LFM2: Liquid Foundation Model 2
Step-by-Step Implementation Guide

Complete Tutorial

September 6, 2025

Contents

1 Introduction

LFM2 (Liquid Foundation Model 2) represents a hybrid neural architecture combining convolution operations for local processing with attention mechanisms for global dependencies. This tutorial walks through each component step-by-step, showing the architecture diagram first, followed immediately by the corresponding implementation.

2 Foundation Components

2.1 Configuration and Setup

Let's start with the basic configuration that defines our model:

```python
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Optional, Tuple, Dict, Any, Union
from dataclasses import dataclass

@dataclass
class LFM2Config:
    """Configuration class for LFM2 model."""
    vocab_size: int = 32768            # Size of the vocabulary
    hidden_size: int = 1024            # Hidden dimension size
    intermediate_size: int = 2816      # Feedforward intermediate dimension
    num_conv_blocks: int = 10          # Number of LIV convolution blocks
    num_attention_blocks: int = 6      # Number of GQA blocks
    num_attention_heads: int = 16      # Number of attention heads
    num_key_value_heads: int = 4       # Number of key-value heads for GQA
    conv_kernel_size: int = 3          # Kernel size for short convolutions
    max_position_embeddings: int = 32768  # Maximum sequence length
    rms_norm_eps: float = 1e-6         # RMS normalization epsilon
    rope_theta: float = 10000.0        # RoPE base frequency
    attention_dropout: float = 0.0     # Dropout rate for attention
    hidden_dropout: float = 0.0        # Dropout rate for hidden layers
    initializer_range: float = 0.02    # Weight initialization std
    use_cache: bool = True             # Enable KV caching
    tie_word_embeddings: bool = True   # Tie input/output embeddings
```

This configuration defines the hybrid architecture: 10 convolution blocks for local processing, followed by 6 attention blocks for global dependencies. The grouped query attention uses 16 query heads but only 4 key-value heads (4:1 ratio) for memory efficiency.

2.2 RMSNorm Implementation

```python
class RMSNorm(nn.Module):
    """Root Mean Square Layer Normalization.

    RMSNorm normalizes using only the root mean square, without centering:
    RMSNorm(x) = (x / RMS(x)) * learnable_scale
    where RMS(x) = sqrt(mean(x^2))
    """

```

```python
    def __init__(self, hidden_size: int, eps: float = 1e-6):
        super().__init__()
        # Learnable scale parameter (no bias unlike LayerNorm)
        self.weight = nn.Parameter(torch.ones(hidden_size))
        self.variance_epsilon = eps

    def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
        # Store original dtype for final output
        input_dtype = hidden_states.dtype

        # Convert to float32 for stable computation
        hidden_states = hidden_states.to(torch.float32)

        # Compute variance: mean of squared values
        variance = hidden_states.pow(2).mean(-1, keepdim=True)

        # Normalize by RMS: x / sqrt(variance + eps)
        hidden_states = hidden_states * torch.rsqrt(variance + self.variance_epsilon)

        # Apply learnable scale and convert back to original dtype
        return self.weight * hidden_states.to(input_dtype)
```

RMSNorm is more stable than LayerNorm because it doesn't center the data (no mean subtraction), only scales by the RMS. This reduces computation and often improves training stability.

2.3 Rotary Positional Embedding (RoPE)

```python
class RotaryPositionalEmbedding(nn.Module):
    """Rotary Positional Embedding implementation.

    RoPE encodes position by rotating query and key vectors by position-dependent angles.
    This allows the model to understand relative positions naturally.
    """

    def __init__(self, dim: int, max_position_embeddings: int = 32768,
                 base: float = 10000.0):
        super().__init__()
        self.dim = dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base

        # Precompute frequency inverse: 1 / (base^(2i/d))
        # This creates different rotation frequencies for each dimension pair
        inv_freq = 1.0 / (self.base **
                          (torch.arange(0, self.dim, 2).float() / self.dim))
        self.register_buffer("inv_freq", inv_freq, persistent=False)

    def forward(self, x: torch.Tensor, seq_len: int) -> Tuple[torch.Tensor, torch.Tensor]:
        # Generate position indices [0, 1, 2, ..., seq_len-1]
        t = torch.arange(seq_len, device=x.device).type_as(self.inv_freq)

        # Compute frequencies for each position: pos * inv_freq
        freqs = torch.outer(t, self.inv_freq)

```

```python
        # Duplicate frequencies for sine and cosine (each dim pair needs both)
        emb = torch.cat((freqs, freqs), dim=-1)

        # Return cosine and sine components
        cos = emb.cos()
        sin = emb.sin()

        return cos, sin

def rotate_half(x: torch.Tensor) -> torch.Tensor:
    """Rotates half the hidden dims of the input.

    For RoPE, we need to rotate pairs of dimensions. This function
    swaps the first and second half and negates the second half.
    """
    x1 = x[..., : x.shape[-1] // 2]  # First half
    x2 = x[..., x.shape[-1] // 2 :]  # Second half
    return torch.cat((-x2, x1), dim=-1)  # Rotate: [-x2, x1]

def apply_rotary_pos_emb(q: torch.Tensor, k: torch.Tensor,
                        cos: torch.Tensor, sin: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    """Apply rotary positional embedding to query and key tensors.

    The rotation is: x_rotated = x * cos + rotate_half(x) * sin
    This applies the rotation matrix in complex number form.
    """
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```

RoPE works by treating pairs of dimensions as complex numbers and rotating them by position-dependent angles. This creates a natural way for the model to understand relative positions between tokens.

3 LFM2 Convolution Block

3.1 Architecture Diagram

3.2 Implementation

```python
class LFM2ConvBlock(nn.Module):
    """LFM2 double-gated short-range convolution block.

    This implements Linear Input-Varying (LIV) convolution:
    1. Project input to 3 components: gates B, C, and values x'
    2. First gating: B * x' (input-dependent gating)
    3. Depthwise convolution for local pattern extraction
    4. Second gating: C * conv_output (input-dependent gating)
    5. Final projection back to hidden size
    """

    def __init__(self, config: LFM2Config):
        super().__init__()
        self.config = config
```

LFM2 Convolution Block

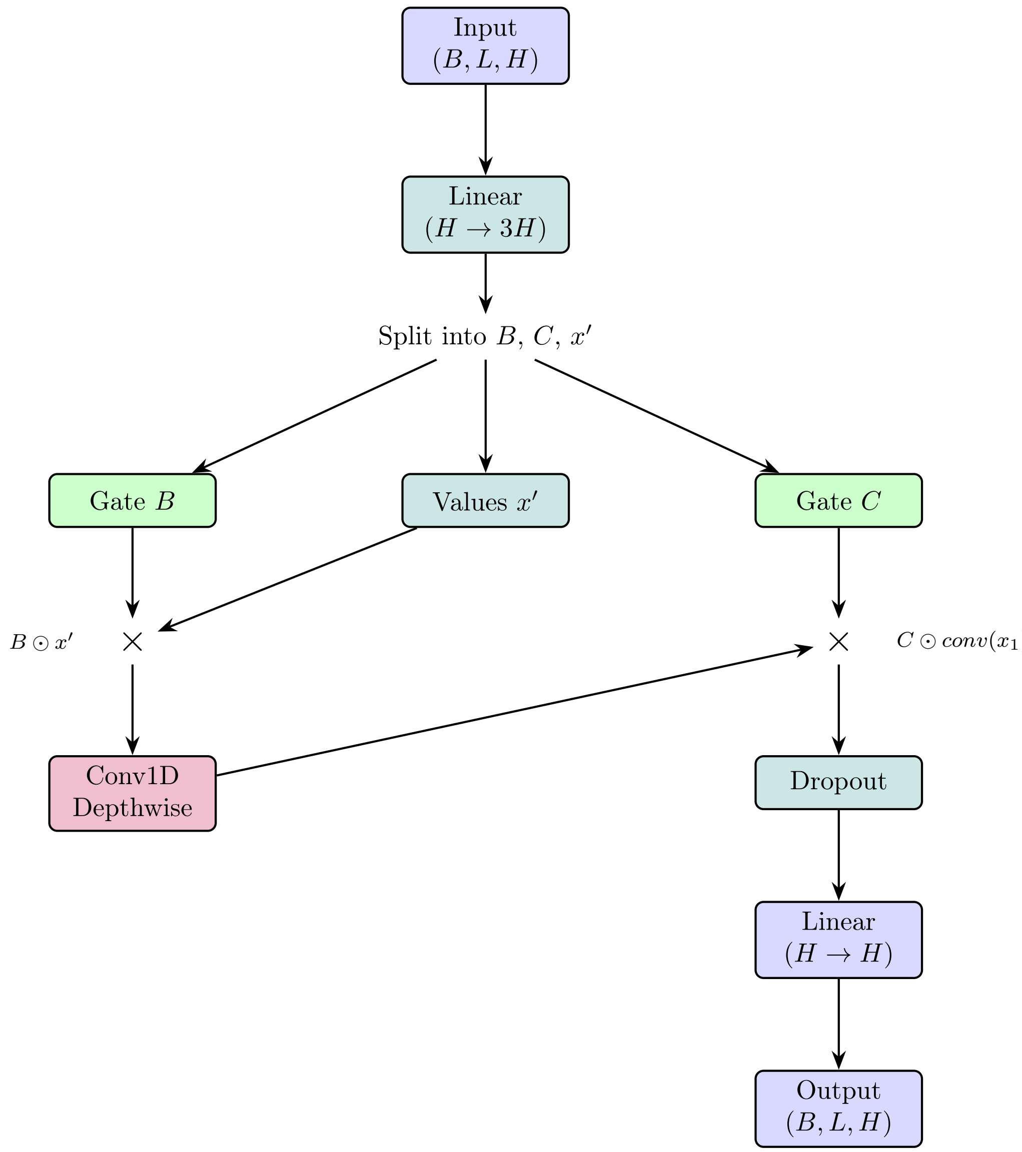

Figure 1: Double-gated Linear Input-Varying convolution with input-dependent gates B and C.

```python
        self.hidden_size = config.hidden_size

        # Input projection to gates and values (splits into B, C, x)
        self.input_projection = nn.Linear(
            self.hidden_size,
            3 * self.hidden_size,  # Output: 3 * hidden_size for B, C, x
            bias=False,
        )

        # Depthwise convolution: each channel processed independently
        # This is memory efficient and captures local patterns
```

```python
        self.conv = nn.Conv1d(
            self.hidden_size,           # Input channels
            self.hidden_size,           # Output channels (same as input)
            kernel_size=config.conv_kernel_size,  # Usually 3
            padding=config.conv_kernel_size // 2,  # Same padding
            groups=self.hidden_size,    # Depthwise: groups = channels
            bias=False,
        )

        # Output projection back to original hidden size
        self.output_projection = nn.Linear(
            self.hidden_size, self.hidden_size, bias=False
        )

        # Dropout for regularization
        self.dropout = nn.Dropout(config.hidden_dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward pass implementing the LIV convolution.

        Args:
            x: Input tensor of shape (batch_size, seq_len, hidden_size)

        Returns:
            Output tensor of same shape as input
        """
        batch_size, seq_len, hidden_size = x.shape

        # Step 1: Project input to gates B, C and values x'
        # Input: (B, L, H) -> Output: (B, L, 3H)
        projected = self.input_projection(x)

        # Split into three equal parts: B, C, x'
        # Each has shape (B, L, H)
        B, C, x_proj = projected.chunk(3, dim=-1)

        # Step 2: First gating - multiply values by gate B
        # This allows input-dependent filtering before convolution
        x_gated = B * x_proj  # Element-wise multiplication

        # Step 3: Apply depthwise convolution
        # Conv1D expects (batch, channels, sequence), so transpose
        x_conv_input = x_gated.transpose(1, 2)  # (B, L, H) -> (B, H, L)
        x_conv = self.conv(x_conv_input)        # Apply convolution
        x_conv = x_conv.transpose(1, 2)         # Back to (B, L, H)

        # Step 4: Second gating - multiply conv output by gate C
        # This provides input-dependent filtering after convolution
        x_gated_2 = C * x_conv

        # Step 5: Apply dropout for regularization
        x_gated_2 = self.dropout(x_gated_2)

        # Step 6: Final output projection
        output = self.output_projection(x_gated_2)

        return output
```

The key innovation here is the double gating mechanism. Unlike standard convolution, both gates B and C depend on the input, making this a "Linear Input-Varying" convolution. The depthwise convolution (groups=hidden$_size)iscomputationallyefficientwhilestillcapturinglocalpatterns.$

4 Grouped Query Attention

4.1 Architecture Diagram

4.2 Implementation

```python
class GroupedQueryAttention(nn.Module):
    """Grouped Query Attention implementation.

    GQA reduces memory usage by using fewer key-value heads than query heads.
    For example: 16 query heads, 4 key-value heads means each KV head
    is shared across 4 query heads (16/4 = 4 groups).
    """

    def __init__(self, config: LFM2Config):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.num_heads = config.num_attention_heads          # e.g., 16
        self.num_key_value_heads = config.num_key_value_heads # e.g., 4
        self.head_dim = self.hidden_size // self.num_heads    # e.g., 64

        # Number of query heads per key-value head
        self.num_key_value_groups = self.num_heads // self.num_key_value_heads

        # Validate configuration
        if self.hidden_size % self.num_heads != 0:
            raise ValueError(f"hidden_size must be divisible by num_heads")

        # Linear projections - note different output sizes
        self.q_proj = nn.Linear(
            self.hidden_size,
            self.num_heads * self.head_dim,          # Full size: 16 * 64
            bias=False,
        )
        self.k_proj = nn.Linear(
            self.hidden_size,
            self.num_key_value_heads * self.head_dim, # Reduced: 4 * 64
            bias=False,
        )
        self.v_proj = nn.Linear(
            self.hidden_size,
            self.num_key_value_heads * self.head_dim, # Reduced: 4 * 64
            bias=False,
        )
        self.o_proj = nn.Linear(
            self.num_heads * self.head_dim,          # Back to full hidden size
            self.hidden_size,
            bias=False,
        )

        # Rotary positional embedding
        self.rotary_emb = RotaryPositionalEmbedding(
```

Grouped Query Attention

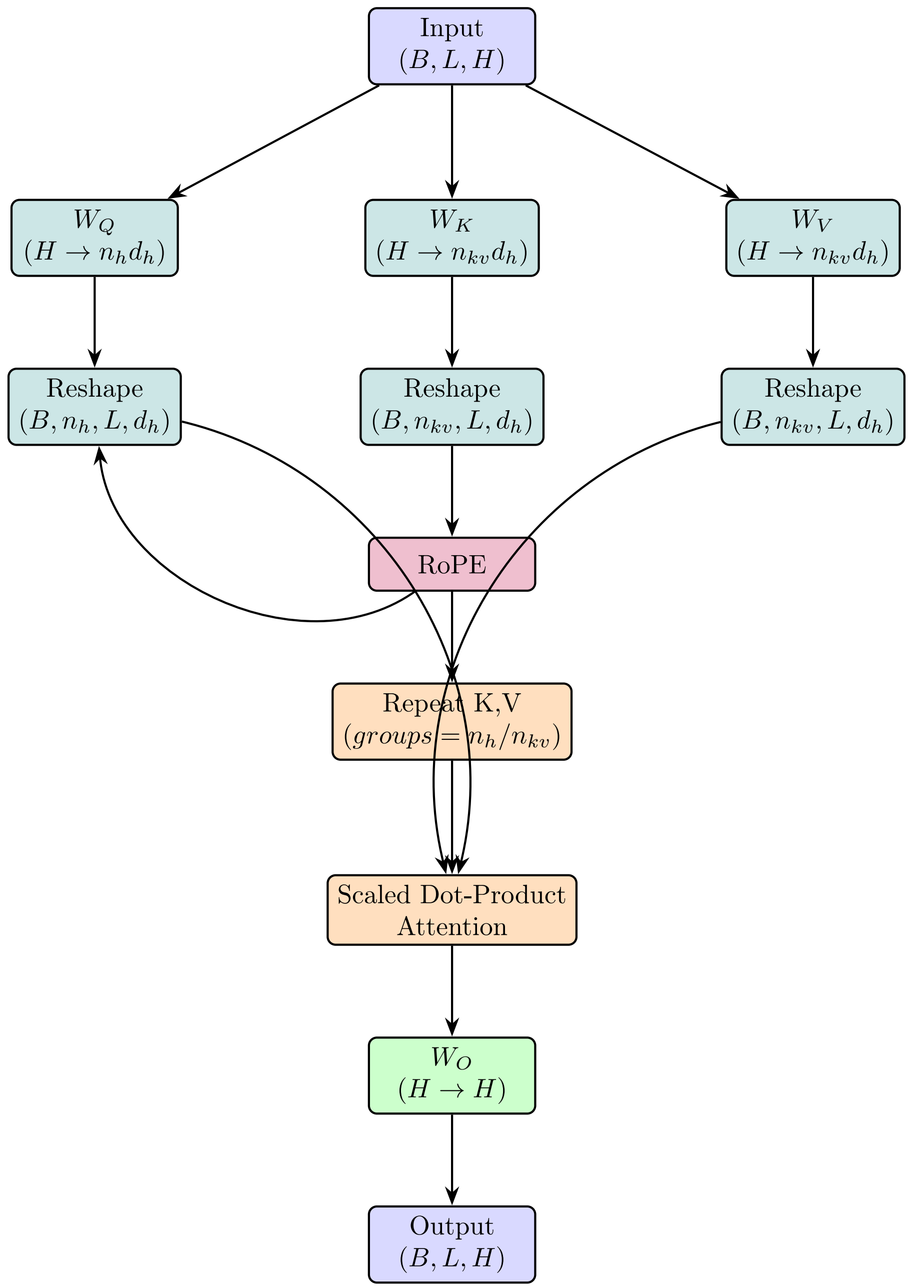

Figure 2: GQA uses fewer key-value heads than query heads, then repeats K,V to match Q heads.

```
            self.head_dim,
            config.max_position_embeddings,
            config.rope_theta,
        )

        # Attention dropout
        self.attention_dropout = nn.Dropout(config.attention_dropout)
```

```python

    def forward(self, hidden_states: torch.Tensor,
                attention_mask: Optional[torch.Tensor] = None,
                position_ids: Optional[torch.Tensor] = None,
                past_key_value: Optional[Tuple[torch.Tensor]] = None,
                output_attentions: bool = False,
                use_cache: bool = False) -> Tuple[torch.Tensor, Optional[torch.Tensor], Optional[Tuple[torch.Tensor]]]:

        bsz, q_len, _ = hidden_states.size()

        # Step 1: Project input to Q, K, V
        query_states = self.q_proj(hidden_states)  # (B, L, n_h * d_h)
        key_states = self.k_proj(hidden_states)    # (B, L, n_kv * d_h)
        value_states = self.v_proj(hidden_states)  # (B, L, n_kv * d_h)

        # Step 2: Reshape for multi-head attention
        # Q: (B, L, n_h * d_h) -> (B, n_h, L, d_h)
        query_states = query_states.view(
            bsz, q_len, self.num_heads, self.head_dim
        ).transpose(1, 2)

        # K, V: (B, L, n_kv * d_h) -> (B, n_kv, L, d_h)
        key_states = key_states.view(
            bsz, q_len, self.num_key_value_heads, self.head_dim
        ).transpose(1, 2)
        value_states = value_states.view(
            bsz, q_len, self.num_key_value_heads, self.head_dim
        ).transpose(1, 2)

        # Step 3: Get sequence length for RoPE (including past context)
        kv_seq_len = key_states.shape[-2]
        if past_key_value is not None:
            kv_seq_len += past_key_value[0].shape[-2]

        # Step 4: Apply rotary positional embedding to Q and K
        cos, sin = self.rotary_emb(value_states, kv_seq_len)
        query_states, key_states = apply_rotary_pos_emb(
            query_states, key_states, cos, sin
        )

        # Step 5: Handle past key-value cache for generation
        if past_key_value is not None:
            # Concatenate past and current key-values
            key_states = torch.cat([past_key_value[0], key_states], dim=2)
            value_states = torch.cat([past_key_value[1], value_states], dim=2)

        # Prepare cache for next iteration
        past_key_value = (key_states, value_states) if use_cache else None

        # Step 6: Repeat K,V to match number of query heads (GQA core)
        # From (B, n_kv, L, d_h) to (B, n_h, L, d_h)
        key_states = key_states.repeat_interleave(
            self.num_key_value_groups, dim=1
        )
        value_states = value_states.repeat_interleave(
            self.num_key_value_groups, dim=1
        )

```

```python
        # Step 7: Compute scaled dot-product attention
        # Q @ K^T / sqrt(d_k)
        attn_weights = torch.matmul(
            query_states, key_states.transpose(2, 3)
        ) / math.sqrt(self.head_dim)

        # Apply attention mask if provided (for padding/causality)
        if attention_mask is not None:
            attn_weights = attn_weights + attention_mask

        # Step 8: Apply softmax and dropout
        attn_weights = F.softmax(attn_weights, dim=-1, dtype=torch.float32)
        attn_weights = attn_weights.to(query_states.dtype)
        attn_weights = self.attention_dropout(attn_weights)

        # Step 9: Apply attention to values
        attn_output = torch.matmul(attn_weights, value_states)

        # Step 10: Reshape back to original format and apply output projection
        attn_output = attn_output.transpose(1, 2).contiguous()
        attn_output = attn_output.reshape(bsz, q_len, self.hidden_size)
        attn_output = self.o_proj(attn_output)

        # Return outputs (optionally include attention weights)
        if not output_attentions:
            attn_weights = None

        return attn_output, attn_weights, past_key_value
```

The memory efficiency of GQA comes from step 6: instead of computing separate K,V for each head, we compute fewer K,V heads and repeat them. This reduces the KV cache size significantly during generation.

5 SwiGLU Feed-Forward Network

5.1 Implementation

```python
class SwiGLU(nn.Module):
    """SwiGLU activation function.

    SwiGLU combines Swish activation with Gated Linear Units:
    SwiGLU(x) = Swish(xW_gate) * (xW_up) * W_down

    Where Swish(x) = x * sigmoid(x) = x *    (x)
    This has been shown to work better than ReLU-based FFNs.
    """

    def __init__(self, config: LFM2Config):
        super().__init__()
        self.hidden_size = config.hidden_size
        self.intermediate_size = config.intermediate_size  # Usually ~2.7x hidden_size

        # Two parallel transformations to intermediate size
        self.gate_proj = nn.Linear(
            self.hidden_size, self.intermediate_size, bias=False
```

```
        )
        self.up_proj = nn.Linear(
            self.hidden_size, self.intermediate_size, bias=False
        )

        # Down projection back to hidden size
        self.down_proj = nn.Linear(
            self.intermediate_size, self.hidden_size, bias=False
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward pass implementing SwiGLU.

        Args:
            x: Input tensor of shape (..., hidden_size)

        Returns:
            Output tensor of same shape as input
        """
        # Apply both linear projections in parallel
        gate = self.gate_proj(x)    # "Gate" branch: controls information flow
        up = self.up_proj(x)        # "Up" branch: carries information

        # Apply SiLU (Swish) activation to gate branch
        # SiLU(x) = x * sigmoid(x) - smooth, non-monotonic activation
        swish_gate = F.silu(gate)

        # Multiply gate and up branches (gating mechanism)
        gated_output = swish_gate * up

        # Project back down to original hidden size
        return self.down_proj(gated_output)
```

SwiGLU has been empirically shown to outperform ReLU-based feed-forward networks. The gating mechanism allows the network to control information flow, while SiLU provides smooth gradients compared to ReLU.

6 LFM2 Transformer Block

6.1 Architecture Diagram

6.2 Implementation

```
class LFM2Block(nn.Module):
    """LFM2 transformer block with GQA and SwiGLU.

    Uses pre-normalization pattern:
    1.  x   = x + Attention(RMSNorm(x))
    2.  x   =  x   + SwiGLU(RMSNorm( x  ))

    Pre-norm helps with training stability compared to post-norm.
    """

    def __init__(self, config: LFM2Config):
        super().__init__()
```

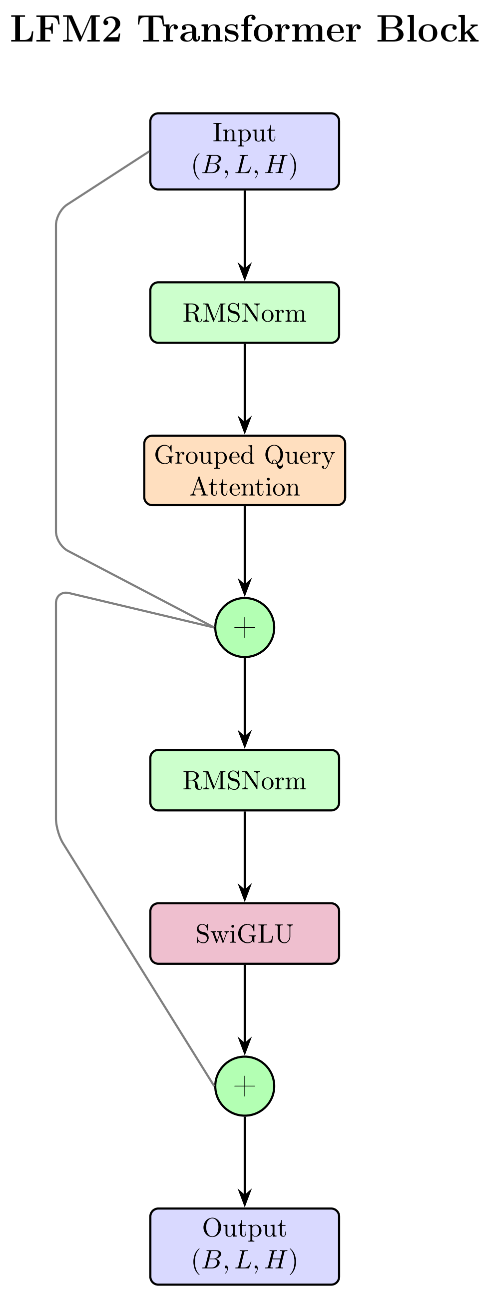

Figure 3: Pre-normalization transformer block with residual connections.

```
        self.config = config
        self.hidden_size = config.hidden_size

        # Main components
        self.self_attn = GroupedQueryAttention(config)
        self.mlp = SwiGLU(config)

        # Layer normalization (pre-norm pattern)
        self.input_layernorm = RMSNorm(
            config.hidden_size, eps=config.rms_norm_eps
```

```python
        )
        self.post_attention_layernorm = RMSNorm(
            config.hidden_size, eps=config.rms_norm_eps
        )

    def forward(self, hidden_states: torch.Tensor,
                attention_mask: Optional[torch.Tensor] = None,
                position_ids: Optional[torch.Tensor] = None,
                past_key_value: Optional[Tuple[torch.Tensor]] = None,
                output_attentions: Optional[bool] = False,
                use_cache: Optional[bool] = False) -> Tuple[torch.Tensor, Optional
                    [torch.Tensor], Optional[Tuple[torch.Tensor]]]:
        """Forward pass implementing pre-norm transformer block."""

        # First residual block: Self-attention
        residual = hidden_states  # Store for residual connection

        # Pre-normalization before attention
        hidden_states = self.input_layernorm(hidden_states)

        # Apply self-attention
        hidden_states, self_attn_weights, present_key_value = self.self_attn(
            hidden_states=hidden_states,
            attention_mask=attention_mask,
            position_ids=position_ids,
            past_key_value=past_key_value,
            output_attentions=output_attentions,
            use_cache=use_cache,
        )

        # Add residual connection
        hidden_states = residual + hidden_states

        # Second residual block: Feed-forward network
        residual = hidden_states  # Store for next residual connection

        # Pre-normalization before MLP
        hidden_states = self.post_attention_layernorm(hidden_states)

        # Apply SwiGLU feed-forward network
        hidden_states = self.mlp(hidden_states)

        # Add residual connection
        hidden_states = residual + hidden_states

        # Prepare outputs
        outputs = (hidden_states,)

        if output_attentions:
            outputs += (self_attn_weights,)

        if use_cache:
            outputs += (present_key_value,)

        return outputs
```

The pre-normalization pattern (norm before operation) has become standard in modern transformers because it provides more stable gradients during training compared to post-normalization.

7 Complete LFM2 Model

7.1 Architecture Diagram

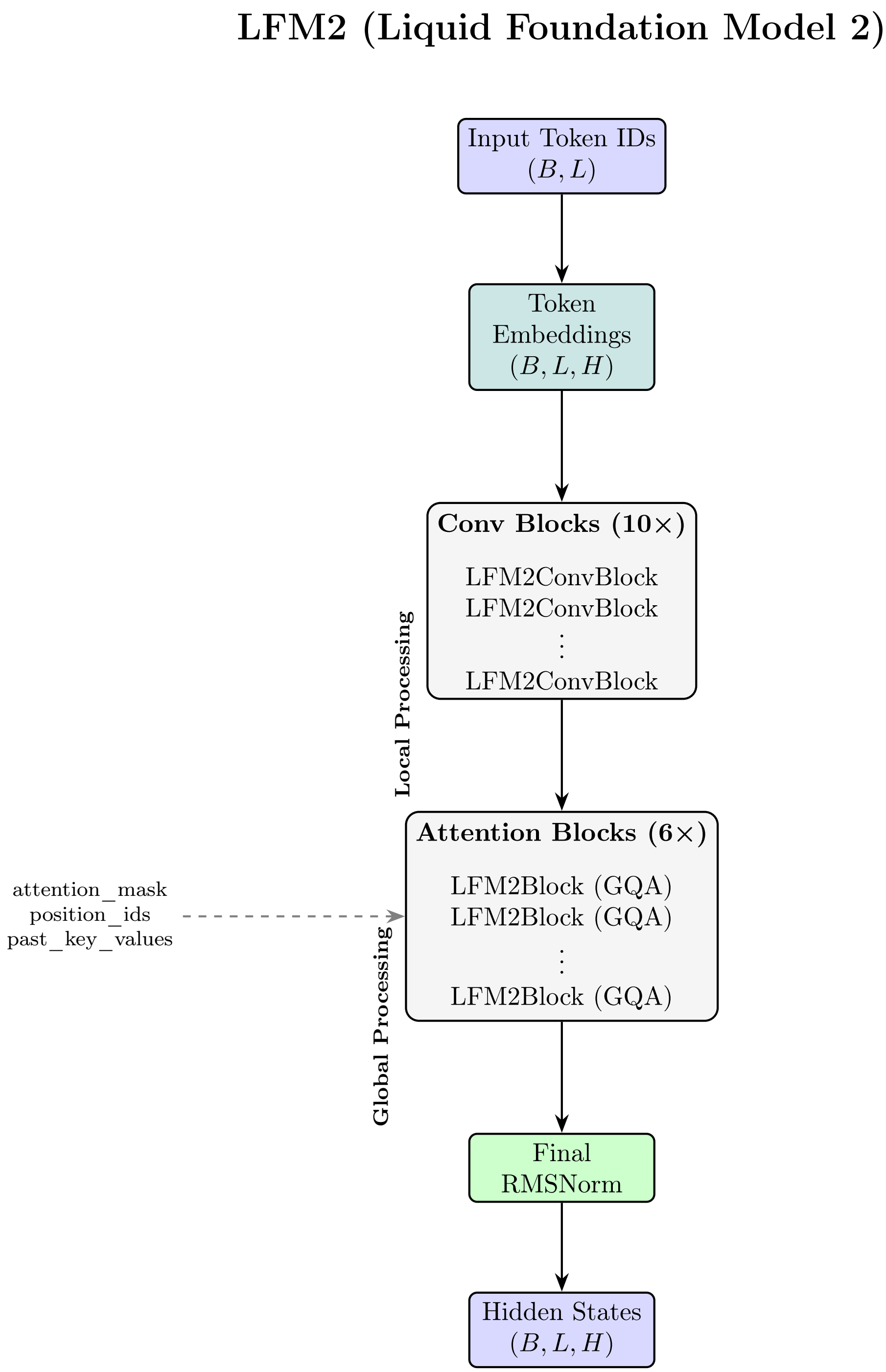

Figure 4: Complete LFM2 hybrid architecture: convolution for local patterns, attention for global dependencies.

7.2 Implementation

```python
class LFM2Model(nn.Module):
    """Complete LFM2 model with hybrid architecture.

    Architecture:
    - Token embeddings
    - 10 LFM2ConvBlocks (local pattern processing)
    - 6 LFM2Blocks with GQA (global dependency modeling)
    - Final RMSNorm

    This hybrid approach allows efficient local processing followed
    by global attention for long-range dependencies.
    """

    def __init__(self, config: LFM2Config):
        super().__init__()
        self.config = config
        self.vocab_size = config.vocab_size
        self.hidden_size = config.hidden_size

        # Token embeddings: convert token IDs to dense vectors
        self.embed_tokens = nn.Embedding(config.vocab_size, config.hidden_size)

        # Model layers: hybrid convolution + attention
        self.layers = nn.ModuleList()

        # Stage 1: Add convolution blocks (local processing)
        for i in range(config.num_conv_blocks):
            self.layers.append(LFM2ConvBlock(config))

        # Stage 2: Add attention blocks (global processing)
        for i in range(config.num_attention_blocks):
            self.layers.append(LFM2Block(config))

        # Final layer normalization
        self.norm = RMSNorm(config.hidden_size, eps=config.rms_norm_eps)

        # Initialize weights
        self.apply(self._init_weights)

    def _init_weights(self, module: nn.Module) -> None:
        """Initialize model weights following standard practices."""
        if isinstance(module, nn.Linear):
            # Normal initialization for linear layers
            torch.nn.init.normal_(
                module.weight, mean=0.0, std=self.config.initializer_range
            )
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            # Normal initialization for embeddings
            torch.nn.init.normal_(
                module.weight, mean=0.0, std=self.config.initializer_range
            )
        elif isinstance(module, nn.Conv1d):
            # Normal initialization for convolutions
            torch.nn.init.normal_(
```

```python
                module.weight, mean=0.0, std=self.config.initializer_range
            )
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)

    def forward(self, input_ids: torch.LongTensor,
                attention_mask: Optional[torch.Tensor] = None,
                position_ids: Optional[torch.LongTensor] = None,
                past_key_values: Optional[Tuple[Tuple[torch.Tensor]]] = None,
                use_cache: Optional[bool] = None,
                output_hidden_states: Optional[bool] = None,
                return_dict: Optional[bool] = None) -> Union[Tuple, Dict[str, torch.Tensor]]:
        """Forward pass through the complete LFM2 model."""

        # Set default values
        use_cache = use_cache if use_cache is not None else self.config.use_cache
        output_hidden_states = (output_hidden_states if output_hidden_states is not None
                                else self.config.output_hidden_states)
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        # Step 1: Convert token IDs to embeddings
        hidden_states = self.embed_tokens(input_ids)
        batch_size, seq_length = hidden_states.shape[:2]

        # Step 2: Initialize position IDs if not provided
        if position_ids is None:
            device = input_ids.device if input_ids is not None else hidden_states.device
            position_ids = torch.arange(seq_length, dtype=torch.long, device=device)
            position_ids = position_ids.unsqueeze(0).expand(batch_size, -1)

        # Step 3: Initialize past key values for caching
        if past_key_values is None:
            past_key_values = [None] * len(self.layers)

        # Step 4: Prepare attention mask for causal modeling
        if attention_mask is not None:
            attention_mask = self._prepare_attention_mask(
                attention_mask, seq_length, batch_size
            )

        # Step 5: Initialize output containers
        all_hidden_states = () if output_hidden_states else None
        next_decoder_cache = () if use_cache else None

        # Step 6: Process through all layers
        for idx, decoder_layer in enumerate(self.layers):
            if output_hidden_states:
                all_hidden_states += (hidden_states,)

            past_key_value = past_key_values[idx] if past_key_values is not None else None

            if isinstance(decoder_layer, LFM2ConvBlock):
                # Convolution blocks: simple forward pass (no attention cache)
```

```
                hidden_states = decoder_layer(hidden_states)
                layer_outputs = (hidden_states,)
            else:
                # Attention blocks: full forward pass with caching
                layer_outputs = decoder_layer(
                    hidden_states,
                    attention_mask=attention_mask,
                    position_ids=position_ids,
                    past_key_value=past_key_value,
                    use_cache=use_cache,
                )
                hidden_states = layer_outputs[0]

            # Collect cache from attention layers only
            if use_cache and isinstance(decoder_layer, LFM2Block):
                next_decoder_cache += (layer_outputs[2],)

        # Step 7: Apply final normalization
        hidden_states = self.norm(hidden_states)

        # Add final hidden state to output
        if output_hidden_states:
            all_hidden_states += (hidden_states,)

        # Prepare final cache
        next_cache = next_decoder_cache if use_cache else None

        # Return in requested format
        if not return_dict:
            return tuple(v for v in [hidden_states, next_cache, all_hidden_states]
                 if v is not None)

        return {
            "last_hidden_state": hidden_states,
            "past_key_values": next_cache,
            "hidden_states": all_hidden_states,
        }

    def _prepare_attention_mask(self, attention_mask: torch.Tensor,
                              seq_length: int, batch_size: int) -> torch.Tensor:
        """Prepare causal attention mask combining padding and causality."""

        # Create causal mask (lower triangular)
        causal_mask = torch.triu(
            torch.ones((seq_length, seq_length), dtype=torch.bool),
            diagonal=1,  # Upper triangle (including diagonal) = True (masked)
        ).to(attention_mask.device)

        # Expand attention mask to 4D: (batch, 1, seq, seq)
        expanded_mask = (attention_mask[:, None, None, :]
                        .expand(batch_size, 1, seq_length, seq_length)
                        .to(torch.float32))

        # Convert to mask format: 1.0 = masked, 0.0 = not masked
        expanded_mask = 1.0 - expanded_mask
        causal_float_mask = causal_mask.to(torch.float32)

        # Combine padding and causal masks
        combined_mask = expanded_mask + causal_float_mask[None, None, :, :]
```

```python

        # Convert to attention weights: 0 = attend, -inf = mask
        return combined_mask.masked_fill(combined_mask > 0, float("-inf"))
```

The LFM2Model orchestrates the entire hybrid architecture. The key insight is processing through convolution blocks first for local patterns, then attention blocks for global dependencies.

7.3 Language Modeling Head

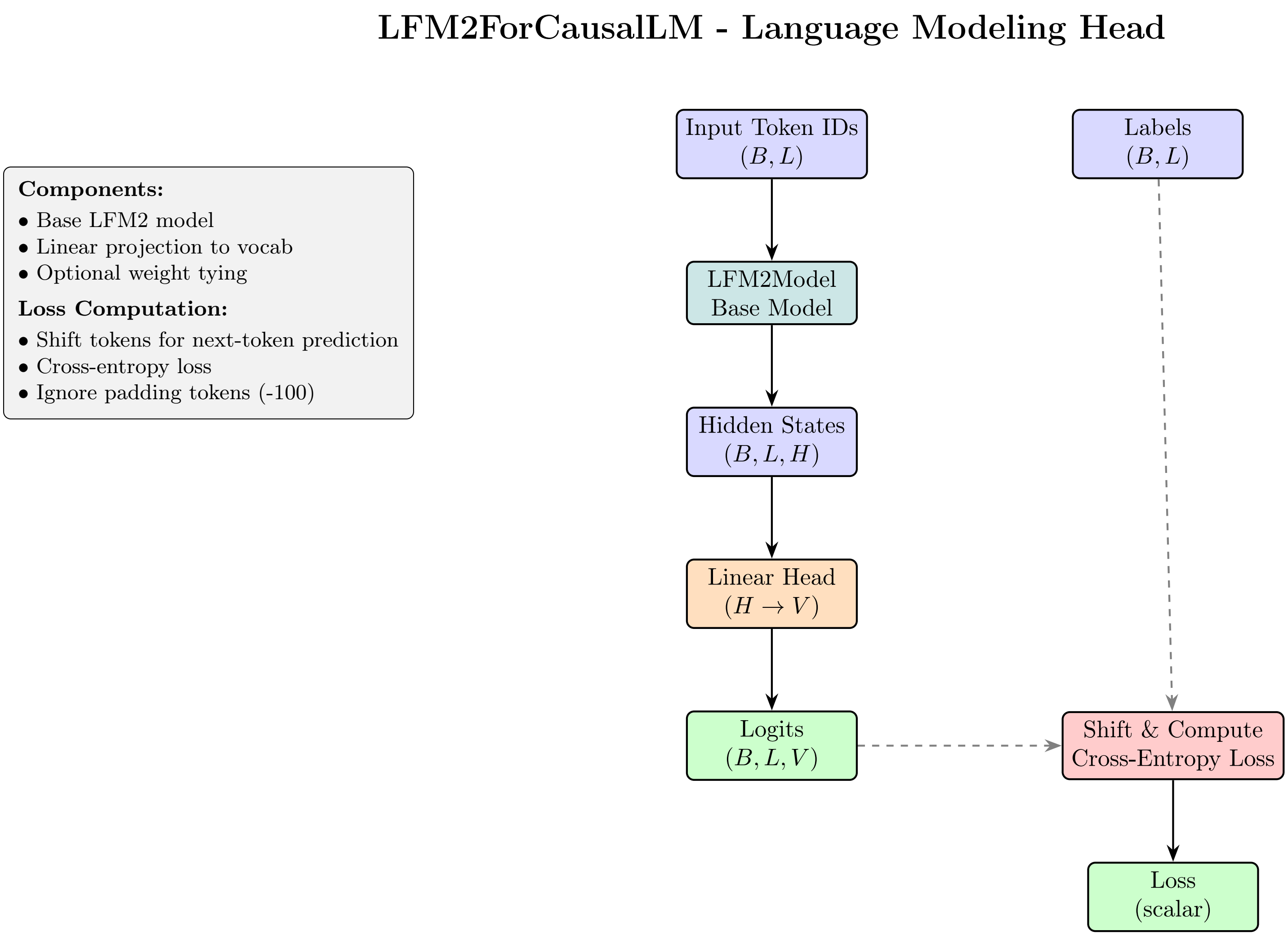

Figure 5: Language modeling head converts hidden states to vocabulary logits for text generation.

7.4 Attention Mask Preparation

```python
class LFM2ForCausalLM(nn.Module):
    """LFM2 model for causal language modeling (text generation)."""

    def __init__(self, config: LFM2Config):
```

Attention Mask Preparation

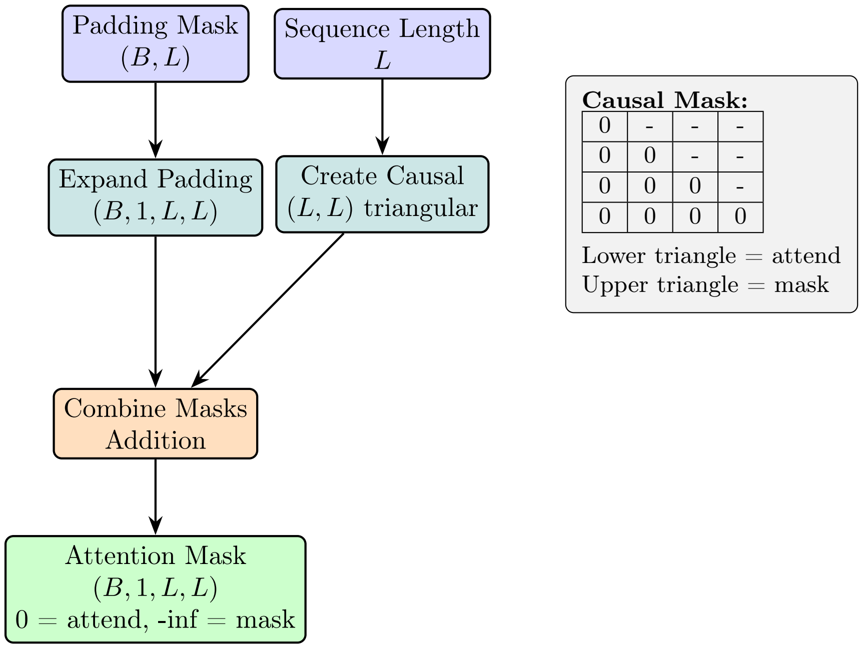

Figure 6: Attention mask combines padding mask with causal mask for autoregressive generation.

```
        super().__init__()
        self.config = config
        self.model = LFM2Model(config)
        self.vocab_size = config.vocab_size

        # Language modeling head: hidden states -> vocabulary logits
        self.lm_head = nn.Linear(config.hidden_size, config.vocab_size, bias=False)

        # Optionally tie input and output embeddings (parameter sharing)
        if config.tie_word_embeddings:
            self.lm_head.weight = self.model.embed_tokens.weight

        self.apply(self._init_weights)

    def forward(self, input_ids: torch.LongTensor,
                attention_mask: Optional[torch.Tensor] = None,
                labels: Optional[torch.LongTensor] = None,
                **kwargs) -> Union[Tuple, Dict[str, torch.Tensor]]:
        """Forward pass for causal language modeling."""

        # Forward through base model
        outputs = self.model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            **kwargs
        )

```

```
        hidden_states = outputs["last_hidden_state"] if isinstance(outputs, dict) else outputs[0]

        # Compute logits for each position and vocabulary item
        logits = self.lm_head(hidden_states)

        # Compute loss if labels are provided (training)
        loss = None
        if labels is not None:
            # Shift for next-token prediction: predict token i+1 from tokens 0..i
            shift_logits = logits[..., :-1, :].contiguous()  # Remove last position
            shift_labels = labels[..., 1:].contiguous()      # Remove first position

            # Cross-entropy loss
            loss_fct = nn.CrossEntropyLoss(ignore_index=-100)
            shift_logits = shift_logits.view(-1, self.vocab_size)
            shift_labels = shift_labels.view(-1)

            # Move to same device for model parallelism
            shift_labels = shift_labels.to(shift_logits.device)
            loss = loss_fct(shift_logits, shift_labels)

        # Return outputs
        if isinstance(outputs, dict):
            return {
                "loss": loss,
                "logits": logits,
                "past_key_values": outputs.get("past_key_values"),
                "hidden_states": outputs.get("hidden_states"),
            }
        else:
            output = (logits,) + outputs[1:]
            return (loss,) + output if loss is not None else output
```

7.5 Model Creation and Testing

```
def create_lfm2_model(model_size: str = "700M") -> LFM2ForCausalLM:
    """Create LFM2 model with predefined configurations."""

    # Predefined size configurations
    size_configs = {
        "350M": {
            "hidden_size": 768,
            "intermediate_size": 2048,
            "num_attention_heads": 12,
            "num_key_value_heads": 3,
        },
        "700M": {
            "hidden_size": 1024,
            "intermediate_size": 2816,
            "num_attention_heads": 16,
            "num_key_value_heads": 4,
        },
        "1.2B": {
```

```
            "hidden_size": 1536,
            "intermediate_size": 4096,
            "num_attention_heads": 24,
            "num_key_value_heads": 6,
        },
    }

    # Create and configure model
    config = LFM2Config()
    config.__dict__.update(size_configs[model_size])

    return LFM2ForCausalLM(config)

def test_lfm2_model():
    """Test LFM2 model with dummy inputs."""
    print("Creating LFM2 model...")
    model = create_lfm2_model("350M")
    model.eval()

    # Create dummy inputs
    batch_size, seq_len = 2, 128
    input_ids = torch.randint(0, model.config.vocab_size, (batch_size, seq_len))
    attention_mask = torch.ones(batch_size, seq_len)

    print(f"Input shape: {input_ids.shape}")
    print(f"Model parameters: {sum(p.numel() for p in model.parameters()):,}")

    # Forward pass
    with torch.no_grad():
        outputs = model(input_ids=input_ids, attention_mask=attention_mask)

    print(f"Output logits shape: {outputs['logits'].shape}")
    print("Test completed successfully!")

# Run the test
test_lfm2_model()
```

8 Conclusion

This step-by-step implementation of LFM2 demonstrates several key innovations:

- **Hybrid Architecture**: Convolution blocks for local patterns + attention blocks for global dependencies
- **Grouped Query Attention**: Reduces memory usage by sharing key-value pairs across query heads
- **Modern Components**: RMSNorm, RoPE, SwiGLU for improved training and performance
- **Efficient Design**: Double-gated convolutions and depthwise operations for computational efficiency

The implementation provides a complete, working LFM2 model suitable for language modeling tasks, with careful attention to memory efficiency and modern best practices.