Natural Language Processing

Week 5: The Speed Revolution

From Sequential Waiting to Parallel Processing

NLP Course 2025

The Nightmare Scenario

You want to train a language model on Wikipedia

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:

- Processing speed: 800 words/second
- Calculate: $\frac{\text{100 billion words}}{\text{800 words/sec}} = 125$ million seconds
- Converting: **3.9 years of continuous training**

The Waiting Game: Press “train”, come back in 2027

Even with parallelization tricks, you’re looking at months of waiting

Why So Slow? The Sequential Trap

RNN must process one word at a time:

Step 1: Process “The” → hidden state h_1
Step 2: Wait for h_1, process “cat” → hidden state h_2
Step 3: Wait for h_2, process “sat” → hidden state h_3
$\vdots$

Human Analogy: Assembly line where each worker waits for previous worker to finish

Your GPU Has:

- 5,120 CUDA cores (NVIDIA A100)
- Can perform 5,120 operations *simultaneously*
- But RNN uses only ONE core at a time
- The other 5,119 sit idle, waiting

Sequential processing = massive underutilization

Your $10,000 GPU Is Bored to Death

Actual GPU Utilization During RNN Training

The Hardware:

- NVIDIA A100: $10,000
- 5,120 parallel processors
- Designed for massive parallelism
- Peak: 312 TFLOPS

What RNN Uses:

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

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
Only 49 working
5,071 standing around waiting

The Waste: This is like hiring 100 people but only giving work to 1

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 (Wikipedia-scale model)

Model	**Days**	**GPU Util**	**BLEU**	**Cost ($)**
RNN	90	1%	28.5	$45,000
RNN+Attention	45	5%	33.2	$22,500
Target?	**1**	**90%**	**34+**	**$500**

Information Theory Perspective:

- Sequential processing: Compute operations $= O(n)$ where $n =$ sequence length
- Parallel potential: Could do all operations simultaneously $= O(1)$
- Theoretical speedup: 100x (if we remove sequential dependency)

The Key Observation:

- Attention was helpful (quality improved)
- RNN was the bottleneck (sequential processing)
- Radical question: **What if we removed the RNN entirely?**

The Radical Idea: Pure Attention

The Observation:

Breaking down RNN+Attention:

- Attention part: Helped quality (selectively focus on relevant words)
- RNN part: Created bottleneck (sequential processing)

The Hypothesis:

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

Old Way (RNN+Attention):

- RNN: Process sequentially
- Build hidden states one-by-one
- Attention: Look back at states
- Sequential bottleneck

New Idea (Pure Attention):

- No RNN at all
- All words connect to all words
- Attention happens simultaneously
- No sequential dependency

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:

Attention is permutation invariant – order doesn't matter!

What Information Got Lost?

Two-Column Analysis: What Survived vs What Died

What Pure Attention CAN See:

- Which words are present
- Semantic relationships
- Word meanings
- Attention weights
- Co-occurrence patterns

Example:

- Knows “cat” and “sat” are related
- Knows “mat” is object
- Understands semantic fields

What Pure Attention CANNOT See:

- Which word came first
- Temporal ordering
- Sequence position
- Left-to-right flow
- Syntactic structure

Example:

- Can’t distinguish subject vs object
- “cat sat” = “sat cat” (same!)
- Word order scrambled

Quantifying the Mismatch:

- Test: Randomly permute word order
- Pure attention performance: 50% (barely better than random: 50%)

The Key Question

How Do We Add Position
Without Going Back to Sequential Processing?

The Dilemma:

- RNN gave us position *automatically* (by processing sequentially)
- But sequential processing is exactly what we want to eliminate
- We need position information *without* sequential dependency

Requirements for a Solution:

1. Inject position information somehow
2. Must be computable in parallel (no sequential dependency)
3. Must work for any sequence length
4. Should preserve relative positions

The Insight Needed:

Human Introspection: How Do YOU Know Order?

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

Honest Self-Observation:

1. You see *spatial layout*: Words from left to right on page
2. You track mentally: “This is the first word, that’s the second...”
3. You use *both* meaning AND position together
4. Position isn’t separate - it’s part of how you understand each word

Key Realizations:

- Position information can be *visual/spatial* (location on page)
- Or it can be *numerical* (counting: 1st, 2nd, 3rd)
- It’s added to meaning, not processed separately
- You process meaning + position *simultaneously*

The Aha Moment:

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

Let's see this with actual numbers:

Example: The word "cat"

- Word embedding (meaning of "cat"): [0.3, 0.2, 0.5, 0.1]

When "cat" is at position 1:

- Position pattern for 1: [0.1, 0.0, 0.0, 0.05]
- Combined: [0.3, 0.2, 0.5, 0.1] + [0.1, 0.0, 0.0, 0.05]
- Result: [0.4, 0.2, 0.5, 0.15] ← This represents "cat at position 1"

When "cat" is at position 2:

- Position pattern for 2: [0.0, 0.1, 0.05, 0.0]
- Combined: [0.3, 0.2, 0.5, 0.1] + [0.0, 0.1, 0.05, 0.0]
- Result: [0.3, 0.3, 0.55, 0.1] ← This represents "cat at position 2"

The Magic:

- Same word, different positions = different number patterns

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: The Complete 3-Step Algorithm

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
- *Why:* Builds meaning from relevant surrounding words

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:

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

Step 2: Softmax to Percentages

- $e^{0.14} = 1.15$, $e^{0.40} = 1.49$, $e^{0.31} = 1.36$
- $\text{Sum} = 1.15 + 1.49 + 1.36 = 4.00$
- 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)

Multi-Head: Multiple Perspectives Simultaneously

One attention mechanism finds one type of relationship

But different relationships matter:

- Head 1: Syntactic dependencies (subject-verb agreement)
- Head 2: Semantic similarity (related meanings)
- Head 3: Positional patterns (nearby words)
- Head 4: Co-reference (pronouns to nouns)
- ... (typically 8-16 heads)

Example: "The bank by the river"

Head 1

Syntax

- bank → the
- river → the
- by → bank

Head 2

Semantics

- bank → river
- Strong connection
- Related concepts

Head 3

Position

- Adjacent words
- Local context
- Sequential flow

Head 4

Global

- Sentence-level
- Broad attention
- Context gathering

Why This Solves the Speed Problem

Architecture Comparison: Sequential vs Parallel

RNN (Sequential):

- Process word 1 → state 1
- Wait... Process word 2 → state 2
- Wait... Process word 3 → state 3
- Time complexity: $O(n)$ steps
- GPU utilization: 1-5%
- Bottleneck: Sequential dependency

Timeline:

Word 1: [—-] (100ms)
Word 2: [—-] (100ms)
Word 3: [—-] (100ms)
Total: 300ms

Transformer (Parallel):

- All words processed simultaneously
- Self-attention: All pairs at once
- Positional encoding: Pre-computed
- Time complexity: $O(1)$ steps
- GPU utilization: 85-92%
- No sequential dependency!

Timeline:

Word 1: [-] (10ms)
Word 2: [-] (10ms) *(parallel)*
Word 3: [-] (10ms) *(parallel)*
Total: 10ms

Experimental Validation: The Numbers Speak

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

Translation Quality (WMT English-German):

Model	Training Time	BLEU	GPU Usage	Parameters
RNN	90 days	24.5	2%	200M
RNN+Attention	45 days	28.4	5%	210M
Transformer (base)	**1 day**	**27.3**	**90%**	65M
Transformer (big)	**3.5 days**	**28.4**	**92%**	213M

Key Observations:

- Transformer base: Same quality as RNN+Attention in 1 day vs 45 days (45x speedup)
- Transformer big: *Better* quality in 3.5 days vs 90 days (25x speedup + better BLEU)
- GPU utilization: 2% → 92% (46x improvement)
- Fewer parameters but better efficiency

Breakthrough Validated: Faster training, better quality, full parallelization achieved!

Simple Implementation: It’s Just Matrix Operations

The complete self-attention mechanism in 40 lines:

```python
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 Complete Transformer Architecture

All Components Working Together

Input Processing:

1. Word embeddings (meaning vectors)
2. + Positional encoding (position patterns)
3. = Complete representation

Core Mechanism:

1. Multi-head self-attention
2. Parallel processing of all words
3. Multiple relationship types
4. Attention weights show focus

Enhancement Layers:

1. Feed-forward networks

The Three Key Innovations:

1. Positional Encoding:
Solved order problem without sequential processing

2. Self-Attention:
All words attend to all words simultaneously

3. Full Parallelization:
100x speedup by using all GPU cores

Typical Configuration:

- 6-24 layers
- 8-16 attention heads
- 512-1024 model dimension
- 10M-1B+ parameters

What We Learned: Principles Beyond Transformers

Conceptual Insights That Transfer to Other Domains:

1. Sequential Processing Is Not Always Necessary

- Order can be encoded explicitly (positional encoding)
- Don't assume sequential processing is required for sequential data
- Remove unnecessary dependencies to enable parallelization

2. Parallelization Through Independence

- Identify what can be computed independently
- Matrix operations enable massive parallelism
- Trade more compute operations for less wall-clock time

3. Selective Attention vs Compression

- Week 4 lesson: Don't compress, selectively attend
- Week 5 extension: Do it all in parallel
- Keep information, let model decide what's relevant

The 2024 Landscape: Transformers Everywhere

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

Language:

- ChatGPT (175B)
- GPT-4 (1.7T)
- Claude (200B)
- Bard/Gemini
- LLaMA

Vision:

- ViT (images)
- DALL-E 3
- Midjourney
- Stable Diffusion
- SAM (segmentation)

Audio:

- Whisper (speech)
- MusicGen
- AudioLM
- Vall-E (voice)

Code & Science:

- Copilot
- AlphaFold
- ESMFold
- Galactica

Timeline of Impact:

- 2017: Paper published ("Attention Is All You Need")
- 2018: BERT revolutionizes NLP (Google Search)
- 2019: GPT-2 shows scale matters
- 2020: GPT-3 demonstrates emergent abilities (175B parameters)
- 2021: Vision Transformers beat CNNs
- 2022: ChatGPT launches (100M users in 2 months)
- 2023: GPT-4, multimodal transformers everywhere
- 2024: Transformers in every AI product

Summary: The Speed Revolution Journey

From Waiting Months to Training in Days

The Journey:

1. **The Problem:** RNNs sequentially process = 90 days training, 2% GPU usage
2. **First Attempt:** Remove RNN, use pure attention = 10x faster BUT lost word order
3. **The Diagnosis:** Attention is permutation invariant - can't tell word order
4. **The Insight:** Add position as explicit signal (positional encoding)
5. **The Breakthrough:** Self-attention + positional encoding = 100x speedup

Key Takeaways:

- Self-attention enables full parallelization (all words simultaneously)
- Positional encoding preserves order without sequential processing
- Result: 1 day training instead of 90 days, 90% GPU usage instead of 2%
- Enabled modern AI: ChatGPT, GPT-4, DALL-E only possible due to speed

The Speed Revolution

From Sequential Waiting to Parallel Processing

Questions?

Next: Lab - Implementing Transformers From Scratch