Building GPT-2 From Scratch (To Undertand How LLMs Work)

Updated: 7 February 2025 version 1.0

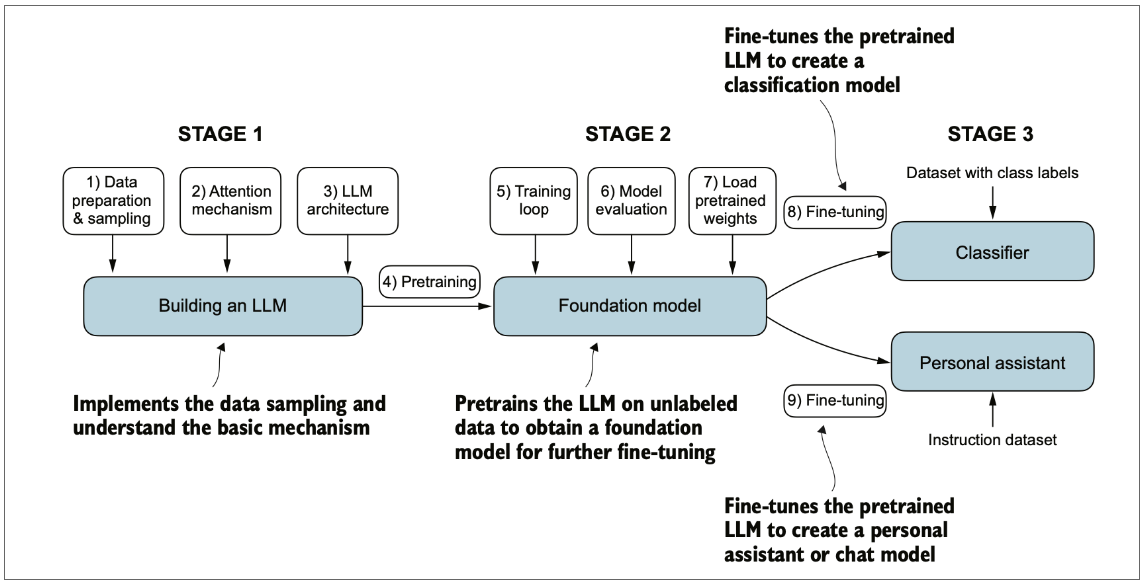
Introduction

This class walks through the process of building a transfomer-based large language model from scratch (i.e. GPT-2) to give students a feel for what is involved and to be understand some key concepts. We will use a combination of lecture, code, and practice examples in this class.

Instructors:

- Dr Leslie Teo, Senior Director, AI Products, AI Singapore
- Dr William Thji, Head, Applied Research in Foundation Models, AI Products, AI Singapore

Figure 0: Overview of Lecture


Fine-tunes the pretrained LLM to create a classification model
STAGE 1
STAGE 2
STAGE 3
1) Data preparation & sampling
2) Attention mechanism
3) LLM architecture
5) Training loop
6) Model evaluation
7) Load pretrained weights
8) Fine-tuning
Dataset with class labels
Classifier
4) Pretraining
Building an LLM
Foundation model
Personal assistant
Implements the data sampling and understand the basic mechanism
Pretrains the LLM on unlabeled data to obtain a foundation model for further fine-tuning
9) Fine-tuning
Instruction dataset
Fine-tunes the pretrained LLM to create a personal assistant or chat model

Part 1: Introduction to Large Language Models (LLMs)

Learning Objectives:

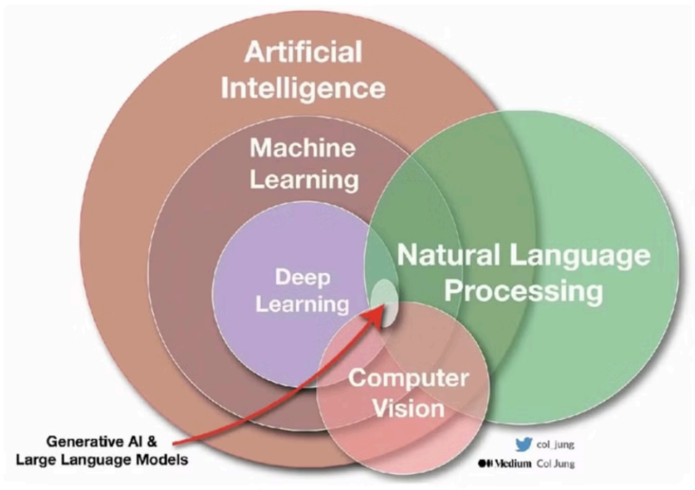
- Understand what Large Language Models (LLMs) are and their capabilities.
- Explore common use cases and applications of LLMs.
- Learn about the brief history of LLMs.
- Understand why we study GPT-2.

1.1 What are LLMs?

- **Definition and Capabilities:**
  - **Large Language Models (LLMs)** are deep learning models trained on massive datasets of text and code. They are designed to **understand and generate human-like text**, **translate languages**, **write different kinds of creative content**, and **answer your questions in an informative way**, even if they are open ended, challenging, or strange. Today LLMs also encompass speech, music (audio), images, and video.
  - They learn **statistical patterns and relationships** between words, phrases, and sentences, enabling them to perform a wide range of language-based tasks.
  - **Key characteristics:**
    - **Large:** These models have billions, even trillions, of parameters (weights and biases).
    - **Language-focused:** They are specifically trained for understanding and generating natural language.
    - **Pre-trained:** They are first trained on a vast corpus of general text data and can then be fine-tuned for specific tasks.

Figure 1. AI and LLMs

LLMs and Gen AI



- **Examples:** * GPT-3, GPT-4 (OpenAI) - These are later, more powerful versions. * BERT (Google) - A different type of Transformer (encoder-only). * LaMDA, PaLM (Google) - Other large-scale models. * BLOOM (BigScience) - An open-source multilingual LLM. * LLaMA (Meta) - Another powerful family of LLMs. * Qwen (Alibaba) - Representing Chinese Big Tech. * Deepseek (Deepseek) - Powerful open weights models. * ChatGPT, Claude, Gemini: Popular applications built on top of LLMs.

- **Generative Nature of LLMs**
    - LLMs are considered "generative" because they **create new text sequences** that are not simply copied from their training data.
    - They generate text by **sampling from a probability distribution over possible word sequences**. At each step, the model predicts the probability of the next word, given the preceding words, and then a word is selected (sampled) based on these probabilities.
    - This probabilistic nature allows LLMs to produce diverse and creative outputs.

Why study GPT-2 if we have newer, bigger models?

- **Foundational Understanding:** GPT-2 is a great model for learning the **fundamentals of transformers and LLMs**. Its architecture is simpler than the later models, making it easier to dissect and understand.
- **Accessibility:** The model and its pre-trained weights are readily available through libraries like Hugging Face Transformers.
- **Resource-Friendly:** You can experiment with and even fine-tune GPT-2 on less powerful hardware compared to the behemoths like GPT-3 or GPT-4.

In [ ]:

```python
# Example: Using SEA-LION
import requests
import json
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

# Access your variables
API_KEY = os.getenv('API_KEY')

url = "https://api.sea-lion.ai/v1/chat/completions"

headers = {
    "accept": "application/json",
    "Content-Type": "application/json",
    "Authorization": "Bearer " + API_KEY,
}

payload = {
    "messages": [{"role": "user", "content": "Hi SEA-LION"}],
    "model": "aisingapore/llama3.1-8b-cpt-sea-lionv3-instruct",
    "stream": False,
}

response = requests.post(url, headers=headers, json=payload)
print(json.dumps(response.json(), indent=2))
```

1.2 Use Cases and Applications of LLMs

- **Common Applications:**
    - **Proprietary Services:** Using pre-built and hosted LLM services like **ChatGPT** for various tasks (e.g., writing assistance, question answering, code generation).
    - **Custom Deployments:** Running your own instance of an LLM (like Llama) locally on your own hardware. This gives you more control over the model, data privacy, and customization.
    - **API Integration:** Deploying LLMs as API endpoints that other applications can interact with. This allows you to integrate LLM capabilities into your own software systems.

Examples

- **Chatbots and Conversational AI:** Creating intelligent agents that can engage in natural language conversations.
- **Text Generation:** Writing stories, articles, scripts, poems, code, and other forms of creative content.
- **Machine Translation:** Translating text from one language to another.
- **Question Answering:** Providing answers to questions based on a given context or general knowledge.
- **Text Summarization:** Condensing large amounts of text into shorter, concise summaries.
- **Code Completion and Generation:** Assisting developers by suggesting code completions and even generating entire code blocks.
- **Sentiment Analysis:** Determining the emotional tone or sentiment expressed in a piece of text.
- **And many more emerging applications...**

1.4 Training an LLM - Let's Use GPT-2

- **Overview of GPT-2**
  - GPT-2 is a **decoder-only Transformer model**. This means it's designed for text generation, predicting the next word in a sequence given the preceding words. In contrast, an encoder-decoder architecture, such as the one originally described in **Attention is All You Need**, is more common in tasks like machine translation where the input and output can have different lengths.
  - It was developed by OpenAI and released in stages due to concerns about potential misuse.
  - It was a significant advancement in the field of NLP, demonstrating the power of scaling up model size and training data.
- **GPT-2 Miniseries and Scaling Laws**
  - OpenAI released different sizes of GPT-2 models (small, medium, large, XL), each with an increasing number of parameters.
  - **Scaling Laws:** Research on GPT-2 and other LLMs has shown that there are **predictable relationships** between model size, dataset size, compute used for training, and the resulting model performance. Generally, **larger models trained on more data with more compute tend to perform better**. This also informs why we are focusing on GPT-2 now, which can help us to learn some of the fundamental concepts.
- **Availability**
  - Pre-trained weights and code for GPT-2 are publicly available, primarily through libraries like **Hugging Face Transformers**. This makes it relatively easy to get started with using and experimenting with GPT-2.
- **Demonstration**
  - We will (later in the course) run a simplified GPT-2 training example and show the **objective (predicting the next word)** and how the **loss function decreases over time** as the model learns.

In [ ]:

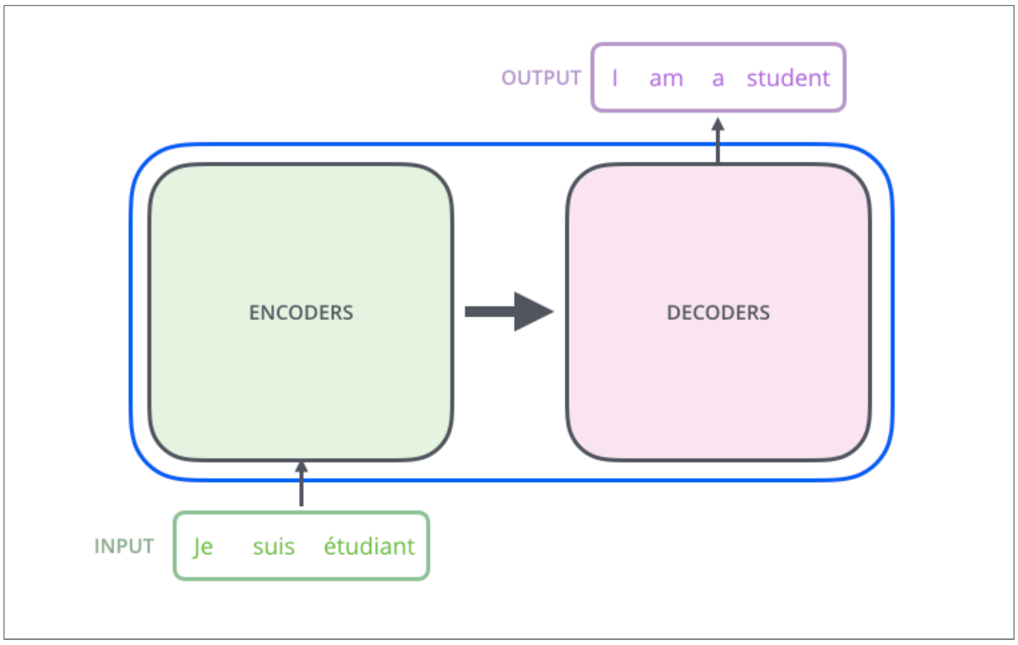
```python
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained("gpt2")
tokenizer = AutoTokenizer.from_pretrained("gpt2")

# Set padding token
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = model.config.eos_token_id

# Prepare input with prompt
prompt = "GPT2 is a model developed by OpenAI. Tomorrow is Chinese New Year"
inputs = tokenizer(
    prompt,
    return_tensors="pt",
    padding=True,
    truncation=True,
    max_length=100,
    add_special_tokens=True
)

# Generate text
gen_tokens = model.generate(
    input_ids=inputs.input_ids,
    attention_mask=inputs.attention_mask,
    do_sample=True,
    temperature=0.9,
    max_length=100,
    pad_token_id=tokenizer.pad_token_id,
    eos_token_id=tokenizer.eos_token_id,
)

# Decode and print the generated text
gen_text = tokenizer.batch_decode(gen_tokens, skip_special_tokens=True)[0]
print(gen_text)
```

1.5 Core Concepts: Transformers and Attention

- **1.5.1 The Transformer Architecture:**
    - **Encoder-Decoder Structure (Original Transformer):**
        - The original Transformer had an **encoder** and a **decoder**.
        - **Encoder:** Processes the input sequence and creates a contextualized representation.
        - **Decoder:** Generates the output sequence based on the encoder's representation and its own previous outputs.
        - **Example:** Machine translation - the encoder processes the source language sentence, and the decoder generates the target language translation.
    - **GPT-2: Decoder-Only Model:**
        - GPT-2 uses **only the decoder** part of the Transformer. It's designed for text generation.
        - It predicts the next word in a sequence given the preceding words.

Reference: The Illustrated Transformer by Jay Alammar

Figure 2: Transformer Architecture (Encoder-Decoder)



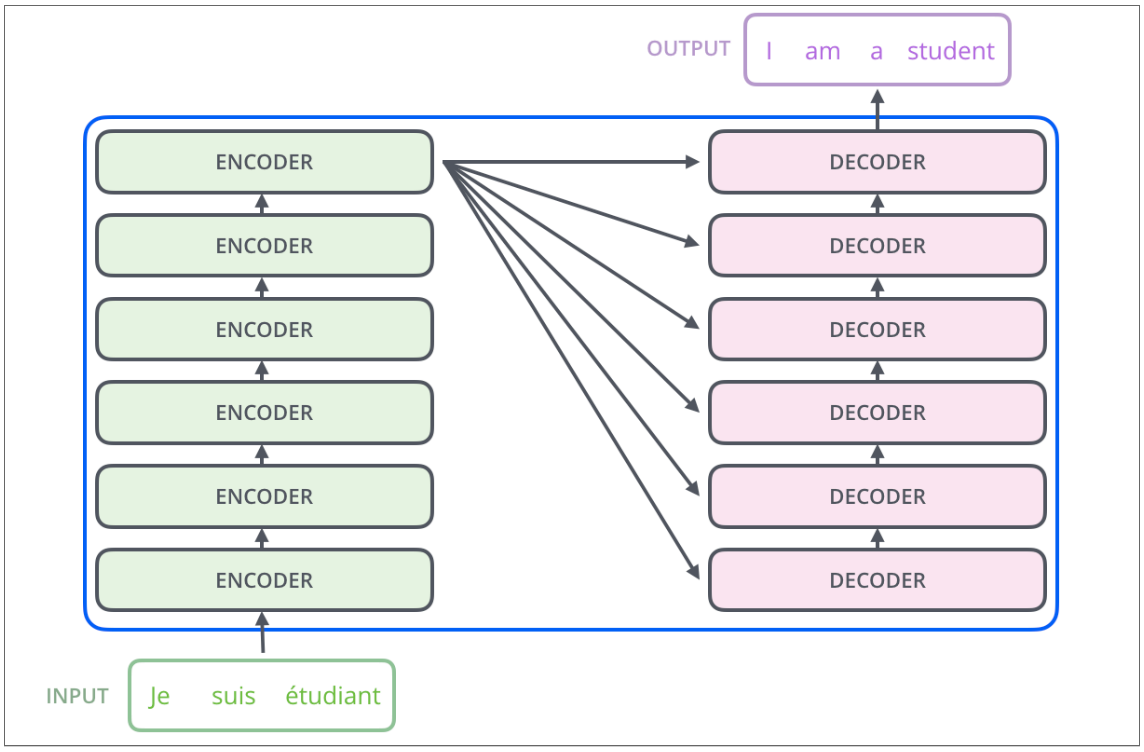
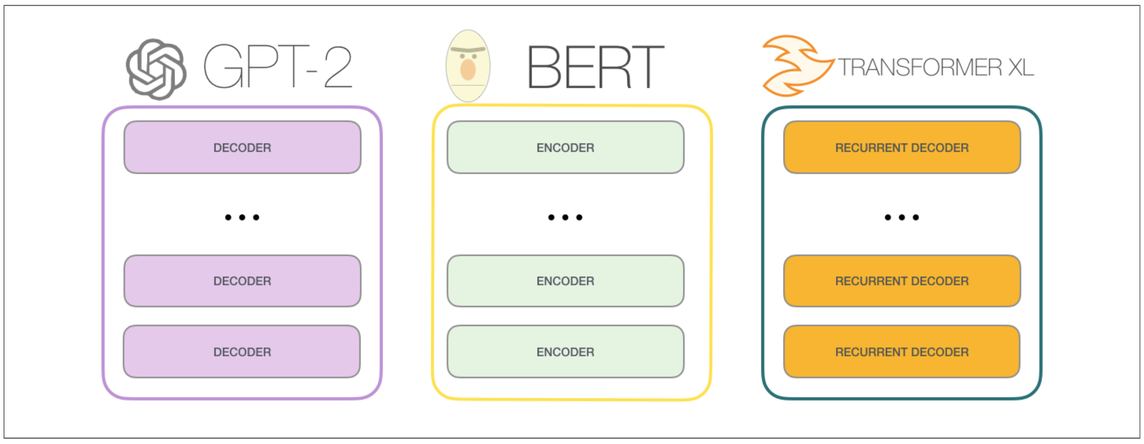

OUTPUT
I am a student
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
DECODER
DECODER
DECODER
DECODER
DECODER
DECODER
INPUT
Je suis étudiant

Figure 3: GPT-2 Architecture (Decoder-Only)



Key Components:

- **Embedding Layer:** Converts words into numerical vectors (**embeddings**). These vectors capture the semantic meaning of words.
- **Positional Encoding:** Adds information about the **position** of words in the sequence to the embeddings. This is crucial because, unlike RNNs, Transformers don't inherently process data sequentially.
- **Decoder Layers (GPT-2 has multiple):** Each layer consists of:
- **Masked Self-Attention:** Allows the model to attend to **preceding** words in the sequence when predicting the next word (more on this below). The "masked" part ensures that the model doesn't "cheat" by looking at future words during training.
- **Feed-Forward Neural Network:** A simple fully connected network that further processes the output of the attention layer.
- **Output Layer:** A linear layer followed by a softmax function that produces a **probability distribution over the vocabulary**, indicating the likelihood of each word being the next word. The model assigns a probability to every possible word in its vocabulary, indicating how likely it is to be the next word.

- **1.5.2 Attention Mechanism:**
  - **Intuition: Attention** allows the model to **focus on the most relevant parts of the input sequence** when making a prediction. Think of it like highlighting the important words in a sentence when you're trying to understand its meaning.
  - **Analogy:** Imagine you're translating a sentence. You don't just translate each word in isolation. You consider the entire sentence, paying more attention to certain words that are crucial for understanding the meaning of the word you're currently translating. This is how attention allows the model to "weigh" the importance of other words.
  - **Self-Attention:** In **self-attention**, the input sequence attends to **itself**. Each word in the sequence attends to every other word in the same sequence.
  - **Masked Self-Attention (in GPT-2):** In GPT-2, the self-attention is "**masked**" during training. This means that when a word is attending to other words, it can **only attend to the words that came before it** in the sequence. This is because GPT-2 is trained to predict the next word, and allowing it to see future words would be like giving it the answers during a test.

How it Works (Simplified):

- Scenario: We have the input sentence "The cat sat on the mat". We want to understand how the attention mechanism works when focusing on the word "sat".
- Simplified Analogy: Imagine you're reading this sentence. When you get to "sat", you instinctively pay more attention to "cat" and "mat" because they tell you who is sitting and where. Attention in Transformers works similarly.

Key Concepts:

- Queries, Keys, and Values: These are vector representations of the words. Think of them as different facets of the word. For our example, let's simplify and say each word has a "meaning" vector.
    - Query: The word we're currently focusing on ("sat" in this case). It's asking "What am I related to?"
    - Keys: The other words in the sentence ("the", "cat", "on", "the", "mat"). They represent "What information do I offer?"
    - Values: Also the other words. They represent the actual information the word carries. (In reality, Keys and Values are often the same). * Scoring: We calculate a "score" between the Query ("sat") and each Key (every word). This score represents how much the other word relates to the current word. A higher score means more attention. We can use a simple dot product for this example. * Softmax: We apply a softmax function to the scores. This converts the scores into probabilities that sum up to 1. These probabilities represent the attention weights. * Weighted Sum: We multiply each Value (the actual word representation) by its corresponding attention weight and sum them up. This weighted sum is the output of the attention mechanism for the word "sat".

Example:
Let's pretend our word vectors are just single numbers (for simplicity). In reality, they'd be much larger vectors.
Word Meaning (Query/Key/Value) The 1 cat 5 sat 3 on 2 the 1 mat 4
Export to Sheets Query (sat): 3
Keys (all words): 1, 5, 3, 2, 1, 4
Scoring (Dot Product):
"sat" x "the" = 3 * 1 = 3 "sat" x "cat" = 3 * 5 = 15 "sat" x "sat" = 3 * 3 = 9 "sat" x "on" = 3 * 2 = 6 "sat" x "the" = 3 * 1 = 3 "sat" x "mat" = 3 * 4 = 12 Softmax (Simplified): Let's just normalize these scores so they sum to 1 (a true softmax is a bit more complex). The exact values aren't crucial for the illustration:
"the": 3 / (3+15+9+6+3+12) ≈ 0.04 "cat": 15 / (3+15+9+6+3+12) ≈ 0.24 "sat": 9 / (3+15+9+6+3+12) ≈ 0.14 "on": 6 / (3+15+9+6+3+12) ≈ 0.10 "the": 3 / (3+15+9+6+3+12) ≈ 0.04 "mat": 12 / (3+15+9+6+3+12) ≈ 0.19 Weighted Sum:
(0.04 * 1) + (0.24 * 5) + (0.14 * 3) + (0.10 * 2) + (0.04 * 1) + (0.19 * 4) ≈ 1.87
Interpretation:
The output for "sat" is approximately 1.87. Notice how "cat" and "mat" have the highest attention weights (0.24 and 0.19 respectively). This means the model paid more attention to "cat" and "mat" when processing "sat", which makes intuitive sense. The number 1.87 is a weighted representation that encodes the context around "sat".

Figure 4: Self-Attention Mechanism

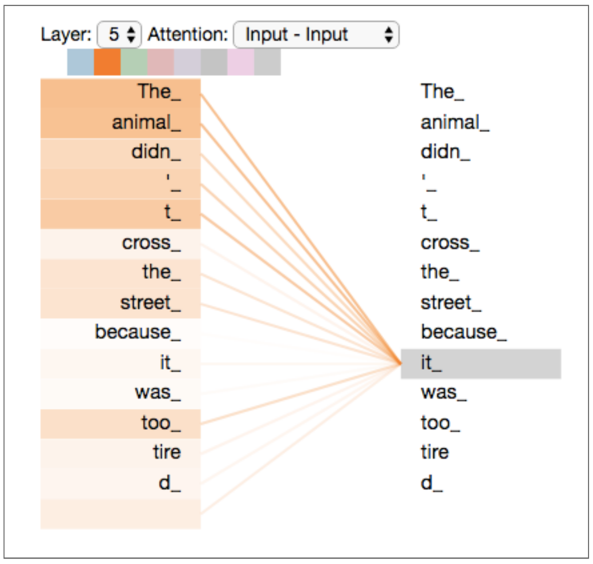
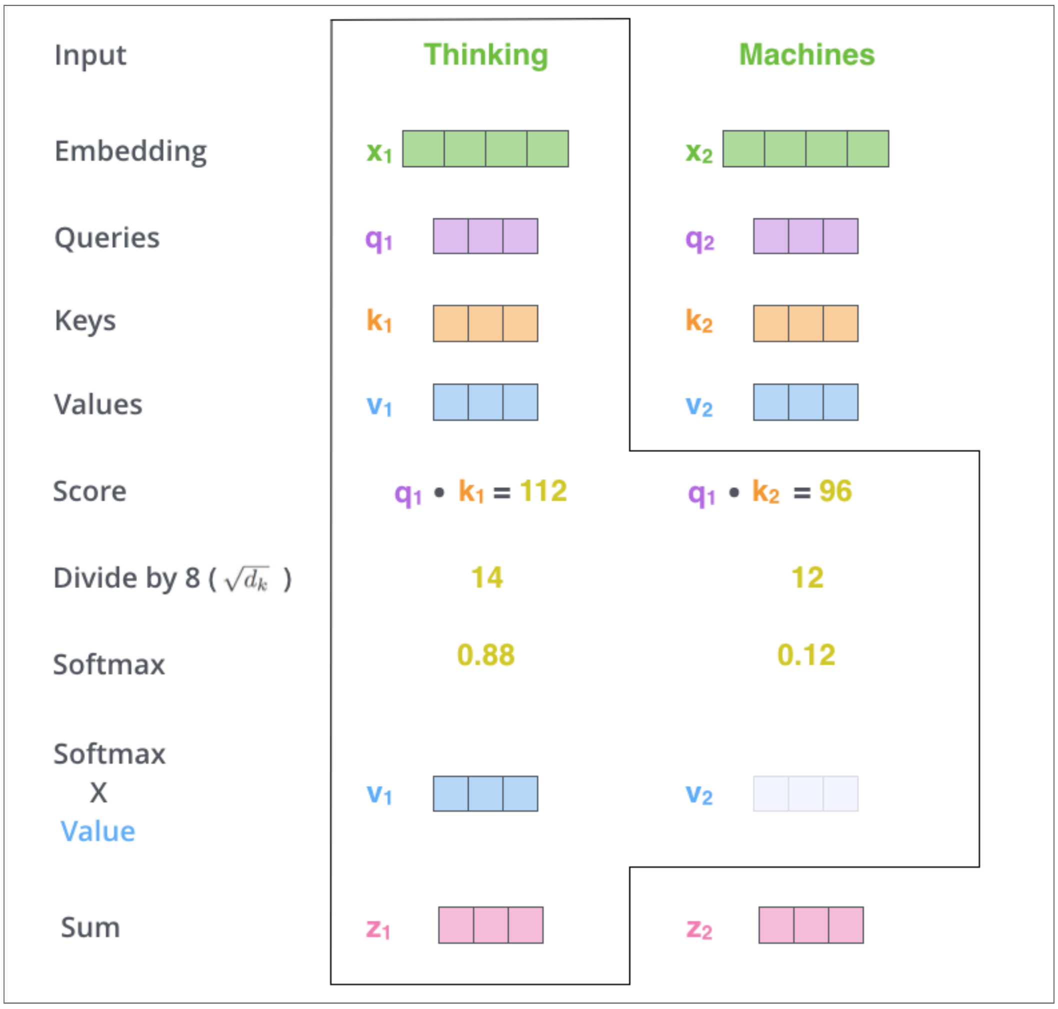


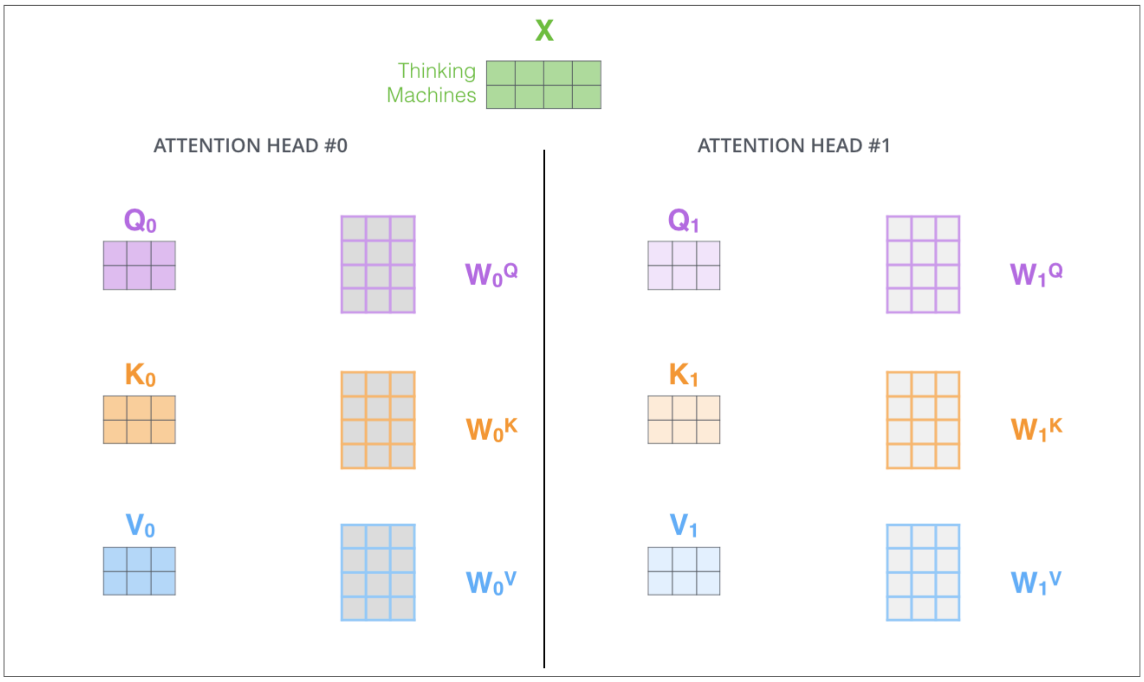
Figure 5: Self-Attention Mechanism (Detailed)



Multi-Head Attention:

- Instead of having just one set of Q, K, and V vectors per word, **multi-head attention** uses multiple sets (multiple "**heads**").
- Each head learns to attend to **different aspects** of the input sequence.
- **Analogy:** Think of it like having multiple experts examining the same sentence, each focusing on a different aspect (e.g., syntax, semantics, coreference).
- The outputs of the multiple heads are concatenated and then passed through a linear layer to produce the final output.

Figure 6: Multi-Head Attention



Part 2: Data Preparation and Tokenization

Learning Objectives:

- Understand the importance of data preparation for LLMs.
- Learn what tokenization is and why it's necessary.
- Understand the different steps in the tokenization process.
- Learn what Byte Pair Encoding is and how it works.
- Be able to use tiktoken to encode and decode text.

2.1 Importance of Data Preparation

- **Data is the Foundation:** The performance and capabilities of an LLM are heavily dependent on the quality and nature of the data it's trained on.
- **Garbage In, Garbage Out:** If you train a model on poorly formatted, biased, or irrelevant data, you'll get a poorly performing, biased, or irrelevant model.
- **LLM Comprehension:** Data preparation, particularly **tokenization**, is crucial for how the LLM "understands" the input text. Tokenization allows the model to map words to numerical vectors, and the way you break down text into tokens directly affects what the model learns.
- **Key Considerations:**
    - **Data Source:** Where does the data come from? Is it representative of the kind of text you want the model to generate? (e.g. WebText for GPT-2, a collection of diverse web pages)
    - **Data Cleaning:** Removing noise, errors, inconsistencies, and irrelevant parts (e.g., HTML tags, special characters).
    - **Data Formatting:** Structuring the data in a way that the model can process (e.g., each training example might be a sequence of text of a certain length).
    - **Dataset Size:** Larger datasets generally lead to better performance, but there are diminishing returns.
    - **Data Bias:** Is the data biased in any way (e.g., gender, race, ideology)? This can lead to a biased model.

2.2 Tokenization and Its Role

- **What is Tokenization?**
    - Tokenization is the process of **splitting text into individual units called tokens**.
    - Tokens can be words, subwords, characters, or even punctuation marks, depending on the tokenization method.
    - Think of it like breaking down a sentence into its building blocks.
- **Why is it Necessary?**
    - **Numerical Representation:** Neural networks work with numbers, not raw text. Tokenization is the first step in converting text into a numerical representation that the model can process.
    - **Vocabulary:** Tokenization defines the **vocabulary** of the model – the set of all unique tokens that the model recognizes.
    - **Input to the Model:** The sequence of tokens (represented as numerical IDs) becomes the input to the LLM.
- **Example:**
    - Sentence: "The quick brown fox jumps over the lazy dog."
    - Tokenization (word-level): ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]

2.3 Tokenization Process Detailed

- **Steps Involved:**
  1. **Text Cleaning (Preprocessing):**
     - This step is often done *before* the core tokenization process.
     - It can involve:
       - Removing HTML tags or other markup.
       - Handling special characters or encoding issues.
       - Lowercasing text (optional, depends on the model and task).
       - Removing irrelevant sections (e.g., headers, footers).

2. **Text Splitting:**
   - The core of tokenization.
   - Text is split into tokens based on a set of rules or an algorithm.
   - Common approaches:
     - **Whitespace Splitting:** The simplest method, splitting on spaces and punctuation. * **Rule-based Splitting:** Using predefined rules to handle special cases (e.g., contractions like "don't").
     - **Subword Tokenization (e.g., Byte Pair Encoding, WordPiece):** More advanced algorithms that split words into smaller units (subwords). We'll focus on Byte Pair Encoding (BPE) later.

3. **Token ID Conversion:**
    - Each unique token in the vocabulary is assigned a unique numerical ID.
    - A **vocabulary file** (or a mapping) stores the correspondence between tokens and their IDs.
    - Example:
        - "The": 0
        - "quick": 1
        - "brown": 2
        - ...and so on...
4. **Embeddings Conversion (Conceptual Overview):**
    - This step happens *inside* the model, but it's important to understand the connection.
    - Token IDs are used to look up corresponding **word embeddings** in an embedding layer.
    - Embeddings are dense vector representations of words that capture semantic meaning.

- **Example (Illustrative):**

```
Text: "Hello, world! This is a sentence."

1. Text Cleaning (e.g., lowercase, remove punctuation):
   "hello world this is a sentence"

2. Text Splitting (e.g., whitespace splitting):
   ["hello", "world", "this", "is", "a", "sentence"]

3. Token ID Conversion (using a hypothetical vocabulary):
   [10, 25, 42, 18, 5, 67]

4. Embeddings Conversion (inside the model):
   The model looks up the embedding vectors for each of these IDs in its embedding
layer.
```

2.4 Byte Pair Encoding (BPE)

- **Introduction to BPE:**
    - BPE is a **subword tokenization algorithm** that is widely used in modern LLMs, including GPT-2.
    - It's a good balance between the granularity of character-level tokenization (which can result in very long sequences) and the limitations of word-level tokenization (which struggles with out-of-vocabulary words).
    - **Key Idea:** BPE iteratively merges the most frequent pair of consecutive tokens into a new, single token.
- **Handling Unknown Words:**
    - BPE can handle words that were not seen during training (out-of-vocabulary or OOV words) by breaking them down into subword units that are present in the vocabulary.
    - For example, if the word "aaabdabc" was not seen during training, but we have "aaab" and "dabc" in the vocabulary from above steps, we can tokenize it as two tokens.

2.5 Hands-on Exercise 1: Data Preparation and Tokenization

- **Activity:** We'll use the `tiktoken` library, which provides a fast implementation of the BPE tokenizer used by GPT-2.
- **Task:**
    1. Take a sample text.
    2. Tokenize the text using the GPT-2 tokenizer from `tiktoken`.
    3. Inspect the generated tokens and their corresponding IDs.
- **Guidance:**
    - Observe how the text is split into tokens.
    - Notice how punctuation is handled.
    - See how subwords are used for words that might not be in the vocabulary (like "blargh" or "pলাপ").
    - Understand the relationship between tokens and their IDs.

Key Takeaways:

- Students should be able to use the `tiktoken` library to tokenize text using the GPT-2 tokenizer.

- They should understand how text is converted into a sequence of token IDs.
- They should observe how BPE handles subwords and potentially unknown words.

In [ ]:

```python
import tiktoken

# Get the GPT-2 tokenizer
tokenizer = tiktoken.get_encoding("gpt2")

# Sample text
text = "This is a sample text for demonstrating GPT-2 tokenization. This includes some

# Encode (tokenize) the text
encoded_text = tokenizer.encode(text)

# Decode (detokenize) the tokens back into text
decoded_text = tokenizer.decode(encoded_text)

# Print the encoded text (token IDs)
print("Encoded text (token IDs):", encoded_text)

# Print the decoded text
print("Decoded text:", decoded_text)

# Print each token and its ID
print("Tokens and their IDs:")
for token_id in encoded_text:
    print(f"Token: '{tokenizer.decode([token_id])}', ID: {token_id}")

# Test with an out of vocabulary word
oov_text = "This is a text with an out of vocabulary word: blarghxyz"
encoded_oov_text = tokenizer.encode(oov_text)
print(f"Encoded OOV text: {encoded_oov_text}")
print(f"Decoded OOV text: {tokenizer.decode(encoded_oov_text)}")
for token_id in encoded_oov_text:
    print(f"Token: '{tokenizer.decode([token_id])}', ID: {token_id}")
```

Part 3: Training GPT-2

Learning Objectives:

- Understand the architecture of GPT-2
- Learn how to implement a GPT-2 model from scratch
- Understand what pre-training is and why it's important
- Learn the core concepts of training
- Run a simplified GPT-2 training example

3.1 GPT-2 Architecture Overview

- **Recap:** GPT-2 is a **decoder-only Transformer model**. It uses the decoder part of the original Transformer architecture (from "Attention is All You Need") for text generation.
- **Components:**
    - **Token Embeddings:**
        - Converts input tokens (represented as numerical IDs) into **dense vector representations** called embeddings.
        - These embeddings capture semantic meaning and relationships between words.
        - The embedding layer is essentially a large lookup table where each row corresponds to the embedding vector for a specific token ID in the vocabulary.
    - **Positional Encodings:**
        - Since Transformers don't inherently process sequential information, positional encodings are added to the token embeddings to provide information about the **position of each token in the sequence**.
        - These encodings can be learned or fixed (e.g., using sine and cosine functions, as in the original Transformer paper).

- **Transformer Blocks (Decoder Layers):**
  - GPT-2 has multiple stacked Transformer blocks (or decoder layers). The number of blocks varies depending on the model size (e.g., 12 blocks in GPT-2 small, 48 in GPT-2 XL).
  - Each block consists of:
    - **Masked Self-Attention:** Allows the model to attend to the **preceding tokens** in the sequence when predicting the next token. The "masking" prevents the model from "seeing" future tokens during training.
    - **Feed-Forward Network:** A two-layer fully connected network that applies a non-linear transformation to the output of the attention layer.
      - **Output Layer:**
  - A linear layer that projects the output of the final Transformer block to the size of the vocabulary.
  - A **softmax function** is applied to produce a probability distribution over the vocabulary, where each element represents the probability of the corresponding token being the next word in the sequence.

Figure 7. GPT-2 Architecture

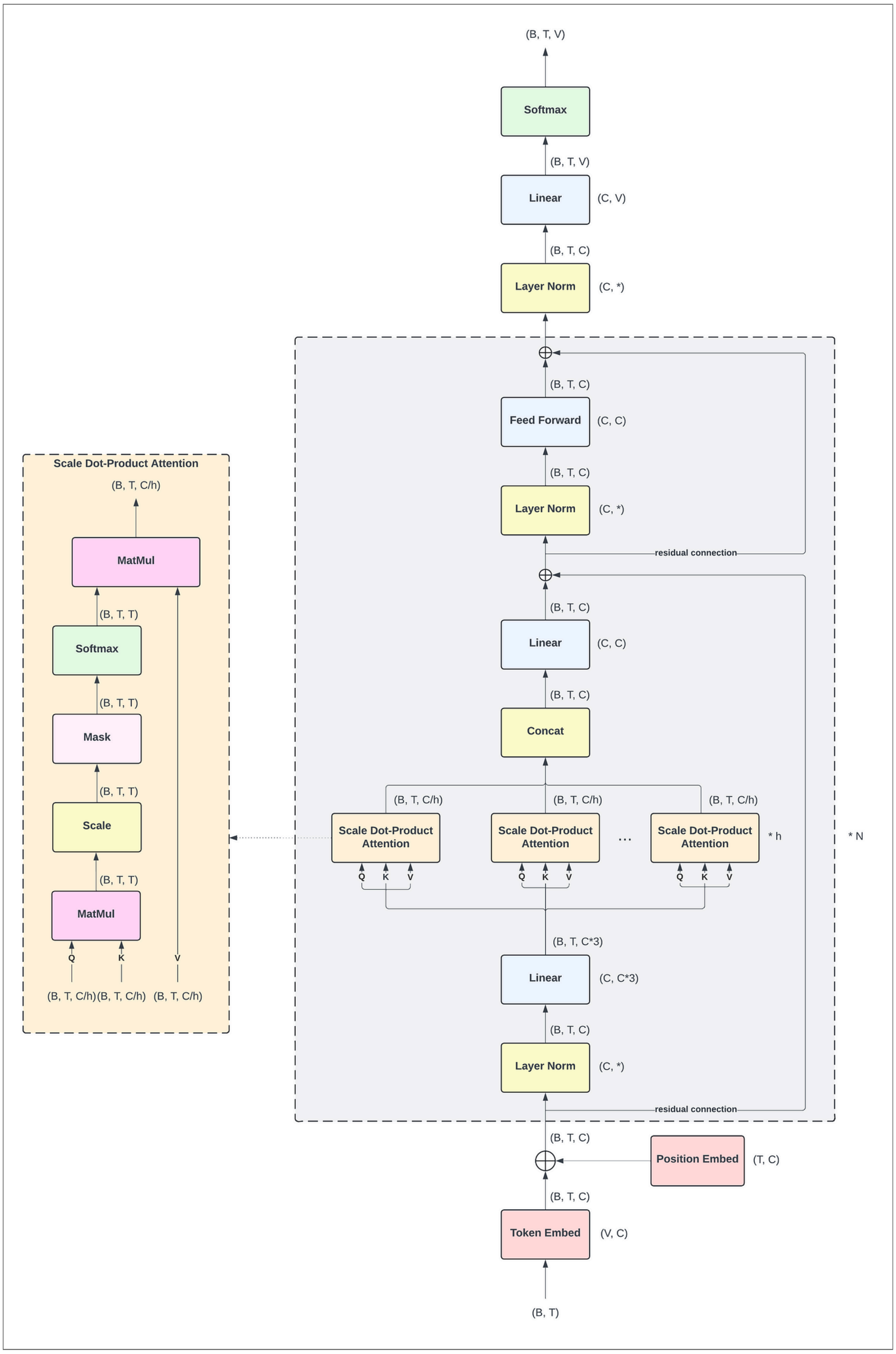
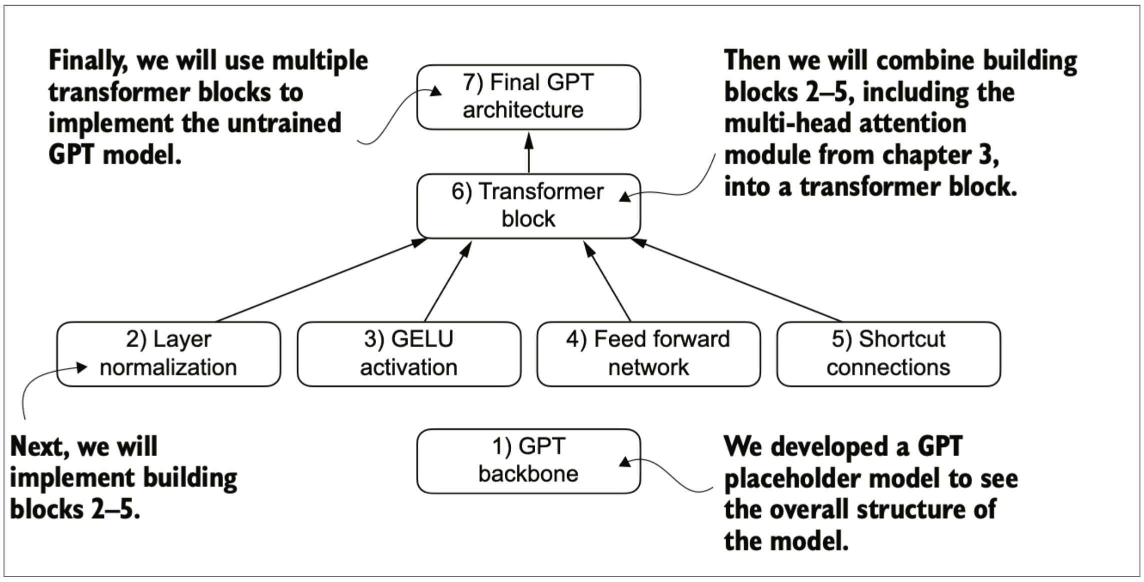


Figure 8. Overview of GPT-2 Model Code


Finally, we will use multiple transformer blocks to implement the untrained GPT model.
7) Final GPT architecture
Then we will combine building blocks 2–5, including the multi-head attention module from chapter 3, into a transformer block.
6) Transformer block
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
Next, we will implement building blocks 2–5.
1) GPT backbone
We developed a GPT placeholder model to see the overall structure of the model.

3.2 Implementing the GPT-2 Model from Scratch

- **Skeleton Code:**

We'll start with a basic framework for the GPT-2 class in PyTorch. The goal of this code is to show you how the different parts of a Transformer model work by implementing them from scratch. Don't worry if you don't understand all of it right away! We will go through it step by step.

```
import torch
import torch.nn as nn
import math

class GPT2(nn.Module):
    def __init__(self, vocab_size, d_model, nhead, num_layers, dim_feedforward, dropout=0.1,
layer_norm_epsilon=1e-5):
        super().__init__()

        # 1. Token Embedding Layer
        self.token_embedding = nn.Embedding(vocab_size, d_model)

        # 2. Positional Encoding Layer (using fixed sinusoidal encodings)
        self.positional_encoding = PositionalEncoding(d_model, dropout)

        # 3. Transformer Blocks (Decoder Layers)
        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(d_model, nhead, dim_feedforward, dropout, layer_norm_epsilon) for _ in
range(num_layers)
        ])

        # 4. Output Layer
        self.output_layer = nn.Linear(d_model, vocab_size)

        # Weight tying (optional but often beneficial)
        self.output_layer.weight = self.token_embedding.weight

        self.d_model = d_model
```

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward, dropout, layer_norm_epsilon):
        super().__init__()

        # Multi-head attention with efficient implementations
        self.self_attn = nn.MultiheadAttention(d_model, nhead, dropout=dropout, batch_first=True)

        # Feed-forward network
        self.ffn = nn.Sequential(
            nn.Linear(d_model, dim_feedforward),
            nn.GELU(),
            nn.Linear(dim_feedforward, d_model),
            nn.Dropout(dropout)
        )

        # Layer normalization
        self.norm1 = nn.LayerNorm(d_model, eps=layer_norm_epsilon)
        self.norm2 = nn.LayerNorm(d_model, eps=layer_norm_epsilon)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, attention_mask):
        # Multi-head attention + residual connection
        attn_output, _ = self.self_attn(x, x, x, attn_mask=attention_mask, need_weights=False)
        x = self.norm1(x + self.dropout(attn_output))

        # Feed-forward network + residual connection
        ffn_output = self.ffn(x)
        x = self.norm2(x + self.dropout(ffn_output))
```

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:x.size(1)].transpose(0, 1)
        return self.dropout(x)
```

3.3 Hands-on Exercise 2: Implementing the GPT-2 Model

- **Activity:** Students will understand how the code works and implement the Transformer model.
- **Task:**
    1. Understand the code for `TransformerBlock` and `GPT2`.
    2. Explain what each line of code does.
- **Guidance:**
    - Refer to the explanations and code snippets in section 3.2.
    - Use the PyTorch documentation for `nn.MultiheadAttention`, `nn.Sequential`, `nn.Linear`, `nn.LayerNorm`, and other relevant modules.
    - Pay attention to the order of operations (attention, residual connection, layer normalization, feed-forward network, etc.).
    - Make sure the dimensions of the tensors are correct at each step.

In [ ]:

```
## Basic code for GPT-2

import torch
import torch.nn as nn
import math


class GPT2(nn.Module):
    def __init__(
        self,
        vocab_size,
        d_model,
        nhead,
        num_layers,
        dim_feedforward,
        dropout=0.1,
        layer_norm_epsilon=1e-5,
    ):
        super().__init__()

        # 1. Token Embedding Layer
        self.token_embedding = nn.Embedding(vocab_size, d_model)

        # 2. Positional Encoding Layer (using fixed sinusoidal encodings)
        self.positional_encoding = PositionalEncoding(d_model, dropout)

        # 3. Transformer Blocks (Decoder Layers)
        self.transformer_blocks = nn.ModuleList(
            [
                TransformerBlock(
                    d_model, nhead, dim_feedforward, dropout, layer_norm_epsilon
                )
                for _ in range(num_layers)
            ]
        )

        # 4. Output Layer
        self.output_layer = nn.Linear(d_model, vocab_size)

        # Weight tying (optional but often beneficial)
        self.output_layer.weight = self.token_embedding.weight

        self.d_model = d_model

    def forward(self, input_ids, attention_mask=None):
        # 1. Get token embeddings
        x = self.token_embedding(input_ids) * math.sqrt(
            self.d_model
        )  # Scale embeddings

        # 2. Add positional encodings
        x = self.positional_encoding(x)

        # 3. Create causal attention mask (to prevent attending to future tokens)
        if attention_mask is None:
            seq_len = input_ids.size(1)
            attention_mask = torch.triu(
                torch.ones(seq_len, seq_len, device=x.device, dtype=torch.bool),
                diagonal=1,
            )

        # Invert the mask for compatibility with PyTorch's masking convention
        attention_mask = attention_mask.logical_not()
```

```python
        # 4. Pass through Transformer blocks
        for block in self.transformer_blocks:
            x = block(x, attention_mask)

        # 5. Calculate output logits
        logits = self.output_layer(x)

        return logits


class TransformerBlock(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward, dropout, layer_norm_epsilon):
        super().__init__()

        # Multi-head attention with efficient implementations
        self.self_attn = nn.MultiheadAttention(
            d_model, nhead, dropout=dropout, batch_first=True
        )

        # Feed-forward network
        self.ffn = nn.Sequential(
            nn.Linear(d_model, dim_feedforward),
            nn.GELU(),
            nn.Linear(dim_feedforward, d_model),
            nn.Dropout(dropout),
        )

        # Layer normalization
        self.norm1 = nn.LayerNorm(d_model, eps=layer_norm_epsilon)
        self.norm2 = nn.LayerNorm(d_model, eps=layer_norm_epsilon)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, attention_mask):
        # Multi-head attention + residual connection
        attn_output, _ = self.self_attn(
            x, x, x, attn_mask=attention_mask, need_weights=False
        )
        x = self.norm1(x + self.dropout(attn_output))

        # Feed-forward network + residual connection
        ffn_output = self.ffn(x)
        x = self.norm2(x + self.dropout(ffn_output))

        return x


class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model)
        )
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + self.pe[: x.size(1)].transpose(0, 1)
        return self.dropout(x)


# Example Usage:
vocab_size = 50257  # GPT-2 vocabulary size
d_model = 768  # Hidden size (embedding dimension)
```

```python
nhead = 12  # Number of attention heads
num_layers = 12  # Number of transformer layers
dim_feedforward = 3072  # Feedforward network dimension
batch_size = 8
seq_length = 1024

model = GPT2(vocab_size, d_model, nhead, num_layers, dim_feedforward)
input_ids = torch.randint(0, vocab_size, (batch_size, seq_length))

# Generate output
output = model(input_ids)
print(output.shape)  # Should be (batch_size, seq_length, vocab_size)
```

3.4 Hands-on Exercise 3: Pre-Training GPT-2

- **Concept and Importance:**
    - **Pre-training** is the process of training a language model on a **massive dataset of general text** to learn the fundamental patterns and structures of language.
    - It's like giving the model a broad education in language before teaching it more specific tasks.
    - **Why is it important?**
        - **Learns General Language Understanding:** The model acquires a rich internal representation of language, including syntax, semantics, and even some world knowledge.
        - **Data Efficiency for Downstream Tasks:** Pre-trained models can be fine-tuned for specific tasks with much less data than training from scratch.
        - **Improved Performance:** Fine-tuned models often achieve better performance on downstream tasks compared to models trained from scratch.
    - **Analogy:** Think of pre-training as learning the basic rules of grammar and vocabulary, and fine-tuning as learning how to write a specific type of essay or answer a specific type of question.

- **Training Objective:**
    - The primary training objective in pre-training GPT-2 is **next-word prediction** (also called **language modeling** or **causal language modeling**).
    - The model is given a sequence of words and is trained to predict the probability distribution of the next word in the sequence.
    - **Example:**
        - Input: "The quick brown fox"
        - Target: "jumps"
        - The model is trained to assign a high probability to "jumps" given the preceding words.
- **Loss Function and Optimization:**
    - **Loss Function:** The standard loss function used is **cross-entropy loss**. It measures the difference between the model's predicted probability distribution over the vocabulary and the actual distribution (where the target word has a probability of 1 and all other words have a probability of 0).
    - **Optimization:** An optimizer (like AdamW) is used to update the model's parameters to minimize the loss function. The gradients are calculated using backpropagation.

In [ ]:

```
## Pre-Training GPT-2

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import math
from tqdm import tqdm  # For a nice progress bar
import tiktoken

#class GPT2(nn.Module):
    # ... (Refer to previous code)

#class TransformerBlock(nn.Module):
    # ... (Refer to previous code)

#class PositionalEncoding(nn.Module):
    # ... (Refer to previous code)

# --- Data Preparation (using a dummy dataset for demonstration) ---

class TextDataset(Dataset):
    def __init__(self, text_data, sequence_length, tokenizer=None):
        self.sequence_length = sequence_length
        if tokenizer is None:
            self.tokenizer = tiktoken.get_encoding("gpt2")

        # Tokenize the entire text at once
        self.tokens = self.tokenizer.encode(text_data)

    def __len__(self):
        return max(0, len(self.tokens) - self.sequence_length - 1)

    def __getitem__(self, idx):
        # Get sequence and ensure it's exactly sequence_length
        input_sequence = self.tokens[idx:idx + self.sequence_length]
        target_sequence = self.tokens[idx + 1:idx + self.sequence_length + 1]

        # Pad if necessary (shouldn't be needed if data is long enough)
        if len(input_sequence) < self.sequence_length:
            padding = [0] * (self.sequence_length - len(input_sequence))
            input_sequence = input_sequence + padding
            target_sequence = target_sequence + padding

        # Truncate if somehow longer
        input_sequence = input_sequence[:self.sequence_length]
        target_sequence = target_sequence[:self.sequence_length]

        return (
            torch.tensor(input_sequence, dtype=torch.long),
            torch.tensor(target_sequence, dtype=torch.long)
        )

# --- Hyperparameters and Configuration ---

vocab_size = 50257  # GPT-2 vocabulary size
d_model = 768  # Hidden size (embedding dimension)
nhead = 12  # Number of attention heads
num_layers = 6 # Number of transformer layers
dim_feedforward = 3072  # Feedforward network dimension
dropout = 0.1
batch_size = 8
sequence_length = 128
learning_rate = 5e-5
```

```
num_epochs = 2 # You can increase this for better results if you have the resources
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# --- Create the Model, Optimizer, and Loss Function ---

model = GPT2(vocab_size, d_model, nhead, num_layers, dim_feedforward, dropout).to(devic
optimizer = optim.AdamW(model.parameters(), lr=learning_rate, weight_decay=0.01)  # Us
criterion = nn.CrossEntropyLoss()

# --- Learning Rate Scheduler (example with linear decay) ---

# We'll use a dummy dataset for this example. Replace with your actual dataset.
text_data = "This is a very long text used as a dummy example for training a GPT-2 mode
# If you have a real dataset, you can uncomment and modify the following lines:
# with open("your_dataset.txt", "r", encoding="utf-8") as f:
#     text_data = f.read()

num_training_steps = num_epochs * (len(text_data) - sequence_length) // batch_size  #
lr_scheduler = optim.lr_scheduler.LinearLR(optimizer, start_factor=1.0, end_factor=0.1

dataset = TextDataset(text_data, sequence_length)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# --- Training Loop ---

model.train()
for epoch in range(num_epochs):
    for batch_idx, (input_ids, target_ids) in enumerate(tqdm(dataloader, desc=f"Epoch
        input_ids = input_ids.to(device)
        target_ids = target_ids.to(device)

        optimizer.zero_grad()

        # Forward pass
        outputs = model(input_ids)

        # Calculate loss
        loss = criterion(outputs.view(-1, vocab_size), target_ids.view(-1))

        # Backward pass
        loss.backward()

        # Clip gradients (optional but often helpful for Transformers)
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        # Update weights
        optimizer.step()

        # Update learning rate
        lr_scheduler.step()

        if batch_idx % 10 == 0:
            print(f"Epoch: {epoch+1}/{num_epochs}, Batch: {batch_idx}/{len(dataloader)

print("Training finished!")

# --- Save the trained model ---
# This saves the trained model's state_dict, which contains the model's learned parame
# You can load the state_dict later to resume training or use the model for inference.
torch.save(model.state_dict(), "trained_gpt2_model.pth")
print("Model saved to trained_gpt2_model.pth")

# --- Text Completion Demonstration ---
# Now let's use the trained model to complete some text.
model.eval()  # Set the model to evaluation mode

tokenizer = tiktoken.get_encoding("gpt2")
prompt = "The quick brown fox"
```

```
input_ids = tokenizer.encode(prompt)
input_ids = torch.tensor([input_ids], dtype=torch.long).to(device)

with torch.no_grad():  # No need to track gradients during inference
    for _ in range(50):  # Generate 50 tokens
        outputs = model(input_ids)
        predicted_token_id = outputs[0, -1, :].argmax().item()  # Get the ID of the mos
        input_ids = torch.cat((input_ids, torch.tensor([[predicted_token_id]], dtype=to
        if predicted_token_id == tokenizer.eot_token:
            break

generated_text = tokenizer.decode(input_ids[0].tolist())  # Decode the generated token
print("Generated text:")
print(generated_text)
```

Part 4: Post-Processing: Fine-Tuning

Learning Objectives:

- Understand the benefits of using pre-trained weights.
- Be able to load pre-trained models and weights.
- Understand fine-tuning and instruction fine-tuning
- Know of existing Instruction Fine-tuning datasets.
- Be able to fine-tune a pre-trained LLM

4.1 Loading Pre-trained Weights

- **Benefits:**
    - **Saves Time and Resources:** You don't have to pre-train the model from scratch, which can take days, weeks, or even months, and requires a lot of computational power.
    - **Better Performance:** Pre-trained weights have already learned a good representation of language, so starting from these weights usually leads to better fine-tuning results.
    - **Accessibility:** Organizations like OpenAI and Hugging Face have made pre-trained weights for various models (including GPT-2) publicly available.
- **Implementation (using Hugging Face `transformers`):**
    - The Hugging Face `transformers` library is the easiest and most common way to load pre-trained GPT-2 weights.
- **Explanation:**
    - `from_pretrained(model_name)` automatically downloads and loads the pre-trained weights and configuration for the specified model name (e.g., "gpt2", "gpt2-medium", etc.).
    - `GPT2LMHeadModel` is the Hugging Face class for GPT-2 used for language modeling (with a language modeling head on top).
    - `GPT2Tokenizer` is the corresponding tokenizer.
- **OPTIONAL Loading into our Custom GPT-2:**
    - You can also load the state dict into our own GPT-2 class.

In [ ]:

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load pre-trained model and tokenizer
model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)

# Set padding token
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = model.config.eos_token_id

# Example inference with proper attention mask
input_text = "The quick brown fox"
inputs = tokenizer(input_text,
                   return_tensors="pt",
                   padding=True,
                   truncation=True,
                   max_length=100)
```

```
output = model.generate(
    input_ids=inputs.input_ids,
    attention_mask=inputs.attention_mask,
    max_length=100,
    pad_token_id=tokenizer.pad_token_id,
    num_return_sequences=1
)

decoded_output = tokenizer.decode(output[0], skip_special_tokens=True)
print(decoded_output)
```

4.2 Fine-tuning for Specific Tasks

- **Concept of Fine-tuning:**
    - **Fine-tuning** is the process of taking a **pre-trained model** and **further training it on a smaller dataset** that is specific to a particular downstream task.
    - The pre-trained weights are used as a starting point, and the model's parameters are adjusted to better perform the new task.
    - **Example Tasks:**
        - **Text Classification:** Classifying text into different categories (e.g., sentiment analysis, topic classification).
        - **Question Answering:** Answering questions based on a given context passage.
        - **Text Summarization:** Generating summaries of longer text documents.
        - **Translation:** (Though less common with decoder-only models like GPT-2, you could potentially fine-tune for a specific language pair).
        - **Instruction Following:** Training the model to follow specific instructions (more on this below).

- **Process:**
    1. **Load Pre-trained Weights:** Start with a pre-trained GPT-2 model (as described in section 4.3).
    2. **Prepare Task-Specific Data:** Create a dataset for your specific task. This dataset will usually be much smaller than the pre-training dataset.
    3. **Add a Task-Specific Output Layer (Optional):**
        - For some tasks (like classification), you might need to add a new output layer on top of the pre-trained model. For example, for binary classification, you could add a linear layer that outputs a single number, followed by a sigmoid activation function to produce a probability between 0 and 1.
        - For text generation tasks, the existing language modeling head (which produces a probability distribution over the vocabulary) might be sufficient.
    4. **Modify Training Objective (if needed):**
        - For example, if you add a classification head, you would use a classification loss (like binary cross-entropy).
        - For text generation
    5. **Train the Model:**
        - Train the model on your task-specific dataset, updating the weights of the entire model (or sometimes just the new output layer and a few of the top layers of the pre-trained model).
        - Use a lower learning rate than in pre-training to avoid drastic changes to the already learned representations.
        - Train for fewer epochs than in pre-training.

- **Instruction Fine-tuning**
  - This is a popular and effective way to fine-tune models to perform a wide range of tasks, by training it to follow instructions.
  - **Idea:** You provide the model with an **instruction** that describes the task you want it to perform, along with the input, and train it to generate the correct output that follows the instruction.
  - **Example:**

```
Instruction: Translate the following English sentence to French.
Input: The cat sat on the mat.
Output: Le chat s'est assis sur le tapis.

Instruction: Summarize the following article.
Input: <Article text>
Output: <Summary of the article>

Instruction: Answer the following question based on the context provided.
Context: <Context text>
Question: What is the capital of France?
Output: Paris
```

  - **Benefits:**
    - **Flexibility:** You can train a single model to perform many different tasks by simply changing the instruction.
    - **Generalization:** Models trained with instructions often generalize better to new, unseen instructions.
    - **Data Efficiency:** Can be more data-efficient than training separate models for each task.

4.3 Hands-on Exercise 4: Post-training Work

- **Activity:** We'll fine-tune a pre-trained GPT-2 model for a simple instruction-following task. We will be using the Hugging Face `transformers` library to simplify the process.
- **Task:** Fine-tune GPT-2 to convert sentences to passive voice based on an instruction.
- **Dataset:** We'll create a very small, illustrative dataset for this example. In a real-world scenario, you would use a much larger dataset.
- **Steps:**
    1. **Load Pre-trained Model and Tokenizer:** We load the pre-trained GPT-2 model and tokenizer from Hugging Face `transformers`.
    2. **Prepare the Dataset:** We create a simple dataset with a few examples of active-to-passive voice conversion instructions.
    3. **Tokenize the Data:** We format our instruction, input and output text into a single string, and use `TextDataset` with `DataCollatorForLanguageModeling` to prepare the data for the `Trainer`.
    4. **Training Arguments:** We set up the training arguments, including the output directory, number of epochs, batch size, etc.
    5. **Trainer:** We use the Hugging Face `Trainer` class, which simplifies the training process. We pass the model, training arguments, data collator, and training dataset to the `Trainer`.
    6. **Fine-tune:** `trainer.train()` starts the fine-tuning process.
    7. **Save:** We save the fine-tuned model using `trainer.save_model()`.
    8. **Inference:** We test the model by giving it a new instruction and input, and generating the output.
- **Guidance:**
    - Run the code and observe the output.
    - Experiment with different instructions and inputs.
    - Try adding more examples to the `instruction_data` to improve the model's performance.

- You can adjust the `max_length` parameter in `model.generate()` to control the length of the generated text.

In [ ]:

```
# Simple instruction fine-tuning

from transformers import GPT2LMHeadModel, GPT2Tokenizer, DataCollatorForLanguageModeli
from transformers import Trainer, TrainingArguments
from torch.utils.data import Dataset
import torch

# Check if MPS is available
device = torch.device("mps") if torch.backends.mps.is_available() else torch.device("c
print(f"Using device: {device}")

# Create a custom dataset class
class InstructionDataset(Dataset):
    def __init__(self, texts, tokenizer, max_length):
        self.encodings = tokenizer(
            texts,
            truncation=True,
            padding=True,
            max_length=max_length,
            return_tensors="pt"
        )

    def __getitem__(self, idx):
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['labels'] = item['input_ids'].clone()
        return item

    def __len__(self):
        return len(self.encodings.input_ids)

# Load pre-trained model and tokenizer
model_name = "gpt2"
model = GPT2LMHeadModel.from_pretrained(model_name)
tokenizer = GPT2Tokenizer.from_pretrained(model_name)

# Add padding token
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})
    model.resize_token_embeddings(len(tokenizer))

# Move model to MPS device
model = model.to(device)

# Prepare the dataset
instruction_data = [
    {
        "instruction": "Convert the following sentence to passive voice.",
        "input": "The cat chased the mouse.",
        "output": "The mouse was chased by the cat."
    },
    {
        "instruction": "Convert the following sentence to passive voice.",
        "input": "John ate the apple.",
        "output": "The apple was eaten by John."
    },
    {
        "instruction": "Convert the following sentence to passive voice.",
        "input": "The teacher graded the tests.",
        "output": "The tests were graded by the teacher."
    },
    {
        "instruction": "Convert the following sentence to passive voice.",
        "input": "The storm damaged the house.",
        "output": "The house was damaged by the storm."
```

```python
    },
    {
        "instruction": "Convert the following sentence to passive voice.",
        "input": "The company hired new employees.",
        "output": "New employees were hired by the company."
    },
    {
        "instruction": "Convert the following sentence to passive voice.",
        "input": "Students submitted their assignments.",
        "output": "The assignments were submitted by the students."
    }
]

# Format the texts
train_texts = [
    f"### Instruction: {data['instruction']}\n### Input: {data['input']}\n### Output: 
    for data in instruction_data
]

# Create dataset using custom class
train_dataset = InstructionDataset(
    texts=train_texts,
    tokenizer=tokenizer,
    max_length=128
)

# Set up training arguments
training_args = TrainingArguments(
    output_dir="./gpt2_passive_voice",
    overwrite_output_dir=True,
    num_train_epochs=3,
    per_device_train_batch_size=2,
    save_steps=10_000,
    save_total_limit=2,
    logging_dir='./logs',
)

# Create trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
)

# Train
trainer.train()
trainer.save_model()

# Test the model
model.eval()
test_instruction = "Convert the following sentence to passive voice."
test_input = "The chef prepared the meal."
prompt = f"Instruction: {test_instruction}\nInput: {test_input}\nOutput:"

inputs = tokenizer(
    prompt,
    return_tensors="pt",
    padding=True,
    truncation=True
)

# Move inputs to MPS device
inputs = {k: v.to(device) for k, v in inputs.items()}

with torch.no_grad():
    output = model.generate(
        input_ids=inputs['input_ids'],
        attention_mask=inputs['attention_mask'],
        max_length=100,
        pad_token_id=tokenizer.pad_token_id,
    )

generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print("\nGenerated text:")
print(generated_text)
```

Part 5: Conclusion and Next Steps

5.1 Recap of Key Concepts

- **Large Language Models (LLMs):**
  - Powerful deep learning models trained on massive text data.
  - Capable of understanding and generating human-like text.
  - Examples: GPT-2, GPT-3, BERT, LaMDA, etc.
- **GPT-2:**
  - A decoder-only Transformer model.
  - Designed for text generation (next-word prediction).
  - Good for learning the fundamentals of LLMs due to its relatively simpler architecture.

- **Transformers:**
  - Revolutionized NLP with the "Attention is All You Need" paper.
  - Key components:
    - **Self-Attention:** Allows the model to weigh the importance of different words in a sequence.
    - **Multi-Head Attention:** Multiple attention mechanisms working in parallel.
    - **Positional Encoding:** Provides information about word order.
    - **Feed-Forward Networks:** Apply non-linear transformations.
    - **Layer Normalization:** Stabilizes training.
    - **Residual Connections:** Help with gradient flow.
  - **Advantages over RNNs:**
    - Parallelization (faster training).
    - Better at capturing long-range dependencies.

- **Data Preparation:**
  - Crucial for LLM performance.
  - **Tokenization:** Breaking down text into tokens (words, subwords, characters).
  - **Byte Pair Encoding (BPE):** A common subword tokenization algorithm used in GPT-2.
- **Building the GPT-2 Architecture:**
  - Implementing the model in PyTorch (or another deep learning framework).
  - Understanding the role of each component (token embeddings, positional encodings, Transformer blocks, output layer).
- **Pre-training:**
  - Training on a massive general text dataset to learn broad language understanding.
  - Objective: Next-word prediction.
  - Loss function: Cross-entropy.
- **Fine-tuning:**
  - Adapting a pre-trained model to a specific task.
  - Training on a smaller, task-specific dataset.
  - **Instruction Fine-tuning:** Training the model to follow instructions.
- **Hugging Face `transformers` Library:**
  - Provides pre-trained models, tokenizers, and training utilities.
  - Simplifies the process of working with LLMs.

5.2 Limitations of this lecture

- **Simplified Implementation:** We built a simplified version of GPT-2 for educational purposes. Real-world implementations are more complex and optimized.
- **Small Dataset:** We used a very small or dummy dataset for demonstration. Training state-of-the-art LLMs requires massive datasets.
- **Limited Compute:** We likely didn't have access to the computational resources (e.g., multiple high-end GPUs, TPUs) needed to train a large model from scratch.
- **Shallow Dive into Advanced Topics:** We touched upon advanced topics like model parallelism, mixed precision training, and optimized kernels, but didn't explore them in depth.
- **No Coverage of Evaluation Metrics:** We didn't cover metrics like perplexity, BLEU score, ROUGE, etc., which are essential for evaluating language models.
- **Ethical Considerations:** We only briefly mentioned the ethical implications of LLMs. A thorough discussion of bias, fairness, safety, and potential misuse is crucial when working with these powerful models.

Endnote

We welcome comments and corrections. Please feel free to contact us at leslie@aisingapore.org and william@aisingapore.org.

In [ ]:

```
dd
```