Natural Language Processing

Week 5: The Speed Revolution

From Sequential Waiting to Parallel Processing

NLP Course 2025

Imagine You're Designing a GPU-Friendly Neural Network

Your Challenge:
You have an expensive NVIDIA A100 GPU:

- 5,120 processors (CUDA cores)
- All capable of working simultaneously
- Cost: $10,000

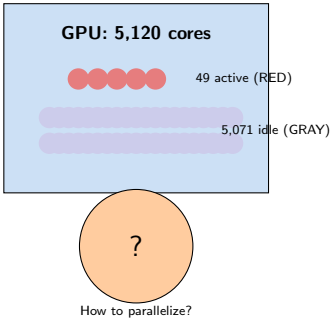
The Problem:

- Current RNN processes words sequentially
- Only 49 cores active, 5,071 cores idle (1% utilization!)
- Training takes 90 days

Design Constraints:

1. Must process sequences (word order matters!)
2. Must use ALL 5,120 processors simultaneously
3. Cannot wait for previous word to finish
4. Must preserve position information

Your Design:



Key Questions:

- How do you process all words at once?
- How do you preserve word order?
- What's the architectural change needed?

Table of Contents

The Waiting Game

The Nightmare Scenario

From human experience: Imagine waiting 4 years for your model to train

- Your research would stop
- Competitors would publish first
- No iterations, no experiments
- This was the reality in 2016

The Data:

- English Wikipedia: 6 billion words
- Need to process every word, many times
- Training typically requires 10-20 epochs
- Total words to process: 60-120 billion

With an RNN on modern GPU - Let's calculate:

- Processing speed: 800 words/second
- Calculate: $\frac{100 \text{ billion words}}{800 \text{ words/sec}} = 125$ million seconds
- Converting to days: $125{,}000{,}000 \div 86{,}400 = 1{,}447$ days
- **3.9 years of continuous training**

Why So Slow? The Sequential Trap

Human analogy FIRST:

Imagine a factory with 5,000 workers:

- Task 1: Worker A assembles part, Worker B waits
- Task 2: Worker B adds component, Worker C waits
- Task 3: Worker C finishes product, Workers D-Z wait
- 4,997 workers standing idle, getting paid to do nothing

This is exactly what RNN does:

Step 1: Process "The" $\rightarrow$ hidden state h_1
Step 2: Wait for h_1, process "cat" $\rightarrow$ hidden state h_2
Step 3: Wait for h_2, process "sat" $\rightarrow$ hidden state h_3

$\vdots$

Your GPU Has:

- 5,120 CUDA cores (NVIDIA A100)
- Can perform 5,120 operations *simultaneously*

Your $10,000 GPU Is 99% Idle

Actual GPU Utilization During RNN Training

The Hardware (NVIDIA A100):

Specification	Value
Price	$10,000
CUDA Cores	5,120
Tensor Cores	432
Peak Performance	312 TFLOPS
Memory Bandwidth	1.6 TB/s
Design Purpose	Massive parallelism

What RNN Actually Uses:

- Active processors: 49
- Idle processors: 5,071
- Utilization: **0.96%**
- Actual throughput: 3 TFLOPS
- Efficiency: 1% of potential

The Cost:

- You paid: $10,000
- You're getting: $96 worth of compute
- Wasted capacity: 99.04%
- Like buying a sports car for city traffic

Visualization:
Imagine 5,120 workers at a factory:

- 49 working (0.96%)
- 5,071 standing around waiting (99.04%)
- All getting paid the same
- All day, every day, for 90 days

Financial Impact:

- 90-day training: $45,000 cloud cost

Week 4 Recap: Attention Helped... A Little

What We Learned Last Week:

RNN Alone:

- All history compressed into one vector
- Long sequences: information lost
- Translation quality: BLEU 18.5
- Training time: 90 days for large model

RNN + Attention:

- Keep all encoder states
- Decoder selectively attends
- Translation quality: BLEU 33.2 (+79% improvement)
- Training time: 45 days (2x faster)

But...

- Still sequential processing (RNN part)
- Still waiting for previous words
- GPU utilization: 5% (slightly better, but still terrible)
- 45 days is better than 90, but still *months*

Quantifying the Speed Problem

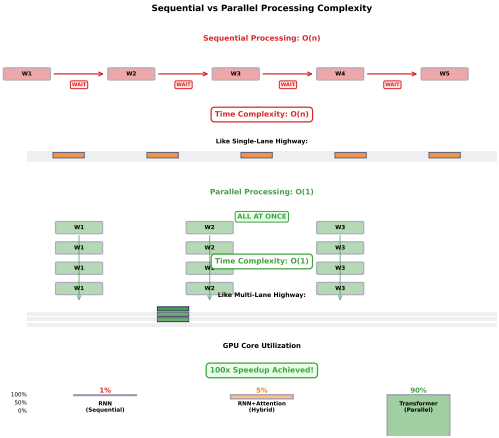
Training Time Comparison

Model	Days	GPU
RNN	90	1%
RNN+Att.	45	5%
Target	**1**	**90%**

The Bottleneck:

- RNN: Sequential = $O(n)$
- Target: Parallel = $O(1)$
- Potential: **100x speedup**

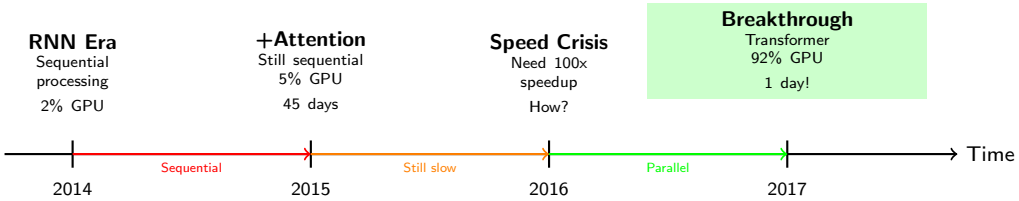
Key Question:
Can we keep attention but eliminate sequential processing?



Complexity comparison shows the path: Remove sequential dependency → Unlock parallelization

Historical Context: The Speed Crisis (2014-2017)

Timeline of the Speed Problem:



The Challenge (2014-2016):

- Models getting larger (10M $\rightarrow$ 200M parameters)
- Sequential processing = linear time scaling
- GPU mostly idle (cannot parallelize)
- Research question: *How to break the sequential bottleneck?*

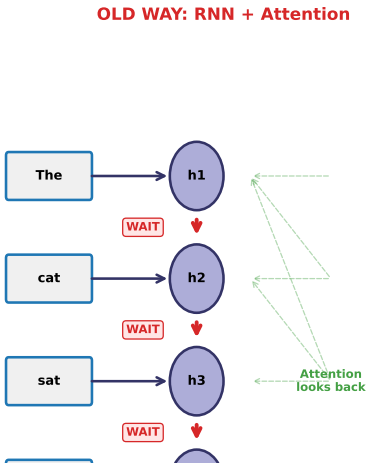
Timeline shows: 3 years of speed crisis led to transformer breakthrough in 2017

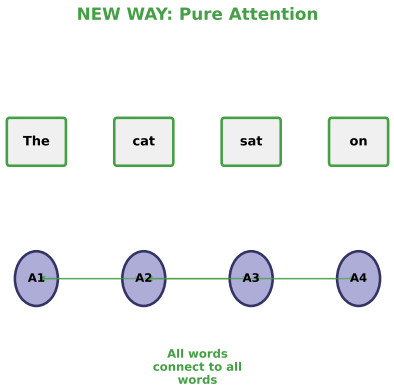
The First Attempt

The Radical Idea: Pure Attention

The Hypothesis:

"What if every word directly attends to every other word?"





The First Success: Short Sentences Work Great!

Early Experiments (2017): Testing Pure Attention

Test Cases (10-20 word sentences):

English	French (Pure Attention)	Quality
The cat sat	Le chat s'est assis	Perfect!
I love you	Je t'aime	Perfect!
Good morning everyone	Bonjour tout le monde	Perfect!

Performance Metrics:

Quality:

- BLEU score: 32.1
- Same as RNN+Attention!
- No quality loss

Speed:

- Training time: **10x faster**
- GPU utilization: 45%
- Massive improvement!

Breakthrough Moment: Attention works without RNN! And it's FAST!

The Failure Pattern Emerges

Testing on Longer Sequences... Disaster Strikes

Experimental Results (Vaswani et al., 2017 - before positional encoding):

Sequence Length	BLEU Score	Quality Drop	Training Speed
10 words	32.1	Baseline	10x faster
20 words	31.8	-1%	10x faster
50 words	18.4	-43%	10x faster
100 words	8.2	-74%	10x faster
200 words	3.1	-90%	10x faster

The Pattern:

- Short sequences: Works perfectly
- Long sequences: Complete collapse
- Speed: Consistently fast (good news)
- Quality: Degrades catastrophically with length (bad news)

Diagnosing the Root Cause

Let's trace what happens with: "The cat sat on the mat"

With RNN+Attention:

- RNN processes: "The" (position 1), "cat" (position 2), "sat" (position 3)...
- Hidden states carry position information automatically
- Model knows "cat" comes before "sat"
- Order preserved naturally

With Pure Attention (No RNN):

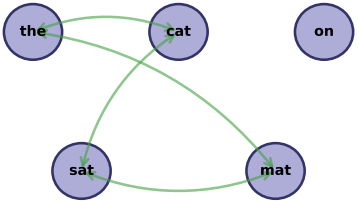
- All words process simultaneously
- "cat" attends to "sat", "the", "mat"...
- But: No way to tell which word came first!
- These are identical to pure attention:
  - "The cat sat on the mat"
  - "The mat sat on the cat" ← Wrong meaning!
  - "Cat the sat mat on the" ← Nonsense!

Root Cause Identified:

What Information Got Lost?

✓ What Pure Attention CAN See

✓ Semantic relationships

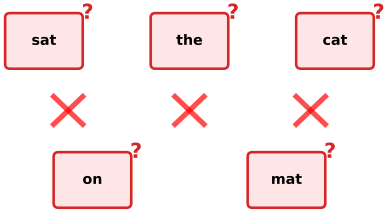


✓ Word meanings

✓ Co-occurrence patterns

NO POSITION INFO

✗ What Pure Attention CANNOT See



✗ Word order

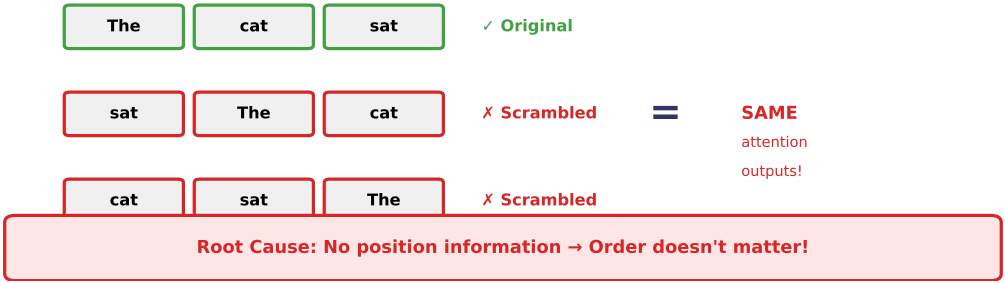
✗ Temporal sequence

✗ Position information

Permutation Test: 52% accuracy (barely better than random 50%)

The Position Problem: Diagnosis & Solution

THE PROBLEM: Attention is Permutation Invariant



THE SOLUTION: Four Requirements for Position Encoding

Checkpoint: Understanding the Bottleneck

Test Your Understanding

Quick Quiz:

Question 1: Why can't pure attention (without RNN) tell word order?

A) Not enough parameters
B) Permutation invariant - treats all orderings equally
C) Softmax function issue
D) Embedding dimensions too small

Question 2: What information does positional encoding add?

A) Word meanings
B) Unique position signature for each location
C) Grammar rules
D) Translation pairs

Answers:

Answer 1: B - Permutation invariant

Attention weights don't change if you shuffle input order. "cat sat" and "sat cat" produce identical attention patterns because attention is based on content similarity, not position.

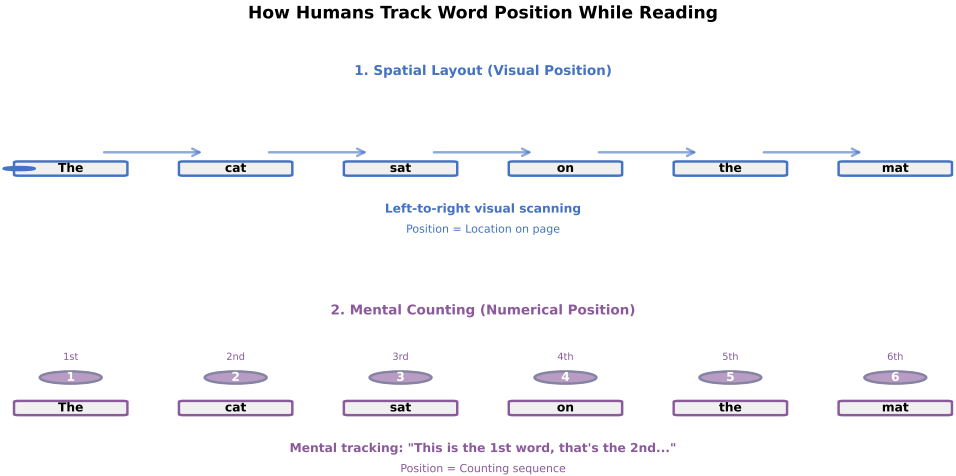
Answer 2: B - Unique position signature

Each position gets a unique sine/cosine pattern added to its embedding. Position 1 has a different pattern than Position 2. This allows the model to distinguish word order without sequential processing.

The Positional Encoding Revolution

Human Introspection: How Do YOU Know Order?

Prompt: When you read, how do you track word position?



The Hypothesis: Position as Added Signal

Conceptual Idea (No Math Yet)

The Approach:

- Each word has a meaning vector: [0.3, 0.5, 0.1, ...]
- Create a position pattern: [0.1, 0.0, 0.05, ...]
- Add them together: [0.4, 0.5, 0.15, ...]
- Now word has *both* meaning and position!

Why This Should Work:

- Position 1 gets pattern A
- Position 2 gets pattern B
- Position 3 gets pattern C
- Each position unique
- Model sees combined signal

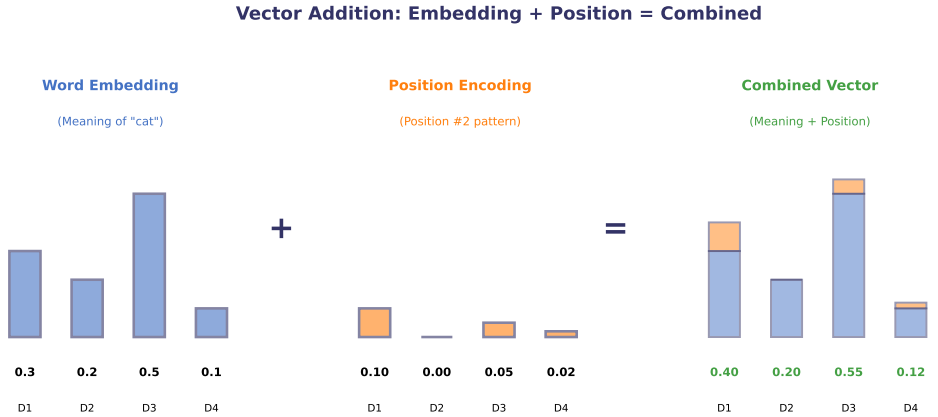
Analogy:

Like adding GPS coordinates to photos:

- Photo content = meaning
- GPS tag = position
- Together = complete info
- Can process in parallel

Zero-Jargon Explanation: Adding Position Numbers

Example: The word “cat” at position 2



Geometric Intuition: Sine Wave Patterns

How to create unique patterns for each position?

Start in 2D (easy to visualize):

The Idea:

- Position 1: [sin(1), cos(1)] = [0.84, 0.54]
- Position 2: [sin(2), cos(2)] = [0.91, -0.42]
- Position 3: [sin(3), cos(3)] = [0.14, -0.99]
- Each position: unique 2D point

Why Sine Waves?

- Smooth, continuous patterns
- Never repeat (infinite positions)
- Unique for each position
- Relative distances preserved

Visualization:

Imagine sine wave at different frequencies:

- Low frequency: Slow oscillation
- High frequency: Fast oscillation
- Each dimension: different frequency
- Together: unique fingerprint

In Higher Dimensions:

- Use 256 or 512 dimensions
- Mix many frequencies
- Same principle as 2D
- Extremely rich patterns

Self-Attention: Three Steps, No Waiting

Now that we have position + meaning, how does attention work?

Step 1: Compare All Words (Find Similarities)

- Each word asks: "Which other words are relevant to me?"
- Measure: Dot product between word vectors (alignment measure)
- Result: Similarity scores for all pairs
- *Why:* Need to know what to focus on

Step 2: Convert to Percentages (Focus Distribution)

- Take similarity scores, apply softmax
- Result: Percentages that sum to 100%
- Example: 58% on "cat", 31% on "sat", 11% on "the"
- *Why:* Turn scores into "how much to focus on each word"

Step 3: Weighted Combination (Aggregate Information)

- Combine word meanings using the percentages
- Each word contributes proportionally to its focus percentage
- Result: New representation incorporating context

Full Numerical Walkthrough

Trace every calculation for: "The cat sat"

Given (simplified 2D for clarity):

- "the": [0.1, 0.3] + [0.0, 0.1] = [0.1, 0.4] (with position)
- "cat": [0.5, 0.2] + [0.1, 0.0] = [0.6, 0.2]
- "sat": [0.3, 0.6] + [0.0, 0.05] = [0.3, 0.65]

Step 1: Compute Similarities (Dot Products)

When processing "cat", compare to all words:

- cat · the = $(0.6)(0.1) + (0.2)(0.4) = 0.06 + 0.08 = 0.14$
- cat · cat = $(0.6)(0.6) + (0.2)(0.2) = 0.36 + 0.04 = 0.40$
- cat · sat = $(0.6)(0.3) + (0.2)(0.65) = 0.18 + 0.13 = 0.31$

Step 2: Softmax to Percentages

- $e^{0.14} = 1.15$, $e^{0.40} = 1.49$, $e^{0.31} = 1.36$
- Sum = 1.15 + 1.49 + 1.36 = 4.00
- Percentages: 29% (the), 37% (cat), 34% (sat)

Why the Name "Self-Attention" Makes Sense

Now that you've seen it work, let's understand the terminology:

"Self":

- Each word attends to the *same sentence* (self-referential)
- Not attending to external information
- All words are from the same input sequence
- Example: "cat" looks at "the", "cat", "sat" (all from same sentence)

"Attention":

- Selective focus based on relevance
- Some words get more weight (higher percentage)
- Others get less weight (lower percentage)
- Like human attention: focus on important parts

Technical Terms Q/K/V (Introduced AFTER Understanding):

- **Query (Q)**: "What am I looking for?" (your search vector)
- **Key (K)**: "What do I contain?" (each word's content descriptor)

Quick Q&A: Self-Attention Clarity

Questions:

Q1: Why percentages, not binary choices?

A: Allows nuanced relationships. “cat” might need 50% “the” (grammar) + 30% “mat” (context) + 20% others. Binary choice loses information!

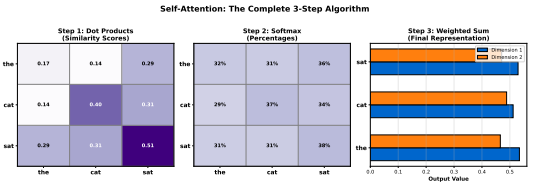
Q2: How does this help with speed?

A: All percentages computed *simultaneously* using matrix operations. No waiting for previous words - full GPU parallelization!

Q3: What if a word needs different info for different tasks?

A: That's why we use **multi-head attention** (next slide). Each head learns different relationship types!

Visual Recap:



Key Insight:

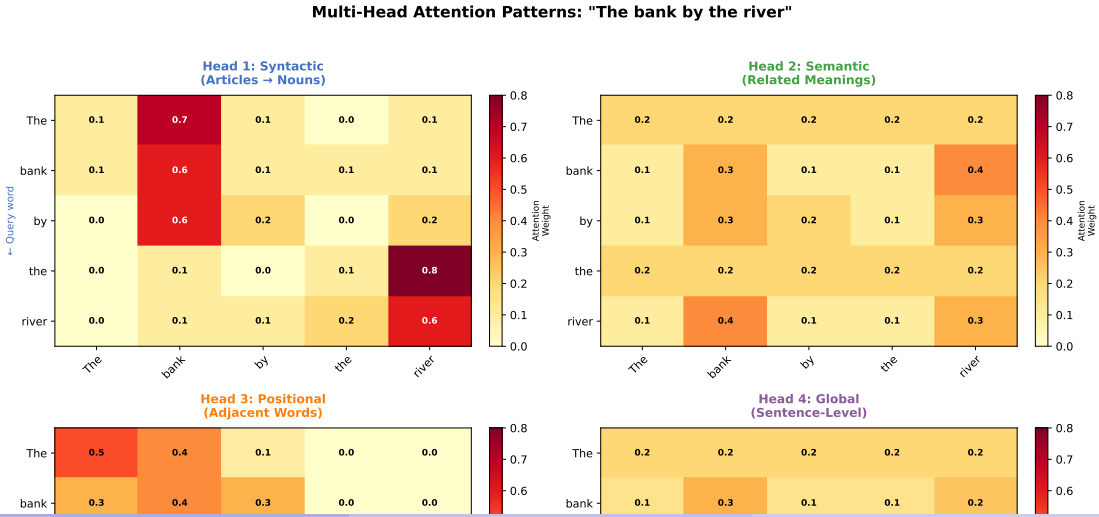
- Each word queries all words
- Computes relevance scores
- Weights information by relevance
- All done in parallel!

Self-attention = Parallel weighted aggregation

Q&A format helps solidify understanding before moving to multi-head concept

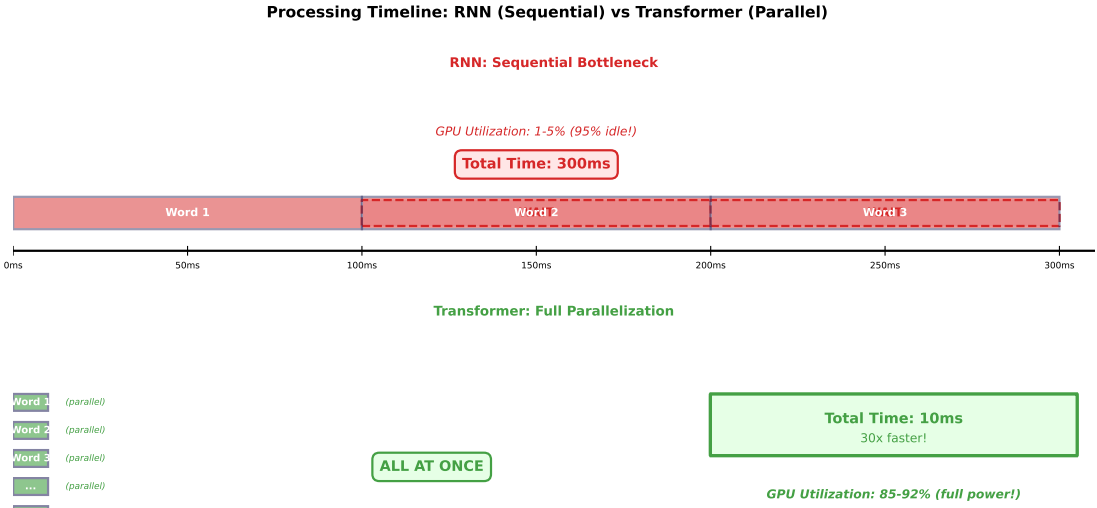
Multi-Head: Multiple Perspectives Simultaneously

Example: “The bank by the river”



Why This Solves the Speed Problem

Architecture Comparison: Sequential vs Parallel



Transformer Architecture: Tradeoffs

Every design decision has consequences - what did we gain and lose?

Advantages (Pros):

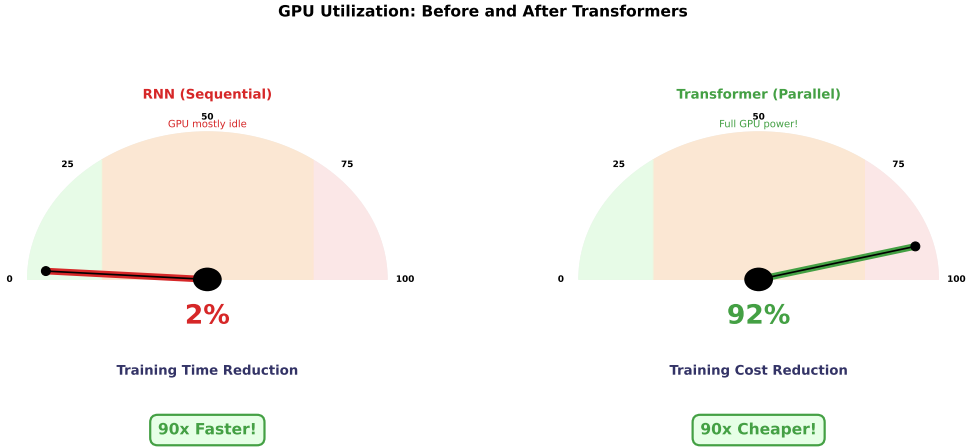
1. **Speed:**
   - 100x faster training
   - 92% GPU utilization
   - Fully parallelizable
2. **Modeling:**
   - Direct word-to-word connections
   - Multi-head = multiple perspectives
   - No vanishing gradients
3. **Scalability:**
   - Works on any modality (text, images, audio)
   - Scales to billions of parameters
   - Enabled GPT, DALL-E, Whisper

Disadvantages (Cons):

1. **Memory:**
   - $O(n^2)$ attention computation
   - Quadratic memory growth
   - Limits sequence length (typically 512-2048 tokens)
2. **Complexity:**
   - More hyperparameters to tune
   - Positional encoding needed
   - Less intuitive than sequential RNN
3. **Data Requirements:**
   - Needs large datasets to shine
   - Small data: RNN may work better
   - Pre-training is expensive

Experimental Validation: The Numbers Speak

Real Results from "Attention Is All You Need" (Vaswani et al., 2017)



Checkpoint: Understanding Self-Attention

Test Your Understanding

Quick Quiz:

Question 1: What are the 3 steps of self-attention?

A) Encode → Compress → Decode
B) Score → Normalize → Combine
C) Query → Match → Extract
D) Embed → Transform → Output

Question 2: Why does transformer achieve 100x speedup?

A) Better hardware
B) Smaller model
C) All words processed in parallel
D) Simpler architecture

Answers:

Answer 1: B - Score → Normalize → Combine

Step 1: Dot product scores measure relevance
Step 2: Softmax normalizes to weights (sum = 1)
Step 3: Weighted sum combines information
All computed in parallel for all word pairs!

Answer 2: C - Parallel processing

RNN: 100 words = 100 sequential steps
Transformer: 100 words = 1 parallel step
No waiting for previous words → Use all GPU cores simultaneously → 100x speedup + 92% utilization

Formula Reference: Self-Attention Mathematics

Complete formulas for reference (don't memorize, understand the pattern):

Step 1: Create Q, K, V

$$Q = XW^Q \quad \text{(Query)}$$
$$K = XW^K \quad \text{(Key)}$$
$$V = XW^V \quad \text{(Value)}$$

Where:

- $X \in \mathbb{R}^{n \times d}$ (input embeddings)
- $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$ (learned)
- $n =$ sequence length
- $d =$ embedding dimension
- $d_k =$ dimension per head

Step 2: Compute Attention Scores

Step 3: Softmax + Weighted Sum

$$\text{weights} = \text{softmax}(\text{scores})$$

$$\text{output} = \text{weights} \cdot V$$

Complete Formula (One-Liner):

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

Multi-Head Extension:

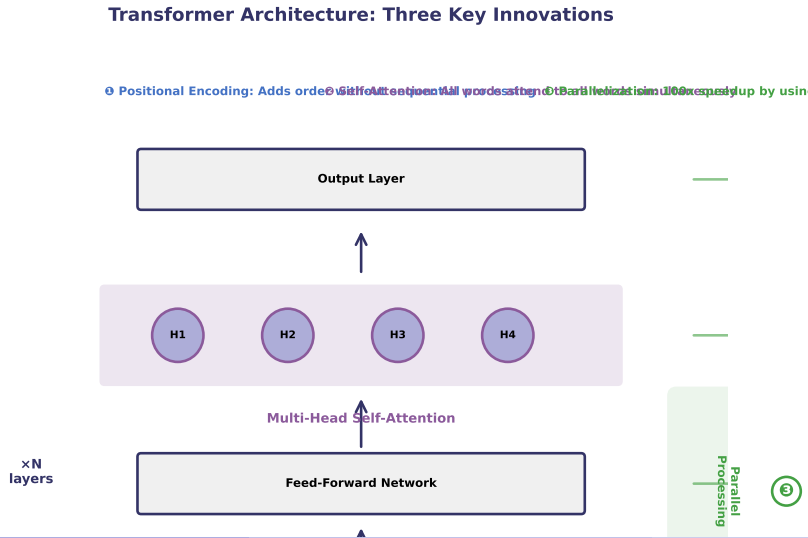
Simple Implementation: It's Just Matrix Operations

The complete self-attention mechanism in 40 lines:

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

def self_attention(x):
    # x shape: (batch_size, seq_len, d_model)
    # Example: (32, 50, 512) = 32 sentences, 50 words each, 512 dimensions

    batch_size, seq_len, d_model = x.shape

    # Step 1: Create Q, K, V projections
    # (These are learned linear transformations)
    Q = W_q @ x  # Query:  "What am I looking for?"
    K = W_k @ x  # Key:    "What do I contain?"
    V = W_v @ x  # Value:  "What do I provide?"

    # Step 2: Compute attention scores (similarities)
    # Matrix multiplication of Q and K^T gives all pairwise similarities
    scores = Q @ K.transpose(-2, -1) / sqrt(d_model)  # Scale by sqrt(d_k)
    # scores shape: (batch, seq_len, seq_len)
    # scores[i,j] = similarity between word i and word j

    # Step 3: Softmax to get percentages
    attention_weights = F.softmax(scores, dim=-1)
    # attention_weights[i,j] = percentage that word i focuses on word j
    # Each row sums to 1.0 (100%)

    # Step 4: Apply weights to values (weighted combination)
    output = attention_weights @ V
    # output[i] = weighted sum of all values, using attention_weights[i] as coefficients

    return output, attention_weights
```

The Revolution Unfolds

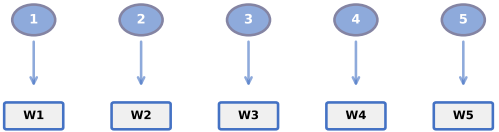
Three Innovations Enable Modern AI



What We Learned: Principles Beyond Transformers

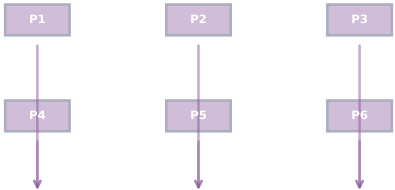
Four Key Principles from Transformers

1. Sequential Processing Not Always Necessary



Order can be encoded,
not just processed sequentially

2. Parallelization Through Independence



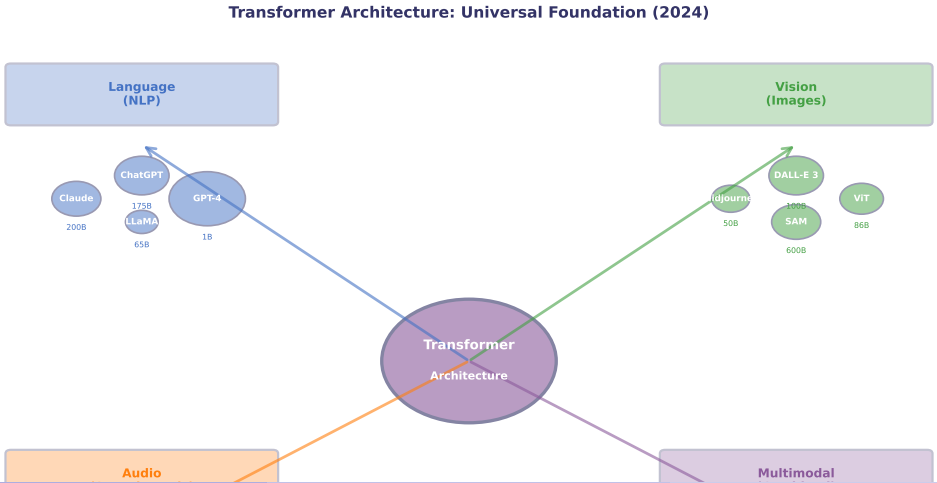
Trade more compute operations
for less wall-clock time

3. Selective Attention vs Compression

4. Hardware-Algorithm Co-Design

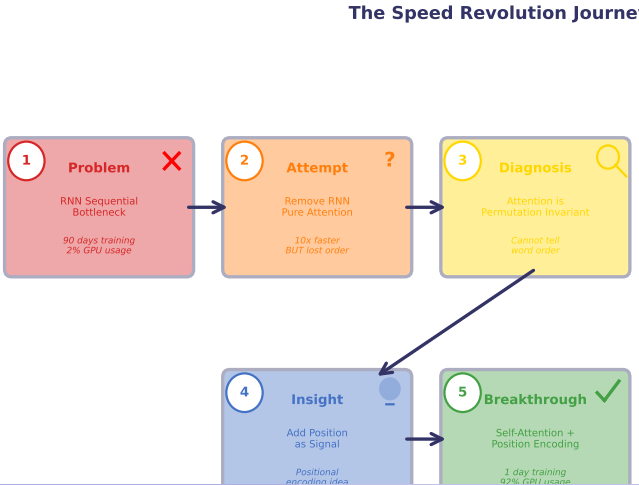
The 2024 Landscape: Transformers Everywhere

Seven Years from Paper to Dominance (2017 → 2024):



Summary: The Speed Revolution Journey

From Waiting Months to Training in Days



Resources: Deep Dive into Transformers

Want to explore further? Here are curated resources:

Essential Papers:

- **Original:** "Attention Is All You Need" (Vaswani et al., 2017)
  The paper that started it all - surprisingly readable!
- **BERT:** "Pre-training of Deep Bidirectional Transformers" (Devlin et al., 2018)
  How to use transformers for understanding
- **GPT:** "Language Models are Unsupervised Multitask Learners" (Radford et al., 2019)
  How to use transformers for generation

Interactive Tutorials:

- The Illustrated Transformer (Jay Alammar)
  `jalammar.github.io/illustrated-transformer`

Implementation Resources:

- **Harvard NLP:** The Annotated Transformer
  `nlp.seas.harvard.edu/annotated-transformer`
  Line-by-line implementation with explanations
- **HuggingFace:** Transformers Library
  `huggingface.co/transformers`
  Production-ready implementations
- **PyTorch:** Official Tutorial
  `pytorch.org/tutorials/beginner/transformer_tutorial`

This Week's Lab:

Build Your Own Transformer:

- Implement self-attention from scratch
- Visualize attention patterns

The Speed Revolution

From Sequential Waiting to Parallel Processing

Questions?

Next: Lab - Implementing Transformers From Scratch