BERT MODEL

Data Mining and Natural Language Processing

Presented by

Fandishe Mussa

May 2025

CONTENT

1. INTRODUCTION

What is Bert?

- BERT (**Bidirectional Encoder Representations from Transformers**) stands as an open-source machine learning framework designed for the natural language processing (NLP).
- It is a breakthrough model by Google in 2018 that set new standards for NLP tasks.
- BERT leverages(uses) a transformer-based neural network to understand and generate human-like language.
- BERT employs an encoder-only architecture.
- In the original **Transformer architecture**, there are both **encoder** and **decoder** modules.
- The decision to use an encoder-only architecture in BERT suggests a primary emphasis on understanding input sequences rather than generating output sequences.

Why is BERT important in NLP?

- Traditional language models process text sequentially, either from left to right or right to left.
- This method limits the model's awareness to the immediate context preceding the target word.
- BERT uses a bi-directional approach considering both the left and right context of words in a sentence, instead of analyzing the text sequentially, BERT looks at all the words in a sentence simultaneously.
- *Example: "**The bank is situated on the _____ of the river.**"*
- *In a **unidirectional mode**l, the understanding of the blank would heavily depend on the preceding words, and the model might struggle to discern whether "**bank**" refers to a financial institution or the side of **the river**.*
- ***BERT**, being bidirectional, simultaneously considers both the left ("**The bank is situated on the**") and right context ("**of the river**"), enabling a more nuanced understanding.*
- *It comprehends that the missing word is likely related to the geographical location of the bank, demonstrating the contextual richness that the bidirectional approach brings.*

2. BACKGROUND

Brief introduction to NLP tasks

- Natural Language Processing helps machines to process, analyze and generate human like content.
- It helps search engines to answer questions, translating languages and summarizing texts.
- NLP is used in various tasks that make human-computer interaction smoother.

1. Text Classification

- Text classification is like sorting different pieces of text into specific groups or categories.
- Imagine you have a bunch of emails and you want to separate them into "**important**" and "not important" piles.
- Text classification helps computers do this automatically by categorizing text into predefined groups.

How It Works?

- To perform text classification we first teach the computer using examples of text already sorted into categories. We provide labeled examples like positive and negative reviews or spam and non-spam emails. Once the computer learns from these examples it can automatically sort new, unseen text into the right category
- Some of its examples are:

Sentiment Analysis: When you figure out if a movie review is happy (positive) or sad (negative). For example, if a review says, "I loved the movie! It was amazing," the sentiment is positive.

Spam Detection: This task helps identify if an email is junk (spam) or real (non-spam). For example, an email saying "Win a free trip to Hawaii!" is likely spam.

2. Token Classification

- Token classification is like labeling each word in a sentence.
- It’s often used to find special words like names, places or companies.
- Imagine you have a sentence and you want to highlight the important words, like a person's name or a city.

How It Works?

- Computer looks at each word and its neighbors to figure out what label fits best.
- The model learns to recognize patterns and contexts, so it can label words like "Apple" as a company and "California" as a location.
- This is particularly useful for tasks like extracting information from text.
- Some of its examples are:

Named Entity Recognition (NER): In the sentence "Apple is based in California," "Apple" is a company and "California" is a place. Token classification helps identify these special words and label them correctly.

CONT'D

3. Question Answering

- Question answering is like playing a trivia game with a computer.
- You ask a question and the computer finds the answer in a piece of text.
- It’s like having a smart assistant that can read a document and answer your questions based on what it reads.

How It Works

- Computer is trained on lots of questions and answers.
- It learns to understand the question and find the right answer in the text.
- Model looks at the context of the question and locates the specific information in the passage to provide an accurate answer.
- This is useful for applications like virtual assistants and search engines.
- Some of its examples are:
  Open-domain Question Answering: If you ask "**Who founded the country?"** and give it a passage about history, the computer finds the answer.
  For example, if the passage says, "**The country was founded by George Washington**," the computer will answer "George Washington."

4. Casual Language Modeling

- **Casual Language Model** is a type of AI model designed to generate text in an informal, conversational tone.
- This type of model is optimized to produce responses that mimic everyday, casual conversation, making it suitable for applications where a relaxed, friendly and approachable tone is desired.

How It Works?

- Casual Language Models are trained on datasets that include informal and conversational text, such as social media posts, chat logs and casual conversations.
- Model learns to generate text that sounds natural and friendly, similar to how people speak in everyday interactions.
- This training process helps the model understand the nuances of casual language, including slang, colloquialisms and informal expressions.
- Some of its examples are:
  Meena: A conversational AI model developed by Google that aims to generate human-like respones in casual conversations.
  DialoGPT: A model trained on Reddit conversations to generate casual and engaging responses.

CONT'D

5. Masked Language Modeling

- Masked language modeling is like a fill-in-the-blank game.
- Computer has to guess a missing word in a sentence.
- It’s a common approach in training language models, especially for models like BERT.

How It Works

- Computer looks at the words around the blank to figure out what word fits best.
- It’s like solving a puzzle where each word gets a tag.
- Model learns to predict the missing word based on the context provided by the surrounding words.
- This helps it understand relationships between words and their context in sentences.
- Some of its examples are:

  Fill-in-the-Blank Tasks: For "The cat is [MASK] on the mat," the computer guesses "sitting." It has to figure out the best word to complete the sentence.

6. Translation

- **Translation is changing words from one language into another.**
- **It helps people who speak different languages understand each other. Imagine you have a sentence in English and you want to turn it into French.**

How It Works?

- Computer is trained on large bilingual datasets. It learns the mapping between words and phrases in one language and their corresponding words in another language. These models can generate accurate translations by understanding sentence structure and context in both languages. This is useful for applications like Google Translate.
- Some of its examples are:

  English to French Translation: Turning "How are you?" into "Comment ça va?"

  Multilingual Translation: Translating content between multiple languages, which is important for global applications and communication.

CONT'D

7. Summarization

- Summarization is making a long piece of text shorter while keeping the important parts.
- It’s like making a quick summary of a long story.
- Imagine you have a long article and you want to concise it into a few key sentences.

How It Works

- Summarization models use techniques like extractive or abstractive summarization.
- In extractive summarization, important sentences are selected directly from the text and in abstractive summarization, the model generates new sentences that convey the main ideas of the text in a concise form.
- This is useful for quickly understanding large documents or articles.

- Some of its examples are:
  News Summarization: Turning a long news article into a few sentences that capture the key points.
  Document Summarization: Creating an abstract or executive summary of a lengthy research paper or report.
- Each of these tasks helps computers understand and process human language in different ways.
- These tasks are fundamental to many applications in natural language processing and help make computers better at understanding and generating human language.

LIMITATIONS OF TRADITIONAL MODELS

1. Bag of Words (BoW)

- **Ignores Word Order:** Sequences like "Dog bites man" and "Man bites dog" are treated the same.
- **Context Loss:** Cannot differentiate between polysemous words (e.g., "bank" in different contexts).
- **Sparse and High-Dimensional:** Creates large vectors (100,000+ dimensions) with mostly zeros.
- **No Semantic Meaning:** Treats synonyms and similar words as completely different.
- **OOV(Out-Of-Vocabulary) Problem:** New words outside the vocabulary are ignored.

2. Recurrent Neural Networks (RNNs)

- **Vanishing and Exploding Gradients:** Struggles with learning long-term dependencies.
- **Short-Term Memory:** Cannot capture relationships between distant words.
- **Sequential Processing:** Slower training due to lack of parallelism.
- **Difficulty with Long Dependencies:** Poor performance on complex, nested sentences.

3. Long Short-Term Memory (LSTM)

- **Improvements over RNNs:** Better at capturing longer sequences using gates.
- **Limitations**:

  Still Sequential: Cannot leverage parallel processing like Transformers.
  Memory Bottleneck: Struggles with sequences longer than 512-1024 tokens.
  High Training Overhead: Computationally expensive due to gating mechanisms.
  Poor at Document-Level Understanding: Underperforms on large text tasks (summarization, QA).
  Struggles with Hierarchical Structures: Hard to model syntax trees and deep grammar relations.

Comparison Table

Feature	Bag of Words	RNN	LSTM	Transformers
Word order captured?	❌ No	✅ Yes	✅ Yes	✅ Yes
Context length	❌ None	❌ Short	⚠️ Medium	✅ Long
Parallelizable?	✅ Yes	❌ No	❌ No	✅ Yes
Sparse vectors?	❌ Yes	✅ Dense	✅ Dense	✅ Dense
Handles semantics?	❌ No	⚠️ Limited	⚠️ Limited	✅ Yes
OOV robustness	❌ Poor	✅ Better	✅ Better	✅ Best
Long document	❌ No	❌ Weak	⚠️	✅ Strong

RISE OF TRANSFORMER MODELS

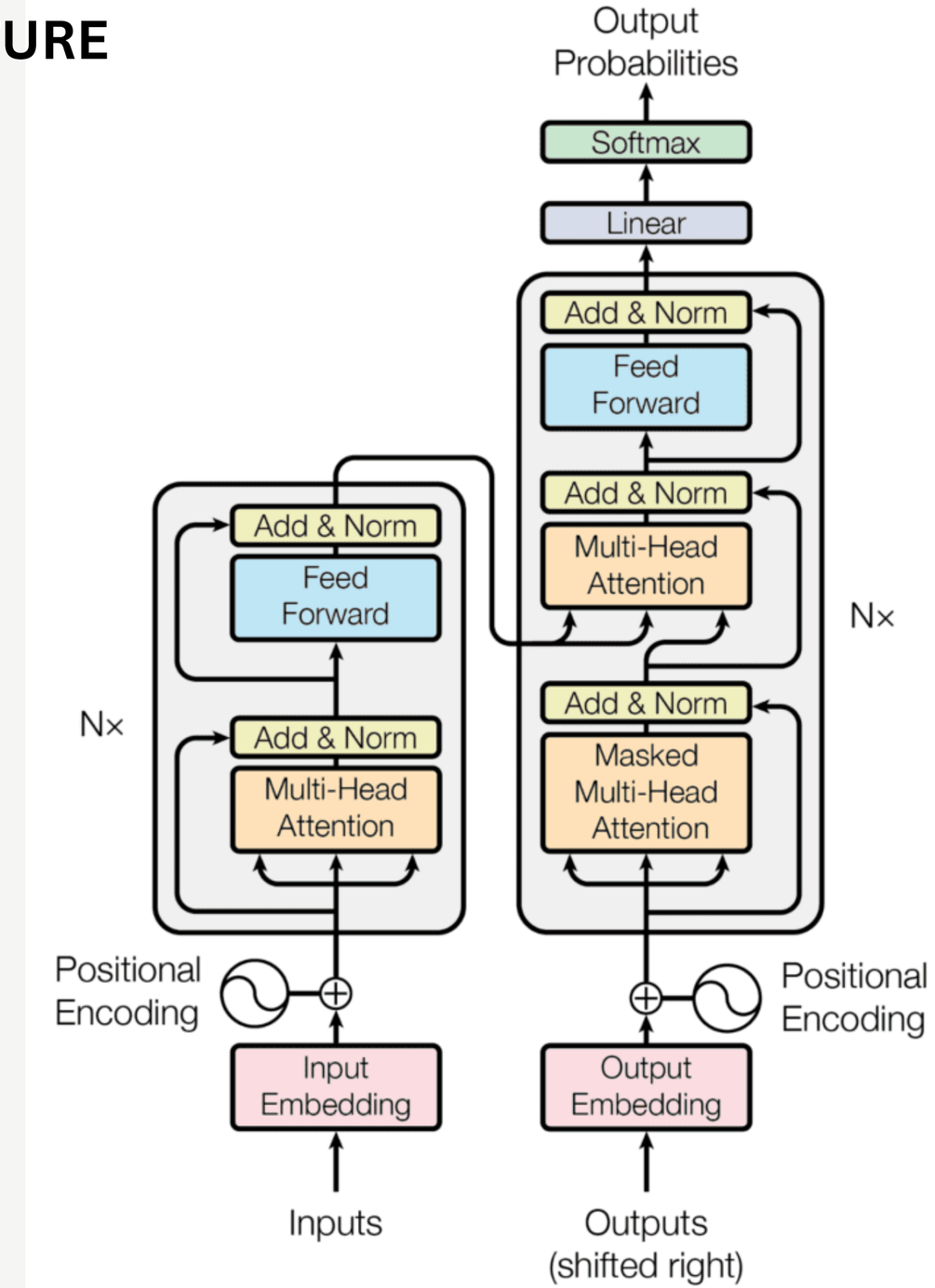
1. Pre-Transformer Era

- Before Transformers, most NLP models were based on:
  Recurrent Neural Networks (RNNs)
  Long Short-Term Memory (LSTM)
  Gated Recurrent Units (GRUs)
- These models processed input sequentially, making them: Slow to train , and Poor at capturing long-range dependencies.

2. Why Transformers Took Over Traditional Models

- Transformer Architecture is a model that uses self-attention to transform one whole sentence into a single sentence.
- This is useful where older models work step by step and it helps overcome the challenges seen in models like **RNNs** and **LSTMs**.
- Traditional models like **RNNs (Recurrent Neural Networks)** suffer from the vanishing gradient problem which leads to long-term memory loss.
- **RNNs** process text sequentially meaning they analyze words one at a time.
- **For Example:** in the sentence: "**XYZ went to France in 2019 when there were no cases of COVID and there he met the president of that country**" the word "**that country**" refers to "**France**".
- However **RNN** would struggle to link "**that country**" to "**France**" since it processes each word in sequence leading to losing context over long sentences. This limitation prevents **RNNs** from understanding the full meaning of the sentence.
- While adding more memory cells in **LSTMs (Long Short-Term Memory networks)** helped address the **vanishing gradien**t issue they still process words one by one. This sequential processing means LSTMs can't analyze an entire sentence at once.
- T**raditional models** struggle with this context dependence , whereas, **Transformer model** through its **self-attention mechanism**, processes the entire sentence in **parallel addressing** these issues and making it significantly more effective at understanding context.

3. TRANSFORMER ARCHITECTURE

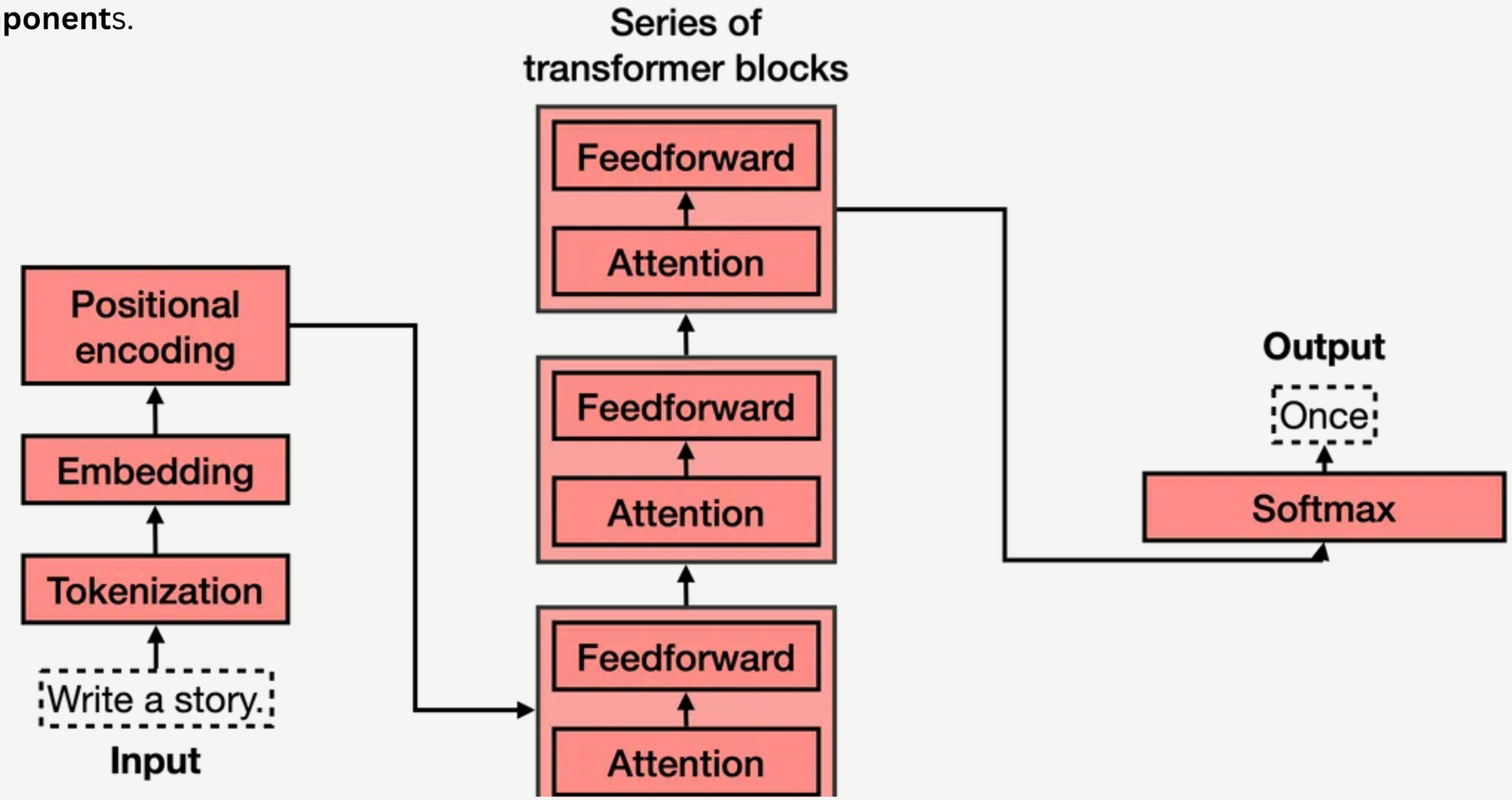


TRANSFORMER ARCHITECTURE

- When we look at **Transformer Architecture,** The transformer has 4 main parts:

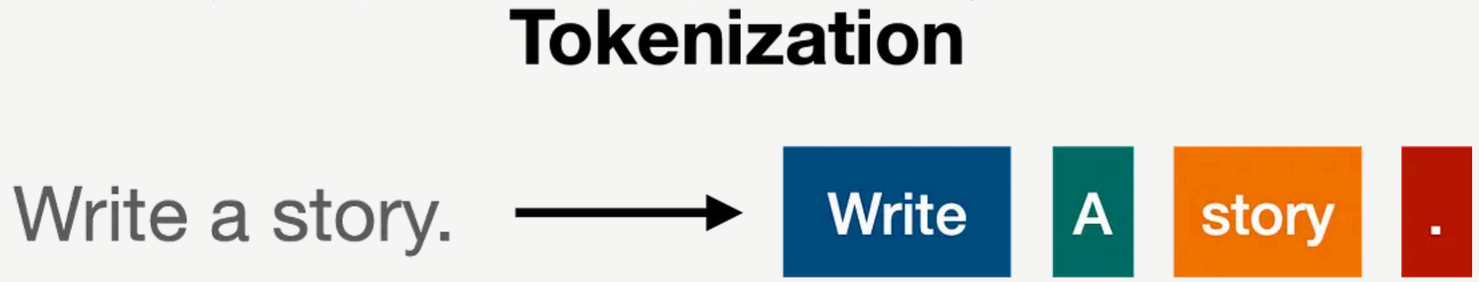
- **Tokenization**
- **Embedding**
- **Positional encoding**
- **Transformer block (several of these)**
- **Output Probabilities**

The fourth one, **the transformer block**, is the most complex of all. Many of these can be concatenated, and each one contains two main parts: The **attention** and the **feedforward component**s.



1. TOKENIZATION

- Tokenization is the most basic step. It consists of a large dataset of tokens, including all the words, punctuation signs, etc.
- The tokenization step takes every word, prefix, suffix, and punctuation signs, and sends them to a known token from the library.

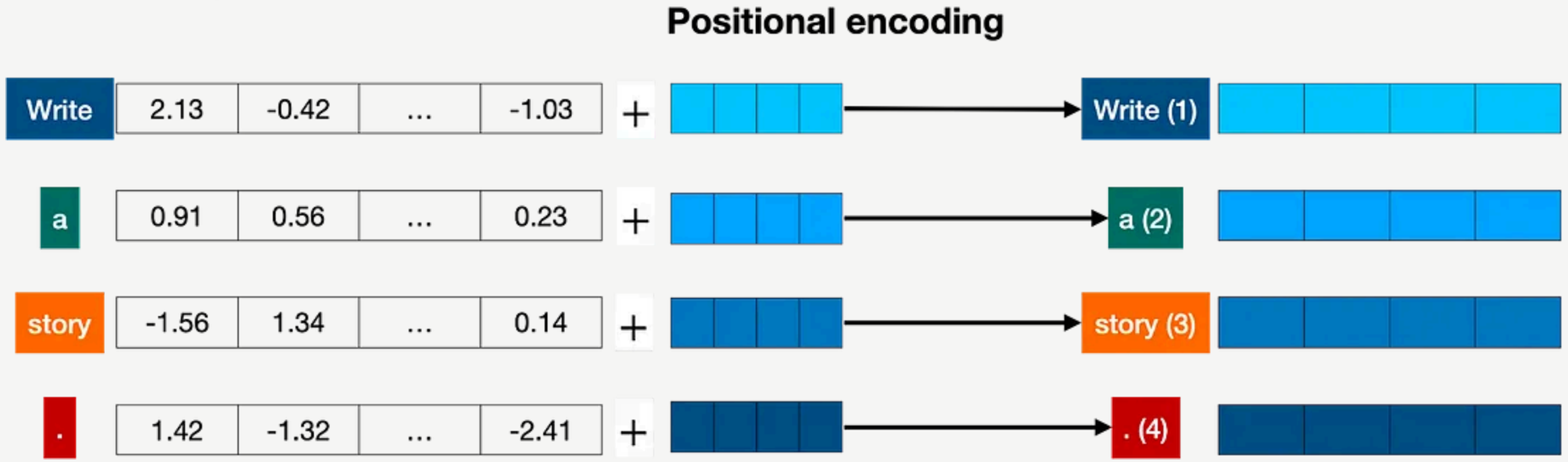


2. EMBEDDING

- Once the input has been tokenized, it’s time to turn words into numbers.
- For this, we use an embedding.

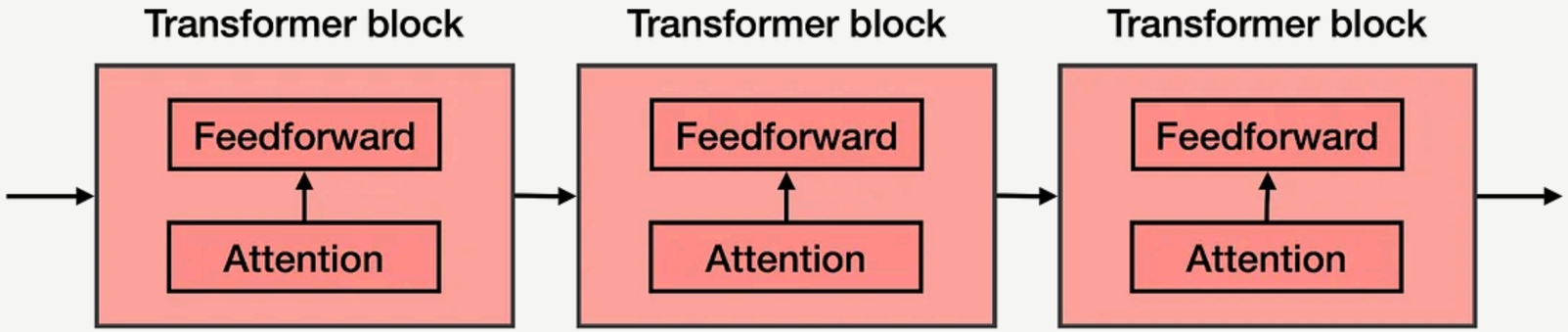
3. POSITIONAL ENCODING

- In natural language processing order of words is very important for understanding its meaning in the tasks like translation and text generation.
- Transformers process all tokens in parallel which speeds up training but they don't naturally capture order of tokens.
- Unlike RNNs, transformers lack an inherent understanding of word order since they process data in parallel.
- To solve this **Positional Encodings** are added to token embeddings providing information about the position of each token within a sequence.
- Positional encoding adds a positional vector to each word, in order to keep track of the positions of the words.



4. TRANSFORMER BLOCK

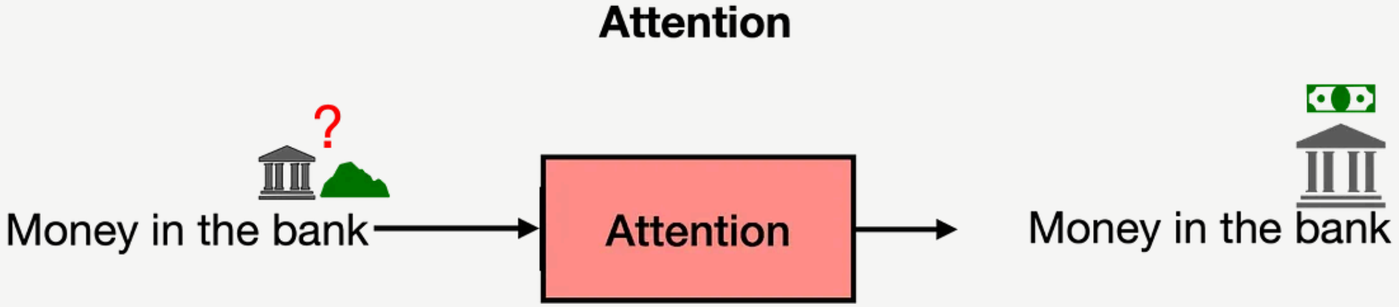
- The words come in and get turned into tokens (tokenization), tokenized words are turned into numbers (embeddings), then order gets taken into account (positional encoding).
- This gives us a vector for every token that we input to the model.
- Now, the next step is to predict the next word in this sentence.
- This is done with a really really large neural network, which is trained precisely with that goal, to predict the next word in a sentence.
- We can train such a large network, but we can vastly improve it by adding a key step: the attention component.
- The attention component is added at every block of the feedforward network.
- Therefore, if you imagine a large feedforward neural network whose goal is to predict the next word, formed by several blocks of smaller neural networks, an attention component is added to each one of these blocks.
- Each component of the transformer, called a transformer block, is then formed by two main components:
- The **attention** component.
- The **feedforward** component.



- The transformer is a concatenation of many transformer blocks.
- Each one of these is composed by an attention component followed by a feedforward component (a neural network).

1. Attention Mechanism

- **Transformer model** is a type of neural network architecture designed to handle sequential data primarily for tasks such as language translation, text generation and many more.
- Unlike traditional **Recurrent Neural Networks** (**RNNs**) or **Convolutional Neural Networks** (**CNNs**), **Transformers** uses A**ttention Mechanism** to capture relationships between all words in a sentence regardless of their distance from each other.
- The **Attention Mechanism** is a technique that allows models to focus on specific parts of the input sequence when producing each element of the output sequence.
- It assigns different weights to different input elements enabling the model to prioritize certain information over others.
- This is particularly useful in tasks like language translation where the meaning of a word often depends on its context.



- The attention mechanism allows transformers to determine which words in a sentence are most relevant to each other. This is done using a scaled dot-product attention approach:

1. Each word in a sequence is mapped to three vectors:
   - **Query (Q)** is the search text you type in the search engine bar.
   - **Key (K)** is represents the possible tokens the query can attend to.
   - **Value (V)** is the actual content of web pages shown. Once we matched the appropriate search term (**Query**) with the relevant results (**Key**), we want to get the content (**Value**) of the most relevant pages.
2. **Attention scores** are computed as: Attention(Q,K,V)= $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
3. **Attention scores** determine how much attention each word should pay to others.

Types of Attention Mechanisms in Transformers

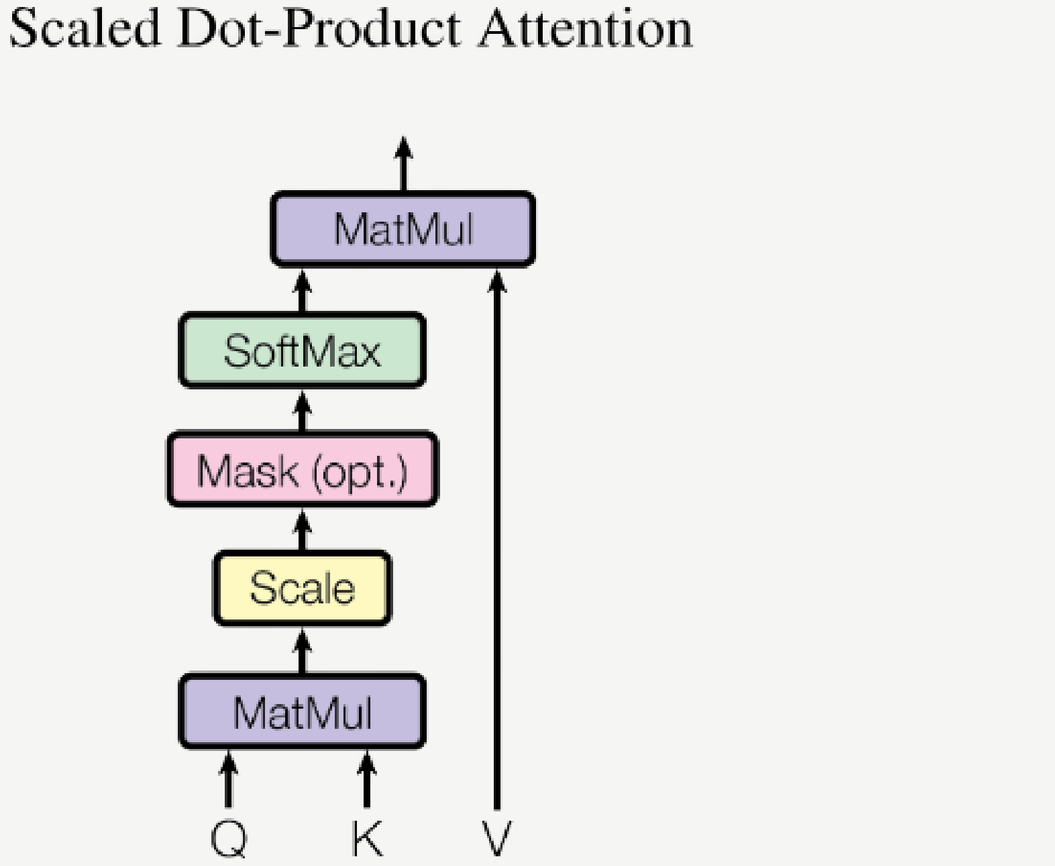
1. Scaled Dot-Product Attention

- The **Scaled Dot-Product Attention** is the fundamental building block of the Transformer's attention mechanism.
- It involves three main components: **queries (Q)**, **keys (K),** and **values (V**).
- The **attention score** is computed as the **dot product** of the **query** and **key vectors**, scaled by the **square root** of the **dimension** of the **key vectors**.
- This score is then passed through a **softmax function** to obtain the attention weights, which are used to compute a **weighted** sum of the **value vectors**.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

where d_k is the dimension of the key vectors.

```
[3]: def scaled_dot_product(q, k, v, mask=None):
         d_k = q.size()[-1]
         attn_logits = torch.matmul(q, k.transpose(-2, -1))
         attn_logits = attn_logits / math.sqrt(d_k)
         if mask is not None:
             attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
         attention = F.softmax(attn_logits, dim=-1)
         values = torch.matmul(attention, v)
         return values, attention
```

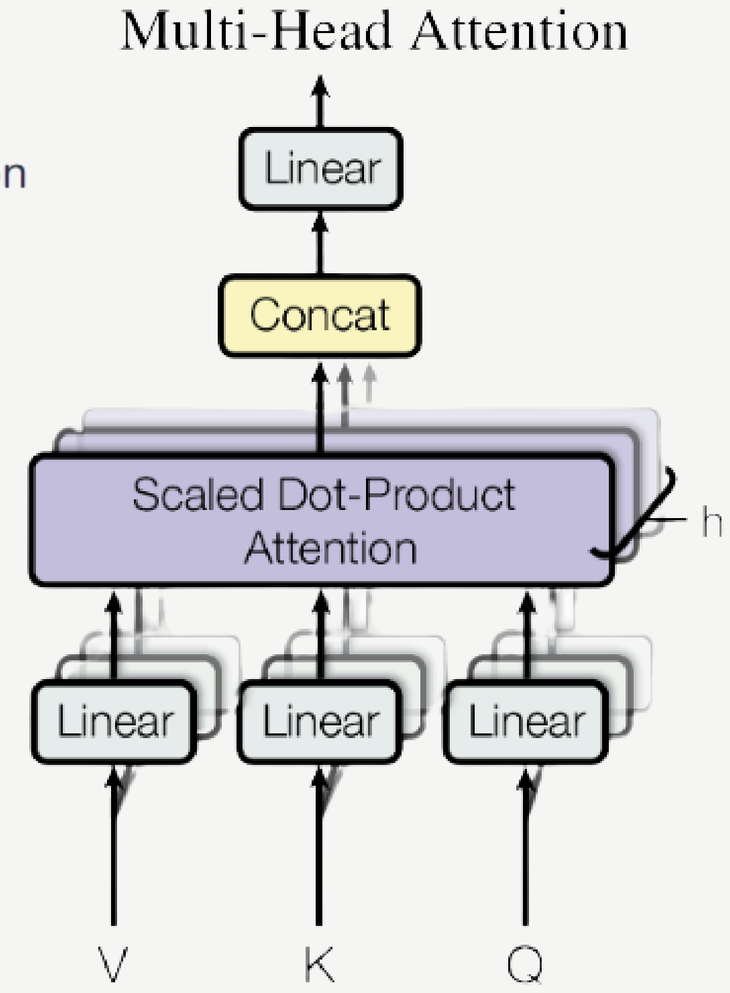


2. Multi-Head Attention

- **Multi-Head Attention** enhances the model's ability to focus on different parts of the input sequence simultaneously.
- It involves **multiple attention heads,** each with its own set of **query**, **key**, and **value** matrices.
- The outputs of these **heads** are concatenated and linearly transformed to produce the final output.
- This allows the model to capture different **features** and **dependencies** in the input sequence.

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \ldots, head_h)W^O$$

where each $Attention(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is the output projection matrix.



3. Self-Attention

- Self-Attention is also known as intra-attention, allows the model to consider different positions of the same sequence when computing the representation of a word.
- In the context of the Transformer, self-attention is applied in both the encoder and decoder layers.
- It enables the model to capture long-range dependencies and relationships within the input sequence.

4. Encoder-Decoder Attention

- Encoder-Decoder Attention also known as cross-attention, is used in the decoder layers of the Transformer.
- It allows the decoder to focus on relevant parts of the input sequence (encoded by the encoder) when generating each word of the output sequence.
- This type of attention ensures that the decoder has access to the entire input sequence, helping it produce more accurate and contextually appropriate translations.

5. Causal or Masked Self-Attention

- Causal or Masked Self-Attention is used in the decoder to ensure that the prediction for a given position only depends on the known outputs at positions before it.
- This is crucial for tasks like language modeling where future tokens should not be visible during training.
- The attention scores for future tokens are masked out, ensuring that the model cannot look ahead.

$$MaskedAttention(Q, K, V) = softmax(\frac{QK^{\iota} + M}{\sqrt{d_k}})V$$

where M is the mask matrix with $-\infty$ in positions that should be masked.

2. Position-wise Feed-Forward Networks

- In addition to **attention sub-layers**, each of the layers in our **encoder** and **decoder** contains a fully connected **feed-forward network**, which is applied to each position separately and identically.
- The **Position-Wise Feed-Forward Network (FFN)** consists of two fully connected **linear** or **dense layers**, or a **multi-layer perceptron (MLP**).
- The **hidden layer**, which is known as **d_ffn**, is generally set to a value about four times that of **d_model**.
- This is why it is sometimes known as an **expand-and-contract network**.
- In other words, the FNN has a size of ***(d_model, d_ffn)*** in its first layer, which means it has to be broadcast across each sequence during tensor multiplication.
- This means each sequence is multiplied by the same weights. If identical sequences are input, the outputs will also be identical.
- This same logic applies to the second dense layer of size ***(d_ffn, d_model)***, which returns the **tensor** to its original size.
- The **ReLU activiation function**, ***max(0, X)***, is used between the layers.
- Any values greater than 0 remain the same, and any values less than or equal to 0 become 0.
- It introduces non-linearity and helps prevent vanishing gradients.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- This transformation helps refine the encoded representation at each position.

ADVANCED ARCHITECTURAL FEATURES

- There are several advanced architectural features that enhance the performance of Transformer models.
- **Layer Normalization**, **Dropout**, and **Residual Connections** are crucial components in Transformer models, particularly during the training phase.

1. Layer Normalization

- Layer Normalization helps to stabilize the training process and improves convergence.
- It works by normalizing the inputs across the features, ensuring that the mean and variance of the activations are consistent.
- This normalization helps mitigate issues related to internal covariate shift, allowing the model to learn more effectively and reducing the sensitivity to the initial weights.
- Layer Normalization is applied twice in each Transformer block, once before the self-attention mechanism and once before the MLP layer.

2. Residual Dropout

- Dropout is a regularization technique used to prevent overfitting in neural networks by randomly setting a fraction of model weights to zero during training.
- This encourages the model to learn more robust features and reduces dependency on specific neurons, helping the network generalize better to new, unseen data.
- During model inference, dropout is deactivated.
- This essentially means that we are using an ensemble of the trained subnetworks, which leads to a better model performance.

3. Residual Connections

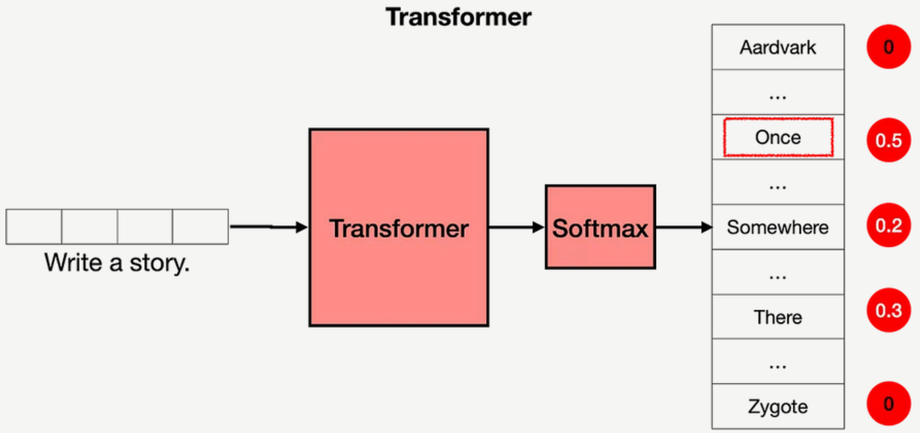
- Residual connections were first introduced in the ResNet model in 2015. This architectural innovation revolutionized deep learning by enabling the training of very deep neural networks.
- Essentially, residual connections are shortcuts that bypass one or more layers, adding the input of a layer to its output.
- This helps mitigate the vanishing gradient problem, making it easier to train deep networks with multiple Transformer blocks stacked on top of each other.
- In GPT-2, residual connections are used twice within each Transformer block: once before the MLP and once after, ensuring that gradients flow more easily, and earlier layers receive sufficient updates during backpropagation.

5. OUTPUT PROBABILITIES

- After the input has been processed through all Transformer blocks, the output is passed through the **final linear layer** to prepare it for token prediction.
- This layer projects the final representations into a 50,257 dimensional space, where every token in the vocabulary has a corresponding value called **logit**.
- Any token can be the next word, so this process allows us to simply rank these tokens by their likelihood of being that next word.
- We then apply the **softmax functio**n to convert the **logits** into a **probability distribution** that sums to one.
- This will allow us to sample the next token based on its likelihood.
- **Logits** are the raw, unnormalized outputs of a **neural network** before applying **softmax**.
- They represent the **confidence scores** (**positive**, **negative**, or **zero**) for each class (e.g., each word in the vocabulary), but they aren't probabilities yet.

The Softmax Layer

- The transformer outputs scores for all the words, where the highest scores are given to the words that are most likely to be next in the sentence.
- The last step of a transformer is a softmax layer, which turns these scores into probabilities (that add to 1), where the highest scores correspond to the highest probabilities.



Why is Softmax Important?

- Softmax is essential in Transformers for two reasons:

1. Meaningful Scores

- The raw scores from the attention mechanism wouldn't be very informative on their own.
- Softmax transforms them into probabilities, making it clear how much attention the model should pay to each word.

2. Decision Making

- During generation tasks like machine translation or text summarization, the Transformer needs to pick the next word in the sequence.
- By having probabilities for each word, the model can make informed decisions about which word is most likely to come next.

Benefits of Softmax for Large Language Models(LLMs)

- **Clear Importance:** Raw scores don't tell the whole story. Softmax provides a clear picture of how much attention each word deserves.
- **Better Predictions:** By understanding word importance through probabilities, the LLM can make more accurate predictions during tasks like text generation and translation.
- **Foundation for Learning:** Softmax output serves as the basis for the LLM to learn and improve its understanding of language relationships.
- In essence, softmax acts as a translator, converting the Transformer's internal calculations into a probability distribution that empowers the LLM to make informed decisions and generate human-like text

4. BERT MODEL

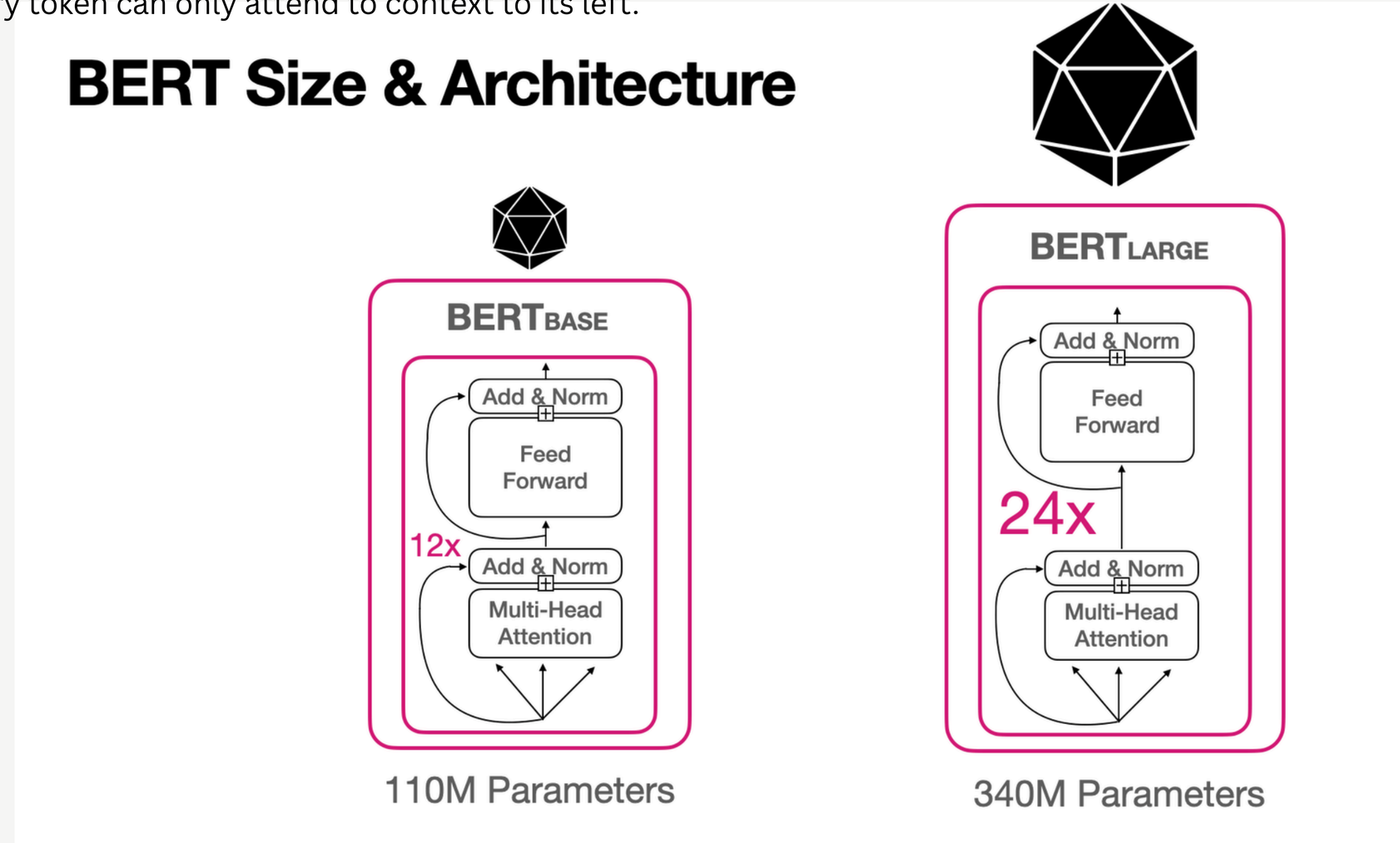
- **BERT's** model architecture is based on **Transformers**. It uses multilayer bidirectional transformer encoders for language representations. Based on the depth of the model architecture, two types of BERT models are introduced namely $BERT_{Base}$ and $BERT_{Large}$. The $BERT_{Base}$ model uses 12 layers of transformers block with a hidden size of 768 and number of self-attention heads as 12 and has around 110M trainable parameters. On the other hand, $BERT_{Large}$ uses 24 layers of transformers block with a hidden size of 1024 and number of self-attention heads as 16 and has around 340M trainable parameters. BERT uses the same model architecture for all the tasks be it NLI, classification, or Question-Answering with minimal change such as adding an output layer for classification.

Why is BERT Important?

- Traditional language models process text sequentially, either from left to right or right to left.
- This method limits the model's awareness to the immediate context preceding the target word.
- **BERT** uses a **bi-directional approach** considering both the left and right context of words in a sentence, instead of analyzing the text sequentially, **BERT** looks at all the words in a sentence simultaneously.
- *Example: "The bank is situated on the _____ of the river."*
- *In a unidirectional model, the understanding of the blank would heavily depend on the preceding words, and the model might struggle to recognize or find out whether "bank" refers to a financial institution or the side of the river.*
- ***BERT**, being bidirectional, simultaneously considers both the left **("The bank is situated on the"**) and right context ("**of the river**"), enabling a more nuanced understanding.*
- *It comprehends that the missing word is likely related to the geographical location of the bank, demonstrating the contextual richness that the bidirectional approach brings.*
- **BERT** understands that the context-driven relationship between words plays a pivotal role in deriving meaning.
- It captures the essence of bidirectionality, allowing it to consider the complete context surrounding each word, revolutionizing the accuracy and depth of language understanding.

5. BERT ARCHITECTURE

- BERT's model architecture is a multi-layer bidirectional Transformer encoder .
- In google research paper work, they denote the **number of layers** (i.e., **Transformer blocks**) as ***L***, the **hidden size** as **H**, and the **number of self-attention heads** as **A**.
- They primarily report results on two model sizes: **BERT**$_{BASE}$ (**L**=12, **H**=768, **A**=12, **Total Parameters**=110M) and **BERT**$_{LARGE}$ (**L**=24, **H**=1024, **A**=16, **Total Parameters**=340M). **BERT**$_{BASE}$ was chosen to have the same model size as **OpenAI GPT** for comparison purposes.
- Critically, however, the **BERT Transformer** uses **bidirectional self-attention**, while the **GPT Transformer** uses **constrained self-attention** where every token can only attend to context to its left.



6. PREPROCESSING FOR BERT

- Before BERT can work its magic on text, it needs to be prepared and structured in a way that it can understand.
- The crucial steps of preprocessing text for BERT is **tokenization**, **input formatting**, and the **Masked Language Model (MLM**) objective.

1. Tokenization

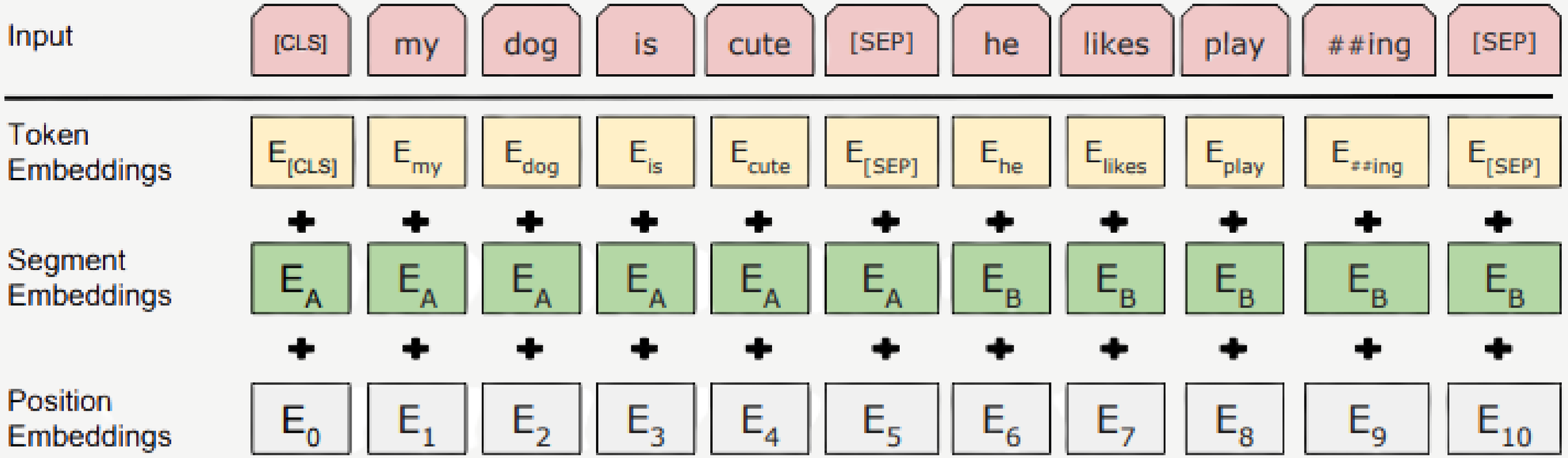
- Imagine we're teaching BERT to read a book. we wouldn't hand in the entire book at once; we'd break it into sentences and paragraphs.
- Similarly, BERT needs text to be broken down into smaller units called tokens.
- BERT uses **WordPiece** embeddings **or** tokenization.
- It splits words into smaller pieces, like turning "running" into "run" and "ning."
- This helps handle tricky words and ensures that BERT doesn't get lost in unfamiliar words.
- Example: **Original Tex**t: "TinyBERT is fascinating." **WordPiece** Tokens:["Tiny", "##B", "##ER", "##T", "is", "fascinating", "."]

2. Input Formatting

- To make BERT handle a variety of down-stream tasks, our input representation is able to unambiguously represent both a single sentence and a pair of sentences (e.g., Question, Answeri) in one token sequence.
- A "**sequence**" refers to the input token sequence to BERT, which may be a single sentence or two sentences packed together.
- BERT use WordPiece embeddings.
- The first token of every sequence is always a special classification token (**[CLS]**).
- The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks.
- Sentence pairs are packed together into a single sequence.
- We differentiate the sentences in two ways. First, we separate them with a special token ([SEP]). Second, we add a learned embedding to every token indicating whether it belongs to sentence **A** or sentence **B**.
- we denote input embedding as **E**, the final hidden vector of the special [**CLS**] token as $\mathbf{C} \in \mathbf{R}^{H}$, and the final hidden vector for the i^{th} input token as $\mathbf{T}_i \in \mathbf{R}^{H}$
- For a given token, its input representation is constructed by summing the corresponding **token**, **segment**, and **position embeddings**.

CONT'D

- The **input embeddings** are the sum of the **token embeddings**, the **segmentation embeddings** and the **position embeddings**.

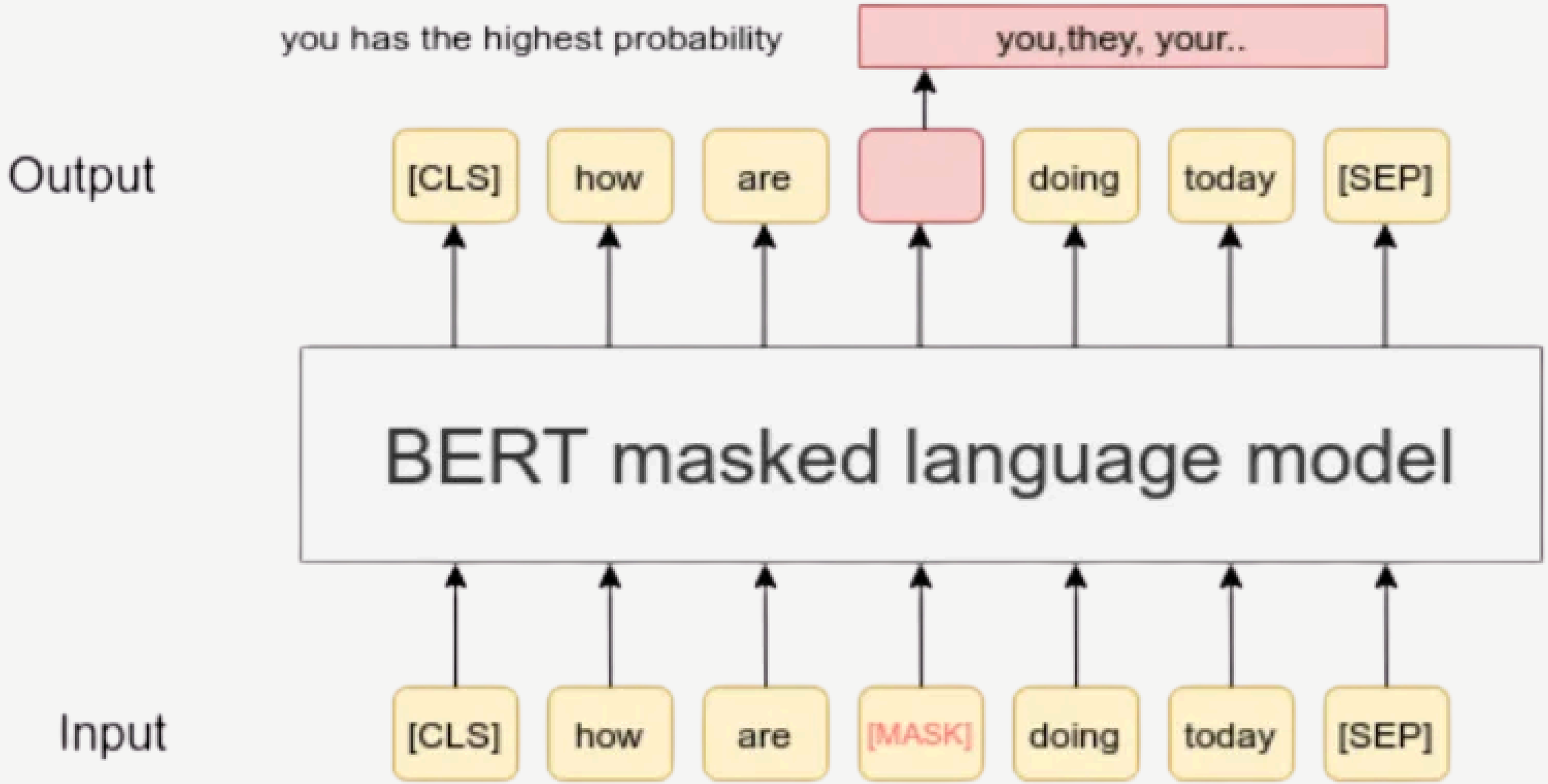


7. PRE-TRAINING TASKS

- Unlike **Recurrent Neural Network(RNN)** and **Long Short-Term Memory networks** (**LSTMs**), We do not use traditional left-to-right or right-to-left language models to pre-train BERT.
- Instead, we pre-train BERT using two **unsupervised tasks: Masked Language Modelling(MLM)** and **Next Sentence Prediction (NSP) or**
- **Upstream Task(Pre-Training):** a BERT is trained on large **unlabeled text** with **self-supervised tasks** like**: Masked Language Modeling (MLM), Next Sentence Prediction (NSP).**

1. Masked Language Modelling (MLM)

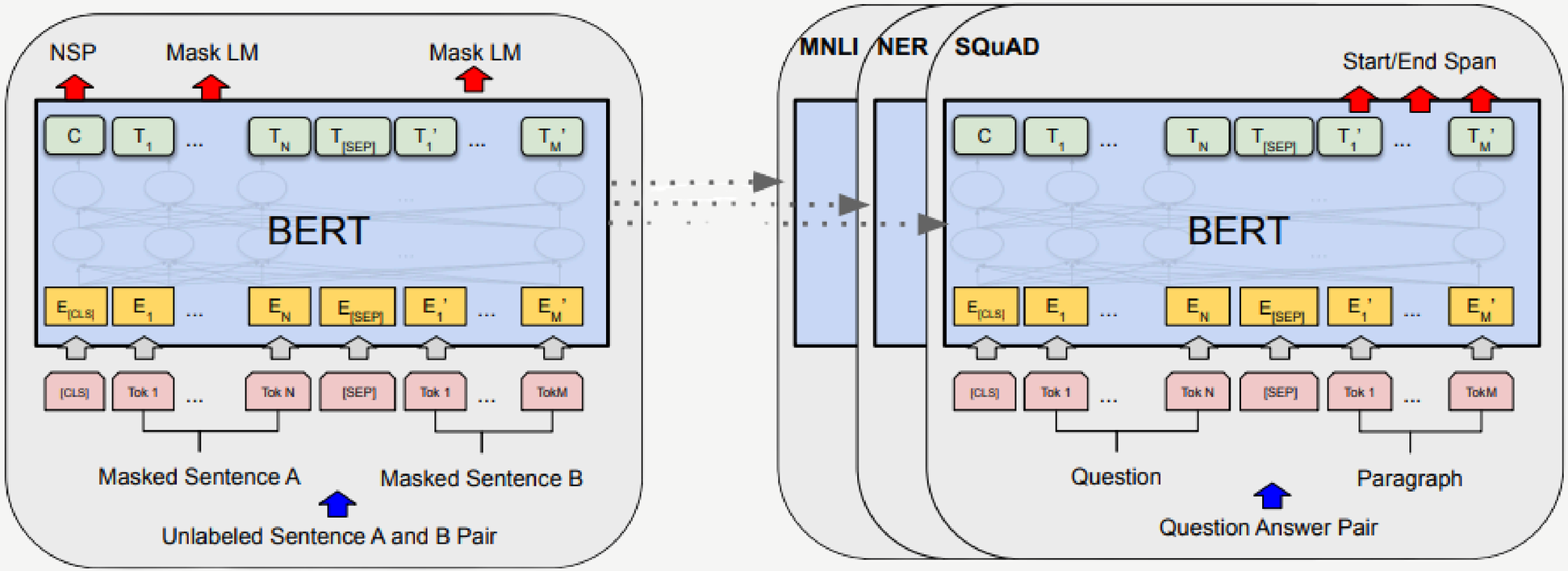
- Intuitively, it is reasonable to believe that a deep bidirectional model is strictly more powerful than either a left-to-right model or the shallow concatenation of a left-toright and a right-to-left model.
- Unfortunately, standard conditional language models can only be trained left-to-right or right-to-left, since bidirectional conditioning would allow each word to indirectly "see itself", and the model could trivially predict the target word in a multi-layered context.
- In order to train a deep bidirectional representation, we simply mask some percentage of the input tokens at random, and then predict those masked tokens.
- We refer to this procedure as a "**masked LM**" (MLM).
- In this case, the final hidden vectors corresponding to the **mask tokens** are fed into an output **softmax** over the vocabulary, as in a standard LM.
- We mask 15% of all **WordPiece** tokens in each sequence at random.
- In contrast to denoising auto-encoders, we only predict the **masked words** rather than reconstructing the entire input.
- Although this allows us to obtain a bidirectional pre-trained model, a downside is that we are creating a mismatch between **pre-training** and **fine-tuning**, since the **[MASK] token** does not appear during **fine-tuning**.
- To mitigate this, we do not always replace "**masked**" words with the **actual [MASK] token**.
- The training data generator chooses 15% of the **token positions** at random for **prediction**.
- If the **i^{th}** token is chosen, we replace the **i^{th}** token with the **[MASK] token** 80% of the time a random token 10% of the time the unchanged **i^{th}** token 10% of the time.
- Then, **T_i** will be used to predict **the original token** with **cross entropy loss**.
- This helps BERT to grasp how words relate to each other, both before and after.
- Example**: Original Sentence: "**The cat is on the mat.**" Masked Sentence: "**The **[MASK]** is on the mat.**"**


you has the highest probability
you,they, your..
Output
[CLS]
how
are
doing
today
[SEP]
BERT masked language model
Input
[CLS]
how
are
[MASK]
doing
today
[SEP]

CONT'D

2. Next Sentence Prediction (NSP)

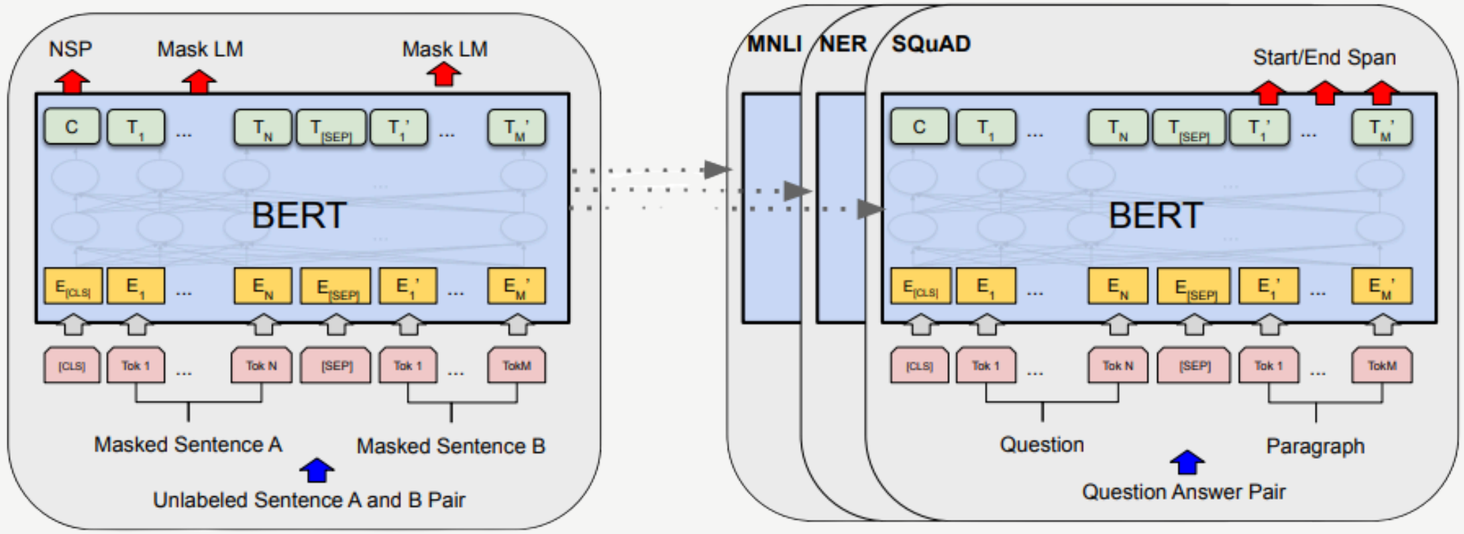
- Many important downstream tasks such as **Question Answering (QA)** and **Natural Language Inference (NLI)** are based on understanding the relationship between two sentences, which is not directly captured by language modeling.
- In order to train a model that understands sentence relationships, we pre-train for a binarized next sentence prediction task that can be trivially or easily generated from any monolingual corpus.
- Specifically, when choosing the sentences **A** and **B** for each pretraining example, **50%** of the tim**e B** is the a**ctual next sentence** that follows **A** (labeled as **IsNext**), and **50%** of the time it is a **random sentence** from the **corpus** (labeled as **NotNext**).
- As we show in the below Figure , **C** is used for **next sentence prediction (NSP)**.



Pre-Training **Fine-Tuning**

8. FINE-TUNING BERT

- **Fine-tuning** is straightforward since the **self-attention mechanism** in the **Transformer** allows BERT to model many **downstream tasks**, whether they involve single text or text pairs, by swapping out the appropriate inputs and outputs.
- A **Downstream Task(Task-Specific Fine-Tuning)** refers to a **specific NLP task** that the model is fine-tuned on after the general-purpose **pre-training phase**.
- For applications involving text pairs, a common pattern is to independently encode text pairs before applying **bidirectional cross attention**.
- BERT instead uses the **self-attention mechanism** to unify these two stages, as encoding a concatenated text pair with **self-attention** effectively includes **bidirectional cross attention** between two sentences.
- For each task, we simply plug in the taskspecific inputs and outputs into BERT and finetune all the parameters end-to-end.
- At the input, **sentence A** and **sentence B** from pre-training are analogous to **sentence pairs** in paraphrasing, **hypothesis-premise pairs** in entailment, **question-passage pairs** in question answering, and a **degenerate text-∅ pair** in text classification or sequence tagging.
- At the output, the token representations are fed into an output layer for token level tasks, such as **sequence tagging** or **question answering**, and the **[CLS]** representation is fed into an output layer for classification, such as **entailment** or **sentiment analysis**.



Pre-Training **Fine-Tuning**

9. FEATURE-BASED APPROACH WITH BERT

- All of the BERT results presented so far have used the fine-tuning approach, where a simple classification layer is added to the pre-trained model, and all parameters are jointly fine-tuned on a downstream task.
- However, the feature-based approach, where fixed features are extracted from the pretrained model, has certain advantages.
- First, not all tasks can be easily represented by a **Transformer encoder architecture**, and therefore require a task-specific model architecture to be added.
- Second, there are major computational benefits to pre-compute an expensive representation of the training data once and then run many experiments with cheaper models on top of this representation.
- To avoid fine-tuning approach, the researchers use the feature-based technique by extracting activations from one or more layers. Before the classification layer, these contextual embeddings are fed into a 768-dimensional BiLSTM with a randomly initialized two-layer structure.
- The results are shown in the below figure. BERTLARGE utilizes cutting-edge technology to provide competitive results. Concatenating the top four hidden layers of the pre-trained Transformer's token representations is the best-performing technique, which is just 0.3 F1 behind fine-tuning the entire model. Regarding fine-tuning and feature-based processes, BERT is an excellent choice.

System	Dev F1	Test F1
ELMo (Peters et al., 2018a)	95.7	92.2
CVT (Clark et al., 2018)	-	92.6
CSE (Akbik et al., 2018)	-	**93.1**
Fine-tuning approach		
$\text{BERT}_{\text{LARGE}}$	96.6	92.8
$\text{BERT}_{\text{BASE}}$	96.4	92.4
Feature-based approach ($\text{BERT}_{\text{BASE}}$)		
Embeddings	91.0	-
Second-to-Last Hidden	95.6	-
Last Hidden	94.9	-
Weighted Sum Last Four Hidden	95.9	-
Concat Last Four Hidden	96.1	-
Weighted Sum All 12 Layers	95.5	-

9. POPULAR BERT VARIANTS

- As the field of Natural Language Processing (NLP) evolves, so does BERT.
- In this and next a few slides , we'll explore recent developments and variants that have taken BERT's capabilities even further, including **RoBERTa**, **ALBERT**, **DistilBERT**, **ELECTRA,** and **TinyBERT.**

1. **RoBERTa (**Robustly Optimized BERT Approach**)**
   - RoBERTa is like BERT's clever sibling. It's trained with a more thorough recipe, involving larger batches, more data, and more training steps.
   - This enhanced training regimen results in even better language understanding and performance across various tasks.
2. **ALBERT(**A Lite BERT**)**
   - Designed by Google Research for parameter efficiency.
   - It's designed to be efficient, using parameter-sharing techniques to reduce memory consumption.
   - Despite its smaller size, ALBERT maintains BERT's power and can be particularly useful when resources are limited.
3. **DistilBERT (**Distilled BERT**)**
   - DistilBERT is a distilled version of BERT. It's trained to mimic BERT's behavior but with fewer parameters.
   - This makes DistilBERT lighter and faster while still retaining a good portion of BERT's performance. I
   - t's a great choice for applications where speed and efficiency matter.
4. **ELECTRA (** Efficiently Learning an Encoder that Classifies Token Replacements Accurately**)**
   - ELECTRA introduces an interesting twist to training. Instead of predicting masked words, ELECTRA trains by detecting whether a replaced word is real or artificially generated.
   - Unlike BERT, which is trained using Masked Language Modeling (MLM), ELECTRA introduces a more efficient pre-training objective called: **Replaced Token Detection (RTD)**
   - This efficient method makes ELECTRA a promising approach for training large models without the full computational cost.
5. **TinyBERT(** Tiny BERT**)**
   - A very compact version of BERT, even smaller than DistilBERT. Uses two-stage knowledge distillation: **General distillation** (on large corpus) , and **Task-specific distillation** (on fine-tuning task)
   - Designed for edge devices or resource-constrained environments.

10. APPLICATIONS OF BERT

1. Text Representation

- BERT is used to generate word embeddings or representation for words in a sentence.

2. Named Entity Recognition (NER))

- BERT can be fine-tuned for named entity recognition tasks, where the goal is to identify entities such as names of people, organizations, locations, etc., in a given text.

3. Text Classification

- BERT is widely used for text classification tasks, including sentiment analysis, spam detection, and topic categorization.
- It has demonstrated excellent performance in understanding and classifying the context of textual data.

4. Question-Answering Systems

- BERT has been applied to question-answering systems, where the model is trained to understand the context of a question and provide relevant answers.
- This is particularly useful for tasks like reading comprehension.

5. Machine Translation

- BERT's contextual embeddings can be leveraged for improving machine translation systems. The model captures the nuances of language that are crucial for accurate translation.

6. Conversational AI

- BERT is employed in building conversational AI systems, such as chatbots, virtual assistants, and dialogue systems. Its ability to grasp context makes it effective for understanding and generating natural language responses.

7. Text Summarization

- BERT can be used for abstractive text summarization, where the model generates concise and meaningful summaries of longer texts by understanding the context and semantics.

8. Semantic Similarity

- BERT embeddings can be used to measure semantic similarity between sentences or documents. This is valuable in tasks like duplicate detection, paraphrase identification, and information retrieval.

11. CONCLUSION

- BERT has proved to be a breakthrough in Natural Language Processing and Language Understanding field similar to that AlexNet has provided in the Computer Vision field.
- It has achieved state-of-the-art results in different task thus can be used for many NLP tasks.
- It is also used in Google Search in 70 languages as Dec 2019.
- Recent empirical improvements due to **transfer learning** with language models have demonstrated that rich, **unsupervised pre-training** is an integral part of many language understanding systems.
- In particular, these results enable even low-resource tasks to benefit from deep unidirectional architectures.
- Google research group major contribution is further generalizing these findings to deep bidirectional architectures, allowing the same pre-trained model to successfully tackle a broad set of NLP tasks.

Thank you

For your attention

fandisheabdurehman@gmail.com

www.ktun.edu.tr

+90-507-070-1196