# Transformers Architecture

## From Self-Attention to Large-Scale Foundation Models

### Introduction

The Transformer architecture revolutionized AI by replacing recurrence with self-attention, enabling parallelization and long-context understanding. It is the foundation of models like GPT, BERT, and Vision Transformers.

### Historical Evolution

Pre-transformer era relied on RNNs and LSTMs, which struggled with long dependencies. The paper 'Attention Is All You Need' (Vaswani et al., 2017) introduced self-attention as a new paradigm. Andrej Karpathy’s implementations (minGPT, nanoGPT) demonstrate this architecture end-to-end.

### High-Level Overview

The transformer consists of: 1. Embedding Layer, 2. Positional Encoding, 3. Multi-Head Self-Attention, 4. Feed Forward Network, 5. Layer Normalization & Residuals. Variants include Encoder-Only (BERT), Decoder-Only (GPT), and Encoder-Decoder (T5).

### Self-Attention Mechanism

Each token is projected into three vectors: Query (Q), Key (K), and Value (V). Attention is computed as Softmax(QK■ / √d■) × V. This allows each token to attend to all others simultaneously. Multi-head attention runs this process multiple times in parallel to learn richer relationships.

### Code Example (from Karpathy's nanoGPT)

```
import torch.nn as nn
self.attn = nn.MultiheadAttention(embed_dim, num_heads, dropout=0.1)
attn_output, _ = self.attn(x, x, x)
```

### Positional Encoding

Transformers lack recurrence, so position must be injected explicitly. Original sinusoidal encoding: PE(pos,2i)=sin(pos/10000^(2i/d)), PE(pos,2i+1)=cos(pos/10000^(2i/d)). Learned embeddings (as used in GPT) are now common for efficiency.

### Transformer Encoder Block

Each encoder layer includes:
- Multi-Head Self-Attention
- Add & Norm
- Feed Forward Network (two linear layers with ReLU/GELU)
- Another Add & Norm. Residual connections stabilize deep networks and improve

gradient flow.

## Decoder Block

Decoder has masked self-attention to prevent looking at future tokens. If used in seq2seq tasks, cross-attention attends over encoder outputs. GPT-style models use only the decoder block with causal masks.

## Feed Forward Network (FFN)

Typically two linear layers with nonlinearity: FFN(x) = max(0, xW■ + b■)W■ + b■. Hidden layer is usually 4× embedding size. This step enables non-linear transformations between attention layers.

## Layer Normalization & Residual Connections

Add & Norm pattern ensures training stability. Residuals help maintain information across layers; normalization prevents exploding gradients.

## Training and Parallelization

Transformers are trained on large corpora using teacher forcing. Self-attention allows full-sequence parallelism, enabling massive scaling on GPUs/TPUs. Techniques like gradient checkpointing, mixed precision, and distributed training (e.g., DeepSpeed) are key for efficiency.

## Karpathy's Perspective

Karpathy emphasizes simplicity — building transformers from scratch using PyTorch to understand each component. His minGPT repository breaks the model into a few hundred lines for clarity. He advocates for inspecting attention matrices and understanding gradients intuitively.

## Optimization Insights (From Stanford CS25)

Stanford’s course highlights scaling laws (model size $\propto$ performance), and shows how compute efficiency and dataset curation critically impact downstream generalization.

## Variants & Extensions

• Vision Transformers (ViT) – treat image patches as tokens
• Efficient Transformers – Linformer, Performer, Reformer
• Mixture-of-Experts (MoE) – sparse activation of sub-models
• Retrieval-Augmented Transformers – integrate external knowledge bases

## Applications Beyond NLP

Transformers are now used in vision, audio, biology (AlphaFold), time series forecasting, and multi-modal learning. Stanford CS25 demos illustrate unified architectures for text, image, and reinforcement learning tasks.

## Strengths & Limitations

Strengths:
- Parallelism in training
- Captures long-range dependencies
- Generalizes across domains

Limitations:
- Quadratic complexity with sequence length
- Data and compute intensive
- Lack of inductive bias for spatial data

## Implementation Pseudocode (Simplified)

```
def transformer_block(x):
  q,k,v = linear(x), linear(x), linear(x)
  attn = softmax(q@k.T/sqrt(d))*v
  x = x + attn
  x = layer_norm(x)
  ff = relu(linear(x))
  return layer_norm(x + ff)
```

## Efficiency Improvements

Recent methods include:
- Sparse attention (Longformer, BigBird)
- Low-rank factorization
- Quantization and pruning
- Flash Attention (optimized CUDA kernels)

These reduce compute cost and memory footprint.

## Scaling Laws & Large Models

Empirical scaling laws (OpenAI, DeepMind): performance improves predictably with data, compute, and parameters. This motivates large models like GPT-4 and PaLM. However, diminishing returns appear beyond optimal scaling regimes.

## Explainability and Visualization

Attention heatmaps reveal which tokens the model focuses on. Layer-wise relevance propagation and attribution methods help interpret transformer behavior.

## Future of Transformer Architecture

Expect hybrid designs combining CNNs, Graph Networks, and Transformers. Focus areas: efficiency, interpretability, and domain adaptation for edge deployment.

## Summary

Transformers unify sequence modeling through attention. From Karpathy’s minimalist GPT to Stanford’s multi-domain applications, the architecture continues to evolve. Mastering attention mechanics and optimization is key to building next-gen AI systems.

## Q&A; / Discussion

Prompt: How would you design a transformer variant for edge deployment while balancing accuracy, size, and latency?