The Transformer Architecture

Week 5 - Attention Is All You Need

NLP Course 2025

September 20, 2025

Using Optimal Readability Template

The Architecture That Changed Everything

The Problem

- RNNs process sequentially
- Can't parallelize training
- Long-range dependencies fail
- Weeks to train large models

The Solution

- **Self-attention** mechanism
- Process all words in parallel
- Direct connections everywhere
- Days to train (not weeks!)

The Impact

- Powers ChatGPT, BERT
- 98% of modern NLP
- Enabled LLM revolution
- Multimodal AI foundation

Core Insight: Let every word attend to every other word directly

Why Google Couldn't Scale Translation Fast Enough

The RNN Bottleneck (2016):

To translate "I love machine learning":

1. Process "I" → wait →
2. Process "love" → wait →
3. Process "machine" → wait →
4. Process "learning" → done

RNNs must process words one at a time - can't parallelize!

The Cost:

- Training large models: Weeks to months
- Can't use modern GPUs effectively (built for parallel computation)
- Google needed 8,000 TPUs for production

Original transformer trained in 3.5 days vs weeks for RNNs

A Radical Idea: Remove RNNs Entirely

The 2017 Breakthrough:
"What if we use ONLY attention mechanisms?"

Revolutionary Insights:

1. Attention captures all relationships directly
2. No sequential processing needed
3. Every word sees every other word
4. Parallelization becomes trivial!

The Paper: Vaswani et al. (2017)
"Attention Is All You Need"
NeurIPS
Started the modern LLM era

The Impact:

- Training time: Weeks $\rightarrow$ Days
- BLEU score: 28.4 (EN-DE)
- Previous best: 25.2
- Spawned GPT, BERT, all LLMs

The Transformer: Process all words in parallel using attention

Key Innovation: Replace sequential processing with parallel attention computation

Transformers Power Everything (2024)

Language Models

- ChatGPT (GPT-4)
- Google Bard (Gemini)
- Claude (Anthropic)
- GitHub Copilot

Multimodal AI

- DALL-E (text $\rightarrow$ image)
- Whisper (speech $\rightarrow$ text)
- CLIP (vision-language)
- Flamingo (understanding)

Key Advantages

- 100x faster training
- Better long-range deps
- Transfer learning
- Scale to trillions

Search & Translation

- Google Search (BERT)
- DeepL Translator
- Every modern NMT

Code Generation

- Codex
- AlphaCode
- TabNine

98% of SOTA NLP uses transformers

The Genius of Self-Attention

How humans read "The cat sat on the mat":
When we see "sat", we instantly know:

- WHO sat? → look at "cat"
- WHERE? → look at "mat"
- **No sequential processing!**

Self-attention does exactly this:

1. Each word asks: "Who should I pay attention to?"
2. Computes attention scores with all other words
3. Creates weighted combination of relevant words
4. All happening simultaneously!

Self-attention = Each word decides what's relevant to it

Example: "The student who studied hard passed"

- "passed" attends to "student"
- "hard" attends to "studied"
- All connections in parallel

Every word connects to every other word directly

Self-Attention Mathematics: Elegantly Simple

The Attention Formula

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

For each word:

- **Q** (Query): "What am I looking for?"
- **K** (Key): "What do I contain?"
- **V** (Value): "What information do I provide?"

In Plain English:

1. Compare query with all keys
2. Scale by $\sqrt{d_k}$ (prevent saturation)
3. Softmax for weights
4. Weighted sum of values

Dot product measures similarity, softmax creates distribution

Building Self-Attention: Complete Implementation

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

class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads=8):
        super().__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        # Linear projections for Q, K, V
        self.queries = nn.Linear(embed_size, embed_size)
        self.keys = nn.Linear(embed_size, embed_size)
        self.values = nn.Linear(embed_size, embed_size)
        self.fc_out = nn.Linear(embed_size, embed_size)

    def forward(self, x, mask=None):
        N, seq_len, _ = x.shape

        # Project to Q, K, V
        Q = self.queries(x)
        K = self.keys(x)
        V = self.values(x)

        # Reshape for multi-head
```

Design Choices:

- 8 heads typical
- Head dim = 64 (512/8)
- Scaling prevents gradients

Multi-Head Benefits:

- Different heads learn different patterns
- One head: **syntax**
- Another: **semantics**
- Another: **position**

Multiple attention heads capture diverse relationships

Original paper used 8 heads. GPT-3 uses 96!

The Position Problem: Order Still Matters!

Self-attention has no notion of position!
These are identical to self-attention:

- "The cat sat on the mat"
- "Mat the on sat the cat"
- "Cat mat the the on sat"

The Solution: Positional Encoding

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d})$$

Why Sinusoids?

- Unique pattern for each position
- Can extrapolate to longer sequences
- Relative positions consistent
- No parameters to learn

Modern Alternatives:

- Learned embeddings (BERT)
- RoPE (LLaMA)
- ALiBi (attention bias)
- Relative encodings

The Transformer Block

```python
class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads,
                 dropout, forward_expansion):
        super().__init__()
        self.attention = SelfAttention(embed_size,
            heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size,
                      forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size,
                      embed_size)
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Self-attention with residual
        attention = self.attention(x, mask)
        x = self.dropout(self.norm1(attention + x))

        # Feed-forward with residual
        forward = self.feed_forward(x)
        out = self.dropout(self.norm2(forward + x))
```

Architecture (Base):

- 6 layers deep
- 512 embedding dim
- 2048 feed-forward
- Residual connections

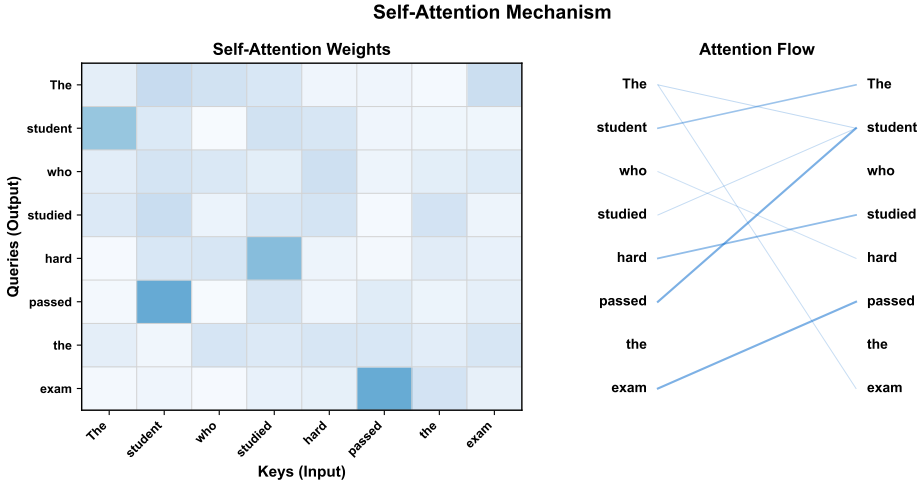
Why Residuals?

- Enable deep networks
- Gradient preservation
- Each layer refines

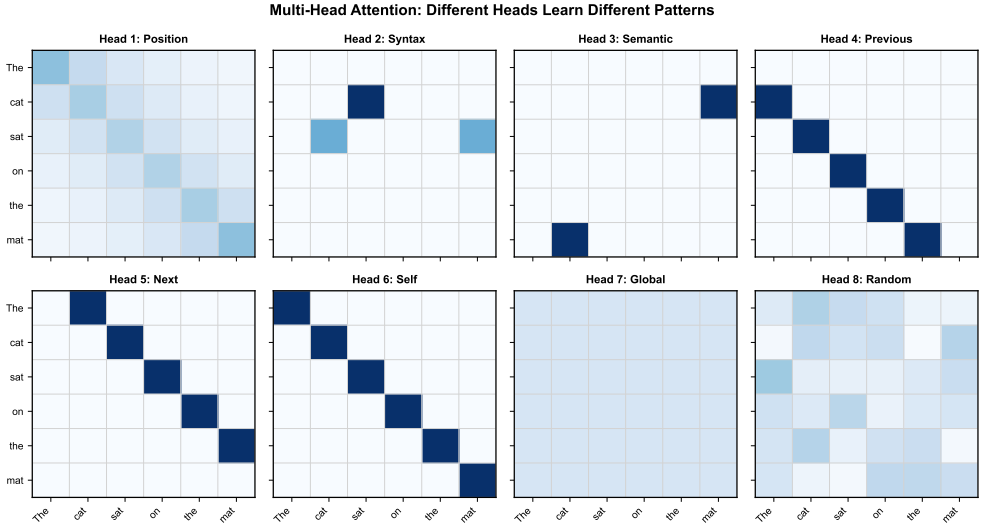
Layer Normalization:

- Stabilizes training
- Faster convergence
- Prevents saturation

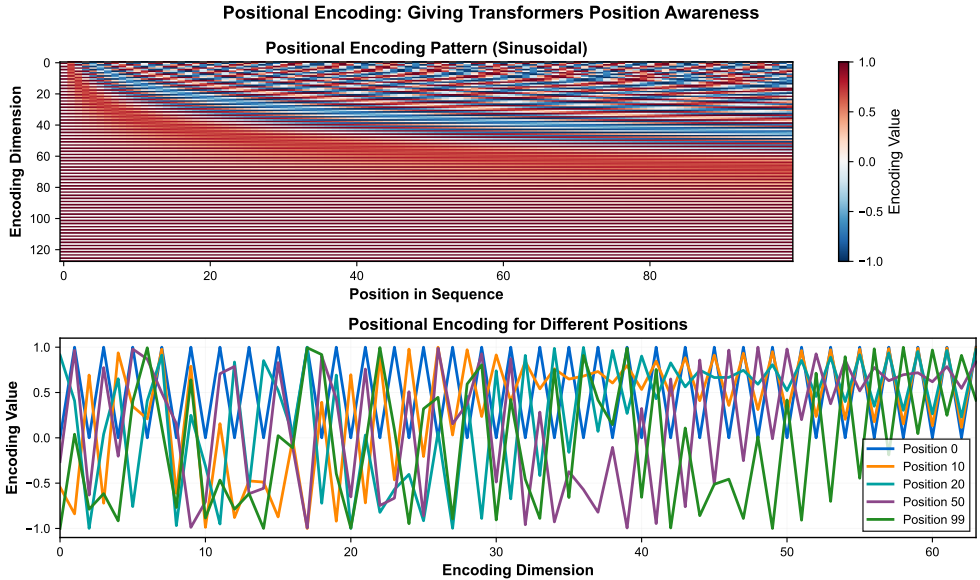
Attention Mechanism Visualization



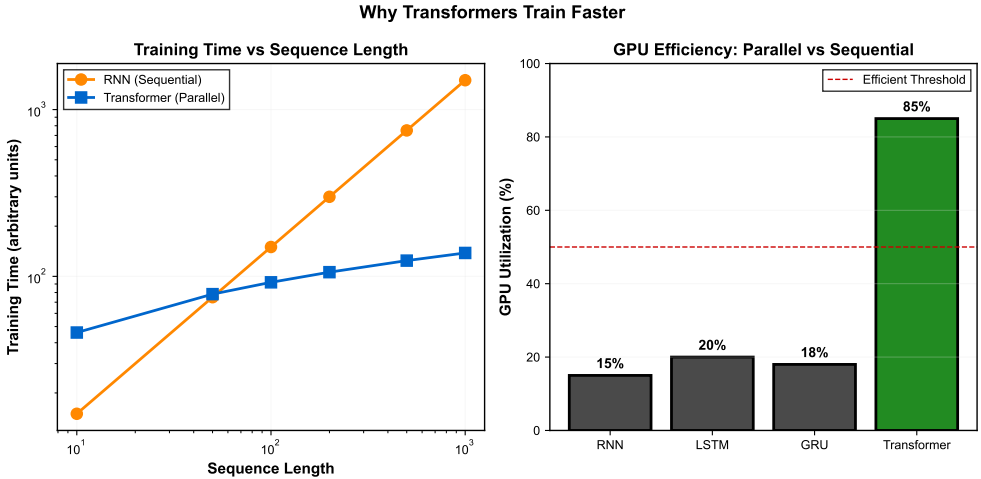
Multi-Head Attention Patterns



Positional Encoding Patterns



Transformer vs RNN Speed Comparison



The Transformer Family Tree (2024)

Encoder-Only (BERT)

- BERT
- RoBERTa
- DeBERTa
- ELECTRA

Used for:

- Classification
- NER
- Question Answering

Decoder-Only (GPT)

- GPT-4
- Claude
- LLaMA
- Mistral

Used for:

- Generation
- Chat
- Code completion

Encoder-Decoder

- T5
- BART
- mT5
- mBART

Used for:

- Translation
- Summarization
- Text rewriting

Transformer Gotchas and Solutions

1. Attention is Quadratic

- Problem: $O(n^2)$ memory
- Solution: Sparse attention
- Example: GPT-3 sparse patterns

2. Position Extrapolation

- Problem: Fails on longer sequences
- Solution: ALiBi, RoPE
- Example: LLaMA 100k+ context

3. Training Instability

- Problem: Large models diverge
- Solution: LR warmup, careful init
- Example: GPT-3 months of tuning

4. Efficiency at Scale

- Problem: Billions of parameters
- Solution: FlashAttention, quantization
- Example: 2-3x speedup

Modern transformers use many optimizations for production

Week 5 Exercise: Build Your Own Mini-GPT

Your Mission: Create a character-level GPT for text generation

Implementation Steps:

1. Implement multi-head attention
2. Add positional encodings
3. Stack 6 transformer blocks
4. Train on Shakespeare
5. Generate new text

Expected Results:

- 10x faster than RNN
- Better long-range coherence
- GPU usage: 15% $\rightarrow$ 90%
- Meaningful attention patterns

Key Experiments:

- Compare 1 vs 8 vs 16 heads
- Try without position encoding
- Measure GPU utilization
- Visualize attention patterns

Bonus Challenges:

- Implement sparse attention
- Add beam search
- Different position schemes
- Build chatbot interface

You'll discover why transformers took over the world!

Week 5 Summary: The Attention Revolution

- Sequential processing was the bottleneck
- Self-attention enables full parallelization
- Every word attends to **every other word**
- Position encodings restore order information
- Transformers scale to trillions of parameters

The Evolution:

Sequential (RNN) $\rightarrow$ Parallel (Transformer) $\rightarrow$ Scale (GPT/BERT)

Next Week: Pre-trained Language Models

How do we use transformers to learn from all human knowledge?

Key Takeaway