Understanding Transformer Neural Networks

Comprehensive Guide to Modern Attention-Based Architectures

April 4, 2025

Abstract

This document provides a comprehensive explanation of transformer neural networks, focusing on their architecture, key components, and operational principles. We examine the attention mechanism that forms the core of these models and explain how transformers process sequential data without recurrence. Using a simple PyTorch implementation as reference, we break down each component of the transformer architecture and explain its function in the overall model.

Contents

1 Introduction to Transformer Architecture

Transformer neural networks, introduced in the paper "Attention Is All You Need" by Vaswani et al. (2017), represent a paradigm shift in sequence processing. Unlike previous architectures such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory networks (LSTMs), transformers do not process data sequentially. Instead, they employ a self-attention mechanism that allows them to consider the entire input sequence simultaneously, enabling more efficient parallel processing and better modeling of long-range dependencies.

The transformer architecture has become the foundation for many state-of-the-art models in natural language processing (NLP), including BERT, GPT, T5, and others. These models have achieved remarkable results in various tasks such as machine translation, text summarization, question answering, and text generation.

2 High-Level Overview of Transformer Architecture

At a high level, a transformer consists of two main components:

1. **Encoder**: Processes the input sequence and generates a continuous representation that captures the contextual information.
2. **Decoder**: Takes the encoder's output and generates the target sequence, typically one token at a time.

Both the encoder and decoder are composed of multiple identical layers stacked on top of each other. Each layer contains two main sub-layers:

1. **Multi-Head Attention**: Allows the model to focus on different parts of the input sequence.
2. **Position-wise Feed-Forward Network**: Applies the same feed-forward transformation to each position independently.

Additionally, the transformer employs residual connections around each sub-layer, followed by layer normalization. This design helps with training deeper networks by allowing gradients to flow more easily through the network.

3 Key Components of Transformer Architecture

3.1 Embedding and Positional Encoding

In a transformer, input tokens (words or subwords) are first converted to continuous vector representations through an embedding layer. However, unlike

RNNs, transformers process all tokens in parallel and thus lose the inherent sequential information. To compensate for this, positional encodings are added to the embeddings to provide information about the position of each token in the sequence.

The positional encoding used in the original transformer paper is defined as:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{1}$$

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

where pos is the position of the token in the sequence, i is the dimension, and d_{model} is the embedding dimension. This creates a unique pattern for each position, allowing the model to distinguish between different positions.

In our implementation, the positional encoding is created as follows:

```
# Create positional encoding matrix
pe = torch.zeros(max_seq_length, d_model)
position = torch.arange(0, max_seq_length, dtype=torch.float).
    unsqueeze(1)
div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.
    log(10000.0) / d_model))

# Apply sine to even indices and cosine to odd indices
pe[:, 0::2] = torch.sin(position * div_term)
pe[:, 1::2] = torch.cos(position * div_term)
```

Listing 1: Positional Encoding Implementation

3.2 Self-Attention Mechanism

The self-attention mechanism is the core innovation of the transformer architecture. It allows the model to weigh the importance of different tokens in the input sequence when processing a specific token. This is particularly useful for capturing long-range dependencies in the data.

The self-attention mechanism operates on three projections of the input: Queries (Q), Keys (K), and Values (V). The attention weights are computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3}$$

where d_k is the dimension of the keys. The scaling factor $\sqrt{d_k}$ is used to prevent the softmax function from entering regions with very small gradients.

In our implementation, the scaled dot-product attention is computed as:

```
# Scaled dot-product attention
scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.d_k)

# Apply mask if provided
if mask is not None:
```

```
    scores = scores.masked_fill(mask == 0, -1e9)

# Apply softmax to get attention weights
attn_weights = torch.softmax(scores, dim=-1)

# Apply attention weights to values
output = torch.matmul(attn_weights, v)
```

Listing 2: Scaled Dot-Product Attention

3.3 Multi-Head Attention

Instead of performing a single attention function, the transformer uses multi-head attention, which allows the model to jointly attend to information from different representation subspaces. This is achieved by linearly projecting the queries, keys, and values multiple times with different learned projections, performing attention on each projected version, and then concatenating the results.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{4}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{5}$$

In our implementation, multi-head attention is achieved by reshaping the projected queries, keys, and values:

```
# Linear projections and reshape for multi-head attention
q = self.q_linear(q).view(batch_size, -1, self.num_heads, self.d_k)
    .transpose(1, 2)
k = self.k_linear(k).view(batch_size, -1, self.num_heads, self.d_k)
    .transpose(1, 2)
v = self.v_linear(v).view(batch_size, -1, self.num_heads, self.d_k)
    .transpose(1, 2)

# ... (attention computation) ...

# Reshape and apply final linear projection
output = output.transpose(1, 2).contiguous().view(batch_size, -1,
    self.d_model)
output = self.out_linear(output)
```

Listing 3: Multi-Head Attention

The key line here is the reshaping operation that transforms the output back from multiple heads to the original dimension:

```
output = output.transpose(1, 2).contiguous().view(batch_size, -1,
    self.d_model)
```

This operation first transposes the dimensions to bring the sequence length back to the second dimension, ensures the tensor is contiguous in memory, and then reshapes it to combine all the attention heads.

3.4 Position-wise Feed-Forward Networks

Each layer in both the encoder and decoder contains a fully connected feed-forward network that is applied to each position separately and identically. This network consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{6}$$

In our implementation, this is represented as:

```python
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, verbose=False):
        super(FeedForward, self).__init__()
        self.verbose = verbose

        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        # First linear layer
        ff1 = self.linear1(x)

        # ReLU activation
        relu_out = self.relu(ff1)

        # Second linear layer
        output = self.linear2(relu_out)

        return output
```

Listing 4: Feed-Forward Network

3.5 Residual Connections and Layer Normalization

To facilitate training of deep networks, the transformer employs residual connections around each sub-layer, followed by layer normalization. This can be expressed as:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{7}$$

where $\text{Sublayer}(x)$ is the function implemented by the sub-layer itself.

In our implementation, this is represented in the encoder and decoder layers:

```python
# Self-attention with residual connection and layer normalization
attn_output = self.self_attn(x, x, x, mask)
x = self.norm1(x + self.dropout(attn_output))

# Feed-forward with residual connection and layer normalization
ff_output = self.feed_forward(x)
x = self.norm2(x + self.dropout(ff_output))
```

Listing 5: Residual Connections and Layer Normalization

4 Encoder-Decoder Architecture

4.1 Encoder

The encoder consists of a stack of identical layers, each containing a multi-head self-attention mechanism and a position-wise feed-forward network. The encoder processes the input sequence and generates a continuous representation that captures the contextual information.

In our implementation, the encoder is represented as a stack of EncoderLayer instances:

```
# Pass through encoder layers
enc_output = src
for i, enc_layer in enumerate(self.encoder_layers):
    enc_output = enc_layer(enc_output, src_mask)
```

Listing 6: Encoder Implementation

4.2 Decoder

The decoder also consists of a stack of identical layers, but each layer has an additional multi-head attention sub-layer that attends to the output of the encoder. The decoder generates the target sequence, typically one token at a time.

In our implementation, the decoder is represented as a stack of DecoderLayer instances:

```
# Pass through decoder layers
dec_output = tgt
for i, dec_layer in enumerate(self.decoder_layers):
    dec_output = dec_layer(dec_output, enc_output, src_mask,
    tgt_mask)
```

Listing 7: Decoder Implementation

4.3 Masking in the Decoder

During training, the decoder uses masking to prevent it from attending to future positions in the target sequence. This is necessary because the model should not be able to "see" the future tokens when predicting the current token.

In our implementation, this is achieved using a square subsequent mask:

```
def generate_square_subsequent_mask(self, sz):
    """Generate a square mask for the sequence. The masked
    positions are filled with float('-inf').
       Unmasked positions are filled with float(0.0).
    """
    mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
    mask = mask.float().masked_fill(mask == 0, float('-inf')).
    masked_fill(mask == 1, float(0.0))
    return mask
```

Listing 8: Masking in the Decoder

5 The Complete Transformer Model

Putting all the components together, the transformer model first embeds the input tokens, adds positional encoding, and then passes the resulting representations through the encoder and decoder stacks. Finally, a linear layer followed by a softmax function is used to convert the decoder output to probabilities over the target vocabulary.

In our implementation, the forward pass of the transformer is represented as:

```
def forward(self, src, tgt, src_mask=None, tgt_mask=None):
    # Embed source and target sequences and add positional encoding
    src = self.dropout(self.positional_encoding(self.src_embedding(
    src) * math.sqrt(self.d_model)))
    tgt = self.dropout(self.positional_encoding(self.tgt_embedding(
    tgt) * math.sqrt(self.d_model)))

    # Create masks if not provided
    if tgt_mask is None:
        tgt_mask = self.generate_square_subsequent_mask(tgt.size(1)
    ).to(tgt.device)

    # Pass through encoder layers
    enc_output = src
    for i, enc_layer in enumerate(self.encoder_layers):
        enc_output = enc_layer(enc_output, src_mask)

    # Pass through decoder layers
    dec_output = tgt
    for i, dec_layer in enumerate(self.decoder_layers):
        dec_output = dec_layer(dec_output, enc_output, src_mask,
    tgt_mask)

    # Final linear layer to get logits
    output = self.output_linear(dec_output)

    return output
```

Listing 9: Transformer Forward Pass

6 Training and Inference

6.1 Training

During training, the transformer is typically trained using teacher forcing, where the ground truth target tokens are provided as input to the decoder. The model is trained to maximize the likelihood of the correct next token given the previous tokens.

The loss function used is typically cross-entropy loss, which measures the difference between the predicted probability distribution and the true distribution (a one-hot encoding of the correct token).

6.2 Inference

During inference, the transformer generates the target sequence one token at a time. At each step, the model takes the previously generated tokens as input to the decoder and predicts the next token. This process continues until an end-of-sequence token is generated or a maximum length is reached.

7 Advantages of Transformer Architecture

The transformer architecture offers several advantages over previous sequence processing models:

1. **Parallelization**: Unlike RNNs, which process tokens sequentially, transformers can process all tokens in parallel, leading to more efficient training.
2. **Long-range dependencies**: The self-attention mechanism allows transformers to capture dependencies between tokens regardless of their distance in the sequence.
3. **Interpretability**: The attention weights can be visualized to understand which parts of the input the model is focusing on when making predictions.
4. **Scalability**: Transformers can be scaled to very large models with billions of parameters, leading to state-of-the-art performance on many tasks.

8 Conclusion

The transformer architecture has revolutionized the field of natural language processing and beyond. Its ability to process sequences in parallel and capture long-range dependencies has made it the foundation for many state-of-the-art models. Understanding the key components of transformers, such as self-attention, multi-head attention, and positional encoding, is essential for working with modern deep learning models.

The simple implementation discussed in this document provides a clear illustration of the transformer architecture and its components. By breaking down the model into its constituent parts, we can better understand how transformers process and generate sequential data.

9 References

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. In Advances in Neural Information Processing Systems.
2. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint arXiv:1810.04805.

3. Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training.

4. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language Models are Few-Shot Learners. arXiv preprint arXiv:2005.14165.