Large language models (LLMs)

Large language models (LLMs) are advanced AI systems that use deep learning, particularly the Transformer architecture, to understand and generate human-like text by learning patterns from vast datasets. Trained on billions of texts and other content, LLMs excel at tasks like text generation, translation, summarization, and question answering. Key factors enabling their power include the Transformer's parallel processing and "attention" mechanisms, increased computational power, and the availability of large, high-quality datasets.

How LLMs Work

1. **Training on Data**:

LLMs are trained on massive amounts of text and data to identify complex patterns, grammar, and factual information.

2. **Transformer Architecture**:

This deep learning architecture, introduced in 2017 by Google, is fundamental to most LLMs.

3. **Attention Mechanisms**:

Within the Transformer, "attention" mechanisms allow the model to focus on the most relevant parts of the input text when processing and generating output, improving understanding of context.

4. **Text Generation**:

LLMs generate responses word-by-word by predicting the most probable next word in a sequence based on the preceding text and its vast training data.

Key Characteristics

- **Vast Parameters**:

LLMs have a huge number of parameters, which are essentially the learned values that enable them to process longer text sequences and handle complex tasks.

- **Contextual Understanding**:

They can understand the relationships between words and phrases within a text, enabling them to produce relevant and in-context content.

- **Self-supervised Learning**:

LLMs are trained using self-supervised machine learning, which allows them to learn from data without requiring explicit human labels.

Applications

- **Content Creation**: Generating articles, stories, and code.
- **Translation**: Translating text between different languages.
- **Summarization**: Condensing long pieces of text into shorter summaries.
- **Question Answering**: Providing answers to questions based on their training data.

Challenges

High Costs: Training LLMs is computationally intensive and expensive.

Bias: LLMs can reflect biases present in their training data.

Hallucinations: They may sometimes generate factually incorrect or nonsensical information.

Ethical Implications: Responsible deployment, data privacy, and potential misuse are significant concerns.

Introduction to Tokenizer

A **tokenizer** is a fundamental component in **Natural Language Processing (NLP)** and **Deep Learning** that breaks down raw text into smaller, manageable units called **tokens**. These tokens can be **words, subwords, characters, or symbols**, depending on the application and model design. Tokenization is the **first and most crucial step** in transforming human language into a numerical format that computers can process and understand.

What Is Tokenization?

Tokenization is the process of splitting a sequence of text into individual pieces (tokens). For example:

Sentence: “Deep learning transforms AI.”
Tokens: [“Deep”, “learning”, “transforms”, “AI”, “.”]

Each token represents a meaningful element of text that the model can analyze. After tokenization, these tokens are usually converted into **numerical IDs** through a **vocabulary** or **embedding layer**, enabling the neural network to perform computations.

Example: Book page 38

Have the bards who preceded me left any theme unsung?

1. Overview

Feature	BERT	GPT
Full Name	Bidirectional Encoder Representations from Transformers	Generative Pre-trained Transformer
Developed By	Google AI	OpenAI
Model Type	Encoder-only Transformer	Decoder-only Transformer
Training Objective	**Masked Language Modeling (MLM)**: Predict missing words in a sentence	**Causal Language Modeling (CLM)**: Predict next word in a sequence
Directionality	**Bidirectional**: reads context from both left and right	**Unidirectional** (left-to-right)
Purpose	Understanding tasks: classification, QA, NER, sentiment analysis	Generation tasks: text completion, summarization, dialogue, story writing
Input Processing	Tokens fed through encoder to produce embeddings representing context	Tokens fed through decoder to generate next token probabilities

2. Tokenization

- **BERT:** Uses **WordPiece tokenizer**, which splits rare words into subwords. Special tokens include [CLS] (classification start) and [SEP] (sentence separator).
- **GPT:** Uses **Byte-Pair Encoding (BPE)**. Adds special handling like Ġ to indicate a space, supporting smooth text generation.

Example: “Have the bards”

- BERT tokens: [CLS], have, the, bards, [SEP]
- GPT tokens: ['Have', 'Ġthe', 'Ġbards']

3. Use Cases

BERT (Understanding):

- Sentiment Analysis
- Question Answering (SQuAD)
- Named Entity Recognition (NER)
- Text Classification

GPT (Generation):

- Text completion
- Story/essay generation
- Dialogue/chatbots
- Summarization and translation

4. Key Differences in Training & Architecture

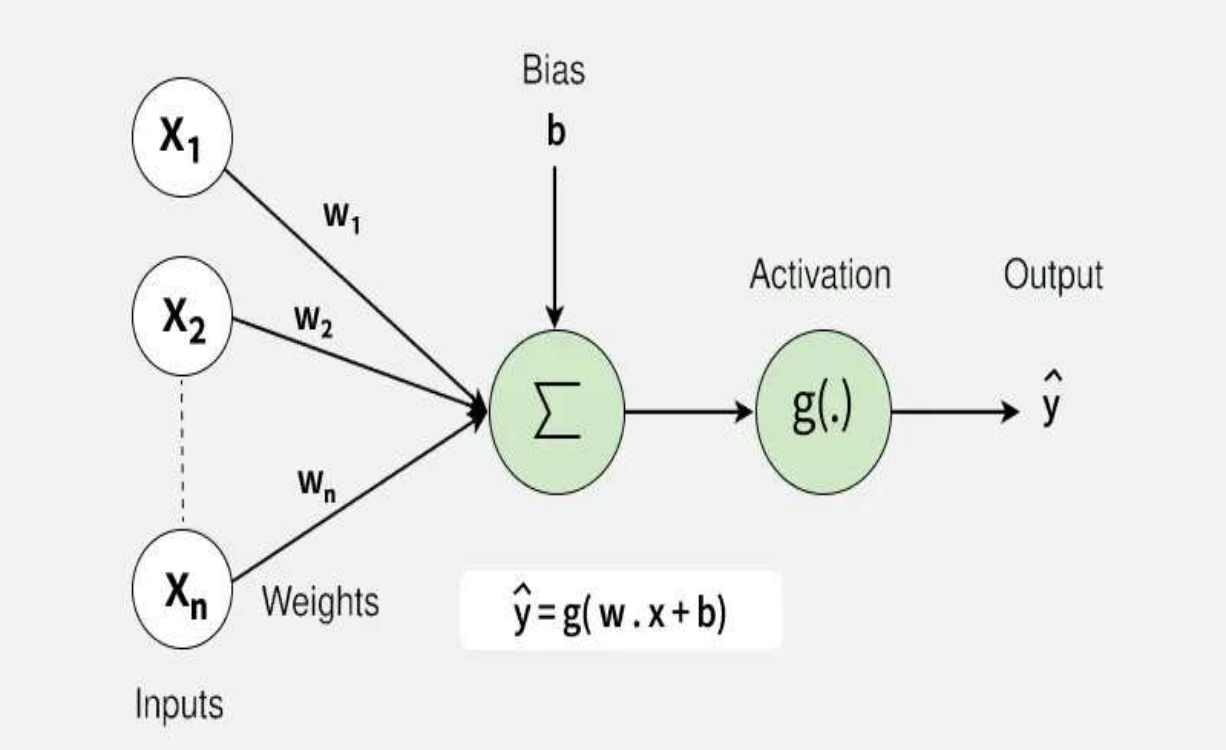
1. **Contextual Understanding**
    - BERT: Bidirectional → understands meaning from **both left and right context**.
    - GPT: Left-to-right → generates coherent text **one word at a time**.
2. **Fine-Tuning**
    - BERT: Fine-tuned on downstream tasks with task-specific layers.
    - GPT: Can be fine-tuned, but often used in **few-shot learning** or **prompt-based generation**.
3. **Output**
    - BERT: Produces embeddings for tokens (good for classification or comprehension).
    - GPT: Produces probabilities for **next token prediction** (good for generation).

Summary

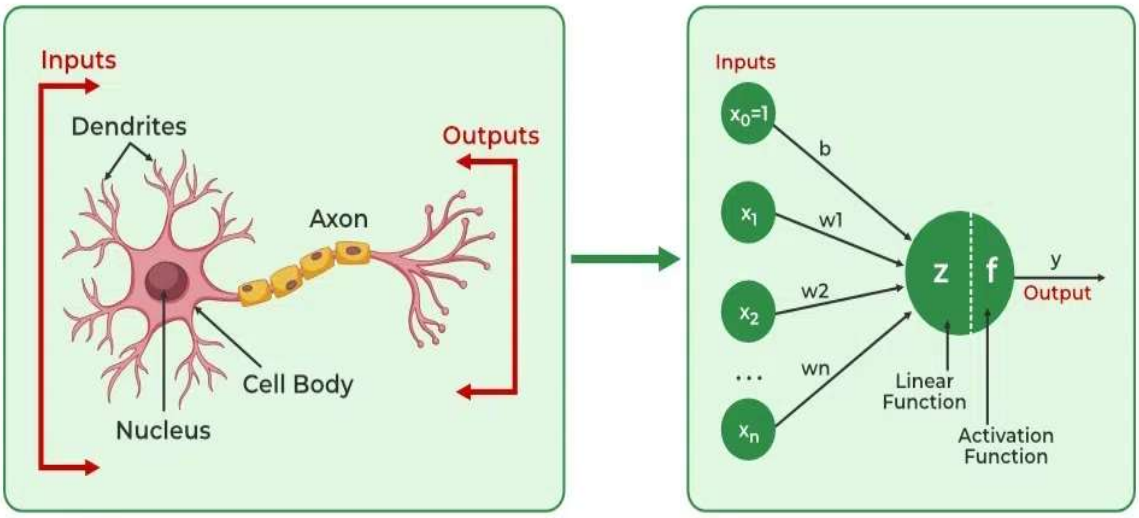
- **BERT = “Reader”** → excels at understanding and analyzing text.
- **GPT = “Writer”** → excels at generating coherent and fluent text.

Neural Network

Neural networks are machine learning models that mimic the complex functions of the human brain. These models consist of interconnected nodes or neurons that process data, learn patterns and enable tasks such as pattern recognition and decision-making.

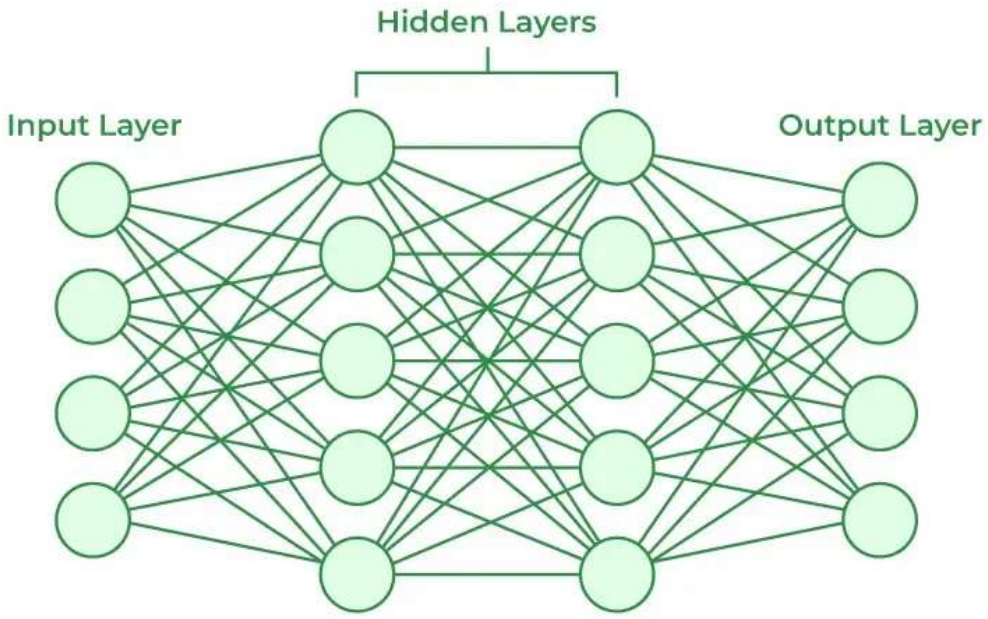


Autoregressive model, Basics of Transformer Architecture, choosing a single token from probability distribution, parallel token processing ***SL. No-9***


Inputs
Dendrites
Outputs
Axon
Cell Body
Nucleus
Inputs
$x_0=1$
b
x_1
w1
x_2
w2
...
wn
x_n
z
f
y
Output
Linear Function
Activation Function

Layers in Neural Network Architecture

1. **Input Layer:** This is where the network receives its input data. Each input neuron in the layer corresponds to a feature in the input data.
2. **Hidden Layers:** These layers perform most of the computational heavy lifting. A neural network can have one or multiple hidden layers. Each layer consists of units (neurons) that transform the inputs into something that the output layer can use.
3. **Output Layer:** The final layer produces the output of the model. The format of these outputs varies depending on the specific task like classification, regression.



Working of Neural Networks

1. Forward Propagation

When data is input into the network, it passes through the network in the forward direction, from the input layer through the hidden layers to the output layer. This process is known as forward propagation. Here's what happens during this phase:

1. Linear Transformation: Each neuron in a layer receives inputs which are multiplied by the weights associated with the connections. These products are summed together and a bias is added to the sum. This can be represented mathematically as:

$$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

where

- w represents the weights
- x represents the inputs
- b is the bias

2. Activation: The result of the linear transformation (denoted as z) is then passed through an activation function. The activation function is crucial because it introduces non-linearity into the system, enabling the network to learn more complex patterns. Popular activation functions include ReLU, sigmoid and tanh.

2. Backpropagation

After forward propagation, the network evaluates its performance using a loss function which measures the difference between the actual output and the predicted output. The goal of training is to minimize this loss. This is where backpropagation comes into play:

- **Loss Calculation:** The network calculates the loss which provides a measure of error in the predictions. The loss function could vary; common choices are mean squared error for regression tasks or cross-entropy loss for classification.
- **Gradient Calculation:** The network computes the gradients of the loss function with respect to each weight and bias in the network.
- **Weight Update:** Once the gradients are calculated, the weights and biases are updated using an optimization algorithm like stochastic gradient descent (SGD). The weights are adjusted in the opposite direction of the gradient to minimize the loss.

3. Iteration

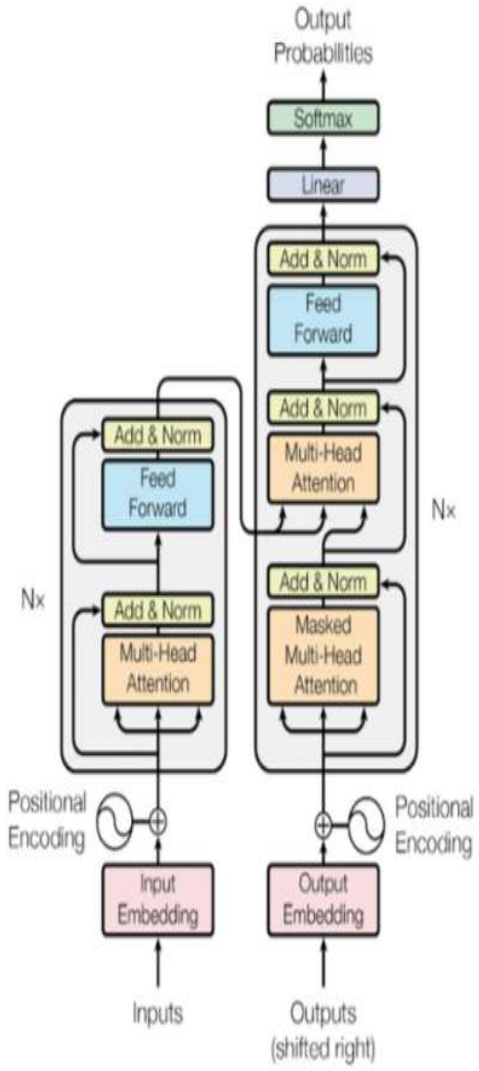
This process of forward propagation, loss calculation, backpropagation and weight update is repeated for many iterations over the dataset. Over time, this iterative process reduces the loss and the network's predictions become more accurate.

Through these steps, neural networks can adapt their parameters to better approximate the relationships in the data, thereby improving their performance on tasks such as classification, regression or any other predictive modelling.

Deep Dive into the Transformer Architecture:

Transformer

Attention Is All You Need



Transformer Architecture represents a paradigm shift in how sequential data is processed for natural language understanding and generation. At its core, the Transformer model abandons the conventional reliance on recurrent or convolutional layers, instead leveraging a novel mechanism known as self-attention to process data in parallel.

The Encoder-Decoder Structure

Central to the Transformer's design is its encoder-decoder structure, consisting of stacks of encoder and decoder layers. Each encoder layer performs two primary functions: self-attention and position-wise feed-forward neural networks. The encoder's role is to process the input sequence and map it into a continuous representation that holds both the semantic and syntactic information of the input.

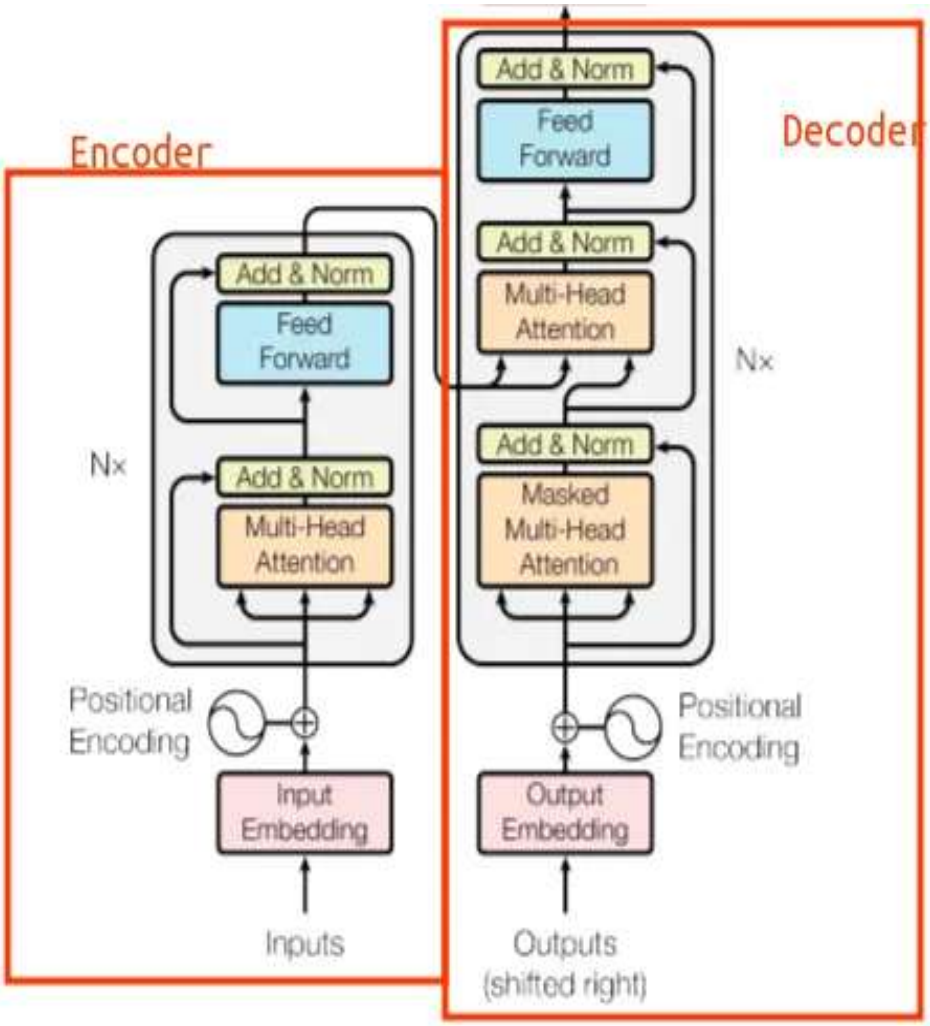
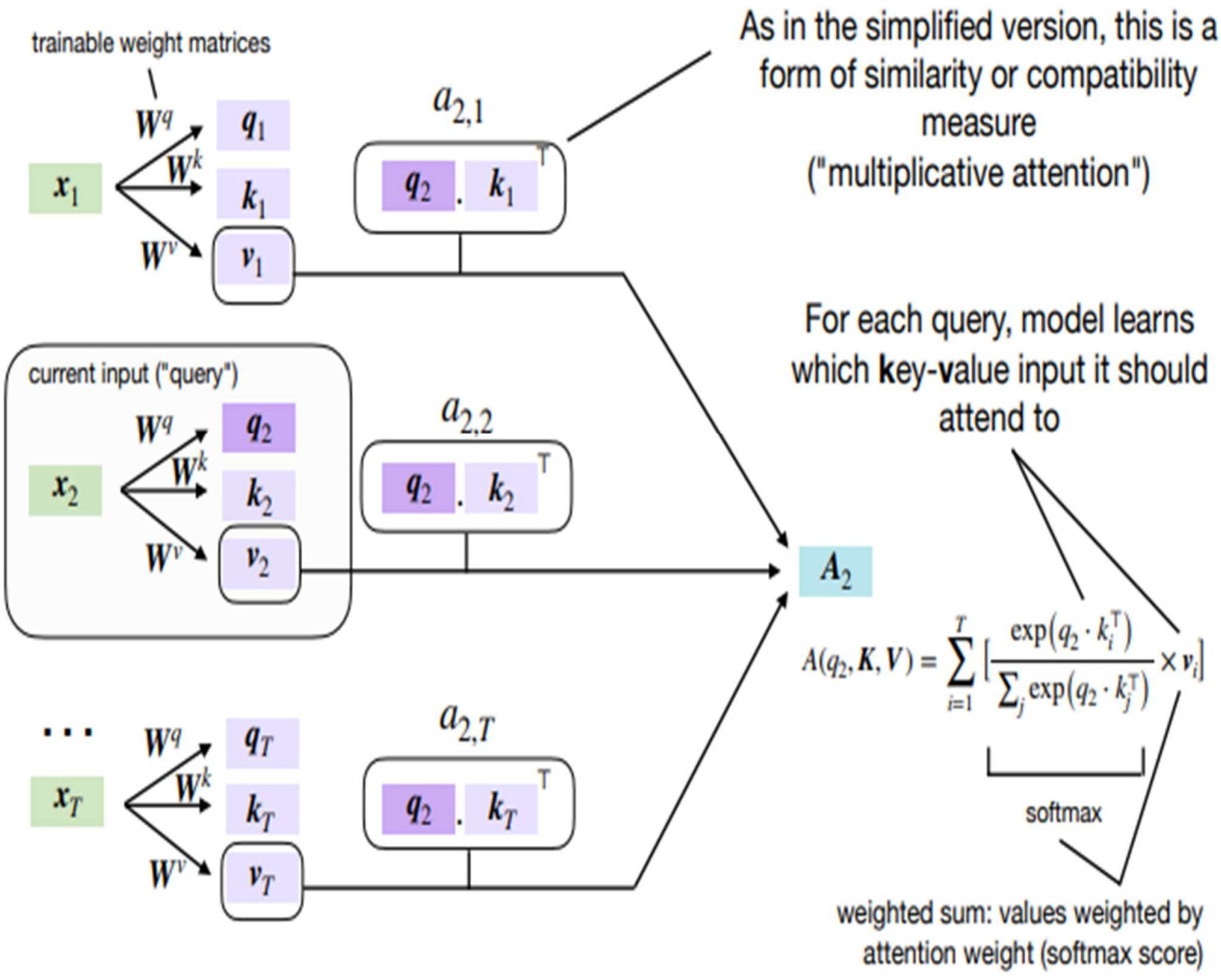


Fig. **The Encoder-Decoder Structure**

Self-Attention Mechanism

The self-attention mechanism is the heart of the Transformer, enabling the model to weigh the importance of different words within the input sequence relative to each other. Unlike traditional attention mechanisms that operate in a query-key-value paradigm, self-attention applies this concept within the input sequence itself, allowing each position to attend to all positions and thus capturing the intricate dependencies regardless of their distance in the sequence. This mechanism is crucial for understanding the context and meaning of words in sentences, enhancing the model's language processing capabilities.

Self-Attention Mechanism



Real-world Applications and Impact of the Transformer Architecture

The Transformer architecture has been foundational in driving advancements across a wide spectrum of NLP tasks. Its ability to process sequential data in parallel and capture long-range dependencies has led to significant improvements in both understanding and generating natural language.

Machine Translation

The initial application of the Transformer model demonstrated its superiority in machine translation tasks. By efficiently handling sequences and understanding the context of entire sentences, the Transformer has achieved state-of-the-art performance in translating between languages, reducing training times and improving accuracy.

Text Summarization

Transformers have revolutionized text summarization by enabling models to generate concise and relevant summaries of lengthy texts. This is achieved through their ability to understand the overall context and identify the most significant parts of the source material, a task that previous models struggled with.

Question Answering

In question-answering systems, the Transformer's ability to understand and process natural language queries has led to more accurate and context-aware responses. By analysing the relationships and significance of words within both the query and the source documents, these models can provide precise answers to a wide range of questions.

Sentiment Analysis

The application of Transformer models in sentiment analysis has allowed for more nuanced and context-sensitive interpretations of text. Their deep understanding of language enables them to discern subtle nuances in sentiment, leading to more accurate classification of texts according to the sentiments expressed.

Language Model Pre-training

Perhaps the most transformative application of the Transformer has been in the development of large pre-trained language models like BERT and GPT. These models, built upon the Transformer architecture, have set new benchmarks in a multitude of NLP tasks by leveraging vast amounts of text data to learn rich representations of language.

Example: ***Problem***

Consider the neural network in Figure 2.11. The network takes two input variables, x_1 and x_2, outputs two continuous variables, y_1 and y_2, and utilises the Sigmoid activation function at each hidden unit. At the current training checkpoint, the weights have the following values: $w_{11}^{(1)} = 0.1$, $w_{21}^{(1)} = 0.2$, $w_{12}^{(1)} = 0.3$, $w_{22}^{(1)} = 0.4$, $w_{11}^{(2)} = 0.5$, $w_{21}^{(2)} = 0.7$, $w_{12}^{(2)} = 0.6$, $w_{22}^{(2)} = 0.8$. The bias terms are $b_1 = 0.25$ and $b_2 = 0.35$.

Given a new training input vector $\mathbf{x} = (x_1, x_2) = (0.1, 0.5)$ and the expected output $\mathbf{t} = (t_1, t_2) = (0.05, 0.95)$, let us calculate the update for using $w_{11}^{(2)}$ stochastic gradient descent and $\eta = 0.1$.

We will first forward propagate through the neural network to store values of hidden units and predicted outputs.

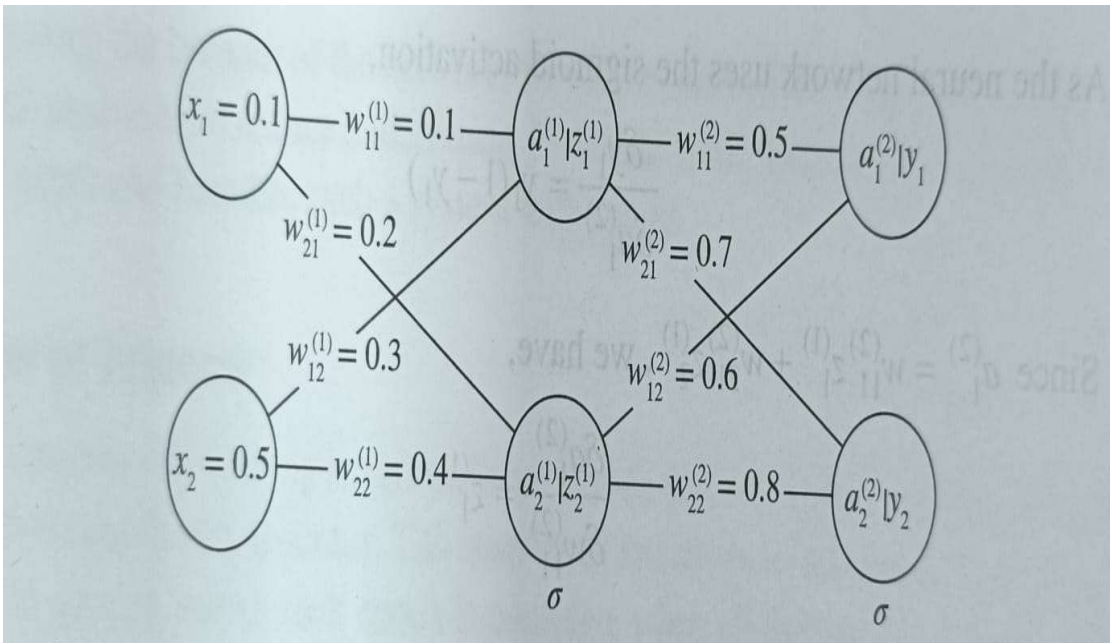


FIGURE 2.11 The neural network architecture for Example 2.5.

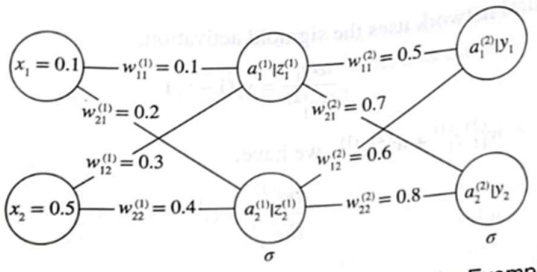


FIGURE 2.11 The neural network architecture for Example 2.5.

Forward Propagation:

$$a_1^{(1)} = 0.1 \cdot 0.1 + 0.5 \cdot 0.3 + 0.25 = 0.410$$

$$z_1^{(1)} = \sigma(a_1^{(1)}) = \frac{1}{1+e^{-0.41}} = 0.601$$

$$a_2^{(1)} = 0.1 \cdot 0.2 + 0.4 \cdot 0.5 + 0.25 = 0.470$$

$$z_2^{(1)} = \sigma(a_2^{(1)}) = \frac{1}{1+e^{-0.47}} = 0.615$$

$$a_1^{(2)} = 0.5 \times 0.601 + 0.6 \times 0.615 + 0.35 = 1.020$$

$$a_2^{(2)} = 0.7 \times 0.601 + 0.8 \times 0.615 + 0.35 = 1.263$$

$$y_1 = \sigma(a_1^{(2)}) = 0.735$$

$$y_2 = \sigma(a_2^{(2)}) = 0.780$$

We will now calculate the error contribution due to this new training input vector.

Computing Total Error:

$$E = \frac{1}{2}(t_1 - y_1)^2 + \frac{1}{2}(t_2 - y_2)^2$$

$$E = \frac{1}{2}(0.735 - 0.05)^2 + \frac{1}{2}(0.780 - 0.950)^2 = 0.234 + 0.014 = 0.248$$

Backward Propagation: We will use the chain rule to calculate the partial derivative of total error E with respect to $w_{11}^{(2)}$:

$$\frac{\delta E}{\delta w_{11}^{(2)}} = \frac{\delta E}{\delta y_1} \cdot \frac{\delta y_1}{\delta a_1^{(2)}} \cdot \frac{\delta a_1^{(2)}}{\delta w_{11}^{(2)}}$$

Differentiating E with respect to y_1,

$$\frac{\delta E}{\delta y_1} = -1 \cdot (t_1 - y_1)$$

As the neural network uses the sigmoid activation,

$$\frac{\delta y_1}{\delta a_1^{(2)}} = y_1(1-y_1)$$

Since $a_1^{(2)} = w_{11}^{(2)} z_1^{(1)} + w_{21}^{(2)} z_2^{(1)}$, we have,

$$\frac{\delta a_1^{(2)}}{\delta w_{11}^{(2)}} = z_1^{(1)}$$

Therefore,

$$\frac{\delta E}{\delta w_{11}^{(2)}} = (y_1 - t_1) \cdot y_1(1-y_1) \cdot z_1^{(1)} = 0.685 \times 0.735 \times 0.265 \times 0.601 = 0.08018$$

The updated weight $w_{11}^{(2)}$ can be obtained using the following equation (assuming $\eta = 0.1$),

$$(w_{11}^{(2)})_{\text{new}} = w_{11}^{(2)} - 0.1 \times 0.08018 = 0.5 - 0.1 \times 0.08018 = 0.49198$$

Parallel token processing

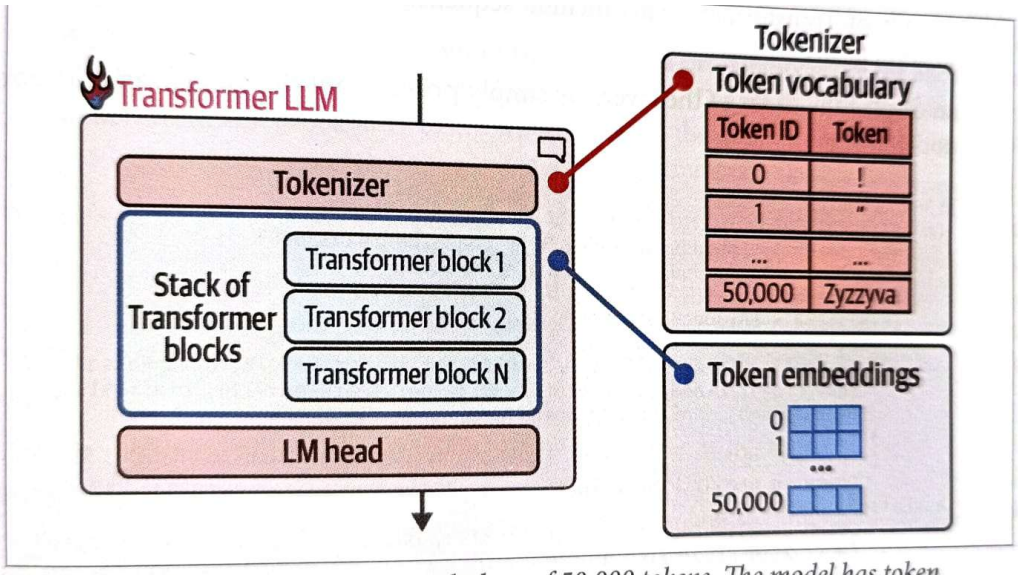


Figure 3-5. The tokenizer has a vocabulary of 50,000 tokens. The model has token embeddings associated with those embeddings.

Parallel Token Processing and Context Size

One of the most compelling features of Transformers is that they lend themselves better to parallel computing than previous neural network architectures in language processing. In text generation, we get a first glance at this when looking at how each token is processed. We know from the previous chapter that the tokenizer will break down the text into tokens. Each of these input tokens then flows through its own computation path (that's a good first intuition, at least). We can see these individual processing tracks or streams in Figure 3-8.

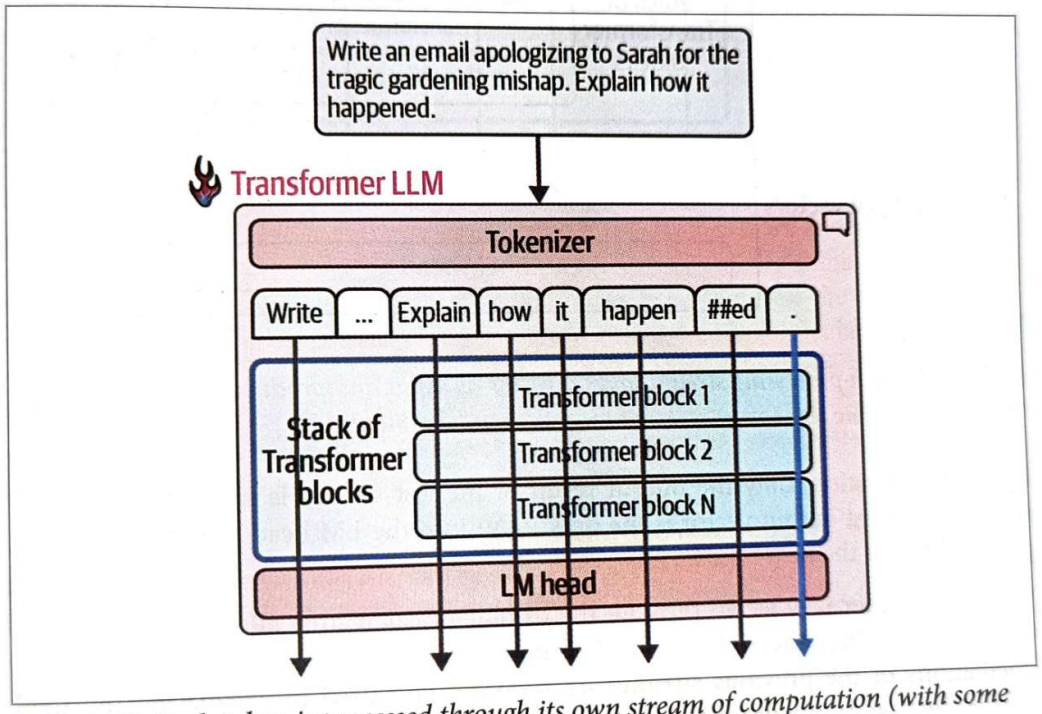


Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

Hyperparameters

Hyperparameters are variables that control different aspects of training. Three common hyperparameters are:

- Learning rate
- Batch size
- Epochs

In contrast, parameters are the variables, like the weights and bias, that are part of the model itself.

Learning rate is a floating-point number you set that influences how quickly the model converges. If the learning rate is too low, the model can take a long time to converge. However, if the learning rate is too high, the model never converges, but instead bounces around the weights and bias that minimize the loss. The goal is to pick a learning rate that's not too high nor too low so that the model converges quickly.

For example, if the gradient's magnitude is 2.5 and the learning rate is 0.01, then the model will change the parameter by 0.025.

The ideal learning rate helps the model to converge within a reasonable number of iterations. In Figure, the loss curve shows the model significantly improving during the first 20 iterations before beginning to converge:

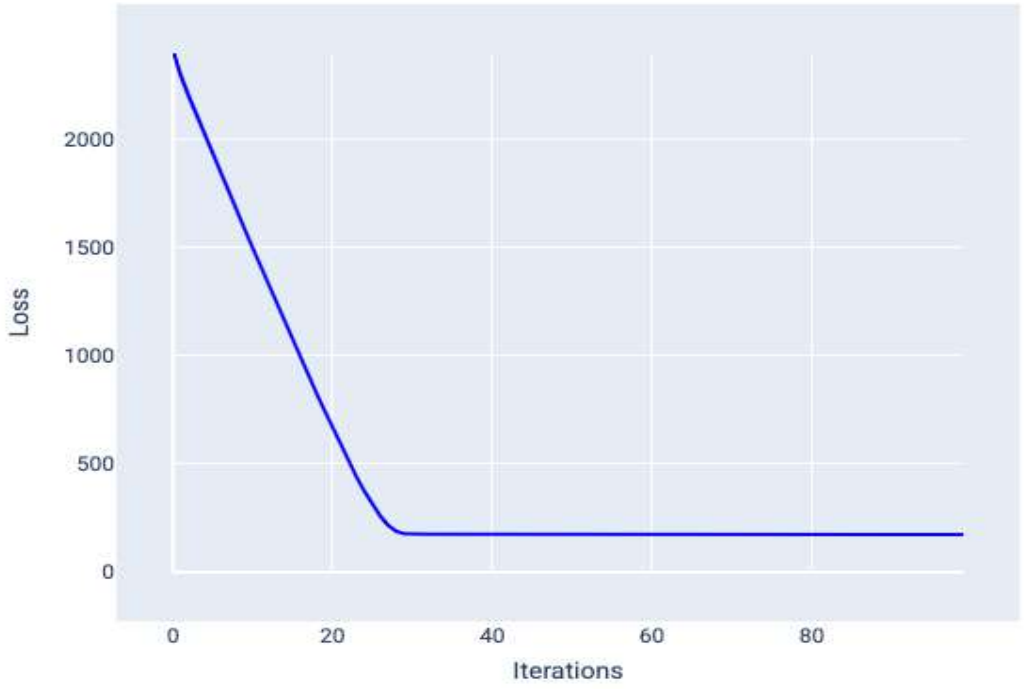


Figure 1. Loss graph showing a model trained with a learning rate that converges quickly.

In contrast, a learning rate that's too small can take too many iterations to converge. In Figure, the loss curve shows the model making only minor improvements after each iteration:

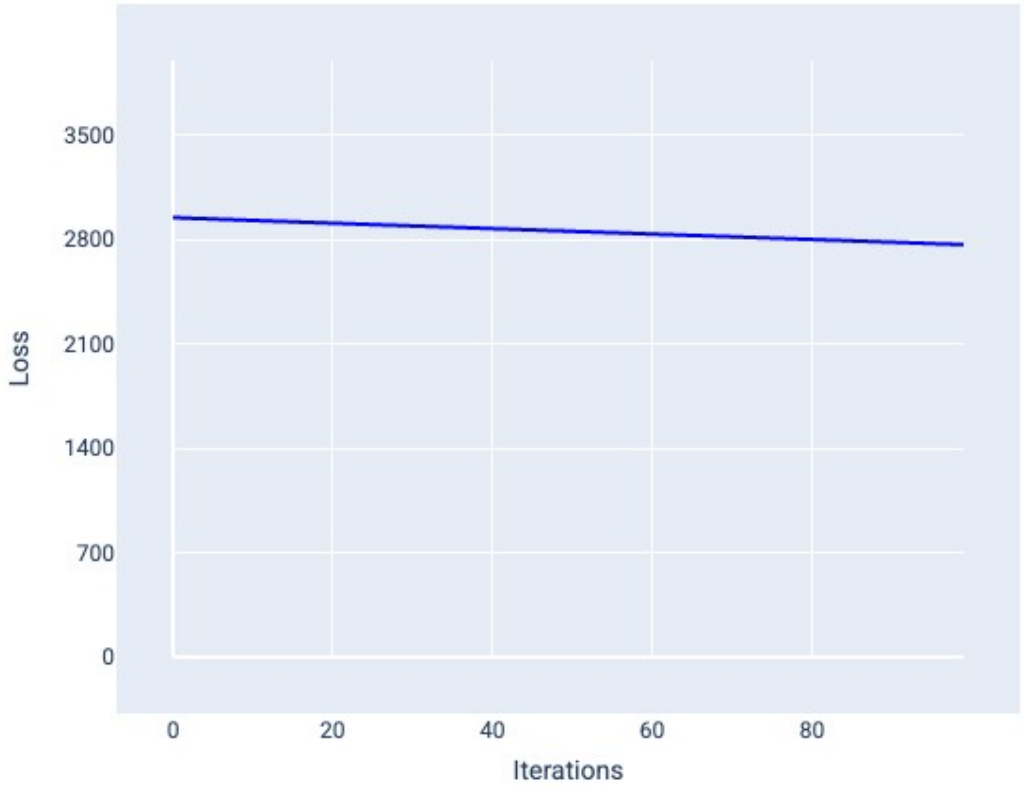


Figure 2. Loss graph showing a model trained with a small learning rate.

Batch size:

Batch size is a hyperparameter that refers to the number of times the model processes before updating its weights and bias.

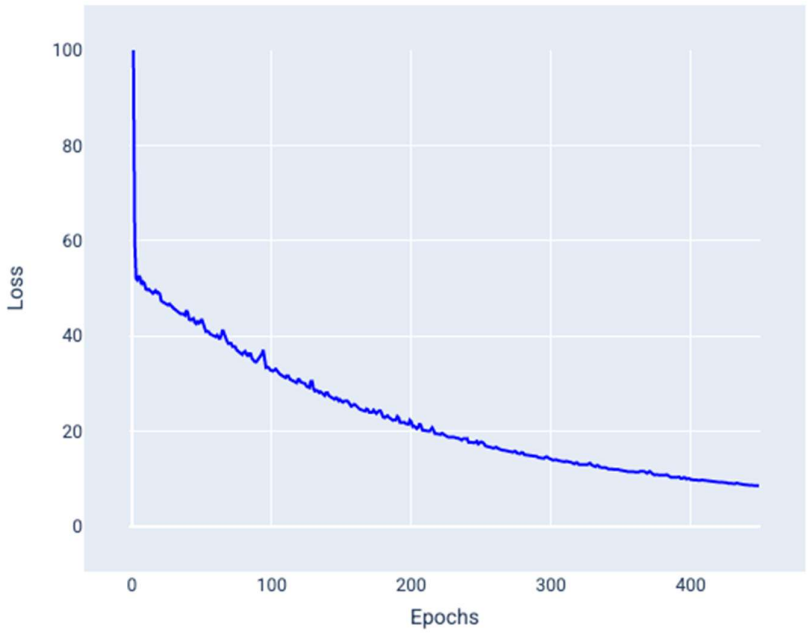


Figure 3. Model trained with mini-batch SGD.

Epochs

During training, an **epoch** means that the model has processed every example in the training set *once*. For example, given a training set with 1,000 examples and a mini-batch size of 100 examples, it will take the model 10 **iterations** to complete one epoch.

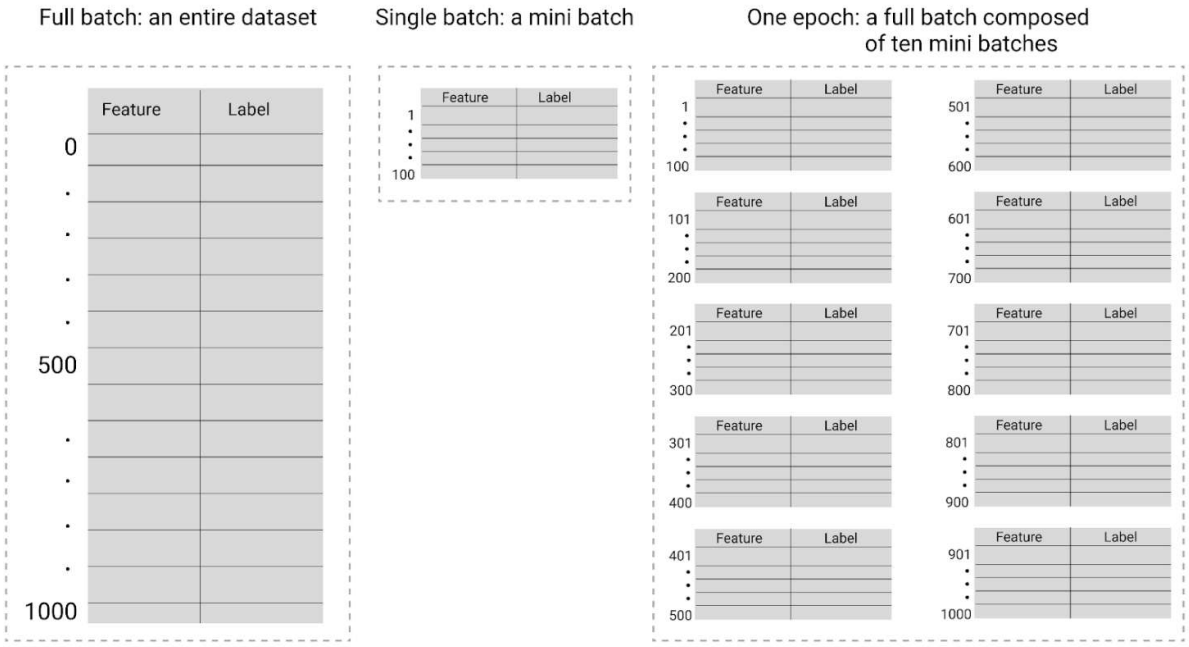


Figure 3. Full batch versus mini batch

EVALUATION METRICS

TP (True Positive), FP (False Positive), and FN (False Negative) are key metrics from a confusion matrix used to evaluate a machine learning model's performance on a binary classification task. A True Positive (TP) is a correct positive prediction, a False Positive (FP) is a positive prediction where the actual value was negative (a Type I error), and a False Negative (FN) is a negative prediction where the actual value was positive (a Type II error).

Precision

Out of all the instances the model predicted as positive, what fraction were actually positive?

Recall

What it is: Out of all the actual positive instances, what fraction did the model correctly identify?

F1-Score

A single metric that combines precision and recall. It is the harmonic mean, which gives a higher weight to lower values. This means the F1-score will be low if either precision or recall is low.

Metric	Formula
True positive rate, recall	$\frac{\text{TP}}{\text{TP+FN}}$
False positive rate	$\frac{\text{FP}}{\text{FP+TN}}$
Precision	$\frac{\text{TP}}{\text{TP+FP}}$
Accuracy	$\frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$
F-measure	$\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$

Example 2.6 Let us map true positives, true negatives, false positives, and false negatives when $y = [1, 1, -1, 1, -1, -1, 1, 1, 1, 1]$ and $\hat{y} = [1, -1, 1, 1, -1, 1, 1, 1, 1, -1]$. Further, based on these counts, we can produce a confusion matrix.

In Table 2.9, we enlist the type of correct/incorrect information captured by the ith index. We can see that TP occurs when $y_i = \hat{y}_i = 1$ and TN at $y_i = \hat{y}_i = -1$. Meanwhile, at indices 2 and 10, we observe the case of $y_i = 1$ but $\hat{y}_i = -1$, causing false negatives. Finally, at indices 3 and 6, we note $y_i = -1$ but $\hat{y}_i = 1$, leading to false positives.

Now, mapping the type count in Table 2.9, we can construct the confusion matrix for the four cases as accounted in Table 2.10.

TABLE 2.9 Mapping true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) for expected labels $y = [1, 1, -1, 1, -1, -1, 1, 1, 1, 1]$ and predicted labels $\hat{y} = [1, -1, 1, 1, -1, 1, 1, 1, 1, -1]$.

Index	1	2	3	4	5	6	7	8	9	10
Expected y	1	1	−1	1	−1	−1	1	1	1	1
Predicted ŷ	1	−1	1	1	−1	1	1	1	1	−1
Type	TP	FN	FP	TP	TN	FP	TP	TP	TP	FN

TABLE 2.10 Confusion matrix for sentiment classification of positive (1) and negative (−1) sentiments for ten sentences. We constructed this from expected labels $y = [1, 1, -1, 1, -1, -1, 1, 1, 1, 1]$ and predicted labels $\hat{y} = [1, -1, 1, 1, -1, 1, 1, 1, 1, -1]$. The tabulations follow from mapping in Table 2.9.

	Predicted	
Actual	**Positive**	**Negative**
Positive	5 (TP)	2 (FN)
Negative	2 (FP)	1 (TN)

Word Embedding

Word embedding is a technique in Natural Language Processing (NLP) that represents words as numerical vectors (lists of numbers) in a continuous vector space, where similar words have similar vector representations.

TABLE 3.1 The term-document matrix for five words in four magazine documents. Each cell contains the number of times a word (row) appears in the document (column).

	Fruit Market Overview	Tropical Fruit Guide	Tech Trends	Healthy Tech Lifestyle
apple	10	0	0	8
banana	0	12	0	0
fruit	7	5	0	5
technology	0	0	15	3
software	0	0	10	4

Example 3.1 Let us determine whether 'fruit' is more similar to 'apple' or 'technology' using the example from Table 3.1.

The vector representations for the words are: 'apple' = [10, 0, 0, 8], 'fruit' = [7, 5, 0, 5], and 'technology' = [0, 0, 15, 3]. Using Equation (3.4), we calculate the cosine similarity by:

$$\begin{aligned}\cos(apple, fruit) &= \frac{apple \cdot fruit}{\| apple \| \| fruit \|} \\ &= \frac{(10\times 7)+(0\times 5)+(0\times 0)+(8\times 5)}{\sqrt{10^2+0^2+0^2+8^2}\sqrt{7^2+5^2+0^2+5^2}} \\ &= \frac{100}{12.81\times 9.95} \\ &= 0.862\end{aligned}$$

$$\begin{aligned}\cos(technology, fruit) &= \frac{technology \cdot fruit}{\| technology \| \| fruit \|} \\ &= \frac{(0\times 7)+(0\times 5)+(15\times 0)+(3\times 5)}{\sqrt{0^2+0^2+15^2+3^2}\sqrt{7^2+5^2+0^2+5^2}} \\ &= \frac{15}{15.30\times 9.95} \\ &= 0.098\end{aligned}$$

These results indicate that in this vector space representation, 'fruit' shows a significantly higher semantic similarity to 'apple' than to 'technology'. This outcome aligns with the intuitive conceptual relationships among these terms.

Term Frequency-Inverse Document Frequency

The co-occurrence matrices discussed above use raw frequency counts to represent relationships between words and documents or between words and other words. However, this approach has significant limitations. For example, consider the vectors: 'apple' = [10, 0, 0, 8], 'fruit' = [7, 5, 0, 5], and 'technology' = [0, 0, 15, 3].

These vectors illustrate some shortcomings of using raw co-occurrence frequencies for word representation. For instance, the high frequency in the third dimension (15) of 'technology' might dominate the vector and potentially overshadow other meaningful relationships. In short, the raw frequency count-based method makes frequent words, such as articles or common verbs, more significant than they are. Conversely, less frequent but potentially more meaningful terms might be underrepresented.

To address these limitations, we introduce an approach called *Term Frequency-Inverse Document Frequency* (TF-IDF). TF-IDF assigns weight to a word in a document based on its statistical significance within the document and across a collection of documents or corpus. It consists of *Term Frequency* (TF) and *Inverse Document Frequency* (IDF).

1. **Term Frequency (TF):** TF measures how frequently a term occurs in a document. It is typically calculated as:

$$TF_{t,d} = f_{t,d} \tag{3.5}$$

where $f_{t,d}$ is the raw frequency of term t in document d. Since the number of documents can be very large, a logarithmic scaling can be used to dampen the effect of large frequency differences:

$$TF_{t,d} = \begin{cases} 1+\log(f_{t,d}), & \text{if } f_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

2. **Inverse Document Frequency (IDF):** IDF measures how important a term is across the entire corpus. It is calculated as:

$$\text{IDF}_t = \log\left(\frac{N}{n_t}\right)$$

where N is the total number of documents in the corpus, and n_t is the number of documents containing the term t.

TF-IDF Calculation. TF-IDF is a weighted score that combines the local importance of a term within a document (TF) with its global significance across the corpus (IDF). This statistical measure aims to reflect how important a word is to a document in a collection or corpus. TF-IDF is calculated by multiplying the TF and the IDF of a term:

$$\text{TF-IDF}_{t,d,D} = \text{TF}_{t,d} \times \text{IDF}_{t,D}$$

Here, $\text{TF-IDF}_{t,d,D}$ is the TF-IDF of term t in document d over the corpus D. A higher TF-IDF score for a term signifies that it is both frequent within the document and rare enough in the corpus, thus making it more relevant to the document's content. TF-IDF balances the frequency of a term in a specific document with its rarity across all documents, providing a more nuanced measure of term importance than raw frequency alone.

Vector Representation Using TF-IDF. The TF-IDF score of each term or vocabulary word in the term-document matrix can also be used to represent a document or the word. Consider the following example:

Example 3.2 Consider a small corpus of 4 documents:

1. Doc 1: '*The quick brown fox jumps over the lazy dog*'.
2. Doc 2: '*The lazy dog sleeps all day*'.
3. Doc 3: '*The quick brown fox hunts in the forest*'.
4. Doc 4: '*A lazy afternoon in the forest*'.

Let us consider these four documents and the terms[1] 'the', 'lazy', 'dog', 'quick', and 'fox'. In Table 3.2, each value is the TF score. For example, the TF for '*lazy*' with respect to Doc 2 in Table 3.2 is computed by: *TF*('*lazy*', Doc 2) = *count*('*lazy*', Doc 2) = 1. Next, we compute the IDF score for each term q:

- IDF('the') = $\log_2(4/3) \approx 0.415$
- IDF('lazy') = $\log_2(4/3) \approx 0.415$
- IDF('dog') = $\log_2(4/2) = 1$
- IDF('quick') = $\log_2(4/2) = 1$
- IDF('fox') = $\log_2(4/2) = 1$

We then calculate the TF-IDF score for each term. We do this by multiplying each TF value by its corresponding IDF, as presented in Table 3.3. For instance, in Table 3.3, the TF-IDF value for the term 'lazy' in Doc 1 is calculated as:

$$\begin{aligned}\text{TF-IDF('lazy', Doc 1)} &= \text{TF('lazy', Doc 1)} \times \text{IDF('lazy')} \\ &= 1 \times 0.415 = 0.415\end{aligned}$$

TABLE 3.2 The TF matrix for terms 'the', 'lazy', 'dog', 'quick', and 'fox' with respect to four documents: Doc 1, Doc 2, Doc 3, and Doc 4.

	the	lazy	dog	quick	fox
Doc 1	2	1	1	1	1
Doc 2	1	1	1	0	0
Doc 3	2	0	0	1	1
Doc 4	1	1	0	0	0

TABLE 3.3 The TF-IDF matrix for the terms, 'the', 'lazy', 'dog', 'quick', and 'fox' with respect to the four documents: Doc 1, Doc 2, Doc 3, and Doc 4.

	the	lazy	dog	quick	fox
Doc 1	0.830	0.415	1	1	1
Doc 2	0.415	0.415	1	0	0
Doc 3	0.830	0	0	1	1
Doc 4	0.415	0.415	0	0	0

Each row vector in Table 3.3 represents the corresponding document. For instance, Doc 2 is represented as [0.415, 0.415, 1, 0, 0]. Each column vector can be used to represent the word vector. For example, the word 'quick' is represented as [1, 0, 1, 0].

Word Embedding–based Generative Models, GloVe and Skip-Gram Models

The **Skip-gram model** is a neural network architecture used for learning word embeddings by predicting the surrounding context words from a given target word. It is the inverse of the CBOW model, where the input is the target word and the output is the probability of words appearing in its context window. This process allows the model to capture semantic relationships between words.

Skip-Gram Model

The Skip-Gram model is one of the two primary models introduced by Word2Vec (Mikolov et al. 2013a,b). The fundamental idea behind the Skip-Gram model is to select a central word from a sentence and use it to predict its surrounding context words within a specified window. This process helps train a classifier to learn the embedding weights, which can later be used as dense vector representations of words.

For instance, consider the example sentence: '*I live in Abu Dhabi*'. If we choose the word 'live' as the centre word and set a context window of ±1, the model would attempt to predict the words 'I' and 'in' as the context words surrounding 'live'. This setup is illustrated in Figure 3.2. The Skip-Gram model thus leverages the prediction of context words to refine the word embeddings, with the aim of capturing semantic and syntactic relationships between words.

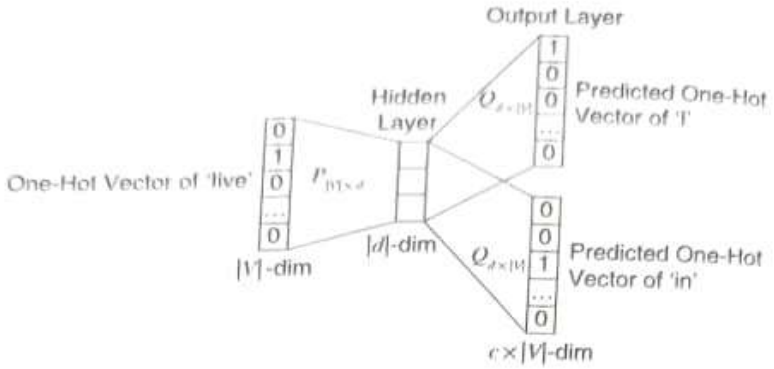


FIGURE 3.2 An illustration of the Skip-Gram model to predict the context words 'I' and 'in' given the input centre word 'life'. The centre word is converted to a one-hot encoder vector of dimension $|V|$, and the output final layers are the one-hot encoding vectors for the two predicted outputs. The model learns the weight matrices, P and Q, which are used as the embedding. Each row of the P matrix is a d-dimensional embedding.

Preliminaries. Let us begin with some foundational concepts. In the Skip-Gram model, words in a sentence are represented as one-hot vectors, which are used to classify whether any word in the vocabulary is a neighbour of the centre word. The centre word is represented by a one-hot vector x, while the context words are denoted by y^j. To facilitate this process, we introduce two matrices: $P \in \mathbb{R}^{|V| \times d}$ and $Q \in \mathbb{R}^{d \times |V|}$. Here, P represents the input word matrix and Q represents the output word matrix. The variable d stands for the embedding dimension, while $|V|$ denotes the vocabulary size, which corresponds to the size of the one-hot vectors. In the input matrix P, the word w_i is associated with a d-dimensional embedding vector located in the ith row of P. This row vector is denoted by p_i. Similarly, in the output matrix Q, the word w_j is associated with a d-dimensional embedding vector located in the jth column of Q, denoted by q_j. The Skip-Gram model aims to learn both the input (p_i) and output (q_j) word vectors for each word w_i. Essentially, the model seeks to learn two embedding vectors, p and q, for every word in the vocabulary. The following steps outline the process for learning these embedding weights:

1. **Generation of One-Hot Input Vector:** Initially, the centre word is represented as a one-hot vector. For a given centre word x, we create a $|V|$-dimensional one-hot vector, where $|V|$ is the vocabulary size. This vector has a value of 1 at the index corresponding to the centre word w in the vocabulary and 0 elsewhere. Mathematically, it is represented as:

$$x = [0, 0, \ldots, 1, \ldots, 0]^T$$

or

$$x = \begin{cases} 1 & \text{if } i = \text{index}(w) \\ 0 & \text{otherwise} \end{cases}$$

Here, index(w) refers to the index of the centre word w in the vocabulary.

2. **Get the Embedded Vector for the Centre Word:** Next, we multiply the one-hot vector x by the input word matrix P to obtain the embedding vector p_c for the centre word. This operation can be expressed as:

$$p_c = P^T x \in \mathbb{R}^d$$

Here, P^T is the transpose of the input word matrix P, and p_c is a d-dimensional vector that represents the embedding of the centre word. This multiplication effectively selects the column in P corresponding to the centre word's index, yielding its embedding vector.

3. **Getting the Similarity Between the Centre and Context Words:** The intuition behind the Skip-Gram model is that the likelihood of a word appearing close to a target word depends on the similarity of their embedding vectors. To measure this similarity, we can use the dot product, where a higher dot product indicates a greater degree of similarity between the words. To compute the similarity score, we multiply the centre word vector p_c by the output word matrix Q, resulting in the score vector z:

$$z = Q^T p_c \tag{3.6}$$

Each element in z represents the dot product score between the context word and the centre word, reflecting the model's prediction of how likely each word is to appear in the context of the centre word.

4. **Turn Similarity Scores into Probability:** The similarity scores obtained in the previous step are not sufficient for our prediction task for several reasons. First, these scores are not probabilities; they range unboundedly between $-\infty$ and $+\infty$. Second, the scores do not account for relative scores across all the words in the vocabulary. For our task, we need a concrete probability that represents the likelihood of a context word given a centre word.

To achieve this, we use the *softmax function* to convert these scores into a probability distribution:

$$\hat{y} = \text{softmax}(z) = \frac{\exp(z)}{\sum_{j=1}^{|V|} \exp(z_j)}$$

Here, $\hat{y}_{c-m}, \ldots, \hat{y}_{c-1}, \hat{y}_{c+1}, \ldots, \hat{y}_{c+m}$ represent the probabilities of observing each context word within a window of size m around the centre word. These probabilities correspond to the respective elements in the $\hat{y}$ vector, making them suitable for the prediction task.

5. **Objective Function:** To learn the Q and P matrices, we need an objective function that measures how well the predicted probabilities $\hat{y}_{c-m}, \ldots, \hat{y}_{c-1}, \hat{y}_{c+1}, \ldots, \hat{y}_{c+m}$ match the true probabilities $y_{c-m}, \ldots, y_{c-1}, y_{c+1}, \ldots, y_{c+m}$. We use an independence assumption to simplify the model by assuming that, given the centre word, all output words are independent of each other. This allows us to break down the joint probability into a product of individual probabilities.

The objective function J that we aim to minimise is the negative log-likelihood of the true context words, given the centre word:

$$\begin{aligned}
\text{minimise } J &= -\log P(w_{c-m},\ldots,w_{c-1},w_{c+1},\ldots,w_{c+m} \mid w_c) \\
&= -\log \prod_{j=0,j\neq m}^{2m} P(w_{c-m+j} \mid w_c) \\
&= -\log \prod_{j=0,j\neq m}^{2m} P(q_{c-m+j} \mid p_c) \\
&= -\log \prod_{j=0,j\neq m}^{2m} \frac{\exp(q_{c-m+j}^T p_c)}{\sum_{k=1}^{|V|} \exp(q_k^T p_c)} \\
&= -\prod_{j=0,j\neq m}^{2m} q_{c-m+j}^T p_c + 2m \log \exp(q_k^T p_c)
\end{aligned}$$

Now, Stochastic Gradient Descent (SGD) (as discussed in Chapter 2) can be used to update all the relevant word vectors q_c and p_c.

Global Vectors for Word Representation

We discussed word representation techniques using count-based methods (like co-occurrence matrices) and predictive methods (like Word2Vec). Count-based methods often fall short in tasks such as analogy, indicating that their vector representations are suboptimal.

Global Vectors for Word Representation (GloVe) combine the strengths of both approaches by leveraging the global statistical information captured by word-word co-occurrence matrices while also benefiting from the local context window approach used in Word2Vec.

The GloVe Model

In this section, we will delve into the inner workings of the GloVe model. Before proceeding, let us establish some notations. GloVe uses a global word-word co-occurrence matrix, denoted by X. Each element in this matrix, X_{ij}, represents the count of occurrences of word j given the context word i. Now, let us define $X_i = \Sigma_k X_{ik}$ as the total count of all words in the context of word i, which is the sum of all co-occurrence counts across the entire corpus when i is the context word. Based on this, we can calculate the probability of word j given the context word i as follows:

$$P_{ij} = P(j \mid i) = \frac{X_{ij}}{X_i}$$

Let us consider an example of how co-occurrence probabilities can be utilised to derive certain aspects of meaning.

.3 Let us consider two words, i and j, that represent movie genres: i = *comedy* and j = *horror*. We will examine the relationship of the words i and j by studying the ratio of their co-occurrence probabilities with some probe words k. In the example shown in Table 3.4, we use k = *laughter* for words related to *comedy* but not with *horror*, and the ratio of P_{ik}/P_{jk} or $P(laughter|comedy)/P(laughter|horror)$ is large (8.89) as expected. Similarly, k = *scary* will be more related to *horror*, and the ratio P_{ik}/P_{jk} is low (0.05). And, the words that are either related to both *comedy* or *horror* or totally unrelated will have a ratio close to one. This example demonstrates that a ratio of probabilities can effectively distinguish words relevant to one genre but not the other (*laughter*, *scary*) and words relevant to either both (*popcorn*) or none (*algebra*). In short, this approach of taking the ratio of co-occurrence probabilities provides a more effective and distinctive measure as against the raw probabilities.

TABLE 3.4 An illustration of co-occurrence probabilities and ratios for the target words *comedy* and *horror* with *k* probe words as context from a hypothetical corpus.

k	$P(k\|comedy)$	$P(k\|horror)$	$P(k\|comedy)/P(k\|horror)$
laughter	0.0080	0.0009	8.89
scary	0.0005	0.0100	0.05
popcorn	0.0050	0.0045	1.11
algebra	0.0001	0.0001	1.00

Objective Function. In Example 3.3, we noticed that the ratio of probabilities is a good starting point for learning word vectors as compared to the raw probabilities. We also came across how P_{ik}/P_{jk} score reflects when the words i and j are associated with the word k. We want to encode this insight into the vector space through a function F as follows:

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{3.10}$$

where, $w \in \mathbb{R}^d$ is the word vector when the word is the centre or the main word, while and $\tilde{w} \in \mathbb{R}^d$ is the word vector for the context. Since vector spaces are linear, we can re-model F in Equation (3.10) with vector differences as in Equation (3.11):

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{3.11}$$

Now, the RHS of Equation (3.11) is a scalar, and the LHS is a vector. Hence, this needs to be addressed to maintain the linear structure by taking the dot product of the arguments as below:

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{3.12}$$

In the word-word co-occurrence matrices, the word and context can be interchangeable. So, we need to make F to be invariable even with the interchange of $w \leftrightarrow \tilde{w}$ and $X \leftrightarrow X^T$. For this, we require F to follow *homomorphism* between the groups $(\mathbb{R}, +)$ and $(\mathbb{R} > 0, \times)$, i.e.,

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)} \tag{3.13}$$

Equation (3.13) is solved using Equation (3.12) as:

$$F(w_i^T \tilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i} \tag{3.14}$$

Now, one solution of Equation (3.13) is $F = exp$, or,

$$w_i^T \tilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i) \tag{3.15}$$

In Equation (3.15), $\log(X_i)$ breaks the symmetry. However, it is independent of k and can be incorporated into a bias term b_i for the main word vector w_i. Finally, an additional bias $\tilde{b}_k$ for w_k is introduced to restore the symmetry,

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik}) \tag{3.16}$$

The GloVe modelling with Equation (3.16) has certain drawbacks, such as when two words never occur, the count is zero, and log (0) is undefined. A naïve approach would be to add a small integer value so that $\log X_{ik}$ becomes $\log(1 + X_{ik})$. This maintains the sparsity of X while avoiding the mathematical divergence. However, this approach gives equal weightage to all the co-occurrences regardless of their frequency. To address this issue, Equation (3.16) can be reformulated to a weighted least square regression model. The final weighted objective function of the GloVe is given by:

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2, \tag{3.17}$$

where $|V|$ is the vocabulary size and $f(X_{ij})$ is the weighing function. In the original GloVe paper (Pennington et al. 2014), the weighing function $f(X_{ij})$ is computed using:

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise.} \end{cases}$$

Pennington et al. (2014) set $x_{\max} = 100$ and $\alpha = 3/4$.

Model Learning. Consider Skip-Gram, where the task is to model the probability distribution Q_{ij} for j to appear given the context i. Now, this probability is calculated using a softmax,

$$Q_{ij} = \frac{\exp(w_i^T \tilde{w}_j)}{\sum_{k=1}^{|V|} \exp(w_i^T \tilde{w}_k)}.$$

Skip-Gram attempts to maximise the log probability as a context window scans over the corpus, and the global objective function can be written as:

$$J = -\sum_{\substack{i \in \text{corpus} \\ j \in \text{context}(i)}} \log Q_{ij} \tag{3.18}$$

Computing the softmax for each term in Equation (3.18) is costly. Therefore, GloVe uses the precomputed co-occurrence matrix X to group terms with the same i and j values.

$$J = -\sum_{i=1}^{|V|}\sum_{j=1}^{|V|} X_{ij} \log Q_{ij} \tag{3.19}$$

This can be further deduced to:

$$J = -\sum_{i,j=1}^{|V|} X_i P_{ij} \log Q_{ij} \tag{3.20}$$

This cross-entropy loss itself is costly because it needs a normalised distribution of Q, which is expensive due to the summation over the entire vocabulary. Therefore, we can use the least square objective where the normalisation of P and Q can be skipped,

$$\hat{J} = \sum_{i,j} X_i (\hat{P}_{ij} - \hat{Q}_{ij})^2 \tag{3.21}$$

where $\hat{P}_{ij} = X_{ij}$ and $\hat{Q}_{ij} = \exp(w_i^T \tilde{w}_j)$ are the unnormalised distributions. Here, X_{ij} can take large values; so instead, logarithms of $\hat{P}$ and $\hat{Q}$ are used,

$$\hat{J} = \sum_{i,j} X_i (\log \hat{P}_{ij} - \log \hat{Q}_{ij})^2 = \sum_{i,j} X_i (w_i^T \tilde{w}_j - \log X_{ij})^2 \tag{3.22}$$

Here, the weighting factor X_i is not guaranteed to be optimal. Hence, a more general weighting function which depends on the context word can be used:

$$\hat{J} = \sum_{i,j} f(X_{ij}) (w_i^T \tilde{w}_j - \log X_{ij})^2 \tag{3.23}$$

This is the final weighted least squared objective function of GloVe, which is used to train the word vectors w_i for the word i.

Statistical Language Model

A Statistical Language Model estimates the probability of a sequence of words occurring in a language. It helps a system understand how likely a sentence is according to learned language patterns.

4.1.1 The Conditional Probability

In probability theory, conditional probability quantifies the likelihood of an event A occurring, given another event B has already taken place or vice versa. For example, let us assume event A represents *The FeverEase capsules that are widely used for treating common fevers, and event B represents A medicine shop named MediHall that is capable of selling 1000 FeverEase capsules each day.* Then $P(A|B)$ indicates the probability of FeverEase being widely popular due to its high sales, while $P(B|A)$ represents the probability of FeverEase's high sales due to its effectiveness in treating common fevers. The conditional probability of A given B is written as:

$$P(A|B) = \frac{n(A \cap B)}{n(B)} = \frac{n(A \cap B)}{N} \times \frac{N}{n(B)} = \frac{P(A \cap B)}{P(B)} \quad (4.1)$$

Here $n(A \cap B)$ denotes the number of events of A and B occurring together, whereas $n(B)$ is the number of events in B. Now, dividing the numerator and the denominator in the second step with N (the total number of events in A and B together) yields $P(A \cap B)$ and $P(B)$. Similarly,

$$P(B|A) = \frac{P(A \cap B)}{P(A)} \quad (4.2)$$

Equating Equations (4.1) and (4.2), we get

$$P(A \cap B) = P(A|B) \times P(B) = P(B|A) \times P(A)$$

Therefore,

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \text{ or } P(B|A) = \frac{P(A|B) \times P(B)}{P(A)} \quad (4.3)$$

Chain Rule

The chain rule of probability (also known as the probability multiplication rule) allows us to express the joint probability of multiple random variables in terms of conditional probabilities.

For two events $A$and B:

$$P(A, B) = P(A) \cdot P(B \mid A)$$

This means the probability that **both A and B** occur equals the probability that **A** occurs multiplied by the probability that **B** occurs given that **A** has already happened.

4.1.2 The Chain Rule of Probability

In probability theory, the chain rule is also known as the general product rule. It provides a method to calculate the probability of the joint distribution of multiple dependent random variables. Let us assume $A_0, A_1, \ldots, A_n$ are n different events, not necessarily independent. We need to estimate the probability of all events occurring simultaneously, i.e., $P(A_0 \cap A_1 \cap \ldots \cap A_n)$. This can be achieved through the use of conditional probabilities and the chain rule:

$$P(A_0 \cap A_1 \cap \ldots \cap A_n)$$
$$= P(A_n|A_{n-1} \cap \ldots \cap A_0) \times P(A_{n-1} \cap \ldots \cap A_0) \text{ (using Equation 4.3)}$$
$$= P(A_n|A_{n-1} \cap \ldots \cap A_0) \times P(A_{n-1}|A_{n-2} \cap A_{n-3} \cap \ldots \cap A_0) \times P(A_{n-2} \cap A_{n-3} \cap \ldots \cap A_0)$$
$$= P(A_n|A_{n-1} \cap \ldots \cap A_0) \times \ldots \times P(A_{n-i}|A_{n-i+1} \cap A_{n-i+2} \cap \ldots \cap A_0) \times \ldots \times P(A_1|A_0) \times P(A_0)$$

or rewriting the previous step in the reverse order,

$$= P(A_0) \times P(A_1|A_0) \times P(A_2|A_1 \cap A_0) \times P(A_3|A_2 \cap A_1 \cap A_0) \times \ldots \times P(A_n|A_{n-1} \cap \ldots \cap A_0)$$

$$= \prod_{i=0}^{n} P(A_i|A_{i-1} \cap \ldots \cap A_0)$$

Therefore,

$$P(A_0 \cap A_1 \cap \ldots \cap A_n) = \prod_{i=0}^{n} P(A_i|A_{i-1} \cap \ldots \cap A_0) = \prod_{i=0}^{n} P(A_i|A_{i-1}, \ldots, A_0) \quad (4.4)$$

Following the above-mentioned generalised chain rule of probability, Equation (4.4), the joint probabilistic distribution over a sequence of words $\{W_0, W_1, W_2, \ldots, W_n\}$ can be expressed as follows:

$$P(W_0 W_1 W_2 \ldots W_n) = \prod_{i=0}^{n} P(W_i|W_{i-1}, \ldots, W_0) = \prod_{i=0}^{n} P(W_i|W_0 : W_{i-1}) \quad (4.5)$$

Here, $P(W_i|W_0, W_1, \ldots, W_{i-1})$ can be calculated using maximum likelihood estimate as $\frac{\text{count}(W_0, W_1, \ldots, W_i)}{\text{count}(W_0, W_1, \ldots, W_{i-1})}$, where count $(W_0, W_1, \ldots, W_i)$ = the number of occurrences of the sequence $(W_0, W_1, \ldots, W_i)$ in the training corpus.

Example 4.1

Find the expression for the joint distribution of the words in the sentence, *'The monsoon season has begun'*.

Solution

Here, we will see each token at ith position, i.e., W_i using $W_{0:i-1}$ as its context in the sentence.

$$P(\text{The monsoon season has begun})$$
$$= P(\text{The}) \times P(\text{monsoon}|\text{The}) \times P(\text{season}|\text{The monsoon})$$
$$\times P(\text{has}|\text{The monsoon season}) \times P(\text{begun}|\text{The monsoon season has}) \quad (4.6)$$

4.1.4 Unigram Language Model

As the term *unigram* suggests, the probabilistic value of a word, W_i, in a sequence of words $\{W_0, W_1, W_2, W_3, \ldots, W_n\}$ depends on itself only. This is also known as the zero-order Markov process. A unigram language model considers no previous context for W_i. The unigram probability of a word W_i can be estimated as the fraction of times it has appeared in the training corpus, w.r.t. the total number of words available in the corpus,

$$P(W_i) = \frac{\text{count}(W_i)}{N}$$

where N = total number of words in the training corpus. Also, the probability of a sequence of words or a sentence can be expressed as a multiplication of the probabilistic estimation of each word, i.e.,

$$P(W_0 W_1 W_2 \ldots W_n) = P(W_0) \times P(W_1) \times P(W_2) \times \cdots \times P(W_i) \times \cdots \times P(W_n) = \prod_{i=0}^{n} P(W_i)$$

Example 4.3 Find the probability of the sentence, '*The lazy cat sells sea shells*' using a unigram language model and the training data (case-insensitive) given below.

Sentence 1: *The cat sat on the mat*
Sentence 2: *A quick brown fox jumps over the lazy dog*
Sentence 3: *She sells sea shells by the sea shore*
Sentence 4: *He reads books every evening before bed*
Sentence 5: *The sun rises in the east and sets in the west*

Solution

There are a total of 41 words (N) in the given corpus. Let us calculate the probability of individual words, *The*, *lazy*, *cat*, *sells*, *sea*, and *shells* of the given input sentence.

$$P(The) = \text{count}(The)/N = 7/41,\ P(lazy) = \text{count}(lazy)/N = 1/41$$
$$P(cat) = \text{count}(cat)/N = 1/41,\ P(sells) = \text{count}(sells)/N = 1/41$$
$$P(sea) = \text{count}(sea)/N = 2/41,\ P(shells) = \text{count}(shells)/N = 1/41$$

Now, we can use the above probabilities to estimate the probability of the input sentence:

$$P(The\ lazy\ cat\ sells\ sea\ shells)$$
$$= P(The) \times P(lazy) \times P(cat) \times P(sells) \times P(sea) \times P(shells)$$
$$= \frac{7}{41} \times \frac{1}{41} \times \frac{1}{41} \times \frac{1}{41} \times \frac{2}{41} \times \frac{1}{41} = \frac{14}{41^6} \approx 2.947 \times 10^{-9}$$

4.1.5 Bigram Language Model

As the term *bigram* suggests, the probability of a word, W_i, in a sequence $\{W_1, W_2, W_3, \ldots, W_n\}$ depends only on the previous word, i.e., W_{i-1}. In short, a bigram language model considers a previous context of length one. The probabilistic value of a word, W_i, can be expressed as,

$$P(W_i|W_{i-k}, W_{i-k+1}, \ldots, W_{i-1}) \approx P(W_i|W_{i-1})$$

and

$$P(W_i|W_{i-1}) = \frac{\text{count}(W_{i-1}, W_i)}{\text{count}(W_{i-1})}$$

Example 4.4

Find the probability of the sentence, '*The cat sells sea shells*' using a bigram language model and the training data (case-insensitive) given below.

Sentence 1: *The cat sat on the mat*
Sentence 2: *The cat sells a sea shell to the lazy dog*
Sentence 3: *She sells sea shells by the sea shore*
Sentence 4: *He reads books every evening before bed*
Sentence 5: *The sun rises in the east and sets in the west*

Solution

Let us introduce the start token '<*START*>' at the beginning of each sentence. Therefore, all the sentences will appear as

Sentence 1: <*START*> *The cat sat on the mat*
Sentence 2: <*START*> *The cat sells a sea shell to the lazy dog*
Sentence 3: <*START*> *He sells sea shells by the sea shore*
Sentence 4: <*START*> *He reads books every evening before bed*
Sentence 5: <*START*> *The sun rises in the east and sets in the west*

Target input **:** <*START*> *The cat sells sea shells*

Now, we need to calculate the likelihood of each bigram token for the input sentence, '*<START> The cat sells sea shells*'. Note that the likelihood of the start token is 1.

$$P(\text{<START> The cat sells sea shells})$$
$$= P(\text{<START>}) \times P(\text{The} \mid \text{<START>}) \times P(\text{cat} \mid \text{The}) \times P(\text{sells} \mid \text{cat}) \times P(\text{sea} \mid \text{sells}) \times P(\text{shells} \mid \text{sea})$$

$$P(\text{<START>}) = 1$$

$$P(\text{The} \mid \text{<START>}) = \frac{\langle \text{<START>}, \text{The} \rangle \text{ appears three times}}{\text{'<START>' appears five times}} = \frac{3}{5}$$

$$P(\text{cat} \mid \text{The}) = \frac{\langle \text{The}, \text{cat} \rangle \text{ appears two times}}{\text{'The' appears eight times}} = \frac{1}{4}$$

$$P(\text{sells} \mid \text{cat}) = \frac{\langle \text{cat}, \text{sells} \rangle \text{ appears one times}}{\text{'cat' appears two times}} = \frac{1}{2}$$

$$P(\text{sea} \mid \text{sells}) = \frac{\langle \text{sells}, \text{sea} \rangle \text{ appears one times}}{\text{'sells' appears two times}} = \frac{1}{2}$$

$$P(\text{shells} \mid \text{sea}) = \frac{\langle \text{sea}, \text{shells} \rangle \text{ appears one times}}{\text{'sea' appears 2 times}} = \frac{1}{2}$$

$$\Rightarrow P(\text{The cat sells sea shells}) = 1 \times \frac{3}{5} \times \frac{1}{4} \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{3}{160} = 0.01875$$

Note that the likelihood of the initial term, w_0 of a sentence s, $P(w_0)$, is different from the w_0's unigram probability. It shall be indifferent with or without the start token *<START>*.

For example, we have already seen the likelihood of the phrase $P(\text{<START> The})$ as $P(\text{<START>}) \times P(\text{The} \mid \text{<START>})$, which is $1 \times \frac{3}{5} = \frac{3}{5}$. Similarly, $P(\text{The})$ can be expressed as the number of occurrences of the initial word out of all sentences, i.e., 3/5.

4.2.2 Smoothing

The smoothing technique allows us to avoid the zero probability mentioned earlier by 'stealing' some probability mass from the tokens having high probability and distributing it among those tokens which do not have sufficient probability (or probability of zero). The simple smoothing technique is *add-1* smoothing or *Laplace* smoothing, in which the original count of an n-gram is increased by one. In particular, the conditional probability after smoothing can be written as:

$$P_{add-1}(W_i|W_{i-1}) = \frac{\text{count}(W_{i-1}, W_i) + 1}{\text{count}(W_{i-1}) + |V|}$$

For an unseen token, the count(W_{i-1}, W_i) is 0, but due to adding 1 to the numerator, it does not reduce to zero. The denominator is also increased by $|V|$, the size of the vocabulary, to make sure that $\Sigma_{v \in V} P_{add-1}(W_v|W_{i-1}) = 1$.

4.6 Find the probability of the sentence, '*The furry cat sells sea shells*' using a bigram language model and the training data.

Sentence 1: *The cat sat on the mat*
Sentence 2: *A quick brown fox jumps over the lazy dog*
Sentence 3: *She sells sea shells by the sea shore*
Sentence 4: *He reads books every evening before bed*
Sentence 5: *The sun rises in the east and sets in the west*

Solution

As we have seen earlier,

$$P(\textit{The furry cat sells sea shells}) = P(\textit{furry}|\textit{The}) \times P(\textit{cat}|\textit{furry}) \times P(\textit{sells}|\textit{cat}) \times P(\textit{sea}|\textit{sells}) \times P(\textit{shells}|\textit{sells})$$

As already noticed, the term '*furry*' is an unseen word. It sets the value of $P(\textit{furry}|\textit{The})$ and $P(\textit{cat}|\textit{furry})$ to zero. But if add-1 smoothing is used, assuming $|V|$ to be 46, we will get

$$P_{add-1}(\textit{furry}|\textit{The}) = \frac{\text{count}(\textit{The furry}) + 1}{\text{count}(\textit{The}) + 46} = \frac{0+1}{7+46} = \frac{1}{53}$$

$$P_{add-1}(\textit{cat}|\textit{furry}) = \frac{\text{count}(\textit{furry cat}) + 1}{\text{count}(\textit{furry}) + 46} = \frac{0+1}{0+41} = \frac{1}{41}$$

However, it has been observed that adding a 1 to the numerator can cause a significant shift in the probabilistic distribution, especially when the underlying training corpus is small or medium in size. To address this issue, the add-1 smoothing formula is further modified, resulting in the add-k smoothing. The formula for add-k smoothing is

$$P_{add-k}(W_i|W_{i-1}) = \frac{\text{count}(W_{i-1}, W_i) + k}{\text{count}(W_{i-1}) + kV}, \text{ where } k \in (0, 1)$$

The drawbacks of this approach are that it assigns the same probability to all unseen words in the test case, and it is challenging to balance the distributional shift. The issue arises due to the involvement of a longer context of W_i, i.e., $\{W_{i-k}, \ldots, W_{i-2}, W_{i-1}\}$. So, if a shorter context is considered instead of using the entire one, it might result in a match. Next, we will discuss two plausible approaches, which opt for a shorter context if the original context causes a no-match.

4.2.3 Back-Off

This algorithm employs recursion to shorten the context size in order to address the unseen tokens. Initially, the algorithm begins with a higher-order n-gram to calculate $P(W_i|W_{i-1}, \ldots, W_{i-k+1})$. If it encounters any challenges, it falls back to lower-order n-grams. For example, assume that $P(fox|A\ quick\ brown)$ is zero. Then, the back-off algorithm will search for $P(fox|quick\ brown)$, and in case of a failure, it will again look for $P(fox|brown)$, i.e., at every step, the algorithm reduces the context size by one. The back-off algorithm can be expressed as,

$$P(W_i|W_{i-k+1}^{i-1}) = \begin{cases} \dfrac{\text{count}(W_{i-k+1}^{i})}{\text{count}(W_{i-k+1}^{i-1})}, & \text{if count}(W_{i-k+1}^{i}) > 0 \\ \alpha \times P(W_i|W_{i-k+2}^{i-1}), & \text{otherwise, where } \alpha \in (0, 1) \end{cases}$$

and

$$P(W_i) = \frac{\text{count}(W_i)}{N}, \text{ where } |V| \text{ is the length of the vocabulary.}$$

Here, α is a penalty for considering a shorter context.

Example 4.7

Find the probability of the sentence '*The furry cat sells sea shells*' w.r.t. the training data and a bigram language model. Assume that $\alpha = 0.4$.

Sentence 1: *The cat sat on the mat*
Sentence 2: *A furry brown fox jumps over the lazy dog*
Sentence 3: *She sells sea shells by the sea shore*
Sentence 4: *He reads books every evening before bed*
Sentence 5: *The sun rises in the east and sets in the west*

Solution

$P(The\ furry\ cat\ sells\ sea\ shells)$
$= P(The) \times P(furry|The) \times P(cat|furry) \times P(sells|cat) \times P(sea|sells)$
$\times P(shells|sells)$

Notice that $P(The)$ indicates the probability of the word '*The*' at the starting position. It means $P(The)$ is not a unigram probability but conditioned on being an initial word, which is equivalent to having the <*START*> token for all the sentences.

$$P(The) = \frac{2}{5}$$

$$P(furry|The) = \alpha \times P(furry) = 0.4 \times \frac{1}{41} = \frac{4}{410}$$

$$P(cat|furry) = \alpha \times P(cat) = 0.4 \times \frac{1}{41} = \frac{4}{410},$$

$$P(sells|cat) = \alpha \times P(sells) = 0.4 \times \frac{1}{41} = \frac{4}{410}$$

$$P(sea|sells) = \frac{1}{1} = 1,$$

$$P(shells|sea) = \frac{1}{2}$$

Now, replacing the bigram probabilities with corresponding values, we get

$$P(\textit{The furry cat sells sea shells}) = \frac{2}{5} \times \frac{4}{410} \times \frac{4}{410} \times \frac{4}{410} \times 1 \times \frac{1}{2} \approx 1.85 \times 10^{-7}.$$

Prompt Engineering

- A prompt is simply the input or instruction you give to a Large Language Model (LLM) — like ChatGPT, GPT-5, or BERT — to tell it what you want it to do.

- Prompt Engineering in Large Language Models (LLMs) means the art and science of designing effective input prompts to get the desired, accurate, and relevant responses from the model.

- LLMs don't "know" your intent unless you clearly guide them. A small change in wording can dramatically change the response quality.

Weak Prompt

"Explain LSTM for Bitcoin prediction."

Output (typical):

LSTM stands for Long Short-Term Memory. It is a type of recurrent neural network used for sequential data. It can be used to predict Bitcoin prices by learning patterns in the data.

Strong (Engineered) Prompt

"You are a data scientist explaining Bitcoin price forecasting to graduate students. Describe step-by-step how an LSTM network uses historical OHLC (Open, High, Low, Close) data to predict the next-day closing price. Include how data is preprocessed, how sequences are created, and how the model learns temporal dependencies."

Output (typical):

In Bitcoin price forecasting, an LSTM (Long Short-Term Memory) model is used to capture long-term temporal patterns in sequential data.

1. **Data preprocessing:** Historical OHLC data is normalized to ensure stable training. Missing values are imputed and outliers handled.
2. **Sequence creation:** The model takes past *n* time steps (e.g., last 60 days) as input to predict the next-day closing price.
3. **Model structure:** LSTM layers maintain memory cells that capture dependencies over time, followed by dense layers for final prediction.
4. **Training:** The model minimizes mean squared error between predicted and actual prices.
5. **Output:** The trained model forecasts future Bitcoin prices with improved temporal accuracy compared to basic RNNs.

Prompt Type	Example	Purpose
Zero-shot	**"Summarize this text."**	No prior examples
Few-shot	**Give example Q&A before asking**	Show desired format
Role-based	**"You are a finance expert..."**	Control tone/role
Chain-of-thought	**"Explain your reasoning..."**	Improve logic & reasoning
Instructional	**"List 5 key points..."**	Precise control over structure

The Basic Ingredients of a Prompt

An LLM is a prediction machine. Based on a certain input, the prompt, it tries to predict the words that might follow it. At its core (illustrated in Figure 6-6), the prompt does not need to be more than a few words to elicit a response from the LLM.

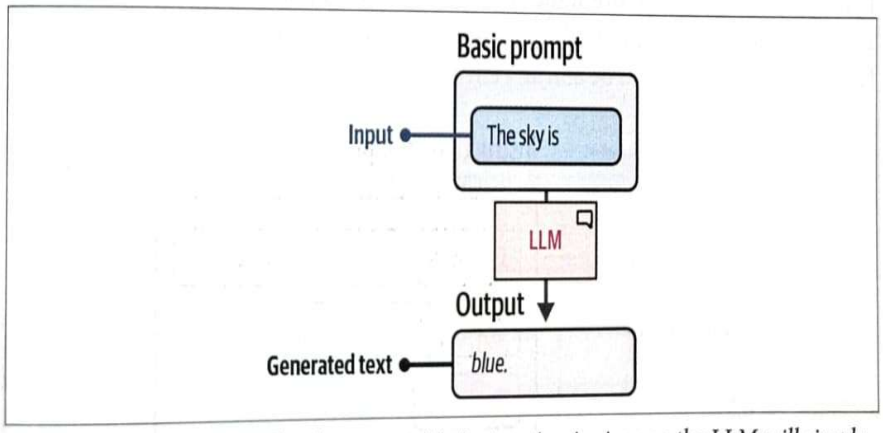


Figure 6-6. A basic example of a prompt. No instruction is given so the LLM will simply try to complete the sentence.

However, although the illustration works as a basic example, it fails to complete a specific task. Instead, we generally approach prompt engineering by asking a specific question or task the LLM should complete. To elicit the desired response, we need a more structured prompt.

For example, and as shown in Figure 6-7, we could ask the LLM to classify a sentence into either having positive or negative sentiment. This extends the most basic prompt

to one consisting of two components—the instruction itself and the data that relates to the instruction.

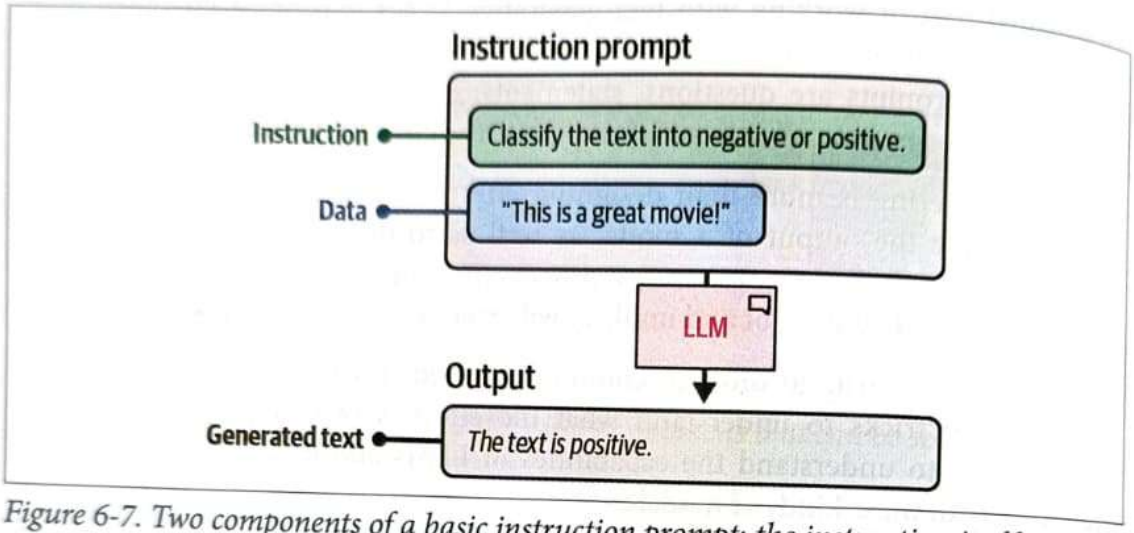


Figure 6-7. Two components of a basic instruction prompt: the instruction itself and the data it refers to.

More complex use cases might require more components in a prompt. For instance, to make sure the model only outputs "negative" or "positive" we can introduce output indicators that help guide the model. In Figure 6-8, we prefix the sentence with "Text:" and add "Sentiment:" to prevent the model from generating a complete sentence. Instead, this structure indicates that we expect either "negative" or "positive." Although the model might not have been trained on these components directly, it was fed enough instructions to be able to generalize to this structure.

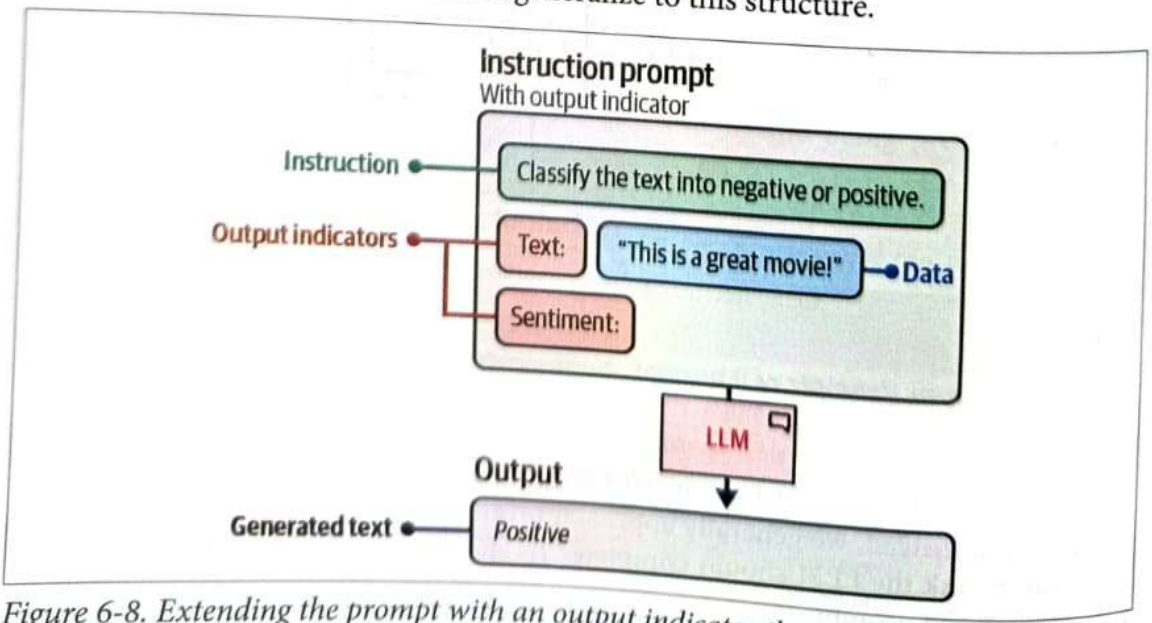


Figure 6-8. Extending the prompt with an output indicator that allows for a specific output.

Instruction based Prompting

Instruction-based prompting is a method where the prompt contains direct and specific instructions guiding the model's behaviour, style, content, or format of the output.

Instruction-Based Prompting

Although prompting comes in many flavors, from discussing philosophy with the LLM to role-playing with your favorite superhero, prompting is often used to have the LLM answer a specific question or resolve a certain task. This is referred to as *instruction-based prompting*.

Figure 6-9 illustrates a number of use cases in which instruction-based prompting plays an important role. We already did one of these in the previous example, namely supervised classification.

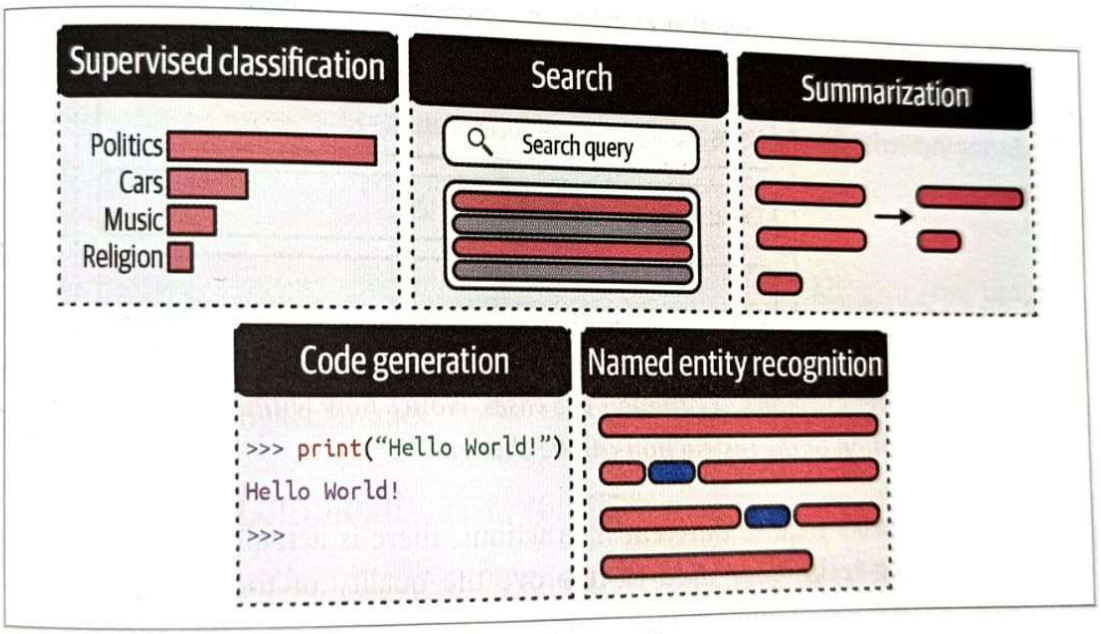


Figure 6-9. Use cases for instruction-based prompting.

Each of these tasks requires different prompting formats and more specifically, asking different questions of the LLM. Asking the LLM to summarize a piece of text will not suddenly result in classification. To illustrate, examples of prompts for some of these use cases can be found in Figure 6-10.

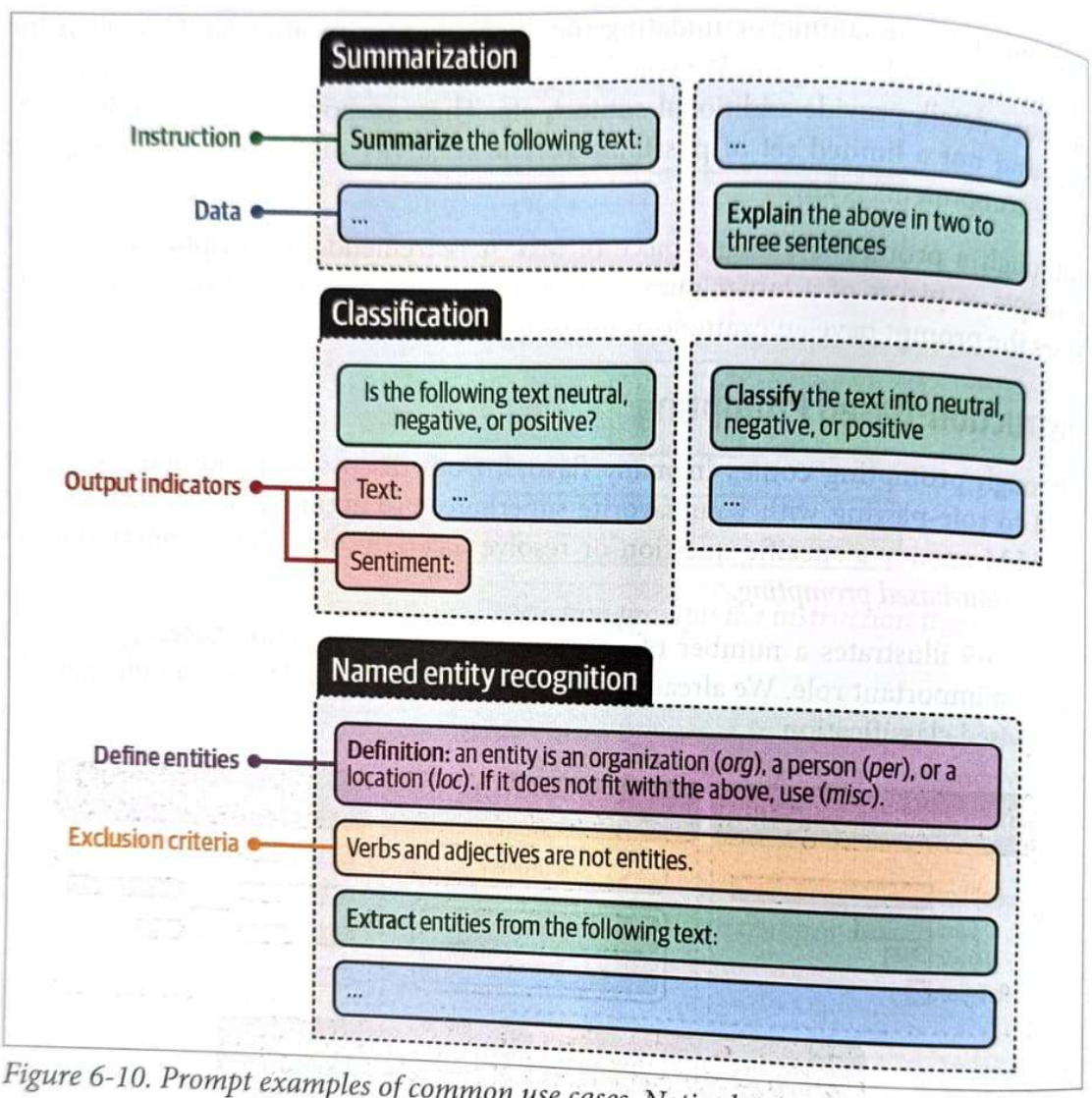


Figure 6-10. Prompt examples of common use cases. Notice how within a use case, the structure and location of the instruction can be changed.

Although these tasks require different instructions, there is actually a lot of overlap in the prompting techniques used to improve the quality of the output. A non-exhaustive list of these techniques includes:

Specificity

Accurately describe what you want to achieve. Instead of asking the LLM to "Write a description for a product" ask it to "Write a description for a product in less than two sentences and use a formal tone."

Hallucination

LLMs may generate incorrect information confidently, which is referred to as hallucination. To reduce its impact, we can ask the LLM to only generate an answer if it knows the answer. If it does not know the answer, it can respond with "I don't know."

Order

Either begin or end your prompt with the instruction. Especially with long prompts, information in the middle is often forgotten.[1] LLMs tend to focus on information either at the beginning of a prompt (primacy effect) or the end of a prompt (recency effect).

Advance Prompt Engineering

Advanced Prompt Engineering is the practice of using expert-level prompting methods to guide an LLM to think, reason, analyse, and generate better results for complex tasks.

What Advanced Prompt Engineering Achieves

✓ Higher accuracy in reasoning

✓ More predictable and consistent output

✓ Better performance on coding, math, NLP tasks

✓ Reduction of hallucinations

✓ Ability to handle multi-step complex workflows

✓ Better control over tone, format, and reasoning style

Where Advanced Prompt Engineering Is Used

- LLM-based apps
- Chatbots
- Research automation
- Code generation
- Data extraction
- Finance & forecasting models
- Academic writing
- Teaching and learning

Model I/O: Loading Quantized Models with LangChain

Using LangChain to load, run, and interact with quantized (compressed) language models that use fewer bits (like 8-bit, 4-bit) so they run faster and use less memory.

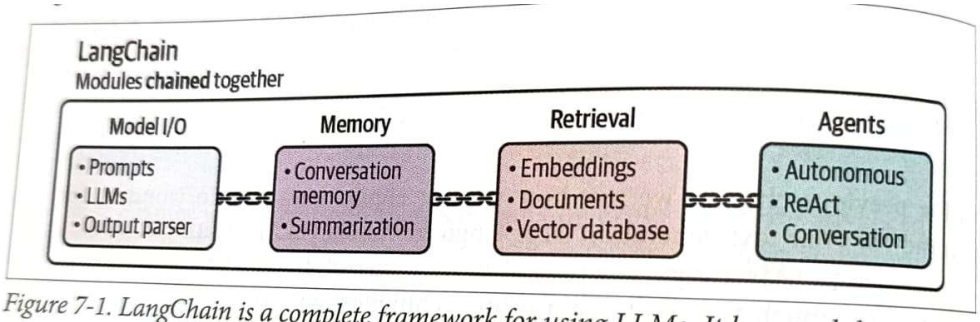


Figure 7-1. LangChain is a complete framework for using LLMs. It has modular components that can be chained together to allow for complex LLM systems.

The most basic form of a chain in LangChain is a single chain. Although a chain can take many forms, each with a different complexity, it generally connects an LLM with some additional tool, prompt, or feature. This idea of connecting a component to an LLM is illustrated in Figure 7-3.

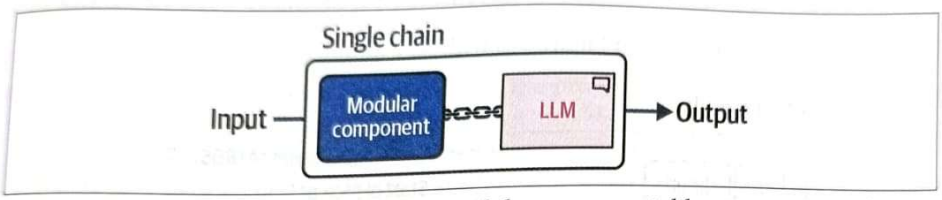


Figure 7-3. A single chain connects some modular component, like a prompt template or external memory, to the LLM.

In practice, chains can become complex quite quickly. We can extend the prompt template however we want and we can even combine several separate chains together to create intricate systems. In order to thoroughly understand what is happening in a chain, let's explore how we can add Phi-3's prompt template to the LLM.

A Single Link in the Chain: Prompt Template

We start with creating our first chain, namely the prompt template that Phi-3 expects. In the previous chapter, we explored how `transformers.pipeline` applies the chat template automatically. This is not always the case with other packages and they might need the prompt template to be explicitly defined. With LangChain, we will use chains to create and use a default prompt template. It also serves as a nice hands-on experience with using prompt templates.

The idea, as illustrated in Figure 7-4, is that we chain the prompt template together with the LLM to get the output we are looking for. Instead of having to copy-paste the prompt template each time we use the LLM, we would only need to define the user and system prompts.

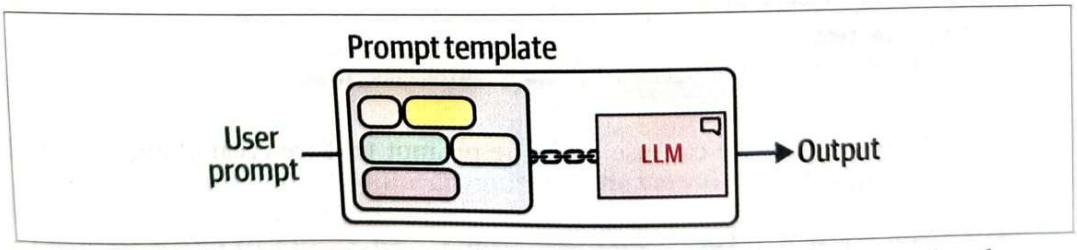


Figure 7-4. By chaining a prompt template with an LLM, we only need to define the input prompts. The template will be constructed for you.

A Chain with Multiple Prompts

In our previous example, we created a single chain consisting of a prompt template and an LLM. Since our example was quite straightforward, the LLM had no issues dealing with the prompt. However, some applications are more involved and require lengthy or complex prompts to generate a response that captures those intricate details.

Instead, we could break this complex prompt into smaller subtasks that can be run sequentially. This would require multiple calls to the LLM but with smaller prompts and intermediate outputs as shown in Figure 7-7.

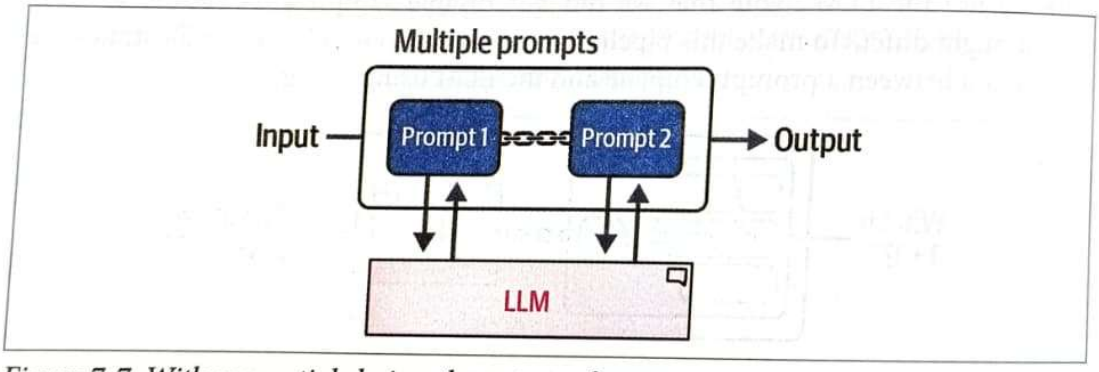


Figure 7-7. With sequential chains, the output of a prompt is used as the input for the next prompt.

This process of using multiple prompts is an extension of our previous example. Instead of using a single chain, we link chains where each link deals with a specific subtask.

For instance, consider the process of generating a story. We could ask the LLM to generate a story along with complex details like the title, a summary, a description of the characters, etc. Instead of trying to put all of that information into a single prompt, we could dissect this prompt into manageable smaller tasks instead.

Let's illustrate with an example. Assume that we want to generate a story that has three components:

- A title
- A description of the main character
- A summary of the story

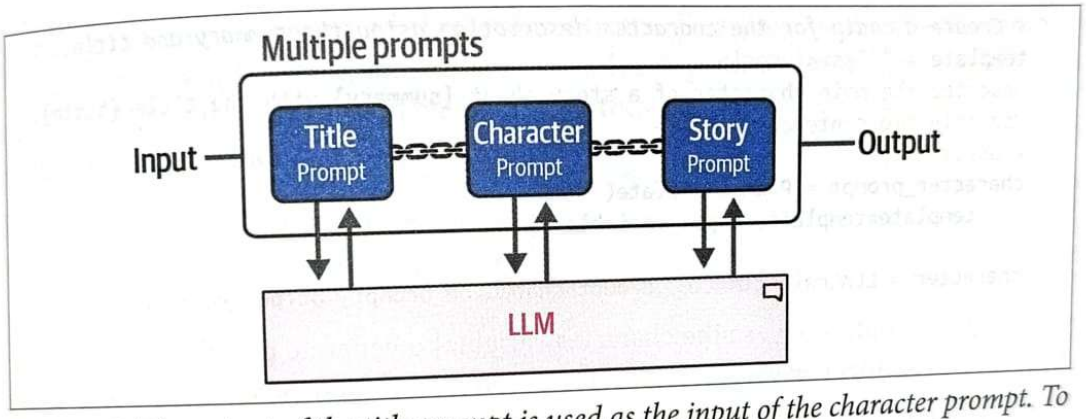


Figure 7-8. The output of the title prompt is used as the input of the character prompt. To generate the story, the output of all previous prompts is used.

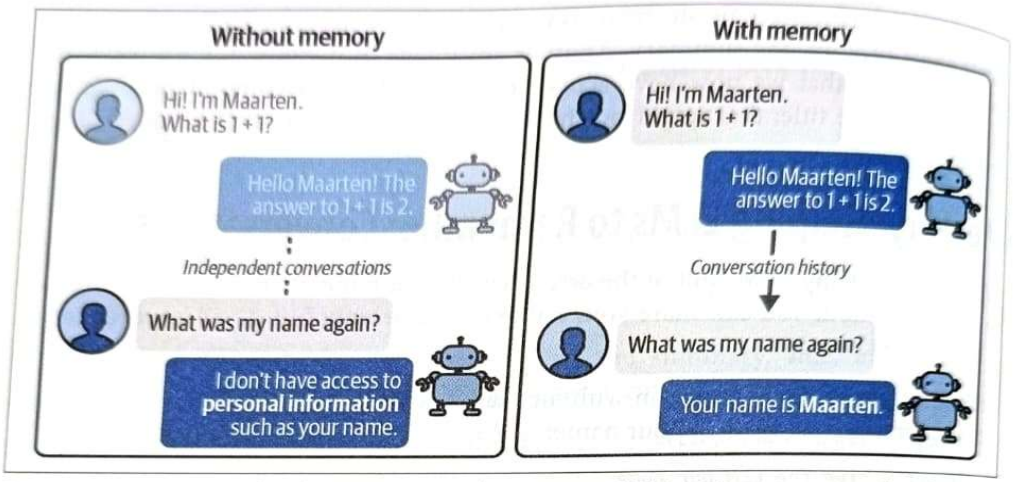


Figure 7-9. An example of a conversation between an LLM with memory and without memory.

Conversation Buffer

One of the most intuitive forms of giving LLMs memory is simply reminding them exactly what has happened in the past. As illustrated in Figure 7-10, we can achieve this by copying the full conversation history and pasting that into our prompt.

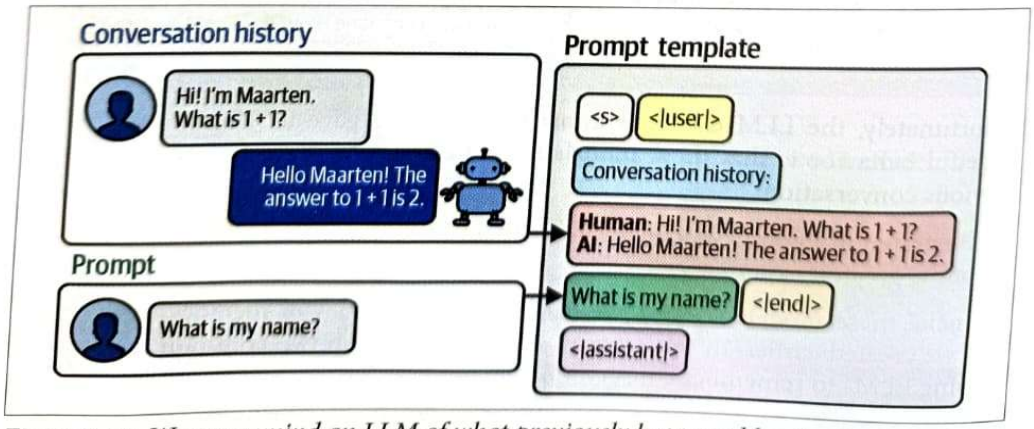


Figure 7-10. We can remind an LLM of what previously happened by simply appending the entire conversation history to the input prompt.

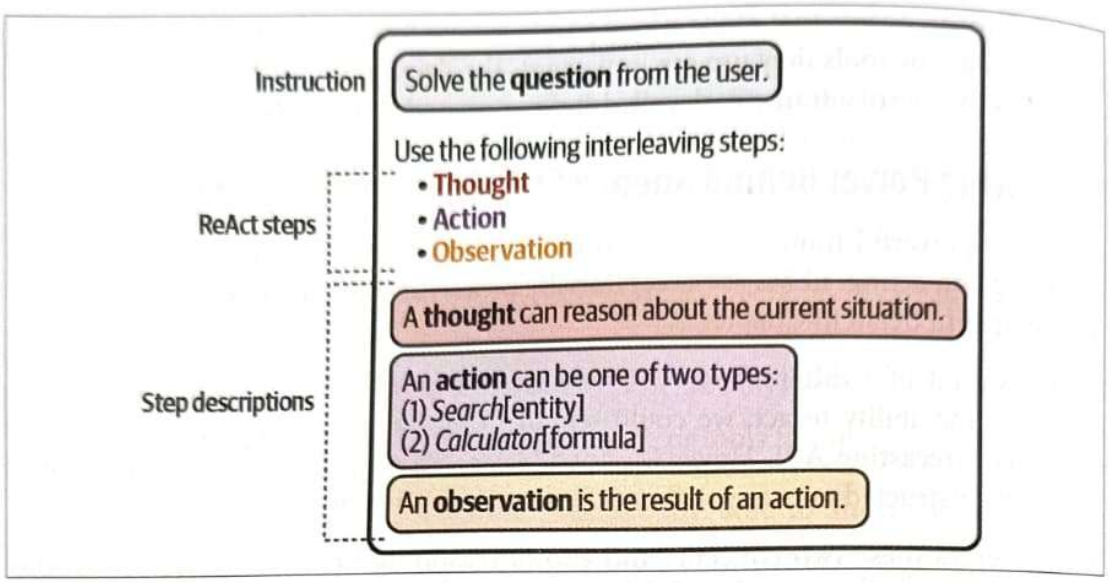


Figure 7-15. An example of a ReAct prompt template.

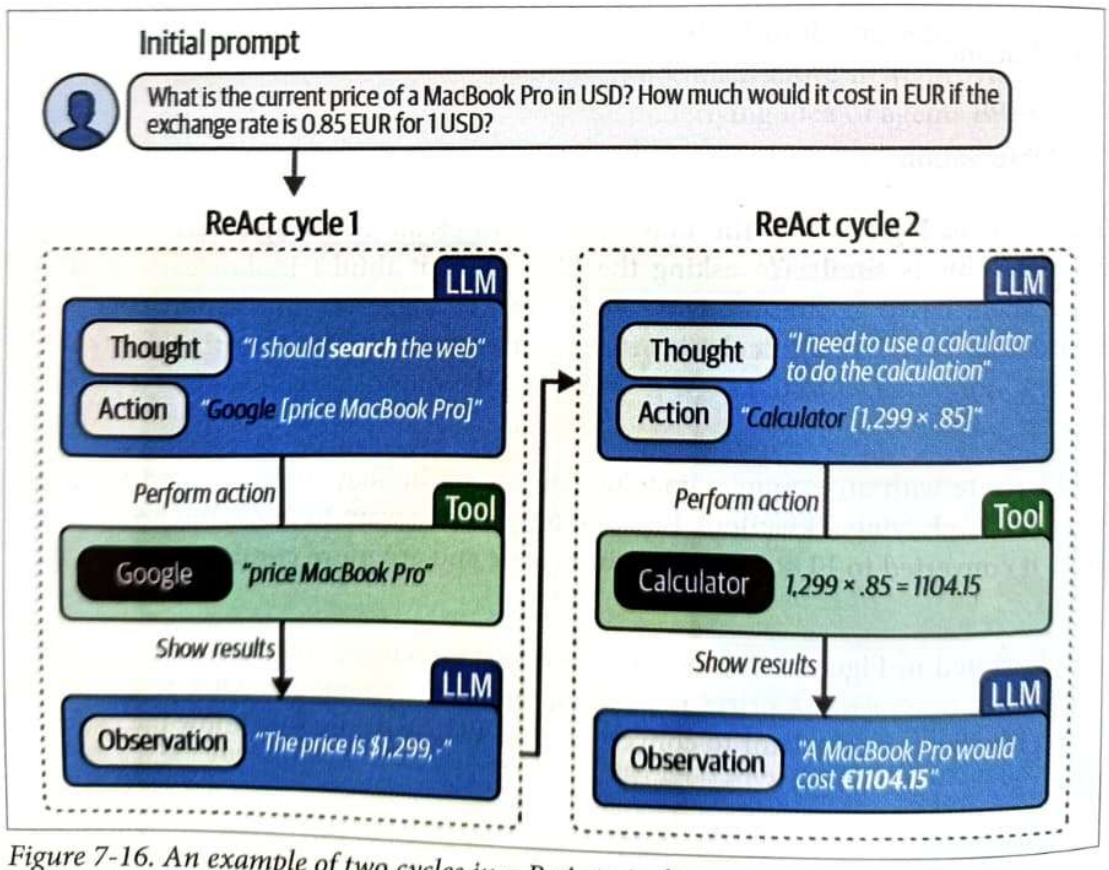


Figure 7-16. An example of two cycles in a ReAct pipeline.

Semantic Search

Semantic Search means searching by meaning, not by exact words.

Instead of matching keywords, an LLM + embeddings system understands the context, intent, and relationships of words to find the most meaningful results.

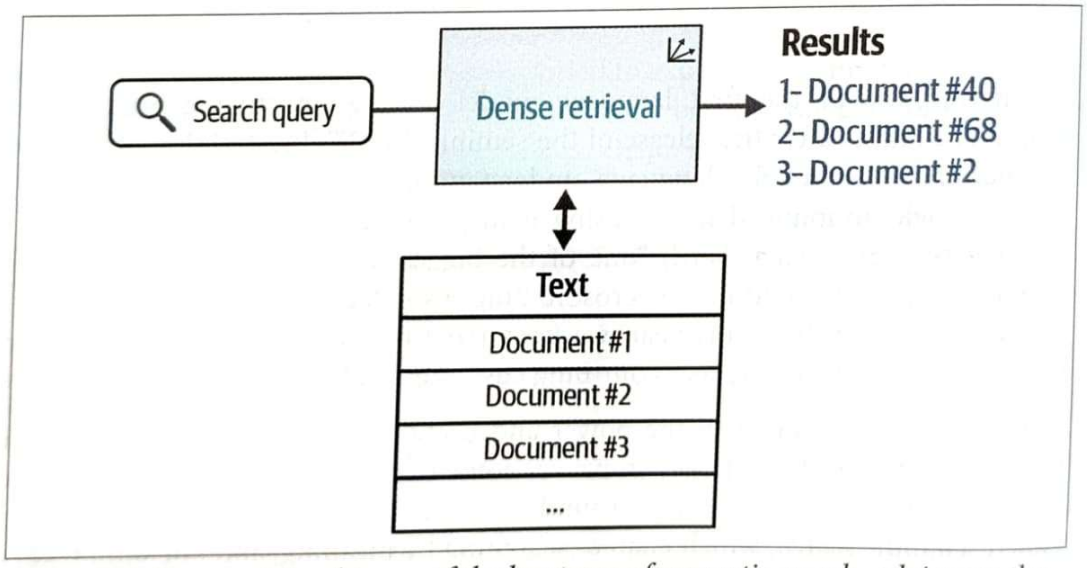


Figure 8-1. Dense retrieval is one of the key types of semantic search, relying on the similarity of text embeddings to retrieve relevant results.

Reranking

Search systems are often pipelines of multiple steps. A reranking language model is one of these steps and is tasked with scoring the relevance of a subset of results against the query; the order of results is then changed based on these scores. Figure 8-2 shows how rerankers are different from dense retrieval in that they take an additional input: a set of search results from a previous step in the search pipeline.

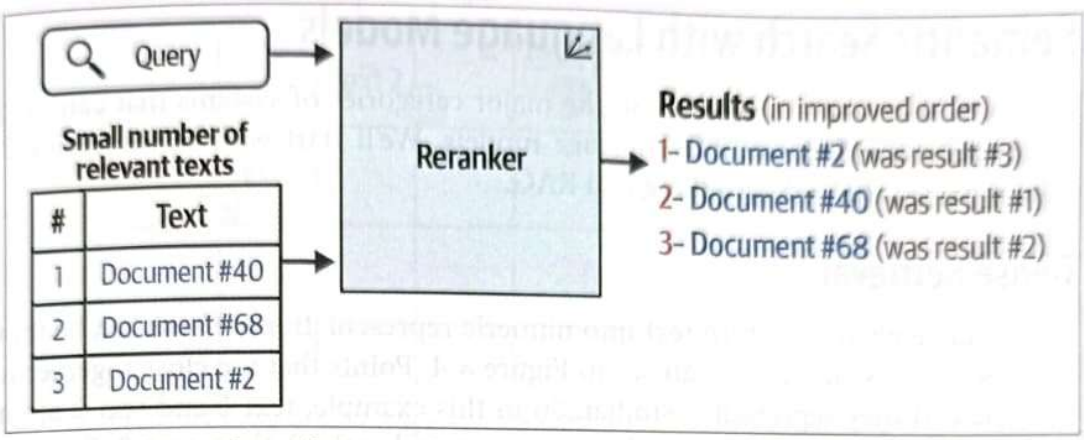


Figure 8-2. Rerankers, the second key type of semantic search, take a search query and a collection of results, and reorder them by relevance, often resulting in vastly improved results.

RAG

The growing LLM capability of text generation led to a new type of search systems that include a model that generates an answer in response to a query. Figure 8-3 shows an example of such a generative search system.

Generative search is a subset of a broader type of category of systems better called RAG systems. These are text generation systems that incorporate search capabilities to reduce hallucinations, increase factuality, and/or ground the generation model on a specific dataset.

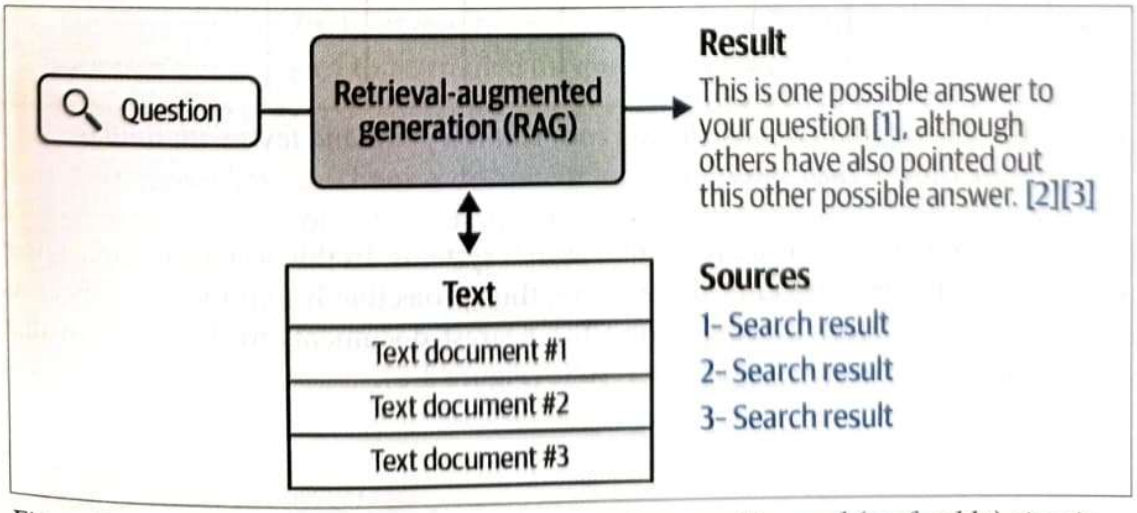


Figure 8-3. A RAG system formulates an answer to a question and (preferably) cites its information sources.

Dense Retrieval

Recall that embeddings turn text into numeric representations. Those can be thought of as points in space, as we can see in Figure 8-4. Points that are close together mean that the text they represent is similar. So in this example, text 1 and text 2 are more similar to each other (because they are near each other) than text 3 (because it's farther away).

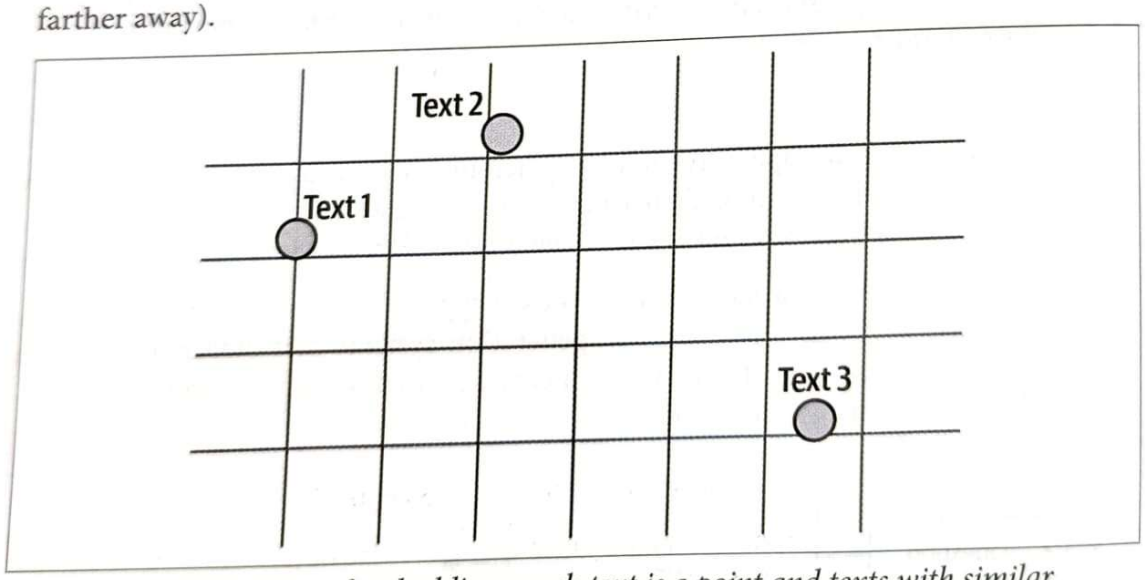


Figure 8-4. The intuition of embeddings: each text is a point and texts with similar meaning are close to each other.

Advanced RAG Techniques

There are several additional techniques to improve the performance of RAG systems. Some of them are laid out here.

Query rewriting

If the RAG system is a chatbot, the preceding simple RAG implementation would likely struggle with the search step if a question is too verbose, or to refer to context in previous messages in the conversation. This is why it's a good idea to use an LLM to rewrite the query into one that aids the retrieval step in getting the right information. An example of this is a message such as:

> User Question: "We have an essay due tomorrow. We have to write about some animal. I love penguins. I could write about them. But I could also write about dolphins. Are they animals? Maybe. Let's do dolphins. Where do they live for example?"

This should actually be rewritten into a query like:

> Query: "Where do dolphins live"

This rewriting behavior can be done through a prompt (or through an API call). Cohere's API, for example, has a dedicated query-rewriting mode for `co.chat`.

Multimodal Large Language Models

Multimodal Large Language Models (MLLMs) are advanced AI systems that can understand and generate information across **multiple types of data**, not just text.

In simple words:
LLMs = models that understand text
MLLMs = models that understand text + images + audio + video + other modalities

What "Multimodal" Means……

Modality = type of data.
MLLMs can process more than one modality, such as:

Text (questions, paragraphs, code)

Images (photos, diagrams, charts)

Audio (speech, music)

Video (clips, scenes)

Sensor data (e.g., from robots)

Applications of Multimodal LLMs…….

1. **Image Understanding**
   – Captioning, object detection, diagram solving
2. **Education**
   – Solve math problems from photos
   – Check handwritten answers
3. **Healthcare**
   – Analyse X-rays with explanations
4. **Finance**
   – Analyse charts + news + text
5. **Robotics**
   – Navigate environments using sensor + image I/O
6. **Creative tasks**
   – Generate images, logos, storyboards

Examples of Popular MLLMs……

GPT and GPT-5 family

Google Gemini

Meta LLaVA / LLaMA Vision

OpenAI CLIP + LLM hybrids

DeepMind Flamingo

PaLM-E (robotics multimodal model)

How MLLMs Work……..

Multimodal LLMs contain:

1. **Encoders**
   Convert each data type into embeddings (numerical representations).
   - Image encoder
   - Audio encoder
   - Video encoder
   - Text encoder (transformer)
2. **Fusion Module**
   Combines information across modalities.
3. **LLM Core**
   A transformer-based model that performs reasoning and generation.
4. **Decoders**
   Convert embeddings back into text, images, or audio.

Key Features of Multimodal LLMs……..

1. Understand multiple data types

Example:
Upload an image → ask questions about it → get answers in text.

2. Combine information across modalities

Example:
Provide a chart + a question → model interprets the chart & provides insights.

3. Generate multimodal outputs

- Generate images from text prompts
- Generate captions for images
- Convert speech to text or text to speech

4. Better reasoning

Because they see multiple forms of information, they can perform:

- Visual reasoning
- Mathematical reasoning with diagrams
- Chart reading
- Code understanding with screenshots
- Spatial or geometric reasoning

When you think about large language models (LLMs), multimodality might not be the first thing that comes to mind. After all, they are *language* models! But we can quickly see that models can be much more useful if they're able to handle types of data other than text. It's very useful, for example, if a language model is able to glance at a picture and answer questions about it. A model that is able to handle text and images (each of which is called a *modality*) is said to be *multimodal*, as we can see in Figure 9-1.

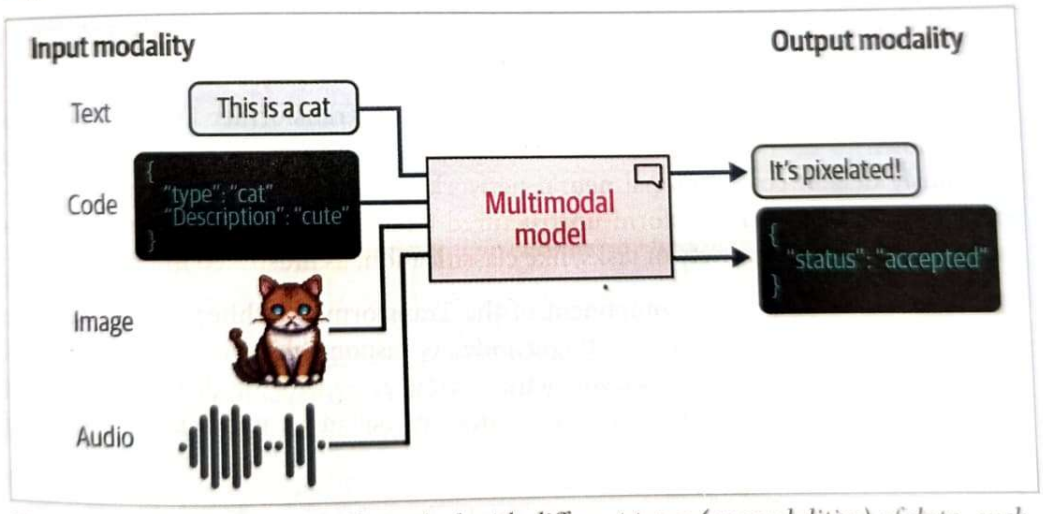


Figure 9-1. Models that are able to deal with different types (or modalities) of data, such as images, audio, video, or sensors, are said to be multimodal. It's possible for a model to accept a modality as input yet not be able to generate in that modality.

Transformers for vision……

Transformers for **vision** in LLMs are models that apply the **Transformer architecture** originally developed for text to process **images**. This capability is a key reason modern **Multimodal Large Language Models (MLLMs)** (like GPT, Gemini etc.) can understand pictures, charts, diagrams, and videos.

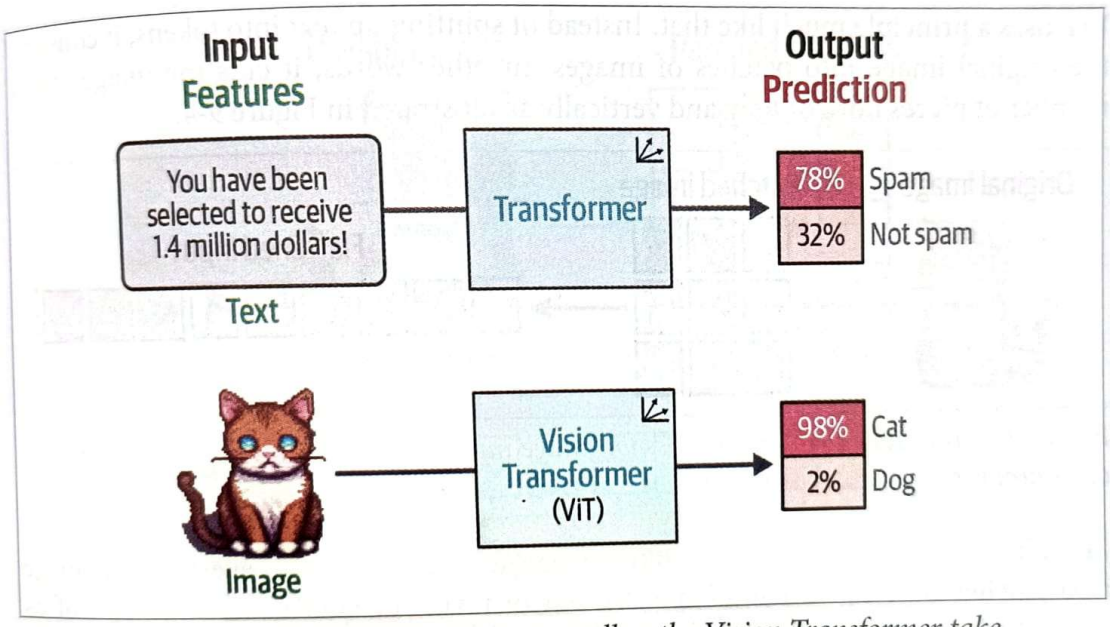


Figure 9-2. Both the original Transformer as well as the Vision Transformer take unstructured data, convert it to numerical representations, and finally use that for tasks like classification.

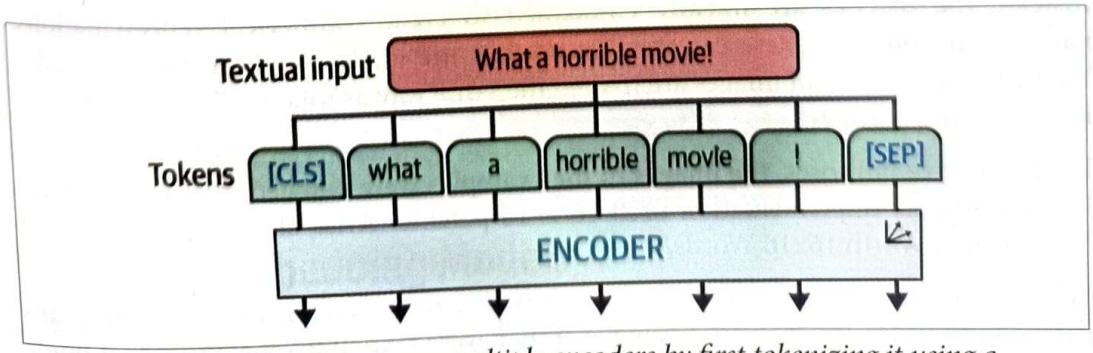


Figure 9-3. Text is passed to one or multiple encoders by first tokenizing it using a tokenizer.

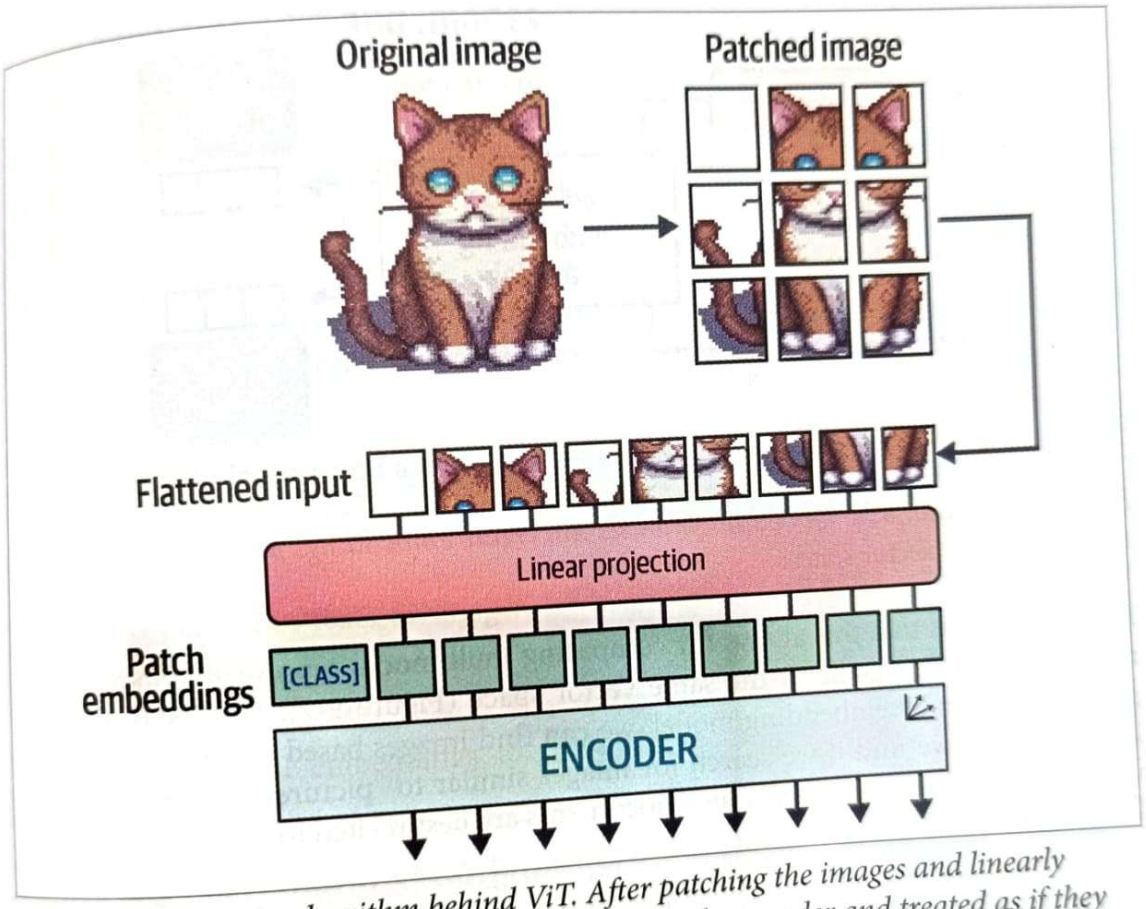


Figure 9-5. The main algorithm behind ViT. After patching the images and linearly projecting them, the patch embeddings are passed to the encoder and treated as if they were textual tokens.

Contrastive Language Image Pre-training……

CLIP: Connecting Text and Images

CLIP is an embedding model that can compute embeddings of both images and texts. The resulting embeddings lie in the same vector space, which means that the embeddings of images can be compared with the embeddings of text.[3] This comparison capability makes CLIP, and similar models, usable for tasks such as:

Zero-shot classification
: We can compare the embedding of an image with that of the description of its possible classes to find which class is most similar.

Clustering
: Cluster both images and a collection of keywords to find which keywords belong to which sets of images.

Search
: Across billions of texts or images, we can quickly find what relates to an input text or image.

Generation
: Use multimodal embeddings to drive the generation of images (e.g., stable diffusion[4]).

How Can CLIP Generate Multimodal Embeddings?

The procedure of CLIP is actually quite straightforward. Imagine that you have a dataset with millions of images alongside captions as we illustrate in Figure 9-8.

Figure 9-8. The type of data that is needed to train a multimodal embedding model.

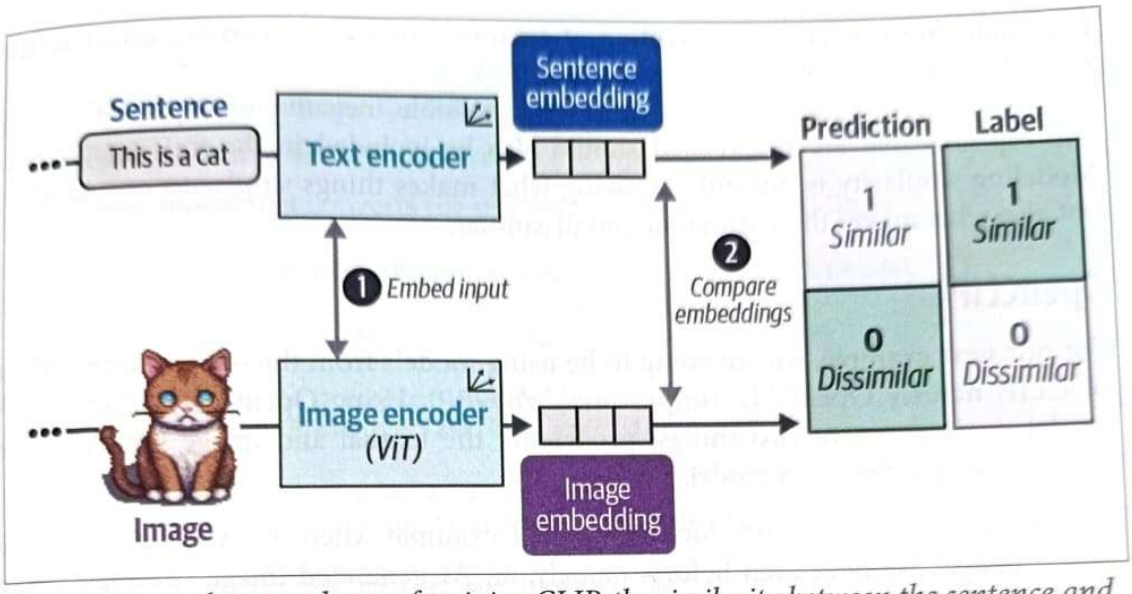


Figure 9-10. In the second step of training CLIP, the similarity between the sentence and image embedding is calculated using cosine similarity.

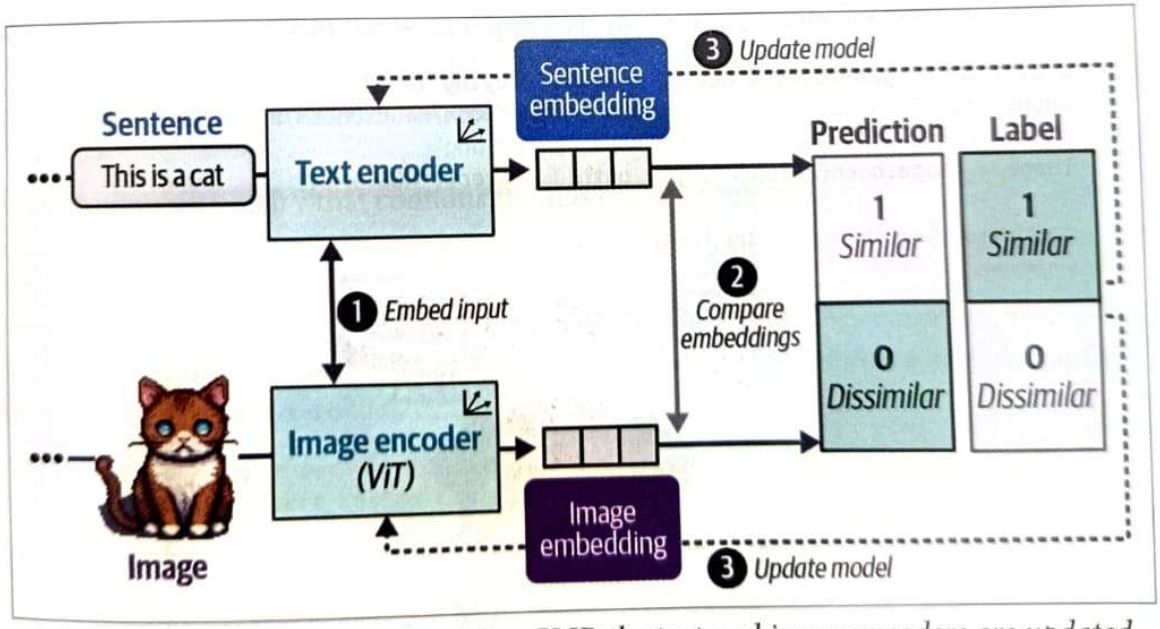


Figure 9-11. In the third step of training CLIP, the text and image encoders are updated to match what the intended similarity should be. This updates the embeddings such that they are closer in vector space if the inputs are similar.

Making Text Generation Models Multimodal……

Making Text Generation Models Multimodal taking a text-only LLM (like GPT, BERT, LLaMA, etc.) and adding the ability to understand and generate other types of data, such as:

- **Images**
- **Audio**
- **Video**
- **Charts**
- **Sensor data**
- **Embedding streams**

This process transforms an LLM into a **Multimodal Large Language Model (MLLM)**.

Why Do We Need Multimodality?

Because **real-world information is not only text**.

Examples:

- A student uploads a math diagram → LLM should solve it.
- A doctor uploads an X-ray → LLM should diagnose it.
- A trader uploads a candlestick chart → LLM should interpret it.
- A user uploads an audio clip → LLM should transcribe and explain it.

A text-only LLM cannot do these tasks.
A multimodal one can.

In the case of text generation models, we would like it to reason about certain input images. For example, we could give it an image of a pizza and ask it what ingredients it contains. You could show it a picture of the Eiffel Tower and ask when it was built or where it is located. This conversational ability is further illustrated in Figure 9-15.

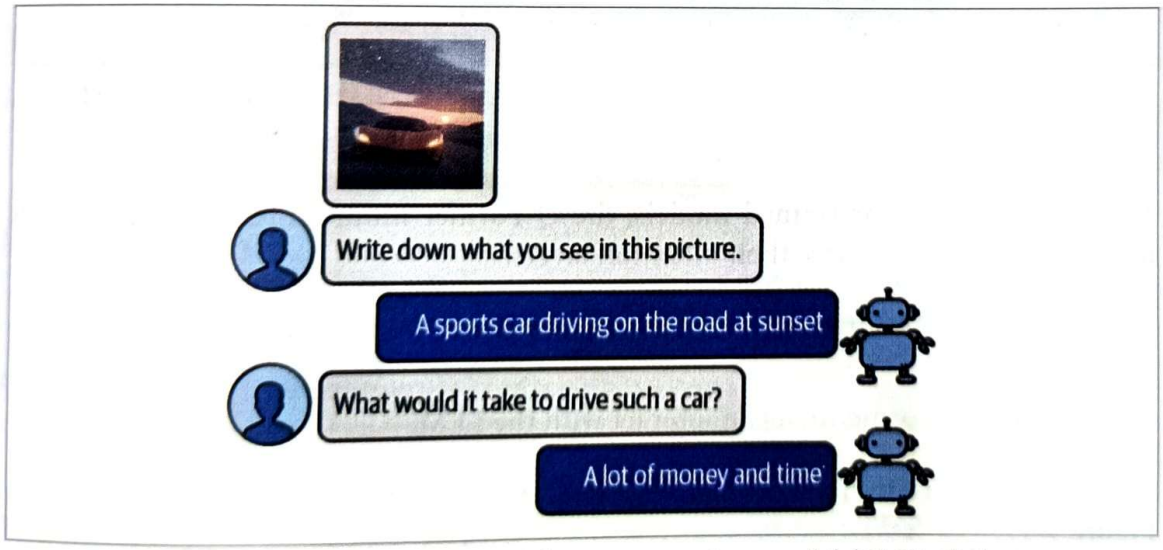


Figure 9-15. An example of a multimodal text generation model (BLIP-2) that can reason about input images.

To bridge the gap between these two domains, attempts have been made to introduce a form of multimodality to existing models. One such method is called *BLIP-2: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation 2*. BLIP-2 is an easy-to-use and modular technique that allows for introducing vision capabilities to existing language models.

Bootstrapping Language-Image Pre-training

It is a **vision-language model (VLM)** designed to connect:

- **Images**
- **Text**

BLIP is mainly used as the **image–text encoder** in many **multimodal LLM systems**.

How BLIP Works .....

BLIP consists of three components:

1. Vision Encoder (ViT)

Converts images into embeddings (like “image tokens”).

2. Text Encoder / Decoder

Reads or generates text.

3. Multimodal Fusion Module

Aligns image and text representations for joint understanding.

BLIP is trained on massive image–text datasets, allowing it to learn:

- visual content
- captions
- relationships between text and images

Benefits of BLIP

1. Strong Image–Text Alignment

BLIP aligns the meaning of text and images very accurately, improving:

Captioning
Question answering
Image-based reasoning

This alignment is crucial for multimodal LLMs.

2. High-Quality Image Captioning

3. BLIP can be used in different modes:

Image-to-text
Text-to-image
Vision–text matching

4. Reduces Noisy Data During Training

BLIP has a **caption-filtering mechanism**, meaning:

It removes low-quality captions

Keeps clean and relevant pairs

This leads to **better training** and more robust multimodal understanding.

BLIP is a vision-language model that helps LLMs understand images, align them with text, and perform tasks like captioning and visual Q&A, making multimodal AI more accurate and more powerful.

Potential Complexity *of a Prompt…*

Potential Complexity of a Prompt refers to how *detailed, layered, and cognitively demanding* a prompt is for an AI model (or a human) to understand and respond to correctly. It measures how much reasoning, structure, and constraint-handling is required to generate a good answer.

Prompt complexity depends on:

Number of instructions

Type of task (explain, analyze, predict, code, summarize, compare, etc.)

Context length

Constraints and rules

Reasoning depth required

Domain knowledge involved

Levels of Prompt Complexity

1. Low-Complexity Prompt

- One direct task
- No constraints
- No reasoning required

Example:

“Define LSTM.”

Output is straightforward.

2. Medium-Complexity Prompt

Multiple instructions

Some reasoning or comparison

Example:

“Explain LSTM and compare it with GRU in terms of performance and structure.”

Requires understanding + comparison.

3. High-Complexity Prompt

- Multi-step reasoning
- Multiple constraints
- Domain-specific
- Structured output required

Example:

“Design a hybrid LSTM-GRU model for Bitcoin price forecasting using OHLC data, justify hyperparameter choices, and explain how volatility affects prediction performance.”

Requires:

- Deep learning
- Financial domain knowledge
- Architecture design
- Analytical justification

Factors That Increase Prompt Complexity

Factor	Effect
Multi-tasking	Increases complexity
Strict formatting	Increases complexity
Logical reasoning	Increases complexity
Mathematical steps	Increases complexity
Domain-specific knowledge	Increases complexity
Long context	Increases complexity

Why Prompt Complexity Is Important in LLMs?

Determines **response quality**

Affects **model accuracy**

Influences **hallucination risk**

Impacts **token usage & cost**

Helps in **evaluating model intelligence**

Some advanced components make a prompt complex, these are….

Persona
Describe what role the LLM should take on. For example, use "You are an expert in astrophysics" if you want to ask a question about astrophysics.

Instruction
The task itself. Make sure this is as specific as possible. We do not want to leave much room for interpretation.

Context
Additional information describing the context of the problem or task. It answers questions like "What is the reason for the instruction?"

Format
The format the LLM should use to output the generated text. Without it, the LLM will come up with a format itself, which is troublesome in automated systems.

Audience
The target of the generated text. This also describes the level of the generated output. For education purposes, it is often helpful to use ELI5 ("Explain it like I'm 5").

Tone
The tone of voice the LLM should use in the generated text. If you are writing a formal email to your boss, you might not want to use an informal tone of voice.

Data
The main data related to the task itself.

To illustrate, let us extend the classification prompt we had earlier and use all of the preceding components. This is demonstrated in Figure 6-11.

This complex prompt demonstrates the modular nature of prompting. We can add and remove components freely and judge their effect on the output. As illustrated in Figure 6-12, we can slowly build up our prompt and explore the effect of each change.

The changes are not limited to simply introducing or removing components. Their order, as we saw before with the recency and primacy effects, can affect the quality of the LLM's output. In other words, experimentation is vital when finding the best prompt for your use case. With prompting, we essentially have ourselves in an iterative cycle of experimentation.

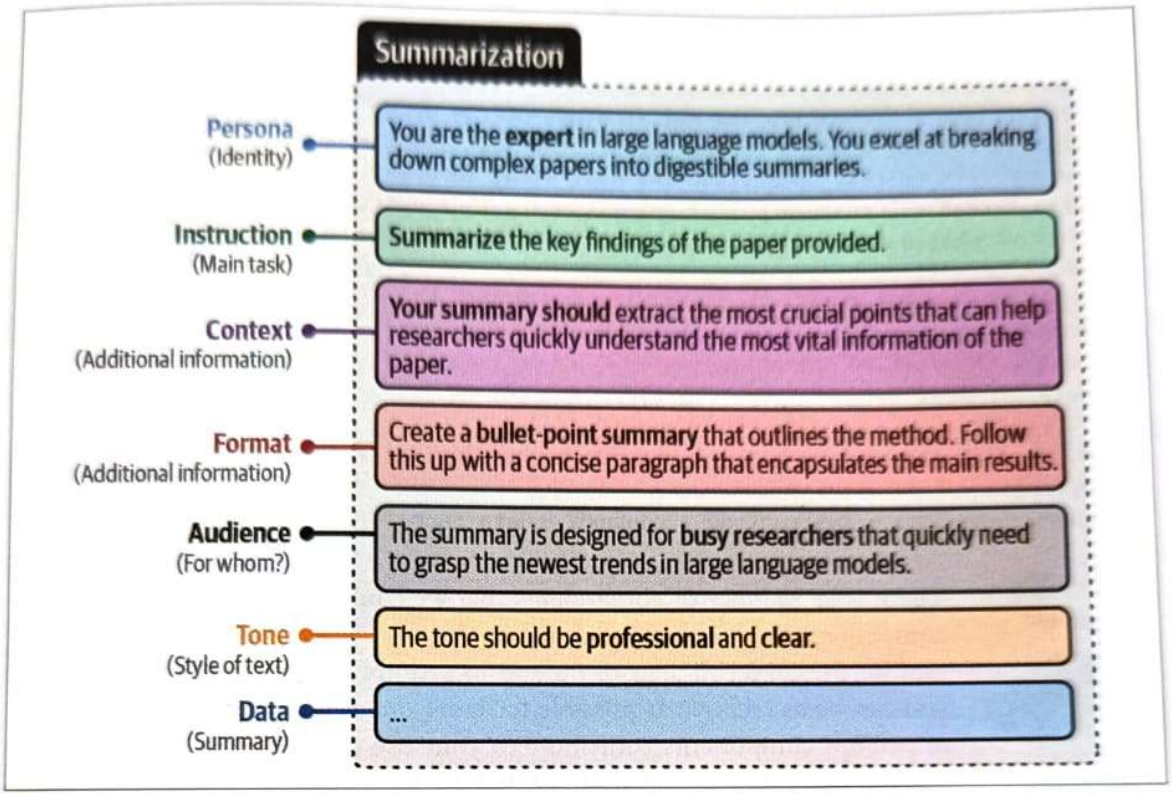


Figure 6-11. An example of a complex prompt with many components.

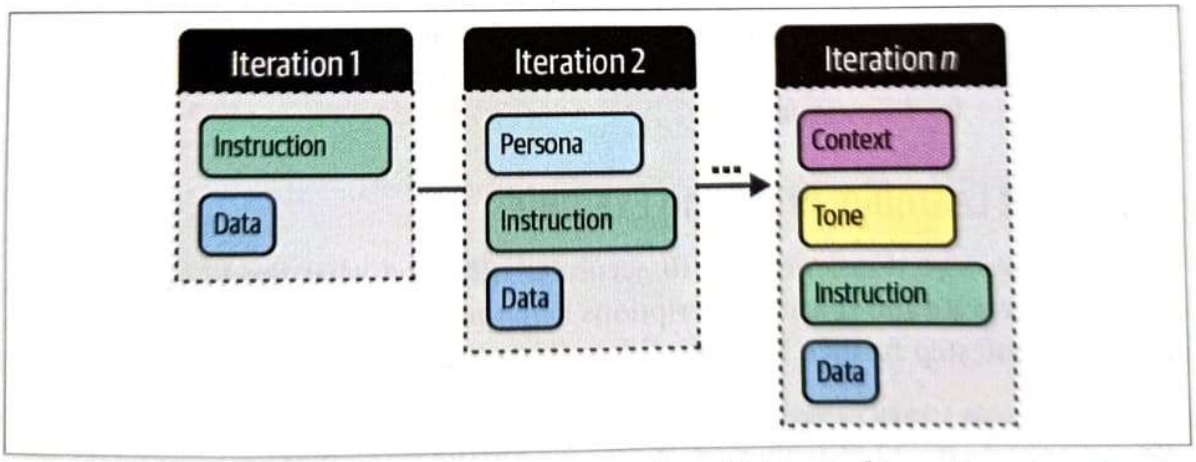


Figure 6-12. Iterating over modular components is a vital part of prompt engineering.

Try it out yourself! Use the complex prompt to add and/or remove parts to observe its impact on the generated output. You will quickly notice when pieces of the puzzle are worth keeping. You can use your own data by adding it to the `data` variable:

Reasoning with Generative Models:

Reasoning with Generative Models refers to the ability of generative AI systems—especially Large Language Models (LLMs) like GPT, Pathways Language Model (PaLM) or LLaMA—to **perform logical thinking, multi-step problem solving, decision-making, and inference**, rather than just generating fluent text.

What Are Generative Models?

Generative models are AI systems that **learn the probability distribution of data and generate new data samples** similar to what they were trained on.

Examples:

- Text → GPT, BERT (decoder side)
- Images → **Generative Adversarial Network** (GANs), Diffusion Models
- Audio → WaveNet

In LLMs, these models generate text **token by token** based on learned patterns.

Types of Reasoning in Generative Models…

Type	Description	Example
Deductive	Rule → Conclusion	If X > Y, X is larger
Inductive	Pattern → Rule	Stock trends
Abductive	Best explanation	Medical diagnosis
Analogical	Similarity-based	Heart = Pump
Causal	Cause → Effect	Rate hike → Price drop
Mathematical	Numeric logic	Algebra, calculus

Chain of Thought:

Chain of Thought is a technique that enables LLMs to solve complex problems by breaking them into step-by-step logical reasoning before producing the final answer.

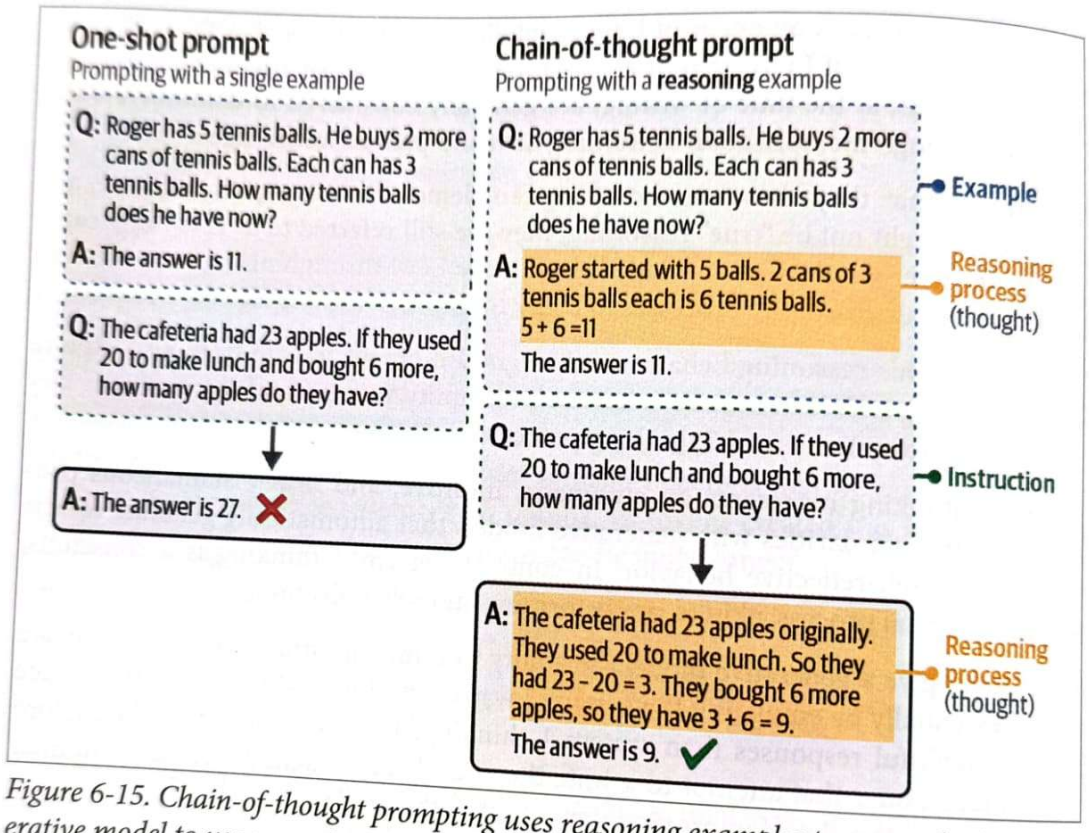


Figure 6-15. Chain-of-thought prompting uses reasoning examples to persuade the generative model to use reasoning in its answer.

Although chain-of-thought is a great method for enhancing the output of a generative model, it does require one or more examples of reasoning in the prompt, which the user might not have access to. Instead of providing examples, we can simply ask the generative model to provide the reasoning (zero-shot chain-of-thought). There are many different forms that work but a common and effective method is to use the phrase "Let's think step-by-step," which is illustrated in Figure 6-16.[7]

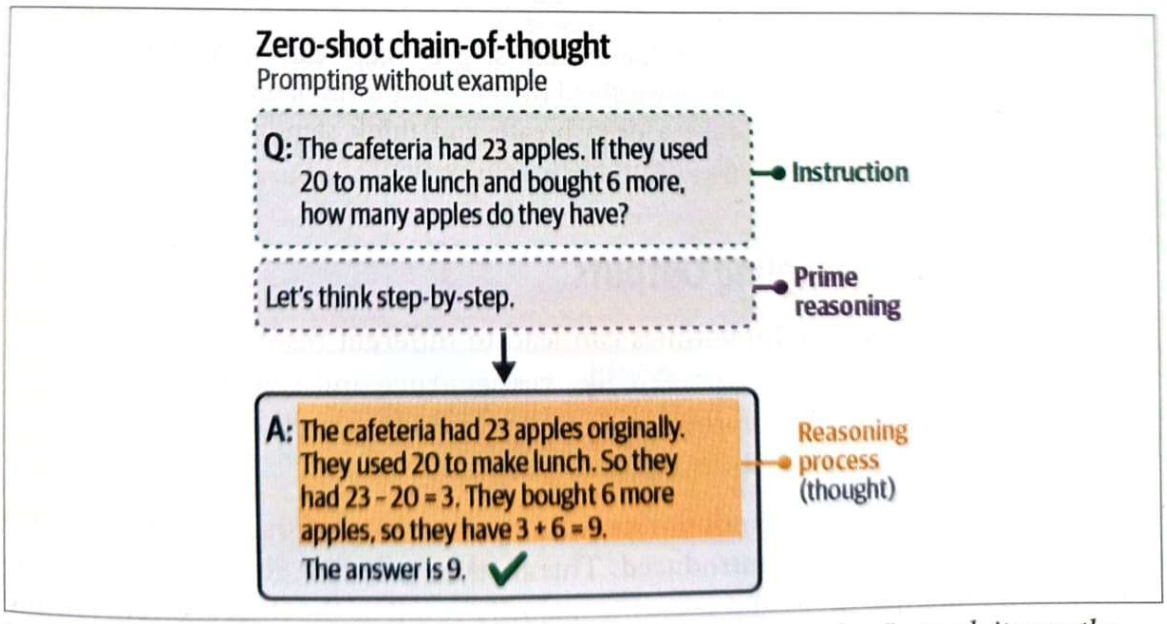


Figure 6-16. Chain-of-thought prompting without using examples. Instead, it uses the phrase "Let's think step-by-step" to prime reasoning in its answer.

Self-Consistency: Sampling Outputs:

Self-Consistency is a decoding and reasoning strategy in Large Language Models where multiple reasoning paths are sampled for the same prompt, and the most frequent final answer is selected as the correct one.

Instead of trusting one single reasoning chain, the model Generates many different solutions path, produces multiple final answers**,** Selects the most consistent (majority-voted) answer. This greatly improves reasoning accuracy.

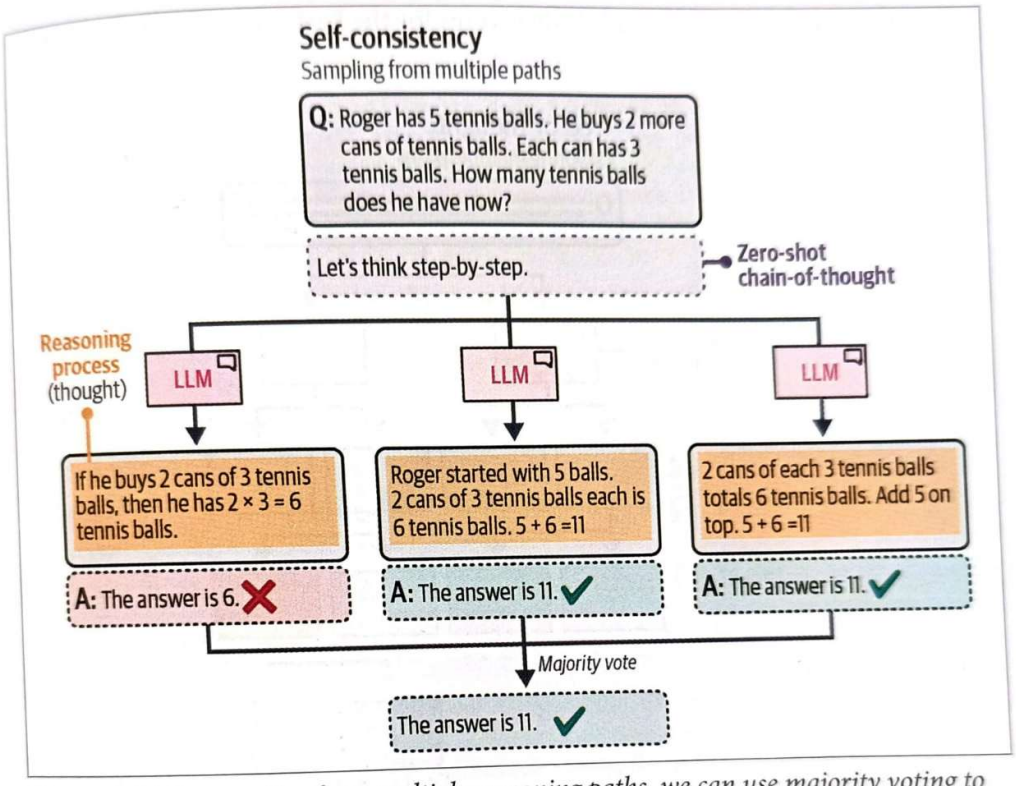


Figure 6-17. By sampling from multiple reasoning paths, we can use majority voting to extract the most likely answer.

Tree of Thoughts:

Tree of Thoughts (ToT) is an advanced reasoning framework for Large Language Models where the model explores **multiple reasoning paths in a tree-like structure**, evaluates them at each step, and selects the best path before reaching a final answer.

Unlike **Chain of Thought (CoT)** which follows a **single linear reasoning path**, ToT allows the model to:

- Branch into **multiple possible thoughts**
- Evaluate each branch
- Prune weak paths
- Continue only with the best ones

This closely resembles **human problem-solving by exploring alternatives**.

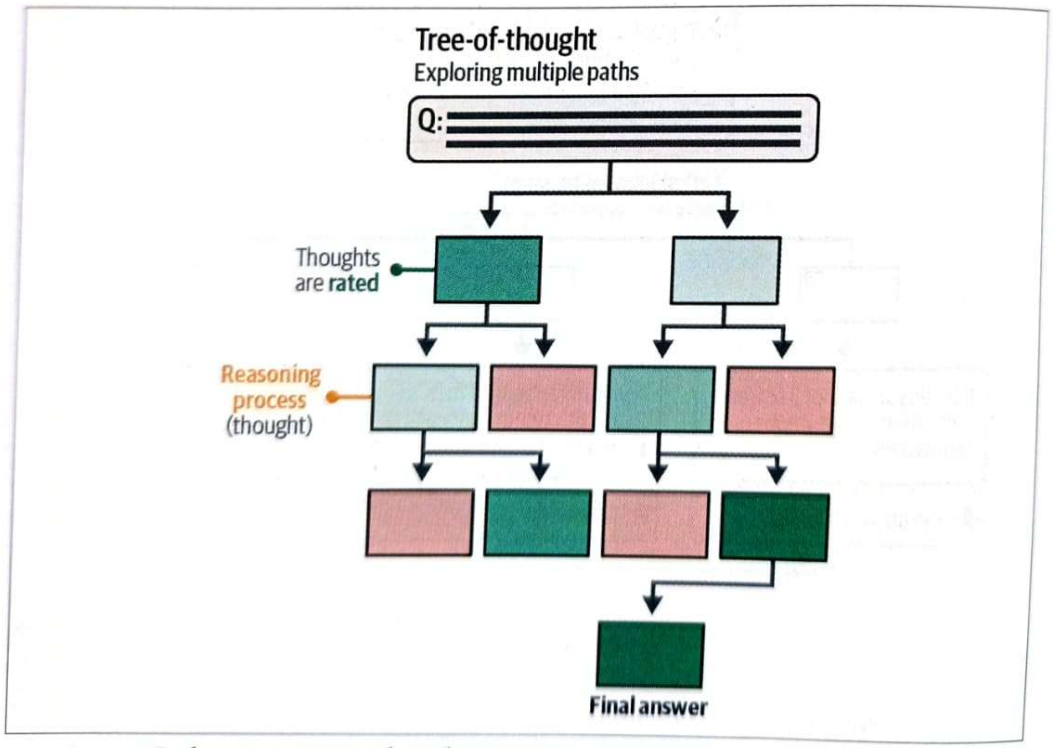


Figure 6-18. By leveraging a tree-based structure, generative models can generate intermediate thoughts to be rated. The most promising thoughts are kept and the lowest are pruned.

Creating Text Embedding Models

Creating text-embedding models in large language models" refers to the process of designing or using a model that converts text into **numeric vectors** (embedding's) that capture meaning. These vectors allow software systems to:

- compare semantic similarity
- perform search & retrieval
- cluster documents
- feed structured inputs into larger LLM pipelines

A text embedding model takes text (a word, sentence, or document) and produces a vector such as:

[0.12, -0.87, 0.45, ...]

These vectors represent the semantic meaning of the text.

Texts with similar meaning produce vectors that is close in vector space.

Examples:

"car" and "automobile" → close vectors

"dog" and "quantum physics" → far apart

How Embedding Models Relate to LLMs

Modern large language models (LLMs), like GPT, LLaMA, and others, internally use embedding's to represent text.

An embedding model is often: a separate small neural network trained specifically for embedding's or a part of a larger LLM, extracted or fine-tuned for embedding's. Many LLM providers offer embedding-specific versions of their main models:

OpenAI: text-embedding-3-large, text-embedding-3-small.

Meta: LLaMA model variants fine-tuned for embeddings

Google: models like Gecko (for embeddings)

Contrastive Learning

Contrastive learning in Large Language Models (LLMs) refers to training strategies where the model learns better representations of text by comparing examples—pulling similar text representations closer and pushing dissimilar ones apart.

While traditional LLM training uses next-token prediction (causal language modeling), contrastive learning is increasingly used to improve embedding's, alignment, reasoning, and robustness.

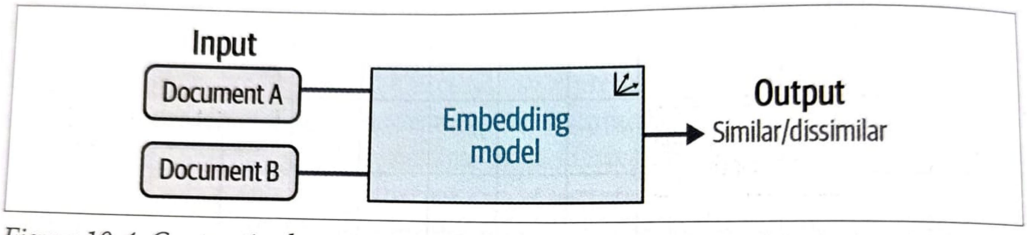


Figure 10-4. Contrastive learning aims to teach an embedding model whether documents are similar or dissimilar. It does so by presenting groups of documents to a model that are similar or dissimilar to a certain degree.

1. Improving Embeddings

LLMs often produce vector embedding's for tasks like search, retrieval, or clustering.
Contrastive learning helps embedding's capture semantic similarity.

Positive pairs might be:

- paraphrases
- question ↔ answer
- similar sentences from augmentation

Negative pairs might be:

- unrelated sentences
- contradictory sentences

This works for models like **E5**, **GTE**, **SimCSE**, and many retrieval-augmented LLMs.

2. Self-Supervised Contrastive Objectives

LLMs can generate their own training pairs. Examples:

SimCSE for LLMs

- Positive pair: two different dropout passes of the same sentence
- Negative pair: other sentences in the batch

This improves sentence embedding's without labels.

Contrastive Decoding

Models learn:

- anchor sentence = original
- positive = perturbed version with same meaning
- negative = perturbed version that changes meaning

3. Contrastive Learning for Instruction Following

LLMs can be aligned using **preference pairs**, e.g.:

- Good answer (positive)
- Bad answer (negative)

Contrastive loss pushes the model to prefer the positive.

This is used in:

- **Direct Preference Optimization (DPO)**
- **Contrastive Preference Optimization (CPO)**

These methods compare embedding's or log-probabilities of *good vs. bad* responses.

4. Contrastive Tuning for Robustness

Applied to make LLMs robust to:

- adversarial prompts
- hallucination
- paraphrased instructions

Example:
Contrast the model's output against a harmful or incorrect output.

5. Multimodal LLMs (e.g., CLIP-like training)

LLMs connected to image/audio/video encoders often use contrastive learning:

- Text encoder ↔ Image encoder
- Positive: caption matches an image
- Negative: mismatched pairs

@Why Contrastive Learning Helps LLMs…

Creates high-quality semantic embedding's

Improves retrieval-augmented generation (RAG) performance

Makes responses more consistent and aligned

Helps in preference optimization (LLM alignment)

Reduces hallucinations by encouraging consistent representations

SBERT

SBERT (Sentence-BERT) in Large Language Models

SBERT is a modification of BERT designed to produce sentence embedding's fixed-size vector representations that capture the semantic meaning of a sentence. While BERT (and many LLMs) can understand text well, they aren't optimized to produce efficient, meaningful sentence-level embedding's out of the box.

SBERT = BERT + Siamese network architecture + contrastive training

[Siamese network architecture]

A Siamese network architecture in the context of LLMs refers to using two identical transformer encoders (with shared weights) to convert two different text inputs into comparable embedding's, so that their similarity can be measured.

Instead, it is a training setup (dual-encoder) built on top of an LLM encoder.

A Siamese architecture for LLMs uses:

Text A → Encoder → Embedding A

Text B → Encoder → Embedding B

The same LLM encoder (same weights) processes A and B.

Then the embeddings are compared using:

cosine similarity

dot product

contrastive loss

triplet loss

So, a Siamese setup is basically a pair of identical LLM encoders used for similarity learning.

It was designed to:

Generate dense semantic vectors.

Enable semantic similarity, clustering, and retrieval.

Run much faster than BERT for pairwise comparisons.

How SBERT Works:

SBERT usually consists of:

Two identical transformer encoders (shared weights → "Siamese")

Sentences encoded into embeddings

A similarity function (cosine similarity)

Concept	SBERT	Modern LLMs
Purpose	Sentence embedding's	Text generation & understanding
Training	Contrastive, NLI	Next-token prediction
Output	Fixed semantic vectors	Probabilistic token distribution
Use case	Semantic similarity, retrieval, clustering	Chabot's, reasoning, generation
Relation	Used *with* LLMs in RAG	Needs SBERT-like training for embedding's

A solution to this overhead is to generate embeddings from a BERT model by averaging its output layer or using the `[CLS]` token. This, however, has shown to be worse than simply averaging word vectors, like GloVe.[4]

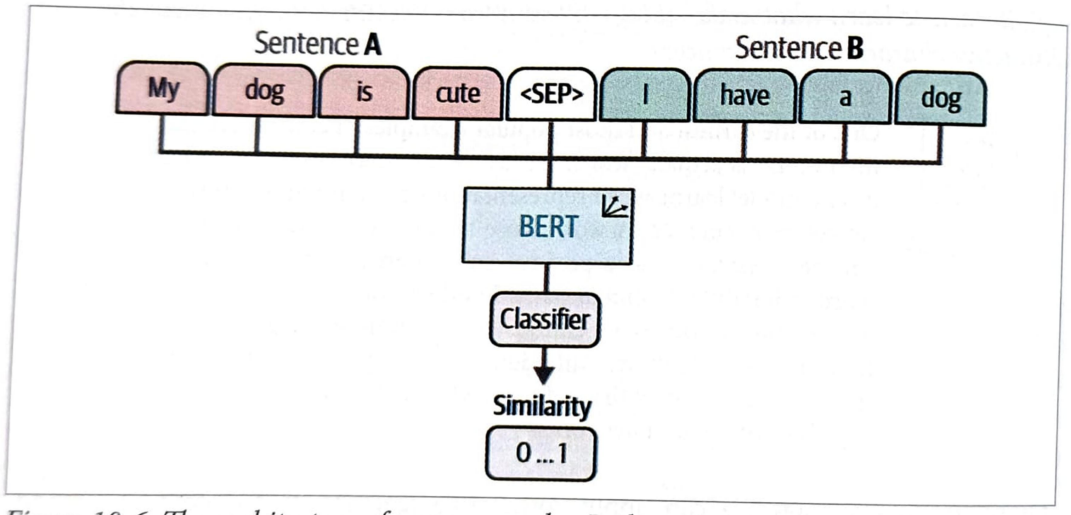


Figure 10-6. The architecture of a cross-encoder. Both sentences are concatenated, separated with a <SEP> token, and fed to the model simultaneously.

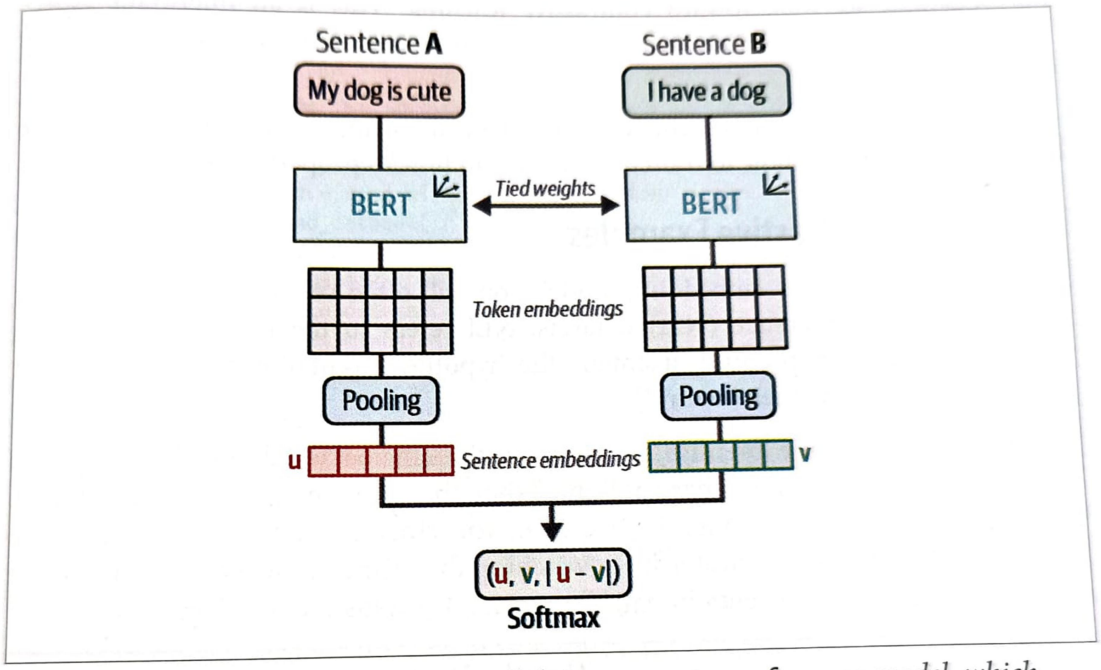


Figure 10-7. The architecture of the original `sentence-transformers` model, which leverages a Siamese network, also called a bi-encoder.

- The original sentence-transformers model uses a Siamese (bi-encoder) architecture.
- Training pairs of sentences involves loss functions that significantly affect model performance.
- During training, embedding's from each sentence are concatenated along with their difference.
- This combined embedding is then optimized using a **softmax classifier**.
- Bi-encoders are fast and create accurate sentence embedding's.
- Cross-encoders, although slower, usually provide better performance but do not generate embedding's.
- Like cross-encoders, bi-encoders use contrastive learning to optimize similarity or dissimilarity between sentence pairs.
- Through this optimization, the model learns what features make sentences similar or different.

SBERT

SBERT (Sentence-BERT) in the context of LLMs refers to using Sentence-BERT as a semantic embedding model alongside or within Large Language Model (LLM) pipelines to improve understanding, retrieval, and reasoning over text.

SBERT vs LLM Embeddings

Feature	SBERT	LLM Embeddings
Speed	Very fast	Slower
Cost	Free (open-source)	Paid (API)
Size	Small	Large
Best for	Retrieval, similarity	Deep semantic understanding
Fine-tuning	Easy	Limited

Generating Contrastive Examples:

Generating contrastive examples in Large Language Models (LLMs) means creating pairs or sets of examples that are very similar in surface form but differ in meaning, label, or outcome, so that the model learns or is evaluated on fine-grained distinctions rather than shallow patterns.

A **contrastive example** consists of:

- **Positive (anchor / correct) example**
- **Negative (hard negative / contrastive) example**

They differ **minimally**, but their **labels or meanings change**.

Simple Example

"Bitcoin price increased due to high demand." → **Positive**

"Bitcoin price increased due to low demand." → **Negative**

Only one word changes, but the meaning flips.

Why Contrastive Examples Are Important in LLMs

Contrastive examples help LLMs:
Learn **robust semantic representations**
Reduce **spurious correlations**
Improve **reasoning and generalization**
Enhance **factual consistency**
Handle **edge cases and ambiguity**

Contrastive is especially useful in:

- Fine-tuning
- Prompt engineering
- Evaluation
- Alignment and safety

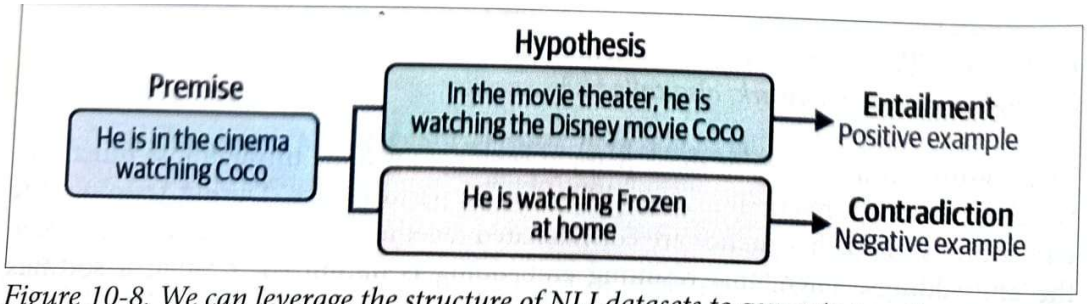


Figure 10-8. We can leverage the structure of NLI datasets to generate negative examples (contradiction) and positive examples (entailments) for contrastive learning.

Cosine Similarity:

Cosine similarity is a core mathematical measure used in Large Language Models (LLMs) to compare text embedding's and determine how semantically similar two pieces of text are.

Why Cosine Similarity Is Used in LLMs

LLMs convert text into **high-dimensional embedding's**.
Cosine similarity is preferred because:

Length-independent
Works well in high-dimensional spaces
Stable for semantic comparison
Computationally efficient

Cosine Similarity Vs Other Distance Metrics:

Metric	Measures	Used When
Cosine Similarity	Angle	Semantic meaning
Euclidean Distance	Absolute distance	Magnitude matters
Dot Product	Projection	Fast but unnormalized
Manhattan Distance	L1 distance	Sparse vectors

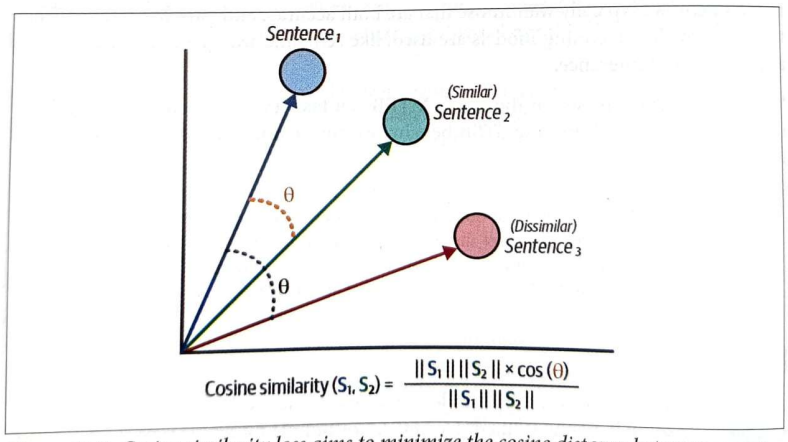


Figure 10-9. Cosine similarity loss aims to minimize the cosine distance between semantically similar sentences and to maximize the distance between semantically dissimilar sentences.

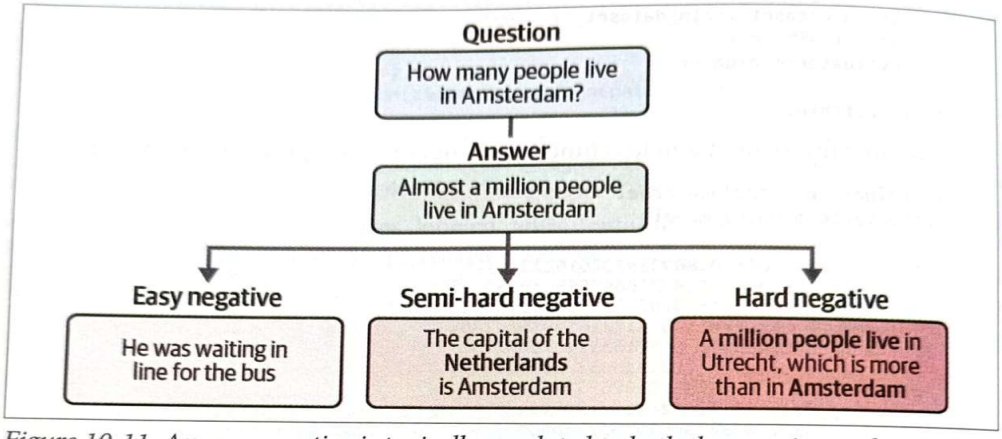


Figure 10-11. An easy negative is typically unrelated to both the question and answer. A semi-hard negative has some similarities to the topic of the question and answer but is somewhat unrelated. A hard negative is very similar to the question but is generally the wrong answer.

Gathering negatives can roughly be divided into the following three processes:

Easy negatives
: Through randomly sampling documents as we did before.

Semi-hard negatives
: Using a pretrained embedding model, we can apply cosine similarity on all sentence embeddings to find those that are highly related. Generally, this does not lead to hard negatives since this method merely finds similar sentences, not question/answer pairs.

Hard negatives
: These often need to be either manually labeled (for instance, by generating semi-hard negatives) or you can use a generative model to either judge or generate sentence pairs.

Make sure to *restart* your notebook so we can explore the different methods of fine-tuning embedding models.

Augmented SBERT

Augmented SBERT refers to enhancing Sentence-BERT (SBERT) using LLMs, external knowledge, or advanced training strategies to produce richer, more robust sentence embedding's for downstream LLM applications such as RAG, semantic search, clustering, and topic modelling.

In short:
SBERT = base semantic encoder
Augmented SBERT = SBERT + LLM/knowledge/data augmentation

Why Do We Need Augmented SBERT?

Standard SBERT embedding's may struggle with:

- Domain-specific language (finance, medicine)
- Logical negation and reasoning
- Sparse or low-resource data
- Hard negative examples

Augmentation improves embedding quality and robustness.

Architecture: Augmented SBERT in LLM Systems:

```
Text / Query
     ↓
LLM Augmentation
(paraphrase, contrast, expand)
     ↓
Augmented Text
     ↓
SBERT Encoder
     ↓
Enhanced Embeddings
     ↓
Vector DB / RAG / BERTopic
```

Key Advantages:

Faster than LLM embedding's
Near-LLM-level semantic quality

Scalable and cost-effective
Ideal for RAG and BERTopic

Aspect	SBERT	Augmented SBERT
Training data	Generic	LLM-augmented
Domain knowledge	Limited	High
Negation handling	Moderate	Improved
Robustness	Medium	High
Retrieval quality	Good	Excellent

Here, a gold dataset is a small but fully annotated dataset that holds the ground truth. A silver dataset is also fully annotated but is not necessarily the ground truth as it was generated through predictions of the cross-encoder.

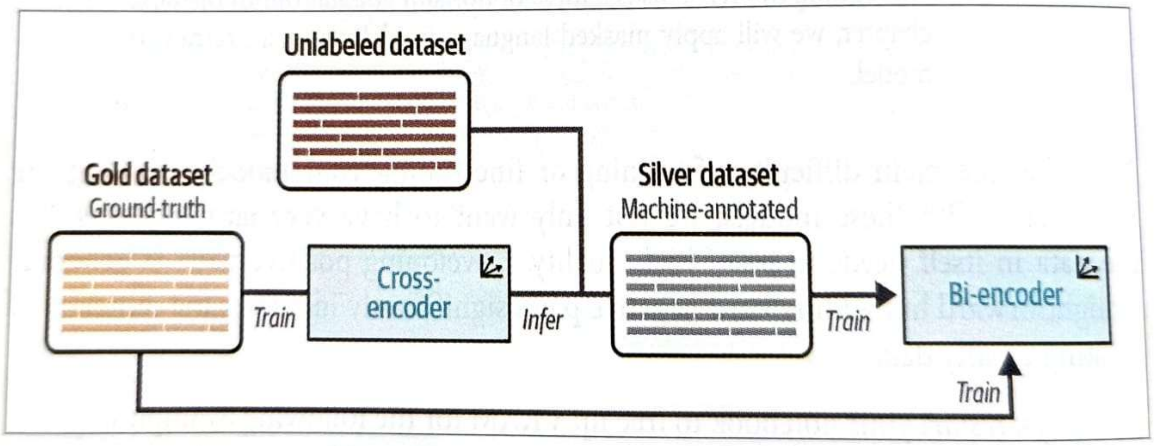


Figure 10-12. Augmented SBERT works through training a cross-encoder on a small gold dataset, then using that to label an unlabeled dataset to generate a larger silver dataset. Finally, both the gold and silver datasets are used to train the bi-encoder.

Supervised Learning

Supervised learning in LLMs uses labelled data, where the correct output (answer, label, or response) is explicitly provided for each input.

Where Supervised Learning Appears in LLMs

1. Supervised Fine-Tuning (SFT)

After pretraining, LLMs are fine-tuned on **instruction–response pairs**.

Example

Input: Explain cosine similarity in NLP.

Output: Cosine similarity measures the angle...

Improves instruction-following
Aligns model behaviour with human expectations

2. Task-Specific Fine-Tuning

Used for:

- Sentiment classification
- NLI
- Named Entity Recognition
- Question answering

3. Embedding Models (SBERT)

SBERT uses **supervised contrastive learning**:

- Positive pairs
- Negative pairs
- Labels indicate similarity or class

4. Human Feedback (RLHF – Partial Supervision)

- Human-labelled preferences
- Rankings or scores guide model optimization

Advantages

✓ High accuracy
✓ Clear objective
✓ Faster convergence

Unsupervised Learning:

Unsupervised learning in LLMs learns patterns from unlabelled data without explicit target outputs.

Where Unsupervised Learning Appears in LLMs

1. Pretraining (Core of LLMs)

LLMs are primarily pretrained using self-supervised learning, a form of unsupervised learning.

Objective

- Predict next token
- Masked language modeling

Example:

"The price of Bitcoin is [MASK]."

2. Representation Learning

- Learning grammar, syntax, semantics
- Capturing world knowledge

3. Topic Modeling & Clustering

- BERTopic
- Document clustering
- Semantic grouping

4. Unsupervised Embedding Training

- Sentence embedding's without labels
- Contrastive learning with in-batch negatives

Advantages

✓ Scales to massive data
✓ No labeling cost
✓ Learns general language structure

Self-Supervised Learning (Bridge Between Both)

LLMs mainly use self-supervised learning, which is:

- Unsupervised in data usage
- Supervised in training signal

Example

- Input = context tokens
- Label = next token (automatically generated)

Supervised vs Unsupervised: Comparison Table

Aspect	Supervised Learning	Unsupervised Learning
Labels	Required	Not required
Main stage	Fine-tuning	Pretraining
Data scale	Smaller	Massive
Cost	High	Low
Control	High	Low
Examples	SFT, SBERT	GPT pretraining