Double-click (or enter) to edit

AI TOOLS

- **You.com**
- **Pi**
- **Perpexility Labs**
- **Groq**

Groq Github

- **Ollama**
- **ChatGPT for DevOps: Introducing Kubiya.ai**
- **My Collection**

Double-click (or enter) to edit

Agents

- **Devika**
- **CrewAI**

Github crewai

- **Autogen**
- **PraisonAI**

GPT Engineer

[MetaGPT]

- [Suno V3]

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
from PIL import Image
from IPython.display import display

def display_image(image_path):
    try:
        img = Image.open(image_path)
        display(img)
    except Exception as e:
        print("Error:", e)
```

GenerativeAI

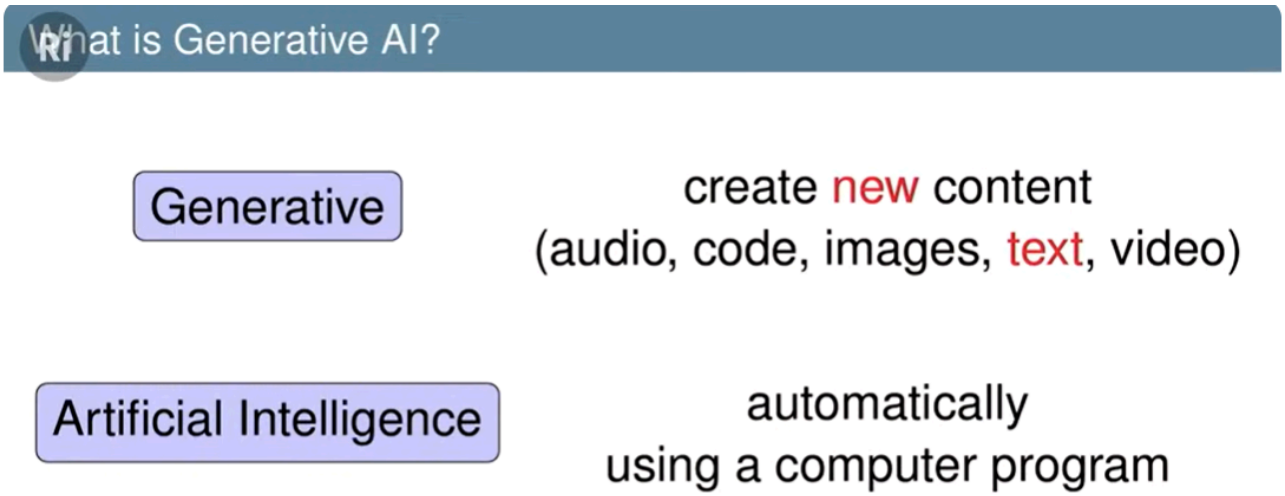
What is GenerativeAI

- Generative AI refers to a class of artificial intelligence techniques and algorithms designed to generate new content, data, or outputs that mimic or are similar to those found in the training data it has been exposed to. Instead of simply recognizing patterns in data or making predictions based on existing data, generative AI models can create entirely new content that has never been seen before.
- Generative AI encompasses various approaches, including generative adversarial networks (GANs), autoregressive models, variational autoencoders (VAEs), and others. These

models are commonly used in fields such as natural language processing, computer vision, music generation, and more.

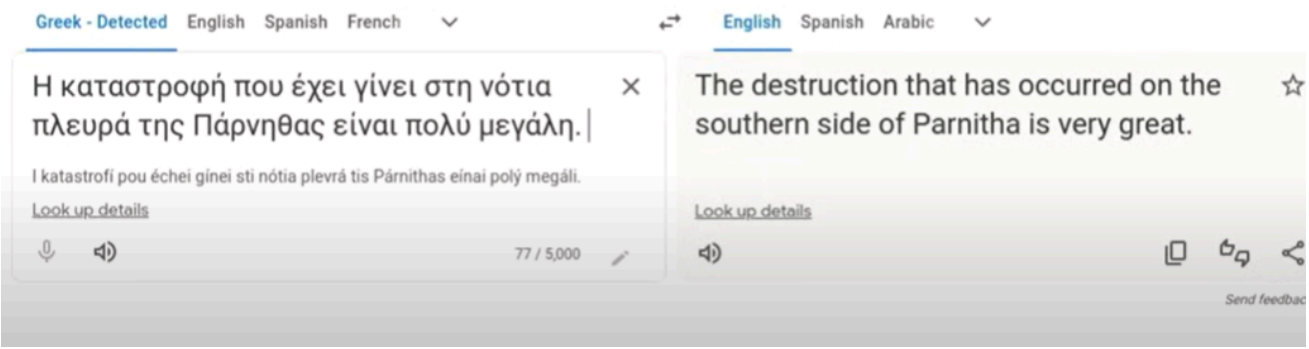
```
Start coding or generate with AI.
```

```
display_image('/content/what_GenAI.png')
```



GenerativeAI is not a new Concept

```
display_image('/content/GOOGLE_Translate.png')
```



Before the advent of large language models (LLMs), such as GPT models, statistical methods were the dominant approach in natural language processing tasks like machine translation. One of the prominent techniques used was Statistical Machine Translation (SMT). Here's how SMT works and how it differs from the techniques used in LLMs:

1.Statistical Machine Translation (SMT):

- **Explanation:** SMT is based on statistical models that learn translation patterns from large bilingual corpora. These models estimate the likelihood of translating a source sentence to a target sentence based on observed translations in the training data.

Example Sentence (Source Language - French):

"Le chat noir dort sur le tapis."

Translation Task:

Translate the sentence from French to English using alignment models, translation models, and language models.

Alignment Model:

The alignment model identifies word correspondences between the source (French) and target (English) languages.

For example, it might align "Le" with "The", "chat" with "cat", "noir" with "black", "dort" with "sleeps", and so on.

Translation Model:

The translation model captures translation patterns between phrases or sub-sentential units in the source and target languages. For example, it learns that "le chat noir" often translates to "the black cat" and "sur le tapis" translates to "on the carpet".

Language Model:

The language model estimates the probability of generating a sequence of words in the target language (English). For example, it ensures that the generated translation is fluent and natural-sounding, taking into account English syntax and semantics.

- **Components:** SMT systems typically consist of alignment models, translation models, and language models. Alignment models learn word alignments between source and target sentences, translation models estimate the translation probabilities of phrases or words, and language models capture the probability of generating target language sentences.

- *Example:** In a basic SMT system, given a source sentence in one language, the model uses statistical probabilities to select the most likely translation in the target language based on the learned patterns from the training data.

- *Tools:** Common tools and frameworks used for building SMT systems include Moses, Phrasal, and Apertium.

```
display_image('/content/voice_assistant.jpg')
```

Early versions of virtual assistants like Siri utilized machine learning techniques to enhance their performance. These included Hidden Markov Models (HMMs) for speech recognition, supervised learning algorithms for understanding language semantics and classifying user queries, natural language processing (NLP) techniques for processing text input, and feature engineering for extracting relevant information. While effective at the time, modern virtual assistants have evolved to employ more advanced deep learning techniques like recurrent neural networks (RNNs), convolutional neural networks (CNNs), and transformers, enabling higher accuracy, robustness, and adaptability to user inputs and contexts.

Hidden Markov Models (HMMs):

Hidden Markov Models are used for tasks like speech recognition, where the goal is to understand spoken words. Imagine you're talking to a virtual assistant like Siri. It listens to your words and tries to figure out what you're saying. HMMs help by modeling the probabilities of different sounds and words occurring together. For example, if you say "hello," the virtual assistant's HMM might recognize the sequence of sounds and guess that you're greeting it.

Supervised Learning Algorithms:

Supervised learning algorithms are used to understand the meaning of language and classify user queries. For example, when you ask Siri a question like "What's the weather like today?" it needs to understand that you're asking about the weather and then find the relevant information for you. Supervised learning algorithms are trained on lots of examples of questions and their correct answers to learn how to classify new questions.

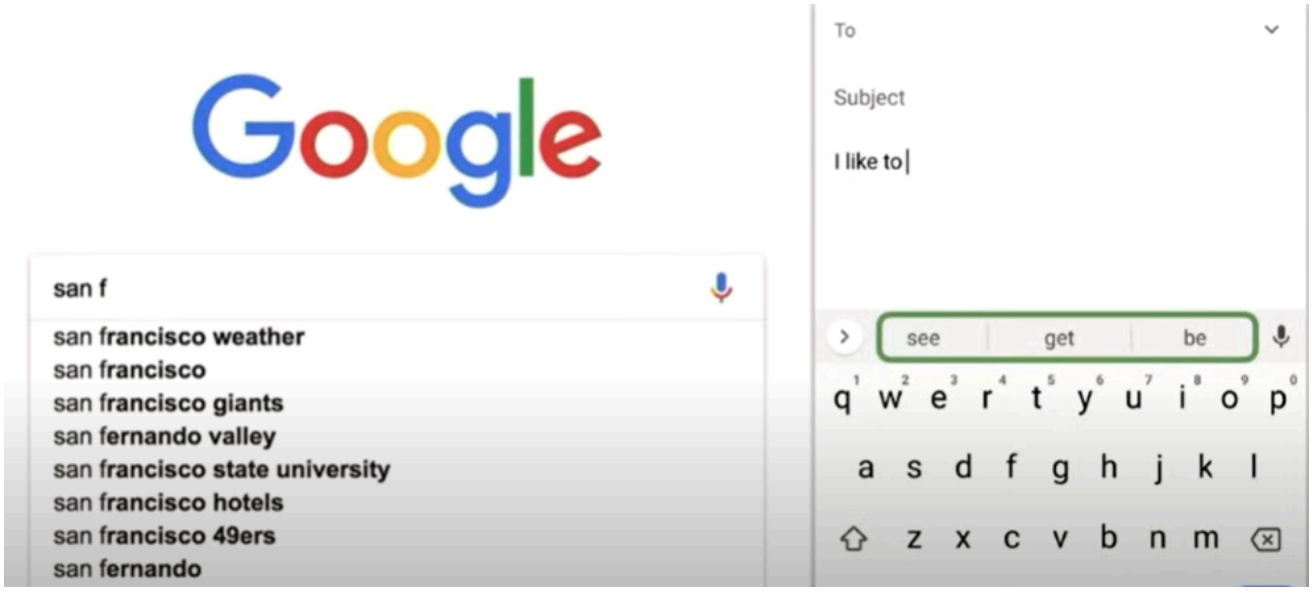
Natural Language Processing (NLP) Techniques:

Natural Language Processing techniques are used to process and understand text input. When you type a message to Siri, it needs to understand the meaning of your words. NLP techniques help by analyzing the structure and meaning of sentences. For example, if you type "Remind me to buy milk tomorrow," NLP techniques help Siri understand that you want to be reminded to buy milk the next day.

Feature Engineering:

Feature engineering involves extracting relevant information from the input data to help machine learning algorithms make better predictions. For example, if Siri is trying to understand a spoken command, it might extract features like the frequency of different sounds or the length of pauses between words to help the HMM recognize the spoken words more accurately.

```
display_image('/content/google_sent_complete.png')
```



[Google Sentence Completion Link]

```
Start coding or generate with AI.
```

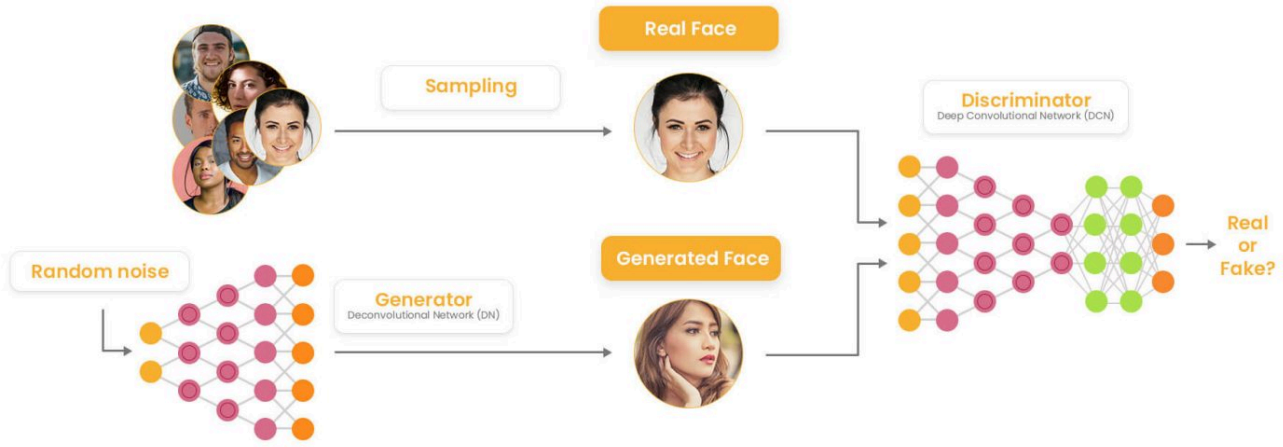
Different GenerativeAI

Generative Adversarial Networks (GANs):

Generative Adversarial Networks (GANs) can be broken down into three parts:

- Generative: To learn a generative model, which describes how data is generated in terms of a probabilistic model.
- Adversarial: The word adversarial refers to setting one thing up against another. This means that, in the context of GANs, the generative result is compared with the actual images in the data set. A mechanism known as a discriminator is used to apply a model that attempts to distinguish between real and fake images.
- Networks: Use deep neural networks as artificial intelligence (AI) algorithms for training purposes.

```
display_image('/content/Generative-Adversarial-Networks-Architecture-scaled.jpg')
```



How does a GAN work?

The steps involved in how a GAN works:

- **Initialization:** Two neural networks are created: a Generator (G) and a Discriminator (D). G is tasked with creating new data, like images or text, that closely resembles real data. D acts as a critic, trying to distinguish between real data (from a training dataset) and the data generated by G.

- **Generator's First Move:** G takes a random noise vector as input. This noise vector contains random values and acts as the starting point for G's creation process. Using its internal layers and learned patterns, G transforms the noise vector into a new data sample, like a generated image.

- **Discriminator's Turn:** D receives two kinds of inputs: Real data samples from the training dataset. The data samples generated by G in the previous step. D's job is to analyze each input and determine whether it's real data or something G cooked up. It outputs a probability score between 0 and 1. A score of 1 indicates the data is likely real, and 0 suggests it's fake.

- **The Learning Process:** Now, the adversarial part comes in: If D correctly identifies real data as real (score close to 1) and generated data as fake (score close to 0), both G and D are rewarded to a small degree. This is because they're both doing their jobs well. However, the key is to continuously improve. If D consistently identifies everything correctly, it won't learn much. So, the goal is for G to eventually trick D.

- **Generator's Improvement:** When D mistakenly labels G's creation as real (score close to 1), it's a sign that G is on the right track. In this case, G receives a significant positive update, while D receives a penalty for being fooled. This feedback helps G improve its generation process to create more realistic data.

- **Discriminator's Adaptation:** Conversely, if D correctly identifies G's fake data (score close to 0), but G receives no reward, D is further strengthened in its discrimination abilities.

This ongoing duel between G and D refines both networks over time. As training progresses, G gets better at generating realistic data, making it harder for D to tell the difference. Ideally, G becomes so adept that D can't reliably distinguish real from fake data. At this point, G is considered well-trained and can be used to generate new, realistic data samples.

Application Of Generative Adversarial Networks (GANs)

GANs, or Generative Adversarial Networks, have many uses in many different fields. Here are some of the widely recognized uses of GANs:

Image Synthesis and Generation : GANs are often used for picture synthesis and generation tasks, They may create fresh, lifelike pictures that mimic training data by learning the distribution that explains the dataset. The development of lifelike avatars, high-resolution photographs, and fresh artwork have all been facilitated by these types of generative networks.

Image-to-Image Translation : GANs may be used for problems involving image-to-image translation, where the objective is to convert an input picture from one domain to another while maintaining its key features. GANs may be used, for instance, to change pictures from day to night, transform drawings into realistic images, or change the creative style of an image.

Text-to-Image Synthesis : GANs have been used to create visuals from descriptions in text. GANs may produce pictures that translate to a description given a text input, such as a phrase or a caption. This application might have an impact on how realistic visual material is produced using text-based instructions.

Data Augmentation : GANs can augment present data and increase the robustness and generalizability of machine-learning models by creating synthetic data samples.

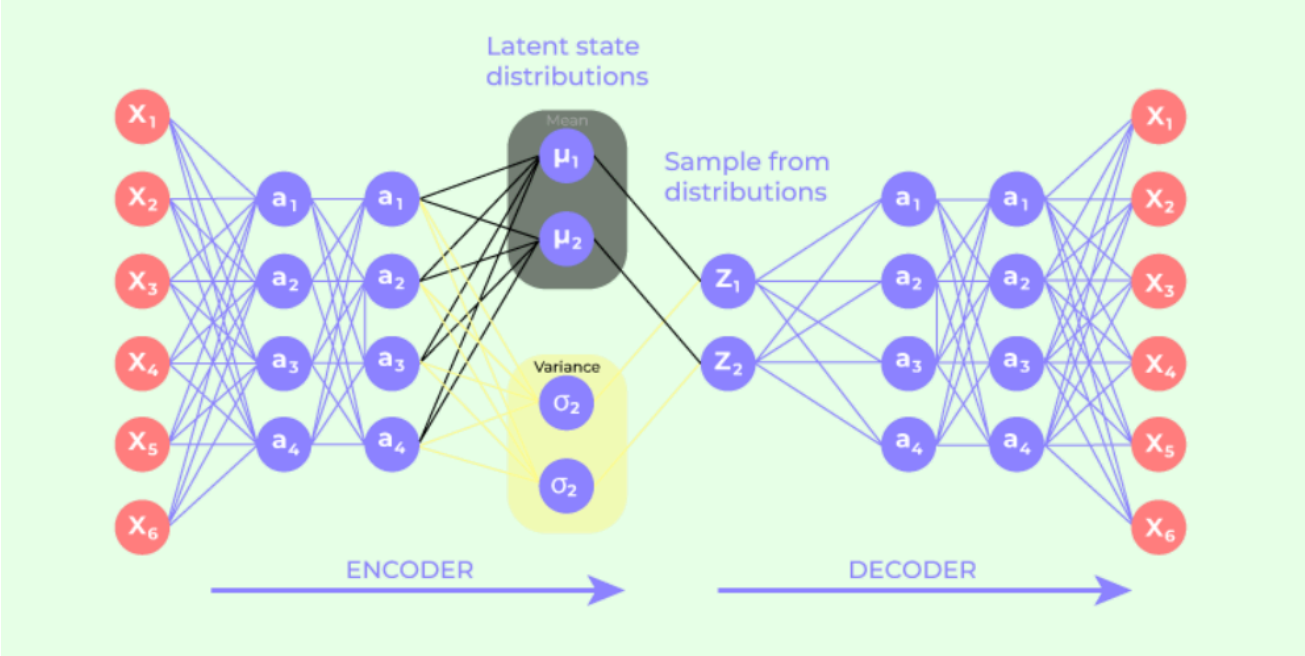
Data Generation for Training : GANs can enhance the resolution and quality of low-resolution images. By training on pairs of low-resolution and high-resolution images, GANs can generate high-resolution images from low-resolution inputs, enabling improved image quality in various applications such as medical imaging, satellite imaging, and video enhancement.

Variational AutoEncoders(VAE)

Variational Autoencoders (VAEs) are a type of generative model used in unsupervised learning tasks, particularly in the field of deep learning and neural networks. They are an extension of traditional autoencoders, which are neural network architectures used for dimensionality

reduction and feature learning. VAEs aim to learn the underlying probability distribution of input data in an unsupervised manner and generate new data samples similar to the training data.

```
display_image('/content/VAE.png')
```



Variational Autoencoder

Why it was introduced:

- Variational Autoencoders (VAEs) were introduced to address some limitations of traditional autoencoders. Autoencoders are neural networks used for unsupervised learning tasks, such as data compression, denoising, and feature learning. They consist of an encoder network that compresses the input data into a latent space representation and a decoder network that reconstructs the original input data from the latent space representation.
- However, traditional autoencoders have some drawbacks:
- They don't provide a probabilistic framework for generating new data samples. They don't explicitly learn a distribution over the latent space. They can generate only a fixed set of data samples from the input distribution. VAEs were introduced to overcome these limitations by introducing a probabilistic approach to learning latent representations. They

provide a way to generate new data samples by sampling from a learned distribution in the latent space, allowing for more flexibility and diversity in the generated samples.

Architecture:

- The architecture of a VAE consists of three main components:
- **Encoder Network:** This part of the network takes the input data and maps it to a distribution in the latent space. Unlike traditional autoencoders, the encoder network of a VAE outputs the parameters (mean and variance) of a Gaussian distribution that represents the latent space.
- **Latent Space:** The latent space is the low-dimensional representation of the input data where the data is mapped by the encoder network. It represents the underlying structure or features of the input data.
- **Decoder Network:** This part of the network takes samples from the latent space (usually drawn from the Gaussian distribution) and reconstructs the original input data. The decoder network learns to generate realistic data samples from the latent space representation.

Example:

- Imagine you have a dataset of handwritten digits (like the MNIST dataset). A traditional autoencoder would learn to compress each image into a fixed-size vector in the latent space, and then reconstruct the original image from this vector. However, it wouldn't capture the variability in how each digit can be written.
- On the other hand, a VAE would learn to map each digit image to a distribution in the latent space, capturing the variability in how each digit can be represented. This allows the VAE to generate new digit images by sampling from this distribution and decoding the samples into realistic images.
- In simpler terms, while autoencoders learn fixed representations of input data, VAEs learn probabilistic representations that capture the uncertainty and variability in the data, allowing for more flexible and diverse generation of new samples.

Example: Generating Molecular Structures

- Imagine you're working in drug discovery, and you want to generate new molecular structures that could potentially be candidates for new drugs. Each molecule can be represented as a sequence of atoms and bonds, and you want to learn a model that can generate new molecular structures.

Traditional Autoencoder Approach:

- With a traditional autoencoder, you would train a neural network to learn a mapping from the input molecular structures to a fixed-size vector in the latent space and then back to the reconstructed molecular structures. The encoder compresses the input molecules into a fixed-size vector, and the decoder reconstructs the original molecules from this vector.
- However, traditional autoencoders may not capture the diverse and realistic variations in molecular structures, limiting their ability to generate new and meaningful molecules.

Variational Autoencoder Approach:

- Now, let's consider using a Variational Autoencoder (VAE) for this task. In a VAE, instead of learning deterministic mappings, we learn probabilistic mappings that capture the variability in the latent space.
- **Encoder Network:** The encoder network takes the input molecular structures and maps them to a distribution in the latent space. Instead of outputting a fixed-size vector, the encoder outputs the parameters (mean and variance) of a multivariate Gaussian distribution that represents the latent space.
- **Sampling from Latent Space:** During training, we sample from this Gaussian distribution to get points in the latent space. These samples represent different variations or styles of molecular structures.
- **Decoder Network:** The decoder network takes these samples from the latent space and reconstructs the original input molecular structures. The decoder learns to generate realistic molecular structures corresponding to different samples from the latent space.

By learning a distribution over the latent space, VAEs capture the diversity in molecular structures and can generate new molecules by sampling from the learned distribution and decoding the samples into realistic molecular structures.

```
Start coding or generate with AI.
```

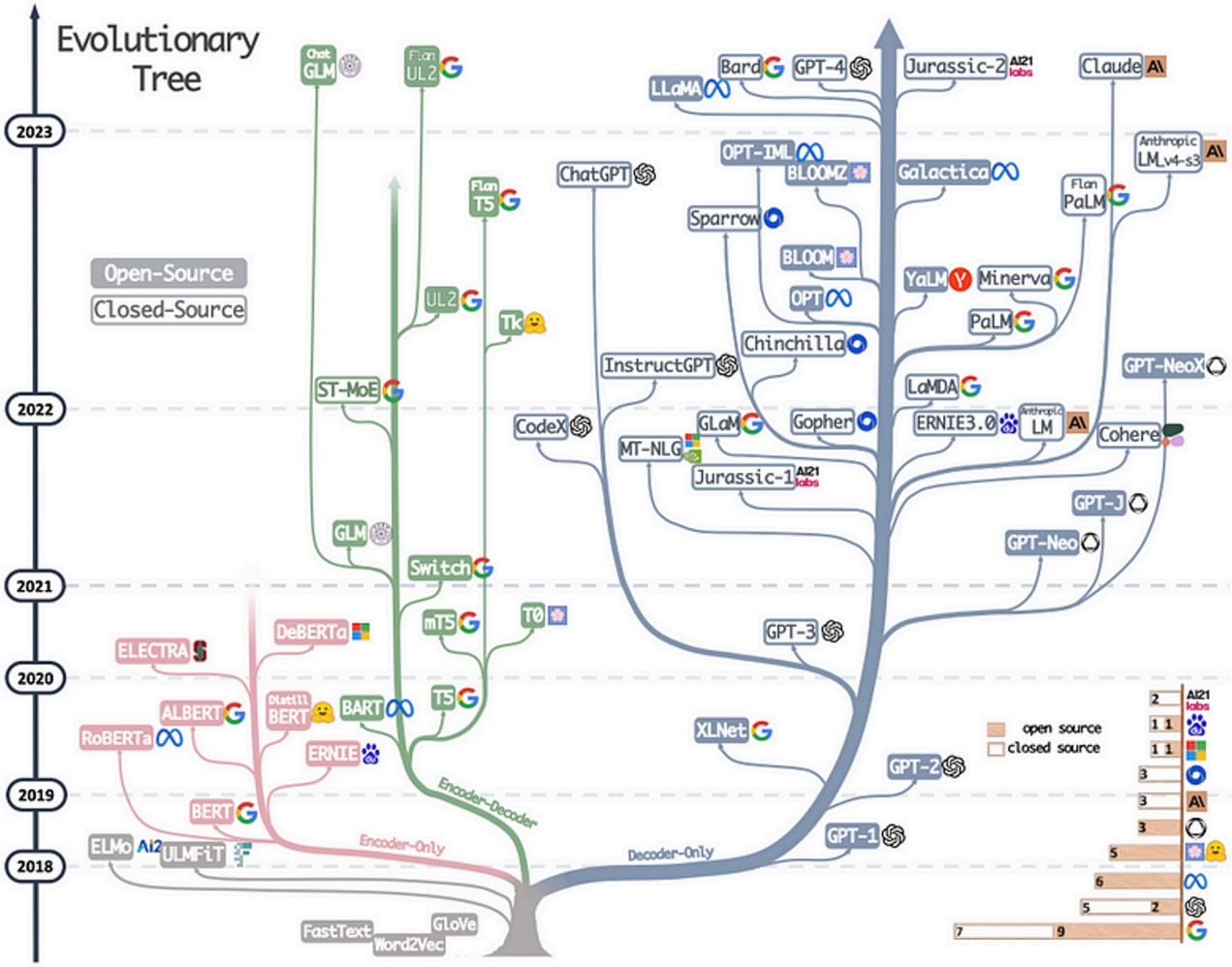
Autoregressive Models:

- Autoregressive models are a type of statistical model used in time series analysis and sequence generation tasks. The term "autoregressive" refers to the idea that each element in the sequence is predicted based on previous elements in the same sequence. This means that the model learns the conditional probability distribution of each element given its past elements.
- The name "autoregressive" stems from the fact that the model regresses on its own past outputs to predict future outputs in a sequence. Each element in the sequence is predicted

based on the previous elements, making it "auto" and "regressive" in nature. This modeling approach has proven to be effective in capturing dependencies and patterns in sequential data, leading to its widespread adoption in various applications, particularly in natural language processing.

Large Language Model

```
display_image('/content/intro_LLM.png')
```



"LLM" stands for "Large Language Model." This term is used to refer to language models that are exceptionally large in terms of the number of parameters and the amount of training data used to train them. LLMs are characterized by their ability to understand and generate human-like text across a wide range of tasks, including natural language understanding (NLU) and natural language generation (NLG).

The best definition of an LLM could be:

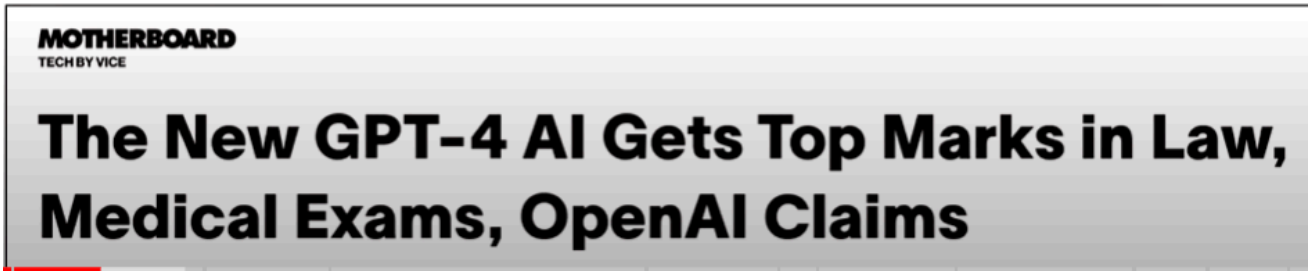
- "A Large Language Model (LLM) is a powerful artificial intelligence model that uses deep learning techniques, typically based on neural networks, to understand and generate human-like text. LLMs are trained on vast amounts of text data and learn to capture the complex patterns and structures of natural language. They are capable of performing a variety of language-related tasks, such as text generation, summarization, translation, question answering, and more. LLMs have significantly advanced the field of natural language processing (NLP) and have applications in various domains, including content generation, virtual assistants, chatbots, sentiment analysis, and information retrieval."

Why it is called language model:

- In the context of "Large Language Models" (LLMs), the term "language model" specifically refers to the neural network architectures used to process and generate human-like text. These models, which are typically trained using deep learning techniques, learn to understand and generate text based on the patterns and structures present in the training data. Therefore, they are called "language models" because they model the language in a way that allows them to generate coherent and contextually relevant text.

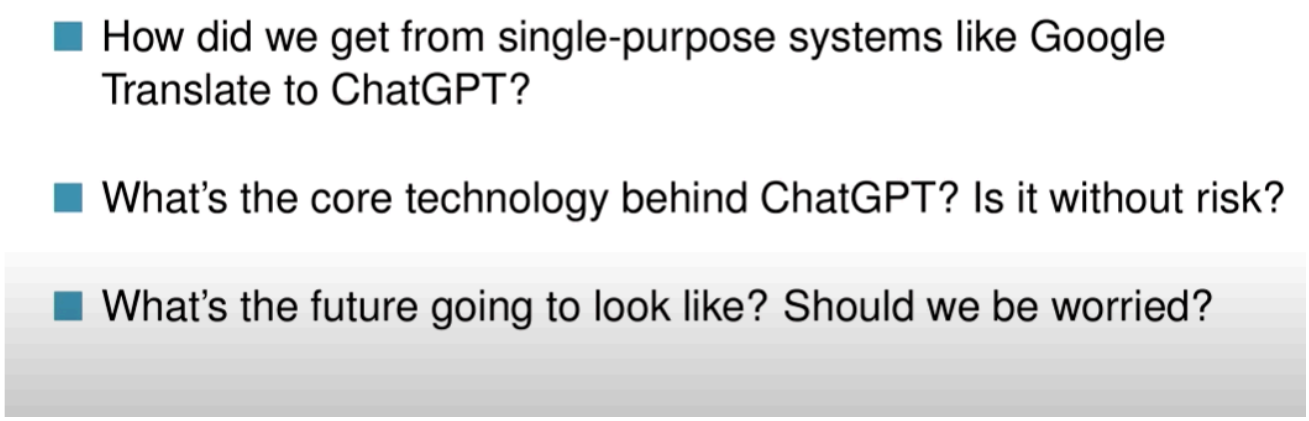
2023 OpenAI announcement

```
display_image('/content/openai_announce.png')
```




```
display_image('/content/time_100M.png')
```


```
display_image('/content/HOWWEREACHChatGPT.png')
```




```
display_image('/content/transformers.jpg')
```

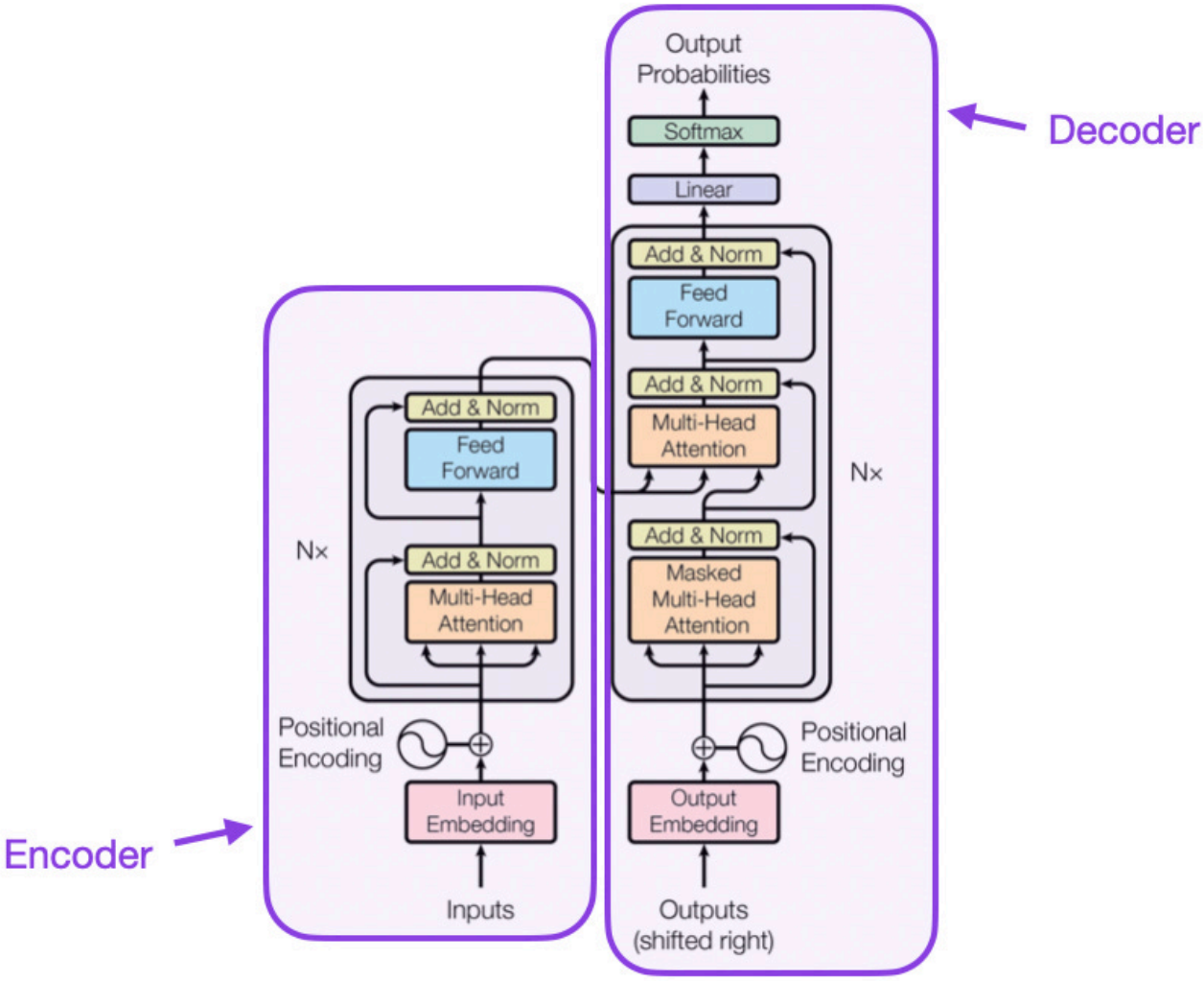


Figure 1: The Transformer - model architecture.

- **Encoder-Only Model:**

An encoder-only model, also known as an encoder or feature extractor, takes an input sequence and produces a fixed-length representation (embedding) that captures the relevant information of the input. It's commonly used in tasks where understanding the input sequence is essential but generating an output sequence isn't necessary. For example, in text classification tasks, an encoder-only model might take in a sentence and produce an embedding that represents the meaning or sentiment of the sentence without generating any additional text.

- **Decoder-Only Model:**

A decoder-only model, also known as a generator or autoregressive model, generates an output sequence based on a given input or context. It's typically used in tasks where generating

coherent and meaningful output sequences is the primary objective. For example, in text generation tasks like language modeling or machine translation, a decoder-only model might take in a sequence of tokens and produce a continuation of the sequence based on the context.

- **Encoder-Decoder Model:**

An encoder-decoder model consists of both an encoder and a decoder, where the encoder processes the input sequence and generates a representation, which is then used by the decoder to generate an output sequence. It's commonly used in tasks where there's a clear mapping between an input sequence and an output sequence, such as machine translation or sequence-to-sequence tasks. For example, in machine translation, the encoder-decoder model takes in a sentence in one language, encodes it into a fixed-length representation, and then decodes that representation into a sentence in another language.

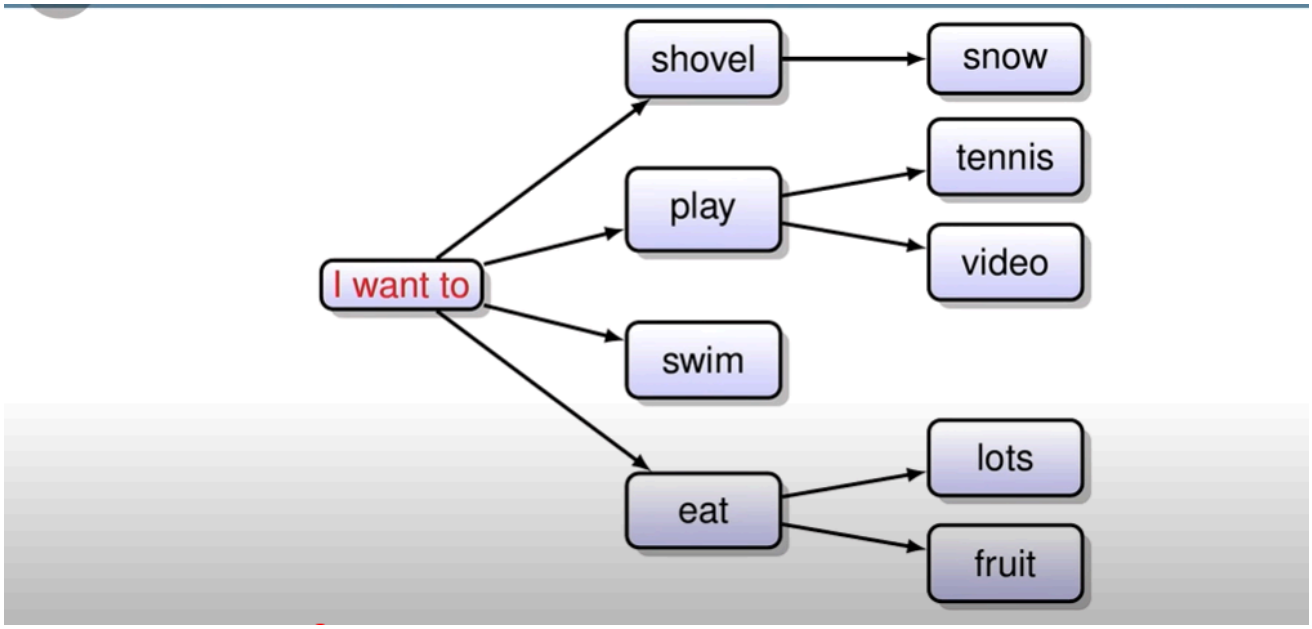
The "Add & Norm"

The "Add & Norm" operation in a Transformer involves adding the input to the output of the feedforward network and then normalizing the combined result. This process helps stabilize training and promotes effective information flow through the network.

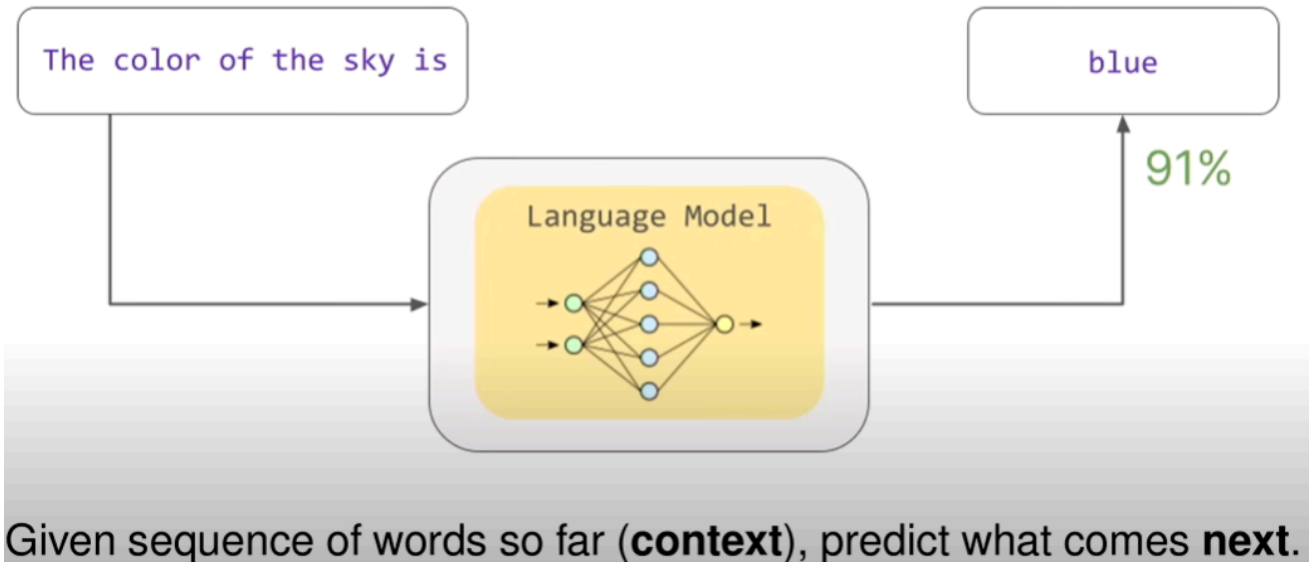
```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
display_image('/content/LanguageModelNextWord.png')
```



```
display_image('/content/NxtWordLanguageModel.png')
```



Recipe for creating own Language Model

```
display_image('/content/creatingownlanguagemodel.png')
```

Step 1: Collect a very large corpus:

- Wikipedia Books, StackOverflow
- Quora, Public Social media,
- Github, Reddit

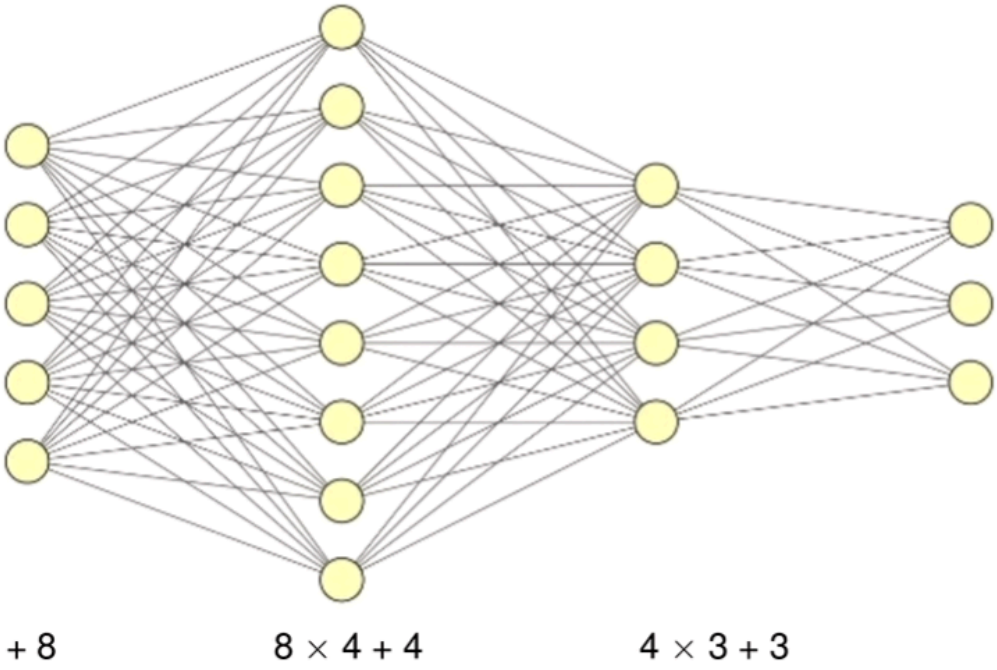
Step 2: Ask LM to predict the next word in a sentence:

- randomly truncate last part of input sentence
- calculates probabilities of missing words
- adjust and feed back to the model to match the ground truth

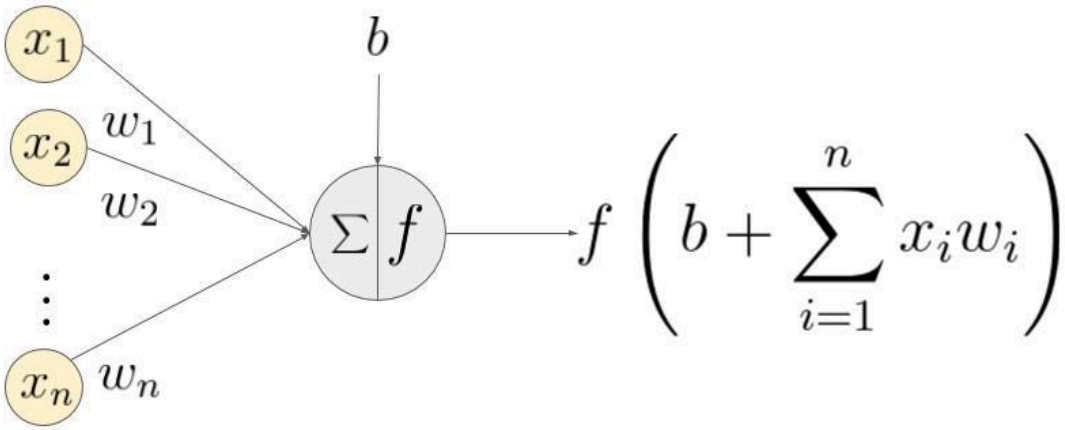
Step 3: Repeat over the whole corpus.

Neural Network Language Model

```
display_image('/content/neuralnetworkLM.png')
```



```
display_image('/content/neural network weights.jpg')
```



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

Here's a breakdown of how this process works:

Input:

Each neuron receives input from the neurons in the previous layer or directly from the input data. This input consists of numerical values (often represented as a vector) that represent features or attributes of the data being processed.

Processing:

The neuron computes a weighted sum of its inputs, where each input is multiplied by a corresponding weight. These weights represent the strength of the connections between the neurons. The neuron also applies an activation function to the weighted sum to introduce non-

linearity into the computation.

Consider a simple neural network with one neuron:

Input: x1, x2, x3 (input features)

Weights: w1, w2, w3 (connection strengths between the inputs and the neuron)

Bias: b (a constant term added to the weighted sum)

Activation function: σ() (non-linear function applied to the weighted sum)

The computation performed by the neuron can be expressed mathematically as follows:

1. Weighted Sum:
    - The neuron computes a weighted sum of its inputs by multiplying each input by its corresponding weight and summing the results: z = w1 * x1 + w2 * x2 + w3 * x3 + b
    - This weighted sum z represents the total input to the neuron, taking into account the strengths of the connections between the inputs and the neuron.
2. Activation Function:
    - The neuron applies an activation function σ() to the weighted sum to introduce non-linearity into the computation and determine the neuron's output: y = σ(z)

Example: Suppose we have the following input values and weights:

- x1 = 2, x2 = 3, x3 = 1 (input features)
- w1 = 0.5, w2 = -1, w3 = 0.3 (connection weights)
- b = 1 (bias)

Using the equations above, we can compute the weighted sum and the neuron's output:

1. Weighted Sum: z = (0.5 * 2) + (-1 * 3) + (0.3 * 1) + 1 z = 1 + (-3) + 0.3 + 1 z = -0.7
2. Activation Function:
    - Let's assume we use the sigmoid activation function: σ(z) = 1 / (1 + e^(-z))
    - Substituting z = -0.7 into the sigmoid function: y = 1 / (1 + e^0.7) y ≈ 1 / (1 + 0.496) y ≈ 1 / 1.496 y ≈ 0.669

So, in this example, the output of the neuron y is approximately 0.669. This computation demonstrates how a neuron processes its inputs using weighted sums and activation functions

to produce an output.

Activation Function:

The activation function takes the weighted sum of inputs and applies a non-linear transformation to it. This allows the neuron to model complex relationships between inputs and outputs and introduces the capability to learn non-linear patterns in the data.

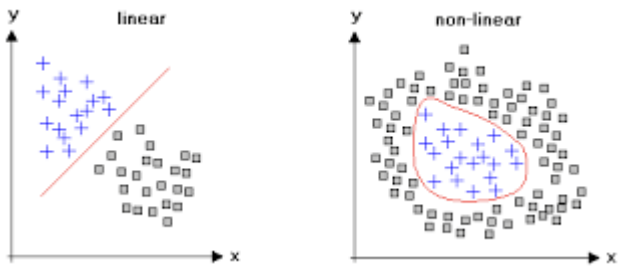
Example why activation function is used:

- Consider a simple classification problem where the data points are distributed in a concentric circle pattern. Using a linear activation function like the identity function (σ(z) = z) would result in a linear decision boundary, unable to separate the classes effectively. However, by applying a non-linear activation function like the ReLU (Rectified Linear Unit) function (σ(z) = max(0, z)), the neural network can learn a non-linear decision boundary, effectively separating the classes.
- In image classification tasks, the input data consists of pixel values representing images. The relationships between pixels within an image are highly non-linear, and the features defining objects or patterns within images can be complex and hierarchical. Non-linear activation functions, such as the sigmoid function (σ(z) = 1 / (1 + exp(-z)) or the hyperbolic tangent function (tanh), allow neural networks to learn hierarchical representations of features in images, capturing the intricate details necessary for accurate classification.

Output:

The result of the activation function becomes the output of the neuron, which is then passed as input to the neurons in the next layer. In a multi-layer neural network, this process is repeated across multiple layers, with each layer transforming the input received from the previous layer before passing it on to the next layer.

```
display_image('/content/nonlinearity.png')
```




```
Start coding or generate with AI.
```

Types of parameters

Here are a few parameters in ChatGPT-3 neural network:

- Weights:

  They are the most fundamental parameters in ChatGPT-3 that model learns from the training data. For instance, if ChatGPT-3 sees the word "cat" followed by the word "meowed," it will assign a higher weight to this word pair. Next time, the model will more likely predict the word "meowed" after the "cat."

  Each connection between neurons in the input and output layers (or between subsequent layers) is associated with a weight. These weights determine the strength of the connection between neurons. During training, the weights are adjusted to minimize the loss function and improve the model's performance.

- Bias:

  This parameter is used as an adjusting factor to tune the prediction of an entire layer to the more accurate side. Because of this reason, it is the same for a specific layer and applied layerwise.

Imagine you're a real estate agent trying to predict house prices based on certain factors like the size of the house, the number of bedrooms, and the neighborhood's safety level. You're using a neural network to help you make these predictions.

Now, think of biases as an additional adjustment factor that the neural network can learn and use to fine-tune its predictions.

Adding Flexibility:

Without biases, your predictions would be limited to a specific range dictated solely by the input features. However, with biases, the neural network gains flexibility. It can adjust its predictions up or down independently of the input features. This flexibility is essential because not all houses with similar features will have the same price due to other factors like market trends or neighborhood desirability.

Capturing Patterns:

Imagine you're analyzing data from a city where houses in certain neighborhoods tend to be more expensive overall. Even if two houses have the same size and number of bedrooms, the one in the more prestigious neighborhood will likely have a higher price. Biases help the neural network learn these patterns by allowing it to account for factors that might not be directly represented in the input features.

Improving Accuracy:

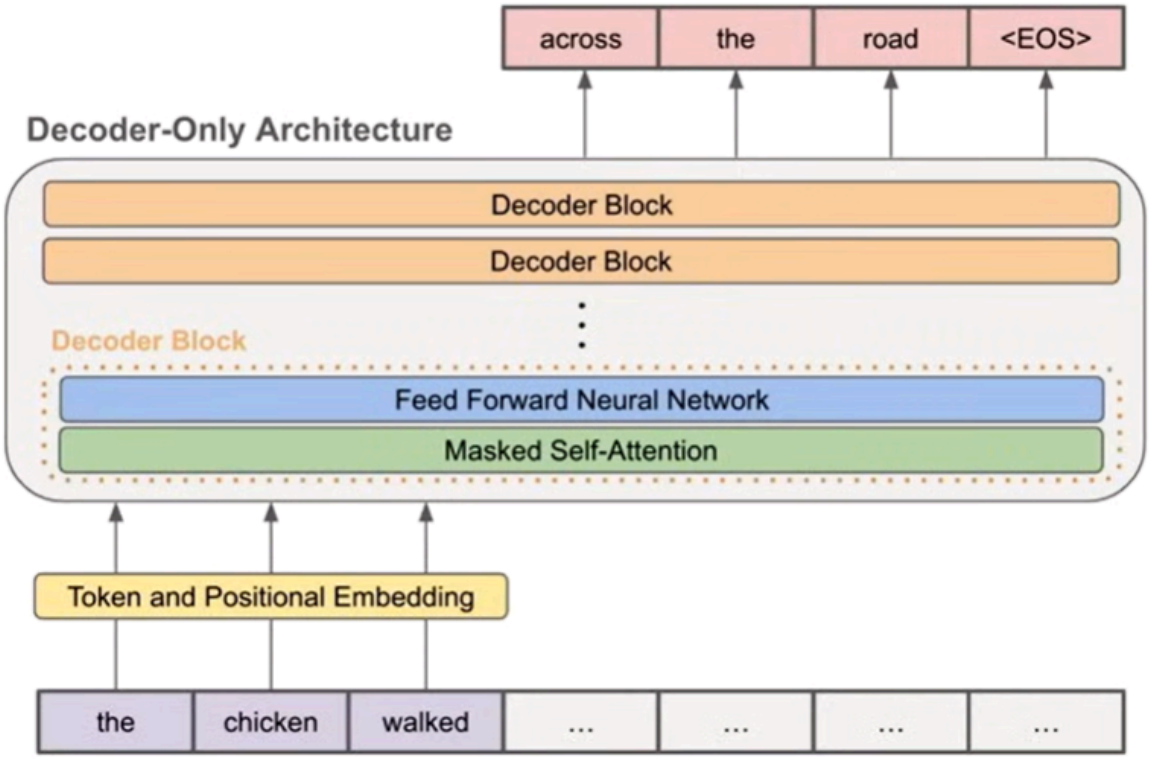
In real estate, there are many factors that influence house prices, some of which may not be captured by the input features alone. Biases enable the neural network to consider these factors, leading to more accurate predictions. For example, if there's a sudden increase in demand for houses in a certain area, biases allow the model to adjust its predictions accordingly, resulting in more accurate price estimates.

- Learning rate:

  This parameter suggests how much the biases and the weights have to be adjusted in order to reduce the error and make the results more accurate. This process of reducing the error is done during backpropagation.

The King of AI Architecture

```
display_image('/content/DecoderArchitecture.png')
```



The decoder architecture is a crucial component in various sequence generation tasks, such as machine translation, text summarization, and language modeling. It takes encoded representations of input sequences (if any) and generates output sequences token by token. Here are the components typically found in a decoder architecture:

Initial State Initialization:

- The decoder initializes its initial hidden state using the context vector obtained from the encoder (in tasks like machine translation) or using a fixed-length vector in the case of autoregressive language modeling.

Token Embedding:

- Each token in the output sequence (e.g., words or subwords) is represented as a dense vector through an embedding layer. This layer maps each token to a high-dimensional embedding space where semantically similar tokens are closer together.

Positional Encoding:

- Positional encoding is added to the token embeddings to provide information about the position of each token in the sequence. This helps the model distinguish between tokens

with the same embedding but at different positions.

A dog chases a cat

A cat chases a dog

Multi-Head Attention:

Multi-head attention mechanisms attend to different parts of the input sequences to capture relevant information. The decoder typically performs self-attention over the previously generated tokens to incorporate context from the output sequence.

Multi-Head Attention Mechanisms:

- In multi-head attention, the model doesn't just focus on one aspect of the input sequence; instead, it attends to different parts of the input simultaneously. Each "head" of attention can learn to pay attention to different aspects of the input, capturing various patterns and information.

- Example:

  Suppose you're translating the English sentence "The cat sits on the mat." into French. One head of attention might focus on the word "cat" to understand its meaning and context in the sentence. Another head might focus on the words "sits" and "mat" to capture the relationship between the actions and objects in the sentence. By attending to different parts of the input, each head contributes to a more comprehensive understanding of the sentence, helping the model generate a more accurate translation.

Decoder Self-Attention:

- In the decoding phase of the sequence-to-sequence model (where the model generates the output sequence), self-attention is performed over the previously generated tokens in the output sequence. This allows the decoder to incorporate context from the output sequence, ensuring that each token is generated in the context of the previously generated tokens.
- Example:

  Let's say the model has already generated the partial French translation "Le chat" (which means "The cat" in French). Now, when generating the next token, "s'assied" (which means "sits" in French), the decoder performs self-attention over "Le chat" to understand the

context in which "s'assied" should be generated. By attending to "Le chat," the decoder ensures that the translation remains coherent and grammatically correct.

If you're trying to understand a story, but there are many different characters and plotlines. Multi-head attention is like having several pairs of eyes, each focused on a different part of the story. This helps you capture important details from different perspectives and understand the overall narrative better. In the decoder, it's like looking back at what you've translated so far and paying attention to different parts to make sure everything fits together coherently.

Masking:

Masking is applied to prevent the model from attending to future tokens during training, ensuring that each token is generated based only on the previously generated tokens.

When you're predicting the next word in a sentence, you shouldn't peek at words that come after it. Masking is like covering up the future words so that you only see what's come before. This prevents you from cheating by looking ahead and helps you focus on predicting the next word based on the context you already have.

Position-wise Feedforward Networks:

Position-wise feedforward networks apply fully connected layers independently to each position in the sequence, enabling the model to capture complex non-linear relationships between tokens.

Imagine you're analyzing a sentence, and you want to understand the relationships between different words. Position-wise feedforward networks are like magnifying glasses that you use to examine each word individually, focusing on its connections with nearby words. By doing this independently for each word, you can capture intricate patterns and relationships between words throughout the sentence.

Output Layer:

The output layer produces the probability distribution over the vocabulary for the next token in the sequence. This distribution is typically computed using a softmax activation function.

Sampling (Optional):

During inference, the decoder may use sampling techniques, such as greedy decoding or beam search, to select the next token based on the probability distribution output by the softmax layer.

Beam Search

Model Output at each step:

Step 1:

Given the prompt "The cat", the model calculates the probabilities of all possible next words. Let's say the top three words and their probabilities are:

"is": 0.6

"sat": 0.3

"runs": 0.1

The model expands each candidate sequence with the top three words, calculates their probabilities, and retains the top three sequences with the highest overall probabilities.

Updated sequences:

"The cat is" (Probability: 0.6)

"The cat sat" (Probability: 0.3)

"The cat runs" (Probability: 0.1)

Step 2:

Now, for each of the updated sequences, the model calculates the probabilities of all possible next words given their respective contexts.

The top three words and their probabilities for each sequence might be:

For "The cat is":

"sleeping": 0.7

"eating": 0.2

"running": 0.1

For "The cat sat":

"on": 0.5

"next": 0.4

"down": 0.1

For "The cat runs":

"fast": 0.6

"slowly": 0.3

"away": 0.1

The model updates the beam with the top three sequences with the highest overall probabilities.

Updated sequences:

"The cat is sleeping" (Probability: 0.6 * 0.7 = 0.42)

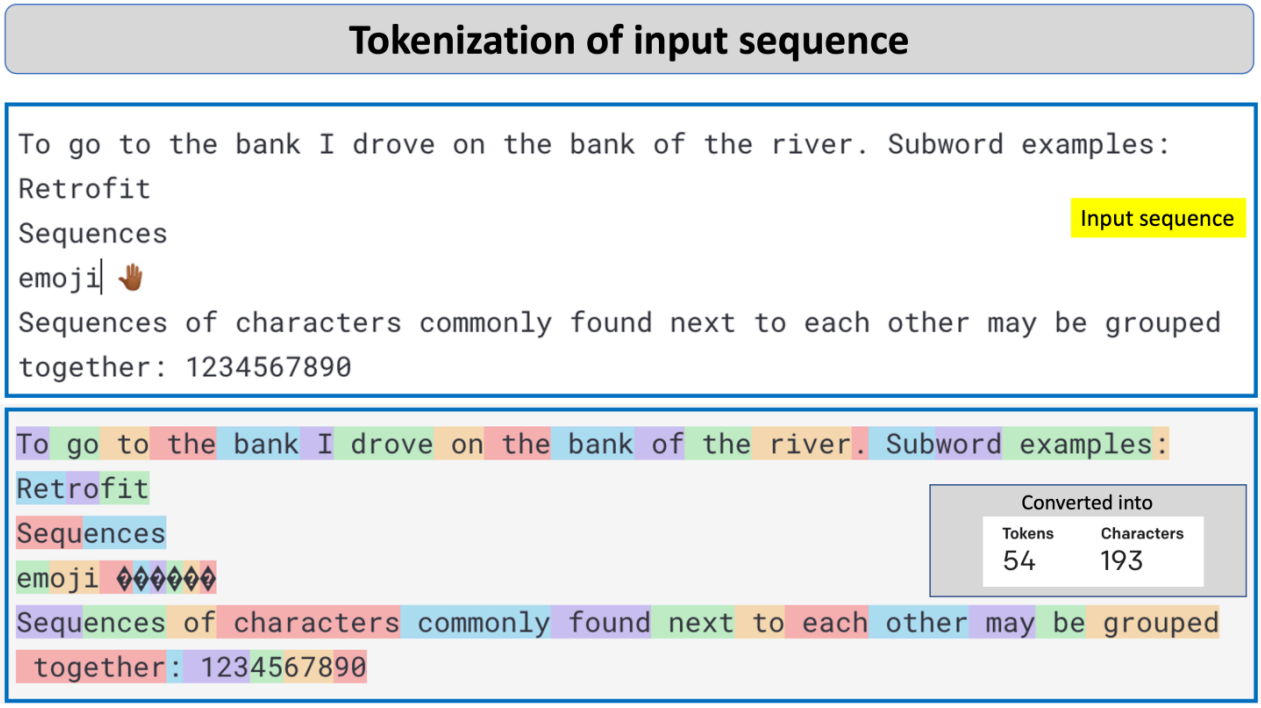
"The cat is eating" (Probability: 0.6 * 0.2 = 0.12)

"The cat is running" (Probability: 0.6 * 0.1 = 0.06)

Step 3: The process continues until a predefined stopping criterion is met (e.g., reaching a maximum sentence length or generating an end-of-sequence token).

Tokenization, vocabulary, and embedding

```
display_image('/content/tokenization.png')
```




```
Start coding or generate with AI.
```

When processing input text, GPT-3 first converts the text into tokens, To do this, GPT-3 uses the concept of words, sub-words, and punctuation.

Words are familiar units of meaning, like "cat" or "dog," but GPT-3 also uses sub-words to break longer words into smaller units.

For example, the word "unbelievable" might be broken down into the sub-words "un," "believe," and "able." This helps the model process a larger vocabulary of words.

Subword-based tokenization

Subword-based tokenization is a solution between word and character-based tokenization. 😎
The main idea is to solve the issues faced by word-based tokenization (very large vocabulary size, large number of OOV tokens, and different meaning of very similar words) and character-based tokenization (very long sequences and less meaningful individual tokens).

Word-based Tokenization:

If we were to use word-based tokenization, each unique word would be treated as a separate token. However, in scientific texts, we often encounter compound words or specialized terms that may not be present in a standard vocabulary. For example, terms like "deoxyribonucleic acid" or "gluconeogenesis" are compound words that may not be included in a typical vocabulary. Treating these compound words as single tokens would lead to a large vocabulary size, making it challenging for the model to learn representations for all possible terms.

Character-based Tokenization:

Character-based tokenization breaks down text into individual characters, which can lead to very long sequences, especially for longer words or compound terms. While character-based tokenization captures the granularity of individual characters, it may not effectively capture the meaningful subunits within compound words or specialized terms. For example, "deoxyribonucleic acid" would be tokenized as ["d", "e", "o", "x", "y", "r", ...], resulting in a long sequence of characters that may not provide meaningful context for the model to learn from.

Subword-based Tokenization:

Subword-based tokenization strikes a balance between word-based and character-based tokenization. It breaks down words into smaller subunits, capturing meaningful parts of compound words or specialized terms. For example, "deoxyribonucleic acid" might be tokenized into subwords like ["deoxy", "ribonucleic", "acid"], which provide more meaningful units for the model to learn from compared to individual characters. Similarly, chemical compound names like "aspirin" might be tokenized into subwords like ["asp", "ir", "in"], capturing meaningful parts of the term.

Large Vocabulary Size:

In word-based tokenization, each unique word is treated as a separate token. This can lead to a large vocabulary size, especially in languages with rich morphology or extensive vocabularies. Subword-based tokenization breaks down words into smaller units called subwords. By doing so, it reduces the number of unique tokens in the vocabulary. For example, instead of having separate tokens for "running," "ran," and "runs," all these words might share the same subword "run." This reduces the vocabulary size and makes it more manageable for the model.

Out-of-Vocabulary (OOV) Tokens:

Words that are not present in the vocabulary are represented as out-of-vocabulary (OOV) tokens. This often occurs with rare or unseen words, which can negatively impact model performance. Subword-based tokenization allows the model to handle unseen words more gracefully. Since subwords are smaller units that are learned based on the training data, they are more likely to capture meaningful parts of unseen words. Even if a word is unseen during training, the model may still be able to represent it using its constituent subwords, leading to fewer OOV tokens and better overall performance.

For Example:-

In a word-based tokenization or character-based tokenization approach, if the word "university" does not appear frequently in the training data or is absent from the vocabulary, it would be treated as an out-of-vocabulary (OOV) token. The model may struggle to handle unseen words like "university" gracefully because it lacks a specific representation for this word.

However, with subword-based tokenization, the word "university" can be broken down into smaller subword units, such as "uni" and "versity." These subwords are more likely to appear in the training data and are learned as part of the vocabulary during training.

So, even if the model has not explicitly seen the word "university" during training, it can still represent it using its constituent subwords. For example, the model may have learned representations for "uni" and "versity" separately. When encountering the unseen word "university" during inference or testing, the model can combine the representations of its subwords to understand its meaning and context.

This ability to handle unseen words more gracefully through subword-based tokenization reduces the reliance on out-of-vocabulary tokens and improves the overall performance of the model, especially when dealing with rare or infrequently occurring words in the input data.

Variations in Word Forms:

Words with similar meanings may have different forms, such as verb conjugations ("run," "running," "ran") or plural forms ("cat," "cats"). Subword-based tokenization addresses this issue by capturing meaningful parts of words as subwords. Similar words with different forms are

likely to share common subwords. For example, the words "run," "running," and "ran" may all share the subword "run." This allows the model to generalize across different word forms more effectively and capture similarities in meaning.

The subword-based tokenization algorithms do not split the frequently used words into smaller subwords. It rather splits the rare words into smaller meaningful subwords. For example, “boy” is not split but “boys” is split into “boy” and “s”. This helps the model learn that the word “boys” is formed using the word “boy” with slightly different meanings but the same root word.

Some of the popular subword tokenization algorithms are **WordPiece, Byte-Pair Encoding (BPE), Unigram, and SentencePiece.** .BPE is used in language models like GPT-2, RoBERTa, XLM, FlauBERT, etc. A few of these models use space tokenization as the pre-tokenization method while a few use more advanced pre-tokenization methods provided by Moses, spaCY, ftfy.

See additional examples in the above fig Punctuation marks are also treated as separate tokens. This allows GPT-3 to understand sentence structure and other grammatical conventions.

Byte-Pair Encoding (BPE)

BPE is a simple form of data compression algorithm in which the most common pair of consecutive bytes of data is replaced with a byte that does not occur in that data. It was first described in the article “A New Algorithm for Data Compression” published in 1994. The below example will explain BPE and has been taken from Wikipedia.

Suppose we have data aaabdaaabac which needs to be encoded (compressed). The byte pair aa occurs most often, so we will replace it with Z as Z does not occur in our data. So we now have ZabdZabac where Z = aa. The next common byte pair is ab so let’s replace it with Y. We now have ZYdZYac where Z = aa and Y = ab. The only byte pair left is ac which appears as just one so we will not encode it. We can use recursive byte pair encoding to encode ZY as X. Our data has now transformed into XdXac where X = ZY, Y = ab, and Z = aa. It cannot be further compressed as there are no byte pairs appearing more than once. We decompress the data by performing replacements in reverse order.

Imagine you have a text document containing the following sentences:

"The quick brown fox jumps over the lazy dog."

You want to compress this text using BPE.

- Identify Repeated Byte Pairs:

  Look for repeated pairs of characters in the text. In our example, let's say the most common repeated pair is "th".

- Replace Repeated Byte Pairs:

  Replace the most common repeated pair with a new symbol that doesn't exist in the text. Let's use "$" for "th". So, our compressed text becomes:

"The quick brown fox jumps over $e lazy dog."

- Repeat the Process:

  Continue identifying and replacing repeated pairs of characters in the compressed text. Let's say the next most common pair is "er". Replace it with another new symbol, let's use "#" for "er". So, our compressed text becomes:

"The quick brown fox jumps ov# $e lazy dog."

- Repeat Until No More Replacements Can Be Made: Keep repeating the process until no more repeated pairs of characters can be found. In our example, there are no more repeated pairs.

- Decompression:

  To decompress the text, reverse the replacements. Replace the symbols with their corresponding original pairs of characters. For example, replace "$" with "th" and "#" with "er". So, our decompressed text becomes:

"The quick brown fox jumps over the lazy dog."

A variant of this is used in NLP. Let us understand the NLP version of it together. 🤗

BPE ensures that the most common words are represented in the vocabulary as a single token while the rare words are broken down into two or more subword tokens and this is in agreement with what a subword-based tokenization algorithm does.

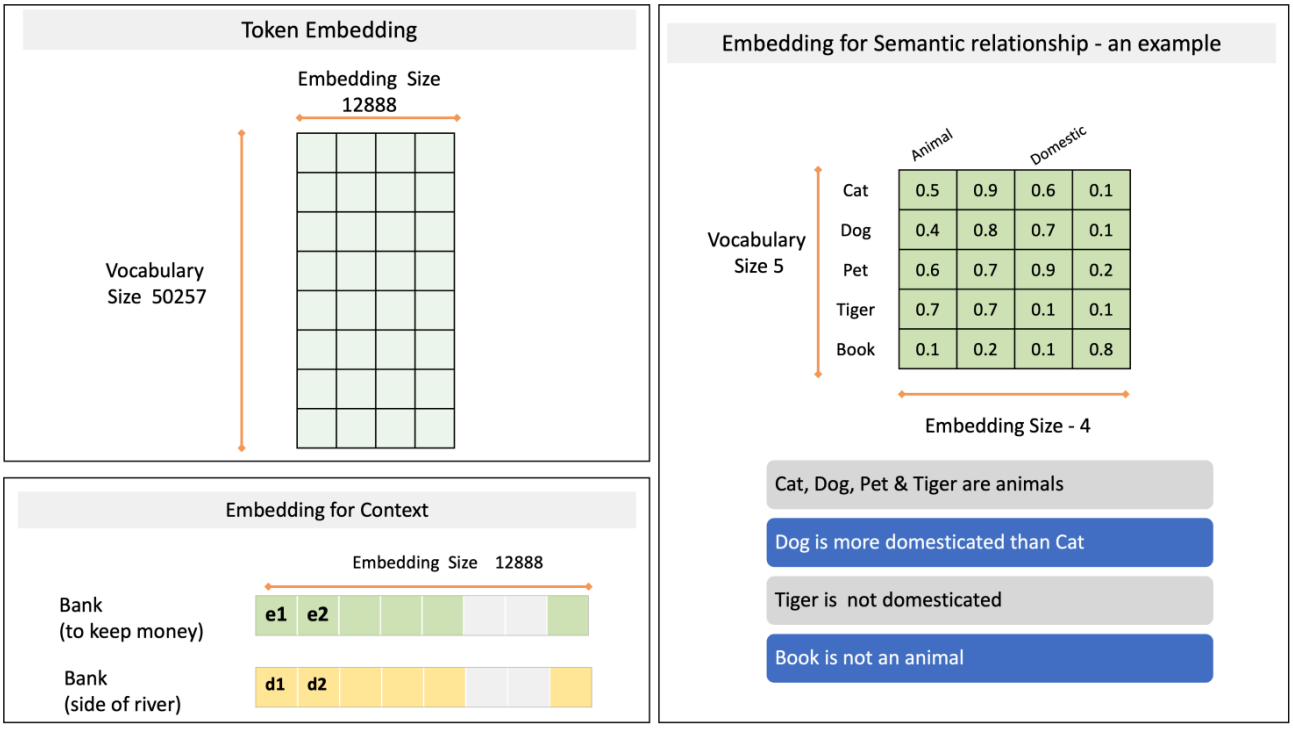
```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
display_image('/content/GPTEmbedding.png')
```



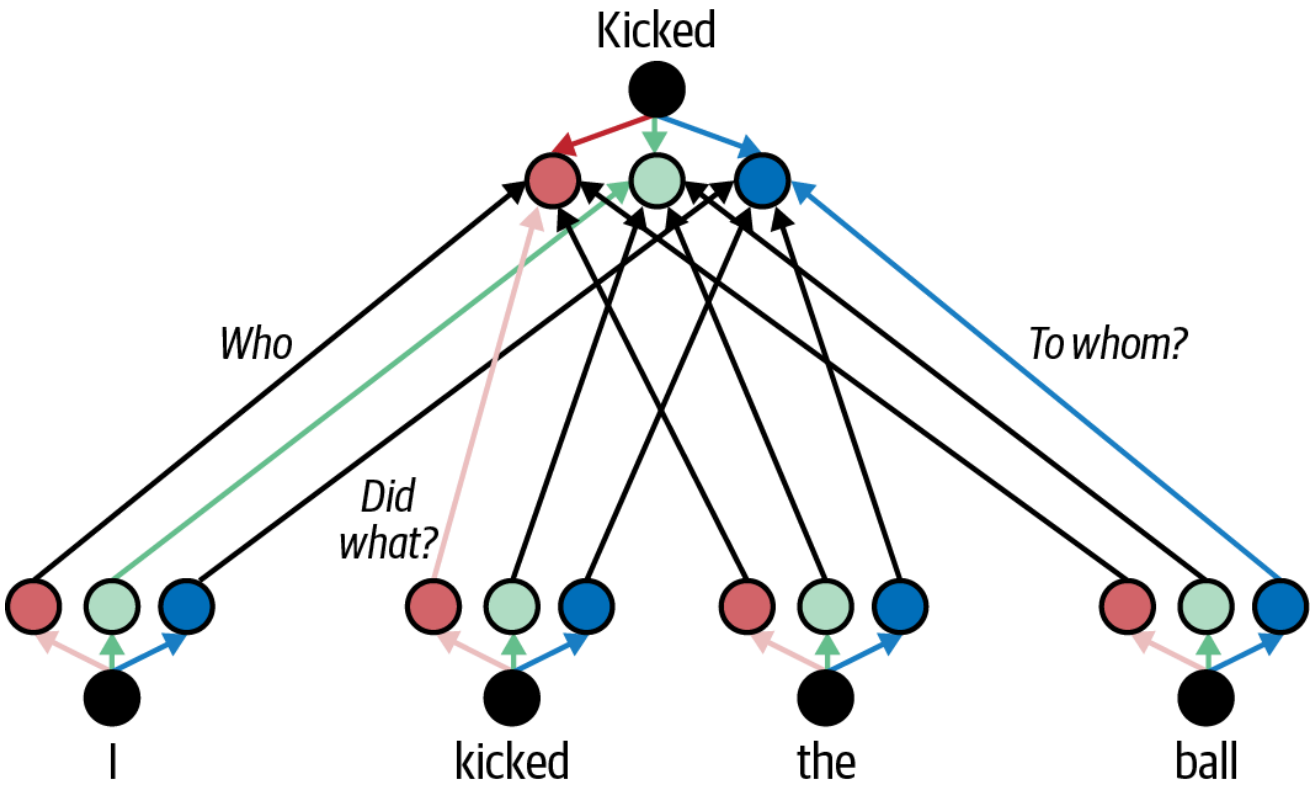
- GPT-3 has a vocabulary of 50,257 tokens, and each token is represented by a vector of 12888 elements.

- This vector representation is called an embedding, an important concept, that helps to capture the semantic meanings of the tokens.

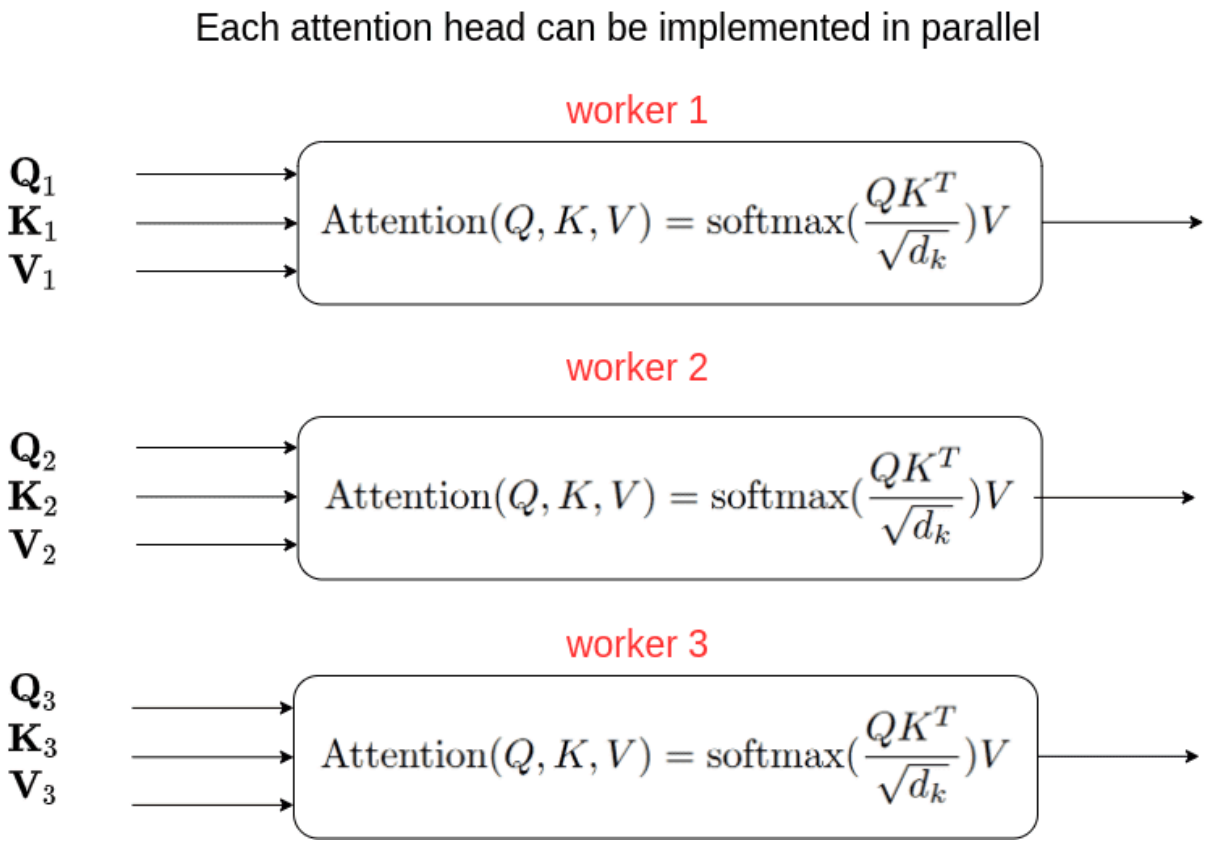
```
Start coding or generate with AI.
```

Multi Head Attention Mechanism

```
display_image('/content/multiheadattention.png')
```



```
display_image('/content/multi-head-self-attention-block-diagram.png')
```



Multi-head attention is a mechanism used in neural network architectures, particularly in sequence-to-sequence tasks like machine translation or text generation. It allows the model to focus on different parts of the input sequence simultaneously, enabling it to capture relevant information more effectively.

Let's elaborate on multi-head attention with an example:

Consider a machine translation task where we want to translate a sentence from English to French. We have an encoder-decoder architecture, and during the decoding phase, the decoder needs to generate each word of the translated sentence. Multi-head attention comes into play in the decoder to help it focus on different parts of the input (encoded source sentence) when generating each word of the output (translated sentence).

Here's how multi-head attention works:

Input Sequences:

- Let's say our input English sentence is: "The cat sat on the mat." Each word in the input sentence is represented as a vector after passing through the encoder.

- *Self-Attention:**
- In the decoder, when generating the first word of the translation, the model performs self-attention over the encoded input sequence. For instance, when generating the first word of the translation (e.g., "Le"), the model needs to pay attention to relevant words in the input sentence ("The cat sat on the mat") that help in translating "Le". The self-attention mechanism computes attention scores for each word in the input sentence with respect to the current word being generated.

- *Multi-Head Attention:**
- Multi-head attention enhances the capability of self-attention by allowing the model to consider different aspects of the input sequence simultaneously.

- The decoder performs multiple parallel self-attention operations, each focusing on a different aspect of the input sequence.

- For example, one attention head might focus on the subject ("cat"), another on the verb ("sat"), and another on the object ("mat"). Each attention head computes its own set of attention scores, capturing different aspects of the context. Integration:
- The attention scores from all the heads are combined, typically through concatenation or averaging, to create a comprehensive representation of the input sequence's context.

- This integrated context vector is then used by subsequent layers in the decoder, such as the position-wise feedforward networks, to generate the next word in the translation.

In summary, multi-head attention allows the decoder to focus on different parts of the input sequence simultaneously, capturing diverse aspects of the context and improving the quality of the generated output. It enhances the model's ability to incorporate relevant information from the input sequence when generating each word of the output sequence in tasks like machine translation.

```
Start coding or generate with AI.
```

Training

- GPT-3 was trained with over 175 billion parameters or weights. Engineers trained it on over 45 terabytes of data from sources like web texts, Common Crawl, books, and Wikipedia. Prior to training, the average quality of the datasets was improved as the model matured from version 1 to version 3.

- GPT-3 trained in a semi-supervised mode. First, machine learning engineers fed the deep learning model with the unlabeled training data. GPT-3 would understand the sentences, break them down, and reconstruct them into new sentences. In unsupervised training, GPT-3 attempted to produce accurate and realistic results by itself. Then, machine learning engineers would fine-tune the results in supervised training, a process known as reinforcement learning with human feedback (RLHF).

- **Unsupervised Pre-training:**

In unsupervised learning, the model is trained on data without explicit labels or annotations. The goal is to uncover patterns, structures, or relationships within the data without any guidance from external sources.

Examples of unsupervised learning tasks include clustering (grouping similar data points together), dimensionality reduction (reducing the number of features while preserving important information), and density estimation (estimating the probability distribution of the data).

In unsupervised learning, the training signal comes solely from the data itself, and the model learns to find meaningful representations or structures in the absence of explicit supervision.

- This means that the training signal is derived from the patterns, similarities, or structures present in the input data.

- For example, in clustering algorithms, the training signal comes from the similarity or dissimilarity between data points, and the model learns to group similar data points together. In dimensionality reduction techniques like Principal Component Analysis (PCA), the training signal comes from the covariance structure of the data, and the model learns to capture the most significant directions of variation.

- **Self Supervised Learning**

In self-supervised learning, the model generates its own supervisory signal from the input data. This means that instead of relying on external labels or annotations provided by humans, the model creates its own training objectives or tasks based on the structure of the data it's trained on.

Example Tasks:

Self-supervised learning tasks are designed to leverage the inherent structure or patterns in the data. Examples include:

Language Modeling: Predicting the next word in a sequence given the preceding context. For example, given the sentence "The cat is on the...", the model predicts the next word ("mat") based on the context provided.

Image or Text Reconstruction: Reconstructing corrupted or masked portions of an image or text. For instance, given an image with a portion missing or obscured, the model learns to fill in the missing parts.

Pretext Tasks: Tasks like rotation prediction or colorization, where the model learns representations by solving auxiliary tasks. For example, the model might be trained to predict the orientation of an image patch after it has been rotated.

- **Fine-tuning with Supervised Data:**

After unsupervised pre-training, the model may undergo further training using labeled (supervised) data for specific downstream tasks.

This fine-tuning process involves adjusting the parameters of the pre-trained model to perform well on the target task.

For example, GPT-3 can be fine-tuned on labeled datasets for tasks such as text classification, question answering, or text generation. During fine-tuning, the model learns task-specific patterns and nuances from the labeled examples.

- **Prompt-based Learning:**

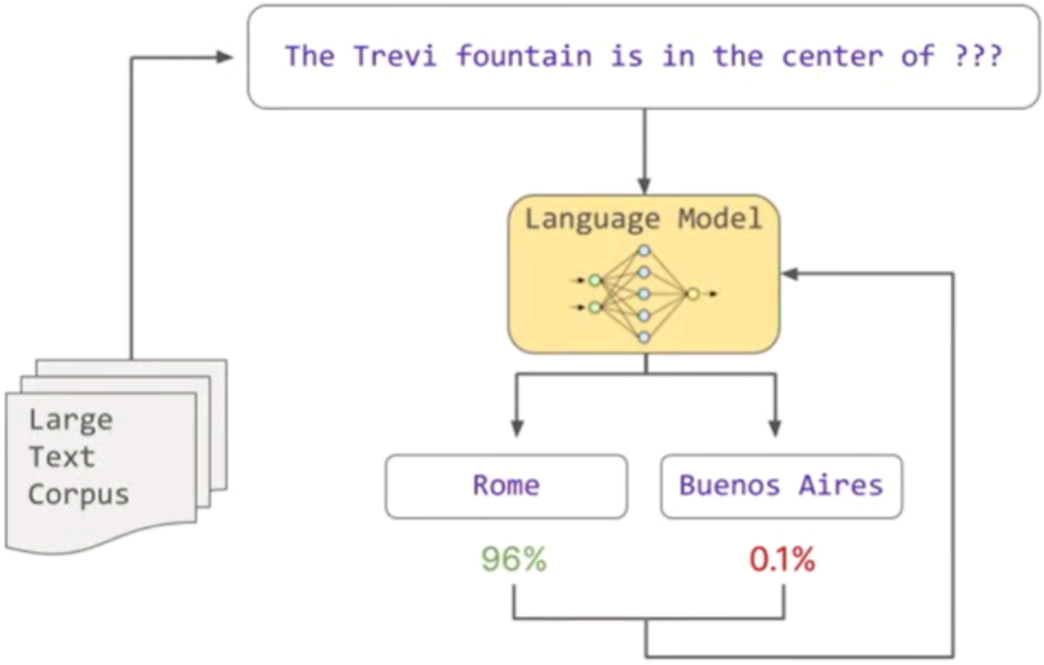
In addition to unsupervised pre-training and supervised fine-tuning, GPT-3 also benefits from prompt-based learning. In this approach, the model is provided with specific prompts or input-output pairs to guide its generation process. By conditioning the model on prompts and desired outputs, users can steer GPT-3 to produce responses that are relevant to particular tasks or contexts

```
Start coding or generate with AI.
```

Double-click (or enter) to edit

What is self supervised learning

```
display_image('/content/SELFSUPERVISEDLEARNING.png')
```



Self-supervised learning is a type of machine learning paradigm where a model learns to make predictions about some aspect of its input data without explicit supervision. Instead of relying on labeled data provided by humans, the model generates its own supervision signal from the input data itself. The primary goal is to enable the model to learn useful representations or features of the input data that can be later used for downstream tasks.

Here's how self-supervised learning typically works:

Input Transformation:

- The input data is transformed in some way to create a supervised learning task. This transformation can involve techniques such as masking, permutation, or generating pretext

tasks. For example, in the context of natural language processing, words in a sentence might be randomly masked and the model is trained to predict the masked words.

Learning Representation:

- The model is then trained to predict the transformed or masked parts of the input data based on the unmasked parts. By doing so, the model learns to understand the underlying structure or semantics of the data.

Transfer Learning:

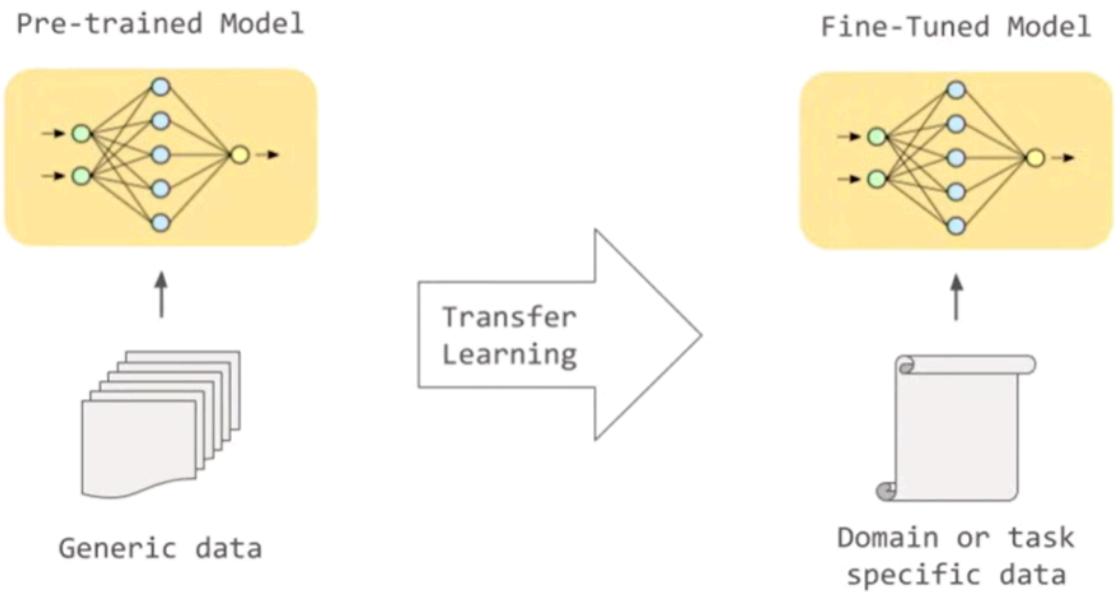
- Once the model has learned useful representations through self-supervised learning, these representations can be transferred to downstream tasks. The model is fine-tuned on a smaller labeled dataset for the specific task of interest, leveraging the learned representations to improve performance.

Self-supervised learning has gained popularity in recent years due to its ability to learn from vast amounts of unlabeled data, which is often readily available, especially in domains like natural language processing, computer vision, and speech recognition. It enables models to learn rich and meaningful representations of data, leading to better generalization and performance on downstream tasks.

Exploiting a Pretrained Model: Fine Tuning

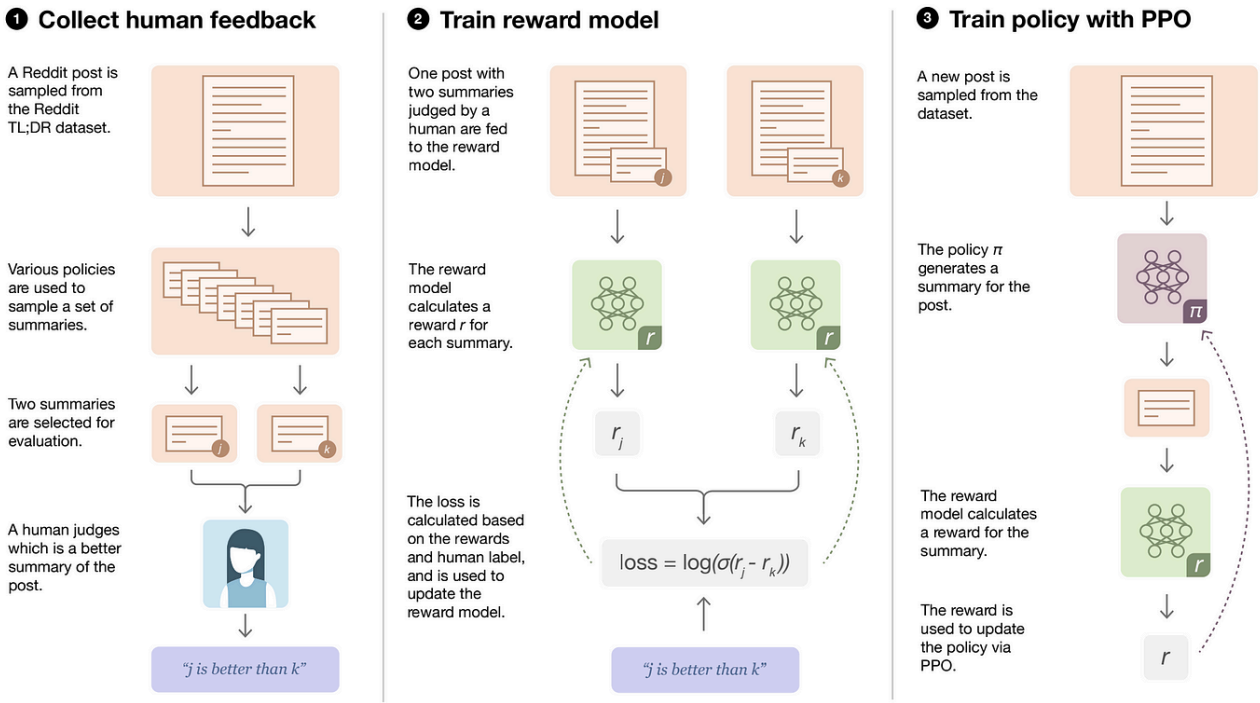
The pretrined model is finetuned to perform special task like translation,summarization,and many other task

```
display_image('/content/finetuning.png')
```



InstructGPT: Reinforcement learning with human feedback

```
display_image('/content/rlhf.png')
```



LLMs have shown remarkable capabilities in a wide range of NLP tasks. However, these models may sometimes exhibit unintended behaviors, e.g., fabricating false information, pursuing inaccurate objectives, and producing harmful, misleading, and biased expressions. For LLMs, the language modeling objective retrains the model parameters by word prediction while lacking the consideration of human values (helpful, honest, harmless) or preferences. To avert these unexpected behaviors, human alignment has been proposed to make LLMs act in line with human expectations.

InstructGPT is a technique developed by OpenAI to guide large language models (LLMs) like GPT in generating more accurate and appropriate outputs by incorporating human feedback. It uses a method called Reinforcement Learning with Human Feedback (RLHF) to train the model to follow human expectations.

Here's a simplified explanation of how InstructGPT works:

Supervised Fine-Tuning (SFT):

First, a team collects examples of desired behavior from humans. These examples show how the model should respond to different inputs. Then, they train the model on this data using supervised learning, adjusting its parameters to match the desired behavior.

Reward Model Training (RM):

Next, the team collects examples where humans compare different model outputs and indicate which they prefer. They train a reward model to predict which outputs humans are likely to prefer based on these comparisons.

Reinforcement Learning Fine-Tuning (RL):

The model is then fine-tuned using reinforcement learning, where it learns to maximize the reward predicted by the reward model. This helps the model generate outputs that are more aligned with human preferences.

In summary, InstructGPT uses a combination of supervised learning, reward modeling, and reinforcement learning to train large language models to generate outputs that better match human expectations.

```
Start coding or generate with AI.
```

Example

Let's consider the scenario of teaching a pet to perform tricks, like fetching a ball.

- Supervised Fine-Tuning (SFT):

Initially, you demonstrate to your pet how to fetch a ball. You show them how to chase the ball, pick it up with their mouth, and bring it back to you. Your pet learns from these demonstrations and practices, gradually improving its ability to fetch the ball accurately.

- Reward Model Training (RM):

Next, you ask a group of friends to observe your pet's fetching behavior and provide feedback on which attempts they prefer. Some attempts may be faster, more accurate, or more enthusiastic. Based on this feedback, you develop a sense of which fetching behaviors are most desirable. You then train a reward model to predict which behaviors your friends would prefer in different situations.

- Reinforcement Learning Fine-Tuning (RL):

Using the reward model, you further refine your pet's fetching behavior. You set up a system where your pet receives a treat or praise whenever it successfully fetches the ball in a preferred manner, according to the predictions of the reward model. Over time, your pet learns to adjust its fetching technique to maximize the rewards it receives. It becomes more consistent in its behavior and starts fetching the ball in a way that closely aligns with what humans prefer.

```
Start coding or generate with AI.
```

What is Reinforcement Learning

- Reinforcement learning (RL) is a type of machine learning paradigm where an agent learns to interact with an environment in order to maximize some notion of cumulative reward. It differs from supervised learning in that the agent learns from trial and error, receiving feedback in the form of rewards or penalties for its actions rather than explicit instructions.
- In RL, the agent takes actions in an environment, and based on those actions, the environment provides feedback in the form of rewards or penalties. The agent's objective is to learn a policy, a mapping from states to actions, that maximizes the cumulative reward over time.

PPO

PPO (Proximal Policy Optimization):

- PPO is a reinforcement learning algorithm designed to train agents to perform tasks in environments where feedback is received through interactions with the environment itself.

- It belongs to the class of policy gradient methods and aims to optimize the policy of the agent to maximize expected rewards.

- PPO specifically addresses the problem of ensuring stable and efficient learning by constraining the policy update steps to prevent large policy changes that could destabilize learning.

- PPO has been widely used in various reinforcement learning applications, including game playing, robotics, and autonomous systems.

```
Start coding or generate with AI.
```

Imagine you're playing a video game where you control a character (the agent) trying to reach a goal while facing obstacles (the environment). The way your character behaves in response to what it sees around it is its "policy."

Now, in this game, you want your character to learn how to play better over time. Traditional methods would be like giving your character a bunch of rules to follow, which might not always work well.

PPO is like a smarter way for your character to learn. Instead of rigid rules, it learns by trial and error, adjusting its actions based on the outcomes it experiences. Here's how it works:

- Learning from Experience:

  Your character tries different actions in different situations and sees what happens. If an action leads to a good outcome (like reaching the goal), it learns to do more of that. If it leads to a bad outcome (like getting stuck), it learns to avoid that action in that situation.

- Balancing Exploration and Exploitation:

  PPO balances between trying new things (exploration) and sticking with what it knows works (exploitation). It doesn't want to keep trying random stuff forever, but it also doesn't want to get stuck in a rut.

- Keeping Things Steady:

  PPO doesn't make huge changes to its behavior all at once. It adjusts its actions gradually, making sure that it doesn't change too much too quickly. This helps in learning without causing chaos.

- Using Feedback to Improve:

  PPO also listens to feedback from the environment. If it gets a reward (like points in the

game), it learns to associate its actions with those rewards. This helps it figure out what works and what doesn't.

So, in simpler terms, PPO is like a method for your game character to learn how to play better by trying things out, learning from the outcomes, and making gradual improvements over time. It's like learning from trial and error, but with a smart strategy to guide the learning process.

PPO vs RLHF

PPO (Proximal Policy Optimization):

- PPO is like a teacher training a student to perform a task, like playing a game or driving a car, by giving feedback on their actions.
- The student (agent) tries different actions in an environment and receives feedback (rewards) based on how well they're doing.
- PPO helps the student learn the best way to act in different situations to get the highest rewards.
- It's designed to make sure the student learns steadily and efficiently without making big, risky changes too quickly.

RLHF (Reinforcement Learning from Human Feedback):

- RLHF is like having a coach or mentor guiding the student's learning process. In addition to feedback from the environment, the student also gets input from the coach (human) on how to improve.
- The coach might provide different types of feedback, like rewards for good performance, demonstrations of the right way to do things, or tips on how to improve.
- RLHF helps the student learn faster and more effectively by combining feedback from both the environment and the coach.

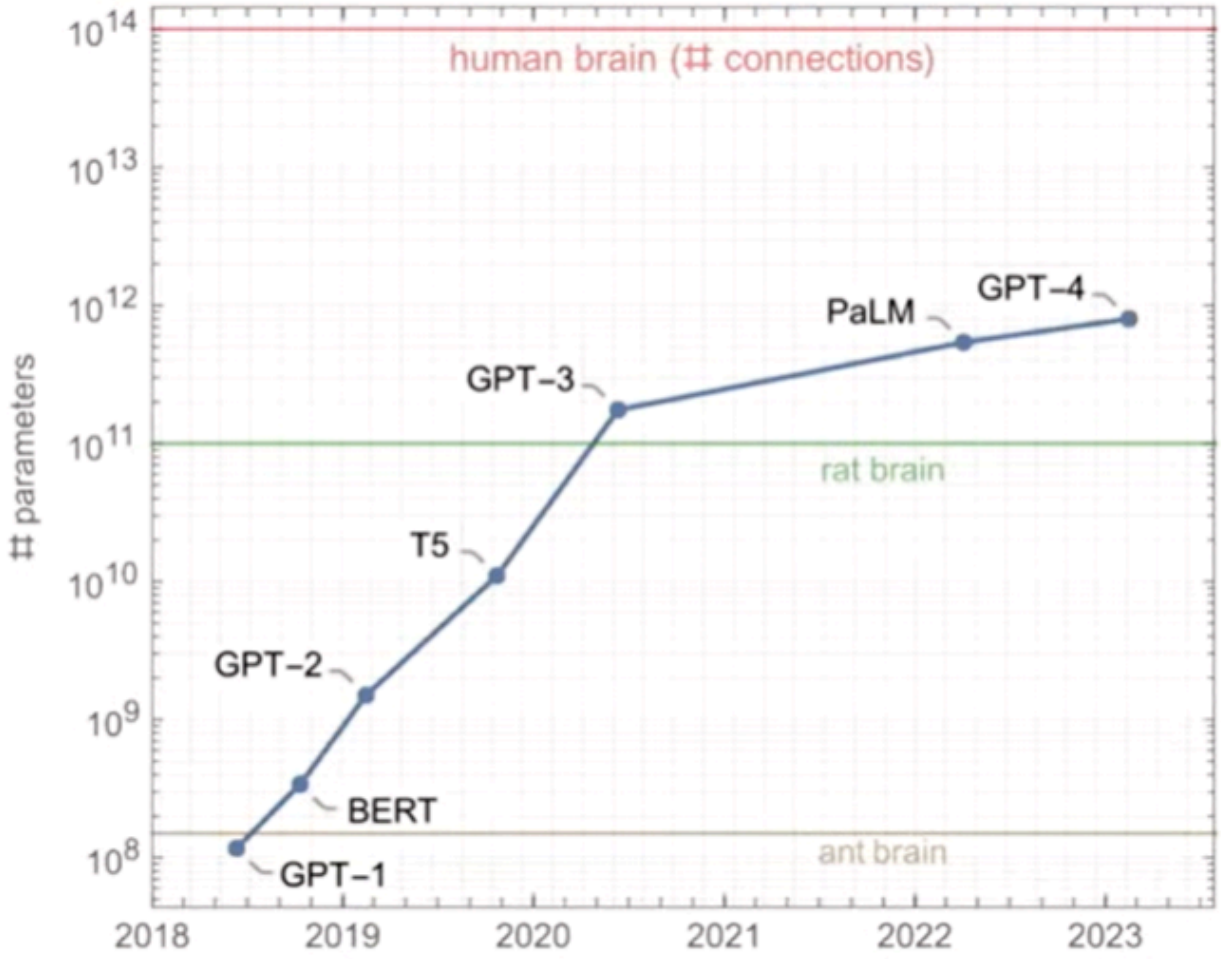
In simpler terms, PPO focuses on training an agent to learn from its interactions with the environment, while RLHF adds human guidance to the learning process to help the agent learn more efficiently. While PPO is a specific algorithm for reinforcement learning, RLHF is a broader

framework that can be applied alongside various reinforcement learning algorithms, including PPO, to incorporate human feedback into the learning process.

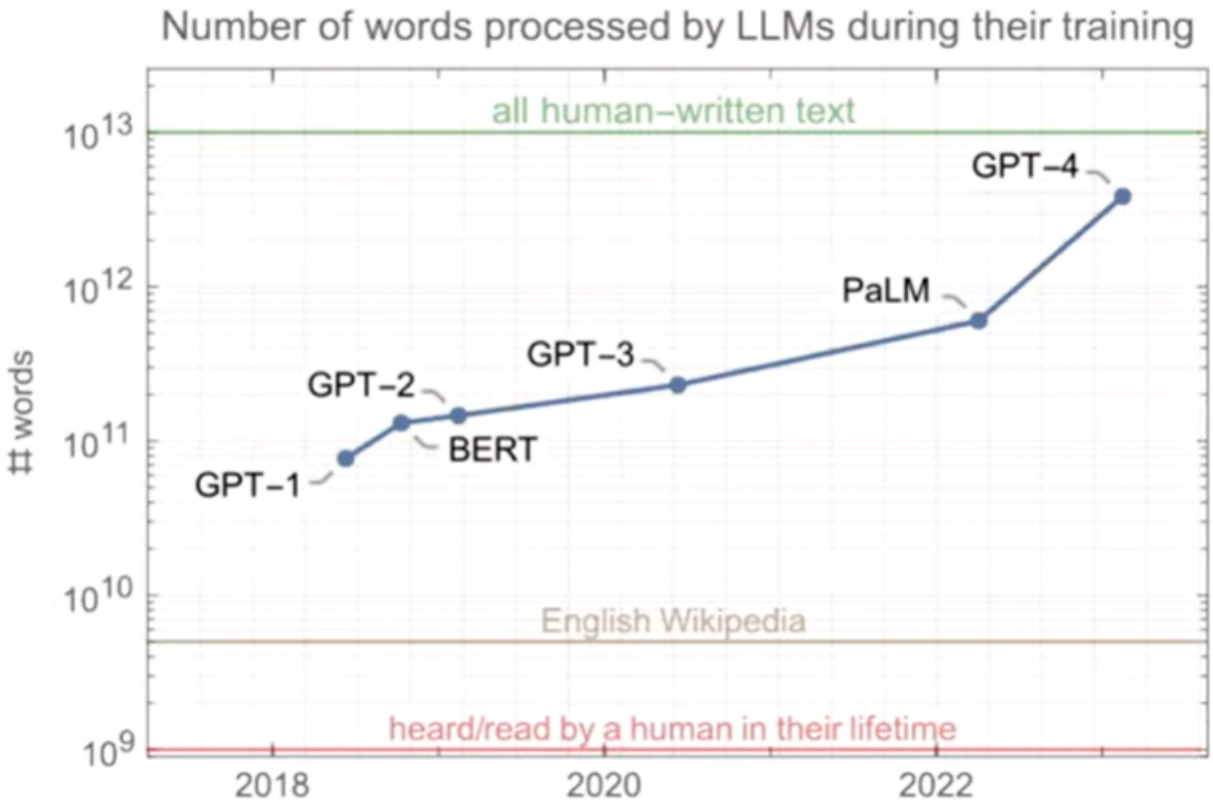
```
Start coding or generate with AI.
```

From Language Model to Large Language Model

```
display_image('/content/noofparameters.png')
```




```
display_image('/content/wordsprocessedbyllm.png')
```




```
display_image('/content/GPTTrainingPrices.png')
```

GPT Parameters

```
display_image('/content/GPT_Parameters.png')
```

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

- **n_params:**

This refers to the total number of parameters in the model, which represents the capacity

or complexity of the model. It includes all the weights and biases of the neural network.

- **n_layers:**

  This refers to the number of layers in the transformer architecture. Each layer consists of multiple sub-layers, typically including self-attention mechanisms and feed-forward neural networks.

- **d_model:**

  This refers to the dimensionality of the model's hidden states or embeddings. It represents the size of the vectors used to represent tokens or words in the model.

- **n_heads:**

  This refers to the number of attention heads in the multi-head attention mechanism used in the transformer architecture. More attention heads allow the model to attend to multiple parts of the input sequence simultaneously.

- **d_heads:**

  This refers to the dimensionality of the queries, keys, and values used in each attention head. Typically, d_heads is calculated as d_model divided by n_heads.

- **batch_size:**

  This refers to the number of input examples (sequences) processed in parallel during each training iteration. A larger batch size can lead to faster training but may require more memory.

- **learning_rate:**

  This refers to the rate at which the model's parameters are updated during training using techniques like gradient descent. It controls the size of the steps taken towards minimizing the loss function during optimization.

GPT Playground

- **Model**:

  Specifies which version of the GPT model you're using. In this case, it's GPT-3.5-turbo, which is a specific variant of the GPT-3.5 model optimized for performance.

- **Temperature**:

  Controls the randomness of the generated responses. A lower temperature produces more deterministic and conservative responses, while a higher temperature allows for more randomness and creativity in the responses.

- **Maximum Length**:

  Sets the maximum length of the generated response in terms of tokens. Tokens are essentially words or subwords in the model's vocabulary. This parameter ensures that the generated response doesn't exceed a certain length.

- **Stop Sequences**:

  Specifies sequences that, when generated, will cause the model to stop generating further text. For example, if "Stop Sequences" is set to "Enter sequence and press Tab," the model will stop generating text once it encounters that sequence.

- **Top P**:

  Also known as nucleus sampling, this parameter controls the diversity of the generated responses by dynamically adjusting the probability distribution of words based on their likelihood of occurrence. A lower value results in a more focused response, while a higher value allows for more diverse responses.

- **Frequency Penalty**:

  Penalizes words based on their frequency of occurrence in the model's training data. This encourages the model to use less common words and phrases in its responses.

penalty_threshold = 10

This ensures that extremely rare words are not penalized too harshly. For example, you might decide to only penalize words that appear more than 10 times in the training data:

- **Presence Penalty**:

  Penalizes words based on their presence in the previous parts of the generated response. This encourages the model to generate responses that are more diverse and avoid repeating the same phrases.

penalty_threshold = 5

Set the Penalty Threshold:

Decide on a threshold position beyond which words will start to be penalized. This ensures that words at the beginning of the generated text are not penalized. For example, you might decide to only penalize words that appear after the first 5 words:

Top P

Nucleus sampling, which is controlled by the Top P parameter, is a method used in generating text that dynamically adjusts the probability distribution of words based on their likelihood of occurrence in the model's vocabulary.

Imagine you're asking a language model to generate the next word in a sentence. The model has a vocabulary of thousands or even millions of words, each with its own probability of being the next word in the sequence. Top P sampling adjusts these probabilities dynamically to control the diversity of the generated responses.

Here's how it works:

- **Probability Calculation:**

  Initially, the model assigns probabilities to each word in its vocabulary based on their likelihood of occurrence given the context of the previous words in the sequence.

- **Sorting by Probability:**

  The model sorts these probabilities in descending order, placing the most likely words at the top.

- **Cumulative Probability:**

  The model calculates the cumulative probability of the sorted words until it exceeds a

certain threshold determined by the Top P parameter. This threshold represents the cumulative probability mass that the model wants to include in its sampling.

- **Word Selection:**

  The model selects words from the sorted list until the cumulative probability exceeds the threshold specified by the Top P parameter. These selected words form the "nucleus" of the probability distribution.

- **Sampling:**

  Finally, the model samples from this nucleus distribution to generate the next word in the sequence. The higher the Top P value, the larger the nucleus, resulting in more diverse and varied responses. Conversely, a lower Top P value leads to a smaller nucleus and more focused responses.

Let's illustrate this with an example using some fictional probabilities for the words "apple," "banana," "cherry," and "date":

Probability of "apple": 0.4

Probability of "banana": 0.3

Probability of "cherry": 0.2

Probability of "date": 0.1

Now, let's say we set the Top P parameter to 0.8. The cumulative probabilities are:

Cumulative Probability of "apple": 0.4

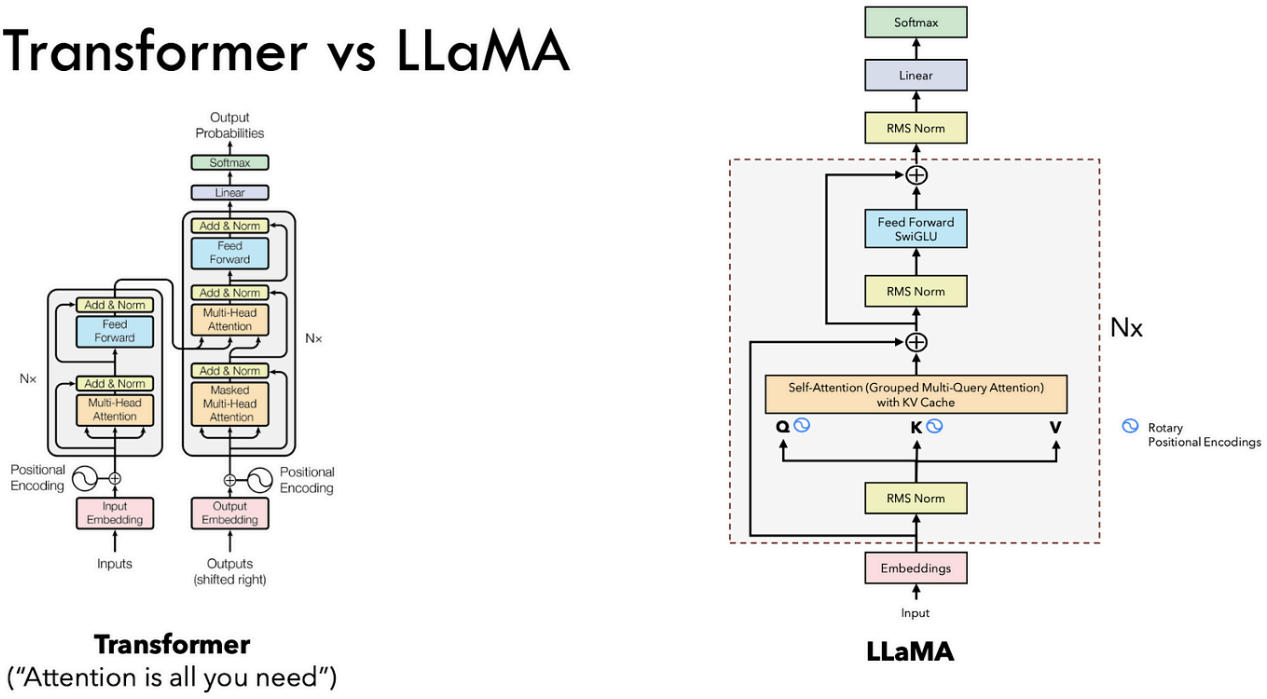
Cumulative Probability of "apple" + "banana": 0.7

Cumulative Probability of "apple" + "banana" + "cherry": 0.9

Cumulative Probability of "apple" + "banana" + "cherry" + "date": 1.0

With a Top P of 0.8, the model would select "apple," "banana," and "cherry" as the nucleus words, resulting in a diverse set of options for the next word.

```
display_image('/content/llama2 vs gpt.png')
```



LLaMA Model Features

RMS-Normalization:

- RMSNorm is a simplification of the original layer normalization (LayerNorm). LayerNorm is a regularization technique that might handle the internal covariate shift issue so as to stabilize the layer activations and improve model convergence. It has been proved quite successful in LLaMA 2.

- Imagine you're baking cookies, and you want each batch to come out just right. RMSNorm is like adjusting the oven temperature to make sure it stays consistent throughout baking.

- In neural networks, RMSNorm helps keep the activations (outputs) of each layer stable during training. It adjusts the scale of these activations based on the average size of the inputs. This helps prevent the activations from getting too big or too small, ensuring smoother and more reliable learning.

```
RMSNorm(x_i) = x_i / sqrt((1/n) * sum(x_i^2))

Where:

xi are the activations of the neurons.

n is the number of neurons.
```

```
x_1 = 3, x_2 = 4, x_3 = 5
RMSNorm(x_1) = 3 / sqrt((1/3) * (3^2 + 4^2 + 5^2))
RMSNorm(x_2) = 4 / sqrt((1/3) * (3^2 + 4^2 + 5^2))
RMSNorm(x_3) = 5 / sqrt((1/3) * (3^2 + 4^2 + 5^2))
```

Layer Normalization:

```
LayerNorm(x_i) = (x_i - mu) / sigma


Where:

μ is the mean of the activations.

σ is the standard deviation of the activations.


mu = 4, sigma = 1
LayerNorm(x_1) = (3 - 4) / 1
LayerNorm(x_2) = (4 - 4) / 1
LayerNorm(x_3) = (5 - 4) / 1
```

Activation Function

LLaMA 2 uses the SwiGLU activation function instead of ReLU, leading to improved training performance.

ReLU (Rectified Linear Unit):

ReLU is a widely used activation function in neural networks. It returns 0 for negative inputs and a linear (identity) function for positive inputs. ReLU helps to introduce non-linearity into the network and has been successful in training deep neural networks.

```
ELU(Exponential Linear Unit.)

f(x) = {
    x               if x >= 0
    alpha * (exp(x) - 1) if x < 0
}


ReLU

f(x) = max(0, x)
```

SwiGLU (Sigmoid-Weighted Linear Unit):

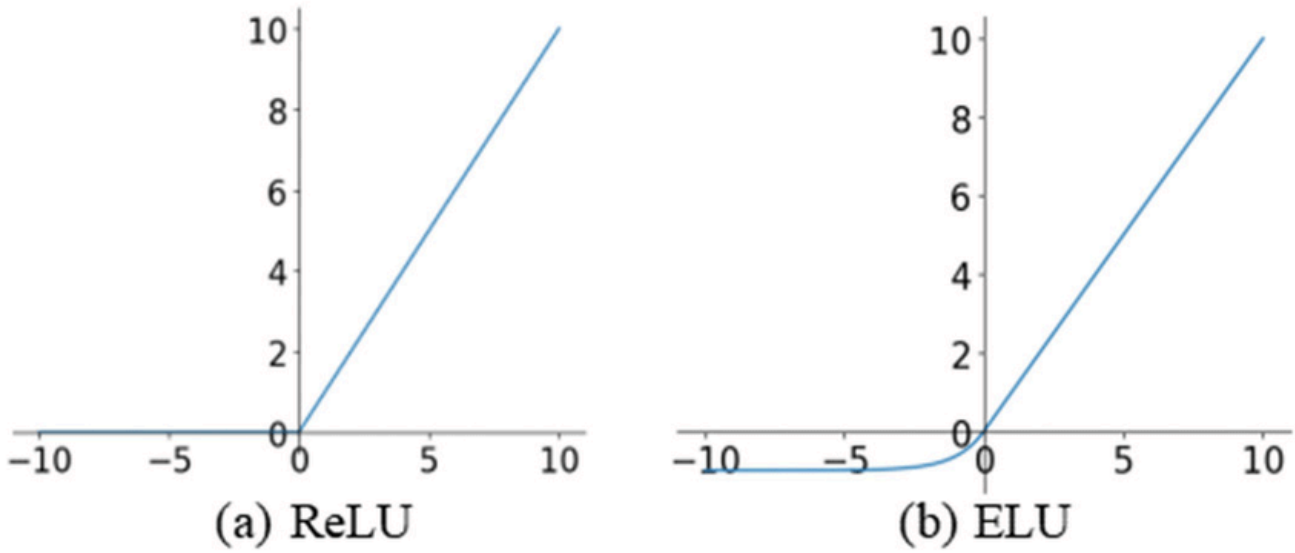
SwiGLU is a relatively new activation function that combines the properties of both Swish and GELU activation functions. It aims to address the drawbacks of GELU, such as vanishing gradients for large negative inputs, while retaining its smoothness and non-monotonic nature.

```
Swish
f(x) = x * sigmoid(beta * x)

SwiGLU
f(x) = (x * sigmoid(x)) + (x * tanh(x))
```

In LLaMA 2, SwiGLU is used as the activation function instead of ReLU. This choice was made because SwiGLU has been shown to improve training performance, convergence speed, and generalization capabilities in deep neural networks.

```
display_image('/content/relu elu.png')
```



Comparison of Activation Function:

- LLaMA 2: Uses SwiGLU activation function
- GPT-3: Uses GeLU activation function
- GPT, GPT-2: Use GELU activation function

GELU (Gaussian Error Linear Unit):

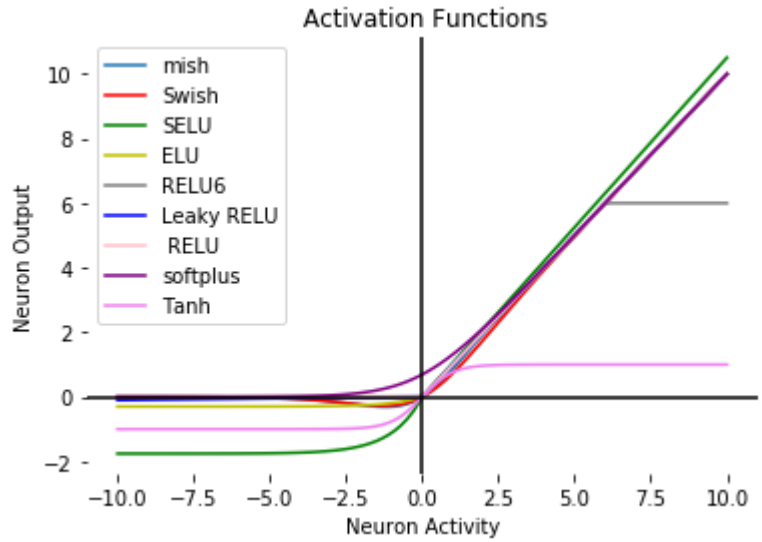
GELU is a smooth, non-monotonic activation function that uses the expected value of a rectified Gaussian distribution as its output. It's defined by the formula: GELU(x) = xΦ(x), where Φ(x) is the cumulative distribution function of the standard normal distribution. GELU is advantageous because it has non-zero derivatives everywhere, making it smooth and differentiable. However, it can encounter issues such as vanishing gradients for large negative inputs.

GeLU (Generalized Error Linear Unit):

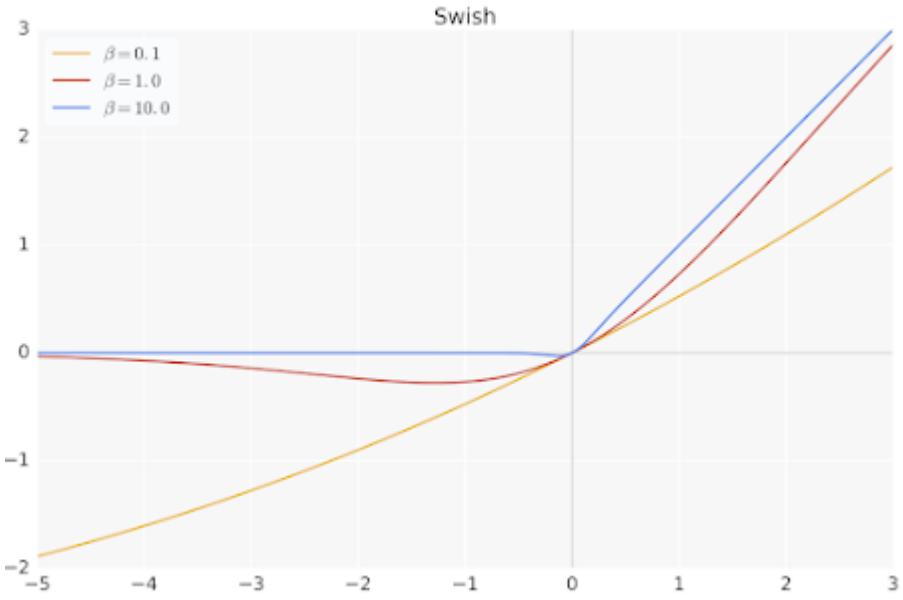
GeLU is a variant of GELU, introduced in GPT-3. It addresses GELU's vanishing gradient problem for large negative inputs while maintaining its smoothness and non-monotonic nature. GeLU introduces a hyperparameter that controls the degree of nonlinearity and smoothness of the activation function. The GeLU activation function is defined as GeLU(x, k) = (1 - k)GELU(x) + k(x/k + 1). Here, k is the hyperparameter that can be tuned to find the best setting for a given model.

```
f(x) = 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))
```

```
display_image('/content/differentactivation.png')
```




```
display_image('/content/swigluactivation.png')
```



Rotary Positional Embeddings (RoPE):

Inspired by the GPT-Neo-X project, LLaMA 2 incorporates rotary positional embeddings at each layer, enhancing the model's positional understanding.

Rotary positional embeddings (RoPE) are a type of positional encoding used in neural network architectures, particularly in transformer-based models like GPT-Neo-X and LLaMA 2.

- They enhance the model's understanding of the positional relationships between tokens in a sequence.
- Traditional positional encodings represent each position in a sequence with a fixed embedding vector. These embeddings are added to the token embeddings to provide

information about the order of tokens in the sequence. However, traditional positional encodings are static and do not change during the model's training or inference process.

- On the other hand, rotary positional embeddings introduce dynamic rotational transformations to the positional encodings. This means that the positional embeddings are modified during the training process based on the input data and the model's parameters.
- The term "rotary" is used because these positional embeddings undergo rotational transformations, which help the model learn more complex positional relationships. This dynamic adjustment allows the model to capture and encode positional information more effectively, potentially leading to improved performance in tasks that require understanding of sequence order.

Now, imagine a scenario where the language model encounters two different types of sequences during training:

- Regular Patterns:

In some sentences, the words follow predictable and regular patterns, where the positional relationships between tokens are consistent and stable. For example, in the sentence "**The cat sat on the mat,**" the order of words follows a straightforward pattern.

- Irregular Patterns:

In other sentences, the words may exhibit more complex and irregular patterns, where the positional relationships between tokens vary or are influenced by contextual factors. For example, in the sentence "**The mat sat on the cat,**" the order of words deviates from the usual pattern due to contextual nuances.

In such scenarios, traditional positional encodings may struggle to effectively capture the diverse positional relationships present in the data. This is where rotary positional embeddings come into play:

```
display_image('/content/rotatoryembedding.png')
```

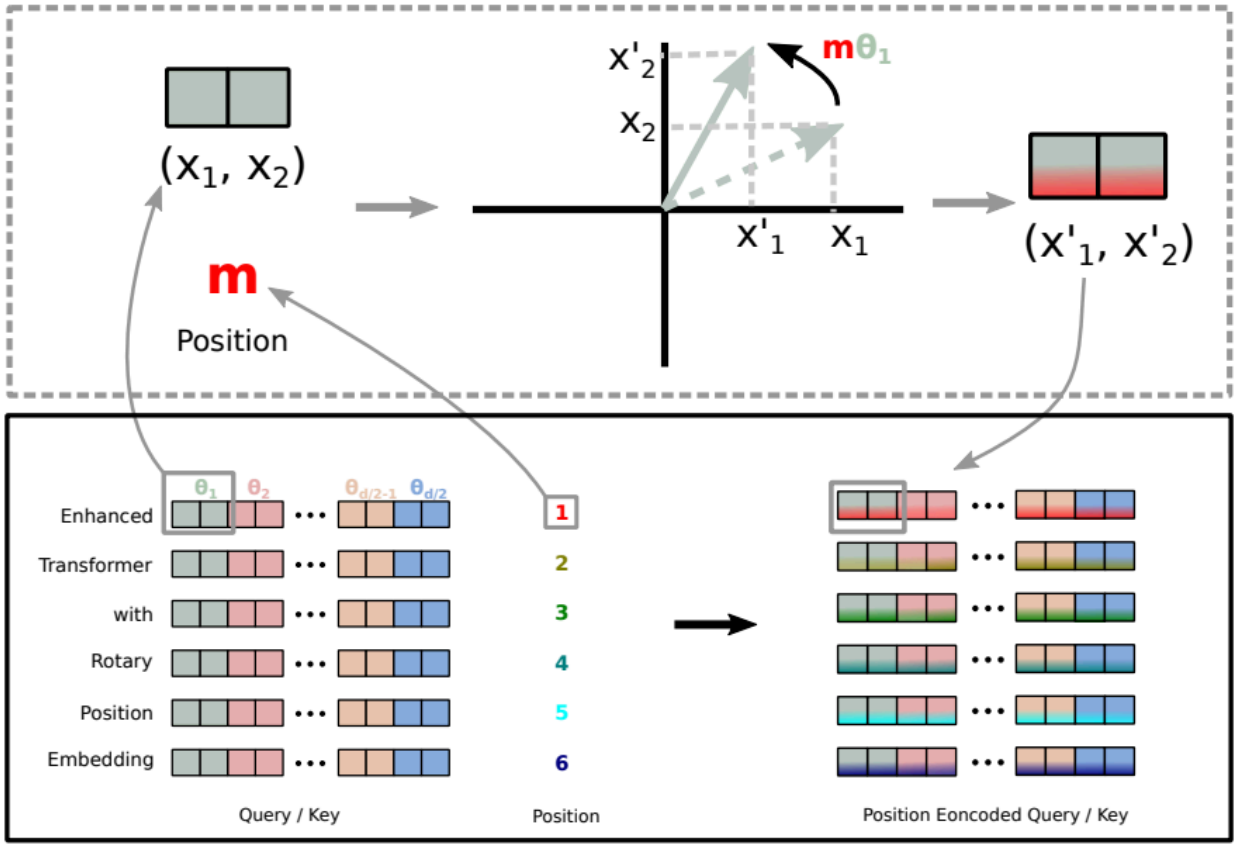


Figure 1: Implementation of Rotary Position Embedding(RoPE).

Increased Context Length and Grouped-Query Attention (GQA):

LLaMA 2 model has a doubled context window (from 2048 to 4096 tokens) and employs grouped-query attention. This allows for better processing of long documents, chat histories, and summarization tasks.

Context window (from 2048 to 4096 tokens)

Increased Context Length:

- The context length refers to the number of tokens or words that the model can consider at once when making predictions.

- In the LLaMA 2 model, the context window has been doubled from 2048 to 4096 tokens. This means that the model can now consider a larger context when processing text data.

Grouped-Query Attention (GQA):

- Attention mechanisms in transformer models allow the model to focus on different parts of the input sequence when making predictions.

- In grouped-query attention, instead of attending to all tokens in the sequence simultaneously, the attention mechanism divides the tokens into groups and attends to each group separately.

- This approach helps improve efficiency and scalability, especially when dealing with long sequences, by reducing the computational complexity of the attention mechanism.

- For example, suppose we have a sequence of 4096 tokens. Instead of attending to all tokens at once, the model may divide the tokens into smaller groups, such as groups of 512 tokens each. The attention mechanism would then attend to each group separately, allowing the model to process the entire sequence more efficiently.

```
display_image('/content/llama2training.webp')
```

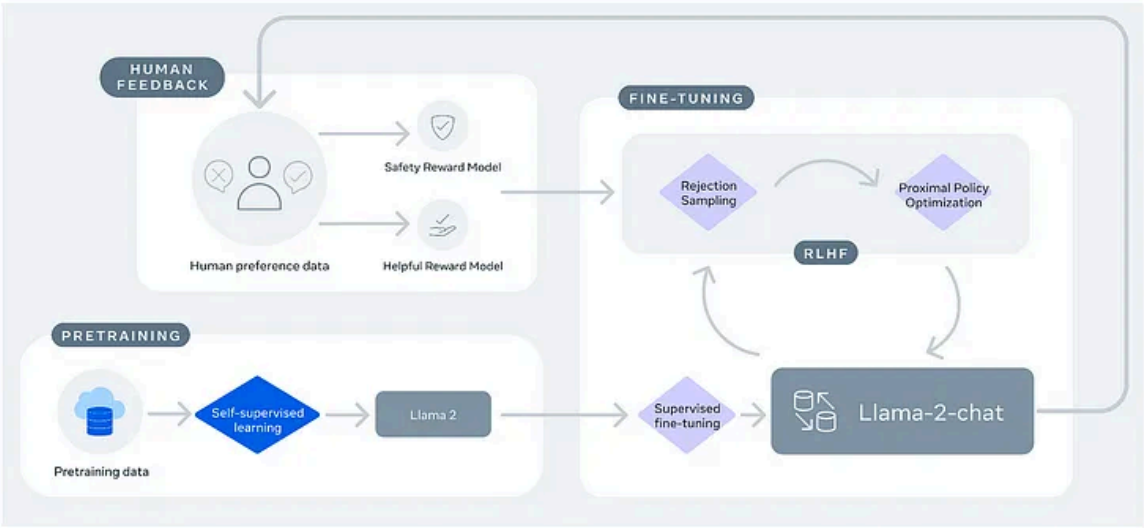


Figure 4: Training of Llama 2-Chat: This process begins with the **pretraining** of Llama 2 using publicly available online sources. Following this, we create an initial version of Llama 2-Chat through the application of **supervised fine-tuning**. Subsequently, the model is iteratively refined using Reinforcement Learning with Human Feedback (**RLHF**) methodologies, specifically through rejection sampling and Proximal Policy Optimization (PPO). Throughout the RLHF stage, the accumulation of **iterative reward modeling data** in parallel with model enhancements is crucial to ensure the reward models remain within distribution.

Rejection Sampling Fine-Tuning:

- In rejection sampling fine-tuning, the model generates multiple candidate outputs (samples) based on a given input or prompt.

- Each generated sample is then scored or evaluated using a reward function or quality metric.

- The sample with the highest score, or the best candidate, is selected as the reference for high-quality outputs.

- The model is fine-tuned using reinforcement learning techniques, such as gradient updates, based on the selected high-quality sample.

- The goal is to encourage the model to generate outputs similar to the selected sample, thereby improving the overall quality of generated outputs.

What are LLM Benchmarks and Why are They Important?

An LLM benchmark is a standardised performance test used to evaluate various capabilities of AI language models. A benchmark usually consists of a dataset, a collection of questions or tasks, and a scoring mechanism. After undergoing the benchmark's evaluation, models are usually awarded a score from 0 to 100.

ARC

- AI2 Reasoning Challenge (ARC) is a question-answer (QA) benchmark that's designed to test an LLM's knowledge and reasoning skills. ARC's dataset consists of 7787 four-option multiple-choice science questions that range from a 3rd to 9th-grade difficulty level. ARC's questions are divided into Easy and Challenge sets that test different types of knowledge such as factual, definition, purpose, spatial, process, experimental, and algebraic.

- ARC was devised to be a more comprehensive and difficult benchmark than previous QA benchmarks, such as the Stanford Question and Answer Dataset (SQuAD) or the Stanford Natural Language Inference (SNLI) corpus, which only tended to measure a model's ability to extract the correct answer from a passage. To achieve this, the ARC corpus provides distributed evidence: typically containing most of the information required to answer a question – but spreading the pertinent details throughout a passage. This requires a language model to solve ARC questions through its knowledge and reasoning abilities instead of explicitly memorising the answers.

Pros and cons of the ARC benchmark

- Pros

Varied and challenging dataset

Pushes AI vendors to improve QA abilities – not just through fact retrieval but by integrating information from several sentences.

- Cons

Only consists of scientific questions

HellaSwag

- HellaSwag (short for Harder Endings, Longer contexts, and Low-shot Activities for Situations with Adversarial Generations) benchmark tests the commonsense reasoning and natural language inference (NLI) capabilities of LLMs through sentence completion exercises. A successor to the SWAG benchmark, each exercise is composed of a segment of a video caption as an initial context and four possible endings, of which only one is correct.

- Each question revolves around common, real-world physical scenarios that are designed to be easily answerable for humans (with an average score of around 95%) but challenging for NLP models.

- HellaSwag's corpus was created through a process called adversarial filtering, an algorithm that increases the complexity by generating deceptive wrong answers, called adversarial endings, which contain words and phrases relevant to the context – but defy conventional knowledge about the world. These adversarial endings are such that they immediately stand out to most people but often prove difficult for LLMs.

Pros and Cons of the HellaSwag Benchmark

- Pros

Similar to ARC, it evaluates a model's common sense and reasoning, as opposed to mere ability to recall facts

Thoroughly curated dataset: all easily completed contexts from the SWAG dataset were

discarded and human assistants sifted through the adversarial endings and chose the best 70,000.

- Cons

General knowledge – doesn't test common sense reasoning for specialised domains.

MMLU

- Massive Multitask Language Understanding (MMLU) is a broad, but important benchmark that measures an LLM's NLU, i.e., how well it understands language and, subsequently, its ability to solve problems with the knowledge to which it was exposed during training. MMLU was devised to challenge models on their NLU capabilities – in contrast to NLP tasks on which a growing number of models were increasingly excelling at the time.

- The MMLU dataset consists of 15,908 questions divided into 57 tasks drawn from a variety of online sources that test both qualitative and quantitative analysis. Its questions cover STEM (science, technology, engineering and mathematics), humanities (language arts, history, sociology, performing and visual arts, etc.), social sciences, and other subjects from an elementary to an advanced professional level. This in itself was a departure from other NLU benchmarks at the time of its release (like SuperGLUE), which focused on basic knowledge rather than the specialised knowledge covered by MMLU.

Pros and Cons of the MMLU Benchmark

- Pros

Tests a broad range of subjects at various levels of difficulty Broad corpus helps identify areas of general knowledge in which models are deficient

- Cons

Limited information on how corpus was constructed Dataset is shown to have numerous errors

TruthfulQA

- While an LLM may be capable of producing coherent and well-constructed responses, it doesn't necessarily mean they're accurate. The TruthfulQA benchmark attempts to address this, i.e., language models' tendency to hallucinate, by measuring a model's ability to generate truthful answers to questions.

- There could be several reasons why an LLM produces inaccurate responses. Chief among them is the model being given a lack of training data for particular subjects, rendering it unable to generate a truthful answer. Similarly, the LLM could have been trained on low-quality data that was full of inaccuracies. Alternatively, the false answers may have been incentivised during the model's training, i.e., faulty training objectives: these are known as imitative falsehoods.

- TruthfulQA's dataset is designed in such a way as to encourage models to choose imitative falsehoods instead of true answers. It assesses the truthfulness of an LLM's response by how much it describes the literal truth about the real world. Consequently, answers that stem from a particular belief system or works of fiction present in the training data are considered false. Additionally, TruthQA measures how informative an answer is – to avoid LLMs attaining high scores by simply responding sceptically with "I don't know" or "I'm not sure".

- The TruthQA corpus consists of 817 questions across 38 categories, such as finance, health, and politics. To calculate a score, each model is put through two tasks. The first requires the model to generate answers to a series of questions. Each response is scored between 0 and 1 by human evaluators, where 0 is false and 1 is true. For the second task, instead of generating an answer, the LLM must choose true or false for a series of multiple-choice questions, which are tallied. The two scores are then combined to produce a final result.

Pros and cons of the TruthfulQA benchmark

- Pros

Diverse dataset Tests LLMs for hallucinations and encourages model accuracy

- Cons

Corpus covers general knowledge, so not a great indicator of truthfulness for specialised domains

SuperGLUE

- The General Language Understanding Evaluation (GLUE) benchmark tests an LLM's NLU capabilities and was notable upon its release for its variety of assessments. SuperGLUE improves upon GLUE with a more diverse and challenging collection of tasks that assess a model's performance across eight subtasks and two metrics, with their average providing an overall score.

Here's a summary of the SuperGLUE benchmark's subtasks and metrics:

Subtasks

Boolean Questions (BoolQ): yes/no QA task

CommitmentBank (CB): truthfulness assessment

Choice of Plausible Alternatives (COPA): causal reasoning task

Multi-Sentence Reading Comprehension (MultiRC): true/false QA task

Reading Comprehension with Commonsense Reasoning Dataset (ReCoRD):

multiple-choice QA task

Recognizing Textual Entailment (RTE): two-class classification task

Word-in-Context (WiC): is a binary classification task

Winograd Schema Challenge (WSC): pronoun resolution problems

Metrics

Broad Coverage Diagnostics: to automatically test an LLM's linguistic, common sense, and general world knowledge

Analysing Gender Bias in Models: an analytical tool for detecting a model's social biases

Pros and cons of the SuperGLUE benchmark

- Pros

A thorough and diverse range of tasks that test a model's NLU capabilities

- Cons

A smaller range of models are tested against SuperGLUE than similar benchmark MMLU

HumanEval

- HumanEval (also often referred to as HumanEval-Python) is a benchmark designed to measure a model's ability to generate functionally correct code; it consists of the HumanEval dataset and the pass@k metric.
- This HumanEval dataset was carefully designed and contains 164 diverse coding challenges that include several unit tests (7.7 on average). The pass@k metric calculates the probability that at least one of k generated code samples pass the coding challenge's unit tests, given that there are c correct samples from n generated samples.
- In the past, the BLEU (bilingual evaluation understudy) metric was used to assess the textual similarity of model-generated coding solutions compared with human ones. The problem with this approach, however as it doesn't evaluate the functional correctness of the generated solution; for more complex problems, the solution could still be functionally correct while appearing different textually from the solution produced by a person. The HumanEval addressed this by utilising unit tests to evaluate a code sample's functionality in a similar way that humans would.

Pros and cons of the HumanEval benchmark

- Pros

A good indication of a model's coding capability Unit testing mirrors the way humans evaluate code functionality

- Cons

The HumanEval dataset doesn't comprehensively capture how coding models are used in practice. For instance, it doesn't test for aspects such as writing tests, code explanation, code infilling, or docstring generation.

MT Bench

- MT-Bench is a benchmark that evaluates a language model's capability to effectively engage in multi-turn dialogues. By simulating the back-and-forth conversations that LLMs would have in real-life situations, MT-Bench provides a way to measure how effectively chatbots follow instructions and the natural flow of conversations.
- MT Bench was developed through the use of Chatbot Arena: a crowd-sourced platform that allows users to evaluate a variety of chatbots by entering a prompt and comparing the two responses side-by-side. Users could then vote for which model provided the best response, which was recorded and tallied to produce a leaderboard of the best-performing LLMs.
- Through this process, the researchers behind Chatbot Arena identified eight main types of user prompts: writing, roleplay, extraction, reasoning, math, coding, knowledge I (STEM), and knowledge II (humanities). Subsequently, they devised 10 multi-turn questions per category, to create a total set of 160 questions. While the results from Chatbot Arena are subjective, MT-Bench is intended to complement it with a more objective measure of a model's conversational capabilities.

Pros and Cons of the MT-Bench Benchmark

- Pros

Measures a model's ability to answer subsequent, related questions

- Cons

Though carefully curated, the dataset is small Hard to simulate the broad and unpredictable nature of conversations

```
display_image('/content/llmbenchmarkspicture.webp')
```

Benchmark (Higher is better)	MPT (7B)	Falcon (7B)	Llama-2 (7B)	Llama-2 (13B)	MPT (30B)	Falcon (40B)	Llama-1 (65B)	Llama-2 (70B)
MMLU	26.8	26.2	45.3	54.8	46.9	55.4	63.4	68.9
TriviaQA	59.6	56.8	68.9	77.2	71.3	78.6	84.5	85.0
Natural Questions	17.8	18.1	22.7	28.0	23.0	29.5	31.0	33.0
GSM8K	6.8	6.8	14.6	28.7	15.2	19.6	50.9	56.8
HumanEval	18.3	N/A	12.8	18.3	25.0	N/A	23.7	29.9
AGIEval (English tasks only)	23.5	21.2	29.3	39.1	33.8	37.0	47.6	54.2
BoolQ	75.0	67.5	77.4	81.7	79.0	83.1	85.3	85.0
HellaSwag	76.4	74.1	77.2	80.7	79.9	83.6	84.2	85.3
OpenBookQA	51.4	51.6	58.6	57.0	52.0	56.6	60.2	60.2
QuAC	37.7	18.8	39.7	44.8	41.1	43.3	39.8	49.3
Winogrande	68.3	66.3	69.2	72.8	71.0	76.9	77.0	80.2

```
display_image('/content/llmbenchmark.png')
```

	Average	Multi-choice Qs	Reasoning	Python coding	Future Capabilties	Grade school math	Math Problems
Claude 3 Opus	84.83%	86.80%	95.40%	84.90%	86.80%	95.00%	60.10%
Gemini 1.5 Pro	80.08%	81.90%	92.50%	71.90%	84%	91.70%	58.50%
Gemini Ultra	79.52%	83.70%	87.80%	74.40%	83.60%	94.40%	53.20%
GPT-4	79.45%	86.40%	95.30%	67%	83.10%	92%	52.90%
Claude 3 Sonnet	76.55%	79.00%	89.00%	73.00%	82.90%	92.30%	43.10%
Claude 3 Haiku	73.08%	75.20%	85.90%	75.90%	73.70%	88.90%	38.90%
Gemini Pro	68.28%	71.80%	84.70%	67.70%	75%	77.90%	32.60%
Palm 2-L	65.82%	78.40%	86.80%	37.60%	77.70%	80%	34.40%
GPT-3.5	65.46%	70%	85.50%	48.10%	66.60%	57.10%	34.1%
Mixtral 8×7B	59.79%	70.60%	84.40%	40.20%	60.76%	74.40%	28.40%

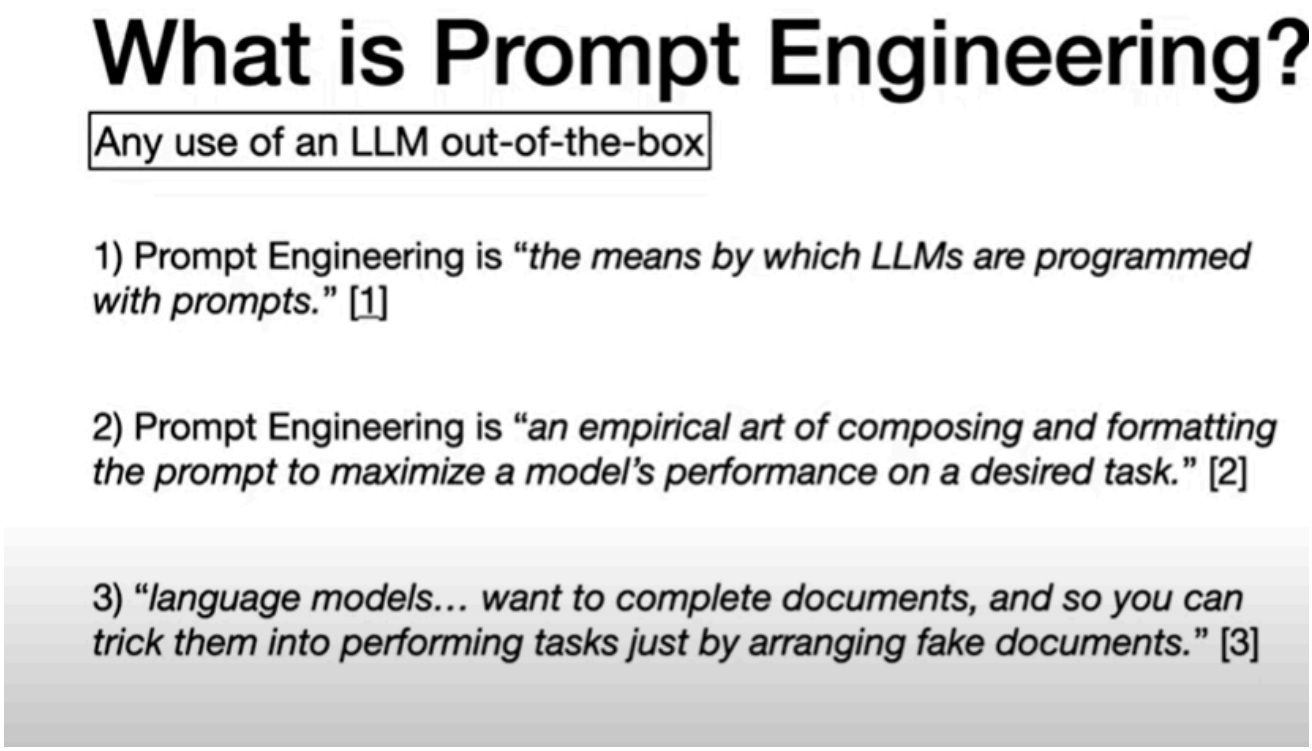
LLM Evaluation Metrics

Medium Blog

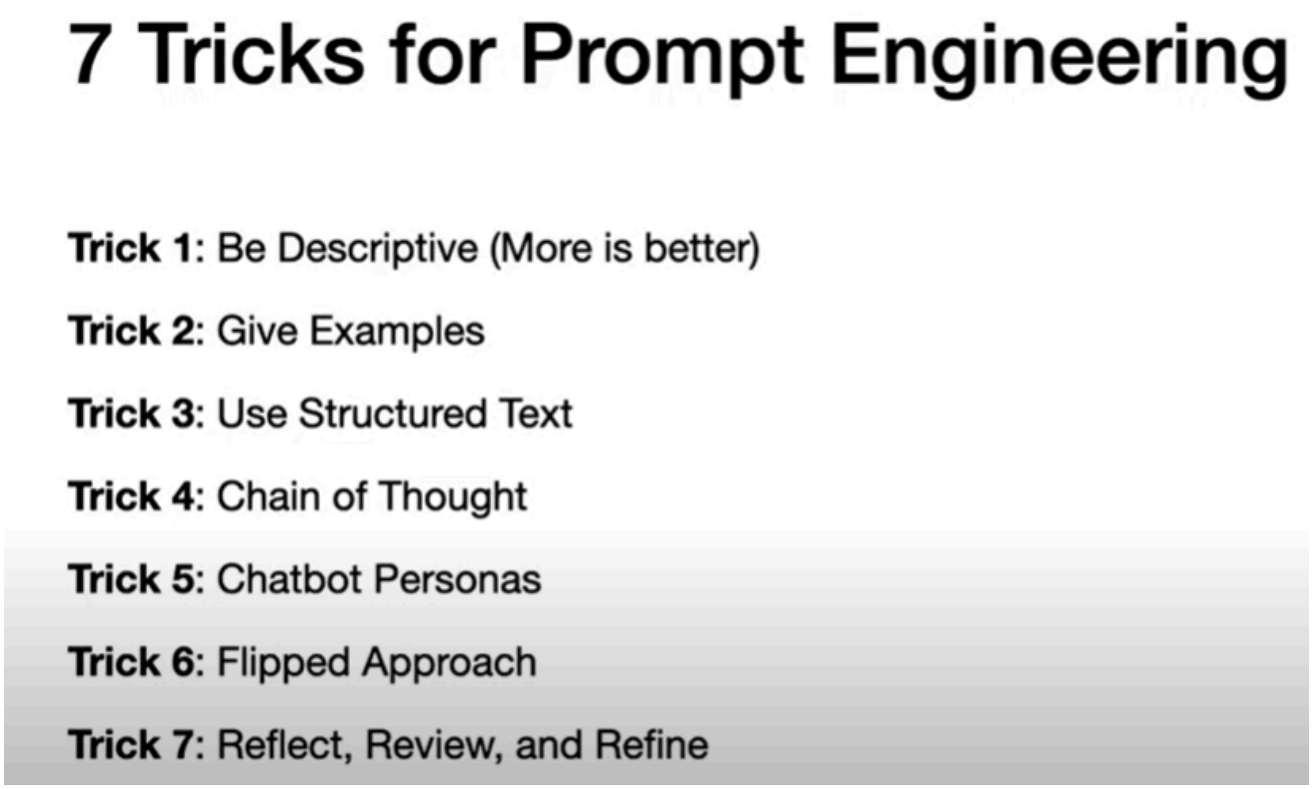
```
Start coding or generate with AI.
```

Prompt Engineering

```
display_image('/content/what_prompt_engineering.png')
```

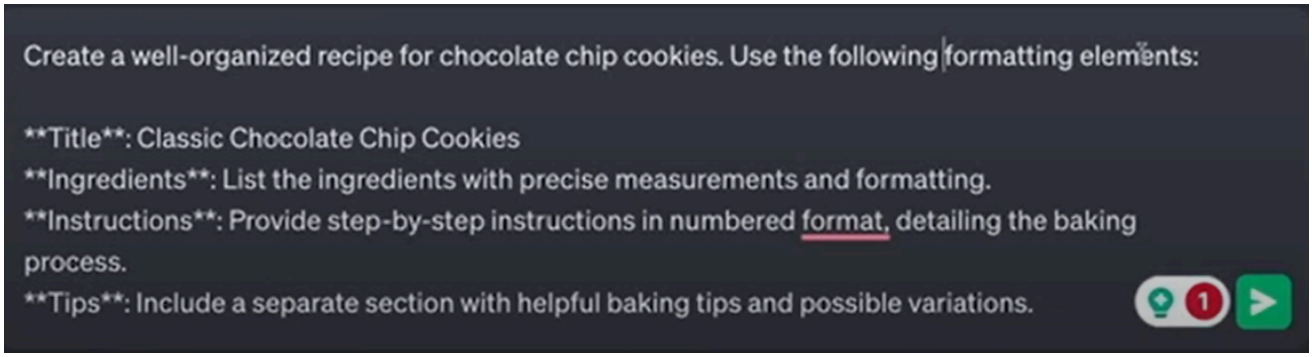



```
display_image('/content/tricks_prompt.png')
```



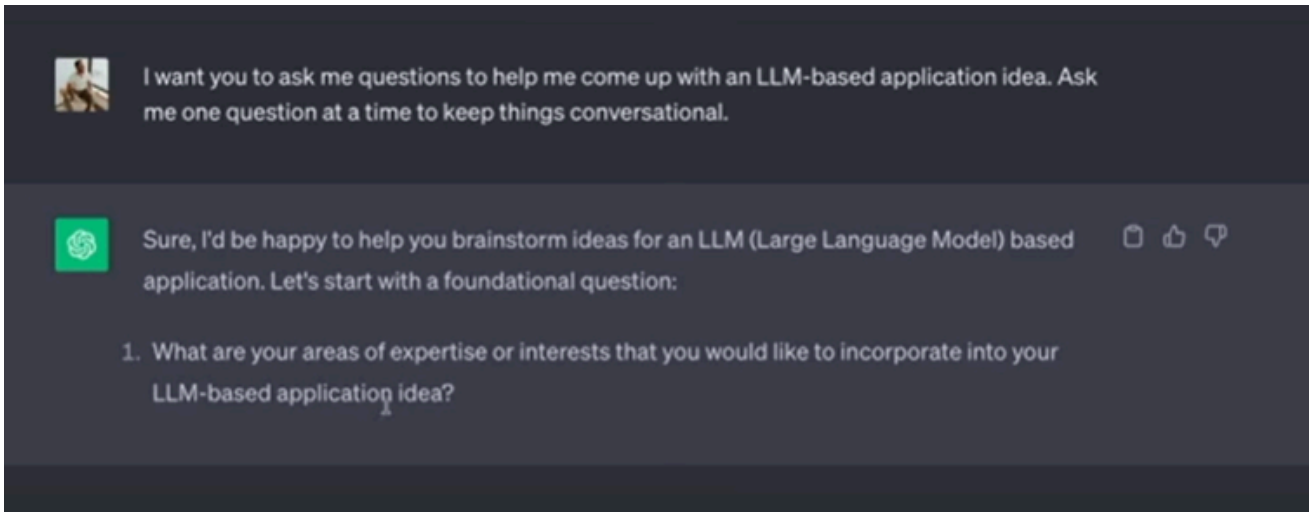
Structured Text

```
display_image('/content/structure_prompt.png')
```



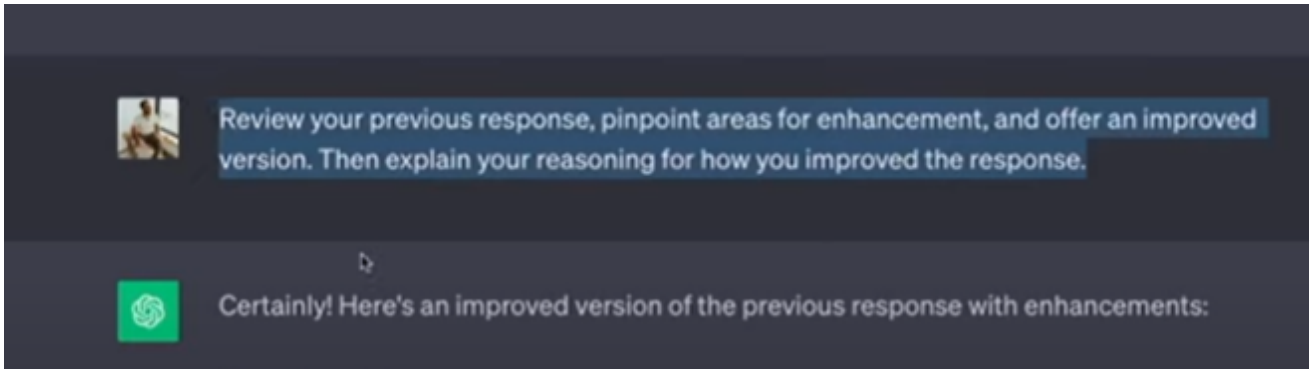
Flipped Approach

```
display_image('/content/keep_conversation_prompt.png')
```



Reflect Review Prompt

```
display_image('/content/review_prmpt.png')
```

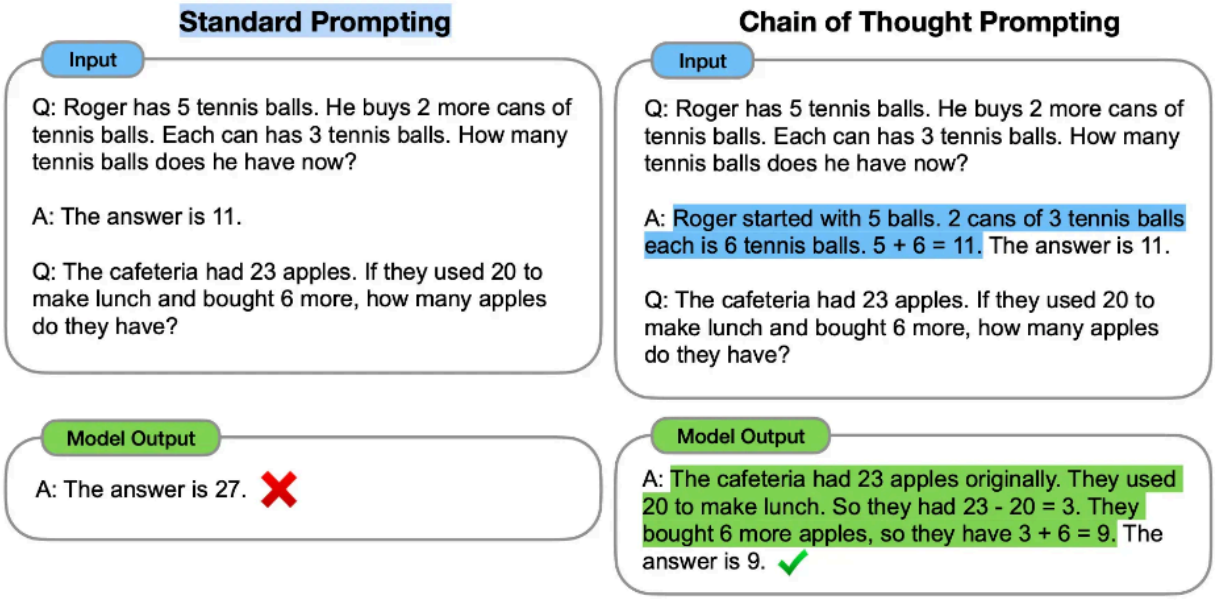


Chain of thought(CoT)

- **Chain of thought (CoT) prompting1** is a recent advancement in prompting methods that encourage Large Language Models (LLMs) to explain their reasoning. This method contrasts with standard prompting by not only seeking an answer but also requiring the model to explain its steps to arrive at that answer.

The below image1 shows a few shot standard prompt (left) compared to a chain of thought prompt (right). This comparison between a few-shot standard prompt and a chain-of-thought prompt illustrates the difference: while the standard approach directly seeks a solution, the CoT approach guides the LLM to unfold its reasoning, often leading to more accurate and interpretable results.

```
display_image('/content/CoT.png')
```



Model size:

- CoT seems to be more effective for larger LLMs with stronger reasoning capabilities. Smaller models might struggle to follow the prompts or generate their own reasoning chains.

```
# Prompt without CoT
prompt = "What is the sum of 5 and 3?"
# Prompt with CoT
cot_steps = [
 "Let's add the first number, 5.",
 "Then, add the second number, 3, to the previously obtained sum.",
 "The answer is the final sum.",
]
# Combine prompt and CoT steps
prompt_with_cot = "\n".join([prompt] + cot_steps)
# Use the prompt with/without CoT to generate the answer
# (the actual code for generating the answer will depend on the LLM platform)
```

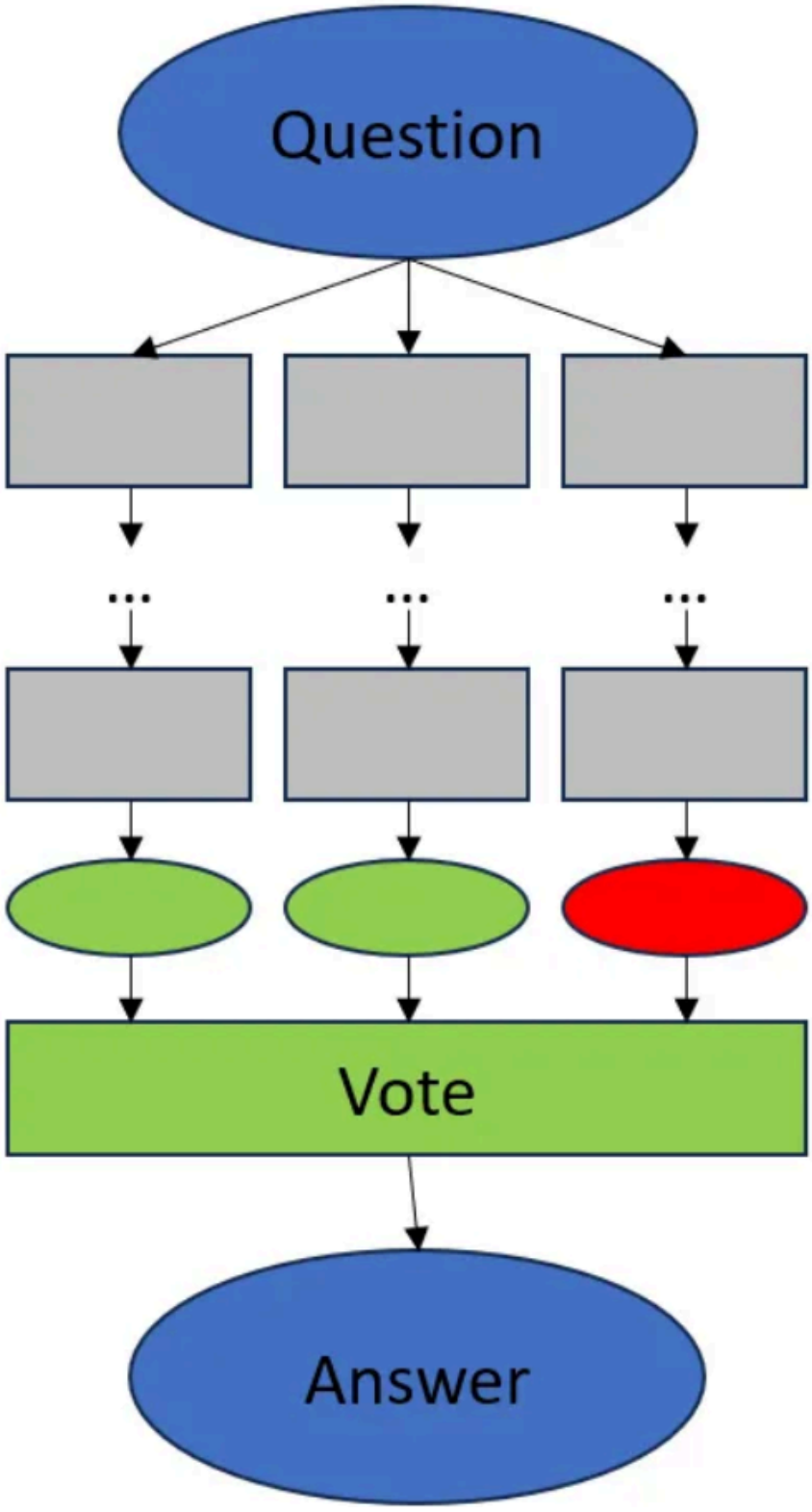
```
# Prompt without CoT
prompt = "Who was the first person to walk on the moon?"
# Prompt with CoT
cot_steps = [
 "The moon landing happened in 1969.",
 "We need to identify the astronaut who first stepped onto the moon during that mission."
 "Based on historical records, Neil Armstrong was the first person to walk on the moon.",
]
# Combine prompt and CoT steps
prompt_with_cot = "\n".join([prompt] + cot_steps)
# Use the prompt with/without CoT to generate the answer
# (the actual code for generating the answer will depend on the LLM platform)
```

```
Start coding or generate with AI.
```

Self-Consistency with Chain of Thought (CoT-SC)

(Main idea) It is assumed that the correct answer is within in the language model and that this correct answer is returned from the language model in the majority of cases while repeatedly asking the model the same question. The iterative process, culminating in a choice of final answers, is illustrated in the figure below. In this example, a majority vote is used for decision-making, i.e., the answer that occurs most frequently is selected as the correct one. In any case, the Chain of Thought with Self-Consistency process ends in a choice that synthesizes one correct answer from all the answers found.

```
display_image('/content/cot_self_consistency.png')
```



Procedure

1. Add „think step-by-step“ to your original question (we’ll call this augmented question the question in the following).

2. Ask the question repeatedly (n times) and collect the answers.
3. Decide for a voting technique and decide which of the collected answers is picked as the final answer.

Advantages / Performance gain Chain of Thought with Self-Consistency helps to overcome 2 major problems of Large Language Models (LLMs) as the mentioned source describes. These partially overcome problems are:

1. **Repetitiveness and Local-Optimality in Greedy Decoding**: During text generation, LLMs pick at each step the most probable next word (with some randomness introduced). It's called greedy because it always picks the next best option. This can lead to local optimality and missing the big picture since the model only thinks about which next word is the best one in the current context and not where it wants to get in the long run.
2. **Stochasticity of a Single Sampled Generation:** This mitigates the prior issue a bit but it introduces a new problem. Choosing the next token not only relies on the best choice for it but introduces some randomness. While this can give more diverse outputs on one hand, on the other it can also generate nonsensical ones.

```
!pip install openai dotenv
```

```
# Chain of Thought with Self-Consistency with majority vote demo
# Code written by OpenAI Code Interpreter
# 2023-08-07
import openai
from dotenv import find_dotenv, load_dotenv
import os
```

```
# Initialize OpenAI API with your key
load_dotenv(find_dotenv(), override=True)
api_key = os.environ.get('OPENAI_API_KEY')
openai.api_key = api_key
engine = "gpt-3.5-turbo"
def chain_of_thought_prompting(initial_prompt, iterations=5):
    current_prompt = initial_prompt + ", think step-by-step"
    messages = [
        {"role": "system", "content":"you answer like a scientist in a brief and precise
        {"role": "user", "content": current_prompt}
    ]

    response = openai.ChatCompletion.create(
        model=engine,
        messages=messages,
        max_tokens=150,
        n=iterations,
    )

    responses = [choice['message']['content'] for choice in response['choices'] if choice
    responses_string = "first answer: " + ', next answer:'
```

Tree Of Thoughts (ToT)

The Tree of Thoughts (ToT) prompting framework uses a tree structure of each step of the reasoning process that allows the language model to evaluate each reasoning step and decide whether or not that step in the reasoning is viable and lead to an answer. If the language model decides that the reasoning path will not lead to an answer the prompting strategy requires it to abandon that path (or branch) and keep moving forward with another branch, until it reaches the final result.

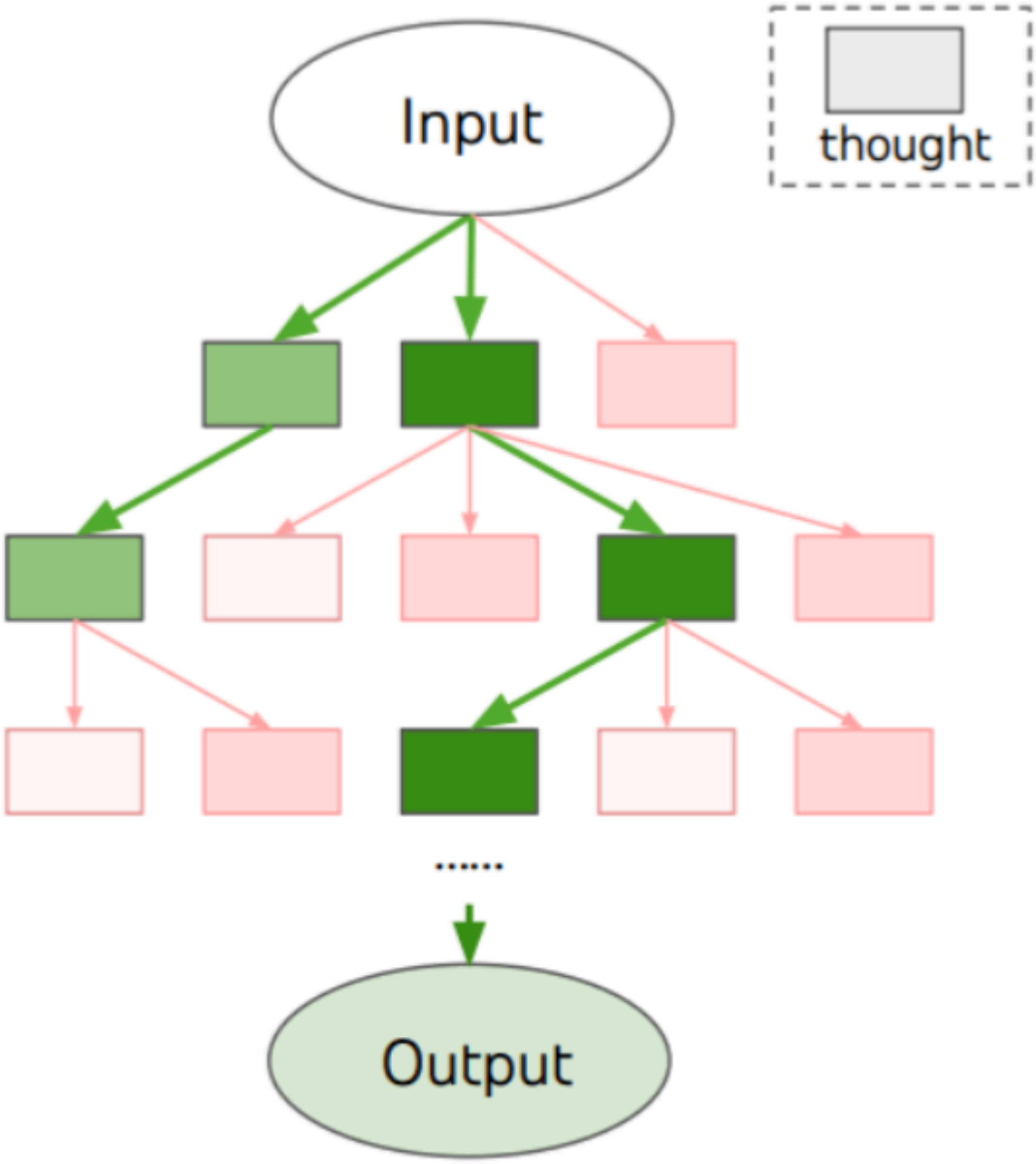
Tree Of Thoughts (ToT) Versus Chain of Thoughts (CoT)

The difference between ToT and and CoT is that ToT is has a tree and branch framework for the reasoning process whereas CoT takes a more linear path.

In simple terms, CoT tells the language model to follow a series of steps in order to accomplish a task, which resembles the system 1 cognitive model that is fast and automatic.

ToT resembles the system 2 cognitive model that is more deliberative and tells the language model to follow a series of steps but to also have an evaluator step in and review each step and if it's a good step to keep going and if not to stop and follow another path.

```
display_image('/content/tot_prompt.png')
```




```
!pip install tree-of-thoughts swarms
```

```
import os
from tree_of_thoughts import ToTAgent, MonteCarloSearch
```