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

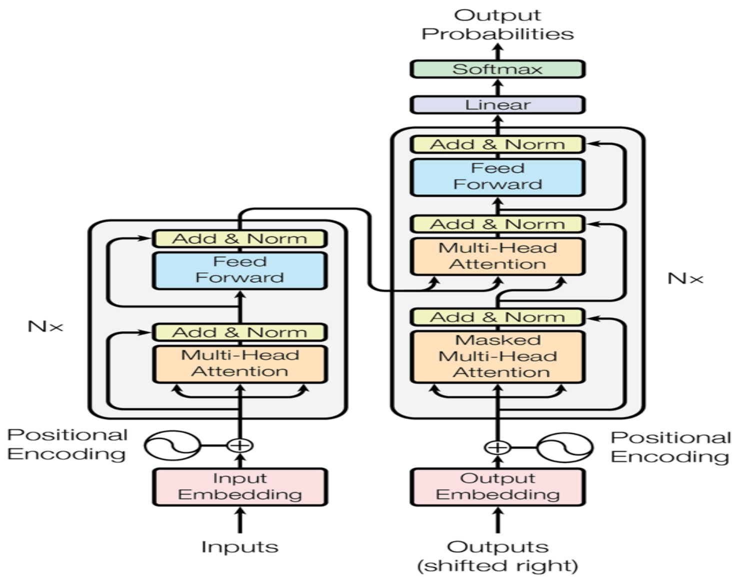
In this lesson, we will dive deeper into Transformers and provide a comprehensive understanding of their various components. We will also cover the network's inner mechanisms.
We will look into the seminal paper “Attention is all you need” and examine a diagram of the components of a Transformer. Last, 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



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.

```
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)
```

The sample code.

```
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.

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

The sample code.

```
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)
      )
    )
  )
)
```

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.

```
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)
```

The sample code.

```
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>)
```

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.

```
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)
```

```
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>)
```

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.

```
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)
```

The sample output.

```
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>)
```

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).
In this Notebook, you can find the code for this lesson.

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

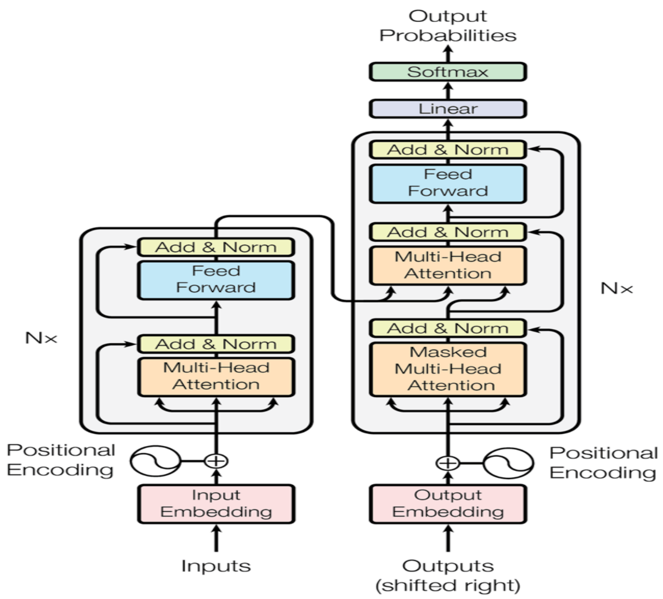


Image taken from the "*Attention is all you need*" *paper*

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.

```
from transformers import AutoModel, AutoTokenizer
BART = AutoModel.from_pretrained("facebook/bart-large")
print(BART)
```

```
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)
  )
)
```

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.

```
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'])
```

```
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

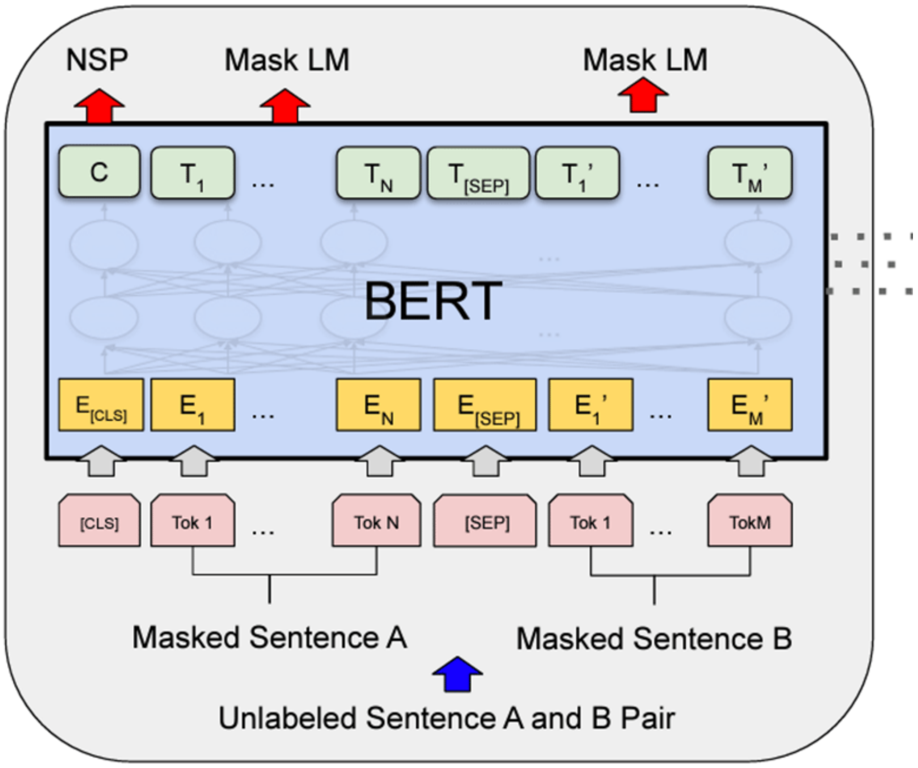


Image taken from the "BERT: Pre-training of Deep Bidirectional Transformers for LanguageUnderstanding" paper

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.

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

print(BERT)
```

```
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()
  )
)
```

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.

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

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

print(lbl)
```

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

The output.

The Decoder-Only Architecture

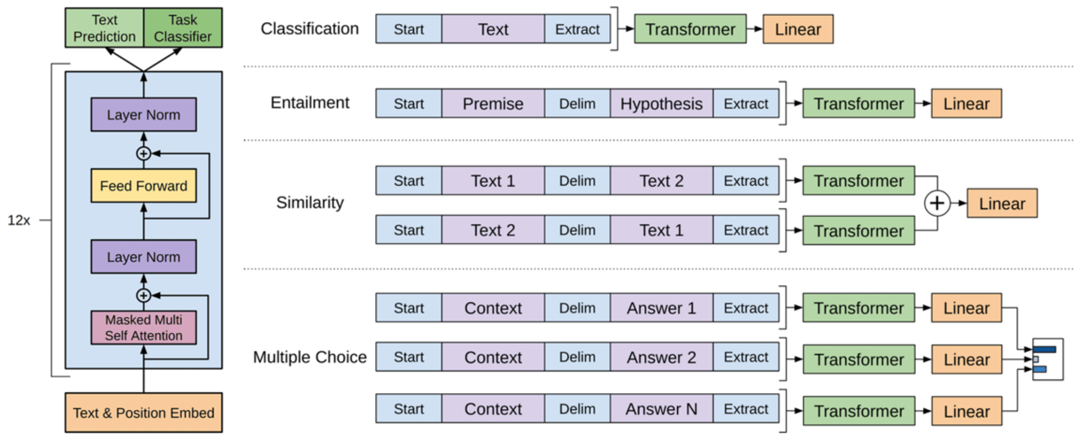


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

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.

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

print(gpt2)


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))
```

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."

```
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'])
```

```
>  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.

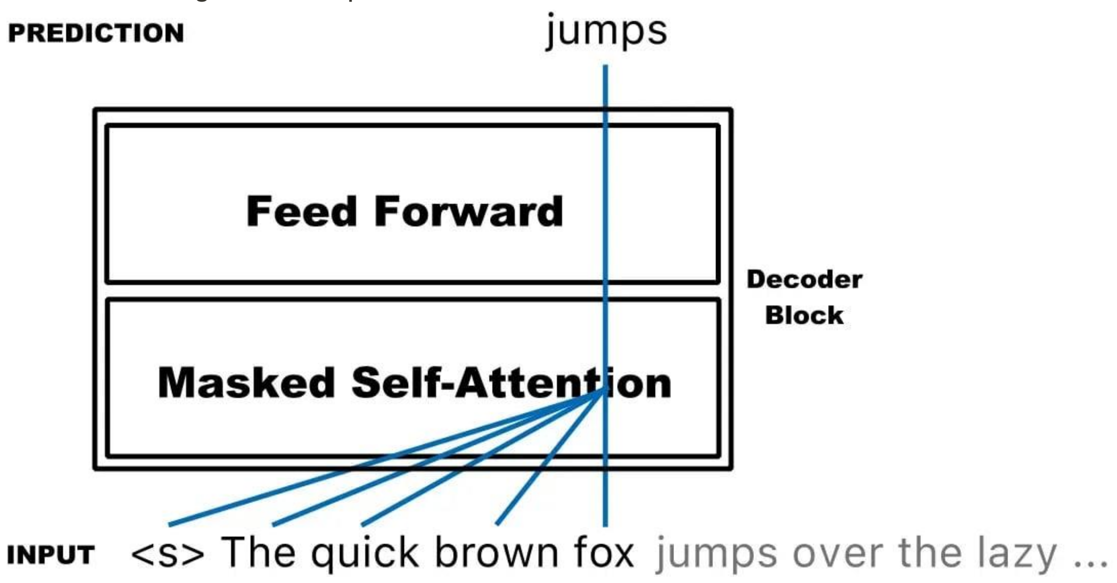
In contrast to conventional Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, the transformer architecture departs from recurrence and adopts self-attention mechanisms, resulting in substantial advancements in speed and scalability. An immensely powerful architecture was unleashed 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.

```
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 outputCopy
```

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`:

```python
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) # 2.1485556947850033
```

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?

```python
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 -> lower
```

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.

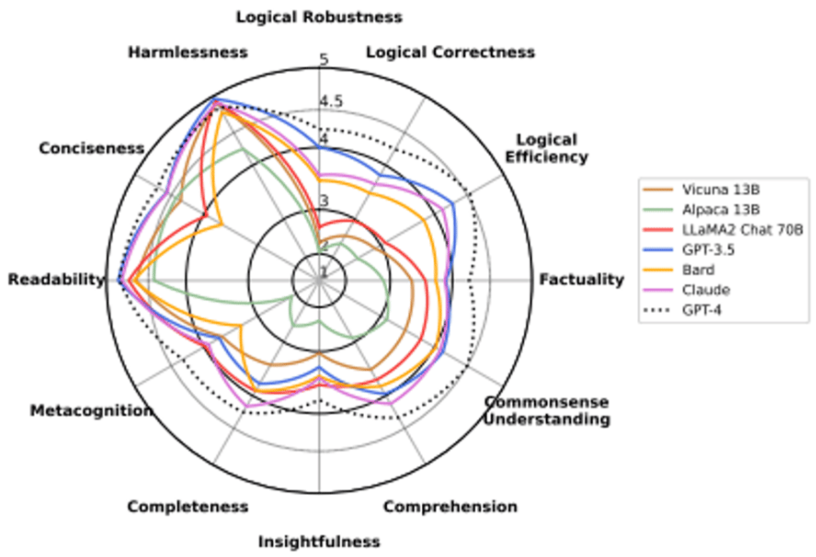


Figure 1: FLASK is a comprehensive evaluation framework for language models considering: `Logical Thinking` (Logical Robustness, Logical Correctness, Logical Efficiency), `Background Knowledge` (Factuality, Commonsense Understanding), `Problem Handling` (Comprehension, Insightfulness, Completeness, Metacognition), `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.

```
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.

```
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'])
```

```
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.

```
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'])
```

```
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.

```python
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'])
```

```
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.

```
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'])
```

```
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 will 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.