifies the number of samples the model processes in each iteration, among others. In our experience, the default parameters worked well, but feel free to experiment with these settings. Once you are ready, click the "Initiate training" button.

That’s it! Cohere will send you an email once the fine-tuning process is completed, providing you with the model ID for use in your APIs.

Extract Disease Names

In the code snippet below, we employ the same prompt as seen in the first section; however, we use the model ID of the network we just fine-tuned. Let’s see if there are any improvements.

```
response = co.generate(
    model='2075d3bc-eacf-472e-bd26-23d0284ec536-ft',
    prompt=prompt,
    max_tokens=200,
    temperature=0.750)
disease_model = response.generations[0].text
print(disease_model)
```

```
- neurodegeneration
- glaucoma
- blindness
```

```
- POAG
- glaucomas
- retinal degenerative diseases
- neurodegeneration
- neurodegenerationCopy
```

The output.

As evident from the output, the model can now identify a broad spectrum of new diseases, highlighting the effectiveness of the fine-tuning approach. The Cohere platform offers both a user-friendly interface and a potent base model to build upon.

Extract Chemical Names

In the upcoming test, we will assess the performance of our custom models in extracting chemical names compared to the baseline model. To eliminate the need for redundant code mentions, we will only present the prompt, followed by the output of each model for easy comparison. We utilized the following prompt to extract information from a text within the test set.

```
prompt = """The following article contains technical terms including diseases, drugs
and chemicals. Create a list only of the chemicals mentioned.


To test the validity of the hypothesis that hypomethylation of DNA plays an important
role in the initiation of carcinogenic process, 5-azacytidine (5-AzC) (10 mg/kg), an
inhibitor of DNA methylation, was given to rats during the phase of repair synthesis
induced by the three carcinogens, benzo[a]-pyrene (200 mg/kg), N-methyl-N-nitrosourea
(60 mg/kg) and 1,2-dimethylhydrazine (1,2-DMH) (100 mg/kg). The initiated hepatocytes
in the liver were assayed as the gamma-glutamyltransferase (gamma-GT) positive foci
formed following a 2-week selection regimen consisting of dietary 0.02% 2-
acetylaminofluorene coupled with a necrogenic dose of CCl4. The results obtained
indicate that with all three carcinogens, administration of 5-AzC during repair
synthesis increased the incidence of initiated hepatocytes, for example 10-20
foci/cm2 in 5-AzC and carcinogen-treated rats compared with 3-5 foci/cm2 in rats
treated with carcinogen only. Administration of [3H]-5-azadeoxycytidine during the
repair synthesis induced by 1,2-DMH further showed that 0.019 mol % of cytosine
residues in DNA were substituted by the analogue, indicating that incorporation of 5-
AzC occurs during repair synthesis. In the absence of the carcinogen, 5-AzC given
after a two thirds partial hepatectomy, when its incorporation should be maximum,
failed to induce any gamma-GT positive foci. The results suggest that hypomethylation
of DNA per se may not be sufficient for initiation. Perhaps two events might be
necessary for initiation, the first caused by the carcinogen and a second involving
hypomethylation of DNA.


List of extracted chemicals:"""
```

The sample code.

First, we will examine the output of the base model.

```
- 5-azacytidine (5-AzC)
- benzo[a]-pyrene
- N-methyl-N-nitrosourea
- 1,2-dimethylhydrazine
- CCl4
- 2-acetylaminofluorene
```

The output. (base model)

Followed by the predictions generated by the custom fine-tuned model.

```
- 5-azacytidine
- 5-AzC
- benzo[a]-pyrene
- N-methyl-N-nitrosourea
- 1,2-dimethylhydrazine
- 1,2-DMH
- 2-acetylaminofluorene
- CCl4
- [3H]-5-azadeoxycytidine
- cytosine
```

The output. (custom model)

It is clear that the custom model is better suited for our specific task and adapts readily based on the samples it has encountered.

Extract Relations

The final test involves employing the model to extract complex relationships between chemicals and the diseases they impact. It is an advanced task that could pose some challenges for the base model. As previously done, we begin by introducing the prompt we employed from the test set.

```
prompt = """The following article contains technical terms including diseases, drugs
and chemicals. Create a list only of the influences between the chemicals and
diseases mentioned.


The yield of severe cirrhosis of the liver (defined as a shrunken finely nodular
liver with micronodular histology, ascites greater than 30 ml, plasma albumin less
than 2.2 g/dl, splenomegaly 2-3 times normal, and testicular atrophy approximately
half normal weight) after 12 doses of carbon tetrachloride given intragastrically in
the phenobarbitone-primed rat was increased from 25% to 56% by giving the initial
"calibrating" dose of carbon tetrachloride at the peak of the phenobarbitone-induced
enlargement of the liver. At this point it was assumed that the cytochrome P450/CCl4
```

```
toxic state was both maximal and stable. The optimal rat size to begin phenobarbitone
was determined as 100 g, and this size as a group had a mean maximum relative liver
weight increase 47% greater than normal rats of the same body weight. The optimal
time for the initial dose of carbon tetrachloride was after 14 days on
phenobarbitone.


List of extracted influences:"""
```

Here is the output generated by the base model.
Copy

```
severe cirrhosis of the liver influences shrinking, finely nodular, ascites, plasma
albumin, splenomegaly, testicular atrophy, carbon tetrachloride, phenobarbitone
```

The output. (base model)

And here are the generations produced by the custom model.

```
- Chemical phenobarbitone influences disease cirrhosis of the liver

- Chemical carbon tetrachloride influences disease cirrhosis of the liver
```

The output. (custom model)

The base model evidently attempts to establish some connections within the text. Nevertheless, it's evident that the custom fine-tuned model excels in producing well-formatted output and distinctly connecting each chemical to the respective disease. This task poses a significant challenge for a general-purpose model; however, it showcases the effectiveness of fine-tuning by simply providing a couple of thousands samples of the task we aim to accomplish.

Conclusion

As we have examined in various lessons within this chapter, the fine-tuning process has demonstrated itself as a potent tool for extending the capabilities of large language models, even when working with a relatively small amount of data, all while maintaining cost efficiency. For individuals new to the field of AI, especially those who are not well-versed in coding, the no-code approach offered by the Cohere service is an exceptionally powerful option. Our custom model demonstrated superior performance over the base model across three distinct tasks by performing one fine-tuning process, showcasing its capability to effectively follow the patterns presented in the dataset.
>> Notebook.

Fine-Tune LLMs with AWS and TorchTune

Introduction

In the rapidly evolving field of artificial intelligence, fine-tuning pre-trained models has become an essential technique for achieving high performance on specific tasks. The LLaMA (Large Language Model Meta AI) series has gained importance due to its versatility and effectiveness across various applications. LLaMA 3, the latest iteration, brings improved architecture and capabilities, making it an attractive choice for developers and researchers.

Fine-tuning a model involves adapting a pre-trained model to new data, enabling it to perform better on a specific task. This process requires careful consideration of the dataset, the computational resources, and the fine-tuning framework. In this article, we will explore how to fine-tune the LLaMA 3 model using Torchtune, a popular framework for model tuning, and a dataset hosted on Amazon Web Services (AWS).

Fine-Tuning LLaMA 3 Using Torchtune and AWS

Step 1: Setting Up AWS

Create an S3 Bucket:

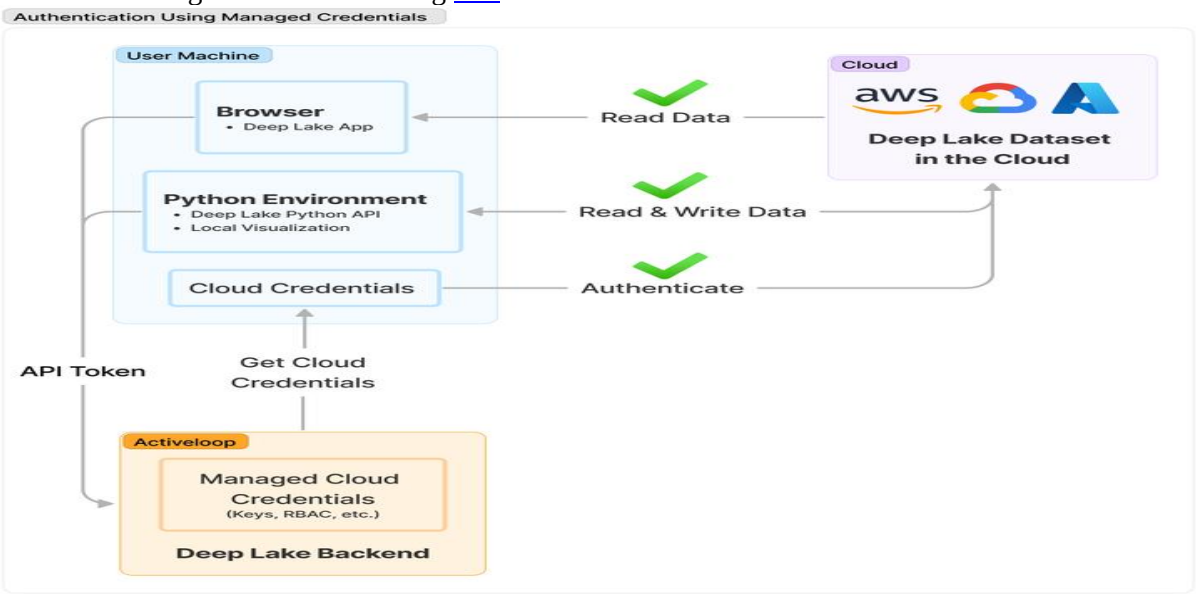
- Log in to your AWS account.
- Navigate to the S3 service and create a new bucket.
- Upload your dataset to the S3 bucket.

Step 2: Connecting Data From Your Cloud Using Deep Lake Managed Credentials

Connecting data from your cloud and managing credentials in Deep Lake offers several key benefits:

- **Access to Performance Features**: Utilize the Deep Lake Compute Engine for enhanced performance.
- **Integration with Deep Lake App** : Access datasets stored in your cloud through the Deep Lake App.
- **Simplified Access via Python API** : Easily access Deep Lake datasets stored in your cloud using the Python API.
- **Credential Management** : Avoid repeatedly specifying cloud access keys in your Python code.

For Deep Lake to access datasets or linked tensors stored in the user's cloud, it must authenticate the respective cloud resources. This can be done using access keys or through role-based access. We can also refer to our official guide at the following link for more detailed instructions.



Default storage allows us to map the Deep Lake path `hub://org_id/dataset_name` to a cloud path of our choosing. This means that all datasets created using the Deep Lake path will be stored at the location you specify and can be accessed using API tokens and managed credentials from Deep Lake. By default, the storage is set to Activeloop Storage, but you can change this through the UI in the Activeloop platform.

Step 3: Preparing the Dataset

If we do not set the Default Storage to our own cloud, we can still connect datasets in our cloud to the Deep Lake App using the Python API below. Once connected to Deep Lake, the dataset is given a Deep Lake path in the format `hub://org_id/dataset_name`, and can be accessed using API tokens and managed credentials from Deep Lake, without the need to repeatedly specify cloud credentials.

1. **Connecting Datasets in the Python API** :

```
# Step 1: Create/load the dataset directly in the cloud using your org_id and
# Managed Credentials (creds_key) for accessing the data (See Managed Credentials above)
ds = deeplake.load('s3://my_bucket/dataset_name',
                   creds={'creds_key': 'managed_creds_key'}, org_id='my_org_id')


# Step 2a: Connect the dataset to Deep Lake, inheriting the dataset_name above
ds.connect()
## ->>> This produces a Deep Lake path for accessing the dataset such as:
## ---- 'hub://my_org_id/dataset_name'


## OR


# Step 2b: Specify your own path and dataset name for future access to the dataset.
# You can also specify different managed credentials, if desired
ds.connect(dest_path = 'hub://org_id/dataset_name',creds_key = 'my_cred_key' )
```

1. **Upload the Dataset to S3**:

Use the following Python script to upload our dataset to an S3 bucket. This script uses Deep Lake's API to handle the upload process.

```
def upload_raft_dataset_to_activeloop(data_to_upload: dict, ds: deeplake.dataset):
    # create column
    with ds:
        ds.create_tensor("id", htype="text", exist_ok=True)
        ds.create_tensor("type", htype="text", exist_ok=True)
        ds.create_tensor("question", htype="text", exist_ok=True)
        ds.create_tensor("oracle_context", htype="text", exist_ok=True)
        ds.create_tensor("cot_answer", htype="text", exist_ok=True)
        ds.create_tensor("instruction", htype="text", exist_ok=True)
```

```
        for num_el in tqdm(range(len(data_to_upload))):

            ds.append(
                {
                    "id": data_to_upload["id"][num_el],
                    "type": data_to_upload["type"][num_el],
                    "question": data_to_upload["question"][num_el],
                    "oracle_context": data_to_upload["oracle_context"][num_el],
                    "cot_answer": data_to_upload["cot_answer"][num_el],
                    "instruction": data_to_upload["instruction"][num_el],
                }
            )
```

In this example we are uploading a dataset specifying which labels we want it to have. The dataset being uploaded in this example is in RAFT format, used to train an LLM as described in this paper.

```
bucketname = "YOUR_BUCKET_NAME"
dataset_name = "YOUR_DATASET_NAME"
s3_dataset_path = f's3://{bucketname}/{dataset_name}'


# use one of these 3 command to create the dataset
ds = deeplake.empty(s3_dataset_path, creds = {"aws_access_key_id": ..., ...}) # Create dataset stored in your cloud using your own credentials.
ds = deeplake.empty(s3_dataset_path, creds = {"creds_key": "managed_creds_key"}, org_id = "my_org_id") # Create dataset stored in your cloud using Deep Lake managed credentials.
ds = deeplake.empty(s3_dataset_path, overwrite=True) # Overwrite the dataset currently at the path


raft_raw_file = "raft_file.jsonl"


jsonObj = pd.read_json(path_or_buf=raft_raw_file, lines=True)


upload_raft_dataset_to_activeloop(jsonObj, ds)
```

1. **Load the Dataset**:

The `deeplake.load` function is a versatile tool for loading datasets from various storage locations. By specifying appropriate parameters and credentials, we can access our data whether it's managed by Deep Lake or stored in our own cloud infrastructure.

Copy

```
ds = deeplake.load("s3://mybucket/my_dataset", creds = {"aws_access_key_id": ..., ...}) # Load dataset stored in your cloud using your own credentials.

# OR

ds = deeplake.load("s3://mybucket/my_dataset", creds = {"creds_key": "managed_creds_key"}, org_id = "my_org_id") # Load dataset stored in your cloud using Deep Lake managed credentials
```

You can find the official documentation here for detailed instructions.

Step 4: Fine-Tuning the Model with TorchTune

Before diving into fine-tuning the model, we need to understand the `PyTorch Dataset` class for loading data from ActiveLoop's DeepLake platform. This class will facilitate the efficient loading of data, which is crucial for the fine-tuning process:

Copy

```
class DeepLakeDataloader(Dataset):
    """A PyTorch Dataset class for loading data from ActiveLoop's DeepLake platform.

    This class serves as a data loader for working with datasets stored in ActiveLoop's DeepLake platform.
    It takes a DeepLake dataset object as input and provides functionality to load data from it
    using PyTorch's DataLoader interface.

    Args:
        ds (deeplake.Dataset): The dataset object obtained from ActiveLoop's DeepLake platform.
    """

    def __init__(self, ds: deeplake.Dataset):
        self.ds = ds

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

    def __getitem__(self, idx):
        column_map = self.ds.tensors.keys()

        values_dataset = {}
```

```
        for el in column_map:  # {"column_name" : value}
            values_dataset[el] = self.ds[el][idx].text().astype(str)

        return values_dataset

def load_deep_lake_dataset(
    deep_lake_dataset: str, **config_kwargs
) -> DeepLakeDataloader:
    """
    Load a dataset from ActiveLoop's DeepLake platform.

    Args:
        deep_lake_dataset (str): The name of the dataset to load from DeepLake.
        **config_kwargs: Additional keyword arguments passed to `deeplake.dataset`.

    Returns:
        DeepLakeDataloader: A data loader for the loaded dataset.
    """
    ds = deeplake.dataset(deep_lake_dataset, **config_kwargs)
    log.info(f"Dataset loaded from deeplake: {ds}")
    return DeepLakeDataloader(ds)
```

The `DeepLakeDataloader` class is specifically designed to interact with the DeepLake dataset object. It implements the `__len__` and `__getitem__` methods to comply with PyTorch's Dataset class, making it easy to integrate with PyTorch's DataLoader for batch processing during training. The `load_deep_lake_dataset` function simplifies the process of loading a dataset from DeepLake, ensuring that all necessary configurations and credentials are handled.

Step 5: Fine-Tuning the Model with TorchTune

To be able to train the model with RAFT technique and Deep Lake Dataloader we need to download the repository and install the requirements:

```
!git clone -b feature/raft-fine-tuning <https://github.com/efenocchi/torchtune.git>
%cd torchtune
!pip install -e .
```

Now we can specify our dataset by replacing the one already present in the `torchtune/datasets/_raft.py` file, if you want to fine-tune a specific dataset change the path here `torchtune/datasets/_raft.py`

To continue, all we have to do is download the `Llama3` weights directly from Hugging Face. Please note that before you can access these files you must accept the Meta terms here.

```
llama3_original_checkpoints_folder = "llama3"
os.makedirs(llama3_original_checkpoints_folder,exist_ok = True)


lora_finetune_output_checkpoints_folder = "lora_finetune_output"
os.makedirs(lora_finetune_output_checkpoints_folder,exist_ok = True)


!tune download meta-llama/Meta-Llama-3-8B --output-dir llama3 --hf-token <YOUR_HF_TOKEN>
```

In the `torchtune/recipes/configs/llama3/8B_lora_single_device_deep_lake_raft.yaml` file change `/tmp/Meta-Llama-3-8B/original` with `llama3/original` and `/tmp/Meta-Llama-3-8B/` with `lora_finetune_output` so that the file we are going to execute during the training phase is able to point to the correct folder.
Now let's make sure we have the necessary resources for the training phase (an A100 GPU was used in this guide) and proceed with the training command:
Copy

```
!tune run lora_finetune_single_device --config
recipes/configs/llama3/8B_lora_single_device_deep_lake_raft.yaml
```

Since Torchtune performs excellently during the training phase but not as well during the testing phase, we decided to convert the PyTorch weights to the standard Hugging Face format and upload them to our space.
Make sure you are in the root project folder and not inside torchtune and install the following packages:

```
%cd ..
!pip install git+https://github.com/huggingface/transformers
%cd ..
!git clone https://github.com/huggingface/transformers
!pip install tiktoken blobfile
!pip install accelerate transformers
```

Move the tokenizer, Llama3 fine-tuned model checkpoint, and params.json file into the `weights` folder.

```
weights_folder = "weights"
os.makedirs(weights_folder,exist_ok = True)
!cp llama3/meta_model_0.pt weights/consolidated.00.pth
!cp llama3/original/params.json weights
!cp llama3/original/tokenizer.model weights
```

Now we can transform the weights into the standard used by Hugging Face and load them into our space:

```
!python transformers/src/transformers/models/llama/convert_llama_weights_to_hf.py ¥¥
--input_dir torchtune/weights ¥¥
--model_size 8B ¥¥
```

```
--output_dir hf_weights ¥¥
--llama_version 3
```

Step 5: Saving and Deploying the Model

We upload the weights on our space by choosing a suitable name:

```
from transformers import AutoTokenizer, AutoModelForCausalLM
tokenizer = AutoTokenizer.from_pretrained("hf_weights")
model = AutoModelForCausalLM.from_pretrained("hf_weights")


hf_repository_name = "llama3_RAFT"


tokenizer.push_to_hub(hf_repository_name)
model.push_to_hub(hf_repository_name)
```

Conclusion

This guide has shown how to adapt pre-trained models to meet our specific needs, enhancing their performance and utility in various applications. The process involves setting up AWS infrastructure, managing credentials with Deep Lake, uploading and connecting datasets, and using Torchtune to fine-tune the model with custom configurations.

By following these steps, we can achieve a tailored AI solution capable of performing specialized tasks with high efficiency. Fine-tuning allows us to utilize the full potential of the LLaMA 3 model, ensuring it is optimized for our unique requirements. This not only showcases the flexibility and power of modern AI frameworks but also highlights the importance of integrating cloud resources and advanced tuning techniques in AI development.