SRM INSTITUTE OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTING TECHNOLOGIES

21CSE356T

NATURAL LANGUAGE PROCESSING

UNIT- 4

S.PRABU
Assistant Professor
C-TECH-SRM-IST-KTR

UNIT 4
RECURRENT NEURAL NETWORKS (RNN) AND TRANSFORMER-BASED MODELS

Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), Attention mechanism, Transformer Based Models, Self-attention, multi-headed attention, BERT, RoBERTa, Fine Tuning for downstream tasks, Text classification and Text generation.

1. Recurrent Neural Networks (RNN)

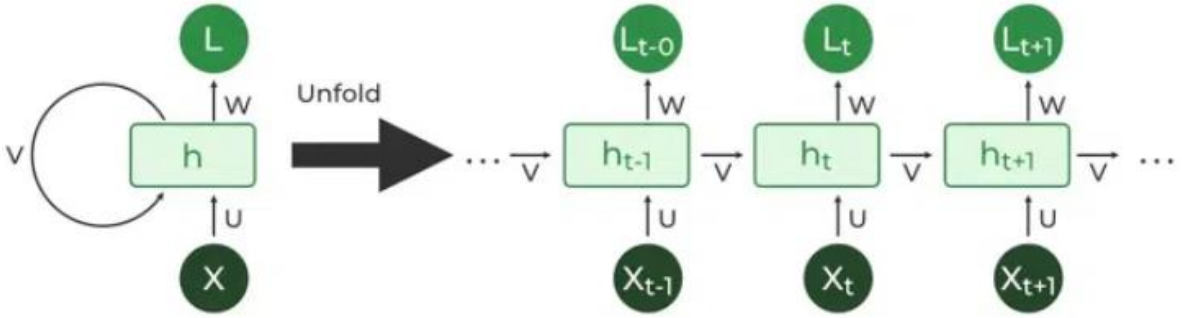
Definition:

Recurrent Neural Networks (RNNs) are a type of neural network designed to process sequential data by maintaining a memory of previous inputs.

Example:

Predicting the next word in a sentence based on previous words.

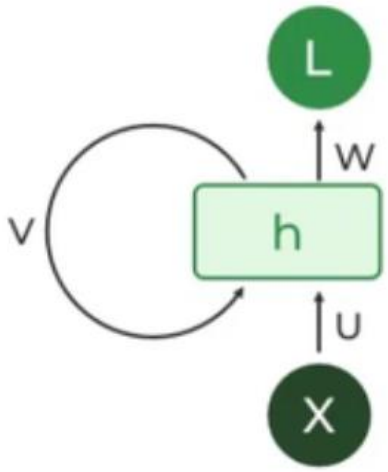
- ✓ Think of it like reading a sentence, when you're trying to predict the next word you don't just look at the current word but also need to remember the words that came before to make accurate guess.
- ✓ RNNs allow the network to "remember" past information by feeding the output from one step into next step.
- ✓ This helps the network understand the context of what has already happened and make better predictions based on that.
- ✓ For example when predicting the next word in a sentence the RNN uses the previous words to help decide what word is most likely to come next.



Key Components of RNNs

1. Recurrent Neurons

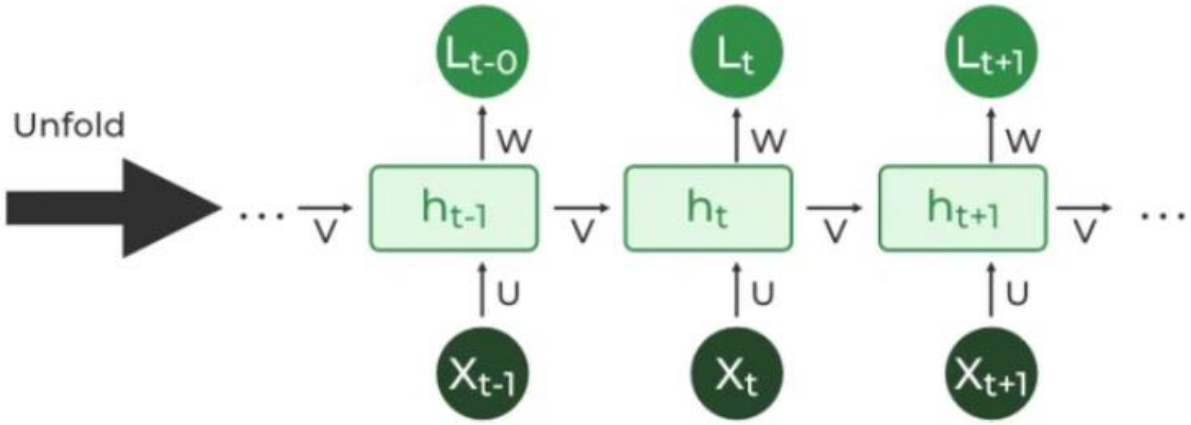
- ✓ The fundamental processing unit in RNN is a Recurrent Unit. Recurrent units hold a hidden state that maintains information about previous inputs in a sequence.
- ✓ Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



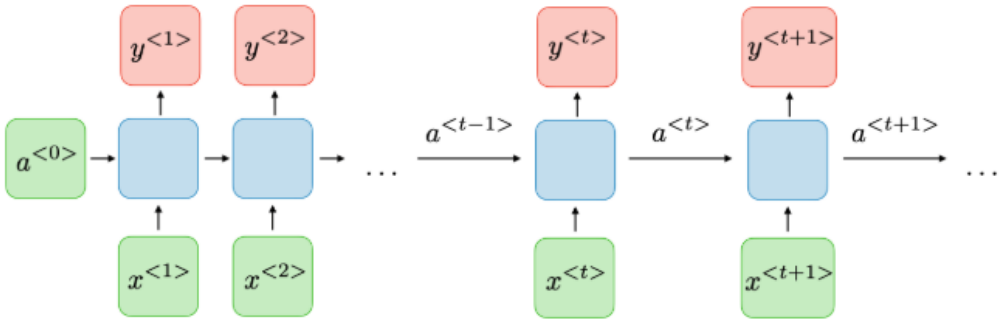
Recurrent Neuron

2. RNN Unfolding

- ✓ RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps.
- ✓ During unfolding each step of the sequence is represented as a separate layer in a series illustrating how information flows across each time step.
- ✓ This unrolling enables backpropagation through time (BPTT) a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data.



Recurrent Neural Network Architecture



For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions.

- ✓ RNNs are a type of neural network with hidden states and allow past outputs to be used as inputs. They usually go like this:
- ✓ Here's a breakdown of its key components:
  - **Input Layer:**
    - ✓ This layer receives the initial element of the sequence data.
    - ✓ For example, in a sentence, it might receive the first word as a vector representation.
  - **Hidden Layer:**
    - ✓ The heart of the RNN, the hidden layer contains a set of interconnected neurons.

- ✓ Each neuron processes the current input along with the information from the previous hidden layer's state.
- ✓ This "state" captures the network's memory of past inputs, allowing it to understand the current element in context.

- **Activation Function:**
  - ✓ This function introduces non-linearity into the network, enabling it to learn complex patterns.
  - ✓ It transforms the combined input from the current input layer and the previous hidden layer state before passing it on.
- **Output Layer:**
  - ✓ The output layer generates the network's prediction based on the processed information.
  - ✓ In a language model, it might predict the next word in the sequence.
- **Recurrent Connection:**
  - ✓ A key distinction of RNNs is the recurrent connection within the hidden layer.
  - ✓ This connection allows the network to pass the hidden state information (the network's memory) to the next time step.
  - ✓ It's like passing a baton in a relay race, carrying information about previous inputs forward

Below are some RNN architectures that can help you better understand this.

- **One To One:** There is only one pair here. A one-to-one architecture is used in traditional neural networks.
- **One To Many:** A single input in a one-to-many network might result in numerous outputs. One too many networks are used in music production, for example.
- **Many To One:** A single output combines inputs from distinct time steps in this scenario. Sentiment analysis and emotion identification use such networks, in which a sequence of words determines the class label.

- **Many to Many:** For many to many, there are numerous options. Two inputs yield three outputs. Machine translation systems, such as English to French or vice versa translation systems, use many-to-many networks.

Advantages and Disadvantages of RNN

Advantages of RNNs:

- Handle sequential data effectively, including text, speech, and time series.
- Process inputs of any length, unlike feedforward neural networks.
- Share weights across time steps, enhancing training efficiency.

Disadvantages of RNNs:

- Prone to vanishing and exploding gradient problems, hindering learning.
- Training can be challenging, especially for long sequences.
- Computationally slower than other neural network architectures.

Key Points:

- RNNs use loops within the network to pass information from one step to the next.
- They are useful for time-series data, speech recognition, and text processing.
- Vanishing gradient problem limits learning long-term dependencies.

2. Long Short-Term Memory (LSTM)

Definition:

LSTM is an advanced RNN variant designed to handle long-term dependencies using memory cells.

Example:

Predicting stock prices based on historical trends.

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) designed to address the vanishing gradient problem, allowing it to effectively learn and retain long-term dependencies in sequential data.

- ✓ Long Short-Term Memory Networks or LSTM in deep learning, is a sequential neural network that allows information to persist.
- ✓ It is a special type of Recurrent Neural Network which is capable of handling the vanishing gradient problem faced by RNN.
- ✓ LSTM was designed by Hochreiter and Schmidhuber that resolves the problem caused by traditional rnns and machine learning algorithms.
- ✓ LSTM Model can be implemented in Python using the Keras library.

What is LSTM?

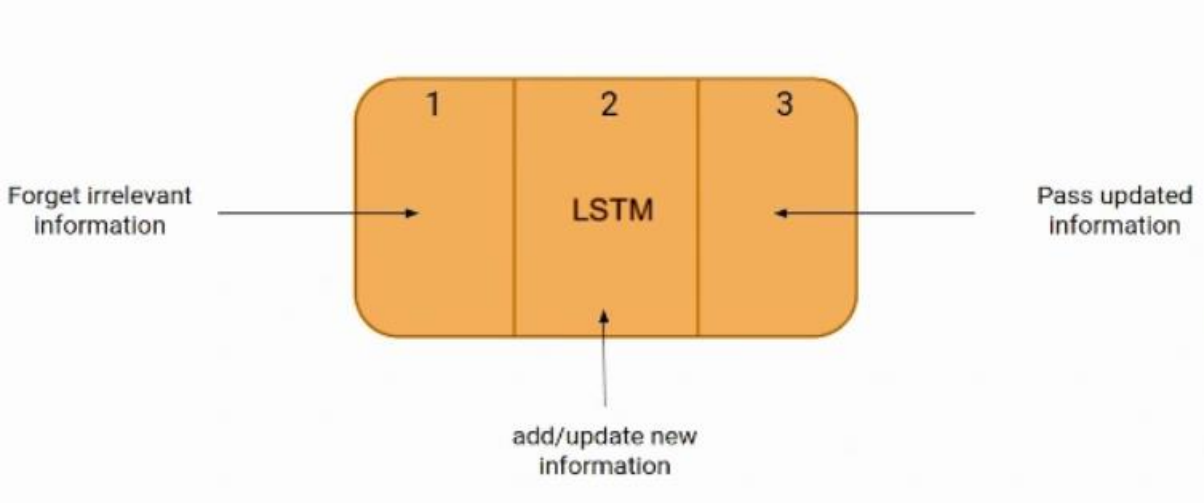
- ✓ LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning.
- ✓ It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks.
- ✓ Unlike traditional neural networks, LSTM incorporates feedback connections, allowing it to process entire sequences of data, not just individual data points.
- ✓ This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.
- ✓ LSTM has become a powerful tool in artificial intelligence and deep learning, enabling breakthroughs in various fields by uncovering valuable insights from sequential data.

LSTM Architecture

- ✓ In the introduction to long short-term memory, we learned that it resolves the vanishing gradient problem faced by RNN.
- ✓ So now, in this section, we will see how it resolves this problem by learning the architecture of the LSTM. At a high level, LSTM works very much like an RNN cell.

✓ Here is the internal functioning of the LSTM network.

✓ The LSTM network architecture consists of three parts, as shown in the image below, and each part performs an individual function.



The Logic Behind LSTM

✓ The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten.

✓ In the second part, the cell tries to learn new information from the input to this cell. At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp.

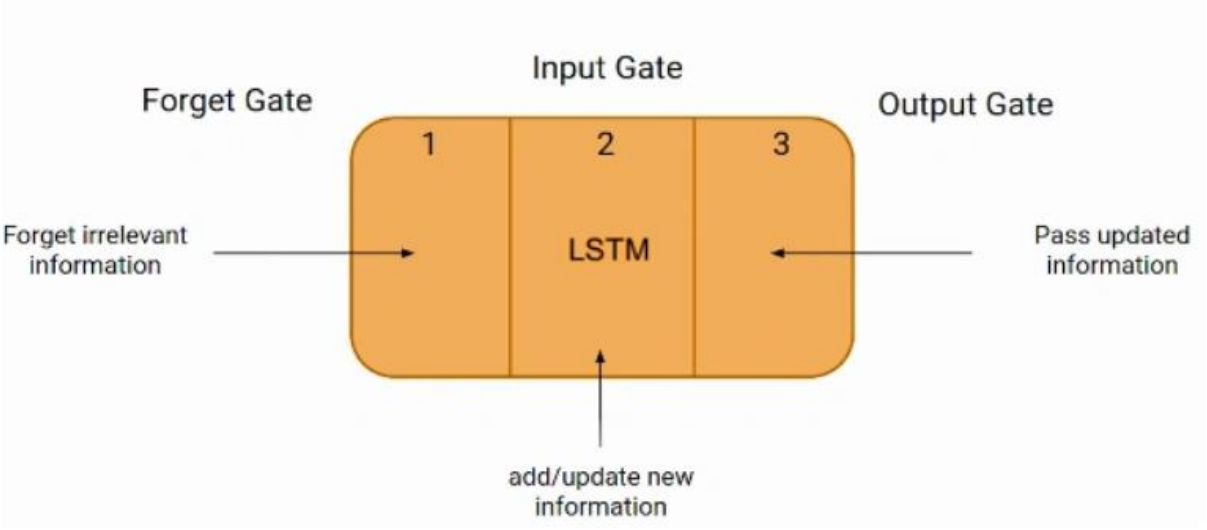
✓ This one cycle of LSTM is considered a single-time step.

✓ These three parts of an LSTM unit are known as gates.

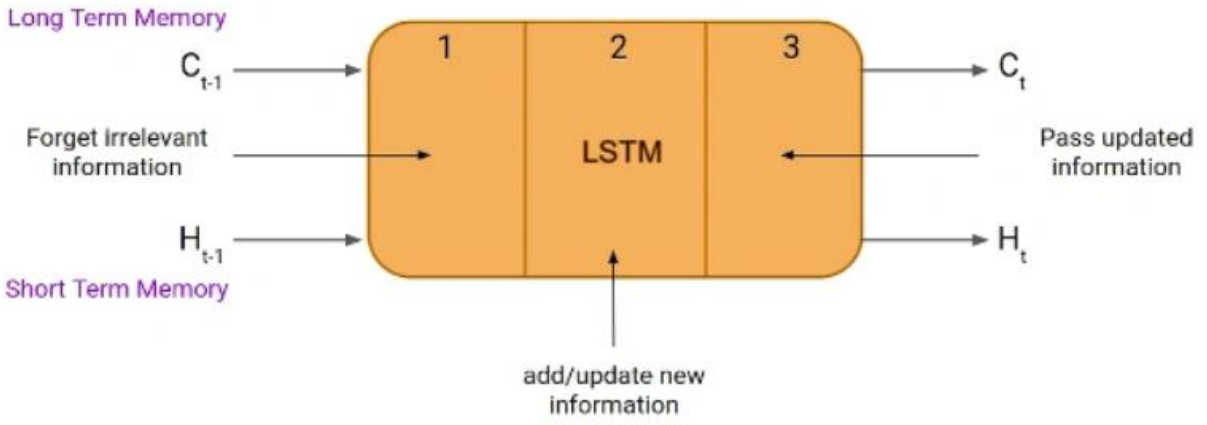
✓ They control the flow of information in and out of the memory cell or lstm cell.

✓ The first gate is called Forget gate, the second gate is known as the Input gate, and the last one is the Output gate.

✓ An LSTM unit that consists of these three gates and a memory cell or lstm cell can be considered as a layer of neurons in traditional feedforward neural network, with each neuron having a hidden layer and a current state.



- ✓ Just like a simple RNN, an LSTM also has a hidden state where H(t-1) represents the hidden state of the previous timestamp and Ht is the hidden state of the current timestamp.
- ✓ In addition to that, LSTM also has a cell state represented by C(t-1) and C(t) for the previous and current timestamps, respectively.
- ✓ Here the hidden state is known as Short term memory, and the cell state is known as Long term memory. Refer to the following image.



- ✓ It is interesting to note that the cell state carries the information along with all the timestamps.

LSTM

Key Points:

- Uses cell states and three gates (input, forget, and output gates) to regulate information flow.
- Solves the vanishing gradient problem.
- Widely used in speech and text applications.

3. Attention Mechanism

Definition:

✓ The attention mechanism is a technique used in machine learning and natural language processing to increase model accuracy by focusing on relevant data.

✓ It enables the model to focus on certain areas of the input data, giving more weight to crucial features and disregarding unimportant ones.

Example:

- ranslating a sentence from English to French
- Let's say we're translating:
- Input (English): The cat sat on the mat
- Target (French): Le chat était assis sur le tapis

Problem Without Attention:

- ✓ A simple Seq2Seq model encodes the whole input sentence into a single vector, then decodes it.
- ✓ This doesn't work well for long or complex sentences, because it squeezes too much info into one vector.

Attention Mechanism to the Rescue:

- ✓ Instead of compressing everything into one vector, the attention mechanism allows the model to look at different parts of the input sentence at each decoding step.

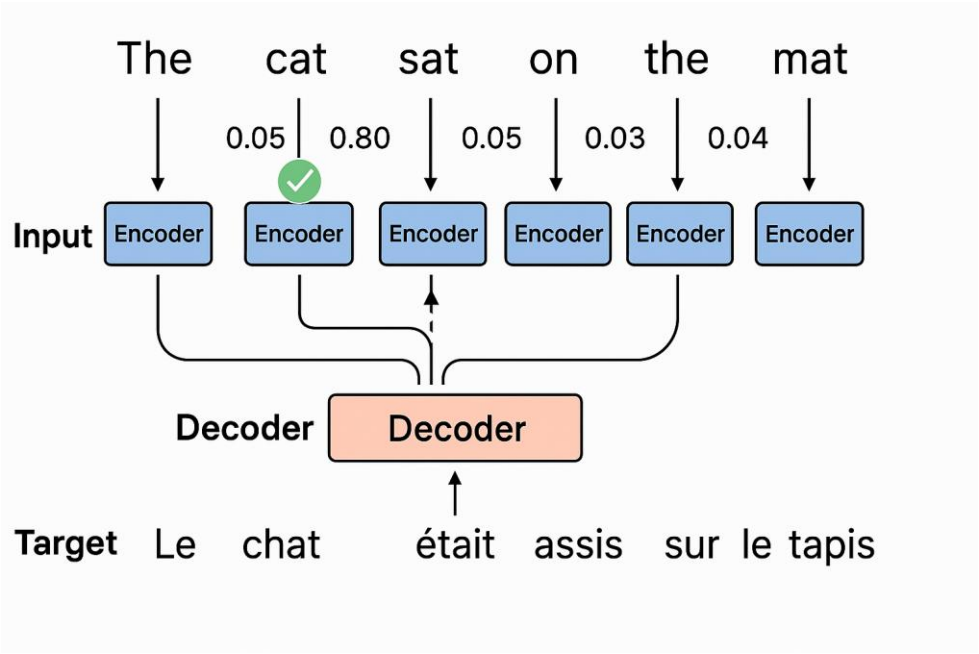
The attention mechanism will:

1. Compare the current decoder state with all encoder hidden states.
2. Compute attention scores (like weights) for each input word.
3. Take a weighted sum of input words' representations.
4. Use this "context vector" to help predict the next word.

- ✓ Imagine the attention scores (weights) when generating "chat" look like this:

Input Word	Attention Score
The	0.05
cat	0.80
sat	0.05
on	0.03
the	0.04
mat	0.03

- ✓ Here, the model gives **most of its "attention" to "cat"**,
- ✓ Because that's the most relevant for generating "chat".



The attention mechanism computes:

$$\text{Attention Score} = \text{Alignment}(h_{\text{decoder}}, h_{\text{encoder}})$$

Then applies **softmax** to get probabilities.

The context vector is:

$$\text{Context} = \sum_i \alpha_i h_i$$

where α_i is the attention weight for word i.

Advantage

- Helps with long sentences.
- Focuses on relevant parts.
- Improves translation, summarization, and many NLP tasks.

4. Transformer-Based Models

Definition:

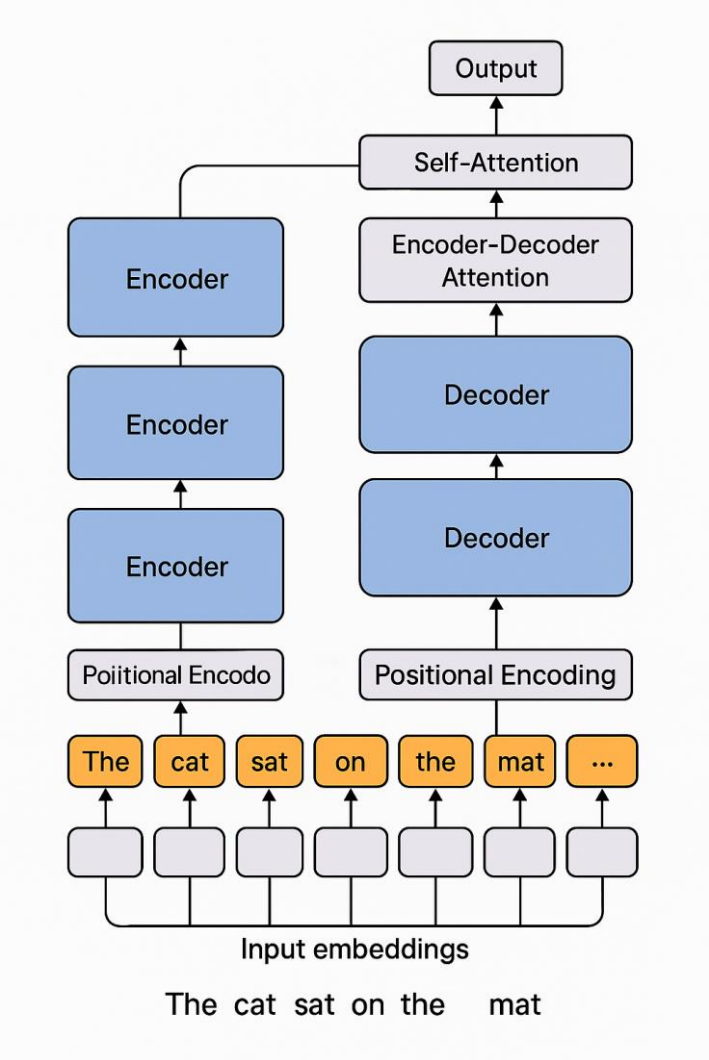
A transformer is a deep learning model that uses **self-attention** to process input data all at once (not sequentially like RNNs), making it faster and better at capturing context in long sequences.

Example:

Google's BERT and OpenAI's GPT.

Key Points:

- No recurrent connections, unlike RNNs.
- Highly parallelized, improving efficiency.
- Forms the foundation for modern NLP models.



The diagram is divided into two main parts:

1. **Encoder (Left Side)**
2. **Decoder (Right Side)**

These two components work together to transform an input sentence into a desired output (e.g., in translation).

Input Embedding + Positional Encoding

- The input sentence "The cat sat on the mat" is first converted into **word embeddings**.
- Then, **positional encoding** is added so the model knows the order of the words.

Encoder Side

- The encoder processes the input using **self-attention layers** and **feed-forward networks**.
- Each word in the sentence is compared with every other word to capture meaning and context.
- The encoder outputs a set of encoded vectors that contain rich information about the sentence.

Decoder Side

- The decoder takes in previously generated words (like "Le", "chat", etc.) and uses:
    - **Masked self-attention** to prevent looking ahead.
    - **Encoder-decoder attention** to focus on relevant parts of the input sentence.
- Based on this, the decoder generates the next word in the output, such as "chat" (French for "cat").

Self-Attention

- Self-attention allows the model to **weigh the importance of each word** in a sentence relative to others, helping it understand meaning and context better.

Final Output

- The decoder continues this process until the full output sentence is generated, such as "Le chat était assis sur le tapis".

This structure is the backbone of powerful models like **BERT**, **GPT**, and **T5**, which are widely used in NLP tasks like translation, summarization, question answering, and more.

5. Self-Attention Mechanism

Definition:

Self-attention computes relationships between different positions of a sequence to weigh their importance.

Example:

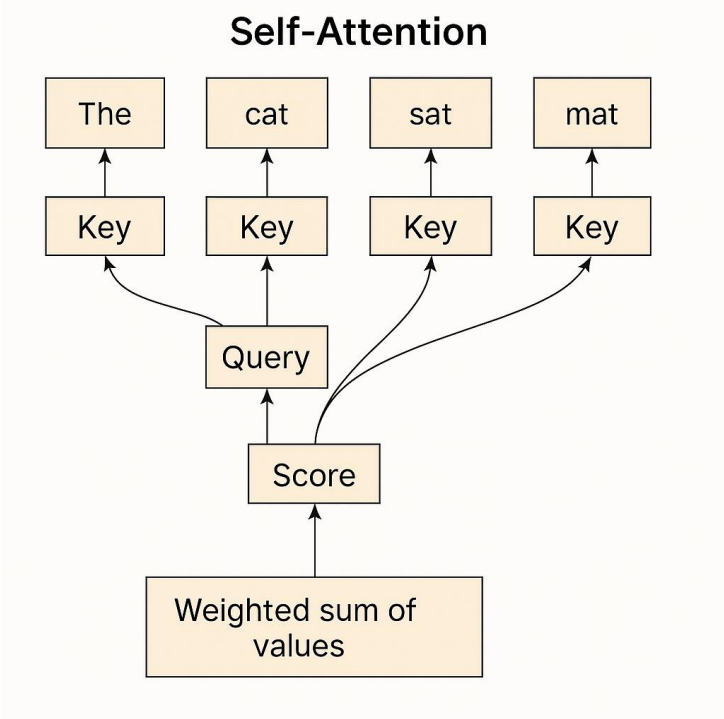
Understanding word meaning based on context in a sentence.

Key Points:

- Helps focus on important words within a sequence.
- Used in transformers to replace recurrence.
- Reduces dependency on past states.

Purpose of Self-Attention

Self-attention helps the model figure out **which words in a sentence should be focused on** when understanding a particular word.



1. **Input Words**

   Example: "The", "cat", "sat", "mat" – these are the words the model is processing.

2. **Query, Key, Value Vectors**

   Each word is transformed into 3 vectors:

   - **Query (Q)**: What the word is **looking for** in others.
   - **Key (K)**: What this word **offers** to others.
   - **Value (V)**: The **actual information** of the word.

3. **Scoring ($Q \cdot K^T$)**

   Each word's **Query** is compared with every word's **Key** using dot product. This gives **attention scores** that tell how much one word should pay attention to another.

4. **Softmax Normalization**

   The scores are passed through **softmax** to turn them into probabilities (attention weights), so they all add up to 1.

5. **Weighted Sum of Values**

   The final output for a word is a **weighted combination of all Value vectors**, based on those attention weights.

Example

If the model is focusing on the word "sat", it might pay:

- high attention to "cat" (subject),
- medium to "mat" (object), and
- low to "The".

So "sat" will be represented by combining information from all other words, with more weight on "cat" and "mat".

Result

The output vector for each word is now **rich in context**, helping the model understand not just the word, but its relationship to others.

6. Multi-Headed Attention

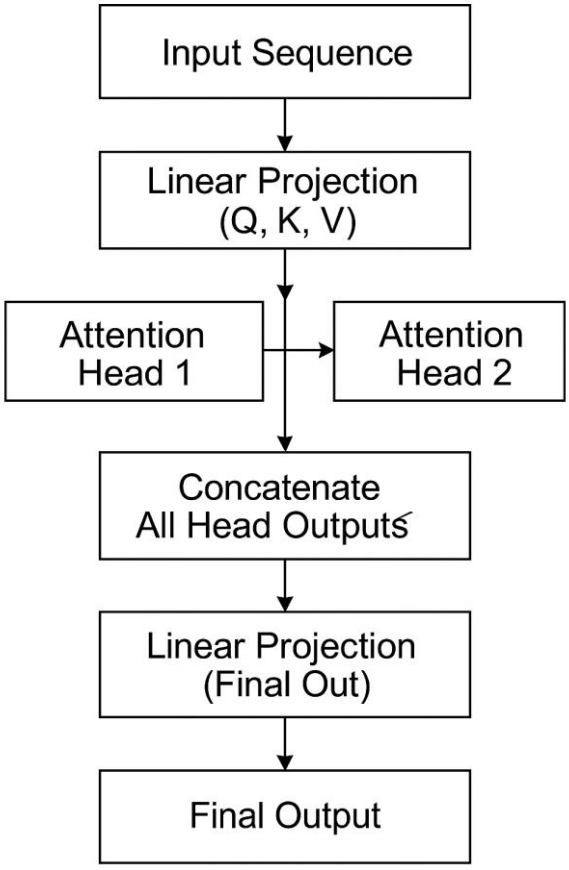
Definition:

Multi-headed attention runs multiple self-attention mechanisms in parallel to capture different types of relationships.

Multi-Headed Attention is a mechanism used in transformer models that allows the model to focus on different parts of a sentence or sequence simultaneously. It uses multiple attention "heads" to capture various types of relationships (e.g., syntactic, semantic) between words in different subspaces.

Example:

Translating a sentence while considering grammar and meaning separately.



1.Input Sequence

- ✓ The process begins with an **input sequence**—this could be a sentence like: **"The cat sat on the mat."**
- ✓ Each word in the sentence is first converted into a numerical vector (called an **embedding**) using word embedding techniques.
- ✓ These embeddings are then passed into the Transformer model.

2: Linear Projections into Q, K, and V

- ✓ For each input vector, we create three versions:
  - **Q** (Query)
  - **K** (Key)
  - **V** (Value)
- ✓ These are obtained by multiplying the input by three different weight matrices.
- ✓ This is called a linear transformation, and it helps the model learn how words relate to each other.

Why do we need Q, K, and V?

- **Query**: what we're looking for.
- **Key**: what each word offers.
- **Value**: what we finally use to compute the output.

3: Multiple Attention Heads

✓ Instead of doing attention once, we do it multiple times in parallel.

✓ Each attention head performs Scaled Dot-Product Attention on the Q, K, and V from the previous step.

Why multiple heads?

- Each head focuses on different parts of the sentence.
- For example:
  - Head 1 might focus on grammatical structure.
  - Head 2 might learn positional relationships.
  - Head 3 might focus on meaning and context.

✓ This is what gives multi-headed attention its power—different perspectives are learned simultaneously, increasing the richness of the model's understanding.

4: Concatenation of Heads

✓ Once each attention head has produced its own output vector (representing what it has focused on), we **concatenate** (combine side-by-side) all of them into a single large vector.

✓ This merged vector contains a diverse set of learned relationships between words.

5: Final Linear Projection

✓ The concatenated vector is passed through another linear layer.

✓ This final transformation brings all the attention outputs together into one coherent output that can be passed to the next layer in the Transformer model (or used directly for predictions).

6: Final Output

- ✓ This is the result of the entire Multi-Headed Attention process.
- ✓ It is a context-aware representation of each word in the sentence.
- ✓ Each word's output vector now contains information not just about itself, but also about how it relates to all other words in the sentence.
- ✓ Multi-Headed Attention is a core component of Transformer models like BERT, GPT, and T5.
- ✓ It allows the model to understand language from multiple angles at once.
- ✓ This mechanism improves accuracy, context understanding, and handling of long-distance dependencies in text.

7. BERT (Bidirectional Encoder Representations from Transformers)

Definition:

- ✓ BERT is a pre-trained language model using transformers for understanding text contextually.
- ✓ BERT stands for Bidirectional Encoder Representations from Transformers.
- ✓ It is a deep learning model developed by Google AI in 2018 and is designed to understand the context of words in a sentence in both directions — left-to-right and right-to-left — simultaneously.

Key Features of BERT:

1. Bidirectional: Unlike earlier models that read text in one direction (e.g., left to right), BERT reads in both directions using the Transformer encoder. This helps it understand the full context of a word based on its surroundings.
2. Based on Transformer: Specifically, BERT uses only the encoder part of the Transformer architecture (no decoder).
3. Pre-trained Language Model: BERT is first pre-trained on large datasets (like Wikipedia and BookCorpus) using self-supervised tasks, and then fine-tuned on specific tasks like sentiment analysis, question answering, etc.

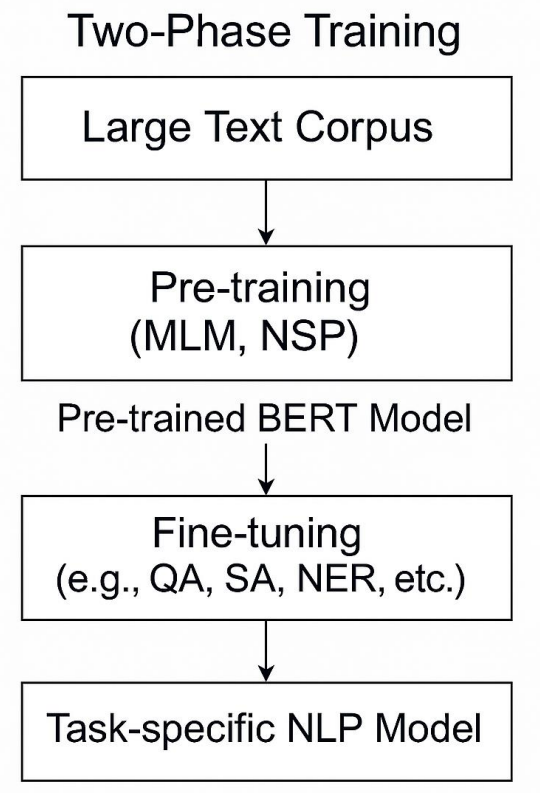
Two-Phase Training of BERT

1. Pre-training Phase

- **Goal:** Train BERT on a large text corpus (like Wikipedia).
- **Techniques**:
  - Masked Language Modeling (MLM): Randomly masks some words in a sentence and trains BERT to predict them.
  - Next Sentence Prediction (NSP): BERT learns if a sentence B follows sentence A.
- **Result**: General language understanding.

2. Fine-tuning Phase

- **Goal**: Adapt BERT to a specific NLP task (e.g., sentiment analysis, question answering).
- **Process**: Add a small task-specific layer on top of BERT and train it on a smaller, labeled dataset.



1. Large Text Corpus

- **Description**: This is a huge amount of unlabeled text data (like Wikipedia, BookCorpus, etc.).
- **Purpose**: Acts as the raw material to teach BERT general language understanding during pre-training.

2. Pre-training (MLM, NSP)

- **Masked Language Modeling (MLM)**:
  - Randomly masks some words in the input sentence.
  - BERT learns to predict the missing/masked words based on surrounding context.
  - Example: "The cat sat on the [MASK]." → Predict "mat".
- **Next Sentence Prediction (NSP)**:
  - BERT is shown sentence pairs and learns to predict if the second sentence follows the first in the original text.
  - Helps BERT understand sentence relationships.
- **Output**: A **pre-trained BERT model** that understands the general structure and flow of natural language.

3. Fine-tuning (e.g., QA, SA, NER, etc.)

- **Description**: This step adapts the pre-trained BERT model to specific NLP tasks.
- **Examples**:
  - **QA (Question Answering)** – Extracting answers from a passage.
  - **SA (Sentiment Analysis)** – Classifying the tone of a text (positive/negative).
  - **NER (Named Entity Recognition)** – Identifying names, places, dates, etc.
- **Process**: Add a task-specific output layer and fine-tune the entire model on labeled data for the specific task.

4. Task-specific NLP Model

- **Description**: The final product after fine-tuning.
- **Result**: A model specialized in performing a certain NLP task using BERT's deep language understanding.

Example:

Input Sentence: "The cat sat on the [MASK]."

BERT learns to fill in "mat" during **pre-training**.

Later, it is **fine-tuned** on a sentiment analysis dataset to classify texts as positive or negative.

BERT Variants and Extensions

Several variants and extensions of BERT have been developed to address its limitations and expand its capabilities:

Several variants and extensions of BERT have been developed to address its limitations and expand its capabilities:

- **RoBERTa** (Robustly Optimized BERT Pretraining Approach): An optimized version of BERT that adjusts key hyperparameters and removes the next-sentence prediction mechanism, improving performance on various NLP tasks. RoBERTa enhances BERT's robustness by training on more data and using longer sequences, making it more effective in capturing context.
- **DistilBERT**: A smaller, faster, and more efficient version of BERT, suitable for environments with limited resources. DistilBERT achieves nearly the same performance as BERT while being 60% faster and using 40% fewer parameters, making it ideal for applications with critical speed and efficiency.
- **ALBERT** (A Lite BERT): A light version of BERT that reduces model size without significantly impacting effectiveness, addressing the resource demands of BERT. ALBERT achieves this reduction by sharing parameters across layers and factorizing the embedding matrix, making it a more resource-efficient alternative while maintaining high accuracy.

- **ColBERT** (Contextualized Late Interaction over BERT): A refined extension of BERT that focuses on efficient retrieval in large text collections. ColBERT is designed for document retrieval and ranking tasks. It balances the need for deep contextual understanding with computational efficiency.
- **BGE-M3** (BERT-based Generative Encoder Multilingual Model): An advanced machine-learning model that extends BERT's capabilities, particularly in multilingual tasks. BGE-M3 leverages BERT's architecture to improve performance across different languages, making it a valuable tool for global applications.
- **Splade** (Sparse Lexical and Dense Retrieval): An evolution in generating learned sparse embeddings, building upon the foundational BERT architecture with a unique methodology to refine embedding sparsity. Splade combines the strengths of sparse and dense retrieval methods, providing a more efficient and effective approach to information retrieval tasks.

✓ These variants and extensions showcase BERT's adaptability and ongoing relevance in the evolving field of statistical natural language processing using.

✓ They represent efforts to overcome the challenges posed by the original BERT model, making it more accessible, efficient, and effective for a broader range of applications.

8. RoBERTa (Robustly Optimized BERT Pretraining Approach)

Definition:

RoBERTa is an improved version of BERT, trained with more data and better techniques.

Example:

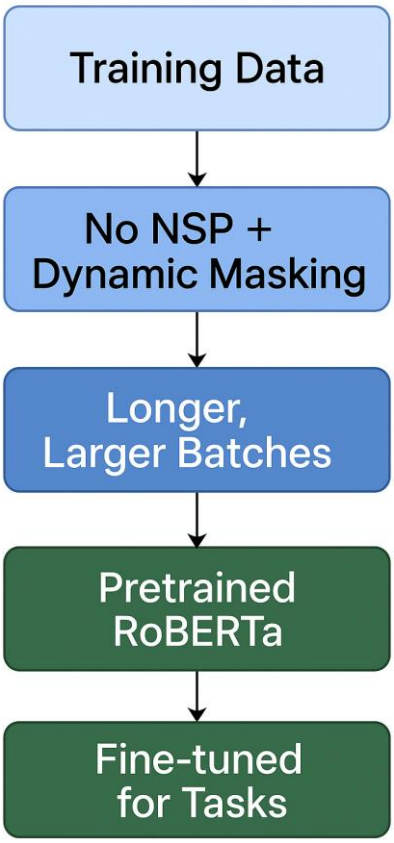
Used in AI applications for better text comprehension.

Key Points:

- Removes the Next Sentence Prediction (NSP) task.
- Uses dynamic masking for better training.
- Achieves better accuracy than BERT.

Key Improvements Over BERT

Feature	BERT	RoBERTa
Training Data	16GB (BooksCorpus + Wikipedia)	160GB (from more diverse sources like CC-News, OpenWebText)
Next Sentence Prediction	Used during pretraining (NSP)	Removed – Found to be not helpful
Batch Size	Small	Much larger
Training Time	Less	Longer training with more data
Dynamic Masking	Static masking (fixed once)	Dynamic masking (changes every epoch)



1. Training Data

- Description: RoBERTa uses 160GB of unlabeled data from diverse sources such as:
    - Common Crawl News (CC-News)
    - OpenWebText
    - BooksCorpus
    - Wikipedia
- Purpose: More data = better language understanding.

2 No NSP + Dynamic Masking

- No NSP (Next Sentence Prediction):
    - RoBERTa removes the NSP task that BERT uses.
    - Found it did not improve performance significantly.

- Dynamic Masking:
    - Instead of fixing masked tokens once, RoBERTa reshuffles the mask tokens in every training epoch.
    - Leads to better generalization and robustness.

3. Longer, Larger Batches

- Batch Size: RoBERTa uses much larger batch sizes.
- Training Time: Trained longer across more steps.
- Effect: Helps the model learn deeper and more stable language representations.

4. Pretrained RoBERTa

- **Result**: A robust, powerful pretrained language model.
- **Purpose**: Now ready to be adapted for downstream tasks like QA, Sentiment Analysis, etc.

5. Fine-tuned for Tasks

- Just like BERT, RoBERTa is **fine-tuned** on **specific labeled datasets**.
- Examples:
    - **SQuAD** → Question Answering
    - **GLUE** → General Language Understanding Evaluation
    - **RACE** → Reading Comprehension

9. Fine-Tuning for Downstream Tasks

Definition:

Fine-tuning adapts a pre-trained model for specific NLP tasks.

Example:

Customizing BERT for sentiment analysis.

Key Points:

- Uses transfer learning.
- Requires labeled data for task-specific training.
- Reduces the need for large datasets.

✓ Fine-tuning is the process of adapting a pretrained language model (like BERT or RoBERTa) to a specific NLP task using a smaller, labeled dataset.

Process of Fine-Tuning

1. Start with a Pretrained Model

- Use a model like RoBERTa, already trained on a massive corpus.
- It understands general language patterns.

2. Add a Task-Specific Layer

- Add a lightweight neural network layer (like a classifier or sequence tagger) on top.
- The structure depends on the task:
    - [CLS] → Softmax for classification
    - Token-wise classifier for Named Entity Recognition

3. Train on Task Dataset

- Use a labeled dataset tailored to the task.
- Adjust the entire model weights (not just the new layer).
- Use a lower learning rate to avoid "forgetting" the pretraining.

4. Evaluate and Deploy

- Check accuracy, F1 score, etc.
- Deploy the model for real-world predictions.

Common Downstream NLP Tasks

Task	Description	Output Example
Sentiment Analysis	Detect tone of a sentence	"Positive", "Negative"
Question Answering (QA)	Extract answers from a passage	"Paris" from "What is the capital of France?"
Named Entity Recognition (NER)	Detect entities like names, dates, places	[Steve] [Jobs] → PERSON
Text Classification	Categorize texts into labels	Email → “Spam” or “Not Spam”
Text Summarization	Generate a short summary of a document	Summary sentence(s)
Textual Entailment	Determine logical relation between sentence pairs	"Entailment", "Contradiction", "Neutral"

Fine-Tuning vs Pretraining

Feature	Pretraining	Fine-tuning
Data Type	Unlabeled text	Labeled task-specific data
Dataset Size	Huge (e.g., 160GB)	Small to medium
Purpose	Learn general language representation	Learn to perform a specific task
Time Required	Days or weeks	Minutes to hours

10. Text Classification

Definition:

Text classification assigns categories to text based on content.

Example:

Classifying emails as spam or not spam.

Key Points:

- Uses NLP models like BERT and LSTM.
- Requires labeled training data.
- Used in sentiment analysis, topic detection.

✓ Text classification in Natural Language Processing (NLP) is the task of assigning predefined labels or categories to text documents, sentences, or phrases based on their content, enabling computers to understand and organize large amounts of unstructured text.

How it works:

NLP techniques and machine learning algorithms analyze text content and assign the most appropriate label or category.

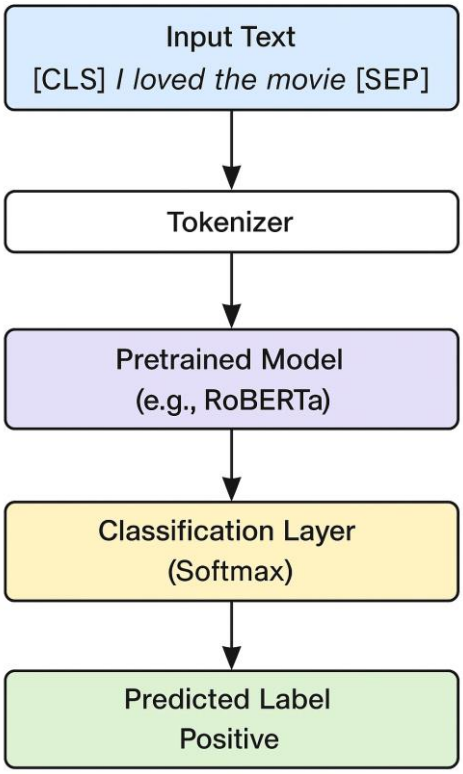
- **Examples:**
  - **Spam detection:** Classifying emails as spam or not spam.
  - **Sentiment analysis:** Determining the sentiment (positive, negative, or neutral) of a piece of text, such as customer reviews.
  - **Topic categorization:** Assigning articles or documents to specific topics.
  - **Customer support ticket classification:** Categorizing support requests based on the problem they describe.

- **Types of Text Classification:**
  - **Binary classification:** Assigning text to one of two categories (e.g., spam/not spam).
  - **Multiclass classification:** Assigning text to one of multiple categories (e.g., news articles to different genres).
  - **Multilabel classification:** Assigning text to multiple categories simultaneously (e.g., a news article could be classified as "sports", "technology", and "breaking news").

Applications:

- Organizing and managing large amounts of unstructured data .
- Improving search results and content recommendations .
- Developing chatbots and other AI applications .
- Detecting hate speech and other harmful content .

Worling Principle:



1. Input Text

- **Example**: *"The product quality is amazing!"*
- This is the raw text that we want to classify.

2. Tokenization

- The sentence is split into subwords or tokens.
- Special tokens like [CLS] and [SEP] are added.
- Example tokens: [CLS] the product quality is amazing ! [SEP]

3. Pretrained Language Model (BERT / RoBERTa)

- The tokenized input is fed into a pretrained model like RoBERTa.
- The model processes the sequence and produces contextual embeddings for each token.
- The embedding of the [CLS] token is treated as a summary of the entire sentence.

4. [CLS] Token Output

- The output vector corresponding to the [CLS] token is extracted.
- This is a dense feature representation of the input text.

5. Classification Layer (Softmax)

- A fully connected dense layer is applied to the [CLS] output.
- A Softmax function is used to convert outputs into probability scores for each class.

6. Predicted Label

- The model selects the class with the highest probability.
- Example output: Positive

Example Summary:

Input:

"The product quality is amazing!"

Model Prediction:

Positive

The model understands sentiment and assigns the appropriate label using learned language patterns.

11. Text Generation

Definition:

Text generation models create new text based on learned patterns.

Example:

AI-generated news articles.

Key Points:

- Uses transformers (GPT) for better fluency.
- Can generate human-like text.
- Applied in chatbots, story writing.

✓ Text generation is the task of automatically generating natural language text from input prompts.

✓ The model learns to predict the next word/token based on the given context.

Common Examples

Input (Prompt)	Output (Generated Text)
"Once upon a time in a forest,"	"there lived a wise old owl who watched over the animals."
"Explain gravity in simple terms."	"Gravity is the force that pulls objects toward each other."
"Translate: Hello, how are you?"	"Hola, ¿cómo estás?"

Types of Text Generation

1. **Open-ended Generation**
    - Story writing, poetry, chat responses
    - Model: GPT-2, GPT-3, LLaMA, etc.
2. **Conditional Generation**
    - Summarization, translation, Q&A
    - Model: T5, BART, mT5, etc.
3. **Masked Generation (less common)**
    - Fill in the blanks
    - Model: BERT (for token-level predictions)

How it works

1. Prompt / Input Text

What it is:

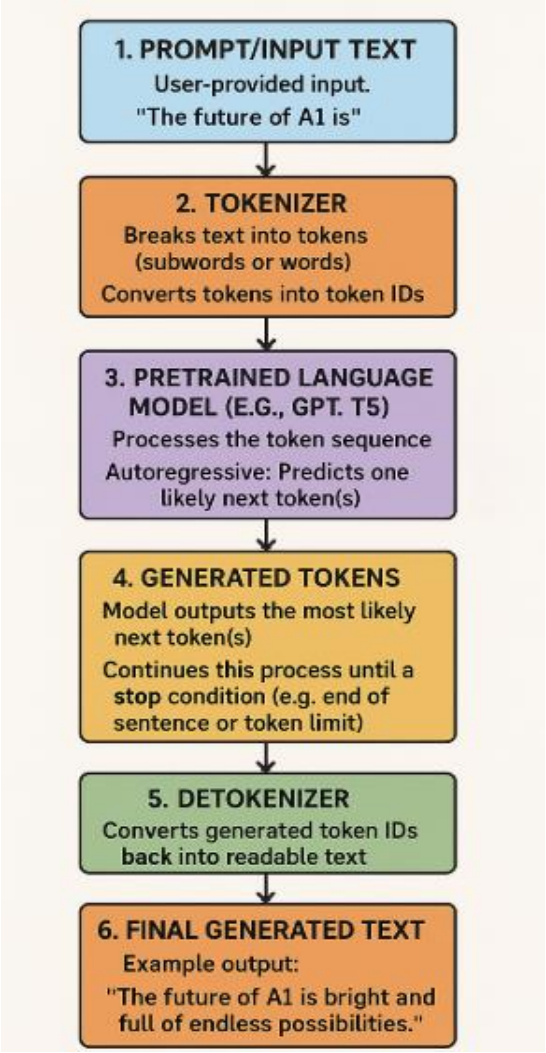
This is the initial text you give to the language model.

Example:

"The future of AI is"

Purpose:

It sets the context. The model uses this to start generating relevant output.



2. Tokenizer

What it does:

- Splits the input into smaller pieces called **tokens** (like words or parts of words).
- Converts those tokens into numbers called **token IDs** that the model can understand.

Example:

"The future of AI is" → tokens like "The", "future", "of", "AI", "is" → token IDs like [627, 1582, 318, 995, 318] (numbers vary by model).

3. Pretrained Language Model (e.g., GPT, T5)

What it does:

- Reads the token ID sequence.

- Predicts the next most likely token based on previous ones.
- Uses an autoregressive process (one token at a time, adding each predicted token back in to predict the next).

Example process:

Given "The future of AI is", it predicts the next token, then uses "The future of AI is bright" to predict the next, and so on.

4. Generated Tokens

What it does:

- Outputs one token at a time.
- Keeps generating until a stop condition is reached:
    - End-of-sentence token
    - Max length
    - Special stop signal

Example:

Generates token IDs like [1123, 290, 1048, 1427], which might represent "bright and full of".

5. Detokenizer

What it does:

- Converts the generated token IDs back into human-readable text.
- Merges subwords and adds spaces or punctuation if needed.

Example:

[1123, 290, 1048, 1427] → "bright and full of"

6. Final Generated Text

What it is:

The complete output after all tokens have been generated and detokenized.

Example output:

"The future of AI is bright and full of endless possibilities."