"Andriy's long-awaited sequel in his "The Hundred-Page" series of machine learning textbooks is a masterpiece of concision."

— Bob van Luijt, CEO and Co-Founder of Weaviate

"Andriy has this almost supernatural talent for shrinking epic AI concepts down to bite-sized, 'Ah, now I get it!' moments."

— Jorge Torres, CEO at MindsDB

"Andriy paints for us, in 100 marvelous strokes, the journey from linear algebra basics to the implementation of transformers."

— Florian Douetteau, Co-founder and CEO at Dataiku

"Andriy's book is an incredibly concise, clear, and accessible introduction to machine learning."

— Andre Zayarni, Co-founder and CEO at Qdrant

"This is one of the most comprehensive yet concise handbooks out there for truly understanding how LLMs work under the hood."

— Jerry Liu, Co-founder and CEO at LlamaIndex

Featuring a foreword by **Tomáš Mikolov** and back cover text by **Vint Cerf**

The Hundred-Page Language Models Book

Andriy Burkov

ISBN 978-1-7780427-2-0

Publisher: True Positive Inc.

To my family, with love

"Language is the source of misunderstandings."
***—Antoine de Saint-Exupéry**, The Little Prince*

"In mathematics you don't understand things. You just get used to them."
—John von Neumann

"Computers are useless. They can only give you answers."
— Pablo Picasso

The book is distributed on the "read first, buy later" principle

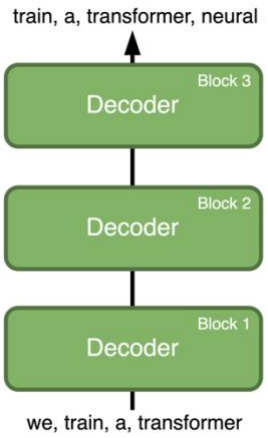
Contents

Chapter 4. Transformer

Transformer models have greatly advanced NLP. They overcome RNNs' limitations in managing long-range dependencies and enable parallel processing of input sequences. There are three main Transformer architectures: encoder-decoder, initially formulated for machine translation; encoder-only, typically used for classification; and decoder-only, commonly found in chat LMs.

In this chapter, we'll explore the decoder-only Transformer architecture in detail, as it is the most widely used approach for training **autoregressive language models**.

The transformer architecture introduces two key innovations: self-attention and positional encoding. Self-attention enables the model to assess how each word relates to all others during prediction, while positional encoding captures word order and sequential patterns. Unlike RNNs, transformers process all tokens simultaneously, using positional encoding to maintain sequential context despite parallel processing of each token. This chapter explores these fundamental elements in detail.

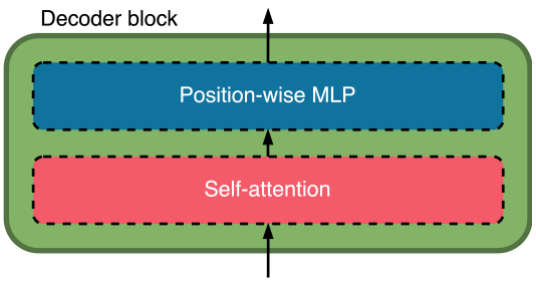


A **decoder-only Transformer** (referred to simply as "decoder" from here on) is made up of multiple identical[5] layers, known as decoder blocks, stacked vertically as shown on the right.

As you can see, training a decoder involves pairing each input sequence with a target sequence that is shifted forward by one token —the same method used for RNN-based language models.

4.1. Decoder Block

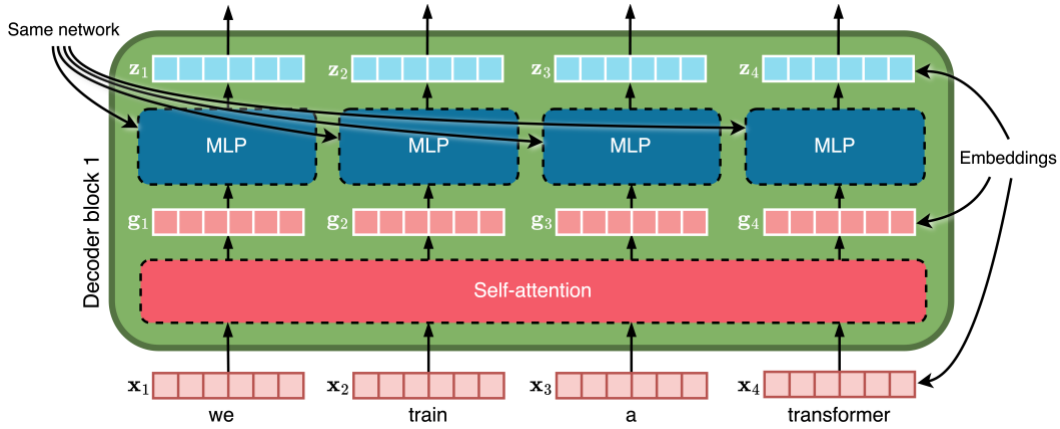
Each decoder block has two sub-layers: self-attention and a position-wise multilayer perceptron (MLP) as shown below:



The illustration simplifies certain aspects to avoid introducing too many new concepts at once. We'll introduce the missing details step by step.

Let's take a closer look at what happens in a decoder block, starting with the first one:

[5] Decoder blocks share the same architecture but have distinct trainable parameters unique to each block.



The first decoder block processes input token embeddings. For this example, we use 6-dimensional input and output embeddings, though in practice these dimensions grow larger with parameter count and token vocabulary. The **self-attention layer**, transforms each input embedding vector $\mathbf{x}_t$ into a new vector $\mathbf{g}_t$ for every token t, from 1 to L, where L represents the input length.

> Here, we simplified each unit as a square, following the same approach we used for the four-unit network in Section 1.5. While our earlier chapters showed information in a neural network flowing from left to right, we've now shifted to a bottom-to-top orientation—the standard convention for high-level language model diagrams in the literature. We'll maintain this vertical orientation from now on.

After self-attention, the position-wise MLP independently processes each vector $\mathbf{g}_t$ one at a time. Each decoder block has its own MLP with unique parameters, and within a block, this same MLP is applied independently to each position's vector, taking one $\mathbf{g}_t$ as input and producing one $\mathbf{z}_t$ as output. When the MLP finishes processing each position sequentially, the number of output vectors $\mathbf{z}_t$ equals the number of input tokens $\mathbf{x}_t$.

The output vectors $\mathbf{z}_t$ then serve as inputs to the next decoder block. This process repeats through each decoder block, preserving a number of output vectors equal to the number of input tokens $\mathbf{x}_t$.

4.2. Self-Attention

To see how **self-attention** works, let's start with an intuitive comparison. Transforming $\mathbf{g}_t$ into $\mathbf{z}_t$ is straightforward: a position-wise MLP takes an input vector and outputs a new vector by applying a learned transformation. This is what feedforward networks are designed to do. However, self-attention can seem more complex.

Consider a 5-token example: ["we," "train," "a," "transformer," "model"], and assume a decoder with a maximum input sequence length of 4.

In each decoder block, the self-attention function relies on three tensors of trainable parameters: $\mathbf{W}^Q$, $\mathbf{W}^K$, and $\mathbf{W}^V$. Here, Q stands for "query," K for "key," and V for "value."

Let's assume these tensors are 6 × 6. This means each of the four 6-dimensional input vectors will be transformed into four 6-dimensional output vectors. Let's use the second token, $\mathbf{x}_2$, representing the word "train," as our illustrative example. To compute the output $\mathbf{g}_2$ for $\mathbf{x}_2$, the self-attention layer works in six steps.

4.2.1. Step 1 of Self-Attention

Compute matrices **Q**, **K**, and **V** as shown below:

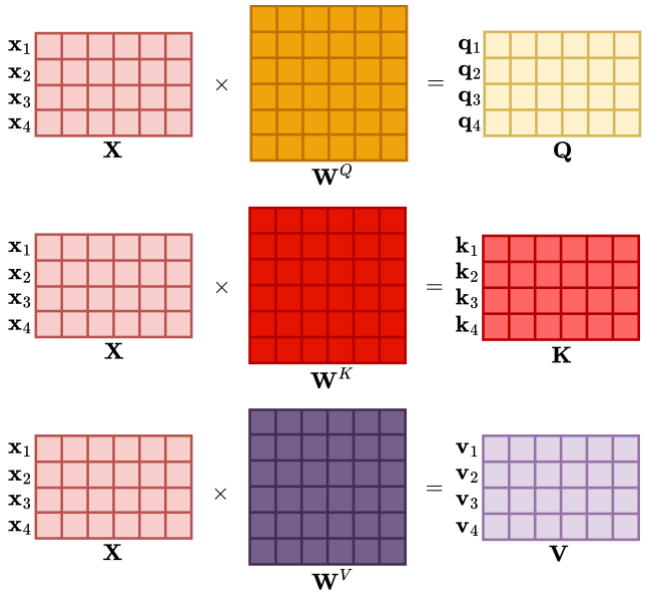


Figure 4.1: Matrix multiplication in the self-attention layer.

In the illustration, we combined the four input embeddings $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, and $\mathbf{x}_4$ into a matrix **X**. Then, we multiplied **X** by the weight matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, and $\mathbf{W}^V$ to create matrices **Q**, **K**, and **V**. These matrices hold 6-dimensional query, key, and value vectors, respectively. Since the process generates the same number of query, key, and value vectors as input embeddings, each input embedding $\mathbf{x}_t$ corresponds to a query vector $\mathbf{q}_t$, a key vector $\mathbf{k}_t$, and a value vector $\mathbf{v}_t$.

4.2.2. Step 2 of Self-Attention

Taking the second token $\mathbf{x}_2$ as our example, we compute **attention scores** by taking the dot product of its query vector $\mathbf{q}_2$ with each key vector $\mathbf{k}_t$. Let's assume the resulting scores are:

$$\mathbf{q}_2 \cdot \mathbf{k}_1 = 4.90, \quad \mathbf{q}_2 \cdot \mathbf{k}_2 = 17.15, \quad \mathbf{q}_2 \cdot \mathbf{k}_3 = 9.80, \quad \mathbf{q}_2 \cdot \mathbf{k}_4 = 12.25$$

In vector format:

$$\mathbf{scores}_2 = [4.90, 17.15, 9.80, 12.25]^\top$$

4.2.3. Step 3 of Self-Attention

To obtain the **scaled scores**, we divide each attention score by the square root of the key vector's dimensionality. In our example, since the key vector has a dimensionality of 6, we divide all scores by $\sqrt{6} \approx 2.45$, yielding:

$$\mathbf{scaled\_scores}_2 = \left[\frac{4.9}{2.45}, \frac{17.15}{2.45}, \frac{9.8}{2.45}, \frac{12.25}{2.45}\right]^\top = [2,7,4,5]^\top$$

4.2.4. Step 4 of Self-Attention

We then apply the **causal mask** to the scaled scores. (If the reason for using the causal mask isn't clear yet, it will be explained in detail soon.) For the second input position, the causal mask is:

$$\mathbf{causal\_mask}_2 \stackrel{\text{def}}{=} [0,0,-\infty,-\infty]^\top$$

We add the scaled scores to the causal mask, resulting in the **masked scores**:

$$\mathbf{masked\_scores}_2 = \mathbf{scaled\_scores}_2 + \mathbf{causal\_mask}_2 = [2,7,-\infty,-\infty]^\top$$

4.2.5. Step 5 of Self-Attention

We apply the **softmax** function to the masked scores to produce the **attention weights**:

$$\mathbf{attention\_weights}_2 = \text{softmax}([2,7,-\infty,-\infty]^\top)$$

Since scores of $-\infty$ become zero after applying the exponential function, the attention weights for the third and fourth positions will be zero. The remaining two weights are calculated as:

$$\mathbf{attention\_weights}_2 = \left[\frac{e^2}{e^2+e^7}, \frac{e^7}{e^2+e^7}, 0, 0\right]^\top \approx [0.0067,0.9933,0,0]^\top$$

> Dividing attention scores by the square root of the key dimensionality helps prevent the dot products from growing too large in magnitude as the dimensionality increases, which could lead to extremely small gradients after applying **softmax** (due to very large negative or positive values pushing the softmax outputs to 0 or 1).

4.2.6. Step 6 of Self-Attention

We compute the output vector $\mathbf{g}_2$ for the input embedding $\mathbf{x}_2$ by taking a weighted sum of the value vectors $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, and $\mathbf{v}_4$ using the attention weights from the previous step:

$$\mathbf{g}_2 \approx 0.0067 \cdot \mathbf{v}_1 + 0.9933 \cdot \mathbf{v}_2 + 0 \cdot \mathbf{v}_3 + 0 \cdot \mathbf{v}_4$$

As you can see, the decoder's output for position 2 depends only on (or, we can say "attends only to") the inputs at positions 1 and 2, with position 2 having a much stronger influence. This effect comes from the **causal mask**, which restricts the model from attending to future positions when generating an output for a given position. This property is essential for maintaining the **autoregressive** nature of language models, ensuring that predictions for each position rely solely on previous and current inputs, not future ones.

> While this token primarily attends to itself in our example, attention patterns vary across different contexts. A token may attend strongly to other tokens providing relevant semantic or syntactic information, depending on sentence structure.

The vectors $\mathbf{q}_t$, $\mathbf{k}_t$, and $\mathbf{v}_t$ can be interpreted as follows: each input position (token or embedding) seeks information about other positions. For example, a token like "I" might look for a name in another position, allowing the model to process "I" and the name in a similar way. To enable this, each position t is assigned a query $\mathbf{q}_t$.

The self-attention mechanism calculates a **dot product** between $\mathbf{q}_t$ and every key $\mathbf{k}_p$ across all positions p. A larger dot-product indicates greater similarity between the vectors. If position p's key $\mathbf{k}_p$ aligns closely with position t's query $\mathbf{q}_t$, then position p's value $\mathbf{v}_p$ contributes more significantly to the final result.

> The concept of attention emerged before the Transformer. In 2014, Dzmitry Bahdanau, while studying under Yoshua Bengio, addressed a fundamental challenge in machine translation: enabling an RNN to focus on the most relevant parts of a sentence. Drawing from his own experience learning English—where he moved his focus between different parts of the text—Bahdanau developed a mechanism for the RNN to "decide" which input words were most important at each translation step. This mechanism, which Bengio then termed attention, became a cornerstone of modern neural networks.

The process used to calculate $\mathbf{g}_2$ is repeated for each position in the input sequence, resulting in a set of output vectors: $\mathbf{g}_1$, $\mathbf{g}_2$, $\mathbf{g}_3$, and $\mathbf{g}_4$. Each position has its own causal mask, so when calculating $\mathbf{g}_1$, $\mathbf{g}_3$, and $\mathbf{g}_4$, a different causal mask is applied for each position. The full causal mask for all positions is shown below:

$$\mathbf{M} \stackrel{\text{def}}{=} \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

As you can see, the first token attends only to itself, the second to itself and the first, the third to itself and the first two, and the last to itself and all preceding tokens.

The general formula for computing attention for all positions is:

$$\mathbf{G} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \stackrel{\text{def}}{=} \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}$$

Here, $\mathbf{Q}$ and $\mathbf{V}$ are $L \times d_k$ query and value matrices. $\mathbf{K}^\top$ is the $d_k \times L$ transposed key matrix. d_k is the dimensionality of the key, query, and value vectors, and L is the sequence length.

While we computed the attention scores explicitly for $\mathbf{x}_2$ earlier, the matrix multiplication $\mathbf{Q}\mathbf{K}^\top$ calculates the scores for all positions at once. This method makes the process much faster.

This completes the definition of self-attention.

4.3. Position-Wise Multilayer Perceptron

After the masked self-attention layer, each output vector $\mathbf{g}_t$ is individually processed by a **multilayer perceptron** (MLP). The MLP applies a sequence of additional transformations:

$$\mathbf{z}_t = \mathbf{W}_2\big(\mathrm{ReLU}(\mathbf{W}_1\mathbf{g}_t + \mathbf{b}_1)\big) + \mathbf{b}_2$$

Here, $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{b}_1$, and $\mathbf{b}_2$ are learned parameters. The resulting vector $\mathbf{z}_t$ is then either passed to the next decoder block or, if it's the final decoder block, used to generate the output vector.

> This component is a position-wise multilayer perceptron, which is why I use that term. The literature may refer to it as a feedforward network, dense layer, or fully connected layer, but these names can be misleading. The entire Transformer is a feedforward neural network. Additionally, dense or fully connected layers typically incorporate one weight matrix, one bias vector, and an output non-linearity. The position-wise MLP in a Transformer, however, utilizes two weight matrices, two bias vectors, and omits an output non-linearity.

4.4. Rotary Position Embedding

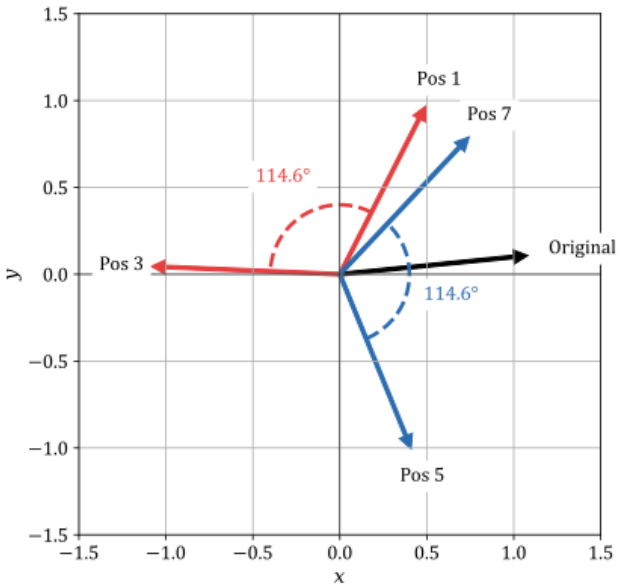
The Transformer architecture, as described so far, does not inherently account for word order. The **causal mask** ensures that each token cannot attend to tokens on its right, but rearranging tokens on the left does not affect the attention weights of a given token. This is unlike RNNs, where hidden states are computed sequentially, each depending on the previous one. Changing word order in RNNs alters the hidden states and, consequently, the output. In contrast, Transformers calculate attention across all tokens at once, without sequential dependency.

To handle word order, Transformers need to incorporate positional information. A widely used method for this is **rotary position embedding** (RoPE), which applies position-dependent rotations to the query and key vectors in the attention mechanism. One key benefit of RoPE is its ability to generalize effectively to sequences longer than those seen during training. This allows models to be trained on shorter sequences—saving time and computational resources—while still supporting much longer contexts at inference.

RoPE encodes positional information by rotating the query and key vectors. This rotation occurs before the attention computation. The illustration on the next page shows how it works in 2D. The black arrow labeled "Original" shows a position-less key or query vector in self-attention. RoPE embeds positional information by rotating this vector according to the token's position.[6] The colored arrows show the resulting rotated vectors for positions 1, 3, 5, and 7.

A key property of RoPE is that the angle between any two rotated vectors encodes the distance between their positions in the sequence. For example, the angle between positions 1 and 3 is the same as the angle between positions 5 and 7, since both pairs are two positions apart.

[6] In practice, RoPE operates by rotating pairs of adjacent dimensions within query and key vectors, rather than rotating the entire vectors themselves, as we will explore shortly.



So, how do we rotate vectors? We use matrix multiplication! **Rotation matrices** are widely used in fields like computer graphics to rotate 3D scenes—one of the original purposes of GPUs (the "G" in GPU stands for graphical) before they were applied to neural network training.

In two dimensions, the rotation matrix for an angle θ is:

$$\mathbf{R}_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Let's rotate the two-dimensional vector $\mathbf{q} = [2,1]^\top$. To do this, we multiply $\mathbf{q}$ by the rotation matrix $\mathbf{R}_\theta$. The result is a new vector, representing $\mathbf{q}$ rotated counterclockwise by an angle θ.

For a 45° rotation ($\theta = \pi/4$ radians), we can use the special values $\cos(\theta) = \sin(\theta) = \frac{\sqrt{2}}{2}$. This gives us the rotation matrix:

$$\mathbf{R}_{45^\circ} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

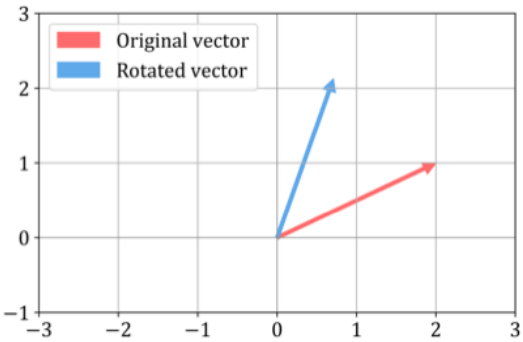
To find the rotated vector, we multiply $\mathbf{R}_{45^\circ}$ by $\mathbf{q}$:

$$\mathbf{q}\text{rotated} = \mathbf{R}_{45^\circ} \cdot \mathbf{q} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Computing this multiplication step by step:

$$\mathbf{q}_{\text{rotated}} = \begin{bmatrix} \frac{\sqrt{2}}{2} \cdot 2 - \frac{\sqrt{2}}{2} \cdot 1 \\ \frac{\sqrt{2}}{2} \cdot 2 + \frac{\sqrt{2}}{2} \cdot 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2}(2-1) \\ \frac{\sqrt{2}}{2}(2+1) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \cdot 1 \\ \frac{\sqrt{2}}{2} \cdot 3 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{3\sqrt{2}}{2} \end{bmatrix}$$

The figure below illustrates $\mathbf{q}$ and its rotated version for $\theta = 45°$:



For a position t, RoPE rotates each pair of dimensions in the query and key vectors defined as:

$$\begin{aligned} \mathbf{q}_t &= \left[q_t^{(1)}, q_t^{(2)}, \ldots, q_t^{(d_q-1)}, q_t^{(d_q)}\right]^\top \\ \mathbf{k}_t &= \left[k_t^{(1)}, k_t^{(2)}, \ldots, k_t^{(d_k-1)}, k_t^{(d_k)}\right]^\top \end{aligned}$$

Here, d_q and d_k are the (even) dimensionality of the query and key vectors. RoPE rotates pairs of dimensions indexed as $(2p-1,\ 2p)$, where each pair's index p ranges from 1 to $d_q/2$.

To split the dimensions of $\mathbf{q}_t$ into $d_q/2$ pairs, we group them like this:

$$\left[q_t^{(1)}, q_t^{(2)}\right]^\top, \left[q_t^{(3)}, q_t^{(4)}\right]^\top, \ldots, \left[q_t^{(d_q-1)}, q_t^{(d_q)}\right]^\top$$

When we write $\mathbf{q}_t(p)$, it represents the pair $\left[q_t^{(2p-1)}, q_t^{(2p)}\right]$. For example, $\mathbf{q}_t(3)$ corresponds to:

$$\left[q_t^{(2\cdot 3-1)}, q_t^{(2\cdot 3)}\right] = \left[q_t^{(5)}, q_t^{(6)}\right]$$

Each pair p undergoes a rotation based on the token position t and a **rotation frequency** θ_p:

$$\text{RoPE}\left(\mathbf{q}_t(p)\right) \overset{\text{def}}{=} \begin{bmatrix} \cos(\theta_p t) & -\sin(\theta_p t) \\ \sin(\theta_p t) & \cos(\theta_p t) \end{bmatrix} \begin{bmatrix} q_t^{(2p-1)} \\ q_t^{(2p)} \end{bmatrix}$$

Applying the **matrix-vector multiplication** rule, the rotation results in the following 2D vector:

$$\text{RoPE}\left(\mathbf{q}_t(p)\right) = \left[q_t^{(2p-1)}\cos(\theta_p t) - q_t^{(2p)}\sin(\theta_p t),\ q_t^{(2p-1)}\sin(\theta_p t) + q_t^{(2p)}\cos(\theta_p t)\right]^\top,$$

where θ_p is the rotation frequency for the p^{th} pair. It is defined as:

$$\theta_p \overset{\text{def}}{=} \frac{1}{\Theta^{2(p-1)/d_q}}$$

Here, Θ is a constant. Initially set to 10,000, later experiments demonstrated that higher values of Θ—such as 500,000 (used in Llama 2 and 3 series of models) or 1,000,000 (in Qwen 2 and 2.5 series)—enable support for larger context sizes (hundreds of thousands of tokens).

The full rotated embedding $\mathrm{RoPE}(\mathbf{q}_t)$ is constructed by concatenating all the rotated pairs:

$$\mathrm{RoPE}(\mathbf{q}_t) \stackrel{\text{def}}{=} \mathrm{concat}\left[\mathrm{RoPE}\big(\mathbf{q}_t(1)\big),\ \mathrm{RoPE}\big(\mathbf{q}_t(2)\big), \ldots,\ \mathrm{RoPE}\left(\mathbf{q}_t\big(d_q/2\big)\right)\right]$$

Note how the rotation frequency θ_p decreases quickly for each subsequent pair because of the exponential term in the denominator. This enables RoPE to capture fine-grained local position information in the early dimensions, where rotations are more frequent, and coarse-grained global position information in the later dimensions, where rotations slow down. This combination creates richer positional encoding, allowing the model to differentiate token positions in a sequence more effectively than using a single rotation frequency across all dimensions.

To illustrate the process, consider a 6-dimensional query vector at position t and $\Theta = 10{,}000$:

$$\mathbf{q}_t = \left[q_t^{(1)}, q_t^{(2)}, q_t^{(3)}, q_t^{(4)}, q_t^{(5)}, q_t^{(6)}\right]^\top \stackrel{\text{def}}{=} [0.8,0.6,0.7,0.3,0.5,0.4]^\top$$

First, we split it into three pairs ($d_q/2 = 3$):

$$\mathbf{q}_t(1) = \left[q_t^{(1)}, q_t^{(2)}\right] = [0.8,0.6]^\top$$
$$\mathbf{q}_t(2) = \left[q_t^{(3)}, q_t^{(4)}\right] = [0.7,0.3]^\top$$
$$\mathbf{q}_t(3) = \left[q_t^{(5)}, q_t^{(6)}\right] = [0.5,0.4]^\top$$

Each pair p undergoes a rotation by angle $\theta_p t$, where:

$$\theta_p = \frac{1}{10000^{2(p-1)/d_q}}$$

Let the position t be 100. First, we calculate the rotation angles for each pair (in radians):

$$\theta_1 = \frac{1}{10000^{2(1-1)/6}} = \frac{1}{10000^{0/6}} = 1.0000, \quad \text{therefore: } \theta_1 t = 100.00$$
$$\theta_2 = \frac{1}{10000^{2(2-1)/6}} = \frac{1}{10000^{2/6}} \approx 0.0464, \quad \text{therefore: } \theta_2 t = 4.64$$
$$\theta_3 = \frac{1}{10000^{2(3-1)/6}} = \frac{1}{10000^{4/6}} \approx 0.0022, \quad \text{therefore: } \theta_3 t = 0.22$$

The rotated pair 1 is:

$$\mathrm{RoPE}\big(\mathbf{q}_{100}(1)\big) = \begin{bmatrix}\cos(100) & -\sin(100)\\ \sin(100) & \cos(100)\end{bmatrix}\begin{bmatrix}0.8\\0.6\end{bmatrix} \approx \begin{bmatrix}0.86 & 0.51\\ -0.51 & 0.86\end{bmatrix}\begin{bmatrix}0.8\\0.6\end{bmatrix} = [0.99,0.11]^\top$$

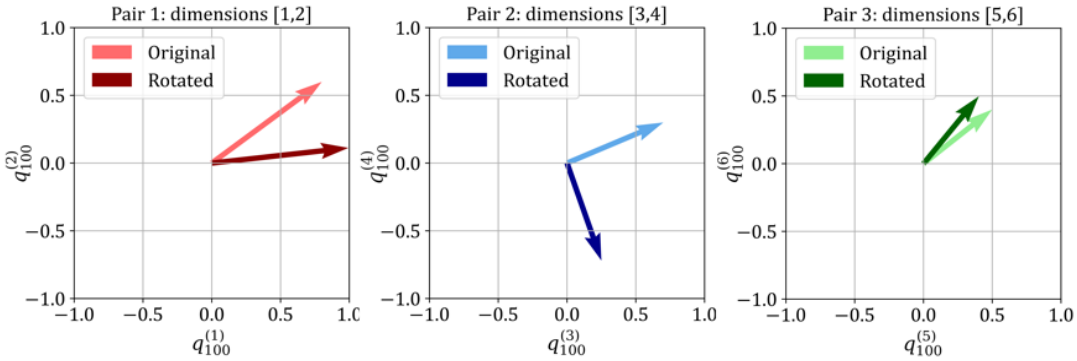
The rotated pair 2 is:

$$\mathrm{RoPE}\big(\mathbf{q}_{100}(2)\big) = \begin{bmatrix}\cos(4.64) & -\sin(4.64)\\ \sin(4.64) & \cos(4.64)\end{bmatrix}\begin{bmatrix}0.7\\0.3\end{bmatrix} \approx \begin{bmatrix}-0.07 & 1.00\\ -1.00 & -0.07\end{bmatrix}\begin{bmatrix}0.7\\0.3\end{bmatrix} = [0.25,-0.72]^\top$$

The rotated pair 3 is:

$$\mathrm{RoPE}\big(\mathbf{q}_{100}(3)\big) = \begin{bmatrix}\cos(0.22) & -\sin(0.22)\\ \sin(0.22) & \cos(0.22)\end{bmatrix}\begin{bmatrix}0.5\\0.4\end{bmatrix} \approx \begin{bmatrix}0.98 & -0.21\\ 0.21 & 0.98\end{bmatrix}\begin{bmatrix}0.5\\0.4\end{bmatrix} = [0.40,0.50]^\top$$

These is what the original and rotated pairs look like when plotted:



The final RoPE-encoded vector is the concatenation of these pairs:

$$\text{RoPE}(\mathbf{q}_{100}) \approx [0.99, 0.11, 0.25, -0.72, 0.40, 0.50]^\top$$

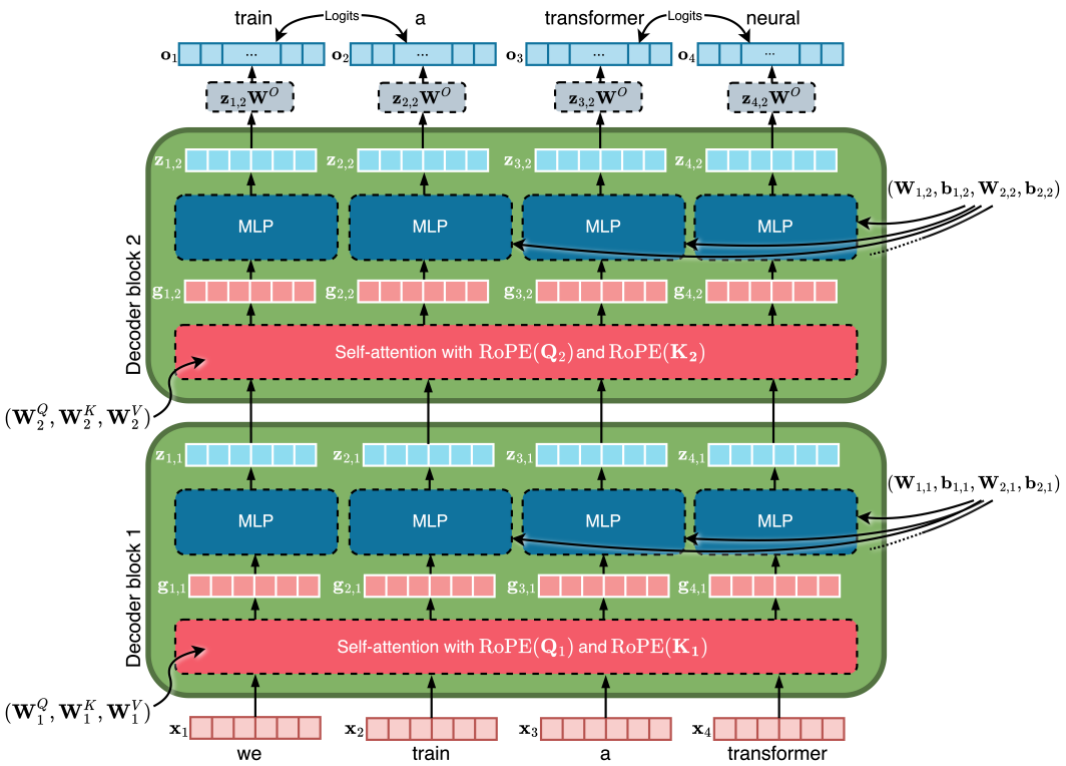
The math for RoPE($\mathbf{k}_t$) is the same as for RoPE($\mathbf{q}_t$). In each decoder block, RoPE is applied to each row of the query ($\mathbf{Q}$) and key ($\mathbf{K}$) matrices within the self-attention mechanism.

> Value vectors only provide the information that is selected and combined after the attention weights are determined. Since the positional relationships are already captured in the query-key alignment, value vectors don't need their own rotary embeddings. In other words, the value vectors simply "deliver" the content once the positional-aware attention has identified where to look.

Recall that $\mathbf{Q}$ and $\mathbf{K}$ are generated by multiplying the decoder block inputs by weight matrices $\mathbf{W}^Q$ and $\mathbf{W}^K$, as illustrated in Figure 4.1. RoPE is applied immediately after obtaining $\mathbf{Q}$ and $\mathbf{K}$, and before the attention scores are calculated.

RoPE is applied across all decoder blocks, ensuring positional information flows consistently throughout the network's depth. The illustration on the next page shows its implementation in two sequential decoder blocks. In this graph, the outputs of the second decoder block are used to compute logits for each position. This is achieved by multiplying the outputs of the final decoder block by a matrix of shape (embedding dimensionality, vocabulary size) shared across all positions. We'll explore this part in more detail when we implement the decoder model in Python.

The self-attention mechanism we've described would work as is. However, transformers typically employ an enhanced version called **multi-head attention**. This allows the model to focus on multiple aspects of information simultaneously. For example, one attention head might capture syntactic relationships, another might emphasize semantic similarities, and a third could detect long-range dependencies between tokens.



4.5. Multi-Head Attention

Once you understand self-attention, understanding multi-head attention is relatively straightforward. For each **head** h, from 1 to H, there is a separate triplet of attention matrices:

$$\left\{\left(\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V\right)\right\}_{h\in 1,\ldots,H}$$

Each triplet is applied to the input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_4$, producing H matrices $\mathbf{G}_h$. For each head, this gives four vectors $\mathbf{g}_{h,1}, \ldots, \mathbf{g}_{h,4}$, as shown in Figure 4.2 for three heads ($H = 3$). As you can see, the multi-head self-attention mechanism processes an input sequence through multiple self-attention "heads." For instance, with 3 heads, each head calculates self-attention scores for the input tokens independently. RoPE is applied separately in each head.

All input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_4$ are processed by all three heads, producing output matrices $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$. Each matrix $\mathbf{G}_h$ has as many rows as there are input tokens, meaning each head generates an embedding for every token. The embedding dimensionality of each $\mathbf{G}_h$ is reduced to one-third of the total embedding dimensionality. As a result, each head outputs lower-dimensional embeddings compared to the original embedding size.

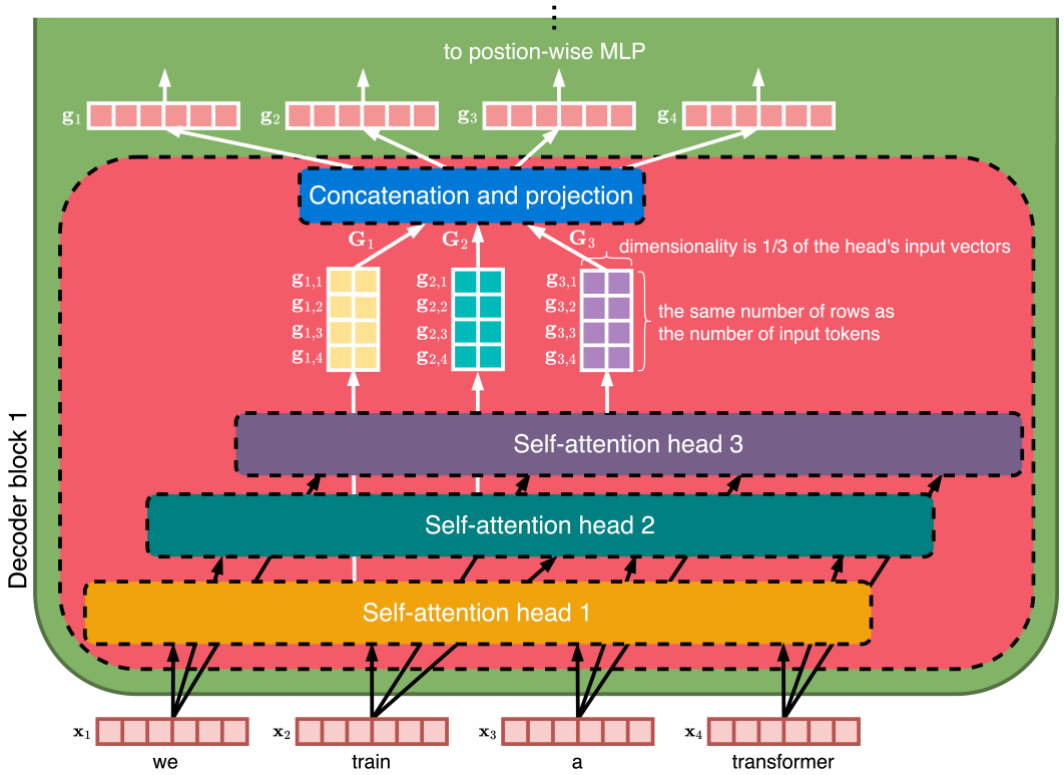
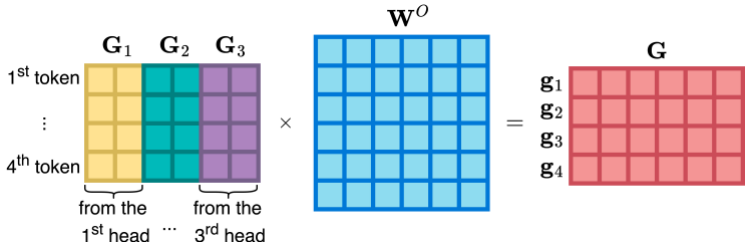


Figure 4.2: 3-head self-attention.

The outputs from the three heads are concatenated along the embedding dimension in the **concatenation and projection layer**, creating a single matrix that integrates information from all heads. This matrix is then transformed by the **projection matrix** $\mathbf{W}^O$, resulting in the final output matrix **G**. This output is passed to the position-wise MLP:



Concatenating the matrices $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$ restores the original embedding dimensionality (e.g., 6 in this case). However, applying the trainable parameter matrix $\mathbf{W}^O$ enables the model to combine the heads' information more effectively than mere concatenation.

> Modern large language models often use up to 128 heads.

At this stage, the reader understands the Transformer model architecture at a high level. Two key technical details remain to explore: layer normalization and residual connections, both essential components that enable the Transformer's effectiveness. Let's begin with residual connections.

4.6. Residual Connection

Residual connections (or **skip connections**) are essential to the Transformer architecture. They solve the vanishing gradient problem in deep neural networks, enabling the training of much deeper models.

A network containing more than two layers is called a **deep neural network**. Training them is called **deep learning**. Before **ReLU** and residual connections, the **vanishing gradient problem** severely limited network depth. Remember that during gradient descent, partial derivatives update all parameters by taking small steps in the opposite direction of the gradient. In deeper networks, these updates become very small in earlier layers (those closer to the input), effectively halting parameter adjustment. Residual connections strengthen these updates by creating pathways for the gradient to "bypass" certain layers, hence the term skip connections.

To better understand the vanishing gradient problem, let's analyze a 3-layer neural network expressed as a **composite function**:

$$f(x) = f_3\left(f_2\left(f_1(x)\right)\right),$$

where f_1 represents the first layer, f_2 represents the second layer, and f_3 represents the third (output) layer. Let these functions be defined as follows:

$$z = f_1(x) \stackrel{\text{def}}{=} w_1 x + b_1$$
$$r = f_2(z) \stackrel{\text{def}}{=} w_2 z + b_2$$
$$y = f_3(r) \stackrel{\text{def}}{=} w_3 r + b_3$$

Here, w_l and b_l are scalar weights and biases for each layer $l \in \{1,2,3\}$.

Let's define the loss function L in terms of the network output $f(x)$ and the true label y as $L(f(x), y)$. The gradient of the loss L with respect to w_1, denoted as $\frac{\partial L}{\partial w_1}$, is given by:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1} = \frac{\partial L}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial w_1},$$

where:

$$\frac{\partial f_3}{\partial f_2} = w_3, \quad \frac{\partial f_2}{\partial f_1} = w_2, \quad \frac{\partial f_1}{\partial w_1} = x$$

So, we can write:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f_3} \cdot w_3 \cdot w_2 \cdot x$$

The vanishing gradient problem occurs when weights like w_2 and w_3 are small (less than 1). When multiplied together, they produce even smaller values, causing the gradient for earlier weights such as w_1 to approach zero. This issue becomes particularly severe in networks with many layers.

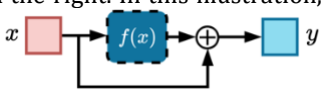
Take large language models as an example. These networks often include 32 or more decoder blocks. To simplify, assume all blocks are fully connected layers. If the average weight value is around 0.5, the gradient for the input layer parameters becomes $0.5^{32} \approx 0.0000000002$. This is extremely small.

After multiplying by the learning rate, updates to the early layers become negligible. As a result, the network stops learning effectively.

Residual connections offer a solution to the vanishing gradient problem by creating shortcuts in the gradient computation path. The basic idea is simple: instead of passing only the output of a layer to the next one, the layer's input is added to its output. Mathematically, this is written as:

$$y = f(x) + x,$$

where x is the input, $f(x)$ is the layer's computed function, and y is the output. This addition forms the residual connection. Graphically, it is shown in the picture on the right. In this illustration, the input x is processed both through the layer (represented as $f(x)$) and added directly to the layer's output.



Now let's introduce residual connections into our 3-layer network. We'll see how this changes gradient computation and mitigates the vanishing gradient issue. Starting with the original network $f(x) = f_3\left(f_2\left(f_1(x)\right)\right)$, let's add residual connections to layers 2 and 3:

$$\begin{aligned} z &\leftarrow f_1(x) \stackrel{\text{def}}{=} w_1 x + b_1 \\ r &\leftarrow f_2(z) \stackrel{\text{def}}{=} w_2 z + b_2 + z \\ y &\leftarrow f_3(r) \stackrel{\text{def}}{=} w_3 r + b_3 + r \end{aligned}$$

Our composite function becomes:

$$f(x) = w_3[w_2(w_1 x + b_1) + b_2 + w_1 x + b_1] + b_3 + w_2(w_1 x + b_1) + b_2 + w_1 x + b_1$$

Now, let's calculate the gradient of the loss L with respect to w_1:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1}$$

Expanding $\frac{\partial f}{\partial w_1}$:

$$\begin{aligned} \frac{\partial f}{\partial w_1} &= \frac{\partial}{\partial w_1}\left[\left(w_3\left(w_2(w_1 x + b_1) + b_2 + (w_1 x + b_1)\right) + b_3\right) + \left(w_2(w_1 x + b_1) + b_2 + (w_1 x + b_1)\right)\right] \\ &= (w_3 w_2 + w_3 + w_2 + 1) \cdot x \end{aligned}$$

Therefore, the full gradient is:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot (w_3 w_2 + w_3 + w_2 + 1) \cdot x$$

Comparing this to our original gradient without residual connections:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot w_3 \cdot w_2 \cdot x$$

We observe that residual connections introduce three additional terms: w_3, w_2, and 1. This guarantees that the gradient will not vanish completely, even when w_2 and w_3 are small, due to the added constant term 1.

For example, if $w_2 = w_3 = 0.5$ as in the previous case:

- **Without residual connections**: $0.5 \cdot 0.5 = 0.25$
- **With residual connections**: $0.5 \cdot 0.5 + 0.5 + 0.5 + 1 = 2.25$

The illustration below depicts an encoding block with residual connections:

As shown, each decoder block includes two residual connections. The layers are now named like Python objects, which we will implement shortly. Additionally, two RMSNorm layers have been added. Let's discuss their purpose.

4.7. Root Mean Square Normalization

The RMSNorm layer applies **root mean square normalization** to the input vector. This operation takes place just before the vector enters the self-attention layer and the position-wise MLP. Let's illustrate this with a three-dimensional vector.

Suppose we have a vector $\mathbf{x} = [x^{(1)}, x^{(2)}, x^{(3)}]^\top$. To apply RMS normalization, we first calculate the **root mean square** (**RMS**) of the vector:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(x^{(i)})^2} = \sqrt{\frac{1}{3}[(x^{(1)})^2 + (x^{(2)})^2 + (x^{(3)})^2]}$$

Then, we normalize the vector by dividing each component by the RMS value to obtain $\tilde{\mathbf{x}}$:

$$\tilde{\mathbf{x}} = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} = \left[\frac{x^{(1)}}{\text{RMS}(\mathbf{x})}, \frac{x^{(2)}}{\text{RMS}(\mathbf{x})}, \frac{x^{(3)}}{\text{RMS}(\mathbf{x})}\right]^\top$$

Finally, we apply the scale factor γ to each dimension of $\tilde{\mathbf{x}}$:

$$\bar{\mathbf{x}} = \text{RMSNorm}(\mathbf{x}) \overset{\text{def}}{=} \boldsymbol{\gamma} \odot \tilde{\mathbf{x}} = \left[\gamma^{(1)}\tilde{x}^{(1)}, \gamma^{(2)}\tilde{x}^{(2)}, \gamma^{(3)}\tilde{x}^{(3)}\right]^\top,$$

where $\odot$ denotes the **element-wise product**. The vector $\boldsymbol{\gamma}$ is a trainable parameter, and each RMSNorm layer has its own independent $\boldsymbol{\gamma}$.

The primary purpose of RMSNorm is to stabilize training by keeping the scale of the input to each layer consistent. This improves numerical stability, helping to prevent gradient updates that are excessively large or small.

Now that we've covered the key components of the Transformer architecture, let's summarize how a decoder block processes its input:

1. The input embeddings $\mathbf{x}_t$ first go through RMS normalization.
2. The normalized embeddings $\bar{\mathbf{x}}_t$ are processed by the multi-head self-attention mechanism, with RoPE applied to key and query vectors.
3. The self-attention output $\mathbf{g}_t$ is added to the original input $\mathbf{x}_t$ (residual connection).
4. This sum, $\hat{\mathbf{g}}_t$, undergoes RMS normalization again.
5. The normalized sum $\bar{\mathbf{g}}_t$ is passed through the multilayer perceptron.
6. The perceptron output $\mathbf{z}_t$ is added to the pre-RMS-normalization vector $\hat{\mathbf{g}}_t$ (another residual connection).
7. The result, $\hat{\mathbf{z}}_t$, is the output of the decoder block, serving as input for the next block (or the final output layer if it's the last block).

This sequence is repeated for each decoder block in the Transformer.

4.8. Key-Value Caching

During **training**, the decoder can process all positions in parallel because at each block it computes the query, key, and value matrices, $\mathbf{Q} = \mathbf{X}\mathbf{W}^Q$, $\mathbf{K} = \mathbf{X}\mathbf{W}^K$, and $\mathbf{V} = \mathbf{X}\mathbf{W}^V$, for the entire sequence $\mathbf{X}$. However, during an autoregressive (left-to-right) **inference**, tokens must be generated one at a time. Normally, each time we generate a new token, we would have to:

1. Calculate the key, query, and value vectors for the new token.
2. Recalculate the key and value matrices for all previous tokens.
3. Merge these with the new token's key and value vectors to compute self-attention for the new token.

Key-value caching skips step 2 by saving the key and value matrices from earlier tokens, avoiding repeated calculations. Since $\mathbf{W}^K$ and $\mathbf{W}^V$ are fixed after training, the key and value vectors of earlier

tokens stay constant during inference. These vectors can be stored ("cached") after being computed once. For every new token:

- Its key and value vectors are computed using $\mathbf{W}^K$ and $\mathbf{W}^V$.
- These vectors are appended to the cached key-value pairs for self-attention.

Query vectors, however, are not cached because they depend on the current token being processed. Every time a new token is added, its query vector must be computed on-the-fly to attend to all cached keys and values.

This approach eliminates reprocessing the rest of the sequence, cutting computation significantly for long sequences. In each decoder block, cached keys and values are stored per attention head with shapes $(L \times d_h)$ for both matrices, where L grows by one with each new token, and d_h is the dimensionality of the query, key, and value vectors for that head. For a model with H attention heads, the combined key and value caches in each decoder block have shapes $(H \times L \times d_h)$.

> RoPE applies position-dependent rotations to vectors, but this doesn't interfere with caching. When a new token arrives, it simply takes the next available position index (if the sequence has L tokens, the new one becomes position $L + 1$), while previously processed tokens retain their original positions from 1 through L. This means the cached keys, already rotated according to their respective positions, remain unchanged. The rotation is only applied to the new token at position $L + 1$.

Now that we understand how the Transformer operates, we're ready to start coding.

4.9. Transformer in Python

Let's begin implementing the decoder in Python by defining the `AttentionHead` class:

```
class AttentionHead(nn.Module):
    def __init__(self, emb_dim, d_h):
        super().__init__()
        self.W_Q = nn.Parameter(torch.empty(emb_dim, d_h))
        self.W_K = nn.Parameter(torch.empty(emb_dim, d_h))
        self.W_V = nn.Parameter(torch.empty(emb_dim, d_h))
        self.d_h = d_h

    def forward(self, x, mask):
        Q = x @ self.W_Q ❶
        K = x @ self.W_K
        V = x @ self.W_V ❷

        Q, K = rope(Q), rope(K) ❸

        scores = Q @ K.transpose(-2, -1) / math.sqrt(self.d_h) ❹
        masked_scores = scores.masked_fill(mask == 0, float("-inf")) ❺
        attention_weights = torch.softmax(masked_scores, dim=-1) ❻
        return attention_weights @ V ❼
```

This class implements a single attention head in the multi-head attention mechanism. In the constructor, we initialize three trainable weight matrices: the query matrix `W_Q`, the key matrix `W_K`, and the value matrix `W_V`. Each of these is a `Parameter` tensor of shape `(emb_dim, d_h)`, where `emb_dim` is the input embedding dimension and `d_h` is the dimensionality of the query, key, and value vectors for this attention head.

In the `forward` method:

- Lines ❶ and ❷ compute the query, key, and value matrices by multiplying the input vector `x` with the respective weight matrices. Given that `x` has shape (`batch_size`, `seq_len`, `emb_dim`), `Q`, `K`, and `V` each have shape (`batch_size`, `seq_len`, `d_h`).

- Line ❸ applies the rotary positional encoding to `Q` and `K`. After the query and key vectors are rotated, line ❹ computes the attention scores. Here's a breakdown:
    - `K.transpose(-2, -1)` swaps the last two dimensions of `K`. If `K` has shape (`batch_size`, `seq_len`, `d_h`), transposing it results in (`batch_size`, `d_h`, `seq_len`). This prepares `K` for matrix multiplication with `Q`.
    - `Q @ K.transpose(-2, -1)` performs batch matrix multiplication, resulting in a tensor of attention scores of shape (`batch_size`, `seq_len`, `seq_len`).
    - As mentioned in Section 4.2, we divide by `sqrt(d_h)` for numerical stability.

> When the matrix multiplication operator `@` is applied to tensors with more than two dimensions, PyTorch uses **broadcasting**. This technique handles dimensions that aren't directly compatible with the `@` operator, which is normally defined only for two-dimensional tensors (matrices). In this case, PyTorch treats the first dimension as the batch dimension, performing the matrix multiplication separately for each example in the batch. This process is known as **batch matrix multiplication**.

- Line ❺ applies the causal mask. The `mask` tensor has the shape (`seq_len`, `seq_len`) and contains 0s and 1s. The `masked_fill` function replaces all cells in the input matrix where `mask == 0` with negative infinity. This prevents attention to future tokens. Since the `mask` lacks the batch dimension while `scores` includes it, PyTorch uses broadcasting to apply the `mask` to the `scores` of each sequence in the batch.
- Line ❻ applies softmax to the `scores` along the last dimension, turning them into attention weights. Then, line ❼ computes the output by multiplying these attention weights with `V`. The resulting output has the shape (`batch_size`, `seq_len`, `d_h`).

Given the attention head class, we can now define the `MultiHeadAttention` class:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()
        d_h = emb_dim // num_heads ❶
        self.heads = nn.ModuleList([
            AttentionHead(emb_dim, d_h)
            for _ in range(num_heads)
        ]) ❷
```

```
        self.W_O = nn.Parameter(torch.empty(emb_dim, emb_dim)) ❸

    def forward(self, x, mask):
        head_outputs = [head(x, mask) for head in self.heads] ❹
        x = torch.cat(head_outputs, dim=-1) ❺
        return x @ self.W_O ❻
```

In the constructor:

- Line ❶ calculates `d_h`, the dimensionality of each attention head, by dividing the model's embedding dimensionality `emb_dim` by the number of heads.
- Line ❷ creates a `ModuleList` containing `num_heads` instances of `AttentionHead`. Each head takes the input dimensionality `emb_dim` and outputs a vector of size `d_h`.
- Line ❸ initializes `W_O`, a learnable **projection matrix** with shape (`emb_dim`, `emb_dim`) to combine the outputs from all attention heads.

In the `forward` method:

- Line ❹ applies each attention head to the input `x` of shape (`batch_size`, `seq_len`, `emb_dim`). Each head's output has shape (`batch_size`, `seq_len`, `d_h`).
- Line ❺ concatenates all heads' outputs along the last dimension. The resulting `x` has shape (`batch_size`, `seq_len`, `emb_dim`) since `num_heads * d_h = emb_dim`.
- Line ❻ multiplies the concatenated output by the projection matrix `W_O`. The output has the same shape as input.

Now that we have multi-head attention, the last piece needed for the decoder block is the position-wise multilayer perceptron. Let's define it:

```
class MLP(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.W_1 = nn.Parameter(torch.empty(emb_dim, emb_dim * 4))
        self.B_1 = nn.Parameter(torch.empty(emb_dim * 4))
        self.W_2 = nn.Parameter(torch.empty(emb_dim * 4, emb_dim))
        self.B_2 = nn.Parameter(torch.empty(emb_dim))

    def forward(self, x):
        x = x @ self.W_1 + self.B_1 ❶
        x = torch.relu(x) ❷
        x = x @ self.W_2 + self.B_2 ❸
        return x
```

In the constructor, we initialize learnable weights and biases.

In the `forward` method:

- Line ❶ multiplies the input `x` by the weight matrix `W_1` and adds the bias vector `B_1`. The input has shape (`batch_size`, `seq_len`, `emb_dim`), so the result has shape (`batch_size`, `seq_len`, `emb_dim * 4`).

- Line ❷ applies the **ReLU** activation function element-wise, adding non-linearity.
- Line ❸ multiplies the result by the second weight matrix `W_2` and adds the bias vector `B_2`, reducing the dimensionality back to (`batch_size`, `seq_len`, `emb_dim`).

The first linear transformation expands to 4 times the embedding dimensionality (`emb_dim * 4`) to provide the network with greater capacity for learning complex patterns and relationships between variables. The 4x factor balances expressiveness and efficiency. After expanding the dimensionality, it's compressed back to the original embedding dimensionality (`emb_dim`). This ensures compatibility with residual connections, which require matching dimensionalities. Empirical results support this expand-and-compress approach as an effective trade-off between computational cost and performance.

With all components defined, we're ready to set up the complete decoder block:

```
class DecoderBlock(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()
        self.norm1 = RMSNorm(emb_dim)
        self.attn = MultiHeadAttention(emb_dim, num_heads)
        self.norm2 = RMSNorm(emb_dim)
        self.mlp = MLP(emb_dim)

    def forward(self, x, mask):
        attn_out = self.attn(self.norm1(x), mask) ❶
        x = x + attn_out ❷
        mlp_out = self.mlp(self.norm2(x)) ❸
        x = x + mlp_out ❹
        return x
```

The `DecoderBlock` class represents a single decoder block in a Transformer model. In the constructor, we set up the necessary layers: two `RMSNorm` layers, a `MultiHeadAttention` instance (configured with the embedding dimensionality and number of heads), and an `MLP` layer.

In the `forward` method:

- Line ❶ applies `RMSNorm` to the input `x`, which has shape (`batch_size`, `seq_len`, `emb_dim`). The output of `RMSNorm` keeps this shape. This normalized tensor is then passed to the multi-head attention layer, which outputs a tensor of the same shape.
- Line ❷ adds a **residual connection** by combining the attention output `attn_out` with the original input `x`. The shape doesn't change.
- Line ❸ applies the second `RMSNorm` to the result from the residual connection, retaining the same shape. This normalized tensor is then passed through the MLP, which outputs another tensor with shape (`batch_size`, `seq_len`, `emb_dim`).
- Line ❹ adds a second residual connection, combining `mlp_out` with its unnormalized input. The decoder block's final output shape is (`batch_size`, `seq_len`, `emb_dim`), ready for the next decoder block or the final output layer.

With the decoder block defined, we can now build the decoder transformer language model by stacking multiple decoder blocks sequentially:

```
class DecoderLanguageModel(nn.Module):
    def __init__(
        self, vocab_size, emb_dim,
        num_heads, num_blocks, pad_idx
    ):
        super().__init__()
        self.embedding = nn.Embedding(
            vocab_size, emb_dim,
            padding_idx=pad_idx
        ) ❶
        self.layers = nn.ModuleList([
            DecoderBlock(emb_dim, num_heads) for _ in range(num_blocks)
        ]) ❷
        self.output = nn.Parameter(torch.rand(emb_dim, vocab_size)) ❸

    def forward(self, x):
        x = self.embedding(x) ❹
        _, seq_len, _ = x.shape
        mask = torch.tril(torch.ones(seq_len, seq_len, device=x.device)) ❺
        for layer in self.layers: ❻
            x = layer(x, mask)
        return x @ self.output ❼
```

In the constructor of the `DecoderLanguageModel` class:

- Line ❶ creates an embedding layer that converts input token indices to dense vectors. The `padding_idx` specifies the ID of the padding token, ensuring that padding tokens are mapped to zero vectors.
- Line ❷ creates a `ModuleList` with `num_blocks` `DecoderBlock` instances, forming the stack of decoder layers.
- Line ❸ defines a matrix to project the last decoder block's output to logits over the vocabulary, enabling next token prediction.

In the `forward` method:

- Line ❹ converts the input token indices to embeddings. The input tensor `x` has shape (`batch_size`, `seq_len`); the output has shape (`batch_size`, `seq_len`, `emb_dim`).
- Line ❺ creates the **causal mask**.
- Line ❻ applies each decoder block to the input tensor `x` with shape (`batch_size`, `seq_len`, `emb_dim`), producing an output tensor of the same shape. Each block refines the sequence and passes it to the next until the final block.
- Line ❼ projects the output of the final decoder block to vocabulary-sized logits by multiplying it with the `self.output` matrix, which has shape (`emb_dim`, `vocab_size`). After this batched matrix multiplication, the final output has shape (`batch_size`, `seq_len`, `vocab_size`), providing scores for each token in the vocabulary at each position in the input sequence. This output can then be used to generate the model's predictions as we will discuss in the next chapter.

The training loop for `DecoderLanguageModel` is the same as for the RNN (Section 3.6), so it is not repeated here for brevity. Implementations of `RMSNorm` and `RoPE` are also skipped. Training data is prepared just like for the RNN: the target sequence is offset by one position relative to the input sequence, as described in Section 3.7. The complete code for training the decoder language model is available in the thelmbook.com/nb/4.1 notebook.

In the notebook, I used these hyperparameter values: `emb_dim = 128`, `num_heads = 8`, `num_blocks = 2`, `batch_size = 128`, `learning_rate = 0.001`, `num_epochs = 1`, and `context_size = 30`. With these settings, the model achieved a perplexity of 55.19, improving on the RNN's 72.23. This is a good result given the comparable number of trainable parameters (8,621,963 for the Transformer vs. 8,292,619 for the RNN). The real strengths of transformers, however, become apparent at larger scales of model size, context length, and training data. Reproducing experiments at such scales in this book is, of course, impractical.

Let's look at some continuations of the prompt "The President" generated by the decoder model at later training steps:

```
The President has been in the process of a new deal to make a decision on the
issue .

The President 's office said the government had `` no intention of making any
mistakes '' .

The President of the United States has been a key figure for the first time i
n the past ## years .
```

The "#" characters in the training data represent individual digits. For example, "##" likely represents the number of years.

***

If you've made it this far, well done! You now understand the mechanics of language models. But understanding the mechanics alone won't let you fully appreciate what modern language models are capable of. To truly understand, you need to work with one.

In the next chapter, we'll explore large language models (LLMs). We'll discuss why they're called *large* and what's so special about the size. Then, we'll cover how to finetune an existing LLM for practical tasks like question answering and document classification, as well as how to use LLMs to address a variety of real-world problems.