Embedding models from architecture to implementation

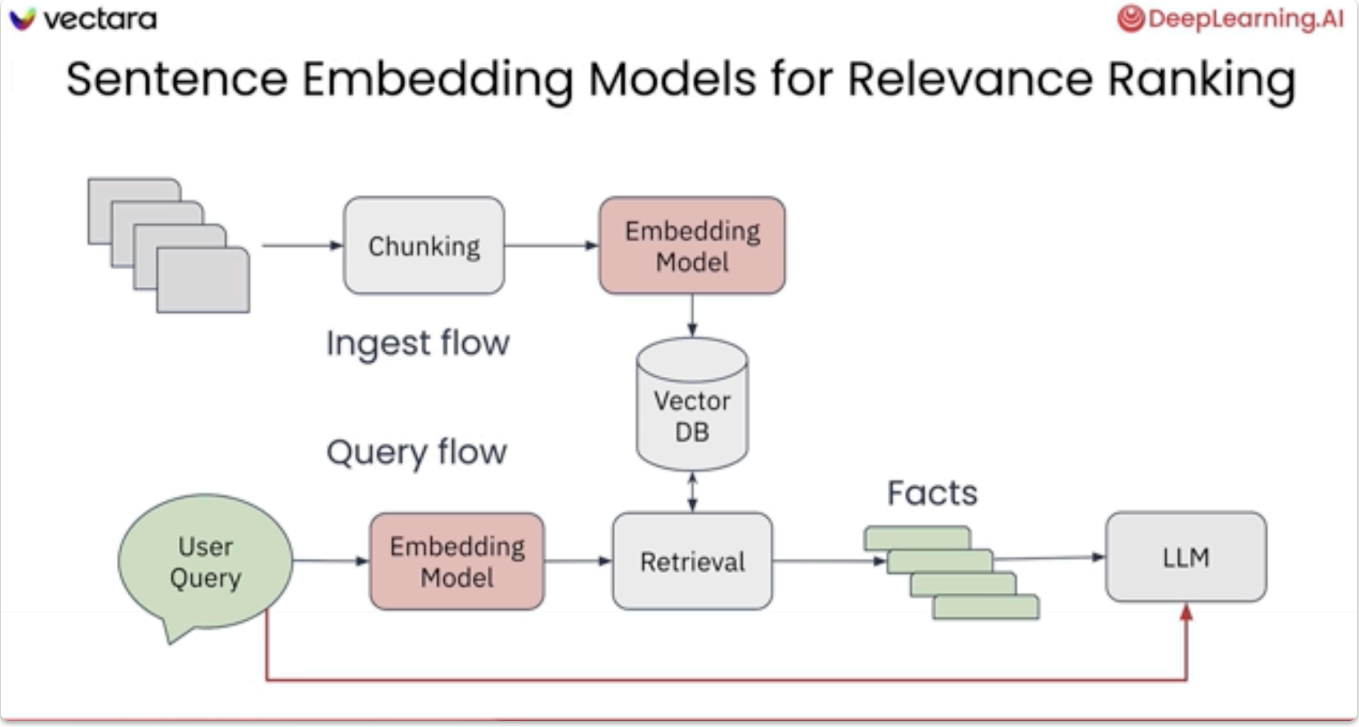
```
@HamadAlrashid
DeepLearning.ai Vectara Course Notes
```

Table of contents

Vanilla RAG

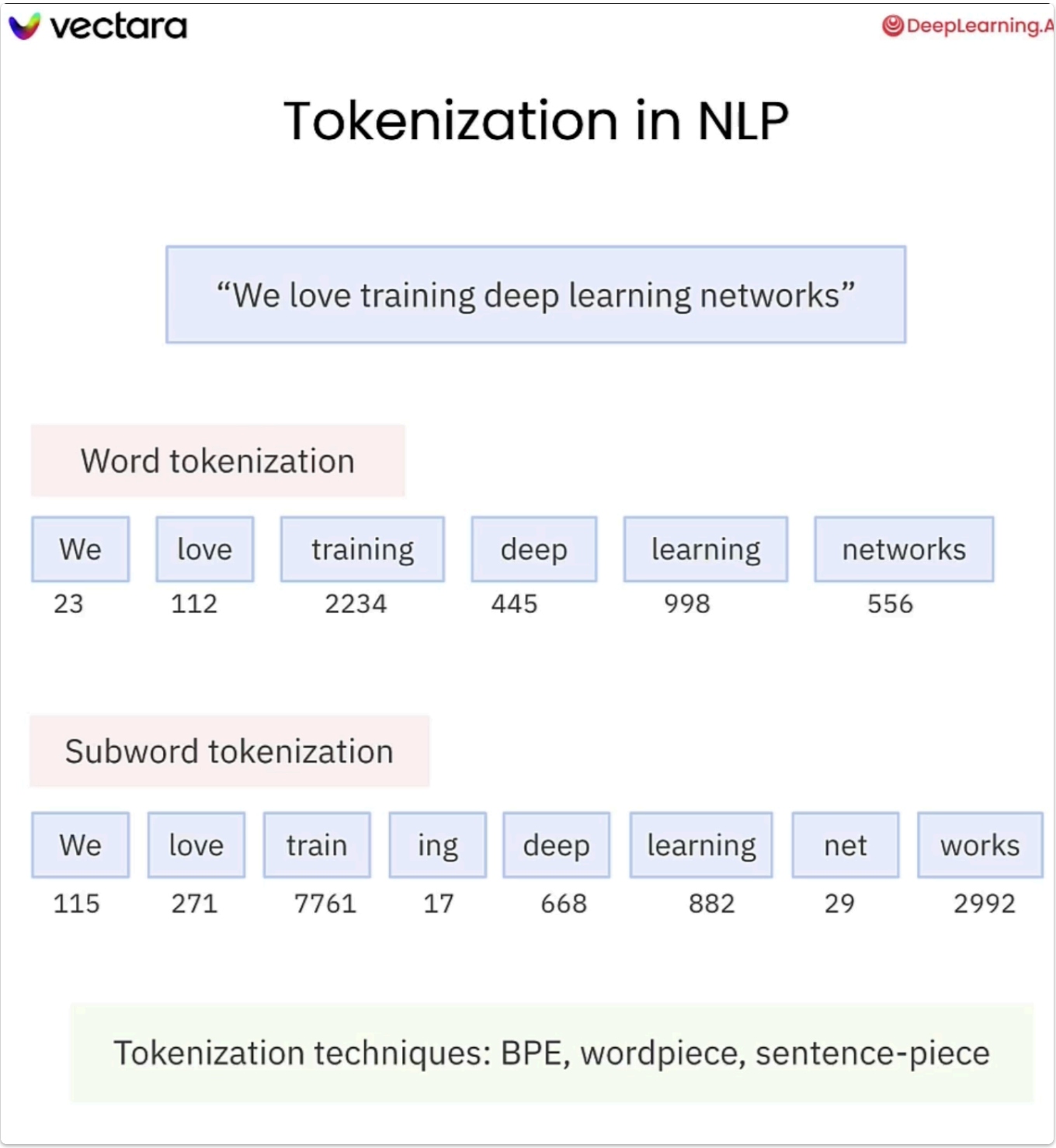
```
 Modern AI systems often rely on external sources of knowledge, involving
embedding models and vector retrieval components. Retrieval Augumented
```

Generation (RAG) is when retrieved information is augmented to the LLM to generate the final output, solving the limitation of large context windows within LLMs. The notes delve into the history, design, and implementaiton of embedding models



Tokenization in NLP

The very first step in NLP is tokenization, which attempts to break down the input text into numbered tokens or segments



- Byte Pair Encoding (BPE): iteratively merge the most frequent pairs of existing tokens. It starts with the base characters and merge the most frequent tokens to create a new token. It was used in GPT, RoBERTa
- WordPiece: Similar to BPE but selects the merges that maximize training data likelihood. It was used in BERT and DistilBERT
- Unigram: Starts with the largest vocabulary and iteratively starts removing tokens that least affect the corpus likelihood.
- Sentence-piece: a system or a toolkit that implements existing subword tokenization algorithms (e.g., BPE and unigram) with additional features such as processing raw text

and being language-agnostic

Transformer architecture

```
This is the underlying NN design that powers LLMs and embedding models
```

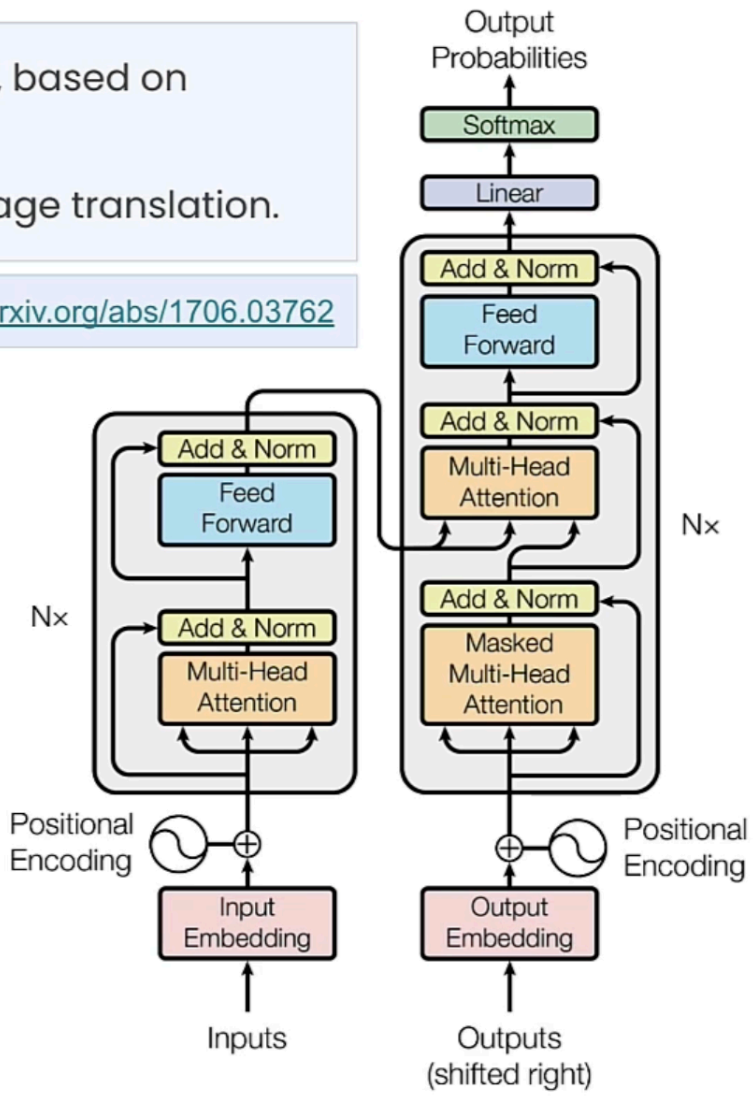
- It is an alternative to Recurrent Neural Networks (RNNs) and LSTMs
  - RNNs doesn't process each input separately, but rather use previous inputs to process the current input, resulting in proper sequential processing. However, it is slow and consider all previous tokens (some of which are unimportant)
  - LSTMs came after RNNs and tried to the improve long-term memory of RNNs by adding multiple gates (forget, input, output gates). It is faster than RNNs but still sequential
- The transformer solved the issues of inference speed by parallelization as opposed to sequential and global context (i.e., the meaning of a word depends on all surrounding words, not just the ones before).
- The transformer follows the encoder decoder architecture
- Introduced in 2017 by google and is based on attention mechanism
- Initially trained for language translation, which is why it has an encoder and a decoder
- **How is it different than word2vec or Glove?**
  - It provides *contextualized token embeddings*
  - This solved a significant problem in NLP, where the same word could have a different meaning in different sentences
  - word2vec is a 2012 model that learns the embedding of a word based on its neighboring words. Glove came in 2015 as a competitor

Transformer to the rescue!

Model introduced in 2017, based on attention mechanism.

Initially trained for language translation.

Attention Is All You Need: https://arxiv.org/abs/1706.03762



Encoder

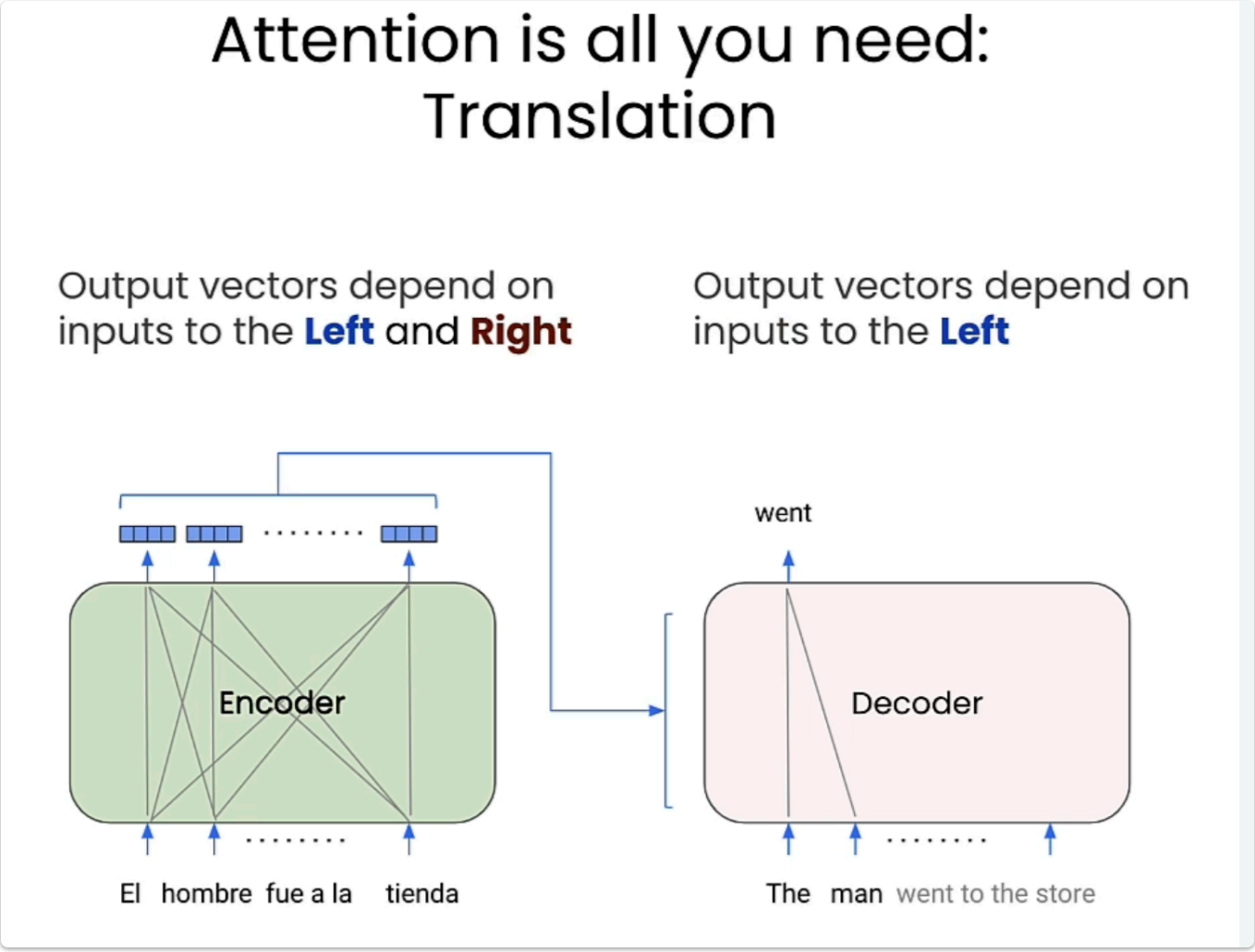
- Input: a sequence of symbol representations (tokens generated by a tokenizer such as Wordpiece of Sentencepiece)
- Output: a sequence of continuous representations (embeddings). i.e., multiple vectors for each input token

Decoder

- Input: continuous representations
- Output: A probability which is used to sample the next token or word

How translation work?

1. A whole sentence in one language is passed at once to the encoder which outputs a sequence of vectors that provide contextual meaning of the input tokens. i.e., a vector for each token in generated
2. The decoder takes the output of the encoder (sequence of word embeddings) and predict one word at a time. To generate nth translated word, the decoder use both the output of the encoder and the previous n-1 decoded word



Other Applications

- The decoder became the basis for many *LLMs* such as GPT2,3,4
- The encoder became the basis for *BERT*, the backbone for many embedding models

Bidirectional Encoder Representation of Transformers (BERT)

- Transformers were not designed to be language models initially. Their design had two main disadvantages when it comes to understanding languages: needs a lot of data and the architecture may not be complex enough.

- To address these concerns, **BERT** was introduced to overcome language understanding, which is the root problem of critical applications such as question-answering, translation, text summarization, and intent classification
- BERT is a stack of *transformer encoders* that generates a sequence of embedding vectors for each input token
- **BERT Network phases**
    1. Pre-training: Understand Language
    2. Fine Tuning: Understand Language specific tasks
- **Advantages over the initial Transformers**
    1. Fine tuning doesn't require obscene amounts of data
    2. Understand the full context words by looking at both ways via attention (*Bidirectional*)
    3. The output is a *representation* that encapsulates meaning of words in a vector format

BERT -

Bidirectional Encoder
Representations from Transformers.

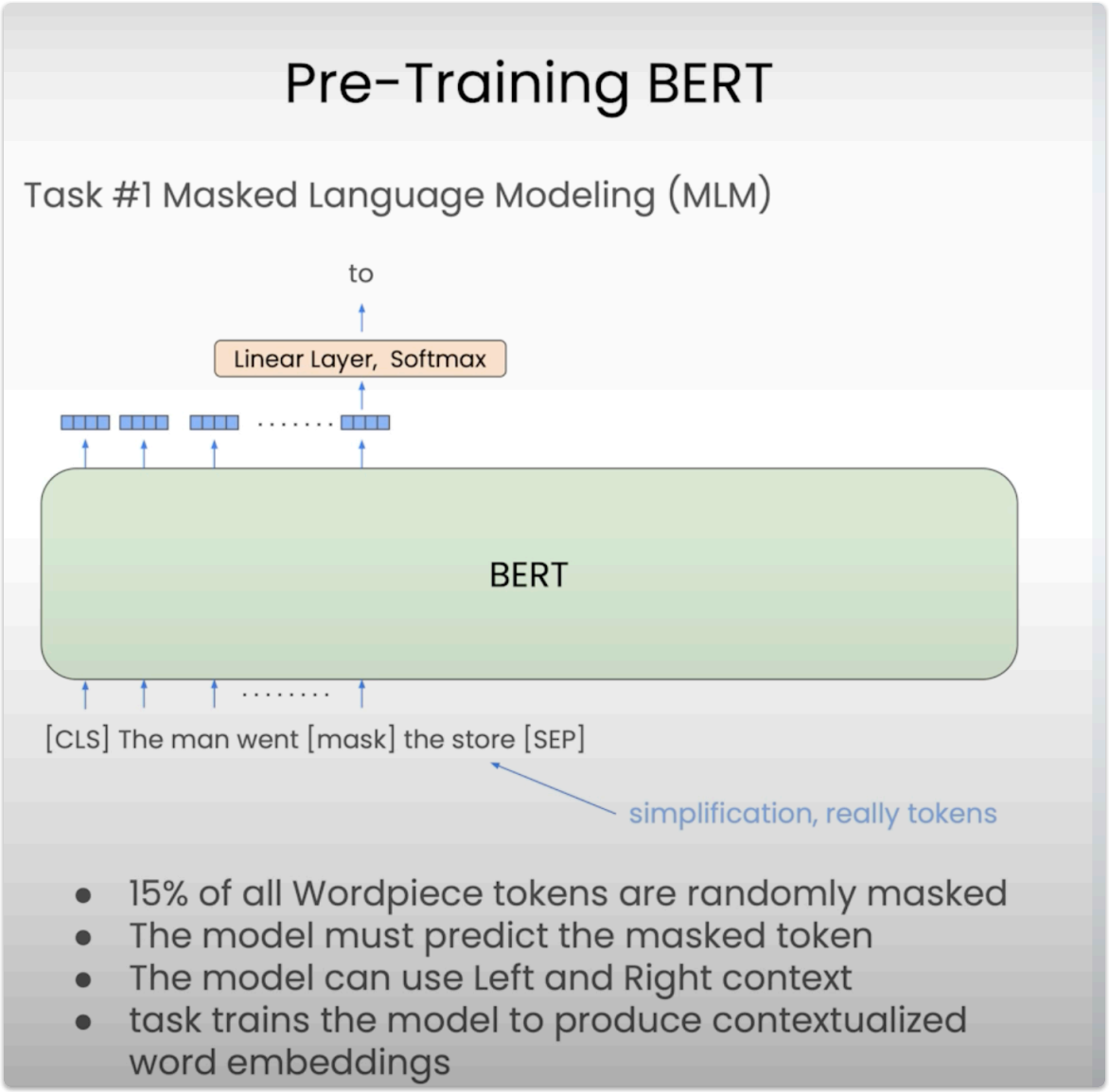
BERT BASE	BERT LARGE
Layers: 12	**Layers:** 24
Hidden Units: 768	**Hidden Units:** 1024
Attention Heads: 12	**Attention Heads:** 16
Parameters: 110 million	**Parameters:** 340 million

Trained on 3.3 Billion Words

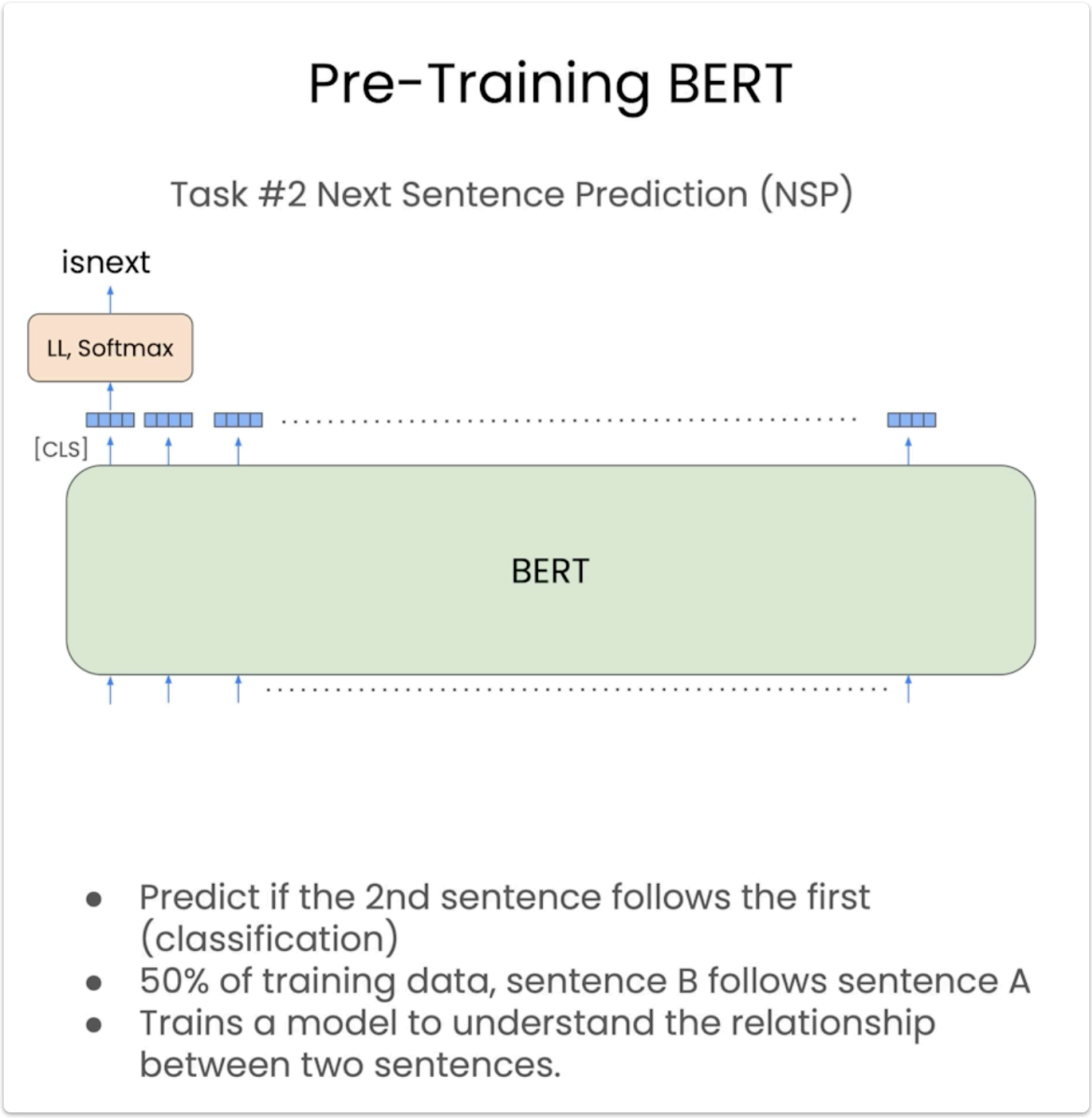
Two phases of training

- Pre-training
- fine-tuning

- The BERT was pre-trained on two tasks:
    1. Task #1 Masked Language Modeling (MLM)
        - The model tries to predict a masked (hidden) word in a sentence
        - This trains the model to produce contextualized word embeddings

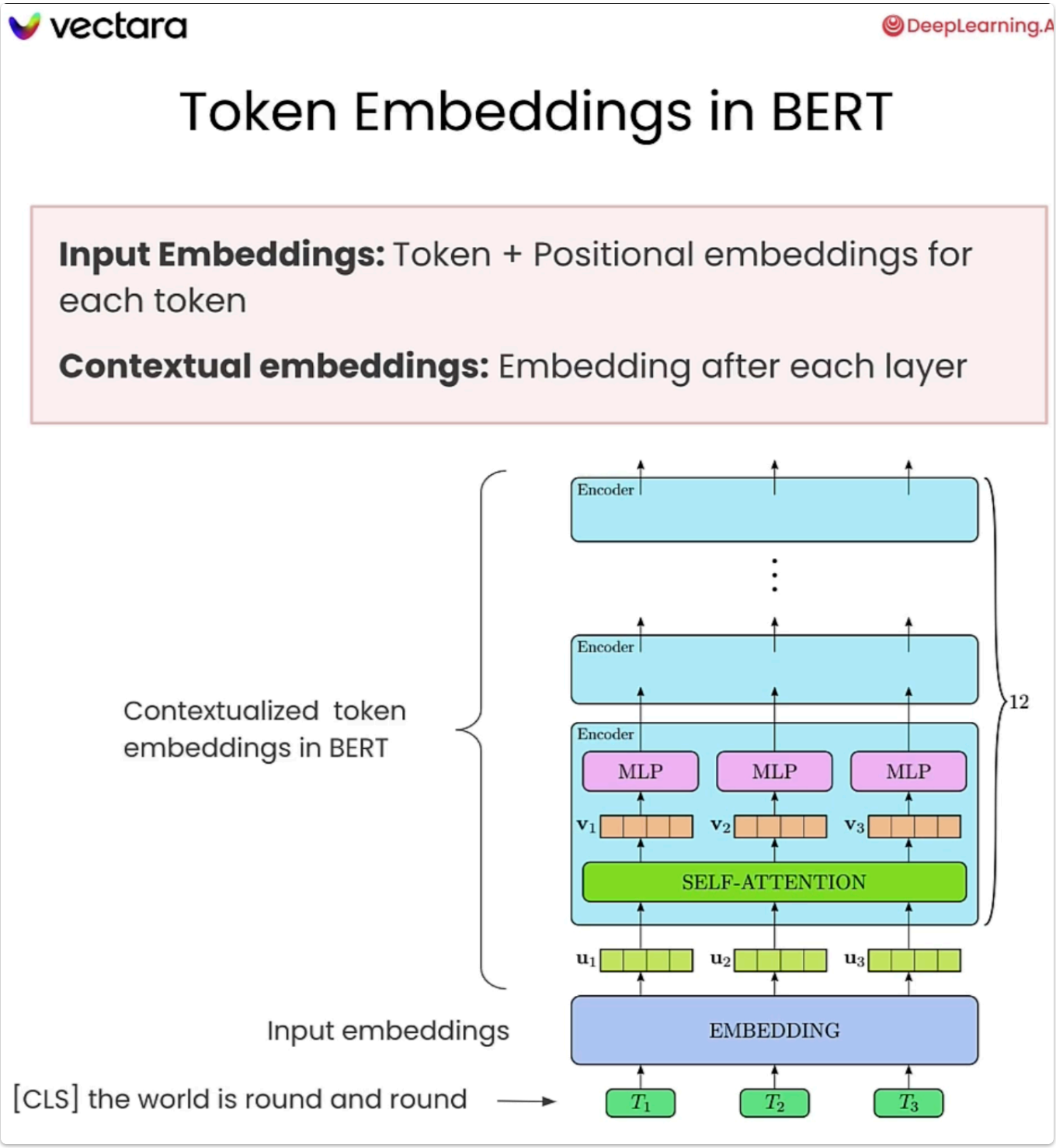


2. Task #2 Next Sentence Prediction (NSP)
    - Predict if the 2nd sentence follows the first sentence (classification T/F)
    - This trains the model to understand the relationship between two sentences



- After pre-training, we can **fine-tune** the model to adapt it to specific tasks. e.g,,
  - text classification
  - named entity recognition
  - question answering
  - reranking via a **cross encoder**, which takes in two sentences separated by a separate SEP token and classify whether they are semantically related or not. This task is called *semantic textual similarity (STS)*. This introduces a computational efficiency problem with many sentence pairs!
- The BERT model generates contextualized token embeddings, which represent multiple vectors that correspond to each token in the input. This doesn't generate a single embedding vector for a whole sentence.

Token Embeddings in BERT

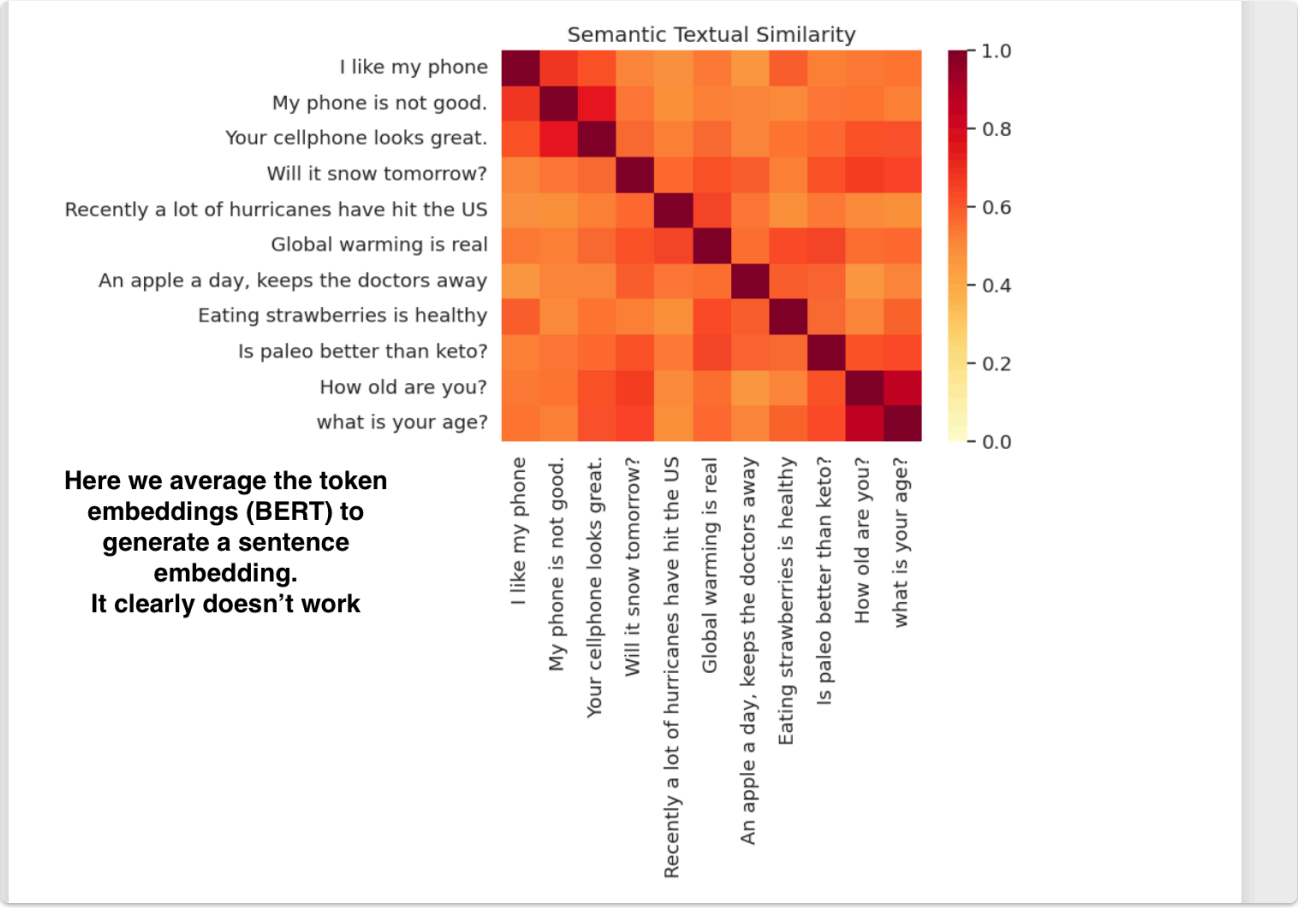


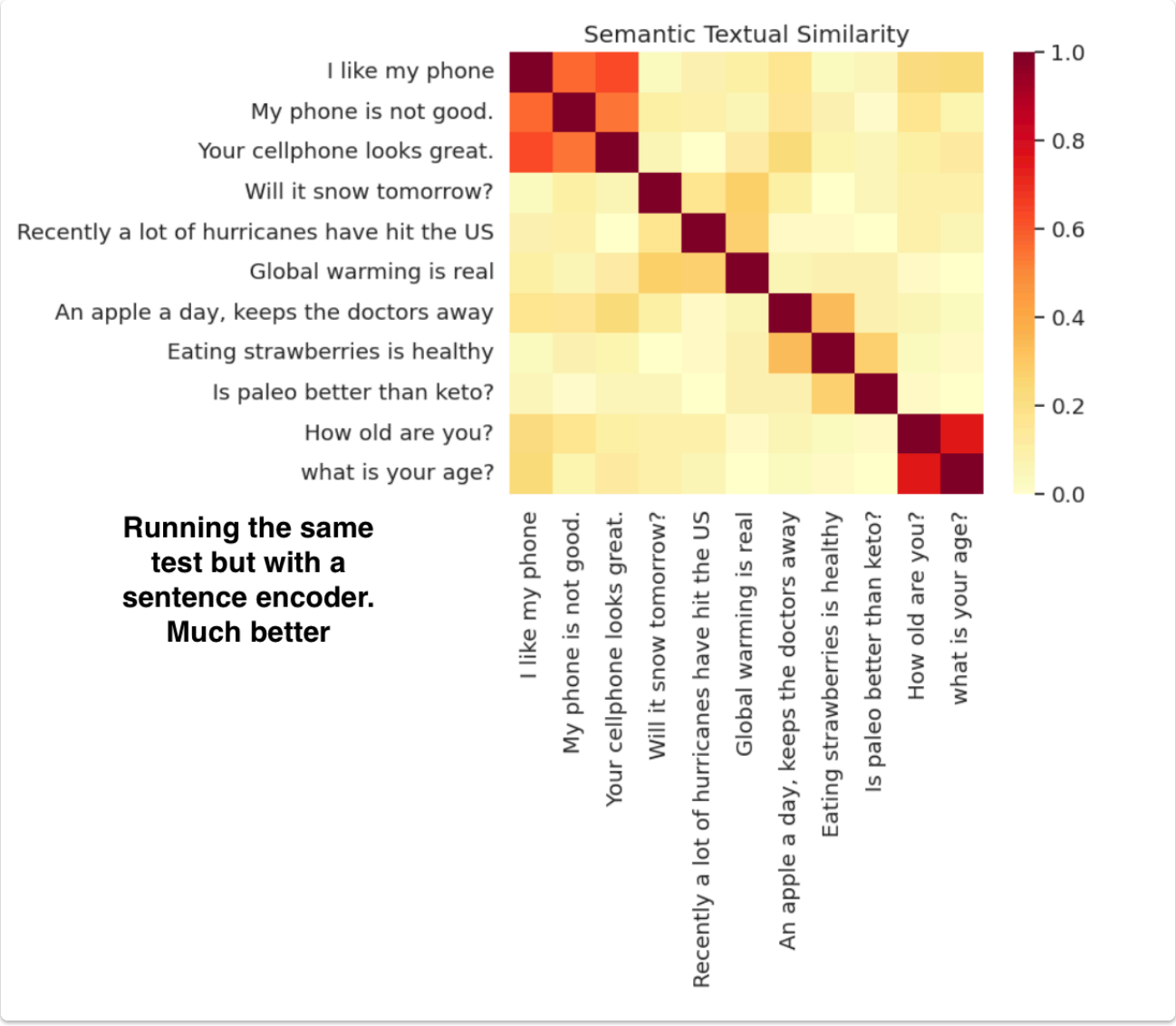
- The words are tokenized and prepended with the CLS token.
- The first layer generates a fixed embedding for each token
- The rest of the layers generate a contextualized token embedding

Sentence Embeddings

```
Recall our goal: We want to build an embedding model for a RAG system.
Therefore, contextualized token embeddings are not enough.
```

- Instead of token embeddings, we want sentence embeddings so that we can compare sentences in a semantic space separately (i.e., embedding of each sentence is generated independently)
- The naive approach was to take the output of the last layer in BERT and *average* the embeddings of each vector (mean pooling). However, this doesn't work and is inaccurate when testing and plotting similarities between sentences:





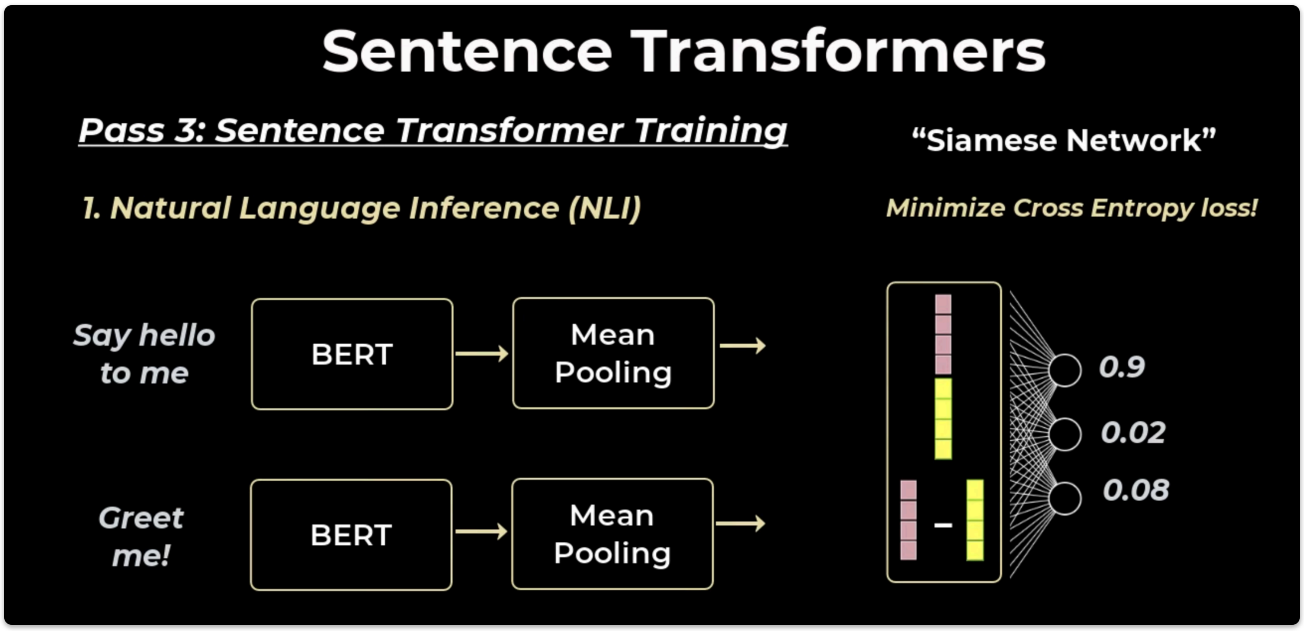
Sentence Transformers

```
SBERT or Sentence transformers was a major update to information retrieval, since it provided an easy to use interface for performing sentence-related NLP tasks
```

- The goal here is to generate a semantic representation of a sentence which can be compared with other generated representation independently
- Obtaining sentence embeddings with accurate semantic representation can be achieved with sentence tasks (fine-tuning):
  - Natural Language Inference (NLI)
  - Sentence Text Similarity (STS)
  - Triplet Dataset

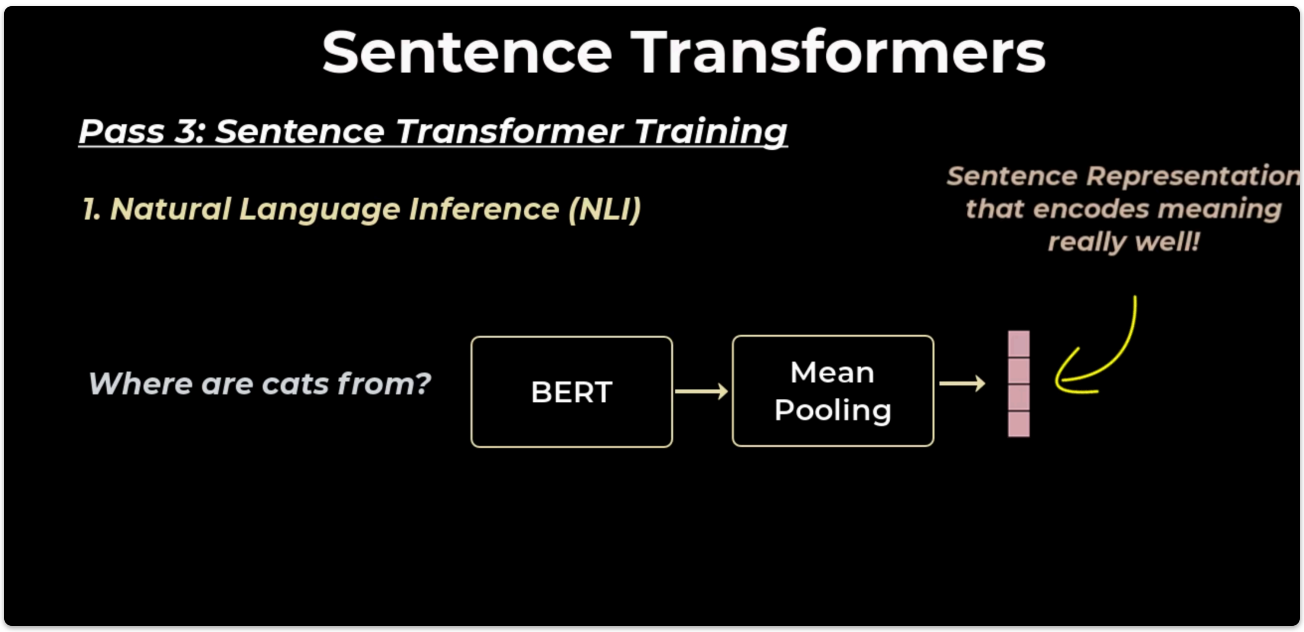
Natural Language Inference (NLI)

- The NLI task takes two sentences as input and check if sentence 1 entails or contradicts sentence 2, or is neutral
- This is done with a *Siamese Network*, which is a twin network used to learn the similarity between the inputs
- Training:



1. Each sentence is passed to one of the twin BERT networks
2. The output of BERT is averaged to obtain a sentence embedding X & Y for each sentence
3. X || Y and X - Y is passed to a softmax classifier that will label the pair sentences as entailment, contradiction, neutral

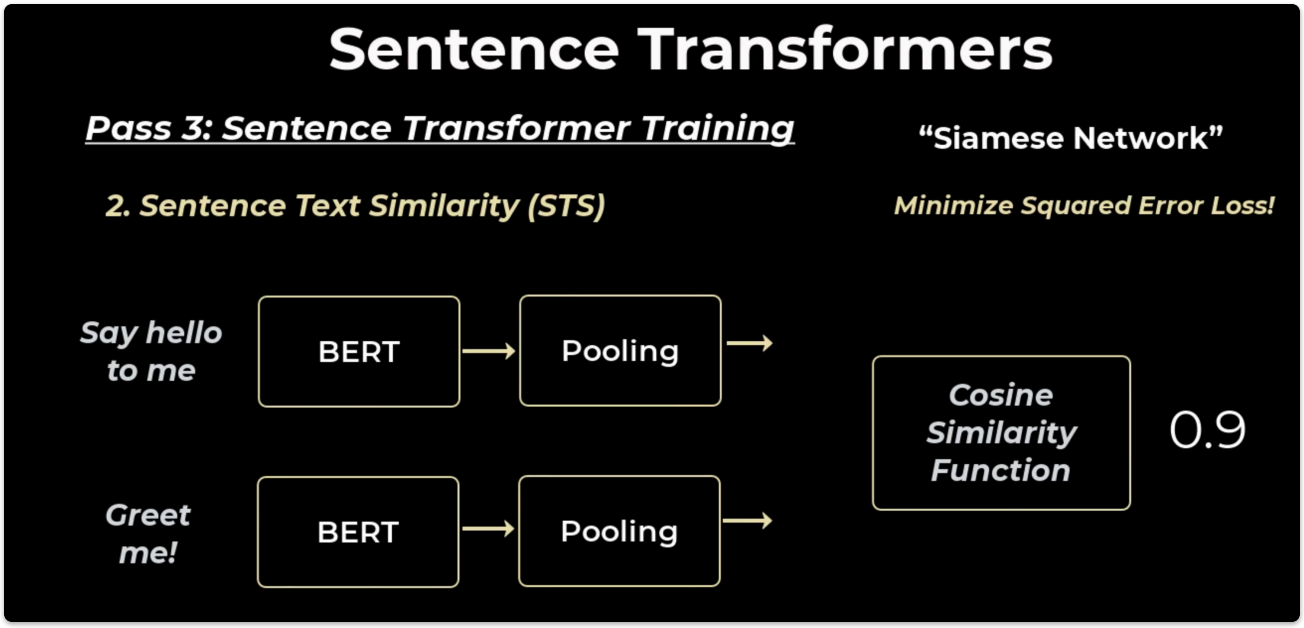
- Inference:



1. We only take one of the twin encoders and use it to obtain sentence embedding

Sentence Text Similarity (STS)

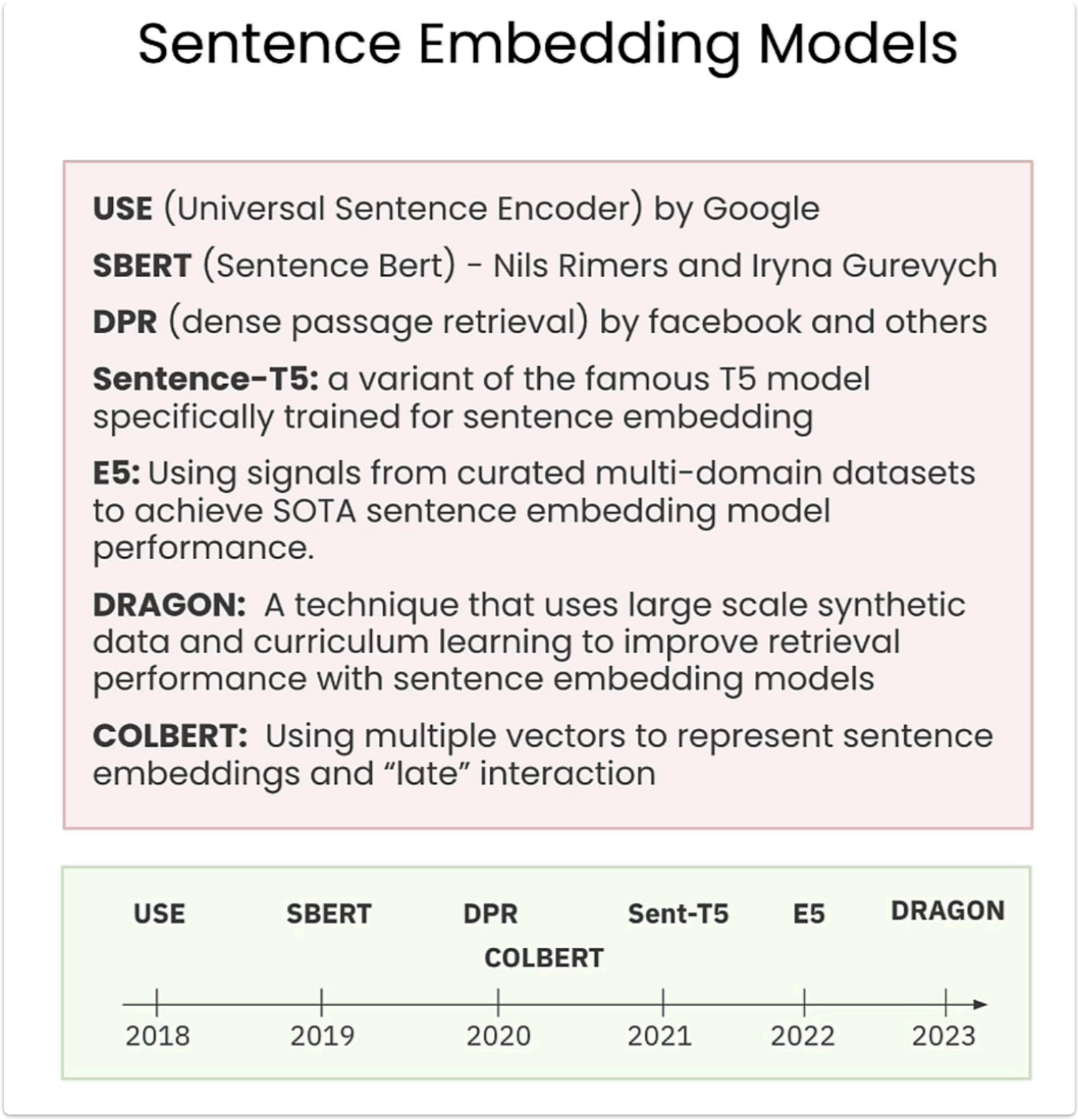
- The STS task takes a pair of sentences and output a value between 0 and 1, with 1 being very similar and 0 being not similar
- Similar to NLI but after the pooling step, we simply compute the cosine similarity and minimize the loss based on the true label:



Triplet Dataset

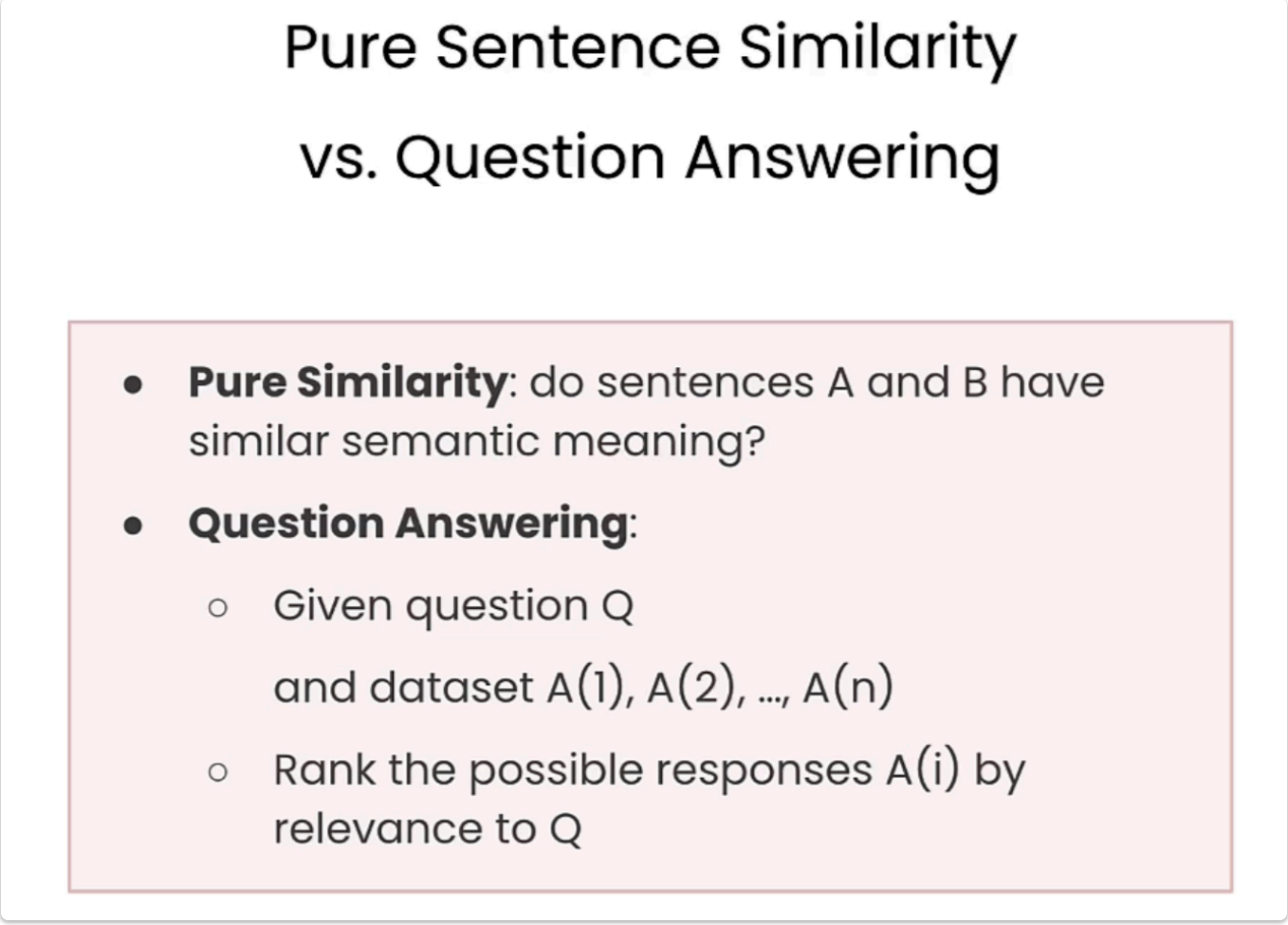
- In this task, we feed the network two related sentences and one unrelated sentence, and minimize the following: Given a triplet of (anchor, positive, negative), the loss minimizes the distance between anchor and positive while it maximizes the distance between anchor and negative.

Sentence Embedding Models History



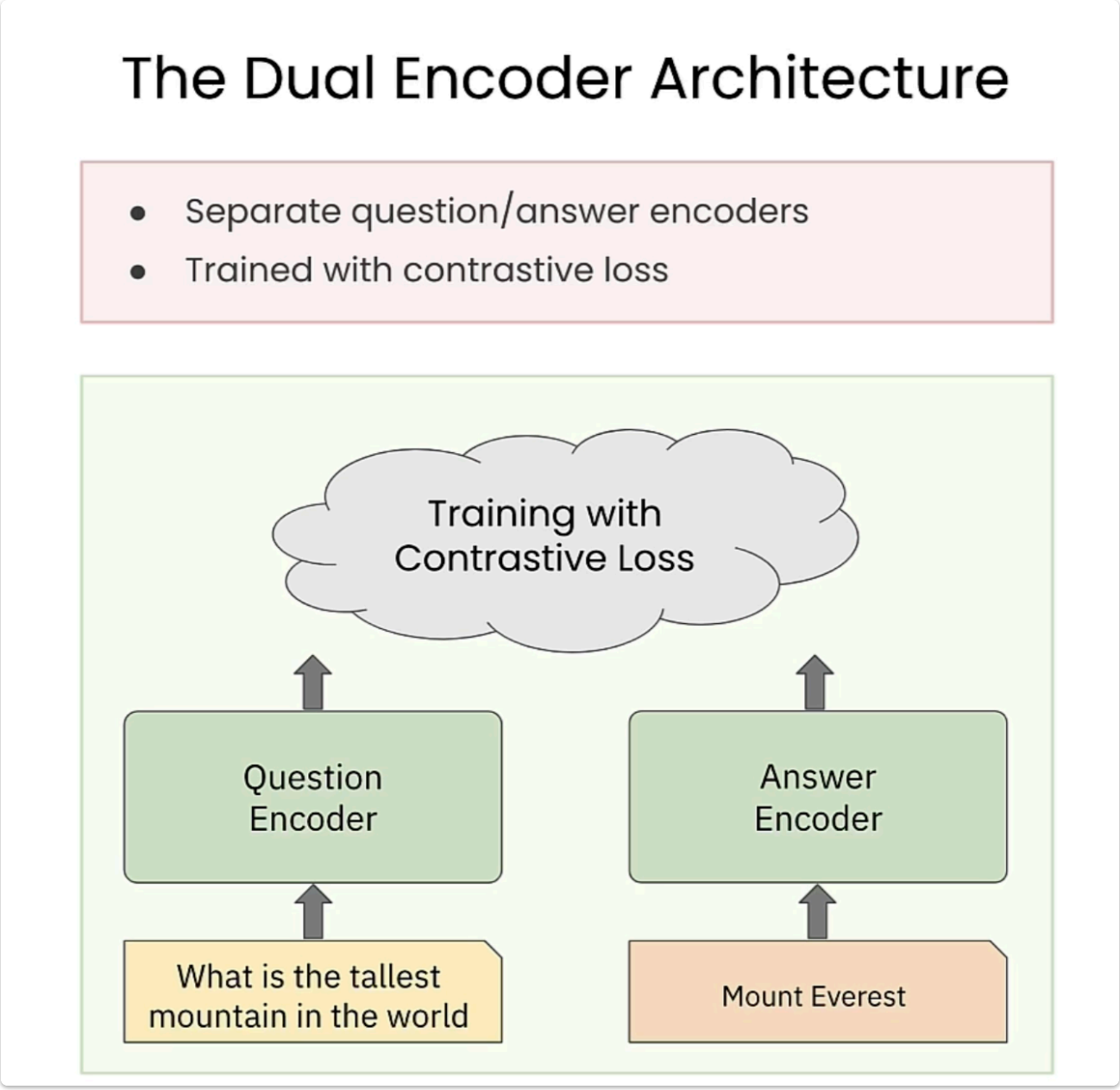
Sentence Similarity vs Question Answering

The goal from the start is to deep dive into the history and building blocks of embedding models. So far, we saw how BERT variants such as SBERT yielded good results in similarity tasks such as STS. However, the question-answering task is totally different since it involves retrieving **answers** and not any semantically similar sentences. This introduces the notion of dual encoders or bi-encoders

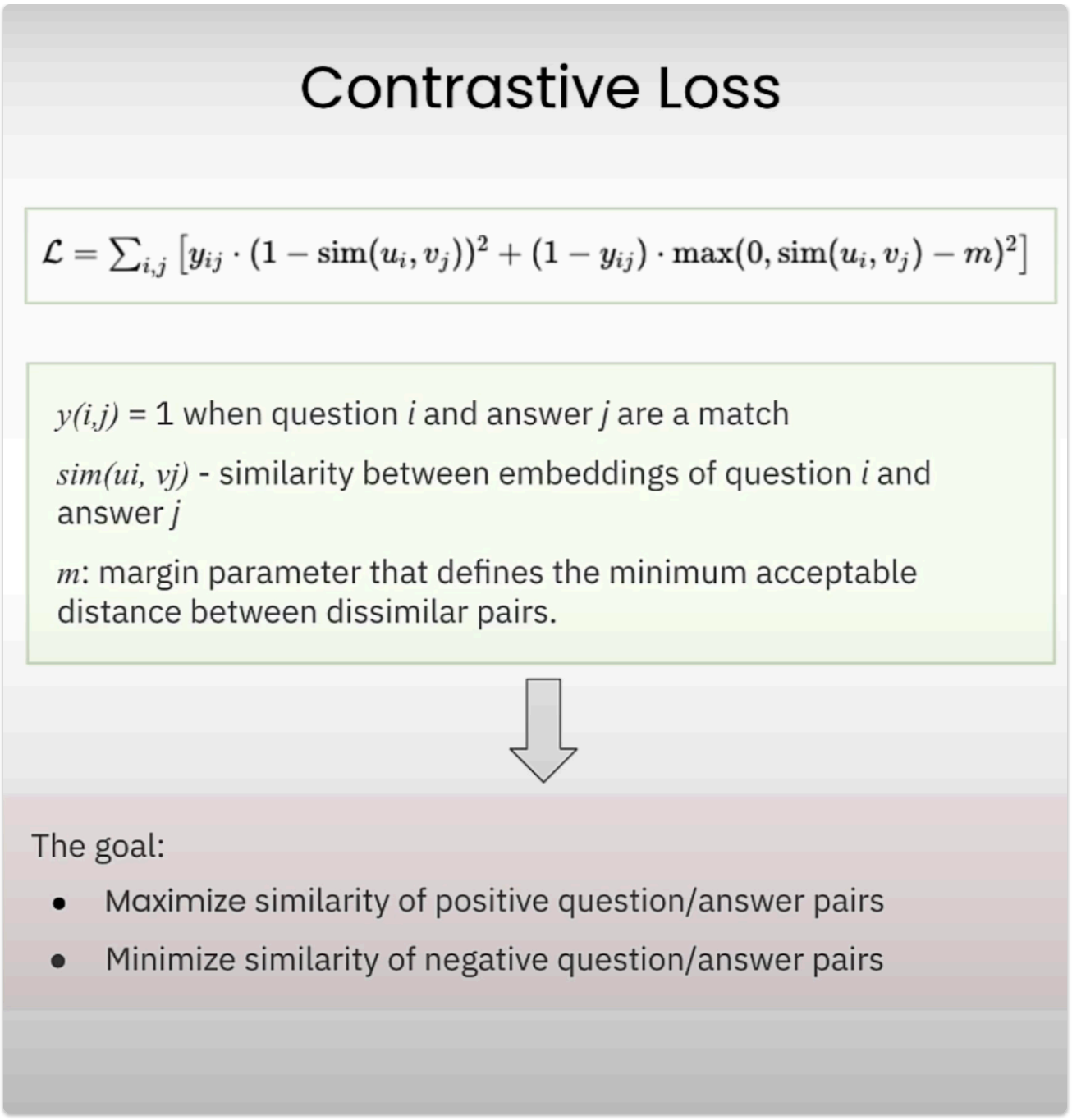


The Dual Encoder Architecture

- The *Dual Encoder or Bi-encoder* (Also called a two tower design) consists of a question encoder and an answer encoder
- The encoders are BERT encoders which outputs a sequence of token embeddings
- The [CLS] token embedding in the final layer is used as the sentence embedding
- The goal is to maximize the similarity of the question and answer embeddings if they are valid pair and to minimize the similarity if they are an invalid pair

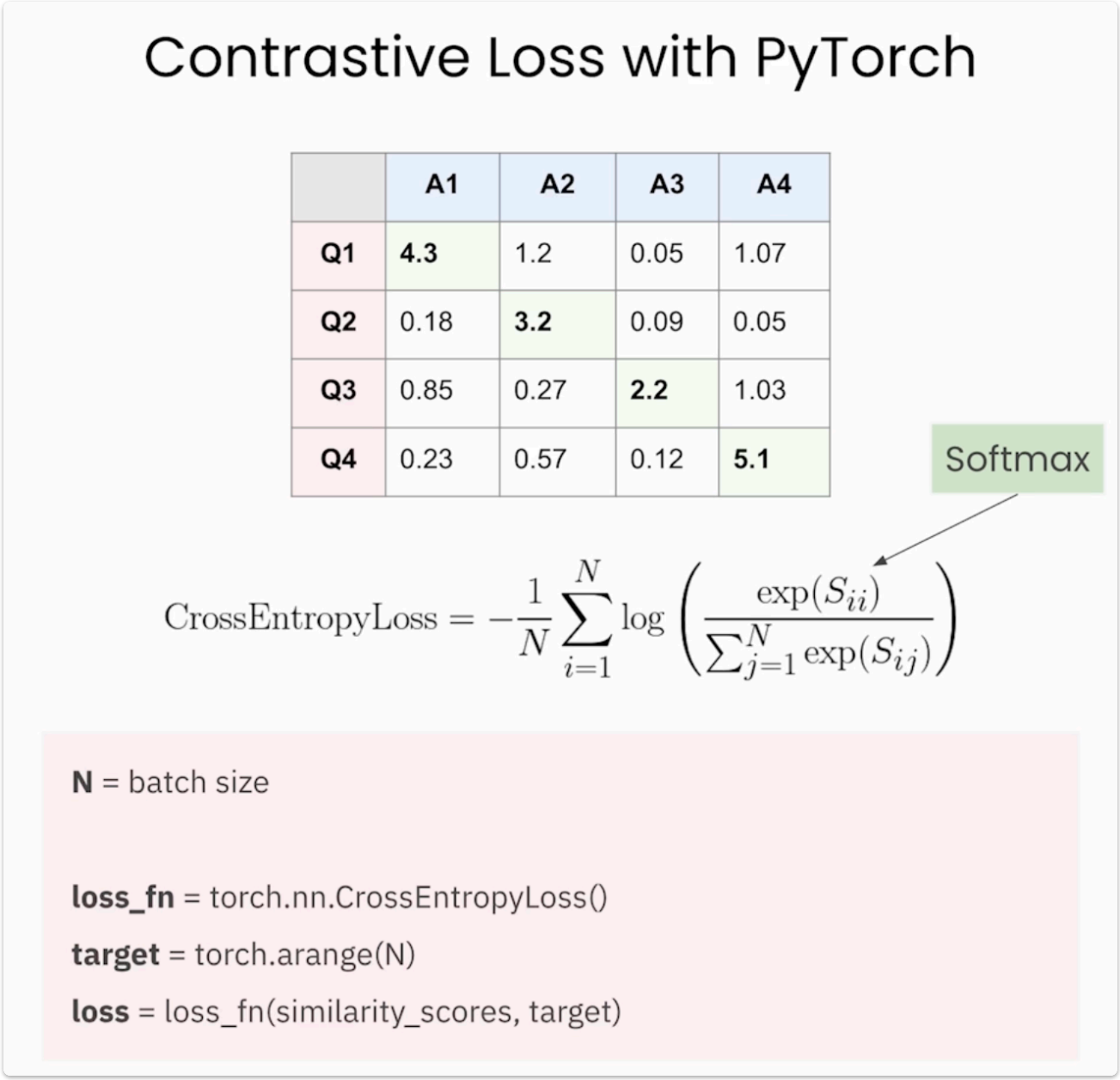


- **Contrastive Loss:** minimize distances between related items (positive pairs) while maximizing distances between unrelated ones (negative pairs) in an embedding space



- During training, we use the following cross-entropy loss for contrastive learning, looking only at the diagonal values which are positive pairs. When the numerator is high (good),

the loss will be closer to 0.

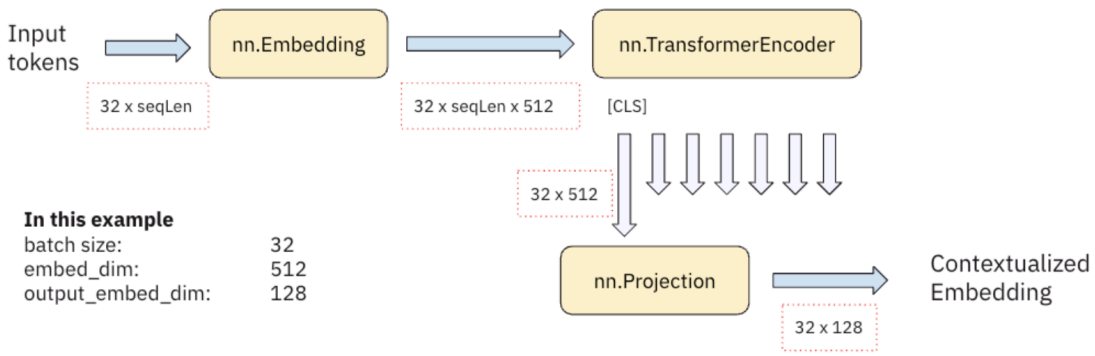


Training the dual encoder

- Basic Design:
  - Tokenize the input -> embedding for each token -> transformer embedding -> projection

The Encoder module.

```
In [8]: class Encoder(torch.nn.Module):
            def __init__(self, vocab_size, embed_dim, output_embed_dim):
                super().__init__()
                self.embedding_layer = torch.nn.Embedding(vocab_size, embed_dim)
                self.encoder = torch.nn.TransformerEncoder(
                    torch.nn.TransformerEncoderLayer(embed_dim, nhead=8, batch_first=True),
                    num_layers=3,
                    norm=torch.nn.LayerNorm([embed_dim]),
                    enable_nested_tensor=False
                )
                self.projection = torch.nn.Linear(embed_dim, output_embed_dim)

            def forward(self, tokenizer_output):
                x = self.embedding_layer(tokenizer_output['input_ids'])
                x = self.encoder(x, src_key_padding_mask=tokenizer_output['attention_mask'].logical_not())
                cls_embed = x[:,0,:]
                return self.projection(cls_embed)
```



- Training:
  - we define token embedding size (embed_size) and the final embedding size (outputs_embed_size) and a batch size

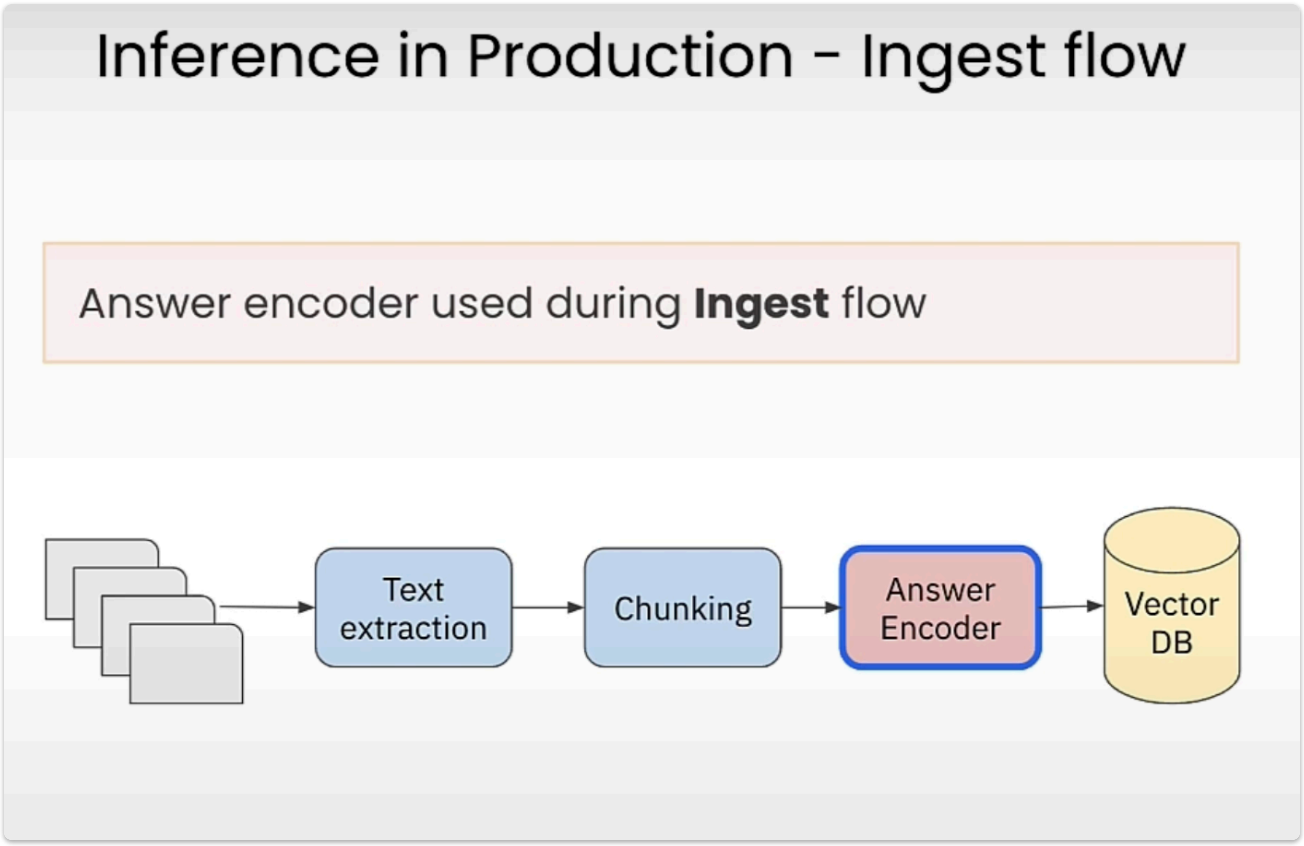
Training in multiple Epochs

In [10]:

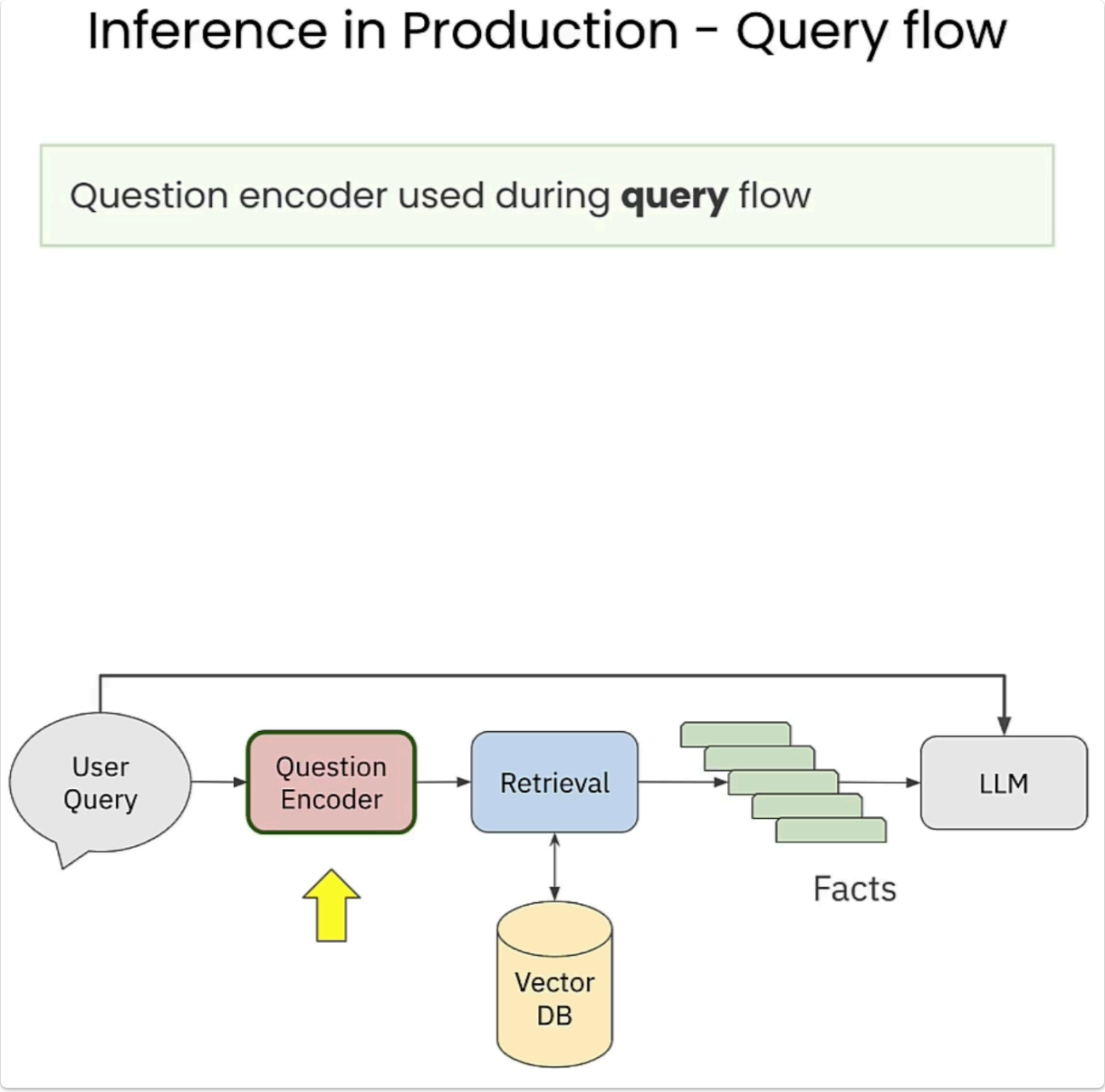
```python
def train(dataset, num_epochs=10):
    embed_size = 512
    output_embed_size = 128
    max_seq_len = 64
    batch_size = 32

    n_iters = len(dataset) // batch_size + 1

    # define the question/answer encoders
    tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
    question_encoder = Encoder(tokenizer.vocab_size, embed_size, output_embed_size)
    answer_encoder = Encoder(tokenizer.vocab_size, embed_size, output_embed_size)

    # define the dataloader, optimizer and loss function
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)
    optimizer = torch.optim.Adam(list(question_encoder.parameters()) + list(answer_encoder.parameters()), lr
    loss_fn = torch.nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        running_loss = []
        for idx, data_batch in enumerate(dataloader):

            # Tokenize the question/answer pairs (each is a batc of 32 questions and 32 answers)
            question, answer = data_batch
            question_tok = tokenizer(question, padding=True, truncation=True, return_tensors='pt', max_lengt
            answer_tok = tokenizer(answer, padding=True, truncation=True, return_tensors='pt', max_length=ma
            if inx == 0 and epoch == 0:
                print(question_tok['input_ids'].shape, answer_tok['input_ids'].shape)

            # Compute the embeddings: the output is of dim = 32 x 128
            question_embed = question_encoder(question_tok)
            answer_embed = answer_encoder(answer_tok)
            if inx == 0 and epoch == 0:
                print(question_embed.shape, answer_embed.shape)

            # Compute similarity scores: a 32x32 matrix
            # row[N] reflects similarity between question[N] and answers[0...31]
            similarity_scores = question_embed @ answer_embed.T
            if inx == 0 and epoch == 0:
                print(similarity_scores.shape)

            # we want to maximize the values in the diagonal
            target = torch.arange(question_embed.shape[0], dtype=torch.long)
            loss = loss_fn(similarity_scores, target)
            running_loss += [loss.item()]
            if idx == n_iters-1:
                print(f"Epoch {epoch}, loss = ", np.mean(running_loss))

            # this is where the magic happens
            optimizer.zero_grad()    # reset optimizer so gradients are all-zero
            loss.backward()
            optimizer.step()
    return question_encoder, answer_encoder
```

Dual Encoder in Production

- Document Ingest Flow



- Query Flow


Inference in Production - Query flow
Question encoder used during **query** flow
User Query
Question Encoder
Retrieval
Facts
LLM
Vector DB

- Search

Inference in Production

Similarity Search

Neural/Vector search

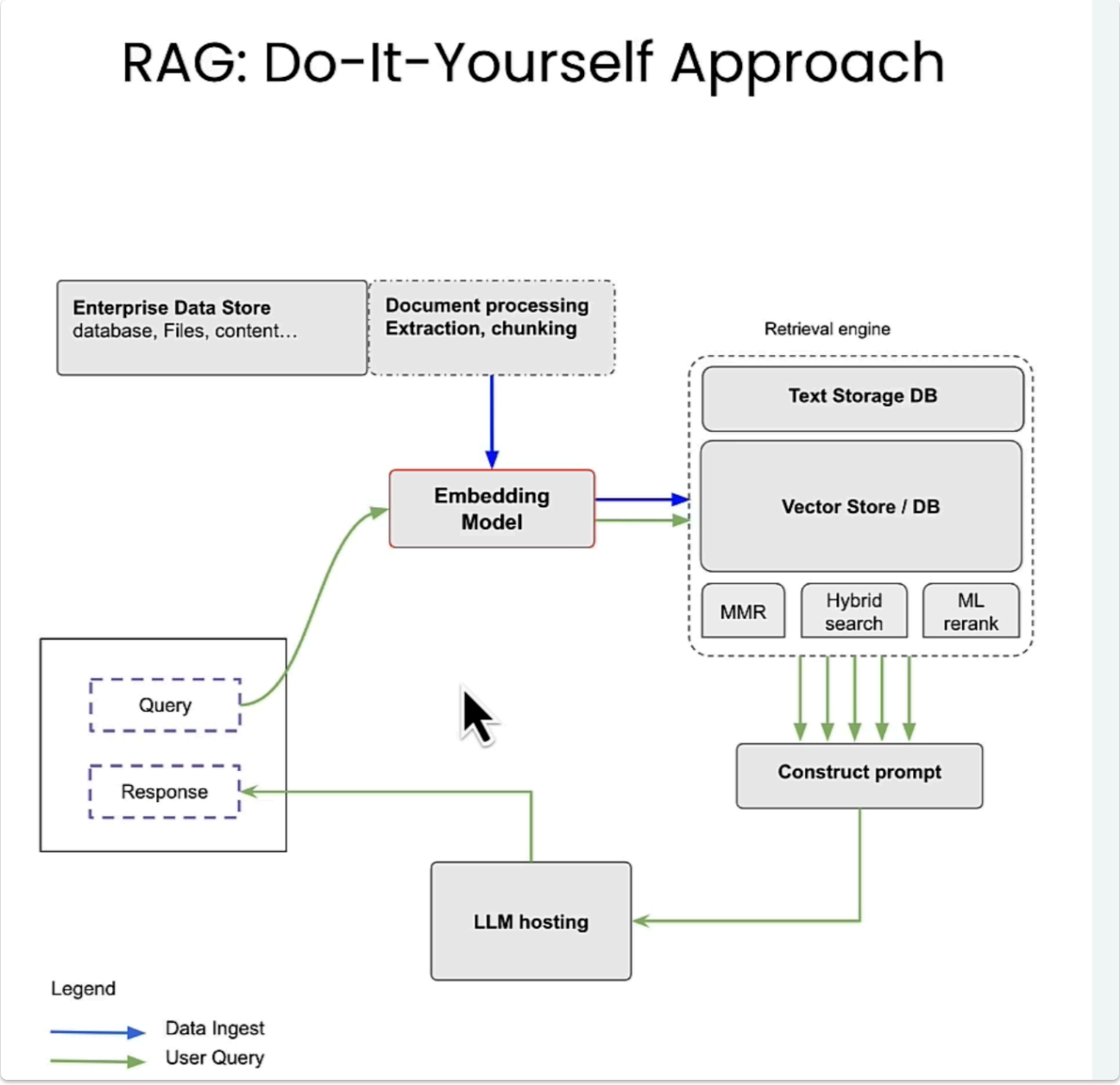
Can't compute similarity against ALL chunks:
`sim(q,a(i)), i=1, …, 1M`

ANN (approximate Nearest Neighbor) to the rescue:

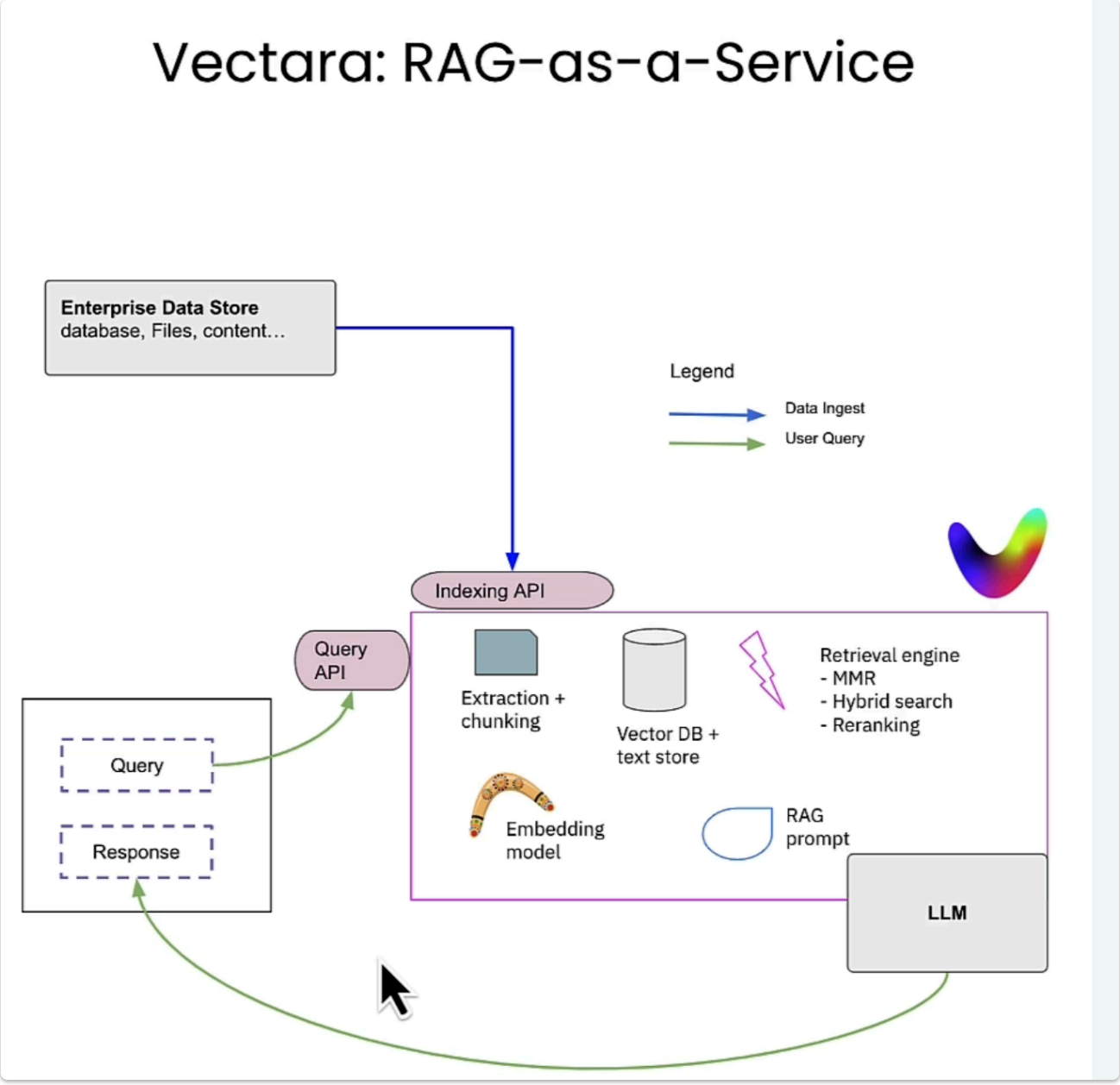
- HNSW, Annoy, FAISS and NMSLib are all algorithms and systems that implement ANN

In production - you need add **persistence**, moving the implementation of these algorithms from in memory to on-disk Vector stores.

- System Design (By yourself)

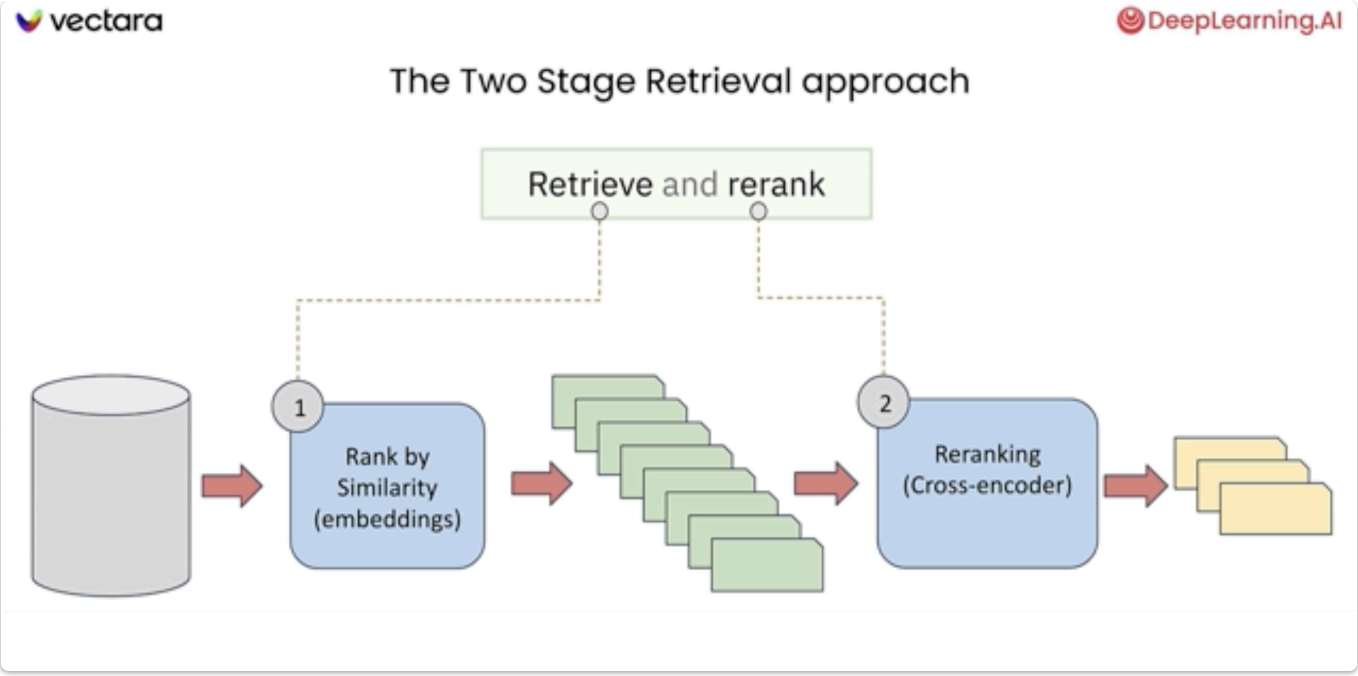


- RAG-as-a-service Design(e.g., Vectara)

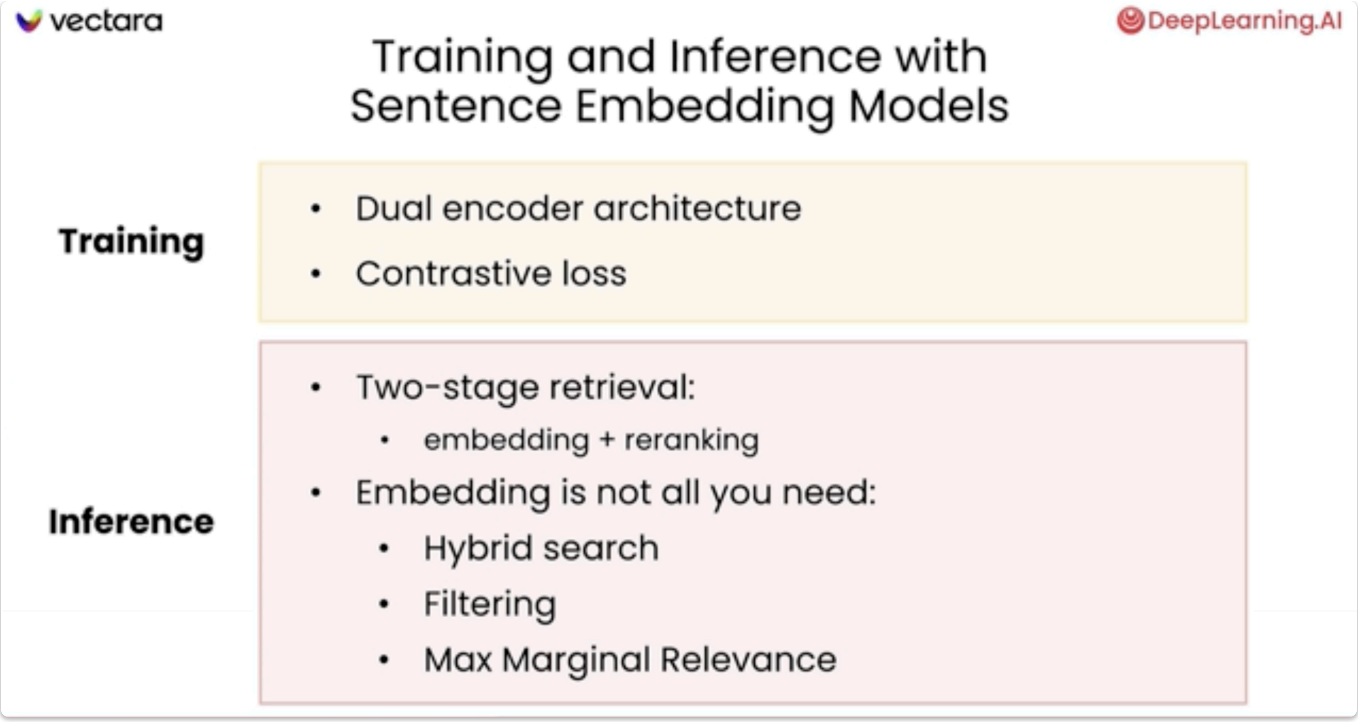


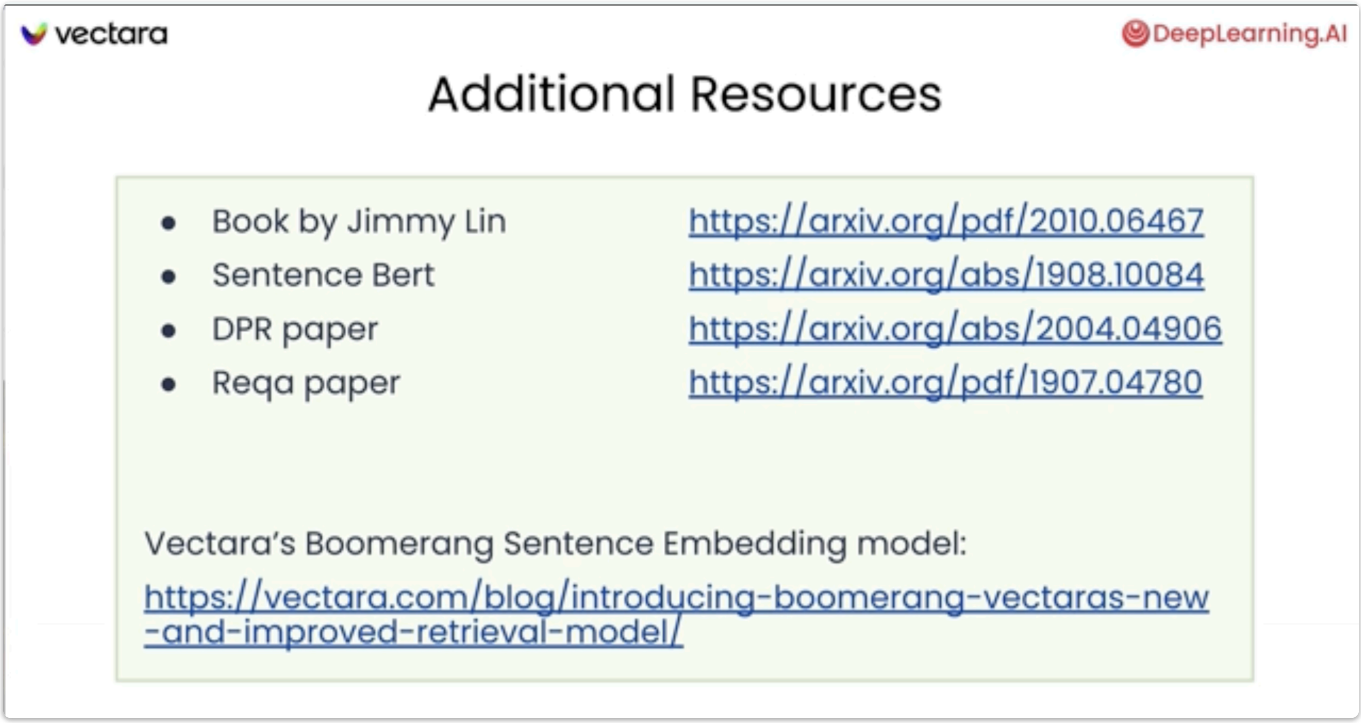
Two Stage Retrieval

```
Retrieve and rerank
```


vectara
DeepLearning.AI
The Two Stage Retrieval approach
Retrieve and rerank
1
Rank by Similarity (embeddings)
2
Reranking (Cross-encoder)

What's next





References:

- deeplearning.ai/courses/embedding-models-from-architecture-to-implementation
- https://www.youtube.com/watch?v=O3xbVmpdJwU&t=340s-
- BERT Paper
- SBERT Paper