INF721 - Deep Learning
L17: Transformers

Prof. Lucas N. Ferreira
Universidade Federal de Viçosa

2024/2

1 Introduction

The Transformer architecture, introduced in the seminal paper "Attention is All You Need" (Vaswani et al., 2017), represents a significant breakthrough in sequence processing. Unlike previous approaches based on recurrent neural networks (RNNs), Transformers process entire sequences in parallel, leading to more efficient training and better handling of long-range dependencies.

2 Motivation: Limitations of RNNs

Traditional RNN-based architectures face two major challenges:

1. **Long-range Dependencies**: RNNs struggle to capture dependencies between distant elements in a sequence due to the vanishing gradient problem.
2. **Sequential Processing**: The inherently sequential nature of RNNs makes parallel processing impossible, leading to longer training times.

3 The Transformer Architecture

The Transformer consists of two main components:

1. **Encoder**: Processes the input sequence $X = \{\mathbf{x}_1, ..., \mathbf{x}_{T_x}\}$ and generates a sequence of encoded representations $C = \{\mathbf{c}_1, ..., \mathbf{c}_{T_x}\}$
2. **Decoder**: Generates the output sequence $Y = \{\mathbf{y}_1, ..., \mathbf{y}_{T_y}\}$ based on the encoded representations C and the previous output tokens $Y = \{\mathbf{y}_1, ..., \mathbf{y}_{t-1}\}$

4 Self-Attention

The key innovation in Transformers is the self-attention mechanism, which allows each element in a sequence to attend to all other elements. For a sequence of length T, the self-attention is difined as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{1}$$

where:

- $\mathbf{Q}$ (Query): Linear transformation of input for querying other elements
- $\mathbf{K}$ (Key): Linear transformation of input for being queried
- $\mathbf{V}$ (Value): Linear transformation of input for aggregating information
- d_k: Dimension of the key vectors

Self-Attention Calculation Example

Step 1: Input Embeddings

Let's use word embeddings of dimension $d_e = 4$ for simplicity. These values represent semantic features of each word and were learned during training with an Embedding layer $E_s = EO_s$, where E is the embedding matrix and O_s is the one-hot encoded input sentence.

$$\begin{aligned}
\mathbf{e}_1 \text{ (Lucas)} &= [1.0, 0.2, 0.1, 0.3] \text{ // Name features} \\
\mathbf{e}_2 \text{ (will)} &= [0.2, 1.0, 0.3, 0.1] \text{ // Modal verb features} \\
\mathbf{e}_3 \text{ (travel)} &= [0.3, 0.2, 1.0, 0.4] \text{ // Action features} \\
\mathbf{e}_4 \text{ (in)} &= [0.1, 0.3, 0.2, 1.0] \text{ // Preposition features} \\
\mathbf{e}_5 \text{ (December)} &= [0.4, 0.1, 0.3, 1.0] \text{ // Time features}
\end{aligned}$$

The input embeddings are concatenated into a matrix X_e, where which row represents the embedding of a word:

$$X_e = \begin{bmatrix} 1.0 & 0.2 & 0.1 & 0.3 \\ 0.2 & 1.0 & 0.3 & 0.1 \\ 0.3 & 0.2 & 1.0 & 0.4 \\ 0.1 & 0.3 & 0.2 & 1.0 \\ 0.4 & 0.1 & 0.3 & 1.0 \end{bmatrix}$$

Step 2: Computing Q, K, V Matrices

Assume the model started with the following weight matrices for W^Q, W^K, and W^V:

$$W^Q = \begin{bmatrix} 0.8 & -0.1 & 0.2 & 0.1 \\ 0.1 & 0.9 & -0.1 & 0.2 \\ 0.2 & 0.1 & 0.8 & -0.1 \\ -0.1 & 0.2 & 0.1 & 0.9 \end{bmatrix}$$

$$W^K = \begin{bmatrix} 0.9 & 0.1 & -0.1 & 0.2 \\ -0.1 & 0.8 & 0.2 & 0.1 \\ 0.2 & -0.1 & 0.9 & 0.1 \\ 0.1 & 0.2 & 0.1 & 0.8 \end{bmatrix}$$

$$W^V = \begin{bmatrix} 0.7 & 0.2 & 0.1 & 0.1 \\ 0.2 & 0.8 & 0.1 & 0.0 \\ 0.1 & 0.1 & 0.9 & 0.0 \\ 0.0 & 0.0 & 0.1 & 0.8 \end{bmatrix}$$

Computing Q, K, V by multiplying input embeddings X_e with respective weight matrices:

$$\mathbf{Q} = X_e \times W^Q = \begin{bmatrix} 0.81 & 0.15 & 0.29 & 0.4 \\ 0.31 & 0.93 & 0.19 & 0.28 \\ 0.42 & 0.33 & 0.88 & 0.33 \\ 0.05 & 0.48 & 0.25 & 0.95 \\ 0.29 & 0.28 & 0.41 & 0.93 \end{bmatrix}$$

$$\mathbf{K} = X_e \times W^k = \begin{bmatrix} 0.93 & 0.31 & 0.06 & 0.47 \\ 0.15 & 0.81 & 0.46 & 0.25 \\ 0.49 & 0.17 & 0.95 & 0.5 \\ 0.2 & 0.43 & 0.33 & 0.87 \\ 0.51 & 0.29 & 0.35 & 0.92 \end{bmatrix}$$

$$\mathbf{V} = X_e \times W^v = \begin{bmatrix} 0.75 & 0.37 & 0.24 & 0.34 \\ 0.37 & 0.87 & 0.4 & 0.1 \\ 0.35 & 0.32 & 0.99 & 0.35 \\ 0.15 & 0.28 & 0.32 & 0.81 \\ 0.33 & 0.19 & 0.42 & 0.84 \end{bmatrix}$$

Step 3: Computing Attention Scores

Calculate scaled dot-product attention scores:

$$\hat{A} = \frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_k}} = \frac{\mathbf{Q} \times \mathbf{K}^T}{2} = \begin{bmatrix} 0.5026 & 0.2382 & 0.44895 & 0.3351 & 0.46305 \\ 0.3598 & 0.4786 & 0.31525 & 0.3841 & 0.37595 \\ 0.3504 & 0.4088 & 0.63145 & 0.4017 & 0.46075 \\ 0.3284 & 0.3744 & 0.4093 & 0.5627 & 0.5631 \\ 0.4091 & 0.3457 & 0.5221 & 0.5614 & 0.6141 \end{bmatrix}$$

Step 4: Apply Softmax

Applying softmax row-wise to get attention weights:

$$A = \text{softmax}(\hat{A}) = \begin{bmatrix} 0.2211 & 0.1697 & 0.2096 & 0.1870 & 0.2125 \\ 0.1952 & 0.2198 & 0.1867 & 0.2000 & 0.1984 \\ 0.1801 & 0.1909 & 0.2385 & 0.1895 & 0.2011 \\ 0.1767 & 0.1850 & 0.1916 & 0.2233 & 0.2234 \\ 0.1835 & 0.1722 & 0.2054 & 0.2137 & 0.2252 \end{bmatrix}$$

Each element in the attention matrix $A_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$ is the dot product of the query vector $\mathbf{q}_i$ and the key vector $\mathbf{k}_j$, and represents the importance of the j-th word for the i-th word.

Step 5: Final Attention Output

The final output of the self-attention mechanism is the weighted sum of the value matrix $\mathbf{V}$ using the attention weights:

$$C = A \times \mathbf{V} = \begin{bmatrix} 0.4001 & 0.3893 & 0.4775 & 0.4955 \\ 0.3885 & 0.4168 & 0.4668 & 0.4822 \\ 0.3839 & 0.4002 & 0.5007 & 0.4861 \\ 0.3752 & 0.3926 & 0.4713 & 0.5141 \\ 0.3795 & 0.3860 & 0.4792 & 0.5137 \end{bmatrix}$$

The self-attention layer learns contextual embebbinds C_i for each input word X_i, where i is a row in the matrix. In this new representaion, each word still is represented in a row, but now with d_v features, instead of d_e. Iin this case, $d_v = d_e$, which is a common choice. This new contextual embeddings C_i have enriched the previous word embeddings X_i with information from other words in the sentence. These enriched embeddings are the one of the main advantages of the self-attention mechanism.

5 Multi-Head Attention

Instead of a single self-attention layer, the Transformer concatenates multiple self-attention representations (called heads) to capture different aspects of the input sequence. Multi-head attention allows the model to jointly attend to information from different representation subspaces:

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}^O \tag{2}$$

where each head is:

$$\text{head}_i = \text{Attention}(\mathbf{X}\mathbf{W}_i^Q, \mathbf{X}\mathbf{W}_i^K, \mathbf{X}\mathbf{W}_i^V) \tag{3}$$

Multi-Head Self-Attention Example

Let's illustrate multi-head self-attention with a simple example. Consider the input sentence "Lucas will travel" and the following parameters:

- $T_x = 3$ (sequence length)
- $d_{\text{model}} = 9$ (embedding dimension)
- $h = 3$ (number of heads)
- $d_k = d_q = d_v = d_{\text{model}}/h = 3$ (dimension per head)

1. Input Embeddings

Each word has a 9-dimensional embedding:

$$\mathbf{E} = \begin{bmatrix} 1.0 & 0.2 & 0.3 & 0.1 & 0.4 & 0.2 & 0.3 & 0.1 & 0.5 \\ 0.2 & 1.0 & 0.1 & 0.3 & 0.2 & 0.4 & 0.1 & 0.5 & 0.2 \\ 0.3 & 0.2 & 1.0 & 0.4 & 0.1 & 0.3 & 0.2 & 0.3 & 0.4 \end{bmatrix}_{3\times 9}$$

2. Weight Matrices for Each Head

For each head i, we have three weight matrices:

- Query matrix $\mathbf{W}_i^Q \in \mathbb{R}^{9\times 3}$
- Key matrix $\mathbf{W}_i^K \in \mathbb{R}^{9\times 3}$
- Value matrix $\mathbf{W}_i^V \in \mathbb{R}^{9\times 3}$

3. Computing Q, K, V for Each Head

Head 1:

$$\mathbf{Q}_1 = \mathbf{E}\mathbf{W}_1^Q = \begin{bmatrix} 0.8 & 0.2 & 0.1 \\ 0.3 & 0.7 & 0.2 \\ 0.1 & 0.3 & 0.8 \end{bmatrix}_{3\times3}$$

$$\mathbf{K}_1 = \mathbf{E}\mathbf{W}_1^K = \begin{bmatrix} 0.7 & 0.3 & 0.2 \\ 0.2 & 0.8 & 0.1 \\ 0.3 & 0.2 & 0.7 \end{bmatrix}_{3\times3}$$

$$\mathbf{V}_1 = \mathbf{E}\mathbf{W}_1^V = \begin{bmatrix} 0.9 & 0.2 & 0.1 \\ 0.1 & 0.8 & 0.2 \\ 0.2 & 0.1 & 0.9 \end{bmatrix}_{3\times3}$$

Head 2:

$$\mathbf{Q}_2 = \mathbf{E}\mathbf{W}_2^Q = \begin{bmatrix} 0.6 & 0.3 & 0.2 \\ 0.2 & 0.6 & 0.3 \\ 0.3 & 0.2 & 0.6 \end{bmatrix}_{3\times3}$$

$$\mathbf{K}_2 = \mathbf{E}\mathbf{W}_2^K = \begin{bmatrix} 0.5 & 0.4 & 0.2 \\ 0.3 & 0.5 & 0.3 \\ 0.2 & 0.3 & 0.6 \end{bmatrix}_{3\times3}$$

$$\mathbf{V}_2 = \mathbf{E}\mathbf{W}_2^V = \begin{bmatrix} 0.7 & 0.3 & 0.2 \\ 0.2 & 0.7 & 0.3 \\ 0.3 & 0.2 & 0.7 \end{bmatrix}_{3\times3}$$

Head 3:

$$\mathbf{Q}_3 = \mathbf{E}\mathbf{W}_3^Q = \begin{bmatrix} 0.4 & 0.4 & 0.3 \\ 0.3 & 0.5 & 0.3 \\ 0.4 & 0.3 & 0.4 \end{bmatrix}_{3\times3}$$

$$\mathbf{K}_3 = \mathbf{E}\mathbf{W}_3^K = \begin{bmatrix} 0.5 & 0.3 & 0.3 \\ 0.4 & 0.4 & 0.3 \\ 0.3 & 0.3 & 0.5 \end{bmatrix}_{3\times3}$$

$$\mathbf{V}_3 = \mathbf{E}\mathbf{W}_3^V = \begin{bmatrix} 0.6 & 0.3 & 0.2 \\ 0.3 & 0.6 & 0.2 \\ 0.2 & 0.3 & 0.6 \end{bmatrix}_{3\times3}$$

4. Computing Attention Scores for Each Head

Head 1:

$$\text{Scores}_1 = \frac{\mathbf{Q}_1\mathbf{K}_1^T}{\sqrt{3}} \rightarrow \text{softmax} \rightarrow \mathbf{A}_1 = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.2 & 0.7 \end{bmatrix}$$

Head 2:

$$\text{Scores}_2 = \frac{\mathbf{Q}_2\mathbf{K}_2^T}{\sqrt{3}} \rightarrow \text{softmax} \rightarrow \mathbf{A}_2 = \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.3 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.5 \end{bmatrix}$$

Head 3:

$$\text{Scores}_3 = \frac{\mathbf{Q}_3\mathbf{K}_3^T}{\sqrt{3}} \rightarrow \text{softmax} \rightarrow \mathbf{A}_3 = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.3 & 0.4 & 0.3 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}$$

5. Output Vectors for Each Head

For each head i:

$$\mathbf{C}_i = \mathbf{A}_i\mathbf{V}_i \in \mathbb{R}^{3\times 3}$$

6. Concatenating Head Outputs

Concatenate the outputs horizontally:

$$\mathbf{C}_{\text{concat}} = [\mathbf{C}_1 \parallel \mathbf{C}_2 \parallel \mathbf{C}_3] \in \mathbb{R}^{3\times 9}$$

7. Final Linear Projection

Apply final output projection $\mathbf{W}^O \in \mathbb{R}^{9\times 9}$:

$$\mathbf{C}_{\text{final}} = \mathbf{C}_{\text{concat}}\mathbf{W}^O \in \mathbb{R}^{3\times 9}$$

Key Observations

- Each head operates on a different 3-dimensional subspace of the 9-dimensional embedding
- Attention patterns differ across heads:
  - Head 1 shows strong self-attention (diagonal)

  - Head 2 shows moderate mixing
  - Head 3 shows more uniform attention

- The final output preserves the original dimensions (3x9)
- Each head can specialize in different types of relationships, such as syntax (e.g., head 1), semantics (e.g., head 2), or encoded position (e.g., head 3).

6 Position Encoding

The multi-head self-attention mechanism does not inherently capture the sequential order of the input sequence. Therefore, positional information must be explicitly added. The original model addes sinusoidal position encodings to the input embeddings to provide information about the position of each token in the sequence:

$$\begin{aligned} PE_{(pos,2i)} &= \sin(pos/10000^{2i/d_{model}}) \text{ for even indices} \\ PE_{(pos,2i+1)} &= \cos(pos/10000^{2i/d_{model}}) \text{ for odd indices} \end{aligned} \tag{4}$$

where pos is the position and i is the dimension.

Positional Embeddings Example

Let's illustrate how positional encodings are added to word embeddings using a simple example. Consider the sentence "Lucas will travel in December" and the following parameters:

- $d_{\text{model}} = 4$ (embedding dimension)
- Sequence length $T_x = 5$ words

1. Computing Scale Factors

When computation scale factors, we have a sine function with frequency $f = 10000^{2i/d_{\text{model}}}$ for even indices and a cosine function with the same frequency f for odd indices. Therefore, we group the embedding dimensions into pairs $[[0, 1], [2, 3]]$ and compute the scale factors for each pair. In this example, each word has dimension $d_{model} = 4$, so we have $d_{model}/2 = 2$ different frequencies:

$$10000^{0/4} = 1 \text{ for } i = 0$$
$$10000^{2/4} = 100 \text{ for } i = 1$$

Just to give another example, if we had $d_{\text{model}} = 6$, we would have three different frequencies.

2. Computing PE Values

For each position ($t = 0$ to $t = 4$) and each dimension ($i = 0$ to $i = 3$):
Position 0 (Lucas):

$$PE_{(t=0,i=0)} = \sin(0/1) = 0.000$$
$$PE_{(t=0,i=1)} = \cos(0/1) = 1.000$$
$$PE_{(t=0,i=2)} = \sin(0/100) = 0.000$$
$$PE_{(t=0,i=3)} = \cos(0/100) = 1.000$$

Position 1 (will):

$$PE_{(t=1,i=0)} = \sin(1/1) = 0.841$$
$$PE_{(t=1,i=1)} = \cos(1/1) = 0.540$$
$$PE_{(t=1,i=2)} = \sin(1/100) = 0.010$$
$$PE_{(t=1,i=3)} = \cos(1/100) = 0.999$$

Position 2 (travel):

$$PE_{(t=2,i=0)} = \sin(2/1) = 0.909$$
$$PE_{(t=2,i=1)} = \cos(2/1) = -0.416$$
$$PE_{(t=2,i=2)} = \sin(2/100) = 0.020$$
$$PE_{(t=2,i=3)} = \cos(2/100) = 0.999$$

Position 3 (in):

$$PE_{(t=3,i=0)} = \sin(3/1) = 0.141$$
$$PE_{(t=3,i=1)} = \cos(3/1) = -0.990$$
$$PE_{(t=3,i=2)} = \sin(3/100) = 0.030$$
$$PE_{(t=3,i=3)} = \cos(3/100) = 0.999$$

Position 4 (December):

$$PE_{(t=4,i=0)} = \sin(4/1) = -0.757$$
$$PE_{(t=4,i=1)} = \cos(4/1) = -0.654$$
$$PE_{(t=4,i=2)} = \sin(4/100) = 0.040$$
$$PE_{(t=4,i=3)} = \cos(4/100) = 0.999$$

3. Complete Positional Encoding Matrix

$$PE = \begin{bmatrix} 0.000 & 1.000 & 0.000 & 1.000 \\ 0.841 & 0.540 & 0.010 & 0.999 \\ 0.909 & -0.416 & 0.020 & 0.999 \\ 0.141 & -0.990 & 0.030 & 0.999 \\ -0.757 & -0.654 & 0.040 & 0.999 \end{bmatrix}$$

4. Adding to Word Embeddings

If we have an input sentence represented with word embeddings $\mathbf{X_e}$, the final input to the Transformer would be:

$$\mathbf{X_{epe}} = \mathbf{X_e} + PE$$

Key Properties of the Positional Encoding

1. **Unique Patterns**
   - Each position has a unique encoding
   - Values vary at different frequencies
   - First two dimensions vary quickly
   - Last two dimensions vary slowly
2. **Bounded Values**
   - All values are between -1 and 1
   - Sum of squares for each position is constant
3. **Linear Combination**
   - Different dimensions capture position at different scales
   - Even indices use sine, odd indices use cosine

- Creates a unique pattern for each position

4. **Properties Preserved**
   - Relative positions can be learned through attention
   - Values don't grow with sequence length
   - Pattern repeats for very long sequences

7 Position-wise Feed-Forward Network

Since the self-attention mechanism in Transformers is purely linear (weighted sum), a position-wise feed-forward network (FFN) is added to introduce non-linearity and position-specific feature refinement. Moreover, the FNN adds learnable parameters to the model, increasing the model capaxity and helping processing complex patterns. The FFN consists of two linear transformations with a ReLU activation:

$$\text{FFN}(\mathbf{C}) = \max(0, \mathbf{C}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

Typically:

- Input/Output dimension: d_{model}
- Inner dimension: d_{ff} (usually $4 \times d_{\text{model}}$)

Position-wise FFN Example Calculation

For sequence "Lucas will travel" with $d_{\text{model}} = 4$ and $d_{\text{ff}} = 8$:

Input After Multihead Attention Layer ($\mathbf{C}$):

$$\mathbf{C} = \begin{bmatrix} 0.5 & 0.3 & -0.2 & 0.1 \\ 0.2 & 0.4 & 0.3 & -0.1 \\ -0.1 & 0.2 & 0.5 & 0.3 \end{bmatrix}_{3\times 4}$$

First Layer ($\mathbf{W}_1 \in \mathbb{R}^{4\times 8}$):

$$\mathbf{C}\mathbf{W}_1 + \mathbf{b}_1 = \begin{bmatrix} 0.7 & -0.3 & 0.4 & 0.2 & -0.5 & 0.1 & 0.3 & -0.2 \\ -0.2 & 0.5 & 0.3 & -0.1 & 0.4 & 0.2 & -0.3 & 0.1 \\ 0.3 & 0.2 & -0.4 & 0.5 & 0.1 & -0.2 & 0.4 & 0.3 \end{bmatrix}$$

After ReLU:

$$\max(0, \mathbf{C}\mathbf{W}_1 + \mathbf{b}_1) = \begin{bmatrix} 0.7 & 0.0 & 0.4 & 0.2 & 0.0 & 0.1 & 0.3 & 0.0 \\ 0.0 & 0.5 & 0.3 & 0.0 & 0.4 & 0.2 & 0.0 & 0.1 \\ 0.3 & 0.2 & 0.0 & 0.5 & 0.1 & 0.0 & 0.4 & 0.3 \end{bmatrix}$$

Final Output ($\mathbf{W}_2 \in \mathbb{R}^{8\times 4}$):

$$\text{FFN}(\mathbf{C}) = \mathbf{C}\mathbf{W}_2 = \begin{bmatrix} 0.6 & 0.2 & -0.1 & 0.3 \\ 0.1 & 0.5 & 0.4 & -0.2 \\ 0.2 & 0.3 & 0.6 & 0.1 \end{bmatrix}$$

Processing Flow

Each position is processed independently:

$$\text{FFN}(\mathbf{C}_i) = f(\mathbf{C}_i) \text{ for each position (row) } i$$

Same transformation applied to each position:

- Weight matrices $\mathbf{W}_1$, $\mathbf{W}_2$ shared across positions
- Allows parameter efficiency
- Maintains position independence

8 Transformer Blocks

8.1 Encoder Block

Each encoder block consists of:

1. Multi-head self-attention
2. Layer normalization
3. Feed-forward neural network
4. Residual connections

Encoder Block Algorithm

Input: Sequence $\mathbf{X}$
$\mathbf{X}' = \mathbf{X} + \text{PositionalEncoding}(\mathbf{X})$
$\mathbf{A} = \text{LayerNorm}(\mathbf{X}' + \text{MultiHeadAttention}(\mathbf{X}'))$
$\mathbf{O} = \text{LayerNorm}(\mathbf{A} + \text{FFN}(\mathbf{A}))$
return $\mathbf{O}$

8.2 Decoder Block

The decoder block includes:

1. Masked multi-head self-attention
2. Multi-head cross-attention with encoder outputs
3. Feed-forward neural network
4. Layer normalization and residual connections

Decoder Block Algorithm

Input: Sequence $\mathbf{Y}$, Encoder outputs $\mathbf{C}$
$\mathbf{Y}' = \mathbf{Y} + \text{PositionalEncoding}(\mathbf{Y})$
$\mathbf{A} = \text{LayerNorm}(\mathbf{Y}' + \text{MaskedMultiHeadAttention}(\mathbf{Y}'))$
$\mathbf{B} = \text{LayerNorm}(\mathbf{A} + \text{MultiHeadAttention}(\mathbf{A}, \mathbf{C}))$
$\mathbf{O} = \text{LayerNorm}(\mathbf{B} + \text{FFN}(\mathbf{B}))$
return $\mathbf{O}$

9 Training the Transformer

During training, the decoder uses teacher forcing with masked attention to prevent attending to future tokens:

Attention Mask

For a sequence of length n, the mask matrix $\mathbf{M}$ is:

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

Teacher forcing consists of feeding the true output sequence to the decoder during training, instead of using the predicted output tokens, as we did in the seq2seq translation models. This approach helps stabilize training and improve convergence. The attention mask M is summed with the scaled dot-product attention scores $\hat{A}$ before applying the softmax function:

$$\hat{A} = \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M} \tag{5}$$

$$A = \text{softmax}(\hat{A}) \tag{6}$$

10 Practical Considerations

In practice, when training Transformers we limit the model to a fixed sequence length. This limitation is primarily due to the quadratic memory requirements of the self-attention mechanism. When computing attention, each token needs to attend to every other token, resulting in an attention matrix of size $n \times n$, where n is the sequence length. This means memory usage grows quadratically $O(n^2)$ with sequence length n. For example, with a sequence length of 512, we need a 512×512 attention matrix for each attention head in each layer.

This quadratic memory requirement forces us to set a maximum sequence length in practice, typically 512 or 1024 tokens, to keep memory usage manageable, especially when processing batches of sequences and using multiple attention heads across multiple layers. While there are techniques like sliding windows or sparse attention to handle longer sequences, the standard Transformer architecture uses fixed-length sequences for efficient batching and predictable memory usage.

11 Conclusion

The Transformer architecture has revolutionized natural language processing by introducing self-attention mechanisms that capture complex relationships in sequences. The multi-head self-attention mechanism allows the model to jointly attend to different aspects of the input sequence, while the position-wise feed-forward network introduces non-linearity and position-specific feature refinement.

The Transformer architecture is composed of encoder and decoder blocks, each with multiple layers of self-attention and feed-forward networks. During training, the decoder uses teacher forcing with masked attention to prevent attending to future tokens. In practice, we limit the model to a fixed sequence length to manage memory usage, typically 512 or 1024 tokens.

The Transformer architecture has become the foundation for many state-of-the-art models in natural language processing, including BERT, GPT, and T5, and continues to drive innovation in the field. Next lecture, we will explore some of these advanced Transformer models and their applications in NLP tasks.