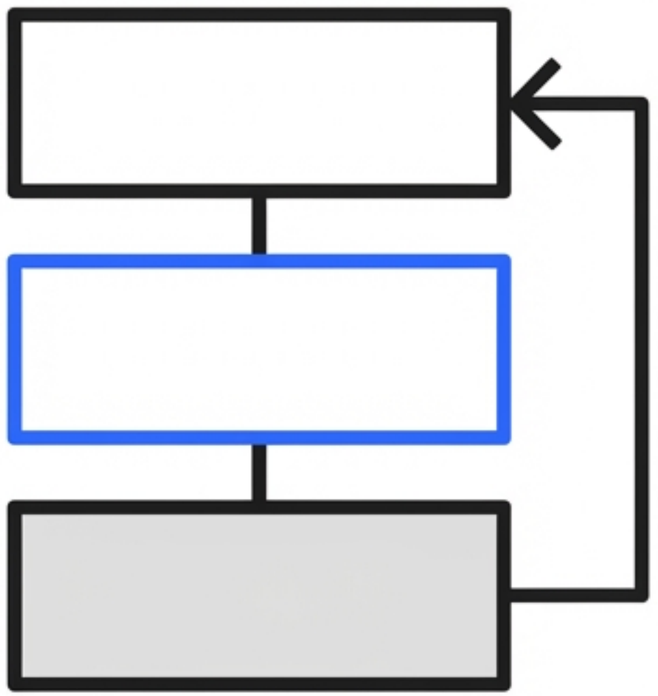
ARCHITECTURE TIER

MODULE 13

GPT Architecture

Building a transformer language model from scratch



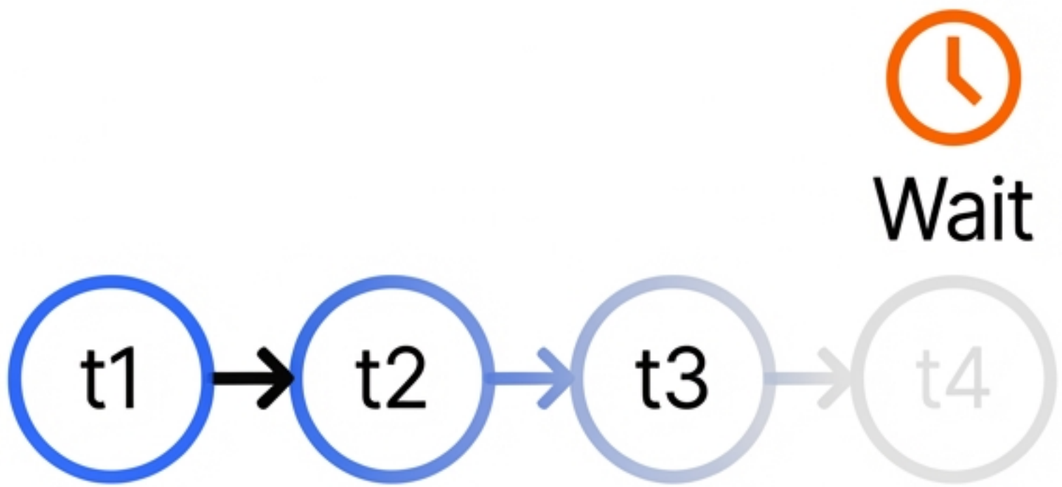
Module 13: Transformers

Architecture Tier | Concept → System → Code

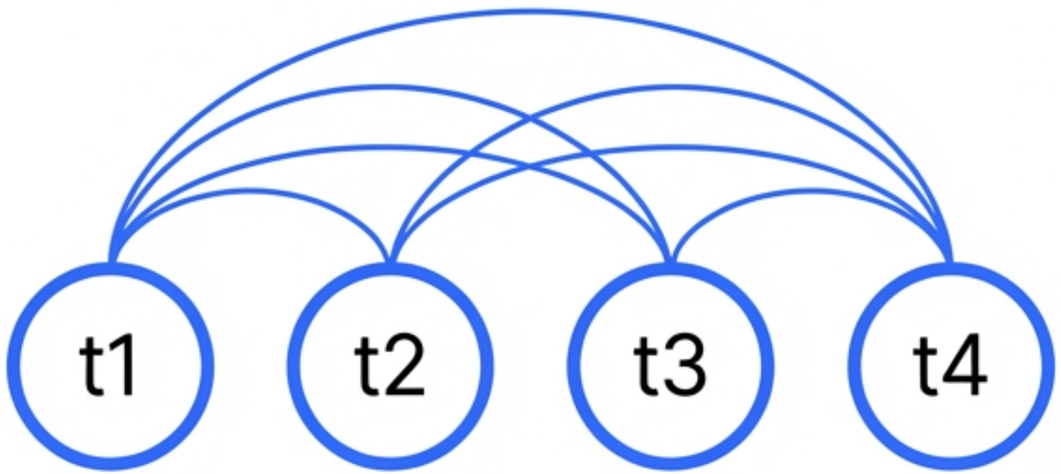
`Building the Engine of Modern Language Models`

From Sequential Steps to Parallel Sets

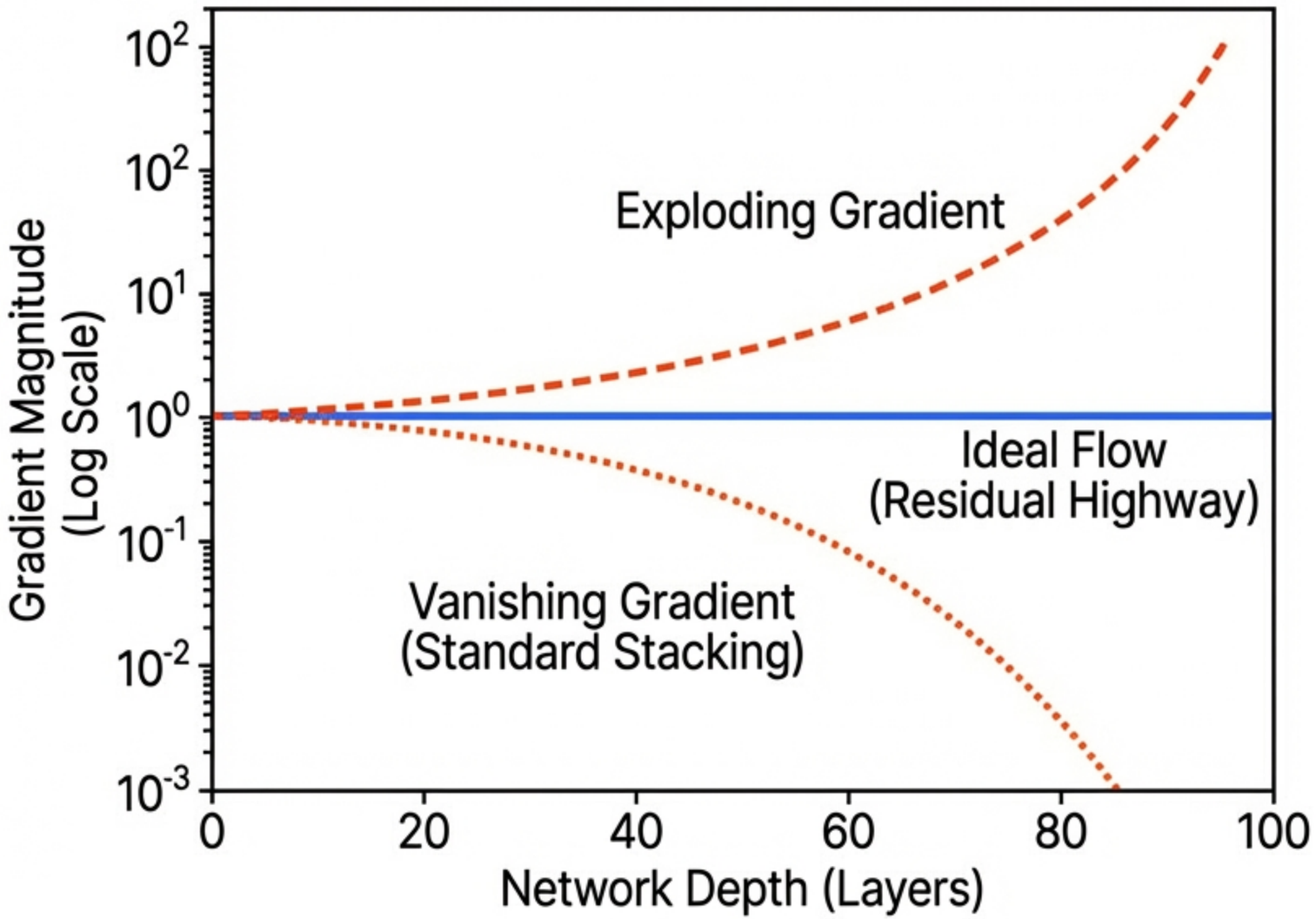
RNN: The Sequential Dependency



Transformer: Parallel + Global Context



The Instability of Deep Networks

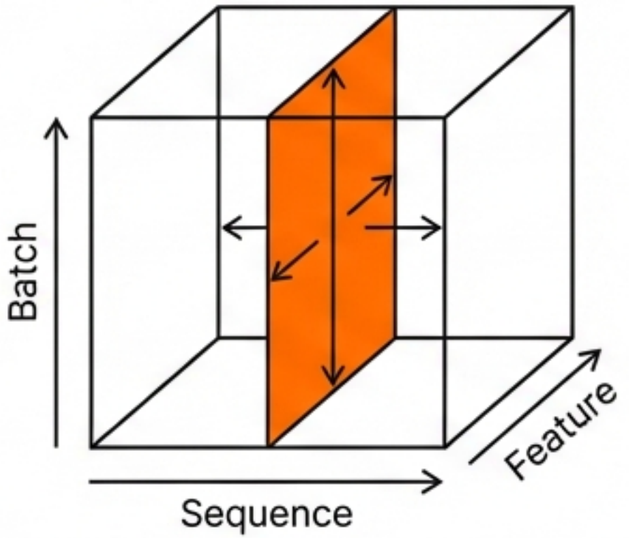


The Depth Problem:

- Stacking 100+ layers causes signal decay.
- Internal Covariate Shift destabilizes training.
- **Solution:** Normalization + Residual Highways.

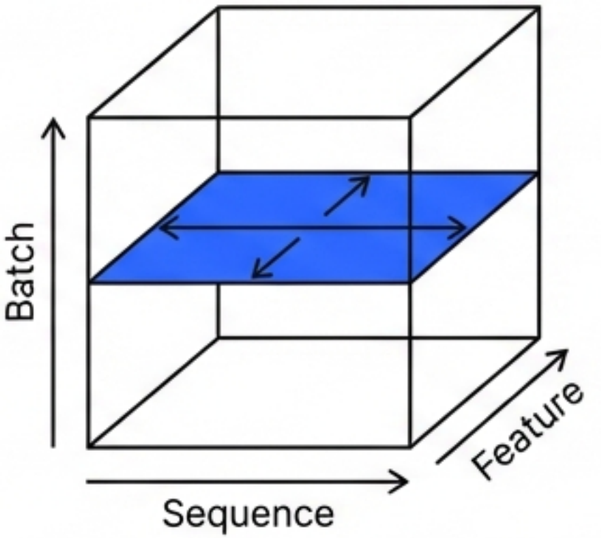
Normalizing Independent Samples

Batch Norm: Dependent on Batch Statistics



Calculated across all samples in a batch.

Layer Norm: Independent per Token



Calculated within a single sample.

Crucial for NLP: Handles variable sequence lengths and small batches.

$$y = \frac{x - \mu}{\sigma} \cdot \gamma + \beta$$

Statistics

Learnable Recovery Parameters

Implementing LayerNorm in TinyTorch

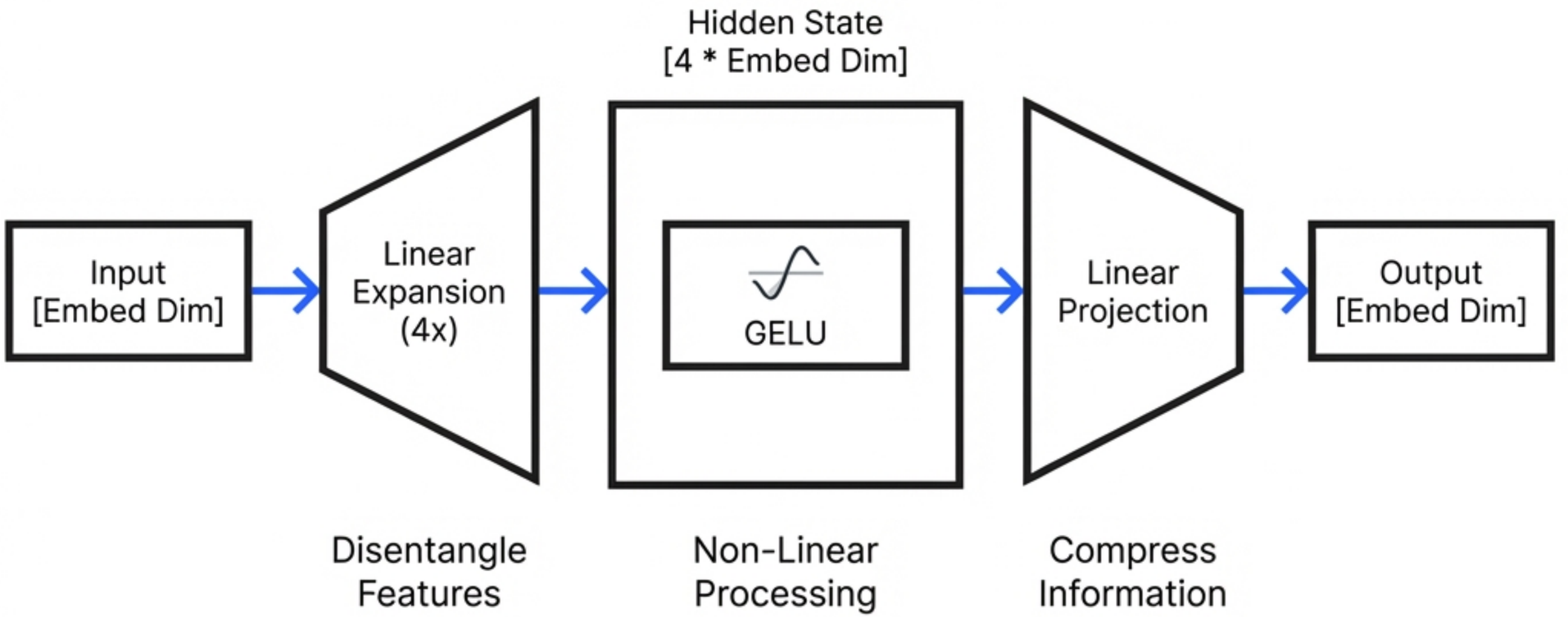
```python
class LayerNorm:
    def __init__(self, normalized_shape, eps=1e-5):
        self.gamma = Tensor(np.ones(normalized_shape)) # Scale
        self.beta = Tensor(np.zeros(normalized_shape)) # Shift
        self.eps = eps

    def forward(self, x):
        # Normalize across features (last dimension)
        mean = x.mean(axis=-1, keepdims=True)
        diff = x - mean
        variance = (diff * diff).mean(axis=-1, keepdims=True)
        std = Tensor(np.sqrt(variance.data + self.eps))

        return (x - mean) / std * self.gamma + self.beta
```

Learnable parameters allow identity recovery

Independent per token

Numerical stability

The MLP: Processing Information



Attention routes information. MLP processes it.

The 4x expansion creates an Information Bottleneck.

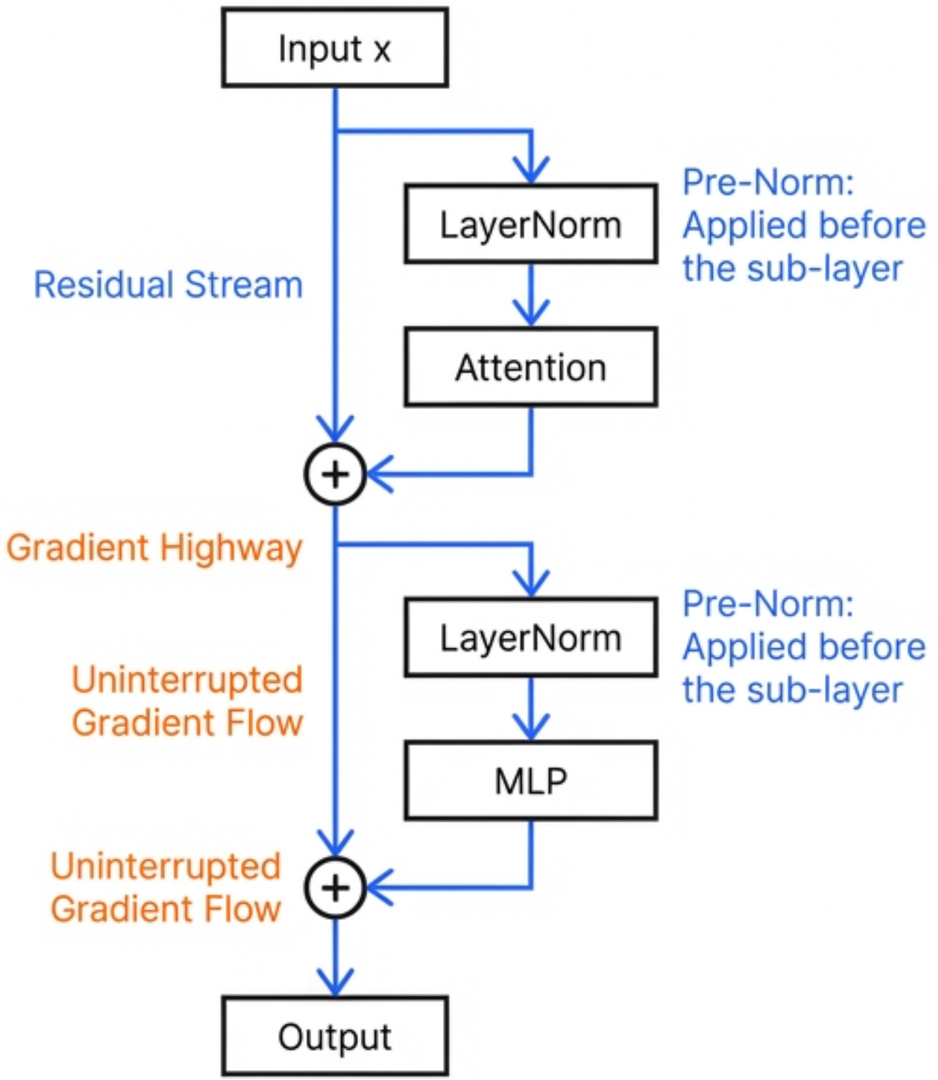
Implementing the Feed-Forward Network

```
class MLP:
    def __init__(self, embed_dim, hidden_dim=None):
        if hidden_dim is None: hidden_dim = 4 * embed_dim
        self.linear1 = Linear(embed_dim, hidden_dim)
        self.gelu = GELU()
        self.linear2 = Linear(hidden_dim, embed_dim)

    def forward(self, x):
        hidden = self.linear1.forward(x)
        hidden = self.gelu.forward(hidden)
        output = self.linear2.forward(hidden)
        return output
```

Shape Invariants

Input Tensor: `(batch, seq, embed_dim)`
Hidden State: `(batch, seq, 4 * embed_dim)`
Output Tensor: `(batch, seq, embed_dim)`

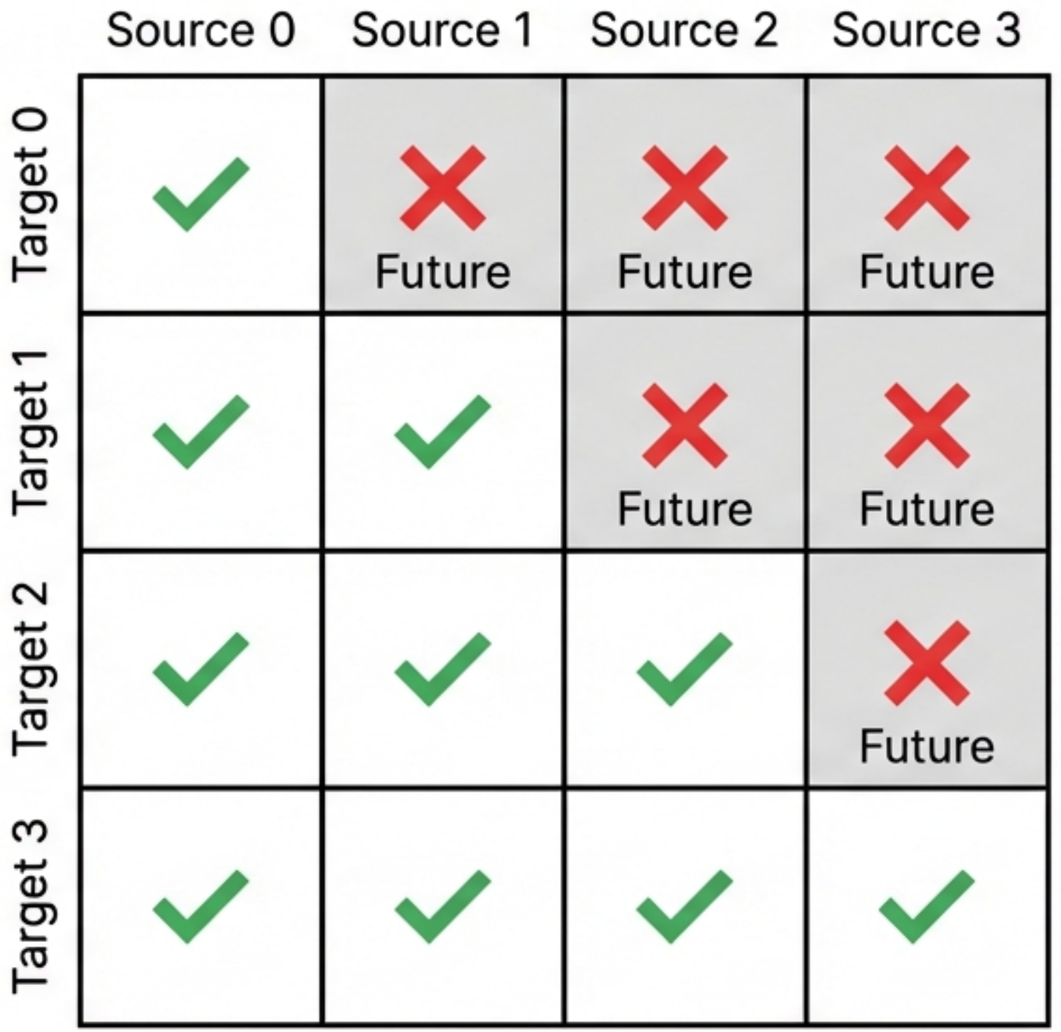
The Pre-Norm Transformer Block



Code: The Transformer Block

```
class TransformerBlock:
    def __init__(self, embed_dim, num_heads):
        self.attention = MultiHeadAttention(embed_dim, num_heads)
        self.ln1 = LayerNorm(embed_dim)
        self.ln2 = LayerNorm(embed_dim)
        self.mlp = MLP(embed_dim, int(embed_dim * 4))

    def forward(self, x, mask=None):
        # Sub-layer 1: Attention
        normed1 = self.ln1.forward(x)
        attn_out = self.attention.forward(normed1, mask)
        x = x + attn_out  # Residual Connection 1

        # Sub-layer 2: MLP
        normed2 = self.ln2.forward(x)
        mlp_out = self.mlp.forward(normed2)
        output = x + mlp_out # Residual Connection 2
        return output
```

Causal Masking: Preventing Time Travel

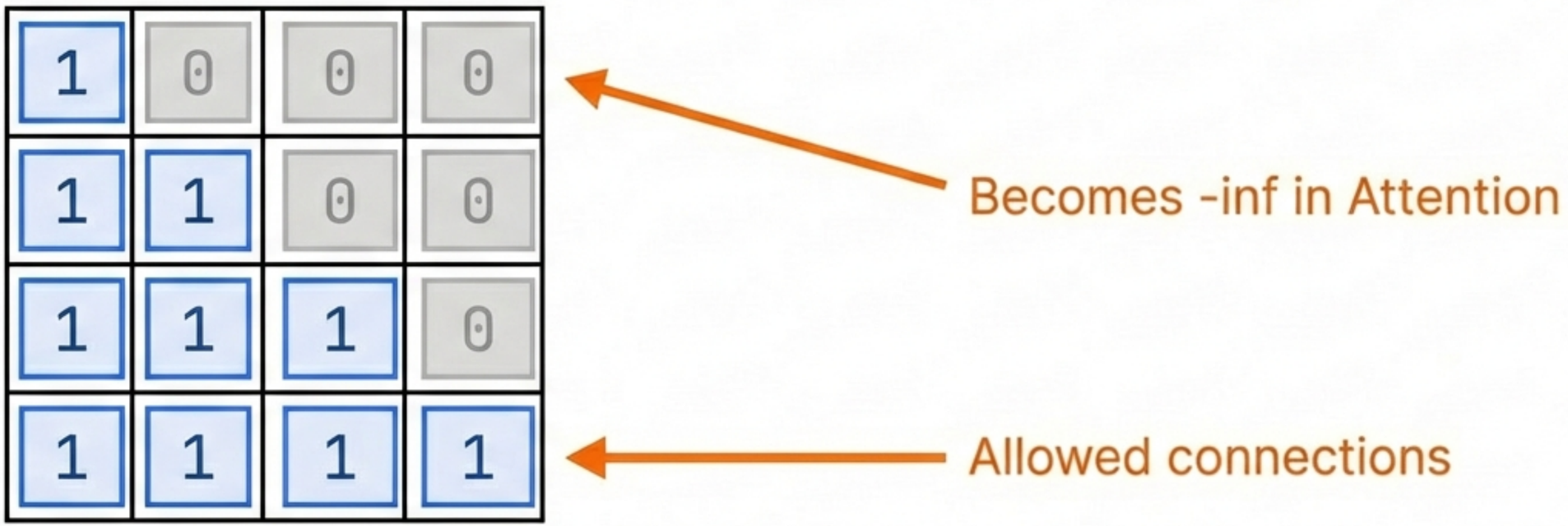


Training Constraint:

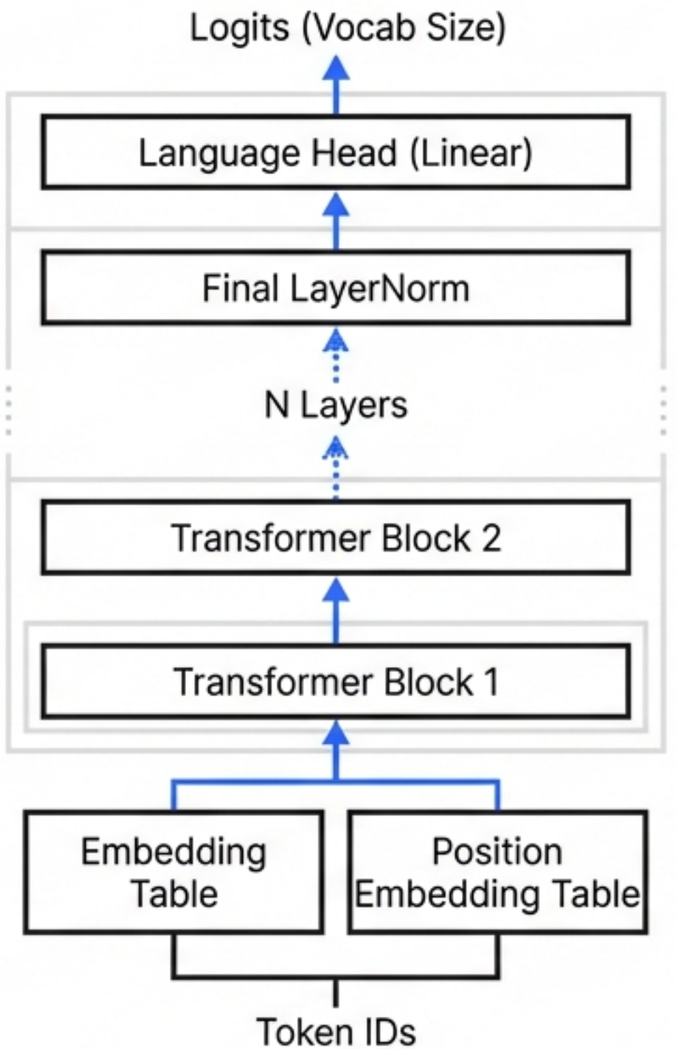
- The model sees the whole sequence at once.
- We must forcefully blind it to future tokens.
- Mechanism: Set attention scores to -infinity before Softmax.

Implementing the Causal Mask

```python
def create_causal_mask(seq_len: int) -> Tensor:
    # 1 = can attend, 0 = cannot attend
    # np.tril creates a Lower Triangular matrix
    mask = np.tril(np.ones((seq_len, seq_len), dtype=np.float32))
    return Tensor(mask[np.newaxis, :, :])
```



The Complete GPT Architecture



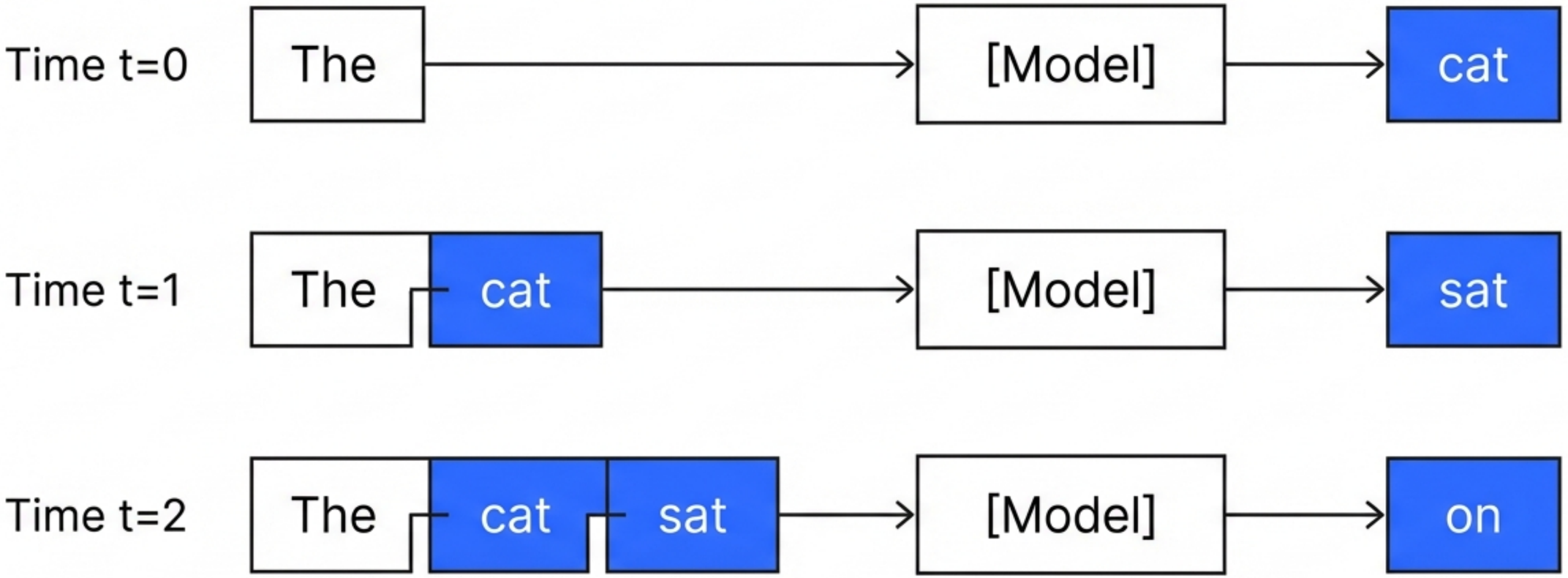
Critical Assembly Details:

- **Positional Embeddings:** Essential because Attention is permutation invariant.
- **Shared Weights:** Often Token Embeddings and Language Head share weights.
- **The Stack:** Deep repetition of the same block structure.

GPT Implementation: The Forward Pass

```
class GPT:
    def __init__(self, vocab_size, embed_dim, num_layers, ...):
        self.token_embedding = Embedding(vocab_size, embed_dim)
        self.position_embedding = Embedding(max_seq_len, embed_dim)
        self.blocks = [TransformerBlock(...) for _ in range(num_layers)]
        self.ln_f = LayerNorm(embed_dim)
        self.lm_head = Linear(embed_dim, vocab_size)

    def forward(self, tokens):
        seq_len = tokens.shape[1]
        pos = Tensor(np.arange(seq_len).reshape(1, seq_len))
        # Add position info to tokens
        x = self.token_embedding(tokens) + self.position_embedding(pos)
        mask = self._create_causal_mask(seq_len)
        # The Deep Stack
        for block in self.blocks:
            x = block(x, mask)

        return self.lm_head(self.ln_f(x))
```

The Generation Loop



Append & Repeat

Autoregressive: The output of t becomes the input of t+1.

Implementing Autoregressive Generation

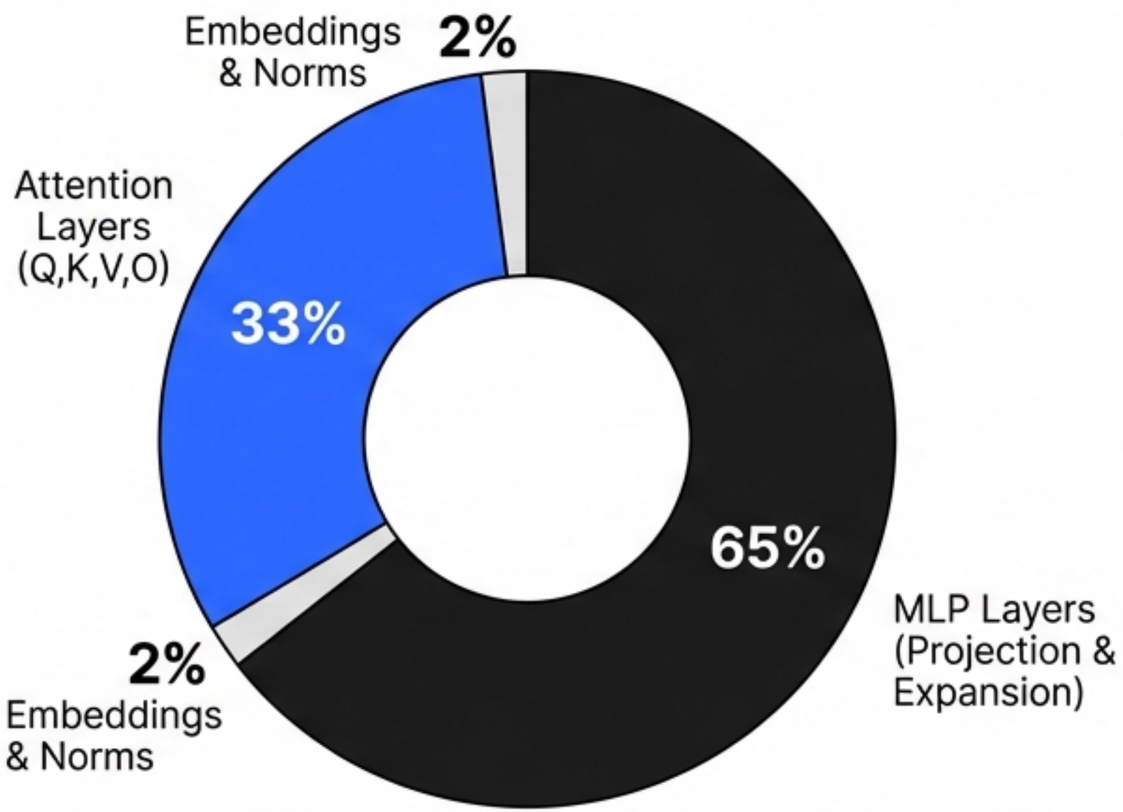
```python
def generate(self, prompt, max_new_tokens, temperature=1.0):
    current = prompt
    for _ in range(max_new_tokens):
        # 1. Forward Pass
        logits = self.forward(current)

        # 2. Focus on last token & Scale
        last_logits = logits.data[:, -1, :] / temperature

        # 3. Softmax & Sample
        probs = softmax(last_logits)
        next_token = sample(probs)

        # 4. Append
        current = concatenate([current, next_token])
    return current
```

Temperature > 1.0 =
More Random/Creative

Inefficient without KV Cache
(recomputing history)

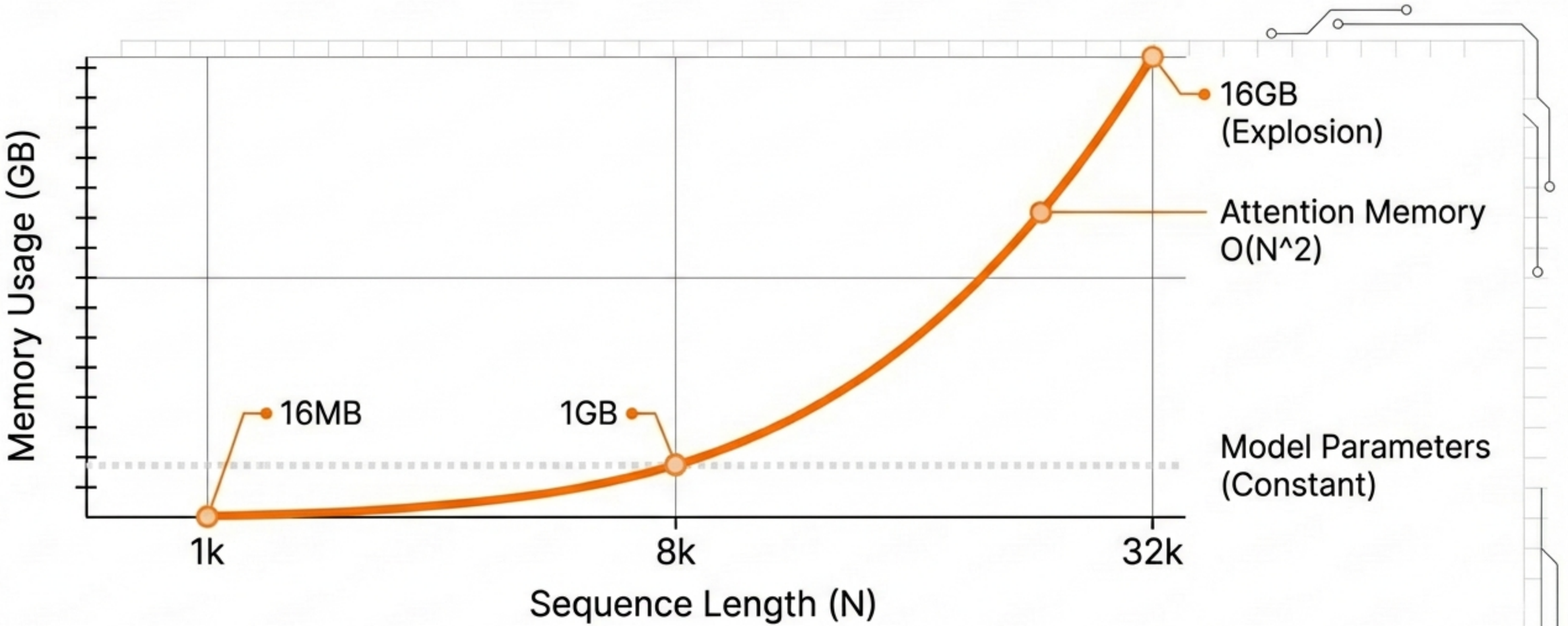
Systems Analysis: Parameter Scaling

Where do the weights live?



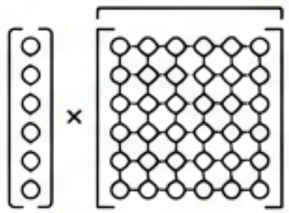
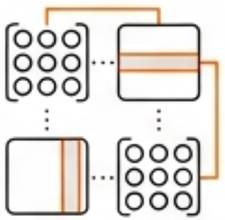
Standard Block (Embed=768)		
MLP Params	~4.7 Million	Due to 4x expansion
Attn Params	~2.3 Million	4 matrices of 768x768
Total Block	~7.0 Million	

The Attention Memory Wall



Doubling context length quadruples memory usage.

TinyTorch vs. Production (PyTorch)

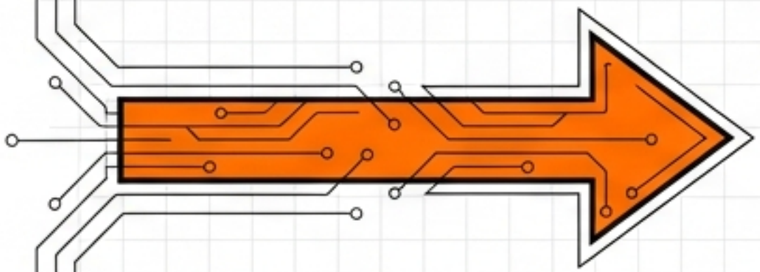
Feature	TinyTorch (You)		Production (PyTorch)
Architecture	Pre-Norm Block	Identical →	Pre-Norm Block
Logic	Autoregressive Masking	Identical →	Autoregressive Masking
Execution	Python Loops `for i in range(N):...`	Speed Difference →	Fused CUDA Kernels `__global__ void fused_attn...`
Attention Math	Full Matrix O(N^2) 	Memory Optimized →	FlashAttention (Memory Optimized) 

You have built the correct logic. Production frameworks optimize the math speed.

What You Have Built

- [x] **LayerNorm:** Stabilization
- [x] **MLP:** Processing Capacity
- [x] **TransformerBlock:** The Composable Unit
- [x] **GPT:** The Generative System

Coming Up: Module 14



Profiling

Now that we've built it, let's find out exactly why it's slow.