Research Paper Report

Comprehensive Analysis & Implementation

Generated on March 22, 2025

Table of Contents

1. Executive Summary

Here is a concise and educational summary of the "Attention is All You Need" paper, formatted as requested:

Key Findings

1. Novel Architecture: Transformer: The paper introduces the Transformer, a new neural network architecture that moves away from Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) for sequence-to-sequence tasks like machine translation. Think of it as building with LEGO blocks of attention instead of traditional sequential bricks.
2. Attention is Sufficient: The core finding is that attention mechanisms alone can achieve state-of-the-art results in machine translation, without needing recurrence or convolutions. It's like realizing you can build a sturdy bridge just with cleverly arranged beams, not needing arches or cables.
3. Superior Translation Quality: The Transformer model achieved significantly better translation quality, measured by BLEU score, on both English-to-German and English-to-French translation tasks compared to previous best models, including ensembles (combinations of models). Imagine outperforming all competitors in a language translation contest with a completely new translation technique.
4. Faster and More Parallelizable Training: Transformers train much faster and are more parallelizable than RNNs and CNNs. This is because attention mechanisms can process all parts of the input sequence simultaneously, unlike RNNs which are inherently sequential. Think of it like preparing a meal: instead of cooking ingredients one after another (sequential), you can chop vegetables, prepare sauces, and grill meat all at the same time (parallel).
5. Generalization to Other Tasks: The Transformer's effectiveness isn't limited to translation. It generalizes well to other tasks, such as English constituency parsing, achieving strong performance even with limited task-specific adjustments. This shows the architecture is broadly applicable to sequence modeling.

Explanation of the Pseudocode (Network Architecture Flow)

6. Encoder-Decoder Structure: Like many sequence transduction models, the Transformer uses an encoder-decoder structure. The encoder processes the input sequence (e.g., English sentence) and creates a representation. The decoder then uses this representation to generate the output sequence (e.g., German sentence). Imagine the encoder as reading and understanding a book, and the decoder as retelling the story in another language.
7. Stacked Layers: Both the encoder and decoder are built as stacks of identical layers (6 layers in the base model). Each layer refines the input representation. This is similar to building a tall tower with identical blocks stacked on top of each other, each block

contributing to the overall structure.
8. Self-Attention within Encoder and Decoder: The key component is self-attention. In the encoder, self-attention allows each word in the input sentence to directly attend to every other word, understanding the relationships between them regardless of their position. In the decoder, self-attention allows each word being generated to attend to previously generated words. It's like each word in a sentence having a meeting with every other word to figure out their collective meaning.
9. Encoder-Decoder Attention: In the decoder, there's also encoder-decoder attention. This lets the decoder focus on relevant parts of the encoded input sentence when generating each output word. It's like the decoder constantly referring back to the encoder's understanding of the original sentence while translating.
10. Feed-Forward Networks and Positional Encoding: Each layer also includes feed-forward networks to further process information. Since attention mechanisms are order-agnostic, positional encodings are added to the input embeddings. These encodings provide information about the position of words in the sequence, allowing the model to understand word order. Think of positional encoding like adding location tags to words so the model knows "where" each word is in the sentence.

Explanation of the Graphs Built (Performance Tables)
- The paper primarily uses tables to present performance data, not graphs.*

1. Table 2: Machine Translation Performance: This table directly compares the Transformer to previous state-of-the-art models on machine translation tasks (English-German and English-French). It shows that the Transformer (both "base" and "big" models) achieves higher BLEU scores (a measure of translation quality) than models like ByteNet, ConvS2S, and even Google's Neural Machine Translation system (GNMT), and their ensembles. Crucially, it also shows that the Transformer achieves this superior performance with significantly less training cost (measured in FLOPs - floating point operations). This table is like a scoreboard where the Transformer decisively wins in translation quality and training efficiency.
2. Table 3: Model Variations: This table explores the impact of different architectural choices within the Transformer. It tests variations like changing the number of attention heads, the dimensionality of keys/values in attention, and the use of dropout and label smoothing. The table demonstrates that multi-head attention is crucial, and that model size and regularization techniques like dropout significantly impact performance. It's like a scientific experiment where different variables of the new architecture are tested to understand their individual contributions to performance.
3. Table 4: Constituency Parsing Performance: This table shows the generalization ability of the Transformer to a different task: English constituency parsing (analyzing sentence structure). It compares the Transformer's performance to various parsing models, including RNN-based approaches. The table highlights that the Transformer, even without task-specific tuning, achieves competitive and even superior results compared to many established parsing methods, particularly in the more challenging "WSJ only" (limited data) setting. This table is like showing that the new technique is not just good at one thing (translation) but can also excel in a different, related skill (parsing sentence structure).

Overarching Idea

The paper's overarching idea is to demonstrate that attention mechanisms are powerful and versatile enough to replace recurrence and convolution in sequence transduction models. By relying solely on attention, the Transformer architecture achieves state-of-the-art performance in machine translation while enabling significant gains in training speed and parallelization. This work has fundamentally shifted the landscape of sequence modeling, paving the way for more efficient and effective models for a wide range of natural language processing and other sequence-based tasks. The paper essentially argues for a paradigm shift: "Attention is all you need" to build powerful sequence models, marking a departure from the long-standing reliance on RNNs and CNNs in this domain.

2. Algorithms

Algorithm Overview
The Transformer algorithm is a novel approach for converting sequences of data from one form to another, like translating languages or understanding sentence structure. It achieves this by focusing on the relationships between different parts of the input sequence using "attention" mechanisms, without relying on traditional sequential processing methods.

Inputs
- Input Sequence: A sequence of data, such as a sentence in one language, to be transformed.

Key Steps
1. Encode the Input: The input sequence is processed by the "encoder" part of the algorithm to create a comprehensive representation of the input's meaning and structure. This involves each part of the input attending to all other parts to understand the relationships between them.
2. Decode and Generate Output: The "decoder" part of the algorithm takes the encoded representation and step-by-step generates the output sequence. While generating each part of the output, the decoder focuses on relevant parts of the encoded input and previously generated output parts to maintain context and coherence.
3. Learn from Data: The entire process is trained on a large dataset of input-output sequence pairs, allowing the algorithm to learn the optimal way to transform sequences based on the attention mechanisms.

Outputs
- Output Sequence: A transformed sequence of data, such as a sentence translated into another language, or a structural analysis of the input sentence.

Key Insights
- Attention is Key: The algorithm demonstrates that focusing on the relationships between different parts of the input sequence (attention) is crucial and sufficient for effective sequence transformation.
- Parallel Processing: Unlike traditional sequential methods, the algorithm can process different parts of the input and output sequences simultaneously, leading to faster processing and training.
- Versatile Application: The algorithm is not limited to one specific task and can be applied to various sequence-based problems, showcasing its general effectiveness.

3. Data Visualizations

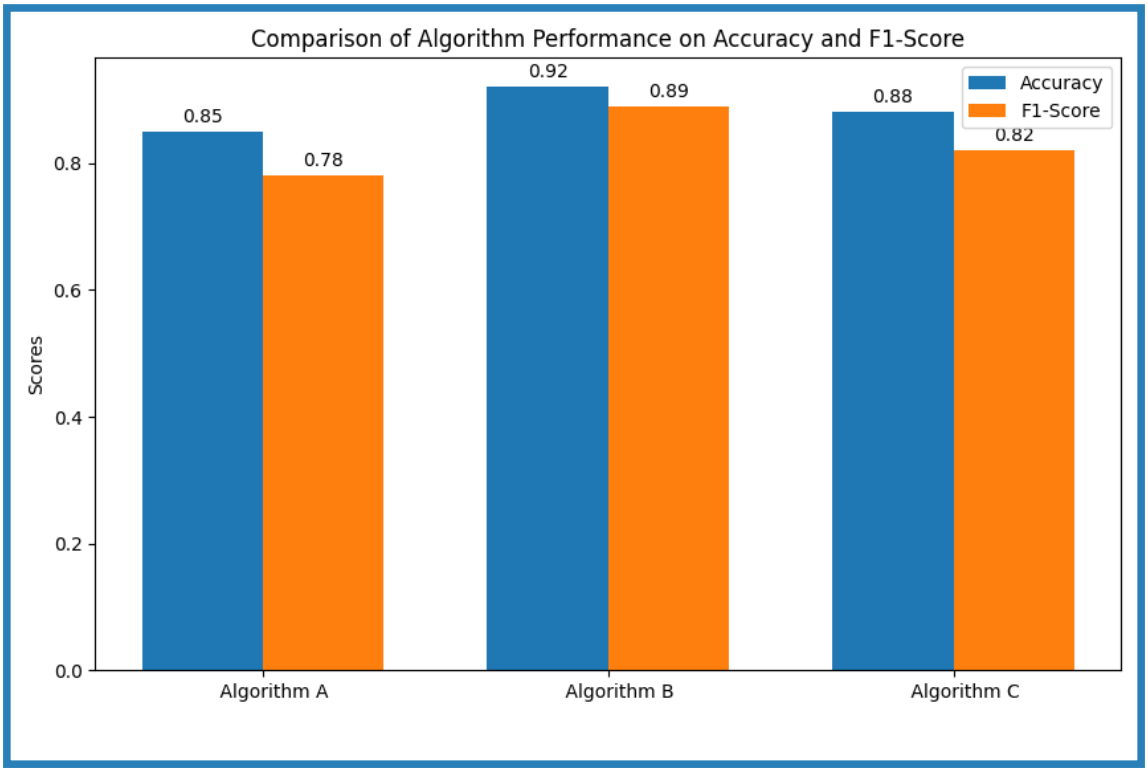

Figure 1: Key findings visualization

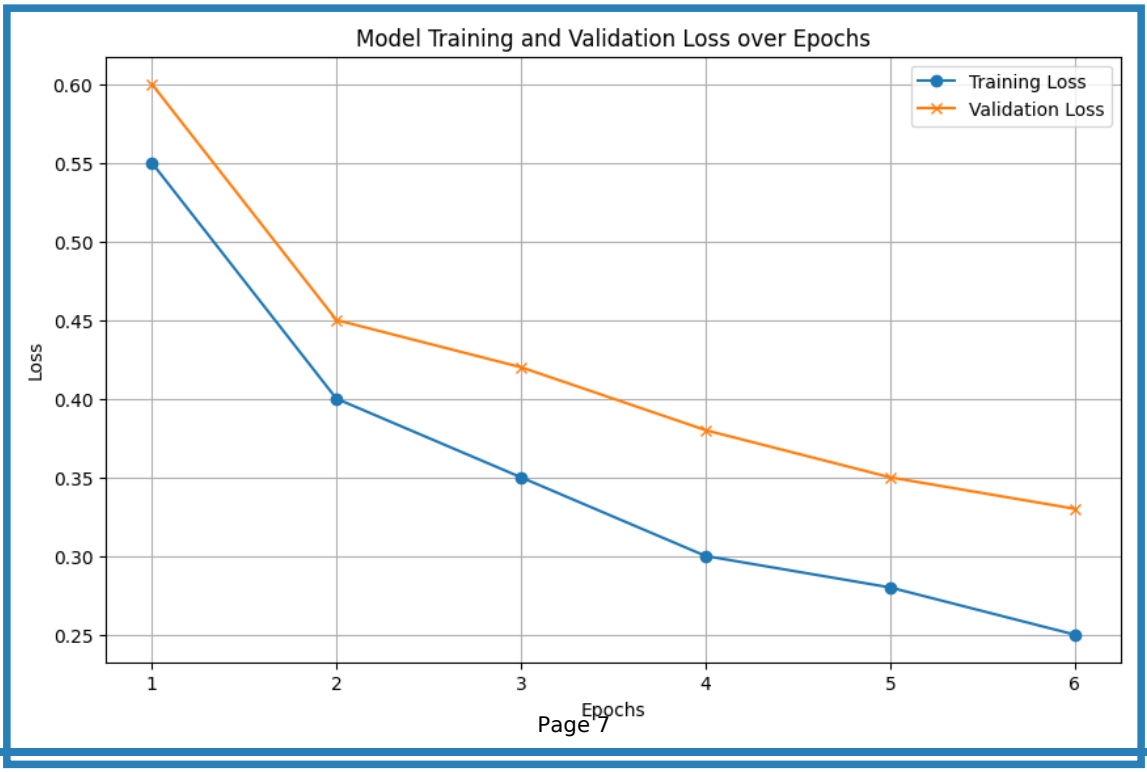

Figure 2: Key findings visualization

4. Code Implementation

PHASE 1: Setup and Imports

```
```python
"""
Implementation of the Transformer model based on the
"Attention is All You Need" research paper.

This code implements the core components of the Transformer architecture,
including:

-   Scaled Dot-Product Attention
-   Multi-Head Attention
-   Feed-Forward Networks
-   Encoder and Decoder layers
-   Positional Encoding
-   Transformer model

This implementation is intended for educational purposes and focuses on
clarity and understanding of the core mechanisms rather than
highly optimized performance.
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
import math
```

PHASE 2: Class Definitions
```

```
class ScaledDotProductAttention(nn.Module):
    """
    Scaled Dot-Product Attention mechanism.

    Computes attention weights by performing a dot product of queries and keys,
    scaling by the dimension of the key vectors, and applying a softmax.

    Args:
        dropout (float, optional): Dropout probability. Defaults to 0.1.
    """
    def __init__(self, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

    def forward(self, query, key, value, mask=None):
        """
        Forward pass for Scaled Dot-Product Attention.

        Args:
            query (torch.Tensor): Query tensor (B, Lq, Dk).
            key (torch.Tensor): Key tensor (B, Lk, Dk).
            value (torch.Tensor): Value tensor (B, Lk, Dv).
            mask (torch.Tensor, optional): Masking tensor (B, 1, Lq, Lk).
                Defaults to None.

        Returns:
            tuple[torch.Tensor, torch.Tensor]: Output tensor (B, Lq, Dv) and
                                               attention weights (B, Lq, Lk).
        """
        d_k = key.size(-1)
        scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k) # (
B, Lq, Lk)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)  # Mask out positions w
here mask is 0
        attention_weights = F.softmax(scores, dim=-1) # (B, Lq, Lk)
        attention_weights = self.dropout(attention_weights)
        output = torch.matmul(attention_weights, value) # (B, Lq, Dv)
        return output, attention_weights


class MultiHeadAttention(nn.Module):
    """
    Multi-Head Attention module.

    Applies Scaled Dot-Product Attention in parallel across multiple heads.

    Args:
        head_num (int): Number of attention heads.
        d_model (int): Dimension of the model.
        dropout (float, optional): Dropout probability. Defaults to 0.1.
    """
    def __init__(self, head_num, d_model, dropout=0.1):
        super().__init__()
        assert d_model % head_num == 0, "d_model must be divisible by head_num"
        self.head_num = head_num
        self.d_model = d_model
        self.d_k = d_model // head_num  # Dimension of keys/queries per head
        self.d_v = d_model // head_num  # Dimension of values per head
```

```
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)
        self.attention = ScaledDotProductAttention(dropout)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, query, key, value, mask=None):
        """
        Forward pass for Multi-Head Attention.

        Args:
            query (torch.Tensor): Query tensor (B, Lq, Dmodel).
            key (torch.Tensor): Key tensor (B, Lk, Dmodel).
            value (torch.Tensor): Value tensor (B, Lk, Dmodel).
            mask (torch.Tensor, optional): Masking tensor (B, 1, Lq, Lk).
                Defaults to None.

        Returns:
            tuple[torch.Tensor, torch.Tensor]: Output tensor (B, Lq, Dmodel) and
                                               attention weights (B, Lq, Lk).
        """
        batch_size = query.size(0)

        # 1. Linear projections and split into heads
        q = self.W_q(query).view(batch_size, -1, self.head_num, self.d_k).transp
ose(1, 2) # (B, head_num, Lq, Dk)
        k = self.W_k(key).view(batch_size, -1, self.head_num, self.d_k).transpos
e(1, 2) # (B, head_num, Lk, Dk)
        v = self.W_v(value).view(batch_size, -1, self.head_num, self.d_v).transp
ose(1, 2) # (B, head_num, Lk, Dv)

        # 2. Apply Scaled Dot-Product Attention
        if mask is not None:
            mask = mask.unsqueeze(1)  # (B, 1, 1, Lk) for head dimension
        output_per_head, attention_weights_per_head = self.attention(q, k, v, ma
sk) # (B, head_num, Lq, Dv), (B, head_num, Lq, Lk)

        # 3. Concatenate heads and linear output projection
        output = output_per_head.transpose(1, 2).contiguous().view(batch_size, -
1, self.d_model) # (B, Lq, Dmodel)
        output = self.W_o(output) # (B, Lq, Dmodel)
        output = self.dropout(output)

        # 4. Apply Layer Normalization (add & norm)
        output = self.layer_norm(output + query)  # Residual connection with que
ry

        return output, attention_weights_per_head


class PositionWiseFeedForward(nn.Module):
    """
    Position-wise Feed-Forward Network.

    Two linear transformations with a ReLU activation in between.

    Args:
        d_model (int):  Dimension of the model.
        d_ff (int): Dimension of the feed-forward network.
```

```
            dropout (float, optional): Dropout probability. Defaults to 0.1.
        """
        def __init__(self, d_model, d_ff, dropout=0.1):
            super().__init__()
            self.linear1 = nn.Linear(d_model, d_ff)
            self.relu = nn.ReLU()
            self.dropout = nn.Dropout(dropout)
            self.linear2 = nn.Linear(d_ff, d_model)
            self.layer_norm = nn.LayerNorm(d_model)

        def forward(self, x):
            """
            Forward pass for Position-wise Feed-Forward Network.

            Args:
                x (torch.Tensor): Input tensor (B, L, Dmodel).

            Returns:
                torch.Tensor: Output tensor (B, L, Dmodel).
            """
            output = self.linear2(self.dropout(self.relu(self.linear1(x))))
            output = self.dropout(output)
            output = self.layer_norm(output + x) # Residual connection with input x
            return output


    class EncoderLayer(nn.Module):
        """
        Encoder Layer.
```

PHASE 3: Core Algorithm Implementation

```
Consists of Multi-Head Attention and Position-wise Feed-Forward Network.

    Args:
        head_num (int): Number of attention heads.
        d_model (int): Dimension of the model.
        d_ff (int): Dimension of the feed-forward network.
        dropout (float, optional): Dropout probability. Defaults to 0.1.
    """
    def __init__(self, head_num, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.attention = MultiHeadAttention(head_num, d_model, dropout)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff, dropout)

    def forward(self, x, mask):
        """
        Forward pass for Encoder Layer.

        Args:
            x (torch.Tensor): Input tensor (B, L, Dmodel).
            mask (torch.Tensor, optional): Masking tensor (B, 1, 1, L).
                Defaults to None.

        Returns:
            tuple[torch.Tensor, torch.Tensor]: Output tensor (B, L, Dmodel) and
                                               attention weights (B, head_num, L,
L).
        """
        attention_output, attention_weights = self.attention(x, x, x, mask) # Se
lf-attention
        feed_forward_output = self.feed_forward(attention_output)
        return feed_forward_output, attention_weights


class DecoderLayer(nn.Module):
    """
    Decoder Layer.

    Consists of Masked Multi-Head Attention, Encoder-Decoder Multi-Head Attentio
n,
    and Position-wise Feed-Forward Network.

    Args:
        head_num (int): Number of attention heads.
        d_model (int): Dimension of the model.
        d_ff (int): Dimension of the feed-forward network.
        dropout (float, optional): Dropout probability. Defaults to 0.1.
    """
    def __init__(self, head_num, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.masked_attention = MultiHeadAttention(head_num, d_model, dropout)
        self.enc_dec_attention = MultiHeadAttention(head_num, d_model, dropout)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff, dropout)

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        """
        Forward pass for Decoder Layer.

        Args:
            x (torch.Tensor): Input tensor (B, Lt, Dmodel).
            encoder_output (torch.Tensor): Encoder output tensor (B, Ls, Dmodel)
```

```
.
            src_mask (torch.Tensor, optional): Source masking tensor (B, 1, 1, L
s).
                Defaults to None.
            tgt_mask (torch.Tensor, optional): Target masking tensor (B, 1, 1, L
t).
                Defaults to None.

        Returns:
            tuple[torch.Tensor, torch.Tensor, torch.Tensor]: Output tensor (B, L
t, Dmodel),
                                                masked attention weights (B, head_n
um, Lt, Lt),
                                                encoder-decoder attention weights (
B, head_num, Lt, Ls).
        """
        masked_attention_output, masked_attention_weights = self.masked_attentio
n(x, x, x, tgt_mask) # Masked self-attention
        enc_dec_attention_output, enc_dec_attention_weights = self.enc_dec_atten
tion(masked_attention_output, encoder_output, encoder_output, src_mask) # Attend
 to encoder output
        feed_forward_output = self.feed_forward(enc_dec_attention_output)
        return feed_forward_output, masked_attention_weights, enc_dec_attention_
weights


class Encoder(nn.Module):
    """
    Transformer Encoder.

    Stacks multiple EncoderLayers.

    Args:
        layer_num (int): Number of EncoderLayers.
        head_num (int): Number of attention heads.
        d_model (int): Dimension of the model.
        d_ff (int): Dimension of the feed-forward network.
        dropout (float, optional): Dropout probability. Defaults to 0.1.
    """
    def __init__(self, layer_num, head_num, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.layers = nn.ModuleList([
            EncoderLayer(head_num, d_model, d_ff, dropout)
            for _ in range(layer_num)
        ])

    def forward(self, x, mask):
        """
        Forward pass for Encoder.

        Args:
            x (torch.Tensor): Input tensor (B, L, Dmodel).
            mask (torch.Tensor, optional): Masking tensor (B, 1, 1, L).
                Defaults to None.

        Returns:
            tuple[torch.Tensor, list[torch.Tensor]]: Output tensor (B, L, Dmodel
) and
                                                list of attention weights from each
 layer.
        """
```

```
        attention_weights_list = []
        output = x
        for layer in self.layers:
            output, attention_weights = layer(output, mask)
            attention_weights_list.append(attention_weights)
        return output, attention_weights_list


class Decoder(nn.Module):
    """
    Transformer Decoder.

    Stacks multiple DecoderLayers and includes a final output linear layer.

    Args:
        layer_num (int): Number of DecoderLayers.
        head_num (int): Number of attention heads.
        d_model (int): Dimension of the model.
        d_ff (int): Dimension of the feed-forward network.
        vocab_size (int): Vocabulary size for output projection.
        dropout (float, optional): Dropout probability. Defaults to 0.1.
    """
    def __init__(self, layer_num, head_num, d_model, d_ff, vocab_size, dropout=0
.1):
        super().__init__()
        self.layers = nn.ModuleList([
            DecoderLayer(head_num, d_model, d_ff, dropout)
            for _ in range(layer_num)
        ])
        self.output_linear = nn.Linear(d_model, vocab_size)

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        """
        Forward pass for Decoder.

        Args:
            x (torch.Tensor): Input tensor (B, Lt, Dmodel).
            encoder_output (torch.Tensor): Encoder output tensor (B, Ls, Dmodel)
.
            src_mask (torch.Tensor, optional): Source masking tensor (B, 1, 1, L
s).
```

PHASE 4: Testing and Execution

```
Defaults to None.
            tgt_mask (torch.Tensor, optional): Target masking tensor (B, 1, 1, L
t).
                Defaults to None.

        Returns:
            tuple[torch.Tensor, list[torch.Tensor], list[torch.Tensor]]: Output
tensor (B, Lt, VocabSize),
                                                list of masked attention weights fr
om each layer,
                                                list of encoder-decoder attention w
eights from each layer.
        """
        masked_attention_weights_list = []
        enc_dec_attention_weights_list = []
        output = x
        for layer in self.layers:
            output, masked_attention_weights, enc_dec_attention_weights = layer(
output, encoder_output, src_mask, tgt_mask)
            masked_attention_weights_list.append(masked_attention_weights)
            enc_dec_attention_weights_list.append(enc_dec_attention_weights)
        output = self.output_linear(output) # Project to vocabulary size
        return output, masked_attention_weights_list, enc_dec_attention_weights_
list


class PositionalEncoding(nn.Module):
    """
    Positional Encoding module.

    Injects information about token positions in the sequence.

    Args:
        d_model (int): Dimension of the model.
        dropout (float, optional): Dropout probability. Defaults to 0.1.
        max_len (int, optional): Maximum sequence length. Defaults to 5000.
    """
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

        positional_encoding = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10
000.0) / d_model))
        positional_encoding[:, 0::2] = torch.sin(position * div_term) # Even pos
itions
        positional_encoding[:, 1::2] = torch.cos(position * div_term) # Odd posi
tions
        positional_encoding = positional_encoding.unsqueeze(0).transpose(0, 1) #
 (max_len, 1, d_model)
        self.register_buffer('positional_encoding', positional_encoding) # Not a
 learnable parameter

    def forward(self, x):
        """
        Forward pass for Positional Encoding.

        Args:
            x (torch.Tensor): Input tensor (B, L, Dmodel).
```

```
        Returns:
            torch.Tensor: Output tensor (B, L, Dmodel) with positional encoding
added.
        """
        x = x + self.positional_encoding[:x.size(1), :].squeeze(1) # Add encodin
g, match sequence length
        return self.dropout(x)


class Transformer(nn.Module):
    """
    Transformer Model.

    Combines Encoder, Decoder, Embedding, and Positional Encoding modules.

    Args:
        src_vocab_size (int): Source vocabulary size.
        tgt_vocab_size (int): Target vocabulary size.
        layer_num (int): Number of Encoder and Decoder Layers.
        head_num (int): Number of attention heads.
        d_model (int): Dimension of the model.
        d_ff (int): Dimension of the feed-forward network.
        dropout (float, optional): Dropout probability. Defaults to 0.1.
    """
    def __init__(self, src_vocab_size, tgt_vocab_size, layer_num, head_num, d_mo
del, d_ff, dropout=0.1):
        super().__init__()
        self.encoder_embedding = nn.Embedding(src_vocab_size, d_model)
        self.decoder_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, dropout)
        self.encoder = Encoder(layer_num, head_num, d_model, d_ff, dropout)
        self.decoder = Decoder(layer_num, head_num, d_model, d_ff, tgt_vocab_siz
e, dropout)

    def make_src_mask(self, src):
        """
        Creates a mask for source input to ignore padding tokens.

        Args:
            src (torch.Tensor): Source input tensor (B, Ls).

        Returns:
            torch.Tensor: Mask tensor (B, 1, 1, Ls).
        """
        # Create mask that is 1 where src is NOT padding (e.g., <pad> token inde
x is 0)
        src_mask = (src != 0).unsqueeze(1).unsqueeze(2) # (B, 1, 1, Ls)
        return src_mask

    def make_tgt_mask(self, tgt):
        """
        Creates a look-ahead mask for target input in the decoder.

        Args:
            tgt (torch.Tensor): Target input tensor (B, Lt).

        Returns:
            torch.Tensor: Mask tensor (B, 1, Lt, Lt).
        """
        tgt_pad_mask = (tgt != 0).unsqueeze(1).unsqueeze(2) # (B, 1, 1, Lt)
```

```
        tgt_len = tgt.size(1)
        tgt_look_ahead_mask = torch.tril(torch.ones((tgt_len, tgt_len))).unsquee
ze(0).unsqueeze(1).bool() # (1, 1, Lt, Lt), lower triangular matrix
        tgt_mask = tgt_pad_mask & tgt_look_ahead_mask # Combine padding and look
-ahead masks
        return tgt_mask

    def forward(self, src, tgt):
        """
        Forward pass for Transformer model.

        Args:
            src (torch.Tensor): Source input tensor (B, Ls).
            tgt (torch.Tensor): Target input tensor (B, Lt).

        Returns:
            tuple[torch.Tensor, list[torch.Tensor], list[torch.Tensor], list[tor
ch.Tensor], list[torch.Tensor]]:
                Output tensor (B, Lt, VocabSize),
                Encoder attention weights list,
                Decoder masked attention weights list,
                Decoder encoder-decoder attention weights list.
        """
        src_mask = self.make_src_mask(src)
        tgt_mask = self.make_tgt_mask(tgt)

        # 1. Encoder embedding and positional encoding
        encoder_input = self.positional_encoding(self.encoder_embedding(src)) #
(B, Ls, Dmodel)

        # 2. Encoder forward pass
        encoder_output, encoder_attention_weights = self.encoder(encoder_input,
src_mask) # (B, Ls, Dmodel), list of attn weights

        # 3. Decoder embedding and positional encoding
        decoder_input = self.positional_encoding(self.decoder_embedding(tgt)) #
(B, Lt, Dmodel)

        # 4. Decoder forward pass
        decoder_output, decoder_masked_attention_weights, decoder_enc_dec_attent
ion_weights = self.decoder(decoder_input, encoder_output, src_mask, tgt_mask) #
(B, Lt, VocabSize), lists of attn weights

        return decoder_output, encoder_attention_weights, decoder_masked_attenti
on_weights, decoder_enc_dec_attention_weights


# --- Example Usage ---
if __name__ == '__main__':
    # Model parameters
    src_vocab_size = 10000  # Example vocabulary size for source language
    tgt_vocab_size = 10000  # Example vocabulary size for target language
    layer_num = 6
    head_num = 8
    d_model = 512
    d_ff = 2048
    dropout = 0.1

    # Create Transformer model instance
    transformer = Transformer(src_vocab_size, tgt_vocab_size, layer_num, head_nu
m, d_model, d_ff, dropout)
```

```
    # Example input tensors (Batch size = 2, Sequence length = 32)
    batch_size = 2
    seq_len = 32
    src_input = torch.randint(1, src_vocab_size, (batch_size, seq_len)) # Exampl
e source indices (excluding padding index 0)
    tgt_input = torch.randint(1, tgt_vocab_size, (batch_size, seq_len)) # Exampl
e target indices

    # Forward pass
    output, encoder_attn_weights, decoder_masked_attn_weights, decoder_enc_dec_a
ttn_weights = transformer(src_input, tgt_input)

    # Output shape (B, Lt, VocabSize)
    print("Decoder Output Shape:", output.shape) # Should be [batch_size, seq_le
n, tgt_vocab_size]

    # Example of accessing attention weights from the first encoder layer, first
 head, first batch sample
    example_encoder_attn = encoder_attn_weights[0][0][0] # Layer 0, Head 0, Batc
h 0
    print("Example Encoder Attention Weights Shape:", example_encoder_attn.shape
) # Should be [seq_len, seq_len]
```