# Transformer Model Implementation: A Comprehensive Guide

IIT Ropar Minor in AI

3rd April, 2025

## 1 Introduction to Transformers: A Machine Translation Case Study

Machine translation is one of the most impactful applications of the Transformer architecture. Consider the challenging task of translating English to German:

**English**: "The black cat sat on the mat." **German**: "Die schwarze Katze saß auf der Matte."

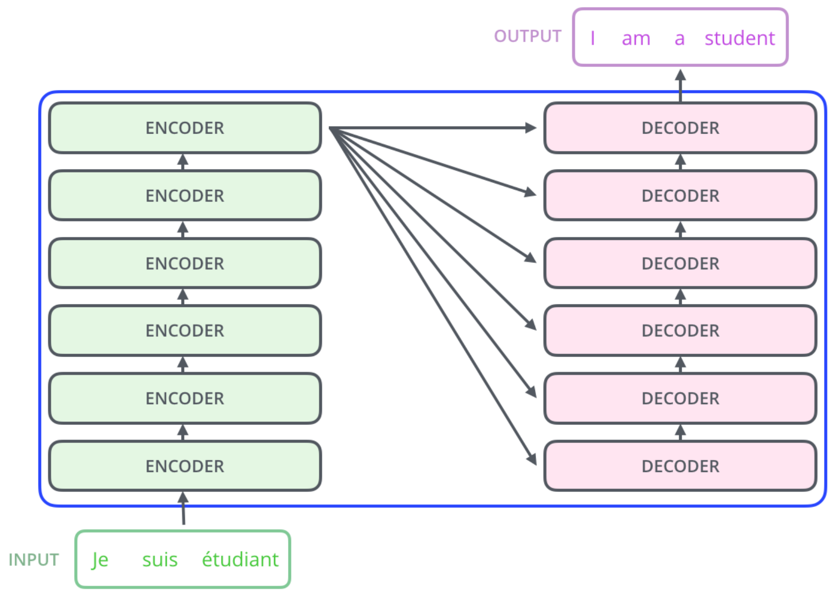
Traditional sequence-to-sequence models with recurrent neural networks (RNNs) struggled with:

- Capturing long-range dependencies
- Parallelization during training
- Maintaining context across sentences

The Transformer architecture, introduced in the seminal "Attention Is All You Need" paper (Vaswani et al., 2017), addressed these limitations through its self-attention mechanism, enabling:

- Direct modeling of dependencies between any two positions in a sequence
- Highly parallelizable computation
- Superior performance on translation tasks

For our example sentence, a Transformer processes the entire input sequence simultaneously rather than word-by-word, attending to relevant context across the entire sentence to produce accurate translations.




Transformer Model (Image Credit: https://jalammar.github.io/illustrated-transformer/)

# 2 Theoretical Foundations

## 2.1 Positional Encoding

**Problem**: Unlike recurrent or convolutional networks, the self-attention mechanism in Transformers is permutation invariant - it doesn't inherently understand the order of tokens.

**Solution**: Positional encodings inject information about token positions directly into the embeddings.

**Mathematical Formulation**:

- For position $pos$ and dimension $i$ in the embedding:
  - For even dimensions: $PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$
  - For odd dimensions: $PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$

These sinusoidal functions create unique positional signatures that:

- Allow the model to attend to relative positions
- Extrapolate to sequence lengths not seen during training
- Are added directly to token embeddings before entering the encoder/decoder stack

## 2.2 Transformer Encoder

The encoder transforms an input sequence into a continuous representation that captures its meaning while preserving positional information.

**Key Components**:

1. **Input Embedding**: Maps tokens to vectors of dimension $d_{model}$
2. **Positional Encoding**: Adds position information to embeddings
3. **Multi-Head Self-Attention**: Processes relationships between all positions in the sequence
4. **Feed-Forward Network**: Processes each position independently with identical parameters
5. **Layer Normalization and Residual Connections**: Stabilize and accelerate training

The encoder creates contextualized representations where each token's embedding contains information about its meaning in relation to all other tokens.

## 2.3 Transformer Decoder

The decoder generates output tokens sequentially while attending to both:

- Previously generated output tokens (via masked self-attention)
- The encoder's representation of the input sequence (via encoder-decoder attention)

**Key Components**:

1. **Output Embedding**: Maps tokens to vectors of dimension $d_{model}$
2. **Positional Encoding**: Adds position information to embeddings
3. **Masked Multi-Head Self-Attention**: Prevents attending to future positions
4. **Multi-Head Encoder-Decoder Attention**: Attends to relevant parts of the input
5. **Feed-Forward Network**: Processes each position independently
6. **Linear and Softmax Layer**: Converts final representations to output probabilities

The decoder's auto-regressive nature enables it to generate coherent sequences while leveraging the complete context provided by the encoder.

### 2.4 Complete Transformer Model

The full Transformer architecture combines the encoder and decoder in an end-to-end system:

1. **Encoder**: Processes the entire input sequence in parallel
2. **Decoder**: Generates output tokens one by one, attending to both:
    - Previously generated tokens
    - The encoder's representation
3. **Training**: Uses teacher forcing with a shifted version of the target sequence

The Transformer excels at capturing both local and global dependencies, making it ideal for sequence transduction tasks like machine translation.

## 3 Code Implementation Analysis

### 3.1 Positional Encoding

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = 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(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
      seq_len = x.size(1)
      return x + self.pe[:, :seq_len, :]
```

**Implementation Details**:

- `d_model`: Dimension of the embedding vectors (e.g., 512)
- `max_len`: Maximum expected sequence length (5000 by default)
- The implementation uses sinusoidal functions as described in the original paper
- Positional encodings are computed once during initialization and stored as a buffer
- The `forward` method adds positional encodings to input embeddings (`x`)

- Sequence truncation is handled by taking only positional encodings up to `seq_len`

This implementation efficiently adds unique positional information to each token embedding while maintaining the same dimensionality.

## 3.2 Transformer Encoder

```python
class TransformerEncoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_heads, hidden_dim, num_layers, max_len):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.pos_encoder = PositionalEncoding(embed_dim, max_len)
        encoder_layer = nn.TransformerEncoderLayer(d_model=embed_dim, nhead=num_heads, dim_feedforward=hidden_dim, batch_first=True)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)

    def forward(self, src, src_key_padding_mask=None):
        x = self.embedding(src)
        x = self.pos_encoder(x)
        x = self.encoder(x, src_key_padding_mask=src_key_padding_mask)
        return x
```

**Implementation Details**:

- The encoder consists of:
  - A token embedding layer that maps integer indices to vectors
  - A positional encoding layer that adds position information
  - A stack of `num_layers` identical encoder layers
- Each encoder layer (created by PyTorch's `nn.TransformerEncoderLayer`) contains:
  - Multi-head self-attention mechanism
  - Position-wise feed-forward network
  - Layer normalization and residual connections
- The `batch_first=True` parameter configures input tensors to have shape `[batch_size, seq_len, embed_dim]`
- `src_key_padding_mask` indicates which positions should be ignored (padded positions)

This implementation leverages PyTorch's built-in Transformer modules while providing a clean interface for the encoder component.

### 3.3 Transformer Decoder

```python
class TransformerDecoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_heads, hidden_dim, num_layers, max_len):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.pos_encoder = PositionalEncoding(embed_dim, max_len)
        decoder_layer = nn.TransformerDecoderLayer(d_model=embed_dim, nhead=num_heads, dim_feedforward=hidden_dim, batch_first=True)
        self.decoder = nn.TransformerDecoder(decoder_layer, num_layers=num_layers)
        self.fc_out = nn.Linear(embed_dim, vocab_size)

    def forward(self, tgt, memory, tgt_mask=None, tgt_key_padding_mask=None, memory_key_padding_mask=None):
        x = self.embedding(tgt)
        x = self.pos_encoder(x)
        x = self.decoder(x, memory, tgt_mask=tgt_mask,
                         tgt_key_padding_mask=tgt_key_padding_mask,
                         memory_key_padding_mask=memory_key_padding_mask)
        return self.fc_out(x)
```

**Implementation Details**:

- Similar to the encoder, the decoder includes:
  - Token embedding layer
  - Positional encoding layer
  - A stack of `num_layers` identical decoder layers
- Each decoder layer (created by PyTorch's `nn.TransformerDecoderLayer`) contains:
  - Masked multi-head self-attention mechanism
  - Multi-head encoder-decoder attention mechanism
  - Position-wise feed-forward network
  - Layer normalization and residual connections
- Key parameters in the `forward` method:
  - `tgt`: Target tokens (already processed tokens during inference)
  - `memory`: Output from the encoder
  - `tgt_mask`: Prevents attending to future positions
  - `tgt_key_padding_mask`: Indicates padded positions in the target sequence
  - `memory_key_padding_mask`: Indicates padded positions in the encoder output

- The final linear layer maps hidden representations to vocabulary distribution

The decoder implements the auto-regressive generation process, enabling one-by-one token generation while attending to the encoder's representation.

## 3.4 Complete Transformer Model

```python
class TransformerModel(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, embed_dim, num_heads, hidden_dim, num_layers, max_len):
        super().__init__()
        self.encoder = TransformerEncoder(src_vocab_size, embed_dim, num_heads, hidden_dim, num_layers, max_len)
        self.decoder = TransformerDecoder(tgt_vocab_size, embed_dim, num_heads, hidden_dim, num_layers, max_len)

    def generate_square_subsequent_mask(self, sz):
        return torch.triu(torch.ones(sz, sz) * float('-inf'), diagonal=1)

    def forward(self, src, tgt, src_padding_mask=None, tgt_padding_mask=None):
        memory = self.encoder(src, src_key_padding_mask=src_padding_mask)
        tgt_seq_len = tgt.size(1)
        tgt_mask = self.generate_square_subsequent_mask(tgt_seq_len).to(tgt.device)
        return self.decoder(tgt, memory, tgt_mask=tgt_mask,
                            tgt_key_padding_mask=tgt_padding_mask,
                            memory_key_padding_mask=src_padding_mask)
```

**Implementation Details**:

- The complete model combines:
  - An encoder for processing the source sequence
  - A decoder for generating the target sequence
- Separate vocabulary sizes for source and target languages
- The `generate_square_subsequent_mask` method creates a causal mask where each position can only attend to prior positions and itself:
  - It creates an upper triangular matrix of `-inf` values above the diagonal
  - During softmax, these `-inf` values become zeros, effectively blocking attention to future positions
- The `forward` method:
  1. Processes the source sequence through the encoder

2. Generates a causal mask for the target sequence
3. Processes the target sequence through the decoder, attending to the encoder output
4. Returns logits that can be passed through a softmax function to get probabilities

This implementation provides a clean, modular approach to the full Transformer architecture while leveraging PyTorch's built-in modules for the core attention mechanisms.

# 4 Practical Applications and Extensions

The Transformer architecture implemented here serves as the foundation for many state-of-the-art models in NLP:

1. **Machine Translation**: The original use case, achieving breakthrough performance on language pairs
2. **Transfer Learning Models**: Pre-trained models like BERT, GPT, and T5 are based on the Transformer architecture
3. **Text Generation**: Transformers excel at generating coherent, contextually relevant text
4. **Document Summarization**: The ability to capture long-range dependencies makes Transformers ideal for summarization
5. **Question Answering**: Transformers can understand relationships between questions and context passages

Extensions to consider:

- **Beam search** for improved decoding
- **Length normalization** to avoid bias toward shorter sequences
- **Label smoothing** to improve generalization
- **Byte-Pair Encoding (BPE)** or **SentencePiece** tokenization
- **Layer-wise learning rate decay** for fine-tuning

# 5 Conclusion

The Transformer architecture represents a fundamental shift in sequence modeling. By replacing recurrence with attention, it enables more effective parallel processing and better modeling of long-range dependencies. The implementation provided here demonstrates the core components of this architecture:

positional encoding, self-attention mechanisms in both encoder and decoder, and the overall sequence-to-sequence framework.

Understanding these components not only helps in implementing Transformers from scratch but also provides insight into the inner workings of modern NLP models that build upon this architecture.