Natural Language Processing

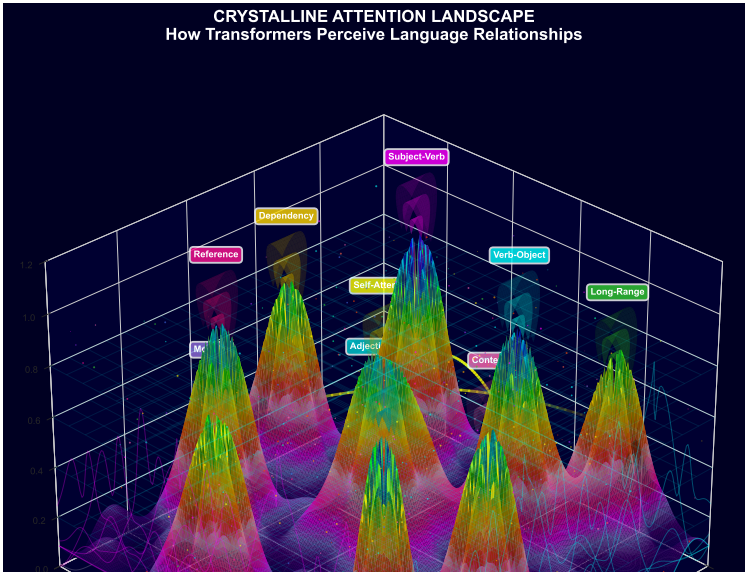
Week 5: The Transformer Revolution

Understanding Attention Mechanisms

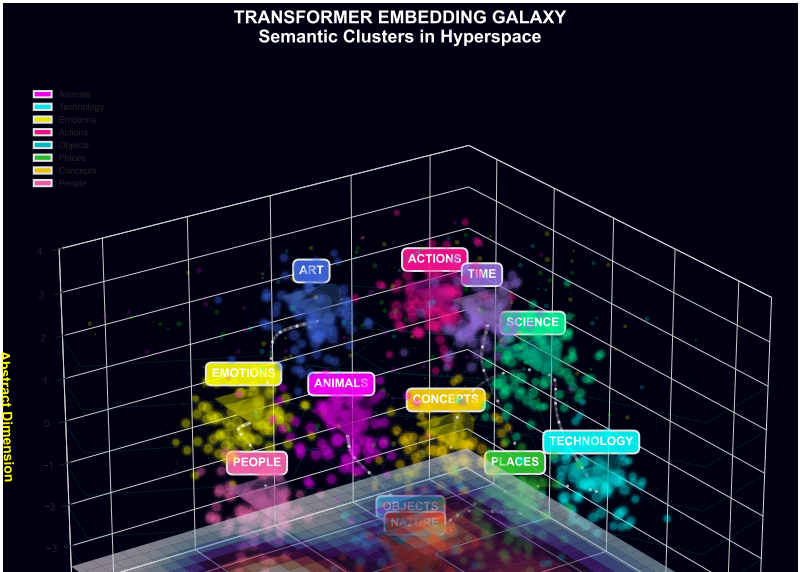
BSc Computer Science

"Attention Is All You Need" - The paper that changed AI forever

The Attention Revolution



The Paradigm Shift in Natural Language Processing



Today's Journey: 4 Parts

Part 1: Why Transformers?

- The GPU bottleneck problem
- Why RNNs couldn't scale
- The parallel processing solution

Part 2: Understanding Attention

- How attention works intuitively
- Query, Key, Value explained
- Multiple heads for different views

Part 3: Building Transformers

- Adding position information
- The complete architecture
- Writing the code

Part 4: Real Applications

- ChatGPT and beyond
- Where transformers excel
- Your hands-on project

Prerequisites Check

Prerequisite

You should know:

- How RNNs work (Week 3)
- Why RNNs have problems with long sequences
- Basic matrix multiplication
- What softmax does

Check Your Understanding

Quick check: Can you explain why RNNs must process words one by one?

If not, review Week 3 first!

Today's Learning Goals:

1. Understand why parallelization matters for modern AI
2. Master the attention mechanism through visual examples
3. Build a mini-transformer from scratch
4. See transformers in action (ChatGPT demo)

Part 1: Why Transformers?

The Problem That Started a Revolution

The 2016 Problem: Wasted Computing Power

Google's Translation Challenge:

Processing "I love machine learning":

1. Process "I" → 100ms
2. Wait for result...
3. Process "love" → 100ms
4. Wait for result...
5. Process "machine" → 100ms
6. Wait for result...
7. Process "learning" → 100ms

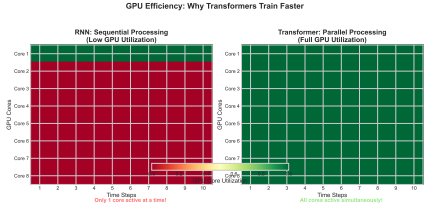
Total: 400ms minimum

Real-World Application

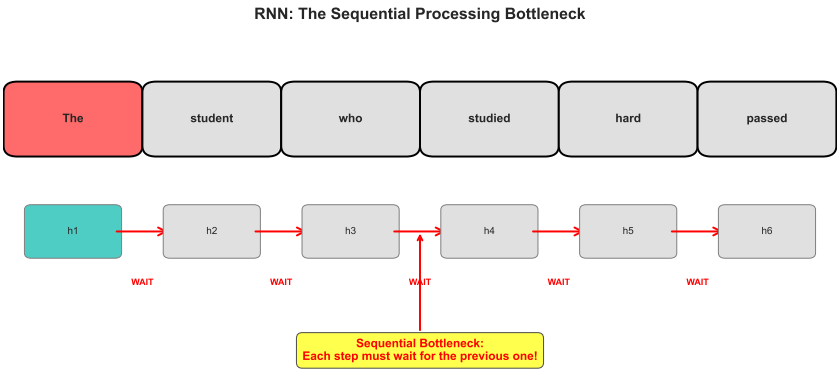
Google Translate handled 100 billion words per day. At 400ms per 4 words, that's 1 million GPU-hours daily!

Intuition

The GPU Paradox:
Imagine having 1000 workers but forcing them to work one at a time. GPUs can do 10,000+ operations simultaneously, but RNNs use only 1!



Why RNNs Must Be Sequential



The Fundamental Constraint:

- Each word needs the hidden state from the previous word
- Can't compute word 5 until word 4 is done
- Like a relay race - must pass the baton!

The Revolutionary Idea: Remove Sequential Processing!

Old Way (RNN):

- Process word 1 ↓
- Process word 2 ↓
- Process word 3 ↓
- Process word 4 ↓

Time: O(n) sequential steps

Like reading one word at a time with your finger

New Way (Transformer):

- Process all words simultaneously!
- Each word looks at all others
- No waiting, no hidden states
- Pure parallel computation

Time: O(1) parallel steps

Like seeing the whole page at once

Result: 100× faster training, better quality, and the birth of ChatGPT!

Key Takeaway:

Transformers solved the parallelization problem by eliminating sequential processing entirely

What we learned:

- RNNs waste 99% of GPU capacity
- Sequential = slow, Parallel = fast
- Removing recurrence was revolutionary

Check Your Understanding

Quick Quiz:
Why can't RNNs use multiple GPUs effectively?

Answer: Each step depends on the previous one

Next: How does attention replace recurrence?

Part 2: Understanding Attention

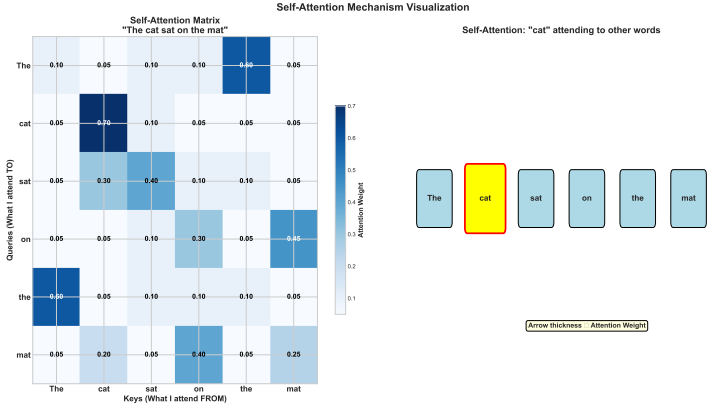
The Mechanism That Powers Modern AI

How Humans Read: The Attention Intuition

Read this sentence: "The student who studied hard passed the exam"

When you see "passed", your brain instantly:

- Looks back at "student" (WHO passed?)
- Looks forward at "exam" (WHAT was passed?)
- Ignores less relevant words like "who" and "the"



The Three Roles: Query, Key, Value

Real-World Application

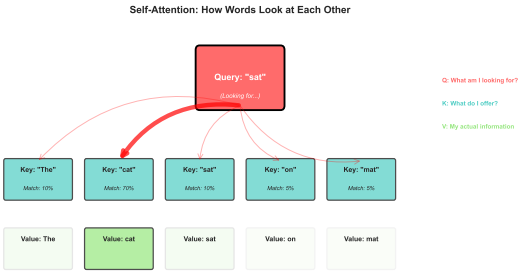
Like searching Google:

- **Query:** Your search term
- **Keys:** Webpage titles
- **Values:** Webpage content

You get back the most relevant content!

In transformers:

- **Q:** "What am I looking for?"
- **K:** "What information do I have?"
- **V:** "What should I retrieve?"



Check Your Understanding

Each word plays all three roles simultaneously. It asks questions (Q), provides answers (K), and gives information (V).

The Attention Formula: Simplified

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

Breaking it down step by step:

1. QK^T: Compute similarity scores
   - How relevant is each key to each query?
   - Higher score = more relevant
2. $\div\sqrt{d_k}$: Scale the scores
   - Prevents numbers from getting too large
   - Keeps gradients stable during training
3. **Softmax**: Convert to probabilities
   - Makes scores sum to 1.0
   - Creates a probability distribution
4. $\times V$: Weighted sum of values
   - Combine values based on attention weights
   - Output = weighted mixture of relevant information

Why QK^T? It's Just Dot Product Similarity!

The Dot Product Measures Similarity:

For two vectors $\vec{a}$ and $\vec{b}$:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$$

What this means:

- Same direction ($\theta = 0$): Maximum similarity
- Perpendicular ($\theta = 90$): No similarity
- Opposite ($\theta = 180$): Negative similarity

Intuition

QK^T computes dot products between all query-key pairs. It's asking: "How similar is what I'm looking for (Q) to what you have (K)?"

Visual Example:

Query: "king" $\rightarrow \vec{q} = [0.9, 0.1, 0.2]$

Keys:

- "queen" $\rightarrow \vec{k}_1 = [0.8, 0.2, 0.3]$
- "car" $\rightarrow \vec{k}_2 = [0.1, 0.9, 0.1]$
- "royal" $\rightarrow \vec{k}_3 = [0.7, 0.0, 0.4]$

Dot products:

- $\vec{q} \cdot \vec{k}_1 = 0.80$ (high similarity!)
- $\vec{q} \cdot \vec{k}_2 = 0.20$ (low similarity)
- $\vec{q} \cdot \vec{k}_3 = 0.71$ (good similarity)

Check Your Understanding

Higher dot product = More attention weight!

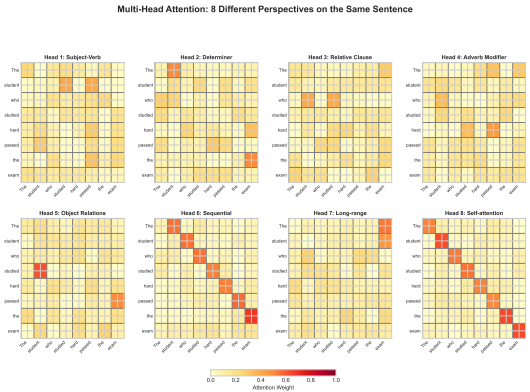
Multi-Head Attention: Different Perspectives

Why multiple heads?
Different heads learn different relationships:

- **Head 1:** Grammar (subject-verb)
- **Head 2:** Meaning (synonyms)
- **Head 3:** Position (nearby words)
- **Head 4:** Topics (related concepts)

Intuition

Like having 8 different experts, each looking for different patterns, then combining their insights!



Check Your Understanding

BERT uses 12 heads, GPT-3 uses 96 heads. More heads = more patterns captured!

Interactive: Visualizing Attention

Exercise: For the sentence "The cat sat on the mat", predict attention weights:

From ↓ To →	The	cat	sat	on	the	mat
sat	0.1	**0.4**	0.2	0.1	0.1	0.1

Your turn: Fill in attention weights for "cat":

From ↓ To →	The	cat	sat	on	the	mat
cat	__	__	__	__	__	__

Intuition

"cat" should pay most attention to "The" (determiner) and "sat" (what the cat did). Remember: weights must sum to 1.0!

Part 2 Summary: Attention Mechanism

Key Takeaway:

Attention allows every word to directly connect to every other word in parallel

What we learned:

- Query-Key-Value mechanism
- Attention formula breakdown
- Multi-head for different patterns
- How to visualize attention

Check Your Understanding

Quick Quiz:
Why do we divide by $\sqrt{d_k}$?

Answer: To prevent softmax saturation when dimensions are large

Next: How to build a complete transformer

Part 3: Building Transformers

From Attention to Architecture

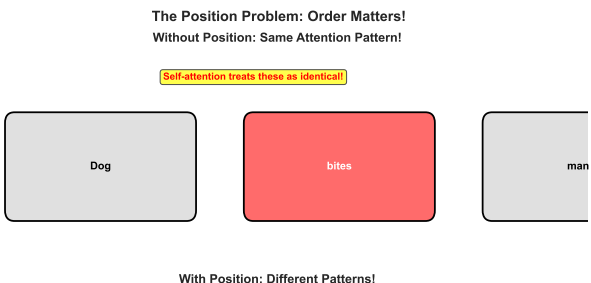
The Position Problem: Order Matters!

Without position information, these are identical to the model:

- "The cat chased the mouse"
- "The mouse chased the cat"
- "Cat mouse the the chased"

Common Misconception

"Attention captures position" - NO!
Attention is permutation invariant. We must add position explicitly.



Positional Encoding: Giving Words an Address

The solution: Add position signals

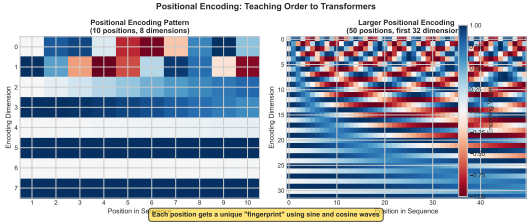
We use sine and cosine waves:

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

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

Intuition

Like GPS coordinates for words!
Each position gets a unique pattern that the model can learn to recognize.



Real-World Application

This allows models to handle sequences longer than anything seen during training!

The Complete Transformer Architecture

Core Components:

1. **Multi-Head Attention**
   - 8-16 parallel attention heads
   - Each head: different pattern
   - Concatenate and project
2. **Feed-Forward Network**
   - Two linear layers
   - ReLU activation between
   - Processes each position independently
3. **Layer Normalization**
   - After each sub-layer
   - Stabilizes training
   - Prevents gradient explosion

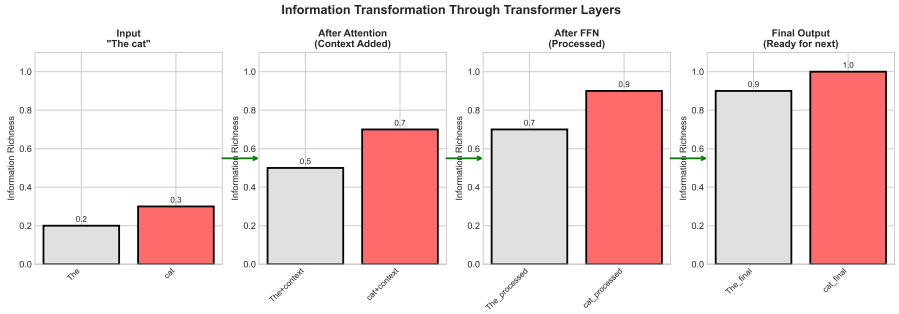
Supporting Mechanisms:

4. **Residual Connections**
   - Skip connections around each layer
   - Output = Layer(x) + x
   - Enables deep networks (96+ layers)
5. **Stacking Layers**
   - 6 layers (original)
   - 12 layers (BERT)
   - 96 layers (GPT-3)
6. **Input/Output**
   - Word embeddings (input)
   - + Positional encoding
   - Final linear projection (output)

Check Your Understanding

Each layer refines the representation. Like an assembly line where each station adds more understanding!

How Information Flows Through Layers



Progressive refinement:

- **Layer 1-2:** Basic patterns (grammar, syntax)
- **Layer 3-4:** Relationships (who did what)
- **Layer 5-6:** High-level meaning (concepts, reasoning)

Check Your Understanding

Each layer builds on the previous one. Like climbing stairs - each step takes you higher!

What Each Component Actually Does

Attention: The Relationship Finder

- Connects every word to every other word
- Finds "who did what to whom"
- Creates a web of relationships
- Example: "cat" $\rightarrow$ "sat", "mat"

Feed-Forward: The Individual Processor

- Works on each position alone
- Adds non-linearity (complexity)
- Like applying a filter to each word
- Transforms: word $\rightarrow$ richer representation

Layer Norm: The Stabilizer

- Keeps values in reasonable range
- Prevents explosion/vanishing
- Like volume control on audio
- Makes training smooth

Residuals: The Memory Lane

- Preserves original information
- Allows model to "remember" input
- Like keeping rough draft while editing
- Enables very deep networks

Intuition

Think of it like editing a document: Attention finds connections (grammar check), FFN processes each word (spell check), Layer Norm keeps formatting consistent, and Residuals keep your original draft!

From Words to Understanding: The Journey

The Complete Transformation Pipeline:

1. **Words → Numbers (Embeddings)**
   - "cat" → [0.2, -0.5, 0.8, ...]
   - Each word gets a unique vector
   - Captures semantic meaning
2. **+ Position → Located in Sequence**
   - Add positional encoding
   - Now model knows "cat" is word #2
   - Preserves word order information
3. **Through Attention → Connected Meanings**
   - Words look at each other
   - Build relationship map
   - "cat" connects strongly to "sat"
4. **Through FFN → Refined Representations**
   - Process each enriched word
   - Add complexity and nuance
   - Deeper understanding emerges
5. **Repeat 6× → Final Understanding**
   - Each layer adds depth
   - From syntax → semantics → reasoning
   - Ready for prediction/generation

Check Your Understanding

It's like reading a book: First you see letters (embeddings), then words in order (position), then sentences

The Code: Simpler Than You Think!

```python
import torch.nn as nn

class SimpleTransformer(nn.Module):
    def __init__(self, vocab_size, d_model=512, n_heads=8, n_layers=6):
        super().__init__()
        # Word embeddings
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoding = PositionalEncoding(d_model)

        # Stack of transformer layers
        self.layers = nn.ModuleList([
            TransformerLayer(d_model, n_heads)
            for _ in range(n_layers)
        ])

        # Output projection
        self.output = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        # Embed and add positions
        x = self.embedding(x)
        x = self.pos_encoding(x)

        # Pass through each layer
        for layer in self.layers:
            x = layer(x)

        # Project to vocabulary
        return self.output(x)
```

Real-World Application

Key Takeaway:

Transformers are surprisingly simple - just attention + position + layers

What we learned:

- Position encoding is essential
- 6 layers of refinement
- Residual connections help training
- Simple code, powerful results

Check Your Understanding

Quick Quiz:
Why do we need residual connections?

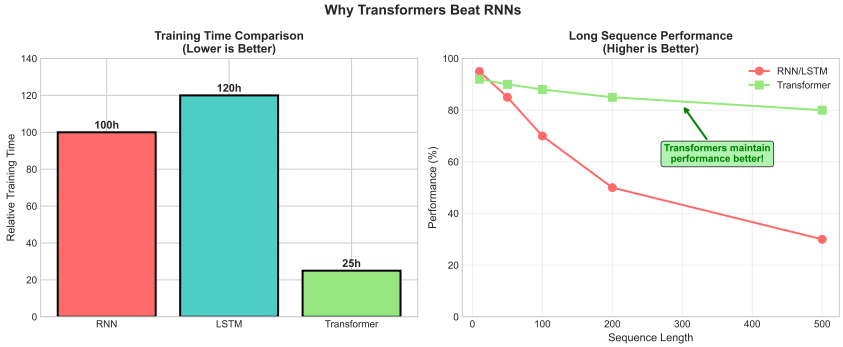
Answer: To prevent vanishing gradients in deep networks

Next: Where are transformers used today?

Part 4: Practical Applications

Transformers in the Real World

Transformers vs RNNs: The Numbers



The advantages are dramatic:

- **Training speed:** 100× faster
- **Accuracy:** 10-20% better on most tasks
- **Long sequences:** Can handle 100,000+ tokens
- **Transfer learning:** Pre-train once, use everywhere

Real-World Application

Transformers Are Everywhere (2024)

Language:

- ChatGPT
- Google Bard
- GitHub Copilot
- DeepL
- Grammarly

Vision:

- DALL-E 3
- Midjourney
- Stable Diffusion
- Medical imaging
- Self-driving cars

Other:

- AlphaFold (proteins)
- Music generation
- Code completion
- Game AI
- Weather prediction

98% of state-of-the-art AI uses transformers

Check Your Understanding

Name an AI application you use daily. It probably uses transformers!

Common Misconceptions About Transformers

Common Misconception

"Transformers are just better RNNs"
No! They're fundamentally different - no recurrence at all.

Common Misconception

"Bigger is always better"
No! A well-tuned 7B model can beat a poorly trained 70B model.

Common Misconception

"Transformers replaced everything"
No! RNNs still win for:

- Streaming data (live transcription)
- Very long sequences (books)
- Edge devices (phones)

Your Lab Project: Build Mini-GPT

What you'll build today:

Part A: Attention (20 min)

- Implement self-attention
- Visualize attention patterns
- Test with examples

Part B: Full Model (20 min)

- Stack transformer layers
- Add position encoding
- Train on Shakespeare

Part C: Analysis (20 min)

- Generate text
- Compare with RNN
- Measure speed difference

Real-World Application

Your model: 10M parameters
ChatGPT: 175B parameters
Same architecture, just scaled!

Resources: Notebook at `week05_transformer_lab.ipynb`

The Transformer Revolution in 5 Points

1. **Parallelization** solved the GPU bottleneck
2. **Self-attention** connects all words directly
3. **Multi-head attention** captures different patterns
4. **Position encoding** preserves word order
5. **Simple architecture**, revolutionary impact

Check Your Understanding

Final Quiz:

- Why is attention $O(n^2)$? *Every word attends to every other word*
- How many heads in BERT? *12 heads*
- What year was the transformer invented? *2017*

Next Week: Pre-trained Language Models (BERT, GPT)

Appendix

Additional Resources and Details

(Optional Material)

Mathematical Details

Complete attention computation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{2}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3}$$

Positional encoding formula:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{4}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{5}$$

Complexity analysis:

- Self-attention: $O(n^2 d)$ where n = sequence length, d = dimension
- RNN: $O(nd)$ but sequential
- Memory: $O(n^2)$ for attention weights

Complete Implementation

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        self.d_k = d_model // n_heads
        self.n_heads = n_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)

        # Linear projections in batch from d_model => h x d_k
        Q = self.W_q(query).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)
        K = self.W_k(key).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)
        V = self.W_v(value).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)

        # Attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        attention_weights = F.softmax(scores, dim=-1)
        context = torch.matmul(attention_weights, V)

        # Concatenate heads
        context = context.transpose(1, 2).contiguous().view(
            batch_size, -1, self.n_heads * self.d_k
        )

        # Final linear projection
        output = self.W_o(context)
        return output
```

References and Further Reading

Essential Papers:

- Vaswani et al. (2017). “Attention Is All You Need” - The original
- Devlin et al. (2019). “BERT” - Bidirectional transformers
- Brown et al. (2020). “GPT-3” - Scaling to 175B parameters

Visual Learning:

- “The Illustrated Transformer” by Jay Alammar
- “Attention is All You Need” video by Yannic Kilcher
- 3Blue1Brown’s neural network series

Hands-on Resources:

- Hugging Face Transformers library
- Andrej Karpathy’s nanoGPT
- Google Colab free GPUs

Recent Advances (2024):

- FlashAttention 2 - 4$\times$ faster training
- Mixture of Experts - Sparse models
- ROPE/ALIBI - Better position encoding

Common Implementation Issues

Problem 1: Out of memory

- Solution: Reduce batch size or sequence length
- Use gradient accumulation
- Try mixed precision training (fp16)

Problem 2: Model not learning

- Check learning rate (try 1e-4 for transformers)
- Ensure proper masking for padding
- Verify position encoding is added correctly

Problem 3: Slow training

- Use torch.compile() in PyTorch 2.0+
- Enable flash attention if available
- Check GPU utilization with nvidia-smi

Problem 4: Poor generation quality

- Increase model size or training data
- Tune temperature and sampling parameters
- Check for repetition in training data