

# Understanding BERT from Scratch

MLP Lab



# Contents

## 01 Introduction of BERT

---

## 02 How BERT works

---

## 03 BERT의 사전학습

---

### 1) BERT's word embedding layer

---

### 2) BERT's Pretraing

---



# 01 Introduction of BERT

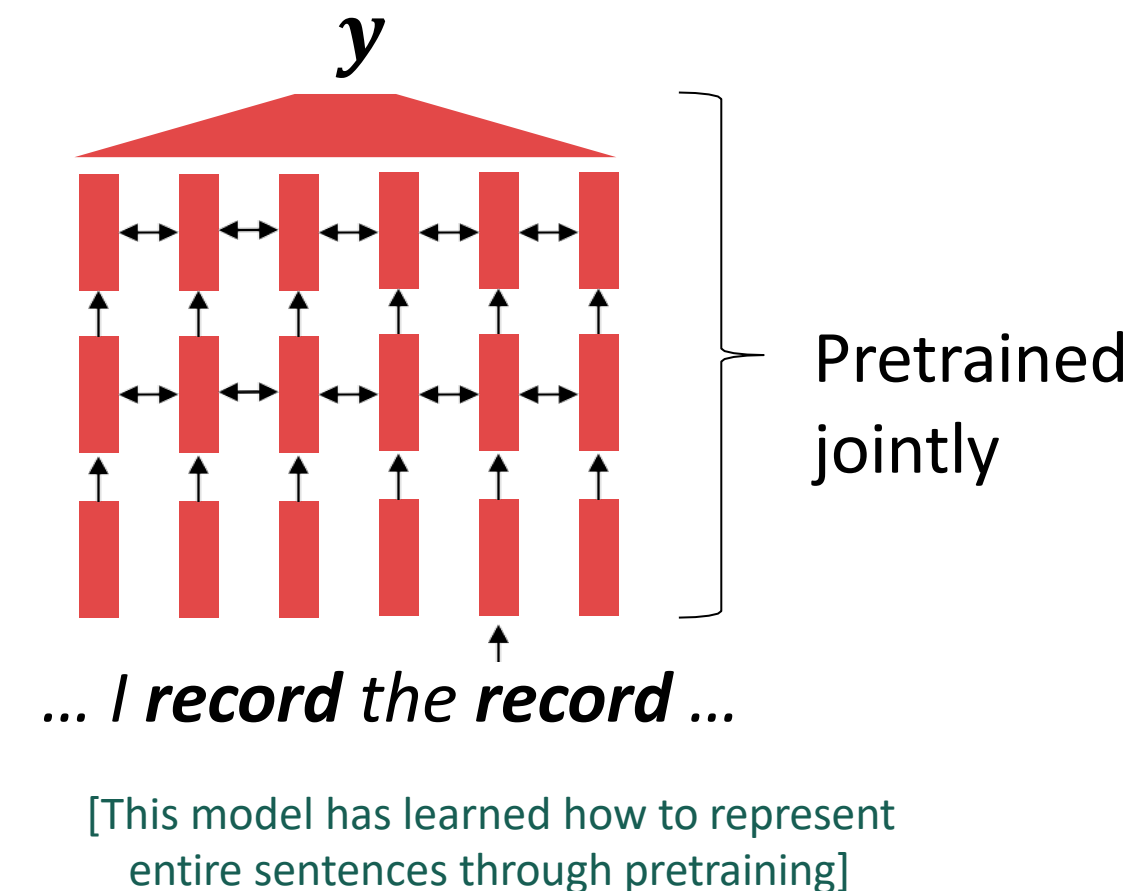
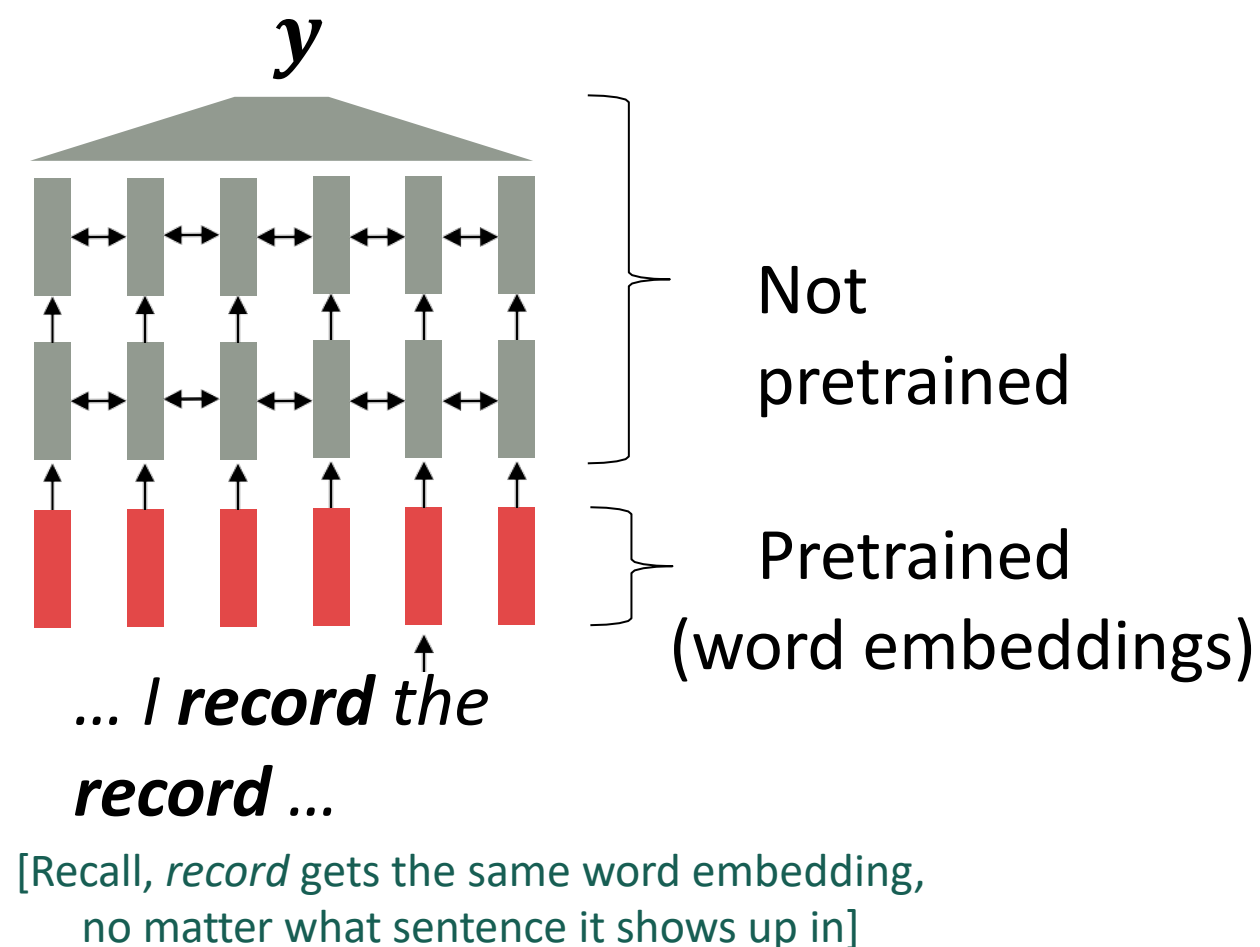
- BERT (Bidirectional Encoder Representations from Transformers) is literally a bi-directional representation of a transformer encoder.
- That is, a bidirectional **word representation** using only the encoder of the transformer.



What??  
Word representation??

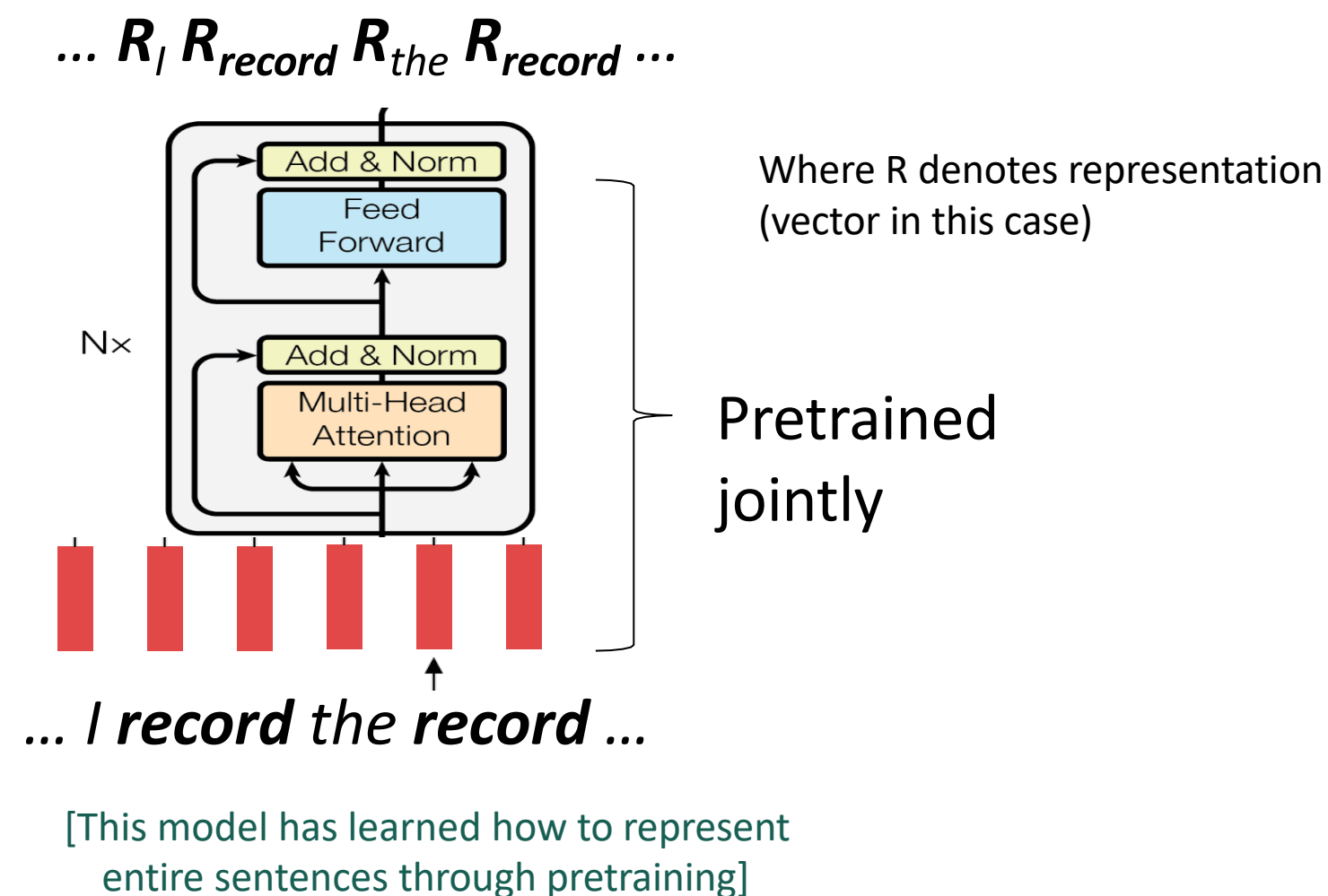
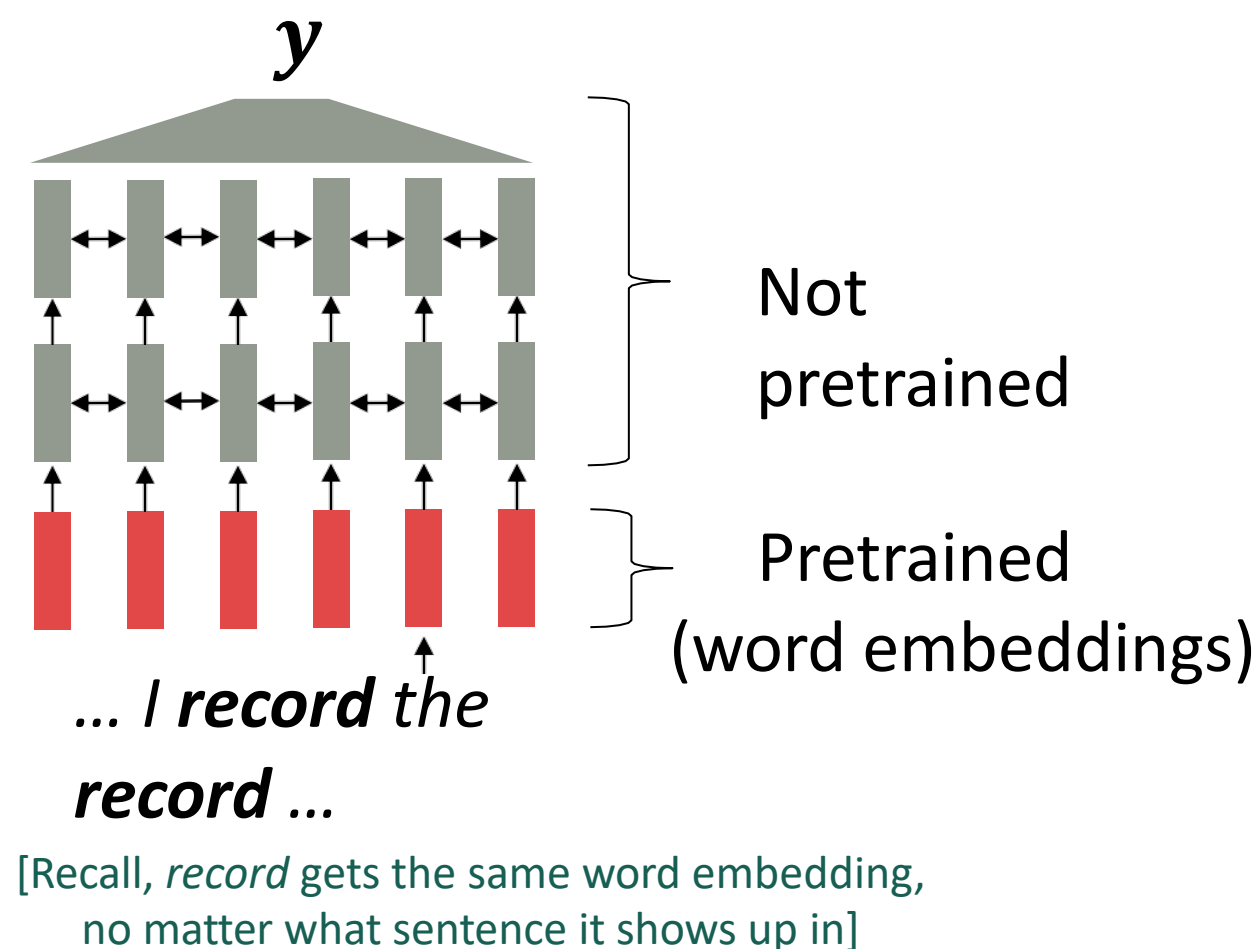
# 01 Introduction of BERT (Motivation)

- In order to build word embeddings as contextualized ones, our system should see the whole sentence with RNNs or Transformer.



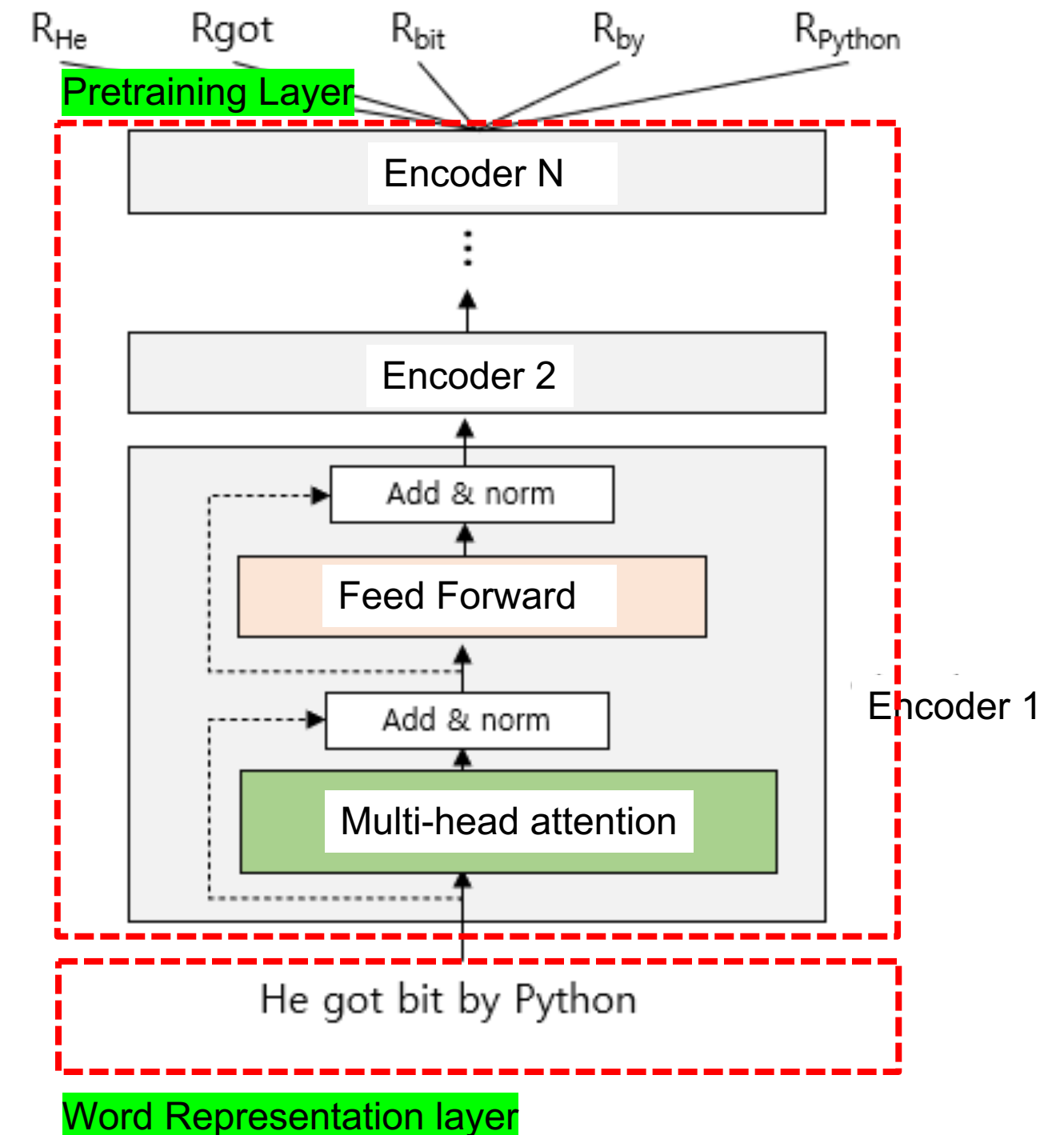
# 01 Introduction of BERT (Motivation)

- In order to build word embeddings as contextualized ones, our system should see the whole sentence with RNNs or Transformer.
- So the idea of **BERT** is to train **word embedding** with Transformer **Encoder** as **Pretraining**



## 02 How BERT works

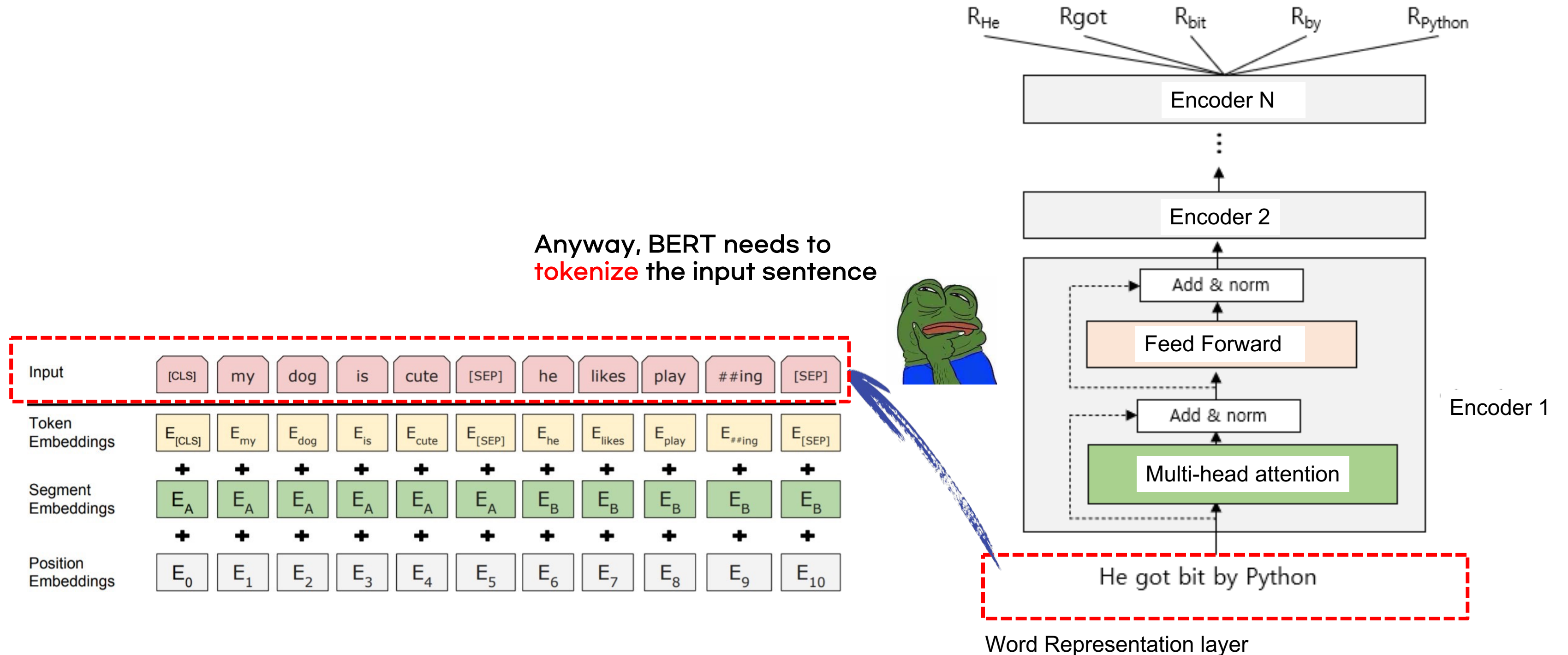
- Since the transformer's encoder can originally read sentences in both directions, BERT also reads sentences in both directions and outputs the representation.
- When **a sentence** comes in as input to the encoder, the encoder uses a multi-head attachment mechanism to **connect all the words** together to **determine** their **relationships** and context, and **outputs a representation of each word** in the sentence.
- There are two layers for training BERT
  - **Word Representation Layer**
  - **Pretraining Layer**





## 02 How BERT works

- Let's look deep inside of **word representation layer** of BERT



## 03 Pretraining BERT

- From word embedding layer in BERT, it tokenizes the input sentence with
  - (1) the WordPiece Tokenizer and then
  - (2) transforms it into the three embeddings below, which are provided as input to BERT
- Token embedding
- Segment embedding
- Position embedding



## 03 Pretraining BERT

- From word embedding layer in BERT, it tokenizes the input sentence with (1) the WordPiece Tokenizer and then

# 03 Pretraining BERT

- From word embedding layer in BERT, it tokenizes the input sentence with  
(1) the **WordPiece Tokenizer**: A sub-word tokenization algorithm
  - WordPiece tokenizer checks to see if the word is in the lexicon, if it is, it uses the word as a token, if not, it splits the word into subwords, checking to see if the subwords are in the lexicon, and so on.
  - This method of splitting into subwords until it reaches an individual character is effective for handling out of vocabulary (OOV) words.

Let's look at the same vocabulary we used in the BPE training example:

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

The splits here will be:

```
'##g', 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##g" "##s", 5)
```

so the initial vocabulary will be ["b", "h", "p", "##g", "##n", "##s", "##u"] (if we forget about special tokens for now). The most frequent pair is ("##u", "##g") (present 20 times), but the individual frequency of "##u" is very high, so its score is not the highest (it's 1 / 36). All pairs with a "##u" actually have that same score (1 / 36), so the best score goes to the pair ("##g", "##s") — the only one without a "##u" — at 1 / 20, and the first merge learned is ("##g", "##s") -> ("##gs").

Note that when we merge, we remove the ## between the two tokens, so we add "##gs" to the vocabulary and apply the merge in the words of the corpus:

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u", "##gs"]  
Corpus: ("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##g" "##s", 5)
```

But why we need to use a sub-word tokenizer??








## 03 Pretraining BERT

- **Why we need a sub-word tokenizer?**

Let's take a look at the assumptions we've made about a language's vocabulary.

We assume a fixed vocab of tens of thousands of words, built from the training set.

All *novel* words seen at test time are mapped to a single UNK.

	word		vocab mapping	embedding
Common words	hat	→	pizza (index)	
	learn	→	tasty (index)	
Variations	taaaaasty	→	UNK (index)	
misspellings	laern	→	UNK (index)	
novel items	Transformerify	→	UNK (index)	

## 03 Pretraining BERT

- **Why we need a sub-word tokenizer?**

Subword modeling in NLP encompasses a wide range of methods for reasoning about structure below the word level. (Parts of words, characters, bytes.)

- The dominant modern paradigm is to learn a vocabulary of **parts of words (subword tokens)**.
- At training and testing time, each word is split into a sequence of known subwords.

**Byte-pair encoding** is a simple, effective strategy for defining a subword vocabulary.

1. Start with a vocabulary containing only characters and an “end-of-word” symbol.
2. Using a corpus of text, find the most common adjacent characters “a,b”; add “ab” as a subword.
3. Replace instances of the character pair with the new subword; repeat until desired vocab size.

Originally used in NLP for machine translation; now a similar method (WordPiece) is used in pretrained models.











## 03 Pretraining BERT

- **Why we need a sub-word tokenizer?**

Common words end up being a part of the subword vocabulary, while rarer words are split into (sometimes intuitive, sometimes not) components.

In the worst case, words are split into as many subwords as they have characters.

	word		vocab mapping	embedding
Common words	hat	→	hat	
	learn	→	learn	
Variations	taaaaasty	→	taa## aaa## sty	  
misspellings	laern	→	la## ern##	 
novel items	Transformerify	→	Transformer## ify	 

## 03 Pretraining BERT

### BERT's tokenizer: WordPiece Tokenizer

- *'let us start pretraining the model'* When this sentence is tokenized with WordPiece Tokenizer, the result is as follows:

```
tokens
✓ 0.0s
['let', 'us', 'start', 'pre', '##train', '##ing', 'the', 'model']
```

- One word “*pretraining*” split into subwords like “pre”, “##train”, “##ing”

## 03 Pretraining BERT

- **BERT's tokenizer: WordPiece Tokenizer**
- Because BERT can receive multiple sentences as input, it adds a [CLS] token at the beginning of the sentence and a [SEP] token at the end.

tokens

✓ 0.0s

```
['[CLS]', 'let', 'us', 'start', 'pre', '##train', '##ing', 'the', 'model', '[SEP]']
```



## 03 Pretraining BERT

- **BERT's tokenizer: WordPiece Tokenizer**
- For two sentences
  - Sentence A: Paris is a beautiful city
  - Sentence B: I love Paris

tokens

✓ 0.0s

```
['Paris', 'is', 'a', 'beautiful', 'city', 'I', 'love', 'Paris']
```

- Add a [CLS] token and a [SEP] token at the end of each sentence

tokens

✓ 0.0s

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

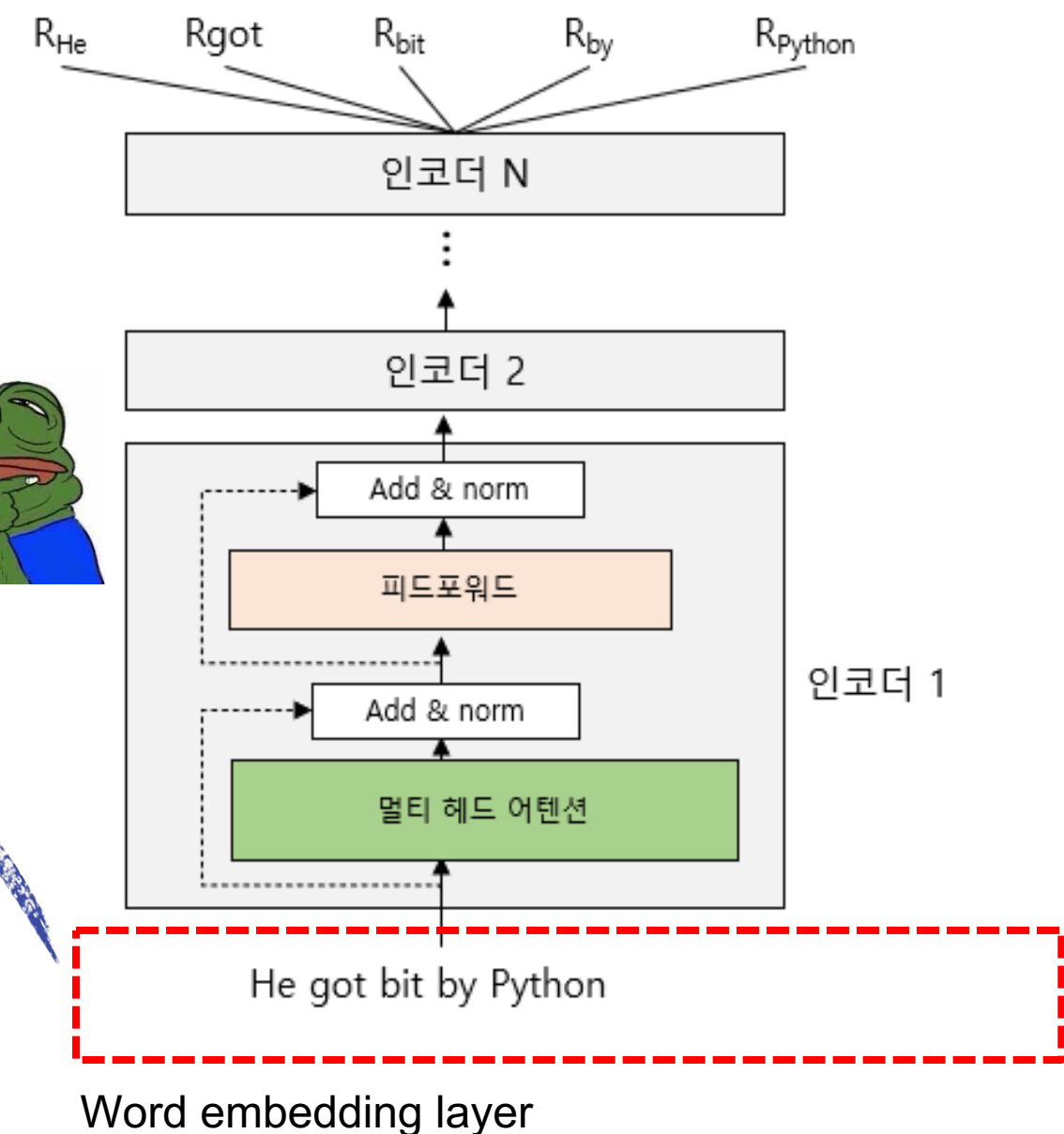
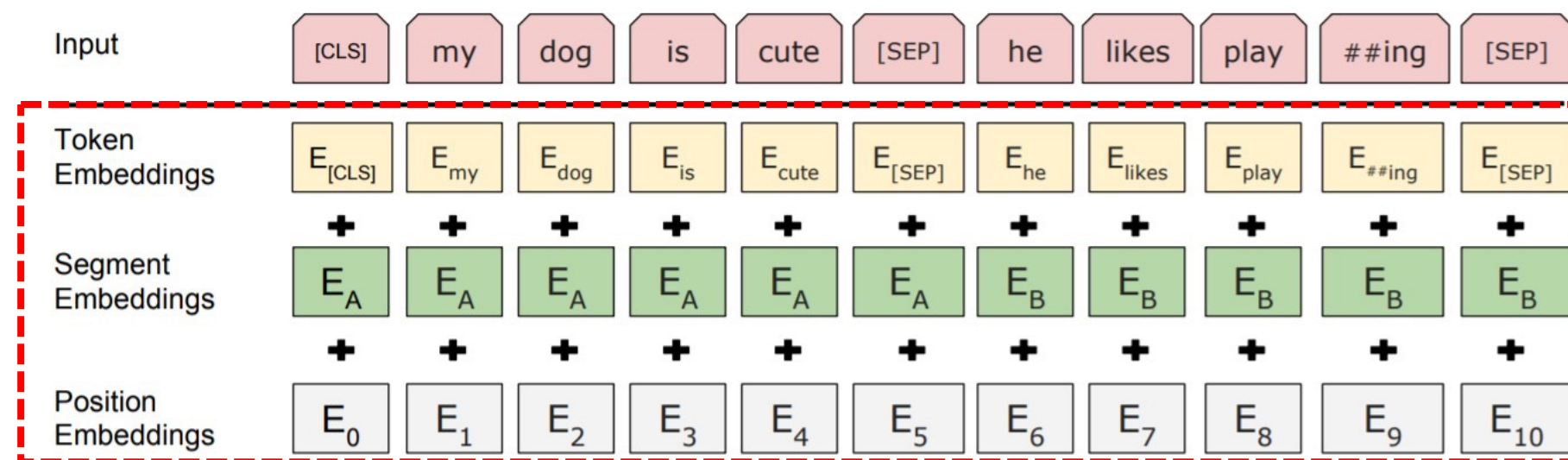
## 03 Pretraining BERT

- From word embedding layer in BERT, it tokenizes the input sentence with
  - (1) the WordPiece Tokenizer and then
  - (2) transforms it into the three embeddings below, which are provided as input to BERT
    - Token embedding
    - Segment embedding
    - Position embedding

# 03 Pretraining BERT

- From word embedding layer in BERT, it tokenizes the input sentence with  
(2) transforms it into the three embeddings below, which are provided as input to BERT
- Token embedding
- Segment embedding
- Position embedding

Anyway, BERT needs **embeddings** from the input sentence



## 03 Pretraining BERT

### 1. BERT's word embedding layer

- **Token embedding:** Convert token to embedding

Input	[CLS]	Paris	is	a	beautiful	city	[SEP]	I	love	Paris	[SEP]
Token embeddings	$E_{\text{CLS}}$	$E_{\text{Paris}}$	$E_{\text{is}}$	$E_{\text{a}}$	$E_{\text{beautiful}}$	$E_{\text{city}}$	$E_{\text{[SEP]}}$	$E_{\text{I}}$	$E_{\text{love}}$	$E_{\text{Paris}}$	$E_{\text{[SEP]}}$

- Token embedding is trained in the pretraining step

- **Segment embedding:** Used to distinguish between two given sentences
- The token corresponding to sentence A has the value  $E_A$  as its segment embedding, and the token corresponding to sentence B has the value  $E_B$  as its segment embedding.

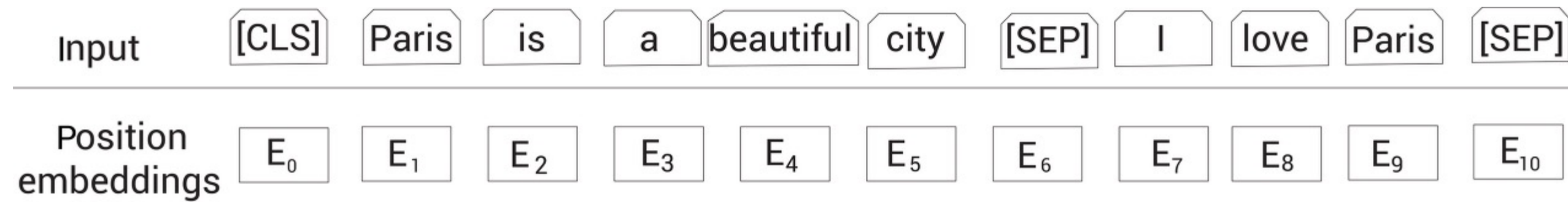
- If the input data is a single sentence, all tokens have the same segment embedding value

[illegible]

## 03 Pretraining BERT

### 1. BERT's word embedding layer

- **Position embedding:** Transformers process all words in parallel, so you need to provide information about the word order.



- ✂ Unlike transformers, BERT uses Learned Positional Embeddings to provide positional information instead of using a sine wave function.

# 03 Pretraining BERT

## 1. BERT’s word embedding layer

- **Final Word Embedding:** The final input representation for BERT takes a given sentence, tokenizes it with WordPiece Tokenizer, finds three embedding values for each token, **sums** them up, and feeds them to BERT (Encoder of Transformer) as input.

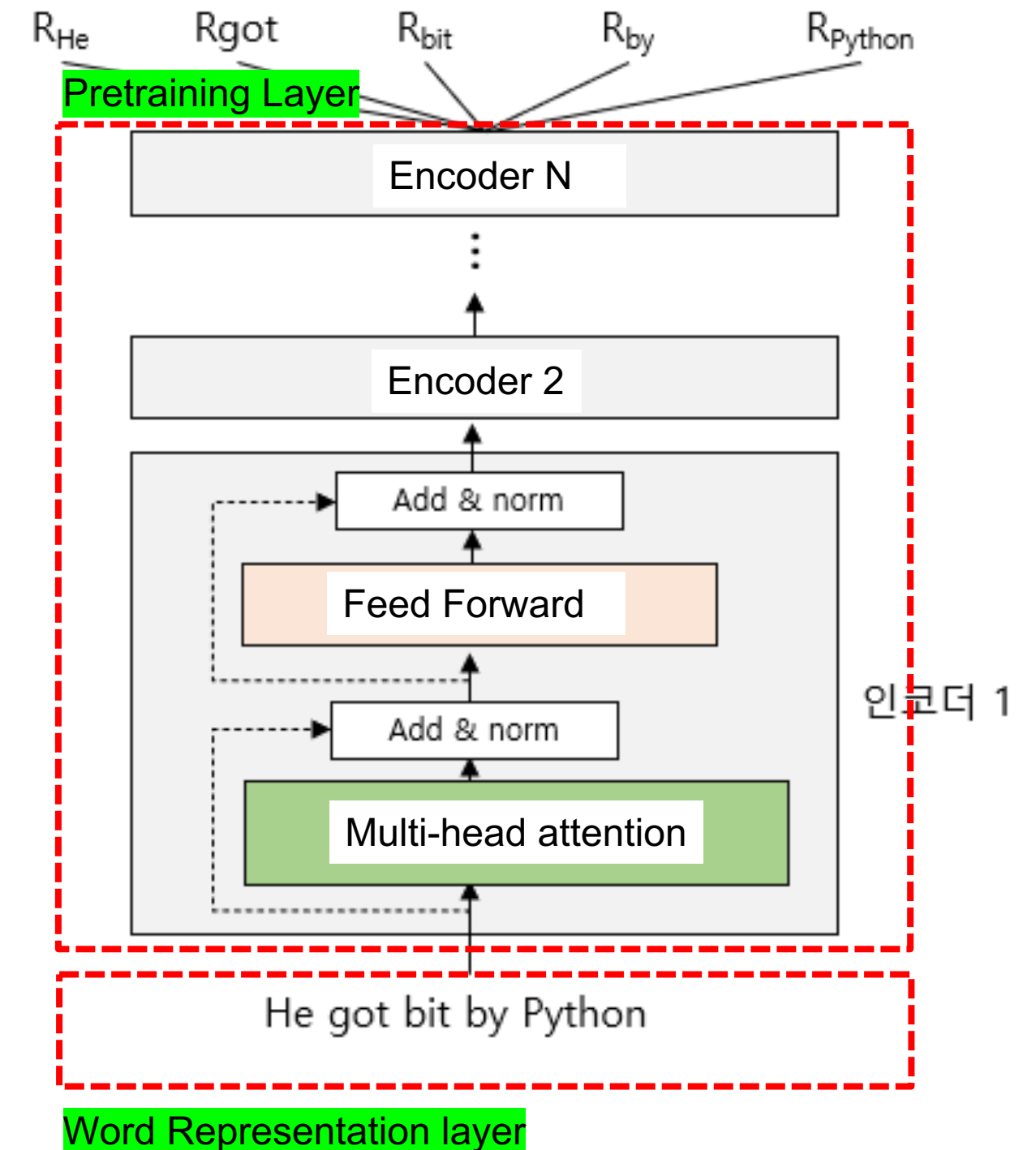
Input	[CLS]	Paris	is	a	beautiful	city	[SEP]	I	love	Paris	[SEP]
Token embeddings	<div>E<sub>[CLS]</sub></div> <div>+</div>	<div>E<sub>Paris</sub></div> <div>+</div>	<div>E<sub>is</sub></div> <div>+</div>	<div>E<sub>a</sub></div> <div>+</div>	<div>E<sub>beautiful</sub></div> <div>+</div>	<div>E<sub>city</sub></div> <div>+</div>	<div>E<sub>[SEP]</sub></div> <div>+</div>	<div>E<sub>I</sub></div> <div>+</div>	<div>E<sub>love</sub></div> <div>+</div>	<div>E<sub>Paris</sub></div> <div>+</div>	<div>E<sub>[SEP]</sub></div> <div>+</div>
Segment embeddings	<div>E<sub>A</sub></div> <div>+</div>	<div>E<sub>A</sub></div> <div>+</div>	<div>E<sub>A</sub></div> <div>+</div>	<div>E<sub>A</sub></div> <div>+</div>	<div>E<sub>A</sub></div> <div>+</div>	<div>E<sub>A</sub></div> <div>+</div>	<div>E<sub>A</sub></div> <div>+</div>	<div>E<sub>B</sub></div> <div>+</div>	<div>E<sub>B</sub></div> <div>+</div>	<div>E<sub>B</sub></div> <div>+</div>	<div>E<sub>B</sub></div> <div>+</div>
Position embeddings	<div>E<sub>0</sub></div>	<div>E<sub>1</sub></div>	<div>E<sub>2</sub></div>	<div>E<sub>3</sub></div>	<div>E<sub>4</sub></div>	<div>E<sub>5</sub></div>	<div>E<sub>6</sub></div>	<div>E<sub>7</sub></div>	<div>E<sub>8</sub></div>	<div>E<sub>9</sub></div>	<div>E<sub>10</sub></div>



## 03 Pretraining BERT

### 2. BERT's pretraining layer

- BERT uses the Toronto BookCorpus and English Wikipedia datasets for pretraining on the following two tasks:
  - MLM (Masked Language Modeling)
  - NSP (Next Sentence Prediction)

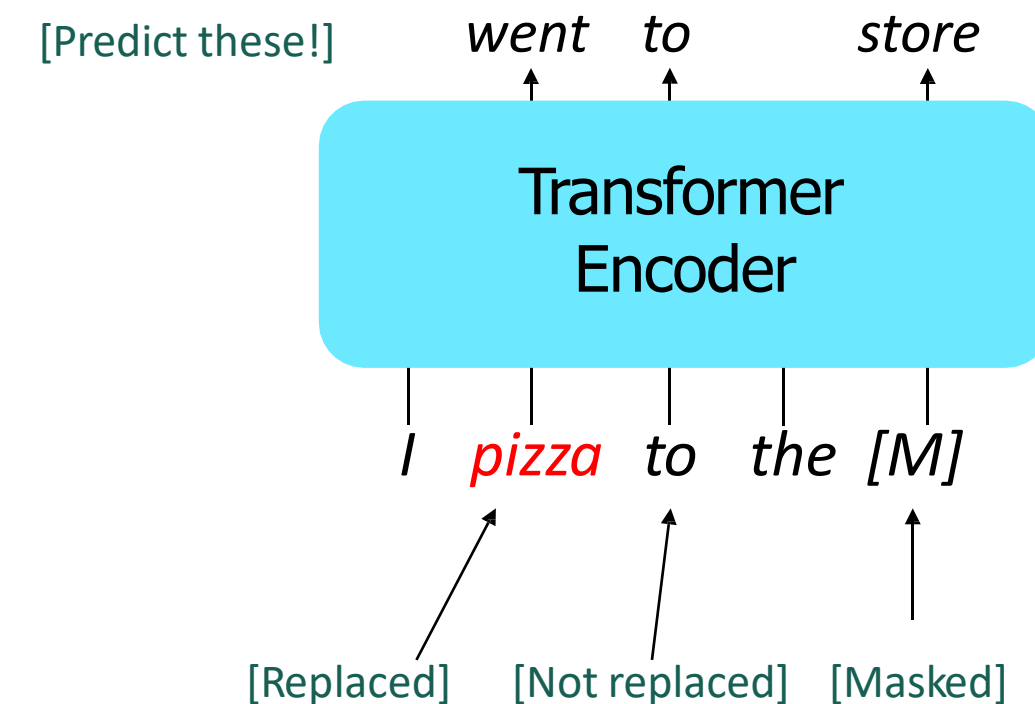


## 03 Pretraining BERT

### 2. BERT's pretraining layer

- **MLM, (Masked Language Modeling): Prediction of hidden words**

- Predict a random 15% of (sub)word tokens.
  - Replace input word with [MASK] 80% of the time
  - Replace input word with a random token 10% of the time
  - Leave input word unchanged 10% of the time (but still predict it!)
- Why? Doesn't let the model get complacent and not build strong representations of non-masked words. (No masks are seen at fine-tuning time!)



[Devlin et al., 2018]

## 03 Pretraining BERT

### 2. BERT's pretraining layer

- **MLM, (Masked Language Modeling): Prediction of hidden words**
- MLM randomly masks 15% of all words in a given input sentence and trains the model to predict the masked words.

tokens

✓ 0.0s

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

- After tokenization as above, add [CLS] and [SEP] tokens and randomly mask 15% of the tokens.

tokens

✓ 0.0s

```
['[CLS]', 'paris', 'is', 'a', 'beautiful', '[MASK]', '[SEP]', 'i', 'love', 'paris', '[SEP]']
```

- 'city' token is masked and replaced with '[MASK]' token

## 03 Pretraining BERT

### 2. BERT's pretraining layer

**MLM** (Masked Language Modeling): Prediction of hidden words

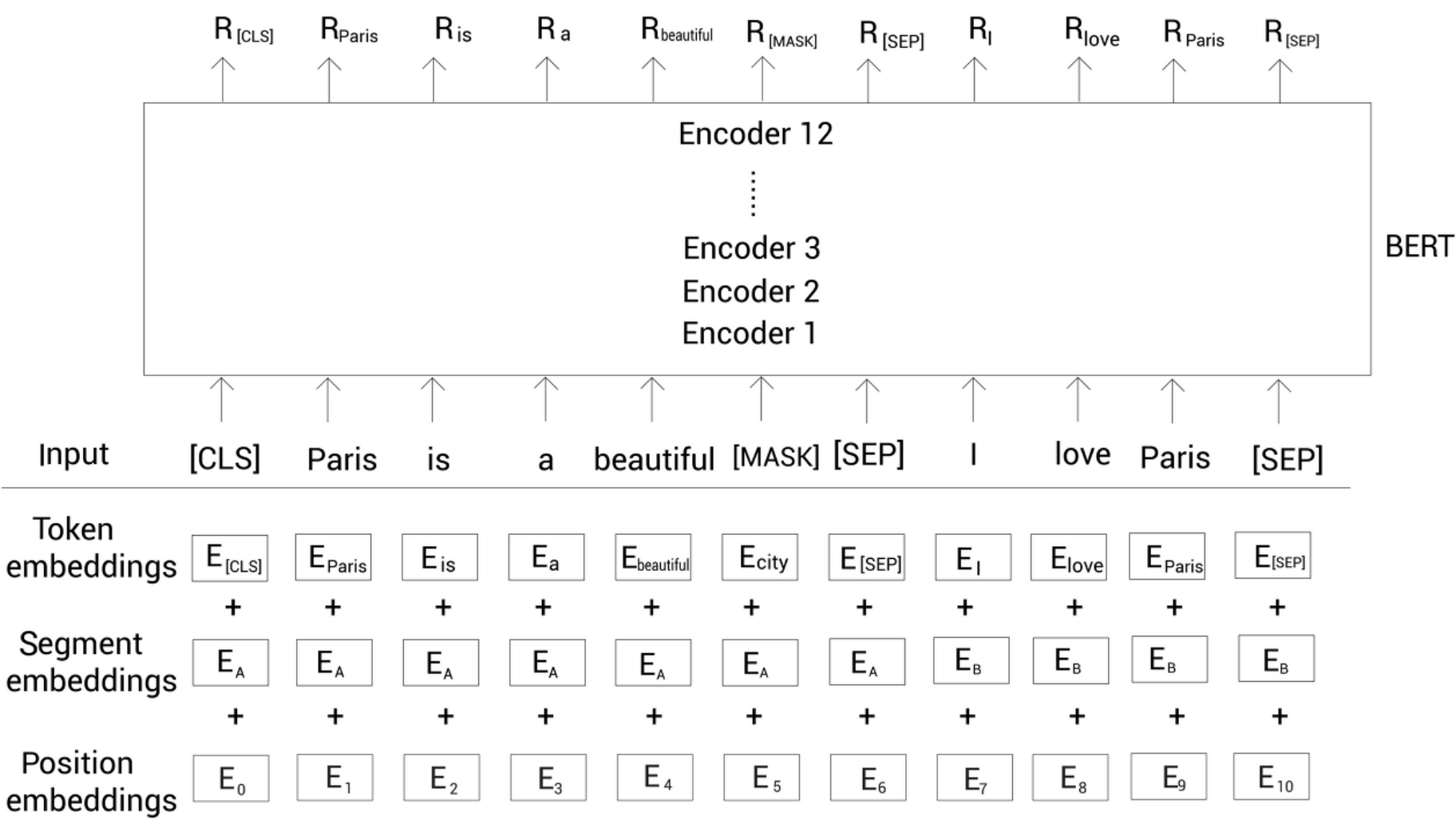
- This approach leads to mismatches between pre-training and downstream tasks.
- For example, when you fine-tune a pre-trained BERT for a sentiment analysis task, the data entered for fine-tuning has discrepancies that do not have a [MASK] token.
- To solve this problem, we introduced the 80-10-10% rule
  - Replace 80% of the 15% with [MASK] tokens
  - Replace 10% of the 15% with a random token (a random word)
  - Make no changes to 10% of the 15%.

# 03 Pretraining BERT

## 2. BERT’s pretraining layer

**MLM** (Masked Language Modeling): Prediction of hidden words

- Once the masking is done, get the input embedding in the embedding layer and give it to BERT

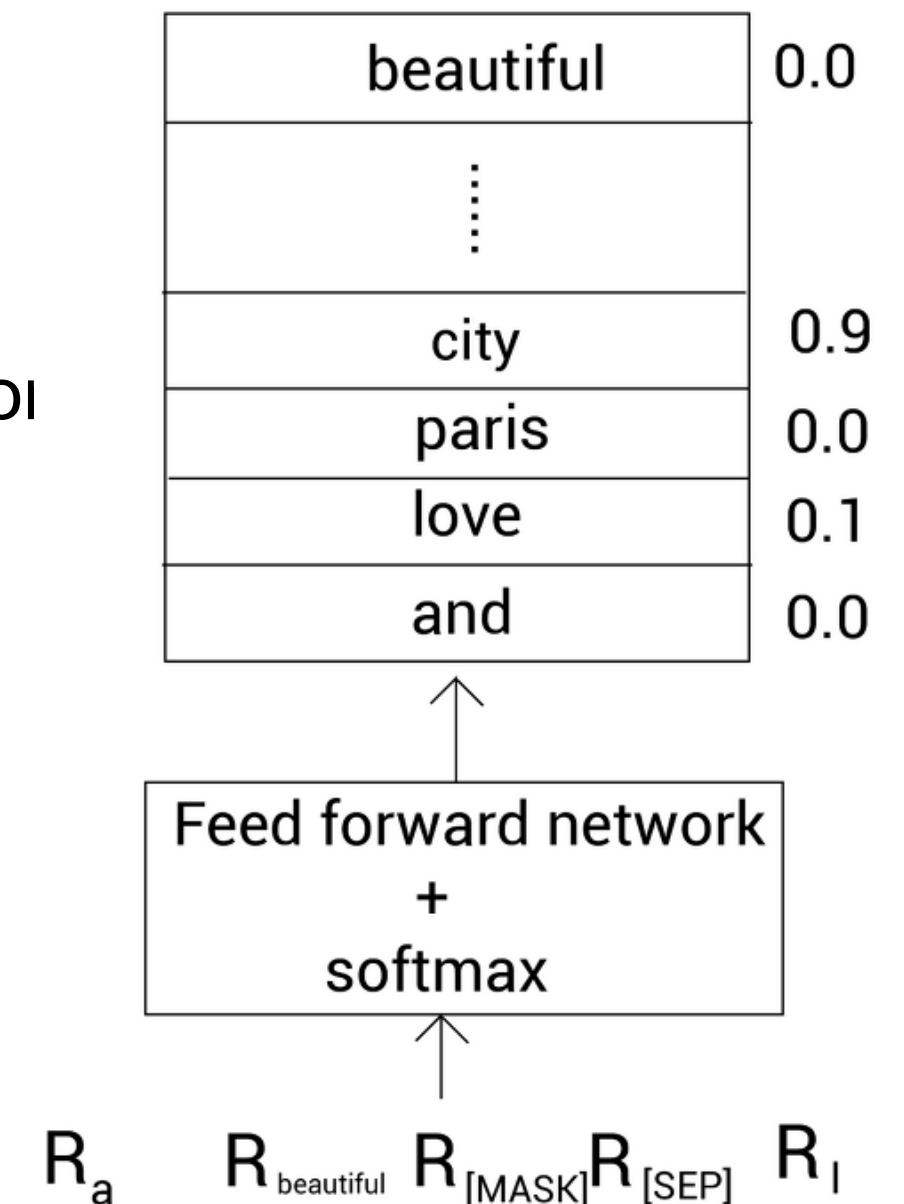


## 03 Pretraining BERT

### 2. BERT's pretraining layer

**MLM** (Masked Language Modeling): Prediction of hidden words

- Once you have the representation of each token ( $R$ ), feed the representation of the masked token  $R_{[MASK]}$  into a feed-forward network (FFN) using the softmax function to predict the masked token.
- At the beginning of learning, the weights of FFN and encoder are not optimal, so the weights are updated by iterative learning through backpropagation to learn the optimal weights.
- MLM Task is also known as a cloze task



# 03 Pretraining BERT

## 2. BERT’s pretraining layer

### NSP (Next Sentence Prediction)

- Given sentences A and B, a binary classification task to predict whether B is the next sentence after A.

문장 쌍	레이블
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	NotNtext



## 03 Pretraining BERT

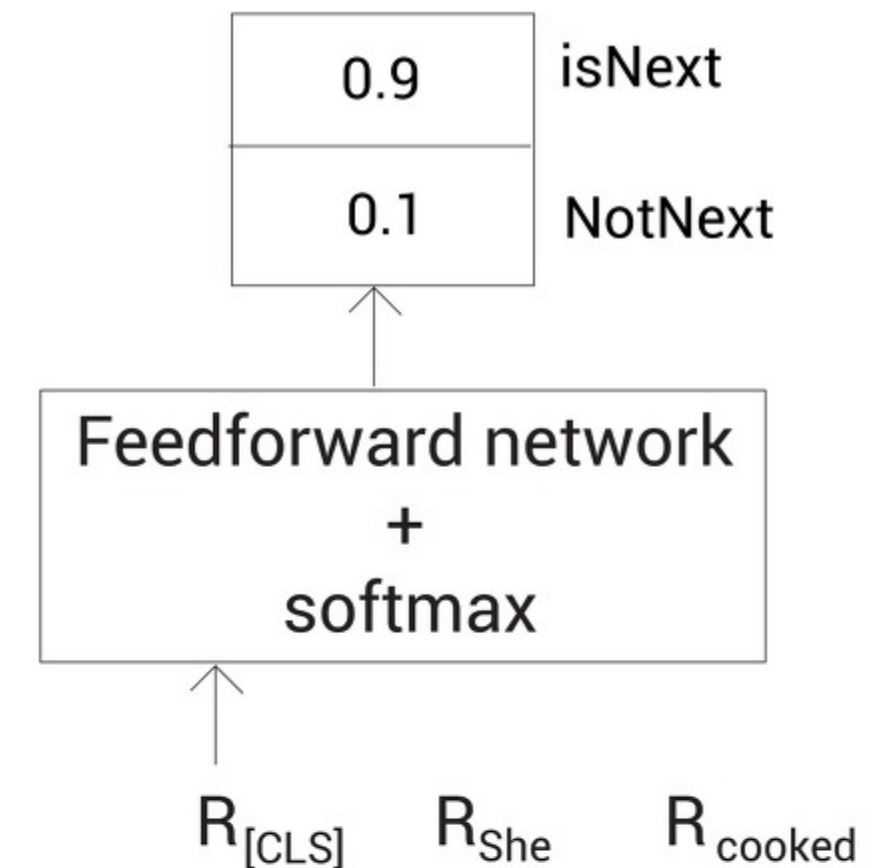
### 2. BERT's pretraining layer

#### NSP (Next Sentence Prediction)

- Obtain the embedding of sentences A, B.
- Feed the representation of the [CLS] token into FFN, softmax for binary classification

Why use only [CLS] tokens?

- ✂ The [CLS] token holds the aggregate representation of all tokens by default, so it holds the representation of the entire sentence



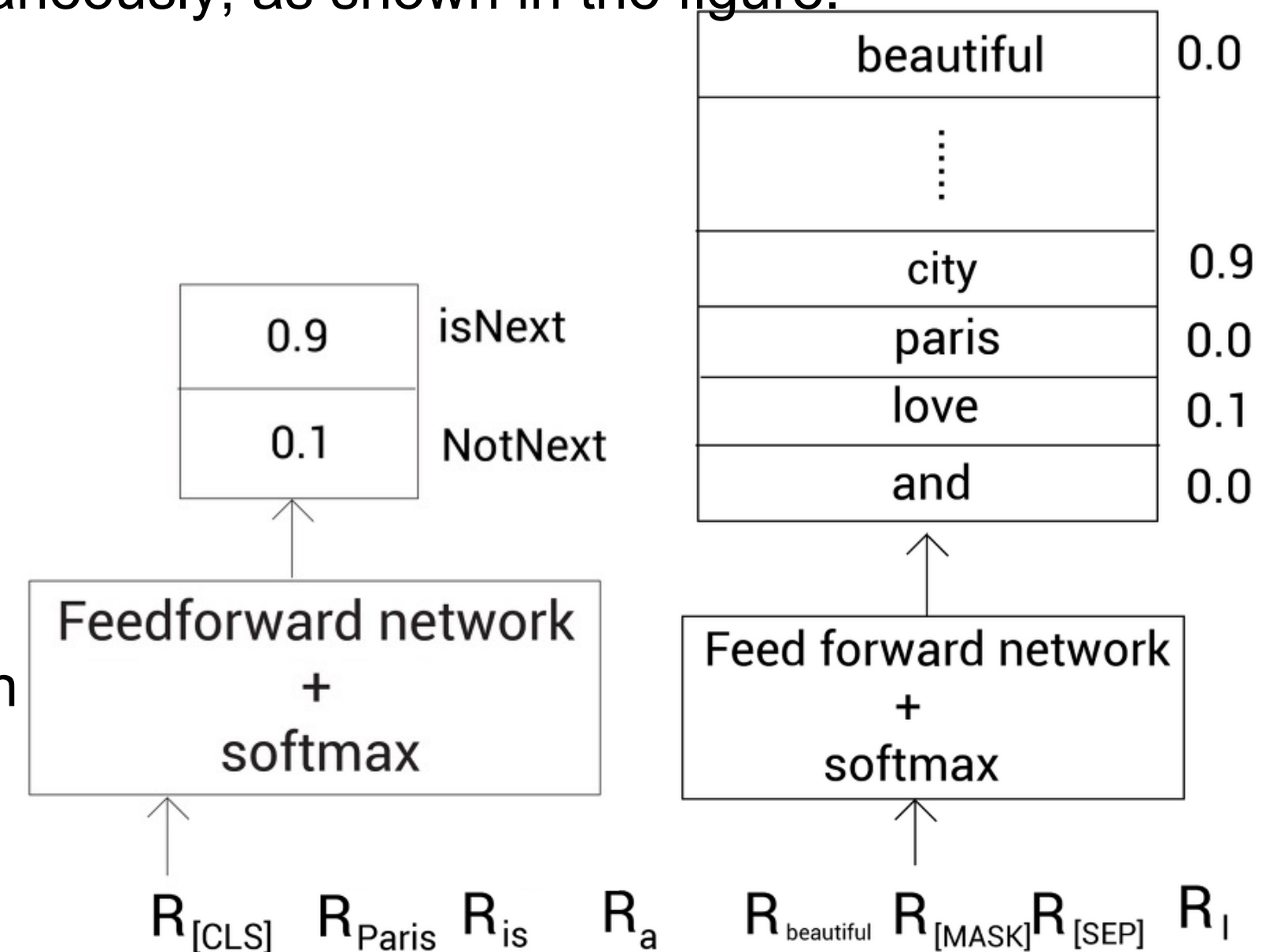
# 03 Pretraining BERT

## 2. BERT's pretraining layer

### NSP (Next Sentence Prediction)

- Pretrain BERT using MLM and NSP tasks simultaneously, as shown in the figure.

- Batch size : 256, 1,000,000 steps, Adam optimizer, lr : 1e-4,  $\beta_1$  : 0.9,  $\beta_2$  : 0.999
- Use a warm-up step with a high learning rate in the beginning and a low learning rate in the end
- Apply dropout to all layers with a dropout probability of 0.1 and use GELU for the activation function



**Thank you**