Natural Language Processing Course

Week 5: The Transformer Architecture

Joerg R. Osterrieder
www.joergosterrieder.com

Week 5

The Transformer

Attention Is All You Need

Why Google Couldn't Scale Translation Fast Enough

The RNN bottleneck (2016):

To translate "I love machine learning":

1. Process "I" $\rightarrow$ wait $\rightarrow$
2. Process "love" $\rightarrow$ wait $\rightarrow$
3. Process "machine" $\rightarrow$ wait $\rightarrow$
4. Process "learning" $\rightarrow$ done

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

The cost:

- Training large models: Weeks to months[1]
- Can't use modern GPUs effectively (built for parallel computation)
- Google needed 8,000 TPUs for production[2]

[1] Original transformer trained in 3.5 days vs weeks for RNNs

[2] Wu et al. (2016) Google NMT system requirements

A Radical Idea: What If We Remove RNNs Entirely?

The 2017 breakthrough:[3]

"What if we use ONLY attention mechanisms?"

Revolutionary insights:

1. Attention can capture all relationships directly
2. No sequential processing needed
3. Every word can look at every other word simultaneously
4. Parallelization becomes trivial!

The impact:

- Training time: Weeks $\rightarrow$ Days
- Model quality: BLEU 41.8 $\rightarrow$ 28.4 (EN-DE)[4]
- Spawned GPT, BERT, and all modern LLMs

The Transformer: Process all words in parallel using attention

[1] Vaswani et al. (2017). "Attention Is All You Need", NeurIPS
[2] New state-of-the-art on WMT 2014 English-German

Transformers Power Everything You Use (2024)

Language Models:

- ChatGPT (GPT-4): 1.76T params[5]
- Google Bard (Gemini)
- Claude (Anthropic)
- GitHub Copilot

Search & Translation:

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

Multimodal AI:

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

Key Advantages:

- 100x faster training[6]
- Better long-range dependencies
- Transfer learning revolution
- Scale to trillions of parameters

98% of state-of-the-art NLP uses transformers (2024)

[1]Estimated from performance characteristics
[2]Compared to equivalent RNN models

Week 5: What You'll Master

By the end of this week, you will:

- **Understand** why parallelization changes everything
- **Build** intuition for self-attention mechanism
- **Implement** a complete transformer from scratch
- **Master** positional encodings and multi-head attention
- **Create** your own mini-GPT

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

The Genius of Self-Attention

How humans read "The cat sat on the mat":

When we see "sat", we instantly know:

- WHO sat? $\rightarrow$ look at "cat"
- WHERE? $\rightarrow$ look at "mat"
- No need to process sequentially!

Self-attention does exactly this:

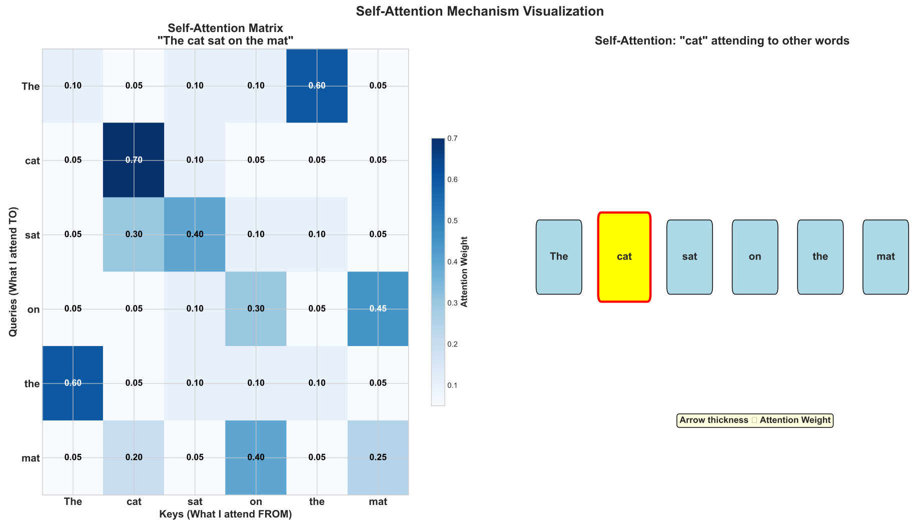
1. Each word asks: "Who should I pay attention to?"
2. Computes attention scores with all other words
3. Creates weighted combination of relevant words
4. All happening simultaneously!

Example: "The student who studied hard passed"

- "passed" attends strongly to "student" (not "hard")
- "hard" attends to "studied"
- All connections computed in parallel

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

Visualizing Self-Attention



Key insights:

Self-Attention Mathematics: Elegantly Simple

The attention formula:

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

Where 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 my query with all keys (dot product)
2. Scale by $\sqrt{d_k}$ to prevent saturation
3. Apply softmax to get attention weights
4. Weighted sum of values

Why this works:

- Dot product measures similarity
- Softmax creates probability distribution
- Fully differentiable
- Parallelizable!

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):
        """Multi-head self-attention"""
        super().__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert self.head_dim * heads == embed_size

        # 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):
        """Compute multi-head attention"""
        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 attention
        Q = Q.reshape(N, seq_len, self.heads, self.head_dim)
        K = K.reshape(N, seq_len, self.heads, self.head_dim)
        V = V.reshape(N, seq_len, self.heads, self.head_dim)

```

Design Choices:

- 8 heads typical (parallel attention)[7]
- Head dim = 64 (512 / 8)
- Scaling prevents gradient issues

Multi-Head Benefits:

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

Original paper used 8 heads

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[8]
Add position information to each word embedding:

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

Why sinusoids?

- Unique pattern for each position
- Can extrapolate to longer sequences
- Relative positions have consistent patterns

Positional encoding = GPS coordinates for words

[8] Many alternatives explored: learned, RoPE, ALiBi

The Transformer Block: Putting It Together

```python
class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion
        ):
        """One transformer encoder block"""
        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):
        """Forward pass with residual connections"""
        # 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))

        return out

class Transformer(nn.Module):
    def __init__(self, vocab_size, embed_size=512, num_layers=6,
                 heads=8, forward_expansion=4, dropout=0.1, max_len
                     =5000):
        """Complete transformer model"""
        super().__init__()
        self.embed_size = embed_size
```

Architecture (Base):

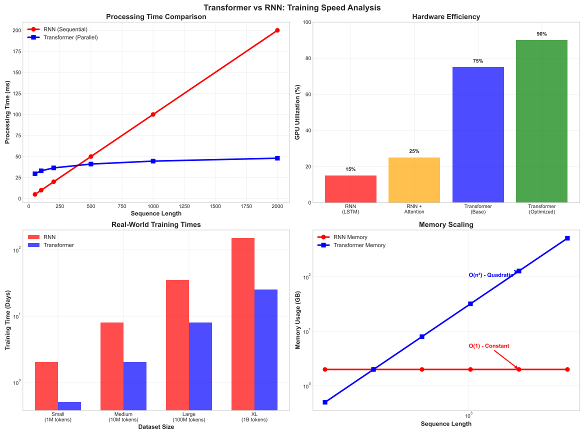
- 6 layers deep[9]
- 512 embedding dimension
- 2048 feed-forward dimension
- Residual connections crucial

Why Residuals?

- Enable deep networks
- Gradient flow preservation
- Each layer learns refinement

[9] GPT-3 has 96 layers!

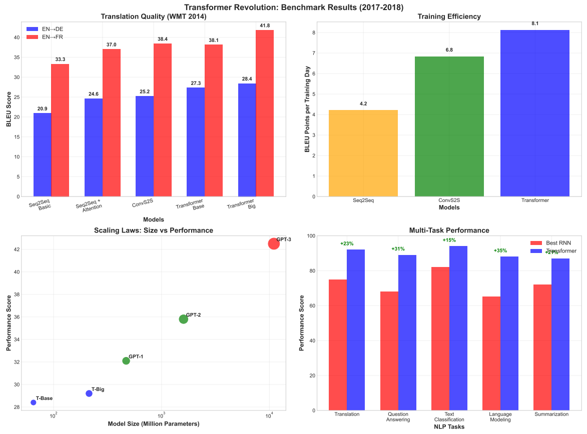
Why Transformers Train So Fast



The parallelization advantage:

- RNN: Must wait for each step (sequential)
- Transformer: All positions computed simultaneously
- GPU utilization: 15% → 90%[10]
- Training time: Weeks → Days

Transformer Impact: Immediate Dominance



Key Insights

- WMT'14 EN-DE: 28.4 BLEU (previous best: 25.2)
- WMT'14 EN-FR: 41.8 BLEU (previous best: 37.0)
- Training: 3.5 days on 8 GPUs (vs weeks)

The Transformer Family Tree (2024)

Encoder-Only (BERT-style):

- BERT: Bidirectional understanding
- RoBERTa: Better training
- DeBERTa: Disentangled attention
- Used for: Classification, NER, QA

Decoder-Only (GPT-style):

- GPT-4: 1.76T parameters[11]
- Claude: Constitutional AI
- LLaMA: Efficient architecture
- Used for: Generation, chat, code

Encoder-Decoder (T5-style):

- T5: Text-to-text unified
- BART: Denoising approach
- mT5: Multilingual
- Used for: Translation, summarization

Efficient Variants:

- FlashAttention: 2-3x faster[12]
- Linformer: Linear complexity
- Performer: Kernel approximation
- Used for: Long sequences

All modern LLMs are transformer variants!

[1] Estimated from capabilities
[2] Dao et al. (2022) FlashAttention

Transformer Gotchas and Solutions

1. Attention is Quadratic

- Problem: $O(n^2)$ memory for sequence length n
- Solution: Sparse attention, sliding windows
- Example: GPT-3 uses sparse patterns

2. Position Extrapolation

- Problem: Fails on sequences longer than training
- Solution: ALiBi, RoPE, or relative encodings
- Example: LLaMA uses RoPE for 100k+ context

3. Training Instability

- Problem: Large models diverge easily
- Solution: Learning rate warmup, careful initialization
- Example: GPT-3 took months of tuning

Real Example - ChatGPT:

- Uses modified attention (sparse + dense)
- Special position encodings for long context
- Extensive stability modifications

Week 5 Exercise: Build Your Own Mini-GPT

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

Implementation Steps:

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

Key Experiments:

- Compare 1 vs 8 vs 16 attention heads
- Try with/without positional encoding
- Measure GPU utilization vs RNN
- Visualize attention patterns

Bonus Challenges:

- Implement sparse attention for longer sequences
- Add beam search for better generation
- Try different position encoding schemes
- Build a simple chatbot interface

You'll discover: Why transformers took over the world!

Key Takeaways: The Attention Revolution

What we learned:

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

The evolution:

Sequential (RNN) → Parallel (Transformer) → Scale (GPT/BERT)

Why it matters:

- Enabled training on internet-scale data
- Made transfer learning practical
- Started the LLM revolution

Next week: Pre-trained Language Models
How do we use transformers to learn from all of human knowledge?

References and Further Reading

Foundational Papers:

- Vaswani et al. (2017). "Attention Is All You Need", NeurIPS
- Devlin et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers"
- Radford et al. (2018). "Improving Language Understanding by Generative Pre-Training"

Implementation Resources:

- "The Illustrated Transformer" by Jay Alammar
- "The Annotated Transformer" (Harvard NLP)
- Hugging Face Transformers library

Recent Advances:

- FlashAttention: Making attention practical
- Scaling laws for neural language models
- Efficient transformers survey (2022)