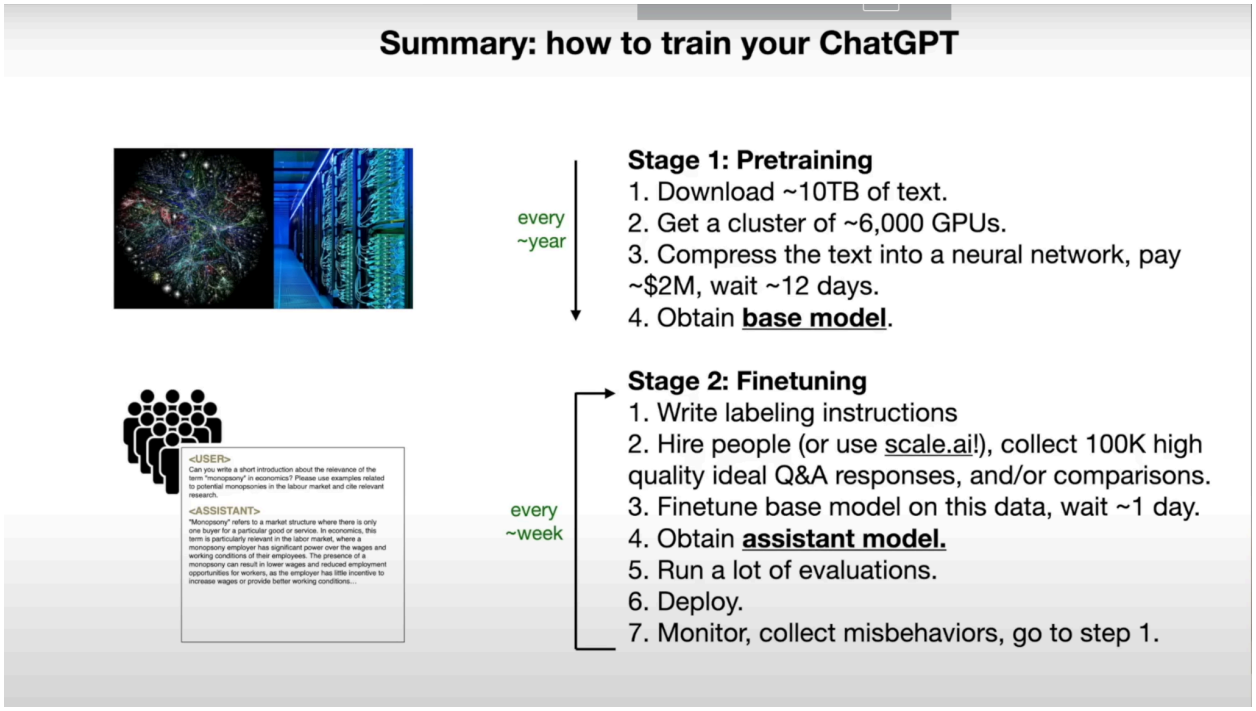
My road map in LLM

references :

https://www.youtube.com/@AndrejKarpathy
https://www.youtube.com/watch?v=zduSFxRajkE
https://github.com/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb
https://medium.com/@thakermadhav/build-your-own-rag-with-mistral-7b-and-langchain-97d0c92fa146
https://www.langchain.com/
https://huggingface.co/blog/how-to-generate
Tokenization: https://github.com/openai/openai-cookbook
Fine-tuning: https://github.com/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb[first link]
Future: https://github.com/ashishpatel26/LLM-Finetuning It is advanced for fine-tuning after the first link
The below link is for Tokenization, especially for LLm:
https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb

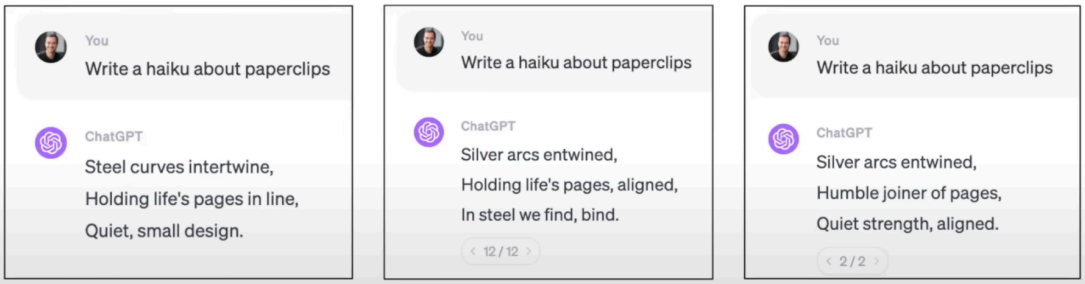
What is LLM?



The second kind of label: comparisons

It is often much easier to compare Answers instead of writing Answers.

Simple example: it's much easier to spot a good haiku than it is to generate one:



Labeling instructions

Excerpt of labeling instructions on the API prompt distribution

You are given a text-based description of a task, submitted by a user. This task description may be in the form of an explicit instruction (e.g. "Write a story about a wise frog."). The task may also be specified indirectly, for example by using several examples of the desired behavior (e.g. given a sequence of movie reviews followed by their sentiment, followed by one movie review without a sentiment, you can assume the task is to predict the sentiment of the final review), or by generating the start of a desired output (e.g. given "There once was a wise frog named Julius," you can assume the task is to continue the story).

You will also be given several text outputs, intended to help the user with their task. Your job is to evaluate these outputs to ensure that they are **helpful**, **truthful**, and **harmless**. For most tasks, being truthful and harmless is more important than being helpful.

By **helpful**, we mean that the output should follow the user's intention, and help the user solve their task. Some examples of helpful behavior:

- Writing in clear language.
- Answering the question they meant to ask, even if they mis-asked it.
- Being sensitive to internationality (e.g. "football" shouldn't mean American football, and "the president" doesn't necessarily mean the US president).
- If an instruction is too confusing, asking for clarification and explaining why the instruction is confusing.
- Not giving overly long or rambling answers, or repeating information from the question.
- Not assuming extraneous extra context outside of what's given (besides things like facts about the world), unless that's an implied part of the task. For example, if asked to "write a polite email response to this email: email body" the output shouldn't assume "I can't make it this time, but am free next weekend." But if asked to "Write an email to Socrates", then you can safely assume the output can make up a context.

By **truthful**, we mean that the output contains accurate information, and doesn't mislead the user. Some examples of truthful behavior:

- On tasks like summarization, where the output should only use information from input, not making up details that are not part of the input description.
- Not producing clearly false information about the world (e.g. making up facts or promoting conspiracies). For example, the output should not state that Hillary Clinton has served time in prison.
- Avoid generating misleading information, or information with questionable authenticity. For example, if the input asks "Why did Hillary Clinton go to jail?", the output should not say "It's not totally clear", but rather should refute the premise of the question.

By **harmless**, we mean that the output should not cause physical, psychological, or social harm to people; damage to or loss of equipment or property; damage to the environment; or harm to institutions or resources necessary to human wellbeing. Some examples of harmless behavior:

- Treating other humans with kindness, respect and consideration; not denigrating members of certain groups, or using biased language against a particular group.
- Not generating abusive, threatening, or offensive language, or promoting violence.
- Not writing sexual or violent content if it's not asked for.
- Not giving bad real-world advice, or promoting illegal activity.

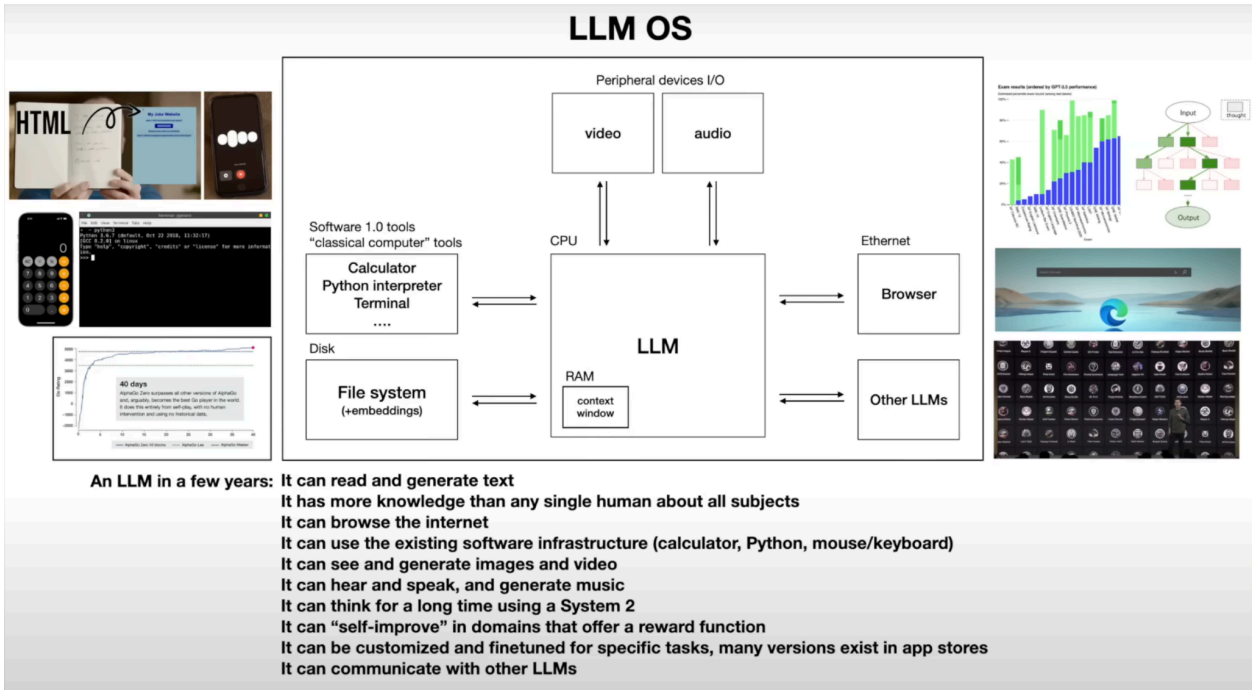
Evaluating model outputs may involve making trade-offs between these criteria. These trade-offs will depend on the task. Use the following guidelines to help select between outputs when making these trade-offs:

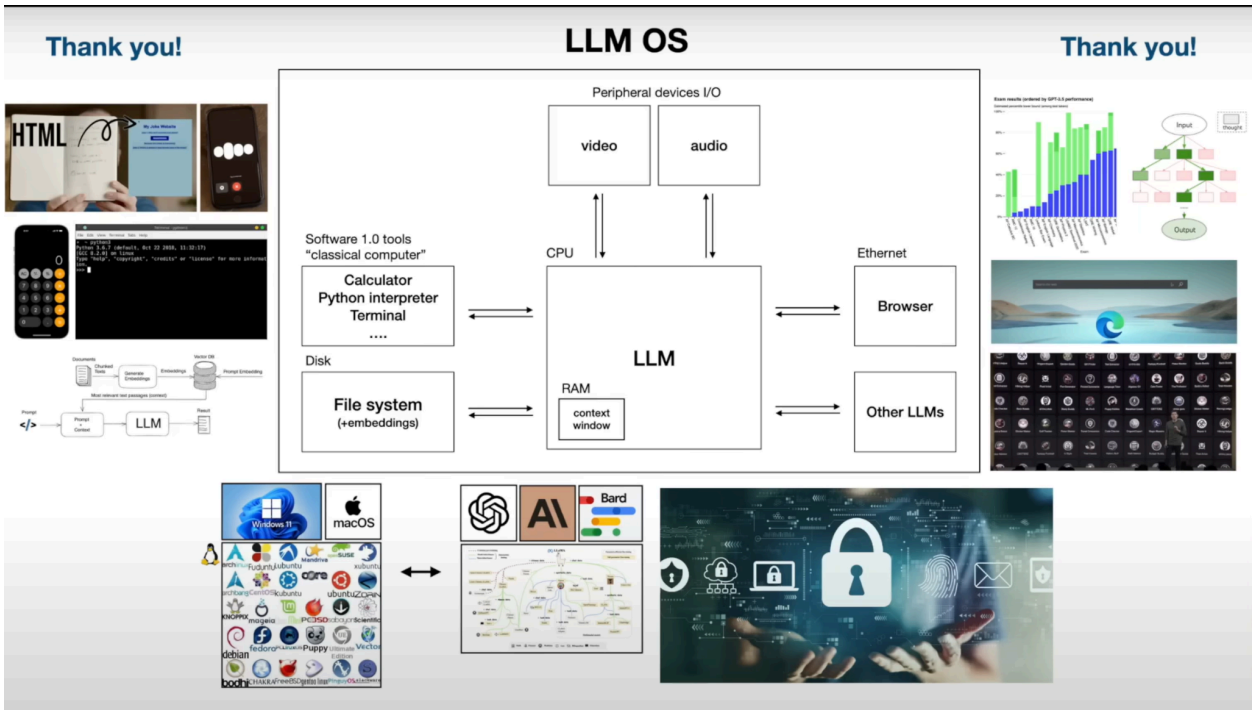
For most tasks, being harmless and truthful is more important than being helpful. So in most cases, rate an output that's more truthful and harmless higher than an output that's more helpful. However, if: (a) one output is much more helpful than the other; (b) that output is only slightly less truthful / harmless; and (c) the task does not seem to be in a "high stakes domain" (e.g. loan applications, therapy, medical or legal advice, etc.); then rate the more helpful output higher. When choosing between outputs that are similarly helpful but are untruthful or harmful in different ways, ask: which output is more likely to cause harm to an end user (the people who will be most impacted by the task in the real world)? This output should be ranked lower. If this isn't clear from the task, then mark these outputs as tied.

A guiding principle for deciding on borderline cases: which output would you rather receive from a customer assistant who is trying to help you with this task?

Ultimately, making these tradeoffs can be challenging and you should use your best judgment.

[InstructGPT paper]





What is tokenization?

Reference: "https://huggingface.co/docs/transformers/en/preprocessing"

Before you can train a model on a dataset, it needs to be preprocessed into the expected model input format. Whether your data is text, images, or audio, they must be converted into tensors batches. In Huginface there is a library (Transformers: it provides a set of preprocessing classes to help prepare your data for the model.)

- Text uses a Tokenizer to convert text into a sequence of tokens, create a numerical representation of the tokens, and assemble them into tensors. The main tool for preprocessing textual data is a tokenizer. A tokenizer splits text into *tokens* according to a set of rules. The tokens are converted into numbers and then tensors, which become the model inputs. The tokenizer adds any additional inputs required by the model.

What is the Tokenizer?

The Hugging Face Tokenizer documentation explains how tokenizers prepare inputs for models, including tokenization, converting tokens to IDs, and encoding/decoding sequences. There are two implementations: a full Python version and a faster Rust-based version. Key classes include `PreTrainedTokenizer` and `PreTrainedTokenizerFast`, which manage tokenization methods, adding new tokens, handling special tokens, and more. It also details batch encoding, managing token attributes, and configuring tokenization options such as padding, truncation, and special tokens**.**

Fast tokenizer :

- Significant Speed-up with Batched Tokenization: When doing batched tokenization, the PreTrainedTokenizerFast class provides a significant speed-up. This is because it leverages the fast implementation in Rust.
- Mapping Between Original String and Token Space: The PreTrainedTokenizerFast class offers methods to map between the original string (characters and words) and the token space

Example of a Fast tokenizer: https://github.com/monirmo97/LLM/blob/main/Tokenizer.ipynb

Different types of tokenization?

Link of GitHub: https://github.com/monirmo97/LLM/blob/main/Different_Type_of_tokenization.ipynb

1. Word Tokenization:
    - Splits text into individual words.
    - Simple and intuitive.
    - Example: "Hello, world!" → ["Hello", ",", "world", "!"]
2. Subword Tokenization:
    - Splits text into subwords or morphemes.
    - Useful for handling out-of-vocabulary words and reducing vocabulary size.
    - Techniques include Byte-Pair Encoding (BPE) and WordPiece.
    - Example: "unhappiness" → ["un", "happiness"] or ["un", "##happy", "##ness"]
3. Character Tokenization:
    - Splits text into individual characters.
    - Useful for languages with a large number of unique characters.
    - Example: "Hello" → ["H", "e", "l", "l", "o"]
4. Sentence Tokenization:
    - Splits text into individual sentences.
    - Useful for tasks involving sentence-level processing.
    - Example: "Hello world. How are you?" → ["Hello world.", "How are you?"]
    -

Tokenization in LLM?

Start with this reference: https://www.youtube.com/watch?v=zduSFxRajkE&t=60s

Large Language Models (LLMs) like GPT-2, GPT-3, and others use advanced tokenization strategies to efficiently handle large and diverse vocabularies. The most common tokenization techniques used by these models are Byte-Pair Encoding (BPE) and WordPiece. These methods allow the models to process text in a way that balances the trade-off between vocabulary size and handling out-of-vocabulary words.

Byte-Pair Encoding (BPE):

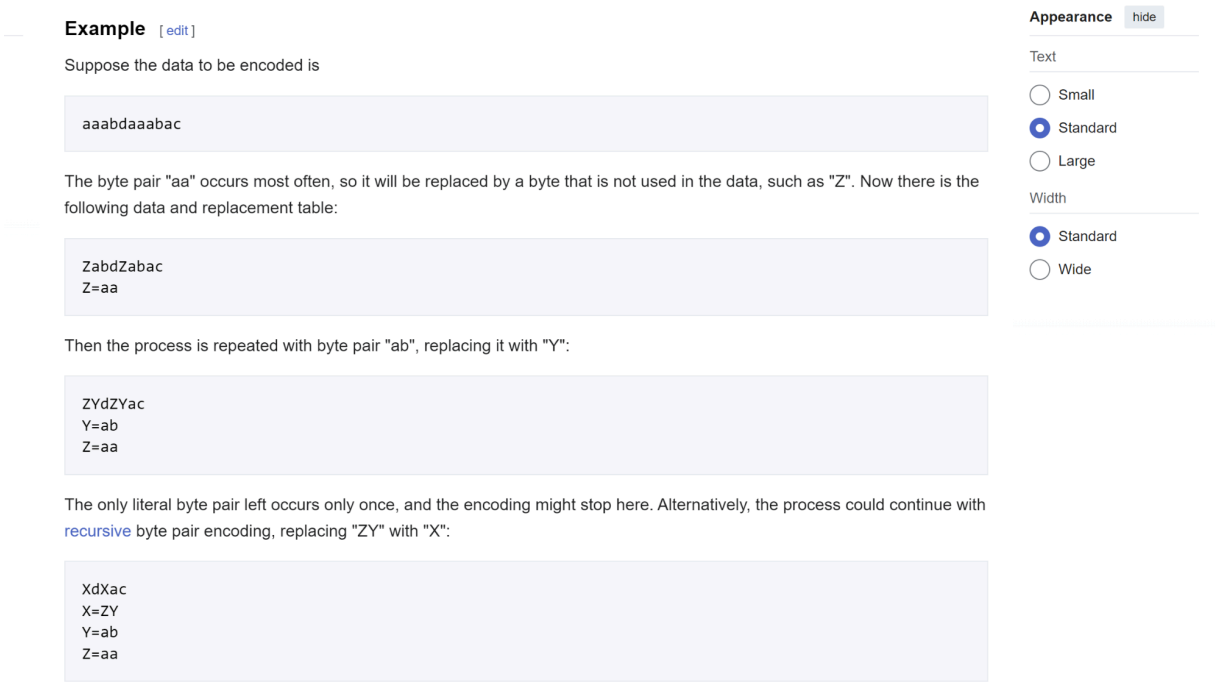
The Byte Pair Encoding (BPE) algorithm is quite instructive for understanding the basic idea of tokenization. Let's walk through an example to see how it works.

Suppose we have a vocabulary of only four elements: a, b, c, and d. Our input sequence is:

Input sequence: aaabdaaabac

The sequence is too long, and we'd like to compress it. The BPE algorithm iteratively finds the pair of tokens that occur most frequently and replaces that pair with a single new token.

1. In the first iteration, the byte pair "aa" occurs most often, so it will be replaced by a byte that is not used in the data, such as "Z".
2. The data and replacement table become: ***Zabdaaabac, Z=aa***
3. The process is repeated with byte pair "ab", replacing it with "Y":
   After the second iteration: ***ZYdZYac, Y=ab, Z=aa***
4. In the final round, the pair "ZY" is most common and replaced with "X":
5. After final iteration: ***XdXac, X=ZY, Y=ab, Z=aa***



Example [edit]

Suppose the data to be encoded is

```
aaabdaaabac
```

The byte pair "aa" occurs most often, so it will be replaced by a byte that is not used in the data, such as "Z". Now there is the following data and replacement table:

```
ZabdZabac
Z=aa
```

Then the process is repeated with byte pair "ab", replacing it with "Y":

```
ZYdZYac
Y=ab
Z=aa
```

The only literal byte pair left occurs only once, and the encoding might stop here. Alternatively, the process could continue with recursive byte pair encoding, replacing "ZY" with "X":

```
XdXac
X=ZY
Y=ab
Z=aa
```

Appearance hide

Text

- Small
- Standard
- Large

Width

- Standard
- Wide

Result: After going through this process, instead of having a sequence of 11 tokens with a vocabulary length of 4, we now have a sequence of 5 tokens with a vocabulary length of 4. The BPE algorithm can be applied in the same way to byte sequences. Starting with a vocabulary size of 256, we iteratively find the byte pairs that occur most frequently, mint new tokens, append them to the vocabulary, and replace occurrences in the data. This results in a compressed dataset and an encoding/decoding algorithm.

implementation of tokenization in large language models (LLMs):

Now, I will start the implementation of tokenization in large language models (LLMs). Tokenization is a crucial component of state-of-the-art LLMs. Still, it is necessary to understand

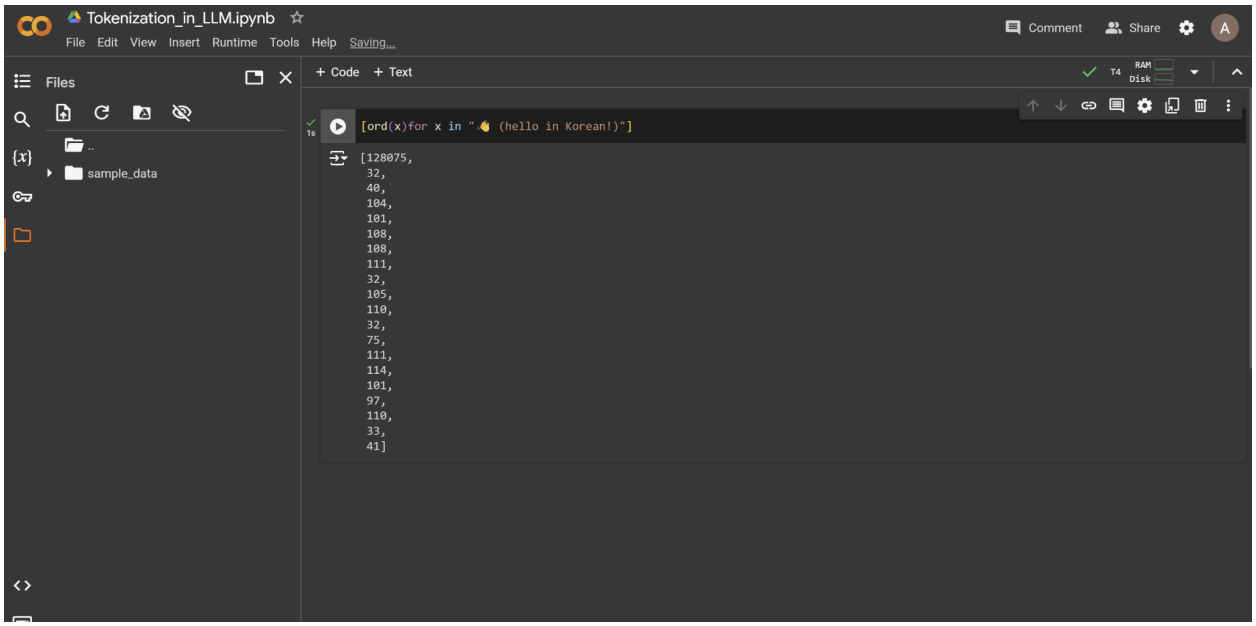
in some detail because a lot of the shining results of LLMs may be attributed to the neural network or otherwise actually traced back to tokenization.

Unicode tokenization:

What is Unicode?

In Python, strings are immutable sequences of Unicode code points. The Unicode Consortium defines Unicode code points as part of the Unicode standard, which currently defines roughly 150,000 characters across 161 scripts. The standard is alive, with the latest version 15.1 released in September 2023.

We can access the Unicode code point for a single character using Python's ord() function. For example:



Using ord() to get Unicode code points for characters in a string.

However, we can't simply use these raw code point integers for tokenization, as the vocabulary would be too large (150,000+) and unstable due to the evolving Unicode standard.

Unicode Byte Encoding: To find a better solution for tokenization, we turn to Unicode byte encodings like ASCII, UTF-8, UTF-16, and UTF-32. These encodings define how to translate the abstract Unicode code points into actual bytes that can be stored and transmitted.

The Unicode Consortium defines three types of encodings: UTF-8, UTF-16 and UTF-32. These encodings are how we can take Unicode text and translate it into binary data or byte streams.

Encodings: UTF-8, UTF-16, and UTF-32

To store or transmit these Unicode characters, we need to convert (or encode) them into a sequence of bytes. UTF-8, UTF-16 and UTF-32 are different encoding schemes that specify how these code points are translated into byte streams.

- **UTF-8**: Variable-length encoding (1 to 4 bytes per code point)
- **UTF-16**: Variable-length encoding (2 or 4 bytes per code point)
- **UTF-32**: Fixed-length encoding (always 4 bytes per code point)

UTF-8 Encoding

UTF-8 is the most common encoding scheme because of its efficiency and compatibility with ASCII. It uses a variable number of bytes to encode each character based on its Unicode code point.

How UTF-8 Works

1. **1 Byte for ASCII**: The first 128 Unicode code points (0 to 127) correspond to ASCII characters and are encoded using a single byte.
2. **2 Bytes for Additional Characters**: The next 1,920 code points (128 to 2,047) are encoded using two bytes.
3. **3 Bytes for BMP**: The following 61,440 code points (2,048 to 65,535) are encoded using three bytes. This covers most of the Basic Multilingual Plane (BMP) characters.
4. **4 Bytes for Supplementary Planes**: Code points above 65,535 (up to 1,114,111) are encoded using four bytes. These include less common characters, such as certain Chinese, Japanese, and Korean (CJK) characters, historic scripts, and mathematical symbols.

Examples

1. **1-Byte Encoding (ASCII)**:
    - Character: 'A' (U+0041)
    - UTF-8 Encoding: 0x41
2. **2-Byte Encoding**:
    - Character: 'é' (U+00E9)
    - UTF-8 Encoding: 0xC3 0xA9
3. **3-Byte Encoding**:
    - Character: 'ह' (U+0939)

    - UTF-8 Encoding: 0xE0 0xA4 0xB9
4. **4-Byte Encoding**:
    - Character: '😊' (U+1F60A)
    - UTF-8 Encoding: 0xF0 0x9F 0x98 0x8A

Detailed Breakdown

1-Byte Encoding

For ASCII characters (0 to 127):

- The byte value is simply the ASCII value.
- Example: 'A' (U+0041) is 0x41 in UTF-8.

2-Byte Encoding

For code points from 128 to 2,047:

- The format is: 110xxxxx 10xxxxxx
- Example: 'é' (U+00E9)
    - Binary: 1110 1001 (U+00E9)
    - UTF-8: 11000011 10101001 (0xC3 0xA9)

3-Byte Encoding

For code points from 2,048 to 65,535:

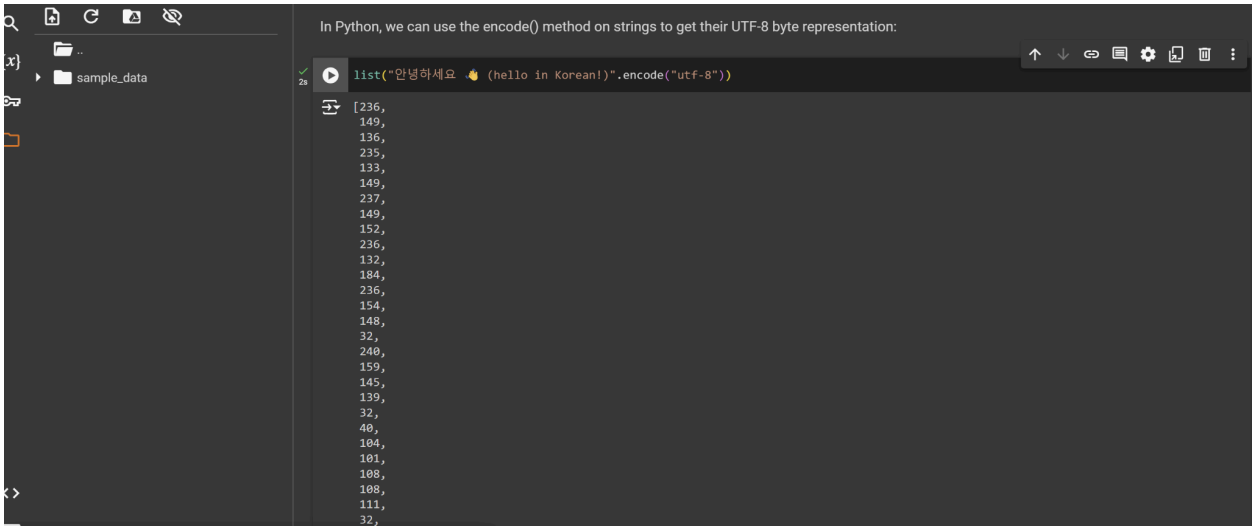
- The format is: 1110xxxx 10xxxxxx 10xxxxxx
- Example: 'ह' (U+0939)
    - Binary: 1001 0011 1001 (U+0939)
    - UTF-8: 11100000 10100100 10111001 (0xE0 0xA4 0xB9)

4-Byte Encoding

For code points from 65,536 to 1,114,111:

- The format is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
- Example: '😊' (U+1F60A)
    - Binary: 0001 1111 0110 0000 1010 (U+1F60A)
    - UTF-8: 11110000 10011111 10011000 10001010 (0xF0 0x9F 0x98 0x8A)

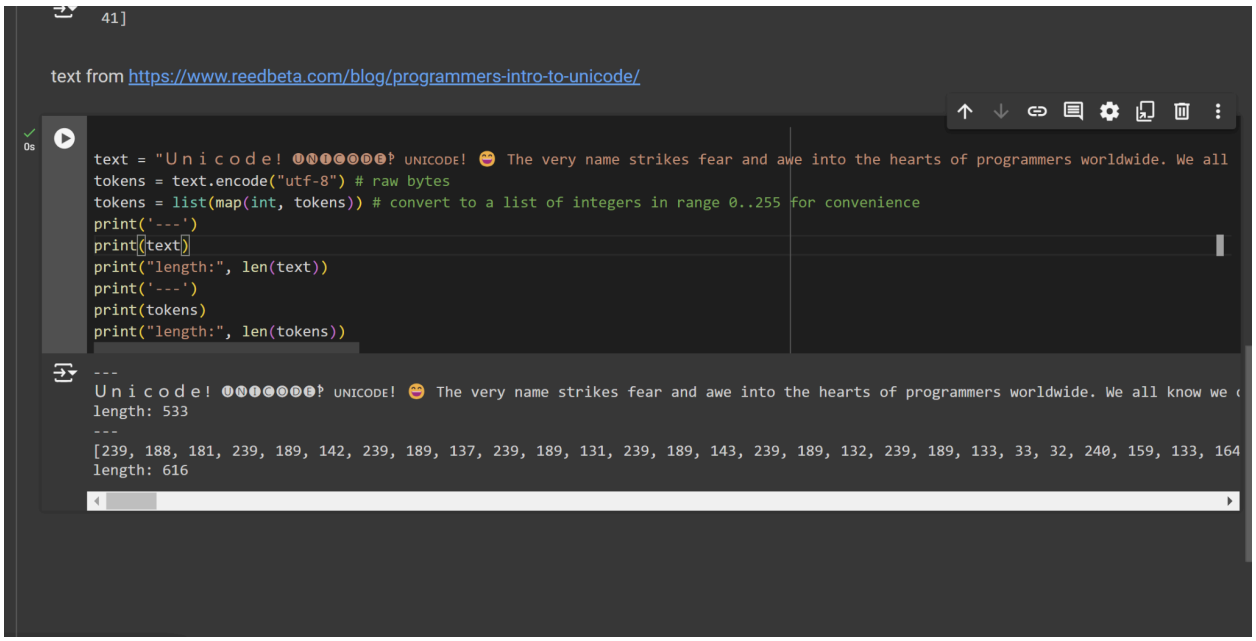
In Python, we can use the `encode()` method on strings to get their UTF-8 byte representation:



However, directly using the raw UTF-8 bytes would be very inefficient for language models. It would lead to extremely long sequences with a small vocabulary size of only 256 possible byte values. This prevents attending to sufficiently long contexts.

The solution uses a byte pair encoding (BPE) algorithm to compress these sequences to a variable amount. This allows efficient text representation with a larger but tunable vocabulary size.

To get the tokens, we take our input text and encode it into UTF-8. At this point, the tokens will be a raw bytes single stream of bytes. To make it easier to work with, we convert all those bytes to integers and create a list out of it for easier manipulation and visualization in Python.



Converting text to a list of token integers

The original paragraph has a length of 533 code points, but after encoding into UTF-8, it expands to 616 bytes or tokens. This is because while many simple ASCII characters become a single byte, more complex Unicode characters can take up to four bytes each.

Finding the most common Byte pair:

As a first step in the algorithm, we want to iterate over the bytes and find the pair of bytes that occur most frequently, as we will then merge them.

Here is one implementation of **Finding the most common Byte pair** in Python:

This function takes the list of token integers, counts the frequency of each consecutive pair, and returns the pair that appears most often. This is a key step in the byte pair encoding algorithm used for tokenization in many LLMs.

Finding Most Common Consecutive Pairs in Tokenized Text:

I will explore how to find the most commonly occurring consecutive pairs in a list of tokenized integers. I'll implement a function called `get_stats` that takes a list of integers and returns a dictionary keeping track of the counts of each consecutive pair.

For deeply understanding of tokenization in LLM, please follow the links below:

https://hundredblocks.github.io/transcription_demo/

What is the library for tokenization?

In Python, several libraries support tokenization for natural language processing (NLP). Each library may support different tokenization methods, including word-level, character-level, subword-level (like Byte Pair Encoding and WordPiece), and sentence-level tokenization. Here's a comprehensive list of some of the most commonly used libraries and their supported tokenization methods:

1. NLTK (Natural Language Toolkit)

- **Supported Tokenization Methods**:
  - Word Tokenization: `nltk.word_tokenize`
  - Sentence Tokenization: `nltk.sent_tokenize`
  - Regular Expression Tokenization: `nltk.RegexpTokenizer`
  - Character Tokenization (customizable via regex)

2. spaCy

- **Supported Tokenization Methods**:
  - Word Tokenization: `spacy.tokens.Token`
  - Sentence Tokenization: `spacy.tokens.Span`
  - Custom Tokenization Rules (via the `tokenizer` attribute)

3. Hugging Face Transformers

- **Supported Tokenization Methods**:
  - WordPiece Tokenization: `BertTokenizer`
  - Byte Pair Encoding (BPE): `GPT2Tokenizer`, `RobertaTokenizer`
  - SentencePiece Tokenization: `AlbertTokenizer`, `T5Tokenizer`, `XLMTokenizer`
  - Unigram Language Model: `SentencePieceUnigramTokenizer`

4. SentencePiece

- **Supported Tokenization Methods**:
  - Byte Pair Encoding (BPE): `sentencepiece.SentencePieceProcessor`
  - Unigram Language Model: `sentencepiece.SentencePieceProcessor`

5. Tokenizers (by Hugging Face)

- **Supported Tokenization Methods**:
    - WordPiece: `tokenizers.BertWordPieceTokenizer`
    - Byte Pair Encoding (BPE): `tokenizers.ByteLevelBPETokenizer`
    - SentencePiece: `tokenizers.SentencePieceBPETokenizer`
    - Unigram: `tokenizers.Unigram`

6. Gensim

- **Website**: Gensim
- **Supported Tokenization Methods**:
    - Simple Preprocessing: `gensim.utils.simple_preprocess`
    - Word Tokenization: `gensim.utils.tokenize`

installation steps for each of the mentioned libraries in Google Colab:

1. NLTK (Natural Language Toolkit)

```
!pip install nltk
```

2. spaCy

```
!pip install spacy

# Download the English model

!python -m spacy download en_core_web_sm
```

3. Hugging Face Transformers

```
!pip install transformers
```

4. SentencePiece

```
!pip install sentencepiece
```

5. Tokenizers (by Hugging Face)

```
!pip install tokenizers
```

6. Gensim

```
!pip install gensim
```

Example of tokenization:

NLTK Example

```
[ ]  !pip install nltk
```

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize
text = "Hello, world!"
print(word_tokenize(text))
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
['Hello', ',', 'world', '!']
```

Explanation:

Downloads the 'punkt' tokenizer models from NLTK.

Uses word_tokenize from NLTK to tokenize the text "Hello, world!" into individual words and punctuation.

spacy

```
[1]  !pip install spacy
     # Download the English model
     !python -m spacy download en_core_web_sm
```

```
[2]  import spacy
     nlp = spacy.load("en_core_web_sm")
     doc = nlp("Hello, world!")
     print([token.text for token in doc])
```

```
['Hello', ',', 'world', '!']
```

Explanation:

Loads the English language model en_core_web_sm for spaCy.

Tokenizes the text "Hello, world!" using spaCy's pipeline and prints each token.

transformers

```
[3] !pip install transformers
```

```
[4] from transformers import BertTokenizer
    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    print(tokenizer.tokenize("Hello, world!"))
```

Explanation:

Loads the BERT tokenizer with a pretrained 'bert-base-uncased' model.

Tokenizes the text "Hello, world!" into subwords as used in BERT models.

sentencepiece

```
[5] !pip install sentencepiece
```

```
0s # Create a sample text file
   with open("example.txt", "w") as f:
       f.write("This is a sample sentence for training the SentencePiece tokenizer.\n" * 2)
```

+ Code + Text

```
[16] import sentencepiece as spm
0s   spm.SentencePieceTrainer.Train('--input=/content/example.txt --model_prefix=m --vocab_size=30')
     sp = spm.SentencePieceProcessor()
     sp.load('m.model')
     print(sp.encode_as_pieces("Hello, world!"))
```

```
['_', 'H', 'e', 'l', 'l', 'o', ',', '_', 'w', 'o', 'r', 'l', 'd!']
```

Explanation:

Trains a SentencePiece model on the text file example.txt with a vocabulary size of 30 and prefix m.

Loads the trained model m.model and tokenizes the text "Hello, world!" into subwords.

tokenizers

```
[7] !pip install tokenizers
```

```
0s from tokenizers import BertWordPieceTokenizer
   tokenizer = BertWordPieceTokenizer()
   tokenizer.train(["/content/example.txt"])
   print(tokenizer.encode("Hello, world!").tokens)
```

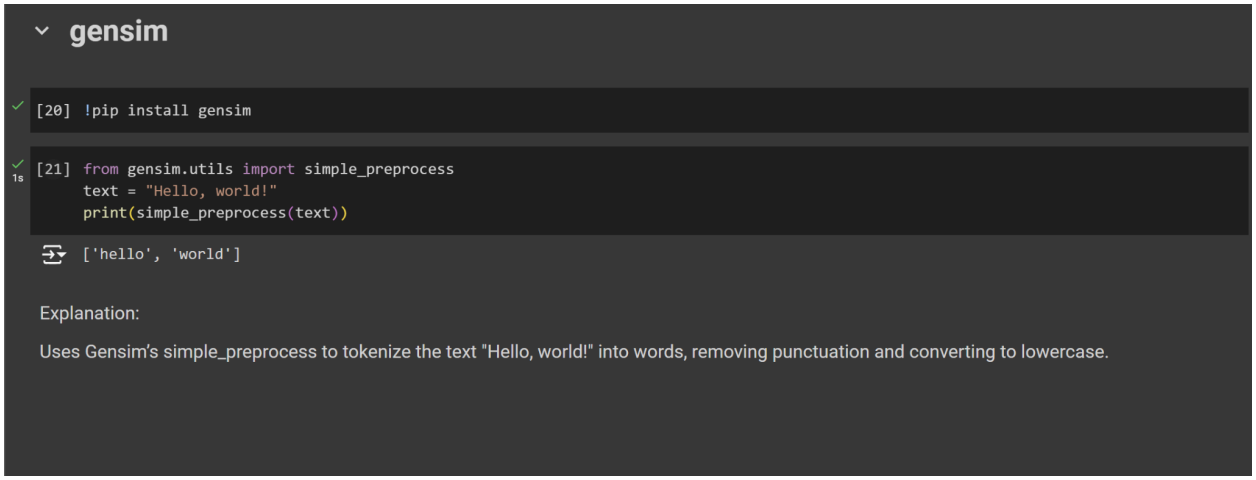
```
['h', '##e', '##l', '##l', '##o', '[UNK]', '[UNK]', '[UNK]']
```

Explanation:

Initializes the BertWordPieceTokenizer.

Trains the tokenizer on the text file example.txt.

Tokenizes the text "Hello, world!" and prints the tokens.



Methods for Working with LLMs:

1. Fine-tuning: Involves training the model further on a specific dataset to adapt it to a particular task or domain.

Fine-tuning methods:

Fine-tuning a Large Language Model (LLM) involves a supervised learning process. In this method, a dataset comprising labeled examples is utilized to adjust the model's weights, enhancing its proficiency in specific tasks. Now, let's delve into some noteworthy techniques employed in the fine-tuning process.

1. **Full Fine Tuning (Instruction fine-tuning)**: Instruction fine-tuning is a strategy to enhance a model's performance across various tasks by training it on examples that guide its responses to queries. The choice of the dataset is crucial and tailored to the specific task, such as summarization or translation. This approach, known as full fine-tuning, updates all model weights, creating a new version with improved capabilities. However, it demands sufficient memory and computational resources, similar to pre-training, to handle the storage and processing of gradients, optimizers, and other components during training.
2. **Parameter Efficient Fine-Tuning (PEFT)** is a form of instruction fine-tuning that is much more efficient than full fine-tuning. Training a language model, especially for full LLM fine-tuning, demands significant computational resources. Memory

allocation is not only required for storing the model but also for essential parameters during training, presenting a challenge for simple hardware. PEFT addresses this by updating only a subset of parameters, effectively "freezing" the rest. This reduces the number of trainable parameters, making memory requirements more manageable and preventing catastrophic forgetting. Unlike full fine-tuning, PEFT maintains the original LLM weights, avoiding the loss of previously learned information. This approach proves beneficial for handling storage issues when fine-tuning for multiple tasks. There are various ways of achieving Parameter efficient fine-tuning. Low-Rank Adaptation **LoRA** & **QLoRA** are the most widely used and effective.

What is LoRa?

LoRA is an improved finetuning method where instead of finetuning all the weights that constitute the weight matrix of the pre-trained large language model, two smaller matrices that approximate this larger matrix are fine-tuned. These matrices constitute the LoRA adapter. This fine-tuned adapter is then loaded into the pre-trained model and used for inference. After LoRA fine-tuning for a specific task or use case, the outcome is an unchanged original LLM and the emergence of a considerably smaller "LoRA adapter," often representing a single-digit percentage of the original LLM size (in MBs rather than GBs). During inference, the LoRA adapter must be combined with its original LLM. The advantage lies in the ability of many LoRA adapters to reuse the original LLM, thereby reducing overall memory requirements when handling multiple tasks and use cases.

What is Quantized LoRA (QLoRA)?

QLoRA represents a more memory-efficient iteration of LoRA. **QLoRA** takes **LoRA** a step further by also quantizing the weights of the LoRA adapters (smaller matrices) to lower precision (e.g., 4-bit instead of 8-bit). This further reduces the memory footprint and storage requirements. In QLoRA, the pre-trained model is loaded into GPU memory with quantized 4-bit weights, in contrast to the 8-bit used in LoRA. Despite this reduction in bit precision, QLoRA maintains a comparable level of effectiveness to LoRA.

Example 1: Fine-tuning All the GPT-2 on a wikitext to improve its performance in forecasting the next token. I know, it is completely computational, and time and memory-consuming, however, on the first try I did it. Link to the project I did for this:

https://github.com/monirmo97/LLM/tree/main/Train_Test_GPT2

Example 2: Fine-tuning partially the GPT-2 on a wikitext to improve its performance in forecasting the next token. In this example, The model is fine-tuned by freezing most of the layers except for the last transformer block and the language modeling head. Link to the project I did for this:

https://github.com/monirmo97/LLM/tree/main/Train_Test_GPT2/gpt2-finetune-freeze

Example 3: In this example, I try to use LoRa and QLoRa for fine-tuning. Link to the project I did for this concept:

https://github.com/monirmo97/LLM/tree/main/Train_Test_GPT2/gpt2-fintuning-LoRa-QRaLo

Example 4: In this example, fine-tune a pre-trained language model to understand and generate context based on personal data.

https://github.com/monirmo97/LLM/tree/main/Fine-tune-microsoft-model

2. Prompt

Of all the inputs to a large language model, by far the most influential is the text prompt. Large language models can be prompted to produce output in a few ways:

- **Instruction:** Tell the model what you want
- **Completion:** Induce the model to complete the beginning of what you want
- **Scenario:** Give the model a situation to play out
- **Demonstration:** Show the model what you want, with either:
    - A few examples in the prompt
    - Many hundreds or thousands of examples in a fine-tuning training dataset

An example of each is shown below.

1. **Instruction prompts:** Write your instruction at the top of the prompt (or at the bottom, or both), and the model will do its best to follow the instructions and then stop. Instructions can be detailed, so feel free to write a paragraph explicitly detailing the output you want, just stay aware of how many tokens the model can process.

   Example of instruction prompt:

   > Extract the name of the author from the quotation below
   >
   > "Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence."
   >
   > Ted Chiang, Exhalation
   >
   > Output: Ted Chiang

2. **Completion prompt:** Completion-style prompts take advantage of how large language models try to write text they think will likely come next. To steer the model, try beginning a pattern or sentence that will be completed by the output you want to see. This mode of steering large language models can take more care and experimentation relative to direct instructions. In addition, the models won't know where to stop, so you will often need stop sequences or post-processing to cut off text generated beyond the desired output.

   Example of completion prompt:

   > "Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence."
   > Ted Chiang, Exhalation
   > The author of this quote is
   > Output: Ted Chiang

3. **Scenario prompt:** Giving the model a scenario to follow or a role to play out can be helpful for complex queries or when seeking imaginative responses. When using a hypothetical prompt, you set up a situation, problem, or story, and then ask the model to respond as if it were a character in that scenario or an expert on the topic.

   Example of scenario prompt:

Your role is to extract the name of the author from any given text

"Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence."
Ted Chiang, Exhalation
Output: Ted Chiang

4. **Demonstration prompt (few-shot learning):** Like completion-style prompts, demonstrations can show the model what you want it to do. This approach is sometimes called few-shot learning, as the model learns from a few examples in the prompt.

   Example of demonstration prompt:

   Quote:

   "When the reasoning mind is forced to confront the impossible repeatedly, it has no choice but to adapt."

   N.K. Jemisin, The Fifth Season
   Author: N.K. Jemisin

   Quote:
   "Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, the night sky's hush is the graveyard's silence."
   Ted Chiang, Exhalation
   Author:

   Output: Ted Chiang

5. **Fine-tuned prompt:** You can fine-tune a custom model with enough training examples. In this case, instructions become unnecessary, as the model can learn the task from the training data provided. However, it can be helpful to include separator sequences (e.g., -> or ### or any string that doesn't commonly appear in your inputs) to tell the model when the prompt has ended and the output should begin. With separator sequences, the model can continue elaborating on the input text rather than starting on the answer you want to see.

Example of fine-tuned prompt: (for a model that has been custom-trained on similar prompt-completion pairs):

"Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence."

Ted Chiang, Exhalation

Output: Ted Chiang

In addition to text prompts, there are other ways to input data into a large language model:

1. **Multimodal Inputs**: Combining text with images or other media types (e.g., CLIP models by OpenAI).
2. **Structured Data**: Feeding in tables, graphs, or JSON objects.
3. **Interactive Systems**: Integrating with applications that provide dynamic, real-time inputs, such as chatbots or virtual assistants.
4. **Sensor Data**: Using IoT devices to send real-time sensor data as inputs.
5. **Voice Input**: Converting speech to text using speech recognition systems.

References[https://github.com/openai/openai-cookbook]