The Transformer Architecture

Chapter 5: Attention Is All You Need - Complete Guide

NLP Course 2025

September 24, 2025

Comprehensive Coverage of Transformer Architecture
45 slides covering theory, implementation, and applications

Learning Objectives

What You Will Master Today

Theoretical Understanding

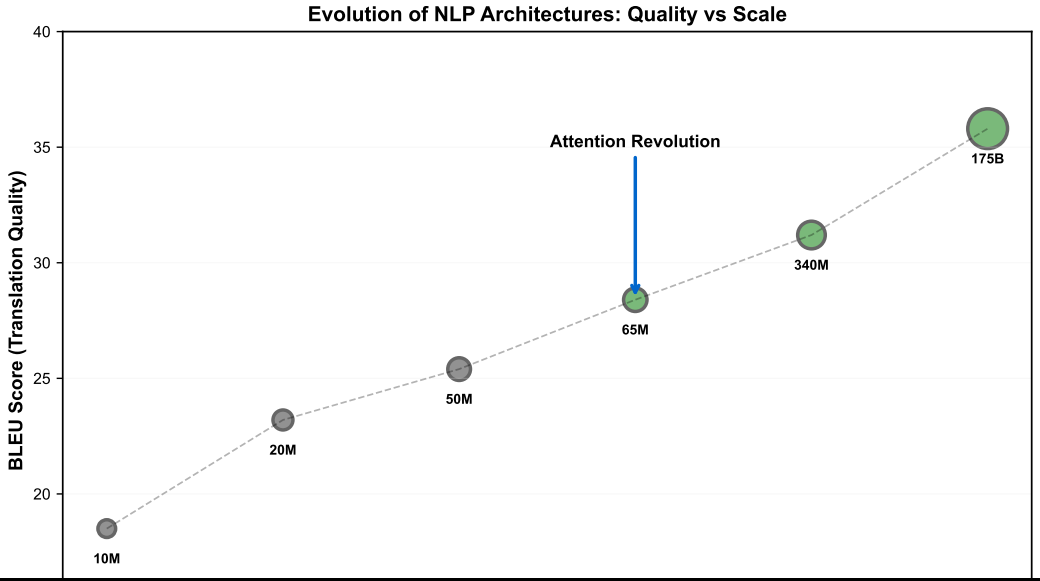
- Self-attention mechanism
- Multi-head attention
- Positional encoding
- Layer normalization
- Complete architecture

Practical Skills

- Implement attention from scratch
- Build a mini-transformer
- Debug attention weights
- Optimize for performance
- Apply to real problems

By the end: You'll understand how ChatGPT, BERT, and all modern LLMs work

The Journey to Transformers



The Sequential Processing Problem

Why RNNs Hit a Wall:

Sequential Processing

- Must process word 1 before word 2
- Can't parallelize across sequence
- Training time: $O(T)$ where T = length
- GPUs sit idle most of the time

Result: Weeks to train large models

Information Bottleneck

- All history compressed into one vector
- Long sequences lose information
- Gradient vanishing/exploding
- Can't capture long-range dependencies

Result: Poor performance on long texts

We needed a completely different approach

The Transformer Revolution (2017-2024)

One Architecture Changed Everything

Training Speed

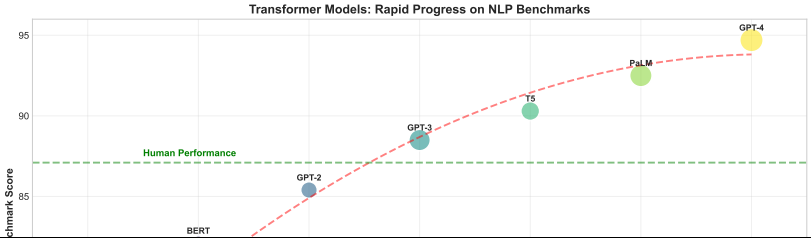
- 100x faster than RNNs
- Full parallelization
- Days not weeks
- Scales with hardware

Performance

- SOTA on all benchmarks
- Better long-range deps
- Transfer learning works
- Multimodal capabilities

Applications

- ChatGPT (175B params)
- Google Search (BERT)
- GitHub Copilot
- DALL-E, Whisper, etc.



Part 2

Core Concepts

Understanding Self-Attention

Self-Attention: The Core Innovation

The Brilliant Insight:

"Let every word directly look at every other word to decide what's relevant"

Example: "The cat sat on the mat because it was tired"

When processing "it":

- Traditional RNN: Only sees previous hidden states sequentially
- **Self-attention**: Directly examines all words:
  - "it" strongly attends to "cat" (0.7 weight)
  - "it" weakly attends to "mat" (0.1 weight)
  - "it" moderately attends to "tired" (0.2 weight)

Every word builds its representation by looking at all other words

This happens for ALL words SIMULTANEOUSLY - that's the magic!

The Attention Formula

Scaled Dot-Product Attention

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

Step-by-step computation:

1. **Create Q, K, V:** Linear projections of input embeddings
$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$
2. **Compute attention scores:** How much should word i attend to word j?
$$\text{score}(i, j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$
3. **Apply softmax:** Convert scores to probability distribution
$$\alpha_{ij} = \frac{\exp(\text{score}(i, j))}{\sum_k \exp(\text{score}(i, k))}$$
4. **Weighted sum:** Combine values using attention weights

Understanding Query, Key, and Value

The Information Retrieval Analogy:

Query (Q)

- "What am I looking for?"
- The current word's search vector
- Dimension: d_k
- Learned through W^Q

Example: "bank" asks "Am I financial or river-related?"

Key (K)

- "What do I contain?"
- Each word's content descriptor
- Dimension: d_k
- Learned through W^K

Example: "river" advertises "I'm about water"

Value (V)

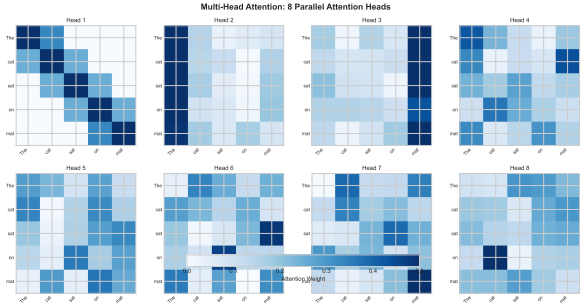
- "What information do I provide?"
- The actual content to aggregate
- Dimension: d_v
- Learned through W^V

Example: "river" provides its semantic content

Q and K determine attention weights; V provides the actual information

Visualizing Attention Patterns

Attention Weights for: "The cat sat on the mat"



What the Matrix Shows:

- Darker = stronger attention
- Row i = where word i looks
- Column j = who attends to word j

Key Observations:

- "cat" and "sat" strongly connected
- "on" attends to "mat"
- Articles attend broadly
- Diagonal = self-attention

Attention patterns reveal semantic relationships

Real transformers have multiple attention heads capturing different relationships

Multi-Head Attention: Multiple Perspectives

Why One Head Isn't Enough:

Different heads capture different relationships:

- Head 1: Syntactic dependencies (subject-verb)
- Head 2: Coreference resolution (pronouns)
- Head 3: Semantic similarity
- Head 4: Positional patterns
- ...

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

where each head is:

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

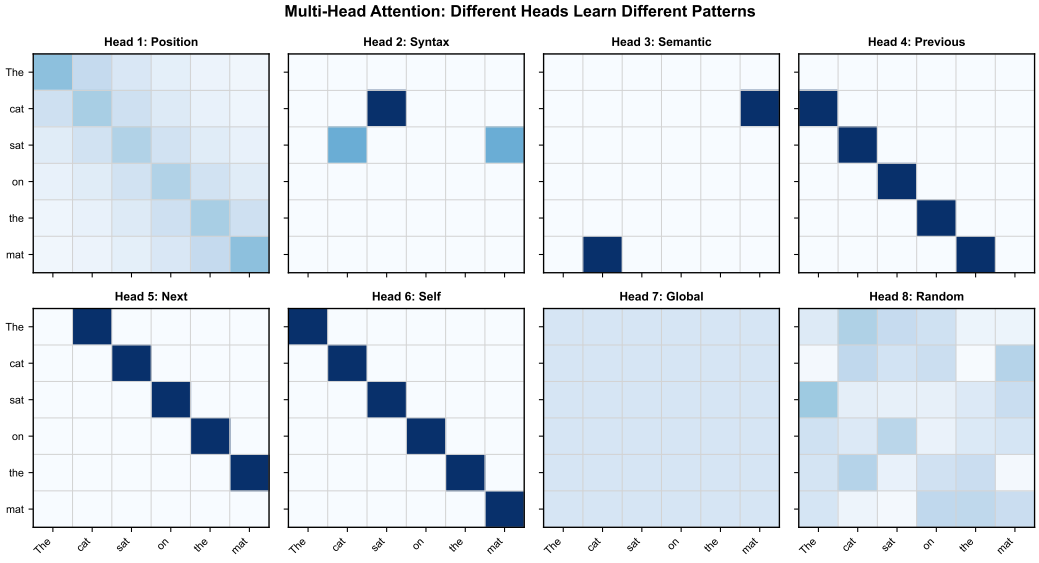
Typical Configuration:

- 8-16 heads in practice
- Each head: $d_k = d_{model}/h$
- Parallel computation

Benefits:

- Richer representations
- Redundancy and robustness
- Different abstraction levels

Multi-Head Attention in Action



The Position Problem

Attention Has No Notion of Order!

Without position information, these are identical to self-attention:

- "The cat chased the mouse"
- "The mouse chased the cat"
- "Cat the chased mouse the"

Self-attention is permutation invariant - order doesn't matter!

The Solution: Positional Encoding

- Add position information to embeddings
- Must be deterministic and smooth
- Should generalize to unseen lengths
- Preserve relative position information

We inject position information before attention computation

Positional Encoding: Sinusoidal Functions

The Elegant Solution:

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

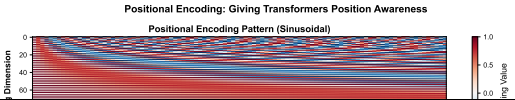
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where:

- pos = position in sequence (0, 1, 2, ...)
- i = dimension index
- d_{model} = model dimension (e.g., 512)

Why Sinusoids?

- Unique pattern for each position
- Smooth interpolation
- Can extrapolate to longer sequences



Layer Normalization: Stabilizing Training

Why Normalization is Critical:

- Deep networks suffer from internal covariate shift
- Attention can produce varying magnitude outputs
- Gradients can vanish or explode
- Training becomes unstable

Layer Normalization:

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sigma + \epsilon} + \beta$$

where:

- μ = mean across features for each sample
- σ = standard deviation across features
- γ, β = learned scale and shift parameters
- ϵ = small constant for numerical stability

Applied after each sub-layer (attention and feed-forward)

Residual Connections: Highway for Gradients

The Deep Network Challenge:

Transformers are DEEP (12-24+ layers). Without residuals:

- Gradients vanish exponentially
- Information gets lost
- Training becomes impossible

Residual Connections to the Rescue:

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

Benefits:

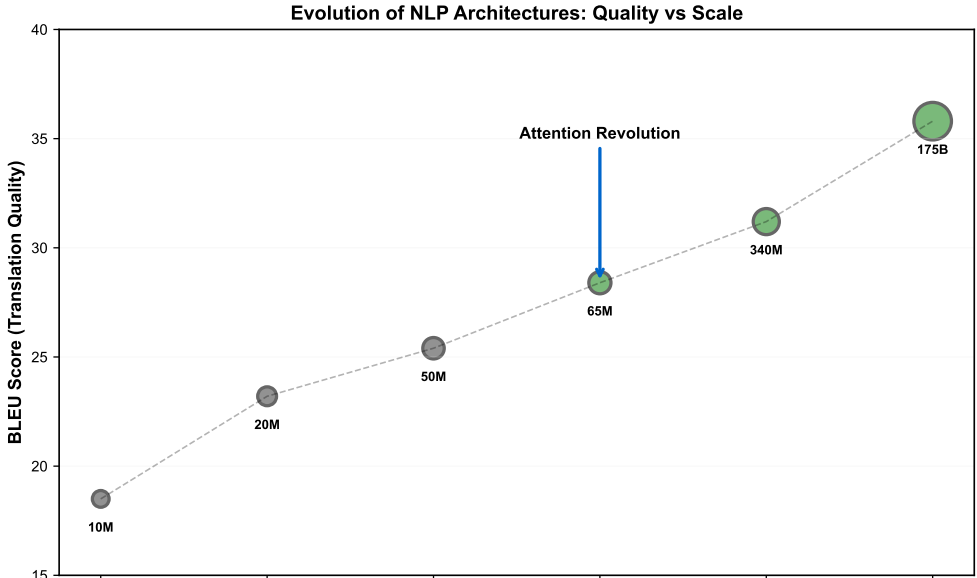
- Direct gradient path
- Preserves information
- Enables very deep networks
- Faster convergence

Implementation:

```
# Attention with residual
attn_out = attention(x)
x = layer_norm(x + attn_out)

# FFN with residual
ffn_out = feed_forward(x)
x = layer_norm(x + ffn_out)
```

The Complete Transformer Architecture



Inside the Encoder

Each Encoder Layer Contains:

1. **Multi-Head Self-Attention**
   - All positions attend to all positions
   - Parallel computation for all words
   - 8-16 attention heads typically

2. **Position-wise Feed-Forward Network**

```
FFN(x) = max(0, xW1 + b1)W2 + b2
```

   - Applied to each position separately
   - Hidden dimension typically 4x model dimension
   - ReLU or GELU activation

3. **Residual Connections** around each sublayer
4. **Layer Normalization** after each sublayer

Dimensions (BERT-base example):

- Model dimension: 768
- Feed-forward dimension: 3072

Inside the Decoder

Three Sublayers per Decoder Layer:

1. **Masked Multi-Head Self-Attention**
   - Prevents looking at future tokens
   - Ensures autoregressive property
   - Critical for generation tasks

2. **Encoder-Decoder Attention**
   - Queries from decoder
   - Keys and Values from encoder output
   - Allows decoder to focus on relevant input

3. **Position-wise Feed-Forward**
   - Same as encoder FFN
   - Independent processing per position

Masking ensures we can't "cheat" during training by looking ahead

GPT models are decoder-only with masked self-attention

Cross-Attention: Connecting Encoder and Decoder

How Decoder Attends to Encoder:

The Mechanism:

- **Q** comes from decoder
- **K, V** come from encoder
- Each decoder position can attend to all encoder positions
- No masking needed here

Translation Example:

- Input (English): "I love cats"
- Generating (French): "J'aime les..."
- When generating "chats":
  - Q from "les" position
  - Attends strongly to "cats"
  - Weakly to other words

$$\text{CrossAttn}(Q_{dec}, K_{enc}, V_{enc})$$

Cross-attention allows decoder to focus on relevant input parts

This is how the model aligns source and target sequences

Part 3

Implementation Details

Building Transformers in Practice

Implementing Self-Attention from Scratch

```python
import torch
import torch.nn.functional as F

class SelfAttention(torch.nn.Module):
    def __init__(self, d_model, d_k):
        super().__init__()
        self.W_q = torch.nn.Linear(d_model, d_k)
        self.W_k = torch.nn.Linear(d_model, d_k)
        self.W_v = torch.nn.Linear(d_model, d_k)
        self.scale = d_k ** 0.5

    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        Q = self.W_q(x)  # (batch, seq_len, d_k)
        K = self.W_k(x)
        V = self.W_v(x)

        # Compute attention scores
        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
        # scores shape: (batch, seq_len, seq_len)

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

        # Apply weights to values
        output = torch.matmul(attn_weights, V)
        return output, attn_weights
```

Implementing Multi-Head Attention

```python
class MultiHeadAttention(torch.nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0
        self.d_k = d_model // n_heads
        self.n_heads = n_heads

        # Linear projections for all heads at once
        self.W_q = torch.nn.Linear(d_model, d_model)
        self.W_k = torch.nn.Linear(d_model, d_model)
        self.W_v = torch.nn.Linear(d_model, d_model)
        self.W_o = torch.nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        batch_size, seq_len, d_model = x.shape

        # Project and reshape for multi-head
        Q = self.W_q(x).view(batch_size, seq_len, self.n_heads, self.d_k)
        K = self.W_k(x).view(batch_size, seq_len, self.n_heads, self.d_k)
        V = self.W_v(x).view(batch_size, seq_len, self.n_heads, self.d_k)

        # Transpose for attention: (batch, n_heads, seq_len, d_k)
        Q = Q.transpose(1, 2)
        K = K.transpose(1, 2)
        V = V.transpose(1, 2)

        # Attention for all heads in parallel
```

Implementing Positional Encoding

```python
import numpy as np

class PositionalEncoding(torch.nn.Module):
    def __init__(self, d_model, max_seq_len=5000):
        super().__init__()

        # Create positional encoding matrix
        pe = torch.zeros(max_seq_len, d_model)
        position = torch.arange(0, max_seq_len).unsqueeze(1).float()

        # Create div_term for the sinusoidal pattern
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                             -(np.log(10000.0) / d_model))

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

        # Register as buffer (not a parameter)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        seq_len = x.size(1)
        # Add positional encoding
        x = x + self.pe[:, :seq_len]
```

The Feed-Forward Network

Position-wise FFN: Simple but Crucial

```python
class FeedForward(torch.nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.linear1 = torch.nn.Linear(d_model, d_ff)
        self.linear2 = torch.nn.Linear(d_ff, d_model)
        self.dropout = torch.nn.Dropout(dropout)
        self.activation = torch.nn.ReLU()

    def forward(self, x):
        # Expand to higher dimension
        x = self.linear1(x)       # (batch, seq_len, d_ff)
        x = self.activation(x)    # Non-linearity
        x = self.dropout(x)       # Regularization
        # Project back to model dimension
        x = self.linear2(x)       # (batch, seq_len, d_model)
        return x
```

Why Important:

- Adds non-linearity (attention is linear)
- Increases model capacity

Training Transformers: Key Considerations

Optimization Challenges:

Learning Rate Schedule

- Warmup is critical
- Linear increase for first steps
- Then decay by $1/\sqrt{step}$
- Prevents early instability

$$lr = d_{model}^{-0.5} \cdot \min(step^{-0.5}, step \cdot warmup^{-1.5})$$

Regularization

- Dropout (typically 0.1)
- Weight decay
- Label smoothing
- Gradient clipping

Batch Size

- Larger is better (if fits)
- Gradient accumulation
- Typically 4k-64k tokens

Training is unstable without proper warmup and initialization

Computational Complexity Analysis

Time and Space Complexity:

Component	Time	Space	Bottleneck
Self-Attention	$O(n^2 \cdot d)$	$O(n^2)$	Quadratic in length
Feed-Forward	$O(n \cdot d^2)$	$O(1)$	Linear in length
Multi-Head Proj	$O(n \cdot d^2)$	$O(1)$	Model dimension
Total per Layer	$O(n^2 \cdot d + n \cdot d^2)$	$O(n^2)$	

where n = sequence length, d = model dimension

Practical Implications:

- Memory grows quadratically with sequence length
- Long sequences (¿2k tokens) become problematic
- Modern solutions: Sparse attention, Flash Attention
- Trade-off: Parallelization vs memory usage

The $O(n^2)$ attention is both the strength and limitation

Memory Requirements in Practice

Where Does the Memory Go?

For BERT-base (110M parameters):

- **Model Parameters:** 440 MB (fp32)
- **Optimizer States:** 880 MB (Adam - 2x params)
- **Gradients:** 440 MB
- **Activations:** Depends on batch size and sequence length

Activation Memory for Batch Size 32, Seq Length 512:

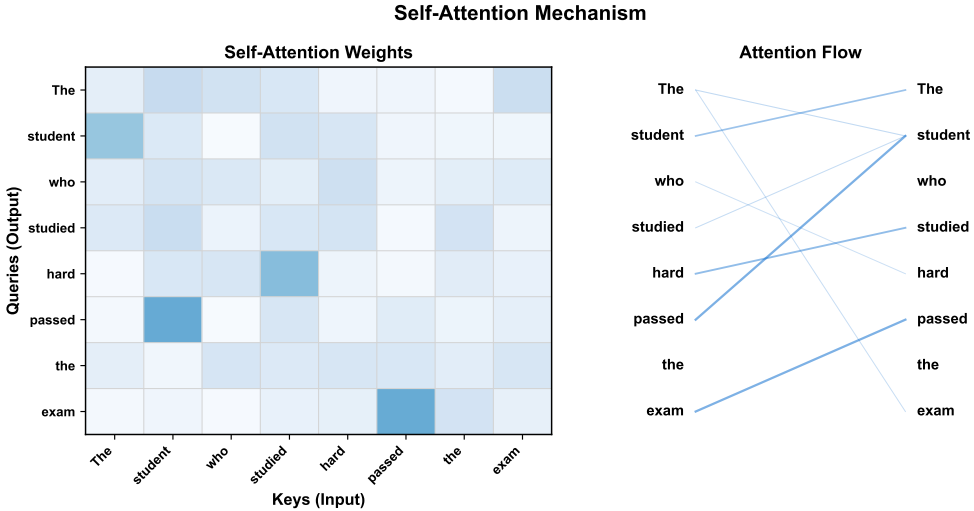
- Attention matrices: $32 \times 12 \times 512 \times 512 \times 4$ bytes $= 402$ MB
- FFN activations: $32 \times 512 \times 3072 \times 4$ bytes $= 201$ MB
- Per layer: 600 MB
- 12 layers: 7.2 GB just for activations!

Memory, not compute, is often the limiting factor

Techniques: Gradient checkpointing, mixed precision, model parallelism

Debugging Attention Patterns

What Should Attention Look Like?



Common Implementation Pitfalls

Mistakes That Cost Days of Debugging:

1. **Forgetting the Scaling Factor**
   - Without $1/\sqrt{d_k}$: Softmax saturates
   - Gradients vanish, model doesn't learn

2. **Wrong Mask Implementation**
   - Must add -inf BEFORE softmax
   - Boolean mask vs additive mask confusion

3. **Positional Encoding Bugs**
   - Not adding to embeddings
   - Wrong dimension ordering
   - Not handling variable lengths

4. **Layer Norm Placement**
   - Pre-norm vs post-norm matters
   - Modern models use pre-norm for stability

Always verify shapes and attention patterns early in training

Part 4

Applications & Impact

From BERT to GPT-4

BERT: Bidirectional Encoder Representations

The Encoder-Only Revolution (2018):

Architecture:

- Encoder-only transformer
- Bidirectional context
- 12/24 layers (Base/Large)
- 110M/340M parameters

Key Innovations:

- Bidirectional pre-training
- Fine-tuning paradigm
- CLS token for classification
- Segment embeddings

Training Objective:

- Masked Language Modeling (MLM)
- Randomly mask 15% of tokens
- Predict masked tokens
- Next Sentence Prediction (NSP)

Impact:

- SOTA on 11 NLP tasks
- Powers Google Search
- Started fine-tuning era
- 50,000+ citations

BERT showed that pre-training + fine-tuning is incredibly powerful

GPT: The Decoder-Only Approach

Generative Pre-trained Transformer Evolution:

Architecture:

- Decoder-only (masked attention)
- Autoregressive generation
- Left-to-right only
- Scales to extreme sizes

Training Approach:

- Next token prediction
- Massive text corpora
- No fine-tuning needed
- In-context learning

Model Progression:

- GPT (2018): 117M params
- GPT-2 (2019): 1.5B params
- GPT-3 (2020): 175B params
- GPT-4 (2023): 1.7T params*

Capabilities:

- Text generation
- Few-shot learning
- Code generation
- Reasoning tasks

GPT proved that scale + simple objective = emergent abilities

Encoder-Only vs Decoder-Only vs Both

Which Architecture When?

Aspect	Encoder-Only	Decoder-Only	Both
Example	BERT, RoBERTa	GPT, LLaMA	T5, BART
Context	Bidirectional	Left-to-right	Flexible
Best for	Understanding	Generation	Translation
Training	MLM	Next token	Varies
Fine-tuning	Required	Optional	Optional
Speed	Fast inference	Slower (sequential)	Slowest
Parameters	Smaller	Can be huge	Medium

Use Cases:

Encoder-Only:

- Classification
- NER, POS tagging
- Sentiment analysis
- Search/retrieval

Decoder-Only:

- Text generation
- Dialogue systems
- Code completion
- Creative writing

Encoder-Decoder:

- Translation
- Summarization
- Question answering
- Text-to-text tasks

Beyond Text: Vision Transformers (ViT)

Transformers Conquer Computer Vision (2020):

Key Idea:

- Divide image into patches (16×16)
- Treat patches as "tokens"
- Add positional embeddings
- Standard transformer encoder

CLS token for classification

Why It Works:

- Global receptive field from start
- Flexible attention patterns
- No inductive bias of convolution
- Scales better than CNNs

Results:

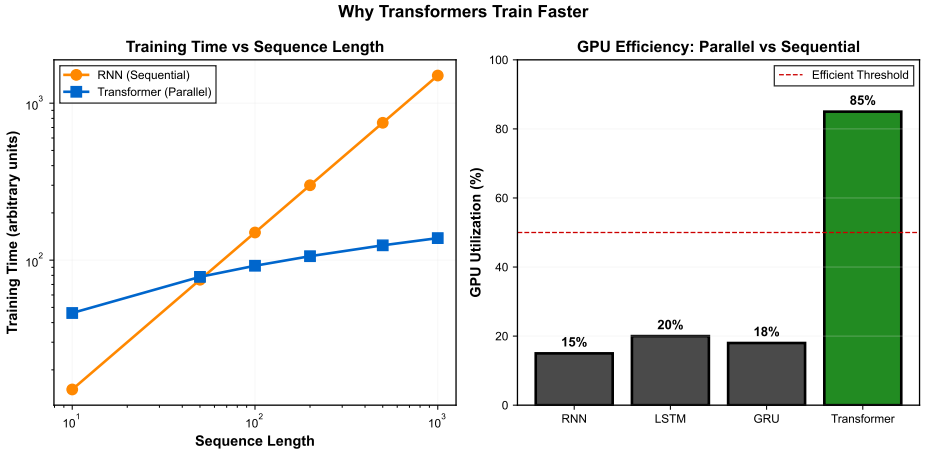
- Beats CNNs at scale
- ImageNet: 88.5% accuracy
- Needs lots of data
- Better transfer learning

Extensions:

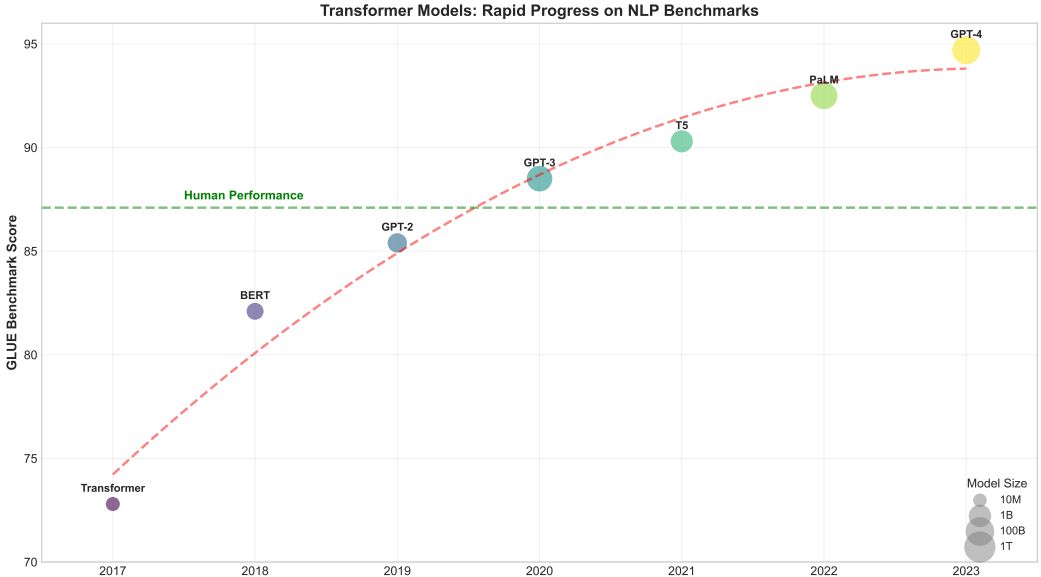
- CLIP: Vision + Language
- DALL-E: Text to Image
- Flamingo: Visual QA
- SAM: Segmentation

Multimodal Transformers: Unified Architecture

One Architecture to Process Everything:



Performance Evolution: Transformers Dominate



Scaling Laws: Bigger is (Usually) Better

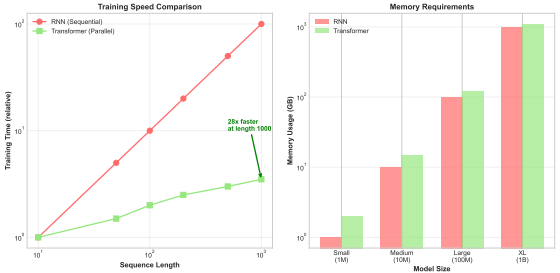
The Kaplan Scaling Laws (2020):

$$\text{Loss} = aN^{-\alpha} + bD^{-\beta} + c$$

where N = parameters, D = data, $\alpha \approx 0.076$, $\beta \approx 0.095$

Key Findings:

- Performance improves predictably
- No plateau in sight
- Data and compute equally important
- Optimal model size grows with budget



Modern Transformer Variants

Innovations to Address Limitations:

Efficiency Improvements:

- **Flash Attention:** 2-4x faster, less memory
- **Sparse Transformers:** $O(n\sqrt{n})$ complexity
- **Linformer:** $O(n)$ attention
- **Performer:** Kernel-based attention

Training Innovations:

- **RoPE:** Better positional encoding
- **ALiBi:** Extrapolate to longer sequences
- **FlashAttention-2:** Even faster
- **GQA:** Grouped query attention

Architectural Changes:

- **Mixture of Experts:** Conditional computation
- **Retrieval-Augmented:** External memory
- **Toolformer:** API calling ability

Current State-of-the-Art:

- LLaMA 2 architecture
- RMSNorm instead of LayerNorm
- SwiGLU activation
- Rotary embeddings

Innovation continues at breakneck pace

Real-World Impact: Transformers Everywhere

Applications Powered by Transformers (2024):

Language:
- ChatGPT
- Google Bard
- Claude
- DeepL
- Grammarly

Code:
- GitHub Copilot
- Replit AI
- Amazon CodeWhisperer
- Tabnine

Creative:
- DALL-E 3
- Midjourney
- Stable Diffusion
- MuseNet

Science:
- AlphaFold
- ESMFold
- Galactica
- BioGPT

Industry Adoption:
- **Microsoft:** Copilot in Office 365
- **Google:** Bard, Search, Workspace
- **Meta:** LLaMA, Make-A-Video
- **Every Tech Company:** Building transformer-based products

Transformers are the foundation of the AI revolution

Part 5

Hands-On & Summary

Bringing It All Together

Let's Build a Complete Mini-Transformer

```python
class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.attention = MultiHeadAttention(d_model, n_heads)
        self.norm1 = nn.LayerNorm(d_model)
        self.ff = FeedForward(d_model, d_ff, dropout)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Self-attention with residual
        attn_out = self.attention(x, mask)
        x = self.norm1(x + self.dropout(attn_out))
        # Feed-forward with residual
        ff_out = self.ff(x)
        x = self.norm2(x + self.dropout(ff_out))
        return x

class MiniTransformer(nn.Module):
    def __init__(self, vocab_size, d_model, n_layers, n_heads, max_len=5000):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoding = PositionalEncoding(d_model, max_len)
        self.blocks = nn.ModuleList([
            TransformerBlock(d_model, n_heads, d_model*4)
            for _ in range(n_layers)
        ])
        self.ln_f = nn.LayerNorm(d_model)
        self.output = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        x = self.embedding(x)
        x = self.pos_encoding(x)
        for block in self.blocks:
            x = block(x)
        x = self.ln_f(x)
```

What to Remember

1. **Core Innovation:** Self-attention enables parallel processing
2. **Key Components:**
   - Multi-head attention (different relationships)
   - Positional encoding (order information)
   - FFN (non-linearity and capacity)
   - Residuals & LayerNorm (training stability)
3. **Advantages:** Parallelization, long-range dependencies, transfer learning
4. **Limitations:** Quadratic complexity, memory intensive
5. **Impact:** Powers essentially all modern NLP and beyond
6. **Future:** Longer context, efficiency, multimodal, reasoning

Transformers are the foundation you need to understand modern AI

Common Misconceptions

Let's Clear These Up:

1. "Transformers are just attention"
   Reality: FFN, residuals, and layer norm are equally critical

2. "Bigger is always better"
   Reality: Depends on task, data, and computational budget

3. "Transformers understand language"
   Reality: They learn statistical patterns, not meaning

4. "Attention weights show what the model thinks"
   Reality: They're one part of a complex computation

5. "Transformers made RNNs obsolete"
   Reality: RNNs still useful for streaming, small models

Practice Problems

Test Your Understanding:

1. **Computation:** For sequence length 100, model dimension 512, 8 heads:
   - What's the dimension of each head?
   - How many parameters in multi-head attention?
   - Memory for attention matrices?

2. **Architecture:** Why do we need positional encoding? What happens without it?

3. **Implementation:** Write the attention masking for decoder self-attention

4. **Analysis:** Why is the scaling factor $\sqrt{d_k}$ and not d_k?

5. **Design:** How would you modify transformers for 10k+ token sequences?

Solutions will be discussed in the lab session

Resources for Deep Dive

Essential Papers:

- Vaswani et al. (2017): "Attention Is All You Need" - The original
- Devlin et al. (2018): "BERT: Pre-training of Deep Bidirectional Transformers"
- Radford et al. (2019): "Language Models are Unsupervised Multitask Learners" (GPT-2)
- Brown et al. (2020): "Language Models are Few-Shot Learners" (GPT-3)
- Dosovitskiy et al. (2020): "An Image is Worth 16x16 Words" (ViT)

Implementation Resources:

- **The Annotated Transformer:** Harvard NLP's line-by-line implementation
- **HuggingFace Transformers:** Industry standard library
- **nanoGPT:** Karpathy's minimal GPT implementation
- **x-transformers:** Cutting-edge variants

Visualization Tools:

- BertViz: Attention visualization
- Tensor2Tensor: Interactive transformer
- Papers with Code: Benchmark tracking

Thank You!

Questions?

Next: Lab Session - Implementing Transformers

We'll build and train a transformer from scratch