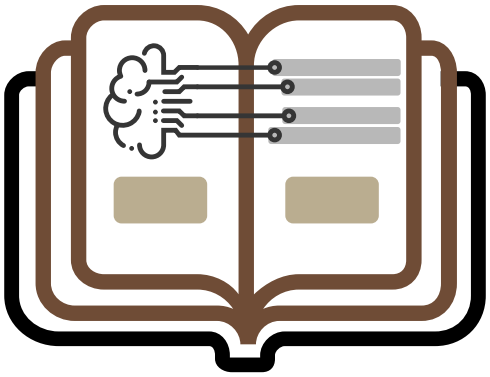
Natural Language Processing



Understanding Language Models

Han Cheol Moon
School of Computer Science and Engineering
Nanyang Technological University
Singapore
hancheol001@e.ntu.edu.sg
March 10, 2025

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Language Models

1.2 Pre-Training

This achievement is largely motivated by pre-training: we separate common components from many neural network-based systems, and then train them on huge amounts of unlabeled data using self-supervision. These pre-trained models serve as foundation models that can be easily adapted to different tasks via fine-tuning or prompting. As a result, the paradigm of NLP has been enormously changed. In many cases, large-scale supervised learning for specific tasks is no longer required, and instead, we only need to adapt pre-trained foundation models.

The discussion of pre-training issues in NLP typically involves two types of problems: sequence modeling (or sequence encoding) and sequence generation.

Optimizing θ on a pre-training task. Unlike standard learning problems in NLP, pre-training does not assume specific downstream tasks to which the model will be applied. Instead, the goal is to train a model that can generalize across various tasks.

Applying the pre-trained model to downstream tasks. To adapt the model to these tasks, we need to adjust the parameters slightly using labeled data or prompt the model with task descriptions.

Part II

Transformers

Chapter 2

Transformer

TLDR

- *Attention is a communication mechanism*, which can be seen as nodes in a directed graph looking at each other and aggregating information with a weighted sum from all nodes that point to them, with data-dependent weights.
- There is no notion of space. Attention simply acts over a set of vectors. This is why we need to positionally encode tokens.
- Each example across batch dimension is of course processed completely independently and never "talk" to each other
- In an "encoder" attention block just delete the single line that does masking with 'tril', allowing all tokens to communicate. This block here is called a "decoder" attention block because it has triangular masking, and is usually used in autoregressive settings, like language modeling.
- "self-attention" just means that the keys and values are produced from the same source as queries. In "cross-attention", the queries still get produced from x, but the keys and values come from some other, external source (*e.g.,* an encoder module)
- "Scaled" attention additional divides 'wei' by $1/sqrt(head\_size)$. This makes it so when input Q,K are unit variance, wei will be unit variance too and Softmax will stay diffuse and not saturate too much. Illustration below

2.1 Attention Mechanism

The attention mechanism assigns *attention scores* (*i.e., weights*) to different parts of the input sequence, indicating how much each part contributes to the current output. In essence, the model "pays attention" to certain parts of the input more than others while processing each token in the sequence.

Assume the encoder produces 3 hidden states (each a 2-dimensional vector):

$$h_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad h_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad h_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

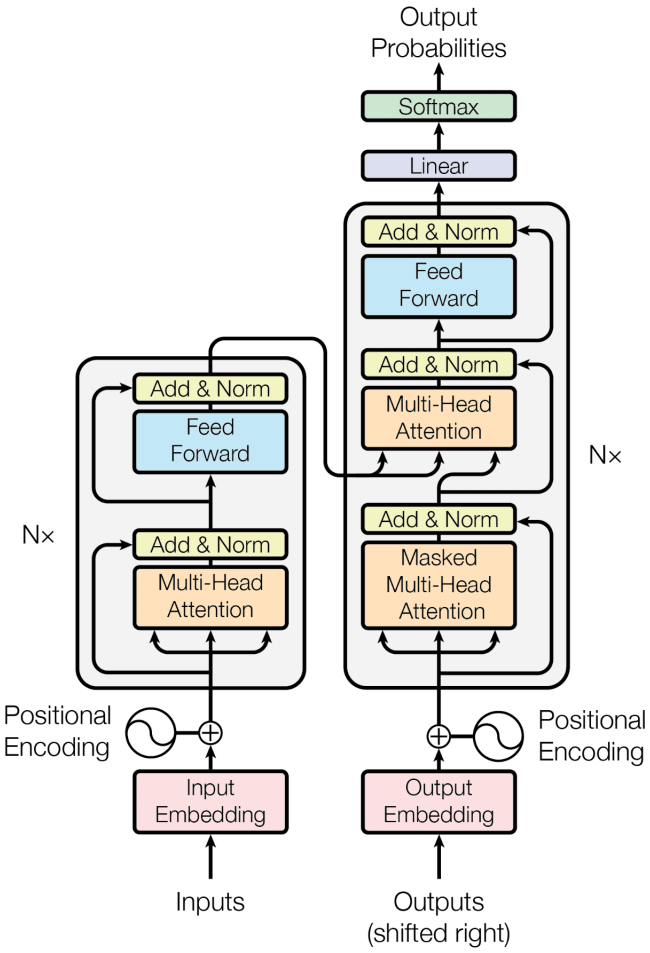


Figure 2.1: An illustration of Transformer architecture.

Let the decoder's previous hidden state at time $t-1$ be:

$$s_{t-1} = \begin{bmatrix} 0.5 \\ 0.2 \end{bmatrix}.$$

Using the *dot product* as our score function, the alignment score for each encoder hidden state is:

$$e_{tj} = s_{t-1}^{\top} h_j.$$

Compute each:

1. For h_1:
$$e_{t1} = \begin{bmatrix} 0.5 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0.5.$$
2. For h_2:
$$e_{t2} = \begin{bmatrix} 0.5 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0.2.$$
3. For h_3:
$$e_{t3} = \begin{bmatrix} 0.5 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 0.7.$$

The attention weight for each encoder time step is given by:

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{3} \exp(e_{tk})}.$$

Calculate the exponentials:

- $\exp(0.5) \approx 1.6487$,
- $\exp(0.2) \approx 1.2214$,
- $\exp(0.7) \approx 2.0138$.

Sum of exponentials:

$$S = 1.6487 + 1.2214 + 2.0138 \approx 4.8839.$$

Now compute each attention weight:

1. For α_{t1}:
$$\alpha_{t1} = \frac{1.6487}{4.8839} \approx 0.3374.$$
2. For α_{t2}:
$$\alpha_{t2} = \frac{1.2214}{4.8839} \approx 0.2501.$$
3. For α_{t3}:
$$\alpha_{t3} = \frac{2.0138}{4.8839} \approx 0.4125.$$

These weights sum to 1 (up to rounding):

$$0.3374 + 0.2501 + 0.4125 \approx 1.0000.$$

The context vector c_t is the weighted sum of the encoder hidden states:

$$c_t = \alpha_{t1} h_1 + \alpha_{t2} h_2 + \alpha_{t3} h_3.$$

Substitute in the values:

$$\begin{aligned} c_t &= 0.3374 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0.2501 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 0.4125 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.3374 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.2501 \end{bmatrix} + \begin{bmatrix} 0.4125 \\ 0.4125 \end{bmatrix} \\ &= \begin{bmatrix} 0.3374 + 0.4125 \\ 0 + 0.2501 + 0.4125 \end{bmatrix} \\ &= \begin{bmatrix} 0.7499 \\ 0.6626 \end{bmatrix}. \end{aligned}$$

So, the context vector is approximately:

$$c_t \approx \begin{bmatrix} 0.75 \\ 0.66 \end{bmatrix}.$$

In a typical decoder, the context vector c_t is combined with the previous hidden state and possibly the previously generated output to update the current hidden state. For example, an update could be:

$$s_t = f(s_{t-1}, y_{t-1}, c_t),$$

or, if you are using a simple formulation with a combined input, it might be:

$$s_t = \tanh(W[s_{t-1}; c_t]),$$

where $[s_{t-1}; c_t]$ denotes the concatenation of s_{t-1} and c_t, and W is a learnable weight matrix. The updated state s_t would then be used to predict the next output token.

Let's use a machine translation task (English to French) as an example. Suppose we are translating the sentence "I am learning" into French.

- Input: Sequence of words in English: '[I, am, learning]'
- Output: Sequence of words in French: '[Je, suis, en, train, d'apprendre]'

Instead of compressing all the input information into a fixed-size context vector (like in traditional encoder-decoder models), the attention mechanism allows the decoder to look at different parts of the input sentence at each step of the decoding process.

	I	am	learning
Je	0.7	0.2	0.1
suis	0.1	0.8	0.1
en	0.05	0.15	0.8

This matrix shows that "Je" strongly attends to "I", "suis" attends mostly to "am", and "en" attends primarily to "learning".

2.1.1 Self-Attention

- n: the number of tokens in the sequence.
- d_{model}: the dimension of the input embeddings.
- d_k: the dimension of the query and key vectors.
- d_v: the dimension of the value vectors (often $d_k = d_v$, but they need not be equal).

Assume we have an input sequence of n tokens. For each token i (with $1 \leq i \leq n$) we start with an embedding vector $\mathbf{x}_i \in \mathbb{R}^{d_{\text{model}}}$. In self-attention, we first linearly project these embeddings into three different spaces to obtain the *query*, *key*, and *value* vectors:

$$\begin{aligned}\mathbf{q}_i &= \mathbf{x}_i\, W^Q,\\ \mathbf{k}_i &= \mathbf{x}_i\, W^K,\\ \mathbf{v}_i &= \mathbf{x}_i\, W^V,\end{aligned}$$

where

- $W^Q, W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ are the query and key projection matrices,
- $W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ is the value projection matrix,
- d_k (and sometimes d_v) is a chosen dimensionality for these spaces.

For a given token i, we compute its output representation as a weighted sum of the value vectors of all tokens. The weights are determined by the similarity between the query $\mathbf{q}_i$ and the keys $\mathbf{k}_j$ of all tokens j in the sequence.

First, compute the *dot-product scores* between the query for token i and every key:

$$s_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j, \quad \text{for } j = 1, 2, \ldots, n.$$

To prevent the dot products from growing too large in magnitude (especially when d_k is large), we *scale the scores* by $\sqrt{d_k}$:

$$\tilde{s}_{ij} = \frac{s_{ij}}{\sqrt{d_k}}.$$

Next, we apply the softmax function over the scaled scores for token i to obtain the attention weights α_{ij}:

$$\alpha_{ij} = \frac{\exp(\tilde{s}_{ij})}{\sum_{l=1}^{n} \exp(\tilde{s}_{il})}.$$

These weights satisfy $\sum_{j=1}^{n} \alpha_{ij} = 1$.

Finally, the output for token i, denoted by $\mathbf{z}_i$, is the weighted sum of the value vectors:

$$\mathbf{z}_i = \sum_{j=1}^{n} \alpha_{ij}\, \mathbf{v}_j.$$

In matrix form for all tokens, if we define matrices Q, K, and V whose rows are the vectors $\mathbf{q}_i$, $\mathbf{k}_i$, and $\mathbf{v}_i$ respectively, the self-attention operation is:

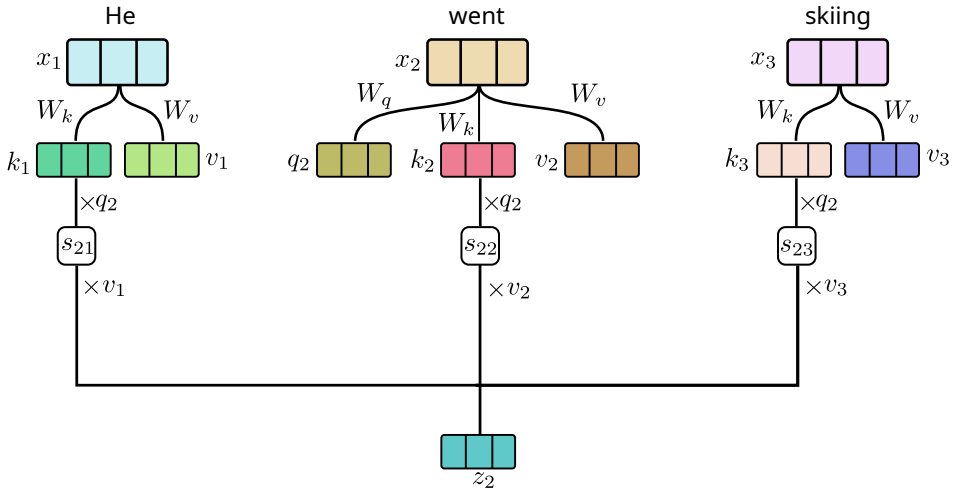


Figure 2.2: An Illustration of the self-attention.

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

Here, $Q, K, V \in \mathbb{R}^{n \times d_{model}}$, QK^T's time complexity is $O(N^2 d)$. This quadratic cost is massive for long input-sequences such as documents to be summarized or character-level inputs.

Input Embeddings Each token i is represented by an embedding:

$$\mathbf{x}_i \in \mathbb{R}^{d_{\text{model}}}.$$

You can think of all the tokens put together as a matrix:

$$X \in \mathbb{R}^{n \times d_{\text{model}}}.$$

Projection Matrices To obtain the queries, keys, and values, we use learned projection matrices:

$$W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k},$$
$$W^K \in \mathbb{R}^{d_{\text{model}} \times d_k},$$
$$W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}.$$

Projected Matrices Multiplying the input X by these weight matrices gives:

$$Q = X\,W^Q \in \mathbb{R}^{n \times d_k},$$
$$K = X\,W^K \in \mathbb{R}^{n \times d_k},$$
$$V = X\,W^V \in \mathbb{R}^{n \times d_v}.$$

Here, each row of Q (or K, or V) corresponds to the query (or key, or value) of one token.

Dot-Product Attention Scores The attention scores between tokens are computed using:

$$QK^\top \in \mathbb{R}^{n\times n}.$$

In this product:

- Q is $n \times d_k$
- $K^\top$ is $d_k \times n$

Therefore, the result is an $n \times n$ matrix where each entry (i, j) represents the (unnormalized) similarity between token i and token j.

Scaled Dot-Product and Softmax Before applying the softmax, the scores are scaled by $\sqrt{d_k}$:

$$\frac{QK^\top}{\sqrt{d_k}} \in \mathbb{R}^{n\times n}.$$

Then, applying the softmax function row-wise produces an attention weight matrix $A \in \mathbb{R}^{n\times n}$:

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right).$$

Final Output Finally, the output of the self-attention layer is computed as:

$$\text{Attention}(Q, K, V) = A\,V.$$

Since:

- A is $n \times n$,
- V is $n \times d_v$,

The final output is:

$$\text{Attention}(Q, K, V) \in \mathbb{R}^{n\times d_v}.$$

2.1.2 Masked Attention

In autoregressive tasks (*e.g.,* language modeling), it is essential that when predicting a token at position ii, the model does not "peek" at any tokens at positions $j > i$. *Masked attention* ensures that each token only attends to tokens at the same or earlier positions. The masked attention is often referred to *cross-attention*. This is just a self-attention in decoder.

$$\text{MA}(Q, K, V) = softmax\left(\frac{Q^T K + M}{\sqrt{d_k}}\right)V,$$

where M

$$M_{ij} = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{if } j > i \end{cases}$$

Note that $-\infty$ will make exp term to be zero.

2.1.3 Multi-Head Attention

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. **Rather than computing a single attention function with full-dimensional queries, keys, and values, the mechanism splits them into multiple "heads" and computes attention in parallel**.

Suppose:

- The input embeddings (or previous layer outputs) form the matrix $X \in \mathbb{R}^{n \times d_{\text{model}}}$,
- d_{model} is the model (or embedding) dimension,
- We decide to use h attention heads.

Each head will work with lower-dimensional projections of the input. In particular, we typically set:

$$d'_k = d'_v = \frac{d_{\text{model}}}{h},$$

so that each head processes queries, keys, and values of dimensions d'_k and d'_v, and the total computation remains efficient.

Linear Projections for Each Head For each head $i \in \{1, \ldots, h\}$, we define learned projection matrices:

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d'_k},$$
$$W_i^K \in \mathbb{R}^{d_{\text{model}} \times d'_k},$$
$$W_i^V \in \mathbb{R}^{d_{\text{model}} \times d'_v}.$$

We then project the input X to obtain:

$$Q_i = XW_i^Q \in \mathbb{R}^{n \times d'_k},$$
$$K_i = XW_i^K \in \mathbb{R}^{n \times d'_k},$$
$$V_i = XW_i^V \in \mathbb{R}^{n \times d'_v}.$$

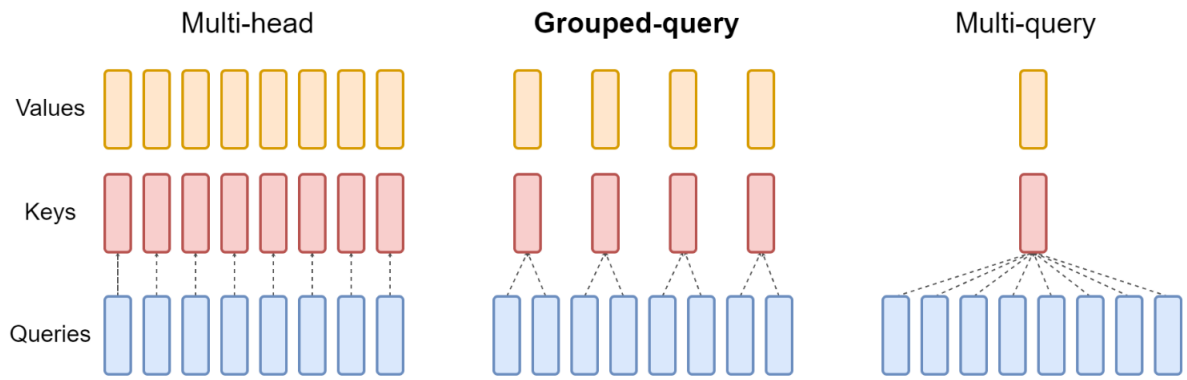
Compute Scaled (Masked) Dot-Product Attention for Each Head For each head i, compute:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i),$$

where the attention function is defined as:

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^\top + M}{\sqrt{d'_k}}\right) V_i.$$

- If masking is not required (*e.g.,* in the encoder or in *non-autoregressive* settings), simply set $M = 0$.
- *For decoder self-attention in autoregressive models*, M is defined as in the Masked Attention section above.



Concatenate the Heads and Project Once all heads are computed, we concatenate their outputs:

$$\text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h) \in \mathbb{R}^{n \times (h \cdot d'_v)}.$$

Finally, we apply a learned linear projection:

$$W^O \in \mathbb{R}^{(h \cdot d'_v) \times d_{\text{model}}},$$

to obtain the final multi-head attention output:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O \in \mathbb{R}^{n \times d_{\text{model}}}.$$

2.2 Various Attention Mechanisms

If we use a single value and a single key over all queries, we can significantly reduce the memory usage for KV caching. This is the basic idea of the multi-query attention (MQA) and the grouped query attention (GQA). However, MQA tends to lower the output quality as the model size increases. GQA achieved a balance between them.

2.3 Positional Embedding

The self-attention mechanism in transformers treats all tokens in a sequence in parallel without an inherent notion of order. This means that, by itself, self-attention is invariant to the order of input tokens. Positional encoding is introduced to inject order information so that the model can differentiate between tokens based on their positions.

- **Absolute Positional Embeddings**: Each position in the sequence is assigned a unique vector. Although straightforward, these embeddings don't scale well to longer sequences and fail to capture the nuances of relative positions between tokens.
- **Relative Positional Embeddings**: These embeddings focus on the distance between tokens, which can improve the model's understanding of token relationships. However, they typically introduce additional complexity into the model architecture.

2.3.1 Permutation Invariance of Self-Attention

Without positional embeddings, the transformer's self-attention would treat the input as a bag of tokens, ignoring the order entirely. The added positional embeddings break this permutation invariance by encoding the position directly into the token representation. As a result, even if the same tokens are present, the model can infer their relative order and roles in the sentence.

Imagine you have a short sentence, "I feel good". If you simply pass this sentence into a transformer, the model wouldn't know the order of the words. The same set of token (*i.e.,* word) embeddings could represent the sentence "good feel I" if no ordering information were provided.

We formulate this as follows: With a permutation matrix P of shape (n, n), the input tokens can be permuted as

$$X' = PX.$$

Let's follow the same self-attention process for X':

$$Q' = X'W_q = PXW_q = PQ,$$

$$K' = X'W_k = PXW_k = PK,$$

$$V' = X'W_v = PXW_v = PV.$$

Then, compute the scores using Q' and K':

$$S' = \frac{Q'(K')^T}{\sqrt{d_k}} = \frac{(PQ)(PK)^T}{\sqrt{d_k}}.$$

Note that

$$(PK)^T = K^T P^T,$$

so

$$S' = \frac{PQK^T P^T}{\sqrt{d_k}} = P\,S\,P^T.$$

The softmax is applied row-wise.

$$A' = \text{Softmax}(S').$$

Since P and P^T are just reordering rows and columns, respectively, the attention weights A are simply permuted like below:

$$A' = P\,A\,P^T.$$

Finally, the output for the permuted input is:

$$\text{Attention}(X') = A'V' = (PAP^T)(PV) = PAV = P\,\text{Attention}(X).$$

As you can see the self-attention mechanism is *equivariant* to permutations. This means that if you permute the input tokens, the output is permuted in the same way. There is no mechanism in the equations above that distinguishes one ordering from another; the operations treat all tokens symmetrically.

Thus, without additional positional encodings, if you were to shuffle the tokens, the model would compute the same set of pairwise interactions—just in a different order. The structure of the equations does not provide any mechanism for the model to know that one token came before or after another.

2.3.2 Sinusoidal Positional Encoding

In a Transformer model, positional encoding vectors are added to the token (word) embeddings before the input is fed into the self-attention layers. This addition gives the model a sense of the order in the sequence, enabling it to capture the sequential relationships between tokens despite processing them in parallel.

- $\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$
- $\mathrm{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$,

where:

- pos: the position of the token in the sequence (starting at 0),
- i: the index along the embedding dimension,
- d_{model}: the total dimension of the model's embeddings.

Let's work through a concrete example with a very small embedding dimension:

- Assume $d_{\text{model}} = 4$.
- We'll compute the positional encoding for two positions: $pos = 0$ and $pos = 1$.

Since $d_{\text{model}} = 4$, we have 4 dimensions indexed $0, 1, 2, 3$. The formulas split the dimensions into even and odd indices:

For $pos = 0$

- Dimension 0 (even index, $i = 0$):
$$\mathrm{PE}(0,0) = \sin\Big(\frac{0}{10000^{\frac{2\cdot 0}{4}}}\Big) = \sin\Big(\frac{0}{10000^{0}}\Big) = \sin(0) = 0.$$
- Dimension 1 (odd index, $i = 0$):
$$\mathrm{PE}(0,1) = \cos\Big(\frac{0}{10000^{\frac{2\cdot 0}{4}}}\Big) = \cos(0) = 1.$$
- Dimension 2 (even index, $i = 1$):
$$\mathrm{PE}(0,2) = \sin\Big(\frac{0}{10000^{\frac{2\cdot 1}{4}}}\Big) = \sin\Big(\frac{0}{10000^{0.5}}\Big) = \sin(0) = 0.$$
- Dimension 3 (odd index, $i = 1$):
$$\mathrm{PE}(0,3) = \cos\Big(\frac{0}{10000^{\frac{2\cdot 1}{4}}}\Big) = \cos(0) = 1.$$
- So, the positional encoding for $pos = 0$ is:
$$[0,\ 1,\ 0,\ 1].$$

For $pos = 1$

- Dimension 0 (even index, $i = 0$):

$$\text{PE}(1,0) = \sin\Big(\frac{1}{10000^{\frac{2\cdot 0}{4}}}\Big) = \sin\Big(\frac{1}{10000^{0}}\Big) = \sin(1) \approx 0.84147.$$

- Dimension 1 (odd index, $i = 0$):

$$\text{PE}(1,1) = \cos\Big(\frac{1}{10000^{\frac{2\cdot 0}{4}}}\Big) = \cos(1) \approx 0.54030.$$

- Dimension 2 (even index, $i = 1$):

$$\text{PE}(1,2) = \sin\Big(\frac{1}{10000^{\frac{2\cdot 1}{4}}}\Big) = \sin\Big(\frac{1}{10000^{0.5}}\Big) = \sin\Big(\frac{1}{100}\Big) = \sin(0.01) \approx 0.00999983.$$

- Dimension 3 (odd index, $i = 1$):

$$\text{PE}(1,3) = \cos\Big(\frac{1}{10000^{\frac{2\cdot 1}{4}}}\Big) = \cos(0.01) \approx 0.99995.$$

- Thus, the positional encoding for $pos = 1$ is approximately:

$$[0.84147,\ 0.54030,\ 0.00999983,\ 0.99995].$$

2.3.3 RoPE

Rather than adding a positional vector to the token embeddings, **RoPE rotates the embeddings by a position-specific angle.** Think of it as "twisting" the embedding in space based on its position. For instance, if you have a simple two-dimensional embedding for the word "dog", you can imagine its vector being rotated by an angle θ if it's the first word, 2θ if it's the second word, and so on.

For high-dimensional embeddings (*e.g.,* in $\mathbb{R}^d$), RoPE divides the vector into $d/2$ pairs (or 2D subspaces). For a token at position i, denote its embedding by:

$$\mathbf{x}_i \in \mathbb{R}^d.$$

We partition $\mathbf{x}_i$ into pairs:

$$\mathbf{x}_i^{(k)} = \begin{pmatrix} x_{i,2k} \\ x_{i,2k+1} \end{pmatrix}, \quad k = 0, 1, \ldots, \frac{d}{2} - 1.$$

Subsequently, for each 2D subspace indexed by k, RoPE defines a rotation angle:

$$\theta_{i,k} = i \cdot \alpha_k,$$

where α_k is a scaling factor that typically depends on the dimension k. A popular choice is:

$$\alpha_k = \frac{1}{10000^{\frac{2k}{d}}}.$$

This scaling mimics the frequency scaling in sinusoidal embeddings (*i.e.,* positional embedding), ensuring that different subspaces capture positional information at different granularities.

In two-dimensional geometry, any rotation by an angle θ can be represented by a *rotation matrix* $R(\theta)$. This matrix is a standard tool in linear algebra and has the form:

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}.$$

- Geometric Interpretation: The rotation matrix $R(\theta)$ rotates any 2D vector by the angle θ (in the counterclockwise direction) while preserving its magnitude. This property makes it ideal for encoding positional shifts—rotating a vector does not change its "content" (its norm) but changes its direction, thereby encoding positional information.
- Inherent Relative Encoding: When different positions correspond to different rotation angles, the relationship between any two positions can be captured by the difference in their rotation angles. This leads to a natural encoding of relative position without having to explicitly compute or store separate relative position vectors.

However, Transformer embeddings are typically high-dimensional. Thus, RoPE adopts a simple trick. They split the high dimensional vector into two halves so that each pair forms a 2D subspace. Thus, we can apply the rotation matrix.

For each 2D subspace, the rotary transformation is applied as follows. Given the subvector:

$$\mathbf{x}_i^{(k)} = \begin{pmatrix} x_{i,2k} \\ x_{i,2k+1} \end{pmatrix},$$

we compute its rotated version:

$$\mathrm{RoPE}_i\big(\mathbf{x}_i^{(k)}\big) = R(\theta_{i,k})\,\mathbf{x}_i^{(k)} = \begin{pmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{pmatrix} \begin{pmatrix} x_{i,2k} \\ x_{i,2k+1} \end{pmatrix}.$$

This yields the updated coordinates:

$$\tilde{x}_{i,2k} = x_{i,2k}\cos(\theta_{i,k}) - x_{i,2k+1}\sin(\theta_{i,k}),$$

$$\tilde{x}_{i,2k+1} = x_{i,2k+1}\cos(\theta_{i,k}) + x_{i,2k}\sin(\theta_{i,k}).$$

After processing all $d/2$ subspaces, we concatenate the results back into a full d-dimensional vector $\tilde{\mathbf{x}}_i$.

In transformer architectures, RoPE is applied to both the query and the key vectors except the value vecotrs:

$$\tilde{\mathbf{q}}_i = \mathrm{RoPE}_i(\mathbf{q}_i),$$
$$\tilde{\mathbf{k}}_j = \mathrm{RoPE}_j(\mathbf{k}_j).$$

Then, the attention score between positions i and j is calculated as:

$$\mathrm{Attention}(i,j) = \frac{\tilde{\mathbf{q}}_i \cdot \tilde{\mathbf{k}}_j}{\sqrt{d}}.$$

A key property of RoPE is that the dot product between the rotated vectors can be reinterpreted to show how relative positions are encoded. In particular, one can derive that:

$$\tilde{\mathbf{q}}_i^\top \tilde{\mathbf{k}}_j = \mathbf{q}_i^\top\, \mathbf{M}(i,j)\, \mathbf{k}_j,$$

where $\mathbf{M}(i,j)$ is a block diagonal matrix that encapsulates the effect of the relative positional difference $j - i$.

Focus on one 2D subspace (indexed by k):

- The query subvector at position i is rotated by $\theta_{i,k} = i\alpha_k$.
- The key subvector at position j is rotated by $\theta_{j,k} = j\alpha_k$.

The dot product in this subspace is:

$$\tilde{\mathbf{q}}_i^{(k)\top} \tilde{\mathbf{k}}_j^{(k)} = \left(\mathbf{q}_i^{(k)}\right)^\top R(\theta_{i,k})^\top R(\theta_{j,k})\, \mathbf{k}_j^{(k)}.$$

Since the transpose of a rotation matrix is its inverse (i.e., $R(\theta)^\top = R(-\theta)$), we have:

$$R(\theta_{i,k})^\top R(\theta_{j,k}) = R(-\theta_{i,k}) R(\theta_{j,k}) = R\big(\theta_{j,k} - \theta_{i,k}\big).$$

Because $\theta_{j,k} - \theta_{i,k} = (j-i)\alpha_k$, the transformation becomes:

$$R\big((j-i)\alpha_k\big).$$

Repeating this for each 2D subspace results in a block diagonal matrix:

$$\mathbf{M}(i,j) = \operatorname{diag}\Big(R\big((j-i)\alpha_0\big),\, R\big((j-i)\alpha_1\big),\, \ldots,\, R\big((j-i)\alpha_{\frac{d}{2}-1}\big)\Big).$$

Each block is the 2×2 rotation matrix:

$$R\big((j-i)\alpha_k\big) = \begin{pmatrix} \cos\big((j-i)\alpha_k\big) & -\sin\big((j-i)\alpha_k\big) \\ \sin\big((j-i)\alpha_k\big) & \cos\big((j-i)\alpha_k\big) \end{pmatrix}.$$

- Relative Positional Bias: The matrix $\mathbf{M}(i,j)$ adjusts the dot product between queries and keys based on their relative positions $(j-i)$. Thus, the value vectors are not modified. Instead of explicitly adding a relative position embedding, the rotation inherently modulates the interaction between tokens.
- Unified Encoding: Since $\mathbf{M}(i,j)$ is built from standard rotation matrices $R(\theta)$, it seamlessly encodes the relative positional difference across all 2D subspaces. This results in a unified treatment where both absolute and relative positional cues are embedded into the attention calculation.
- Elegant Mathematical Foundation: The use of $R(\theta)$ comes directly from classical geometry and linear algebra. It leverages the well-known properties of rotations in 2D—specifically, that rotations preserve vector norms and that the composition of rotations is itself a rotation (with the angle being the sum or difference of the individual angles). This mathematical elegance translates into an efficient and effective mechanism for positional encoding.

2.4 Encoder-Decoder

2.4.1 Encoder

2.4.2 Decoder

The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

```
def forward(self, tgt: Tensor, memory: Tensor) -> Tensor:
  seq_len, dimension = tgt.size(1), tgt.size(2)
  tgt += position_encoding(seq_len, dimension)
  for layer in self.layers:
    tgt = layer(tgt, memory)
  return torch.softmax(self.linear(tgt), dim=-1)
```

2.5 Inference of Autoregressive Model

In an autoregressive transformer (such as GPT), tokens are generated one at a time. During inference, the model must compute attention for the next token based on all previously generated tokens. However, because recomputing the entire attention from scratch at each time step would be inefficient, these models use *caching* to store intermediate results (the keys and values) from previous time steps. Below is an explanation with equations and clear notations.

Inference Overview in Autoregressive Models

- Training: The model can process the entire sequence in parallel using masked self-attention. A mask (typically a triangular matrix) prevents tokens from "seeing" future tokens.
- Inference: The model generates one token at a time. At time step $t+1$, it uses the already generated tokens $[x_1, x_2, \ldots, x_t]$ to compute the probability distribution for the next token.

To avoid recomputing keys and values for tokens $x_1, \ldots, x_t$ at every step, the model stores them (usually for each layer). When a new token is generated, only its query needs to be computed, and then the cached keys and values are used to compute the attention.

The technique of storing and reusing the computed keys and values from previous tokens during autoregressive generation is commonly called *KV caching* (short for Key-Value Caching).

Step 1: Previous Tokens and Cached Representations Assume that by time step t the model has generated tokens:

$$x_1,\, x_2,\, \ldots,\, x_t.$$

For a given transformer layer, let the cached key and value matrices be:

$$K_{\leq t} \in \mathbb{R}^{t \times d_k},$$
$$V_{\leq t} \in \mathbb{R}^{t \times d_v},$$

where d_k is the key (and query) dimension and d_v is the value dimension.

Step 2: Compute the Query for the New Token When generating the next token x_{t+1}, its input (often the embedding of the previously generated token or a special "start" symbol) is used to compute a query vector for each layer:

$$q_{t+1} \in \mathbb{R}^{d_k}.$$

This is computed by a linear projection:

$$q_{t+1} = x_{t+1} W^Q,$$

where $W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ is the learned projection matrix.

Step 3: Compute the Attention Scores The new query q_{t+1} is compared with all the cached keys. In a single attention head, the (scaled) dot-product attention scores are computed as:

$$s_{t+1,j} = \frac{q_{t+1} \cdot k_j}{\sqrt{d_k}} \quad \text{for } j = 1, 2, \ldots, t,$$

and often, for implementation convenience, the new token's own key k_{t+1} is also computed and appended to the cache. In that case, you would have:

$$s_{t+1,j} = \frac{q_{t+1} \cdot k_j}{\sqrt{d_k}} \quad \text{for } j = 1, 2, \ldots, t+1.$$

Since the model is autoregressive, the mask is implicit. There are no "future" tokens beyond $t+1$ at inference time. (If you do compute for all $t+1$ positions, a mask would ensure that token $t+1$ only attends to tokens 1 through $t+1$.)

Step 4: Apply the Softmax to Get Attention Weights The scores are then normalized with the softmax function to obtain the attention weights:

$$\alpha_{t+1,j} = \frac{\exp(s_{t+1,j})}{\sum_{j'=1}^{t+1} \exp(s_{t+1,j'})}, \quad j = 1, \ldots, t+1.$$

These weights determine how much the new token attends to each of the previous tokens (and its own representation, if included).

Step 5: Compute the Weighted Sum of Values The output of the attention layer for the new token is then computed as:

$$z_{t+1} = \sum_{j=1}^{t+1} \alpha_{t+1,j} \, v_j,$$

where each v_j is the value vector from the cache (or computed for the new token in the case of $j = t+1$):

$$v_j = x_j W^V, \quad j = 1, \ldots, t+1.$$

This z_{t+1} is then passed on through the rest of the transformer layer (including feed-forward sub-layers, layer normalization, etc.) to eventually produce logits over the vocabulary.

Step 6: Generate the Next Token and Update the Cache

- The model uses the final output (after all transformer layers) to compute a probability distribution over the vocabulary.
- A token is chosen (*e.g.,* via sampling or greedy decoding) and appended to the sequence.
- The new token's key and value vectors (from each layer) are computed and added to the cache so that future tokens can attend to it.

2.5.1 Inference without Caching

Let's take a look at the following example:

- Number of tokens so far: $t = 3$. We have already generated tokens $\{x_1, x_2, x_3\}$. We now want to generate token x_4.
- Model dimensionality: To keep it simple, let's say each token embedding is 2-dimensional ($d_{\text{model}} = 2$) and the attention uses a single head with key/query dimension $d_k = 2$ and value dimension $d_v = 2$.
- Token embeddings (just made-up numbers):

$$x_1 = \begin{bmatrix}1\\2\end{bmatrix}, \quad x_2 = \begin{bmatrix}3\\4\end{bmatrix}, \quad x_3 = \begin{bmatrix}5\\6\end{bmatrix}, \quad x_4 = \begin{bmatrix}?\\?\end{bmatrix}$$

  The forth one is the one we want to generate. We will compute attention for tokens x_1, x_2, x_3, x_4 all at once.
- Projection matrices (W^Q, W^K, W^V) are each 2×2 for this example. For instance, we have the following matrices:

$$W^Q = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}, \quad W^K = \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}, \quad W^V = \begin{bmatrix}0.5 & -0.5\\1.0 & 0.5\end{bmatrix}.$$

Step A: Compute Queries, Keys, and Values

- Queries:

$$q_i = x_i\, W^Q$$

  For $i = 1, 2, 3, 4$:
  - $q_1 = [1\ \ 2] \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} = \begin{bmatrix}1 & 2\end{bmatrix}$
  - $q_2 = [3\ \ 4] \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} = \begin{bmatrix}3 & 4\end{bmatrix}$
  - $q_3 = [5\ \ 6] \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} = \begin{bmatrix}5 & 6\end{bmatrix}$
  - $q_4 = [?,\ ?] \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} = \begin{bmatrix}? & ?\end{bmatrix}$
- Keys:

$$k_i = x_i\, W^K$$

  For $i = 1, 2, 3, 4$:
  - $k_1 = [1\ \ 2] \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix} = \begin{bmatrix}1 & 4\end{bmatrix}$
  - $k_2 = [3\ \ 4] \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix} = \begin{bmatrix}3 & 10\end{bmatrix}$
  - $k_3 = [5\ \ 6] \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix} = \begin{bmatrix}5 & 16\end{bmatrix}$

  - $k_4 = [?,\ ?] \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} ? & ? \end{bmatrix}$

- Values:

$$v_i = x_i\, W^V$$

  For $i = 1, 2, 3, 4$:

  - $v_1 = [1\ \ 2] \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 2.5 & 0.5 \end{bmatrix}$
  - $v_2 = [3\ \ 4] \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 5.5 & 0.5 \end{bmatrix}$
  - $v_3 = [5\ \ 6] \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 8.5 & 0.5 \end{bmatrix}$
  - $v_4 = [?,\ ?] \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 0.5 \end{bmatrix} = \begin{bmatrix} ? & ? \end{bmatrix}$

All of the above must be computed at the current time step if we do not use caching.

With KV caching, you would *not* recalculate k_1, k_2, k_3 and v_1, v_2, v_3. Instead:

- You already have $\{k_1, k_2, k_3\}$ and $\{v_1, v_2, v_3\}$ stored from the previous steps.
- You only compute:
  - q_4 (the query for the new token),
  - k_4, v_4 (the new key and value to add to the cache).

In other words, you skip re-projecting and re-computing every key and value from tokens $\{1, 2, 3\}$. This saves a substantial amount of computation when generating long sequences, especially in large transformers (like GPT).

Note that storing data in the cache uses up memory space. Systems with limited memory resources may struggle to accommodate this additional memory overhead, potentially resulting in out-of-memory errors. This is especially the case when long inputs need to be processed, as the memory required for the cache grows linearly with the input length and the batch size.

Computational Cost In sum, the vanilla self-attention on n tokens,

- each token needs to look at all n tokens to get attention scores
- Thus, the cost is $O(n^2)$

For instance, if we given a sentence with three tokens,

1. First token: Look at 1 token (cost: $O(1^2)$)
2. Second token: Look at 2 tokens (cost: $O(2^2)$)
3. Third token: Look at 3 tokens (cost: $O(3^2)$)

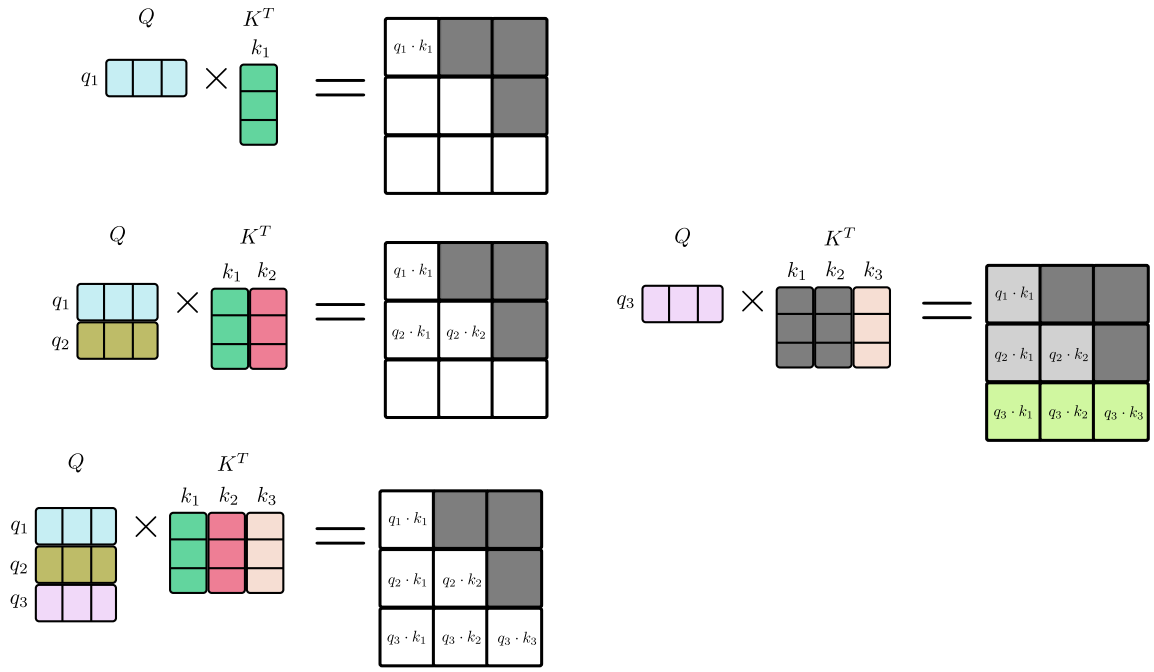


Figure 2.3: An example of KV caching

If we add up all these costs for generating a sequence of length n, we get:

$$O(1^2 + 2^2 + 3^2 + \cdots + n^2) \approx O(n^3)$$

With caching,

1. Process 1 token cost $O(1)$
2. Process 1 new token + look at 1 cached token cost $O(2)$
3. Process 1 new token + look at 2 cached tokens cost $O(3)$

Adding these up:

$$O(1 + 2 + 3 + \cdots + n) = O(n^2)$$

2.6 Example

Part III

Advanced

2.7 LoRA

LoRA (Low-Rank Adaptation) is a popular method under the umbrella of Parameter-Efficient Fine-Tuning (PEFT). Instead of updating all the parameters of a large pre-trained model during fine-tuning, LoRA only learns a small number of additional parameters that capture task-specific adjustments. This approach makes the training more memory- and compute-efficient while often achieving comparable performance to full fine-tuning.

2.7.1 The Core Idea of LoRA

In standard fine-tuning, one might update a large weight matrix W in a neural network layer to adapt the model for a new task. LoRA takes a different approach:

- Freeze the Original Weights: The original weight matrix W remains fixed.
- Learn a Low-Rank Update: Instead of modifying W directly, LoRA introduces a learnable update ΔW that is constrained to be low-rank. The changes made to W during fine-tuning are collectively represented by ΔW, such that the updated weights can be expressed as $W + \Delta W$. This update is typically expressed as the product of two smaller matrices:

$$\Delta W = AB$$

where:

- $A \in \mathbb{R}^{d \times r}$
- $B \in \mathbb{R}^{r \times k}$
- r (the rank) is chosen such that $r \ll \min(d, k)$

Thus, during fine-tuning, the effective weight becomes:

$$W' = W + \Delta W = W + AB$$

2.7.2 Low-Rank Decomposition

Suppose W is of size $d \times k$. In a full fine-tuning scenario, updating W would involve learning $d \times k$ parameters. In LoRA, the low-rank update involves learning:

$$d \times r \quad (\text{from } A) \quad + \quad r \times k \quad (\text{from } B)$$

When r is much smaller than d and k, the total number of trainable parameters is drastically reduced.

Optional Scaling: In many implementations, a scaling factor α is applied to control the magnitude of the update:

$$\Delta W = \frac{\alpha}{r} AB$$

This scaling ensures that the contributions of the low-rank matrices are appropriately balanced with the fixed weights W.

Forward Pass Computation Example in a Transformer Layer: For an input vector h, a typical linear transformation with weight W becomes:

$$h\,W' = h\left(W + \frac{\alpha}{r}AB\right) = h\,W + \frac{\alpha}{r}\,h\,AB$$

Only A and B are updated during training, while W stays constant.

Below is a simplified PyTorch example of a LoRA-enabled linear layer. In a transformer, you could replace, say, the query projection with this LoRA-enabled layer:

```
import torch
import torch.nn as nn
import math

class LoRALinear(nn.Module):
    def __init__(self, in_features, out_features, r=4, alpha=1.0):
        super(LoRALinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.r = r
        self.alpha = alpha
        # The original weight is frozen
        self.weight = nn.Parameter(torch.Tensor(out_features, in_features),
    requires_grad=False)
        # Initialize LoRA parameters
        self.A = nn.Parameter(torch.Tensor(r, in_features))
        self.B = nn.Parameter(torch.Tensor(out_features, r))
        # Initialize weights
        nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))
        nn.init.zeros_(self.B)
        # Scaling factor for the low-rank update
        self.scaling = self.alpha / self.r

    def forward(self, x):
        # Standard linear output using the frozen weight
        base_output = torch.nn.functional.linear(x, self.weight)
        # LoRA update
        lora_output = torch.nn.functional.linear(x, self.B @ self.A) * self.
    scaling
        # Return the sum of the base output and the LoRA update
        return base_output + lora_output

# Example usage in a transformer module:
class TransformerAttention(nn.Module):
    def __init__(self, d_model, num_heads, r=4, alpha=1.0):
        super(TransformerAttention, self).__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        # Let's assume d_model is divisible by num_heads.
        self.head_dim = d_model // num_heads

        # Replace standard linear layers with LoRA-enhanced layers for query,
    key, and value
        self.q_proj = LoRALinear(d_model, d_model, r=r, alpha=alpha)
        self.k_proj = LoRALinear(d_model, d_model, r=r, alpha=alpha)
        self.v_proj = LoRALinear(d_model, d_model, r=r, alpha=alpha)

        # The output projection can be standard or also LoRA-enhanced based on
    design choices
        self.out_proj = nn.Linear(d_model, d_model)

    def forward(self, x):
        # x shape: (batch_size, seq_length, d_model)
```

```
        Q = self.q_proj(x)
        K = self.k_proj(x)
        V = self.v_proj(x)
        # (Further processing like splitting into heads, scaled dot-product attention, etc.)
        # Here, we just illustrate the projections.
        return self.out_proj(Q)  # Simplified output
```

In this example, only the query, key, and value projections are augmented with LoRA.