Introduction to LLMs Module

Introduction to LLMs

This module uncovers the core principles of Large Language Models (LLMs), zooming in on their foundational underpinnings. We provide a historical perspective, highlighting the emergence of Transformers and the difference between proprietary and open-source LLMs. Key attention is on recognizing and mitigating inherent issues like hallucinations and biases within these models.

The module is structured as follows, concisely describing each lesson.

- **What are Large Language Models?** This lesson dives into the core principles of LLMs, highlighting the capabilities of notable models such as GPT-3 and GPT-4. We introduce the concepts of tokens, few-shot learning, emergent abilities, and the significance of scaling laws. As we explore the functions and outputs of these models, we emphasize the potential challenges of hallucinations and biases. Additionally, we also discuss the context size limitation in LLMs.
- **The Evolution of LLMs and Transformers**: This lesson provides a chronological narrative of the progression in language modeling techniques. Starting with the foundational Bag of Words model from 1954, we navigate through significant milestones like TF-IDF, the groundbreaking Word2Vec with its semantic-rich word embeddings, and the sequence-processing capabilities of RNNs. Central to our exploration is the 2017 Transformer architecture, which set the stage for powerhouses like BERT, RoBERTa, and ELECTRA. This lesson only offers an overview without the deep technical intricacies, presenting a panoramic view of the model evolution in NLP.
- **A timeline of Large Language Models**: This lesson steers towards a comprehensive overview of the advancements in the Large Language Models landscape, spotlighting models that marked distinct milestones such as GPT-3, PaLM, and Galactica. Recognizing the significant role of techniques like scaling and alignment tuning in the unprecedented capabilities exhibited by LLMs, we untangle the principles that steer these giants. From exploring the enigmatic emergent abilities to decoding the scaling laws, this lesson explores the phenomena driving the potency and performance of LLMs.
- **Emergent Abilities in LLMs**: This lesson covers the unexpected skills that surface in Large Language Models as they grow beyond certain thresholds. As models expand, they exhibit unique capabilities influenced by factors like training compute. These emergent skills indicate performance leaps in LLMs as they scale, revealing unforeseen learning beyond what was initially anticipated.
- **Proprietary LLMs**: This lesson introduces prominent proprietary Large Language Models such as GPT-4, ChatGPT, and Cohere, among others. We'll weigh the advantages and drawbacks of proprietary models against open-source counterparts. Practical demonstrations will guide students in executing API calls for select models.
- **Open-Source LLMs**: This lesson offers insights into open-source Large Language Models, with a focus on LLaMA 2, Open Assistant, Dolly, and Falcon. We will explore

their unique features, capabilities, and licensing details. Additionally, we'll discuss potential commercial uses and emphasize any restrictions within their licenses.

- **Understanding Hallucinations and Bias in LLMs**: This lesson focuses on the challenges posed by hallucinations and biases in Large Language Models. We'll define hallucinations, provide examples, and discuss their impact on LLM use cases. We'll also explore methods to minimize these issues, such as retriever architectures. The session also covers the concept of bias, its origin in LLMs, and potential mitigation strategies, including approaches like constitutional AI.
- **Applications and Use-Cases of LLMs:** This lesson highlights the leading applications and emerging trends of Large Language Models across industries. By referencing real-world news and examples, we illustrate the transformative impact of LLMs across sectors. While emphasizing the vast potential benefits, the module also underscores the importance of recognizing LLMs' limitations and potential challenges.

This section provided a comprehensive overview of Large Language Models, highlighting their evolution and significant milestones. Topics ranged from understanding emergent abilities in LLMs to discerning between proprietary and open-source models. Critical challenges like hallucinations and biases were also addressed.

What are Large Language Models

Introduction

Welcome to our introductory module on Large Language Models or **LLMs**.
LLMs, or Large Language Models, are a specific category of neural network models characterized by having an exceptionally high number of parameters, often in the billions. These parameters are essentially the variables within the model that allow it to process and generate text. They are trained on vast quantities of textual data, which provides them with a broad understanding of language patterns and structures. The main goal of LLMs is to comprehend and produce text that closely resembles human-written language, enabling them to capture the subtle complexities of both syntax (the arrangement of words in a sentence) and semantics (the meaning conveyed by those words).
These models undergo training with a simple objective: predicting the subsequent word in a sentence. However, they develop a range of **emergent abilities** during this training process. For example, they can perform tasks such as arithmetic calculations and word unscrambling and even achieve remarkable feats like successfully passing professional-level exams such as the US Medical Licensing Exam.
They generate text in an autoregressive manner, generating the next tokens one by one based on the tokens they have previously generated.
The **attention mechanism** plays a key role in enabling these models to establish connections between words and produce coherent and contextually relevant text.
LLMs have significantly advanced the natural language processing (NLP) field, revolutionizing our approach to tasks like machine translation, natural language generation, part-of-speech tagging, parsing, information retrieval, and more.
As we dive further into this module, we will explore the capabilities of these models, their practical applications, and their exciting future possibilities.

Language Modeling

Language modeling is a fundamental task in Natural Language Processing (NLP). It involves explicitly learning the probability distribution of the words in a language. This is generally learned by predicting the next token in a sequence. This task is typically approached using statistical methods or deep learning techniques.
LLMs are trained to predict the next token (word, punctuation, etc.) based on the previous tokens in the text. The models achieve this by learning the distribution of tokens in the training data.

Tokenization

The first step in this process is tokenization, where the input text is broken down into smaller units called tokens. Tokens can be as small as individual characters or as large as whole words. The choice of token size can significantly affect the model's performance. Some models even use subword tokenization, where words are broken down into smaller units that capture meaningful linguistic information.
For example, let’s consider the sentence "The child’s book.”

We could split the text whenever we find white space characters. The output would be:
Copy

```
["The", "child's", "book."]
```

As you can see, the punctuation is still attached to the words *"child's"* and *"book."*
Otherwise, we could split the text according to white spaces and punctuation. The output would be:
Copy

```
["The", "child", "'", "s", "book", "."]
```

Importantly, tokenization is model-specific, meaning different models require different tokenization processes, which can complicate pre-processing and multi-modal modeling.

Model Architecture and Attention

The core of a language model is its architecture. Recurrent Neural Networks (RNNs) were traditionally used for this task, as they are capable of processing sequential data by maintaining an internal state that captures the information from previous tokens. However, they struggle with long sequences due to the vanishing gradient problem.
To overcome these limitations, transformer-based models have become the standard for language modeling tasks. These models use a mechanism called **attention**, which allows them to weigh the importance of different tokens when making predictions. This allows them to capture long-range dependencies between tokens and generate high-quality text.

Training

The model is trained on a large corpus of text to predict the next token of a sentence correctly. The goal is to adjust the model's parameters to maximize the probability of the observed data.
Typically a model is trained on a very large general dataset of texts from the Internet, such as The Pile or CommonCrawl. Sometimes also more specific datasets are used, such as the Stackoverflow Posts dataset.
The model learns to predict the next token in a sequence by adjusting its parameters to maximize the probability of outputting the correct next token from the training data.

Prediction

Once the model is trained, it can be used to generate text by predicting the next token in a sequence. This is done by feeding the sequence into the model, which outputs a probability distribution over the possible subsequent tokens. The next token is then chosen based on this distribution. This process can be repeated to generate sequences of arbitrary length.

Fine-Tuning

The model is often fine-tuned on a specific task after pre-training. This involves continuing the training process on a smaller, task-specific dataset. This allows the model to adapt its learned knowledge to the specific task (e.g. text translation) or specialized domain (e.g. biomedical, finance, etc), improving its performance.

This is a brief explanation, but the actual process can be much more complex, especially for state-of-the-art models like GPT-4. These models use advanced techniques and large amounts of data to achieve impressive results.

Context Size

The context size, or context window, in LLMs is the maximum number of tokens that the model can handle in one go. The context size is significant because it determines the length of the text that can be processed at once, which can impact the model's performance and the results it generates.

Different LLMs have different context sizes. For instance, the OpenAI "gpt-3.5-turbo-16k" model has a context window of 16,000 tokens. There is a natural limit to the number of tokens a model can produce. Smaller models can go up to 1k tokens, while larger models can go up to 32k tokens, like GPT-4.

Let's Generate Some Text

Let's try generating some text with LLMs. You must first generate an API key to use OpenAI's models in your Python environment. You can follow the below steps to generate the API key:

1. After creating an OpenAI account, log in.
2. After logging in, choose Personal from the top-right menu, then choose "View API keys."
3. The "Create new secret key" button is on the page containing API keys once step 2 has been finished. Clicking on that generates a secret key. Save this because it will be required in further lessons.

After that, you can save your key in a .env file like this:

Copy

```
OPENAI_API_KEY="<YOUR-OPENAI-API-KEY>"
```

Every time you start a Python script with the following lines, your key will be loaded into an environment variable called OPENAI_API_KEY. This environment variable will then be used by the openai library whenever you want to generate text.

Copy

```
from dotenv import load_dotenv

load_dotenv()
```

We are now ready to generate some text! Here's an example of it.

Copy

```
from dotenv import load_dotenv

load_dotenv()

import os
```

```
import openai


# English text to translate
english_text = "Hello, how are you?"


response = openai.ChatCompletion.create(
 model="gpt-3.5-turbo",
 messages=[
  {"role": "system", "content": "You are a helpful assistant."},
  {"role": "user", "content": f'Translate the following English text to French: "{english_text}"'}
 ],
)


print(response['choices'][0]['message']['content'])
```

Code to run

Copy

```
Bonjour, comment ça va?
```

The output

By using dotenv, you can safely store sensitive information, such as API keys, in a separate file and avoid accidentally exposing it in your code. This is particularly important when working with open-source projects or sharing your code with others, as it ensures that the sensitive information remains secure.

Few-Shot Learning

Few-shot learning in the context of LLMs refers to providing the model with a few examples before making predictions. These examples "teach" the model how to reason and act as "filters" to help the model search for relevant patterns in the dataset.

The idea of few-shot learning is fascinating as it suggests that the model can be quickly reprogrammed for new tasks. While LLMs like GPT3 excel at language modeling tasks like machine translation, they may struggle with more complex reasoning tasks.

The few-shot examples are helping the model search for relevant patterns in the dataset. The dataset, which is effectively compressed into the model's weights, can be searched for patterns that strongly respond to these provided examples. These patterns are then used to

generate the model's output. The more examples provided, the more precise the output becomes.
Here’s an example of few-shot learning:
Copy

```
from dotenv import load_dotenv
load_dotenv()
import os
import openai

# Prompt for summarization
prompt = """
Describe the following movie using emojis.

{movie}: """

examples = [
        { "input": "Titanic", "output": "🚢🌊💔🧊🎶💧🚢💔👫🎻" },
        { "input": "The Matrix", "output": "👓💊💥👾🔮🏙️🕴️🔁🔓🥋" }
]

movie = "Toy Story"
response = openai.ChatCompletion.create(
 model="gpt-3.5-turbo",
 messages=[
     {"role": "system", "content": "You are a helpful assistant."},
     {"role": "user", "content": prompt.format(movie=examples[0]["input"])},
     {"role": "assistant", "content": examples[0]["output"]},
```

```
    {"role": "user", "content": prompt.format(movie=examples[1]["input"])},
    {"role": "assistant", "content": examples[1]["output"]},
    {"role": "user", "content": prompt.format(movie=movie)},
 ]
)


print(response['choices'][0]['message']['content'])
```

Code to run

Copy

The output

Scaling Laws

Scaling laws refer to the relationship between the model's performance and factors such as the number of parameters, the size of the training dataset, the compute budget, and the network architecture. They were discovered after a lot of experiments and are described in the Chinchilla paper. These laws provide insights into how to allocate resources when training these models optimally.

The main elements characterizing a language model are:

1. The number of parameters (N) reflects the model's capacity to learn from data. More parameters allow the model to capture complex patterns in the data.
2. The size of the training dataset (D) is measured in the number of tokens (small pieces of text ranging from a few words to a single character).
3. FLOPs (floating point operations per second) measure the compute budget used for training.

The researchers trained the Chinchilla model, which has 70B parameters, on 1.4 trillion tokens. This aligns with the rule of thumb proposed in the paper**: for a model with X parameters, it is optimal to train it on approximately X * 20 tokens.** For example, in the context of this rule, a model with 100 billion parameters would be optimally trained on approximately 2 trillion tokens.

Applying this rule, the Chinchilla model, though smaller, performed better than other LLMs. It showed gains in language modeling and task performance and needed less memory and computing power. You can read more about Chinchilla in its paper "Training Compute-Optimal Large Language Models".

Emergent Abilities in LLMs

Emergent abilities in LLMs refer to the sudden appearance of new capabilities as the size of the model increases. These abilities, which include performing arithmetic, answering questions, summarizing passages, and more, are not explicitly trained in the model. Instead, they seem to arise spontaneously as the model scales, hence the term "emergent."
LLMs are probabilistic models that learn patterns in natural language. When these models are scaled up, they not only improve quantitatively in their ability to learn patterns, but they also exhibit qualitative changes in their behavior.

Traditionally, the models require task-specific fine-tuning and architectural modifications to perform specific tasks. However, when scaled, these models can perform these tasks without any architectural modifications or task-specific training. They can do this simply by phrasing the tasks in terms of natural language. This capability of LLMs to perform tasks without fine-tuning is remarkable in itself.
What's even more intriguing is how these abilities appear. As LLMs grow, they rapidly and unpredictably transition from near-zero to sometimes state-of-the-art performance. This phenomenon suggests that these abilities are emergent properties of the model's scale rather than being explicitly programmed into the model.
This concept of emergent abilities in LLMs has significant implications for the field of AI, as it suggests that scaling up models can lead to the spontaneous development of new capabilities.

Prompts

The text containing the instructions that we pass to LLMs is commonly known as prompts. Prompts are instructions given to AI systems like OpenAI's GPT-3 and GPT-4, providing context to generate human-like text. The more detailed the prompt, the better the model's output.

Shorter, concise, and descriptive prompts tend to yield better results as they leave room for the LLM's creativity. Specific words or phrases can help narrow down potential outcomes and ensure relevant content generation.
Writing effective prompts requires a clear goal, simplicity, strategic use of keywords, and actionability. Testing the prompts before publishing ensures the output is relevant and error-free.
Here are some prompting tips:

1. Use **precise language** when crafting a prompt – this will help ensure accuracy in the generated output:
    - Less Precise Prompt: "Write about dogs."
    - More Precise Prompt: "Write a 500-word informative article about the dietary needs of adult Golden Retrievers."
2. Provide **enough context** around each prompt – this will give a better understanding of what kind of output should be produced:
    - Less Contextual Prompt: "Write a story."

    - More Contextual Prompt: "Write a short story set in Victorian England featuring a young detective solving his first major case."
3. Test different **variations** of each prompt – this allows you to experiment with different approaches until you find one that works best:
    - Initial Prompt: "Write a blog post about the benefits of yoga."
    - Variation 1: "Compose a 1000-word blog post detailing the physical and mental benefits of regular yoga practice."
    - Variation 2: "Create an engaging blog post that highlights the top 10 benefits of incorporating yoga into daily routine."
4. **Review** generated outputs before publishing them – while most automated systems produce accurate results, occasionally mistakes occur so it's always wise to double-check everything before releasing any content into production environments:
    - Before Review: "Yoga is a great way to improve your flexibility and strength. It can also help reduce stress and improve mental clarity. However, it's important to remember that all yoga poses are suitable for everyone."
    - After Review (correcting inaccurate information): "Yoga is a great way to improve your flexibility and strength. It can also help reduce stress and improve mental clarity. However, it's important to remember that not all yoga poses are suitable for everyone. Always consult with a healthcare professional before starting any new exercise regimen."

Hallucinations and Biases in LLMs

The term **hallucinations** refers to instances where AI systems generate outputs, such as text or images, that don't align with real-world facts or inputs. For example, ChatGPT might generate a plausible-sounding answer to an entirely incorrect factual question.
Hallucinations in LLMs refer to instances where the model generates outputs that do not align with real-world facts or context. This can lead to the propagation of misinformation, especially in critical sectors like healthcare and education where the accuracy of information is of utmost importance. Similarly, bias in LLMs can result in outputs that favor certain perspectives over others, potentially leading to the reinforcement of harmful stereotypes and discrimination.

Consider an interaction where a user asks, "Who won the World Series in 2025?" If the LLM responds with, "The New York Yankees won the World Series in 2025," it's a clear case of hallucination. As of now (July 2023), the 2025 World Series hasn't taken place, so any claim about its outcome is a fabrication.
Bias in AI and LLMs is another significant issue. It refers to these models' inclination to favor specific outputs or decisions based on their training data. If the training data is predominantly from a specific region, the model might show a bias toward that region's

language, culture, or perspectives. If the training data contains inherent biases, such as gender or racial bias, the AI system might produce skewed or discriminatory outputs.
For example, if a user asks an LLM, "Who is a nurse?" and it responds with, "She is a healthcare professional who cares for patients in a hospital," it shows a gender bias. The model automatically associates nursing with women, which doesn't accurately reflect the reality where both men and women can be nurses.
Mitigating hallucinations and bias in AI systems involves refining model training, using verification techniques, and ensuring the training data is diverse and representative. Finding a balance between maximizing the model's potential and avoiding these issues remains challenging.
Interestingly, in creative domains like media and fiction writing, these "hallucinations" can be beneficial, enabling the generation of unique and innovative content.
The ultimate goal is to develop LLMs that are not only powerful and efficient but also reliable, fair, and trustworthy. By doing so, we can maximize the potential of LLMs while minimizing their risks, ensuring that the benefits of this technology are accessible to all.

Conclusion

In this introductory module, we explored the fascinating world of LLMs. These powerful models, trained on vast amounts of text data, can understand and generate human-like text. They're built on transformer architectures, allowing them to capture long-range dependencies in language and generate text in an autoregressive manner.
We covered the capabilities of LLMs, discussing their impact on the field of NLP. We've learned about few-shot learning, scaling laws, and the emergent abilities of these models. We also acknowledged the challenges that come with these models, including hallucinations and biases, emphasizing the importance of mitigating these issues.
In the next lesson, we'll see a timeline of machine learning models used for language modeling up to the beginning of Large Language Models.

The Evolution of Language Modeling up to LLMs

Introduction

In this lesson, we'll see the most popular models used for language modeling, starting from statistical ones up to the first Large Language Models (LLMs). This lesson is meant to be more like a narrative on the evolution of the models rather than a technical explanation. Therefore, don't worry if you can't understand every model in detail.

The Evolution of Language Modeling

The evolution of NLP models has been a remarkable journey marked by continuous innovation and improvement. It began with the Bag of Words model in 1954, which simply counted word occurrences in documents. This was followed by TF-IDF in 1972, which adjusted these scores based on the rarity or commonality of words.

The advent of Word2Vec in 2013 marked a significant leap forward, introducing the concept of word embeddings that captured semantic relationships between words.

This was then further enhanced by Recurrent Neural Networks (RNNs), which could learn sequence patterns and handle documents of any length.

The introduction of the Transformer architecture in 2017 revolutionized the field, with its attention mechanism allowing the model to focus on the most relevant parts of the input when generating output. This was the foundation for BERT in 2018, which used bidirectional Transformers to achieve impressive results in traditional NLP tasks.

The subsequent years saw a flurry of advancements, with models like RoBERTa, XLM, ALBERT, and ELECTRA each introducing their own improvements and optimizations.

Model's Timeline

- **[1954]** Bag of Words (BOW)
  BOW is a simple model that counts word occurrences in documents, using these counts as features. It was a basic yet effective way to analyze text. However, it did not account for word order or context.
- **[1972]** TF-IDF
  TF-IDF enhanced BOW by giving more weight to rare words and less to common ones. This improved the model's ability to discern document relevance. However, it still did not account for word context.
- **[2013]** Word2Vec
  Word2Vec introduced word embeddings, high-dimensional vectors that capture semantic relationships. These embeddings were learned by a neural network trained on a large corpus of text. This model marked a significant advancement in capturing semantic meaning in text.
- **[2014]** RNNs in Encoder-Decoder architectures
  RNNs (Recurrent Neural Networks) compute document embeddings, leveraging word context in sentences, which was not possible with word embeddings alone. Later evolved with **LSTM** [1997] to capture long-term dependencies and

to **Bidirectional RNN** [1997] to capture left-to-right and right-to-left dependencies. Eventually, **Encoder-Decoder RNNs** [2014] emerged, where an RNN creates a document embedding (i.e., the encoder), and another RNN decodes it into text (i.e., the decoder).

- **[2017]** Transformer
  The Transformer is an encoder-decoder model that leverages attention mechanisms to compute better embeddings and to align output better to input. This model marked a significant advancement in NLP tasks.
- **[2018]** BERT
  BERT is a bidirectional Transformer pre-trained using a combination of Masked Language Modeling and Next Sentence Prediction objectives. It uses global attention.
- **[2018]** GPT
  GPT is the first autoregressive model based on the Transformer architecture. Later, it evolved into **GPT-2** [2019], a bigger and optimized version of GPT pre-trained on WebText, and **GPT-3** [2020], a further bigger and optimized version of GPT-2, pre-trained on Common Crawl.
- **[2019]** CTRL
  CTRL, similar to GPT, introduced control codes for conditional text generation. This allowed for more control over the generated text.
- **[2019]** Transformer-XL
  Transformer-XL reused previously computed hidden states to attend to a longer context. This allowed the model to handle longer sequences of text.
- **[2019]** ALBERT
  ALBERT is a lighter version of BERT where (1) Next Sentence Prediction is replaced by Sentence Order Prediction, and (2) parameter-reduction techniques are used for lower memory consumption and faster training.
- **[2019]** RoBERTa
  RoBERTa is a better version of BERT, where (1) the Masked Language Modeling objective is dynamic, (2) the Next Sentence Prediction objective is dropped, (3) the BPE tokenizer is employed, and (4) better hyperparameters are used.
- **[2019]** XLM
  XLM, a multilingual Transformer, was pre-trained using objectives like Causal Language Modeling, Masked Language Modeling, and Translation Language Modeling.
- **[2019]** XLNet
  It's a Transformer-XL with a generalized autoregressive pre-training method that enables learning bidirectional dependencies.
- **[2019]** PEGASUS
  PEGASUS, a bidirectional encoder and left-to-right decoder, was pre-trained with Masked Language Modeling and Gap Sentence Generation objectives.
- **[2019]** DistilBERT

It is the same as BERT but smaller and faster while preserving over 95% of BERT's performances. Trained by distillation of the pre-trained BERT model.

- **[2019]** XLM-RoBERTa
  XLM-RoBERTa is a multilingual version of RoBERTa, trained on a multilanguage corpus with the Masked Language Modeling objective.
- **[2019]** BART
  BART, a bidirectional encoder and left-to-right decoder, was trained by corrupting text with an arbitrary noising function and learning a model to reconstruct the original text.
- **[2019]** ConvBERT
  ConvBERT replaced self-attention blocks with new ones that leveraged convolutions to better model global and local contexts.
- **[2020]** Funnel Transformer
  It's a type of Transformer that gradually compresses the sequence of hidden states to a shorter one, reducing the computation cost.
- **[2020]** Reformer
  Reformer is a more efficient Transformer thanks to local-sensitive hashing attention, axial position encoding, and other optimizations.
- **[2020]** T5
  T5, a bidirectional encoder and left-to-right decoder, was pre-trained on a mix of unsupervised and supervised tasks.
- **[2020]** Longformer
  Longformer replaced the attention matrices with sparse matrices for higher training efficiency. This made the model faster and more memory-efficient.
- **[2020]** ProphetNet
  ProphetNet was trained with the Future N-gram Prediction objective and with a novel self-attention mechanism.
- **[2020]** ELECTRA
  Lighter and better than BERT, ELECTRA was trained with the Replaced Token Detection objective. This made the model more efficient and improved its performance on NLP tasks.
- **[2021]** Switch Transformers
  Switch Transformers introduced a sparsely-activated expert Transformer model, aiming to simplify and improve over Mixture of Experts. This allowed the model to handle a wider range of tasks.

The years 2020 and 2021 are the ones where Large Language Models truly arose. Up to 2020, most language models were able to generate good-looking texts. After this date, the best language models could follow instructions and solve various tasks aside from simple text generation.

The Transformer

The most crucial model of the previous pipeline is, without doubt, the Transformer, introduced in the very popular paper "Attention Is All You Need." The Transformer is a type of neural network that is used today by all of the best Large Language Models like GPT-4, Claude, and LLaMA.

Central to Transformers is the encoder-decoder structure, which excels at modeling long-range dependencies and capturing contextual information.

The Encoder processes the input text, identifying key elements and creating word embeddings based on their relevance to other words in the sentence. In the original Transformer architecture, designed for text translation, the attention mechanism was employed in two distinct ways: encoding the source language and decoding the encoded embedding back into the target language.

On the other hand, **the Decoder** takes the encoder's output, an embedding, and transforms it back into text. Some models may opt to use only the decoder, bypassing the encoder entirely. The decoder's attention mechanism differs slightly from the encoder's, functioning more like a conventional language model by focusing on previous words during text processing. This approach is particularly useful for tasks like language generation, which is why models like GPT, primarily designed for text generation in response to an input text sequence, utilize the decoder part of the Transformer.

Later in the course, we'll learn more about the Transformer architecture.

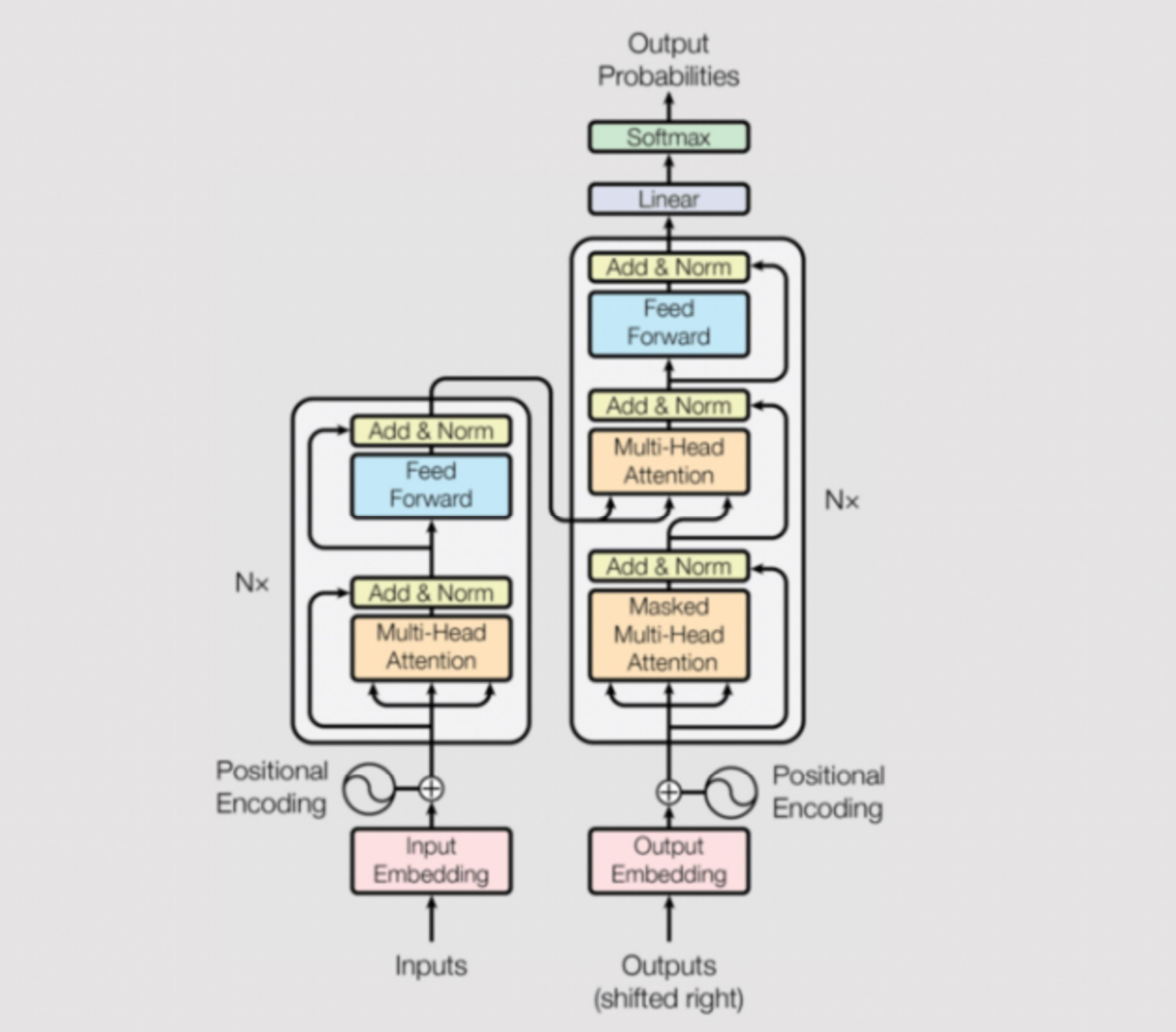


Image from the paper **Attention is All You Need.**

Scaling Transformers: What Lead to Large Language Models

The effectiveness of the Transformer models was further improved by **scaling**, i.e., increasing the number of parameters and training on more data. This scaling led to models with more than 100B parameters that could perform tasks using few-shot or zero-shot approaches, eliminating the need for fine-tuning specific tasks.

The increase in the size of these models and the datasets used for training them (and thus the associated costs) led to the large language models that we see today, like Cohere Command, GPT-4, and LLaMA.

Conclusion

In this lesson, we navigated through the rich history of Natural Language Processing, tracing the path from the rudimentary Bag of Words model to the advanced Transformer family.

This timeline underscored the continuous innovation in NLP, spotlighting the progression of models in sophistication and proficiency.
In the next lesson, we'll continue the timeline of popular models from 2020 (with GPT-3) up to today.

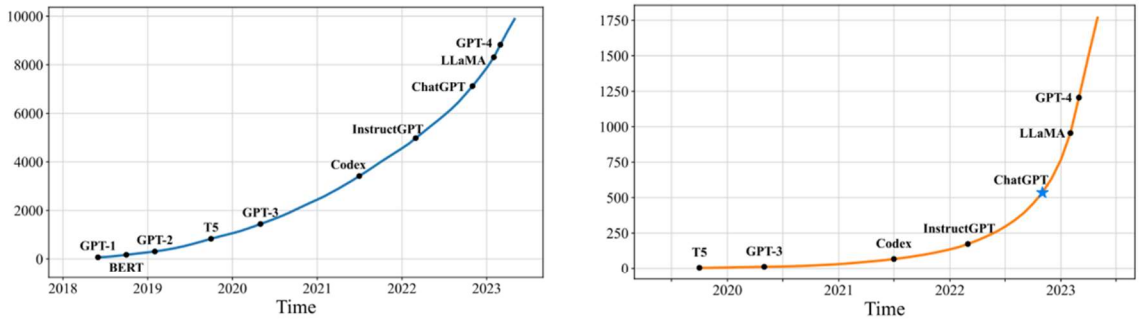
A Timeline of Large Language Models

Introduction

In this lesson, we'll explore the transformative shifts that have reshaped AI, exploring the key features that set LLMs apart from their predecessors. We'll see how scaling laws, emergent abilities, and innovative architectures have propelled LLMs to tackle complex tasks and define the current landscape of popular LLMs.

From Language Models to Large Language Models

The evolution of language models has undergone a transformative shift from pre-trained language models (LMs) to the emergence of large language models (LLMs).
LMs, like ELMo and BERT, initially captured context-aware word representations through pre-training and fine-tuning for specific tasks. However, the introduction of LLMs, exemplified by GPT-3 and PaLM, demonstrated that scaling model size and data can unlock emergent abilities, exceeding the capabilities of their smaller counterparts. These LLMs can tackle more complex tasks through in-context learning.
The following image shows the trends of the cumulative numbers of arXiv papers containing the keyphrases "language model" and "large language model," emphasizing the growing interest in them in the last years.



(a) Query="Language Model"

(b) Query="Large Language Model"

Fig. 1: The trends of the cumulative numbers of arXiv papers that contain the keyphrases *"language model"* (since June 2018) and *"large language model"* (since October 2019), respectively. The statistics are calculated using exact match by querying the keyphrases in title or abstract by months. We set different x-axis ranges for the two keyphrases, because "language models" have been explored at an earlier time. We label the points corresponding to important landmarks in the research progress of LLMs. A sharp increase occurs after the release of ChatGPT: the average number of published arXiv papers that contain *"large language model"* in title or abstract goes from 0.40 per day to 8.58 per day (Figure 1(b)).

From "A Survey of Large Language Models" paper

Key Characterizing Features of LLMs:

==========

Here are the main characteristics that differentiate LLMs from the previous models:

1. **Scaling Laws for Enhanced Capacity:** Scaling laws play a crucial role in LLM development, indicating a relationship between model performance, model size,

dataset size, and training compute. The KM scaling laws emphasize the impact of these factors, revealing distinct formulas for their influence on cross-entropy loss. The Chinchilla scaling laws provide an alternate approach, optimizing compute allocation between model and data size.

2. **Emergent Abilities:** LLMs possess emergent abilities, defined as capabilities that manifest in large models but are absent in smaller counterparts. One prominent emergent ability is *in-context learning* (ICL), showcased by models like GPT-3. ICL allows LLMs to generate unexpected outputs based on natural language instructions, eliminating the need for further training.
3. **Instruction following**: LLMs can be fine tuned to follow text instructions, which further enhances generalization for new tasks.
4. **Step-by-Step Reasoning:** LLMs can perform *step-by-step reasoning* using the chain-of-thought (CoT) prompting strategy. This mechanism enables them to solve complex tasks by breaking them down into intermediate reasoning steps, which is particularly beneficial for tasks involving multiple steps like mathematical word problems.

A Timeline of the Most Popular LLMs:

Here's an overview of the timeline of the most popular LLMs in the last few years.

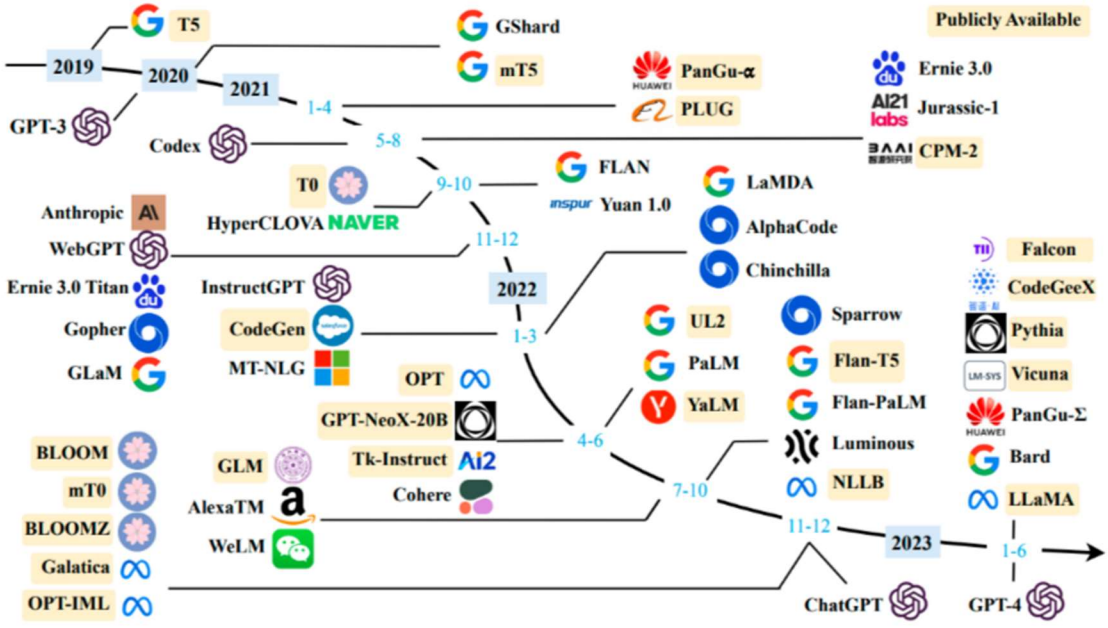


Fig. 2: A timeline of existing large language models (having a size larger than 10B) in recent years. The timeline was established mainly according to the release date (*e.g.*, the submission date to arXiv) of the technical paper for a model. If there was not a corresponding paper, we set the date of a model as the earliest time of its public release or announcement. We mark the LLMs with publicly available model checkpoints in yellow color. Due to the space limit of the figure, we only include the LLMs with publicly reported evaluation results.

From "A Survey of Large Language Models" paper

Here is a brief description of some of them.

- **[2018]** GPT-1
  GPT-1 (Generative Pre-Training 1) was introduced by OpenAI in 2018. It laid the foundation for the GPT-series models by employing a generative, decoder-only Transformer architecture. It combined unsupervised pretraining and supervised fine-tuning to predict the next word in natural language text.
- **[2019]** GPT-2
  Building upon the architecture of GPT-1, GPT-2 was released in 2019 with an increased parameter scale of 1.5 billion. This model demonstrated potential for solving a variety of tasks using language text as a unified format for input, output, and task information.
- **[2020]** GPT-3
  Released in 2020, GPT-3 marked a significant capacity leap by scaling the model to 175 billion parameters. It introduced the concept of in-context learning (ICL), enabling LLMs to understand tasks through few-shot or zero-shot learning. GPT-3 showcased excellent performance in numerous NLP tasks, including reasoning and domain adaptation, highlighting the potential of scaling up model size.
- **[2021]** Codex
  Codex was introduced by OpenAI in July 2021 as a fine-tuned version of GPT-3 specifically trained on a large corpus of GitHub code. It demonstrated enhanced ability in solving programming and mathematical problems, showcasing the potential of training LLMs on specialized data.
- **[2021]** LaMDA
  LaMDA (Language Models for Dialog Applications) was introduced by researchers from DeepMind. LaMDA focuses on enhancing dialog applications and dialog generation tasks. It has a significant number of parameters, with the largest model consisting of 137 billion parameters, making it slightly smaller than GPT-3.
- **[2021]** Gopher
  In 2021, DeepMind introduced Gopher, a language model with an impressive parameter scale of 280 billion. Notably, Gopher demonstrated a remarkable capability to approach human expert performance on the Massive Multitask Language Understanding (MMLU) benchmark. However, like its predecessors, Gopher exhibited certain limitations, including tendencies for repetition, biases, and propagation of incorrect information.
- **[2022]** InstructGPT
  In 2022, InstructGPT was proposed as an enhancement to GPT-3 for human alignment. It utilized reinforcement learning from human feedback (RLHF) to improve the model's instruction-following capacity and mitigate issues like generating harmful content. This approach proved valuable for training LLMs to align with human preferences.

- **[2022]** Chinchilla
  Chinchilla, introduced in 2022 by DeepMind, is a family of large language models that build upon the discovered scaling laws of LLMs. With a focus on efficient utilization of compute resources, Chinchilla boasts 70 billion parameters and achieves a remarkable 67.5% accuracy on the MMLU benchmark—a 7% improvement over Gopher.
- **[2022]** PaLM
  Pathways Language Model (PaLM) was introduced by Google Research in 2022, showcasing a leap in model scale with a whopping 540 billion parameters. Leveraging the proprietary Pathways system for distributed computation, PaLM exhibited great few-shot performance across an array of language understanding, reasoning, and code-related tasks.
- **[2022]** ChatGPT
  In November 2022, OpenAI released ChatGPT, a conversation model based on GPT-3.5 and GPT-4. Specially optimized for dialogue, ChatGPT exhibited great abilities in communicating with humans, reasoning, and aligning with human values.
- **[2023]** LLaMA
  LLaMA (Large Language Model Meta AI) emerged in February 2023 from Meta AI. It introduced a family of large language models available in varying sizes from 7 billion to 65 billion parameters. LLaMA's release marked a departure from the limited access trend, as its model weights were made available to the research community under a noncommercial license. Subsequent developments, including Llama 2 and other chat models, further emphasized accessibility, this time with a license for commercial use.
- **[2023]** GPT-4
  In March 2023, GPT-4 was released, extending text input to multimodal signals. With stronger capacities than GPT-3.5, GPT-4 demonstrated significant performance improvements on various tasks.

If you want to dive deeper into these models, I suggest reading the paper "A Survey of Large Language Models." Here's a table summarizing the architectural and training details of all the mentioned models (and others).

	Model	Release Time	Size (B)	Base Model	Adaptation IT	Adaptation RLHF	Pre-train Data Scale	Latest Data Timestamp	Hardware (GPUs / TPUs)	Training Time	Evaluation ICL	Evaluation CoT
Publicly Available	T5 [73]	Oct-2019	11	-	-	-	1T tokens	Apr-2019	1024 TPU v3	-	✓	-
	mT5 [74]	Oct-2020	13	-	-	-	1T tokens	-	-	-	✓	-
	PanGu-α [75]	Apr-2021	13*	-	-	-	1.1TB	-	2048 Ascend 910	-	✓	-
	CPM-2 [76]	Jun-2021	198	-	-	-	2.6TB	-	-	-	-	-
	T0 [28]	Oct-2021	11	T5	✓	-	-	-	512 TPU v3	27 h	✓	-
	CodeGen [77]	Mar-2022	16	-	-	-	577B tokens	-	-	-	✓	-
	GPT-NeoX-20B [78]	Apr-2022	20	-	-	-	825GB	-	96 40G A100	-	✓	-
	Tk-Instruct [79]	Apr-2022	11	T5	✓	-	-	-	256 TPU v3	4 h	✓	-
	UL2 [80]	May-2022	20	-	-	-	1T tokens	Apr-2019	512 TPU v4	-	✓	✓
	OPT [81]	May-2022	175	-	-	-	180B tokens	-	992 80G A100	-	✓	-
	NLLB [82]	Jul-2022	54.5	-	-	-	-	-	-	-	✓	-
	GLM [83]	Oct-2022	130	-	-	-	400B tokens	-	768 40G A100	60 d	✓	-
	Flan-T5 [64]	Oct-2022	11	T5	✓	-	-	-	-	-	✓	✓
	BLOOM [69]	Nov-2022	176	-	-	-	366B tokens	-	384 80G A100	105 d	✓	-
	mT0 [84]	Nov-2022	13	mT5	✓	-	-	-	-	-	✓	-
	Galactica [35]	Nov-2022	120	-	-	-	106B tokens	-	-	-	✓	✓
	BLOOMZ [84]	Nov-2022	176	BLOOM	✓	-	-	-	-	-	✓	-
	OPT-IML [85]	Dec-2022	175	OPT	✓	-	-	-	128 40G A100	-	✓	✓
	LLaMA [57]	Feb-2023	65	-	-	-	1.4T tokens	-	2048 80G A100	21 d	✓	-
	CodeGeeX [86]	Sep-2022	13	-	-	-	850B tokens	-	1536 Ascend 910	60 d	✓	-
	Pythia [87]	Apr-2023	12	-	-	-	300B tokens	-	256 40G A100	-	✓	-
Closed Source	GPT-3 [55]	May-2020	175	-	-	-	300B tokens	-	-	-	✓	-
	GShard [88]	Jun-2020	600	-	-	-	1T tokens	-	2048 TPU v3	4 d	-	-
	Codex [89]	Jul-2021	12	GPT-3	-	-	100B tokens	May-2020	-	-	✓	-
	ERNIE 3.0 [90]	Jul-2021	10	-	-	-	375B tokens	-	384 V100	-	✓	-
	Jurassic-1 [91]	Aug-2021	178	-	-	-	300B tokens	-	800 GPU	-	✓	-
	HyperCLOVA [92]	Sep-2021	82	-	-	-	300B tokens	-	1024 A100	13.4 d	✓	-
	FLAN [62]	Sep-2021	137	LaMDA-PT	✓	-	-	-	128 TPU v3	60 h	✓	-
	Yuan 1.0 [93]	Oct-2021	245	-	-	-	180B tokens	-	2128 GPU	-	✓	-
	Anthropic [94]	Dec-2021	52	-	-	-	400B tokens	-	-	-	✓	-
	WebGPT [72]	Dec-2021	175	GPT-3	-	✓	-	-	-	-	✓	-
	Gopher [59]	Dec-2021	280	-	-	-	300B tokens	-	4096 TPU v3	920 h	✓	-
	ERNIE 3.0 Titan [95]	Dec-2021	260	-	-	-	-	-	-	-	✓	-
	GLaM [96]	Dec-2021	1200	-	-	-	280B tokens	-	1024 TPU v4	574 h	✓	-
	LaMDA [63]	Jan-2022	137	-	-	-	768B tokens	-	1024 TPU v3	57.7 d	-	-
	MT-NLG [97]	Jan-2022	530	-	-	-	270B tokens	-	4480 80G A100	-	✓	-
	AlphaCode [98]	Feb-2022	41	-	-	-	967B tokens	Jul-2021	-	-	-	-
	InstructGPT [61]	Mar-2022	175	GPT-3	✓	✓	-	-	-	-	✓	-
	Chinchilla [34]	Mar-2022	70	-	-	-	1.4T tokens	-	-	-	✓	-
	PaLM [56]	Apr-2022	540	-	-	-	780B tokens	-	6144 TPU v4	-	✓	✓
	AlexaTM [99]	Aug-2022	20	-	-	-	1.3T tokens	-	128 A100	120 d	✓	✓
	Sparrow [100]	Sep-2022	70	-	-	✓	-	-	64 TPU v3	-	✓	-
	WeLM [101]	Sep-2022	10	-	-	-	300B tokens	-	128 A100 40G	24 d	✓	-
	U-PaLM [102]	Oct-2022	540	PaLM	-	-	-	-	512 TPU v4	5 d	✓	✓
	Flan-PaLM [64]	Oct-2022	540	PaLM	✓	-	-	-	512 TPU v4	37 h	✓	✓
	Flan-U-PaLM [64]	Oct-2022	540	U-PaLM	✓	-	-	-	-	-	✓	✓
	GPT-4 [46]	Mar-2023	-	-	✓	✓	-	-	-	-	✓	✓
	PanGu-Σ [103]	Mar-2023	1085	PanGu-α	-	-	329B tokens	-	512 Ascend 910	100 d	✓	-

From "A Survey of Large Language Models" paper

Moreover, here's an image showing the evolution of the LLaMA models into other fine-tuned models made by online communities, highlighting the great interest it sparked.

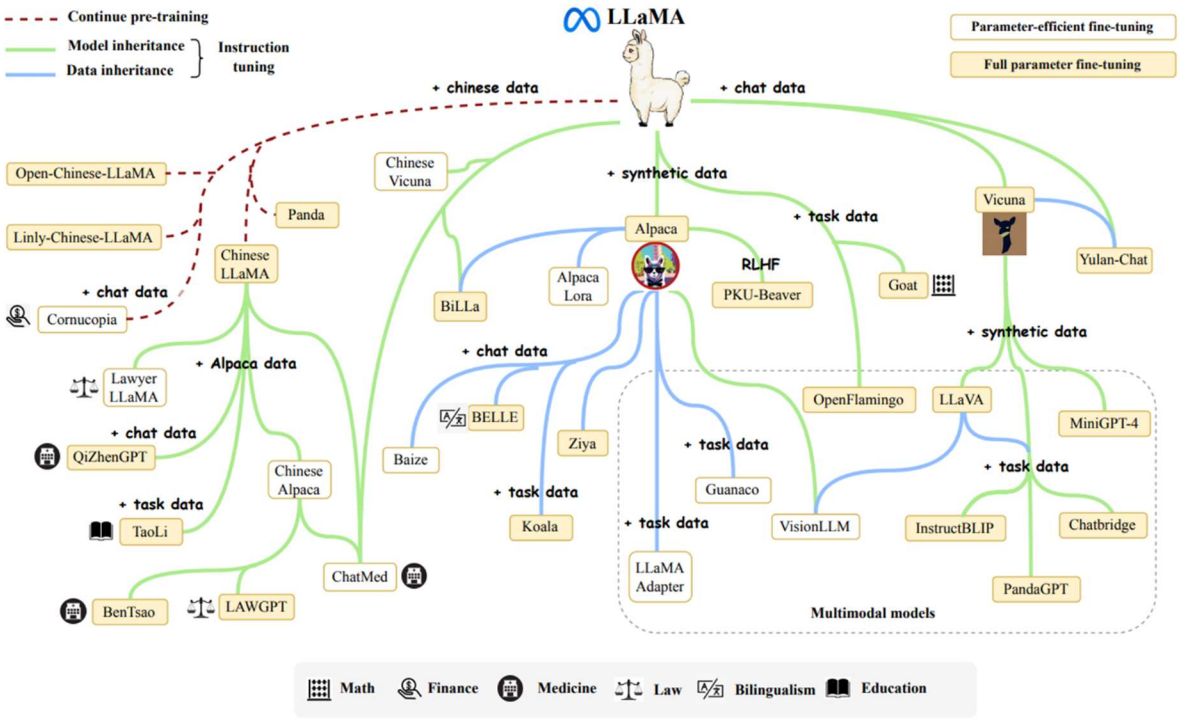


Fig. 4: An evolutionary graph of the research work conducted on LLaMA. Due to the huge number, we cannot include all the LLaMA variants in this figure, even much excellent work. To support incremental update, we share the source file of this figure, and welcome the readers to include the desired models by submitting the pull requests on our GitHub page.

From "A Survey of Large Language Models" paper

Conclusion

In this lesson, we learned more about the transition from pre-trained language models (LMs) to the emergence of large language models (LLMs). We explored the key differentiating features of LLMs, including the influence of scaling laws and the manifestation of emergent abilities like in-context learning, step-by-step reasoning strategies, and instruction following.

We also saw a brief timeline of the most popular LLMs: from the foundational GPT-1 to the revolutionary GPT-3, the specialized Codex, LaMDA, Gopher, and Chinchilla, to PaLM, ChatGPT, and LLaMA.

Emergent Abilities in LLMs

An ability is considered as emergent when larger models exhibit it, but it's absent in smaller models - a key factor contributing to the success of Large Language Models.

Introduction

In this lesson, we'll dive more into the concept of **emergent abilities**, the empirical phenomenon of the new abilities that language models get when their size increases over specific thresholds.

Emergent abilities become apparent as we scale up the models and are influenced by factors such as training compute and model parameters.

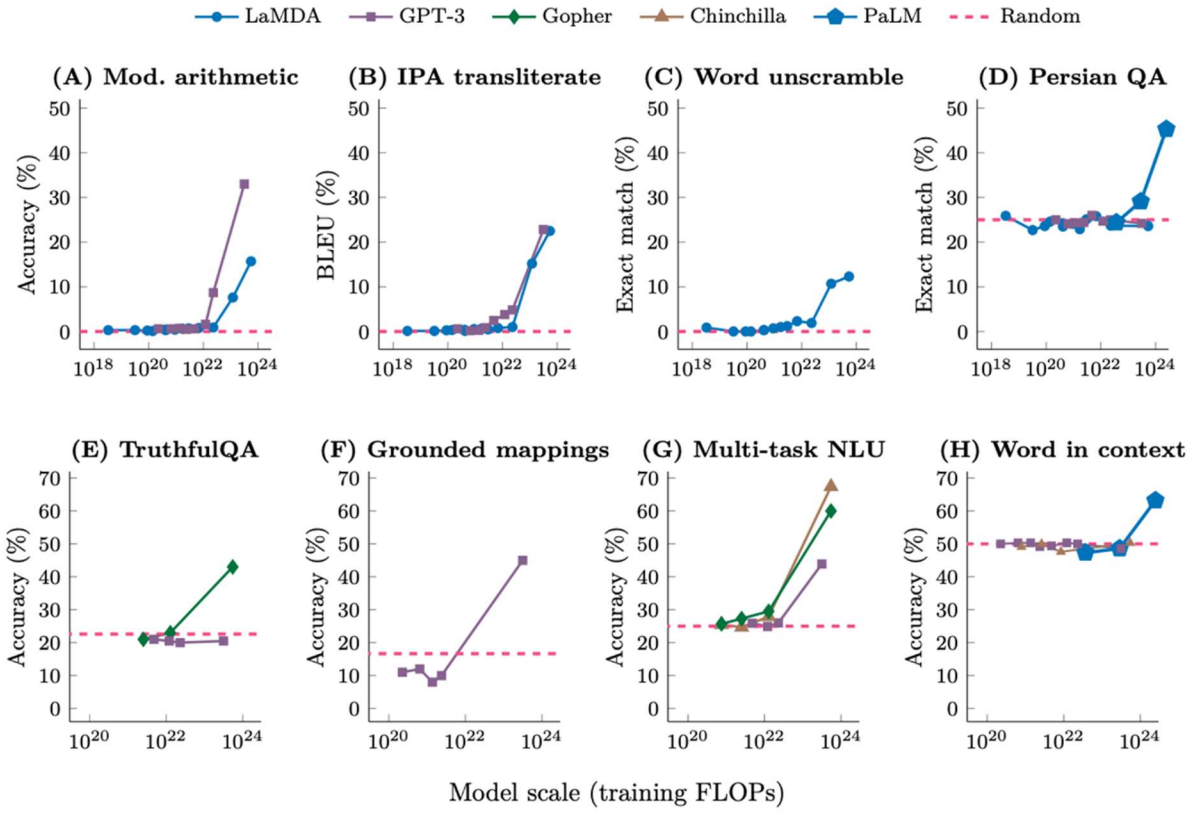
We'll also explore various instances of these emergent abilities, focusing on scenarios like few-shots and augmented prompting, and examine the reasons behind the emergence of these abilities and whether further scaling could reveal more of them.

What Are Emergent Abilities

Emergent abilities in LLMs are defined as significant improvements in task performance that become apparent as the model size or scale increases. These abilities, which are not present or noticeable in smaller or less complex models, become evident in larger or more complex models. This suggests that the model is learning and generalizing from its pre-training in ways that were not explicitly programmed or expected.

When visualized on a scaling curve, emergent abilities show a pattern where performance is almost random until a certain scale threshold, after which performance increases significantly. This is known as a phase transition, a dramatic change in behavior that could not have been predicted by examining smaller-scale systems.

In the following image, taken from the paper "Emergent Abilities of Large Language Models," we see several charts showing the emergence of abilities of LLMs (whose performance is shown on the y-axis) with respect to the model scale (shown on the x-axis).



From the paper "Emergent Abilities of Large Language Models"

Language models have been scaled primarily along computation amount, model parameters, and training dataset size. The emergence of abilities may occur with less training computation or fewer model parameters for models trained on higher-quality data.

It also depends on factors such as the amount of data, its quality, and the number of parameters in the model.
Emergent abilities in LLMs appear as the models scale up and cannot be predicted by simply extrapolating from smaller models.

Evaluation Benchmarks for Emergent Abilities

Several benchmarks are used to evaluate the emergent abilities of language models. These include the BIG-Bench suite, TruthfulQA, the Massive Multi-task Language Understanding (MMLU) benchmark, and the Word in Context (WiC) benchmark.

1. The first of these is the **BIG-Bench suite**, a comprehensive set of over 200 benchmarks that test a model's capabilities across a variety of tasks. These tasks include **arithmetic operations** where the model is expected to perform the four basic operations (example: "Q: What is 132 plus 762? A: 894), transliteration from the International Phonetic Alphabet (IPA) to measure if the model is able to manipulate and use rare words (example: "English: The 1931 Malay census was an alarm bell. IPA: ðə 1931 ˈmeɪleɪ ˈsɛnsəs wɑz ən əˈlɑrm bɛl."), word unscrambling that analyzes the model's ability to work with alphabets. A large number of benchmarks can be found within the Github repository where you can delve into their specific details. The performance of models like GPT-3 and LaMDA on these tasks starts near zero but jumps to significantly above random at a certain scale, demonstrating emergent abilities.
2. Another benchmark is **TruthfulQA**, which measures a model's capacity to provide truthful responses when addressing questions. The evaluation consists of two tasks: 1) Generation: The model will be asked to answer a question with 1 or 2 sentences. 2) Multiple-choices: The second task involves multiple-choice questions, where the model must choose the correct answer from either 4 options or True/False statements. When the Gopher model is scaled up to its largest size, its performance jumps to more than 20% above random, indicating the emergence of this ability.
3. **The Massive Multi-task Language Understanding (MMLU)** is another key benchmark. The primary objective of this benchmark is to evaluate models for their ability to demonstrate a broad range of world knowledge and problem-solving skills. The test encompasses 57 tasks, spanning areas such as elementary mathematics, US history, computer science, law, and more. GPTs, Gopher, and Chinchilla models of a specific scale do not perform better than guessing on average of all the topics, but scaling up to a larger size enables performance to surpass random, indicating the emergence of this ability.
4. Finally, the **Word in Context (WiC)** is a semantic understanding benchmark. WiC is a binary classification task for context-sensitive word embeddings. It involves target words (verbs or nouns) with two provided contexts, aiming to determine if they share the same meaning. Chinchilla fails to achieve the one-shot performance of better than random, even when scaled to its largest model size. Above-random performance eventually emerged when PaLM was scaled to a much larger size, suggesting the emergence of this ability at a larger scale.

Other Factors That Could Give Rise To Emergent Abilities

- Multi-step reasoning is a strategy where a model is guided to produce a sequence of intermediate steps before giving the final answer. This strategy, known as **chain-of-thought prompting**, only surpasses standard prompting when applied to a sufficiently large model.
- **Instruction following** is another strategy that involves fine-tuning a model on a mixture of tasks phrased as instructions. This strategy only improves performance when applied to a model of a specific size.

Risks With Emergent Abilities

As we scale up language models, we also need to be aware of the emergent risks that come with it. These risks could be societal issues related to truthfulness, bias, and toxicity. These risks can be avoided by applying strategies, such as giving model prompts that encourage them to be "helpful, harmless, and honest."

The WinoGender benchmark, which measures gender bias in occupations, has shown that scaling can improve performance but also increase bias in ambiguous contexts. Larger models were found to be more likely to memorize training data, although deduplication methods can reduce this risk.

Emergent risks also include phenomena that might only exist in future language models or that have not yet been characterized in current models. These could include backdoor vulnerabilities or harmful content synthesis.

A Shift Towards General-Purpose Models

The emergence of abilities has led to sociological changes in how the community views and uses these models. Historically, NLP focused on task-specific models. Scaling models has led to an explosion in research on "general purpose" models that aim to perform a range of tasks not explicitly encoded in the training data.

This shift towards general-purpose models is evident when scaling enables a few-shot prompted general-purpose model to outperform prior state-of-the-art held by fine-tuned task-specific models. For example, GPT-3 achieved a new state-of-the-art on the TriviaQA and PiQA question-answering benchmarks; PaLM achieved a new state-of-the-art on three arithmetic reasoning benchmarks; and the multimodal Flamingo model achieved a new state of the art on six visual question answering benchmarks.

The ability of general-purpose models to perform unseen tasks, given only a few examples, has also led to many new applications of language models outside the NLP research community. For instance, language models have been used by prompting to translate natural language instructions into actions that are executable by robots, interact with users, and facilitate multi-modal reasoning.

Conclusion

Emergent abilities in LLMs are capabilities that appear as the models scale up and are a key factor in their success. These abilities, unpredictable from smaller models, become evident after reaching a certain scale threshold. They have been observed in various contexts, such as in a few-shot prompting and augmented prompting strategies. Scaling up LLMs also introduces emergent risks like increased bias and toxicity, which can be avoided with appropriate strategies. The emergence of these abilities has led to a shift towards general-

purpose models and opened up new applications outside the traditional NLP research community.
In the next lesson, we'll dive into today's most popular proprietary LLMs and describe the tradeoffs between proprietary and open-source LLMs.

Emergent Abilities in LLMs

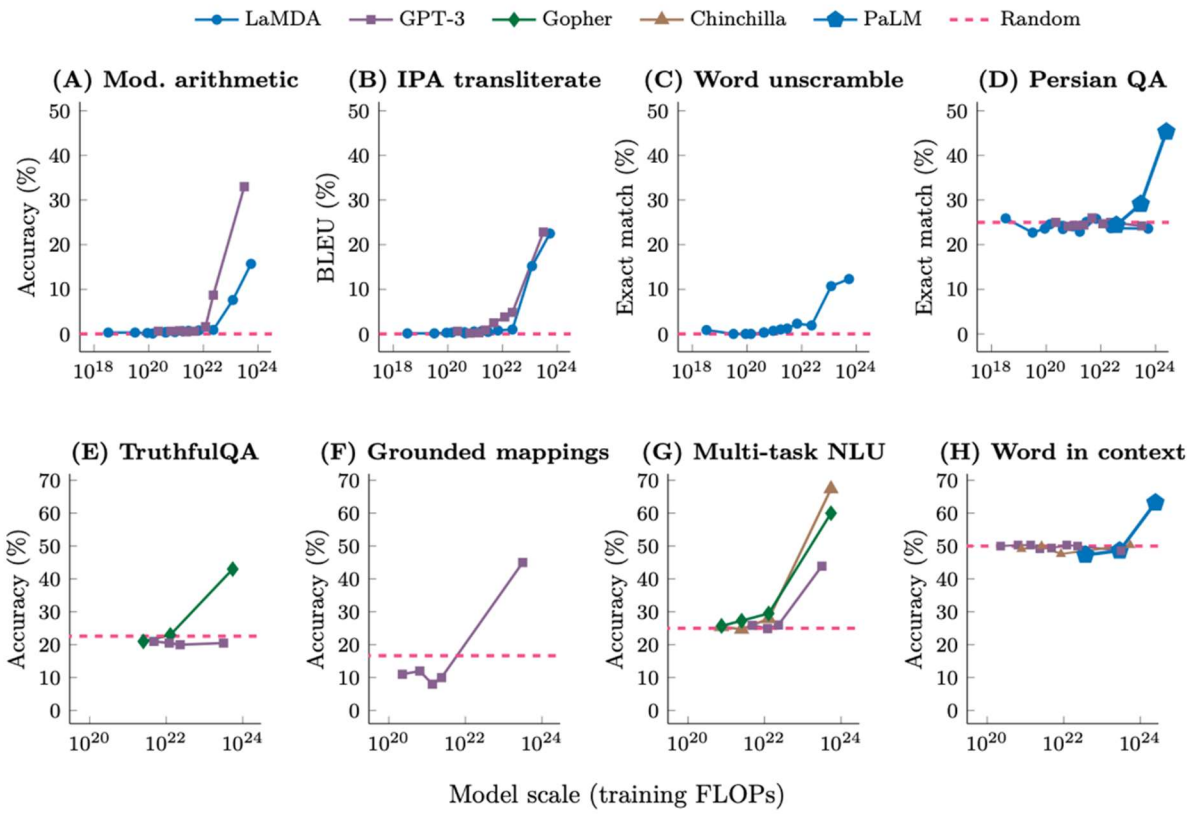
An ability is considered as emergent when larger models exhibit it, but it's absent in smaller models - a key factor contributing to the success of Large Language Models.

Introduction

In this lesson, we'll dive more into the concept of **emergent abilities**, the empirical phenomenon of the new abilities that language models get when their size increases over specific thresholds.
Emergent abilities become apparent as we scale up the models and are influenced by factors such as training compute and model parameters.
We'll also explore various instances of these emergent abilities, focusing on scenarios like few-shots and augmented prompting, and examine the reasons behind the emergence of these abilities and whether further scaling could reveal more of them.

What Are Emergent Abilities

Emergent abilities in LLMs are defined as significant improvements in task performance that become apparent as the model size or scale increases. These abilities, which are not present or noticeable in smaller or less complex models, become evident in larger or more complex models. This suggests that the model is learning and generalizing from its pre-training in ways that were not explicitly programmed or expected.
When visualized on a scaling curve, emergent abilities show a pattern where performance is almost random until a certain scale threshold, after which performance increases significantly. This is known as a phase transition, a dramatic change in behavior that could not have been predicted by examining smaller-scale systems.
In the following image, taken from the paper "Emergent Abilities of Large Language Models," we see several charts showing the emergence of abilities of LLMs (whose performance is shown on the y-axis) with respect to the model scale (shown on the x-axis).



From the paper "Emergent Abilities of Large Language Models"

Language models have been scaled primarily along computation amount, model parameters, and training dataset size. The emergence of abilities may occur with less training computation or fewer model parameters for models trained on higher-quality data. It also depends on factors such as the amount of data, its quality, and the number of parameters in the model.
Emergent abilities in LLMs appear as the models scale up and cannot be predicted by simply extrapolating from smaller models.

Evaluation Benchmarks for Emergent Abilities

Several benchmarks are used to evaluate the emergent abilities of language models. These include the BIG-Bench suite, TruthfulQA, the Massive Multi-task Language Understanding (MMLU) benchmark, and the Word in Context (WiC) benchmark.

1. The first of these is the **BIG-Bench suite**, a comprehensive set of over 200 benchmarks that test a model's capabilities across a variety of tasks. These tasks include **arithmetic operations** where the model is expected to perform the four basic operations (example: "Q: What is 132 plus 762? A: 894), transliteration from the International Phonetic Alphabet (IPA) to measure if the model is able to manipulate and use rare words (example: "English: The 1931 Malay census was an alarm bell. IPA: ðə 1931 ˈmeɪleɪ ˈsɛnsəs wɑz ən əˈlɑrm bɛl."), word unscrambling that analyzes the model's ability to work with alphabets. A large number of benchmarks can be found within the Github repository where you can delve into their specific details. The

performance of models like GPT-3 and LaMDA on these tasks starts near zero but jumps to significantly above random at a certain scale, demonstrating emergent abilities.

2. Another benchmark is **TruthfulQA**, which measures a model's capacity to provide truthful responses when addressing questions. The evaluation consists of two tasks: 1) Generation: The model will be asked to answer a question with 1 or 2 sentences. 2) Multiple-choices: The second task involves multiple-choice questions, where the model must choose the correct answer from either 4 options or True/False statements. When the Gopher model is scaled up to its largest size, its performance jumps to more than 20% above random, indicating the emergence of this ability.
3. **The Massive Multi-task Language Understanding (MMLU)** is another key benchmark. The primary objective of this benchmark is to evaluate models for their ability to demonstrate a broad range of world knowledge and problem-solving skills. The test encompasses 57 tasks, spanning areas such as elementary mathematics, US history, computer science, law, and more. GPTs, Gopher, and Chinchilla models of a specific scale do not perform better than guessing on average of all the topics, but scaling up to a larger size enables performance to surpass random, indicating the emergence of this ability.
4. Finally, the **Word in Context (WiC)** is a semantic understanding benchmark. WiC is a binary classification task for context-sensitive word embeddings. It involves target words (verbs or nouns) with two provided contexts, aiming to determine if they share the same meaning. Chinchilla fails to achieve the one-shot performance of better than random, even when scaled to its largest model size. Above-random performance eventually emerged when PaLM was scaled to a much larger size, suggesting the emergence of this ability at a larger scale.

Other Factors That Could Give Rise To Emergent Abilities

- Multi-step reasoning is a strategy where a model is guided to produce a sequence of intermediate steps before giving the final answer. This strategy, known as **chain-of-thought prompting**, only surpasses standard prompting when applied to a sufficiently large model.
- **Instruction following** is another strategy that involves fine-tuning a model on a mixture of tasks phrased as instructions. This strategy only improves performance when applied to a model of a specific size.

Risks With Emergent Abilities

As we scale up language models, we also need to be aware of the emergent risks that come with it. These risks could be societal issues related to truthfulness, bias, and toxicity. These risks can be avoided by applying strategies, such as giving model prompts that encourage them to be "helpful, harmless, and honest."

The WinoGender benchmark, which measures gender bias in occupations, has shown that scaling can improve performance but also increase bias in ambiguous contexts. Larger models were found to be more likely to memorize training data, although deduplication methods can reduce this risk.

Emergent risks also include phenomena that might only exist in future language models or that have not yet been characterized in current models. These could include backdoor vulnerabilities or harmful content synthesis.

A Shift Towards General-Purpose Models

The emergence of abilities has led to sociological changes in how the community views and uses these models. Historically, NLP focused on task-specific models. Scaling models has led to an explosion in research on "general purpose" models that aim to perform a range of tasks not explicitly encoded in the training data.

This shift towards general-purpose models is evident when scaling enables a few-shot prompted general-purpose model to outperform prior state-of-the-art held by fine-tuned task-specific models. For example, GPT-3 achieved a new state-of-the-art on the TriviaQA and PiQA question-answering benchmarks; PaLM achieved a new state-of-the-art on three arithmetic reasoning benchmarks; and the multimodal Flamingo model achieved a new state of the art on six visual question answering benchmarks.

The ability of general-purpose models to perform unseen tasks, given only a few examples, has also led to many new applications of language models outside the NLP research community. For instance, language models have been used by prompting to translate natural language instructions into actions that are executable by robots, interact with users, and facilitate multi-modal reasoning.

Conclusion

Emergent abilities in LLMs are capabilities that appear as the models scale up and are a key factor in their success. These abilities, unpredictable from smaller models, become evident after reaching a certain scale threshold. They have been observed in various contexts, such as in a few-shot prompting and augmented prompting strategies. Scaling up LLMs also introduces emergent risks like increased bias and toxicity, which can be avoided with appropriate strategies. The emergence of these abilities has led to a shift towards general-purpose models and opened up new applications outside the traditional NLP research community.

In the next lesson, we'll dive into today's most popular proprietary LLMs and describe the tradeoffs between proprietary and open-source LLMs.

Open-Source LLMs

Introduction

In this lesson, we will discuss several open-source LLMs and their features, capabilities, and licenses. This overview will cover LLaMA 2, Open Assistant, Dolly (by Databricks), and Falcon as the most used LLMs. We will also explore the licenses and potential commercial usage of these models. Additionally, we will discuss limitations or restrictions that may be present in their licenses.

LLaMA 2

LLaMA 2 is a cutting-edge large language model developed by Meta, released on July 18, 2023, with an open license for both research and commercial use.

The architecture of LLaMA 2 is described in great detail in the 77-page paper, making it easier for data scientists to recreate and fine-tune the models for their specific needs. The model's training data comprises an impressive 2 trillion tokens. It has been trained on a massive

scale, outperforming all open-source benchmarks and demonstrating performance comparable to GPT3.5 in terms of human evaluation.
LLaMA 2 is available in three parameter variations: 7B, 13B, and 70B, and there are also instruction-tuned versions known as LLaMA-Chat.
The fine-tuning process is done through Supervised Fine-Tuning (SFT) and Reinforcement Learning with Human Feedback (RLHF), using a novel approach to segment data based on helpfulness and safety prompts.
The reward models are crucial to LLaMA 2's performance, allowing it to balance safety and helpfulness effectively. The safety reward model and helpfulness reward model are trained to evaluate the quality of generated responses.
The impact of LLaMA 2 in Generative AI is substantial, outperforming other open innovation models like Falcon or Vicuna.
You can find the LLaMA 2 models on the Hugging Face Hub here. Here, we test the meta-llama/Llama-2-7b-chat-hf model. For this, you'll first have to request access to the model on this page.
First, let's download the model. It takes some time as the model weighs about 14GB.
Copy

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch


# download model
model_id = "meta-llama/Llama-2-7b-chat-hf"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
        model_id,
        trust_remote_code=True,
        torch_dtype=torch.bfloat16
)
```

Then, we generate a completion with it. This step will take a lot of time if you're generating text using CPUs instead of GPUs!
Copy

```
# generate answer
prompt = "Translate English to French: Configuration files are easy to use!"
inputs = tokenizer(prompt, return_tensors="pt", return_token_type_ids=False)
outputs = model.generate(**inputs, max_new_tokens=100)
```

```
# print answer
print(tokenizer.batch_decode(outputs, skip_special_tokens=True)[0])
```

Falcon

The Falcon models, developed and trained by the Technology Innovation Institute (TII) of Abu Dhabi, have gained significant attention since their release in May 2023. These models are causal large language models (LLM), similar to GPT, and are also known as "decoder-only" models. They excel in predicting the next token in a sequence of tokens with their attention focused solely on the left context during training, while the right context remains masked.

The Falcon models are distributed under the Apache 2.0 License, allowing even commercial use.

The largest of these models, Falcon-40B, has shown great performance, outperforming other causal LLMs like LLaMa-65B and MPT-7B. Falcon-7B, a slightly smaller version, was designed to be fine-tuned on consumer hardware and has half the number of layers and embedding dimensions compared to Falcon-40B.

The training data for Falcon models primarily comes from the "Falcon RefinedWeb dataset," which is meticulously curated and multimodal-friendly, preserving links and alt texts of images. This dataset and curated corpora make up 75% of the pre-training data for the Falcon models. While it primarily covers English, additional versions like "RefinedWeb-Europe" have been prepared to include several European languages.

The instruct versions of Falcon-40B and Falcon-7B perform even better, with fine-tuning done on a mixture of chat/instruct datasets sourced from various places, including GPT4all and GPTeacher.

You can find the Falcon models on the Hugging Face Hub here. Here, we test the tiiuae/falcon-7b-instruct model. You can use the same code previously used for the LLaMA example by changing the model_id.

Copy

```
model_id = "tiiuae/falcon-7b-instruct"
```

Dolly

Dolly is an open-source LLM introduced by Databricks. It was first unveiled as Dolly 1.0, a language model that showcased ChatGPT-like human interactivity. The team has now released Dolly 2.0, a better instruction-following LLM.

One of the critical features of Dolly 2.0 is that it is built on a new, high-quality human-generated instruction dataset called "databricks-dolly-15k". This dataset consists of 15,000 prompt/response pairs designed explicitly for instruction tuning large language models. Unlike many instruction-following models, Dolly 2.0's dataset is open-source and licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. This means that anyone can use, modify, or extend the dataset for any purpose, including commercial applications.

The Dolly 2.0 model is based on the EleutherAI Pythia-12 b architecture, comprising 12 billion parameters, which makes it capable of high-quality instruction following behavior. Despite being smaller than some other models, such as Alpaca, Dolly 2.0 has demonstrated great performance due to its reliance on real-world, human-generated training records rather than synthesized data.
You can find the Databricks models on the Hugging Face Hub here. Here, we test the databricks/dolly-v2-3b model. You can use the same code previously used for the LLaMA example by changing the model_id.
Copy

```
model_id = "databricks/dolly-v2-3b"
```

Open Assistant

The Open Assistant project is an initiative aiming to make high-quality large language models accessible to everyone through an open-source and collaborative approach. Unlike some other ChatGPT open-source alternatives with restricted licenses, Open Assistant seeks to provide a versatile chat-based language model comparable to ChatGPT and GPT-4 that can be used for commercial purposes.
The heart of the project lies in its commitment to openness and inclusivity. They have collected a substantial dataset from over 13,000 volunteers, comprising more than 600,000 interactions, 150,000 messages, and 10,000 fully annotated conversation trees on various topics and in multiple languages. This dataset serves as the foundation for training various models hosted on platforms like Hugging Face.
Users can explore the potential of Open Assistant by interacting with the model through the Hugging Face demo or the official chat interface, both designed to solicit user feedback to help improve the chatbot's responses. The project encourages community involvement and contributions, allowing users to participate in data collection and ranking tasks to enhance the capabilities of the language model.
As with most open-source large language models, Open Assistant does have some limitations, particularly in answering math and coding questions, as they are trained on fewer interactions in these domains. However, the model is generally adept at generating interesting and human-like responses, though occasional inaccuracies may occur.

Mistral

Mistral, in September 2023, has released their language model Mistral 7B under the Apache 2.0 license. This model, with 7.3 billion parameters, has shown superior performance compared to the Llama 2 13B and Llama 1 34B models on all and many benchmarks respectively. It also approaches the performance of CodeLlama 7B on code while maintaining proficiency in English tasks.
Mistral 7B uses Grouped-query attention (GQA) for faster inference and Sliding Window Attention (SWA) to handle longer sequences more cost-effectively. This, along with modifications to FlashAttention and xFormers, has led to a 2x speed improvement for sequence lengths of 16k with a window of 4k.

The model can be downloaded and used anywhere, including locally, with the team's reference implementation. It can also be deployed on any cloud (AWS/GCP/Azure) using the vLLM inference server, or used on HuggingFace.
Mistral 7B is easily fine-tuned for any task. As a demonstration, the team has provided a model fine-tuned for chat, which outperforms the Llama 2 13B chat model. The fine-tuned model, Mistral 7B Instruct, outperforms all 7B models on MT-Bench and is comparable to 13B chat models.

The Hugging Face Open LLM Leaderboard:

Hugging Face hosts an **LLM leaderboard**. This leaderboard is created by evaluating community-submitted models on text generation benchmarks on Hugging Face's clusters. It's an excellent resource for checking the new best-performant open-source LLMs.
If you can't find the language or domain you're looking for, you can filter them out and find the one that meets your specific requirements.

Conclusion:

In this lesson, we explored several open-source LLMs) and their features, capabilities, and licenses. We discussed LLaMA 2, Falcon, Dolly, and Open Assistant as some of the most prominent open-source LLMs available.

- LLaMA 2, developed by Meta, is a cutting-edge language model with impressive performance and is available in various parameter variations. It has been trained on a massive scale and demonstrates remarkable performance comparable to GPT3.5.
- Falcon models, developed and trained by the Technology Innovation Institute (TII) of Abu Dhabi, have gained attention for their decoder-only approach and have shown great performance, especially the Falcon-40B model.
- Dolly, introduced by Databricks, is an open-source LLM with a focus on instruction following. It has a high-quality human-generated instruction dataset and is licensed under Creative Commons, allowing for versatile use, including commercial applications.
- Open Assistant is an ambitious project aiming to make high-quality LLMs accessible to everyone through openness and inclusivity. It encourages community involvement and contributions to enhance the capabilities of the language model.

It is essential to acknowledge the importance of open-source LLMs in advancing the field of natural language processing and enabling wider access to state-of-the-art language models for research and commercial purposes.
In the next lesson, we will explore an equally important aspect of LLMs - hallucinations and bias. Hallucinations refer to the generation of fake or incorrect information by LLMs, while bias entails the perpetuation of prejudiced or discriminatory content. Understanding and

addressing these challenges are crucial to ensuring the responsible and ethical use of large language models in various applications.

Understanding Hallucinations and Bias

Introduction

In this lesson, we'll cover the concept of **hallucinations** in LLMs, highlighting their influence on AI applications and demonstrating how to mitigate them using techniques like the retriever's architectures. We'll also explore **bias** within LLMs with examples.

Hallucinations in LLMs

In Large Language Models, hallucinations refer to cases where the model produces text that's incorrect and not based on reality. An AI hallucination is a confident response by the model that cannot be grounded in any of its training data.

There are several possible reasons for these types of hallucinations:

- An LLM could be trained on a dataset that doesn't have the knowledge required to answer a question.
- An LLM does not have a reliable way to check the factual accuracy of its responses. It just outputs a response that may be wholly incorrect yet convincing.
- The training dataset used to train the LLM may include fictional content and subjective content, like opinions and beliefs.
- LLMs are not generally optimized to say "I don't know" when they don't have enough information. Therefore, when the LLM has no answer, it generates whatever is the most probable sequence of text as a response. But note that "most probable" does not necessarily mean "true." Hallucination is the result of this misalignment. Note that the goal of language modeling is learning the underlying distribution of the words in a language, not to distinguish what is true and what isn't.

Let's see a clear example of a hallucination.

On April 12th, 2023, Databricks announced their "Dolly 2.0" model, an open-source 12B parameter language model fine-tuned on a human-generated instruction-following dataset. GPT-3.5 has been trained on a dataset with a cutoff year of 2021. Therefore, it can't know about Dolly 2.

When GPT-3.5 is asked: "Does Databricks Dolly provide value to the open source community?" it responds with something like this:

GPT 3.5: *Databricks Dolly is a tool developed by Databricks, which is designed to help users migrate their data from one platform to another, particularly from on-premise data centers to cloud-based platforms such as AWS or Azure. While Databricks Dolly is not an open source tool, it can still provide value to the open source community in a few ways.[…]*

Except for "Databricks Dolly is a tool developed by DataBricks," the rest of the text is entirely incorrect! Dolly is not a tool to migrate data; it is open-source, contrary to the response. Still, GPT-3.5 knows about Databricks as a company, and therefore, it generates a plausible description of Dolly 2 as a generic product of Databricks.

While this example shows OpenAI GPT-3.5 hallucinating, it's important to note that this phenomenon applies to other similar LLMs like Bard or LLama.

Strategies to mitigate hallucinations include tuning the text generation parameters, cleaning up the training data, precisely defining prompts (prompt engineering), and using retriever architectures to ground responses in specific retrieved documents.

Misinformation Spreading

One significant risk associated with hallucinations in LLMs is their potential to generate content that, while appearing credible, is factually incorrect. Due to their limited capacity to understand the context and verify facts, LLMs can unintentionally spread misinformation. There's the potential for individuals with malicious intent to exploit LLMs to spread disinformation deliberately, creating and promoting false narratives. A study by Blackberry found that nearly half of the respondents (49%) believed that GPT-4 could be used to spread misinformation. The unrestricted spread of such false information via LLMs can lead to widespread negative impacts across societal, cultural, economic, and political landscapes. It's crucial to address these issues related to LLM hallucinations to ensure the ethical use of these models.

Tuning the Text Generation Parameters

The generated output of LLMs is greatly influenced by various model parameters, including temperature, frequency penalty, presence penalty, and top-p. We'll learn more about them in a later lesson in the course.

Higher temperature values promote randomness and creativity, while lower values make the output more deterministic. Increasing the frequency penalty value encourages the model to use repeated tokens more conservatively. Similarly, a higher presence penalty value increases the likelihood of generating tokens not yet included in the generated text. The "top-p" parameter controls response diversity by setting a cumulative probability threshold for word selection.

Leveraging External Documents with Retrievers Architectures

Response accuracy can be improved by providing domain-specific knowledge to the LLM in the form of external documents. Augmenting the **knowledge base** with domain-specific information allows the model to ground its responses in the knowledge base. After a question from a user, we could retrieve documents relevant to the questions (leveraging a module called "retriever") and use them in a prompt to produce the answer. This type of process is implemented into architectures typically called "retrievers architectures".

In these architectures:

1. When a user poses a question, the system computes an embedding representation of it.
2. The embedding of the question is then used for executing a **semantic search** in the database of documents (by comparing their embeddings and computing similarity scores).
3. The top-ranked documents are used by the LLM as context to give the final answer. Usually, the LLM is asked to extract the answer from those context passages precisely and not to write anything that can't be inferred from them.

Retrieval-augmented generation (RAG) is a technique that enhances language model capabilities by sourcing data from external resources and integrating it with the context provided in the model's prompt.

Providing access to external data sources during the prediction process enriches the model's knowledge and grounding. By leveraging external knowledge, the model can generate more accurate, contextually appropriate responses and be less prone to hallucination.

Bias in LLMs

Large language models like GPT-3.5 and GPT-4 have raised serious privacy and ethical concerns. Research has shown that these models are prone to inherent bias, leading to the generation of prejudiced or hateful language, intensifying the concerns regarding their use and governance.

Biases in LLMs arise from various sources: the data, the annotation process, the input representations, the models, and the research design.

For instance, training data that don't represent the diversity of language can lead to demographic biases, resulting in a model's inability to understand and accurately represent certain user groups. Misrepresentation can vary from mild inconveniences to more covert, gradual declines in performance, which can unfairly impact certain demographic groups. LLMs can unintentionally intensify harmful biases through their hallucinations, creating prejudiced and offensive content.

The data used to train LLMs frequently includes stereotypes, which the models may unknowingly reinforce. This imbalance can lead the models to generate prejudiced content that discriminates against underrepresented groups, potentially targeting them based on factors like race, gender, religion, and ethnicity.

This can be exemplified when an LLM produces content that presents women as inferior or portrays certain ethnicities as intrinsically violent or unreliable. Also, if a model is trained on data biased towards a younger, technologically savvy demographic, it may generate outputs that overlook older individuals or those from less technologically equipped regions. If the model is steeped in data from sources promoting hate speech or toxic content, it might produce damaging and prejudiced outputs, amplifying the diffusion of harmful stereotypes and biases.

These examples underscore the urgent need for constant monitoring and ethical management in the use of these models.

Constitutional AI

Constitutional AI' is a conceptual framework crafted by researchers at Anthropic. It aims to align AI systems with human values, ensuring that they become beneficial, safe, and trustworthy.

In the beginning, the model is trained to self-review and modify its responses based on a set of predetermined principles and a small set of process examples. The next phase involves reinforcement learning training. At this point, the model leans on AI-generated feedback, grounded in the given principles, as opposed to human feedback, to choose the least harmful response.

Constitutional AI employs methodologies like **self-supervision training**. These techniques allow the AI to learn to conform to its constitution, without the need for explicit human labeling or supervision.

The approach also includes developing constrained optimization techniques. These ensure that the AI pursues helpfulness within the boundaries set by its constitution rather than pursuing unbounded optimization, potentially forgetting helpful knowledge.

Conclusion

The risks of hallucinations and biases in LLMs present significant issues in producing reliable and accurate outputs. The presence of biases can further damage the accuracy and fairness of the outputs, resulting in the ongoing progression of harmful stereotypes and misinformation.

It's imperative to formulate strategies to mitigate these risks. Such strategies should incorporate pre-processing and input control measures, model configuration adjustments, improvement mechanisms, and context and knowledge enhancement techniques.

Integrating the ethical guidelines is essential to ensure that the models generate fair and trustworthy outputs, ultimately achieving responsible use of these powerful technologies.

Applications and Use-Cases of LLMs

Introduction

In this lesson, we will explore the diverse applications and use cases of LLMs and generative AI across various industries.

We dive into how LLMs are revolutionizing healthcare and medical research by improving diagnosis, drug discovery, and patient care. Additionally, we will uncover their impact on finance, copywriting, education, programming, and the legal industry.

While LLMs offer immense potential, we will also address the risks and ethical considerations associated with their deployment in real-world scenarios, emphasizing the importance of responsible AI implementation and human oversight.

Healthcare and Medical Research

Generative AI offers promising applications that can enhance patient care, drug discovery, and operational efficiency in the industry.

Generative AI is being utilized for diagnosis, patient monitoring, and resource optimization. By incorporating large language models into digital pathology, accuracy for detecting diseases such as cancer has improved significantly. Furthermore, the technology aids in automating administrative tasks, which streamlines workflows and allows clinical staff to focus on more critical aspects of patient care.

In the pharmaceutical industry, generative AI has become a game-changer in drug discovery. It accelerates the process and improves precision in medicine therapies, leading to shorter drug development timelines and reduced costs. This advancement paves the way for more personalized treatments and targeted therapies, ultimately benefiting patients.

Medtech companies are exploring the potential of generative AI to create personalized devices for patient-centered care. Integrating generative AI into the design process optimizes medical devices for specific patient needs, improving treatment outcomes and increasing patient satisfaction.

For example, Med-PaLM is a large language model designed by Google to provide high quality answers to medical questions. It functions as a multimodal generative model capable of processing diverse biomedical data such as clinical text, medical images, and genomics, all using the same set of model parameters. Another example is BioMedLM, a domain-specific LLM for biomedical text, made by the Stanford Center for Research on Foundation Models (CRFM) and MosaicML.

Finance

LLMs like GPT have proven to be powerful tools for analyzing and processing financial data, revolutionizing how financial institutions interact with their clients and manage risks.
One of the key applications of LLMs in finance is in customer interactions with digital platforms, where models can be utilized to enhance user experience through chatbots or AI-based apps. These applications enable seamless and efficient customer support, providing real-time responses to queries and concerns.
The analysis of financial time-series data is another area where LLMs and generative AI have proven worthy. By leveraging large datasets of stock exchange information, these models can offer valuable insights for macroeconomic analysis and stock exchange prediction. Predicting market trends and identifying potential investment opportunities are crucial for making informed financial decisions. LLMs play a significant role in this aspect.
For example, Bloomberg trained an LLM on a mix of general purpose and domain specific documents, calling it BloombergGPT. BloombergGPT outperforms similarly-sized open models on financial NLP tasks, without sacrificing performance on general LLM benchmarks.

Copywriting

Large Language Models and generative AI are influencing the field of copywriting by providing powerful tools for creating content.
The applications of generative AI in copywriting are diverse. It can be utilized to speed up the writing process, overcome writer's block, and reduce costs by improving overall productivity. Additionally, generative AI helps maintain a consistent brand image by learning a company's language patterns and style, ensuring cohesive marketing activities.
Some prominent use cases include generating website content and blog posts, crafting social media posts, creating product descriptions, and optimizing content for SEO. Generative AI can also contribute to developing content for mobile apps, tailoring it to suit different platforms and user experiences.
A popular copywriting tool that uses LLMs is Jasper, which makes it easy to generate diverse kinds of content using generative AI.

Education

LLMs can help a lot in online learning and personalized tutoring. By analyzing individual learning progress, LLMs offer personal feedback, adaptive testing, and personalized interventions.
These models can address the challenges of teacher shortages by providing scalable solutions such as virtual teachers or supporting para-teachers with advanced tools. They empower educators to become mentors and guides, offering personalized support and interactive learning experiences.
AI can analyze individual student performance and personalize the learning experience.

For example, an application of LLMs for education is Khanmigo of Khan Academy. LLMs serve as virtual tutors, offering explanations and examples for better subject understanding. LLMs aid language learning, generating sentences for grammar and vocabulary practice.

Programming

LLMs and generative AI can significantly help in coding by providing powerful tools for developers. LLMs like GPT-4 and its predecessors can generate code snippets based on natural language prompts, significantly enhancing programmers' efficiency. These models are trained on vast corpora of code samples and can understand context, enabling them to generate more relevant and accurate code over time.

The applications of LLMs for coding are diverse. They can assist in code completion by suggesting code snippets as developers type, saving time and reducing errors. Additionally, LLMs are employed for unit test generation, automating the creation of test cases. This not only enhances code quality but also assists in software maintenance.

However, the use of generative AI in coding also presents challenges. While it can boost productivity, developers must exercise caution and review the generated code, as it may contain errors or security vulnerabilities. Furthermore, the potential for model biases and "hallucinations" (fabricating incorrect information) necessitates careful scrutiny.

A popular product using LLMs for programming is GitHub Copilot, which is trained on billions of lines of code. Copilot can turn natural language prompts into coding suggestions across dozens of languages.

Legal Industry

LLMs and generative AI have emerged as powerful tools for the legal industry, offering a range of applications and use cases. These models can be designed to handle the complexities of legal language, interpretations, and the dynamic nature of law. LLMs have the potential to assist legal practitioners in various tasks, such as providing legal advice, understanding complex legal documents, and analyzing court case texts.

One key application is reducing hallucinations, a common challenge with early legal LLMs. These models can produce more accurate and reliable results by integrating domain-specific knowledge through reference modules and reliable data from knowledge bases. They can also identify legal feature words within users' input and quickly analyze legal situations.

Risks and Ethical Considerations of Using LLMs in the Real World

As we learned in previous lessons, deploying Large Language Models (LLMs) in real-world applications poses certain risks and ethical concerns.

One significant risk is "**hallucinations**," where the LLM generates false but plausible-sounding information. This could lead to serious consequences, particularly in critical domains like healthcare, finance, and law.

Another concern is "**bias,**" as LLMs can inadvertently perpetuate societal biases present in their training data. This could result in unfair treatment in areas such as healthcare and finance. Addressing bias requires rigorous data evaluation, inclusivity efforts, and continuous improvement in fairness.

Data privacy and security are crucial as LLMs might memorize sensitive information, potentially leading to privacy breaches. Organizations must implement measures like data anonymization and strict access controls.

Additionally, the impact on employment should be considered, balancing automation and human involvement to preserve human expertise. Dependence on LLMs without human judgment can be risky, necessitating a responsible approach that combines AI benefits with human oversight.

Conclusion

This lesson explored the wide-ranging applications and use cases of LLMs and generative AI across diverse industries. From healthcare and medical research to finance, copywriting, education, programming, and the legal industry, LLMs are powerful tools with immense potential.

However, alongside their benefits, we must be mindful of the risks and ethical considerations associated with their deployment. Addressing issues like hallucinations, bias, data privacy, and the impact on human employment is crucial for responsible AI implementation.

In the next module, we will study Transformer architectures, the backbone of LLMs, to gain insights into their working principles and understand how these models process and generate language with great accuracy and efficiency.

Understanding Transformers and GPTs Introduction

Understanding Transformers and GPTs

Goals: Equip students with foundational theoretical knowledge of transformers and GPTs, ensuring a robust understanding beneficial for effective LLM training and utilization.

This section comprehensively examines the inner workings and components surrounding Transformers and GPT. Participants begin with a detailed study of the Transformer architecture, understanding its foundational concepts and essential parts. The progression covers evaluations of LLMs, their control mechanisms, and the nuances of prompting, pretraining, and finetuning. Each lesson is designed to impart intricate details, practical knowledge, and a tiered understanding of these technologies.

- **Understanding Transformers**: This lesson provides an in-depth look at Transformers, breaking down their complex components and the network's essential mechanics. We begin by examining the paper "Attention is all you need." We conclude by highlighting the use of these components in Hugging Face's transformers library.
- **Transformers Architectures**: This chapter is a concise guide to Transformer architectures. We will first dissect the encoder-decoder framework, which is pivotal for sequence-to-sequence tasks. Next, we provide a high-level overview of the GPT model, known for its language generation capabilities. We also spotlight BERT, emphasizing its significance in understanding the context within textual data.
- **Deep Dive on the GPT architecture**: This section explores the GPT architecture. We shed light on the structural specifics, the objective function, and the principles of causal modeling. This technical session is designed for individuals seeking an in-depth understanding of the intricate details and mathematical foundations of GPT.
- **Evaluating LLM Performance**: This lesson explores the nuances of evaluating Large Language Model performance. We differentiate between objective functions and

metrics and transition into perplexity, BLEU, and ROUGE metrics. We also provide an overview of popular benchmarks in the domain.

- **Controlling LLM Outputs**: This lesson delves into decoding techniques like Greedy and Beam Search, followed by concepts such as Temperature and the use of stop sequences. We will also discuss the importance of these methods, with references to frameworks like ReAct. It also presents concepts like Frequency and Presence Penalties.
- **Prompting and few-shot prompting**: This lesson provides an overview of how carefully crafted prompts can guide LLMs in tasks like answering questions and generating text. We will progress from zero-shot prompting, where LLMs operate without specific examples, to in-context and few-shot prompting, teaching the model to manage intricate tasks with sparse training data.
- **Pretraining and Finetuning**: This module examines the foundational concepts of pretraining and finetuning in the context of Large Language Models. In subsequent chapters, we will discern the differences between pretraining, finetuning, and instruction tuning, setting the stage for deeper dives. While the lesson touches upon various types of instruction tuning, detailed exploration of specific methods like SFT and RLHF will be reserved for later sessions, ensuring a progressive understanding of the topic.

After navigating the diverse terrain of Transformers and LLMs, participants now deeply understand significant architectures like GPT and BERT. The sessions shed light on model evaluation metrics, advanced control techniques for optimal outputs, and the roles of pretraining and finetuning. The upcoming module dives into the complexities of deciding when to train an LLM from scratch, the operational necessities of LLMs, and the sequential steps crucial for the training process.

Understanding Transformers

Introduction

In this lesson, we will dive deeper into Transformers and provide a comprehensive understanding of their various components. We will also cover the network's inner mechanisms.

We will look into the seminal paper "Attention is all you need" and examine a diagram of the components of a Transformer. Last, we see how Hugging Face uses these components in the popular transformers library.

Attention Is All You Need

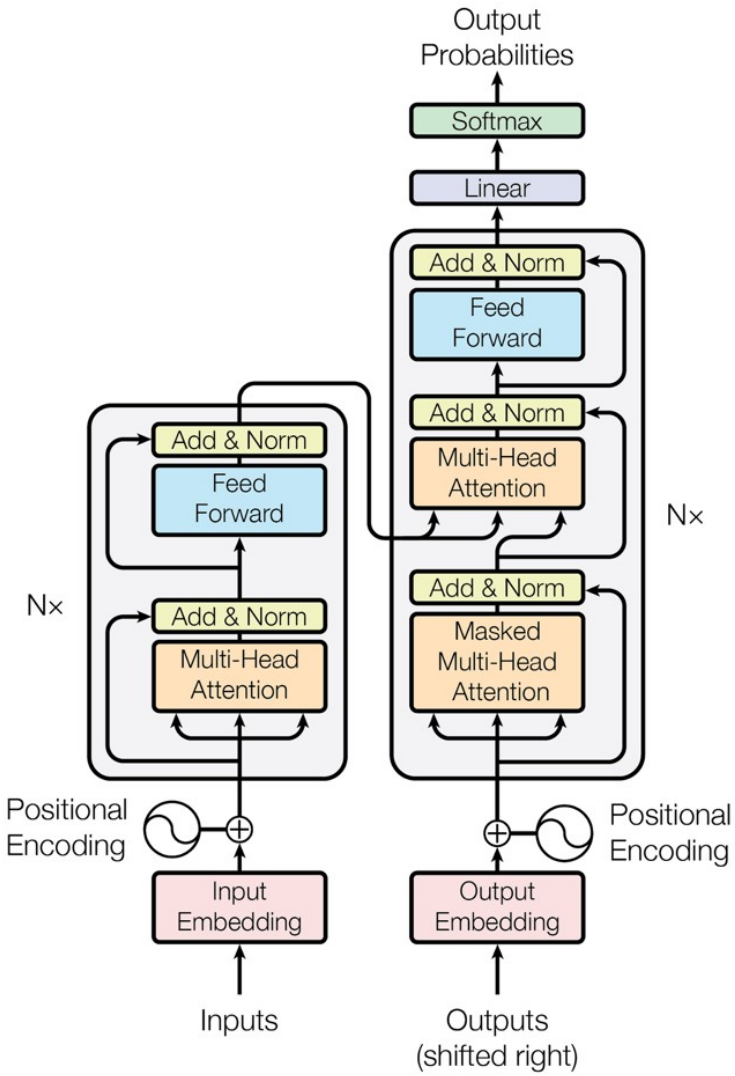
The Transformer architecture was proposed as a collaborative effort between Google Brain and the University of Toronto in a paper called "Attention is All You Need." It presented an encoder-decoder network powered by attention mechanisms for automatic translation tasks, demonstrating superior performance compared to previous benchmarks (WMT 2014 translation tasks) at a fraction of the cost. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

While Transformers have proven to be highly effective in various tasks such as classification, summarization, and, more recently, language generation, their proposal of training highly parallelized networks is equally significant.
The expansion of the architecture into three distinct categories allowed for greater flexibility and specialization in handling different tasks:

- The **encoder-only** category focused on extracting meaningful representations from input data. An example model of this category is BERT.
- The **encoder-decoder** category enabled sequence-to-sequence tasks such as translation and summarization or training multimodal models like caption generators. An example model of this category is BART.
- The **decoder-only** category specializes in generating outputs based on given instructions, as we have in Large Language Models. An example model of this category is GPT.

The Architecture

Now, let's examine the crucial elements of the Transformer model in more detail.



The overview of Transformer architecture. The left component is called the encoder, which is connected to the decoder using a cross-attention mechanism. (Image taken from the "*Attention is all you need" paper*)

Input Embedding

The initial procedure involves translating the input tokens into embeddings. These embeddings are acquired vectors symbolizing the input tokens, facilitating the model's ability to grasp the semantic meanings of the words. The size of the embedding vector varied based on the model's scale and design preferences. For instance, OpenAI's GPT-3 uses a 12,000-dimensional embedding vector, while smaller models like BERT could have a size as small as 768.

Positional Encoding

Given that the Transformer lacks the recurrence feature found in RNNs to feed the input one at a time, it necessitates a method for considering the position of words within a sentence.

This is accomplished by adding positional encodings to the input embeddings. These encodings are vectors that keep the location of a word in the sentence.

Self-Attention Mechanism

At the core of the Transformer model lies the self-attention mechanism, which calculates a weighted sum of the embeddings of all words in a sentence for each word. These weights are determined based on some learned "attention" scores between words. The terms with higher relevance to one another will receive higher "attention" weights.

Based on the inputs, this is implemented using Query, Key, and Value vectors. Here is a brief description of each vector.

- **Query Vector**: It represents the word or token for which the attention weights are being calculated. The Query vector determines which parts of the input sequence should receive more attention. Multiplying word embeddings with the Query vector is like asking, "What should I pay attention to?"
- **Key Vector**: It represents the set of words or tokens in the input sequence that are compared with the Query. The Key vector helps identify the relevant or essential information in the input sequence. Multiplying word embeddings with the Key vector is like asking, "What is important to consider?"
- **Value Vector**: It contains the input sequence's associated information or features for each word or token. The Value vector provides the actual data that will be weighted and combined based on the attention weights calculated between the Query and Key. The Value vector answers the question, "What information do we have?"

Before the advent of the transformer architecture, the attention mechanism was mainly utilized to compare two portions of texts. For example, the model could focus on different parts of the input article while generating the summary for a task like summarization.

The self-attention mechanism enabled the models to highlight the important parts of the content for the task. It is helpful in encoder-only or decoder-only models to create a powerful representation of the input. The text can be transformed into embeddings for encoder-only scenarios, whereas the text is generated for decoder-only models.

The effectiveness of the attention mechanism significantly increases when applied in a multi-head setting. In this configuration, multiple attention components process the same information, with each head learning to focus on distinct aspects of the text, such as verbs, nouns, numbers, and more, throughout the training process.

The Architecture In Action

This section will demonstrate the functioning of the above components from a pre-trained large language model, providing an insight into their inner workings using the transformers Hugging Face library.

To begin, we load the model and tokenizer using AutoModelForCausalLM and AutoTokenizer, respectively. Then, we proceed to tokenize a sample phrase, which will serve as our input in the following steps.

Copy

```
from transformers import AutoModelForCausalLM, AutoTokenizer


OPT = AutoModelForCausalLM.from_pretrained("facebook/opt-1.3b", load_in_8bit=True)
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")


inp = "The quick brown fox jumps over the lazy dog"
inp_tokenized = tokenizer(inp, return_tensors="pt")
print(inp_tokenized['input_ids'].size())
print(inp_tokenized)Copy
```

The sample code.

Copy

```
torch.Size([1, 10])
{'input_ids': tensor([[    2,   133,  2119,  6219, 23602, 13855,    81,     5, 22414,  2335]]),
'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}Copy
```

The output.

We load Facebook's Open Pre-trained Transformer model with 1.3B parameters (facebook/opt-1.3b) in the 8-bit format, a memory-saving approach to efficiently utilize GPU resources. The tokenizer object loads the required vocabulary to interact with the model and will be used to convert the sample input (inp variable) to the token IDs and attention mask.
Let's look at the model's architecture by accessing its .model method.
Copy

```
print(OPT.model)Copy
```

The sample code.

Copy

```
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): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
     (v_proj): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
     (q_proj): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
     (out_proj): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
    )
    (activation_fn): ReLU()
    (self_attn_layer_norm): LayerNorm((2048,), eps=1e-05, elementwise_affine=True)
    (fc1): Linear8bitLt(in_features=2048, out_features=8192, bias=True)
    (fc2): Linear8bitLt(in_features=8192, out_features=2048, bias=True)
    (final_layer_norm): LayerNorm((2048,), eps=1e-05, elementwise_affine=True)
   )
  )
 )
)Copy
```

The output.

The model is decoder-only, a common characteristic among transformer-based language models. Consequently, we must utilize the decoder key to access its inner components. Furthermore, the examination of the layers key reveals that the decoder component is composed of 24 stacked layers with the same architecture. To begin, we look at the embedding layer.

Copy

```
embedded_input = OPT.model.decoder.embed_tokens(inp_tokenized['input_ids'])
print("Layer:\t", OPT.model.decoder.embed_tokens)
print("Size:\t", embedded_input.size())
print("Output:\t", embedded_input)Copy
```

The sample code.

Copy

```
Layer:    Embedding(50272, 2048, padding_idx=1)
Size:      torch.Size([1, 10, 2048])
Output:  tensor([[[-0.0407,  0.0519,  0.0574,  ..., -0.0263, -0.0355, -0.0260],
       [-0.0371,  0.0220, -0.0096,  ...,  0.0265, -0.0166, -0.0030],
       [-0.0455, -0.0236, -0.0121,  ...,  0.0043, -0.0166,  0.0193],
       ...,
       [ 0.0007,  0.0267,  0.0257,  ...,  0.0622,  0.0421,  0.0279],
       [-0.0126,  0.0347, -0.0352,  ..., -0.0393, -0.0396, -0.0102],
       [-0.0115,  0.0319,  0.0274,  ..., -0.0472, -0.0059,  0.0341]]],
     device='cuda:0', dtype=torch.float16, grad_fn=<EmbeddingBackward0>)Copy
```

The output.

The embedding layer is accessible through the .embed_tokens method under the decoder component and passes our tokenized inputs to the layer. As you can see, the embedding layer will transform a list of IDs with [1, 10] size to [1, 10, 2048]. This representation will then be used and passed through the decoder layers.
Subsequently, the positional encoding component utilizes the attention masks to generate a vector that imparts a sense of positioning within the model. The following code uses the .embed_positions method from the decoder to generate the positional embeddings. As seen, the layer generates a distinct vector for each position, which is added to the output of the embedding layer. This process introduces supplementary positional information to the model.
Copy

```
embed_pos_input =
OPT.model.decoder.embed_positions(inp_tokenized['attention_mask'])
print("Layer:\t", OPT.model.decoder.embed_positions)
print("Size:\t", embed_pos_input.size())
print("Output:\t", embed_pos_input)Copy
```

The sample code.

Copy

```
Layer:    OPTLearnedPositionalEmbedding(2050, 2048)
Size:      torch.Size([1, 10, 2048])
Output:  tensor([[[-8.1406e-03, -2.6221e-01,  6.0768e-03,  ...,  1.7273e-02,
```

```
       -5.0621e-03, -1.6220e-02],
      [-8.0585e-05,  2.5000e-01, -1.6632e-02,  ..., -1.5419e-02,
       -1.7838e-02,  2.4948e-02],
      [-9.9411e-03, -1.4978e-01,  1.7557e-03,  ...,  3.7117e-03,
       -1.6434e-02, -9.9087e-04],
      ...,
      [ 3.6979e-04, -7.7454e-02,  1.2955e-02,  ...,  3.9330e-03,
       -1.1642e-02,  7.8506e-03],
      [-2.6779e-03, -2.2446e-02, -1.6754e-02,  ..., -1.3142e-03,
       -7.8583e-03,  2.0096e-02],
      [-8.6288e-03,  1.4233e-01, -1.9012e-02,  ..., -1.8463e-02,
       -9.8572e-03,  8.7662e-03]]], device='cuda:0', dtype=torch.float16,
    grad_fn=<EmbeddingBackward0>)Copy
```

The output.

Lastly, the self-attention component! We use the first layer's self-attention component by indexing through the layers and accessing the .self_attn method.
Copy

```
embed_position_input = embedded_input + embed_pos_input
hidden_states, _, _ = OPT.model.decoder.layers[0].self_attn(embed_position_input)
print("Layer:\t", OPT.model.decoder.layers[0].self_attn)
print("Size:\t", hidden_states.size())
print("Output:\t", hidden_states)Copy
```

The sample output.

Copy

```
Layer:    OPTAttention(
 (k_proj): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
 (v_proj): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
 (q_proj): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
 (out_proj): Linear8bitLt(in_features=2048, out_features=2048, bias=True)
```

```
)
Size:         torch.Size([1, 10, 2048])
Output:   tensor([[[-0.0119, -0.0110,  0.0056,  ...,  0.0094,  0.0013,  0.0093],
         [-0.0119, -0.0110,  0.0056,  ...,  0.0095,  0.0013,  0.0093],
         [-0.0119, -0.0110,  0.0056,  ...,  0.0095,  0.0013,  0.0093],
         ...,
         [-0.0119, -0.0110,  0.0056,  ...,  0.0095,  0.0013,  0.0093],
         [-0.0119, -0.0110,  0.0056,  ...,  0.0095,  0.0013,  0.0093],
         [-0.0119, -0.0110,  0.0056,  ...,  0.0095,  0.0013,  0.0093]]],
       device='cuda:0', dtype=torch.float16, grad_fn=<MatMul8bitLtBackward>)Copy
```

The output.

The self-attention component comprises the mentioned query, key, and value layers, culminating in a final projection for the output. It takes the sum of the embedded input and the positional encoding vector as input. In a real-world example, the model also provides the attention mask to the component, enabling it to identify which portions of the input should be disregarded or ignored. (removed from the sample code for simplicity)
The rest of the architecture applies non-linearity (e.g., RELU), feedforward, and batch normalization layers.
💡

If you are interested in learning the Transformer architecture in more detail and implement a GPT-like network from scratch, we recommend watching the following video from Andrej Karpathy: https://www.youtube.com/watch?v=kCc8FmEb1nY.

Conclusion

This lesson provides an overview of the transformer architecture and dives deeper into the model's structure by loading a pre-trained model and extracting its essential components. We also look into what occurs within an LLM under the surface. In particular, the attention mechanism serves as the core component of the model.
In the next lesson, we will cover the diverse architectures of the transformer: encoder-decoder, decoder-only (like the GPTs), and encoder-only (like BERT).

Transformers Architectures

Introduction

The transformer architecture has demonstrated its versatility in various applications. The original network was presented as an encoder-decoder architecture for translation tasks. The next evolution of transformer architecture began with the introduction of encoder-only models like BERT, followed by the introduction of decoder-only networks in the first iteration of GPT models.

The differences extend beyond just network design and also encompass the learning objectives. These contrasting learning objectives play a crucial role in shaping the model's behavior and outcomes. Understanding these differences is essential for selecting the most suitable architecture for a given task and achieving optimal performance in various applications.

In this lesson, we will explore the distinctions between these architectures by loading pre-trained models. The goal is to dive deeper into each architecture.

The Encoder-Decoder Architecture

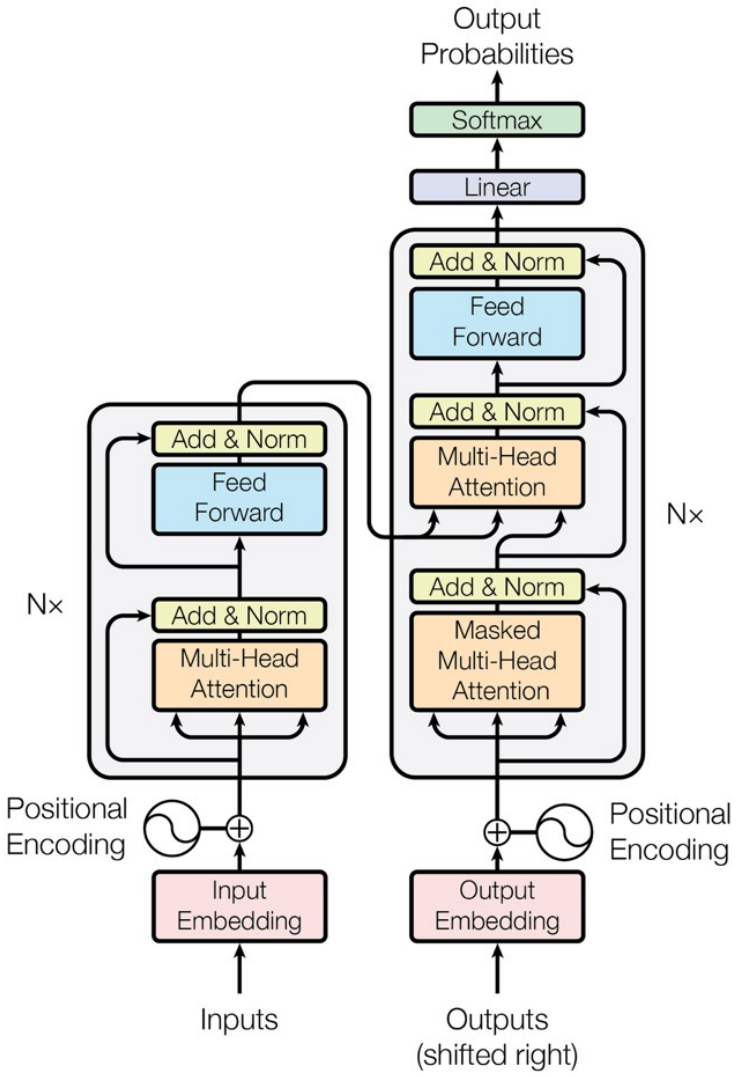


Image taken from the "*Attention is all you need" paper*

The encoder-decoder, also known as the full transformer architecture, comprises multiple stacked encoder components connected to several stacked decoder components through a cross-attention mechanism.
It is notably well-suited for sequence-to-sequence (i.e., handling text as both input and output) tasks such as translation or summarization, mainly when designing models with multi-modality, like image captioning with the image as input and the corresponding caption as the expected output. Cross-attention will help the decoder focus on the most important part of the content during the generation process.
A notable example of this approach is the BART pre-trained model. The architecture incorporates a bi-directional encoder responsible for creating a comprehensive representation of the input, while an autoregressive decoder generates the output one

token at a time. The model takes in a randomly masked input along with the input shifted by one token and attempts to reconstruct the original input as a learning objective.
The provided code below loads the BART model so we can examine its architecture.
Copy

```
from transformers import AutoModel, AutoTokenizer


BART = AutoModel.from_pretrained("facebook/bart-large")
print(BART)Copy
```

The sample code.

Copy

```
BartModel(
 (shared): Embedding(50265, 1024, padding_idx=1)
 (encoder): BartEncoder(
  (embed_tokens): Embedding(50265, 1024, padding_idx=1)
  (embed_positions): BartLearnedPositionalEmbedding(1026, 1024)
  (layers): ModuleList(
   (0-11): 12 x BartEncoderLayer(
    (self_attn): BartAttention(
     (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
     (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
     (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
     (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    (activation_fn): GELUActivation()
    (fc1): Linear(in_features=1024, out_features=4096, bias=True)
    (fc2): Linear(in_features=4096, out_features=1024, bias=True)
    (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
   )
```

```
  )
  (layernorm_embedding): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
)
(decoder): BartDecoder(
  (embed_tokens): Embedding(50265, 1024, padding_idx=1)
  (embed_positions): BartLearnedPositionalEmbedding(1026, 1024)
  (layers): ModuleList(
    (0-11): 12 x BartDecoderLayer(
      (self_attn): BartAttention(
        (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
        (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
        (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
        (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
      )
      (activation_fn): GELUActivation()
      (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
      (encoder_attn): BartAttention(
        (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
        (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
        (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
        (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
      )
      (encoder_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
      (fc1): Linear(in_features=1024, out_features=4096, bias=True)
      (fc2): Linear(in_features=4096, out_features=1024, bias=True)
      (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    )
  )
  (layernorm_embedding): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
```

```
 )
)Copy
```

The output.

We are already familiar with most of the layers in the BART model. The model is comprised of both encoder and decoder components, with each component consisting of 12 layers. Additionally, The decoder component, in particular, contains an additional encoder_attn layer, referred to as cross-attention. The cross-attention component will condition the decoder's output based on the encoder representations.
We can use the fine-tuned version of this model for summarization using the Transformer's pipeline functionality.
Copy

```
from transformers import pipeline


summarizer = pipeline("summarization", model="facebook/bart-large-cnn")

sum = summarizer("""Gaga was best known in the 2010s for pop hits like "Poker Face" and avant-garde experimentation on albums like "Artpop," and Bennett, a singer who mostly stuck to standards, was in his 80s when the pair met. And yet Bennett and Gaga became fast friends and close collaborators, which they remained until Bennett's death at 96 on Friday. They recorded two albums together, 2014's "Cheek to Cheek" and 2021's "Love for Sale," which both won Grammys for best traditional pop vocal album.""", min_length=20, max_length=50)

print(sum[0]['summary_text'])Copy
```

The sample code.

Copy

```
Bennett and Gaga became fast friends and close collaborators. They recorded two albums together, 2014's "Cheek to Cheek" and 2021's "Love for Sale"
```

The output.

The Encoder-Only Architecture

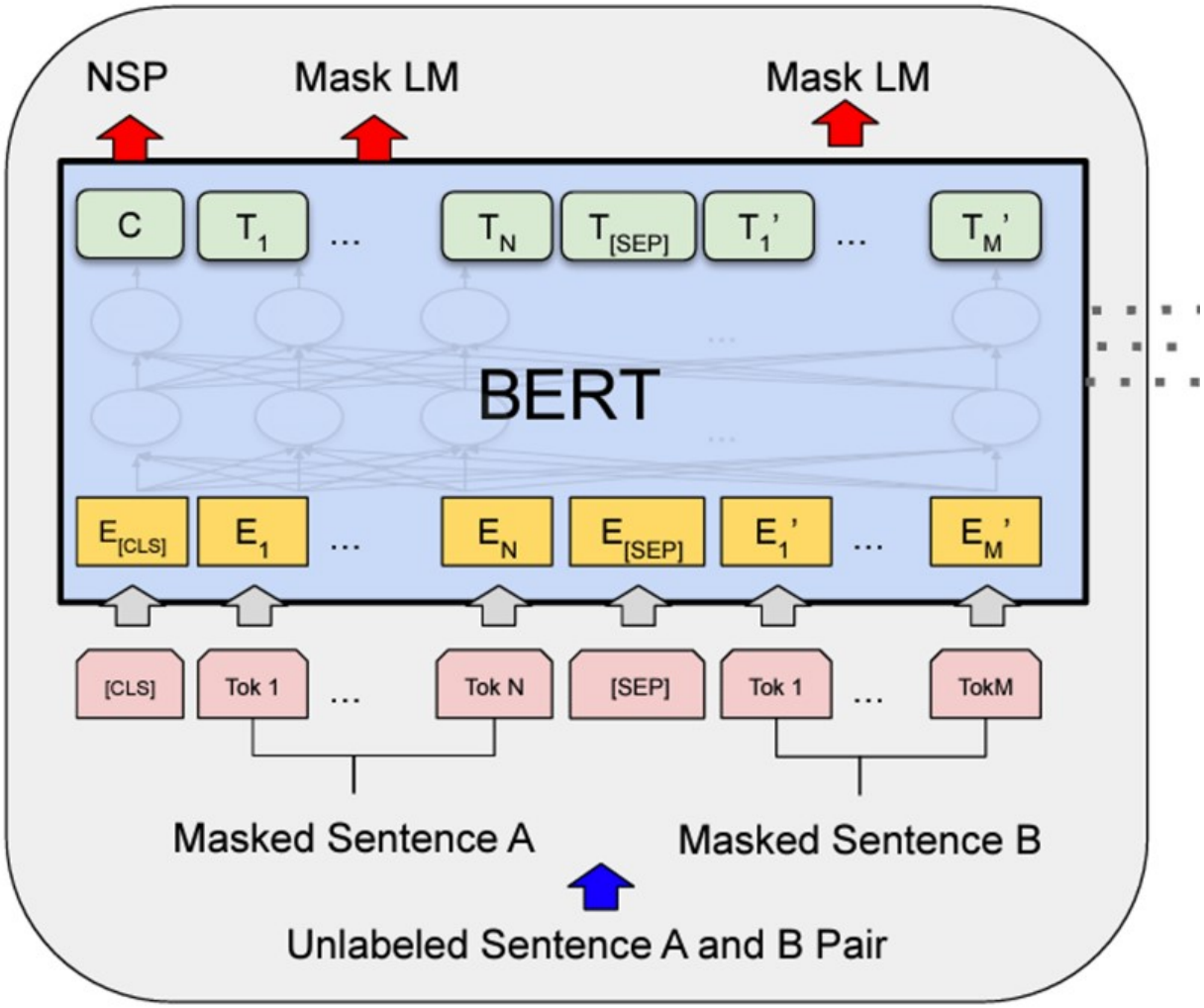


Image taken from the "BERT: Pre-training of Deep Bidirectional Transformers for LanguageUnderstanding" paper

As implied by the name, the encoder-only models are formed by stacking multiple encoder components. As the encoder output cannot be connected to another decoder, its output can be directly used as a text-to-vector method, for instance, to measure similarity. Alternatively, it can be combined with a classification head (feedforward layer) on top to facilitate label prediction (it is also known as a Pooler layer in libraries such as Huggingface). The primary distinction in the encoder-only architecture lies in the absence of the Masked Self-Attention layer. As a result, the encoder can handle the entire input simultaneously. This differs from decoders, where future tokens need to be masked during training to prevent "cheating" when generating new tokens. Due to this property, they are ideally suited for creating representations from a document while retaining complete information.

The BERT paper (or an improved variant like RoBERTa) introduced a widely recognized pre-trained model that significantly improved the state-of-the-art scores on numerous NLP tasks. The model undergoes pre-training with two learning objectives:

1. Masked Language Modeling: masking random tokens from the input and attempting to predict them.
2. Next Sentence Prediction: Present sentences in pairs and assess the likelihood of the second sentence in the subsequent sequence of the first sentence.

Copy

```
BERT = AutoModel.from_pretrained("bert-base-uncased")
print(BERT)Copy
```

The sample code.

Copy

```
BertModel(
 (embeddings): BertEmbeddings(
  (word_embeddings): Embedding(30522, 768, padding_idx=0)
  (position_embeddings): Embedding(512, 768)
  (token_type_embeddings): Embedding(2, 768)
  (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  (dropout): Dropout(p=0.1, inplace=False)
 )
 (encoder): BertEncoder(
  (layer): ModuleList(
   (0-11): 12 x BertLayer(
    (attention): BertAttention(
     (self): BertSelfAttention(
      (query): Linear(in_features=768, out_features=768, bias=True)
      (key): Linear(in_features=768, out_features=768, bias=True)
      (value): Linear(in_features=768, out_features=768, bias=True)
      (dropout): Dropout(p=0.1, inplace=False)
     )
     (output): BertSelfOutput(
```

```
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
       )
      )
      (intermediate): BertIntermediate(
       (dense): Linear(in_features=768, out_features=3072, bias=True)
       (intermediate_act_fn): GELUActivation()
      )
      (output): BertOutput(
       (dense): Linear(in_features=3072, out_features=768, bias=True)
       (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
       (dropout): Dropout(p=0.1, inplace=False)
      )
     )
    )
   )
   (pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
   )
)Copy
```

The output.

The BERT model adopts the conventional transformer architecture for input embedding and 12 encoder blocks. However, the network's output will be passed on to a pooler layer, which is a feed-forward linear layer followed by non-linearity that will generate the final representation. This representation will subsequently be utilized for various tasks, such as classification or similarity assessment.

The following code uses the fine-tuned version of the BERT model for sentiment analysis.

Copy

```
classifier = pipeline("text-classification", model="nlptown/bert-base-multilingual-uncased-sentiment")

lbl = classifier("""This restaurant is awesome.""")


print(lbl)Copy
```

The sample code.

Copy

```
[{'label': '5 stars', 'score': 0.8550480604171753}]Copy
```

The output.

The Decoder-Only Architecture

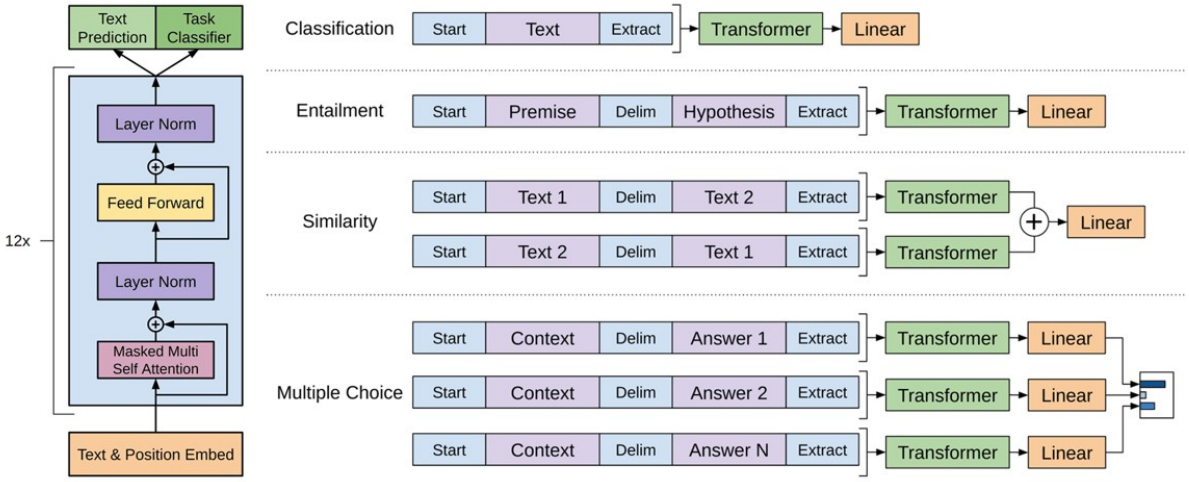


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

Image taken from "Improving language understanding with unsupervised learning" paper

The decoder-only networks continue to serve as the foundation for most large language models today, with slight variations in some instances. Because of the implementation of masked self-attention, their primary use case revolves around the next-token-prediction task, which sparked the concept of prompting.
Research demonstrated that scaling up the decoder-only models can significantly enhance the network's language understanding and generalization capabilities. As a result, they can excel at a diverse range of tasks simply by using different prompts. Large pre-trained models

like GPT-4 and LLaMA 2 exhibit the ability to perform tasks such as classification, summarization, translation, etc., by leveraging the appropriate prompt.
The large language models, such as those in the GPT family, undergo pre-training using the Causal Language Modeling objective. This means the model aims to predict the next word, while the attention mechanism can only attend to previous tokens on the left. This implies that the model can solely rely on the previous context to predict the next token and is unable to peek at future tokens, preventing any form of cheating.
Copy

```
gpt2 = AutoModel.from_pretrained("gpt2")
print(gpt2)Copy
```

The sample code.

Copy

```
GPT2Model(
 (wte): Embedding(50257, 768)
 (wpe): Embedding(1024, 768)
 (drop): Dropout(p=0.1, inplace=False)
 (h): ModuleList(
  (0-11): 12 x GPT2Block(
   (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
   (attn): GPT2Attention(
    (c_attn): Conv1D()
    (c_proj): Conv1D()
    (attn_dropout): Dropout(p=0.1, inplace=False)
    (resid_dropout): Dropout(p=0.1, inplace=False)
   )
   (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
   (mlp): GPT2MLP(
    (c_fc): Conv1D()
    (c_proj): Conv1D()
    (act): NewGELUActivation()
    (dropout): Dropout(p=0.1, inplace=False)
```

```
    )
   )
 )
 (ln_f): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)Copy
```

The output.

When examining the architecture, you will notice the standard transformer decoder block with the cross-attention removed. The GPT family also employs different linear layers (Conv1D) that transpose the weights. (Please note that this should not be confused with PyTorch's convolutional layer.) This design choice is specific to OpenAI, while other open-source large language models use the standard linear layer. The provided code illustrates how the pipeline can be used to incorporate the GPT2 model for text generation. It generates four different alternatives to complete the phrase "This movie was a very."
Copy

```
generator = pipeline(model="gpt2")

output = generator("This movie was a very", do_sample=True, top_p=0.95, num_return_sequences=4, max_new_tokens=50, return_full_text=False)


for item in output:
  print(">", item['generated_text'])Copy
```

The sample code.

Copy

```
>  hard thing to make, but this movie is still one of the most amazing shows I've seen in years. You know, it's sort of fun for a couple of decades to watch, and all that stuff, but one thing's for sure —

>  special thing and that's what really really made this movie special," said Kiefer Sutherland, who co-wrote and directed the film's cinematography. "A lot of times things in our lives get passed on from one generation to another, whether

>  good, good effort and I have no doubt that if it has been released, I will be very pleased with it."


Read more at the Mirror.
```

> enjoyable one for the many reasons that I would like to talk about here. First off, I'm not just talking about the original cast, I'm talking about the cast members that we've seen before and it would be fair to say that none of

The output.

💡

Please be aware that running the above code will yield different outputs due to the randomness involved in the generation process.

Conclusion

In this lesson, we explored the various types of transformer-based models and their areas of maximum effectiveness. While LLMs may appear to be the ultimate solution for every task, it's essential to note that there are instances where smaller, more focused models can produce equally good results while operating more efficiently. Using a small model like DistilBERT on your local server to measure similarity could be more suitable for specific applications while offering a cost-effective alternative to using proprietary models and APIs. Moreover, the transformer paper introduced an effective architecture. However, various architectures have been experimented with minor code changes, such as different embedding sizes and hidden dimensions. Recent experiments have also shown that relocating the batch normalization layer before the attention mechanism can enhance the model's capabilities. Keep in mind that there could be slight variations in the architecture, especially for proprietary models like GPT-3 that have not released their code.
In this Notebook, you can find the code for this lesson.

Focus on the GPT Architecture

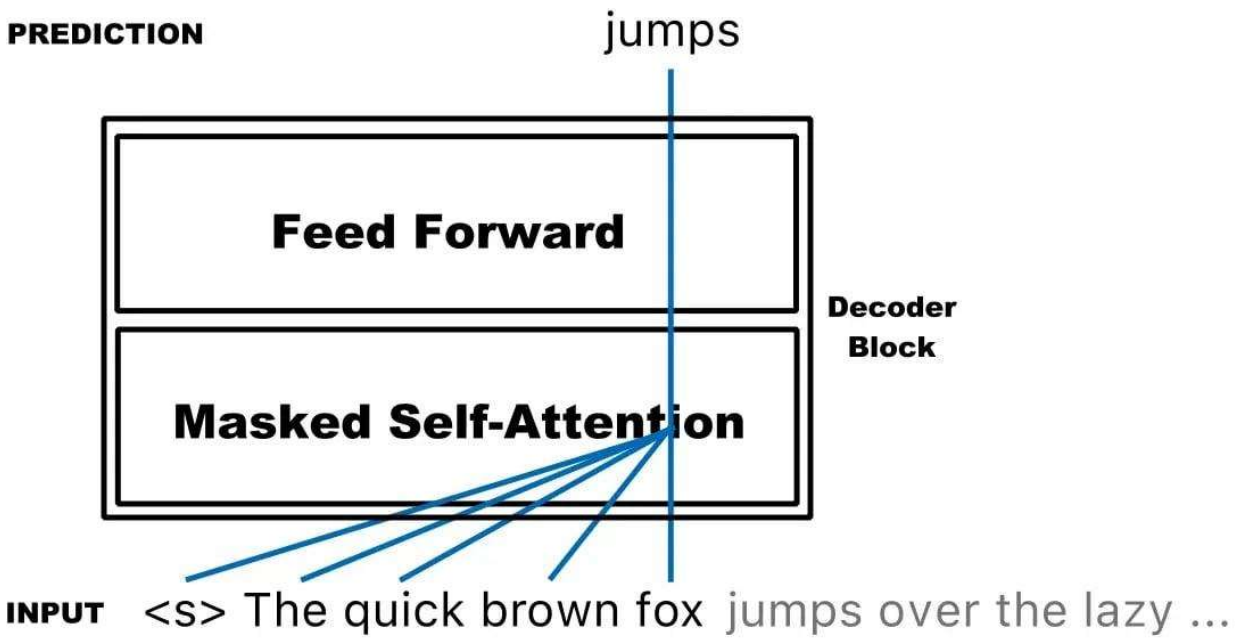
Introduction

The Generative Pre-trained Transformer (GPT) is a type of transformer-based language model developed by OpenAI. The 'transformer' part of its name refers to its transformer architecture, which was introduced in the research paper "Attention is All You Need" by Vaswani et al.
You should have a good understanding of the fundamental elements comprising the transformer architecture. In this session, we will cover the decoder-only networks that play an essential role in developing large language models. We will explore their unique attributes and the reasons behind their effectiveness.
In contrast to conventional Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, the transformer architecture departs from recurrence and adopts self-attention mechanisms, resulting in substantial advancements in speed and scalability. An immensely powerful architecture was unleashed by harnessing the potential for parallelization within the network (simultaneously running multiple head attentions) along with the abundant small cores available in a GPU.

The GPT Architecture

The GPT family comprises decoder-only models, wherein each block in the stack is comprised of a self-attention mechanism and a position-wise fully connected feed-forward network. The self-attention mechanism, also known as scaled dot-product attention, allows the model to weigh the importance of each word in the input when generating the next word in the sequence. It computes a weighted sum of all the words in the sequence, where the weights are determined by the attention scores.

The critical aspect to focus on is the addition of "masking" to the self-attention that prevents the model from attending to certain positions/words.



Illustrating which tokens are attended to by masked self-attention at a particular timestamp. (Image taken from NLPiation) As you see in the figure, we pass the whole sequence to the model, but the model at timestep 5 tries to predict the next token by only looking at the previously generated tokens, masking the future tokens. This prevents the model from "cheating" by predicting tokens leveraging future tokens.

The following code simply implements the "masked self-attention" mechanism.

Copy

```
import numpy as np


def self_attention(query, key, value, mask=None):
    # Compute attention scores
```

```
    scores = np.dot(query, key.T)

    if mask is not None:
        # Apply mask by setting masked positions to a large negative value
        scores = scores + mask * -1e9

    # Apply softmax to obtain attention weights
    attention_weights = np.exp(scores) / np.sum(np.exp(scores), axis=-1, keepdims=True)

    # Compute weighted sum of value vectors
    output = np.dot(attention_weights, value)

    return output
```

The first step is to compute a Query, Key, and Value vector for each word in the input sequence using separate learned linear transformations of the input vector. It is a simple feedforward linear layer that the model learns during training.

Then, we can calculate the attention scores by taking the dot product of its Query vector with the Key vector of every other word. Currently, the application of masking is feasible by setting the scores in specific locations to a large negative number. This effectively informs the model that those words are unimportant and should be disregarded during attention. To get the attention weights, apply the SoftMax function to the attention scores to convert them into probabilities. This gives the weights of the input words and effectively turns the significant negative scores to zero. Lastly, multiply each Value vector by its corresponding weight and sum them up. This produces the output of the masked self-attention mechanism for the word.

The provided code snippet illustrates the process of a single self-attention head, but in reality, each layer contains multiple heads, which could range from 16 to 32 heads, depending on the architecture. These heads operate simultaneously to enhance the model's performance.

Causal Language Modeling

LLMs utilize a **self-supervised learning** process for pre-training. This process eliminates the need to provide explicit labels to the model for learning, making it capable of acquiring knowledge autonomously. For instance, when training a summarization model using supervised learning, it is necessary to provide articles and their corresponding summaries as reference points during the training process. However, LLMs employ the causal language modeling objective to acquire knowledge from any textual data without the explicit need

for human-provided labels. Why is it called “causal”? Because the prediction at each step depends only on earlier steps in the sequence and not on future steps.
This process involves feeding a segment of the document to the model and asking it to predict the next word.

Subsequently, the predicted word is concatenated to the original input and fed back to the model to predict a new token. This iterative loop continues, consistently feeding the newly generated token back into the network. During the pre-training process, the network acquires substantial knowledge about language and grammar. We can then fine-tune the pre-trained model using a supervised approach for different tasks or a specific domain.
Compared to other well-known objectives, the advantage of this approach is that it models how humans naturally write or speak. In contrast to other objectives like masked language modeling, where masked tokens are introduced in the input, the causal language modeling approach constructs sentences one word at a time. This key difference ensures that our model's performance is not adversely affected when dealing with real-world passages lacking masking tokens.
Moreover, we can utilize extensive, high-quality, human-generated content spanning centuries. This content can be derived from books, Wikipedia, news websites, and more. Familiar datasets and repositories, such as ActiveLoop and Huggingface, provide convenient access to some well-known datasets. We will cover this topic in more detail in later lessons.

MinGPT

Numerous implementations of the GPT architecture exist, each designed for specific purposes. In upcoming lessons, we will thoroughly explore alternative libraries that are better suited for production environments. However, we are introducing a lightweight repository implemented by Andrej Karpathy, named minGPT. This represents a minimal implementation of OpenAI's GPT-2 model.
In his own words, this serves as an educational implementation that strives to remove all complexities, achieving a length of just 300 lines of code and using the PyTorch library. This valuable resource provides an excellent opportunity to read and enhance your understanding of what's happening under the hood. Abundant comments in the code describe the processes and act as a helpful guide.
Three main files can be found within the repository. First, model.py handles the definition of architecture details. Second, bpe.py is responsible for the tokenization process using the BPE algorithm. Lastly, train.py represents the implementation of a generic training loop for any neural network, not limited to the GPT architecture. Furthermore, the demo.ipynb file contains a notebook that demonstrates the complete utilization of the code, including the inference process. The code can be executed on a MacBook Air, making it accessible for use on your local PC. Alternatively, you can fork the repository and utilize services like Colab.

Conclusion

The decoder-only architecture and GPT-family models have driven the recent advancements in large language models. It is essential to possess a strong grasp of the transformer architecture and comprehend the distinctive features that set the decoder-only models apart, making them well-suited for language modeling. We have explored the shared

components and delved deeper into what makes their architecture unique. Subsequent lessons will cover various other aspects of language models.

Evaluating LLM Performance

Introduction

In this lesson, we will explore two crucial aspects of language model evaluation: objective functions and evaluation metrics.

Objective functions, also known as loss functions, play a vital role in guiding the learning process during model training. On the other hand, evaluation metrics provide interpretable measures of the model's capabilities and are used to assess its performance on various tasks.

We will dive into the perplexity evaluation metric, commonly used for LLMs, and explore several benchmarking frameworks, such as GLUE, SuperGLUE, BIG-bench, HELM, and FLASK, that help comprehensively evaluate language models across diverse scenarios.

Objective Functions and Evaluation Metrics

Objective functions and evaluation metrics are essential components in machine learning models.

The **objective function**, also known as the **loss function**, is a mathematical formula used during the training phase. It gives a loss score to the model in function of the model parameters. During training, the learning algorithm computes gradients of the loss function and updates the model parameters to minimize it. As a consequence, to guarantee (smooth) learning, the loss function needs to be differentiable and have an excellent smooth form.

The objective function typically used for LLMs is the **cross-entropy loss**. In the case of causal language modeling, the model predicts the next token from a fixed list of tokens, essentially making it a classification problem.

On the other hand, **evaluation metrics** are used to assess the model's performance in an interpretable way for people. Unlike the objective function, evaluation metrics are not directly used during training. As a consequence, evaluation metrics don’t need to be differentiable, as we won’t have to compute gradients for them. Standard evaluation metrics include accuracy, precision, recall, F1-score, and mean squared error.

Typical evaluation metrics for LLMs can be:

- **Intrinsic metrics**, i.e., metrics strictly related to the training objective. A popular example is the **perplexity** metric.
- **Extrinsic metrics** are metrics that aim to assess performance on several downstream tasks and are not strictly related to the training objective. The GLUE, SuperGLUE, BIG-bench, HELM, and FLASK benchmarks are popular examples.

The Perplexity Evaluation Metric

Perplexity is an evaluation metric used to assess the performance of LLMs. It measures how well a language model predicts a given sample or sequence of words, such as a sentence. The lower the perplexity value, the better the language model is at predicting the sample.

LLMs are designed to model the probability distributions of words within sentences. They can generate sentences resembling human writing and assess the sentences' quality. Perplexity is a measure that quantifies the uncertainty or "perplexity" a model experiences when assigning probabilities to sequences of words.

The first step in computing perplexity is to calculate the probability of a sentence by multiplying the probabilities of individual words according to the language model. Longer sentences tend to have lower probabilities due to the multiplication of factors smaller than one. To make comparisons between sentences with different lengths possible, perplexity normalizes the probability by dividing it by the number of words in the sentence and taking the geometric mean.

Perplexity Example

Consider an example where a language model is trained to predict the subsequent word in a sentence: "A red fox." For a competent LLM, the predicted word probabilities could be as follows, step by step.

P("a red fox.") =

= P("a") * P("red" | "a") * P("fox" | "a red") * P("." | "a red fox") =
= 0.4 * 0.27 * 0.55 * 0.79 =
= 0.0469

It would be nice to compare the probabilities assigned to different sentences to see which sentences are better predicted by the language model. However, since the probability of a sentence is obtained from a product of probabilities, the longer the sentence, the lower its probability (since it's a product of factors with values smaller than one). We should find a way of measuring these sentence probabilities without the influence of the sentence length. This can be done by normalizing the sentence probability by the number of words in the sentence. Since the probability of a sentence is obtained by multiplying many factors, we can average them using the geometric mean.

Let's call *Pnorm(W)* the normalized probability of the sentence *W*. Let *n* be the number of words in *W*. Then, applying the geometric mean:

Pnorm(W) = P(W) ^ (1 / n)

Using our specific sentence, "*a red fox.*":

Pnorm("a red fox.") = P("a red ") ^ (1 / 4) = 0.465

Great! This number can now be used to compare the probabilities of sentences with different lengths. The higher this number is over a well-written sentence, the better the language model.

So, what does this have to do with perplexity? Well, perplexity is just the reciprocal of this number.

Let's call *PP(W)* the perplexity computed over the sentence *W*. Then:

PP(W) = 1 / Pnorm(W) =

= 1 / (P(W) ^ (1 / n))
= (1 / P(W)) ^ (1 / n)

Let's compute it with numpy:

Copy

```
import numpy as np
```

```
probabilities = np.array([0.4, 0.27, 0.55, 0.79])
sentence_probability = probabilities.prod()
sentence_probability_normalized = sentence_probability ** (1 / len(probabilities))
perplexity = 1 / sentence_probability_normalized
print(perplexity) # 2.1485556947850033Copy
```

Suppose we further train the LLM, and the probabilities of the next best word become higher. How would the final perplexity be, higher or lower?
Copy

```
probabilities = np.array([0.7, 0.5, 0.6, 0.9])
sentence_probability = probabilities.prod()
sentence_probability_normalized = sentence_probability ** (1 / len(probabilities))
perplexity = 1 / sentence_probability_normalized
print(perplexity) # 1.516647134682679 -> lowerCopy
```

The GLUE Benchmark

The GLUE (General Language Understanding Evaluation) benchmark comprises nine diverse English sentence understanding tasks categorized into three groups.

- The first group, Single-Sentence Tasks, evaluates the model's ability to determine grammatical correctness (CoLA) and sentiment polarity (SST-2) of individual sentences.
- The second group, Similarity, and Paraphrase Tasks, focuses on assessing the model's capacity to identify paraphrases in sentence pairs (MRPC and QQP) and determine the similarity score between sentences (STS-B).
- The third group, Inference Tasks, challenges the model to handle sentence entailment and relationships. This includes recognizing textual entailment (RTE), answering questions based on sentence information (QNLI), and resolving pronoun references (WNLI).

The final GLUE score is obtained by averaging performance across all nine tasks. By providing a unified evaluation platform, GLUE facilitates a deeper understanding of the strengths and weaknesses of various NLP models.

The SuperGLUE Benchmark

The SuperGLUE benchmark builds upon the GLUE benchmark but introduces more complex tasks to push the boundaries of current NLP approaches. The key features of SuperGLUE are:

1. Tasks: SuperGLUE consists of eight diverse language understanding tasks. These tasks include Boolean question answering, textual entailment, coreference

resolution, reading comprehension with commonsense reasoning, and word sense disambiguation.

2. Difficulty: The benchmark retains the two hardest tasks from GLUE and adds new tasks based on the challenges faced by current NLP models, ensuring greater complexity and relevance to real-world language understanding scenarios.
3. Human Baselines: Human performance estimates are included for each task, providing a benchmark for evaluating the performance of NLP models against human-level understanding.
4. Evaluation: NLP models are evaluated on these tasks, and their performance is measured using a single-number overall score obtained by averaging the scores of all individual tasks.

The BIG-Bench Benchmark

BIG-bench is a large-scale and diverse benchmark designed to evaluate the capabilities of large language models. It consists of 204 or more language tasks that cover a wide range of topics and languages. These are challenging and not entirely solvable by current models.
The benchmark supports two types of tasks: JSON-based and programmatic tasks. JSON tasks involve comparing output and target pairs to evaluate performance, while programmatic tasks use Python to measure text generation and conditional log probabilities.
The tasks include writing code, common-sense reasoning, playing games, linguistics, and more.
The researchers found that aggregate performance improves with model size but still falls short of human performance. Model predictions become better calibrated with increased scale, and sparsity offers benefits.
This benchmark is considered a "living benchmark," accepting new task submissions for continuous peer review. The code for BIG-bench is open-source on GitHub, and the research paper is available on arXiv.

The HELM Benchmark

The HELM (Holistic Evaluation of Language Models) benchmark addresses the lack of a unified standard for comparing language models and aims to assess them in their totality.
The benchmark has three main components:

1. Broad Coverage and Recognition of Incompleteness: HELM evaluates language models over a diverse set of scenarios, considering different tasks, domains, languages, and user-facing applications. It acknowledges that not all scenarios can be covered but explicitly identifies major scenarios and missing metrics to highlight improvement areas.
2. Multi-Metric Measurement: HELM evaluates language models based on multiple criteria, unlike previous benchmarks that often focus on a single metric like accuracy. It measures 7 metrics: accuracy, calibration, robustness, fairness, bias, toxicity, and efficiency. This multi-metric approach ensures that non-accuracy desiderata are not overlooked.
3. Standardization: HELM aims to standardize the evaluation process for different language models. It specifies an adaptation procedure using few-shot prompting,

making it easier to compare models effectively. By evaluating 30 models from various providers, HELM improves the overall landscape of language model evaluation and encourages a more transparent and reliable infrastructure for language technologies.

The FLASK Benchmark

The FLASK (Fine-grained Language Model Evaluation based on Alignment Skill Sets) benchmark is an evaluation protocol for LLMs. It breaks down the evaluation process into 12 specific instance-wise skill sets, each representing a crucial aspect of a model's capabilities. These skill sets comprise logical correctness, logical efficiency, factuality, commonsense understanding, comprehension, insightfulness, completeness, metacognition, readability, conciseness, and harmlessness.

By breaking down the evaluation into these specific skill sets, FLASK allows for a precise and comprehensive assessment of a model's performance across various tasks, domains, and difficulty levels. This approach provides a more detailed and nuanced understanding of a language model's strengths and weaknesses, enabling researchers and developers to improve the models in targeted ways and address specific challenges in natural language processing.

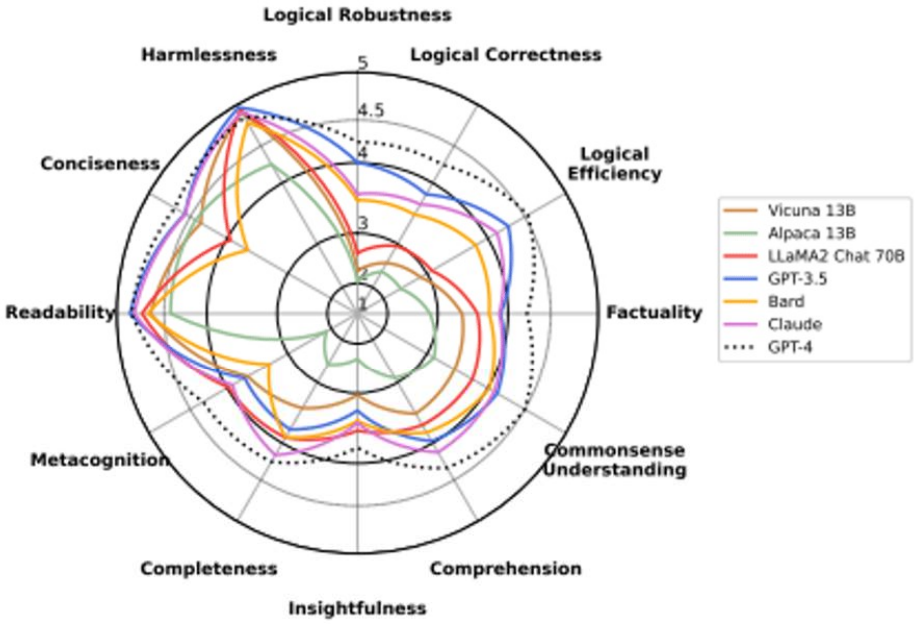


Figure 1: FLASK is a comprehensive evaluation framework for language models considering: `Logical Thinking` (Logical Robustness, Logical Correctness, Logical Efficiency), `Background Knowledge` (Factuality, Commonsense Understanding), `Problem Handling` (Comprehension, Insightfulness, Completeness, Metacognition), `User Alignment` (Readability, Conciseness, Harmlessness). Exact numbers are reported in Table 1.

Conclusion

In this lesson, we explored the essential concepts of evaluating LLM performance through objective functions and evaluation metrics. The objective or loss function plays a critical role during model training. It guides the learning algorithm to minimize the loss score by updating model parameters. For LLMs, the common objective function is the cross-entropy loss.

On the other hand, evaluation metrics are used to assess the model's performance more interpretably, though they are not directly used during training. Perplexity is one such intrinsic metric used to measure how well an LLM predicts a given sample or sequence of words.

Additionally, the lesson introduced several popular extrinsic evaluation benchmarks, such as GLUE, SuperGLUE, BIG-bench, HELM, and FLASK, which evaluate language models on diverse tasks and scenarios, covering aspects like accuracy, fairness, robustness, and more.

By understanding these concepts and using appropriate evaluation metrics and benchmarks, researchers and developers can gain valuable insights into language models' strengths and weaknesses, leading to improving these technologies.

Controlling LLM Outputs

Introduction

In this lesson, we will look into various methods and parameters that can be used to control the outputs of Large Language Models. We will discuss different decoding strategies and how they influence the generation process. We will also explore how certain parameters can be adjusted to fine-tune the output.

Decoding Methods

Decoding methods are fundamental strategies used by LLMs to generate text. Each method has its unique advantages and limitations.

At each decoding step, the LLM gives a score to each of its vocabulary tokens. A high score is related to a high probability of that token being the next token, according to the patterns learned by the model during training.

However, is the token with the highest probability always the best token to predict? By predicting the best token at step 1, the model may then find only tokens with low probabilities at step 2, thus having a low joint probability of the two consecutive tokens. Instead, predicting a slightly lower token at step 1 leads to a high probability token at step 2, thus having an overall higher joint probability of the tokens. Ideally, we'd want to do these computations for all the tokens in the model vocabulary and a large number of steps. However, this can't be done in practice because it would require heavy computations.

All the decoding methods in this lesson try to find the right balance between:

- Being "greedy" and instantly selecting the next token with higher probability.
- A bit of exploration and trying to predict more tokens at once.

Greedy Search

Greedy Search is the simplest of all the decoding methods.

With Greedy Search, the model selects the token with the highest probability as its next output token. While this method is computationally efficient, it can often result in repetitive

or less optimal responses due to its focus on immediate reward rather than long-term outcomes.

Sampling

Sampling introduces randomness into the text generation process, where the model randomly selects the next word based on its probability distribution. This method allows for more diverse and varied output but can sometimes produce less coherent or logical text.

Beam Search

Beam Search is a more sophisticated method. It selects the top N (with N being a parameter) candidate subsequent tokens with the highest probabilities at each step, but only up to a certain number of steps. In the end, the model generates the sequence of tokens (i.e., the beam) with the highest joint probability.
This significantly reduces the search space and produces more consistent results. However, this method might be slower and lead to suboptimal outputs as it can miss high-probability words hidden behind a low-probability word.

Top-K Sampling

Top-K Sampling is a variant of the sampling method where the model narrows down the sampling pool to the top K (with K being a parameter) of the most probable words. This method provides a balance between diversity and relevance by limiting the sampling space, thus offering more control over the generated text.

Top-p (Nucleus) Sampling

Top-p, or Nucleus Sampling, selects words from the smallest possible set of tokens whose cumulative probability exceeds a certain threshold P (with P being a parameter). This method offers fine-grained control and avoids the inclusion of rare or low-probability tokens. However, the dynamically determined shortlist sizes can sometimes be a limitation.

Parameters That Influence Text Generation

Apart from the decoding methods, several parameters can be adjusted to influence text generation using LLMs. These include temperature, stop sequences, frequency, and presence penalties.
These parameters can be adjusted with the most popular LLM APIs and Hugging Face models.

Temperature

The temperature parameter influences the randomness or determinism of the generated text. A lower value makes the output more deterministic and focused, while a higher value increases the randomness, leading to more diverse outputs.
It controls the randomness of predictions by scaling the logits before applying softmax during the text generation process. It's a crucial factor in the trade-off between diversity and quality of the generated text.
Here's a more technical explanation:

1. **Logits**: When a language model makes a prediction, it generates a vector of logits, one for the next possible token. These logits represent the raw, unnormalized prediction scores for each token.

2. **Softmax**: The softmax function is applied to these logits to convert them into probabilities. The softmax function also ensures that these probabilities sum up to 1.
3. **Temperature**: The temperature parameter is used to control the randomness of the model's output. It does this by dividing the logits by the temperature value before the softmax step.
    - **High Temperature (e.g., > 1)**: The logits are scaled down, which makes the softmax output more uniform. This means the model is more likely to pick less likely words, resulting in more diverse and "creative" outputs, but potentially with more mistakes or nonsensical phrases.
    - **Low Temperature (e.g., < 1)**: The logits are scaled up, which makes the softmax output more peaked. This means the model is more likely to pick the most likely word. The output will be more focused and conservative, sticking closer to the most probable outputs but potentially less diverse.
    - **Temperature = 1**: The logits are not scaled, preserving the original probabilities. This is a kind of "neutral" setting.

In summary, the temperature parameter is a knob for controlling the trade-off between diversity (high temperature) and accuracy (low temperature) in the generated text.

Stop Sequences

Stop sequences are specific sets of character sequences that halt the text generation process once they appear in the output. They offer a way to guide the length and structure of the generated text, providing a form of control over the output.

Frequency and Presence Penalties

Frequency and presence penalties are used to discourage or encourage the repetition of certain words in the generated text. A frequency penalty reduces the likelihood of the model repeating tokens that have appeared frequently, while a presence penalty discourages the model from repeating any token that has already appeared in the generated text.

Conclusion

This lesson provided an overview of the various decoding methods and parameters that can be used to control the outputs of Large Language Models.

We've explored decoding strategies such as Greedy Search, Sampling, Beam Search, Top-K Sampling, and Top-p (Nucleus) Sampling, each with its unique approach to balancing the trade-off between immediate reward and long-term outcomes.

We've also discussed parameters like temperature, stop sequences, and frequency and presence penalties, which offer additional control over text generation.

Adjusting these parameters can help in guiding the model to produce the desired results, whether deterministic, focused outputs or more diverse, creative ones.

Prompting and Few-Shot Prompting

Introduction

In this lesson, we will explore prompting and prompt engineering, which allow us to interact with LLMs effectively for various applications. We can leverage LLMs to perform tasks such as answering questions, text generation, and more by crafting specific prompts.

We will delve into zero-shot prompting, where the model produces results without explicit examples, and then transition to in-context learning and few-shot prompting, where the model learns from demonstrations to handle complex tasks with minimal training data.

Prompting and Prompt Engineering

Prompting is a very important technique that involves designing and optimizing prompts to interact effectively with LLMs for various applications. The process of **prompt engineering** enables developers and researchers to harness the capabilities of LLMs and utilize them for tasks such as answering questions, arithmetic reasoning, text generation, and more.

At its core, prompting involves presenting a specific task or instruction to the language model, which then generates a response based on the information provided in the prompt. A prompt can be as simple as a question or instruction or include additional context, examples, or inputs to guide the model towards producing desired outputs. The quality of the results largely depends on the precision and relevance of the information provided in the prompt.

Let's incorporate these ideas in the code examples. Before running them, remember to load your environment variables from your .env file as follows.

Copy

```
from dotenv import load_dotenv

load_dotenv()
```

Example: Story Generation

In this example, the prompt sets up the start of a story, providing initial context ("a world where animals could speak") and a character ("a courageous mouse named Benjamin"). The model's task is to generate the rest of the story based on this prompt.

Note that in this example we are defining separately a prompt_system and a prompt. This is because the OpenAI API works this way, requiring a “system prompt” to steer the model behaviour. This is different from other LLMs that require only a standard prompt.

Copy

```
import openai


prompt_system = "You are a helpful assistant whose goal is to help write stories."


prompt = """Continue the following story. Write no more than 50 words.


Once upon a time, in a world where animals could speak, a courageous mouse named Benjamin decided to"""


response = openai.ChatCompletion.create(
```

```
  model="gpt-3.5-turbo",
  messages=[
    {"role": "system", "content": prompt_system},
    {"role": "user", "content": prompt}
  ]
)

print(response.choices[0]['message']['content'])
```

Code to run.

Copy

```
embark on a journey to find the legendary Golden Cheese. With determination
in his heart, he ventured through thick forests and perilous mountains,
facing countless obstacles. Little did he know that his bravery would
lead him to the greatest adventure of his life.
```

The output.

Example: Product Description

Here, the prompt is a request for a product description with key details ("luxurious, hand-crafted, limited-edition fountain pen made from rosewood and gold"). The model is tasked with writing an appealing product description based on these details.

Copy

```
import openai

prompt_system = "You are a helpful assistant whose goal is to help write product descriptions."

prompt = """Write a captivating product description for a luxurious, hand-crafted, limited-edition fountain pen made from rosewood and gold.
Write no more than 50 words."""

response = openai.ChatCompletion.create(
```

```
  model="gpt-3.5-turbo",
  messages=[
    {"role": "system", "content": prompt_system},
    {"role": "user", "content": prompt}
  ]
)

print(response.choices[0]['message']['content'])
```

Code to run.

Copy

```
Experience the epitome of elegance with our luxurious limited-edition
fountain pen. Meticulously handcrafted from exquisite rosewood and
shimmering gold, this writing instrument exudes sophistication in every
stroke. Elevate your writing experience to new heights with this opulent
masterpiece.
```

The output.

Zero-Shot Prompting

In the context of prompting, **"zero-shot prompting"** is where we directly ask for the result without providing reference examples for the task. For many tasks, LLMs are smart enough to produce great results. This is exactly what we did in the examples above. Here's a new example where we ask an LLM to write a short poem about summer.

Copy

```
import openai

prompt_system = "You are a helpful assistant whose goal is to write short poems."

prompt = """Write a short poem about {topic}."""

response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
```

```
  messages=[
    {"role": "system", "content": prompt_system},
    {"role": "user", "content": prompt.format(topic="summer")}
  ]
)

print(response.choices[0]['message']['content'])
```

Code to run.

Copy

```
In the realm of golden rays,
Summer dances in perfect sway,
Nature's canvas aglow with hues,
Kissing warmth upon the dews.

Breezes whisper through the trees,
Serenading the humming bees,
Joyful laughter fills the air,
As sunshine gleams without a care.

Sand between our toes, so fine,
Waves crashing in rhythmic rhyme,
Picnics filled with sweet delight,
Summer's pleasures, pure and bright.

Days stretch long, nights invite,
Stargazing dreams take flight,
Fireflies dance in twilight's haze,
Summer's magic shall never fade.
```

The output.

The generated poem is nice, but what if we have a specific style of the poem we'd like it to generate? We could try the descriptive approach or simply provide relevant examples of what we need in the prompt.

In-Context Learning And Few-Shot Prompting

In the context of LLMs, **in-context learning** is a powerful approach where the model learns from demonstrations or exemplars provided within the prompt. **Few-shot prompting** is a technique under in-context learning that involves giving the language model a few examples or demonstrations of the task at hand to help it generalize and perform better on complex tasks.

Few-shot prompting allows language models to learn from a limited amount of data, making them more adaptable and capable of handling tasks with minimal training samples. Instead of relying solely on zero-shot capabilities (where the model predicts outputs for tasks it has never seen before), few-shot prompting leverages the in-context demonstrations to improve performance.

In few-shot prompting, the prompt typically includes multiple questions or inputs along with their corresponding answers. The language model learns from these examples and generalizes to respond to similar queries.

Copy

```
import openai


prompt_system = "You are a helpful assistant whose goal is to write short poems."


prompt = """Write a short poem about {topic}."""


examples = {
  "nature": "Birdsong fills the air,\nMountains high and valleys deep,\nNature's music sweet.",
  "winter": "Snow blankets the ground,\nSilence is the only sound,\nWinter's beauty found."
}


response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[
    {"role": "system", "content": prompt_system},
```

```
    {"role": "user", "content": prompt.format(topic="nature")},
    {"role": "assistant", "content": examples["nature"]},
    {"role": "user", "content": prompt.format(topic="winter")},
    {"role": "assistant", "content": examples["winter"]},
    {"role": "user", "content": prompt.format(topic="summer")}
  ]
)

print(response.choices[0]['message']['content'])
```

Code to run.

Copy

```
Golden sunbeams shine,
Warm sands between toes divine,
Summer memories, mine.
```

The output.

Limitations of Few-shot Prompting

Despite its effectiveness, few-shot prompting does have limitations, especially for more complex reasoning tasks. In such cases, advanced techniques like chain-of-thought prompting have gained popularity. Chain-of-thought prompting breaks down complex problems into multiple steps and provides demonstrations for each step, enabling the model to reason more effectively.

Conclusion

In this lesson, we explored prompting in the context of language models. Prompting involves presenting specific tasks or instructions to an LLM to generate desired responses. We learned that the quality of results largely depends on the precision and relevance of the information provided in the prompt.

Through code examples, we saw how to use prompts for story generation and product descriptions. We also explored zero-shot prompting, where the model can perform tasks without explicit reference examples. However, we introduced few-shot prompting, a powerful in-context learning approach to improve the model's performance on more complex tasks. Few-shot prompting allows the model to learn from a limited number of examples, making it more adaptable and capable of handling tasks with minimal training data.

However, we also recognized that few-shot prompting has its limitations, particularly for complex reasoning tasks. In such cases, advanced techniques like chain-of-thought

prompting are gaining popularity by breaking down complex problems into multiple steps with demonstrations for each step.

Pretraining and Fine-Tuning of LLMs

Introduction

This lesson will explore how pretrained LLMs learn from vast amounts of text, becoming great at language tasks. Then, we'll discover the power of finetuning, a process that molds these models into specialized experts, enabling them to tackle complex tasks. We'll also cover instruction finetuning, where we guide the models with explicit instructions, making them versatile and responsive to our needs. This lesson introduces several fine-tuning techniques we will use to finetune models later in the course.

Pretraining LLMs

Pretrained LLMs have catalyzed a paradigm shift in AI. These models are trained on massive text corpora sourced from the Internet, honing their linguistic knowledge through the prediction of the following words within sentences. By training on billions of sentences, these models acquire an excellent grasp of grammar, context, and semantics, enabling them to capture the nuances of language effectively.

Aside from being good at generating text, pretrained LLMs are also good at other tasks, as was found in 2020 with the GPT3 paper "Language Models are Few-Shot Learners." The paper showed that big enough LLMs are "few-shot learners"; that is, they are able to perform other tasks aside from text generation with the help of just a few examples of that task (hence the name "few-shot learners"). With those examples, the LLM is able to understand the logic behind what the user wants.

This was a huge step forward in the field, where each NLP task had very different models for each task. Now, a single model can do several of them and do them well.

The Power of Finetuning

Finetuning complements pretraining for specialized tasks.

While pretrained LLMs are undeniably impressive, their true potential is unlocked through finetuning. Although pretrained models possess a deep understanding of language, they require further adaptation to excel in complex tasks. For example, if the task is to answer questions about medical texts, the model would be finetuned on a dataset of medical question-answer pairs.

Finetuning helps these models become specialized. Finetuning exposes pretrained models to task-specific datasets, enabling them to recalibrate internal parameters and representations to align with the intended task. This adaptation enhances their ability to handle domain-specific challenges effectively.

The necessity for finetuning arises from the inherent non-specificity of pretrained models. While they possess a wide-ranging grasp of language, they lack task-specific context. For instance, finetuning is essential when tackling sentiment analysis of financial news.

In the early days of GPT3 in 2020 and 2021, finetuning also allowed an LLM to be tuned for a specific task without the need for multiple few-shot examples in the prompt.

Instruction Finetuning: Making General-Purpose Assistants out of LLMs

Instruction finetuning adds precise control over model behavior, making it a general-purpose assistant. The goal of instruction finetuning is to obtain an LLM that interprets prompts as instructions instead of text. It's just a special type of finetuning.
For example, consider the following prompt.
What is the capital of France?

An LLM with instruction finetuning would likely interpret the prompt as an instruction, giving the following answer.
Paris.

However, a plain LLM without instruction finetuning could think that we are writing a list of exercises for our geography students, therefore merely continuing the text with a new question.
What is the capital of Italy?

Instruction finetuning takes things up a notch. Imagine giving precise instructions to our model: "Analyze the sentiment of this text and tell us if it's positive." It's like coaching the model to paint exactly what you envision. With instruction finetuning, we provide explicit guidance, shaping the model's behavior to match our intentions.
Instruction tuning offers several advantages. It trains models on a collection of tasks described via instructions, granting LLMs the capacity to generalize to new tasks prompted by additional instructions. This sidesteps the need for vast amounts of task-specific data and instead uses textual instructions to guide learning.
While traditional finetuning acquaints models with task-specific data, instruction finetuning adds an extra layer by incorporating explicit instructions to guide model behavior. This approach empowers developers to shape desired outputs, encourage specific behaviors, and steer model responses.

Finetuning Techniques

There are several finetuning methods. We'll learn more about them later in the course.
There are multiple methods in fine-tuning with a focus on the number of parameters, such as:

- **Full Finetuning:** This method is based on adjusting all the parameters in the pretrained LLM models in order to adapt to a specific task. However, this method is relatively resource-intensive, requiring extensive computational power.
- **Low-Rank Adaptation (LoRA):** LoRA 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.

Multiple methods are focusing on the learning algorithm used for finetuning, such as:

- **Supervised Finetuning (SFT):** SFT involves doing standard supervised finetuning with a pretrained LLM on a small amount of demonstration data.

- **Reinforcement Learning from Human Feedback (RLHF):** RLHF is a training methodology where models are trained to follow human feedback over multiple iterations.

Later in this course, we'll see how to finetune a model using SFT and RLHF, both using LoRA.

Conclusion

In this lesson, we covered the pretraining and finetuning of LLMs. Pretraining equips LLMs with a profound grasp of language by immersing them in vast text corpora.
Finetuning then bridges the gap between general understanding and specialized knowledge, allowing LLMs to perform well in specialized domains. Instruction finetuning makes LLMs become versatile assistants, enabling precise control over their behavior through explicit guidance.
From full finetuning to the resource-efficient Low-Rank Adaptation (LoRA) and from Supervised Finetuning (SFT) to Reinforcement Learning from Human Feedback (RLHF), we also learned about the most popular finetuning techniques.
In the next module, we'll do some hands-on exercises up to launch a pretraining of an LLM on the cloud.

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 specialised 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.
Copy

```
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

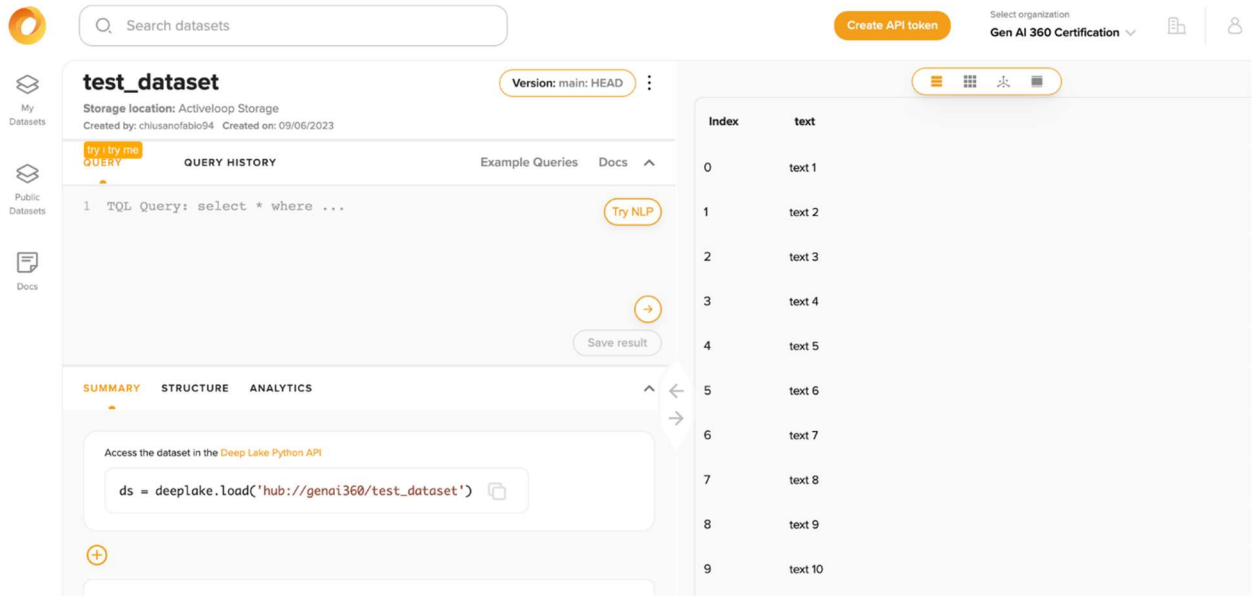
your username on Activeloop. You can easily find it in the URL of your webpage, which should have the form https://app.activeloop.ai/<YOUR_ACTIVELOOP_USERNAME>/home.
Copy

```
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.
Copy

```
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.

Copy

```
# 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.

Copy

```
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.

Copy

```
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.

Copy

```
# 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.

Copy

```
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.
Copy

```
ds_view = ds.query("select * where contains(text, '1')")


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

Copy

```
Batch 0
Sample 0: text 1
Sample 1: text 10
```

Now, we can save our dataset view as follows.
Copy

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

And we can read from it as follows.
Copy

```
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.

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

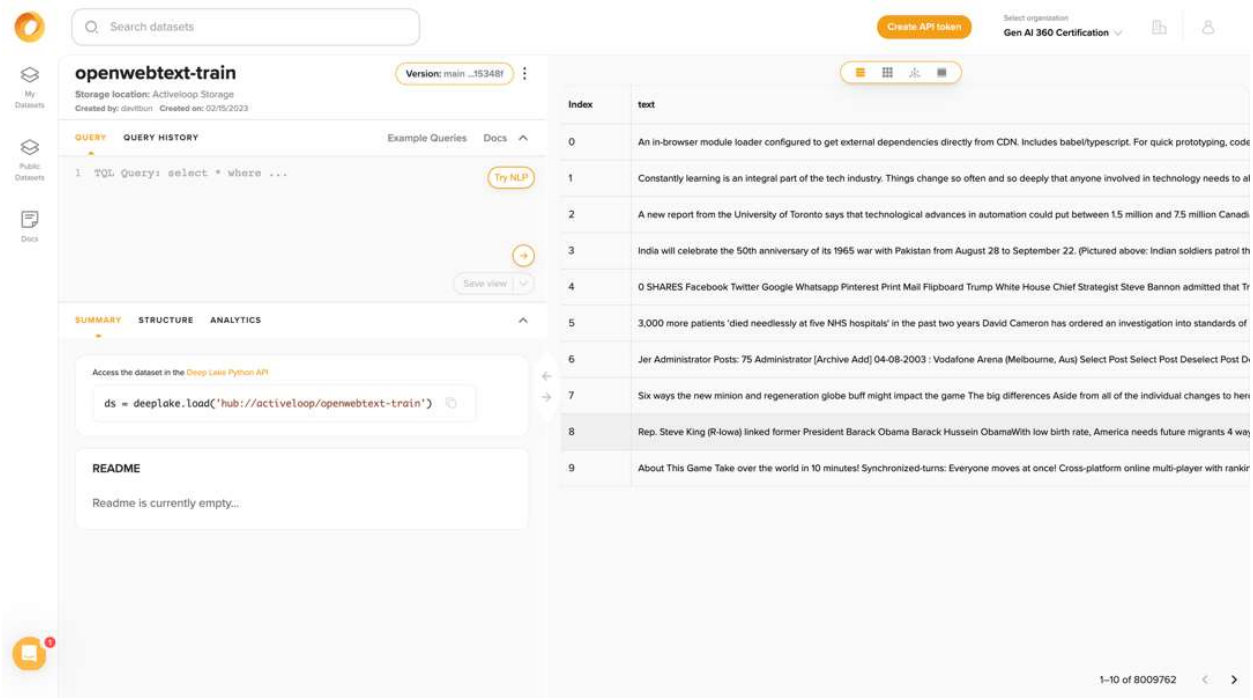
As we're going to spend a lot of money in training our LLM and ensure that everything is progressing smoothly, we'll log the training metrics to Weights and Biases, allowing us to see the metrics in real time in a suitable dashboard.

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.

Copy

```
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.

Copy

```
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.
Copy

```
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.
Copy

```
# 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.
Copy

```
from transformers import AutoConfig


config = AutoConfig.from_pretrained("gpt2")
print(config)
```

The sample code.

Copy

```
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.
Copy

```
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.

Copy

```
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.
Copy

```
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.
Copy

```
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.

Copy

```
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.
Copy

```
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.
Copy

```
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.
Copy

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


trainer.train()
```

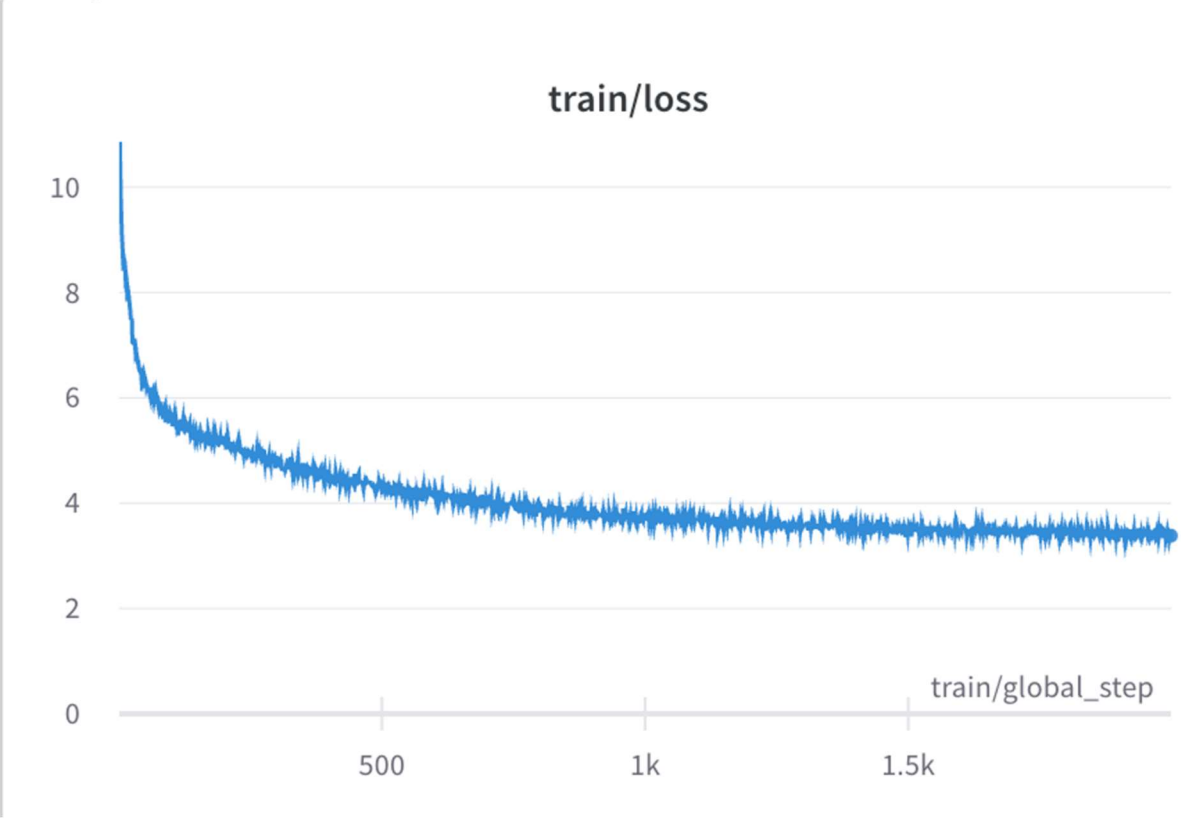
The sample code.

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.

Copy

```
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.
Copy

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


completion = pipe(txt, num_return_sequences=1)
print(completion)
```

The sample code.

Copy

```
[{'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.

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

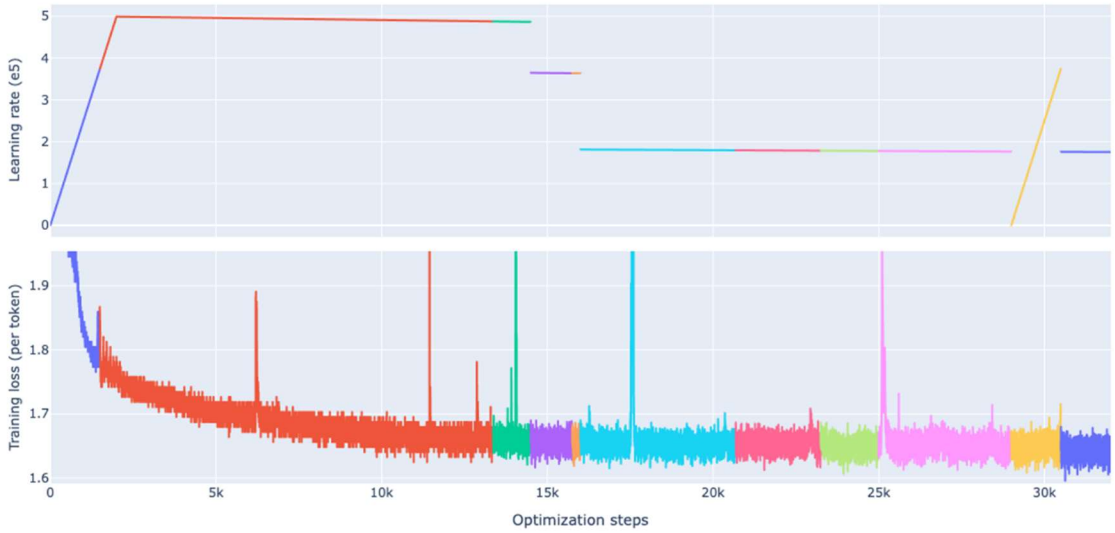
The Hugging Face Accelerate library allows you to leverage DeepSpeed's ZeRO features by making very few code changes. By using Accelerate and DeepSpeed ZeRO, we can significantly increase the maximum batch size that our hardware can handle without running into OOM errors.

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.

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.

Copy

```
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.

Copy

```
from lm_eval import tasks


print(tasks.ALL_TASKS)
```

The sample code.

Copy

```
['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.)

Copy

```
python main.py \
  --model hf-causal \
```

```
  --model_args pretrained=facebook/opt-1.3b \
  --tasks hellaswag \
  --device cuda:0
```

The sample code.

Copy

```
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|
```

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.
Copy

```
export OPENAI_API_SECRET_KEY=YOUR_KEY_HERE


python main.py \
  --model gpt3 \
  --model_args engine=davinci \
  --tasks hellaswag
```

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.

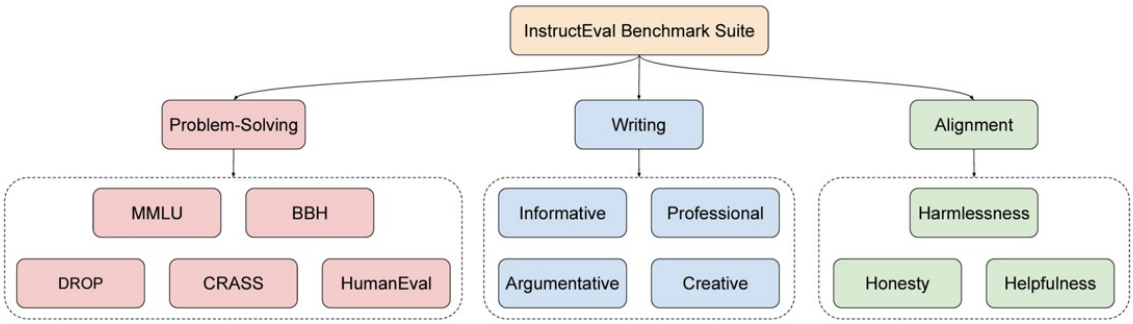


Figure 1: Overview of INSTRUCTEVAL, our holistic evaluation suite for Instructed LLMs

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.

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.

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 backpropagating 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.

Star on GitHub

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.

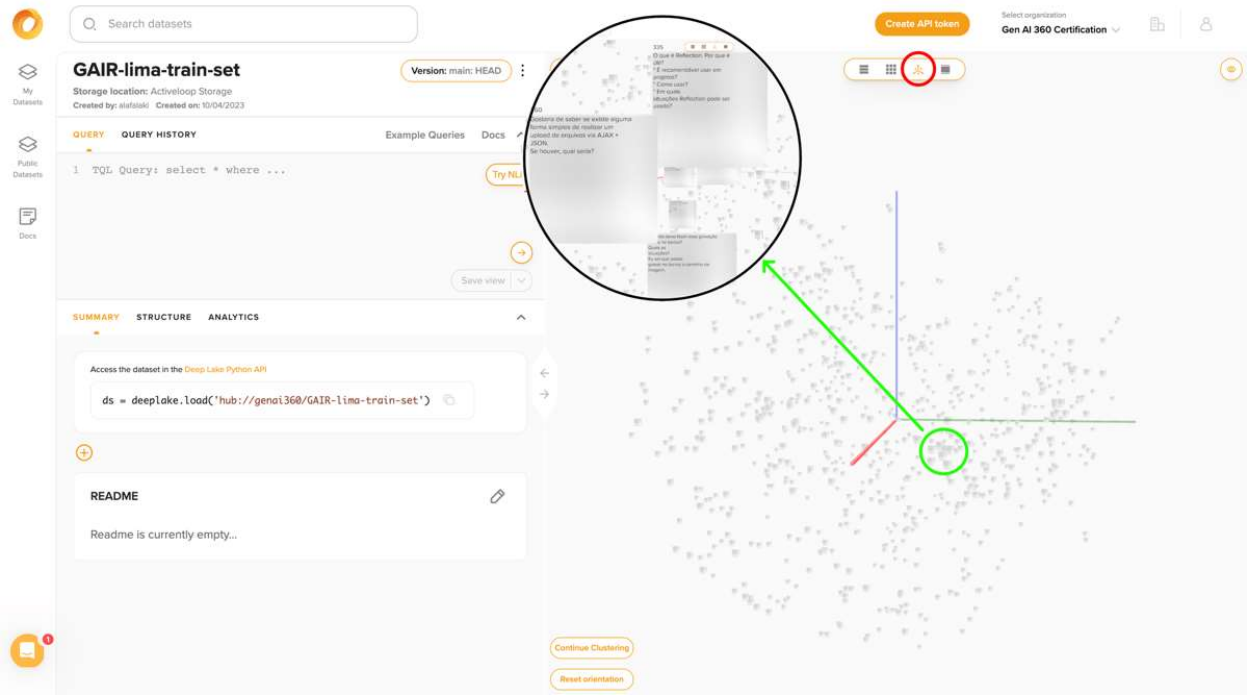
⚠️

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.
Copy

```
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)
```

The sample code.

Copy

```
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.
Copy

```
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.
Copy

```
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.
Copy

```
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)
```

The sample code.

Copy

```
{'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.
Copy

```
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.
Copy

```
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.
Copy

```
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.
Copy

```
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.
Copy

```
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,
)
```

The sample code.

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.

Copy

```
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) )
```

The sample code.

Copy

```
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.
Copy

```
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.
Copy

```
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.
Copy

```
from peft import PeftModel


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

The sample code.

Copy

```
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.

Copy

```
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.

Copy

```
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.

Copy

```
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. You can toggle the outputs by clicking on the right arrow icon.

▸

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.

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.

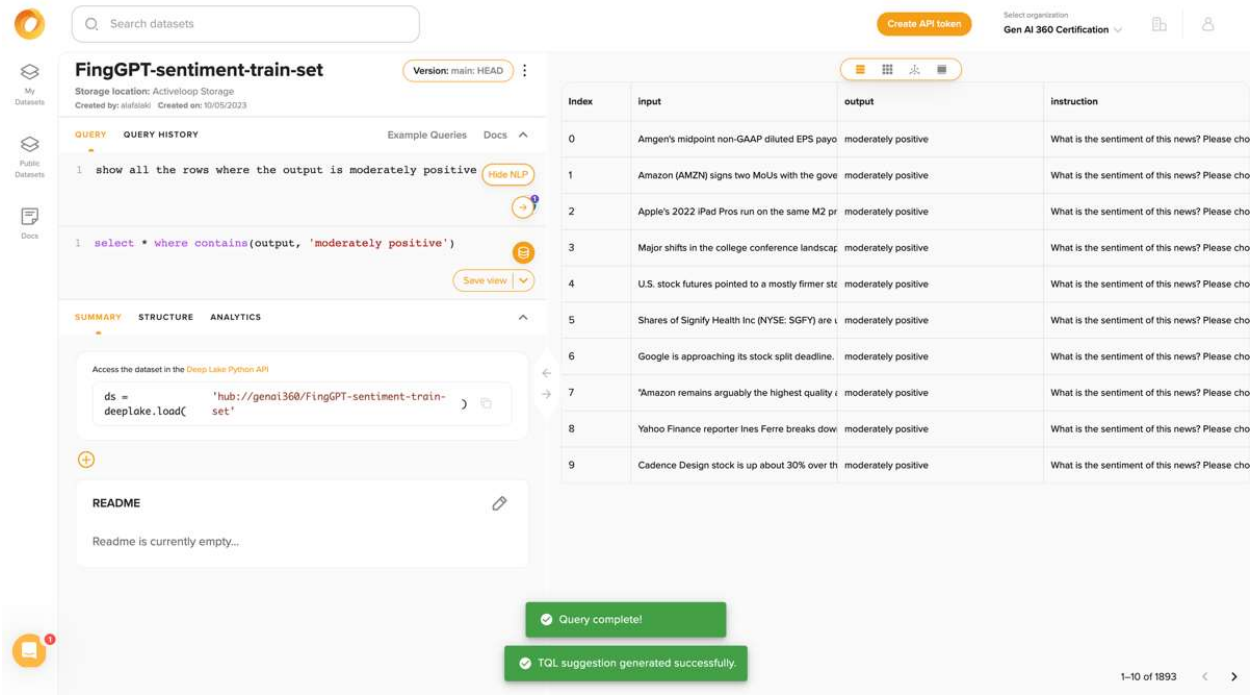
⚠️

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.

Copy

```
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.

Copy

```
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.
Copy

```
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.
Copy

```
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.
Copy

```
# 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.

Copy

```
{'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.

Copy

```
# 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.
Copy

```
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.
Copy

```
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.
Copy

```
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")
```

The sample code.

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.
Copy

```
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]) )
```

The sample code.

Copy

```
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 $300 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.

Improving LLMs with RLHF Module

Improving LLMs with RLHF

Goals: Equip students with knowledge and practical skills for implementing RLHF.

This short module is focussed on Reinforcement Learning from Human Feedback (RLHF). We learn how to incorporate human feedback into the training process through a reward model that learns the desired patterns to amplify the model's output.

Deep Dive into RLHF: This lesson explores the mechanics and applications of RLHF. We will provide a robust understanding of how RLHF functions and its significance in LLM training and optimization.

Improving trained models with RLHF: This lesson provides a practical guide on RLHF as a fine-tuning technique for LLMs. We build upon our previous fine tuning example by implementing RLHF.

Deep Dive into RLHF

Introduction

In this lesson, we will dive deeper into Reinforcement Learning from Human Feedback (RLHF), a method that combines human feedback and reinforcement learning to enhance the alignment and efficiency of Large Language Models.

We explore the RLHF training process, compare it with Supervised Fine-Tuning (SFT), and discuss its alternatives, such as Direct Preference Optimization (DPO) and Reinforced Self-Training (ReST).

By the end of this lesson, you'll have a comprehensive understanding of how RLHF and its alternatives are used to improve the performance and safety of LLMs.

Understanding RLHF

Reinforcement Learning from Human Feedback (RLHF) is a method that integrates human feedback and reinforcement learning into LLMs, enhancing their alignment with human objectives and improving their efficiency.

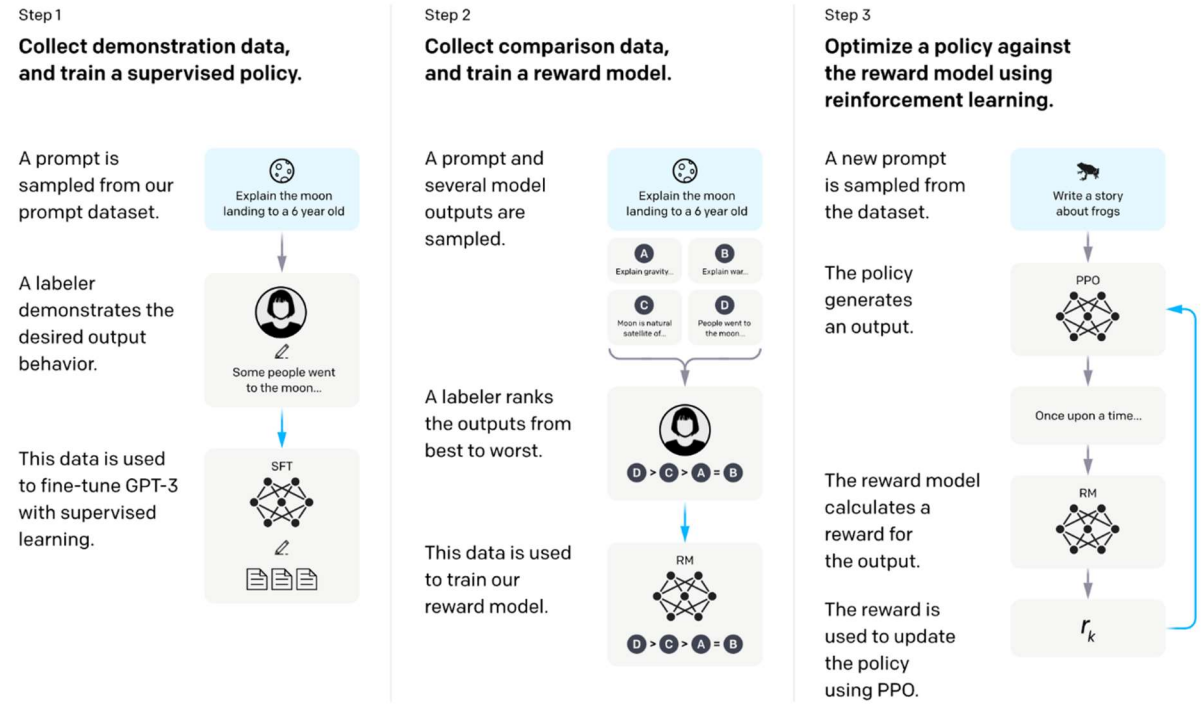
RLHF has shown significant promise in making LLMs safer and more helpful. It was used for the first time for creating InstructGPT, a finetuned version of GPT3 for following instructions, and it's used nowadays in the last OpenAI models ChatGPT (GPT-3.5-turbo) and GPT-4.

RLHF leverages human-curated rankings that act as a signal to the model, directing it to favor specific outputs over others, thereby encouraging the production of more reliable, secure responses that align with human expectations. All of this is done with the help of a reinforcement learning algorithm, namely PPO, that optimizes the underlying LLM model, leveraging the human-curated rankings.

RLHF Training Process

RLHF can be useful in guiding LLMs to generate appropriate texts by treating text generation as a reinforcement learning problem. In this approach, the language model

serves as the RL agent, the possible language outputs represent the action space, and the reward is based on how well the LLM's response aligns with the context of the application and the user's intent.
RLHF must be done on an already pretrained LLM. A language model must be trained in advance on a large corpus of text data collected from the internet.
The RLHF training process can then be broken down into the following steps.
(Optional) Finetune the LLM by following instructions: This is an optional step, but some sources recommend fine-tuning raw LLM in advance by following instructions, using a specialized dataset for it. This step should make the following RL finetuning of RLHF converge faster.
RLHF dataset creation: The LLM is used to generate a lot of text completions from a set of instructions. For each instruction, we collect multiple completions from the model.
Collecting human feedback: Human labelers then rank the generated completions to the same instruction from best to worst. Humans can be asked to rank the completions, keeping into account several aspects, such as completeness, relevancy, accuracy, toxicity, bias, etc. It's possible to convert these ranks into scores that we can assign to the text completions in our dataset, where a high score means that the completion is good.
Training a Reward Model: The RLHF dataset is used to train a reward model, which means a model that, when provided with an instruction and a text completion, assigns a score to the completion. In this context, a high score indicates that the completion is good. The reward model does a very similar job to what human labelers did on the dataset. The reward model is expected to learn, from the RLHF dataset, how to assign scores according to all the aspects taken into account during the labeling process (completeness, relevancy, accuracy, toxicity, bias, etc.).
Fine-tuning the Language Model with Reinforcement Learning and the Reward Model: Starting from a random instruction, our pretrained LLM generates multiple completions. These completions are then assigned scores by the reward model, and these scores are utilized by a reinforcement learning algorithm (PPO) to update the parameters of the LLM. This process aims to make the LLM more likely to produce completions with higher scores. To prevent the LLM from forgetting helpful information during fine-tuning, the RLHF fine-tuning process also aims to maintain a small Kullback-Leibler (KL) divergence between the fine-tuned LLM and the original LLM. This ensures that the token distribution predicted by it remains relatively consistent. After repeating this process for several iterations, we will have our final, finalized LLM.



RLHF vs SFT

As seen in the previous lessons, aligning LLM to follow instructions with human values is possible by doing simple SFT (with or without LoRA) with a high-quality dataset (see the LIMA paper). So, what's the tradeoff between RLHF and SFT?

In reality, it's still an open question. Empirically, it seems that RLHF can better teach the "human alignment" aspects of its dataset if it's sufficiently large and of high quality. However, in contrast, it's more expensive and time-consuming. Reinforcement learning, in this context, is still quite unstable, meaning that the results are very sensitive to the initial model parameters and training hyperparameters. It often falls into local optima, and the loss diverges multiple times, necessitating multiple restarts. This makes it less straightforward than plain SFT with LoRA.

Alternatives to RLHF

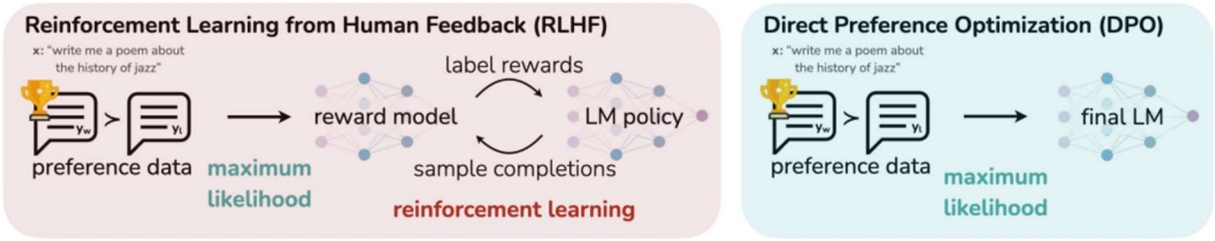
Over time, several alternatives to RLFH have been researched. Here are the most popular of them.

Direct Preference Optimization

Direct Preference Optimization (DPO) is a novel method for finetuning LLMs as an alternative to RLHF.

Unlike RLHF, which requires complex reward functions and careful balance to ensure sensible text generation, DPO simplifies the process by directly optimizing the language model using a binary cross-entropy loss. It bypasses the need for a reward model and RL-based optimization. Instead, it directly optimizes the language model on preference data. This is accomplished through an analytical mapping from the reward function to the optimal RL policy. It involves directly transforming the RL loss, which typically involves the reward and reference models, into a loss over the reference model.

As a result, DPO potentially simplifies the fine-tuning process of LLMs by eliminating the need for complex RL techniques or a reward model.

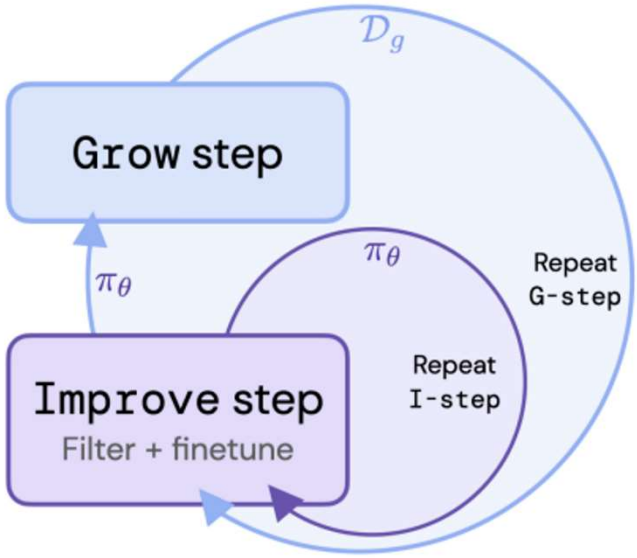


Reinforced Self-Training

Google DeepMind's Reinforced Self-Training (ReST) is a more cost-effective alternative to Reinforcement Learning from Human Feedback. The ReST algorithm operates in a cyclical manner, involving two main steps that are repeated iteratively.

The first step, referred to as the 'Grow' step, involves the use of an LLM to generate multiple output predictions for each context. These predictions are then used to augment a training dataset.

Following this, the 'Improve' step comes into play. In this phase, the augmented dataset is ranked and filtered using a reward model that has been trained based on human preferences. Subsequently, the LLM is fine-tuned on this filtered dataset using an offline reinforcement learning objective. The fine-tuned LLM is then used in the subsequent Grow step.

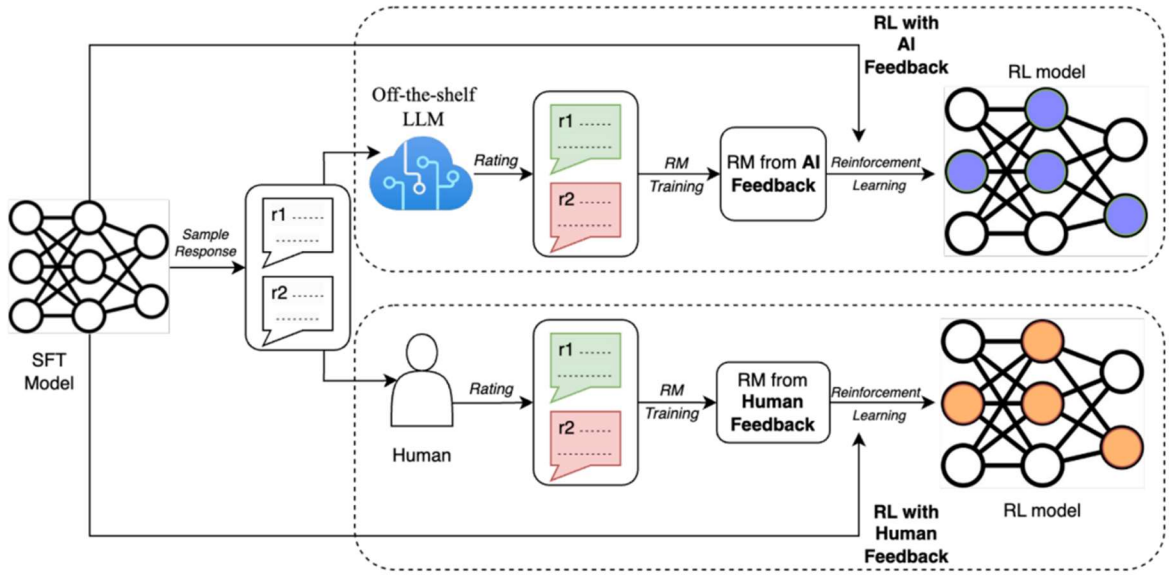


The ReST methodology offers several advantages over RLHF.

It significantly reduces the computational load compared to online reinforcement learning. This is achieved by leveraging the output of the Grow step across multiple Improve steps.

The quality of the policy is not limited by the quality of the original dataset, as is the case with offline reinforcement learning. This is because new training data is sampled from an improved policy during the Grow step.

Decoupling the Grow and Improve steps allows for easy inspection of data quality and potential diagnosis of alignment issues, such as reward hacking.
The ReST approach is straightforward and stable and only requires tuning a small number of hyperparameters, making it a user-friendly and efficient tool in the machine learning toolkit.
Reinforcement Learning from AI Feedback (RLAIF)
Another innovative alternative to RLHF is Reinforcement Learning from AI Feedback (RLAIF). Developed by Anthropic, RLAIF aims to address some of the limitations of RLHF, particularly concerning the subjectivity and scalability of human feedback.
In RLAIF, instead of relying on human feedback, an AI Feedback Model is used to provide feedback for training the AI assistant. This Feedback Model is guided by a constitution provided by humans, outlining the essential principles for the model's judgment. This approach allows for a more objective and scalable supervision technique, as it is not dependent on a small pool of human preferences.
The RLAIF process begins with the creation of a dataset of ranked preferences generated automatically by the AI Feedback Model. This dataset is then used to train a Reward Model similar to RLHF. The Reward Model serves as the reward signal in a reinforcement learning schema for an LLM.



RLAIF offers several advantages over RLHF. Firstly, it maintains the helpfulness of RLHF models while making improvements in terms of harmlessness. Secondly, it reduces subjectivity as the AI assistant's behavior is not solely dependent on a small pool of humans and their particular preferences. Lastly, RLAIF is significantly more scalable as a supervision technique, making it a promising alternative for the future development of safer and more efficient LLMs.
A recent paper from Google did more experiments with RLAIF and found that humans prefer both RLAIF and RLHF to standard SFT at almost equal rates, indicating that they could be alternatives.
Conclusion

This lesson provided a more in-depth exploration of Reinforcement Learning from Human Feedback, a method that combines human feedback and reinforcement learning to enhance the performance and safety of Large Language Models.
We covered the RLHF training process, highlighting its steps and how it leverages human-curated rankings and reinforcement learning to finetune the LLM. We also compared RLHF with Supervised Fine-Tuning (SFT), discussing the trade-offs between the two.
Furthermore, we explored alternatives to RLHF, such as Direct Preference Optimization (DPO) and Reinforced Self-Training (ReST), which offer different approaches to fine-tuning LLMs.
As we continue to refine these techniques, we move closer to our goal of creating LLMs that are more aligned with human values, efficient, and safer to use.

Improving Trained Models with RLHF

Introduction

As previously stated, the RLHF process involves incorporating human feedback into the training process through a reward model that learns the desired patterns to amplify the model's output. For instance, if the goal is to enhance politeness, the reward model will guide the model to generate more polite responses by assigning higher scores to polite outputs. This process can be resource-intensive due to the need to train an additional reward model using a dataset curated by humans, which can be costly. Nevertheless, we will leverage available open-source models and datasets whenever feasible to explore the technique thoroughly while maintaining acceptable costs.
It is recommended to begin the procedure by conducting a supervised fine-tuning phase, which enables the model to adjust to a conversational manner. This procedure can be accomplished by using the SFTTrainer class. The next phase involves training a reward model with the desired traits using the RewardTrainer class in section 2. Finally, the Reinforcement Learning phase employs the models from the preceding steps to construct the ultimate aligned model, utilizing the PPOTrainer class in section 3.
After each subsection, the fine-tuned models, the reports generated from the weights and biases, and the file detailing the requirements for the library can be accessed. Note that different steps might necessitate distinct versions of libraries. We employed the OPT-1.3B model as the foundational model and fine-tuned the DeBERTa (300M) model as the reward model for our experiments. While these are more compact models and might not incorporate the insights of recent larger models like GPT-4 and LLaMA2, the procedure we are exploring in this tutorial can be readily applied to other existing networks by simply modifying the model key name in the code.
We'll be using a set of 8x100 A100 GPUs in this lesson.

GPU Cloud - Lambda

In this lesson, we'll leverage Lambda, Lambda, the GPU cloud designed by ML engineers for training LLMs & Generative AI, for renting GPUs. We can create an account on it, link a billing account, and rent one instance of the following GPU servers with associated costs. The cost

is for the time your instance is up and not only for the time you're training your model, so remember to turn the instance off. For this lesson, we rented an 8x NVIDIA A100 instance comprising 40GB of memory at the price of $8.80/h.

⚠

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

To ensure everything is progressing smoothly, we'll log the training metrics to Weights & Biases, allowing us to see the metrics in real-time in a suitable dashboard.

1. Supervised Fine-Tuning

We thoroughly covered the SFT phase in the previous lessons. If any steps are unclear, refer to the previous lessons.

In this tutorial, the differences are in how we use a distinct dataset called OpenOrca and in applying the QLoRA method, which we will elaborate on in the subsequent sections.

The OpenOrca dataset comprises 1 million interactions with the language model extracted from the original OpenOrca dataset. Each interaction in this collection is comprised of a question paired with a corresponding response. This phase aims to familiarize the model with the conversational structure, thereby teaching it to answer questions rather than relying on its standard auto-completion mechanism.

Begin the process by installing the necessary libraries.

Copy

```
pip install -q transformers==4.32.0 bitsandbytes==0.41.1 accelerate==0.22.0 deeplake==3.6.19 trl==0.5.0 peft==0.5.0 wandb==0.15.8
```

The sample code.

1.1. The Dataset

The initial phase involves streaming the dataset via the Activeloop's performant dataloader to facilitate convenient accessibility. As previously indicated, we employ a subset of the original dataset containing 1 million data points. Nevertheless, the complete dataset (4 million) is accessible at this URL.

Copy

```
import deeplake

# Connect to the training and testing datasets
ds = deeplake.load('hub://genai360/OpenOrca-1M-train-set')
ds_valid = deeplake.load('hub://genai360/OpenOrca-1M-valid-set')

print(ds)
```

The sample code.

Copy

```
Dataset(path='hub://genai360/OpenOrca-1M-train-set', read_only=True, tensors=['id',
'question', 'response', 'system_prompt'])
```

The output.

The dataset has three significant columns. These encompass question, also referred to as prompts, which are the queries we have from the LLM: response, i.e., the model's output or answers to the questions, and finally, system_prompt, i.e., the initial directives guiding the model in establishing its context, such as "you are a helpful assistant."

For simplicity, we exclusively utilize the initial two columns. It could also be beneficial to incorporate the system prompts while formatting the text. This template formats the text in the structure of Question: xxx\n\nAnswer: yyy, where the question-and-answer sections are divided by two newline characters. There's room for experimentation with diverse formats, like trying out System: xxxnnQuestion: yyynnAnswer: zzz, to effectively integrate the system prompts from the dataset.

Copy

```
def prepare_sample_text(example):
   """Prepare the text from a sample of the dataset."""
   text = f"Question: {example['question'][0]}\n\nAnswer: {example['response'][0]}"
   return text
```

The sample code.

Moving forward, the next step is loading the OPT model tokenizer.

Copy

```
from transformers import AutoTokenizer

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

The output.

Next, the ConstantLengthDataset class will come into play, serving to aggregate samples to maximize utilization within the 2K input size constraint and enhance the efficiency of the training process.

Copy

```
from trl.trainer import ConstantLengthDataset

train_dataset = ConstantLengthDataset(
   tokenizer,
   ds,
   formatting_func=prepare_sample_text,
   infinite=True,
   seq_length=2048
)

eval_dataset = ConstantLengthDataset(
   tokenizer,
   ds_valid,
   formatting_func=prepare_sample_text,
   seq_length=1024
```

```
)

iterator = iter(train_dataset)
sample = next(iterator)
print(sample)

train_dataset.start_iteration = 0
```
The sample code.
Copy
```
{'input_ids': tensor([ 16, 358, 828,  ..., 137,  79, 362]), 'labels': tensor([ 16, 358, 828,  ..., 137,
79, 362])}
```
The output.
1.2. Initialize the Model and Trainer
Finally, we initialize the model and apply LoRA, which effectively keeps memory requirements low when fine-tuning a large language model.
Copy
```
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.
Now, we instantiate the TrainingArguments, which define the hyperparameters governing the training loop.
Copy
```
from transformers import TrainingArguments

training_args = TrainingArguments(
   output_dir="./OPT-fine_tuned-OpenOrca",
   dataloader_drop_last=True,
   evaluation_strategy="steps",
   save_strategy="steps",
   num_train_epochs=2,
   eval_steps=2000,
   save_steps=2000,
   logging_steps=1,
   per_device_train_batch_size=8,
   per_device_eval_batch_size=8,
   learning_rate=1e-4,
   lr_scheduler_type="cosine",
```

```
    warmup_steps=100,
    gradient_accumulation_steps=1,
    bf16=True,
    weight_decay=0.05,
    ddp_find_unused_parameters=False,
    run_name="OPT-fine_tuned-OpenOrca",
    report_to="wandb",
)
```

The sample code.

Now, we are using the BitsAndBytes library to execute the quantization process and load the model in a 4-bit format. We will employ the NF4 data type designed for weights and implement the nested quantization approach, which effectively reduces memory usage with negligible decline in performance. Finally, we indicate that the training process computations should be carried out using the bfloat16 format.

The QLoRA method is a recent approach that combines the LoRA technique with quantization to reduce memory usage. When loading the model, it's necessary to provide the quantization_config.

Copy

```
import torch
from transformers import BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(
  load_in_4bit=True,
  bnb_4bit_quant_type="nf4",
  bnb_4bit_use_double_quant=True,
  bnb_4bit_compute_dtype=torch.bfloat16
)
```

The sample code.

The following code segment utilizes the AutoModelForCasualLM class to load the pre-trained weights of the OPT model, which holds 1.3 billion parameters. It's important to note that a GPU is required in order to make use of this capability.

Copy

```
from transformers import AutoModelForCausalLM
from accelerate import Accelerator

model = AutoModelForCausalLM.from_pretrained(
  "facebook/opt-1.3b",
        quantization_config=quantization_config,
        device_map={"": Accelerator().process_index}
)
```

The sample code.

Prior to initializing the trainer object, we introduce modifications to the model architecture for efficiency. This involves casting specific layers of the model to full precision (32 bits), including LayerNorms and the final language modeling head.

Copy

```
from torch import nn

for param in model.parameters():
  param.requires_grad = False
  if param.ndim == 1:
    param.data = param.data.to(torch.float32)

model.gradient_checkpointing_enable()
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)
```

Lastly, The SFTTrainer class will utilize the initialized dataset and model, in combination with the training arguments and LoRA technique, to start the training process.

Copy

```
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.

The SFTTrainer instance will automatically create checkpoints during the training process, as specified by the save_steps parameter, and store them in the ./OPT-fine_tuned-OpenOrca directory.

We're required to merge the LoRA layers with the base model to form a standalone network, a procedure outlined earlier. The subsequent section of code will handle the merging process.

Copy

```
from transformers import AutoModelForCausalLM
import torch

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

```
from peft import PeftModel

# Load the Lora model
model = PeftModel.from_pretrained(model, "./OPT-fine_tuned-OpenOrca/<step>")
model.eval()

model = model.merge_and_unload()

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

The standalone model will be accessible on the ./OPT-supervised_fine_tuned/merged directory. This checkpoint will come into play in Section 3.

▸

Resources

2. Training a Reward Model

The second step is training a reward model, which learns human preferences from labeled samples and guides the LLM during the final step of the RLHF process. This model will be provided with samples of favored behavior compared to not expected or desired behavior. The reward model will learn to imitate human preferences by assigning higher scores to samples that align with those preferences.

The reward models essentially perform a classification task, where they select the superior choice from a pair of sample interactions using input from human feedback. Various network types can be used and trained to function as reward models. Conversations are focused on the idea that the reward model should match the size of the base model in order to possess sufficient knowledge for practical guidance. Nonetheless, smaller models like DeBERTa or RoBERTa proved to be effective. Indeed, with enough resources available, experimenting with larger models is great. Both these models need to be loaded in the next phase of RLHF, which is Reinforcement Learning.

Begin the process by installing the necessary libraries. (We use a different version of the TRL library just in this subsection.)

Copy

```
pip install -q transformers==4.32.0 deeplake==3.6.19 sentencepiece==0.1.99 trl==0.6.0
```

The sample code.

2.1. The Dataset

Please note that the datasets used in this step contain inappropriate language and offensive words. Nevertheless, this approach is the preferred way to align the model's behavior by exposing it to such language and instructing the model not to replicate it.

We will utilize the "helpfulness/harmless" (hh) dataset from Anthropic, specifically curated for the Reinforcement Learning from Human Feedback (RLHF) procedure (you can read more about it here). The Activeloop datasets hub provides access to a dataset that allows us to stream the content effortlessly using a single line of code. The subsequent code snippet will establish the data loader object for both the training and validation sets.

Copy

```
import deeplake

ds = deeplake.load('hub://genai360/Anthropic-hh-rlhf-train-set')
ds_valid = deeplake.load('hub://genai360/Anthropic-hh-rlhf-test-set')

print(ds)
```

The sample code.

Copy

```
Dataset(path='hub://genai360/Anthropic-hh-rlhf-train-set', read_only=True,
tensors=['chosen', 'rejected'])
```

The output.

Moving forward, we need to structure the dataset appropriately for the Trainer class. However, before that, let's load the pretrained tokenizer for the DeBERTa model we will use as the reward model. The code should be recognizable; the AutoTokenizer class will locate the suitable initializer class and utilize the .from_pretrained() method to load the pretrained tokenizer.

Copy

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-v3-base")
```

The sample code.

As presented in earlier lessons, the Dataset class in PyTorch is in charge of formatting the dataset for various downstream tasks. A pair of inputs is necessary to train a reward model. The first item will denote the chosen (favorable) conversation, while the second will represent a conversation rejected by labelers, which we aim to prevent the model from replicating. The concept revolves around the reward model, which assigns a higher score to the chosen sample while assigning lower rankings to the rejected ones. The code snippet below initially tokenizes the samples and then aggregates the pairs into a single Python dictionary.

Copy

```
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, dataset):
        self.dataset = dataset

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

    def __getitem__(self, idx):

        chosen = self.dataset.chosen[idx].text()
        rejected = self.dataset.rejected[idx].text()
```

```
    tokenized_chosen = tokenizer(chosen, truncation=True, max_length=max_length,
padding='max_length')
    tokenized_rejected = tokenizer(rejected, truncation=True, max_length=max_length,
padding='max_length')

    formatted_input = {
      "input_ids_chosen": tokenized_chosen["input_ids"],
      "attention_mask_chosen": tokenized_chosen["attention_mask"],
      "input_ids_rejected": tokenized_rejected["input_ids"],
      "attention_mask_rejected": tokenized_rejected["attention_mask"],
    }

    return formatted_input
```

The sample code.

The Trainer class anticipates receiving a dictionary containing four keys. This includes the tokenized forms for both chosen and rejected conversations (input_ids_chosen and input_ids_rejected) and their respective attention masks (attention_mask_chosen and attention_mask_rejected). As we employ a padding token to standardize input sizes (up to the model's maximum input size, 512 in this case), it's important to inform the model that certain tokens at the end don't contain meaningful information and can be disregarded. This is why attention masks are important.

We can create a dataset instance using the previously defined class. Additionally, we can extract a single row from the dataset using the iter and next methods to verify the output keys and confirm that everything functions as intended.

Copy

```
train_dataset = MyDataset(ds)
eval_dataset = MyDataset(ds_valid)

# Print one sample row
iterator = iter(train_dataset)
one_sample = next(iterator)
print(list(one_sample.keys()))
```

The sample code.

Copy

```
['input_ids_chosen', 'attention_mask_chosen', 'input_ids_rejected',
'attention_mask_rejected']
```

The output.

2.2. Initialize the Model and Trainer

The next steps are quite straightforward. We begin by loading the pretrained DeBERTa model using AutoModelForSequenceClassification, as our aim is to employ the network for a classification task. Specifying the number of labels (num_labels) as 1 is equally important, as we only require a single score to assess a sequence's quality. This score can indicate whether the content is aligned, receiving a high score, or, if it's unsuitable, receiving a low score.

Copy

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(
   "microsoft/deberta-v3-base", num_labels=1
)
```

The sample code.

Then, we can create an instance of TrainingArguments, setting the hyperparameters we intend to utilize. There is flexibility to explore various hyperparameters based on the selection of pre-trained models and available resources. For example, if an Out of Memory (OOM) error is encountered, a smaller batch size might be needed.

Copy

```
from transformers import TrainingArguments

training_args = TrainingArguments(
   output_dir="DeBERTa-reward-hh_rlhf",
   learning_rate=2e-5,
   per_device_train_batch_size=24,
   per_device_eval_batch_size=24,
   num_train_epochs=20,
   weight_decay=0.001,
   evaluation_strategy="steps",
   eval_steps=500,
   save_strategy="steps",
   save_steps=500,
   gradient_accumulation_steps=1,
   bf16=True,
   logging_strategy="steps",
   logging_steps=1,
   optim="adamw_hf",
   lr_scheduler_type="linear",
   ddp_find_unused_parameters=False,
   run_name="DeBERTa-reward-hh_rlhf",
   report_to="wandb",
)
```

The sample code.

Finally, the RewardTrainer class from the TRL library will tie everything together and execute the training loop. It's essential to provide the previously defined variables, such as the model, tokenizer, and dataset.

Copy

```
from trl import RewardTrainer

trainer = RewardTrainer(
   model=model,
   tokenizer=tokenizer,
```

```
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    max_length=max_length
)

trainer.train()
```

The sample code.

The trainer will automatically save the checkpoints, which can be utilized in the next and final steps.

▸

Resources

3. Reinforcement Learning (RL)

The final step of RLHF! This section will integrate the models we trained earlier. Specifically, we will utilize the previously trained reward model to further align the fine-tuned model with human feedback. In the training loop, a custom prompt will be employed to generate a response from the fine-tuned OPT. The reward model will then assign a score based on how closely the response resembles a hypothetical human-generated output. Within the RL process, mechanisms are also in place to ensure that the model retains its acquired knowledge and doesn't deviate too far from the original model's foundation. We will proceed by introducing the dataset, followed by a more detailed examination of the process in the subsequent subsections.

Begin the process by installing the necessary libraries.

Copy

```
pip install -q transformers==4.32.0 accelerate==0.22.0 peft==0.5.0 trl==0.5.0
bitsandbytes==0.41.1 deeplake==3.6.19 wandb==0.15.8 sentencepiece==0.1.99
```

The sample code.

3.1. The Dataset

Given that the process falls under the realm of unsupervised learning, we have the flexibility to choose the dataset for this step. Since the reward model assesses the output without relying on a label, the learning process does not require a question-answer pair. In this section, we will employ Alpaca's OpenOrca dataset, a subset of the OpenOrca dataset.

Copy

```
import deeplake

# Connect to the training and testing datasets
ds = deeplake.load('hub://genai360/Alpaca-OrcaChat')
print(ds)
```

The sample code.

Copy

```
Dataset(path='hub://genai360/Alpaca-OrcaChat', read_only=True, tensors=['input',
'instruction', 'output'])
```

The output.

The dataset comprises three columns: input, which denotes the user's prompt to the model; instruction, which represents the model's directive; and output, which holds the model's response. We only need to use the input feature for the RL process. Before defining a dataset class for proper formatting, it's necessary to load the pre-trained tokenizer corresponding to the fine-tuned model in the first section.

Copy

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b", padding_side='left')
```

The sample code.

In the subsequent subsection, the trainer requires both the query and its tokenized variant. Thus, the query will remain in text format, whereas the input_ids will represent the token IDs. The dataset class format should be recognizable by now. An important point to highlight is that the query variable acts as a template for crafting user prompts, structured as follows: Question: XXX\n\nAnswer: in alignment with the format employed during the supervised fine-tuning (SFT) step.

Copy

```
from torch.utils.data import Dataset

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

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

  def __getitem__(self, idx):

    query = "Question: " + self.ds.input[idx].text() + "\n\nAnswer: "
    tokenized_question = tokenizer(query, truncation=True, max_length=400,
padding='max_length', return_tensors="pt")

    formatted_input = {
      "query": query,
      "input_ids": tokenized_question["input_ids"][0],
    }

    return formatted_input

# Define the dataset object
myTrainingLoader = MyDataset(ds)
```

The sample code.

Additionally, we must establish a collator function responsible for transforming individual samples from the data loader into data batches. This function will later be passed to the Trainer class.

Copy

```
def collator(data):
   return dict((key, [d[key] for d in data]) for key in data[0])
```

The sample code.

3.2. Initialize the SFT Models

In this section, we are required to load two models. To begin, let's initiate the process by loading the fine-tuned model, referred to as OPT-supervised_fine_tuned in section 1, utilizing the configuration provided by the PPOConfig class. The majority of the parameters have been elaborated on in earlier lessons, except adapt_kl_ctrl and init_kl_coef. These arguments will be used to control the KL divergence penalty to ensure the model doesn't stray significantly from the pre-trained model. Otherwise, it runs the risk of generating nonsensical sentences.

Copy

```
from trl import PPOConfig

config = PPOConfig(
   task_name="OPT-RL-OrcaChat",
   steps=10_000,
   model_name="./OPT-fine_tuned-OpenOrca/merged",
   learning_rate=1.41e-5,
   batch_size=32,
   mini_batch_size=4,
   gradient_accumulation_steps=1,
   optimize_cuda_cache=True,
   early_stopping=False,
   target_kl=0.1,
   ppo_epochs=4,
   seed=0,
   init_kl_coef=0.2,
   adap_kl_ctrl=True,
   tracker_project_name="GenAI360",
   log_with="wandb",
)
```

The sample code.

We also need to use the set_seed() function to set the random state for reproducibility, and the current_device variable will store your current device id and will be used later on the code.

Copy

```
from trl import set_seed
from accelerate import Accelerator
```

```
# set seed before initializing value head for deterministic eval
set_seed(config.seed)

# Now let's build the model, the reference model, and the tokenizer.
current_device = Accelerator().local_process_index
```
The sample code.
The next three code blocks are used to load the supervised fine-tuned (SFT) model. It starts by setting the details for the LoRA to accelerate the fine-tuning process.
Copy
```
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.
The LoRA config can then be used alongside the AutoModelForCausalLMwithValueHead class to load the pre-trained weights. We utilize the load_in_8bit argument to load the model, employing a quantization technique that reduces weight precision. This helps conserve memory during model training. This model is intended for utilization within the RL loop.
Copy
```
from trl import AutoModelForCausalLMWithValueHead

model = AutoModelForCausalLMWithValueHead.from_pretrained(
    config.model_name,
    load_in_8bit=True,
    device_map={"": current_device},
    peft_config=lora_config,
)
```
The sample code.
3.3. Initialize the Reward Models
Utilizing the Huggingface pipeline feature makes loading the Reward model straightforward. We need to define the task we are undertaking. In this case, we selected sentiment-analysis, as our fundamental objective revolves around binary classification. Furthermore, it is essential to indicate the path to the pre-trained reward model from the previous section using the model parameter. Alternatively, it could be possible to utilize a model's name from the HuggingFace Hub if a pre-trained reward model is accessible.
The pipeline will automatically load the appropriate tokenizer, and we can initiate classification by providing any text to the defined object.
Copy

```
from transformers import pipeline
import torch

reward_pipeline = pipeline(
   "sentiment-analysis",
   model="./DeBERTa-v3-base-reward-hh_rlhf/checkpoint-1000",
   tokenizer="./DeBERTa-v3-base-reward-hh_rlhf/checkpoint-1000",
   device_map={"": current_device},
   model_kwargs={"load_in_8bit": True},
   return_token_type_ids=False,
)
```

The sample code.

The reward_pipe variable containing the reward model will be employed within the reinforcement learning (RL) training loop.

3.4. PPO Training

The final stage involves employing Proximal Policy Optimization (PPO) to enhance the stability of the training loop. It will restrict alterations to the model by preventing excessively large updates. Empirical findings indicate that introducing minor adjustments accelerates the convergence of the training process, it is one of the intuitions behind the PPO.

Before beginning the actual training loop, certain variables need to be defined for integration within the loop. Initially, we establish the output_length_sampler object, which draws samples from a specified range spanning from a defined minimum to a maximum number. We want the outputs to be in the range of 32 to 128 tokens.

Copy

```
from trl.core import LengthSampler

output_length_sampler = LengthSampler(32, 400) #(OutputMinLength, OutputMaxLength)
```

The sample code.

We need to define two sets of dictionaries that will control the generation process for fine-tuned and reward models. We have the ability to configure arguments that oversee the sampling procedure, truncation, and batch size for each respective network during inference. The code block was concluded by setting the save_freq variable, which determines the interval for checkpoint preservation.

Copy

```
sft_gen_kwargs = {
   "top_k": 0.0,
   "top_p": 1.0,
   "do_sample": True,
   "pad_token_id": tokenizer.pad_token_id,
   "eos_token_id": 100_000,
}

reward_gen_kwargs = {
```

```
    "top_k": None,
    "function_to_apply": "none",
    "batch_size": 16,
    "truncation": True,
    "max_length": 400
}

save_freq = 50
```

The sample code.

The final action before the actual training loop involves the instantiation of the PPO trainer object. The PPOTrainer class will take as input an instance of PPOConfig, which we defined earlier, the directory of the fine-tuned model from section 1, and the training dataset.

It's worth noting that we have the option to provide a reference model using the ref_model parameter, which will serve as the guide for the KL divergence penalty. In cases where this parameter is not specified, the trainer will default to using the original pre-trained model as the reference.

Copy

```
from trl import PPOTrainer

ppo_trainer = PPOTrainer(
    config,
    model,
    tokenizer=tokenizer,
    dataset=myTrainingLoader,
    data_collator=collator
)
```

The sample code.

Now, we can proceed to the last component, which is the training loop. The process begins with obtaining a single batch of samples and utilizing the input_ids, which are the formatted and tokenized prompts (refer to section 3.1) to generate responses using the fine-tuned model. Subsequently, these responses are decoded and combined with the prompt before being fed to the reward model. This allows the reward model to assess their proximity to a human-generated response by assigning scores.

Finally, the PPO object will adjust the model based on the scores by the reward model.

Copy

```
from tqdm import tqdm
tqdm.pandas()

for step, batch in tqdm(enumerate(ppo_trainer.dataloader)):
    if step >= config.total_ppo_epochs:
        break
    question_tensors = batch["input_ids"]

    response_tensors = ppo_trainer.generate(
```

```
        question_tensors,
        return_prompt=False,
        length_sampler=output_length_sampler,
        **sft_gen_kwargs,
    )
    batch["response"] = tokenizer.batch_decode(response_tensors,
skip_special_tokens=True)

    # Compute reward score
    texts = [q + r for q, r in zip(batch["query"], batch["response"])]
    pipe_outputs = reward_pipeline(texts, **reward_gen_kwargs)

    rewards = [torch.tensor(output[0]["score"]) for output in pipe_outputs]

    # Run PPO step
    stats = ppo_trainer.step(question_tensors, response_tensors, rewards)
    ppo_trainer.log_stats(stats, batch, rewards)

    if save_freq and step and step % save_freq == 0:
        print("Saving checkpoint.")
        ppo_trainer.save_pretrained(f"./OPT-RL-OrcaChat/checkpoint-{step}")
```

The sample code.

Remember to merge the LoRA adaptors with the base model to ensure that the network can be utilized independently as a standalone model in the future. Simply ensure to modify the directory of the saved checkpoint adaptor according to the results.

Copy

```
from transformers import AutoModelForCausalLM
import torch

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

from peft import PeftModel

# Load the Lora model
model = PeftModel.from_pretrained(model, "./OPT-RL-OrcaChat/checkpoint-400/")
model.eval();

model = model.merge_and_unload()

model.save_pretrained("./OPT-RL-OrcaChat/merged")
```

The sample code.

▸

Resources

QLoRA

Earlier, we employed an argument named load_in_8bit during the loading of the base model. This quantization technique significantly reduces the memory requirement when loading large models. A 32-bit floating-point format was utilized for model training in the early stages of neural network development. This entailed the representation of each weight using 32 bits, requiring 4 bytes for storage per weight.

Researchers developed diverse methods to mitigate this constraint with the growth of models and the escalating memory requirements. This led to the utilization of lower-precision values for the loading model. Employing an 8-bit representation for numbers reduces the storage requirement to a mere 1 byte.

In more recent times, an additional advancement allows for models to be loaded in a 4-bit format, further reducing memory demands. It is possible to use the BitsAndBytes library while loading a pre-trained model, as the following code presents.

Copy

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
import torch

model = AutoModelForCausalLM.from_pretrained(
        model_name_or_path='/name/or/path/to/your/model',
        load_in_4bit=True,
        device_map='auto',
        torch_dtype=torch.bfloat16,
        quantization_config=BitsAndBytesConfig(
                load_in_4bit=True,
                bnb_4bit_compute_dtype=torch.bfloat16,
                bnb_4bit_use_double_quant=True,
                bnb_4bit_quant_type='nf4'
        ),
        )
```

The sample code.

It's crucial to remember that this approach relates exclusively to storing model weights and does not affect the training process. Additionally, there's a constant balance to strike between employing lower-precision numbers and potentially compromising the language comprehension capabilities of models. While this trade-off is justified in many instances, it's important to acknowledge its presence.

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, the key used for loading the tokenizer/model can be modified to models with any size like LLaMA2.)

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. The initial stage involves loading the tokenizer and the model, followed by decoding the output generated by the model. We employ the beam search decoding approach with a limitation to generate a maximum of 128 tokens. (Explore these techniques further in the in-depth blog post by Huggingface.)

Copy

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

from transformers import AutoModelForCausalLM
from accelerate import Accelerator

model = AutoModelForCausalLM.from_pretrained(
   "./OPT-RL-OrcaChat/merged", device_map={"": Accelerator().process_index}
)
model.eval();

inputs = tokenizer("Question: In one sentence, describe what the following article is about:\n\nClick on "Store" along the menu toolbar at the upper left of the screen. Click on "Sign In" from the drop-down menu and enter your Apple ID and password. After logging in, click on "Store" on the toolbar again and select "View Account" from the drop-down menu. This will open the Account Information page.  Click on the drop-down list and select the country you want to change your iTunes Store to.  You'll now be directed to the iTunes Store welcome page. Review the Terms and Conditions Agreement and click on "Agree" if you wish to proceed. Click on "Continue" once you're done to complete changing your iTunes Store..\n\n Answer: ", return_tensors="pt").to("cuda:0")
generation_output = model.generate(**inputs,
                        return_dict_in_generate=True,
                        output_scores=True,
                        max_new_tokens=128,
                        num_beams=4,
                        do_sample=True,
                        top_k=10,
                        temperature=0.6)
print( tokenizer.decode(generation_output['sequences'][0]) )
```

The sample code.

The following entries represent the outputs generated by the model using various prompts.

- In one sentence, describe what the following article is about…
- Answer the following question given in this paragraph…
-

What the following paragraph is about?...

▸

What the following paragraph is about?... (2)

As evidenced by the examples, the model displays the ability to follow instructions and extract information from lengthy content. However, it falls short in terms of answering open-ended questions such as "Explain the raining process?" This is primarily attributed to the model's smaller size, which entails fewer parameters, approximately ranging from 30x to 70x less than state-of-the-art models.

Deploying LLMs Module

Deploying LLMs

Goals: Familiarize students with efficient LLM deployment techniques, emphasizing quantization and pruning. Offer hands-on experiences with deployments on platforms like GCP and Intel® CPUs.

This module dives into deploying Large Language Models. Model quantization and pruning are central to these strategies, each serving as an effective tool to optimize model performance without compromising efficiency. With a foundation in research articles and real-world applications, this module also introduces participants to deployment on cloud platforms.

- **Challenges of LLM deployment**: The lesson covers challenges during LLM deployment such as the sheer size of models, associated costs, and potential latency issues. We also provide a survey on optimizations, rooted in a research article on Transformer inference and a deepened perspective on potential solutions in addressing these challenges.
- **Model Quantization**: This lesson centers on Quantization, highlighting its role in streamlining LLM deployments. We research the balance between model performance and efficiency by understanding its usefulness and various techniques.
- **Model Pruning**: This module discusses model pruning, showcasing its place in LLM optimization. We will introduce various pruning techniques backed by recent research.
- **Deploying an LLM on a Cloud CPU**: This module uncovers the advantages, considerations, and challenges of deploying large language models on cloud-based CPUs. This lesson needs a server instance equipped with an Intel® Xeon® processor.

By the end of this module, students have gained a robust understanding of the intricacies involved in LLM deployment. The exploration of model quantization, pruning, and practical deployment strategies has provided them with the tools necessary to navigate real-world challenges. Moving beyond foundational concepts, the next section offers a deep dive into the advanced topics and future directions in the realm of LLMs.

After navigating the diverse terrain of Transformers and LLMs, participants now deeply understand significant architectures like GPT and BERT. The sessions shed light on model evaluation metrics, advanced control techniques for optimal outputs, and the roles of pretraining and finetuning. The upcoming module dives into the complexities of deciding

when to train an LLM from scratch, the operational necessities of LLMs, and the sequential steps crucial for the training process.

Challenges of LLM Deployment

Introduction

In this lesson, we will study the challenges of deploying Large Language Models, with a focus on the importance of latency and memory. We also explore optimization techniques with concepts like quantization and sparsity and how they can be applied using the Hugging Face Optimum and Intel® Neural Compressor libraries. We also discuss the role of Intel's® optimization technologies in efficiently running LLMs. This lesson will provide a deeper understanding of how to optimize LLMs for better performance and user experience.

Importance of Latency and Memory

Latency is the delay before a transfer of data begins following an instruction. It is a crucial factor in LLM applications. High latency in real-time or near-real-time applications can lead to a poor user experience. For instance, in a conversational AI application, a delay in response can disrupt the natural flow of conversation, leading to user dissatisfaction. Therefore, reducing latency is a critical aspect of LLM deployment.

Considering an average human reading speed of ~250 words per minute (translated to ~312 tokens per minute) that is about 5 tokens per second, therefore a latency of 200ms per token. Usually, acceptable latency for near-real-time LLM applications is between 100ms and 200ms per token.

Transformers can be computationally intensive and memory-demanding, due to their complex architecture and large size. However, several optimization techniques can be employed to enhance their efficiency without significantly compromising their performance.

Quantization

Quantization is a technique used for compressing neural network models, including Transformers, by lowering the precision of model parameters and/or activations. This method can significantly reduce memory usage. It leverages low-bit precision arithmetic and decreases the size, latency, and energy consumption.

However, it's important to strike a balance between performance gains through reduced precision and maintaining model accuracy. Techniques such as mixed-precision quantization, which assign higher bit precision to more sensitive layers, can mitigate accuracy degradation.

We'll learn different quantization methods later in the course.

Sparsity

Sparsity, usually achieved by pruning, is another technique for reducing the computational cost of LLMs by eliminating redundant or less important weights and activations. This method can significantly decrease off-chip memory consumption, the corresponding memory traffic, energy consumption, and latency.

Pruning can be broadly divided into types: weight pruning and activation pruning.

- **Weight pruning** can be further categorized into unstructured pruning and structured pruning. Unstructured pruning allows any sparsity pattern, and structured pruning imposes an additional constraint on the sparsity pattern. While structured pruning can provide benefits in terms of memory, energy consumption, and latency without additional hardware support, it is known to achieve a lower compression rate than unstructured pruning.
- On the other hand, **activation pruning** prunes redundant activations during inference, which can be especially effective for Transformer models. However, this requires support to detect and zero out unimportant activations at run-time dynamically.

We'll study different pruning methods later in the course.

Utilizing Optimum and Intel® Neural Compressor Libraries

The Hugging Face Optimum and the Intel® Neural Compressor libraries provide a suite of tools helpful in optimizing models for inference, especially for Intel® architectures.

- The Hugging Face Optimum library serves as an interface between the Hugging Face transformers and diffuser libraries and the various tools provided by Intel®.
- The Intel® Neural Compressor is an open-source library that facilitates the application of popular compression techniques such as quantization, pruning, and knowledge distillation. It supports automatic accuracy-driven tuning strategies, enabling users to generate quantized models easily. This library allows users to apply static, dynamic, and aware-training quantization approaches while maintaining predefined accuracy criteria. It also supports different weight pruning techniques, allowing for the creation of pruned models that meet a predefined sparsity target.

These libraries provide a practical application of the quantization and sparsity techniques, and their usage will be of great use in optimizing the deployment of LLMs.

Intel® Optimization Technologies for LLMs

Intel's® optimization technologies play a significant role in running LLMs efficiently on CPUs. The 4th Gen Intel® Xeon® Scalable processors are equipped with AI-infused acceleration known as Intel® Advanced Matrix Extensions (Intel® AMX). These processors have built-in BF16 and INT8 GEMM (general matrix-matrix multiplication) accelerators in every core, which significantly accelerate deep learning training and inference workloads.

The Intel® Xeon® Proecssor Max Series offers up to 128GB of high-bandwidth memory, which is particularly beneficial for LLMs, as these models are often memory-bandwidth bound.

By (1) running model optimizations like quantization and pruning and (2) leveraging the Intel® hardware acceleration technologies, it's possible to achieve a good latency for LLMs too. Take a look at this page to see the performance improvements (better throughput, with less memory size) of several optimized models.

Conclusion

In this lesson, we have explored the challenges of deploying Large Language Models, with a particular focus on latency and memory.

We also discussed optimization techniques like quantization and sparsity, which can significantly reduce LLMs' computational cost and memory usage. We introduced the Hugging Face Optimum and Intel® Neural Compressor libraries, which provide practical tools for applying these techniques. Furthermore, we have highlighted the role of Intel's® optimization technologies, such as the 4th Gen Intel® Xeon Scalable

processors and the Intel® Xeon CPU Max Series, in efficiently running neural networks. By understanding and applying these concepts, we can optimize the deployment of LLMs, achieving better performance and user experience.

For more information on Intel® Accelerator Engines, visit this resource page*. Learn more about Intel® Extension for Transformers, an Innovative Transformer-based Toolkit to Accelerate GenAI/LLM Everywhere* here*.*

Model Quantization

Introduction

As AI models, including large language models, grow more advanced, their increasing number of parameters leads to significant memory usage. This, in turn, increases the costs of hosting and deploying these tools.

In this lesson, we will learn about **quantization,** a process that can be employed to **diminish the memory requirements** of these models. We will explore the various types of quantization, such as scalar and product quantization. We will also learn how fine-tuning techniques like QLoRA use quantization.

Finally, we will examine applying these techniques to AI models using a CPU with methods implemented in the Intel® neural compressor library.

Overview of Quantization

In deep learning, quantization is a technique that **reduces the numerical precision** of model parameters, such as the weights and biases. This reduction helps decrease the model's memory footprint and computational requirements, enabling easier deployment on resource-constrained devices such as mobile phones, smartwatches, and other embedded systems.

Everyday Example

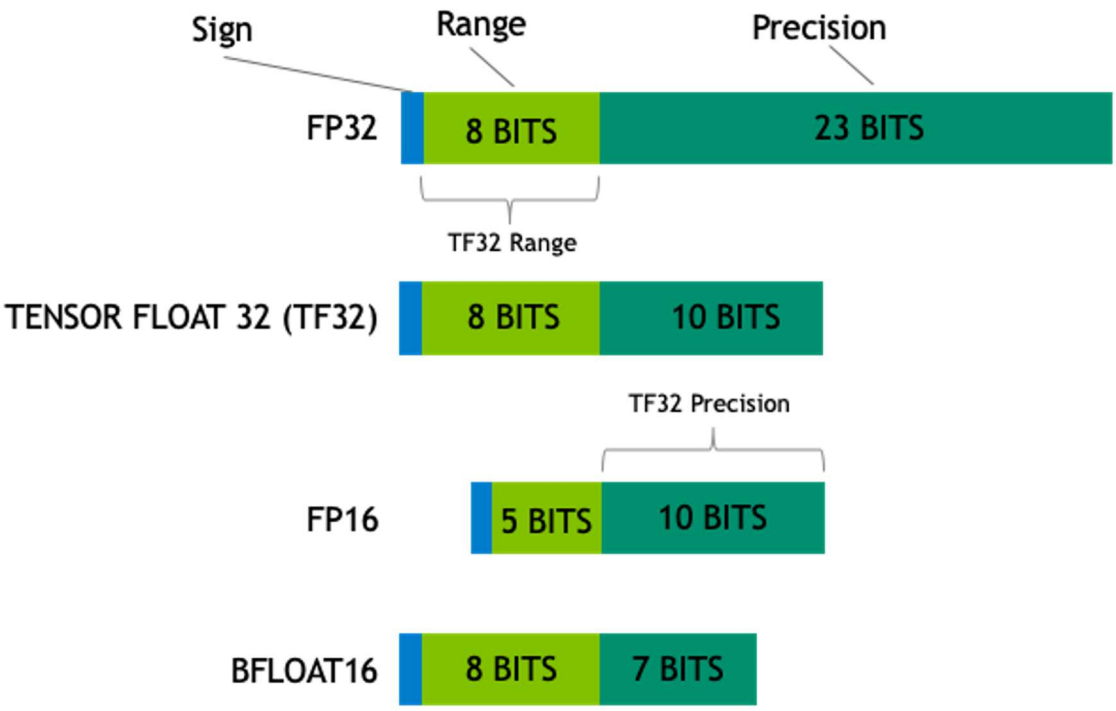
To understand the concept of quantization, consider an everyday scenario. Imagine two friends, Jay and John. Jay asks John, "What's the time?" John can reply with the exact time, 10:58 p.m., or he can say it's around 11 p.m. In the latter response, John simplifies the time, making it less precise but easier to communicate and understand. This is a basic example of quantization, which is analogous to the process in deep learning, where the precision of

model parameters is reduced to make the model more efficient, albeit at the cost of some accuracy.

Quantization in Machine Learning

In Machine Learning, different floating point data types can be used for model parameters, a characteristic also called precision. The precision of the data types affects the amount of memory required by the model. Defining the parameters in higher precision types, like Float32 or Float64, provides greater accuracy but requires more memory, while lower precision types, like Float16 or BFloat16, use less memory but may result in a loss of accuracy.

In the figure below, you can see the main floating point data types.



From "A Gentle Introduction to 8-bit Matrix Multiplication for transformers at scale using Hugging Face Transformers, Accelerate and bitsandbytes" blog post.

We can estimate the memory required for an AI model with its number of parameters. For example, consider the **Llama2 70B** model that uses Float16 precision for its parameters. Each parameter requires **two bytes**. To calculate the memory required in gigabytes (GB), where 1GB = 1024^3 bytes, the calculation is as follows:

(70,000,000,000∗2)/10243=130.385��(70,000,000,000∗2)/10243=130.385GB

Now, let's explore the different basic quantization techniques.

Scalar Quantization

In scalar quantization, each dimension of the dataset is treated independently. The maximum and minimum values are calculated for each dimension across the dataset. The range between the maximum and minimum values in each dimension is then divided into equal-sized bins. Each value in the dataset is mapped to one of these bins, effectively quantizing the data.

For example, consider a dataset of 2000 vectors with 256 dimensions sampled from a Gaussian distribution. The goal is to perform scalar quantization on this dataset.

Copy

```
import numpy as np


dataset = np.random.normal(size=(2000, 256))


# Calculate and store minimum and maximum across each dimension
ranges = np.vstack((np.min(dataset, axis=0), np.max(dataset, axis=0)))
```

Now, calculate each dimension's start value and step size. The start value is the minimum value, and the step size is determined by the number of discrete bins in the integer type being used. This example uses 8-bit unsigned integers (**uint8**), providing 256 bins.

Copy

```
starts = ranges[0,:]
steps = (ranges[1,:] - ranges[0,:]) / 255
```

The quantized dataset is then calculated as follows:

Copy

```
scalar_quantized_dataset = np.uint8((dataset - starts) / steps)
```

The overall scalar quantization process can be encapsulated in a function:

Copy

```
def scalar_quantisation(dataset):
    # Calculate and store minimum and maximum across each dimension
    ranges = np.vstack((np.min(dataset, axis=0), np.max(dataset, axis=0)))
    starts = ranges[0,:]
    steps = (ranges[1,:] - starts) / 255
    return np.uint8((dataset - starts) / steps)
```

Product Quantization

In scalar quantization, the data distribution in each dimension should ideally be considered to avoid loss of information. Product quantization can preserve more information by dividing each vector into sub-vectors and quantizing each sub-vector independently.
For example, consider the following array:
Copy

```
array = [ [ 8.2, 10.3, 290.1, 278.1, 310.3, 299.9, 308.7, 289.7, 300.1],
                                                    [ 0.1, 7.3, 8.9, 9.7, 6.9, 9.55, 8.1, 8.5, 8.99] ]
```

Quantizing this array to a 4-bit integer using scalar quantization results in significant information loss:
Copy

```
quantized_array = [[ 0  0 14 13 15 14 14 14 14]
                                                                    [ 0 0 0 0 0 0 0 0
0]]
```

In contrast, **product quantization** involves the following steps:

1. Divide each vector in the dataset into *m* disjoint sub-vectors.
2. For each sub-vector, cluster the data into *k* centroids (using *k*-means, for example).
3. Replace each sub-vector with the index of the nearest centroid in the corresponding codebook.

Let's proceed with the Product Quantization of the given array with m=3 (number of sub-vectors) and k=2 (number of centroids)
Copy

```
from sklearn.cluster import KMeans
import numpy as np

# Given array
array = np.array([
   [8.2, 10.3, 290.1, 278.1, 310.3, 299.9, 308.7, 289.7, 300.1],
   [0.1, 7.3, 8.9, 9.7, 6.9, 9.55, 8.1, 8.5, 8.99]
])

# Number of subvectors and centroids
m, k = 3, 2
```

```
# Divide each vector into m disjoint sub-vectors
subvectors = array.reshape(-1, m)

# Perform k-means on each sub-vector independently
kmeans = KMeans(n_clusters=k, random_state=0).fit(subvectors)

# Replace each sub-vector with the index of the nearest centroid
labels = kmeans.labels_

# Reshape labels to match the shape of the original array
quantized_array = labels.reshape(array.shape[0], -1)

# Output the quantized array
quantized_array
```

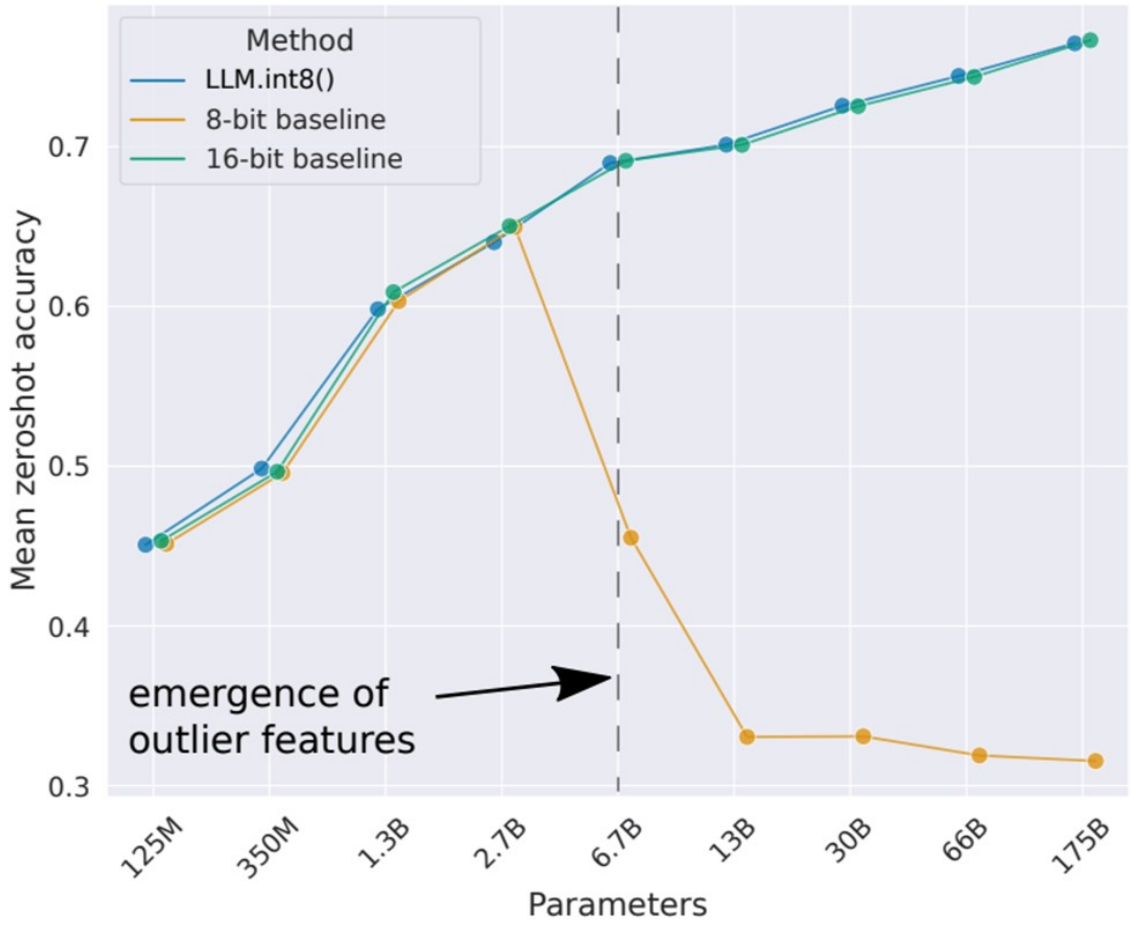
Copy

```
# Result
array([[0, 1, 1],
       [0, 0, 0]], dtype=int32)
```

By quantizing the vectors and storing only the indices of the centroids, the memory footprint is significantly reduced.
This method can help preserve more information than scalar quantization, especially when the distributions of different dimensions are diverse.
Product quantization can significantly reduce memory footprint and speed up the nearest neighbor search but at the cost of accuracy. The tradeoff in product quantization is based on the **number of centroids** and the number of sub-vectors we use. The more centroids we use, the better the accuracy, but the memory footprint would not decrease and vice versa.

Quantizing Large Models

We learned about two relatively basic quantization techniques that can be used with deep learning models. While these simple techniques can work well enough with models with few parameters, they usually lead to a drop in accuracy for larger models with billions of parameters.



From "LLM.int8(): 8bit Matrix Multiplication for Transformers at Scale" paper

Large models contain a **greater amount of information** in their parameters. With more neurons and layers, large models can represent more complex functions. They can capture deeper and more intricate relationships in the data, which smaller models might not be able to handle.
Thus, the quantization process, which reduces the precision of these parameters, can significantly lose this information, resulting in a substantial drop in model accuracy and performance.
Optimizing the quantization process for large models is also more difficult due to the larger parameter space. Finding the optimal quantization strategy that minimizes the loss of accuracy while reducing the model size is a more complex task for larger models.

Popular (Post-Training Quantization) Methods for LLMs
Fortunately, more sophisticated quantization techniques have been released to address these problems, aiming to maintain the accuracy of large models while effectively reducing their size.

LLM.int8()

This research paper observes that activation outliers (activation values significantly different from the others) break the quantization of larger models and proposes keeping them in higher precision. By keeping doing that, the performance of the model is not negatively affected.

GPTQ

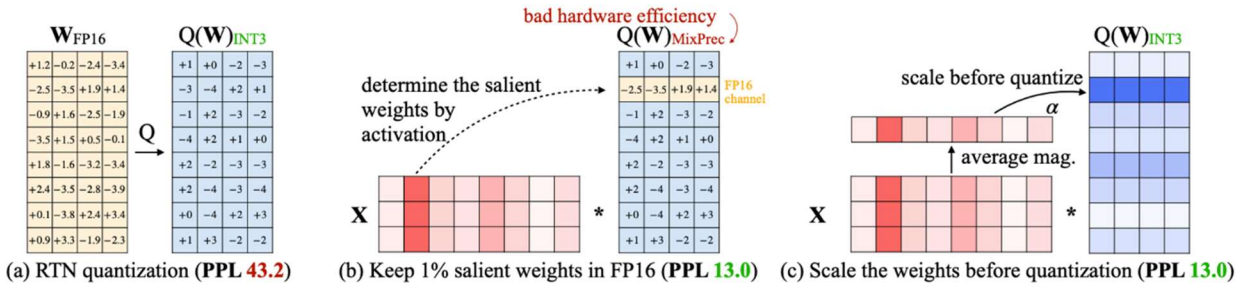
This technique allows for faster text generation. The quantization is done layer by layer, minimizing the mean squared error (MSE) between the quantized and full-precision weights when given an input.

The algorithm uses a mixed int4-fp16 quantization scheme where weights are quantized as int4 while activations remain in float16. During inference, weights are de-quantized on the fly, and the actual compute is performed in float16. This method makes use of a calibration dataset. The GPTQ algorithm requires calibrating the quantized weights of the model by making inferences on the quantized model.

AWQ

This method is grounded in the observation that not all weights contribute equally to Large Language Models performance. It identifies a small fraction (0.1%-1%) of 'important' or 'salient' weights, the quantization of which, if skipped, can substantially mitigate quantization loss.

Unlike traditional approaches that focus on weight distribution, the AWQ method selects these salient weights based on the magnitude of their activations. This approach leads to a notable enhancement in performance. By maintaining only 0.1%-1% of the weight channels, corresponding to larger activations, in the FP16 format, the method significantly boosts the performance of quantized models.



From "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration" paper

The authors note that retaining certain weights in FP16 format can cause hardware inefficiency due to using mixed-precision data types. To address this, they propose a method where all weights, including the salient ones, are quantized to avoid mixed-precision data types. However, before the quantization process, the weights are scaled. This scaling step is crucial as it helps protect the outlier weight channels during quantization, ensuring that the important information they hold is not lost or significantly altered during the quantization process. This method aims to strike a balance, allowing the model to benefit from the quantization efficiency while preserving the essential information in the salient weights.

Using Quantized models

Many open-source LLMs are available for download in a quantized format. As we learned in this lesson, these models will have reduced memory requirements.

You can look at the model's section on HuggingFace to find and use a quantized model. This platform hosts a variety of models. For instance, you can try the latest Mistral-7B-Instruct model, which has been quantized using the GPTQ method.

Quantizing Your Own LLM

You can use the Intel® Neural Compressor Library to quantize your own Large Language Model. This library offers various techniques for model quantization, some of which have been discussed in this module.

To get started, follow the step-by-step guide provided in the **repository**. This guide will walk you through quantizing a model, ensuring you have all the necessary components and knowledge to proceed.

Before beginning the quantization process, ensure you have installed the **neural-compressor** library and **lm-evaluation-harness**. Inside the cloned neural compressor directory, navigate to the appropriate directory and install the required packages by running the following commands:

Copy

```
cd examples/pytorch/nlp/huggingface_models/language-modeling/quantization/ptq_weight_only

pip install -r requirements.txt
```

As an example, to quantize the **opt-125m** model with the GPTQ algorithm, use the following command:

Copy

```
python examples/pytorch/nlp/huggingface_models/language-modeling/quantization/ptq_weight_only/run-gptq-llm.py \
  --model_name_or_path facebook/opt-125m \
  --weight_only_algo GPTQ \
  --dataset NeelNanda/pile-10k \
  --wbits 4 \
  --group_size 128 \
  --pad_max_length 2048 \
  --use_max_length \
  --seed 0 \
  --gpu
```

This command will quantize the **opt-125m** model using the specified parameters.

How Quantization is used in QLoRA

We saw in a previous lesson how fine-tuning can be achieved using fewer resources using QLoRa, a popular variant of LoRA that makes fine-tuning large language models even more accessible.

In the course, we saw that QLoRA involves backpropagating gradients through a frozen, 4-bit quantized pre-trained language model into Low-Rank Adapters. To accomplish this, QLoRA employs a novel data type, the **4-bit NormalFloat (NF4)**, which is theoretically optimal for normally distributed weights.

This optimality stems from quantile quantization, a technique particularly suited for normally distributed values. It ensures that each quantization bin holds **an equal number of values** from the input tensor, minimizing quantization error and providing a more uniform data representation.

Since pre-trained neural network weights typically exhibit a **zero-centered normal distribution** with a standard deviation (σ), QLoRA transforms all weights into a unified fixed distribution. This transformation is achieved by scaling σ to ensure the distribution aligns perfectly within the range of the NF4 data type, further enhancing the efficiency and accuracy of the quantization process.

This new fine-tuning technique shows no accuracy degradation in their experiments and matches BFloat16 performance.

Table 4: Mean 5-shot MMLU test accuracy for LLaMA 7-65B models finetuned with adapters on Alpaca and FLAN v2 for different data types. Overall, NF4 with double quantization (DQ) matches BFloat16 performance, while FP4 is consistently one percentage point behind both.

	Mean 5-shot MMLU Accuracy								
LLaMA Size	7B		13B		33B		65B		Mean
Dataset	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	
BFloat16	38.4	45.6	47.2	50.6	57.7	60.5	61.8	62.5	53.0
Float4	37.2	44.0	47.3	50.0	55.9	58.5	61.3	63.3	52.2
NFloat4 + DQ	39.0	44.5	47.5	50.7	57.3	59.2	61.8	63.9	53.1

From "QLoRA: Efficient Finetuning of Quantized LLMs" paper

Conclusion

In this lesson, we explored the concept of quantization, a technique that can reduce the memory requirements of large models and, in some cases, enhance the text generation speed for language models. We delved into some state-of-the-art quantization techniques suitable for models with billions of parameters, examining the unique contributions of each method.

We also learned how to quantize our own models using the Intel® Neural Compressor Library, which supports many popular quantization methods.

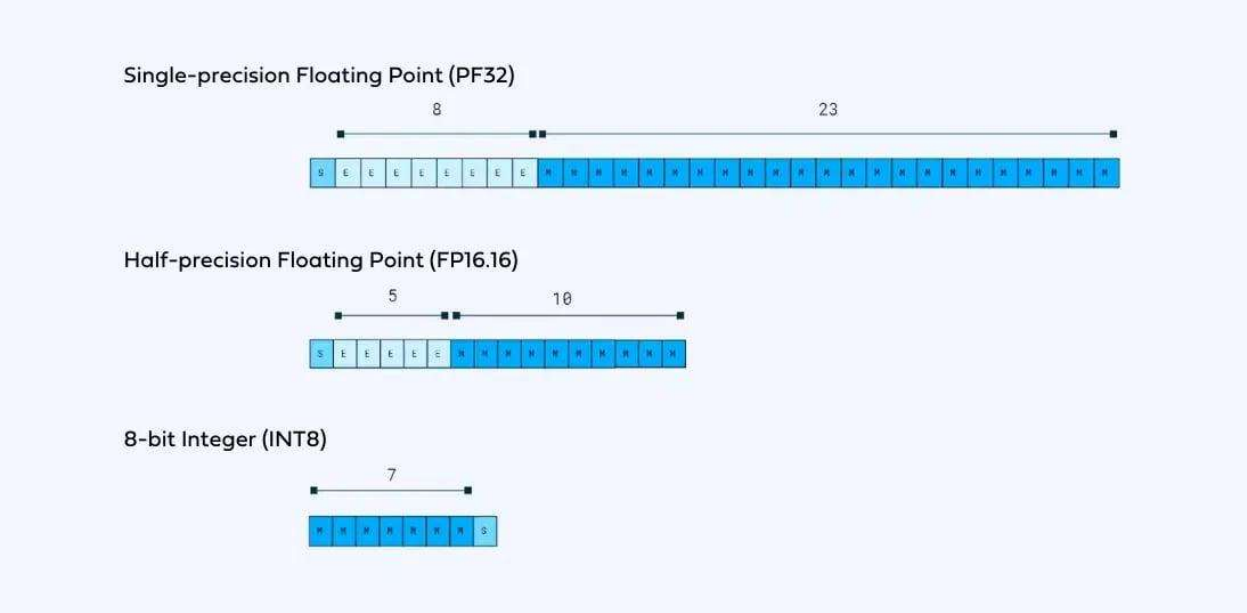
Lastly, we revisited QLoRA, understanding how it leverages quantization to make the fine-tuning of models more accessible to a broader audience.

Deploying an LLM on a Cloud CPU

Introduction

Training a language model can be costly, and expenses associated with deploying it can quickly accumulate over time. Utilizing optimization techniques that enhance the efficiency of the inference process is crucial for minimizing hosting expenses. In this lesson, we will discuss the utilization of the Intel® Neural Compressor library to implement quantization techniques. This approach aims to enhance the cost-effectiveness and speed of models when running on CPU instances (it supports also AMD CPU, ARM CPU, and NVidia GPU through ONNX Runtime, but with limited testing).

Various techniques can be employed for optimizing a network. Pruning involves trimming the parameter count by targeting less important weights, while knowledge distillation transfers insights from a larger model to a smaller one. Lastly, quantization decreases weight precision from 32 bits to 8 bits. It will significantly decrease the memory needed for loading models and generating responses with minimal accuracy loss.



Credit: Deci.ai

The primary focus of this lesson is the quantization technique. We will apply it to an LLM and demonstrate how to perform inference using the quantized model. Ultimately, we will execute several experiments to assess the resulting acceleration.

We'll begin by setting up the necessary libraries. Install the optimum-intel package directly from its GitHub repository.

Copy

```
pip install git+https://github.com/huggingface/optimum-intel.git@v1.11.0

pip install onnx===1.14.1 neural_compressor===2.2.1Copy
```

The sample code.

Simple Quantization (using CLI)
You can utilize the optimum-cli command within the terminal to execute dynamic quantization. Dynamic quantization stands as the recommended approach for transformer-based neural networks. You have the choice to either specify the path to your custom model or select a model from the Huggingface Hub, which will be designated using the --model parameter. The --output parameter determines the name of the resulting model. We are conducting tests on Facebook's OPT model with 1.3 billion parameters.
Copy

```
optimum-cli inc quantize --model facebook/opt-1.3b --output opt1.3b-quantizedCopy
```

The sample code.

The script above will automatically load the model and handle the quantization process. I worth noting that if the script fails to recognize your model, you can employ the --task parameter. You might use --task text-generation for language models. Check the source code for a complete list of supported tasks.
The mentioned approach can be easily incorporated by simply executing a single command. However, for more complex applications where greater control over the process is needed, it might not provide the desired flexibility. In the following section we will use the same Intel Neural Compressor package to perform a more targeted quantization process.

Flexible Quantization (using Code)
As previously discussed, the library includes a constrained quantization method that allows you to specify a precise quantization target. In this approach, we must code and implement the function, which requires more steps while providing more control over the process. For example, you can employ an evaluation function to request quantization of the model while experiencing no more than a 1% decrease in accuracy. To begin, let's install the necessary packages which also includes the Intel neural compressor from the previous section. This will allow us to load the model and carry out the quantization process.
Copy

```
pip install transformers===4.34.0 evaluate===0.4.0 datasets===2.14.5Copy
```

The sample code.

The mentioned packages will help with loading the large language model (transformers), defining an evaluation metric to measure how close we are to the target (evaluate), and

importing a dataset for the evaluation process (datasets). Now, we can load the model's weights and its accompanying tokenizer.
Copy

```
model_name "aman-mehra/opt-1.3b-finetune-squad-ep-0.4-lr-2e-05-wd-0.01"

tokenizer = AutoTokenizer.from_pretrained(model_name, cache_dir="./opt-1.3b")

model = AutoModelForQuestionAnswering.from_pretrained(model_name, cache_dir="./opt-1.3b")Copy
```

The sample code.

We are initializing a model specifically tailored for the question-answering task. Keep in mind that the model you chose must be fine-tuned for question answering tasks before performing the quantization, we chose a fine-tuned version of the OPT-1.3 model. Choosing a task is required to define an objective for our quantization target function and control the process. The task and evaluation metrics can vary widely, ranging from text generation with perplexity, summarization with ROUGE, translation with BLEU, to even classification based on simple accuracy. The next is to define the evaluation metric to assess the model's accuracy and the benchmark dataset that compliments it.
Copy

```
task_evaluator = evaluate.evaluator("question-answering")


eval_dataset = load_dataset("squad", split="validation", cache_dir="./squad-ds")

eval_dataset = eval_dataset.select(range(64)) # Ues a subset of datasetCopy
```

The sample code.

The .evaluator() method will load the essential functions required for evaluating the question-answering task. (Further information on various options is available in the Hugging Face documentation.) You can then employ the load_dataset function from the Hugging Face library to bring a dataset into memory. This function allows you to specify parameters such as the dataset name, which splits to download (train, test, or validation), and the location for storing the dataset. Using the mentioned variables to create the evaluation function is now possible.
Copy

```
qa_pipeline = pipeline("question-answering", model=model, tokenizer=tokenizer)


def eval_fn(model):
   qa_pipeline.model = model
```

```
   metrics = task_evaluator.compute(model_or_pipeline=qa_pipeline, data=eval_dataset,
metric="squad")

   return metrics["f1"]Copy
```

The sample code.

We need to create a pipeline that ties the model with the tokenizer to be able to calculate the model's performance by calling the task evaluator's .compute() function along with the mentioned pipeline and the evaluation dataset. The eval_fn function will compute the accuracy and return the percentage. The quantization process requires several configurations for guidance.
Copy

```
# Set the accepted accuracy loss to 1%

accuracy_criterion = AccuracyCriterion(tolerable_loss=0.01)



# Set the maximum number of trials to 10

tuning_criterion = TuningCriterion(max_trials=10)



quantization_config = PostTrainingQuantConfig(

   approach="dynamic", accuracy_criterion=accuracy_criterion,
tuning_criterion=tuning_criterion

)Copy
```

The sample code.

The PostTrainingQuantConfig config variable will set the required parameters for the quantization process. We are employing the dynamic approach for quantization while accepting a maximum of 1% loss in accuracy, controlled by defining the AccuracyCriterion class. Use the TuningCriterion class to set the maximum number of trials before finishing the quantization process. Lastly, we will define a quantizer object using the INCQuantizer class, which accepts both the model and evaluation function. It could initiate the quantization process by calling the .quantize() method.
Copy

```
quantizer = INCQuantizer.from_pretrained(model, eval_fn=eval_fn)



quantizer.quantize(quantization_config=quantization_config, save_directory="opt1.3b-
quantized")Copy
```

The sample code.

Please note that it is impossible to execute the codes in this section on the Google Colab instance due to memory constraints. However, you can replace the model ("facebook/opt-1.3b") with a smaller model like "distilbert-base-cased-distilled-squad".

Inference

Now, the model is ready for inference purposes. In this section, we will focus on how to load these models and present the outcomes of our benchmark tests, highlighting the impact of quantization on the speed of the generation process. Prior to conducting the inference process, it's essential to load the pre-trained tokenizer using the AutoTokenizer class. As the quantization technique doesn't alter the model's vocabulary, we will employ the same tokenizer as the base model.

Copy

```
from transformers import AutoTokenizer


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

The sample code.

For loading the model, we utilize the INCModelForCasualLM class provided by the Optimum package. Additionally, it offers a range of loaders tailored for various tasks, including INCModelForSequenceClassification for classification and INCModelForQuestionAnswering for tasks involving question answering. The .from_pretrained() method should be provided with the path to the quantized model from the previous section.

Copy

```
from optimum.intel import INCModelForCausalLM


model = INCModelForCausalLM.from_pretrained("./opt1.3b-quantized")Copy
```

The sample code.

Finally, we can employ the identical .generate method from the Transformers library to input the prompt to the model and get the response.

Copy

```
inputs = tokenizer("<PROMPT>", return_tensors="pt")


generation_output = model.generate(**inputs,
                                   return_dict_in_generate=True,
```

```
                output_scores=True,
                min_length=512,
                max_length=512,
                num_beams=1,
                do_sample=True,
                repetition_penalty=1.5)Copy
```

The sample code.

The last step is to convert the generated token ids from the model to the words. It is the same decoding process as we saw in previous lessons.
Copy

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

The sample code.

Copy

```
What does life mean? Describe in great details.\nI have no idea. I don't know how to describe it. I don't know what I'm supposed to do with my life. I don't know what I want to do with my life...
```

The output.

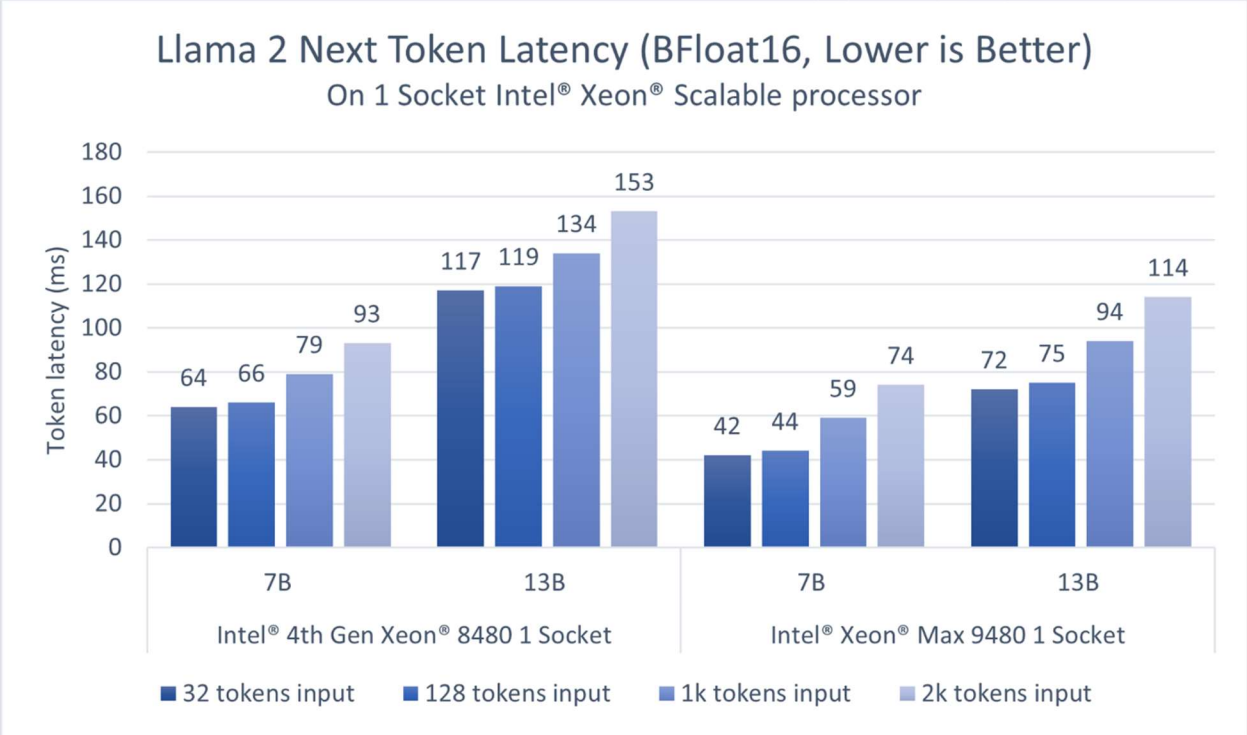
The chosen OPT model is not a instructed tuned model. so it tries to complete the sequence which is fed to it. As evident, it enters into a repetitive loop, reiterating the same words, since we have instructed it to produce exact 512 tokens. Even if the model desires to stop, it is unable to do so!
As mentioned, we force the model to produce 512 tokens by explicitly setting minimum and maximum length parameters. The rationale is maintaining a uniform token count between the standard model and the quantized version, facilitating a valid comparison of their generation times. We also experimented with the beam search decoding strategy.

Decoding Method	Vanilla *(seconds)*	Quantized *(seconds)*
Greedy	58.09	**26.847**
Beam Search (K=4)	144.77	**40.73**

The most significant enhancement involves implementing beam search with a batch size of 1, which led to a 3.5x acceleration in the inference process. All the mentioned experiments

were conducted on a server instance equipped with the 4th Gen Intel® Xeon® Scalable processor with 8 vCPU (4 cores) and 64GB of memory. The utilized instance is the entry tier of the Intel CPUs, featuring the least number of cores among the Intel® Scalable Processor family. This highlights the feasibility of performing inference on CPU instances to mitigate costs and latency effectively.

On the other hand, the highest tier series of 4th Gen Intel Xeon Scalable processors, accessible on the Google Cloud Platform, offers 176 virtual CPUs with 88 cores. It's worth highlighting that the most powerful processor within the SPR family can have up to 112 physical cores. In a recent report released by Intel comparing the performance of their Intel 4th Gen Xeon to the latest Intel® Xeon® Max, the report highlights an impressive speed of 114ms for processing 2K tokens using the latest Max processors with the LLaMA 2 model, which has 13B parameters. Intel takes the lead in advancing and fine-tuning the CPU backend of torch.compile, a prominent feature in PyTorch 2.0. Additionally, Intel provides the Intel® Extension for PyTorch to implement advanced optimizations specifically designed for Intel CPUs before their integration into the official PyTorch distribution.



Llama 2 7B and 13B inference (BFloat16) performance on Intel® Xeon® Scalable Processors. (source: Intel Blog)

Deployment Frameworks

Deploying large language models into production is the final stage in harnessing their capabilities for a diverse array of applications. Creating an API is the most efficient and flexible approach among the various methods available. APIs allow developers to seamlessly integrate these models into their code, enabling real-time interactions with web or mobile applications. There are several ways to create such APIs, each with its advantages and trade-offs.

There exist specialized libraries, such as vLLM and TorchServe, designed for handling specific use cases. These libraries are capable of loading models from various sources and creating endpoints for convenient accessibility. In most cases, these libraries even offer optimization methods to enhance the speed of the inference process, batching income requests, and efficient memory management. On the other hand, there exist standard backend libraries such as FastAPI that facilitate the creation of any endpoints. While it may not be specifically designed for serving AI models, you can effortlessly integrate it into your development process to generate other APIs as needed.
Regardless of the chosen method, a well-designed API ensures that large language models can be deployed robustly, enabling organizations to leverage their capabilities in chatbots, content generation, language translation, and many other applications.

Deploying a model on CPU using Compute Engine with GCP
Follow these steps to deploy a language model on Intel® CPUs using Compute Engine with Google Cloud Platform (GCP):

1. **Google Cloud Setup**: Sign in to your Google Cloud account. If you don't have one, create it and set up a new project.
2. **Enable Compute Engine API**: Navigate to APIs & Services > Library. Search for "Compute Engine API" and enable it.
3. **Create a Compute Engine instance**: Go to the Compute Engine dashboard and click on “Create Instance”. Choose an CPU for your machine type. Here are several machine types that can be used in GCP and sporting Intel CPUs.

CPU processor	Processor SKU	Supported machine series and types	Base frequency (GHz)	All-core turbo frequency (GHz)	Single-core max turbo frequency (GHz)
Intel Xeon Scalable Processor (Sapphire Rapids) 4th generation	Intel® Xeon® Platinum 8481C Processor	• C3	1.9	3.0	3.3
Intel Xeon Scalable Processor (Ice Lake) 3rd Generation	Intel® Xeon® Platinum 8373C Processor	• N2*	2.6	3.4	3.5
		• M3	2.6	3.4	3.5
Intel Xeon Scalable Processor (Cascade Lake) 2nd Generation	Intel® Xeon® Gold 6268CL Processor	• N2*	2.8	3.4	3.9
	Intel® Xeon® Gold 6253CL Processor	• C2	3.1	3.8	3.9
	Intel® Xeon® Platinum 8280L Processor	• M2	2.5	3.4	4.0
	Intel® Xeon® Platinum 8273CL Processor	• A2 • G2	2.2	2.9	3.7
Intel Xeon Scalable Processor (Skylake) 1st Generation	Intel® Xeon® Scalable Platinum 8173M Processor	• E2 • `m1-megamem` memory-optimized machine types • N1	2.0	2.7	3.5
Intel Xeon E7 (Broadwell E7)	Intel® Xeon® E7-8880V4 Processor	• `m1-ultramem` memory-optimized machine types	2.2	2.6	3.3
Intel Xeon E5 v4 (Broadwell E5)	Intel® Xeon® E5-2696V4 Processor	• E2 • N1	2.2	2.8	3.7
Intel Xeon E5 v3 (Haswell)	Intel® Xeon® E5-2696V3 Processor	• N1	2.3	2.8	3.8
Intel Xeon E5 v2 (Ivy Bridge)	Intel® Xeon® E5-2696V2 Processor	• N1	2.5	3.1	3.5

Once the instance is up and running:

1. **Deploy the model**: SSH into your instance. Install the necessary libraries and dependencies and copy your server code (FastAPI, vLLM, etc) to the machine.
2. **Run the model**: Once the setup is complete, run your language model. If it's a web-based model, start your server.

Remember, Google Cloud charges based on the resources used, so make sure to stop your instance when not in use.
A similar process can be done for AWS too using EC2. You can find AWS machine types here.

Conclusion

In this lesson, we explored the potential of harnessing 4th Generation Intel Xeon Scalable Processors for the inference process and the array of optimization techniques available that make it a practical choice. Our focus was on the quantization approach aimed at enhancing the speed of text generation while conserving resources. It is fairly straightforward to perform the optimization process courtesy of a series of libraries from Intel such as Intel Extension for PyTorch and Intel® Extension for Transformers.

The results demonstrate the advantages of applying this technique across various configurations. It is worth noting that there are additional techniques available to optimize the models further. The upcoming chapter will discuss advanced topics within language models, including aspects like multi-modality and emerging challenges.

>> Notebook.

For more information on Intel® Accelerator Engines, visit this resource page*. Learn more about Intel® Extension for Transformers, an Innovative Transformer-based Toolkit to Accelerate GenAI/LLM Everywhere* here*.*

Introduction to Large Multimodal Models

Introduction

In this lesson, we will examine an emerging field of interest: The next progression in the evolution of LLMs, **Large multimodal models (LMMs)**. This topic adds a new layer to the material we've covered throughout the course.

Simply put, multimodal models are designed to handle and interpret different data types - or **"modalities"** - such as text, images, audio, and video, all within a single coordinated system. This integration allows for a more comprehensive analysis and understanding than models processing only one data type, such as text in standard LLMs. For instance, supplementing a text prompt with voice or image inputs can enable these models to capture a more complex representation of the conveyed information. This is achieved by analyzing additional layers of data, such as the tone and cadence of your voice or the visual context provided by images, thus enhancing the depth and richness of the analysis.

With the recent increase in the popularity of large language models, it is unsurprising that researchers are now exploring the potential of extending these models to handle multiple data types, aiming to create more versatile and valuable **general-purpose assistants**. Models that can solve arbitrary tasks specified by the user.

In the following sections, we will explore the current implementations of LMMs and introduce key concepts on how they manage multimodality. We will also learn about their **emergent abilities** and explore the idea of **Instruction-tuned** LMMs.

Finally, in this lesson, we will learn how **Deep Lake** by ActiveLoop can be helpful in training or fine-tuning large multimodal models.

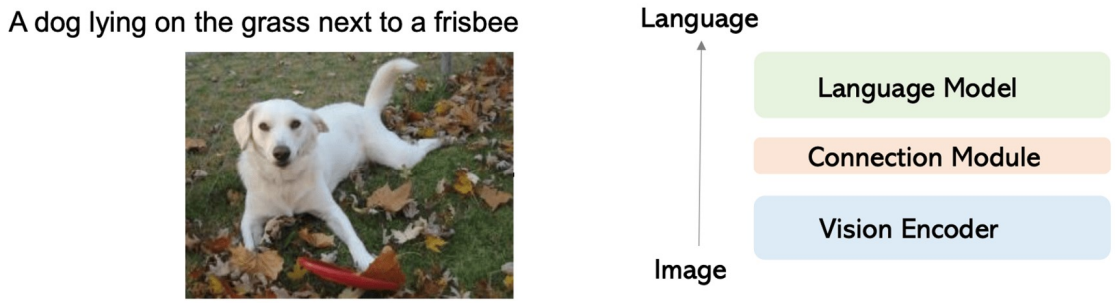
Common Architectures and Training Objectives

By definition, multimodal models are designed to process various input modalities, such as text, images, and videos, and generate outputs in multiple modalities. However, a notable

subset of currently popular LMMs primarily focuses on accepting image inputs and is limited to generating text outputs.
These specialized LMMs often leverage pre-trained large-scale vision or language models as their foundation. We can categorize them as 'Image-to-Text Generative Models,' also known as visual language models (VLMs). They generally perform tasks related to image understanding, such as question answering and image captioning. Examples include GIT by Microsoft, BLIP2 by SalesForce, and Flamingo by DeepMind.

Model Architecture

These models use an **image encoder** to extract visual features and a standard LLM to output a text sequence. The image encoder can be based on convolutional neural networks (CNNs), such as ResNet, or a transformer-based architecture like the Vision Transformer (ViT).
The image encoder and the language model can be trained from scratch or using pre-trained models. Most state-of-the-art models opt for the latter approach; an example is the pre-trained image encoder from the model CLIP by OpenAI. The options for language models are also extensive: one could choose from various open-source pre-trained models, such as Meta's OPT, Llama 2, or Google's instruction-trained FlanT5 models.
Optionally, models like BLIP2 introduce a trainable lightweight connection module connecting the vision and language modalities. Since BLIP2 only trains this light module, it is cheaper and faster than other methods while still managing a strong zero-shot performance on image understanding tasks.



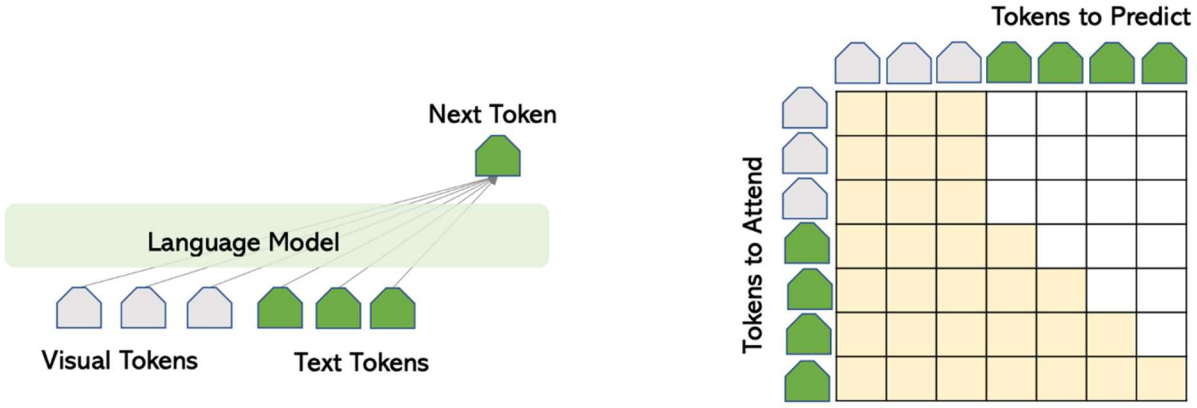
(a) Left: An example of image-to-text generation task; Right: model architecture.

From “Multimodal Foundation Models: From Specialists to General-Purpose Assistants” paper

Training Objective

Similar to what we've seen in the course, LMMs are trained using an **auto-regressive loss** function applied to the output text tokens. When using a Vision Transformer architecture, the concept of '**image tokens**,' which is analogous to text tokenization, is introduced. Just like text can be divided into smaller units like sentences, words, or sub-words for easier processing, images can be segmented into smaller, non-overlapping patches, known as 'tokens.'
The exact attention mechanisms come into play in the Transformer architecture employed by these LMMs. Image tokens can 'attend' to each other, meaning they can influence each

other's representation in the model. Meanwhile, the generation of each text token depends on all the image tokens and the previously generated text tokens. Check out our lesson about **Understanding Transformers** if you are still getting familiar with these concepts.



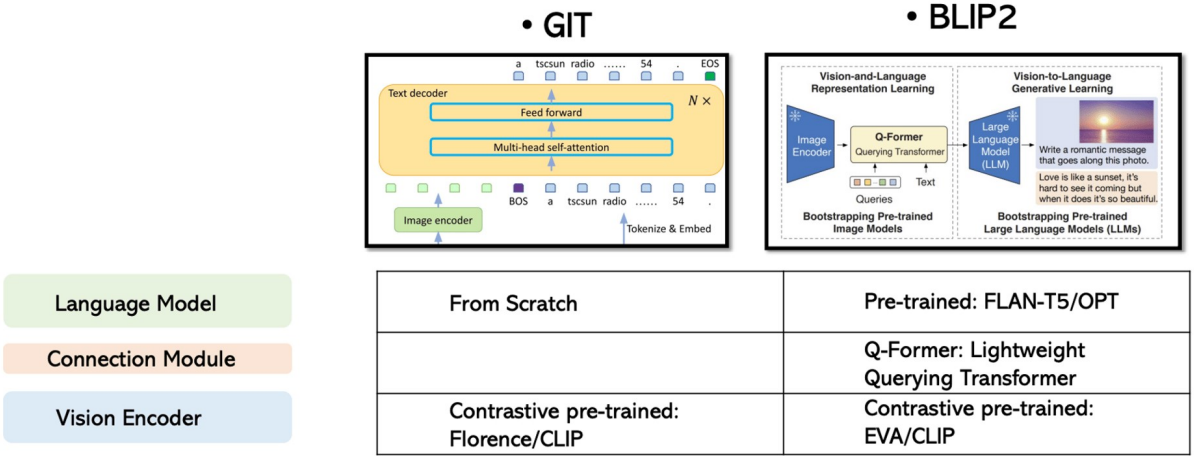
(b) Training objective and attention mask. For each row, the yellow elements indicate that the prediction token attends the tokens on the left.

Figure 3: Illustration of image-to-text generation task, architecture, and training objective.

From "Multimodal Foundation Models: From Specialists to General-Purpose Assistants" paper

Differences in Training Schemes

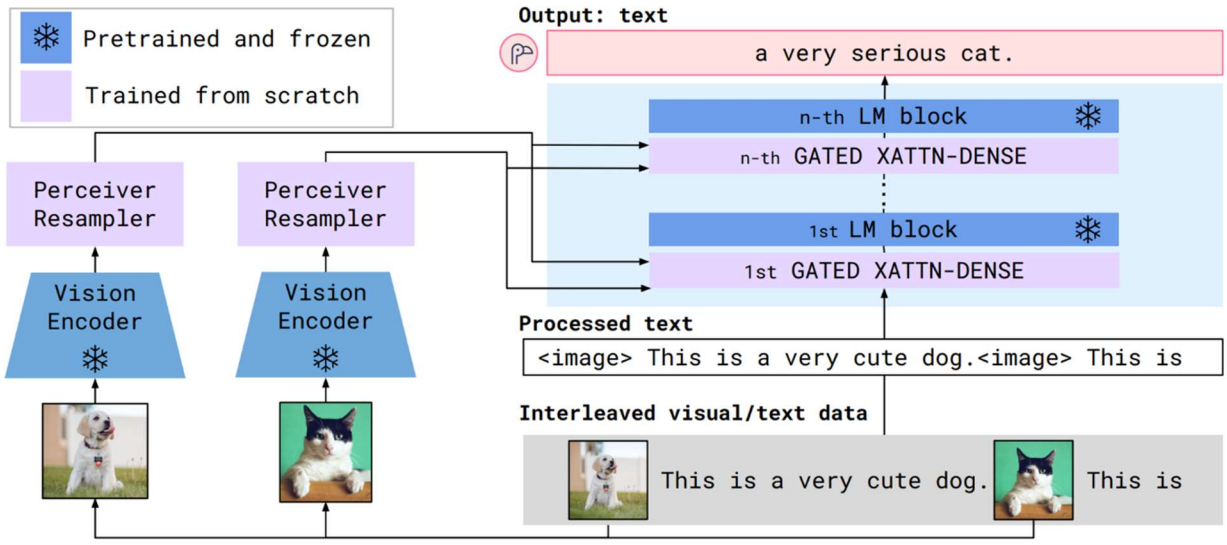
While having the same training objective, some variations emerge in the training schemes of different Language-Multimodal Models (LMMs). Most models, such as GIT and BLIP2, employ only image-text pairs for training. This approach allows them to establish connections between the text and image representations effectively but requires a large, curated image-text pairs dataset.



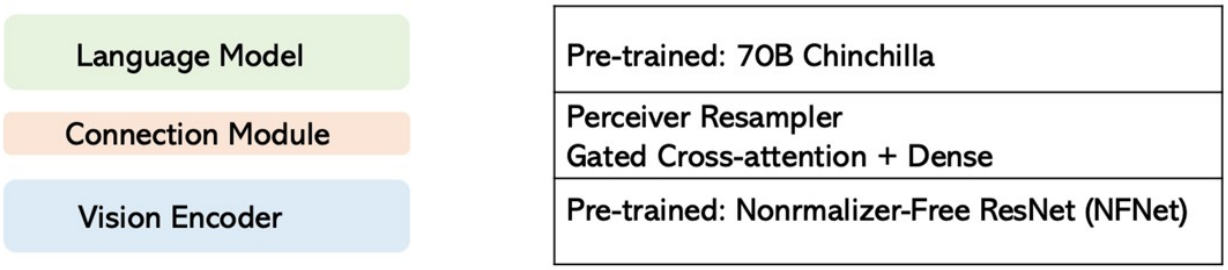
(a) Example 1: LMM with Image-Text Pairs.

From “Multimodal Foundation Models: From Specialists to General-Purpose Assistants” paper

On the other hand, the Flamingo model has some architectural innovations that allow for unlabeled web training data. They extract the text and images from the HTML of 43M webpages. They also determine the positions of images relative to the text based on the relative positions of the text and image elements in the Document Object Model (DOM). This model can ingest a **multimodal prompt** containing images and/or videos interleaved with text as input and generate text in an open-ended manner. It can produce text for tasks such as image captioning or visual question-answering.
The system connects the different modalities and enables multimodal prompting through steps. Initially, a **Perceiver Resampler** module receives spatiotemporal features from visual data, such as an image or video, processed by the pre-trained Vision Encoder. The Perceiver then generates a fixed number of 'visual tokens.'
These visual tokens serve as inputs to condition a frozen language model, a pre-trained language model that is not updated during this process. The conditioning is facilitated by adding newly initialized **cross-attention layers** interleaved with the language model's pre-existing layers. These new layers are not frozen and will be updated during training.
While this architecture is less efficient by having more parameters to train than the one from BLIP2, it provides a powerful way for the language model to incorporate visual cues.

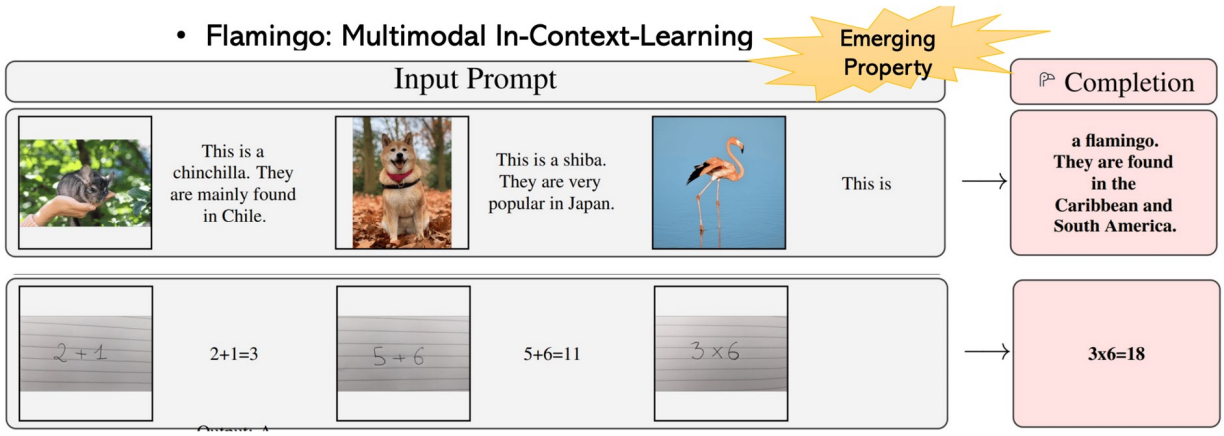


From “Flamingo: a Visual Language Model for Few-Shot Learning” paper



From "Multimodal Foundation Models: From Specialists to General-Purpose Assistants" paper

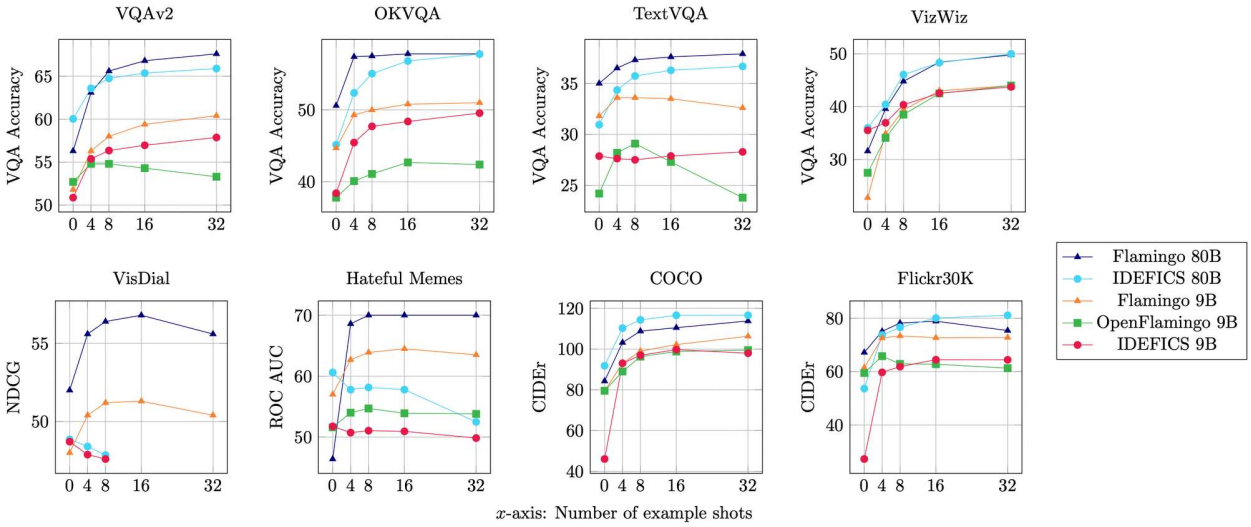
Discovering Emergent Abilities - Few-shot In-Context-Learning
Its flexible architecture allows Flamingo to be trained with multimodal prompts that interleave text with visual tokens. This enables the model to demonstrate emergent abilities, such as few-shot in-context learning, **analogous to GPT-3**. You can see some examples in the figure below.



From "Multimodal Foundation Models: From Specialists to General-Purpose Assistants" paper

Open-sourcing Flamingo
The state-of-the-art results reported in the Flamingo paper are exciting and clearly show significant progress in the field of LMMs. However, DeepMind has yet to make the Flamingo model publicly available.
To fill this gap, HuggingFace's team took the initiative to create an open-source reproduction of Flamingo, known as **IDEFICS**. This replica is constructed entirely using publicly accessible resources, including the LLaMA v1 and OpenCLIP models. IDEFICS is offered in the 'base' and the 'instructed' variants. Both of these are available in two sizes—9 billion parameters and 80 billion parameters. IDEFICS offers comparable results to Flamingo.

The team used a mixture of openly available datasets such as Wikipedia, Public Multimodal Dataset, and LAION to train these models. They also created a new 115B token dataset called **OBELICS**. It has 141 million interleaved image-text documents scraped from the web and contains 353 million images, replicating the dataset described by DeepMind in the Flamingo paper.
IDEFICS is available through the Transformers library, and a demo of it is available here.
Another open-source replication of Flamingo is called Open Flamingo at the 9B parameter size, offering similar performance to the original model.



From **"Introducing IDEFICS: An Open Reproduction of State-of-the-Art Visual Language Model"** blog post.

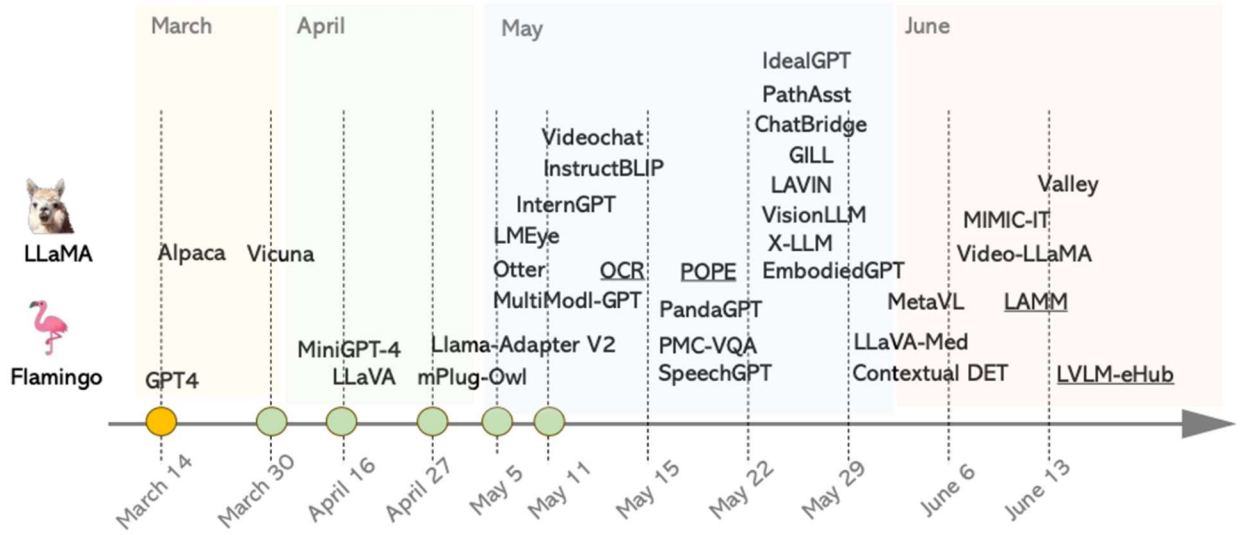
Instruction-tuned LMMs
As demonstrated by GPT-3's emergent abilities with few-shot prompting, where the model could tackle tasks it hadn't seen during training, there's been a rising interest in instruction-fine-tuned LMMs. By allowing the models to be instruction-tuned, we can expect these models to perform a broader set of tasks and better alignment with human intents. This is line with the work done by OpenAI with InstructGPT and, more recently, GPT-4.
OpenAI has showcased the capability of their newer "GPT-4 with vision" model to follow instructions using visual inputs in their GPT4 technical report and GPT-4V(ision) System Card.

GPT-4 visual input example, Extreme Ironing:	
User	What is unusual about this image?
	 Source: https://www.barnorama.com/wp-content/uploads/2016/12/03-Confusing-Pictures.jpg
GPT-4	The unusual thing about this image is that a man is ironing clothes on an ironing board attached to the roof of a moving taxi.

Table 16. Example prompt demonstrating GPT-4's visual input capability. The prompt requires image understanding.

From the "GPT-4 Technical Report"

Following the announcement of OpenAI's **multimodal GPT-4,** there has been a surge in related research. As a result, multiple research labs have introduced instruction-tuned LMMs, including **LLaVA, MiniGPT-4,** and **InstructBlip.** They feature similar network architectures as previous LMMs but train on instruction-following datasets.



From "Multimodal Foundation Models: From Specialists to General-Purpose Assistants" paper

Exploring LLaVA - an instruction-tuned LMM

The network architecture of LLaVA resembles the one we reviewed before. This model connects a pre-trained CLIP visual encoder and the Vicuna language model via a projection matrix. In other words, they consider a simple linear layer to connect image features into the word embedding space. Specifically, they apply a trainable projection matrix called W to convert the image features into language embedding tokens with the same dimensionality as the word embedding space in the language model.

The authors of LLaVA chose these new linear projection layers that are more lightweight than the Q-Former connection module we saw for BLIP2 and the Perceiver Resampler and cross-attention layers from Flamingo.

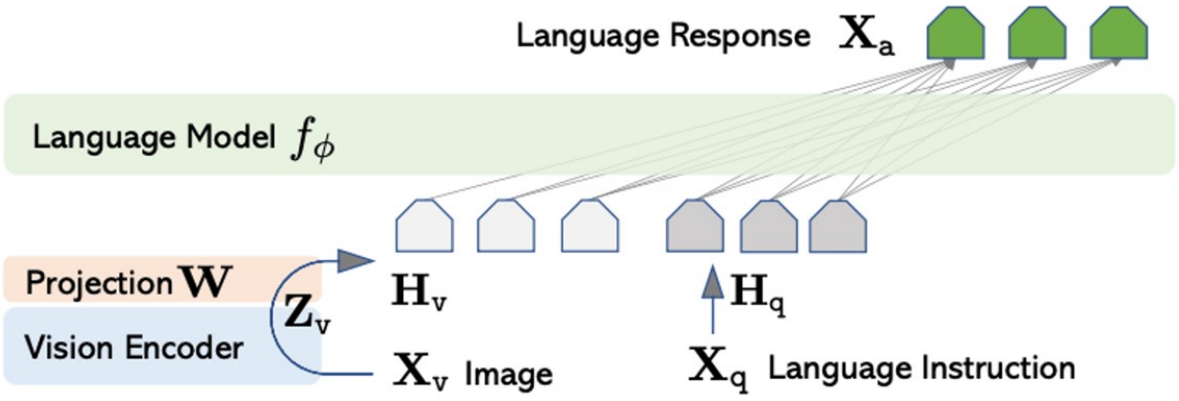
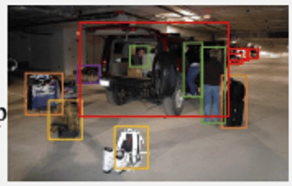


Figure 1: LLaVA network architecture.

From "Visual Instruction Tuning" paper

The authors then adopt a two-stage instruction-tuning procedure to train the model. First, they pre-train the projection matrix using a subset of the CC3M dataset, consisting of images and captions. Then, the model is finetuned end-to-end. Both the projection matrix and the LLM are trained on the proposed multimodal instruction-following data for daily user-oriented applications.
They also leverage GPT-4 to create a **synthetic dataset** consisting of multimodal instructions, drawing from widely available image-pair data.
In the dataset construction process, GPT-4 is shown symbolic representations of images using **captions** and the **coordinates of bounding boxes**, as depicted in the figure below. These representations are derived from the COCO dataset. This information is fed into GPT-4 as a prompt to generate training samples. The generated samples fall into **three categories**: question-answer conversations, detailed descriptions, and complex reasoning questions and answers.
They create 158K training samples in total.

Context type 1: Captions



A group of people standing outside of a black vehicle with various luggage.
Luggage surrounds a vehicle in an underground parking area
People try to fit all of their luggage in an SUV.
The sport utility vehicle is parked in the public garage, being packed for a trip
Some people with luggage near a van that is transporting it.

Context type 2: Boxes

person: [0.681, 0.242, 0.774, 0.694], person: [0.63, 0.222, 0.686, 0.516], person: [0.444, 0.233, 0.487, 0.34], backpack: [0.384, 0.696, 0.485, 0.914], backpack: [0.755, 0.413, 0.846, 0.692], suitcase: [0.758, 0.413, 0.845, 0.69], suitcase: [0.1, 0.497, 0.173, 0.579], bicycle: [0.282, 0.363, 0.327, 0.442], car: [0.786, 0.25, 0.848, 0.322], car: [0.783, 0.27, 0.827, 0.335], car: [0.86, 0.254, 0.891, 0.3], car: [0.261, 0.101, 0.787, 0.626]

Response type 1: conversation

Question: What type of vehicle is featured in the image?
Answer: The image features a black sport utility vehicle (SUV).
Question: Where is the vehicle parked?
Answer: The vehicle is parked in an underground parking area, likely in a public garage.
Question: What are the people in the image doing?
Answer: The people are trying to fit all of their luggage into the SUV, likely preparing for a trip.

Response type 2: detailed description

The image is an underground parking area with a black sport utility vehicle (SUV) parked. There are three people in the scene, with one person standing closer to the left side of the vehicle, another person in the middle, and the third person on the right side. They are all working together to pack their luggage into the SUV for a trip.

In the parking area, various luggage items are scattered around the vehicle. There are two backpacks, one located near the left rear wheel and the other closer to the right side of the vehicle. Additionally, there are two suitcases, one on the right side of the car and another further away near the center of the parking area. A bicycle can also be seen on the left side of the vehicle.

Other cars are parked around the main SUV, with one car positioned behind it and slightly to the left, another behind and slightly to the right, and the third car further behind on the right side.

Response type 3: complex reasoning

Question: What challenges do these people face?

Answer: In the image, a group of people is standing outside a black SUV in a parking area, surrounded by various pieces of luggage, including suitcases and backpacks. They are facing the challenge of fitting all their luggage into the black SUV. There are multiple suitcases and backpacks to be packed, which suggests that the group has a significant amount of belongings to accommodate. They might have to strategize and arrange the luggage efficiently to ensure that everything fits properly into the vehicle. Additionally, they need to consider the comfort of the passengers and visibility while driving, so the placement of the luggage must not obstruct the driver's view or make the passengers uncomfortable during the trip.

Table 1: One example to illustrate the instruction-following data. The top block shows the contexts such as captions and boxes used to prompt GPT, and the bottom block shows the three types of responses. Note that the visual image is not used to prompt GPT, we only show it here as a reference.

From "Visual Instruction Tuning" paper

The LLaVA model demonstrates the effectiveness of visual instruction tuning using language-only GPT-4. They show its capabilities by prompting the model with the same question and image as in the GPT-4 report. You can see the result below. The authors also

report a new SOTA by fine-tuning ScienceQA, a benchmark that contains 21k multimodal multiple-choice questions with rich domain diversity across three subjects, 26 topics, 127 categories, and 379 skills.

Visual input example, Extreme Ironing:

Source: https://www.barnorama.com/wp-content/uploads/2016/12/03-Confusing-Pictures.jpg

User	What is unusual about this image?
LLaVA	The unusual aspect of this image is a man ironing clothes on the back of a minivan or van. This is not a typical place to perform this activity, as one would usually iron clothes in a more stationary and safe location, such as a home, using a regular ironing board. The scene depicted in the image is peculiar as it involves a makeshift ironing setup on a vehicle, which can be both unsafe and unconventional. Additionally, it is not clear how the man is able to maintain balance and stability while ironing clothes in such an unstable environment.

From “Visual Instruction Tuning” paper

Beyond vision and language

In recent months, Image-to-text generative models have dominated the Large Multimodal Model (LMM) landscape. However, newer models have emerged that embrace a wider range of modalities beyond just vision and language.

For instance, **PandaGPT** is designed to handle any input data type, thanks to its integration with the **ImageBind** encoder.

There's also **SpeechGPT**, a model that integrates text and speech data and generates speech alongside text.

NExT-GPT stands out as a versatile model capable of receiving and producing outputs in any modality."

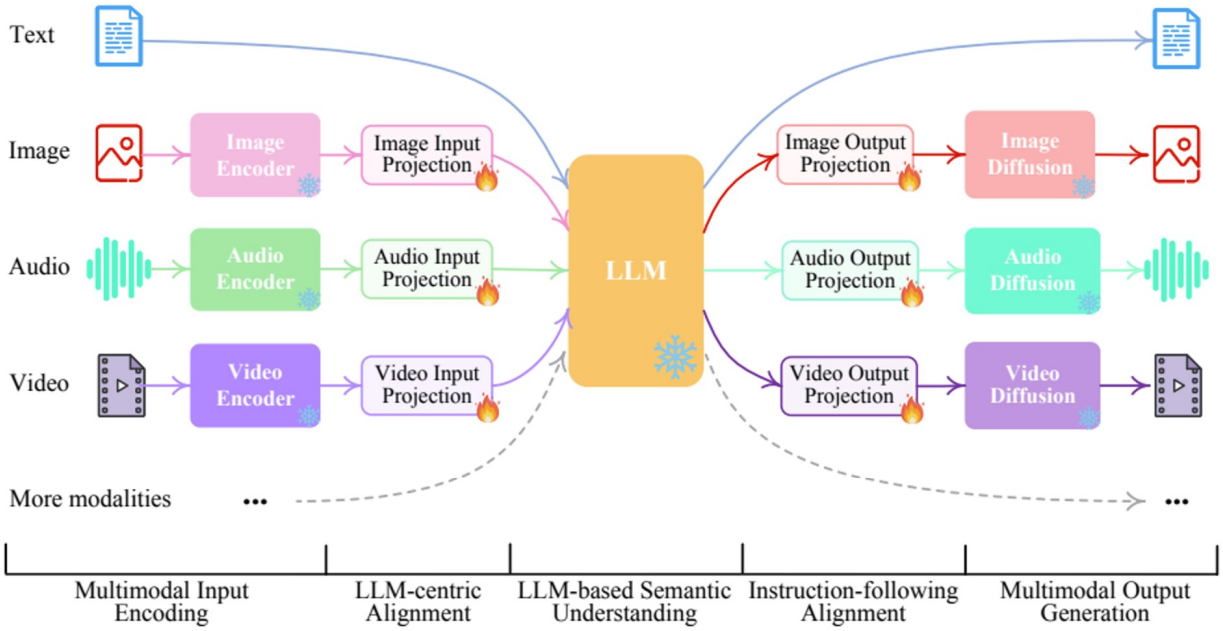
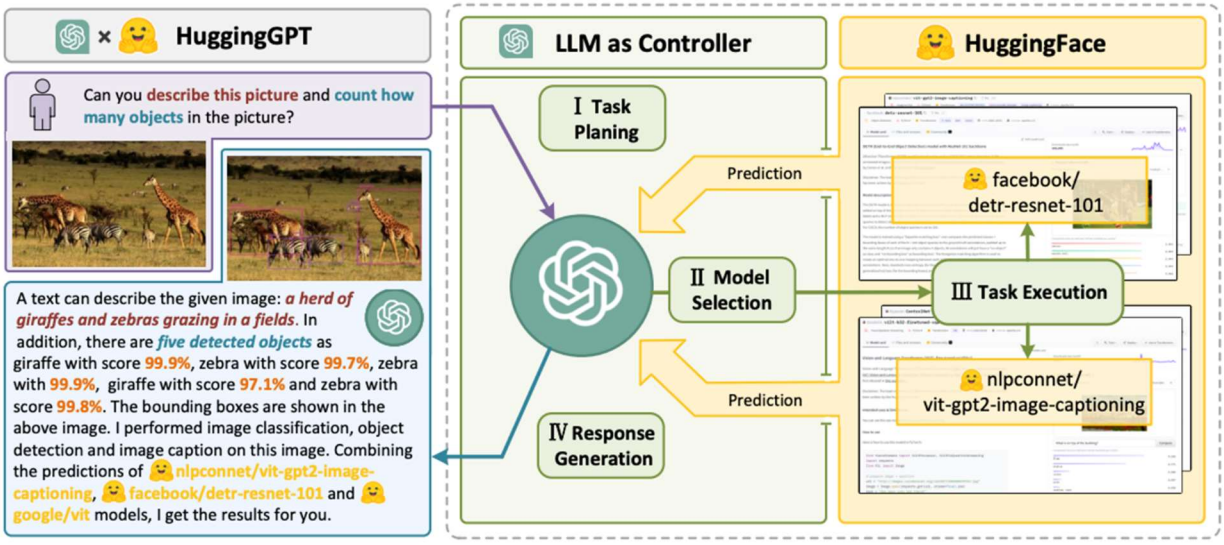


Figure 1: By connecting LLM with multimodal adaptors and diffusion decoders, NExT-GPT achieves universal multimodal understanding and any-to-any modality input and output.

From "NExT-GPT: Any-to-Any Multimodal LLM" paper

HuggingGPT is a novel system that integrates with the HuggingFace platform. It employs a Large Language Model (LLM) as its central controller. This LLM determines which specific model on HuggingFace is best suited for a task, selects that model, and then returns the model's output.



From "HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face" paper

Deep Lake and Multimodal LLMs

Deep Lake, differently from many other data lake and vector store products, is multi-modal and can store any data, from texts to images, videos, or audio. If you're interested in building multi-modal LLMs, this is something to consider, as you can store the different types of data you need in the same place. See this code example to see how to store images and texts in the same dataset. Moreover, here's the full list of data types that can be managed with Deep Lake.

HTYPE	DTYPE	COMPRESSION
generic	None	None
image	uint8	Required arg
image.rgb	uint8	Required arg
image.gray	uint8	Required arg
video	uint8	Required arg
audio	float64	Required arg
class_label	uint32	None
bbox	float32	None
bbox.3d	float32	None
intrinsics	float32	None
segment_mask	uint32	None
binary_mask	bool	None
keypoints_coco	int32	None
point	int32	None
polygon	float32	None
text	str	None
json	Any	None
list	List	None
dicom	None	dcm
nifti	None	Required arg
point_cloud	None	las
mesh	None	ply
instance_label	uint32	None
embedding	None	None
link	str	None
sequence	None	None

Conclusion

In this module, we delved into the emergent field of LMMs. We examined the leading models that combine both vision and language modalities. We learned that instruction-tuning allows these models to achieve greater generalization on tasks they haven't encountered before. Furthermore, we were introduced to advanced LMMs capable of integrating an even wider range of modalities. Lastly, we discussed the utility of Deep Lake in fine-tuning LMMs.

Expanding the Context Window

Introduction

In this lesson, we will discuss context windows in language models, their importance, and the limitations of the original Transformer architecture in handling large context lengths. We explore various optimization techniques that have been developed to expand the context window, including ALiBi Positional Encoding, Sparse Attention, FlashAttention, Multi-Query Attention, and the use of large RAM GPUs.

We also introduce the latest advancements in this field, such as FlashAttention-2 and LongNet, which aim to push the context window to an unprecedented scale.

The Importance of The Context Length

The context window refers to the number of input tokens the model can process simultaneously. In current models like GPT-4, this context window is around 32K tokens. To put this into perspective, this roughly translates to the size of 50 pages. However, recent advancements have pushed this limit to an impressive 100K tokens (check Claude by Anthropic), equivalent to 156 pages.

The context length of an LLM is a critical factor for several reasons. Firstly, it allows the model to process larger amounts of data at once, providing a more comprehensive understanding of the context. This is particularly useful when you want to feed a large amount of custom data into an LLM and ask questions about this specific data.

For instance, you might want to input a large document related to a specific company or problem and ask the model questions about this document. With a larger context window, the LLM can scan and retain more of this custom information, leading to more accurate and personalized responses.

Limitations of the Original Transformer Architecture

The original Transformer architecture, however, has some limitations when it comes to handling large context lengths. The main issue lies in the computational complexity of the Transformer architecture. Specifically, the attention layer computations in the Transformer architecture have a quadratic time and space complexity with respect to the number of input tokens n. This means that as the context length increases, the computational resources required for training and inference increase exponentially.

To understand this better, let's understand the computational complexity of the Transformer architecture. The complexity of the attention layer in the Transformer model is

$O(n^2d+nd^2)$, where n is the context length (number of input tokens) and d is the embedding size.
This complexity arises from two main operations in the attention layer: linear projections to get Query, Key, and Value matrices (complexity ~ $O(nd^2)$) and multiplications of these matrices (complexity ~ $O(n^2d)$). As the context length or embedding size increases, the computational complexity grows quadratically, making it increasingly challenging to process larger context lengths.

Optimization Techniques to Expand the Context Window
Despite these challenges, researchers have developed several optimization techniques to speed up the Transformer and increase the context length to 100K tokens. Let's explore some of these techniques:

1. **ALiBi Positional Encoding**: The original Transformer uses Positional Sinusoidal Encoding, which lacks the ability to extrapolate to larger context lengths. ALiBi, or Attention with Linear Biases, is a positional encoding technique that can be used to train the model on a small context and then fine-tune it on a larger one.
2. **Sparse Attention**: This technique reduces the number of computations by considering only some tokens when calculating the attention scores. This makes the computation linear with respect to n, significantly reducing the computational complexity.
3. **FlashAttention**: This is an efficient implementation of the attention layer for GPU. It optimizes the memory utilization of the GPU by splitting the input matrices into blocks and computing the attention output with respect to these blocks.
4. **Multi-Query Attention (MQA)**: MQA optimizes the memory consumption of the key/value decoder cache by sharing weights across all attention heads when linearly projecting Key and Value matrices.
5. **Large RAM GPUs**: You need a lot of RAM in the GPU to fit a large context. Therefore, models with larger context windows are often trained on GPUs with large RAM, such as 80GB A100 GPUs.

FlashAttention-2
Building on the success of FlashAttention, researchers have recently developed FlashAttention-2, a more efficient version of the algorithm that further optimizes the attention layer's speed and memory usage. This new version has been completely rewritten from scratch, leveraging the new primitives from Nvidia. The result is a version that is about 2x faster than its predecessor, reaching up to 230 TFLOPs/s on A100 GPUs.
FlashAttention-2 introduces several improvements over the original FlashAttention.

- Firstly, it reduces the number of non-matmul FLOPs, which are 16x more expensive than matmul FLOPs, by tweaking the algorithm to spend more time on matmul FLOPs.
- Secondly, it optimizes parallelism by parallelizing over batch size, number of heads, and the sequence length dimension. This results in significant speedup, especially for long sequences.

- Lastly, it improves work partitioning within each thread block to reduce the amount of synchronization and communication between different warps, resulting in fewer shared memory reads/writes.
- In addition to these improvements, FlashAttention-2 also introduces new features, such as support for head dimensions up to 256 and multi-query attention (MQA), further expanding the context window.

With these advancements, FlashAttention-2 is a step forward in expanding the context window (without overcoming the fundamental limitations of the original Transformer architecture).

LongNet: A Leap Towards Billion-Token Context Window

Building on the advancements in Transformer optimization, a recent innovation comes from the paper "LONGNET: Scaling Transformers to 1,000,000,000 Tokens". This paper introduces a novel approach to handling the computational complexity of the Transformer architecture, pushing the context window potentially to an unprecedented 1 billion tokens.

The core innovation in LongNet is the introduction of "dilated attention." This novel attention mechanism expands the attentive field exponentially as the distance between tokens grows, thereby decreasing attention allocation exponentially as the distance increases. This design principle helps to balance the limited attention resources with the necessity to access every token in the sequence.

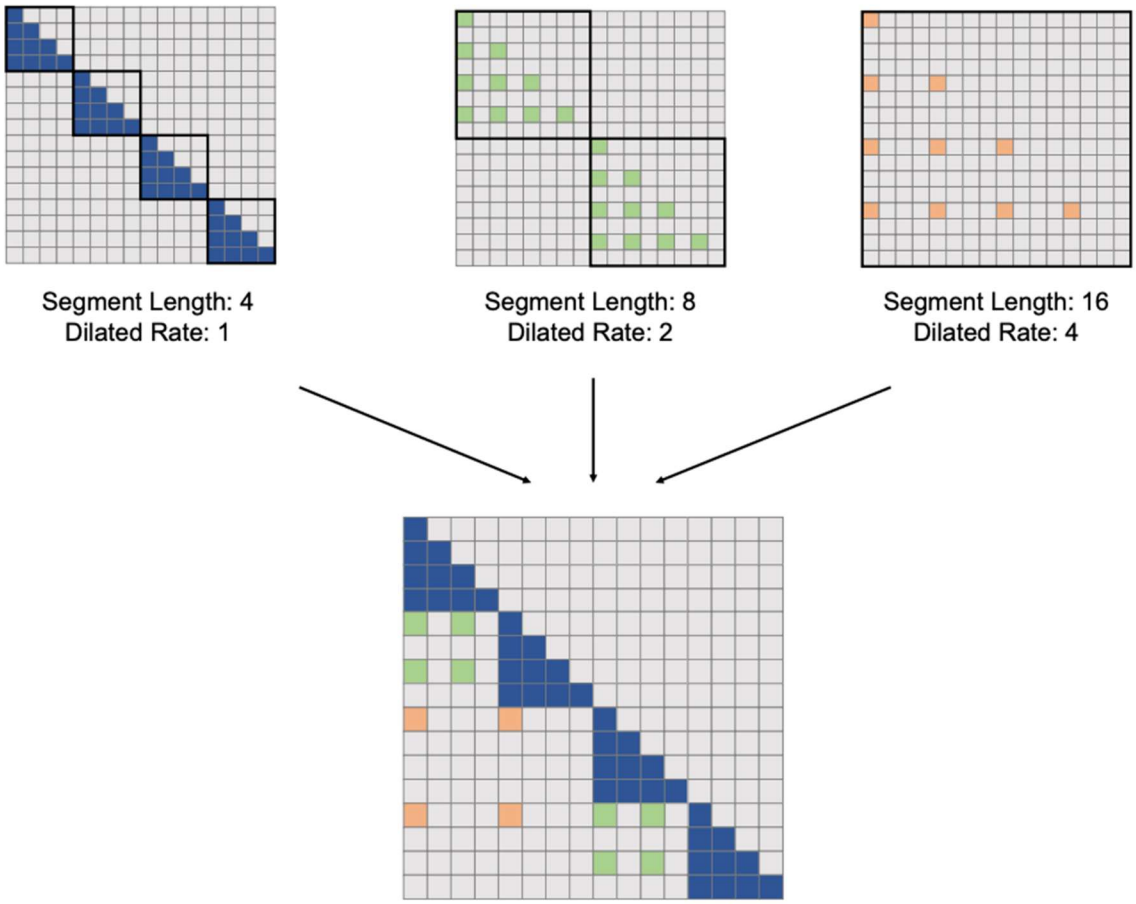


Image from the paper "LONGNET: Scaling Transformers to 1,000,000,000 Tokens". Building blocks of dilated attention used in LONGNET. It consists of a series of attention patterns for modeling short- and long-range dependency. The number of attention patterns can be extended according to the sequence length.

The dilated attention mechanism in LongNet achieves a linear computational complexity, a significant improvement over the quadratic complexity of the standard Transformer.

Method	Computation Complexity
Recurrent	$\mathcal{O}(Nd^2)$
Vanilla Attention	$\mathcal{O}(N^2d)$
Sparse Attention	$\mathcal{O}(N\sqrt{N}d)$
Dilated Attention (This Work)	$\mathcal{O}(Nd)$

Image from the paper "LONGNET: Scaling Transformers to 1,000,000,000 Tokens".

Comparison of computation complexity among different methods. N is the sequence length, and d is the hidden dimension.

Conclusion

In this lesson, we examined the limitations of the original Transformer architecture in handling large context lengths, primarily due to its quadratic computational complexity. We then explored various optimization techniques developed to overcome these limitations, including ALiBi Positional Encoding, Sparse Attention, FlashAttention, Multi-Query Attention, and the use of large RAM GPUs.

We also discussed the latest advancements in this field, such as FlashAttention-2, which further optimizes the speed and memory usage of the attention layer, and LongNet, a novel approach that introduces "dilated attention" to potentially expand the context window to an unprecedented 1 billion tokens.

These advancements are critical in pushing the boundaries of language models, enabling them to process larger amounts of data at once and providing a more comprehensive understanding of the context, leading to more accurate and personalized responses.