Memory updated

Transformers - A Deep Dive

1. Why Transformers?

Problem With Traditional Sequence Models

Before Transformers, sequence models were dominated by **Recurrent Neural Networks (RNNs)** and **Long Short-Term Memory (LSTMs)**.
However, they had **major drawbacks**:

- ❌ **Sequential Computation**: Each word had to be processed one by one → **Slow training**
- ❌ **Long-Term Dependency Issue**: Information from earlier words **gets diluted**
- ❌ **Gradient Vanishing Problem**: Gradients disappear as sequences get longer

💡 **Transformers solved these issues using:** ✅ **Self-Attention**: Looks at all words **simultaneously** instead of sequentially
✅ **Positional Encoding**: Keeps track of word order without recurrence
✅ **Parallel Processing**: Training is **much faster** than RNNs

2. When to Use & Avoid Transformers

✅ **Use Transformers When**:

- You need **long-range dependencies** (e.g., translating entire paragraphs).
- You require **parallel computation** for faster training.
- You need **state-of-the-art NLP performance** (ChatGPT, BERT, GPT, etc.).

❌ **Avoid Transformers When**:

- Your dataset is **small** → Transformers have **millions of parameters**, require lots of data.
- You need **low-latency, lightweight models** (e.g., on mobile devices, CNNs might be better).
- You have **limited computational power** → Training Transformers is GPU-intensive.

3. Transformer Architecture - Step-by-Step

A Transformer consists of **two main parts**:

- **Encoder** (Processes input text)
- **Decoder** (Generates output text)

Each of these has **stacked identical layers** with three main components:

1. **Multi-Head Self-Attention**
2. **Feedforward Neural Network (FFN)**
3. **Layer Normalization & Residual Connections**

Input Text → Token Embeddings → Positional Encoding → Encoder Blocks (Self-Attention + FFN) → Decoder Blocks (Self-Attention + Cross-Attention + FFN) → Output Text

4. Transformer Pseudo Code

plaintext Copy Edit

```
# Step 1: Tokenization & Embedding
1. Convert input text into tokens.
2. Map tokens to dense embeddings (word vectors).
3. Add positional encoding to embeddings.

# Step 2: Encoding Process
FOR each encoder block:
    1. Compute Self-Attention for all words.
    2. Apply Layer Normalization.
    3. Pass result through Feedforward Neural Network.
    4. Apply another Layer Normalization.

# Step 3: Decoding Process
FOR each decoder block:
    1. Compute Masked Self-Attention for decoder input.
    2. Compute Cross-Attention between encoder & decoder outputs.
    3. Apply Layer Normalization & Feedforward Neural Network.

# Step 4: Output Generation
1. Convert decoder output into probabilities using Softmax.
2. Pick the most probable next token.
3. Repeat until the sentence is complete.
```

1. Pseudo Code for Training & Testing a Transformer

plaintext Copy Edit

```
# Step 1: Initialize Transformer Model
1. Define model hyperparameters (d_model, num_heads, learning rate, etc.).
2. Initialize weight matrices for:
    - Multi-Head Attention (Q, K, V, Output)
    - Feedforward Network (W1, W2)
    - Output Projection (Final Layer)

# Step 2: Training Loop
FOR each epoch:
    1. Forward Propagation:
        a. Apply positional encoding to input sequences.
        b. Pass input through the Encoder:
            - Compute Self-Attention
            - Apply Layer Normalization
            - Pass through Feedforward Network
        c. Pass decoder input through the Decoder:
            - Compute Masked Self-Attention
            - Compute Cross-Attention between encoder & decoder
            - Apply Layer Normalization
            - Pass through Feedforward Network
        d. Compute final output with softmax activation.

    2. Compute Loss:
        - Use Cross-Entropy Loss for sequence generation.

    3. Backpropagation:
        a. Compute gradients of loss w.r.t. output layer.
        b. Compute gradients w.r.t. decoder (Cross-Attention, Self-Attention, FFN).
        c. Compute gradients w.r.t. encoder (Self-Attention, FFN).
        d. Compute gradients w.r.t. embeddings.
```

```
    4. Update Weights:
        a. Apply Gradient Descent / Adam optimizer to:
            - Attention weights (Wq, Wk, Wv, Wo)
            - Feedforward network weights (W1, W2)
            - Output projection weights

    5. Print Loss & Track Progress.

# Step 3: Testing / Inference
1. Take new input sequence.
2. Pass through Encoder to obtain hidden state.
3. Generate first token using Decoder.
4. Loop until output sequence is complete:
    - Use previous output token as new decoder input.
    - Compute Decoder output.
    - Convert output probability to next token.
    - Append to output sequence.
5. Return generated sequence.
```

---

2. Breakdown of Key Steps in Pseudo Code

✅ **Positional Encoding** → Injects order information into word embeddings.
✅ **Multi-Head Attention** → Computes relationships between all tokens.
✅ **Feedforward Layers** → Applies transformations to encoder & decoder representations.
✅ **Backpropagation** → Computes gradients and updates weights for learning.
✅ **Inference** → Uses trained weights to generate text sequences.

Transformers: Intuitive Explanation of Every Step

Transformers are **deep learning models designed for sequence processing**, particularly excelling in **natural language processing (NLP)** tasks like translation, text generation, and summarization.

---

1. Step-by-Step Breakdown of the Transformer

The Transformer model follows these steps:

1. **Tokenization & Embedding** – Converting words into numerical representations.
2. **Positional Encoding** – Adding sequence order information.
3. **Multi-Head Self-Attention** – Understanding relationships between words.
4. **Feedforward Neural Networks (FFN)** – Processing contextualized representations.
5. **Layer Normalization & Residual Connections** – Stabilizing training.
6. **Encoder-Decoder Architecture** – Handling input and output sequences.

Let's go through each of these **intuitively**.

---

2. Tokenization & Embedding: Converting Text to Vectors

What Problem Are We Solving?

Neural networks don't understand raw text, so we must **convert words into numerical values**.

How Does Tokenization Work?

Tokenization breaks text into **smaller meaningful units**:

- **Word-Level Tokenization** → "I love transformers" → `[I, love, transformers]`
- **Subword Tokenization (BPE, WordPiece, SentencePiece)** → Breaks words into meaningful parts:
  - `"unhappiness"` → `["un", "happiness"]`
  - `"unreal"` → `["un", "real"]`
- **Character-Level Tokenization** → `"hello"` → `[h, e, l, l, o]`

Converting Tokens to Vectors

Once tokenized, each token is **mapped to an embedding vector**, a high-dimensional numerical representation of words. This is done using an **embedding matrix**.

For example, a token ID (from tokenization) is mapped to a **dense vector**:

$$\text{Embedding Matrix} \times \text{Token ID} = \text{Word Vector}$$

If our **embedding matrix** is of size (V, d_{model}) (where V is vocabulary size and d_{model} is the embedding size), each token gets a d_{model}**-dimensional vector**.

---

3. Positional Encoding: Adding Order Information

Why Do We Need This?

- Unlike **RNNs**, transformers **don't process sequences in order**.
- Positional encoding adds **positional information** to embeddings.

How Does It Work?

We add a **unique encoding** to each word position:

$$PE_{\text{pos},2i} = \sin(\text{pos}/10000^{2i/d_{\text{model}}})$$

$$PE_{\text{pos},2i+1} = \cos(\text{pos}/10000^{2i/d_{\text{model}}})$$

- This generates unique values for each position **without learning parameters**.
- Helps the model understand **word order**.

4. Attention Mechanism: Finding Important Words

Why Do We Need This?

- In a sentence like **"The cat sat on the mat"**, the model should focus on **"cat"** when predicting **"sat"**.
- Attention assigns a **weight** to each word based on its importance.

How Does Attention Work?

Each word in the input is **compared with every other word** to determine its importance.

1. **Create Query, Key, and Value Vectors**:
    - Each word is converted into **3 different vectors**:
        - **Query (Q)** → What am I looking for?
        - **Key (K)** → How relevant is this word?
        - **Value (V)** → The actual meaning of the word.

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

2. **Compute Attention Scores**:

   - We take the **dot product** between the Query and Key to measure similarity:

$$\text{Score}(Q_i, K_j) = Q_i \cdot K_j$$

- **Softmax** converts scores into probabilities:

$$\alpha_{ij} = \frac{\exp(Q_i \cdot K_j)}{\sum_k \exp(Q_i \cdot K_k)}$$

3. **Apply Attention Weights to Values**:

$$\text{Output} = \sum_j \alpha_{ij} V_j$$

The result is a **weighted combination** of words, **focusing on the most relevant words**.

---

5. Single-Head vs. Multi-Head Attention

Single-Head Attention:

- Uses **one** attention mechanism.
- **Limitation:** Focuses on **one relationship at a time**.

Multi-Head Attention:

- Uses **multiple attention heads in parallel**.
- Each head **focuses on different aspects of meaning** (e.g., syntax vs. semantics).
- The outputs of all heads are **concatenated and transformed**.

Mathematically, for h heads:

$$\text{Multi-Head Output} = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W_O$$

This makes attention **more robust**.

6. Feedforward Network (FFN)

Each word, after attention, is passed through a **fully connected neural network**:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

This **processes** and **enhances** word representations.

---

7. Layer Normalization & Residual Connections

- **Residual Connections** → Helps gradient flow, making training stable:

$$X + \text{Attention Output}$$

$$X + \text{FFN Output}$$

- **Layer Normalization** → Stabilizes activations and speeds up training:

$$\frac{X - \mu}{\sigma}$$

---

8. Encoder-Decoder Architecture

Encoder:

- Processes input and creates a **context representation**.

Decoder:

- Generates **new tokens step by step**, using encoder outputs as context.

Masked Attention in the Decoder

- Future words are **masked** (hidden) so the model doesn't cheat:

$$\text{Mask: Only attend to past words}$$

---

6. Fully Commented Manual Implementation (From Scratch)

```python
import numpy as np

class Transformer:
    """
    A simple Transformer with Encoder, Decoder, Training, and Testing.
    """
    def __init__(self, d_model=512, num_heads=8, seq_length=10, learning_rate=0.001):
        """
        Initialize Transformer parameters.
        """
        self.d_model = d_model
        self.num_heads = num_heads
        self.seq_length = seq_length
        self.learning_rate = learning_rate

        # Initialize weights for multi-head attention
        self.Wq = np.random.randn(d_model, d_model) / np.sqrt(d_model)
        self.Wk = np.random.randn(d_model, d_model) / np.sqrt(d_model)
        self.Wv = np.random.randn(d_model, d_model) / np.sqrt(d_model)
        self.Wo = np.random.randn(d_model, d_model) / np.sqrt(d_model)

        # Feedforward network weights
        self.W1 = np.random.randn(d_model, d_model * 4) / np.sqrt(d_model)
        self.W2 = np.random.randn(d_model * 4, d_model) / np.sqrt(d_model)

        # Output projection weights
        self.W_out = np.random.randn(d_model, d_model) / np.sqrt(d_model)

        # Positional encoding
        self.pos_encoding = self.positional_encoding(seq_length, d_model)

    def positional_encoding(self, seq_length, d_model):
        """Generate sinusoidal positional encodings."""
        PE = np.zeros((seq_length, d_model))
        for pos in range(seq_length):
            for i in range(0, d_model, 2):
                PE[pos, i] = np.sin(pos / (10000 ** (i / d_model)))
                if i + 1 < d_model:
                    PE[pos, i + 1] = np.cos(pos / (10000 ** (i / d_model)))
        return PE

    def softmax(self, x):
        """Compute softmax over the last axis for attention weights."""
        exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

    def self_attention(self, X):
        """Compute self-attention mechanism."""
        Q = np.dot(X, self.Wq)
        K = np.dot(X, self.Wk)
        V = np.dot(X, self.Wv)

```

```
        scores = np.dot(Q, K.T) / np.sqrt(self.d_model)
        attention_weights = self.softmax(scores)
        return np.dot(attention_weights, V)

    def multi_head_attention(self, X):
        """Multi-head self-attention implementation."""
        outputs = []
        for _ in range(self.num_heads):
            outputs.append(self.self_attention(X))
        multi_head_output = np.concatenate(outputs, axis=-1)
        return np.dot(multi_head_output, self.Wo)

    def feedforward(self, X):
        """Apply a 2-layer feedforward network."""
        return np.maximum(0, np.dot(X, self.W1)) @ self.W2

    def encoder_block(self, X):
        """Encoder block containing self-attention and feedforward layers."""
        X += self.pos_encoding[:X.shape[0], :]
        X = X + self.multi_head_attention(X)
        return X + self.feedforward(X)

    def decoder_block(self, X, encoder_output):
        """Decoder block containing masked self-attention, cross-attention, and feedforw
        X += self.pos_encoding[:X.shape[0], :]
        X = X + self.multi_head_attention(X)
        X = X + self.multi_head_attention(encoder_output)
        return X + self.feedforward(X)

    def forward(self, encoder_input, decoder_input):
        """Forward pass through the full Transformer model."""
        encoder_output = self.encoder_block(encoder_input)
        return self.decoder_block(decoder_input, encoder_output)

    def compute_loss(self, predicted, target):
        """Compute cross-entropy loss."""
        return -np.sum(target * np.log(predicted + 1e-9)) / target.shape[0]
```

```python

    def backward(self, encoder_input, decoder_input, target):
        """Backpropagation through the Transformer."""
        output = self.forward(encoder_input, decoder_input)
        loss = self.compute_loss(output, target)
        dLoss = output - target  # Gradient of loss w.r.t. output

        # Backpropagate through layers
        dW_out = np.dot(dLoss.T, self.W_out)
        self.W_out -= self.learning_rate * dW_out.T

        # Update attention and feedforward network weights
        self.Wq -= self.learning_rate * np.dot(dLoss.T, self.Wq)
        self.Wk -= self.learning_rate * np.dot(dLoss.T, self.Wk)
        self.Wv -= self.learning_rate * np.dot(dLoss.T, self.Wv)
        self.Wo -= self.learning_rate * np.dot(dLoss.T, self.Wo)
        self.W1 -= self.learning_rate * np.dot(dLoss.T, self.W1)
        self.W2 -= self.learning_rate * np.dot(dLoss.T, self.W2)

        return loss

    def train(self, encoder_input, decoder_input, target, epochs=10):
        """Training loop for the Transformer."""
        for epoch in range(epochs):
            loss = self.backward(encoder_input, decoder_input, target)
            if epoch % 1 == 0:
                print(f"Epoch {epoch+1}, Loss: {loss:.4f}")

    def test(self, encoder_input, decoder_input):
        """Testing the Transformer on unseen data."""
        return self.forward(encoder_input, decoder_input)

```

```
!pip install datasets
```

```
        )

    def forward(self, x):
        return self.layers(x)

class TransformerBlock(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()
        # TODO: initialize the multihead self attention, feed forward layer, and
        # self.attention = ...
        # self.feed_forward = ...
        # self.norm1 = ...
        # self.norm2 = ...
        self.attention = MultiHeadSelfAttention(num_heads, emb_dim // num_heads)
        self.feed_forward = FeedForwardLayer(emb_dim)
        self.norm1 = nn.LayerNorm(emb_dim)
        self.norm2 = nn.LayerNorm(emb_dim)

    def forward(self, x):
        # TODO: implement the forward logic
        # x = ...
        # x = ...
        x = x + self.attention(self.norm1(x))
        x = x + self.feed_forward(self.norm2(x))
        return x

class TransformerLanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        # self.token_embeddings = ...
        # self.position_embeddings = ...
        self.token_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.position_embeddings = nn.Embedding(context_length, embedding_dim)
        self.transformer_blocks = nn.Sequential(*[TransformerBlock(embedding_dim,
        self.final_norm = nn.LayerNorm(embedding_dim)
        self.head = nn.Linear(embedding_dim, vocab_size)

    def forward(self, idx, targets=None):
        # B, T = ...
        # tok_emb = ...
        B, T = idx.shape
        tok_emb = self.token_embeddings(idx)
        pos_emb = self.position_embeddings(torch.arange(T, device=device))
        x = tok_emb + pos_emb
        x = self.final_norm(self.transformer_blocks(x))
```

```
        # self.query_proj = ...
        # self.value_proj = ...
        self.key_proj = nn.Linear(embedding_dim, head_dim, bias=False)
        self.query_proj = nn.Linear(embedding_dim, head_dim, bias=False)
        self.value_proj = nn.Linear(embedding_dim, head_dim, bias=False)
        self.register_buffer('mask', torch.tril(torch.ones(context_length, contex
        self.dropout = nn.Dropout(drop_prob)

    def forward(self, x):
        # B, T, C = ...
        B, T, C = x.shape
        # keys, queries, values = ...
        keys = self.key_proj(x)
        queries = self.query_proj(x)
        values = self.value_proj(x)
        scores = (queries @ keys.transpose(-2, -1)) * (C ** -0.5)
        scores = scores.masked_fill(self.mask[:T, :T] == 0, float('-inf'))
        # TODO: apply softmax and dropout
        # attention_weights = ...
        # attention_weights = ...
        attention_weights = F.softmax(scores, dim=-1)
        attention_weights = self.dropout(attention_weights)
        return attention_weights @ values

class MultiHeadSelfAttention(nn.Module):
    def __init__(self, num_heads, head_dim):
        super().__init__()
        self.heads = nn.ModuleList([SelfAttention(head_dim) for _ in range(num_he
        self.output_proj = nn.Linear(embedding_dim, embedding_dim)
        self.dropout = nn.Dropout(drop_prob)

    def forward(self, x):
        # TODO: combine multiple attention heads
        # x = ...
        x = torch.cat([h(x) for h in self.heads], dim=-1)
        return self.dropout(self.output_proj(x))

class FeedForwardLayer(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(emb_dim, 4 * emb_dim),
            nn.ReLU(),
            nn.Linear(4 * emb_dim, emb_dim),
            nn.Dropout(drop_prob)
```

```
torch.manual_seed(1337)
with open('input.txt', 'r', encoding='utf-8') as file:
    text_data = file.read()

unique_chars = sorted(set(text_data))
vocab_size = len(unique_chars)
char_to_index = {ch: i for i, ch in enumerate(unique_chars)}
index_to_char = {i: ch for i, ch in enumerate(unique_chars)}

def encode_text(s): return [char_to_index[c] for c in s]
def decode_text(l): return ''.join([index_to_char[i] for i in l])

# Split data for training and validation
data_tensor = torch.tensor(encode_text(text_data), dtype=torch.long)
train_size = int(0.9 * len(data_tensor))
train_data, val_data = data_tensor[:train_size], data_tensor[train_size:]

def generate_batch(split):
    data_src = train_data if split == 'train' else val_data
    indices = torch.randint(0, len(data_src) - context_length, (batch_sz,))
    inputs = torch.stack([data_src[i:i + context_length] for i in indices])
    targets = torch.stack([data_src[i + 1:i + context_length + 1] for i in indice
    return inputs.to(device), targets.to(device)

@torch.no_grad()
def evaluate_loss():
    model.eval()
    losses = {'train': [], 'val': []}
    for split in ['train', 'val']:
        for _ in range(eval_steps):
            batch_x, batch_y = generate_batch(split)
            _, batch_loss = model(batch_x, batch_y)
            losses[split].append(batch_loss.item())
    model.train()
    return {split: torch.tensor(losses[split]).mean().item() for split in losses}


# YOU WILL CHANGE CODE IN THIS CELL
# Implement a Transformer model with PyTorch. Fill out the provided skeleton.

class SelfAttention(nn.Module):
    def __init__(self, head_dim):
        super().__init__()
        # TODO: initialize key, query, and value as linear layers. Set bias=False
        # self.key_proj = ...
```

```
!pip install tqdm
```

```
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages
```

```
!wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakesp
```

```
--2025-02-20 05:37:18--  https://raw.githubusercontent.com/karpathy/char-rnn/
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.10
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.1
HTTP request sent, awaiting response... 200 OK
Length: 1115394 (1.1M) [text/plain]
Saving to: 'input.txt'

input.txt           100%[===================>]   1.06M  --.-KB/s    in 0.009s

2025-02-20 05:37:18 (115 MB/s) - 'input.txt' saved [1115394/1115394]
```

```
# DO NOT MODIFY ANY OF THIS CODE

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim.lr_scheduler import ReduceLROnPlateau
from tqdm import tqdm


# DO NOT MODIFY ANY OF THIS CODE

# Hyperparameters
batch_sz = 16
context_length = 32
max_iterations = 30000
log_interval = 200
init_lr = 1e-3
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_steps = 200
embedding_dim = 64
num_heads = 4
num_blocks = 4
drop_prob = 0.0

# Load and prepare the data
```

```
        logits = self.head(x)

        if targets is None:
            return logits, None
        else:
          logits = logits.view(B * T, vocab_size)
          # targets = ...
          # loss = ...
          targets = targets.view(B*T)
          loss = F.cross_entropy(logits, targets)
        return logits, loss

    def generate_text(self, idx, max_tokens):
        for _ in range(max_tokens):
            idx_cond = idx[:, -context_length:]
            logits, _ = self(idx_cond)
            probs = F.softmax(logits[:, -1, :], dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, next_token), dim=1)
        return idx


# DO NOT MODIFY ANY OF THIS CODE

# Initialize and train the model
model = TransformerLanguageModel().to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=init_lr)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=5, verbos
best_val_loss = float('inf')
no_progress = 0
max_patience = 10

for step in tqdm(range(max_iterations)):
    if step % log_interval == 0 or step == max_iterations - 1:
        current_losses = evaluate_loss()
        print(f"Step {step}: train loss {current_losses['train']:.4f}, val loss {cu
        scheduler.step(current_losses['val'])

        if current_losses['val'] < best_val_loss:
            best_val_loss = current_losses['val']
            no_progress = 0
        else:
            no_progress += 1

    batch_x, batch_y = generate_batch('train')
```

```
    _, batch_loss = model(batch_x, batch_y)
    optimizer.zero_grad()
    batch_loss.backward()
    optimizer.step()

start_context = torch.zeros((1, 1), dtype=torch.long, device=device)
generated_output = decode_text(model.generate_text(start_context, max_tokens=1000))

with open("output.txt", "w", encoding="utf-8") as out_file:
    out_file.write(generated_output)
```

```
# DO NOT MODIFY ANY OF THIS CODE

# Generate from the model
with torch.no_grad():
    context = torch.tensor(encode_text("JULIET: "), dtype=torch.long).unsqueeze(0).
    generated_text = decode_text(model.generate_text(context, max_tokens=200)[0].tc
    print(generated_text)
```

```
JULIET: who liege I are none dear litts and things
True it fearful rohor defines, not tell of men that seember'd my libert.
Your sent say--tarrant be tapp'd but good
A wife! 'Tis are I we may be a happy liege
```

4. Scikit-Learn-Compatible Implementation

Since **Scikit-Learn does not have built-in Transformers**, the best alternative is **Hugging Face's** `transformers` **library** or PyTorch's built-in Transformer API.

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define Transformer model using PyTorch's built-in Transformer module
class TransformerModel(nn.Module):
    def __init__(self, input_dim, output_dim, d_model=512, num_heads=8, num_layers=6):
        super(TransformerModel, self).__init__()
        self.encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=num_heads)
        self.encoder = nn.TransformerEncoder(self.encoder_layer, num_layers=num_layers)
        self.decoder_layer = nn.TransformerDecoderLayer(d_model=d_model, nhead=num_heads)
        self.decoder = nn.TransformerDecoder(self.decoder_layer, num_layers=num_layers)
        self.fc = nn.Linear(d_model, output_dim)

    def forward(self, src, tgt):
        memory = self.encoder(src)
        output = self.decoder(tgt, memory)
        return self.fc(output)

# Initialize Transformer
model = TransformerModel(input_dim=512, output_dim=512)

# Dummy input
src = torch.rand(10, 32, 512)  # (seq_length, batch_size, embedding_dim)
tgt = torch.rand(10, 32, 512)

# Forward pass
output = model(src, tgt)
```

How to Calculate the Number of Parameters in a Transformer

To calculate the **total trainable parameters** in a Transformer, we must sum up the parameters from **each component**:

1. **Embedding Layer**
2. **Multi-Head Self-Attention**
3. **Feedforward Neural Network (FFN)**
4. **Positional Encoding (No Parameters)**
5. **Output Projection Layer**

---

1. Token Embedding Layer Parameters

Each token is mapped to a vector of size d_{model} using an **embedding matrix**.

- **Number of parameters**:

$$\text{Embedding Params} = V \times d_{\text{model}}$$

  where:

  - V = vocabulary size
  - d_{model} = embedding dimension

✅ **Example**:
For **vocab size = 30,000**, **d_model = 512**:

$$30,000 \times 512 = 15,360,000 \text{ parameters}$$

---

2. Multi-Head Self-Attention Parameters

Each self-attention layer contains:

1. **Query, Key, and Value Matrices**
2. **Output Projection Matrix**

Each matrix has dimensions:

- $W_Q, W_K, W_V \rightarrow d_{\text{model}} \times d_{\text{model}}$
- $W_O \rightarrow d_{\text{model}} \times d_{\text{model}}$

The total parameters for a **single** attention head:

$$(W_Q + W_K + W_V + W_O) = 4 \times (d_{\text{model}} \times d_{\text{model}})$$

For **multi-head attention** with h heads, we multiply by h:

$$\text{Multi-Head Attention Params} = 4 \times d_{\text{model}} \times d_{\text{model}} \times h$$

✅ **Example**:
For $d_{\text{model}} = 512$, $h = 8$:

$$4 \times 512 \times 512 = 1,048,576 \text{ parameters per layer}$$

3. Feedforward Neural Network (FFN) Parameters

Each encoder/decoder block has a **feedforward network (FFN)**, typically with:

- First layer: $d_{\text{model}} \rightarrow 4 \times d_{\text{model}}$
- Second layer: $4 \times d_{\text{model}} \rightarrow d_{\text{model}}$

Total FFN parameters:

$$(W_1 + b_1) + (W_2 + b_2) = (d_{\text{model}} \times 4d_{\text{model}} + 4d_{\text{model}}) + (4d_{\text{model}} \times d_{\text{model}} + d_{\text{model}})$$

Simplifying:

$$\text{FFN Params} = 8 \times d_{\text{model}}^2 + 5 \times d_{\text{model}}$$

✅ **Example**:
For $d_{\text{model}} = 512$:

$$8 \times 512^2 + 5 \times 512 = 2,101,760 + 2560 = 2,104,320 \text{ parameters per layer}$$

---

4. Positional Encoding Parameters

- Uses a **fixed sinusoidal function**, meaning **no learnable parameters**.
  ✅ **Total Positional Encoding Params = 0**

---

5. Output Projection Layer Parameters

- Projects the final output to vocabulary size V.

$$\text{Output Params} = d_{\text{model}} \times V$$

✅ **Example**:
For $d_{\text{model}} = 512$, $V = 30,000$:

$$512 \times 30,000 = 15,360,000$$

---

6. Total Parameters for a Full Transformer

For a Transformer with:

- L = 6 encoder layers, 6 decoder layers
- (d_{\text{model}} = 512, h = 8 (multi-head attention)
- Vocab size = 30,000

Summing Up the Parameters:

Component	Formula	Total Parameters
Embedding Layer	$V \times d_{\text{model}}$	15,360,000
Multi-Head Self-Attention (Encoder + Decoder)	$2 \times L \times 4 \times d_{\text{model}}^2$	12,582,912
Feedforward Networks (Encoder + Decoder)	$2 \times L \times (8d_{\text{model}}^2 + 5d_{\text{model}})$	25,251,840
Output Projection	$V \times d_{\text{model}}$	15,360,000
Total Transformer Parameters	Sum of all	68,554,752

Summing Up the Parameters:

Component	Formula	Total Parameters
Embedding Layer	$V \times d_{\text{model}}$	15,360,000
Multi-Head Self-Attention (Encoder + Decoder)	$2 \times L \times 4 \times d_{\text{model}}^2$	12,582,912
Feedforward Networks (Encoder + Decoder)	$2 \times L \times (8d_{\text{model}}^2 + 5d_{\text{model}})$	25,251,840
Output Projection	$V \times d_{\text{model}}$	15,360,000
Total Transformer Parameters	Sum of all	68,554,752

Final Summary

✅ **Self-Attention is the biggest contributor** (next to FFN layers).
✅ **Increasing `d_model` exponentially increases parameter count**.
✅ **Using `h` heads in attention only scales linearly** (less impactful than FFN).
