The Transformer Architecture

NLP Course 2025

September 30, 2025

A pedagogical exploration of the architecture that revolutionized AI

47 slides • 4 sections • 2 checkpoints

What You Will Master Today

Conceptual Understanding

- Why RNNs hit a computational wall
- How self-attention solves parallelization
- Multi-head attention intuition
- Position encoding necessity
- Architecture components

Practical Skills

- Implement attention mechanism
- Build mini-transformers
- Debug attention patterns
- Apply to real problems
- Understand modern LLMs

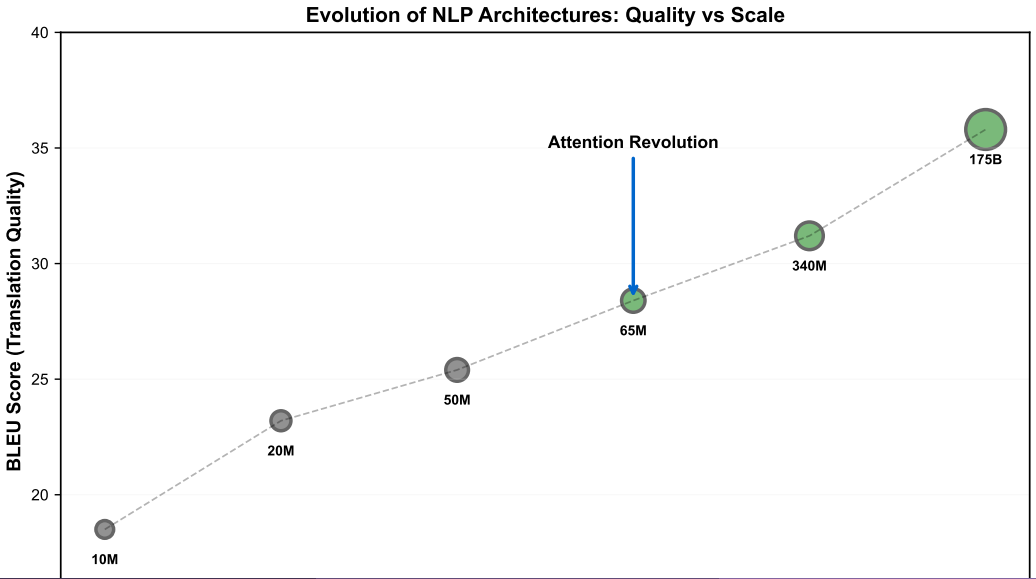
By the end: You'll understand exactly how ChatGPT, BERT, and modern AI work

We'll build understanding step-by-step, with checkpoints along the way

Section 1

The Challenge

Sequential Processing Bottleneck


Evolution of NLP Architectures: Quality vs Scale
BLEU Score (Translation Quality)
40
35
30
25
20
Attention Revolution
10M
20M
50M
65M
340M
175B

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

Warning: 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

Warning: Result: Poor performance on long texts

We needed a completely different approach - one that could process all words simultaneously

The sequential nature of RNNs is fundamentally incompatible with modern parallel hardware

Why This Problem Matters

Real-World Impact of Sequential Bottleneck:

Training Limitations

- LSTM on Wikipedia: 3 weeks
- Can't scale to internet-size data
- Limited model size (memory constraints)
- Expensive iteration cycles

Performance Ceiling

- Max context: 100 words effectively
- Document understanding impossible
- Multi-turn dialogue fails
- Cross-sentence reasoning limited

The Vision: What if we could...

- Process all words in parallel?
- Let every word see every other word directly?
- Scale to thousands of words context?
- Train models 100x faster?

The answer: Self-Attention - a mechanism that revolutionized NLP

Section 2

The Foundation

Self-Attention Mechanism

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 simultaneously

This parallel attention computation is what enables transformer speed

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?"
- 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?"
- Actual content to aggregate
- Dimension: d_v
- Learned through W^V

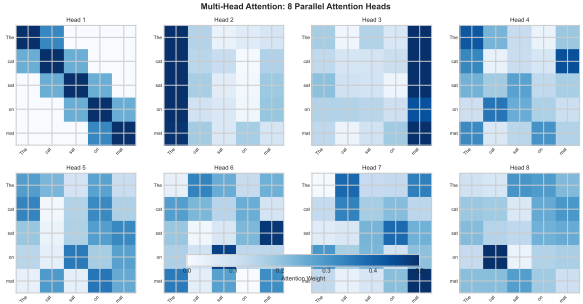
Example: "river" provides its semantic content

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

Think of it as a soft, differentiable database lookup

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 learned semantic and syntactic relationships

Real transformers have multiple heads capturing different relationship types

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
- Concatenated and projected

Benefits:

- Richer representations
- Redundancy and robustness
- Different abstraction levels
- Better generalization

Test Your Understanding

Quick Quiz:

Q1: Why do we scale by $\sqrt{d_k}$?

A) Make computation faster
B) Prevent softmax saturation
C) Reduce memory usage
D) Improve accuracy

Q2: What's the main advantage of self-attention over RNNs?

A) Less parameters
B) Better accuracy
C) Parallel processing
D) Simpler implementation

Answers:

A1: B - Prevent softmax saturation

- Large dot products $\rightarrow$ extreme softmax
- Scaling keeps values in reasonable range
- Critical for gradient flow

A2: C - Parallel processing

- All positions computed simultaneously
- No sequential dependencies
- Enables 100x faster training

A3: B - 64

- $d_k = d_{model}/h = 512/8 = 64$
- Each head gets equal dimension

Section 3

The Architecture

Building Complete Transformers

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"

Warning: 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:

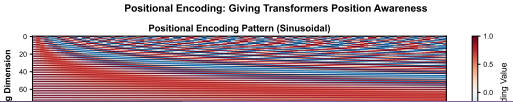
$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

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 without normalization

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

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

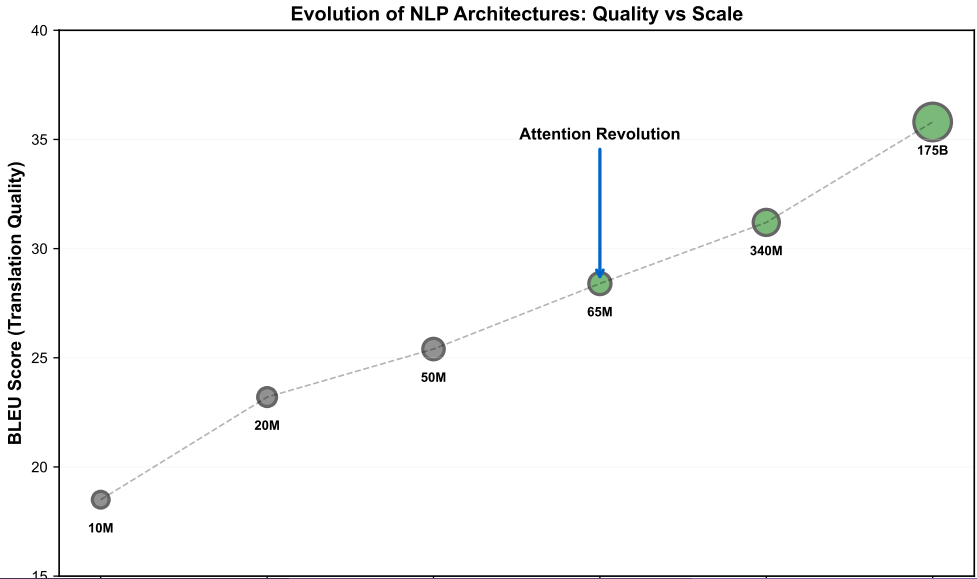
Implementation:

```
# 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)
```

Every sublayer in the transformer has a residual connection

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: 3072, Heads: 12, Layers: 12

The feed-forward network provides the non-linearity that attention lacks

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

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

GPT models are decoder-only with masked self-attention

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
```

Matrix multiplication enables parallel computation across all positions

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

Warning: Training is unstable without proper warmup and initialization

Modern training uses mixed precision and distributed training for efficiency

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

Test Your Architecture Knowledge

Architecture Quiz:

Q1: Why do we need positional encoding?

A) Speed up training
B) Self-attention is order-invariant
C) Reduce memory usage
D) Improve accuracy

Q2: What's the purpose of residual connections?

A) Reduce parameters
B) Speed up inference
C) Enable gradient flow
D) Save memory

Q3: Why is the FFN dimension typically 4x the model

Answers:

A1: B - Self-attention is order-invariant

- Attention treats input as a set
- Position encoding adds order info
- Critical for sequence understanding

A2: C - Enable gradient flow

- Direct path through layers
- Prevents vanishing gradients
- Allows very deep networks

A3: B - Add capacity and non-linearity

- Attention is linear transformation
- FFN adds non-linear processing

Section 4

The Revolution

Impact and Applications

One Architecture Changed Everything

Training Speed

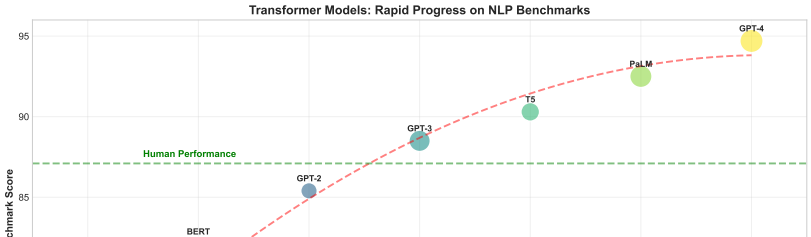
- **100x faster** than RNNs
- Full parallelization
- Days not weeks
- Scales with hardware

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



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

Choose architecture based on your task requirements

Beyond Text: Vision Transformers (ViT)

Transformers Conquer Computer Vision (2020):

Key Idea:

- Divide image into patches (16x16)
- 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

Scaling Laws: Bigger is (Usually) Better

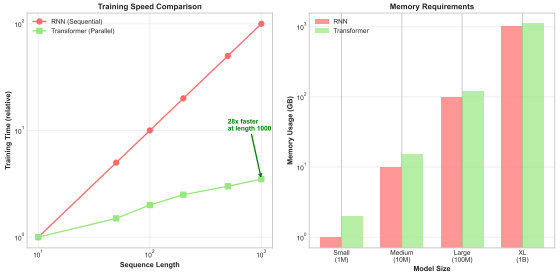
The Kaplan Scaling Laws (2020):

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

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
- **Sparse Transformers:** $O(n\sqrt{n})$
- **Linformer:** $O(n)$ attention
- **Performer:** Kernel-based

Training Innovations:

- **RoPE:** Better positions
- **ALiBi:** Longer sequences
- **FlashAttention-2:** Even faster
- **GQA:** Grouped queries

Architectural Changes:

- **MoE:** Conditional computation
- **Retrieval:** External memory
- **Toolformer:** API calling

Current SOTA:

- LLaMA 2 architecture
- RMSNorm
- SwiGLU activation
- Rotary embeddings

Innovation continues at breakneck pace

Real-World Impact: Transformers Everywhere

Applications Powered by Transformers (2024):

Language:
- ChatGPT
- Claude
- Gemini
- DeepL
- Grammarly

Code:
- GitHub Copilot
- Cursor
- 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 current AI revolution

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

Practice Problems

Deepen 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

Summary: The Journey We've Taken

From Challenge to Revolution:

The Challenge:

- RNNs' sequential bottleneck
- Couldn't parallelize
- Limited context
- Slow training

The Architecture:

- Complete transformer design
- Positional encoding
- Residuals and normalization
- Encoder/Decoder variants

The Foundation:

- Self-attention mechanism
- Query-Key-Value framework
- Multi-head perspectives
- Parallel processing

The Revolution:

- BERT, GPT, and beyond
- Multimodal capabilities
- Universal architecture
- AI transformation

Thank You!

Questions?

Next: Lab Session - Building Transformers

We'll implement a complete transformer from scratch