Building Blocks of GPT-2 LLM

Introduction

This tutorial is designed to

- Explore the architecture of LLMs with a special focus on GPT-2 model
- Introduce fundamental components of LLM (i.e., building blocks of LLM architecture)
- Provide overall understanding of basic LLM architecture and link to resources that help readers gain deeper comprehension of each of the component

Objective

- Understand why each building block (key component) of a LLM important
- Understand how these building blocks work
- Understand the dataflow and data parallelization of LLMs

At the end of the tutorial, learners will gain knowledge to interpret the processes that enable LLMs to predict the next word from a sequence of input words.

⚙ Prerequisites

Prerequisites

- Basic understanding of Deep learning concepts and methods
- Python programming
- Basic understanding of PyTorch implementation

Introduction to Large Language Models (LLMs)

❗ Objectives

Understand:

- Why the filed of language modeling needed LLMs?
- What are LLMs?
- How do LLMs differ from other machine learning approaches?

Language modeling (LM)

- LM was introduced in early 1980s with the introduction of Recurrent Neural Networks (RNNs)
- With the advances in the field of LM, more advance techniques to RNNs were introduced to
  - preserve gradients and maintain information (1997-2014; Gating mechanisms)
  - handle long-term memory (2015; Attention for RNNs)
  - manage variable-length input output sequences (2014; Encoder-decoder for RNNs)

Why does the LM field needs LLMs?

- RNNs process inputs sequentially and the attention mechanism was not build into the core architecture
- RNNs are slow and lead to scalability challenges
- Transformer technology introduced in the paper "Attention Is All You Need" addressed this limitations in Modern RNNs
- Transformer technology (therefore LLMs) eliminate sequential dependency:
  - all positions can be computed in parallel
  - Scalable model training and inference

LLMs

Transformer-based neural networks with large number of parameters (billions to trillions) that employ self-attention mechanisms and trained on vast amounts data (billions to trillions of tokens)

LLMs vs other ML approaches

- Behavior of traditional ML approaches are specifically tied to the training objectives
- LLMs exhibits capabilities that were not explicitly trained
  - i.e., Simple training objectives lead to complex capabilities
  - e.g., LLM's ability to translate despite never being specifically trained for translation
- These capabilities are referred "Emergent Behavior" of LLMs
- This complexity emerges from:
  - Large scale + rich data + powerful architecture
  - Emergent mechanism is still not fully understood

GPT - Generative Pretrained Transformer model

Objectives

Understand:

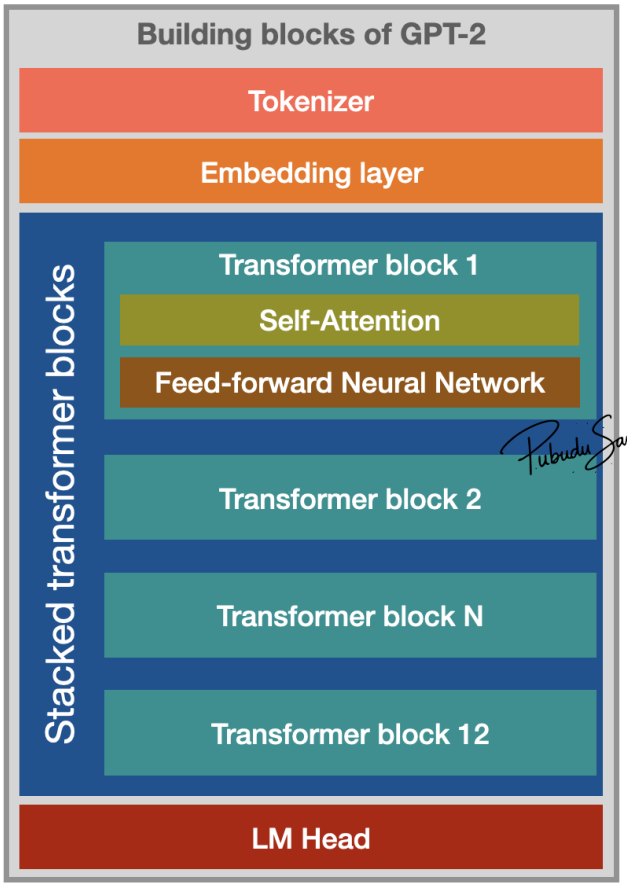
- What is GPT?
- What are the main components (building blocks) of GPT-2 model?

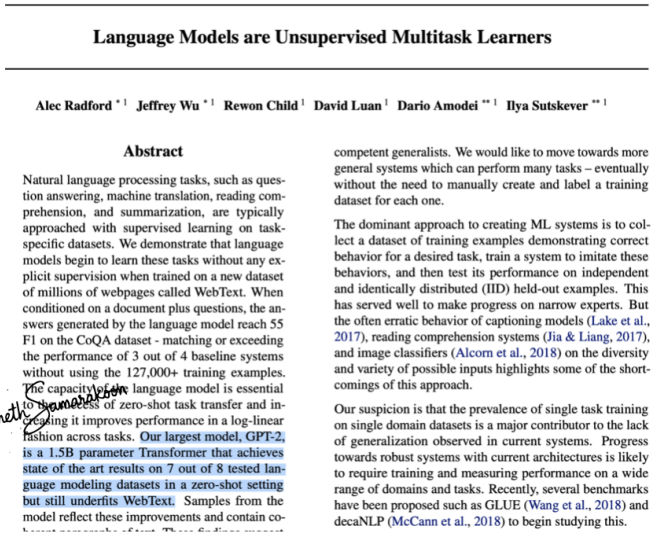
- **Generative**: The model can generate tokens auto-regressive manner (generate one token at a time)

- **Pretrained**: Trained on a large corpus of data
- **Transformer**: The model architecture is based on the transformer, introduced in the 2017 paper "Attention is All You Need" (Self-Attention Mechanism)

GPT-2

- Original publication: "Language Models are Unsupervised Multitask Learners"
- GPT-2 original publication lists four models
  - Smallest GPT-2 model:
    - 17 million parameters; 12 transformer blocks; Model dimensions: 768
  - Largest GPT-2 model:
    - 1542 million parameters; 48 transformer blocks; Model dimensions: 1600





Language Models are Unsupervised Multitask Learners

Alec Radford [*1] Jeffrey Wu [*1] Rewon Child [1] David Luan [1] Dario Amodei [**1] Ilya Sutskever [**1]

Abstract

Natural language processing tasks, such as question answering, machine translation, reading comprehension, and summarization, are typically approached with supervised learning on task-specific datasets. We demonstrate that language models begin to learn these tasks without any explicit supervision when trained on a new dataset of millions of webpages called WebText. When conditioned on a document plus questions, the answers generated by the language model reach 55 F1 on the CoQA dataset - matching or exceeding the performance of 3 out of 4 baseline systems without using the 127,000+ training examples. The capacity of the language model is essential to the success of zero-shot task transfer and increasing it improves performance in a log-linear fashion across tasks. Our largest model, GPT-2, is a 1.5B parameter Transformer that achieves state of the art results on 7 out of 8 tested language modeling datasets in a zero-shot setting but still underfits WebText. Samples from the model reflect these improvements and contain co-

competent generalists. We would like to move towards more general systems which can perform many tasks – eventually without the need to manually create and label a training dataset for each one.

The dominant approach to creating ML systems is to collect a dataset of training examples demonstrating correct behavior for a desired task, train a system to imitate these behaviors, and then test its performance on independent and identically distributed (IID) held-out examples. This has served well to make progress on narrow experts. But the often erratic behavior of captioning models (Lake et al., 2017), reading comprehension systems (Jia & Liang, 2017), and image classifiers (Alcorn et al., 2018) on the diversity and variety of possible inputs highlights some of the shortcomings of this approach.

Our suspicion is that the prevalence of single task training on single domain datasets is a major contributor to the lack of generalization observed in current systems. Progress towards robust systems with current architectures is likely to require training and measuring performance on a wide range of domains and tasks. Recently, several benchmarks have been proposed such as GLUE (Wang et al., 2018) and decaNLP (McCann et al., 2018) to begin studying this.

https://github.com/openai/gpt-2

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

Architecture hyperparameters for the four GPT-2 model sizes as shown in the original paper

Key components of GPT-2 model

- Tokenizer
- Embedding layer
- Transformer block
  - Self-attention layer
  - Feedforward neural network
- Language modeling head

Introduction to tokenization

Objectives

Understand:

- What is tokenization?
- Why it is important?
- How tokenizer process input test

Tokenizers

- What is tokenization?
  - Tokenizers take text as input and generate a sequence of integer representations of input text
- Why it is important?
  - This serves as the foundation for converting text to numerical representations that deep learning models can process

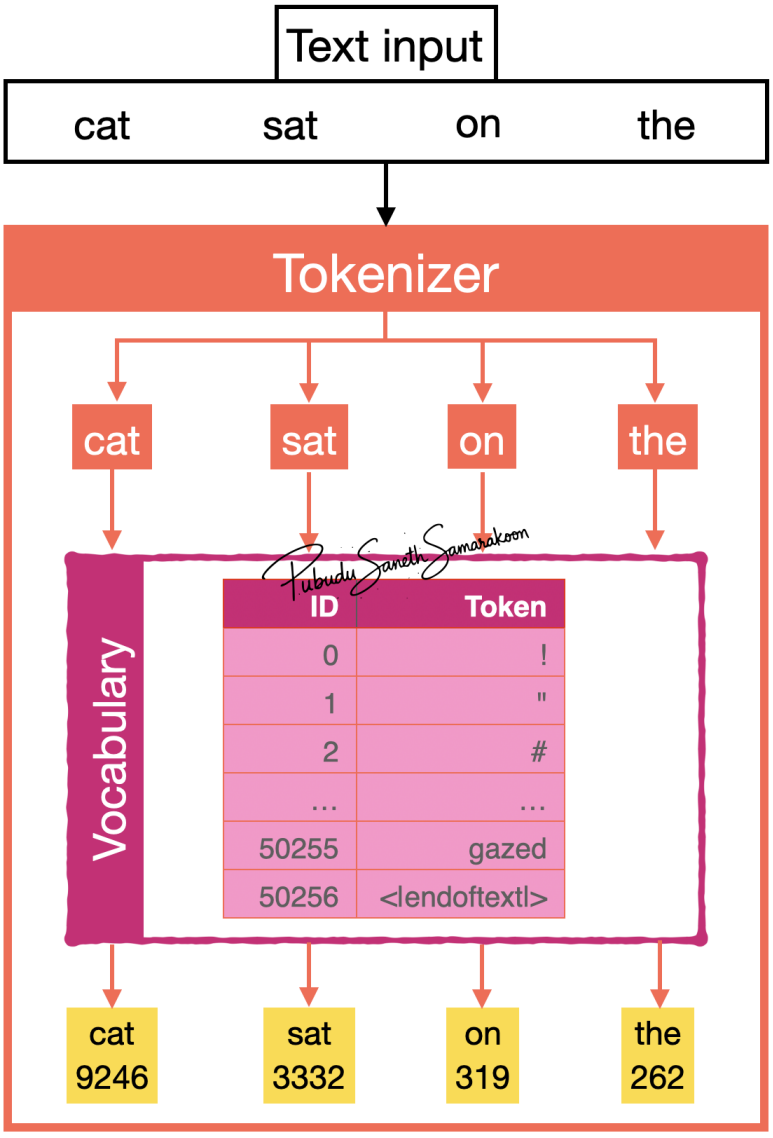
Text to sub-word units

- Tokenizers process these input text by converting them into discrete sub-word units
  - i.e, Split input test in to discrete sub-word units
- These “discrete sub-word units” are tokens
- Token are mapped to corresponding Token IDs using the model vocabulary

Tokenization step-by-step

1. Tokenizer accepts text `cat sat on the` as input
2. Split test into words: `cat`, `sat`, `on`, `the`
    1. Tokenizer split text into sub-words. In this case, sub-words from the tokenizer are equal to words in the text
3. Tokenizer uses model’s vocabulary as a lookup table to map tokens to integer IDs
    1. Vocabulary: A dictionary of unique tokens and unique numerical identifiers assigned to each token (Token ID)
    2. This provides a consistent mapping system that converts variable-length text into fixed numerical representations
4. Return corresponding token-IDs of the tokens from input text

Vocabulary is build from training data by mapping each unique token to a token ID, with special tokens added to handle unknown words and document boundaries, enabling LLMs to process diverse text inputs effectively. The vocabulary size is managed to balance expressiveness with computational efficiency



GPT-2 tokenizer

Text to Token-IDs

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
print(f"Length of the vocabulary: {len(tokenizer)}")

sentence = "cat sat on the"
token_ids = tokenizer.encode(sentence)
print(f"Token IDs of the sentence '{sentence}': {token_ids}")

decoded_sentence = tokenizer.decode(token_ids)
print(f"Decoded sentence: {decoded_sentence}")
```

Text to subword units

```
summarization_token_ids = tokenizer.encode("summarization")
print("Token IDs for word `summarization`:", summarization_token_ids)

print("Mapping of tokens to token IDs:")
for token_id in summarization_token_ids:
    print(f"'{tokenizer.decode(token_id)}' -> {token_id}")
```

Output

▶ Text to Token-IDs

▶ Text to subword units

Introduction to embedding

Objectives

Understand:

- What is embedding?
- Why it is important?
- How does the embedding layer in LLMs process tokens?

Token embedding

What is token embedding?

- Token embedding is the process of converting discrete tokens (specifically token-IDs) into vectors
- These vectors can be represented in a high-dimensional space
  - Token ID -> vector with length `N` (points in N-dimensional space)
- Representing tokens in a high-dimensional space enables to effectively capture complex patterns and relationships

What token embedding is important?

- Token-IDs are just arbitrary integers assigned during tokenization that do not have mathematical relationship between these integers
  - Tokens are discrete numerical labels with no geometric or relational structure
- Token embedding convert these numerical labels to structured representations - vectors (points in a high-dimensional space) in a way that captures the relationships between tokens
- In this high-dimensional space, semantically related tokens like “dog”, “cat”, “animal” cluster together

- This structure is learned during model training process so that words with similar contextual roles have similar vector representations (similarity between vectors represent relationships between tokens)

👀 Demo

Explore token embeddings (Word2Vec embeddings):

- Word2Vec embedding was widely used before the introduction of LLM technology

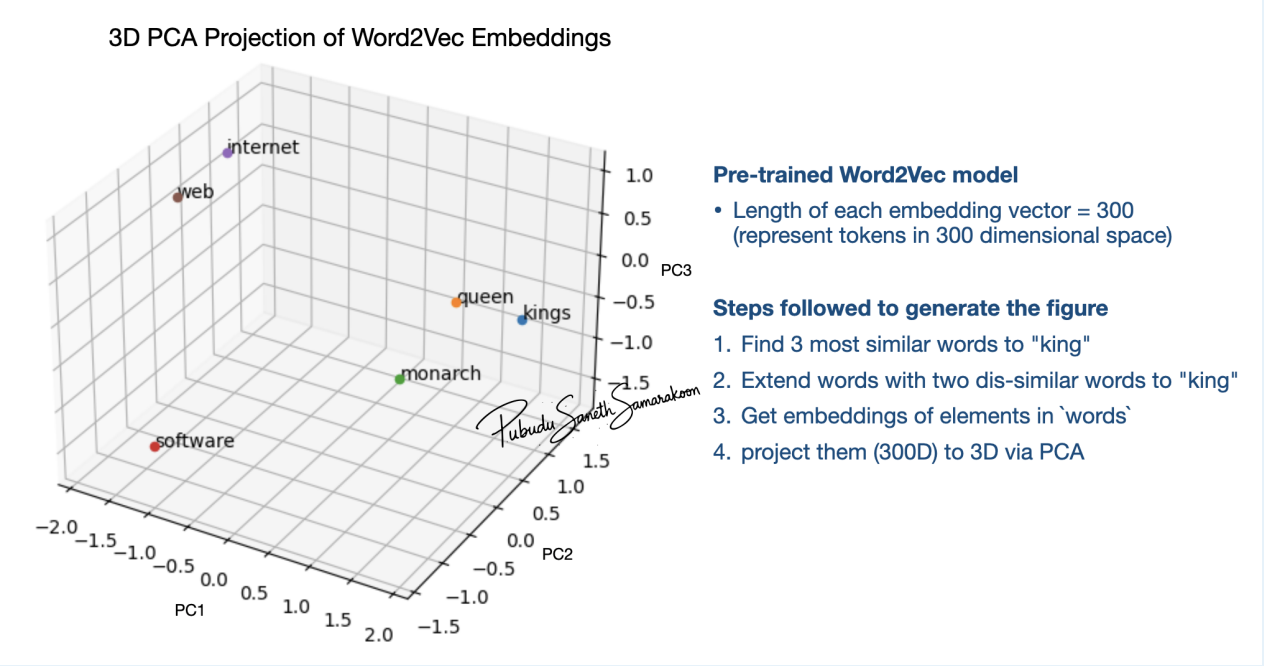


Figure shows that the words with similar roles in natural language cluster together when represented in high-dimensional space

▶ Python implementation

Token embeddings in LLMs

- Tokenizer in GPT-2 smallest model has a vocabulary of 50257 tokens
- This GPT-2 tokenizer maps tokens to integers 0-50256 with no mathematical relationship in those assignments
- For example, tokenizer maps input - `cat sat on the` to tokens `[9246, 3332, 319, 262]` that do not have inherent relationship between these numbers themselves
- Token embeddings convert these arbitrary Token-IDs into dense vectors in a continuous space of 768 dimensions
- Now `cat sat on the` aren't just different numbers—they're points in a high-dimensional space where the similarity between vectors has meaning
- Capturing semantic meaning through token embeddings is learned during LLM pre-training process (detailed later)

GPT-2 Token embeddings: Step-by-Step Breakdown

1. Initiate a learnable matrix (dimensions `[vocab_size × embedding_dim]`)
   - Each row represents one token's embedding vector
2. Initiate the matrix with small random values (e.g., -2.84 to 1.58) to break symmetry
   - if all embeddings were identical, tokens couldn't differentiate during training
3. As the model processes training examples, it makes predictions using these random embeddings
4. Prediction errors generate gradients that flow back through the network to the embedding layer
5. Token embeddings are optimize through backpropagation (Tokens appearing in similar contexts receive similar updates)
6. Through thousands of iterations in the pre-training process, random vectors evolve into meaningful representations where "cat" and "dog" cluster together
7. The final optimized embeddings encode semantic relationships learned entirely from the training data patterns

Explore LLM token embeddings

Embedding Dimensions:

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained("gpt2",)

# Access the word token embedding layer
wte = model.transformer.wte

# Get vocabulary size and embedding dimension
print(f"Vocabulary Size: {wte.num_embeddings}; Embedding Dimension:
{wte.embedding_dim}")

# The embedding matrix is stored in the 'weight' attribute
print(f"Shape of the embedding matrix: {wte.weight.shape}")
```

LLM Embedding of made-up words:

```
text_rand = "rand_xyz"
rand_token_ids = tokenizer.encode(text_rand)
print(f"Token IDs for text '{text_rand}': {rand_token_ids}")
print(f"Decoded text: {tokenizer.decode(rand_token_ids)}")
print()

# Use evaluation mode and not gradient calculation (training)
with torch.no_grad():
    rand_token_embeddings = wte(torch.tensor(rand_token_ids))
print(f"Shape of the random token embeddings: {rand_token_embeddings.shape}")
print()

for i in range(len(rand_token_ids)):
    print(f"Token {i}: {tokenizer.decode(rand_token_ids[i])} ->
{rand_token_ids[i]};\n\tEmbeddings (first 5): {rand_token_embeddings[i][:5]}")
```

Output

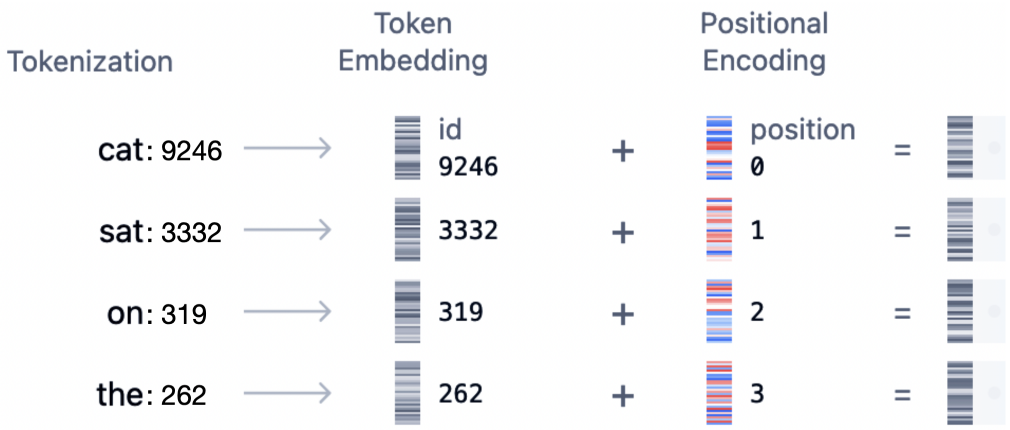
▶ Embedding Dimensions
▶ LLM Embedding of made-up words

Position embeddings

- Token embeddings process (converting unstructured token-ids to dense vectors) help capture relationships between tokens
- Token embeddings process treats all positions equally
- Token embedding alone makes models unable to distinguish token order without position information
  - Unable to distinguish between “dog bites man” and “man bites dog”
- Position embeddings injects position information to embedding vectors

Embedding layer of LLMs

- Token embeddings convert discrete token IDs into vector representations through a learnable matrix
- Positional embeddings added to inject sequence order information to LLM embeddings
- Embedding vectors (combined token and position embeddings) are parses to the next layer of the LLM - transformer layer/block



Source: transformer-explainer

👀 Demo

Token and position embeddings:

Steps in the following script:

- Tokenize input text `bank is near the river bank`
- Pass input to token and position embeddings
- Calculate the similarity between 1st and last token

```
import torch

text_1 = "bank is near the river bank"
token_ids = tokenizer.encode(text_1)

for i in range(0,len(token_ids)):
    print(f"Token {i}: {tokenizer.decode(token_ids[i])} -> {token_ids[i]}")

wte = model.transformer.wte # Token embedding
wpe = model.transformer.wpe # Position embedding

bank_1_id = token_ids[0]
wte_bank_1 = wte(torch.tensor(bank_1_id))
wpe_bank_1 = wpe(torch.tensor(0))

bank_2_id = token_ids[-1]
wte_bank_2 = wte(torch.tensor(bank_2_id))
wpe_bank_2 = wpe(torch.tensor(1))


cosine_sim = torch.nn.functional.cosine_similarity(
    wte_bank_1 + wpe_bank_1,
    wte_bank_2 + wpe_bank_2,
    dim=0
)
print(f"Similarity: {cosine_sim.item():<.4f}")  # Less than 1.0 - they're
different!
```

- The contextualization happens through the Transformer layers of the LLM by updating vector embeddings
- Transformer update vectors from embedding layer to reflect the attention patterns that capture semantic relationships like “river” → “bank” (as in riverbank)

Output

▶ LLM Embedding

Transformer blocks

Objectives

Gain a basic understanding of:

- Transformer technology and why it is important?
- Transformer block and its main components?

Limitations in traditional LM

- RNN based traditional LM failed to track long-range dependencies like understanding how a word at the start of a paragraph relates to one at the end
- RNN based models that processed words one by one (not scalable)

- **Ambiguity Resolution**: Can't differentiate specific linguistic problems like determining what "it" refers to in several sentences

Transformer technology

- Transformer technology was introduced in the paper "Attention Is All You Need" to address several limitations in RNN based language modeling (LM)

Limitations and solutions

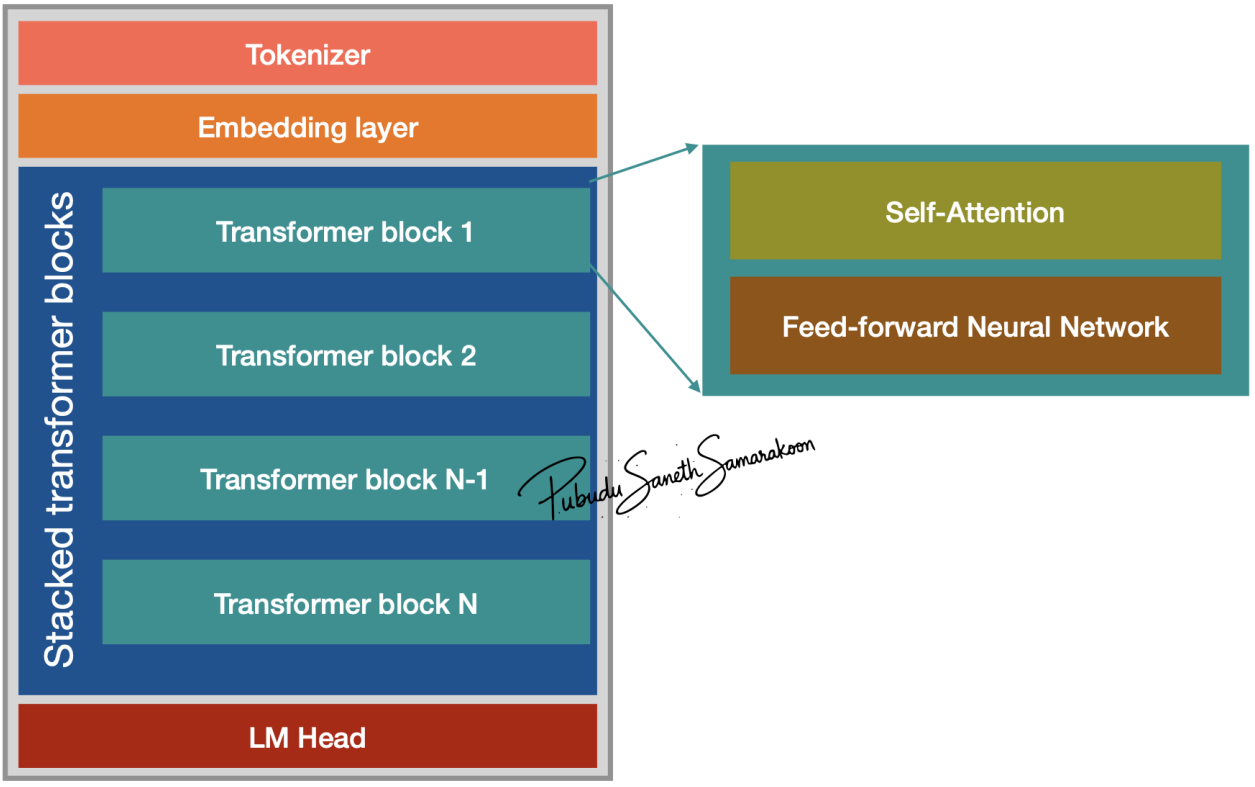
Limitation	Solution
Long-range dependencies	Contextual Understanding via self-attention mechanism
not scalable	Parallel Processing of tokens
specific linguistic problems	Ambiguity Resolution via self-attention mechanism

Why Is Transformer technology important?

- Scalability:
  - Allows for massive scaling (in terms of parameters and training data size)
  - Leading to the "Large" in LLMs.
- Architectural versatility:
  - The same underlying transformer block architecture is used across various state-of-the-art models (like GPT, Llama, and BERT)
- Versatility performance/behaviour:
  - Enables models to generate coherent, contextually appropriate text and perform a wide range of tasks—from translation to coding—that were previously impossible for computers
  - Effective for both understanding and generating human language

What is a Transformer block?

- Transformer block is the fundamental architectural unit of a LLMs
- LLMs - constructed by stacking these blocks on top of one another
  - Each block processes the input it receives from the previous layer and passes the result to the next
  - Stacked transformer blocks progressively refining the model's understanding of the text



Main Components of a transformer block

- Attention mechanism
- Feed Forward neural Network

Self-attention mechanism

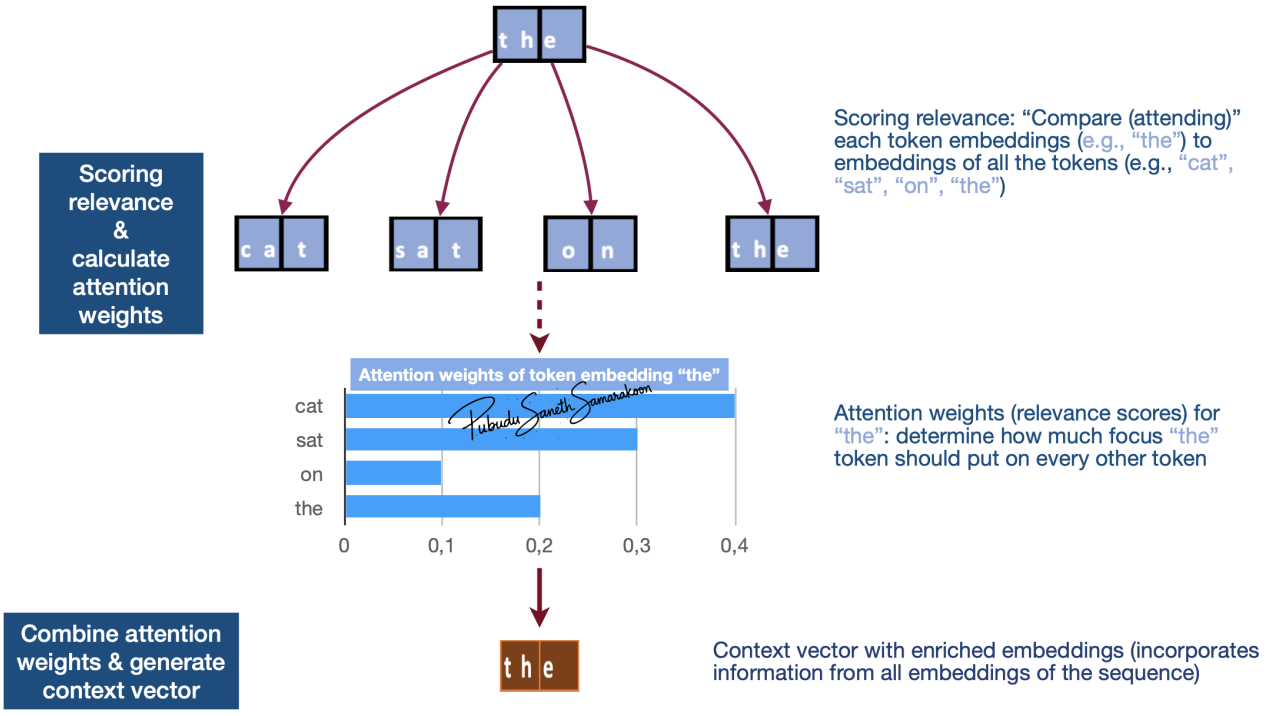
Objectives

Gain a basic understanding of:

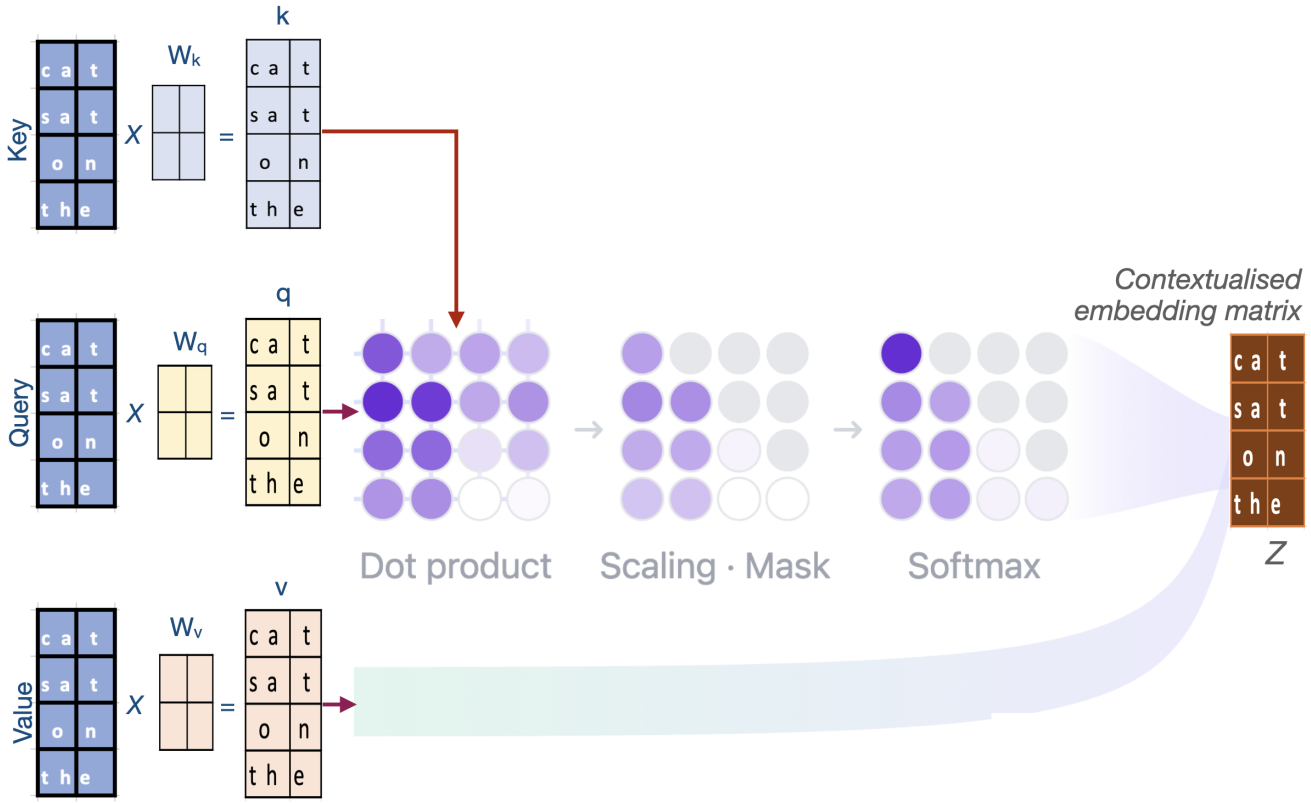
- Self-attention mechanism
- How attention weights are calculated & context vector is generated?

What is self-attention mechanism?

- Self-attention: create a new, enriched representation (context vector) by incorporating information from all token embeddings in the sequence
- Two main steps mechanism:
  1. Scoring relevance (“attending to”/”considering” all tokens) & calculate attention weights (relevance scores)
  2. Combine attention weights and generate context vector (new enriched representation)
- Context vector (enriched representation):
  - Captures the specific meaning of a token embeddings within its surrounding embeddings
  - Allow the model to understand relationships and dependencies between words, regardless of how far apart they are in the sentence



Self-attention with Q, K, V weight matrix



Source (modified): transformer-explainer

- Q, K and V: matrices: Representation of input token embeddings
- Q_{matrix}: Queries
  - Token representations that are used as queries in relevance scoring (embeddings that are used as queries for the “comparison”)
- K_{matrix}: Keys
  - Token representations that get compared to queries
- V_{matrix}: Values

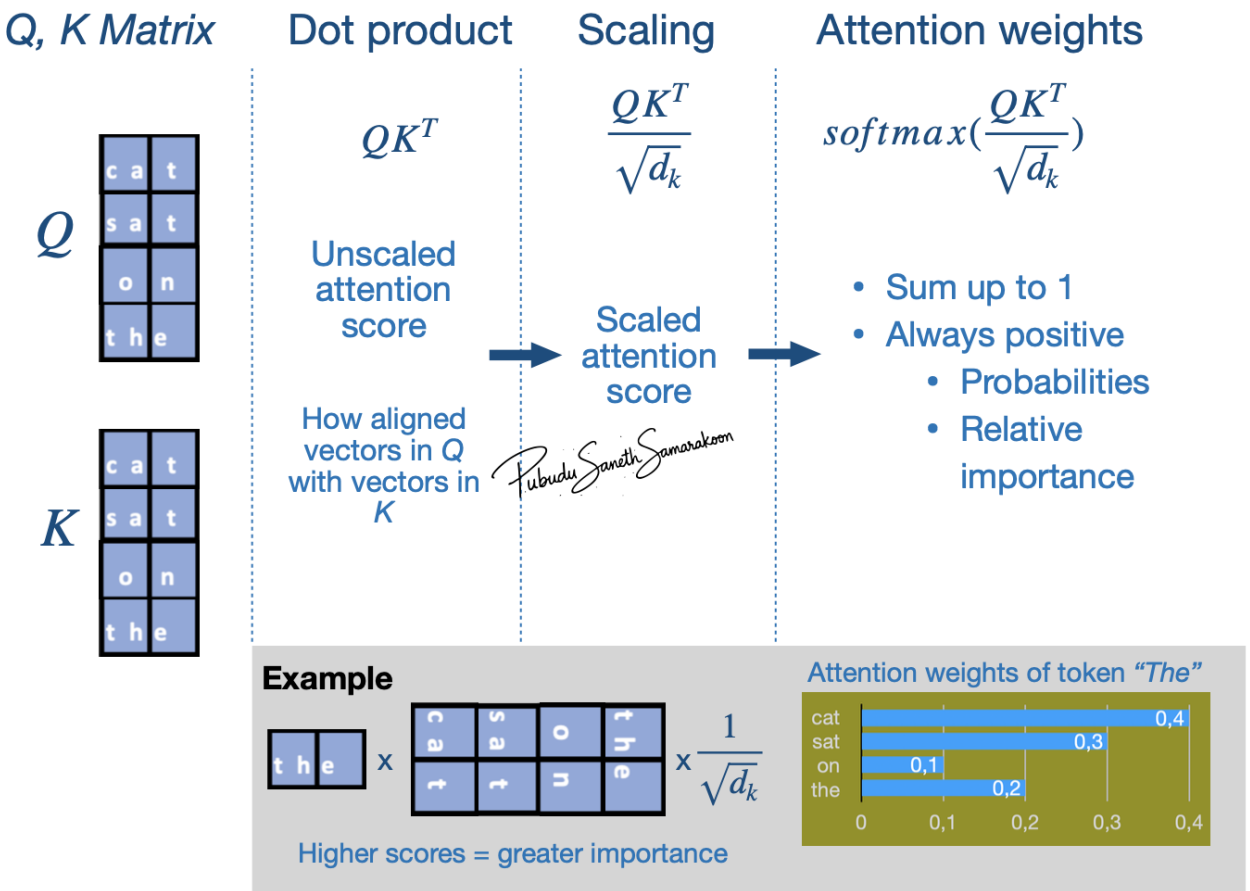
- Token representations that are used to combine attention weights and generate context vector

Calculate attention weights

- Attention weights: \(softmax(\frac{QK^T}{\sqrt{d_k}})\)

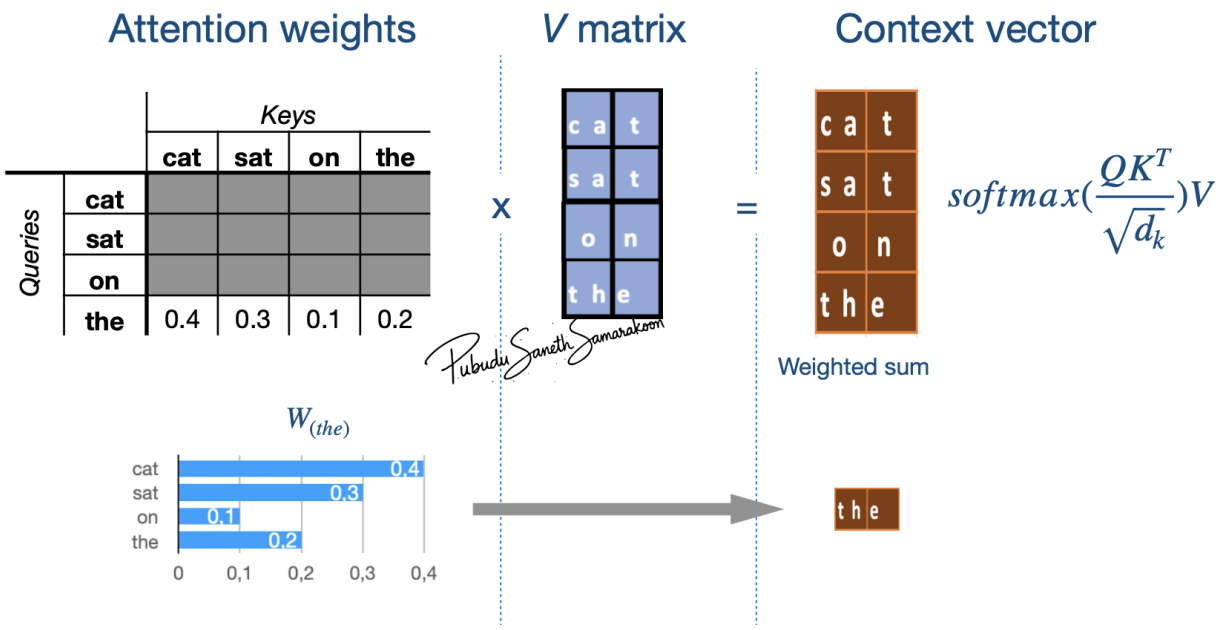
Main Stages

- **Stages 1**: Dot product to calculate attention score (matrix manipulation: \({QK^T}\)):
  - Provides unscaled attention score (initial relevance scores) - A higher dot product means the two tokens are more aligned (similar context)
  - Indicates how aligned vectors in \(Q_{matrix}\) with vectors in \(K_{matrix}\)
  - i.e., how much focus \(Q_{matrix}\) vectors should put on \(K_{matrix}\) vectors
  - Matrix manipulation enables simultaneously compare all the vectors in \(Q_{matrix}\) to \(K_{matrix}\)
- **Stage 2**: Scaling: Scaled attention score
  - Help avoid high-values in attention score and stabilize gradients
- **Stage 3**: Calculate "Attention weights"
  - Apply `softmax` function to scaled attention scores and calculate "Attention weights"
  - `softmax` function makes values to be positive and sums up 1 (convert to probabilities)
  - i.e., Convert attention scores to attention weights (probabilities) what shows "relative importance" \(Q_{matrix}\) vectors put on \(K_{matrix}\) vectors



Generate context vector

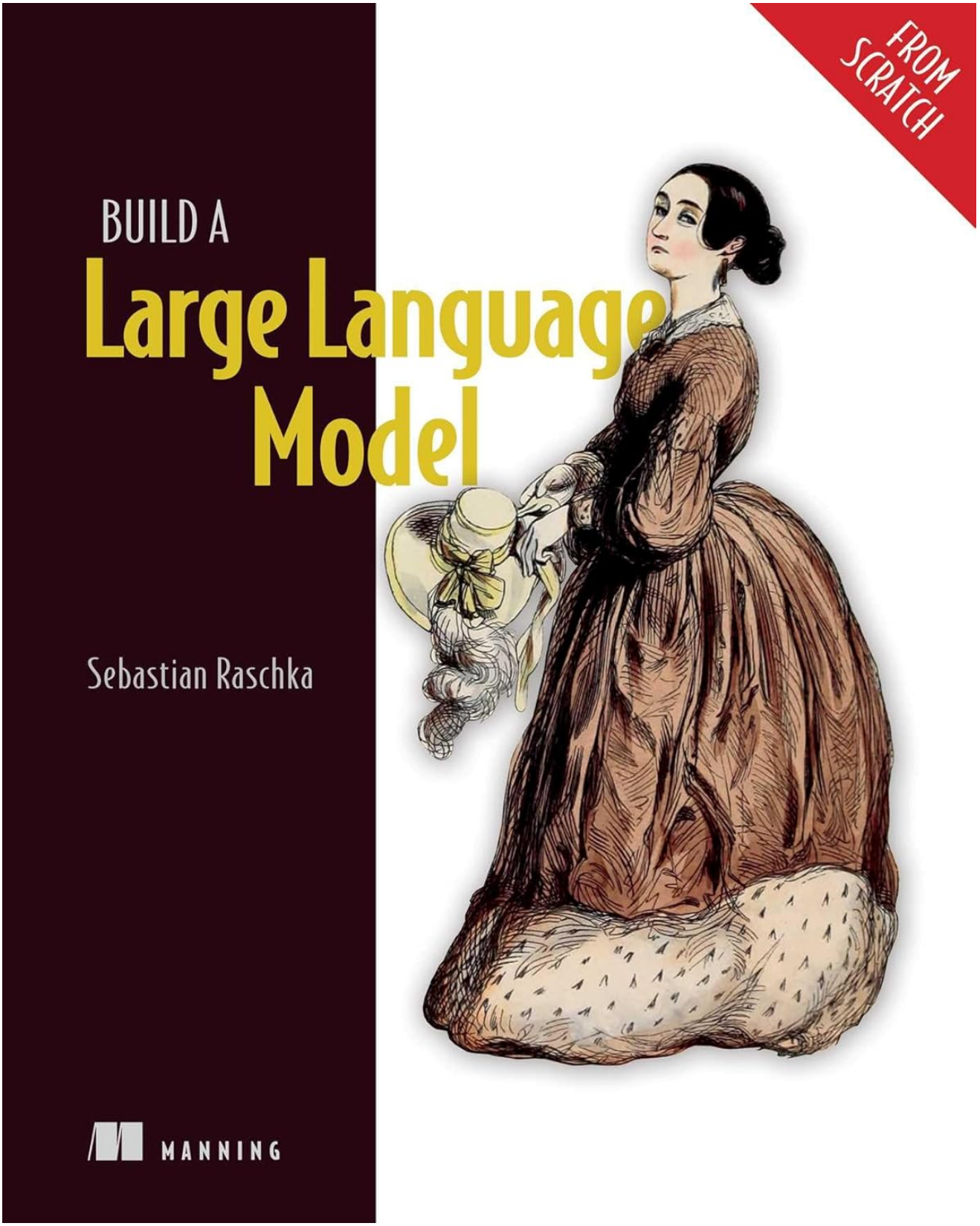
- Multiply these attention weights by the Value vectors (\(V_{matrix}\)) and produce final context vector



- \(W_{the}\): To what extent token “the” attend to (focus on) each input token (attention weights)
- \(V_{matrix}\): Representation of input token embedding matrix

Reference

Book “Build a Large Language Model (From Scratch)”, Sebastian Raschka



Who is the tutorial for?

This tutorial is for individuals with deep learning knowledge and want to have a basic overview of the architecture of LLMs. The tutorial is designed not to dive deep into each component but to interpret the underline processes of LLM's key components.

About the course

See also

Credits