AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

BERT

Encoder Only Model

BERT-

Bidirectional Encoder Representations from Transformers is a transformer-based machine learning technique for natural language processing pre-training developed by Google. BERT was created and published in 2018 by Jacob Devlin and his colleagues from Google

Courtesy:, analytics vidya,fast.ai, coursera: AndrewNG, http://jalammar.github.io/

In 2018, Google introduced and open-sourced BERT (11 NLP tasks).

In December 2019, BERT was applied to more than 70 different languages

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin **Ming-Wei Chang** **Kenton Lee** **Kristina Toutanova**
Google AI Language
{jacobdevlin,mingweichang,kentonl,kristout}@google.com

Abstract

We introduce a new language representation model called **BERT**, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answering Test F1 to 93.2 (1.5 point absolute improvement) and SQuAD v2.0 Test F1 to 83.1 (5.1 point absolute improvement).

1 Introduction

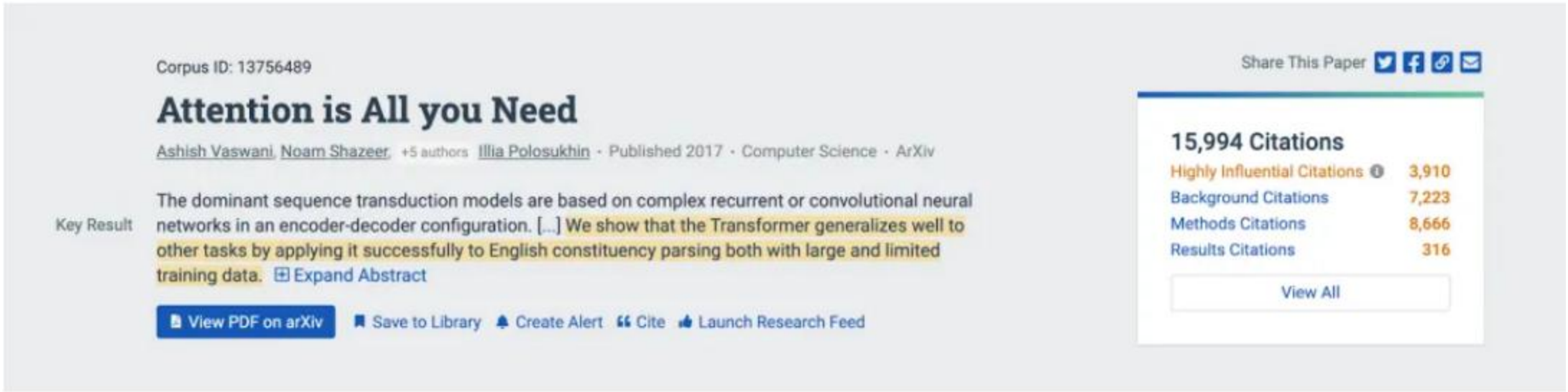
There are two existing strategies for applying pre-trained language representations to downstream tasks: *feature-based* and *fine-tuning*. The feature-based approach, such as ELMo (Peters et al., 2018a), uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT) (Radford et al., 2018), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning *all* pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

We argue that current techniques restrict the power of the pre-trained representations, especially for the fine-tuning approaches. The major limitation is that standard language models are unidirectional, and this limits the choice of architectures that can be used during pre-training. For example, in OpenAI GPT, the authors use a left-to-right architecture, where every token can only attend to previous tokens in the self-attention layers of the Transformer (Vaswani et al., 2017). Such re-

2020 status

As of March 10, 2025, the "Attention is All You Need" paper has been cited over 88,778 times. The "BERT" paper has been cited over 88,000 times.

Now :citations 41978



Corpus ID: 13756489

Attention is All you Need

Ashish Vaswani, Noam Shazeer, +5 authors Illia Polosukhin · Published 2017 · Computer Science · ArXiv

Key Result

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks in an encoder-decoder configuration. [...] We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data. ⊞ Expand Abstract

View PDF on arXiv | Save to Library | Create Alert | Cite | Launch Research Feed

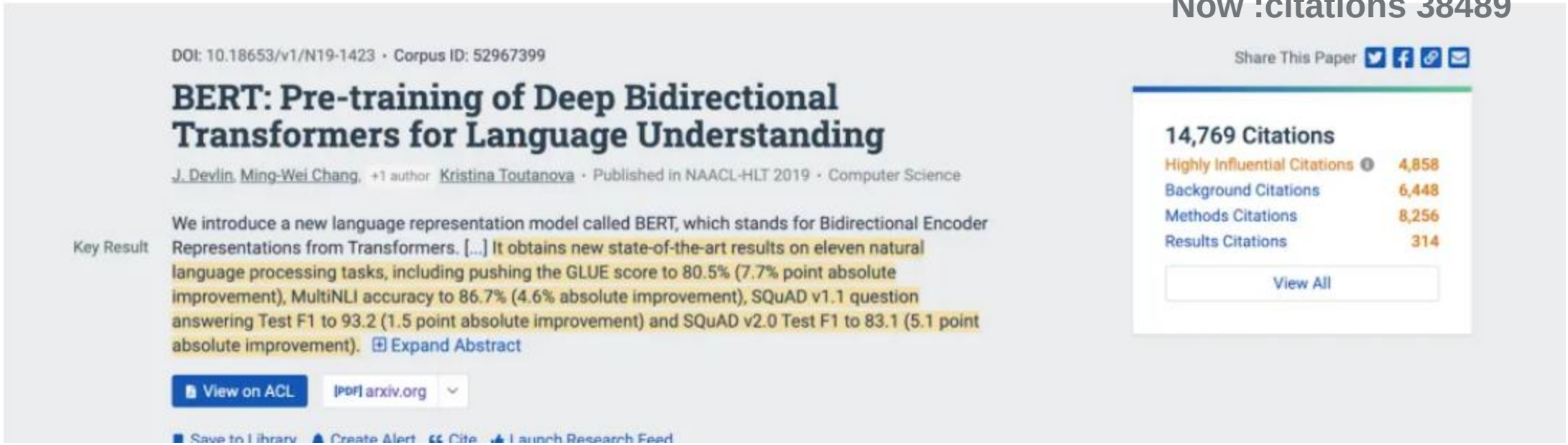
Share This Paper

15,994 Citations

Highly Influential Citations	3,910
Background Citations	7,223
Methods Citations	8,666
Results Citations	316

View All

Now :citations 38489



DOI: 10.18653/v1/N19-1423 · Corpus ID: 52967399

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

J. Devlin, Ming-Wei Chang, +1 author Kristina Toutanova · Published in NAACL-HLT 2019 · Computer Science

Key Result

We introduce a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. [...] It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answering Test F1 to 93.2 (1.5 point absolute improvement) and SQuAD v2.0 Test F1 to 83.1 (5.1 point absolute improvement). ⊞ Expand Abstract

View on ACL | [PDF] arxiv.org

Save to Library | Create Alert | Cite | Launch Research Feed

Share This Paper

14,769 Citations

Highly Influential Citations	4,858
Background Citations	6,448
Methods Citations	8,256
Results Citations	314

View All

NLP's ImageNet moment

BERT basically uses the concept of pre-training the model on a very large dataset in an unsupervised manner for language modeling. A pre-trained model on a very large dataset has the capability to better understand the context of the input sentence. After pre-training, the model can be fine-tuned on the task-specific supervised dataset to achieve good results.

BERT is pre-trained from unlabeled data extracted from BooksCorpus (800M words) **and English Wikipedia** (2,500M words)

- Sequence-to-sequence based language generation tasks such as:
  - Question answering
  - Abstract summarization
  - Sentence prediction
  - Conversational response generation
- Natural language understanding tasks such as:
  - Polysemy and Coreference (words that sound or look the same but have different meanings) resolution
  - Word sense disambiguation
  - Natural language inference
  - Sentiment classification

One Hot Encoding, TF-IDF is a statistical measure used to determine the mathematical significance of words in documents. Term frequency and Inverse Document Frequency

- .

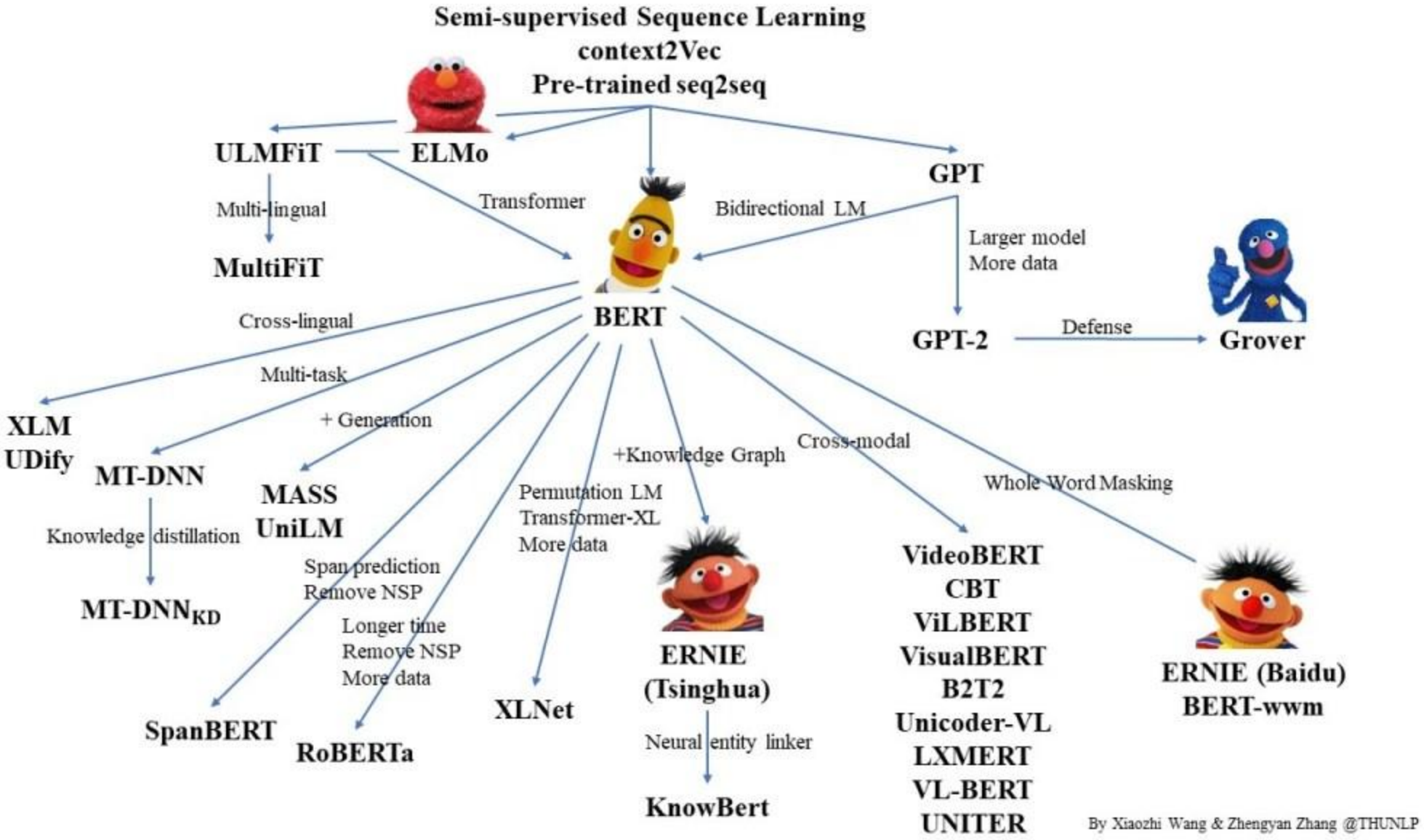
Word2Vec and **Glove** Word-embeddings showed that we can use a vector (a list of numbers) to properly represent words in a way that captures *semantic* or meaning-related relationships : Skip Gram and Common Bag Of Words (CBOW)

ELMo: Context Matters: ELMo looks at the entire sentence before assigning each word in it an embedding. It uses a bi-directional LSTM trained on a specific task to be able to create those embeddings

Universal Language Model FIne-Tuning(ULMFIT) is a transfer learning technique which can help in various NLP tasks : involves a 3-layer AWD-LSTM- Weight-Dropped LSTM

OpenAI Transformer: Pre-training a Transformer Decoder for Language Modeling

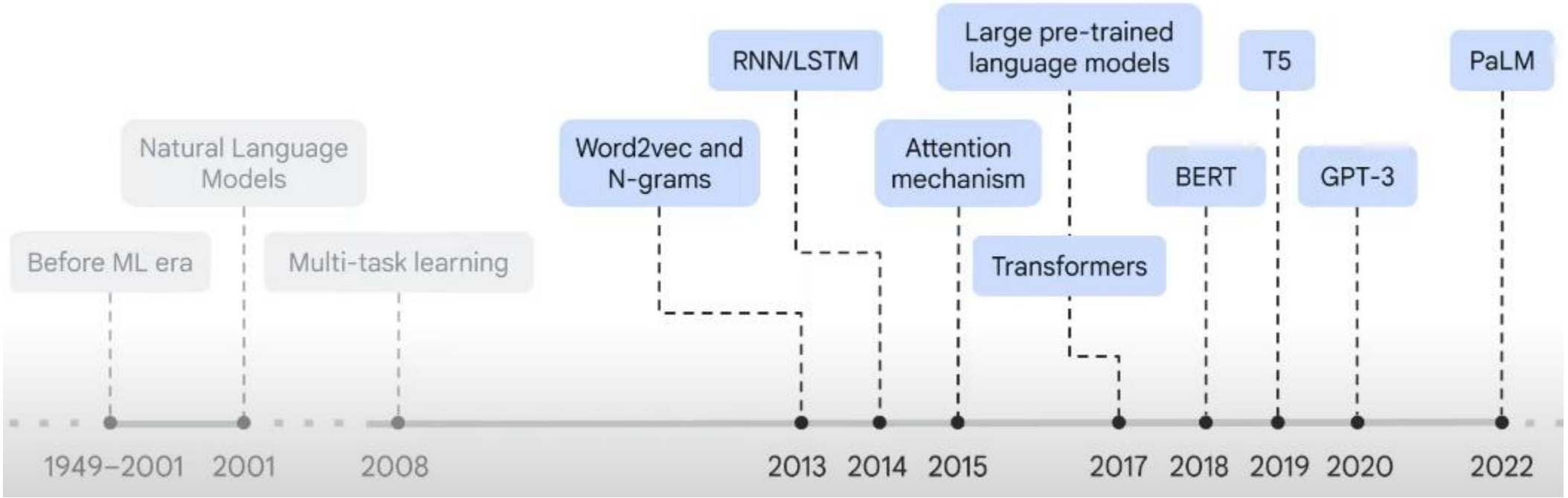
BERT: The openAI transformer gave us a fine-tunable pre-trained model based on the Transformer. But something went missing in this transition from LSTMs to Transformers. ELMo's language model was bi-directional Could we build a transformer-based model whose language model looks both forward and backwards - BERT


Semi-supervised Sequence Learning
context2Vec
Pre-trained seq2seq
ELMo
ULMFiT
Multi-lingual
MultiFiT
Transformer
BERT
Bidirectional LM
GPT
Larger model
More data
GPT-2
Defense
Grover
Cross-lingual
XLM
UDify
Multi-task
MT-DNN
Knowledge distillation
MT-DNN$_{KD}$
+ Generation
MASS
UniLM
Span prediction
Remove NSP
SpanBERT
Longer time
Remove NSP
More data
RoBERTa
Permutation LM
Transformer-XL
More data
XLNet
+Knowledge Graph
ERNIE
(Tsinghua)
Neural entity linker
KnowBert
Cross-modal
VideoBERT
CBT
ViLBERT
VisualBERT
B2T2
Unicoder-VL
LXMERT
VL-BERT
UNITER
Whole Word Masking
ERNIE (Baidu)
BERT-wwm
By Xiaozhi Wang & Zhengyan Zhang @THUNLP

Research groups and separate factions of Google are fine-tuning the BERT model architecture with supervised training to either optimize it for efficiency (modifying the learning rate, for example) or specialize it for certain tasks by pre-training it with certain contextual representations. Some examples include:

- **patentBERT** - a BERT model fine-tuned to perform patent classification.
- **docBERT** - a BERT model fine-tuned for document classification.
- **bioBERT** - a pre-trained biomedical language representation model for biomedical text mining.
- **VideoBERT** - a joint visual-linguistic model for process unsupervised learning of an abundance of unlabeled data on Youtube.
- **SciBERT** - a pretrained BERT model for scientific text
- **G-BERT** - a BERT model pretrained using medical codes with hierarchical representations using graph neural networks (GNN) and then fine-tuned for making medical recommendations.
- **TinyBERT** by Huawei - a smaller, "student" BERT that learns from the original "teacher" BERT, performing transformer distillation to improve efficiency. TinyBERT produced promising results in comparison to BERT-base while being 7.5 times smaller and 9.4 times faster at inference.
- **DistilBERT** by HuggingFace - a supposedly smaller, faster, cheaper version of BERT that is trained from BERT, and then certain architectural aspects are removed for the sake of efficiency.

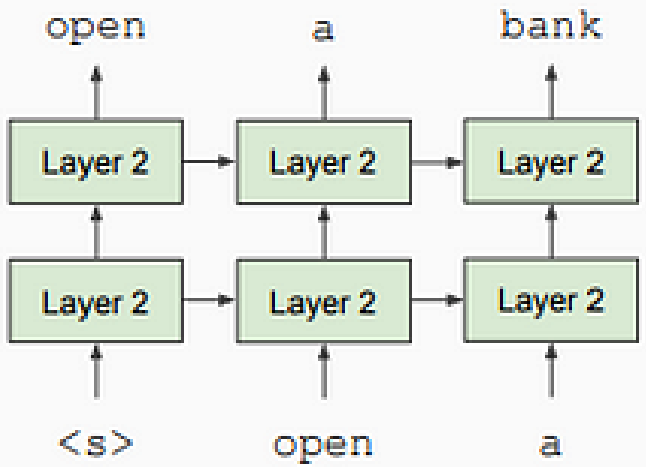
Language Modelling History



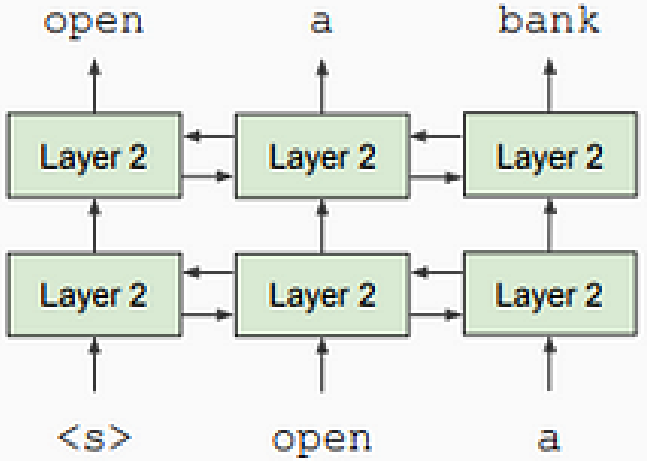
BERT

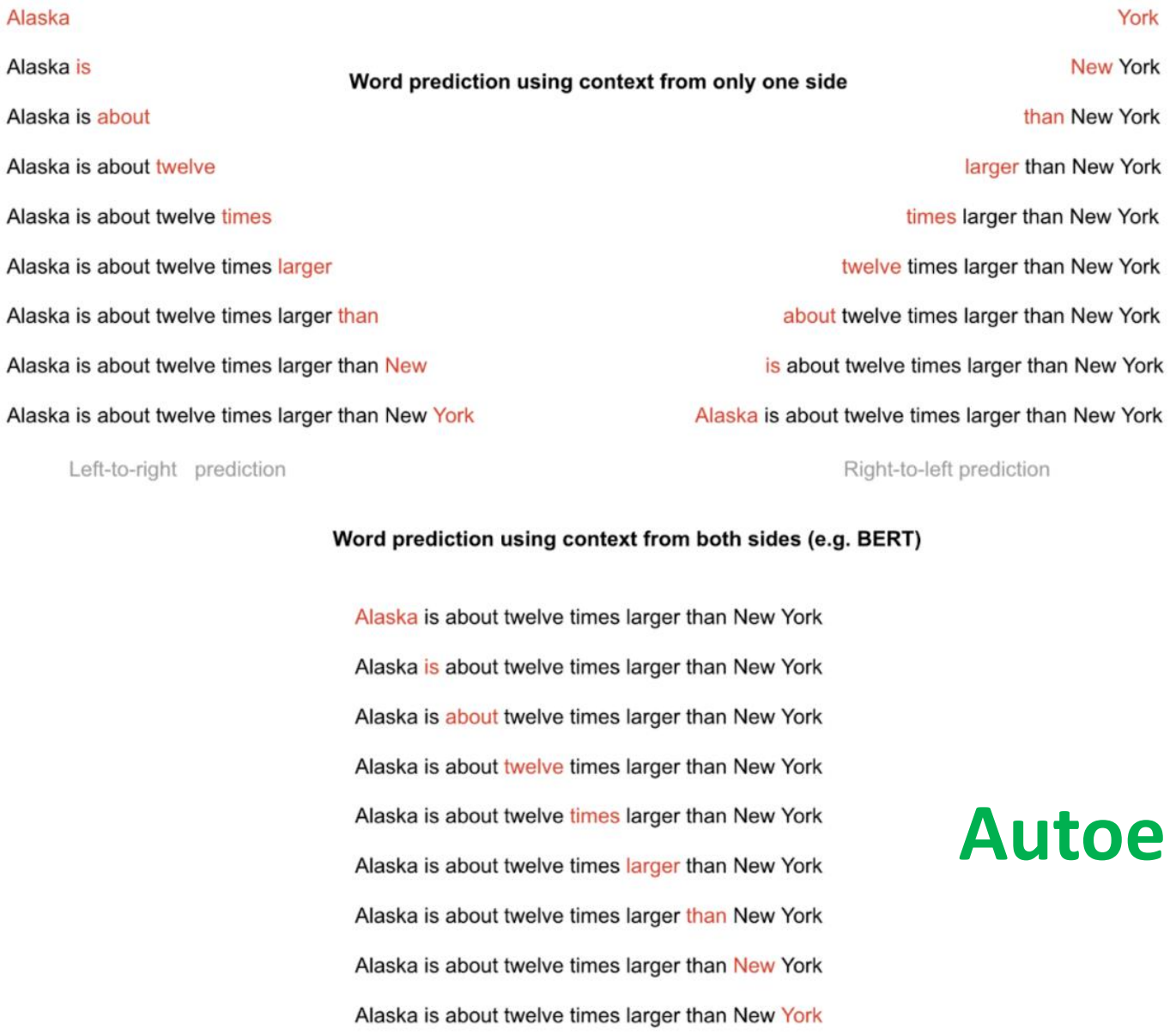
- Bidirectional Encoder Representation from Transformer

Unidirectional context
Build representation incrementally



Bidirectional context
Words can “see themselves”



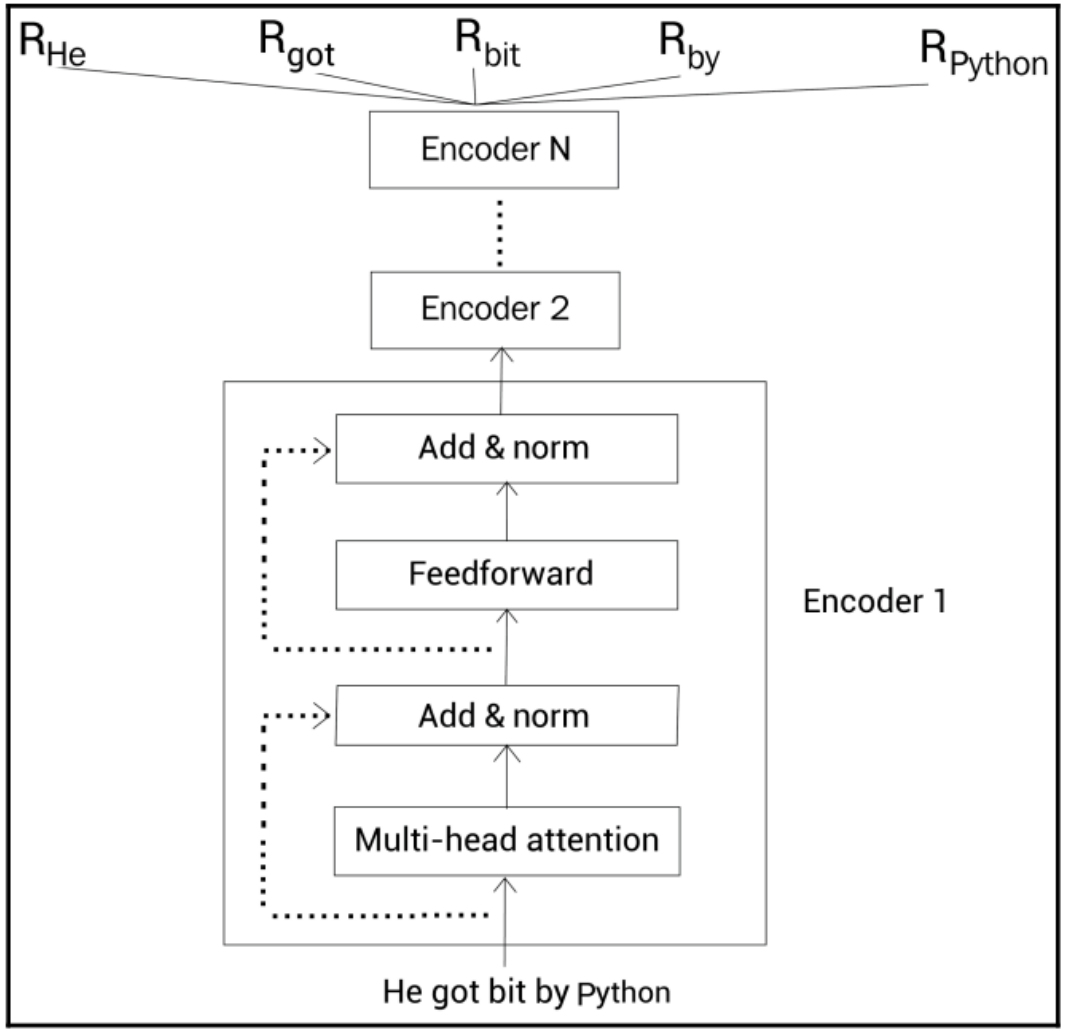


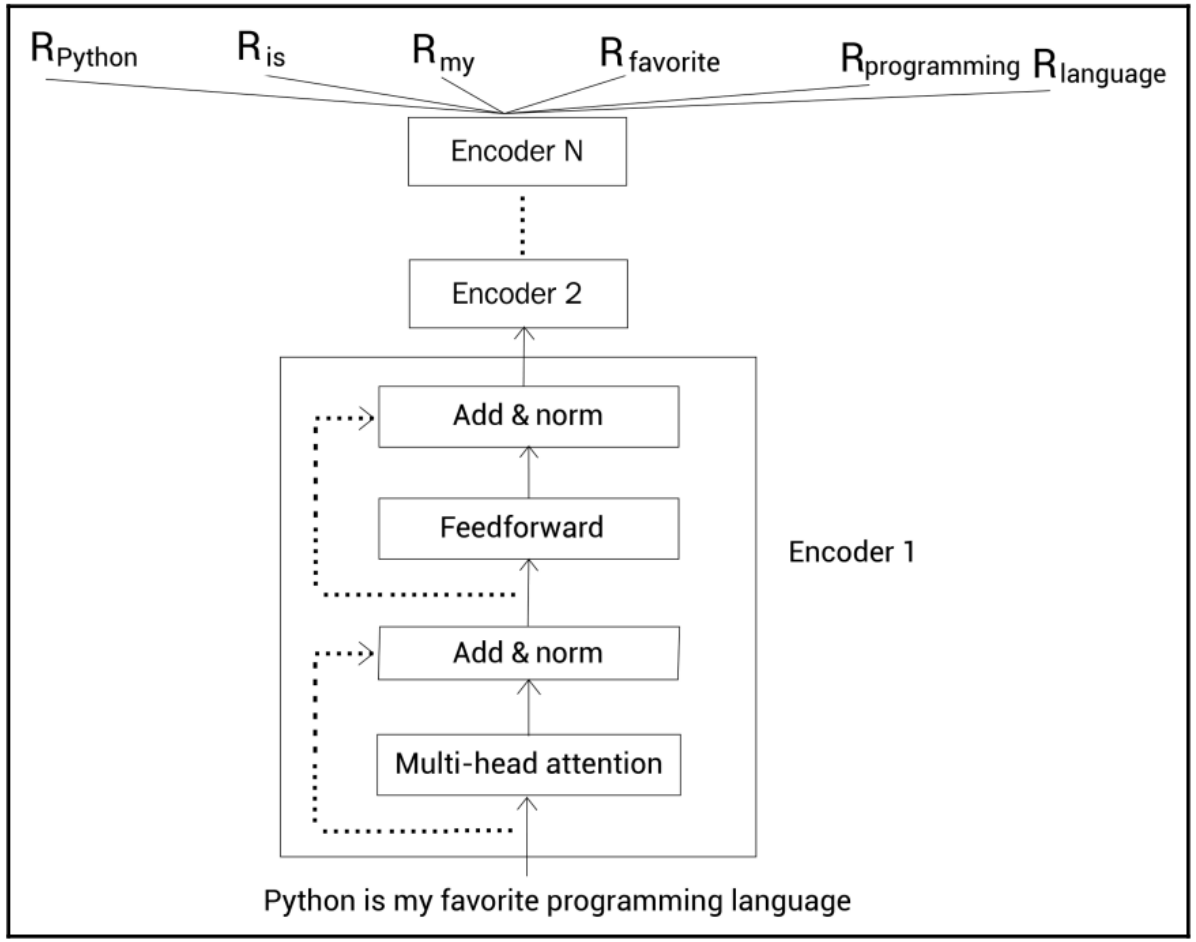
Autoregressive Model

Unlike context-free models such as **word2vec**, which generate **static embeddings** irrespective of the context, **BERT** generates **dynamic embeddings** based on the context.

Autoencoding Model

BERT - contextual representation of words





If we get embeddings for the word *'Python'* in the preceding two sentences using an embedding model such as word2vec, the embedding of the word *'Python'* would be the same in both sentences,. This is because word2vec is a context-free model,
BERT, on the other hand, is a context-based model. It will understand the context and then generate the embedding for the word based on the context. So, for the preceding two sentences, it will give different embeddings for the word *'Python'* based on the context.

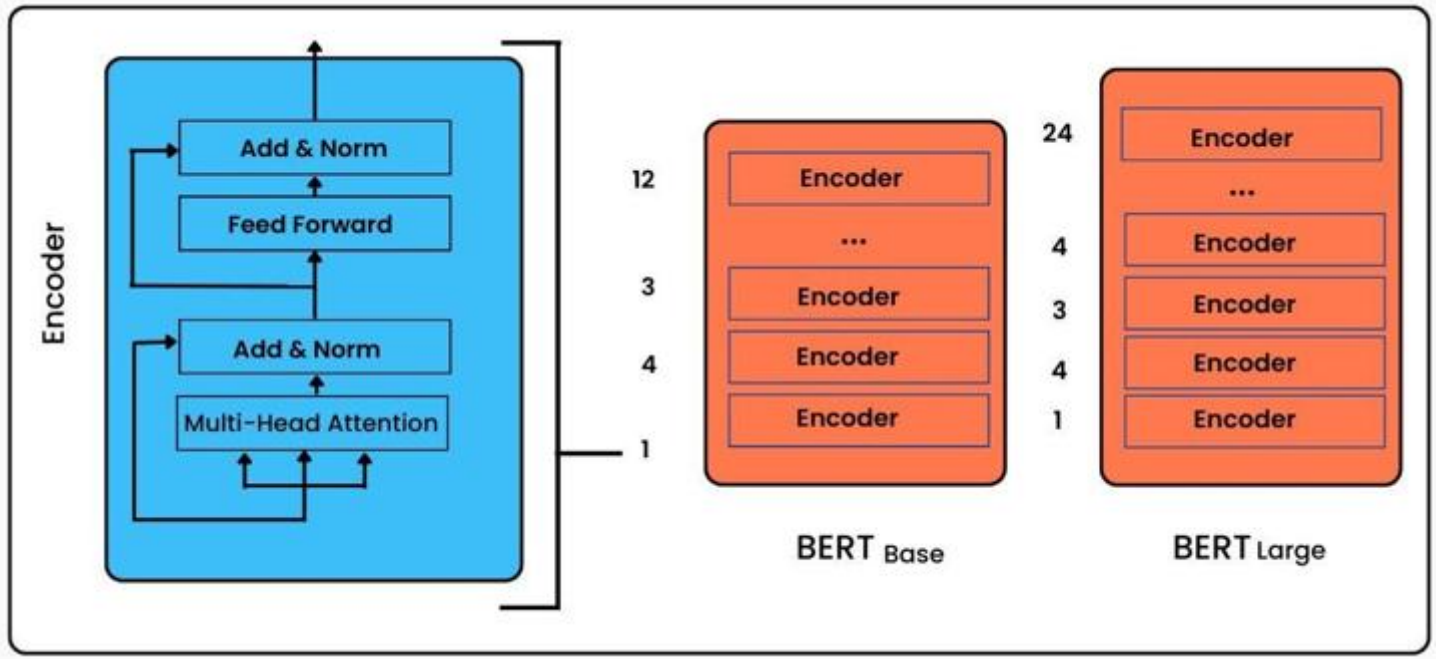
BERT – Architecture Versions

BERT-base

- The number of encoder layers L =12
- The attention head A = 12
- The hidden unit H = 768 (each token gets represented as a **768-dimensional vector**.)
- The total number of parameters - 110 million

BERT-large

- The number of encoder layers L =24
- The attention head A = 16
- The hidden unit H = 1024
- The total number of parameters - 340 million



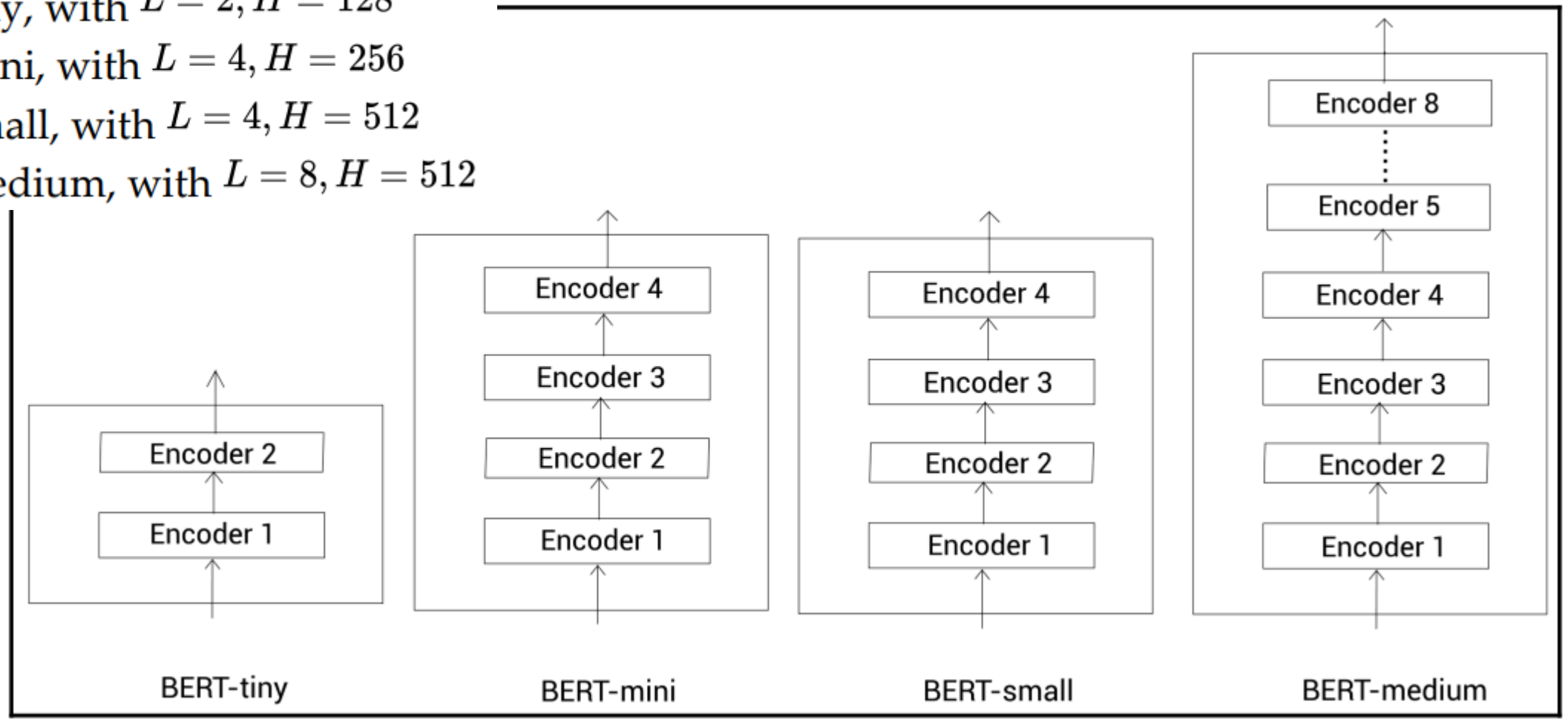
Other configurations of BERT

Bert-tiny, with $L = 2, H = 128$

Bert-mini, with $L = 4, H = 256$

Bert-small, with $L = 4, H = 512$

Bert-medium, with $L = 8, H = 512$



Self supervised/Unsupervised Pretraining

1 - Semi-supervised training on large amounts of text (books, wikipedia..etc).

The model is trained on a certain task that enables it to grasp patterns in language. By the end of the training process, BERT has language-processing abilities capable of empowering many models we later need to build and train in a supervised way.

Semi-supervised Learning Step

Model:

Dataset:

Objective:

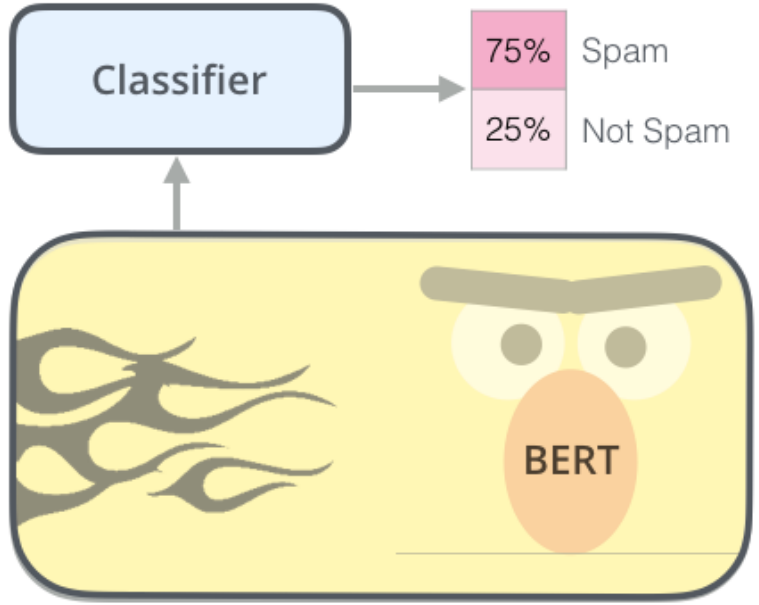
Predict the masked word (langauge modeling)

Finetuning

2 - Supervised training on a specific task with a labeled dataset.

Supervised Learning Step

Model: (pre-trained in step #1)



Dataset:

Email message	Class
Buy these pills	Spam
Win cash prizes	Spam
Dear Mr. Atreides, please find attached…	Not Spam

Auto-regressive language modeling

We can categorize the auto-regressive language modeling as follows:

- Forward (left-to-right) prediction
- Backward (right-to-left) prediction

Paris is a beautiful __.

__. I love Paris

Auto-encoding language modeling

Auto-encoding language modeling takes advantage of both forward (left-to-right) and backward (right-to-left) prediction. That is, it reads the sentence in both directions while making a prediction. Thus, we can say that the auto-encoding language model is bidirectional in nature. As we can observe from the following, to predict the bank, the auto-encoding language model reads the sentence in both directions, that is, left-to-right and right-to-left:

Paris is a beautiful __. I love Paris

BERT uses two self-supervised strategies: Masked Language Model(MLM) and Next Sentence prediction(NSP) as part of pre-training.

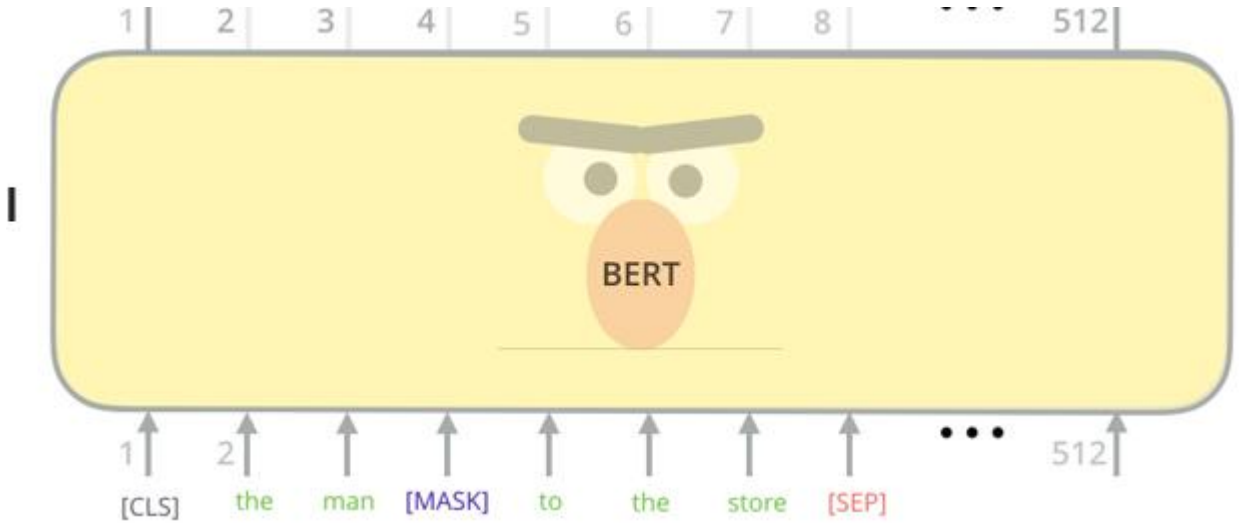
Masked LM (MLM)
Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence.

Next Sentence Prediction (NSP)
In the BERT training process, the model receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document. During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. The assumption is that the random sentence will be disconnected from the first sentence.

```
tokens = [ [CLS], Paris, is, a beautiful, [MASK], [SEP], I, love,
Paris, [SEP] ]
```

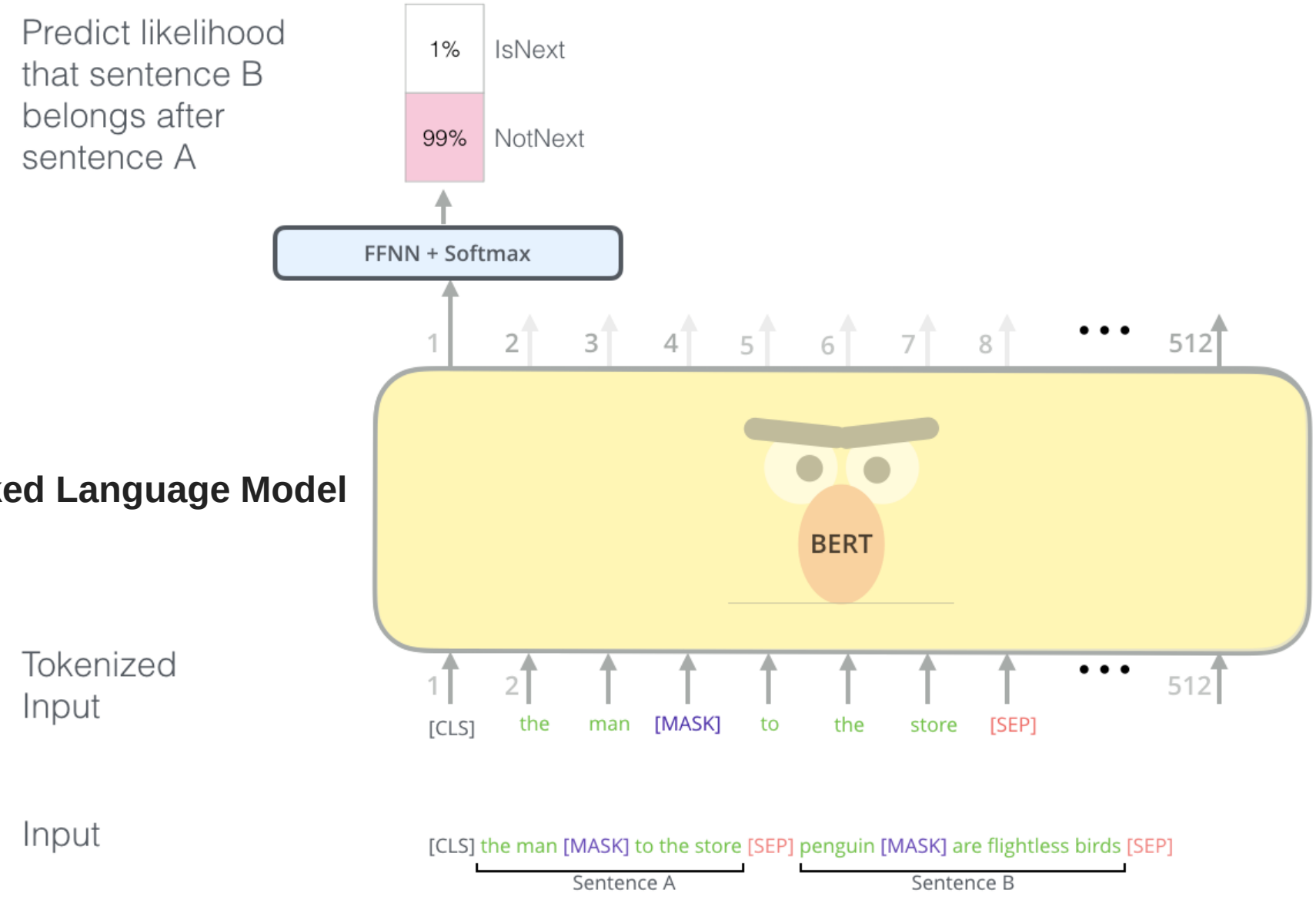
[CLS] and [SEP]

- The whole input to the BERT has to be given a single sequence. BERT uses special tokens [CLS] and [SEP] to understand input properly. [SEP] token has to be inserted at the end of a single input.
- When a task requires more than one input such as NLI and Q-A tasks, [SEP] token helps the model to understand the end of one input and the start of another input in the same sequence input.
- [CLS] is a special classification token and the last hidden state of BERT corresponding to this token ($h_{[CLS]}$) is used for classification tasks



BERT uses two unsupervised strategies: Masked Language Model(MLM) and Next Sentence prediction(NSP) as part of pre-training.

Masked Language Model



Token embedding

- S1: 'Paris is a beautiful city',
- S2: 'I love Paris'.

- Tokenize:

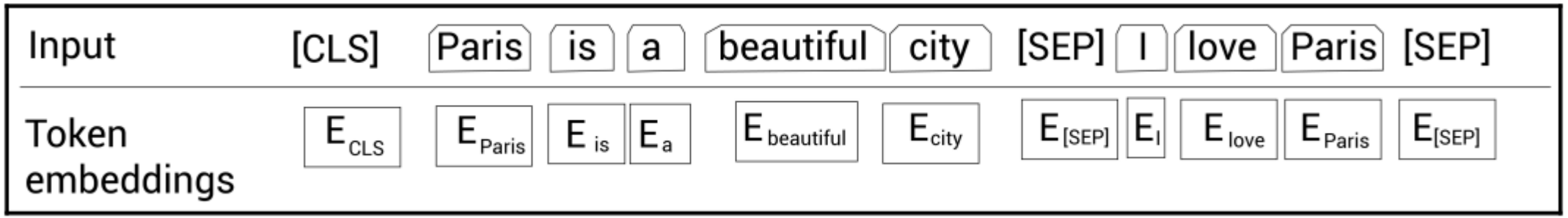
```
tokens = [Paris, is, a beautiful, city, I, love, Paris]
```

- Add the [CLS] token at the beginning of the first sentence and the [SEP] token at the end of every sentence

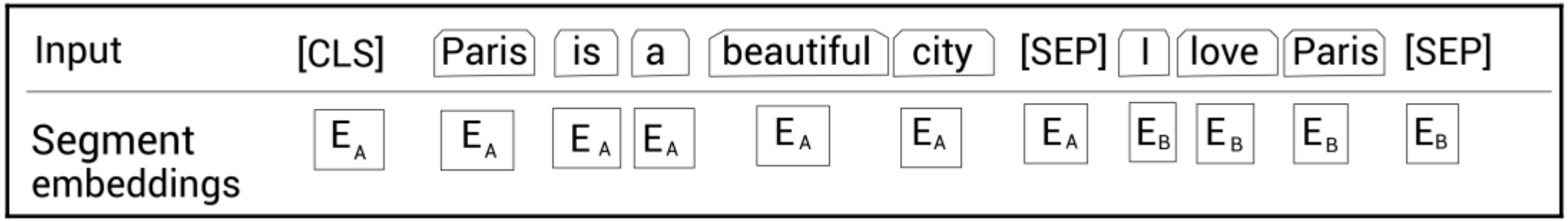
```
tokens = [ [CLS], Paris, is, a beautiful, city, [SEP], I, love, Paris,
[SEP] ]
```

- randomly mask 15% of the tokens

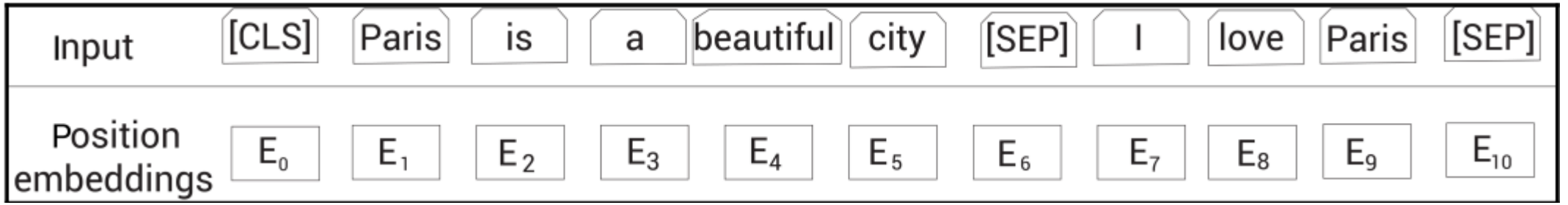
```
tokens = [ [CLS], Paris, is, a beautiful, [MASK], [SEP], I, love, Paris,
[SEP] ]
```



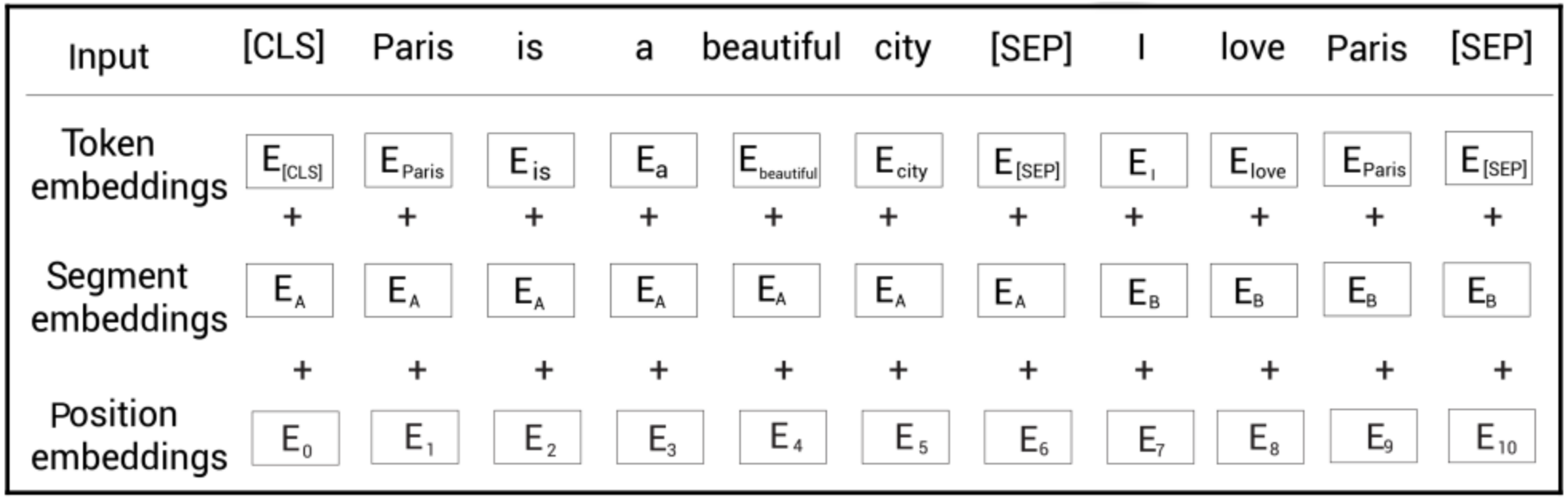
Segment embedding/Sentence Embedding



Position embedding



Final representation



Masking token create a discrepancy in Fine tuning

- have [MASK] tokens in the input
- for 15% the masked tokens, apply 80-10-10% rule
  - For 80% of the time, we replace the token (actual word) with the [MASK] token

```
tokens = [ [CLS], Paris, is, a beautiful, [MASK], [SEP], I, love,
Paris, [SEP] ]
```

  - For 10% of the time, we replace the token (actual word) with a random token

```
tokens = [ [CLS], Paris, is, a beautiful, love, [SEP], I, love,
Paris, [SEP] ]
```

  - For 10% of the time, we don't make any changes

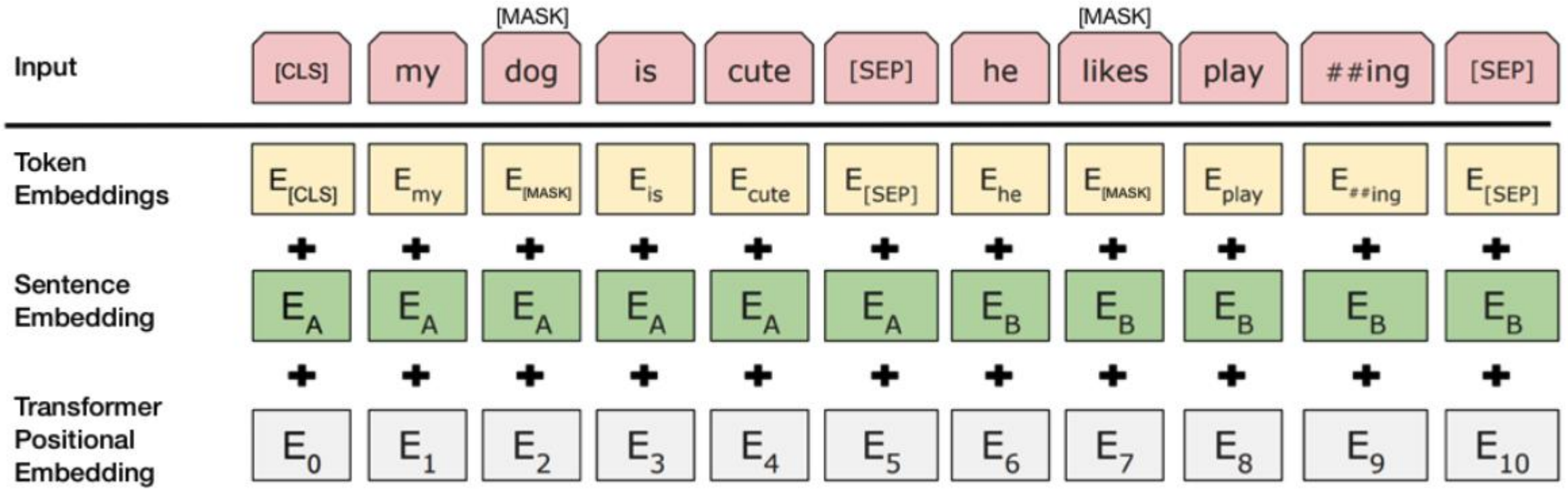
```
tokens = [ [CLS], Paris, is, a beautiful, city, [SEP], I, love,
Paris, [SEP] ]
```

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

1.A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.

2.A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar in concept to token embeddings with a vocabulary of 2.

3.A positional embedding is added to each token to indicate its position in the sequence. The concept and implementation of positional embedding are presented in the Transformer paper.



How BERT learn deep contextual representations?

Pretraining strategies

- **Masked Language Modeling (MLM)**
  - learn **bidirectional context** by randomly **masking some words** in the input sentence and asking the model to predict them
- **Next Sentence Prediction (NSP)**
  - BERT understand **relationships between sentences** by training it to determine whether one sentence follows another

Masked language modeling

given input sentence, we randomly mask 15% of the words and train the network to predict the masked words

- S1: 'Paris is a beautiful city',
- S2: 'I love Paris'.

- Tokenize:

```
tokens = [Paris, is, a beautiful, city, I, love, Paris]
```

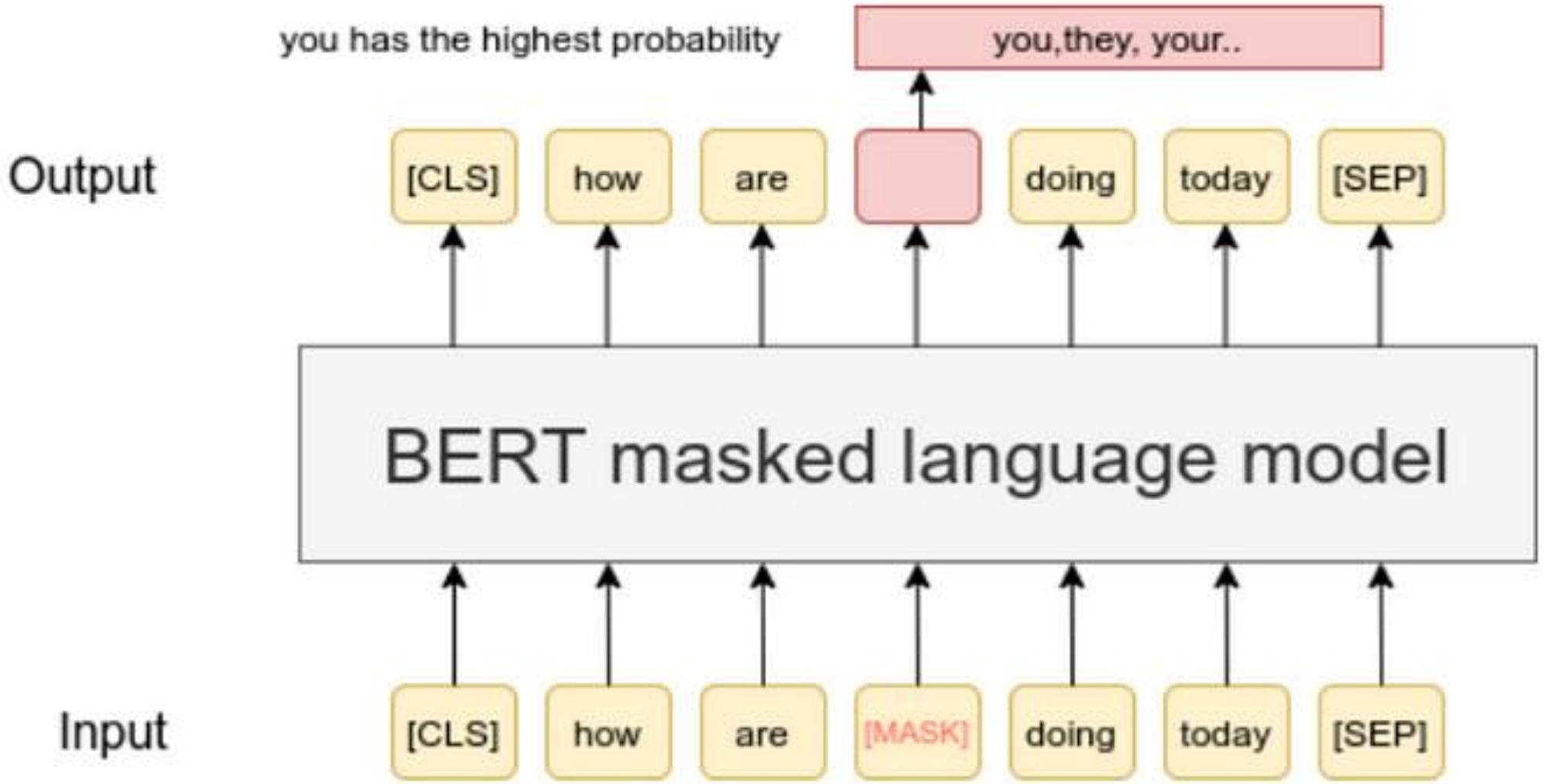
- Add the [CLS] token at the beginning of the first sentence and the [SEP] token at the end of every sentence

```
tokens = [ [CLS], Paris, is, a beautiful, city, [SEP], I, love, Paris,
[SEP] ]
```

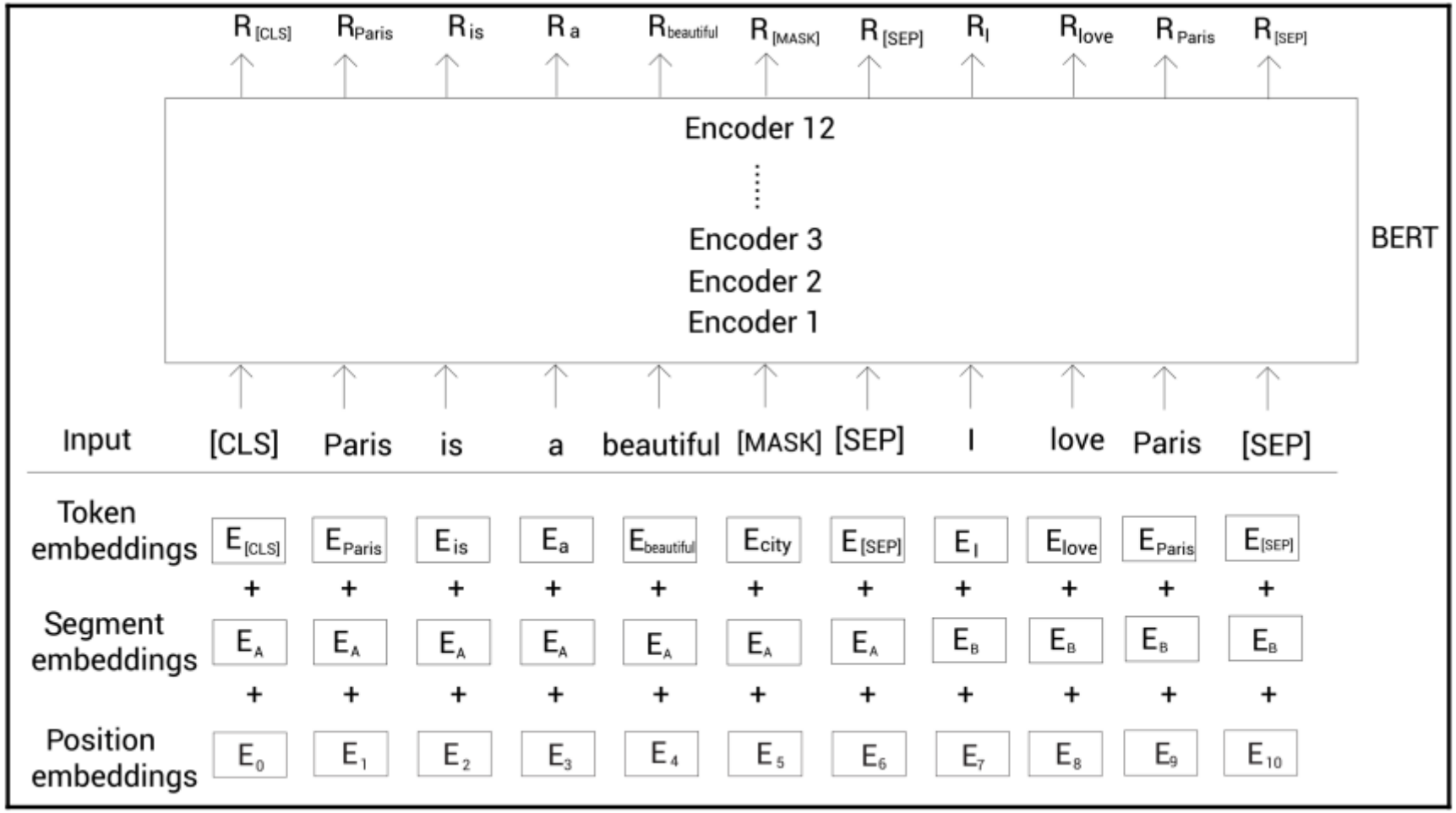
- randomly mask 15% of the tokens

```
tokens = [ [CLS], Paris, is, a beautiful, [MASK], [SEP], I, love, Paris,
[SEP] ]
```

Masked language modeling



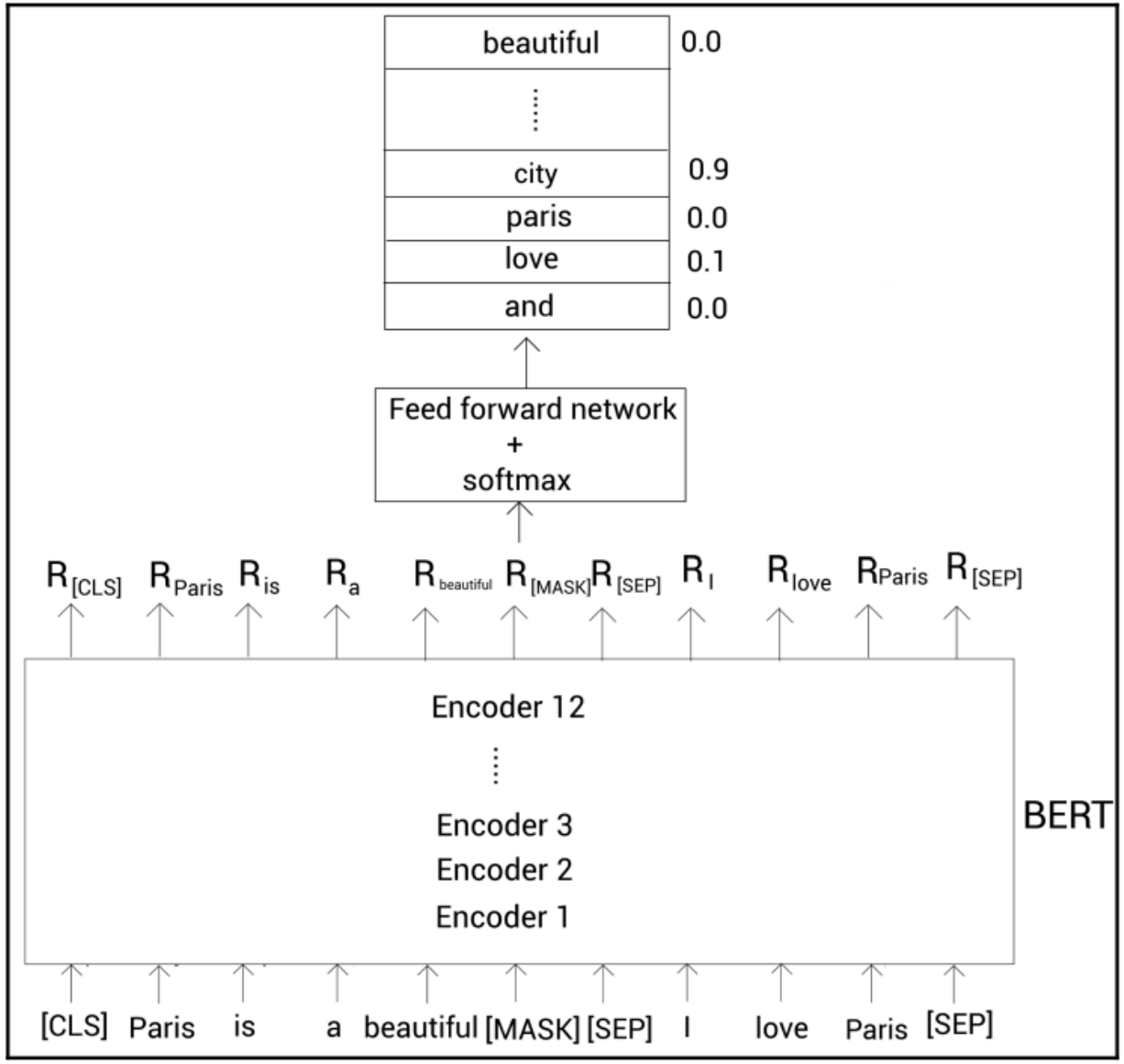
Representation of tokens in BERT



How to predict the masked token with these representations?

$$\mathrm{FFN}(x) = GELU(W_1 x + b_1) \cdot W_2 + b_2$$

In **BERT**, the **Gaussian Error Linear Unit (GELU)** activation function is used **inside each Transformer Encoder Layer**, specifically in the **Feed-Forward Network (FFN)**.



MLM- Mathematical formulation

Given a sequence of tokens:

$$X = (x_1, x_2, ..., x_n)$$

and a set of masked positions M (e.g., index 3 is masked: $X = (x_1, x_2, [MASK], x_4, ..., x_n)$), the MLM objective is to compute:

$$P(x_i | X_{\text{masked}})$$

for each masked token x_i in M, meaning:

$$P(x_3 | x_1, x_2, [\text{MASK}], x_4, ..., x_n)$$

This probability represents **how likely** the model thinks the original word at position 3 should be "**apple**", for example.

$$P(y_i | x_1, x_i = [mask], \cdots, x_T) = ?$$

Next Sentence Prediction

Sentence A: She cooked pasta.

Sentence B: It was delicious.

Sentence B follows from sentence A

isNext

Sentence A: Turn the radio on.

Sentence B: She bought a new hat.

Sentence B does not follows from sentence A

NotNext

Model understand the relation between sentences which is important for tasks such as Question Answering Text generation

Sentence Pair	Label
She cooked pasta It was delicious	isNext
Jack loves songwriting He wrote a new song	isNext
Birds fly in the sky He was reading	NotNext
Turn the radio on She bought a new hat	NotNext

50% data in **IsNext** category

50% data in **NotNext** Category

Example

Let's take the first data point in the preceding sample dataset. First, we will tokenize the sentence pair, as shown here:

```
tokens = [She, cooked, pasta, It, was, delicious]
```

Next, we add a `[CLS]` token just at the beginning of the first sentence, and an `[SEP]` token at the end of every sentence, as shown here:

```
tokens = [[CLS], She, cooked, pasta, [SEP], It, was, delicious, [SEP]]
```

Now, we feed the input tokens to the token, segment, and position embedding layers and get the input embeddings. Then, we feed the input embeddings to BERT and obtain the representation of each token. As shown in the following diagram, $R_{[CLS]}$ denotes the representation of the token `[CLS]`, R_{she} denotes the representation of the token *She*, and so on:

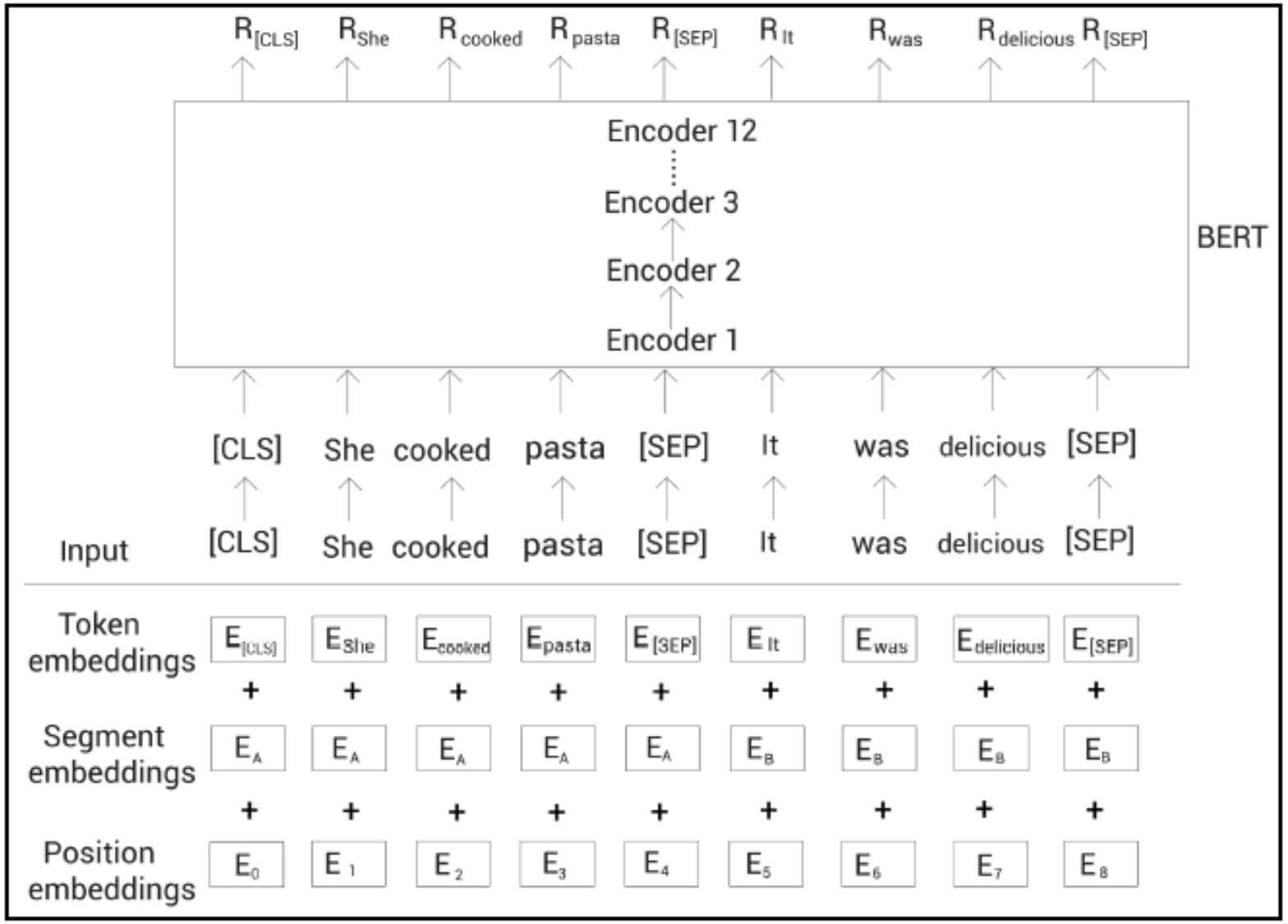


Figure 2.16 – BERT

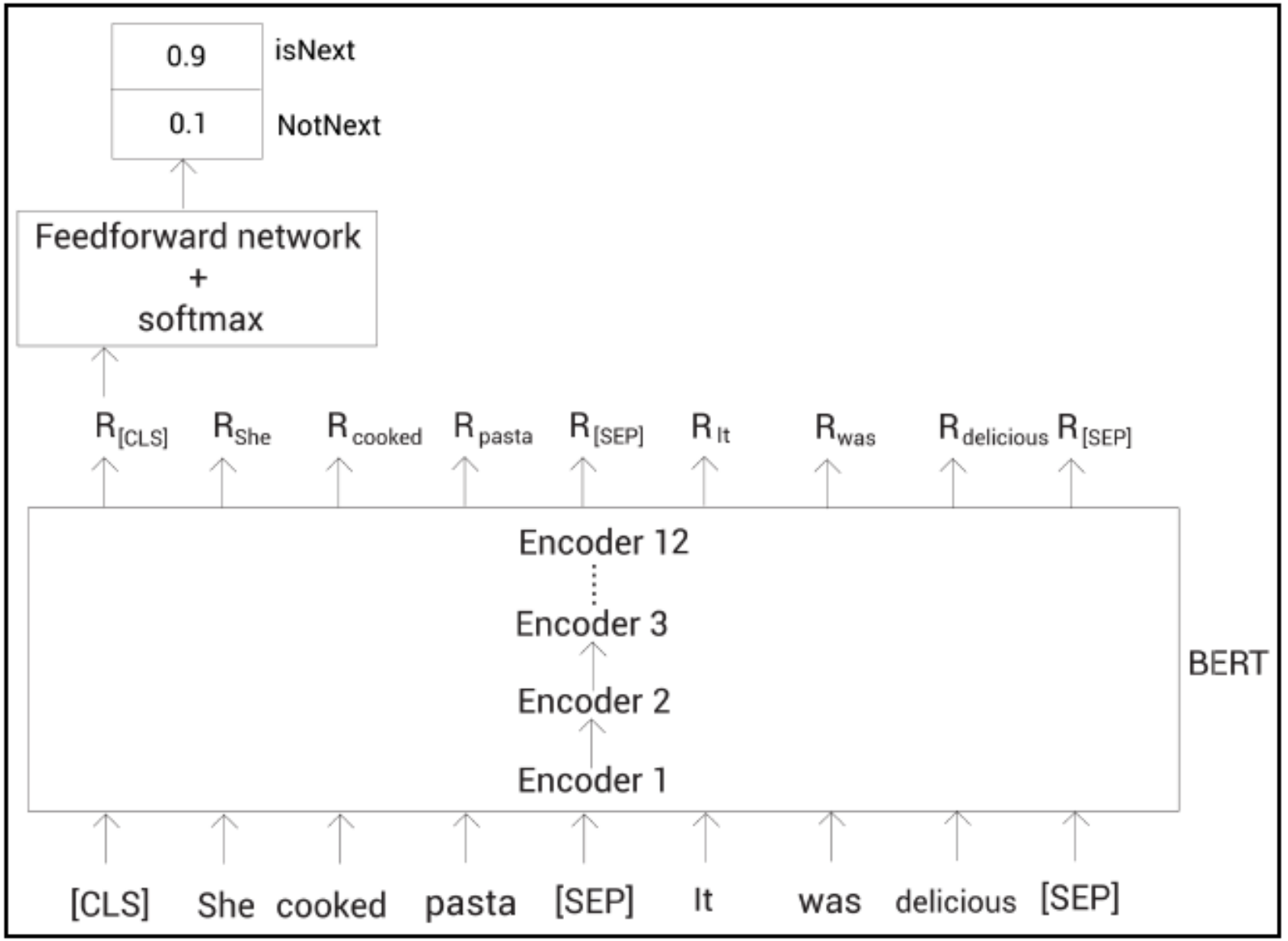


Figure 2.17 – NSP task

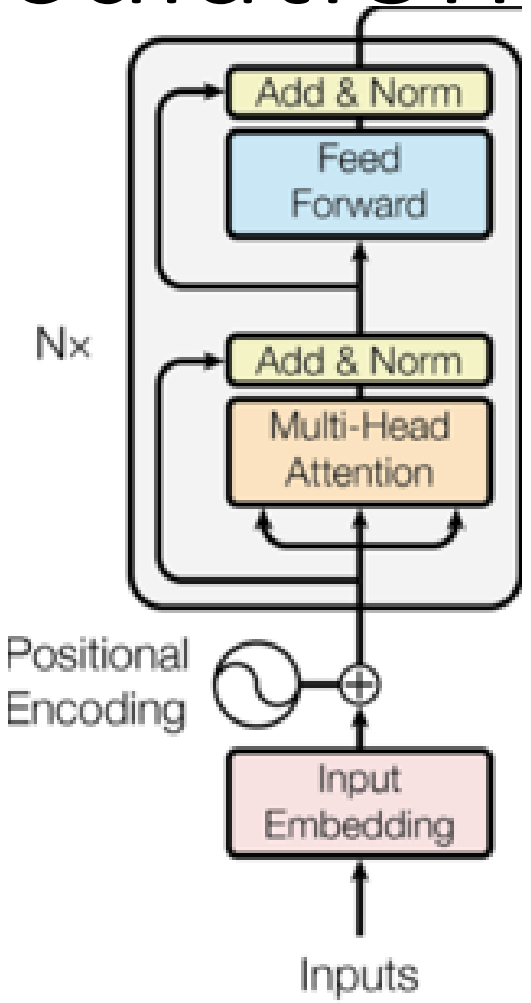
Why output taken from [CLS]?

To perform classification, we simply take the representation of the **[CLS]** token and feed it to the feedforward network with the softmax function, which then returns the probability of our sentence pair being **isNext**

. **Why do we need to take the embedding of the [CLS] token alone? Why not the embeddings of other tokens?**

[CLS] holds the aggregate representation of our sentences. Thus, we can ignore the representation of all the other tokens and simply take the representation of the [CLS] token R[CLS] and feed it to the feedforward layer with a softmax function, which returns the probability.

No: of Parameters Calculation- $BERT_{base}$



Total Parameters - $BERT_{base}$

Maximum Input Sequence length of BERT- 512 tokens

Vocabulary Size- 30,000

Summing up all components:

Component	Parameters
Embedding Layer	23.4M Token Embedding + Segment Embedding + position embedding
Transformer Layers (12 × 7M)	84M
Total Parameters	107.4M (~110M in the original BERT paper)

BERT-base

The number of encoder layers L =12

The attention head A = 12

The hidden unit H = 768 (each token gets represented as a 768-dimensional vector.)

The total number of parameters - 110 million

- **Explicitly counted parameters: 107.4M**
- **Bias terms + LayerNorm + other small components: ~2.5M**
- **Final total (rounded in the paper): ~110M**

$BERT_{base}$– Parameter Calculation

• Embedding Layer

Maximum Input Sequence length of BERT- 512 tokens

•Each token (word or subword) is mapped to a **768-dimensional vector**.
•**vocabulary lookup matrix** of size **30,000(v)×768(d)**.

$$\text{Embedding Matrix} \in \mathbb{R}^{(V \times d)}$$

1 Token Embeddings:

$$30,000 \times 768 \approx 23M$$

- BERT has a **vocabulary size of 30,000 words.**
- Each token (word) is represented by a **768-dimensional embedding vector.**
- The total number of parameters for token embeddings is:

$$30,000 \times 768 = 23,040,000 \approx 23M$$

learned token embeddings are updated during pretraining using **Masked Language Modeling (MLM)**.

$BERT_{base}$– Parameter Calculation

- **Embedding Layer-** Segment Embedding

2 Segment Embeddings:

$$2 \times 768 \approx 1536$$

- Segment embeddings help BERT differentiate between **two sentences** in a pair (Sentence A and Sentence B).
- There are **two segment embeddings** (E_A and E_B), each of size **768**.
- The total number of parameters is:

$$2 \times 768 = 1536$$

$BERT_{base}$– Parameter Calculation

- **Embedding Layer-** Position embedding

3 Position Embeddings:

$$512 \times 768 \approx 0.4M$$

- BERT **does not have recurrence** (like RNNs), so it needs **position embeddings** to capture word order.
- The **maximum input sequence length** in BERT is **512 tokens**.
- Each of the **512 positions** is represented by a **768-dimensional embedding vector**.
- The total number of parameters is:

$$512 \times 768 = 393,216 \approx 0.4M$$

$BERT_{base}$– Parameter Calculation

- **Embedding Layer-** Summary

Component	Shape	Parameter Calculation	Total Parameters
Token Embeddings	$30,000 \times 768$	$30,000 \times 768$	**23.04M** (~23M)
Segment Embeddings	2×768	2×768	1,536 (~1.5K)
Position Embeddings	512×768	512×768	0.39M (~0.4M)
Total (Embedding Layer)	—	—	~23.4M

No of Parameters- Transformer layers

Recap-Steps in Self Attention –

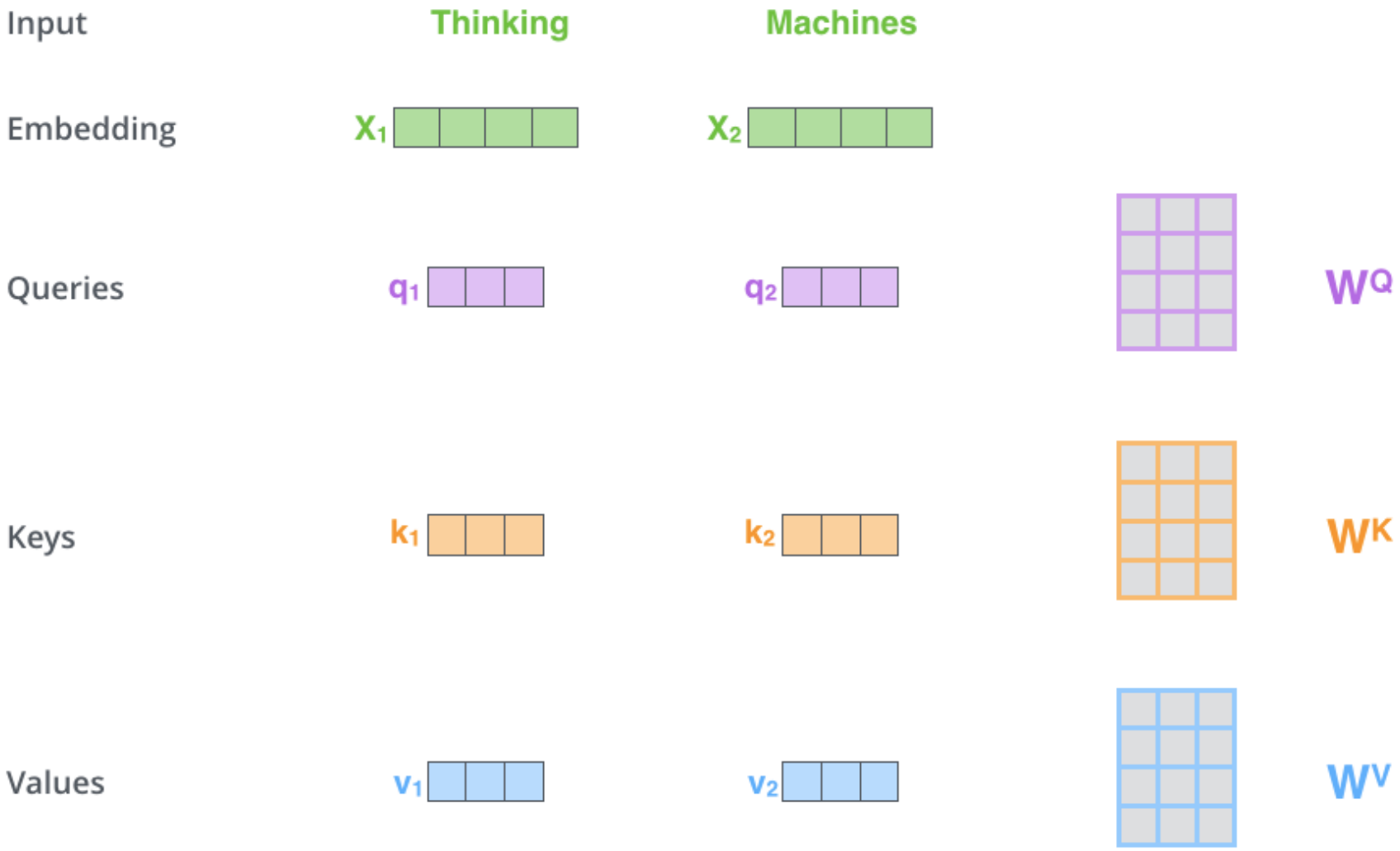
First step

Create 3 vectors

- **Query vector (qi)**
- **Key vector (ki)**
- **Value vector (vi)**

From each of the encoder's input vectors –ie the embedding of each word).

These 3 vectors are created by multiplying the embedding by three matrices that we trained during the training process

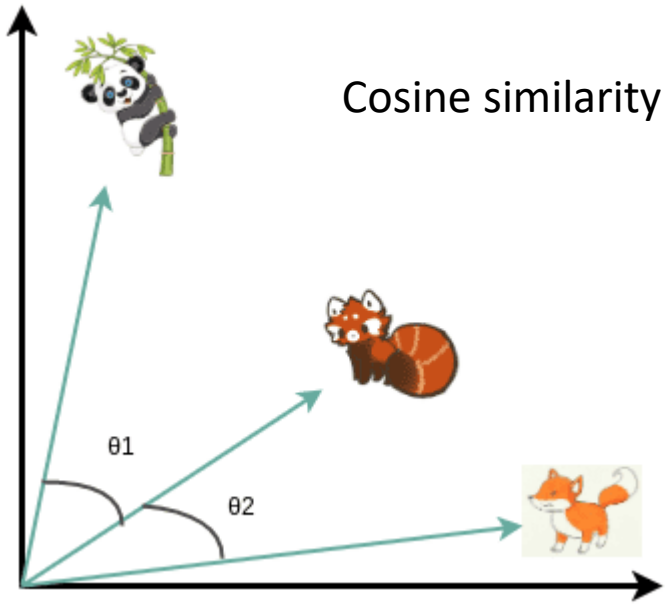


Steps in Self Attention

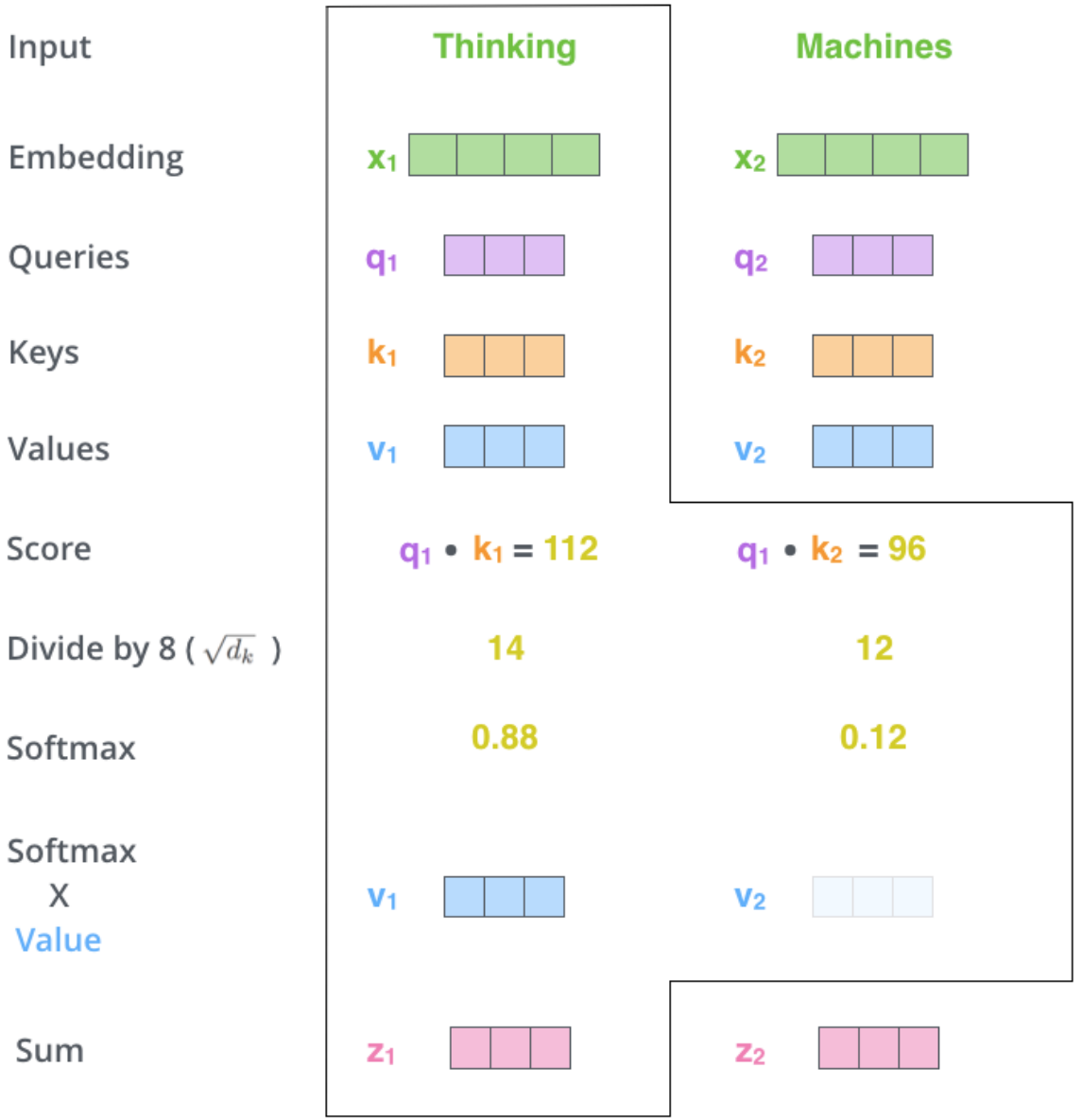
The **second step**

- Calculate a score : dot product of the query vector with the key vector of the respective word

$q_1 \cdot k_1 = 112$ $q_1 \cdot k_2 = 96$



Cosine similarity



“Multi-headed” attention

1) This is our input sentence*

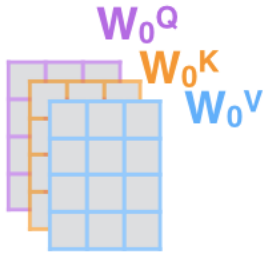
2) We embed each word*

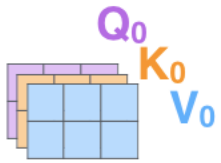
3) Split into 8 heads. We multiply X or R with weight matrices

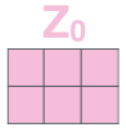
4) Calculate attention using the resulting Q/K/V matrices

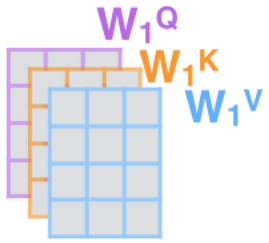
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

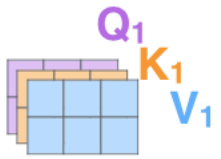
Thinking Machines

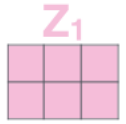




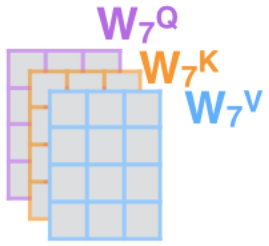


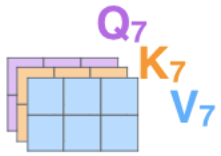


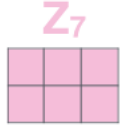


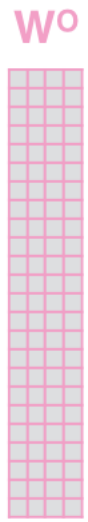


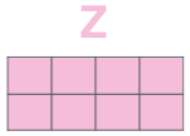
* In all encoders other than #0, we don’t need embedding. We start directly with the output of the encoder right below this one

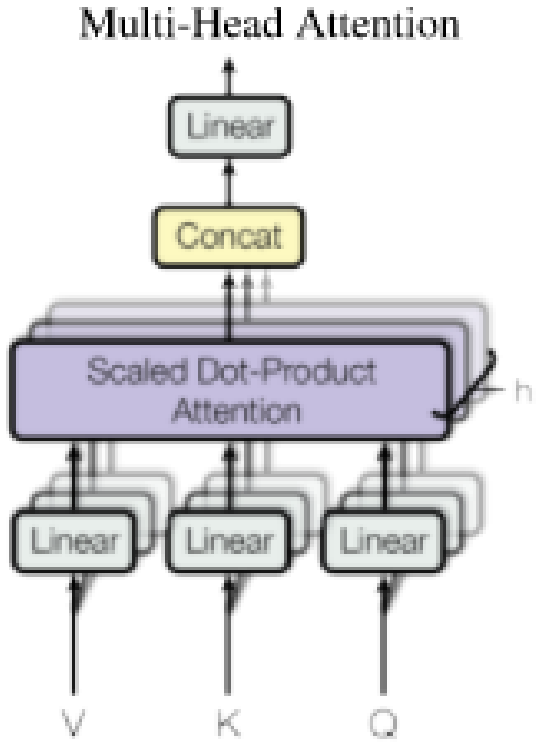












multiple “representation subspaces”

multiple sets of Query/Key/Value weight matrices

$BERT_{base}$– Parameter Calculation

- **Transformer Encoder Layers**

1 Self-Attention Mechanism (Ignoring Bias)

BERT uses **multi-head self-attention**, where each layer has **12 attention heads.**

- **Key (W_K), Query (W_Q), Value (W_V), and Attention Output (A) matrices**
- Each weight matrix has dimensions:

$$(768 \times 64) \times 3 \quad (\text{for } \mathrm{W}_K, \mathrm{W}_Q, \mathrm{W}_V)$$

  - Each head projects from 768 to **64** dimensions.
  - There are **3 projection matrices** (W_K, W_Q, W_V).
  - The number of heads = **12**, so we multiply by **12.**

$$(768 \times 64 \times 3) \times 12 = 1.7M$$

- **Final projection of concatenated heads back to 768-dimension**

$$768 \times 768 = 0.6M$$

Total parameters for **Self-Attention:**

$$1.7M + 0.6M = 2.3M$$

Each of the **12 attention heads** computes attention scores independently, producing **12 different 64-dimensional outputs.**

- After computing attention scores, each head **outputs a 64-dimensional vector.**
- These **12 vectors** are **concatenated** into a **single 768-dimensional vector:**

$$(12 \times 64) = 768$$

This **reconstructs** the original input size **768** but now enriched with self-attention context.

3 Final Linear Projection: 768 × 768

After concatenation, the **768-dimensional vector** (formed by stacking 12 attention heads) is projected **back to 768 dimensions** using a weight matrix W_O:

$$W_O \in \mathbb{R}^{768 \times 768}$$

This transformation **allows BERT to mix information across all heads** into a single vector per token. The number of parameters in this projection layer is:

$$768 \times 768 = 0.6M$$

Without the **linear projection layer**, the **outputs from different heads remain separate**, and their information is **not integrated** effectively.

$BERT_{base}$– Parameter Calculation

Summing up-Self attention

- **Transformer Encoder Layers**

4 Total Parameters for Self-Attention

Adding both components:

- Multi-Head Attention Weights: 1.7M
- Final Linear Projection: 0.6M
- Total for Self-Attention: 2.3M

BERT base Parameter Calculation –cont.

- **Transformer Encoder Layers**

2 Feed-Forward Network (FFN)

BERT's **FFN (Feed-Forward Network)** per layer consists of:

1. A linear transformation **expanding** from **768 to 3072**.
2. A **GELU activation function**.
3. A linear transformation **compressing** from **3072 back to 768**.

$$FFN = 768 \times 3072 + 3072 \times 768 + (3072 + 768) = 4.7M$$

- The extra **(3072 + 768)** represents biases in the linear layers.

Total parameters for **Feed-Forward Network**:

(a) First Linear Transformation (Expansion)

- **Input:** 768-dimensional vector.
- **Output:** 3072-dimensional vector.
- **Weight matrix size:** 768×3072.
- **Number of parameters:**

$$768 \times 3072 = 2.36M$$

(b) Second Linear Transformation (Compression)

- **Input:** 3072-dimensional vector.
- **Output:** 768-dimensional vector.
- **Weight matrix size:** 3072×768.
- **Number of parameters:**

$$3072 \times 768 = 2.36M$$

(c) Bias Terms

Each linear layer has **biases**:

- First layer has **3072 biases**.
- Second layer has **768 biases**.
- Total biases:

$$3072 + 768 = 3840 \approx 0.004M$$

Adding everything together:

$$768 \times 3072 + 3072 \times 768 + (3072 + 768) = 4.7M$$

Summary

1 Embedding Layer Parameters

Component	Shape	Parameters
Token Embeddings	$30,000 \times 768$	23M
Segment Embeddings	2×768	1.5K
Position Embeddings	512×768	0.4M
Total Embedding Layer	—	23.4M

Component	Shape	Parameters
Self-Attention (W_K, W_Q, W_V)	$(768 \times 64) \times 3 \times 12$	1.7M
Final Self-Attention Projection	768×768	0.6M
Feed-Forward Network (FFN)	$768 \times 3072 + 3072 \times 768$	4.7M
Total per Transformer Layer	—	7M

Since BERT Base has **12 Transformer Encoder Layers**, we multiply:

$$7M \times 12 = 84M$$

Summing up all components:

Component	Parameters
Embedding Layer	23.4M
Transformer Layers (12 × 7M)	84M
Total Parameters	107.4M (~110M in the original BERT paper)

BERT dimensions

Maximum number of words (sequence length, N) = 512

Embedding size (H) = 768

Number of attention heads (h) = 12

Head dimension (d_k) = $H/h = 768/12 = 64$

X has a dimension of 512×768.

Query weight per head: $W_Q^{\text{head}} \in \mathbb{R}^{768\times64}$

Key weight per head: $W_K^{\text{head}} \in \mathbb{R}^{768\times64}$

Value weight per head: $W_V^{\text{head}} \in \mathbb{R}^{768\times64}$

$$Q^{\text{head}}, K^{\text{head}}, V^{\text{head}} \in \mathbb{R}^{512\times64}$$

$$QK^T \Rightarrow (512 \times 64) \times (64 \times 512) = 512 \times 512$$

$$(QK^T)V \Rightarrow (512 \times 512) \times (512 \times 64) = 512 \times 64$$

each **head output** is 512×64,

after concatenation

$$Z \in \mathbb{R}^{512\times(12\times64)}$$

$$Z \in \mathbb{R}^{512\times768}$$

The concatenated output Z is passed through a **final linear projection layer** using:

$$W_O \in \mathbb{R}^{768\times768}$$

$$Z_{\text{final}} \in \mathbb{R}^{512\times768}$$

BERT Loss function

Pre-training

Dataset:

BookCorpus

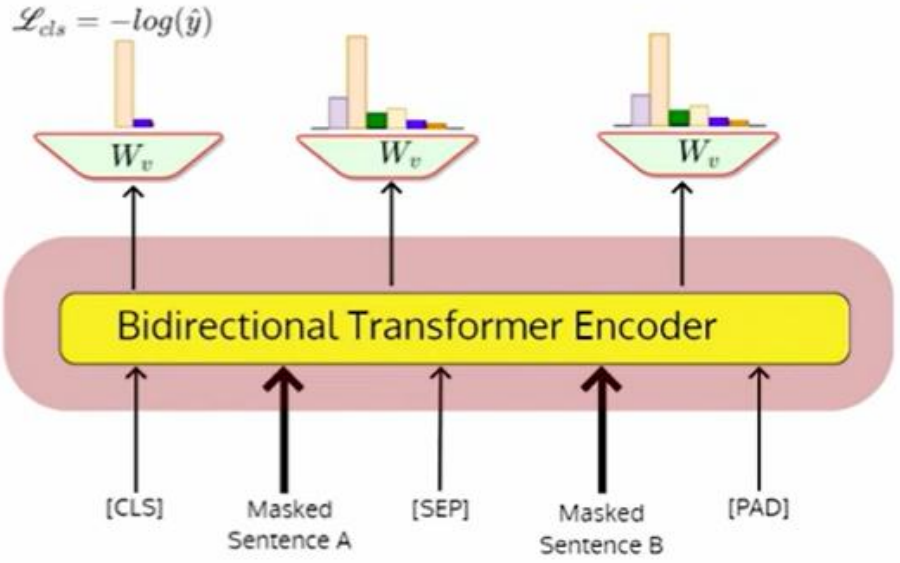
800M words

WikiPedia

2500M words

Vocabulary size: 30,000

$$\mathscr{L} = \frac{1}{N} \sum_{y_i \in \mathcal{M}} -\log(\hat{y}_i) + \mathscr{L}_{cls}$$



1. MLM Loss (Cross-Entropy Loss for Masked Tokens)

- In MLM, BERT randomly masks some tokens in the input and tries to predict them.
- The loss is computed using **Cross-Entropy Loss** over only the masked tokens.
- Let p_i be the predicted probability distribution for the masked token at position i, and y_i be the true token index. The MLM loss is:

$$L_{MLM} = -\frac{1}{N} \sum_{i=1}^{N} \log p_i(y_i)$$

where N is the number of masked tokens in the batch.

Given Input Sentence:

"The cat sat on the [MASK] ."

True Token:

The masked token is "**mat**".

BERT's Predicted Token Probabilities for [MASK]:

$$P = \{"\text{dog}" : 0.1, "\text{mat}" : 0.6, "\text{carpet}" : 0.3\}$$

- The correct word "**mat**" has a probability of **0.6**.

MLM Loss Calculation (Cross-Entropy Loss)

Using the cross-entropy loss formula:

$$L_{MLM} = -\log(P_{\text{correct word}})$$

$$L_{MLM} = -\log(0.6) = 0.51$$

log(1)=0
Log(.999) =-.0004
Log(.0001)=-4
Log(0)=-infinity

2. NSP Loss (Binary Cross-Entropy for Sentence Relationship Prediction)

- BERT learns to predict if the second sentence in a pair follows the first (IsNext) or is a random sentence (NotNext).
- The loss is computed using **Binary Cross-Entropy Loss**:

$$L_{NSP} = -\frac{1}{M}\sum_{j=1}^{M}[y_j \log p_j + (1-y_j)\log(1-p_j)]$$

where M is the number of sentence pairs in the batch, y_j is the ground truth label (1 for IsNext, 0 for NotNext), and p_j is the predicted probability.

Term1 -Maximise probability/minimise Loss for predicting class 1 correctly.
Term 2- Maximise probability/minimise Loss for predicting class 0 correctly.

Given Sentence Pair:

1. **Sentence A:** *"The cat sat on the mat."*
2. **Sentence B:** *"It was sleeping peacefully."*

True Label:

- 1 (IsNext) → Sentence B is the actual next sentence.

BERT's Predicted Probability for IsNext:

$$P_{\text{IsNext}} = 0.7$$

NSP Loss Calculation (Binary Cross-Entropy Loss)

$$L_{NSP} = -\left(y \log P + (1-y) \log(1-P)\right)$$

$$L_{NSP} = -\left(1 \times \log(0.7) + 0 \times \log(1-0.7)\right)$$

$$L_{NSP} = -\log(0.7) = 0.36$$

The **Binary Cross-Entropy (BCE) loss** ensures that the model learns to **assign high probabilities to correct outputs** and **low probabilities to incorrect outputs**. It does this by penalizing incorrect predictions heavily and rewarding correct predictions.

Loss is very low → Model is rewarded for making a correct and confident prediction.
Loss is very high → Model is strongly penalized for being confident but wrong.
Loss is moderate → Model is uncertain, so it's penalized, but not as heavily as a confident wrong answer.

log(1)=0
Log(0)=-infinity

🔹 Step 1: Compute Loss for Each Sample

For sentence pair 1 (A, B) → $y_1 = 1, p_1 = 0.9$

$$L_1 = -[1 \cdot \log(0.9) + (1-1) \cdot \log(1-0.9)]$$

$$= -\log(0.9) = 0.105$$

🔹 Loss is low because the prediction (0.9) is close to ground truth (1).

For sentence pair 2 (C, D) → $y_2 = 0, p_2 = 0.2$

$$L_2 = -[0 \cdot \log(0.2) + (1-0) \cdot \log(1-0.2)]$$

$$= -\log(0.8) = 0.223$$

🔹 Loss is low because the prediction (0.2) correctly indicates that sentence B does not follow A (NotNext, $y = 0$).

For sentence pair 3 (E, F) → $y_3 = 1, p_3 = 0.6$

$$L_3 = -[1 \cdot \log(0.6) + (1-1) \cdot \log(1-0.6)]$$

$$= -\log(0.6) = 0.511$$

🔹 Loss is higher because the model is uncertain (0.6 is close to 50%) and not confident about the correct label (1).

Case 1: Completely Wrong Prediction with High Confidence

Suppose we have a sentence pair (G, H) where:

- **Ground truth** $y_4 = 1$ (meaning Sentence B should follow Sentence A).
- **Model prediction** $p_4 = 0.01$ (model is very confident that Sentence B does NOT follow Sentence A).

The loss is computed as:

$$L_4 = -[1 \cdot \log(0.01) + (1-1) \cdot \log(1-0.01)]$$

$$= -\log(0.01) = 4.605$$

log(1)=0
Log(0)=-infinity

- **Step 2: Compute the Final NSP Loss**

Now, we compute the average loss across the batch:

$$L_{NSP} = \frac{1}{3}\sum_{j=1}^{3} L_j$$

$$= \frac{1}{3}(0.105 + 0.223 + 0.511)$$

$$= \frac{0.839}{3} = 0.28$$

1. Forward Pass

BERT processes an input sequence through **multiple transformer layers**:

- **Embedding Layer**: Converts tokens to embeddings.
- **Multi-Head Self-Attention Layer**: Computes contextual relationships.
- **Feedforward Layers**: Applies non-linear transformations.
- **Output Layer**: Produces predictions for MLM (Masked Language Model) and NSP (Next Sentence Prediction).

The loss L_{total} is computed as:

$$L_{\text{total}} = L_{\text{MLM}} + L_{\text{NSP}}$$

where:

- L_{MLM} is the cross-entropy loss for masked words.
- L_{NSP} is the binary cross-entropy loss for sentence prediction.

2. Compute Gradients (Backpropagation)

Backpropagation applies the **chain rule** to compute gradients layer by layer:

(a) Compute Loss Gradients for Output Layer

- The loss is **differentiated w.r.t the output logits**.
- For MLM:

$$\frac{\partial L_{\text{MLM}}}{\partial \hat{y}_{\text{MLM}}} = \text{softmax}(\hat{y}_{\text{MLM}}) - y_{\text{true}}$$

- For NSP:

$$\frac{\partial L_{\text{NSP}}}{\partial \hat{y}_{\text{NSP}}} = \text{sigmoid}(\hat{y}_{\text{NSP}}) - y_{\text{true}}$$

(b) Compute Gradients for Feedforward Layers

- Backpropagate the loss through the **fully connected layers**.
- Compute derivatives w.r.t weights W_{FFN} and biases b_{FFN}:

$$\frac{\partial L}{\partial W_{\text{FFN}}} = \frac{\partial L}{\partial Y_{\text{FFN}}} \cdot \frac{\partial Y_{\text{FFN}}}{\partial W_{\text{FFN}}}$$

$$L_{MLM} = -\sum_{i} y_{\text{true},i} \log(\text{softmax}(\hat{y}_i))$$

$$L_{NSP} = -\left[y_{\text{true}} \log(\text{sigmoid}(\hat{y})) + (1 - y_{\text{true}}) \log(1 - \text{sigmoid}(\hat{y}))\right]$$

(c) Compute Gradients for Multi-Head Self-Attention

- Compute gradients for **Query (Q), Key (K), and Value (V) matrices**:

$$\frac{\partial L}{\partial Q}, \quad \frac{\partial L}{\partial K}, \quad \frac{\partial L}{\partial V}$$

- Use these to compute gradients for the **attention scores and output**:

$$\frac{\partial L}{\partial W_Q}, \quad \frac{\partial L}{\partial W_K}, \quad \frac{\partial L}{\partial W_V}$$

(d) Compute Gradients for Embedding Layer

- The embedding layer weights W_{emb} are updated based on backpropagated errors from the self-attention layers.

3. Weight Update (Gradient Descent)

Once gradients are computed, weights are updated using the **Adam optimizer**:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla L$$

where:

- η is the learning rate.
- ∇L is the gradient of the loss function.
- θ represents all trainable parameters.

Fine-tuning BERT for downstream tasks

- Fine-tuning implies that we are not training BERT from scratch; instead, we are using the pre-trained BERT and updating its weights according to our task.

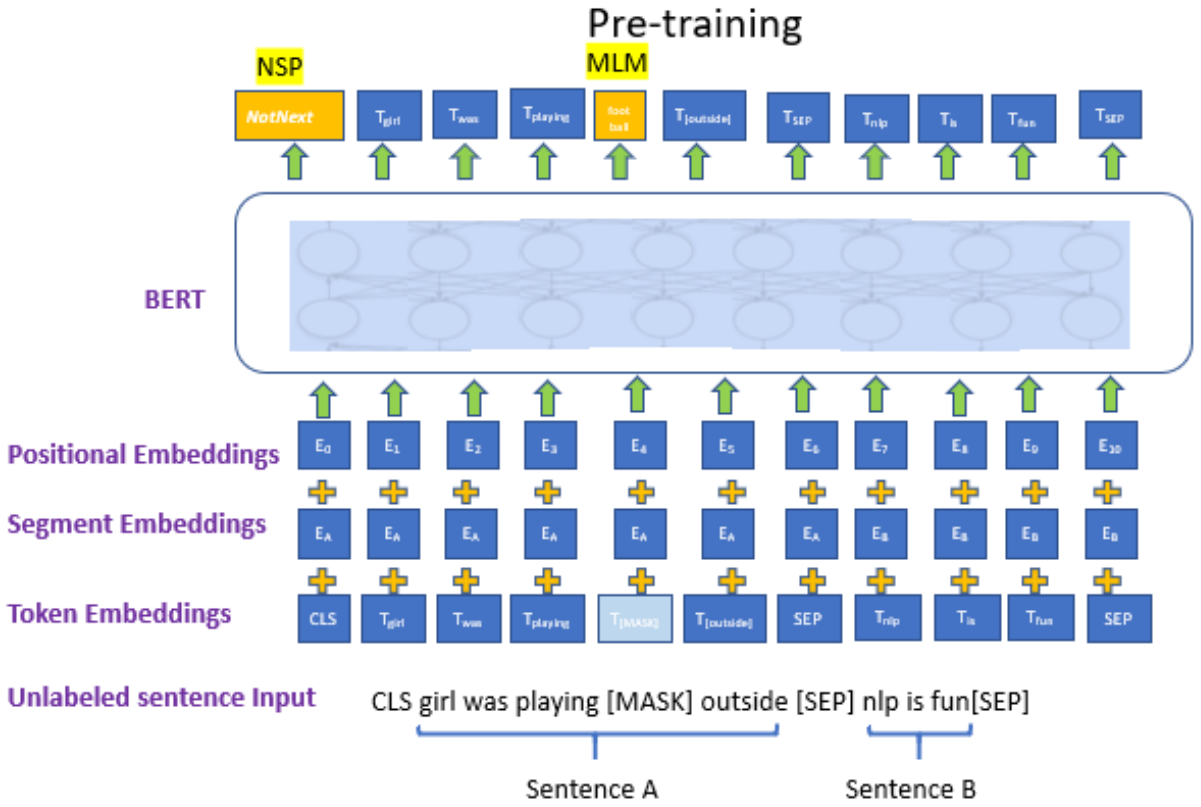
  During fine-tuning, we can adjust the weights of the model in the following two ways:

  - Update the weights of the pre-trained BERT model along with the classification layer

  .

  - Update only the weights of the classification layer and not the pre-trained BERT model. When we do this, it becomes the same as using the pre-trained BERT model as a feature extractor.

For fine-tuning the BERT model, we first initialize with the pre-trained parameters, and then all of the parameters are fine-tuned using labeled data from the downstream tasks.



Fine tuning- Sentiment Classification

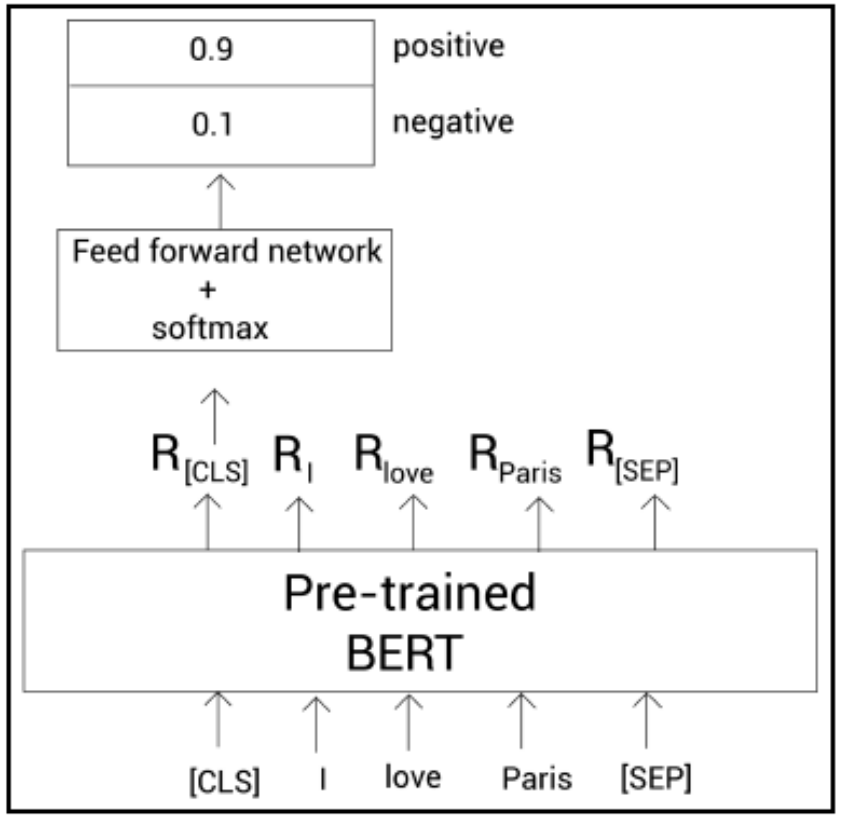
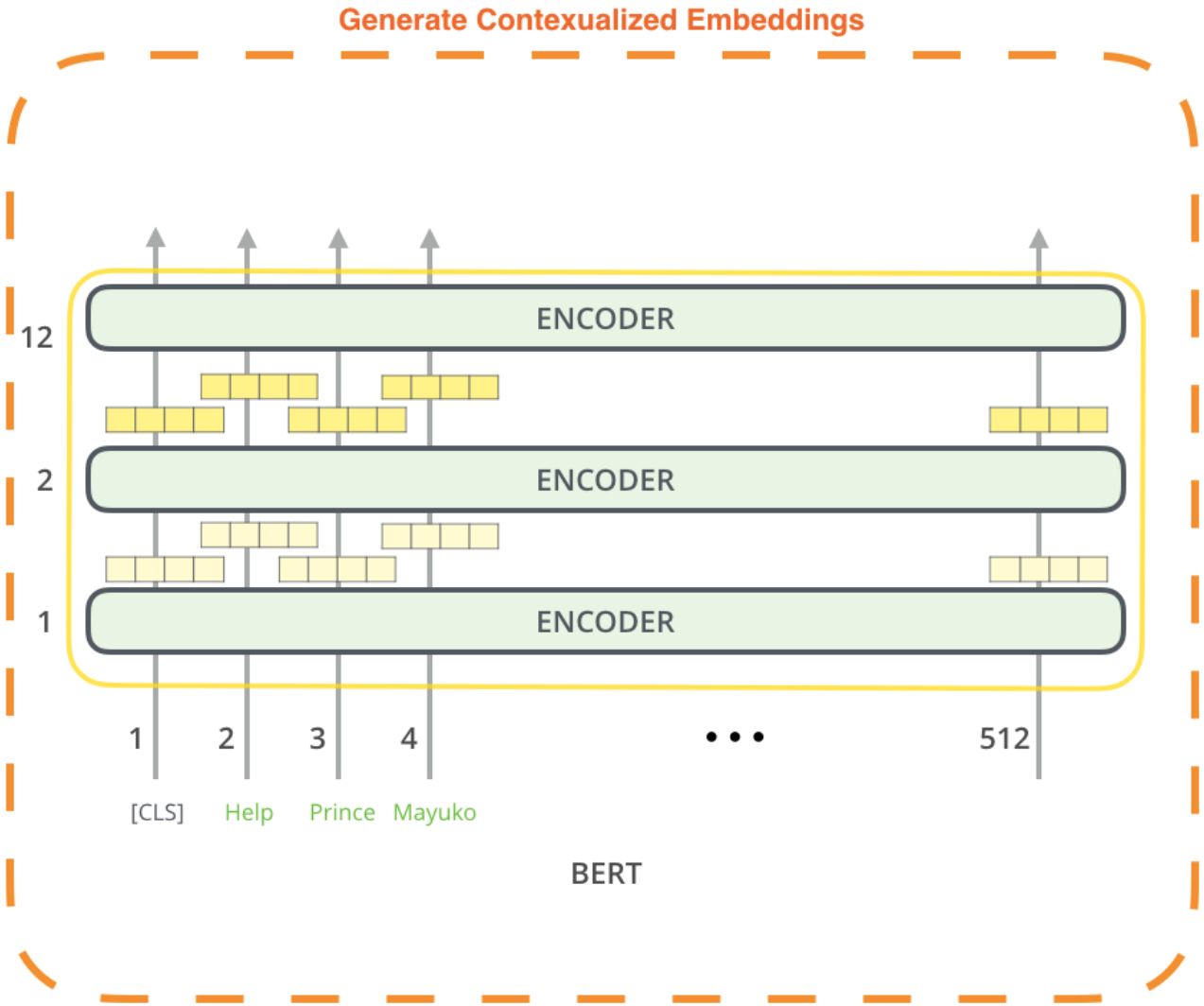


Figure 3.6 – Fine-tuning the pre-trained BERT model for text classification

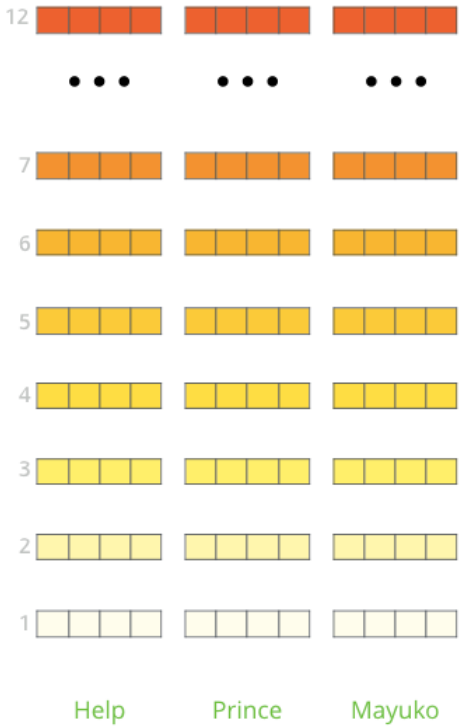
Fine-tuning is adding a layer of untrained neurons as a feedforward layer on top of the pre-trained BERT.

BERT for feature extraction(BERT Embedding)

The fine-tuning approach isn't the only way to use BERT. Just like ELMo, you can use the pre-trained BERT to create contextualized word embeddings. Then you can feed these embeddings to your existing model



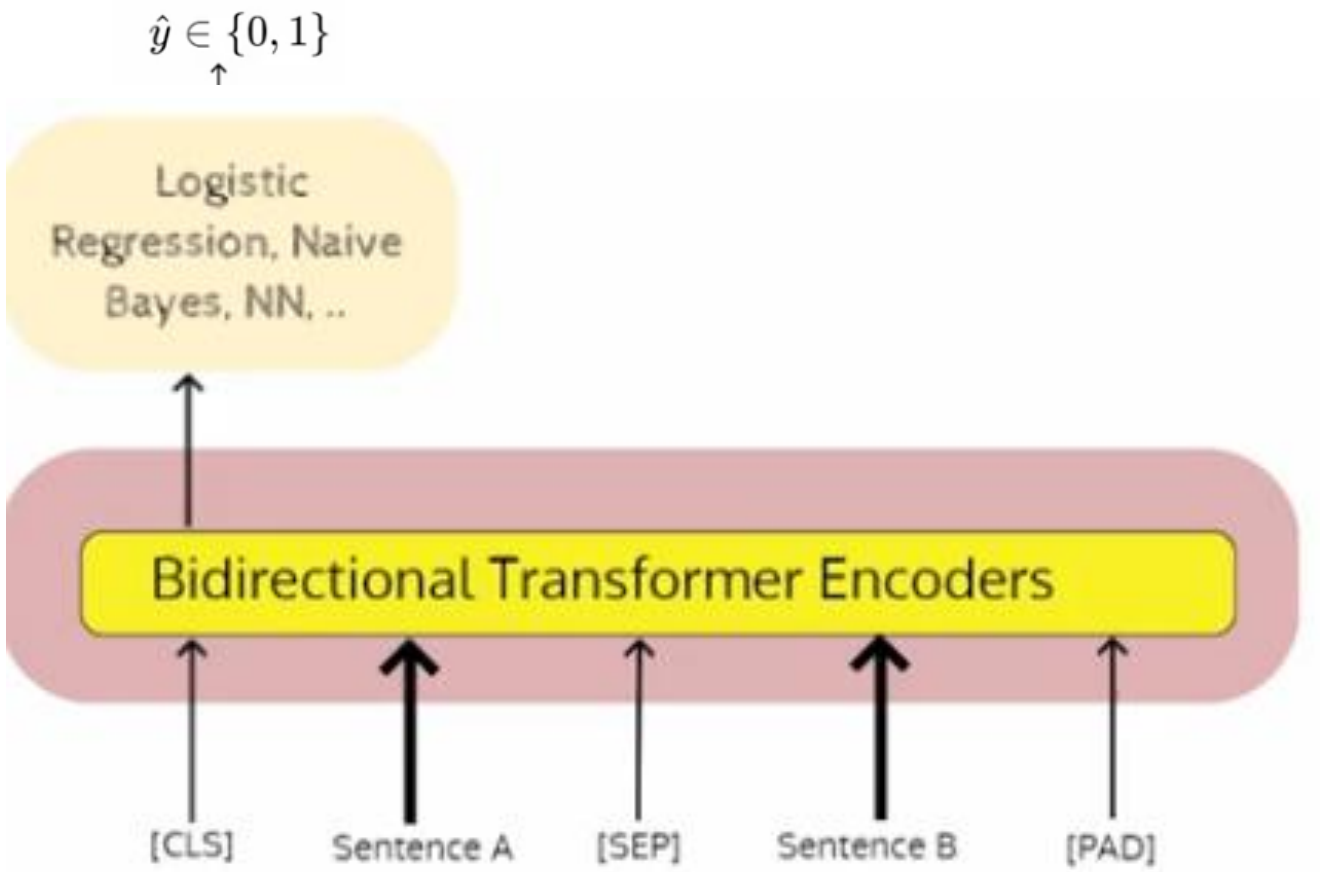
The output of each encoder layer along each token's path can be used as a feature representing that token.



- Context Awareness
- Bidirectional Learning
- Handles OOV Words
- Sentence-Level Understanding

But which one should we use?

BERT as a feature Extractor



Subword tokenization algorithms

- Subword tokenization is popularly used in many state-of-the-art natural language models, including BERT and GPT-3. It is very effective in handling OOV words.

```
vocabulary = [game, the, I, played, walked, enjoy]
```

Let's consider an input sentence ***"I played the game".*** In order to create tokens from the sentence, first, we split the sentence by white space and obtain all the words in the sentence.

Since all the words are present in the vocabulary, our final tokens for the given sentence will be as follows:

```
tokens = [I, played, the, game]
```

Let's consider another sentence***: "I enjoyed the game".*** To tokenize the sentence, we split the given sentence by white space and obtain the words. Then we have ***[I, enjoyed, the, game].*** Now, we check whether we have all the words (I, enjoyed, the, game) present in the vocabulary. We can observe that we have all the words present in the vocabulary except for the word enjoyed. Since the word enjoyed is not present in the vocabulary, we replace it with an unknown token, ***<UNK>.*** Thus, our final tokens will be as follows:

```
tokens = [ I, <UNK>, the, game]
```

In subword tokenization, we split the words into subwords. Suppose we split the word ***played*** into subwords ***[play, ed]*** and the word ***walked*** into subwords ***[walk, ed].*** After splitting the subwords, we will add them to the vocabulary. Note that the vocabulary consists of only unique words. So, our vocabulary now consists of the following:

```
vocabulary = [game, the, I, play, walk, ed, enjoy]
```

Let's consider the same sentence we saw earlier: ***"I enjoyed the game".*** To tokenize the sentence, we split the given sentence by white space and obtain the words. So we have ***[I, enjoyed, the, game].*** Now, we check whether we have all the words ***(I, enjoyed, the, game)*** present in the vocabulary.

We can notice that we have all the words present in the vocabulary except for the word ***enjoyed***. Since the word ***enjoyed*** is not present in the vocabulary, we split it into subwords ***[enjoy, ed].*** Now, we check whether we have the subwords ***enjoy*** and ***ed*** present in the vocabulary, and since they are present in the vocabulary, our tokens become the following:

```
tokens = [ I, enjoy, ##ed, the, game]
```

The ***##*** signs are added just to indicate that it is a subword and that it is preceded by another word.

In this way, subword tokenization handles the unknown words, that is, words that are not present in the vocabulary.

Subword tokenization algorithms

Subword tokenization algorithms that are used to create the vocabulary. After creating the vocabulary, we can use it for tokenization. Let's understand the following three popularly used subword tokenization algorithms:

- **Byte pair encoding**
- **Byte-level byte pair encoding**
- **WordPiece**

Byte Pair Encoding

Byte pair encoding

Let's understand how **Byte Pair Encoding (BPE)** works with the help of an example. Let's suppose we have a dataset. First, we extract all the words from the dataset along with their count. Suppose the words extracted from the dataset along with the count are **(cost, 2), (best, 2), (menu, 1), (men, 1), and (camel, 1).**

Character sequence	Cost
C o s t	2
b e s t	2
m e n u	1
m e n	1
c a m e l	1

The steps involved in BPE are provided here:

1. Extract the words from the given dataset along with their count.
2. Define the vocabulary size.
3. Split the words into a character sequence.
4. Add all the unique characters in our character sequence to the vocabulary.
5. Select and merge the symbol pair that has a high frequency.
6. Repeat step 5 until the vocabulary size is reached.

- Vocabulary size decided as 14

Character sequence	Cost
C o s t	2
b e s t	2
m e n u	1
m e n	1
c a m e l	1

Vocabulary
a, b, c, e, l, m, n, o, s, t, u

Figure 2.20 – Creating a vocabulary with all unique characters

Character sequence	Cost
C o s t	2
b e s t	2
m e n u	1
m e n	1
c a m e l	1

Vocabulary
a, b, c, e, l, m, n, o, s, t, u

Figure 2.21 – Finding the most frequent symbol pair

Character sequence	Cost	Vocabulary
C o **st**	2	a, b, c, e, l, m, n, o, s, t, u ,**st**
b e **st**	2	
m e n u	1	
m e n	1	
c a m e l	1	

Figure 2.22 – Merging the symbols s and t

Character sequence	Cost	Vocabulary
C o st	2	a, b, c, e, l, m, n, o, s, t, u ,st
b e st	2	
m e n u	1	
m e n	1	
c a m e l	1	

Figure 2.23 – Finding the most frequent symbol pair

Character sequence	Cost
C o st	2
b e st	2
me n u	1
me n	1
c a me l	1

Vocabulary: a, b, c, e, l, m, n, o, s ,t , u, st, **me**

Figure 2.24 – Merging the symbols m and e

Character sequence	Cost
C o st	2
b e st	2
me n u	1
me n	1
c a me l	1

Vocabulary: a, b, c, e, l, m, n, o, s ,t , u, st, me

Figure 2.25 – Finding the most frequent symbol pair

- As we reached the vocab size 14, stopped

Character sequence	Cost	Vocabulary
C o st	2	a, b, c, e, l, m, n, o, s, t, u, st, me, men
b e st	2	
men u	1	
men	1	
c a me l	1	

Figure 2.26 – Merging the symbols me and n

Byte-level Byte Pair Encoding

Byte-level byte pair encoding (BBPE)

Byte-level byte pair encoding (BBPE) is another popularly used algorithm. It works very similarly to BPE, but instead of using a character-level sequence, it uses a **byte-level sequence**. Let's understand how BBPE works with the help of an example.

Let's suppose our input text consists of just the word **'best'**. We know that in BPE, we convert the word into a character sequence, so we will have the following:

Character sequence: b e s t

Whereas in BBPE, instead of converting the word to a character sequence, we convert it to a byte-level sequence. Hence, we convert the word **'best'** into a byte sequence:

Byte sequence: 62 65 73 74

represents a **Chinese word** (你好) in **UTF-8 byte encoding**.

Byte sequence: e4 bd a0 e5 a5 bd

UTF-8 (Unicode Transformation Format - 8-bit) is a **variable-length character encoding** that represents **Unicode characters** using **1 to 4 bytes**.

WordPiece Tokenisation

We don't merge symbol pairs based on frequency; instead, we merge them based on **likelihood**. First, we check the likelihood of the language model (which is trained on a given training set) for every symbol pair. Then we merge the symbol pair that has the highest likelihood. The **likelihood** of the symbol pair **s** and **t** is computed, as shown here:

$$\frac{p(st)}{p(s)p(t)}$$

Likelihood = How probable the merged token is in a language model compared to individual tokens.

If the likelihood is high, we simply merge the symbol pair and add them to the vocabulary. In this way, we compute the likelihood of all symbol pairs and merge the one that has the maximum likelihood and add it to the vocabulary. The following steps help us to understand the algorithm better:

1. Extract the words from the given dataset along with their count.
2. Define the vocabulary size.
3. Split the words into a character sequence.
4. Add all the unique characters in our character sequence to the vocabulary.
5. Build the language model on the given dataset (training set).
6. Select and merge the symbol pair that has the maximum likelihood of the language model trained on the training set.
7. Repeat **step 6** until the vocabulary size is reached.

Tokenization in BERT: WordPiece

Tokenization in GPT: Byte Pair Encoding (BPE)

Why WordPiece for BERT?

1. **BERT is an Encoder-Based Model (Bidirectional Context)**
    - Since BERT **processes full sentences at once**, it benefits from **efficient vocabulary reduction**.
2. **Handles Out-of-Vocabulary (OOV) Issues**
    - Instead of assigning `<UNK>` tokens to unknown words, **WordPiece breaks them into known subwords**.
3. **Likelihood-Based Merging**
    - WordPiece merges subwords based on the probability of sequences appearing in training data, **improving semantic understanding**.

WordPiece prioritizes linguistic meaning while BPE primarily focuses on statistical frequency when merging subword units. Resulting pieces are more aligned with how words are used in context, making them more meaningful.

Why BPE for GPT?

1. **GPT is an Autoregressive Model (Generates One Token at a Time)**
    - GPT **predicts the next token**, so it benefits from BPE's **efficient compression** of common words.
2. **Memory Efficiency**
    - Byte-level BPE compresses rare words into shorter subwords, allowing **smaller vocabulary sizes**.
3. **Handles Multilingual Texts Better (Byte-Level BPE in GPT-3 & Later)**
    - Instead of using predefined subword units, **BBPE operates directly on byte sequences**, allowing better handling of **multiple languages and unseen words**.

Namah shivaya