Transformer Architecture Documentation

A Comprehensive Guide to the Implementation in `Transformer_from_scratch.py`

Based on "Attention Is All You Need"

February 8, 2026

Abstract

This document provides a detailed explanation of the Transformer architecture implementation found in `Transformer_from_scratch.py`. The implementation follows the original "Attention Is All You Need" paper with an innovative addition: *Multi-Head Latent Attention* which compresses key-value pairs for improved efficiency. Each component is explained mathematically and conceptually, with code snippets and dimensional analysis.

Contents

1 Introduction

The Transformer architecture revolutionized natural language processing by replacing recurrent layers with *attention mechanisms*. Unlike RNNs which process sequences sequentially, Transformers process entire sequences in parallel through self-attention.

1.1 Key Innovations in This Implementation

This implementation includes two attention mechanisms:

1. **Standard Multi-Head Attention** – The classic attention from the original paper
2. **Multi-Head Latent Attention (MLA)** – A memory-efficient variant that compresses keys and values

The `build_transformer()` function uses MLA for all encoder and decoder blocks.

2 Core Components

2.1 Input Embeddings

```python
class InputEmbeddings(nn.Module):
    def __init__(self, d_model: int, vocab_size: int) -> None:
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        return self.embedding(x) * math.sqrt(self.d_model)
```

Listing 1: InputEmbeddings class

Purpose: Convert integer token IDs into dense vector representations.
Mathematical Operation:

$$\text{Embedding}(x) = \text{lookup}(x) \times \sqrt{d_{\text{model}}} \tag{1}$$

Dimensions:

- Input: `(batch, seq_len)` – integer token indices
- Output: `(batch, seq_len, d_model)` – embedding vectors

Why multiply by $\sqrt{d_{\text{model}}}$? The original paper scales embeddings to match the magnitude of positional encodings, preventing the positional information from drowning out semantic information.

2.2 Positional Encoding

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, seq_len: int, dropout: float) -> None:
        super().__init__()
        pe = torch.zeros(seq_len, d_model)
        position = torch.arange(0, seq_len).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)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        x = x + (self.pe[:, :x.shape[1], :]).requires_grad_(False)
        return self.dropout(x)
```

Listing 2: PositionalEncoding class

Purpose: Inject information about token position since attention has no inherent sense of sequence order.

Mathematical Formula:

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

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{3}$$

Key Properties:

- Unique encoding for each position
- Relative positions can be derived via linear transformations
- Values bounded between $[-1, 1]$
- `requires_grad_=False` – positions are fixed, not learned

2.3 Layer Normalization

```
class LayerNormalization(nn.Module):
    def __init__(self, features: int, eps: float = 10**-6) -> None:
        super().__init__()
        self.eps = eps
        self.alpha = nn.Parameter(torch.ones(features))
        self.bias = nn.Parameter(torch.zeros(features))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        std = x.std(dim=-1, keepdim=True)
        return self.alpha * (x - mean) / (std + self.eps) + self.bias
```

Listing 3: LayerNormalization class

Purpose: Stabilize training by normalizing activations within each layer.

Mathematical Operation:

$$\text{LayerNorm}(x) = \alpha \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{4}$$

Where:

- μ = mean across last dimension
- σ = standard deviation across last dimension
- α = learnable scale parameter
- β = learnable shift parameter
- ϵ = small constant for numerical stability

Note: This implementation uses *pre-normalization* (normalization before sublayers) via `ResidualConnection`.

2.4 Feed-Forward Network

```python
class FeedForwardBlock(nn.Module):
    def __init__(self, d_model: int, d_ff: int, dropout: float) -> None:
        super().__init__()
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        return self.linear_2(self.dropout(torch.relu(self.linear_1(x))))
```

Listing 4: FeedForwardBlock class

Purpose: Apply non-linear transformations to each position independently.
Architecture:

$$\text{FFN}(x) = W_2 \cdot \text{Dropout}(\text{ReLU}(W_1 \cdot x + b_1)) + b_2 \tag{5}$$

Dimensions:

- W_1: $(d_{\text{model}} \times d_{ff})$ – expands dimension
- W_2: $(d_{ff} \times d_{\text{model}})$ – projects back
- Typical: $d_{ff} = 4 \times d_{\text{model}}$ (e.g., 2048 for $d_{\text{model}} = 512$)

2.5 Residual Connections

```python
class ResidualConnection(nn.Module):
    def __init__(self, features: int, dropout: float) -> None:
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.norm = LayerNormalization(features)

    def forward(self, x, sublayer):
        return x + self.dropout(sublayer(self.norm(x)))
```

Listing 5: ResidualConnection class

Purpose: Enable gradient flow through deep networks and stabilize training.
Operation:

$$\text{Output} = x + \text{Dropout}(\text{Sublayer}(\text{LayerNorm}(x))) \tag{6}$$

This is **pre-normalization** (norm before sublayer), which has been shown to train more stably than post-normalization for deep Transformers.

3 Attention Mechanisms

3.1 Standard Multi-Head Attention

```python
class MultiHeadAttentionBlock(nn.Module):
    def __init__(self, d_model: int, h: int, dropout: float) -> None:
        super().__init__()
        self.d_model = d_model
        self.h = h
        assert d_model % h == 0, "d_model is not divisible by h"
        self.d_k = d_model // h

        self.w_q = nn.Linear(d_model, d_model, bias=False)
        self.w_k = nn.Linear(d_model, d_model, bias=False)
        self.w_v = nn.Linear(d_model, d_model, bias=False)
        self.w_o = nn.Linear(d_model, d_model, bias=False)
        self.dropout = nn.Dropout(dropout)
```

Listing 6: MultiHeadAttentionBlock class

Core Idea: Split the model dimension into h heads, allowing the model to attend to information from different representation subspaces.

Scaled Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{7}$$

Multi-Head Attention:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{8}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{9}$$

Dimensions:

Tensor	Shape Before	Shape After
Query	(batch, seq, d_{model})	(batch, h, seq, d_k)
Key	(batch, seq, d_{model})	(batch, h, seq, d_k)
Value	(batch, seq, d_{model})	(batch, h, seq, d_k)
Attention Scores	–	(batch, h, seq, seq)
Output	(batch, h, seq, d_k)	(batch, seq, d_{model})

Why $\sqrt{d_k}$? Prevents dot products from growing too large, which would push softmax into regions with extremely small gradients.

3.2 Multi-Head Latent Attention (Innovation)

```python
class MultiHeadLatentAttentionBlock(nn.Module):
    def __init__(self, d_model: int, h: int, dropout: float):
        super().__init__()
        self.d_model = d_model
        self.h = h
        assert d_model % h == 0, "d_model is not divisible by h"
        self.d_k = d_model // h

        assert d_model % 4 == 0, "d_model is not divisible by 4"
        self.latent_dim = d_model // 4  # Compression factor of 4

```

```
        self.w_q = nn.Linear(d_model, d_model, bias=False)
        # KEY INNOVATION: Compression + Decompression
        self.w_kv_compress = nn.Linear(d_model, self.latent_dim, bias=False)
        self.w_k_decompress = nn.Linear(self.latent_dim, d_model, bias=False)
        self.w_v_decompress = nn.Linear(self.latent_dim, d_model, bias=False)
        self.w_o = nn.Linear(d_model, d_model, bias=False)
        self.dropout = nn.Dropout(dropout)
```

Listing 7: MultiHeadLatentAttentionBlock class

Key Innovation: Instead of computing keys and values at full dimension, MLA:

1. **Compresses** input to a latent representation: $c = W_{\text{kv\_compress}} \cdot x$
2. **Decompresses** to get keys and values: $K = W_{\text{k\_decompress}} \cdot c$, $V = W_{\text{v\_decompress}} \cdot c$

Memory Savings:

- Standard: Store $K, V \in \mathbb{R}^{d_{\text{model}}}$
- MLA: Store latent $c \in \mathbb{R}^{d_{\text{model}}/4}$
- **Reduction:** 75% memory savings for KV cache!

Mathematical Flow:

$$c = W_{\text{compress}} \cdot x \quad \in \mathbb{R}^{d_{\text{model}}/4} \tag{10}$$

$$K = W_{\text{k\_decompress}} \cdot c \quad \in \mathbb{R}^{d_{\text{model}}} \tag{11}$$

$$V = W_{\text{v\_decompress}} \cdot c \quad \in \mathbb{R}^{d_{\text{model}}} \tag{12}$$

$$Q = W_q \cdot x \quad \in \mathbb{R}^{d_{\text{model}}} \tag{13}$$

Trade-offs:

✓ Reduced memory footprint

✓ Faster inference (smaller KV cache)

× Slight capacity reduction (information bottleneck)

× Additional computation for compression/decompression

4 Encoder and Decoder

4.1 Encoder Block

```python
class EncoderBlock(nn.Module):
    def __init__(self, features: int,
                 self_attention_block: MultiHeadLatentAttentionBlock,
                 feed_forward_block: FeedForwardBlock, dropout: float) -> None:
        super().__init__()
        self.self_attention_block = self_attention_block
        self.feed_forward_block = feed_forward_block
        self.residual_connections = nn.ModuleList([
            ResidualConnection(features, dropout) for _ in range(2)
        ])

    def forward(self, x, src_mask):
        x = self.residual_connections[0](x,
            lambda x: self.self_attention_block(x, x, x, src_mask))
        x = self.residual_connections[1](x, self.feed_forward_block)
        return x
```

Listing 8: EncoderBlock class

Architecture:

1. Self-Attention with residual: $x = x + \text{Attn}(\text{LN}(x))$
2. Feed-Forward with residual: $x = x + \text{FFN}(\text{LN}(x))$

Self-Attention: Each position attends to all positions in the same sequence (uses `src_mask` for padding).

4.2 Encoder Stack

```python
class Encoder(nn.Module):
    def __init__(self, features: int, layers: nn.ModuleList) -> None:
        super().__init__()
        self.layers = layers
        self.norm = LayerNormalization(features)

    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

Listing 9: Encoder class

Stacks N (default 6) `EncoderBlocks` with a final layer normalization.

4.3 Decoder Block

```python
class DecoderBlock(nn.Module):
    def __init__(self, features: int,
                 self_attention_block: MultiHeadLatentAttentionBlock,
                 cross_attention_block: MultiHeadLatentAttentionBlock,
                 feed_forward_block: FeedForwardBlock, dropout: float) -> None:
        super().__init__()
        self.self_attention_block = self_attention_block
        self.cross_attention_block = cross_attention_block
        self.feed_forward_block = feed_forward_block
        self.residual_connections = nn.ModuleList([
```

```
            ResidualConnection(features, dropout) for _ in range(3)
        ])

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        x = self.residual_connections[0](x,
            lambda x: self.self_attention_block(x, x, x, tgt_mask))
        x = self.residual_connections[1](x,
            lambda x: self.cross_attention_block(x, encoder_output,
                                                 encoder_output, src_mask))
        x = self.residual_connections[2](x, self.feed_forward_block)
        return x
```

Listing 10: DecoderBlock class

Architecture (3 sublayers):

1. **Masked Self-Attention:** Each position attends to previous positions only (causal masking)
2. **Cross-Attention:** Queries from decoder attend to Keys/Values from encoder output
3. **Feed-Forward:** Position-wise transformation

Cross-Attention:

$$\text{CrossAttn}(Q_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}}) \tag{14}$$

This is where the decoder "looks at" the source sequence representation.

4.4 Decoder Stack

```
class Decoder(nn.Module):
    def __init__(self, features: int, layers: nn.ModuleList) -> None:
        super().__init__()
        self.layers = layers
        self.norm = LayerNormalization(features)

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, encoder_output, src_mask, tgt_mask)
        return self.norm(x)
```

Listing 11: Decoder class

Stacks N `DecoderBlocks` with a final layer normalization.

5 Complete Transformer

5.1 Projection Layer

```python
class ProjectionLayer(nn.Module):
    def __init__(self, d_model, vocab_size) -> None:
        super().__init__()
        self.proj = nn.Linear(d_model, vocab_size)

    def forward(self, x) -> None:
        return self.proj(x)
```

Listing 12: ProjectionLayer class

Converts decoder output from d_{model} dimensions to vocabulary size for softmax prediction. **Output:** `(batch, seq_len, vocab_size)` – logits for each token.

5.2 Transformer Class

```python
class Transformer(nn.Module):
    def __init__(self, encoder: Encoder, decoder: Decoder,
                 src_embed: InputEmbeddings, tgt_embed: InputEmbeddings,
                 src_pos: PositionalEncoding, tgt_pos: PositionalEncoding,
                 projection_layer: ProjectionLayer) -> None:
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.src_pos = src_pos
        self.tgt_pos = tgt_pos
        self.projection_layer = projection_layer

    def encode(self, src, src_mask):
        src = self.src_embed(src)
        src = self.src_pos(src)
        return self.encoder(src, src_mask)

    def decode(self, encoder_output, src_mask, tgt, tgt_mask):
        tgt = self.tgt_embed(tgt)
        tgt = self.tgt_pos(tgt)
        return self.decoder(tgt, encoder_output, src_mask, tgt_mask)

    def project(self, x):
        return self.projection_layer(x)
```

Listing 13: Transformer class

Three Main Operations:

1. `encode(src, src_mask)`:
   - Embed + positional encode source
   - Pass through encoder stack
   - Returns: encoder representation
2. `decode(encoder_output, src_mask, tgt, tgt_mask)`:
   - Embed + positional encode target
   - Pass through decoder stack (with cross-attention)

- Returns: decoder representation

3. `project(x)`:
   - Linear projection to vocabulary
   - Returns: logits for next-token prediction

5.3 Model Construction

```python
def build_transformer(src_vocab_size: int, tgt_vocab_size: int,
                      src_seq_len: int, tgt_seq_len: int,
                      d_model: int = 512, N: int = 6, h: int = 8,
                      dropout: float = 0.1, d_ff: int = 2048):
    # Embeddings
    src_embed = InputEmbeddings(d_model, src_vocab_size)
    tgt_embed = InputEmbeddings(d_model, tgt_vocab_size)

    # Positional encodings
    src_pos = PositionalEncoding(d_model, src_seq_len, dropout)
    tgt_pos = PositionalEncoding(d_model, tgt_seq_len, dropout)

    # Encoder blocks with MultiHeadLatentAttentionBlock
    encoder_blocks = []
    for _ in range(N):
        encoder_self_attention = MultiHeadLatentAttentionBlock(d_model, h,
    dropout)
        feed_forward = FeedForwardBlock(d_model, d_ff, dropout)
        encoder_block = EncoderBlock(d_model, encoder_self_attention,
                                     feed_forward, dropout)
        encoder_blocks.append(encoder_block)

    # Decoder blocks with MultiHeadLatentAttentionBlock
    decoder_blocks = []
    for _ in range(N):
        decoder_self_attention = MultiHeadLatentAttentionBlock(d_model, h,
    dropout)
        decoder_cross_attention = MultiHeadLatentAttentionBlock(d_model, h,
    dropout)
        feed_forward = FeedForwardBlock(d_model, d_ff, dropout)
        decoder_block = DecoderBlock(d_model, decoder_self_attention,
                                     decoder_cross_attention, feed_forward,
    dropout)
        decoder_blocks.append(decoder_block)

    # Assemble
    encoder = Encoder(d_model, nn.ModuleList(encoder_blocks))
    decoder = Decoder(d_model, nn.ModuleList(decoder_blocks))
    projection_layer = ProjectionLayer(d_model, tgt_vocab_size)

    transformer = Transformer(encoder, decoder, src_embed, tgt_embed,
                              src_pos, tgt_pos, projection_layer)

    # Xavier initialization
    for p in transformer.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return transformer
```

Listing 14: build_transformer function

Default Hyperparameters:

Parameter	**Value**	**Description**
d_{model}	512	Model dimension
N	6	Number of layers
h	8	Number of attention heads
d_{ff}	2048	Feed-forward hidden dimension
dropout	0.1	Dropout rate

6 Data Flow Summary

6.1 Training Forward Pass

1. **Source Processing:**

$$\text{src} \rightarrow \text{Embedding} \rightarrow \text{Positional Encoding} \tag{15}$$
$$\rightarrow \text{Encoder} \rightarrow \text{encoder\_output} \tag{16}$$

2. **Target Processing:**

$$\text{tgt} \rightarrow \text{Embedding} \rightarrow \text{Positional Encoding} \tag{17}$$
$$\rightarrow \text{Decoder (with cross-attention)} \tag{18}$$
$$\rightarrow \text{decoder\_output} \tag{19}$$

3. **Prediction:**

$$\text{decoder\_output} \rightarrow \text{Projection} \rightarrow \text{Softmax} \rightarrow \text{Loss} \tag{20}$$

6.2 Inference (Greedy Decoding)

1. Encode source sequence once: memory = encode(src)
2. Start with [SOS] token
3. For each position i:
   - Decode: output = decode(memory, generated_tokens)
   - Project: logits = project(output)
   - Take argmax: $\text{token}_i = \arg\max(\text{logits})$
   - Append to generated sequence
4. Stop when [EOS] token generated

7 Key Differences from Original Paper

1. **Multi-Head Latent Attention:** This implementation uses KV compression, reducing memory by 75%.
2. **Pre-Normalization:** Layer norm happens before attention/FFN (not after).
3. **Xavier Initialization:** Uses `xavier_uniform_` instead of the paper's scheme.

8 Conclusion

This implementation provides a clean, modular Transformer architecture with the innovative addition of Multi-Head Latent Attention. The code is well-structured with clear separation of concerns:

- **Embeddings:** Handle input representation
- **Attention:** Captures relationships (with memory-efficient MLA)
- **Encoder/Decoder:** Process sequences bidirectionally/unidirectionally
- **Residual Connections:** Enable deep network training

The use of `MultiHeadLatentAttentionBlock` throughout makes this implementation particularly suitable for resource-constrained environments where KV cache memory is a bottleneck.