Natural Language Processing
Course code: CSE3015

Module 6
NLP latest Techniques and applications

Prepared by

Dr. Venkata Rami Reddy Ch

SCOPE

Syllabus

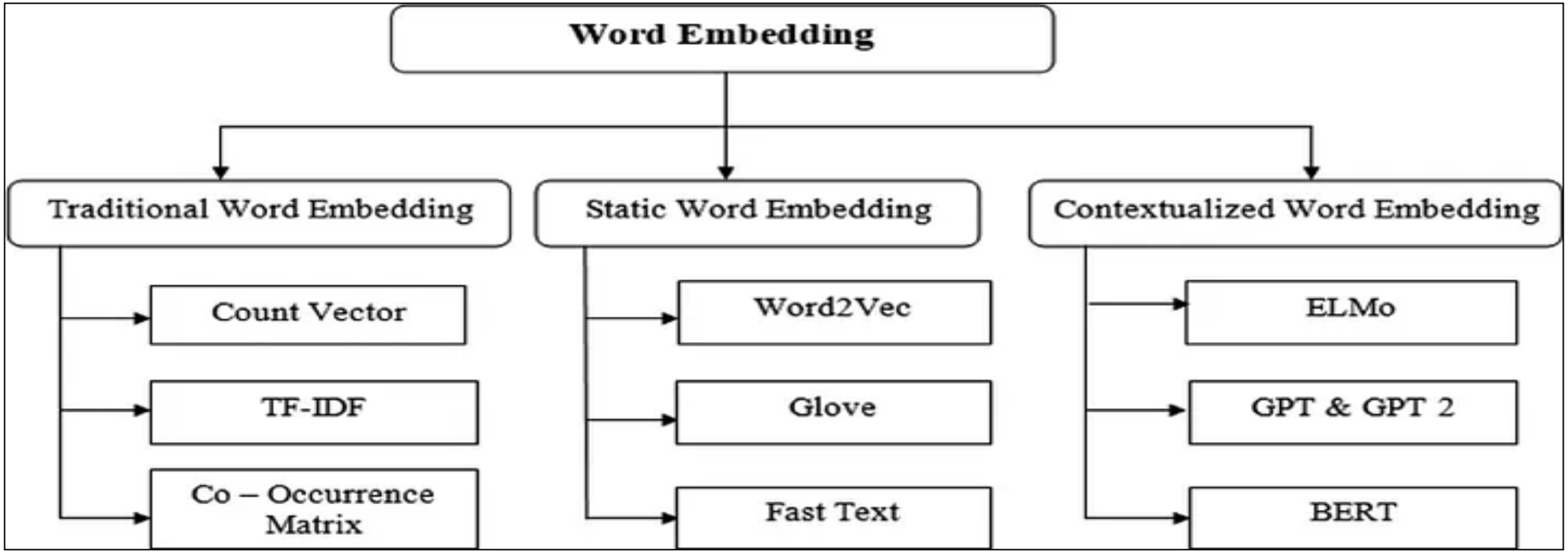
Contextualized word embeddings: ELMo, BERT, GPT,

Pre-trained Language Models (PLMs): BERT, GPT, ELMo, Large Language Models (LLMs),

Applications of NLP: Sentiment Analysis, Information retrieval, Question and Answering

Word embeddings

- Word embeddings in NLP are a type of word representation where words or phrases are mapped to numerical vectors in a continuous vector space.
- These vectors capture semantic and syntactic meanings of words such that similar words (in meaning or context) are represented by similar vectors.



Static Embeddings

- **Static embeddings** are a class of word embeddings in NLP where each word is represented by a fixed vector, regardless of the context in which it is used.
- Each word has one vector that remains the same across all contexts. For example, the word "bank" would have the same embedding whether it refers to a financial institution or a riverbank.

Models:

- **Word2Vec**
- **GloVe**
- **FastText**

Ex: Consider the word **"bank":**

"I deposited money in the bank" → bank_vector = [0.2, -0.5, 0.8, …]
"I sat by the river bank" → bank_vector = [0.2, -0.5, 0.8, …]
In static embeddings, both instances of "bank" would have the same vector, despite their different meanings.

No Context Awareness

- Each word has **only one meaning**, no matter the sentence.
- ➤ "bat" in "cricket bat" and "bat flew" = same vector (which is wrong).

Cannot Handle Polysemy

- Polysemy = words with multiple meanings.
- Static embeddings treat them as the same.

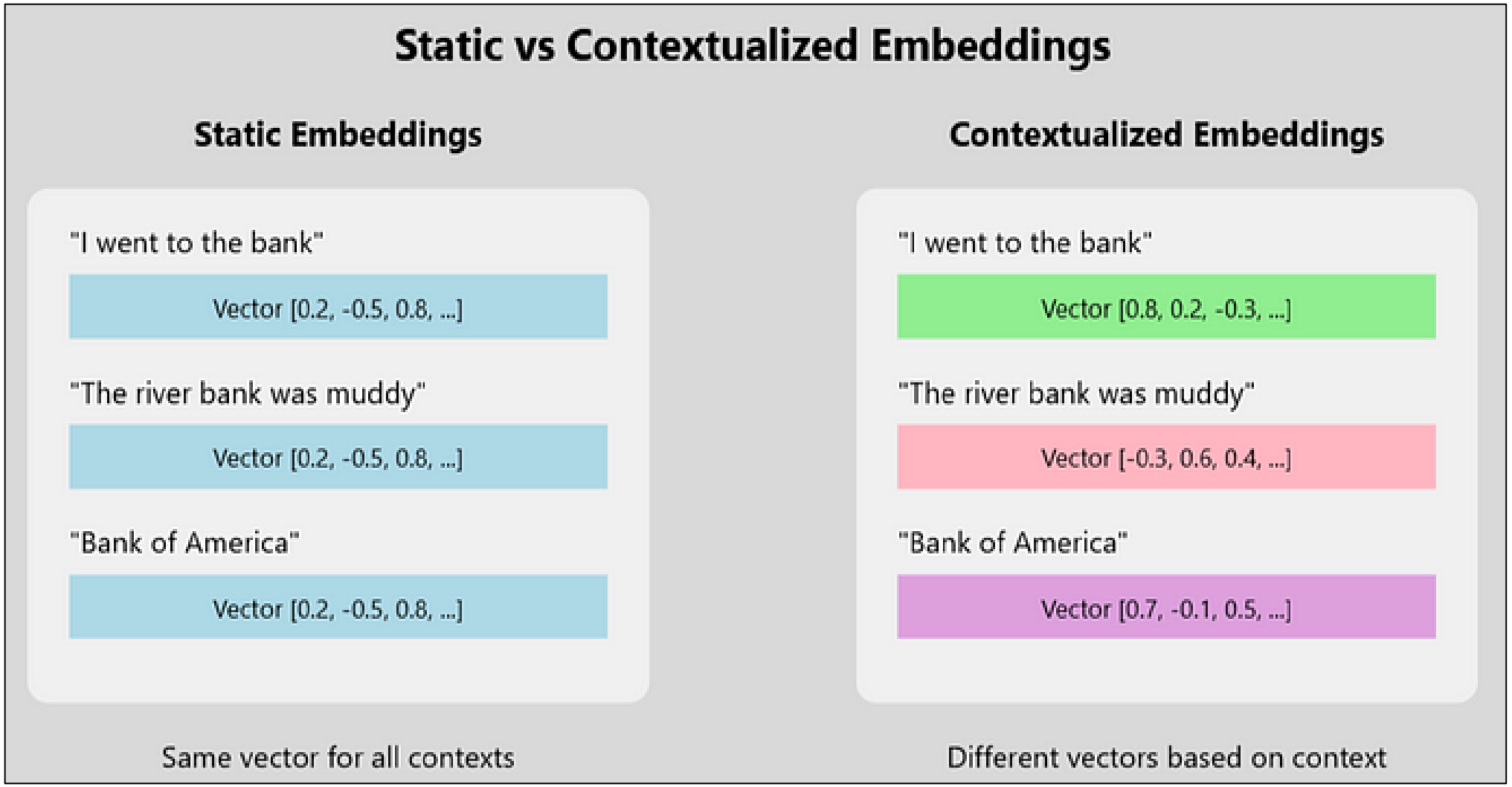
Contextualized word embeddings

- **Contextualized word embeddings** are word vectors that **change depending on the context** a word appears in.
- The **same word** will have **different vectors** if it's used in **different sentences** or **different meanings**.

Characteristics:

- **Context-dependent**: The embedding of a word is influenced by the words around it, meaning the same word can have different embeddings in different sentences.
  - For example, the word **"bank"** will have a different embedding when it appears in the sentence "I went to the bank to deposit money" versus "I sat on the bank of the river."
- **Dynamic Computation**: Embeddings are computed using deep neural networks (such as transformers), which take the full context into account.
- Handles polysemy (words with multiple meanings) effectively.

Contextualized word embeddings



Contextualized word embeddings

Feature	Static Embeddings	Contextual Embeddings
Context Sensitivity	No context consideration	Dynamic, context-dependent
Representation	Single, fixed vector for each word	Multiple vectors for the same word depending on context
Model Type	Shallow models (e.g., Word2Vec, GloVe)	Deep models (e.g., BERT, GPT)
Handling Polysemy	Struggles with polysemy	Handles polysemy effectively
Computational Complexity	Lower, faster to train and use	Higher, slower to train and use
Example Use Cases	Word similarity, text classification	Named Entity Recognition, Question Answering, Translation

What is a Large language model?

- A **language model** is a type of artificial intelligence (AI) system designed to understand, generate, and work with human language.
- At its core, a language model learns the patterns and structures of a language by analyzing vast amounts of text data.
- A language model predicts the next word in a sentence, given the previous words.
- A language model is a probabilistic model that assign probabilities to sequence of words.
- In practice, a language model allows us to compute the following:

- We usually train a neural network to predict these probabilities.

Large Language Models (LLMs):

- A neural network trained on a large corpora of text is known as a Large Language Model (LLM).
- Trained on massive datasets using transformer architectures. Examples: GPT, BERT, T5.

Contextualized word embeddings

Contextualized word embeddings models:

- ELMo
- BERT
- GPT

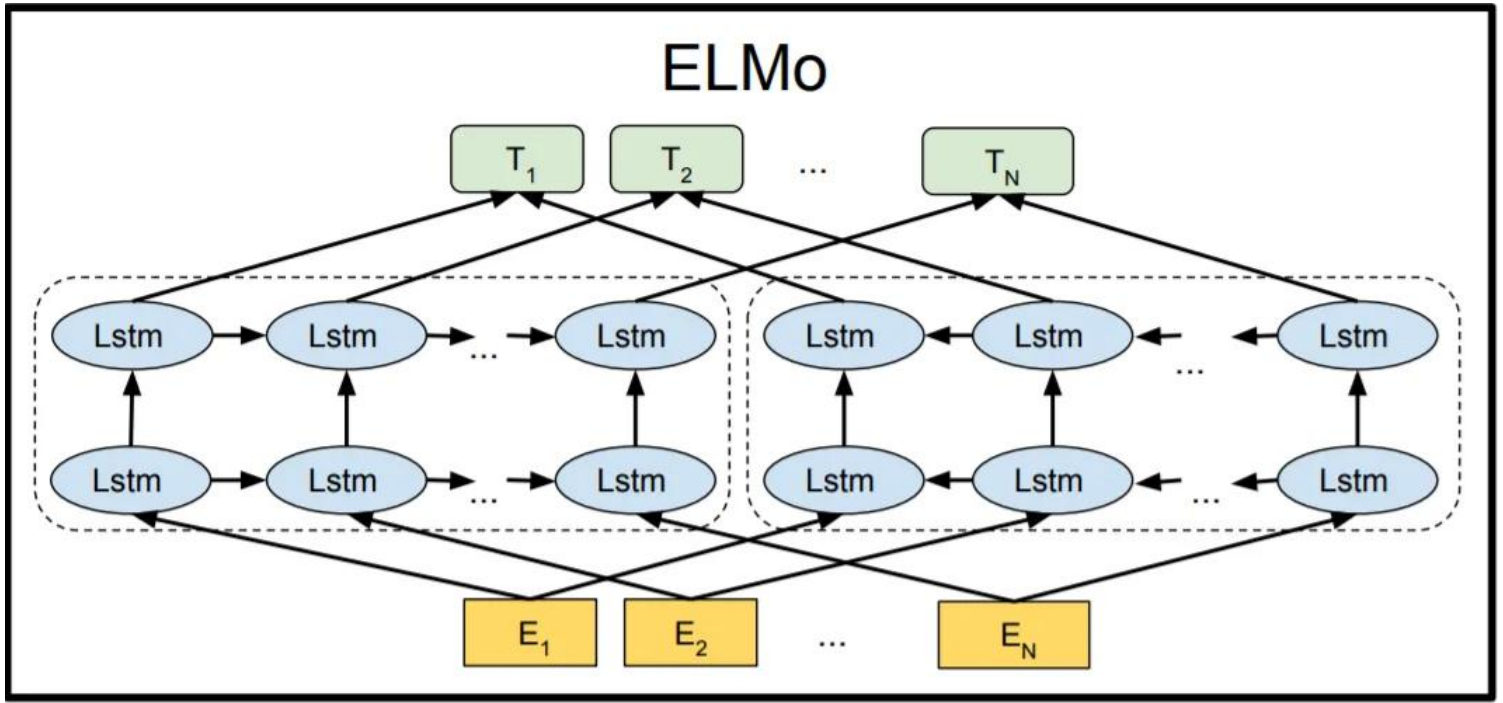
- At its core, ELMo uses bidirectional LSTMs to perform its processing, while BERT uses a Transformer encoder and GPT a Transformer decoder.

ELMo (Embeddings from Language Models)

- **ELMo (Embeddings from Language Models)** is a deep contextualized word representation technique developed by researchers at AllenNLP.
- ELMo is a deep learning technique that utilizes a bi-directional language model (biLM) to generate contextual representations of words.
- ELMo considers the surrounding words to capture the context-dependent meaning of each word.
- Unlike traditional word embeddings like Word2Vec or GloVe, which assign a fixed vector to each word, **ELMo generates dynamic embeddings based on the entire context of the sentence**.

ELMo Architecture

- The heart of ELMo is a biLM, which consists of two independent LSTM (Long Short-Term Memory) networks, each processing the input sequence in opposite directions: forward and backward.
- The outputs of these two LSTMs are concatenated to form a single vector representation for each word in the input sequence.
- This concatenation of forward and backward LSTM outputs represents the contextual information surrounding each word.



How it works

ELMo Input Embedding

- Tokenize word into characters
- Each character is mapped to a vector using a **character embedding matrix**.
- CNN is applied over character sequence and returns final word-level vector

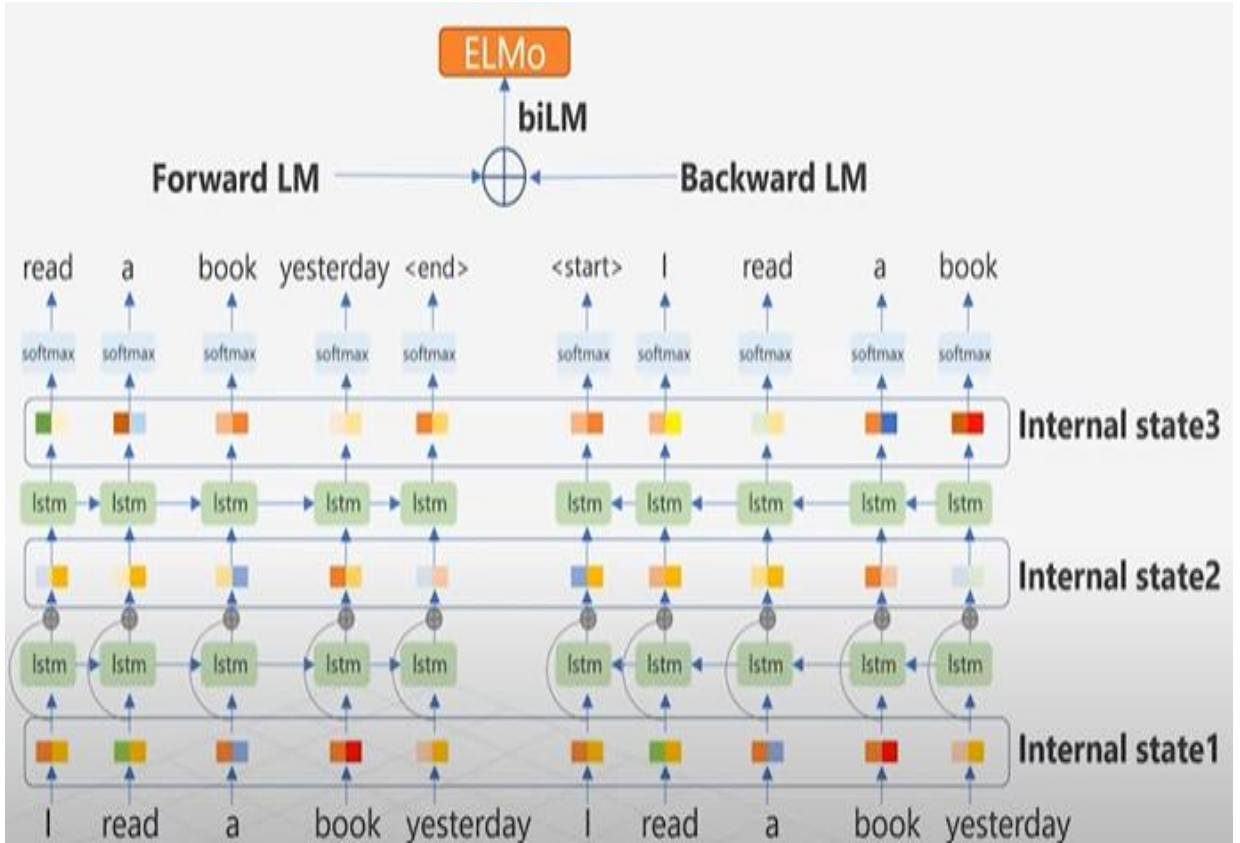
Bidirectional LM

- A **Bidirectional Language Model** is a model that learns to understand language by reading **in two directions**:

•**Forward**: Left to right (normal reading)

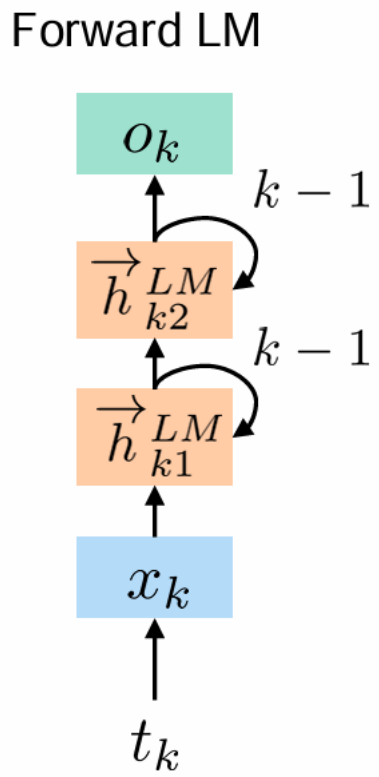
•**Backward**: Right to left (reverse reading)

- This helps it understand the **full context** around each word.



How it works

forward LM



- A **Forward LM** processes text from **left to right**.
- It trains to predict the **next word** in a sequence, based on the previous words.
- Each word is first turned into a vector.
- These word vectors feed into a **LSTM (Long Short-Term Memory)** network.
- The LSTM processes the sequence **one word at a time** from **left to right**.
- At each step, it outputs a **hidden state** that represents all words **seen so far**.

Example: Sentence:"The cat sat on the“

The Forward LM would try to predict:"mat“

So it's learning: P("mat"|"The cat sat on the")

The forward LM is a deep LSTM that goes over the sequence from start to end to predict token t_k based on the prefix $t_1 \ldots t_{k-1}$:

$$p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s)$$

Parameters: token embeddings Θ_x LSTM $\overrightarrow{\Theta}_{LSTM}$ softmax Θ_s

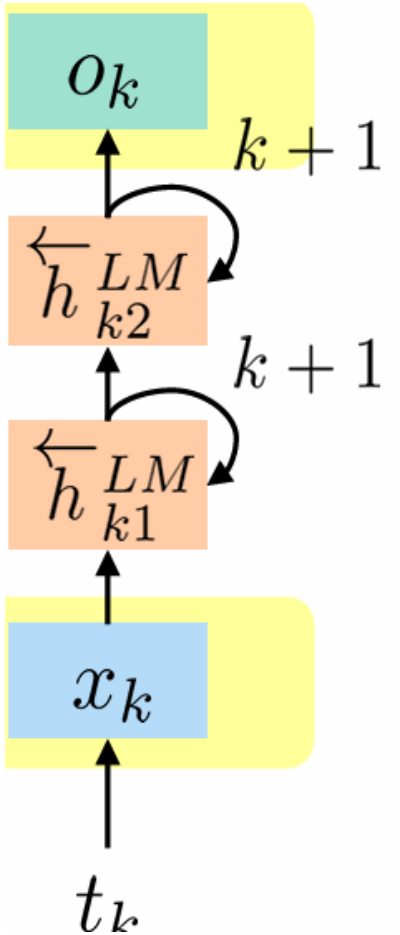
How it works

backword LM

- It processes the text from right to left.
- It trains to predict the previous word in a sentence, given the next words.

How It Works (in ELMo):

1. Each word in the sentence is converted into an embedding vector
2. These word vectors are then passed through a **Backward LSTM** network.
3. The **Backward LSTM** processes the sentence from **right to left** (starting from the last word).
4. For each word, the **Backward LSTM** outputs a hidden state that captures **all the future words** after it.



The backward LM is a deep LSTM that goes over the sequence from end to start to predict token t_k based on the suffix $t_{k+1} \ldots t_N$:

$$p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s)$$

How it works

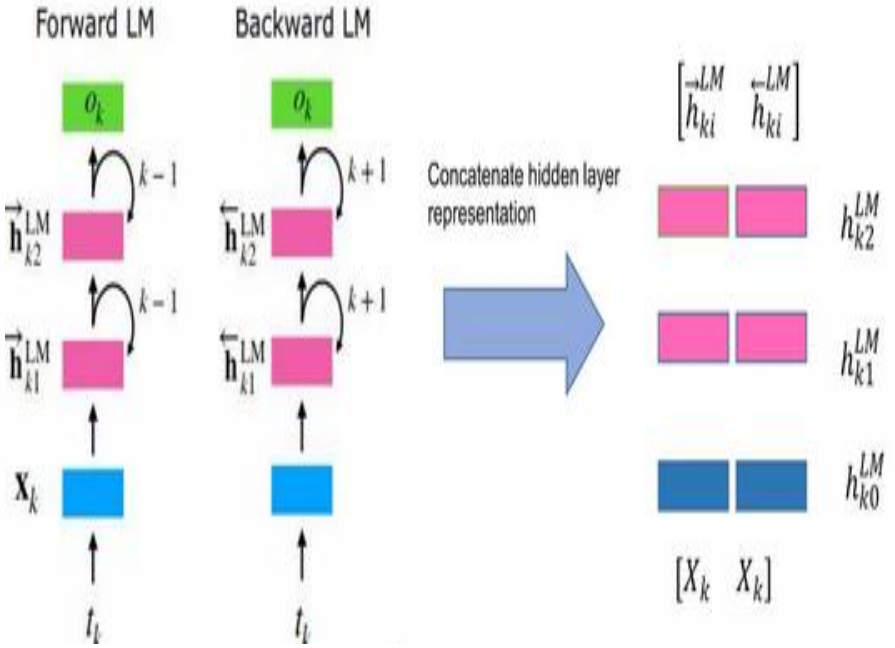
ELMo's token representations:

Given a token representation $\mathbf{x}_k$, each layer *j* of the LSTM language models computes a vector representation $\mathbf{h}_{k,j}$ for every token *k*.

With *L* layers, ELMo represents each token as

$$R_k = \{\mathbf{x}_k^{LM}, \overrightarrow{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, \ldots, L\}$$
$$= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, \ldots, L\},$$

where $\mathbf{h}_{k,j}^{LM} = [\overrightarrow{\mathbf{h}}_{k,j}^{LM}; \overleftarrow{\mathbf{h}}_{k,j}^{LM}]$ and $\mathbf{h}_{k,0}^{LM} = \mathbf{x}_k$

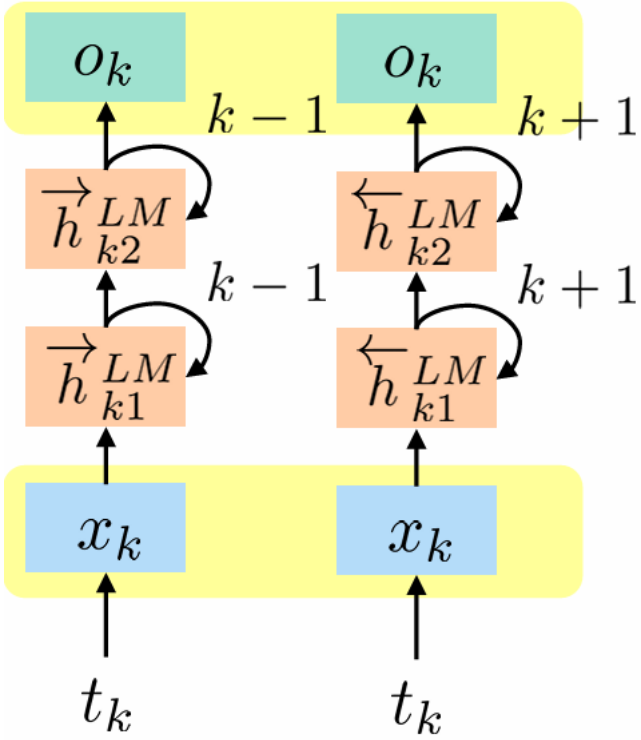


ELMo learns softmax weights s_j^{task} to collapse these vectors into a single vector and a task-specific scalar γ^{task}:

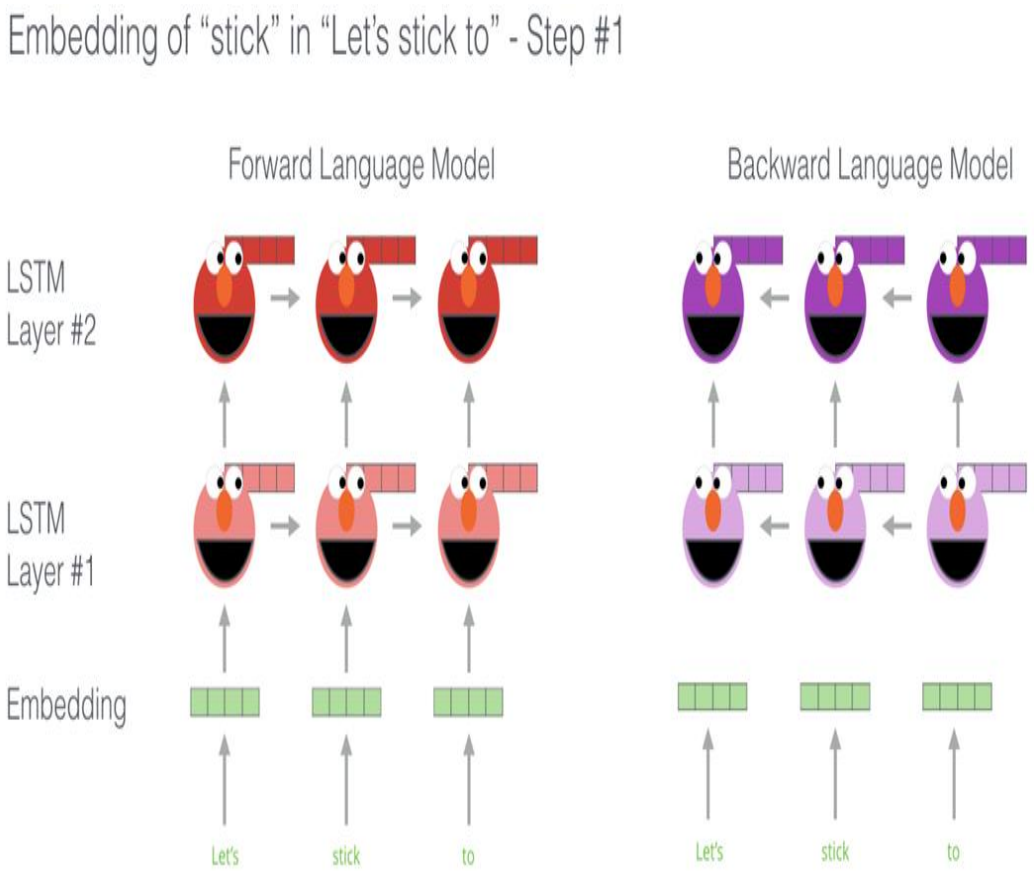
$$\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,j}^{LM}.$$

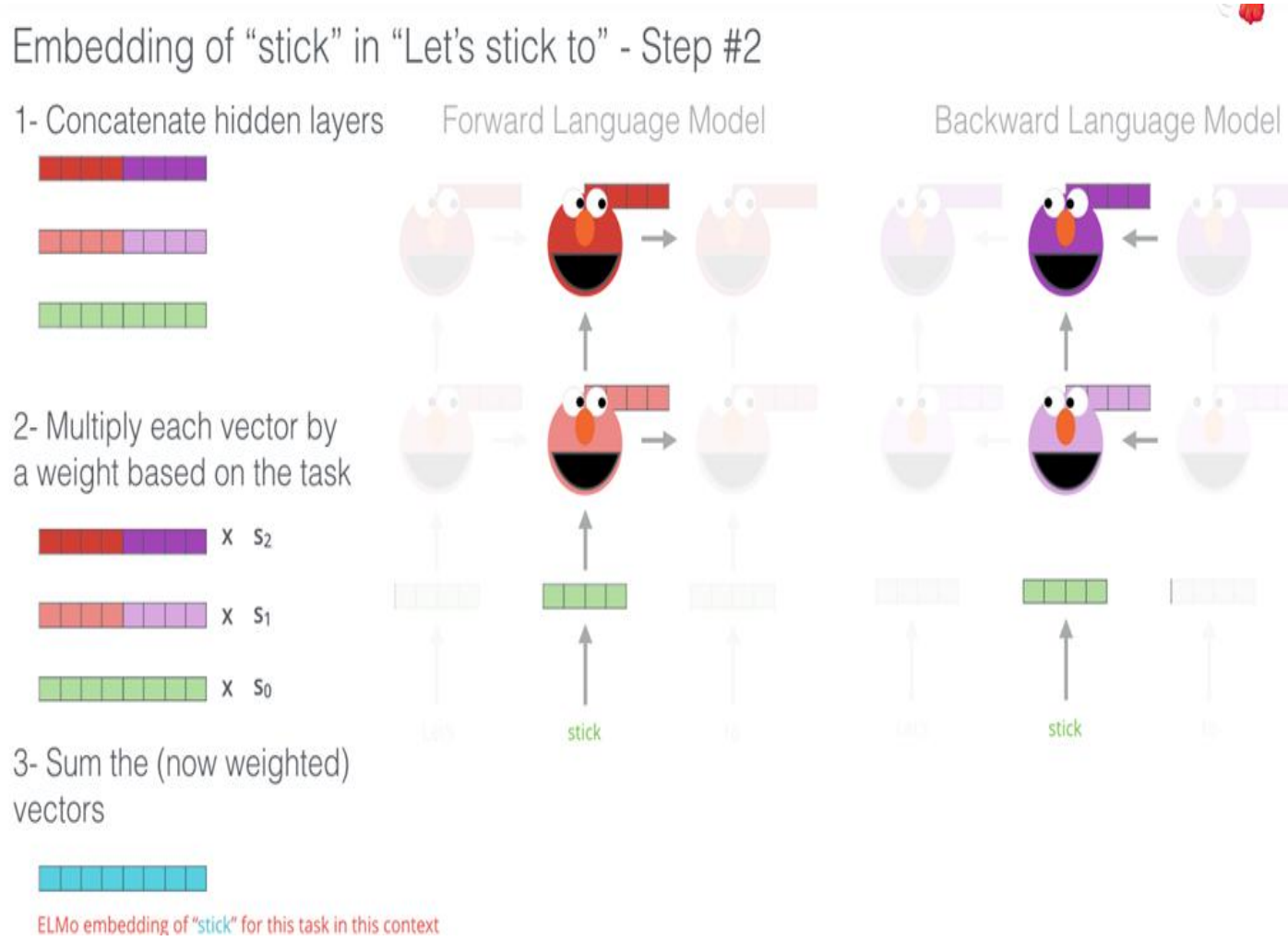
How it works

$$\mathrm{ELMo}_k^{\mathrm{task}} = \gamma^{\mathrm{task}} \times \sum \begin{cases} s_2^{\mathrm{task}} \times h_{k2}^{LM} & \overrightarrow{h}_{k2}^{LM} \quad \overleftarrow{h}_{k2}^{LM} \\ s_1^{\mathrm{task}} \times h_{k1}^{LM} & \overrightarrow{h}_{k1}^{LM} \quad \overleftarrow{h}_{k1}^{LM} \\ s_0^{\mathrm{task}} \times h_{k0}^{LM} & x_k \quad x_k \end{cases}$$



ELMo Illustration





ELMo Pretrained Model

```
from allennlp.commands.elmo import ElmoEmbedder

  # Load the model (downloads weights automatically the first time)
elmo = ElmoEmbedder()
sentence = ["The", "cat", "sat", "on", "the", "mat"]

  # Get ELMo embeddings: returns (3 layers, num_tokens, 1024)
vectors = elmo.embed_sentence(sentence)

  # Example: Average over the 3 layers
import numpy as np

  word_vectors = np.average(vectors, axis=0) # shape: (num_tokens, 1024)

  # Print each word and its vector (first 5 dims)
for word, vec in zip(sentence, word_vectors):
•    print(f"{word:10s} → {vec[:5]} ...")
```

What is a Large language model?

- A **language model** is a type of artificial intelligence (AI) system designed to understand, generate, and work with human language.
- At its core, a language model learns the patterns and structures of a language by analyzing vast amounts of text data.
- A language model **predicts the next word** in a sentence, given the previous words.
- A language model is a probabilistic model that assign probabilities to sequence of words.
- practice, a language model allows us to compute the following:

- We usually train a neural network to predict these probabilities.

Large Language Models (LLMs):

- A neural network trained on a large corpora of text is known as a Large Language Model (LLM).
- Trained on massive datasets using transformer architectures. Examples: GPT, BERT, T5.

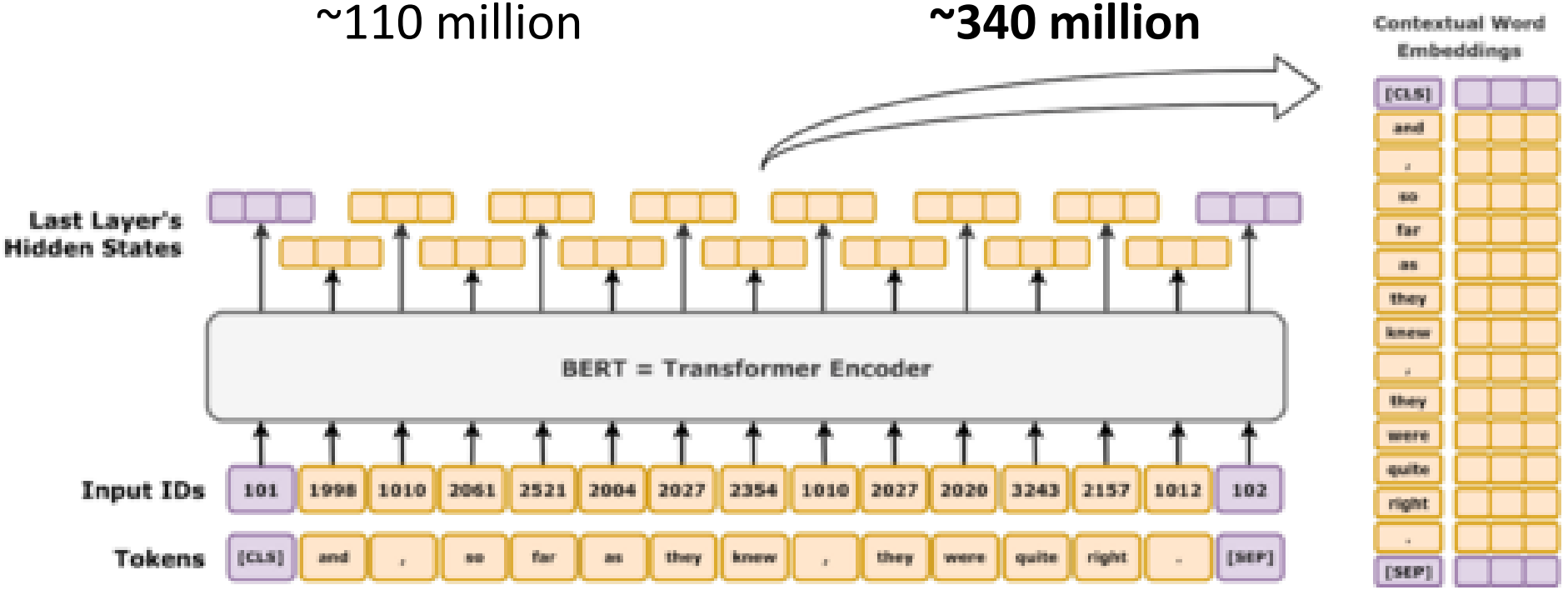
BERT: Bidirectional Encoder Representations from Transformers

- BERT is one of the modern large language model developed by Google in late 2018.
- Unlike static embeddings, **BERT generates different embeddings for the same word depending on its context**.
- BERT uses a transformers (encoders only), that processes all words in a sentence in parallel.
- This speeds up training and allows BERT to handle large datasets more effectively.
- BERT reading the text in both directions simultaneously.
- This bidirectional approach allows BERT to get the full context/meaning of a sentence by looking at the words that come before and after each word.
- Vaswani's transformer was trained on a translation task (English-German and English-French), while BERT was trained on the two tasks Masked Language Modeling (MLM) and Next Sentence Prediction (NSP).
- With these changes, BERT makes a split between creating embeddings of natural language and executing on natural language tasks.
- This split is put concretely in BERT's two-step framework pre-training, and fine-tuning.

BERT: Bidirectional Encoder Representations from Transformers

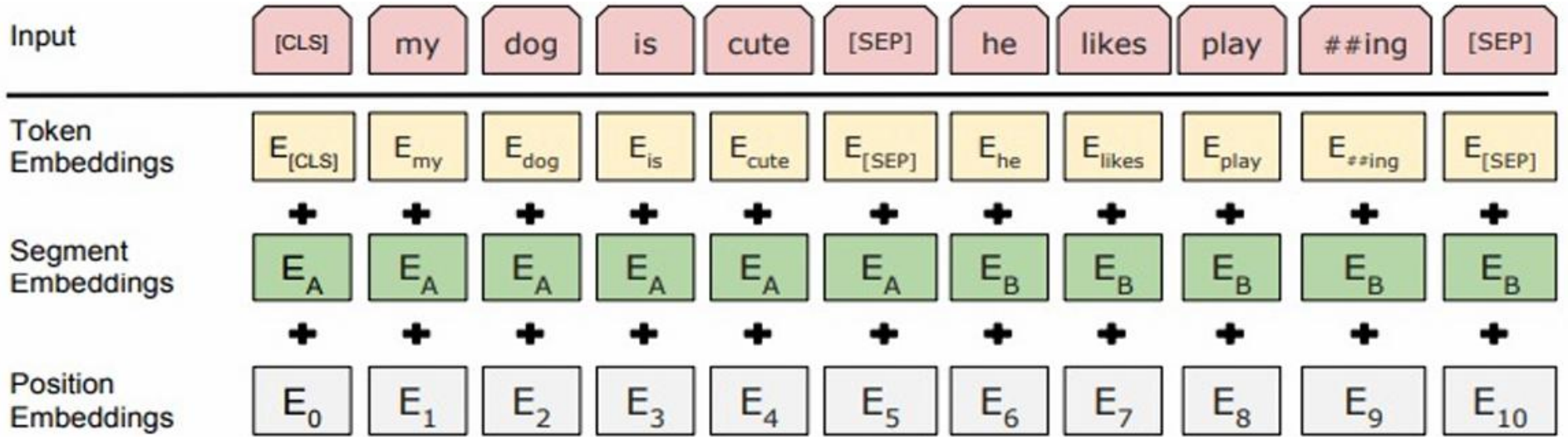
- two popular and widely used BERT models

Feature	BERT-Base	BERT-Large
Layers (Encoders)	12	**24**
Hidden Size	768	**1024**
Attention Heads	12	**16**
Parameters	~110 million	**~340 million**



Input/Output representations

- The first token of every input sequence to BERT must be the special classification token ([CLS]).
- BERT separates the two sentences with a special separation token ([SEP]).
- BERT adds a segment encoding to every token indicating whether it belongs to sentence A or sentence B.
- The input embeddings are the sum of the token encoding, the segmentation encodings and the positional encodings

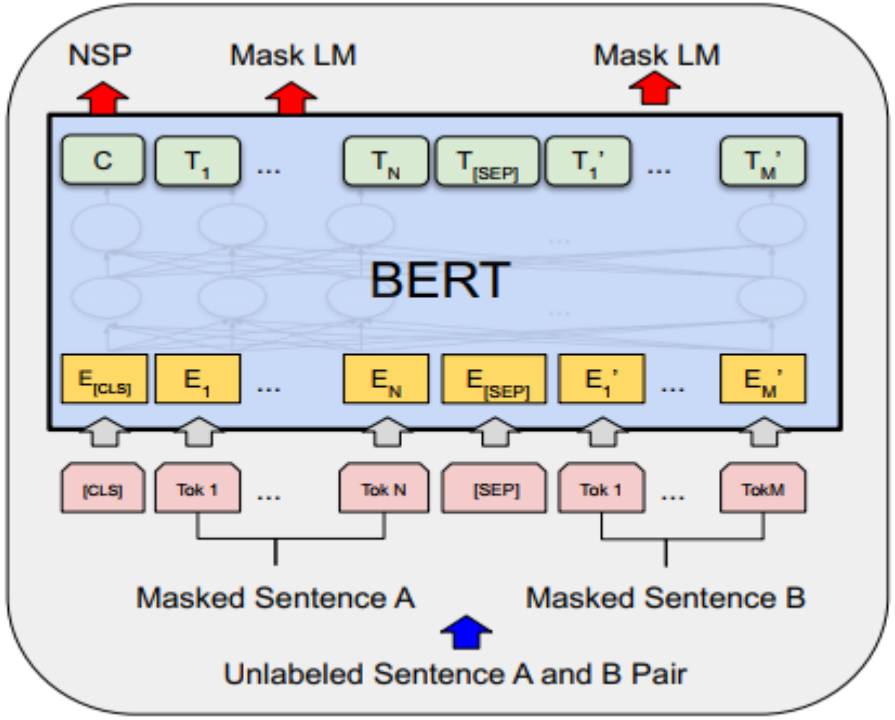


BERT pre-training

- During pre-training, the model is trained on unlabeled data over the different pre-training tasks.
- The result of the pre-training is the embeddings.
- While the embeddings of both BERT and Word2vec are bidirectional, but BERT's embeddings are contextual.
- BERT was pre-trained on two corpora, totaling about 3,300M words (800M from BooksCorpus, 2,500M from English Wikipedia)

BERT was pre-trained using two different pre-training tasks.

1. Masked Language modelling
2. Next Sentence Prediction



Masked Language Modeling

- MLM is a **self-supervised learning** task where BERT learns to understand the context of a word by predicting missing (masked) words in a sentence.
- In Masked Language Modeling (MLM), one word in each sentence is masked by replacing it with a [MASK] token, and the task is predicting it.
- In BERT, randomly selected words in a sentence are masked, and the model must predict the right word given the left and right context.

By doing MLM, BERT learns:

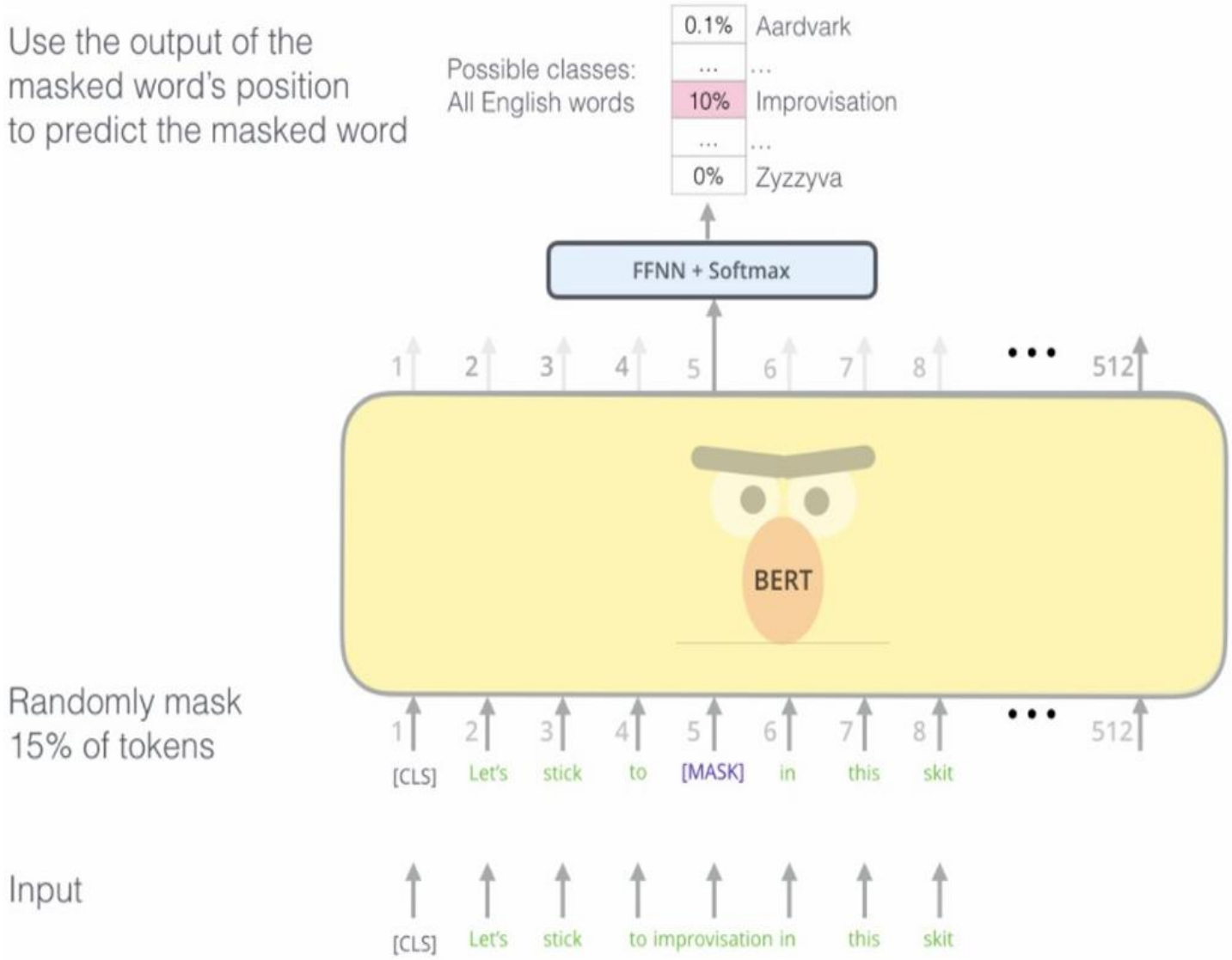
- Word **meanings** in different contexts
- How words **relate** to each other
- a **deeper understanding** of language for downstream/specific tasks

Masked Language Modeling

- The pre-training procedure selects 15% of the tokens from the sentence to be masked.
- BERT does not always replace the "masked" word with the actual [MASK] token.
- BERT replaces the "masked“ word with the [MASK] token 80% of the time a random word 10% of the time and does not change the word at all 10% of the time.
- For instance, the sentence "my dog is hairy" would 80% of the time appear as "my dog is [MASK]", 10% of the time as "my dog is apple" (where the word ’apple’ is randomly sampled) and 10% of the time as "my dog is hairy".

Masked Language Modeling-Training

- Take a normal sentence
- Mask some words (15%)
- Tokenize and encode the sentence
- Feed into BERT
- BERT predicts the masked word
- Compute loss (if training), update model



Next Sentence Prediction

- The second task that BERT was pre-trained on is Next Sentence Prediction (NSP).
- Two sentences were fed to BERT and the task is to determine if the second sentence follows the first sentence in the training corpus.

Why does BERT use NSP?

- To help BERT **understand relationships between sentences**, not just words.

Many real tasks involve **more than one sentence**, like:

- **Question answering** (e.g., find the answer in a paragraph)
- **Natural language inference**
- **Dialogue systems**

- To be good at those, BERT needs to understand how sentences **connect or don't connect**.

During Pretraining:

1.Pick Two Sentences:

From your training data (like Wikipedia), pick:

- **50% of the time**: A real pair (Sentence B actually follows Sentence A).
- **50% of the time**: A random pair (Sentence B comes from somewhere else in the corpus).

Next Sentence Prediction

2. Create Input Format

[CLS] Sentence A [SEP] Sentence B [SEP]

- [CLS]: Special token at the start
- [SEP]: Separates Sentence A and Sentence B

3.Add token + segment + position embeddings

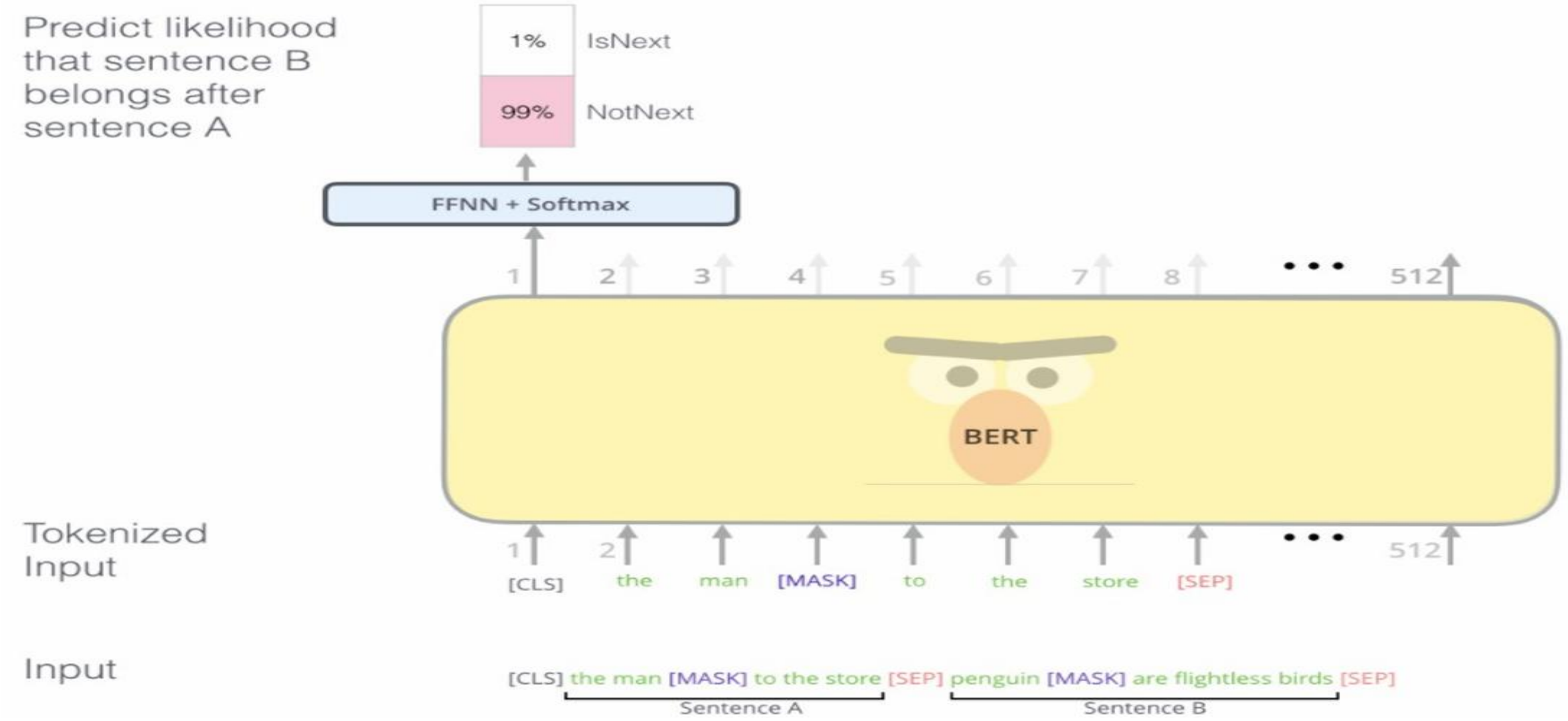
4. Feed into BERT

- The input goes into the transformer layers.

5. Prediction Head

- BERT uses the embedding from the [CLS] token.
- Only the [CLS] token output is used for NSP prediction
- It passes it to a classifier.
- It predicts:
  - **"IsNext"** (1) — if B follows A
  - **"NotNext"** (0) — if B is random

Next Sentence Prediction



Pre-Training datasets and details

- Training loss L is the sum of the mean masked LM likelihood and mean next sentence prediction likelihood.

Dataset:

- BooksCorpus (800M words), about 7,000 unique unpublished books from a variety of genres including Adventure, Fantasy, and Romance.
- English Wikipedia (2,500M words), excluding lists, tables, headers.
- Sequence length 512; Batch size 256; trained for 1M steps (approximately 40 epochs);

BERTBASE: N = 6, $d$model = 512, h = 12, Total Parameters=110M

4 cloud TPUs in Pod configuration

BERTLARGE: N = 24, $d$model = 1024, h = 16, Total Parameters=340M,16 Cloud TPUs

- Each pretraining took 4 days to complete.

Fine-tuning with BERT

- Fine-tuning BERT means taking a pre-trained BERT model and training it further on a specific task like text classification, question answering, NER, etc.
- For fine-tuning, the BERT model is first initialized with its pre-training parameters and then its architecture is modified for the specific task.
- This modification usually only involves adding a head layer in order to plug in task-specific inputs and outputs into BERT.

Add a task-specific head:

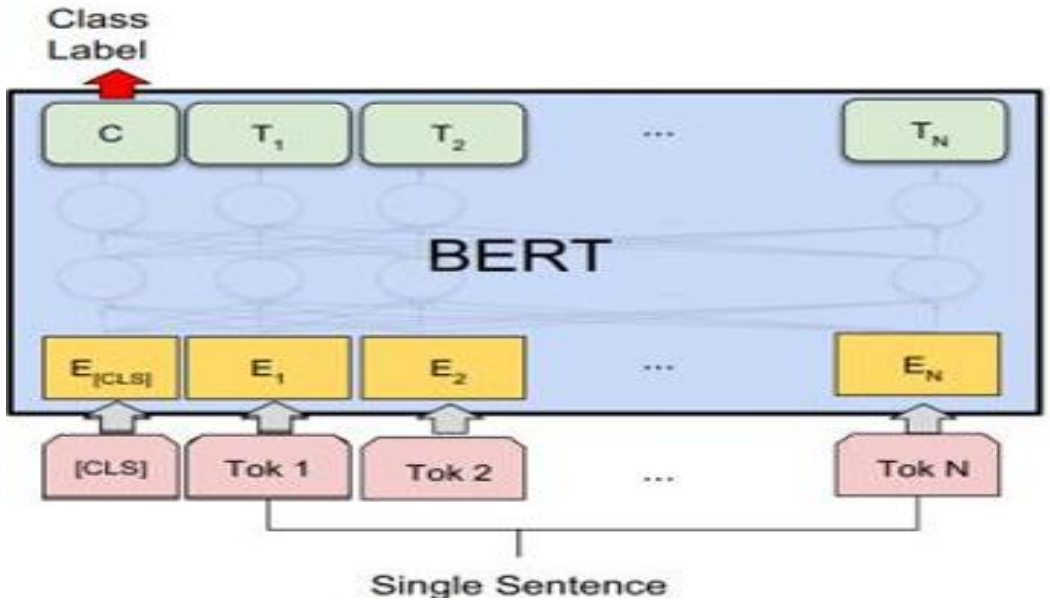
  - Classification: Add a linear layer on top of the [CLS] token.
  - NER / Token classification: Add a linear layer on each token's embedding.
  - QA: Add two linear layers to predict start and end positions.

- The big advantage of this approach lies in the training time of the fine-tuning procedure, which at most takes a few hours on a GPU.
- Contrast this with the training time of pre-training, which took 4 days on multiple TPUs

Fine-tuning with BERT

Text Classification:

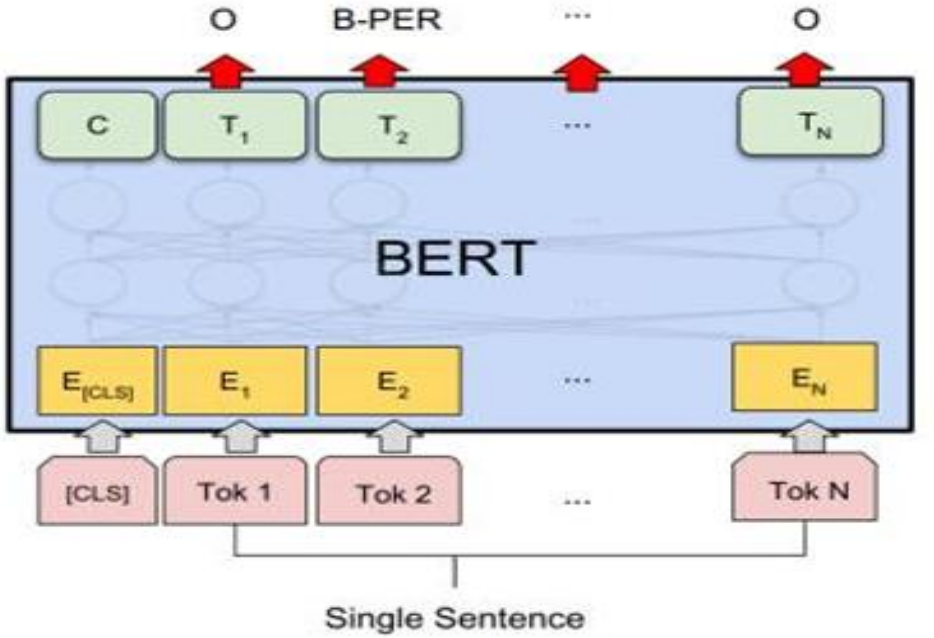
- Add a **classification head** on top of the [CLS] token (a Linear layer + optional dropout)
- In a **text classification task**, each input text is associated with a label (e.g., 0 or 1 for binary classification, or 0 to *n-1* for multi-class)



Named Entity Recognition (NER) :

- The architecture adds a **linear layer** on top of **each token output**, not just the [CLS] token

Output: A label for each token (e.g., “B-PER", B-ORG", “B-LOC")

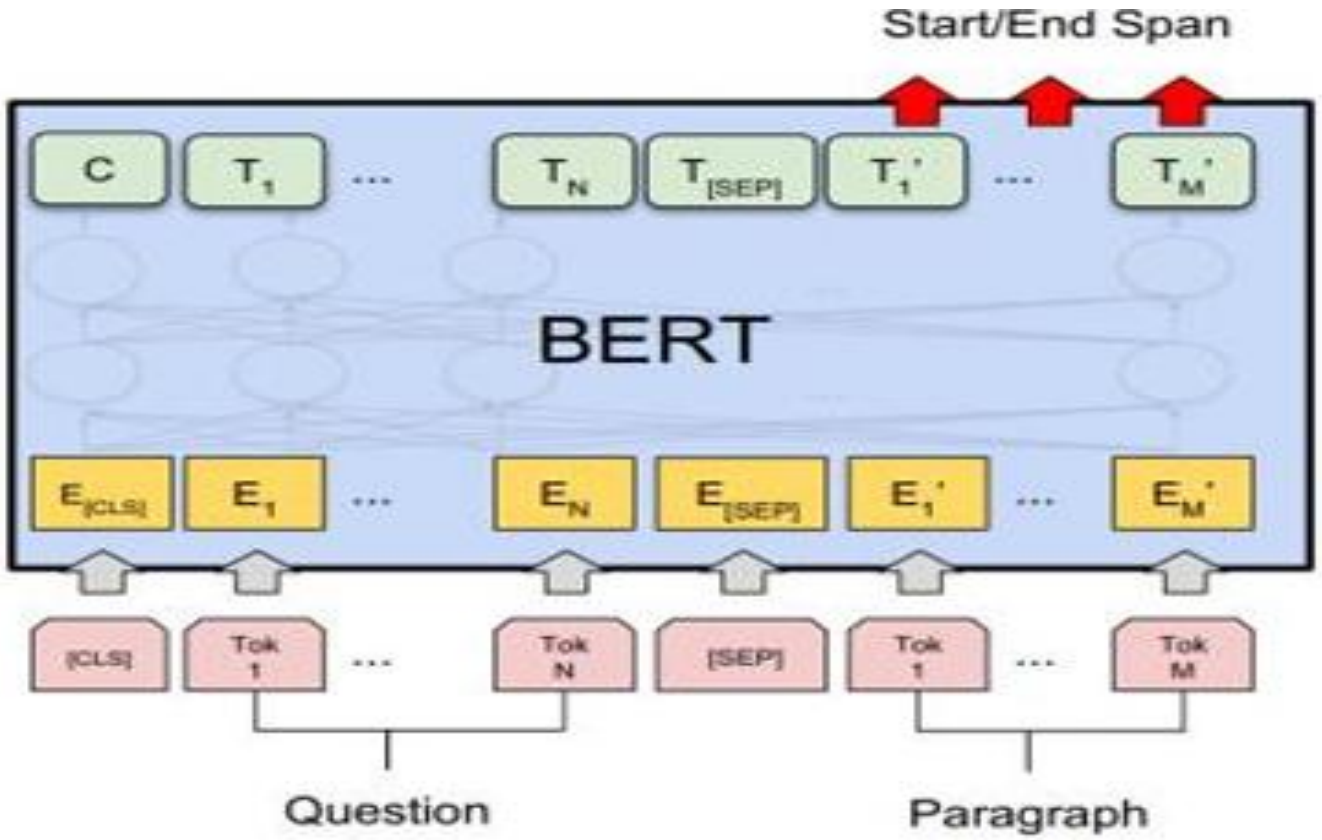


Fine-tuning with BERT

Question Answering:

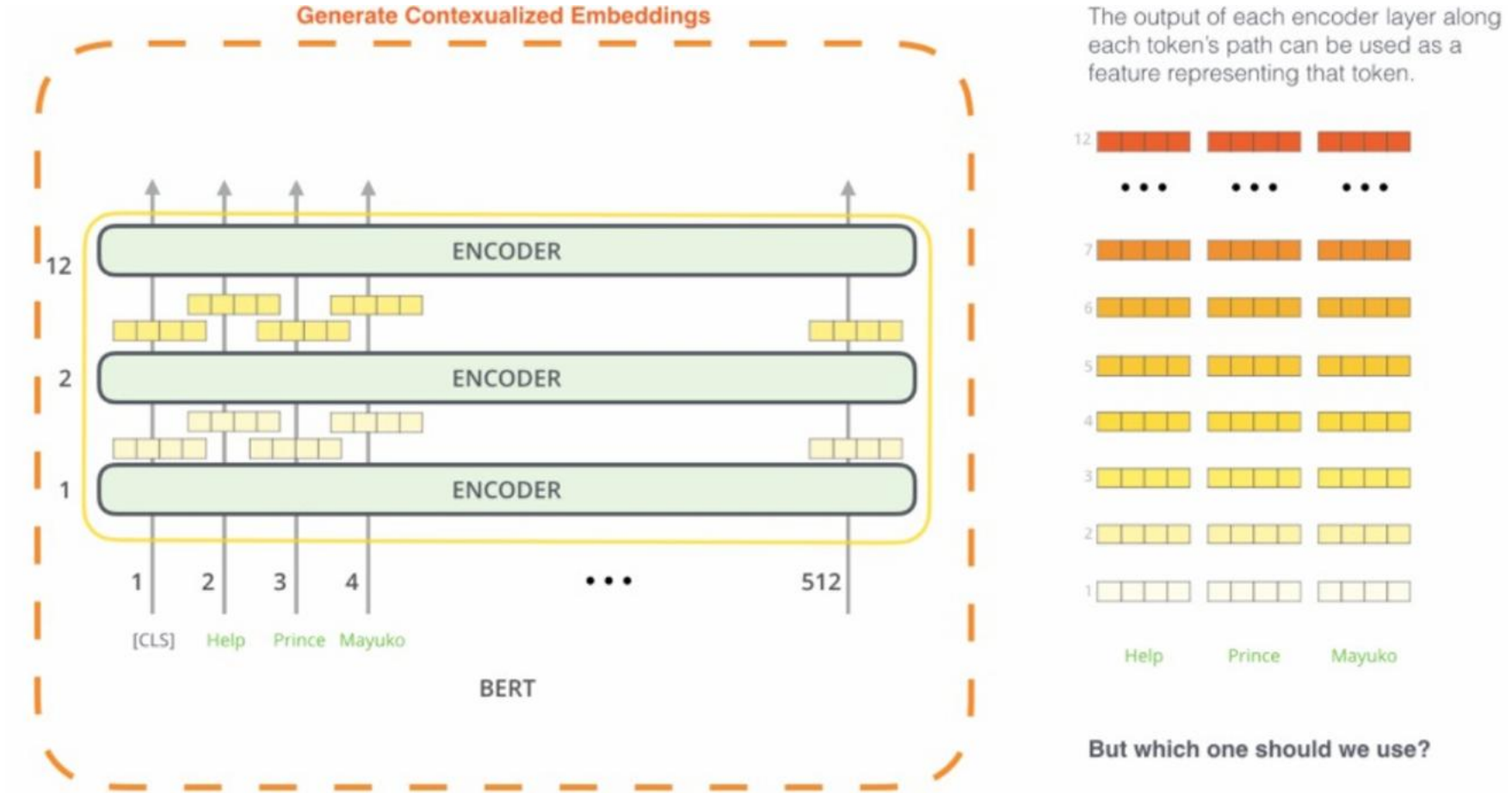
Input: A context paragraph + a question

- Add two linear layers to predict start and end positions
- **Output**: The start and end positions of the answer in the context



Fine-tuning with BERT

contextualized embeddings



Last layers have the best contextualized embeddings

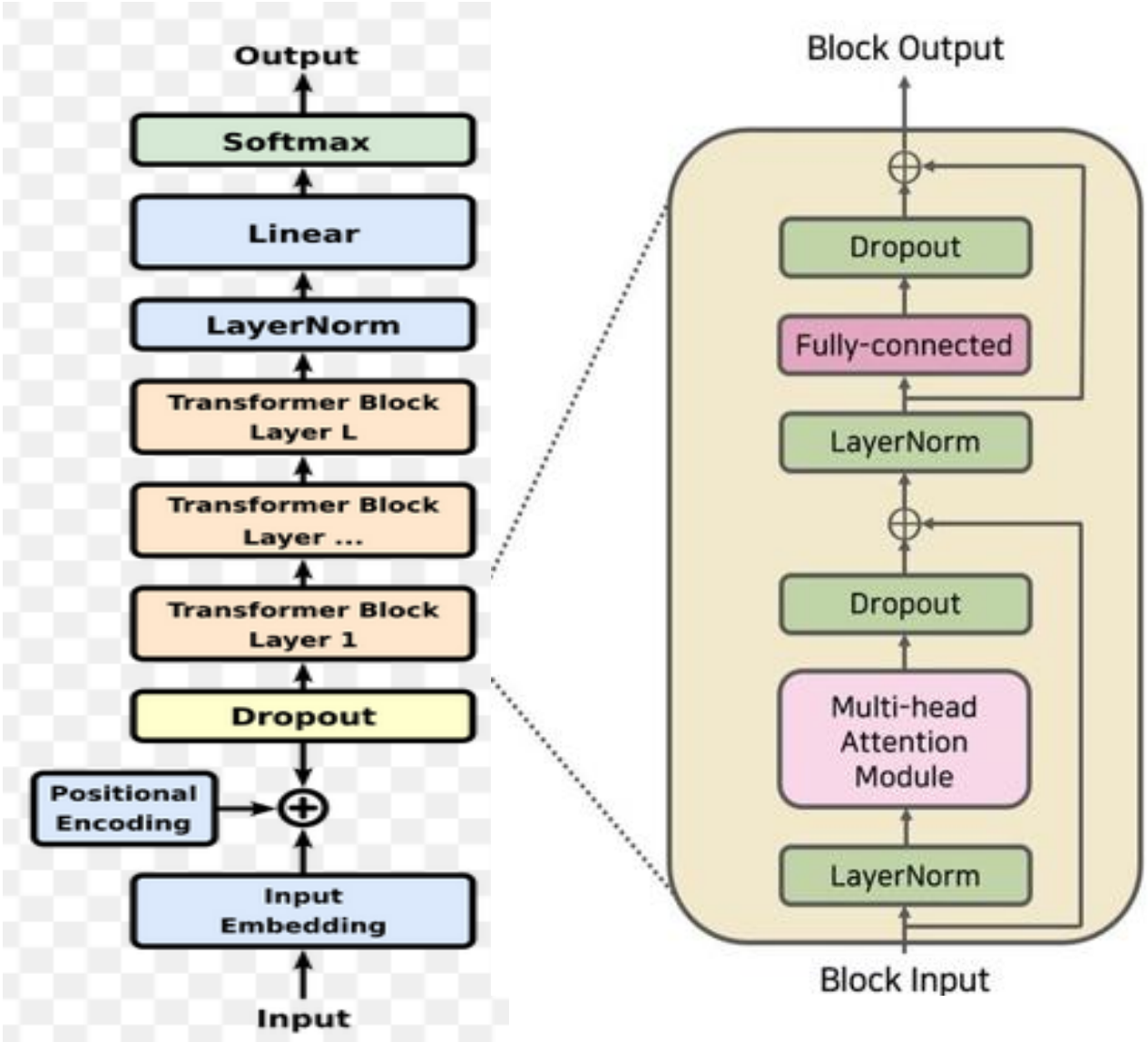
GPT(Generative Pre-trained Transformer)

- GPT is a transformer-based model that generates context-aware embeddings by processing text in a unidirectional manner (left-to-right).
- GPT uses the **decoder part of the transformer,** to process the input text.
- The model consists of multiple layers of self-attention mechanisms, which allow it to capture dependencies between words in a sentence.
- AI research firm OpenAI introduced the first GPT model, GPT-1, in 2018.
- The most recent GPT model is GPT-4, which released in early 2023.
- In May 2024, OpenAI announced the multilingual and multimodal GPT-4o, capable of processing audio, visual and text inputs in real time
- By leveraging the Transformer architecture in a novel way, GPT has set new standards in language modeling, allowing machines to generate human-like text, perform complex reasoning, and even engage in creative tasks such as writing poetry or generating code

GPT(Generative Pre-trained Transformer)

- GPT is a language model built using the Transformer architecture, specifically focusing on the autoregressive approach to text generation.
- Unlike models that use bidirectional attention (like BERT), GPT is trained to predict the next word in a sequence based on preceding words, making it particularly suited for tasks that involve text generation, rather than just understanding.
- The fundamental idea behind GPT is to pre-train the model on a massive corpus of text in an unsupervised manner, and then fine-tune it on smaller, task-specific datasets to improve performance for particular applications.
- This two-stage training approach, called unsupervised pre-training and supervised fine-tuning, has become a staple for modern large-scale NLP models.

GPT Architecture



GPT Architecture

Input Embeddings:

- The input to the GPT model is first tokenized and embedded into continuous vectors.
- These vectors represent the individual tokens (words or subwords) and are fed into the Transformer decoder layers.

Positional Encoding:

- Since the Transformer architecture does not inherently handle sequence order, GPT incorporates positional encodings to inject information about the order of tokens in the input sequence.

Self-Attention Mechanism:

- Each token in the input attends to every other token in the sequence, enabling the model to understand context. GPT employs masked self-attention, which prevents the model from "seeing" future tokens during training.
- This ensures the model generates text one token at a time, where each token is conditioned only on preceding tokens.

GPT Architecture

Feedforward Neural Network (FFN):

- After the attention mechanism, each token's output is passed through a feedforward neural network, allowing the model to learn more complex representations of the input.

Residual Connections and Layer Normalization:

- These techniques help stabilize training and allow gradients to flow more easily through the network, improving learning efficiency and preventing overfitting.

After this step, **each token now has a refined, contextualized embedding**, informed by all other tokens.

Final Linear + Softmax (for output prediction)

After passing through all Transformer blocks:

- A **linear layer** maps each token's final embedding to vocabulary size.
- A **softmax** layer produces probabilities for the **next token**.

Training Process: Unsupervised Pre-Training

- GPT follows a two-phase training process that enables it to learn a general understanding of language and then specialize in specific tasks:
- In the pre-training phase, GPT is exposed to a massive, diverse dataset of text (e.g., books, websites, articles) without specific task labels.
- The model learns to predict the next word in a sentence using the objective of maximum likelihood estimation (MLE), which adjusts the model's parameters to make the predicted words as close as possible to the actual words.
- This phase allows GPT to acquire a broad understanding of language structure, grammar, facts, and common sense knowledge.

Supervised Fine-Tuning

- Fine-tuning represents a sophisticated process that refines pre-trained models (like GPT) for specific tasks or domains, leveraging the model's extensive foundational knowledge acquired during initial training on diverse datasets.
- This involves adjusting the model's parameters based on task-specific data, enhancing its performance, and enabling it to handle particular applications with greater precision and efficiency.
- After pre-training, GPT is fine-tuned on a smaller, labeled dataset for specific downstream tasks, such as sentiment analysis, question-answering, or summarization.
- Fine-tuning is a supervised process, where the model is trained to optimize task-specific loss functions (e.g., cross-entropy loss for classification tasks).
- The fine-tuning phase refines the model's behavior for particular applications, improving its performance on those tasks.

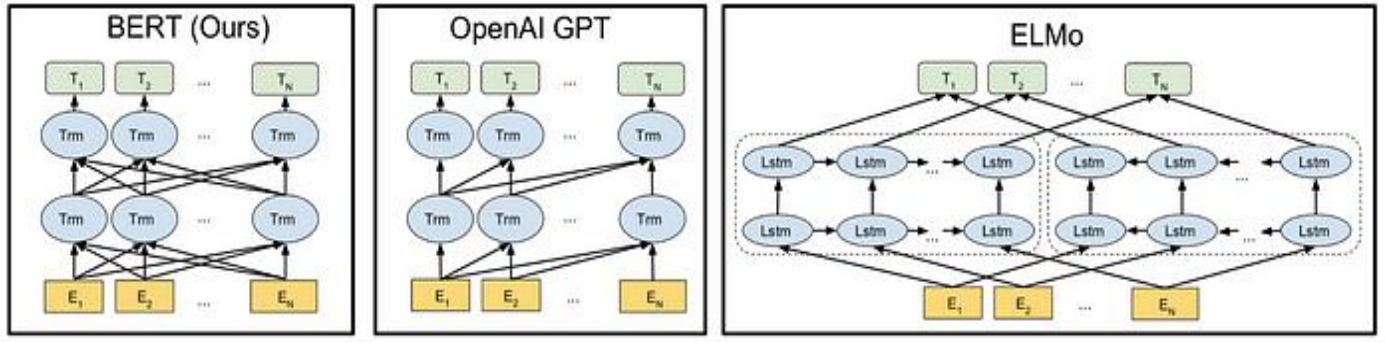
Compare ELMO, BERT & GPT word embedding's

Feature	ELMo	BERT	GPT
Model Type	BiLSTM	Transformer (bidirectional)	Transformer (unidirectional)
Contextualized?	Yes	Yes	Yes
Pretraining Objective	Language modeling (bidirectional LSTM)	Masked Language Modeling (MLM) + Next Sentence Prediction	Causal Language Modeling (predict next token)
Word Embeddings	Contextual (from LSTM layers)	Contextual (from transformer layers)	Contextual (from transformer layers)
Directionality	Bidirectional	Deeply bidirectional	Left-to-right only
Static or Dynamic Embeddings?	Dynamic	Dynamic	Dynamic
Architecture Type	LSTM-based	Transformer encoder	Transformer decoder
Embedding Layers Used	Combination of layers	Usually last 4 layers (or weighted sum)	Final hidden state per token
Output for Word	Different for each sentence (context-aware)	Different for each sentence (context-aware)	Different for each sentence (context-aware)

Compare Pretrained LMs: ELMo, BERT & GPT

1. Model Architecture

Feature	ELMo	BERT	GPT
Base Architecture	Bi-directional LSTM	Transformer (Encoder only)	Transformer (Decoder only)
Directionality	Bi-directional (via LSTM)	Deeply bidirectional	Uni-directional (left to right)
Layers	2-layer BiLSTM	Typically 12 (Base) or 24 (Large)	Varies (GPT-2: 12-48 layers)



Pretrained LMs: ELMo, BERT & GPT

2. Pretraining Objective

Feature	ELMo	BERT	GPT
Objective	Language Modeling (forward & backward separately)	Masked Language Modeling (MLM) + Next Sentence Prediction (NSP)	Causal Language Modeling (CLM)
Context type	Contextual (sentence-level)	Deep bidirectional context	left to right

Pretrained LMs: ELMo, BERT & GPT

3. Pretraining Data

Feature	ELMo	BERT	GPT
Dataset	1 Billion Word Benchmark	BooksCorpus + English Wikipedia	BookCorpus, WebText (for GPT-2)
Size	1 Billion Words	~3.3B words	~40GB (GPT-2), more for GPT-3

Applications of NLP

- Sentiment Analysis
- Question and Answering
- Information retrieval

Sentiment Analysis

- Sentiment classification (also known as sentiment analysis) is the process of determining the emotional tone behind a body of text.
- It’s commonly used to understand opinions, reviews, or customer feedback. Here are the main steps involved:

1. Data Collection

- Collect textual data along with sentiment labels (e.g., positive, negative, neutral).
- Sources: IMDb, Twitter, product reviews, customer feedback.

2. Text Preprocessing

- Clean and normalize the text:
  - Lowercasing
  - Removing punctuation, numbers, HTML tags, etc.
  - Tokenization (splitting into words or subwords)
  - Stop-word removal (optional)
  - Lemmatization or stemming (optional for classical ML)

Sentiment Analysis

3. Text Representation

You need to convert text into numerical features.

- **Traditional Methods**:
  - Bag of Words (BoW),TF-IDF
- **Deep Learning**:
  - Word Embeddings (Word2Vec, GloVe)
  - Transformer-based (BERT, RoBERTa, etc.)

4. Model Selection

- **Traditional ML Models**: Naive Bayes, SVM, Logistic Regression
- **Deep Learning Models**: LSTM, GRU, CNN
- **Transformer Models**: BERT

5. Model Training

- Split data into train/validation/test sets.
- Train the model using labelled data.

6. Prediction (Inference)

- Feed new text to the trained model.
- Get sentiment prediction (positive, negative, etc.)

Sentiment Analysis using Pre-trained BERT Model

```python
from transformers import pipeline

# Load sentiment analysis pipeline with a BERT model
sentiment_pipeline = pipeline("sentiment-analysis", model="nlptown/bert-base-multilingual-uncased-sentiment")

texts = [
    "I absolutely loved this movie! It was fantastic.",
    "The product quality is terrible and the customer service is worse.",
    "It's okay, not the best but not the worst either."
]

# Analyze each text
for text in texts:
    result = sentiment_pipeline(text)[0]
    print(f"Text: {text}")
    print(f"Sentiment: {result['label']} (Confidence: {result['score']:.2f})\n")
```

```
Text: I absolutely loved this movie! It was fantastic.
Sentiment: 5 stars (Confidence: 0.95)

Text: The product quality is terrible and the customer service is worse.
Sentiment: 1 star (Confidence: 0.64)

Text: It's okay, not the best but not the worst either.
Sentiment: 3 stars (Confidence: 0.89)
```

Question Answering (Q&A)

- **Question Answering (Q&A)** is a system designed to automatically answer questions posed in natural language.
- Unlike basic search engines that return documents, QA systems try to **return direct answers**.

Components of a QA System

Question Processing

•Tokenization, POS tagging, NER, dependency parsing

•Determine question type (who, what, where, etc.)

Context Processing

- Preprocess the context (tokenization, sentence splitting, etc.).

Answer Extraction

- Extract a span from the document or passage.
- Use models (like BERT, RoBERTa) to identify the span of text in the context that answers the question.

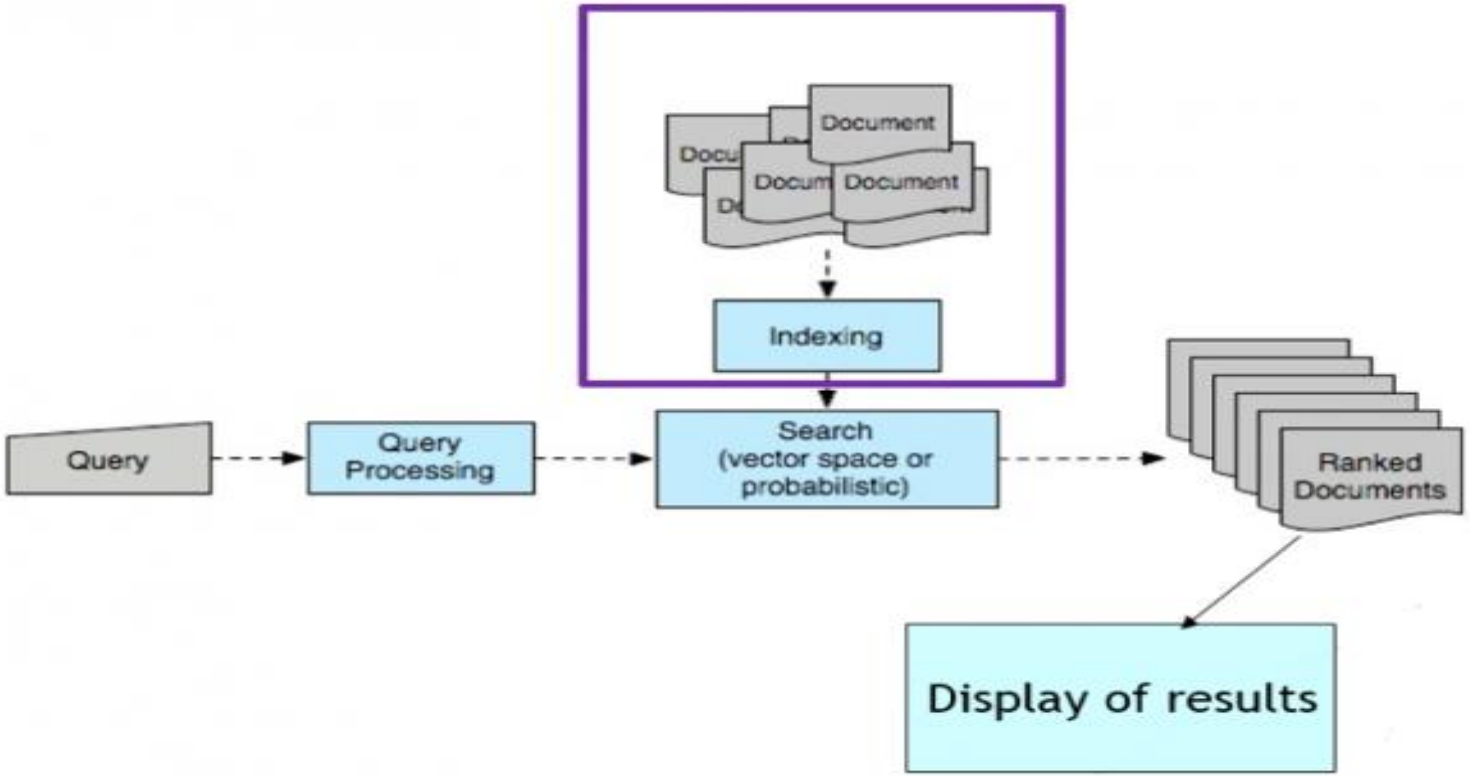
Question Answering (Q&A) using Pre-trained BERT Model

```
from transformers import pipeline

# Load a question-answering pipeline using a BERT model trained on SQuAD
qa_pipeline = pipeline("question-answering", model="bert-large-uncased-whole-word-masking-
finetuned-squad")

context = """
The Nile River is the longest river in the world, flowing through northeastern Africa for over
6,600 kilometers.
It is a vital water source for Egypt and Sudan. The river has two major tributaries: the White
Nile and the Blue Nile.
The White Nile is considered to be the headwaters and primary stream of the Nile itself.
"""

question = "What is the longest river in the world?"

# Run the QA model
result = qa_pipeline({
    'context': context,
    'question': question
})

print("Answer:", result['answer'])
print("Score:", result['score'])
```

```
Answer: The Nile River
Score: 0.5444401502609253
```

Information Retrieval

- An Information Retrieval (IR) system is a software-based framework designed to efficiently and effectively retrieve relevant information from a collection of data or documents in response to user queries.
- These systems are integral to various applications, such as search engines, recommendation systems, document management systems, and chatbots.
- **Google Search** is the most famous example of information retrieval.



Information Retrieval

Indexing:

- **Indexing** processes a collection of documents and builds a table that associates each term (word) in a vocabulary with the set of items from the collection in which it occurs.
- This type of table is called an “inverted” index,
- To make searching efficient, IR systems build an **inverted index**, which maps **terms to documents** containing them.
- This allows for fast lookup and ranking.

Inverted Index Example

- Page1: "Machine learning is fun"
- Page2: "Learning is essential"
- Page3: "Deep learning with Python“

Word	Pages
machine	Page1
learning	Page1, Page2, Page3
is	Page1, Page2
fun	Page1
essential	Page2
deep	Page3
with	Page3
python	Page3

Information Retrieval

Search/Matching

- **searching** and **matching** are fundamental operations that help users find relevant documents or information from a large collection.
- **Searching** refers to the process where a user submits a **query**, and the IR system returns a list of relevant documents ranked by relevance.

Common Search Models:

Boolean Model:

- Uses logical operators (AND, OR, NOT) on keywords.

Vector Space Model (VSM)

- Represents documents and queries as vectors in a multi-dimensional space.
- Relevance is computed using **cosine similarity** between query and document vectors.

Probabilistic Models (e.g., BM25)

- Estimate the probability that a document is relevant to the query.
- More sophisticated and often more accurate than VSM.

Language Models:

- Treat each document as a probability distribution over terms.
- Rank based on the likelihood of the query being generated by the document model.

Steps in Information Retrieval Application

1. **Document Collection**
2. **Text Preprocessing**
   - Tokenization, Stop-word removal, Stemming/Lemmatization
3. **Indexing**
   - **Create Inverted Index**:
   - use:
     - **TF-IDF vectors**
     - **Dense embeddings** (BERT/Transformers)
4. **Query Processing**
   - Preprocess the user query similarly to the documents (tokenize, normalize, remove stop-words, etc.).
5. **Matching and Ranking**
   - Uses **cosine similarity** to rank documents.
6. **Retrieval**
   - Retrieve top-K ranked documents or passages.

Information Retrieval

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
# Step 1: Sample document corpus
documents = [
    "The Eiffel Tower is in Paris.",
    "The Great Wall of China is visible from space.",
    "The Statue of Liberty is in New York.",
    "Paris is the capital of France.",
    "Space exploration is fascinating."
]
# Step 2: Preprocessing & Vectorization (TF-IDF)
vectorizer = TfidfVectorizer(stop_words='english')
doc_vectors = vectorizer.fit_transform(documents)
# Step 3: Define a user query
query = "Where is the Eiffel Tower?"
# Step 4: Preprocess & Vectorize query
query_vector = vectorizer.transform([query])
# Step 5: Compute Cosine Similarity
similarities = cosine_similarity(query_vector, doc_vectors).flatten()
# Step 6: Rank documents by similarity
ranked_indices = similarities.argsort()[::-1]
# Step 7: Display top results
print("Top results for the query:\n")
for idx in ranked_indices[:3]:  # Top 3 results
    print(f"Score: {similarities[idx]:.4f} | Document: {documents[idx]}")
```

```
Top results for the query:

Score: 0.8686 | Document: The Eiffel Tower is in Paris.
Score: 0.0000 | Document: Space exploration is fascinating.
Score: 0.0000 | Document: Paris is the capital of France.
```