Mathematics of Transformers: A Tutorial

Esmaeil Rezaei

January 8, 2026

Abstract

This tutorial provides a comprehensive mathematical treatment of Transformer architectures, the foundation of modern natural language processing models. We begin with fundamental concepts such as tokenization and embeddings, then systematically develop the mathematical framework underlying positional encodings, self-attention mechanisms, multi-head attention, and complete Transformer architectures. Through detailed examples and rigorous mathematical derivations, we illuminate how these components work together to enable models like BERT, GPT, and their variants to achieve state-of-the-art performance on language tasks. This document is designed for readers with basic knowledge of linear algebra and calculus who wish to understand the precise mathematical operations that make Transformers effective.

Contents

1 Introduction

The Transformer architecture, introduced by Vaswani et al. [**?**] in their seminal 2017 paper "Attention is All You Need," revolutionized natural language processing and has since been adapted to domains including computer vision, speech processing, and protein folding prediction. Unlike previous sequential models such as recurrent neural networks (RNNs) and long short-term memory networks (LSTMs), Transformers process entire sequences in parallel through a mechanism called **self-attention**. In simple terms, rather than processing words one by one in order, self-attention allows each word to directly consider its relationships with all other words in the sentence, capturing how its position and meaning relate to the words before and after it.

This tutorial provides a rigorous mathematical exposition of Transformer architectures. We develop the subject systematically, beginning with preliminaries and building toward complete understanding of both encoder and decoder architectures.

1.1 Prerequisites

This section assumes that the reader has a foundational background in mathematics and machine learning. While the core ideas of the Transformer architecture will be developed step by step, a basic familiarity with the following topics will help readers better understand both the intuition and the mathematical formulations presented throughout this tutorial.

Readers should be familiar with:

- Linear algebra: vectors, matrices, matrix multiplication, dot products
- Basic calculus: derivatives, chain rule, gradient descent
- Machine learning fundamentals: neural networks, activation functions, backpropagation
- Basic probability: probability distributions, expectation

1.2 Notation

Throughout this document, we use the following notation conventions:

- Scalars: lowercase letters (x, y, α)
- Vectors: lowercase bold letters ($\mathbf{v}$, $\mathbf{x}$)
- Matrices: uppercase bold letters ($\mathbf{W}$, $\mathbf{Q}$)
- Sets: uppercase calligraphic letters ($\mathcal{V}$, $\mathcal{D}$)
- Sequence length: n
- Embedding dimension: d_{model} or simply d
- Number of attention heads: h
- Dimension per attention head: $d_k = d_v = d_{\text{model}}/h$

2 From Text to Vectors: Tokenization and Embeddings

Before any mathematical operations can be performed, raw text must be converted into numerical representations that neural networks can process.

2.1 Tokenization

When working with text, a Transformer model cannot read an entire sentence at once the way humans do. Instead, the sentence must first be *broken into smaller parts*. This step is important because neural networks operate on numbers, and these numbers are assigned to pieces of text.

A simple way to break a sentence is to split it into full words. For example, the sentence *"Transformers are powerful models"* can be split into the words *"Transformers"*, *"are"*, *"powerful"*, and *"models"*. However, this approach has a limitation: the model would need to store every possible word it might ever see in its dictionary. Since natural language contains millions of words, variations, and misspellings, this dictionary can quickly become very large and impractical.

To solve this problem, modern models break sentences into *tokens* instead of full words. Tokens are smaller pieces of text. A token can be a whole word, part of a word, or even a single character. By using tokens, the model can work with a much smaller dictionary while still being able to represent a wide variety of words. For example, the word *"learning"* can be represented as *"learn"* and *"ing"*, allowing the model to reuse these pieces across many different words.

Definition 2.1 (Tokenization)**.** *Tokenization is the process of splitting text into discrete units called* ***tokens****. Formally, a tokenizer is a function* $\tau : \mathcal{S} \to \mathcal{V}^*$ *that maps an input string* $s \in \mathcal{S}$ *to a sequence of tokens drawn from a vocabulary* $\mathcal{V}$*.*

Several tokenization strategies are commonly used:

1. **Word-level tokenization**: Each word is treated as a single token, which often leads to a very large vocabulary.
2. **Subword tokenization**: Words are broken into smaller, reusable parts (for example, Byte-Pair Encoding or WordPiece), reducing vocabulary size while maintaining flexibility.
3. **Character-level tokenization**: Each character becomes a token, resulting in a very small vocabulary but much longer token sequences.

Most modern Transformer models use **subword tokenization** because it offers a practical balance between vocabulary size and expressive power.

2.2 Token Embeddings

After splitting text into tokens, the next step is to convert these tokens into numbers that the model can understand. Simply assigning each token a unique number is not enough, because the model would have no sense of similarity between tokens. For example, the words *"cat"* and *"dog"* would be just two unrelated numbers, even though they are semantically similar. To address this, Transformers use **embeddings**.

Intuitively, an embedding is a way to represent each token as a list of numbers—a vector—so that the model can capture the meaning and relationships between tokens. Tokens that are related in meaning or usage will have vectors that are closer together in this numeric space.

Definition 2.2 (Embedding)**.** *An embedding is a learned mapping*

$$E : \mathcal{V} \to \mathbb{R}^{d_{model}}$$

that assigns each token in the vocabulary $\mathcal{V}$ *to a dense vector in* $\mathbb{R}^{d_{model}}$*.*

Formally, the embedding layer is represented by an **embedding matrix**

$$\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}},$$

where $|\mathcal{V}|$ is the vocabulary size and d_{model} is the size of each embedding vector. For a token with index $i \in \{1, 2, \ldots, |\mathcal{V}|\}$, its embedding is simply the i-th row of the matrix:

$$\mathbf{e}_i = \mathbf{E}[i, :] \in \mathbb{R}^{d_{\text{model}}}.$$

For a sequence of n tokens with indices $(t_1, t_2, \ldots, t_n)$, the embedding operation produces a matrix:

$$\mathbf{X}_{\text{tok}} = \begin{bmatrix} \mathbf{e}_{t_1}^T \\ \mathbf{e}_{t_2}^T \\ \vdots \\ \mathbf{e}_{t_n}^T \end{bmatrix} \in \mathbb{R}^{n \times d_{\text{model}}}.$$

Traditionally, these embeddings are learned during training via backpropagation, allowing semantically similar tokens—like *"cat"* and *"dog"*—to have vectors close to each other. However, you do not always need to train a large model from scratch to obtain useful embeddings. Pre-trained Transformer models like **BERT** and **DistilBERT** provide embeddings that can be directly used. You can feed a token or a sentence into these models and extract the output vectors, which can be treated as the token's *fingerprint*. These embeddings already capture rich semantic information learned from massive text corpora, saving you both time and computational resources.

For instance:

- **BERT-base**: A full Transformer model with 12 layers and 110 million parameters.
- **DistilBERT**: A smaller, faster version with 6 layers and 66 million parameters, retaining about 97% of BERT's performance.

3 Positional Encodings in Transformer Architectures

Both BERT and DistilBERT share the same overall structure: an embedding layer followed by multiple transformer encoder layers. The key architectural difference is depth—BERT uses more transformer layers than DistilBERT—but the internal mechanics of each layer are the same.

Each transformer layer contains multi-head self-attention, normalization steps, and feed-forward networks. However, there is a critical component that sits *between* the embedding layer and the transformer layers: **positional encodings**.

3.1 Why Positional Encodings Are Needed

Transformers process input tokens **in parallel**, not sequentially. Unlike recurrent models, they do not read a sentence from left to right or right to left. This parallelism is what makes transformers efficient, but it introduces a problem: **word order is not inherently known to the model**.

Without additional information, a transformer cannot distinguish between:

``The cat sat on the mat''

and

``On the mat sat the cat''

Both sentences contain the same tokens, but their meanings differ because of word order.

Positional encodings solve this problem by injecting information about **token position** into the model.

3.2 How Positional Encodings Work

So far, we have split a sentence into tokens and converted each token into an embedding vector. While these embeddings capture the meaning of each token, they do not contain any information about the order of the tokens in the sentence. For example, the sentences:

``The cat chased the dog'' and ``The dog chased the cat''

would have the same set of embeddings, even though their meanings are different. To address this, Transformers use **positional encodings**. They are special vectors that represent the position of each token in the sequence. These vectors are **added directly to the token embeddings** before the data enters the Transformer layers. This way, the model knows not only the meaning of each token but also where it appears in the sentence.

Mathematically, this can be written as:

$$\mathbf{X} = \mathbf{X}_{\text{tok}} + \mathbf{P},$$

where $\mathbf{X}_{\text{tok}} \in \mathbb{R}^{n \times d_{\text{model}}}$ is the matrix of token embeddings for a sequence of length n, and $\mathbf{P} \in \mathbb{R}^{n \times d_{\text{model}}}$ is the matrix of positional encodings for the same sequence.

Each position in the sentence has a unique positional vector, which can encode either absolute positions (like the first, second, third token) or relative positions (how far one token is from another). By combining token embeddings with positional encodings, the Transformer can learn patterns such as word order, distances between words, and sentence structure.

Example: Suppose we have a very short sentence of 3 tokens: *"I love AI"*. We use an embedding size $d_{\text{model}} = 4$ (so each token is represented as a 4-dimensional vector; in reality, depending on the model, this size can be several thousand).

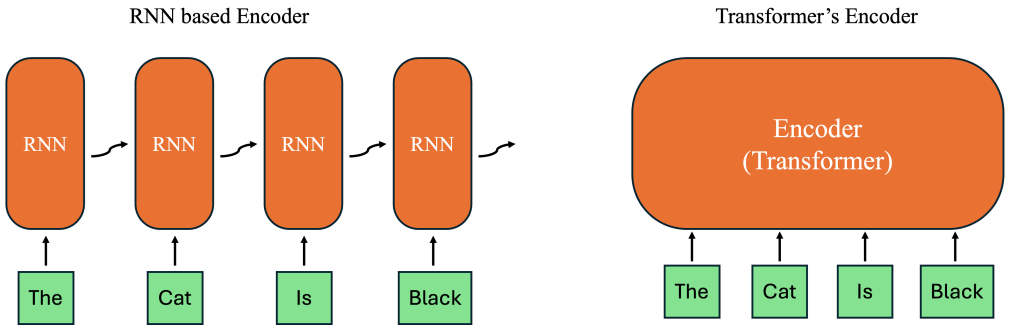

Figure 1: RNN vs. Transformer encoding.

Token embeddings: (rows represent *I*, *love*, *AI*, respectively)

$$\mathbf{X}_{\text{tok}} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \end{bmatrix}$$

Positional encodings: These vectors encode the position of each token. In practice, they are computed by the model, but for simplicity, we assume we already have them:

$$\mathbf{P} = \begin{bmatrix} 0.01 & 0.02 & 0.03 & 0.04 \\ 0.05 & 0.06 & 0.07 & 0.08 \\ 0.09 & 0.10 & 0.11 & 0.12 \end{bmatrix}$$

Adding them together:

$$\mathbf{X} = \mathbf{X}_{\text{tok}} + \mathbf{P} = \begin{bmatrix} 0.11 & 0.22 & 0.33 & 0.44 \\ 0.55 & 0.66 & 0.77 & 0.88 \\ 0.99 & 1.10 & 1.21 & 1.32 \end{bmatrix}$$

Now, $\mathbf{X}$ contains both **semantic meaning** from the token embeddings and **positional information** from the positional encodings. This matrix is what gets passed into the Transformer layers.

3.3 Comparison with Sequential Models

In traditional RNN-based encoders (Figure 1), tokens are processed one at a time.

Input sequence: ‘‘`the cat is black`’’

Processing order:

- “the” → first
- “cat” → second
- “is” → third
- “black” → fourth

Because the sequence is processed step by step, the model naturally learns word order.

In contrast, a transformer encoder receives all tokens at once:

```
["the", "cat", "is", "black"]
```

All tokens are processed simultaneously. Without positional encodings, the model has no way to know which word came first or last.

3.4 Sinusoidal Positional Encodings

So far, we have seen that positional encodings (PE) are added to token embeddings to give the model information about the position of each token in the sequence. The original Transformer paper introduced a specific way to compute these positional encodings using **sine and cosine functions**, which has several useful properties.

Definition 3.1 (Sinusoidal Positional Encoding)**.** *For a token at position* $pos \in \{0, 1, \ldots, n-1\}$ *and embedding dimension* $i \in \{0, 1, \ldots, d_{model} - 1\}$*, the positional encoding vector is defined as:*

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right).$$

Here, PE is the **positional encoding matrix** that we add to the token embeddings. Each row corresponds to a token position in the sentence, and each column corresponds to a dimension of the embedding.

- pos is the **position index** of the token in the sentence (starting from 0).
- i is the **dimension index** of the embedding (starting from 0).
- Even dimensions (0, 2, 4, ...) use sine.
- Odd dimensions (1, 3, 5, ...) use cosine.

Why sine and cosine?

- **Unique encoding for each position:** Each position gets a unique combination of sine and cosine values across the embedding dimensions, so the model can distinguish one position from another.
- **Smooth change across positions:** Sine and cosine vary smoothly, so small changes in position lead to small changes in the positional vector. This helps the model capture relative distances between tokens.
- **Easily learn relative positions:** Using sine and cosine allows the model to compute the relative position between tokens using simple linear operations, because of trigonometric identities (e.g., $\sin(a+b)$ and $\cos(a+b)$).

The formula also uses a **geometric progression of wavelengths** from 2π to $10000 \cdot 2\pi$, so that some dimensions change quickly with position (capturing short-range relationships) and others change slowly (capturing long-range relationships). Indeed, sinusoidal positional encodings give the Transformer a way to **know the order of tokens** without relying on recurrence or convolution, while keeping the representations continuous and differentiable.

3.5 Properties of Sinusoidal Encodings

Theorem 3.1 (Linear Offset Property)**.** *For any fixed offset* k*, the positional encoding at position* $pos + k$ *can be represented as a linear function of the encoding at position pos.*

This property allows the model to learn relative positions easily. The proof relies on trigonometric identities:

$$\sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta,$$
$$\cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta.$$

3.6 Practical Computation Example

Here is a example of how sinusoidal positional encodings are computed.

- **Sentence:** "The cat is black"

- **Embedding dimension:** $d_{\text{model}} = 6$

For position pos = 1 (the second word "cat"):

$$
\begin{aligned}
PE_{(1,0)} &= \sin\left(\frac{1}{10000^{0/6}}\right) = \sin(1) \approx 0.8415, \\
PE_{(1,1)} &= \cos\left(\frac{1}{10000^{0/6}}\right) = \cos(1) \approx 0.5403, \\
PE_{(1,2)} &= \sin\left(\frac{1}{10000^{2/6}}\right) = \sin(1/21.5443) \approx \sin(0.0464) \approx 0.0464, \\
PE_{(1,3)} &= \cos\left(\frac{1}{10000^{2/6}}\right) \approx \cos(0.0464) \approx 0.9989, \\
PE_{(1,4)} &= \sin\left(\frac{1}{10000^{4/6}}\right) = \sin(1/464.159) \approx 0.0022, \\
PE_{(1,5)} &= \cos\left(\frac{1}{10000^{4/6}}\right) \approx \cos(0.0022) \approx 1.0000.
\end{aligned}
$$

Thus, the positional encoding vector for the word "cat" is approximately:

$$\mathbf{p}_1 \approx [0.8415, 0.5403, 0.0464, 0.9989, 0.0022, 1.0000]^T$$

Note that for each dimension index i (e.g., $i = 0$), the positional encoding formula produces both a sine and a cosine value. Therefore, it is better not to think of even and odd i as directly corresponding to sine and cosine, respectively.

3.7 Learned Positional Embeddings

An alternative approach, used in BERT and GPT, employs **learned positional embeddings**. Instead of fixed sinusoidal functions, a learnable embedding matrix $\mathbf{P} \in \mathbb{R}^{n_{\max} \times d_{\text{model}}}$ is initialized randomly and optimized during training, where $n_{\max}$ is the maximum sequence length. Both approaches have trade-offs:

- **Sinusoidal**: Deterministic, generalizes to longer sequences, mathematically interpretable
- **Learned**: More flexible, potentially better performance on specific tasks, limited to training sequence lengths

4 Self-Attention Mechanism

Self-attention is the core innovation of the Transformer architecture. It allows each token to attend to all other tokens in the sequence, dynamically determining which tokens are most relevant for understanding the current token.

4.1 The Core Problem: Understanding Context

To understand why we need attention mechanisms, let's start with a concrete example. Consider the sentence: *"The artist painted the portrait in the studio."*

When we, as humans, read this sentence, we automatically understand the relationships between words. We know that "artist" is the one performing the action of "painted," and that "portrait" is what's being painted. However, for a machine learning model, establishing these relationships is far from trivial.

The word "painted" has different degrees of relationship with different words in the sentence. It has a strong semantic connection to "artist" because the artist is the one who performs the action of painting. We call this a subject-verb relationship in linguistics. Similarly, "painted" has a strong connection to "portrait" because the portrait is the direct object receiving the action—it's what is being painted.

The connection between "painted" and "studio" is weaker—while the studio provides contextual information about where the painting took place, it's not as central to understanding the core action. The word

"the" appears twice in the sentence, but these articles have minimal semantic connection to "painted" since they primarily serve grammatical functions rather than conveying substantive meaning.

The attention mechanism allows the model to learn and represent these varying degrees of relevance mathematically. Instead of treating all words equally, the model learns to assign different importance weights to different words based on their relevance to the word being analyzed. This is the core insight that makes transformers so powerful.

4.2 The Three Learned Transformations

At the heart of the attention mechanism are three learned linear transformations that convert our input embeddings into three different representations. These are called the Query, Key, and Value transformations, and they are represented mathematically as:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{1}$$

Let's break down what each symbol means. The matrix $X \in \mathbb{R}^{n \times d_{model}}$ represents our input embedding matrix, where n is the number of tokens in our sequence and d_{model} is the dimension of each embedding vector (typically 768 for BERT). The matrices $W^Q, W^K, W^V \in \mathbb{R}^{d_{model} \times d_k}$ are the learned weight matrices—these contain parameters that the model learns during training through backpropagation. Finally, Q, K, V are the resulting query, key, and value matrices that we'll use to compute attention.

4.3 Understanding Each Component Through Analogy

The names "query," "key," and "value" come from information retrieval systems, and this analogy is quite helpful for understanding their roles.

Query Matrix (W^Q): Think of the query as a question being asked by each word. When we create a query vector for a word like "painted," we're essentially asking "What information do I need from the other words in this sentence to fully understand my role and meaning?" The query transformation learns to encode what each word is looking for from other words in the context. During training, the model learns weight matrices that transform token embeddings into query vectors that effectively ask these questions.

Key Matrix (W^K): The key vectors can be thought of as labels or identifiers that describe what information each word contains. When we create a key vector for "artist," we're creating a representation that signals "I contain information about the agent performing an action." The key transformation learns to create representations that can be matched against queries. If a query is asking for information about who performed an action, and a key represents an agent, these should have high similarity.

Value Matrix (W^V): Once we've determined which words are relevant (through query-key matching), we need to actually retrieve the information from those words. The value vectors contain the actual semantic content that will be aggregated. While queries and keys are used to determine relevance, values carry the information that gets passed forward in the network. The value transformation learns to extract and encode the most useful information from each token for downstream processing.

4.4 Why Three Separate Transformations?

You might wonder why we need three different transformations rather than just one or two. This design choice is actually quite elegant and serves several important purposes.

First, this separation allows the model to decouple the "matching" process from the "information retrieval" process. The query-key interaction determines which words should attend to which other words—this is purely about relevance and similarity. The value vectors, on the other hand, carry the actual information that gets aggregated. This separation means the model can learn to identify relevant relationships (via Q and K) independently from learning what information to extract and pass forward (via V).

Second, having separate transformations allows each component to learn different aspects of token relationships. The query and key matrices can specialize in learning similarity patterns—what makes two words semantically related—while the value matrix can specialize in extracting useful features for the specific task at hand.

Third, this architecture provides the flexibility to capture both syntactic and semantic dependencies simultaneously. Different attention heads (which we'll discuss later) can use their Q, K, V transformations to focus on different types of relationships—some might learn grammatical dependencies while others learn thematic relationships.

It's important to note that these three matrices contain learned parameters. During training, the model adjusts these parameters through backpropagation to minimize the loss on the training task. This means that for a well-trained model, the W^Q matrix has learned to transform embeddings into effective queries, W^K into effective keys, and W^V into effective values—all optimized for the specific task the model is being trained on.

5 Detailed Mathematical Walkthrough

5.1 Setup

Now that we understand the conceptual foundation, let's work through a concrete example with actual numbers. This will help solidify your understanding of how attention works in practice. We'll use simplified dimensions to keep the calculations manageable and comprehensible, but the same principles apply to full-scale models.

Input Sentence: [CLS] The artist painted the portrait [SEP]

This sentence has been tokenized into 7 tokens total. The [CLS] token is a special classification token that BERT adds at the beginning of every sequence—it's used to aggregate sequence-level information for classification tasks. The [SEP] token is a separator token added at the end to mark the boundary of the sequence.

Embedding Dimension: For pedagogical purposes, we use $d_{model} = 4$. In actual BERT models, this dimension is 768, which provides much richer representations but makes hand calculations impractical. The principles remain exactly the same regardless of dimension—we're just using 4 to make the mathematics transparent and followable.

5.2 Token Embeddings

After tokenization, each token is converted into an embedding vector through an embedding lookup table. This table was learned during pretraining on massive text corpora. For our example, suppose we obtain the following embedding matrix:

$$X = \begin{bmatrix} 1.0 & 0.2 & 0.3 & 0.1 \\ 0.3 & 0.9 & 0.2 & 0.4 \\ 0.2 & 0.1 & 0.8 & 0.3 \\ 0.4 & 0.3 & 0.2 & 0.9 \\ 0.1 & 0.7 & 0.4 & 0.2 \\ 0.5 & 0.2 & 0.6 & 0.3 \\ 0.8 & 0.1 & 0.2 & 0.5 \end{bmatrix} \tag{2}$$

Each row in this matrix represents one token. The first row $[1.0, 0.2, 0.3, 0.1]$ is the embedding for [CLS], the second row $[0.3, 0.9, 0.2, 0.4]$ is the embedding for "The", the third row is "artist", and so on. These embeddings capture semantic information about each word—similar words will have similar embedding vectors.

It's worth noting that these embeddings are not random. They were learned during pretraining so that words with similar meanings or that appear in similar contexts have vectors that are close together in this high-dimensional space. For example, the embeddings for "artist" and "painter" would be very close to each other, while "artist" and "computer" would be far apart.

Figure 2 visualizes these vectors in a 2D space after reducing the embedding matrix to two dimensions using Principal Component Analysis (PCA). Each point corresponds to a token, and the coordinates represent the values along the first two principal components, which capture the majority of the variance in the embeddings.

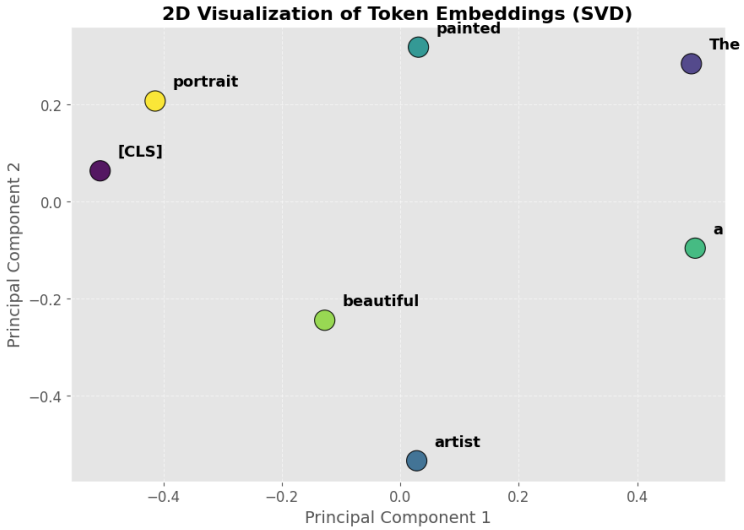

Figure 2: 2D visualization of token embeddings.

From the visualization, we can observe semantic relationships between words. For instance, the words "artist" and "beautiful" are positioned close to each other, indicating that their embeddings capture related semantic information. Similarly, the words "painted" and "portrait" are close in this 2D space, reflecting their contextual similarity in the sentence. Additionally, the tokens "a" and "The" are near each other, showing that the model recognizes their similar grammatical roles. This demonstrates how the embedding space encodes both semantic and syntactic patterns learned during pretraining.

The embedding values used in this example are **purely illustrative and made-up**. In practice, actual token embeddings from models like BERT or GPT will be high-dimensional and learned, but the resulting 2D visualization using SVD or PCA would show a similar structure and relationships among tokens.

5.3 Learned Weight Matrices

The next component we need are the weight matrices that will transform our embeddings into queries, keys, and values. In a real training scenario, these matrices would be randomly initialized and then gradually optimized through backpropagation as the model learns from data. For our example, let's assume the following learned parameters (which we'll pretend have already been optimized through training):

$$W^Q = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \\ 1.3 & 1.4 & 1.5 & 1.6 \end{bmatrix} \tag{3}$$

$$W^K = \begin{bmatrix} 0.2 & 0.3 & 0.4 & 0.5 \\ 0.6 & 0.7 & 0.8 & 0.9 \\ 1.0 & 1.1 & 1.2 & 1.3 \\ 1.4 & 1.5 & 1.6 & 1.7 \end{bmatrix} \tag{4}$$

$$W^V = \begin{bmatrix} 0.3 & 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 & 1.0 \\ 1.1 & 1.2 & 1.3 & 1.4 \\ 1.5 & 1.6 & 1.7 & 1.8 \end{bmatrix} \tag{5}$$

These matrices are 4×4 because our embeddings are 4-dimensional and we're keeping the query/key/value dimensions the same for simplicity. In BERT, you have 768×64 matrices—the model dimension is 768, but each attention head uses only 64 dimensions (and then 12 heads are concatenated to recover the full 768

dimensions, but we'll discuss multi-head attention later).

An important point to understand is that these weight matrices are shared across all positions in the sequence. The same W^Q matrix is used to transform every token's embedding into its query vector. This parameter sharing is crucial—it means the model learns general transformation rules rather than position-specific ones, allowing it to generalize to sequences of any length.

5.4 Computing Query Vectors

The following numerical example illustrates the attention computations for a *single token*, namely the word *"painted"*. All calculations shown below are performed only for this token to demonstrate the mechanics of query generation; in practice, the same operations are applied to every token in the sequence.

$$q_{\text{painted}} = x_{\text{painted}} \cdot W^Q \tag{6}$$
$$= [0.4, 0.3, 0.2, 0.9] \cdot W^Q \tag{7}$$
$$= [1.41, 1.53, 1.65, 1.77] \tag{8}$$

We compute query vectors for all tokens:

$$Q = X \cdot W^Q = \begin{bmatrix} q_{[CLS]} \\ q_{\text{The}} \\ q_{\text{artist}} \\ q_{\text{painted}} \\ q_{\text{the}} \\ q_{\text{portrait}} \\ q_{[SEP]} \end{bmatrix} \tag{9}$$

5.5 Computing Key Vectors

Similarly, we compute key vectors for all tokens. For "artist":

$$k_{\text{artist}} = x_{\text{artist}} \cdot W^K \tag{10}$$
$$= [0.2, 0.1, 0.8, 0.3] \cdot W^K \tag{11}$$
$$= [1.26, 1.38, 1.50, 1.62] \tag{12}$$

We compute key vectors for all tokens:

$$K = X \cdot W^K = \begin{bmatrix} k_{[CLS]} \\ k_{\text{The}} \\ k_{\text{artist}} \\ k_{\text{painted}} \\ k_{\text{the}} \\ k_{\text{portrait}} \\ k_{[SEP]} \end{bmatrix} \tag{13}$$

5.6 Computing Value Vectors

As before, value vectors are obtained by applying a linear transformation to the input embeddings. For the token *"artist"*, the value vector is computed as:

$$v_{\text{artist}} = x_{\text{artist}} \cdot W^V \tag{14}$$
$$= [0.2, 0.1, 0.8, 0.3] \cdot W^V \tag{15}$$
$$= [1.10, 1.22, 1.34, 1.46] \tag{16}$$

Collectively, the value matrix is given by:

$$V = X \cdot W^V = \begin{bmatrix} v_{[CLS]} \\ v_{\text{The}} \\ v_{\text{artist}} \\ v_{\text{painted}} \\ v_{\text{the}} \\ v_{\text{portrait}} \\ v_{[SEP]} \end{bmatrix} \tag{17}$$

At this stage, the query, key, and value matrices Q, K, and V have been constructed. The subsequent attention computation uses the interaction between Q and K to produce attention weights, which are then applied to the value matrix V to obtain context-aware token representations.

6 Why Dot Products Measure Similarity

Before we proceed with softmax normalization, it's crucial to understand why dot products are used to measure similarity between vectors. This is not just a mathematical convenience—there's deep geometric intuition behind this choice.

6.1 Geometric Interpretation

The dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$ can be expressed in two equivalent ways. The algebraic form is simply the sum of element-wise products: $\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$. But there's also a geometric form:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos(\theta) \tag{18}$$

where $|\mathbf{a}|$ and $|\mathbf{b}|$ are the lengths (magnitudes) of the vectors, and θ is the angle between them. This geometric interpretation reveals why dot products measure similarity. The cosine of the angle between vectors captures their directional alignment. When two vectors point in exactly the same direction, $\theta = 0$ and $\cos(0) = 1$, giving us the maximum positive dot product (scaled by the magnitudes). When vectors are perpendicular—pointing in completely unrelated directions—we have $\theta = 90$ and $\cos(90) = 0$, yielding a dot product of zero. When vectors point in opposite directions, $\theta = 180$ and $\cos(180) = -1$, resulting in a negative dot product.

In the context of attention, when a query vector and a key vector have a high dot product, it means they're pointing in similar directions in the embedding space. Since these vectors were created from word embeddings that place semantically similar words close together, a high dot product between a query and key indicates semantic relevance.

6.2 Why This Matters for Attention

Let's think about what this means for our example sentence. When we compute the query vector for "painted" and take its dot product with the key vector for "artist," we're measuring how aligned these representations are in the transformed embedding space. The model has learned (through training) to transform embeddings such that queries for verbs align well with keys for their subjects.

Similarly, when we compute the dot product between the query for "painted" and the key for "portrait," we get a high score because the model has learned that verbs should attend to their objects. The dot

product between "painted" and "the" is low because the model has learned that content words (verbs, nouns) generally don't need to strongly attend to function words (articles, prepositions).

7 Scaled Dot-Product Attention

7.1 The Scaling Factor

Now we arrive at an important technical detail that might seem like a minor adjustment but is actually crucial for training stability. When we compute dot products between query and key vectors, especially in high-dimensional spaces, these products can grow quite large in magnitude. This creates problems during training.

To understand why, consider what happens when we feed large values into the softmax function. The softmax function uses exponentials: e^x. When x is large, e^x becomes enormous. For instance, $e^{10} \approx 22026$, but $e^{20} \approx 485,165,195$. If one attention score is 20 and another is 10, after softmax the first will receive nearly all the probability mass—the distribution becomes extremely peaked.

This extreme peaking causes two problems. First, during forward propagation, almost all attention weight goes to a single token, which limits the model's ability to consider multiple relevant words. Second, and more critically, during backpropagation, the gradients of the softmax in these regions become extremely small (the function is nearly flat), leading to vanishingly small gradient updates. This is the vanishing gradient problem, and it can essentially halt learning. The solution is remarkably simple: we scale the dot products by dividing by $\sqrt{d_k}$, where d_k is the dimension of the key vectors. The complete scaled dot-product attention formula is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{19}$$

7.2 Why Scale by $\sqrt{d_k}$?

The choice of $\sqrt{d_k}$ as the scaling factor isn't arbitrary—it comes from statistical considerations. Let's think about what happens when we compute a dot product between two vectors whose elements are random variables with mean 0 and variance 1 (which is approximately what we get after proper initialization and normalization).

For two such random vectors q and k of dimension d_k, the dot product is $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$. Each product $q_i k_i$ has expected value $\mathbb{E}[q_i k_i] = \mathbb{E}[q_i]\mathbb{E}[k_i] = 0 \cdot 0 = 0$ (assuming independence). The variance of each product is $\text{Var}(q_i k_i) = \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = 1 \cdot 1 = 1$. Now, because the variance of a sum of independent random variables is the sum of their variances, we have: $\text{Var}(q \cdot k) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = d_k$. So the variance of the dot product grows linearly with the dimension. This means that in a 768-dimensional space (like BERT), dot products will have variance 768—they'll be spread out over a much wider range than in, say, a 4-dimensional space where the variance is only 4. By dividing by $\sqrt{d_k}$, we're dividing by the standard deviation, which normalizes the variance back to 1. This keeps the dot products in a reasonable range regardless of dimensionality, preventing the saturation of the softmax function.

7.3 Computing Attention Scores

The attention score between "painted" and each other token is computed via dot product:

$$\text{score}(q_{\text{painted}}, k_i) = q_{\text{painted}} \cdot k_i^T \tag{20}$$

For "painted" and "artist":

$$\begin{aligned}
\text{score}_{\text{artist}} &= [1.41, 1.53, 1.65, 1.77] \cdot [1.26, 1.38, 1.50, 1.62]^T && (21)\\
&= 1.41(1.26) + 1.53(1.38) + 1.65(1.50) + 1.77(1.62) && (22)\\
&= 1.777 + 2.111 + 2.475 + 2.867 && (23)\\
&= 9.230 && (24)
\end{aligned}$$

Computing scores for all tokens relative to "painted":

$$\text{Scores} = \begin{bmatrix} \text{score}([CLS]) = 5.432 \\ \text{score}(\text{The}) = 6.123 \\ \text{score}(\text{artist}) = 9.230 \\ \text{score}(\text{painted}) = 10.156 \\ \text{score}(\text{the}) = 6.891 \\ \text{score}(\text{portrait}) = 8.745 \\ \text{score}([SEP]) = 5.678 \end{bmatrix} \tag{25}$$

The highest score is with "painted" itself at 10.156. This self-attention is actually very important—the model needs to retain information about the word it's currently processing. The second highest score is with "artist" at 9.230, reflecting the strong subject-verb relationship. The third highest is with "portrait" at 8.745, capturing the verb-object relationship. These high scores tell us that when the model processes "painted," it should pay strong attention to who did the painting and what was painted.

Medium scores like "the" at 6.891 indicate moderate relevance—these grammatical words provide some context but aren't as semantically central. Low scores like [CLS] at 5.432 show minimal semantic connection—the special token doesn't convey meaning relevant to understanding "painted."

This pattern emerges naturally from the training process. The model wasn't explicitly told that verbs should attend to their subjects and objects. Instead, through millions of training examples, it learned that this attention pattern helps it predict masked words correctly, understand sentence structure, and solve other language understanding tasks.

7.4 Applying Scaling to Our Example

In our example, we're working with $d_k = 4$, so we divide all our attention scores by $\sqrt{4} = 2$. Taking our previously computed scores:

$$\text{Scaled Scores} = \begin{bmatrix} 5.432/2 = 2.716 \\ 6.123/2 = 3.062 \\ 9.230/2 = 4.615 \\ 10.156/2 = 5.078 \\ 6.891/2 = 3.446 \\ 8.745/2 = 4.373 \\ 5.678/2 = 2.839 \end{bmatrix} \tag{26}$$

Notice that while the relative ordering stays the same—"painted" still has the highest attention to itself, then to "artist," then to "portrait"—the absolute values are now in a more manageable range. These scaled scores will produce a smoother probability distribution after softmax, with better gradient properties for training.

8 Softmax Normalization

8.1 Converting Scores to Probabilities

At this point, we have computed scaled attention scores that tell us, roughly speaking, how relevant each token is to our query token. However, these scores are still just arbitrary real numbers. To make them more interpretable and useful, we need to convert them into probabilities—values between 0 and 1 that sum to 1 across all tokens.

This is where the softmax function comes in. The softmax function is a generalization of the logistic function to multiple dimensions. It takes a vector of arbitrary real numbers and squashes them into a probability distribution. The formula is:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{27}$$

The key properties of softmax are: (1) all outputs are positive (since $e^x > 0$ for all x), (2) all outputs are between 0 and 1, and (3) all outputs sum to exactly 1. This makes the output a valid probability distribution.

The exponential function e^x has an important property: it preserves order while amplifying differences. If $z_1 > z_2$, then $e^{z_1} > e^{z_2}$, but the ratio e^{z_1}/e^{z_2} grows exponentially with the difference $z_1 - z_2$. This means softmax naturally emphasizes the largest values—a score that's only slightly larger than another will receive noticeably more probability mass.

8.2 Step-by-Step Calculation

Let's walk through the softmax calculation for our scaled scores in complete detail. This will help you understand exactly what's happening at each step.

First, we compute the exponential of each scaled score:

$$e^{2.716} = 15.130 \tag{28}$$

$$e^{3.062} = 21.384 \tag{29}$$

$$e^{4.615} = 101.019 \tag{30}$$

$$e^{5.078} = 160.588 \tag{31}$$

$$e^{3.446} = 31.346 \tag{32}$$

$$e^{4.373} = 79.315 \tag{33}$$

$$e^{2.839} = 17.087 \tag{34}$$

Notice how the exponential function dramatically increases the spread. The largest score (5.078) was less than twice the smallest (2.716), but after exponentiation, 160.588 is more than ten times larger than 15.130. Next, we sum all these exponentials to get the normalization constant:

$$\sum_{j=1}^{7} e^{z_j} = 15.130 + 21.384 + 101.019 + 160.588 + 31.346 + 79.315 + 17.087 = 425.869$$

Finally, we divide each exponential by this sum to get the normalized attention weights:

$$\alpha = \begin{bmatrix} 15.130/425.869 = 0.0355 \\ 21.384/425.869 = 0.0502 \\ 101.019/425.869 = 0.2372 \\ 160.588/425.869 = 0.3770 \\ 31.346/425.869 = 0.0736 \\ 79.315/425.869 = 0.1862 \\ 17.087/425.869 = 0.0401 \end{bmatrix} \tag{35}$$

Let's verify our calculation is correct by checking that these probabilities sum to 1: $0.0355 + 0.0502 + 0.2372 + 0.3770 + 0.0736 + 0.1862 + 0.0401 = 0.9998 \approx 1.0$. The tiny discrepancy is due to rounding errors in our calculations—with infinite precision, the sum would be exactly 1.

8.3 Interpreting the Attention Weights

Now we can interpret these probabilities as attention weights. When processing "painted," the model should allocate its attention as follows: 37.7% to "painted" itself (maintaining self-information), 23.7% to "artist" (the subject), 18.6% to "portrait" (the object), 7.4% to "the" (grammatical context), and smaller amounts to the other tokens.

These percentages tell us how much each token will contribute to the final representation of "painted." The high weight on "artist" means that understanding "artist" is crucial for understanding "painted" in this context. The high weight on "portrait" similarly indicates the importance of knowing what was painted.

8.4 Weighted Aggregation

Now comes the crucial step where everything comes together. We have attention weights α that tell us how much to focus on each token, and we have value vectors v_i that contain the information from each token. We combine these through weighted aggregation.

The output representation for "painted" is computed as a weighted sum of all value vectors:

$$o_{\text{painted}} = \sum_{i=1}^{n} \alpha_i v_i \tag{36}$$

Let's expand this to see all the terms:

$$\begin{aligned} o_{\text{painted}} = {} & 0.0355 \cdot v_{[CLS]} + 0.0502 \cdot v_{\text{The}} && (37)\\ & + 0.2372 \cdot v_{\text{artist}} + 0.3770 \cdot v_{\text{painted}} && (38)\\ & + 0.0736 \cdot v_{\text{the}} + 0.1862 \cdot v_{\text{portrait}} && (39)\\ & + 0.0401 \cdot v_{[SEP]} && (40) \end{aligned}$$

This is a weighted average, but not a simple average. Tokens with high attention weights contribute more to the output. Since "painted" has an attention weight of 0.3770 to itself, 37.7% of its output representation comes from its own value vector. Another 23.7% comes from "artist" and 18.6% from "portrait"—the semantically relevant words we identified earlier.

8.5 Interpretation of the Output

The resulting output vector o_{painted} is what we call a context-aware representation. Let's break down what makes it special:

First, it's context-aware because it incorporates information from other tokens in the sequence. The output for "painted" isn't just based on the word "painted" in isolation—it includes weighted contributions from "artist," "portrait," and other words in the sentence.

Second, the weighting is learned and adaptive. The model learned during training that verbs should attend strongly to their subjects and objects. This wasn't hard-coded—it emerged from the data.

Third, it's task-relevant. The value transformation W^V learned to extract features that are useful for the downstream task. If the model was trained for sentiment analysis, the values might encode emotional connotations. If trained for named entity recognition, they might encode entity type information.

The beauty of this mechanism is that the same architecture can learn completely different attention patterns depending on the task and training data. A model trained on scientific text might learn different attention patterns than one trained on conversational text, even though they use the same fundamental attention mechanism.

9 Multi-Head Attention

9.1 Motivation: Why One Head Isn't Enough

So far, we've been working with a single attention mechanism—one set of query, key, and value matrices that transform our embeddings and compute attention as shown in Figure 3. This is called single-head attention. While powerful, single-head attention has fundamental limitations that become apparent when we think about the complexity of language.

Consider our example sentence again: "The artist painted the portrait." There are multiple types of relationships we might want to capture simultaneously. There's the syntactic relationship: "artist" is the subject, "painted" is the verb, "portrait" is the object. There's the semantic relationship: "artist" and "painted" are thematically related (one performs the other), while "painted" and "portrait" have a different relationship (one affects the other). There might be positional relationships: adjacent words often have special significance in English.

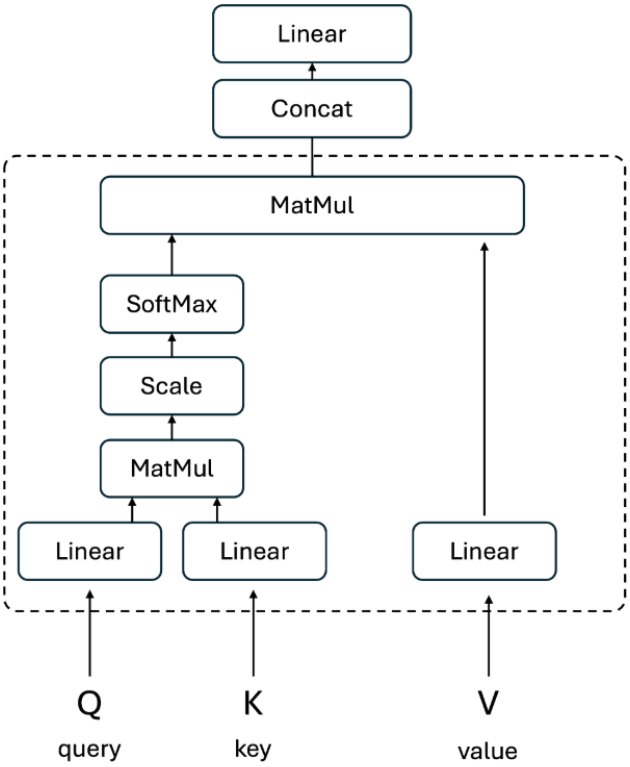

Figure 3: Attention mechanism for one head.

A single attention head can only learn one transformation of the embedding space. It might learn to capture subject-verb relationships well, but in doing so, it might sacrifice its ability to capture other types of relationships. This is where multi-head attention comes in.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. Instead of having one attention mechanism, we have multiple attention mechanisms operating in parallel, each potentially learning to focus on different aspects of the linguistic structure. Figure 4 illustrates a transformer layer incorporating multi-head attention.

9.2 Mathematical Formulation

We begin by recalling the attention mechanism in its simplest form. In *single-head attention*, a single set of projection matrices is used to map the input representations X into queries, keys, and values:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V,$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$. The attention operation is then applied once using these projected representations.

Multi-head attention generalizes this idea by allowing the model to attend to information from multiple representation subspaces simultaneously. Instead of using a single projection for queries, keys, and values, the model employs h distinct sets of projection matrices. Concretely, the original projection matrices are conceptually partitioned into h smaller matrices,

$$W^Q = [W_1^Q \mid W_2^Q \mid \ldots \mid W_h^Q], \quad W^K = [W_1^K \mid \ldots \mid W_h^K], \quad W^V = [W_1^V \mid \ldots \mid W_h^V],$$

where each $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$.

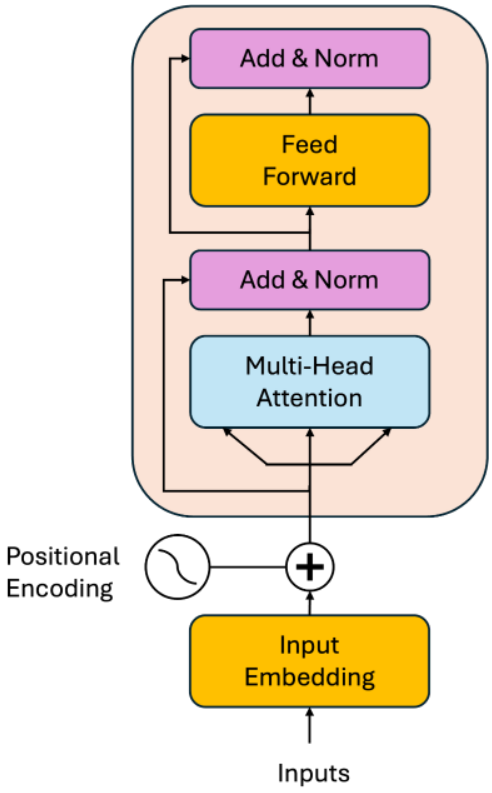

Figure 4: Multi-head attention mechanism

The multi-head attention mechanism is then defined as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O, \tag{41}$$

where each head is computed independently as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V). \tag{42}$$

Each head has its own learned parameters W_i^Q, W_i^K, and W_i^V, enabling different heads to focus on different relational patterns within the sequence. Importantly, these projection matrices are applied *directly to the input representations* X, not to previously computed queries, keys, or values. In other words, one should not interpret the formulation as replacing an already-formed Q with QW_i^Q.

For clarity, the correct per-head projections are

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V,$$

rather than applying an additional projection to a shared Q, K, or V. The notation

$$\text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

is therefore a compact mathematical expression indicating per-head projections from the same input, not a sequential re-projection of previously computed matrices.

After computing all heads independently, their outputs are concatenated and passed through a final linear transformation $W^O \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, which projects the combined representation back to the original model dimension.

BERT-base configuration: BERT-base has 12 transformer layers stacked on top of each other. Each layer contains 12 attention heads operating in parallel. This gives us a total of $12 \times 12 = 144$ attention heads

throughout the entire model. The model dimension is $d_{model} = 768$, which means each token is represented as a 768-dimensional vector. For attention, this 768-dimensional space is split across the 12 heads, giving each head $d_k = d_v = 768/12 = 64$ dimensions to work with.

DistilBERT configuration: DistilBERT is a distilled (compressed) version of BERT that maintains much of its performance while being smaller and faster. It has only 6 transformer layers (half of BERT), but each layer still has 12 attention heads, giving $6 \times 12 = 72$ total attention heads. The model dimension remains $d_{model} = 768$, so each head still works with $d_k = d_v = 64$ dimensions.

This design choice—splitting the model dimension across heads rather than giving each head the full dimension—is intentional. It allows different heads to specialize in different subspaces without exploding the parameter count.

9.3 Concatenation and Projection

Let's work through what happens to the dimensions in multi-head attention. Suppose we're working with BERT-base and processing our 7-token sentence. Each of the 12 attention heads processes the input independently. Each head outputs a 7×64 matrix (7 tokens, 64 dimensions per head). When we concatenate all 12 heads, we get:

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_{12}) \in \mathbb{R}^{7 \times (12 \times 64)} = \mathbb{R}^{7 \times 768} \tag{43}$$

Notice that by concatenating, we've reconstructed the original model dimension of 768. Now we have a 7×768 matrix where each token's representation is the concatenation of its representations from all 12 heads.

Finally, we apply the output projection:

$$\text{Output} = \text{Concat}(\text{heads})W^O \tag{44}$$

where $W^O \in \mathbb{R}^{768 \times 768}$ is a learned weight matrix. This final transformation allows the model to mix information across the different heads. Without this projection, information from different heads would remain isolated—the first 64 dimensions would only contain information from head 1, the next 64 only from head 2, and so on. The output projection allows the model to learn how to optimally combine the different perspectives provided by different heads.

10 Specialized Roles of Attention Heads

10.1 Layer-wise Behavior in BERT

So far, we have described the attention mechanism and how queries, keys, and values are computed for a single layer. In practice, BERT and similar Transformer models are composed of multiple stacked layers—12 layers in BERT-base, and up to 24 in BERT-large. Each layer contains multiple attention heads, and each head can learn to capture different patterns in the input sequence.

Because layers are stacked, early layers tend to focus on more basic, local patterns, while later layers gradually capture higher-level, more abstract relationships (see Figure 5). Below, we describe typical patterns observed across layers.

10.2 Early Layers: Syntactic and Positional Patterns

Layers 1-2 of BERT typically learn fundamental syntactic structure. Some heads in these layers become experts at identifying syntactic dependencies—relationships like subject-verb agreement, determiner-noun pairs, and preposition-object connections. For instance, one head might learn to make "artist" attend strongly to "painted" because it recognizes the grammatical subject-verb relationship.

Other heads in early layers focus on part-of-speech patterns. They learn to recognize that certain positions in a sentence are likely to be occupied by certain types of words—nouns following determiners, verbs following subjects, adjectives preceding nouns. This helps the model build a structural scaffold for understanding the sentence.

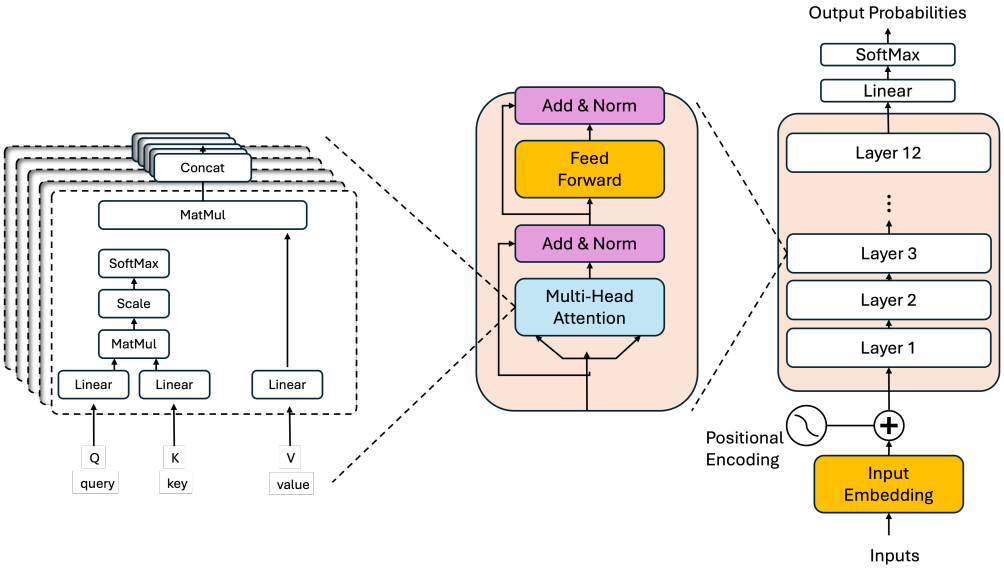

Figure 5: The Transformer - model architecture.

Some early-layer heads attend primarily to adjacent tokens, capturing local context. In English, adjacent words often form meaningful units (noun phrases, verb phrases), so this positional attention helps identify these linguistic chunks. For example, in "the artist," the word "the" might attend strongly to "artist" simply because they're adjacent and form a determiner-noun phrase.

10.3 Middle Layers: Semantic Relationships

Layers 3-6 (in BERT-base) transition to more semantic focus. These layers contain heads that specialize in capturing meaning-based relationships rather than purely structural ones.

Some middle-layer heads excel at coreference resolution—understanding when different words refer to the same entity. For example, in "The artist painted the portrait. She was talented," a coreference head would learn to make "She" attend to "artist," understanding they refer to the same person.

Other heads in these layers focus on semantic role labeling—identifying who did what to whom. They learn patterns like agent-action-patient relationships. In our example, such a head might learn that "artist" is the agent, "painted" is the action, and "portrait" is the patient (the thing being affected).

Named entity patterns also emerge in middle layers. Some heads learn to attend between words that are part of the same named entity or between entities that are related. If our sentence mentioned "Vincent van Gogh" and "Starry Night," a head might learn to connect these as a painter-painting relationship.

10.4 Deep Layers: Abstract Reasoning

Layers 7-12 (in BERT-base) perform the most abstract processing. These layers contain heads that engage in complex semantic reasoning that goes beyond simple word relationships.

Some deep-layer heads specialize in discourse structure—understanding how sentences relate to each other and how information flows through a paragraph. They might learn that certain words signal cause-and-effect relationships, or that certain phrases indicate contrasting ideas.

Other heads in deep layers become highly task-specific. If BERT is fine-tuned for sentiment analysis, some deep heads might learn to attend from sentiment words to the aspects they modify. If fine-tuned for question answering, some heads might learn to connect question words to relevant parts of the passage containing the answer.

The remarkable aspect is that these specializations aren't hard-coded. The same architecture, initialized with random weights, learns completely different attention patterns depending on the training data and task.

A model trained on medical text develops different head specializations than one trained on legal text, even though they use identical architectures.

10.5 Example: Different Heads Analyzing the Same Input

To make this concrete, let's consider how three different heads might analyze our example sentence "The artist painted the portrait" differently.

Head 1 (Syntactic Focus): This head has learned syntactic dependencies during training. When processing the sentence, it creates strong attention links that follow grammatical structure. The word "artist" attends strongly to "painted" (subject → verb), "painted" attends to "portrait" (verb → object), and "the" attends to the nouns it modifies. This head is essentially parsing the sentence structure, identifying the grammatical skeleton.

Head 2 (Positional Focus): This head has specialized in local context. It creates strong attention primarily between adjacent tokens. "The" attends to "artist," "artist" to "painted," "painted" to "the," "the" to "portrait." This sliding window of attention helps capture the local semantic units and phrasal structure without necessarily understanding the long-range syntactic relationships.

Head 3 (Semantic Focus): This head has learned thematic role relationships. It creates a different attention pattern focused on meaning rather than syntax or position. "Artist" attends strongly to both "painted" and "portrait" because these three words form a semantic unit—they describe a complete event with an agent, action, and patient. "Painted" might also attend to both "artist" and "portrait" to understand the full context of the action.

The power of multi-head attention is that the model gets all three of these perspectives simultaneously. It doesn't have to choose between understanding syntax, position, or semantics—it can learn to use all of them together to build a rich, multi-faceted representation of the sentence.

11 Python Implementation

In this section, we demonstrate how to prepare textual data for input into transformer models using the Hugging Face `transformers` library. The goal is to convert raw text into a format that BERT can process, which involves tokenization, encoding, and proper formatting with special tokens, padding, and attention masks. We will walk through the key variables involved in this preprocessing pipeline, provide detailed examples for single and multi-sentence inputs, show how to extract embeddings from the model, and conclude with the complete preprocessed data structure ready for the transformer.

11.1 Key Variables in Text Preprocessing

Before exploring examples, it is important to understand the key variables produced during the tokenization and encoding process:

- **Tokens**: The subword units produced by breaking down the input text according to the model's vocabulary. For example, "mathematics" might remain as one token, while rarer words could be split into multiple subwords.
- **Token IDs (input_ids)**: Numerical representations of tokens, where each token is mapped to a unique integer from the model's vocabulary. These IDs are what the model actually processes.
- **Special Tokens**: Transformers require special tokens to mark sentence boundaries. The `[CLS]` token (ID 101) appears at the start of every input, and `[SEP]` (ID 102) marks the end of each sentence.
- **Attention Mask**: A binary mask where 1 indicates a real token and 0 indicates padding. This tells the model which tokens to attend to and which to ignore.
- **Token Type IDs (Segment IDs)**: Used to distinguish between multiple sentences in a single input. The first sentence gets 0s, the second gets 1s, and so on.

- **Padding**: When processing multiple questions or sentences together (called a batch), they often have different lengths. For example, one question might be "What is AI?" (3 tokens) while another is "How does machine learning work?" (5 tokens). To process them efficiently together, we pad the shorter sequences with zeros until all sequences in the batch have the same length (`max_length`). This ensures uniform dimensions so the model can process them as a single batch. The attention mask tells the model to ignore these padding zeros.
- **Embeddings**: Dense vector representations of tokens produced by the model. Unlike static embeddings, these are contextualized, meaning the same word receives different representations depending on its surrounding context.

11.2 Loading the Tokenizer and Model

We begin by importing the necessary libraries and loading a pre-trained BERT model along with its tokenizer.

```
from transformers import BertTokenizer, BertModel
import torch

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertModel.from_pretrained("bert-base-uncased")
```

The `bert-base-uncased` model is a 12-layer transformer trained on English text, with all text converted to lowercase. The tokenizer handles the conversion from text to token IDs using BERT's 30,000-word vocabulary.

11.3 Basic Tokenization Example

Let us start with a simple example to understand how text is converted into tokens and then into numerical IDs.

```
sentence = "I love mathematics!"
tokens = tokenizer.tokenize(sentence)
token_ids = tokenizer.convert_tokens_to_ids(tokens)

print("Tokens:", tokens)
print("Token IDs:", token_ids)
```

Output:

```
Tokens: ['i', 'love', 'mathematics', '!']
Token IDs: [1045, 2293, 5597, 999]
```

Here, the sentence is split into four tokens. Each token is then mapped to its corresponding ID in BERT's vocabulary. For instance, the token "i" maps to ID 1045, "love" to 2293, and so on. Note that the tokenizer converts text to lowercase because we are using the uncased version of BERT.

11.4 Complete Preprocessing with `encode_plus`

While basic tokenization works for simple cases, transformer models require a more structured input format. We now switch to DistilBERT's tokenizer to demonstrate the `encode_plus` method, which handles all preprocessing steps in one call.

```
from transformers import DistilBertTokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')

inputs = tokenizer.encode_plus(
    "I love mathematics!",
    add_special_tokens=True,
    max_length=20,
    padding='max_length',
```

```
    truncation=True,
    return_token_type_ids=True,
    return_attention_mask=True
)

print(f"Input IDs: {inputs['input_ids']}")
print(f"Attention Mask: {inputs['attention_mask']}")
print(f"Token type ids: {inputs['token_type_ids']}")
```

Output:

```
Input IDs: [101, 1045, 2293, 5597, 999, 102, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0]
Attention Mask: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0]
Token type ids: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0]
```

Let us break down what happened:

1. **Input IDs**: The sequence starts with `[CLS]` (101), followed by the token IDs for our sentence (1045, 2293, 5597, 999), then `[SEP]` (102) to mark the end. Since we set `max_length=20`, the remaining positions are padded with zeros.
2. **Attention Mask**: The first six positions (corresponding to real tokens including special tokens) have value 1, while the 14 padding positions have value 0. This tells the model to ignore the padded positions during self-attention computation.
3. **Token Type IDs**: All values are 0 because we only have one sentence. These IDs become important when processing sentence pairs, as we will see next.

The parameters used in `encode_plus` serve the following purposes:

- `add_special_tokens=True`: Automatically adds `[CLS]` and `[SEP]` tokens.
- `max_length=20`: Sets a fixed sequence length, ensuring consistent dimensions across inputs.
- `padding='max_length'`: Pads shorter sequences with zeros up to `max_length`.
- `truncation=True`: Truncates sequences longer than `max_length`.
- `return_token_type_ids=True`: Returns segment IDs for distinguishing sentences.
- `return_attention_mask=True`: Returns the binary mask for attention.

11.5 Two-Sentence Example

When working with sentence pairs (common in tasks like question answering or natural language inference), the token type IDs become crucial for distinguishing between the two sentences.

```
inputs = tokenizer.encode_plus(
    "I love mathematics!",
    "Linear algebra is at the core of machine learning",
    add_special_tokens=True,
    max_length=20,
    padding='max_length',
    truncation=True,
    return_token_type_ids=True,
    return_attention_mask=True
)

```

```
print(f"Input IDs: {inputs['input_ids']}")
print(f"Attention Mask: {inputs['attention_mask']}")
print(f"Token type ids: {inputs['token_type_ids']}")
```

Output:

```
Input IDs: [101, 1045, 2293, 5597, 999, 102, 7399, 11208, 2003,
            2012, 1996, 4563, 1997, 3698, 4083, 102, 0, 0, 0, 0]
Attention Mask: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 0, 0, 0, 0]
Token type ids: [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 0, 0, 0, 0]
```

Now the structure is more complex. The input starts with `[CLS]` (101), followed by the first sentence tokens (1045, 2293, 5597, 999), then `[SEP]` (102) to mark the end of the first sentence. The second sentence follows (7399, 11208, 2003, ...), ending with another `[SEP]` (102), and finally padding zeros.

The token type IDs now show 0 for all tokens belonging to the first sentence (including its `[SEP]`), and 1 for all tokens in the second sentence. The padding positions also receive token type ID 0 by convention, though they are ignored due to the attention mask.

11.6 Extracting Contextualized Embeddings

Having preprocessed our text, we now feed it through BERT to obtain contextualized embeddings. These embeddings capture both the semantic meaning of each token and its relationship to surrounding tokens.

```
sentence = "I love mathematics!"
tokens = tokenizer.tokenize(sentence)
token_ids = tokenizer.convert_tokens_to_ids(tokens)

# Convert token IDs to tensor and add batch dimension
input_ids = torch.tensor([token_ids])

# Get embeddings from BERT
with torch.no_grad():
    outputs = model(input_ids)
    embeddings = outputs.last_hidden_state

print("Embeddings shape:", embeddings.shape)

# Print embeddings for each token (first 10 dimensions)
for token, emb in zip(tokens, embeddings[0]):
    print(f"Token: {token}")
    print(f"Embedding (first 10 dims): {emb[:10]}")
```

Output:

```
Embeddings shape: torch.Size([1, 4, 768])
Token: i
Embedding (first 10 dims): tensor([ 0.0648,  0.3697,  0.0214,
                                    0.0242, -0.2595,  0.0748,
                                    0.3394,  0.0936, -0.2900,
                                   -0.3007])
Token: love
Embedding (first 10 dims): tensor([ 0.4542,  0.9776,  0.3335,
                                   -0.0131, -0.0401, -0.1888,
                                    0.4279, -0.0346, -0.2702,
                                   -0.5418])
Token: mathematics
Embedding (first 10 dims): tensor([ 0.0373,  1.0036,  0.0657,
```

```
                                    -0.0548,  0.0196, -0.1038,
                                     0.2634,  0.1846, -0.4612,
                                    -0.2633])
Token: !
Embedding (first 10 dims): tensor([ 0.1830,  0.9067,  0.3583,
                                    -0.0487, -0.2368, -0.2107,
                                     0.6332, -0.0442, -0.4583,
                                    -0.3439])
```

Let us understand what happened step by step:

1. **Tensor Conversion**: We convert the token IDs list into a PyTorch tensor and wrap it in a list to add the batch dimension. The shape changes from (4) to $(1, 4)$, where 1 is the batch size and 4 is the sequence length.
2. **Model Inference**: We use `torch.no_grad()` to disable gradient computation, which saves memory and speeds up inference. The `outputs` object contains various components from BERT, including hidden states from all layers.
3. **Extracting Embeddings**: We access `outputs.last_hidden_state`, which contains the final layer's output. This tensor has shape $(1, 4, 768)$, meaning 1 batch, 4 tokens, and 768 dimensions per token (BERT-base's hidden size).
4. **Inspecting Embeddings**: We iterate through tokens and their corresponding embeddings, displaying only the first 10 of 768 dimensions for brevity. Each token now has a unique 768-dimensional vector representation.

These embeddings are *contextualized*, meaning the vector for "love" in this sentence would be different from "love" in another context like "I love physics!" This is fundamentally different from static word embeddings (like Word2Vec or GloVe), where each word always has the same vector regardless of context.

11.7 Complete Preprocessed Data for Transformer Models

In practice, when feeding data to transformer models, we typically use the complete preprocessing pipeline provided by `encode_plus` rather than manually handling token IDs. The preprocessed data structure includes all necessary components:

```python
# Complete preprocessing for model input
inputs = tokenizer.encode_plus(
    "I love mathematics!",
    add_special_tokens=True,
    max_length=20,
    padding='max_length',
    truncation=True,
    return_tensors='pt',  # Return PyTorch tensors
    return_token_type_ids=True,
    return_attention_mask=True
)

# Feed to model
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print("Final embeddings shape:", embeddings.shape)
```

By setting `return_tensors='pt'`, the tokenizer directly returns PyTorch tensors instead of lists, making the data immediately ready for the model. The `**inputs` syntax unpacks the dictionary, passing `input_ids`, `attention_mask`, and `token_type_ids` as separate arguments to the model. This is the standard pipeline for preparing text data for transformer models in production systems.

Acknowledgments

Esmaeil Rezaei, Ph.D.

Thank you for reading this guide! I hope this tutorial has helped you understand how to preprocess text data for transformer models. If you found this useful, I would greatly appreciate your support by connecting with me on my professional networks.

GitHub
esmaeil-rezaei

LinkedIn
esmaeil-rezaei

Feel free to explore my repositories on GitHub for more machine learning projects and connect with me on LinkedIn to stay updated on my latest work. Your feedback and suggestions are always welcome!