Understanding Transformers: From First Principles to Mastery

A Progressive Learning System with Hand-Calculable Examples

ToyAI Educational Project

2025-11-28

Contents

Understanding Transformers: From First Principles to Mastery

A Progressive Learning System with Hand-Calculable Examples

Learn transformers from first principles using 2x2 matrices you can compute by hand.

Chapter 1

Chapter 1: Why Transformers?

1.1 The Problem: Sequence Modeling

Imagine you're reading a sentence: "The cat sat on the mat." To understand this, you need to: - Remember that "cat" is the subject - Connect "sat" to "cat" (the cat did the sitting) - Understand "on the mat" describes where the cat sat

This is **sequence modeling**: understanding how elements in a sequence relate to each other.

Real-world applications: - **Language translation**: "Hello" → "Hola" (but context matters!) - **Text generation**: Given "The weather is", predict "nice" or "terrible" - **Question answering**: "Who wrote Hamlet?" requires understanding context - **Code completion**: IDE suggests next token based on previous code

1.2 The Challenge: Long-Range Dependencies

In the sentence "The cat that I saw yesterday sat on the mat", the word "sat" must connect to "cat" even though many words separate them. This is a **long-range dependency**.

Why this is hard: - Information must flow across many positions - Context from early in the sequence affects later predictions - Traditional models struggle with this

1.3 Previous Solutions: RNNs and Their Limitations

Recurrent Neural Networks (RNNs) were the previous solution: - Process sequence one token at a time - Maintain hidden state that carries information forward - Can theoretically handle long sequences

But RNNs have problems: 1. **Sequential bottleneck**: Must process tokens one-by-one (can't parallelize) 2. **Vanishing gradients**: Information gets lost over long sequences 3. **Forgetting**: Early context fades as sequence gets longer

Example of RNN limitation:

```
Input: "The cat that I saw yesterday in the park near my house sat on the mat"
RNN state: [cat] → [saw] → [yesterday] → [park] → [house] → [sat] → [mat]
                    ↑                                     ↑
              "cat" info                            "cat" info is weak!
```

By the time we reach “sat”, the RNN has forgotten much about “cat”.

1.4 The Transformer Solution: Attention

Transformers solve this with attention: - Every position can directly attend to every other position - No sequential bottleneck - all positions processed in parallel - Information flows directly where needed

Key insight: Instead of forcing information through a sequential chain, let each position “look” at all other positions and decide what’s relevant.

Example with attention:

```
Position "sat" can directly attend to:
- "cat" (high attention - subject)
- "mat" (high attention - object)
- "yesterday" (medium attention - time context)
- "the" (low attention - not very informative)
```

1.5 Why Attention is Powerful

1. **Direct connections**: No information loss through sequential processing
2. **Parallel computation**: All positions computed simultaneously (fast on GPUs)
3. **Interpretable**: Can see what the model is “paying attention to”
4. **Scalable**: Works well with very long sequences

1.6 Real-World Impact

Transformers power: - **GPT models**: ChatGPT, GPT-4 (text generation) - **BERT**: Google search, language understanding - **Code models**: GitHub Copilot, Codex - **Translation**: Google Translate - **Image models**: Vision transformers (ViT)

1.7 The Core Innovation

The transformer’s innovation isn’t a single breakthrough, but a combination: 1. **Self-attention**: Each position attends to all positions 2. **Parallel processing**: No sequential dependency 3. **Scaled dot-product**:

Efficient attention computation 4. **Stacked layers**: Multiple attention layers for complex patterns

1.8 Learning Objectives Recap

- Understand sequence modeling challenges
- See why RNNs struggle with long-range dependencies
- Understand how attention solves these problems
- Connect to real-world transformer applications

1.9 Key Concepts Recap

- **Sequence-to-sequence tasks**: Input sequence → output sequence
- **Long-range dependencies**: Connections across many positions
- **Parallel computation**: All positions processed simultaneously
- **Attention mechanism**: Direct connections between positions

---

Chapter 2

Chapter 2: The Matrix Core

2.1 Why Matrices?

Neural networks are fundamentally built on **matrix operations**. Every layer, every transformation, every computation involves matrices. But why?

The answer: Matrices are the mathematical tool that lets us: 1. Transform data efficiently 2. Learn patterns from examples 3. Compute gradients for training 4. Parallelize on GPUs/TPUs

2.2 What is a Matrix?

A **matrix** is a rectangular array of numbers. For our 2×2 case:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Why 2×2? Small enough to compute by hand, but captures all the essential operations.

2.3 Matrix Multiplication: The Core Operation

Matrix multiplication is how neural networks transform data.

For matrices A (2×2) and B (2×2):

$$C = AB = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

What this means: - Each element of C is a **weighted combination** of elements from A and B - The weights come from the matrix structure itself - This is how networks "mix" information

Example:

```
A = [1, 0]    B = [0.5, 0.5]
    [0, 1]        [0.5, 0.5]


C = A × B = [0.5, 0.5]
            [0.5, 0.5]
```

The identity matrix A doesn't change B - this is like a "pass-through" layer.

2.4 Linear Transformations

Matrix multiplication = linear transformation

When we multiply a vector by a matrix, we: - **Rotate** the vector in space - **Scale** its components - **Project** it to a new space

Example:

```
Vector: [1, 0]  (pointing in x-direction)
Matrix: [0, -1]  (rotation matrix)
        [1,  0]


Result: [0, 1]  (now pointing in y-direction - rotated 90°!)
```

Why this matters for learning: - Different matrices = different transformations - Learning = finding the right transformation - Weights in matrices are what get updated during training

2.5 Vector Spaces

Vectors are points in space. For 2D: - $[1, 0]$ = point at (1, 0) - $[0, 1]$ = point at (0, 1) - $[0.5, 0.5]$ = point at (0.5, 0.5)

Vector space = all possible points/vectors

Why this matters: - Embeddings live in vector spaces - Attention computes similarity in vector space - Learning = moving points in space to create patterns

2.6 Dot Products: Measuring Similarity

Dot product of two vectors measures how "aligned" they are:

$$\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2$$

Properties: - **High dot product** = vectors point in similar direction = similar - **Low dot product** = vectors point in different directions = different - **Zero dot product** = vectors are perpendicular = unrelated

Example:

```
[1, 0] · [1, 0] = 1×1 + 0×0 = 1    (same direction)
[1, 0] · [0, 1] = 1×0 + 0×1 = 0    (perpendicular)
[1, 0] · [-1, 0] = 1×(-1) + 0×0 = -1  (opposite direction)
```

In attention: Dot product between Query and Key measures how relevant they are!

2.7 Transpose: Changing Perspective

Transpose swaps rows and columns:

$$A^T = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

Why transpose? - Matrix multiplication requires compatible dimensions - $A \times B$ works if A has n columns and B has n rows - Transpose lets us align dimensions: $A \times B^T$

In attention: We compute $Q \times K^T$ to get all pairwise dot products at once!

2.8 Why Matrices Enable Learning

1. Expressiveness: - Linear transformations can represent any linear relationship - With non-linearities (ReLU), can approximate any function - Multiple layers = composition of transformations = complex patterns

2. Gradient Flow: - Matrix operations have clean derivatives - Chain rule works beautifully: $\frac{d}{dW}(f(g(x))) = \frac{df}{dg}\frac{dg}{dW}$ - Enables backpropagation

3. Parallelization: - GPUs have "tensor cores" optimized for matrix multiply - Can process thousands of operations simultaneously - Makes training feasible

4. Composition: - Stack matrices: $f(g(x))$ where f and g are matrix operations - Each layer adds complexity - Deep networks = many composed transformations

2.9 Matrix Calculus Basics

For $C = AB$: - $\frac{\partial L}{\partial A} = \frac{\partial L}{\partial C}B^T$ - $\frac{\partial L}{\partial B} = A^T\frac{\partial L}{\partial C}$

Why this matters: - Backpropagation needs these rules - Gradients flow backward through matrix operations - Enables training

2.10 Learning Objectives Recap

- Understand why matrices are fundamental
- Master matrix multiplication
- See how linear transformations work
- Understand gradient flow through matrices

2.11 Key Concepts Recap

- **Matrix multiplication**: Core operation for transformations
- **Linear transformations**: How matrices change vectors
- **Vector spaces**: Where embeddings and computations live
- **Transpose**: Tool for dimension alignment
- **Matrices enable learning**: Expressiveness + gradients + parallelization

Chapter 3

Chapter 3: Embeddings: Tokens to Vectors

3.1 The Problem: Discrete vs. Continuous

Tokens (words, characters, subwords) are **discrete**: - "cat" is just a symbol - No mathematical relationship between "cat" and "dog" - Can't do arithmetic: "cat" + "dog" = ???

Neural networks need **continuous** values: - Matrix operations require numbers - Gradients need smooth functions - Learning needs measurable similarity

Solution: Convert discrete tokens to continuous vectors = **embeddings**

3.2 What are Embeddings?

Embeddings map each token to a vector (point in space):

```
Token "cat" → Vector [0.3, 0.7, -0.2, ...]
Token "dog" → Vector [0.4, 0.6, -0.1, ...]
Token "mat" → Vector [-0.1, 0.2, 0.8, ...]
```

Key insight: Similar tokens should have similar vectors!

3.3 One-Hot Encoding: The Starting Point

One-hot encoding is the simplest embedding: - Vocabulary size = V - Each token gets a vector of length V - Only one element is 1, rest are 0

Example (vocab: A, B, C, D):

```
A → [1, 0, 0, 0]
B → [0, 1, 0, 0]
C → [0, 0, 1, 0]
```

```
D → [0, 0, 0, 1]
```

Problems with one-hot: - Vectors are orthogonal (no similarity) - Dimension = vocabulary size (huge for large vocabs!) - No semantic relationships

3.4 Learned Embeddings: The Solution

Learned embeddings are vectors that get updated during training: - Start random - Learn to capture semantic relationships - Similar meanings → similar vectors

Example (learned):

```
"cat" → [0.3, 0.7, -0.2]
"dog" → [0.4, 0.6, -0.1]  (similar to "cat"!)
"mat" → [-0.1, 0.2, 0.8]  (different from "cat")
```

How it works: - Embedding matrix: $E \in \mathbb{R}^{V \times d}$ - V = vocabulary size - d = embedding dimension (e.g., 2 for our examples, 768 for BERT) - Lookup: token i → row i of E

3.5 Embedding Dimensions

Dimension choice: - **Too small**: Can't capture enough information - **Too large**: Overfitting, slow computation - **Sweet spot**: Balance capacity and efficiency

Our examples: $d = 2$ (hand-calculable!) **Real models:** $d = 768$ (BERT), $d = 12,288$ (GPT-3)

3.6 Semantic Spaces

Embeddings create a "semantic space": - Tokens with similar meanings are close together - Tokens with different meanings are far apart - Relationships emerge: "king" - "man" + "woman" ≈ "queen"

Example in 2D:

```
A = [1, 0]  (corner of space)
B = [0, 1]  (different corner)
C = [1, 1]  (combination)
D = [0, 0]  (origin)
```

3.7 Fixed vs. Learned Embeddings

Fixed embeddings (our examples): - Pre-defined, don't change - Simple for learning - Example: A=[1,0], B=[0,1]

Learned embeddings (real models): - Updated during training - Capture task-specific semantics - Much more powerful

In our examples: We use fixed embeddings to focus on attention and training mechanics.

3.8 Embedding Lookup

Process: 1. Token index: "cat" → index 42 2. Lookup: $E[42]$ → vector $[0.3, 0.7, -0.2, ...]$ 3. Use vector in computations

Mathematically:

$$\text{embedding}(i) = E[i]$$

Where E is the embedding matrix and i is the token index.

3.9 Why Embeddings Matter

1. Enable computation: - Can't do math on "cat" - Can do math on $[0.3, 0.7, -0.2]$

2. Capture relationships: - Similar tokens → similar vectors - Enables attention to find relevant tokens

3. Learnable: - Embeddings adapt to task - Better embeddings = better model

3.10 Learning Objectives Recap

- Understand why embeddings are needed
- See how discrete tokens become vectors
- Understand embedding spaces and dimensions
- Know difference between fixed and learned embeddings

3.11 Key Concepts Recap

- **Token vocabulary**: Set of all possible tokens
- **Embedding matrices**: Map tokens to vectors
- **Vector representations**: Continuous, learnable
- **Semantic spaces**: Where meaning lives

3.12 Mathematical Foundations Recap

- **One-hot encoding**: Simple but limited
- **Embedding lookup**: $E[i]$ for token i
- **Embedding dimensions**: Balance capacity and efficiency

Chapter 4

Chapter 4: Attention Intuition

4.1 The Core Question

When processing a sequence, each position needs to ask: > **"Which other positions contain information relevant to me?"**

Example: In "The cat sat on the mat" - Position "sat" needs to know about "cat" (subject) - Position "mat" needs to know about "sat" (verb) - Position "the" (first) is less relevant to "mat"

Attention is the mechanism that answers this question.

4.2 The Query/Key/Value Metaphor

Think of attention like a **library search system**:

4.2.0.1 Query (Q): "What am I looking for?"

Query represents what information a position needs: - "sat" needs: "What is the subject?" - "mat" needs: "What verb describes location?"

In vectors: Query is a learned representation of "what I'm searching for"

4.2.0.2 Key (K): "What do I have to offer?"

Key represents what information each position contains: - "cat" offers: "I am a noun, I am the subject" - "sat" offers: "I am a verb, I describe action" - "the" offers: "I am an article, I'm not very informative"

In vectors: Key is a learned representation of "what I advertise"

4.2.0.3 Value (V): "What is my actual content?"

Value is the actual information to retrieve: - Once we decide "cat" is relevant, we retrieve its value - Value contains the semantic content we actually use

In vectors: Value is what gets weighted and combined

4.3 The Search Process

Step 1: Match Query to Keys

```
Query "sat": "I need the subject"
Key "cat": "I am a noun, I am the subject"  ← Match!
Key "the": "I am an article"                ← No match
```

Step 2: Compute Relevance Scores - High score = Query matches Key = relevant - Low score = Query doesn't match Key = not relevant

Step 3: Convert to Probabilities (Softmax) - Scores → probabilities (attention weights) - Sum to 1.0 (probability distribution)

Step 4: Retrieve Values - Weighted sum of values - High attention weight → more contribution from that value

4.4 Search Engine Analogy

Google Search: 1. **Query**: Your search terms ("transformer attention") 2. **Keys**: Keywords on web pages 3. **Relevance**: How well keywords match query 4. **Values**: Actual webpage content 5. **Result**: Weighted combination of relevant pages

Transformer Attention: 1. **Query**: What position needs 2. **Keys**: What each position offers 3. **Relevance**: Dot product (similarity) 4. **Values**: Actual content to retrieve 5. **Output**: Weighted combination of values

Same idea, different domain!

4.5 Database Query Analogy

SQL Query:

```sql
SELECT content FROM pages
WHERE keywords MATCH "transformer attention"
ORDER BY relevance DESC
```

Attention:

```
SELECT values FROM positions
WHERE keys MATCH query
WEIGHT BY attention_scores
```

Attention is like a learned, differentiable database query!

4.6 Why Three Components (Q, K, V)?

Why not just one? Because they serve different purposes:

1. **Q and K determine relevance** (what to attend to)
2. **V provides content** (what to retrieve)

Separation allows: - Learning what to search for (Q) - Learning what to advertise (K) - Learning what content to provide (V)

Example: A position might: - Search for "subject" (Q) - Advertise "I'm a noun" (K) - Provide "cat" meaning (V)

All three are learned separately!

4.7 Attention Weights as Probabilities

Attention weights are probabilities: - Each position gets a weight - Weights sum to 1.0 - Higher weight = more attention

Example:

```
Position "sat" attending to:
- "cat": 0.6  (60% attention - subject!)
- "the": 0.1  (10% attention - not very relevant)
- "on": 0.2   (20% attention - preposition)
- "mat": 0.1  (10% attention - object)
Sum: 1.0 □
```

Interpretation: "sat" pays 60% attention to "cat" because it's the subject.

4.8 How Relevance is Computed

Dot product measures alignment:

$$\text{score} = Q \cdot K = \sum_i Q_i K_i$$

Why dot product? - High when Q and K point in similar direction - Low when they point in different directions - Zero when perpendicular (unrelated)

After dot product: 1. Scale by $\sqrt{d_k}$ (prevents large values) 2. Apply softmax (convert to probabilities) 3. Use as weights for values

4.9 Information Retrieval Perspective

Classic IR: Given query, find relevant documents

Attention: Given query vector, find relevant position vectors

Both: - Compute similarity (dot product) - Rank by relevance - Retrieve and combine content

Difference: Attention is **learned** - the model discovers what "relevant" means!

4.10 The Magic: Learned Relevance

Fixed relevance (like keyword matching): - "cat" always matches "cat" - Can't learn new relationships

Learned relevance (attention): - Model learns what makes positions relevant - "cat" might become relevant to "feline" even if they don't share words - Adapts to task

This is why transformers are powerful!

4.11 Learning Objectives Recap

- Understand Q/K/V metaphor
- See attention as search mechanism
- Understand relevance computation
- Connect to information retrieval

4.12 Key Concepts Recap

- **Query**: "What am I looking for?"
- **Key**: "What do I have to offer?"
- **Value**: "What is my actual content?"
- **Attention weights**: Probabilities of relevance

4.13 Intuitive Explanations Recap

- **Search engine**: Query matches keywords, retrieves pages
- **Database query**: SELECT WHERE MATCH, ORDER BY relevance

- **Information retrieval**: Find relevant content, combine it

---

Chapter 5

Example 1: Minimal Forward Pass

5.1 Theory

Goal: Understand how a transformer makes predictions (no training yet)

What You'll Learn: - Forward pass computation - Attention mechanism step-by-step - How context is created - How predictions are made

5.2 The Task

Given input sequence "A B", predict the next token. We'll compute probabilities for each possible token (A, B, C, D) without any training - just to see how the forward pass works.

5.3 Model Architecture

- Fixed token embeddings
- Fixed Q, K, V projection matrices
- Scaled dot-product attention
- Output projection to vocabulary
- Softmax to get probabilities

5.4 Step-by-Step Computation

1. **Token Embeddings**: Convert "A" and "B" to 2D vectors
2. **Q/K/V Projections**: Create Query, Key, Value vectors
3. **Attention Scores**: Compute similarity between queries and keys
4. **Attention Weights**: Apply softmax to get probability distribution
5. **Context Vector**: Weighted sum of values

6. **Output Logits**: Project context to vocabulary space
7. **Probabilities**: Apply softmax to get final predictions

5.5 Hand Calculation Guide

See worksheet for step-by-step template.

5.5.0.1 Attention Formula

The scaled dot-product attention is:

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

Where: - Q: Query matrix (what we're looking for) - K: Key matrix (what information is available) - V: Value matrix (the actual content) - d_k: Dimension of keys (scaling factor)

5.5.0.2 Why Scaling?

Without the $\sqrt{d_k}$ scaling, dot products grow with dimension, causing: - Softmax saturation (probabilities near 0 or 1) - Vanishing gradients - Numerical instability

Scaling keeps variance approximately constant.

5.5.0.3 Softmax Properties

For input vector **x** $= [x_1, x_2, \dots, x_n]$:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Properties: - All outputs are positive - Outputs sum to 1 (probability distribution) - Differentiable everywhere - Preserves relative ordering

5.6 Exercises

1. Compute attention scores by hand for given Q, K matrices
2. Verify softmax computation
3. Trace through complete forward pass
4. Compare hand calculation to code output

5.7 Hand Calculation Worksheet

5.7.1 Forward Pass Only - No Training

5.7.1.1 Initial Values

Token Embeddings: - A = [1, 0] - B = [0, 1] - C = [1, 1] - D = [0, 0]

Projection Weights:

```
WQ = [0.1, 0.0]
     [0.0, 0.1]

WK = [0.1, 0.0]
     [0.0, 0.1]

WV = [0.1, 0.0]
     [0.0, 0.1]
```

Output Projection:

```
WO = [0.1, 0.0]
     [0.0, 0.1]
```

5.7.1.2 Step 1: Token Embeddings

Input sequence: [A, B]

```
X = [1, 0]   ← Token A
    [0, 1]   ← Token B
```

5.7.1.3 Step 2: Q, K, V Projections

Q = X × WQ:

```
Q[0][0] = 1×0.1 + 0×0.0 = 0.1
Q[0][1] = 1×0.0 + 0×0.1 = 0.0
Q[1][0] = 0×0.1 + 1×0.0 = 0.0
Q[1][1] = 0×0.0 + 1×0.1 = 0.1

Q = [0.1, 0.0]
    [0.0, 0.1]
```

K = X × WK: (same as Q, since WK = WQ)

```
K = [0.1, 0.0]
    [0.0, 0.1]
```

V = X × WV: (same as Q)

```
V = [0.1, 0.0]
    [0.0, 0.1]
```

5.7.1.4 Step 3: Attention Scores

For position 1 (B), compute attention to positions 0 and 1:

K^T = K^T:

```
K^T = [0.1, 0.0]
      [0.0, 0.1]
```

Raw Scores = Q × K^T:

```
For position 1:
score[1][0] = Q[1] · K[0] = [0.0, 0.1] · [0.1, 0.0] = 0.0×0.1 + 0.1×0.0 = 0.0
score[1][1] = Q[1] · K[1] = [0.0, 0.1] · [0.0, 0.1] = 0.0×0.0 + 0.1×0.1 = 0.01
```

Scale by 1/√2 ≈ 0.707:

```
scaled_score[1][0] = 0.0 × 0.707 = 0.0
scaled_score[1][1] = 0.01 × 0.707 = 0.00707
```

5.7.1.5 Step 4: Attention Weights (Softmax)

For position 1, scores = [0.0, 0.00707]

Softmax: 1. Find max: max(0.0, 0.00707) = 0.00707 2. Subtract max: [0.0 - 0.00707, 0.00707 - 0.00707] = [-0.00707, 0.0] 3. Exponentiate: [exp(-0.00707), exp(0.0)] ≈ [0.9929, 1.0] 4. Sum: 0.9929 + 1.0 = 1.9929 5. Normalize: [0.9929/1.9929, 1.0/1.9929] ≈ [0.498, 0.502]

Attention weights for position 1: - Weight on position 0 (A): ≈ 0.498 - Weight on position 1 (B): ≈ 0.502

5.7.1.6 Step 5: Context Vector

Context = weighted sum of values:

```
context[0] = 0.498 × V[0][0] + 0.502 × V[1][0]
           = 0.498 × 0.1 + 0.502 × 0.0
           = 0.0498

context[1] = 0.498 × V[0][1] + 0.502 × V[1][1]
           = 0.498 × 0.0 + 0.502 × 0.1
           = 0.0502
```

```
Context = [0.0498, 0.0502]
```

5.7.1.7 Step 6: Output Logits

Logits = Context × WO (adapted for 4 tokens):

For simplicity, we'll compute: - logit(A) = context[0] × 0.1 + context[1] × 0.0 = 0.0498 × 0.1 = 0.00498 - logit(B) = context[0] × 0.0 + context[1] × 0.0 = 0.0 - logit(C) = context[0] × 0.0 + context[1] × 0.1 = 0.0502 × 0.1 = 0.00502 - logit(D) = context[0] × 0.0 + context[1] × 0.0 = 0.0

5.7.1.8 Step 7: Probabilities (Softmax)

Logits = [0.00498, 0.0, 0.00502, 0.0]

1. Max: 0.00502
2. Subtract: [-0.00004, -0.00502, 0.0, -0.00502]
3. Exponentiate: [exp(-0.00004), exp(-0.00502), exp(0.0), exp(-0.00502)] ≈ [0.99996, 0.995, 1.0, 0.995]
4. Sum: ≈ 3.99
5. Normalize: [0.250, 0.249, 0.251, 0.249]

Final Probabilities: - P(A) ≈ 0.250 (25.0%) - P(B) ≈ 0.249 (24.9%) - P(C) ≈ 0.251 (25.1%) - P(D) ≈ 0.249 (24.9%)

5.7.1.9 Verification

- Probabilities sum to 1.0 (approximately)
- All probabilities are positive
- Roughly equal probabilities (model hasn't learned yet)

5.7.1.10 Common Mistakes

1. **Forgetting scaling factor**: Always divide by √d_k
2. **Softmax numerical issues**: Always subtract max before exponentiating
3. **Dimension mismatches**: Check matrix dimensions at each step
4. **Wrong attention position**: Make sure you're computing attention for the correct position

5.7.1.11 Next Steps

Compare your hand calculation to the code output. They should match (within rounding error).

Then proceed to Example 2 to see how training changes these probabilities!

5.8 Code Implementation

```
/**
 * ============================================================================
 * Example 1: Minimal Forward Pass
 * ============================================================================
 *
 * Goal: Understand how a transformer makes predictions (no training yet)
 *
 * This example demonstrates:
 * - Token embeddings
 * - Q, K, V projections
 * - Scaled dot-product attention
 * - Output projection
 * - Softmax to get probabilities
 *
 * All computations use 2x2 matrices that can be verified by hand.
 */

#include "../../src/core/Matrix.hpp"
#include "../../src/core/Embedding.hpp"
#include "../../src/core/LinearProjection.hpp"
#include "../../src/core/Attention.hpp"
#include "../../src/core/Softmax.hpp"
#include <iostream>
#include <vector>
#include <iomanip>

int main() {
    std::cout << std::string(70, '=') << "\n";
    std::cout << "Example 1: Minimal Forward Pass\n";
    std::cout << "Goal: Understand how predictions are made\n";
    std::cout << std::string(70, '=') << "\n\n";

    // ========================================================================
    // Setup: Define the model
    // ========================================================================

    std::cout << "MODEL SETUP\n";
    std::cout << std::string(70, '-') << "\n\n";
    // Token embeddings (fixed)
    Embedding embedding;
    embedding.print();
    std::cout << "\n";

    // Q, K, V projection weights (fixed, small values)
    Matrix WQ(0.1, 0.0, 0.0, 0.1);  // Scaled identity
    Matrix WK(0.1, 0.0, 0.0, 0.1);
    Matrix WV(0.1, 0.0, 0.0, 0.1);

    LinearProjection projQ(WQ, false);  // Not trainable
    LinearProjection projK(WK, false);
    LinearProjection projV(WV, false);

    std::cout << "Projection Weights:\n";
    WQ.print("WQ");
    WK.print("WK");
    WV.print("WV");
    std::cout << "\n";

    // Output projection: maps 2D context → 4 logits
    // For simplicity, we'll use a 2x2 matrix and extract relevant elements
    // In practice, this would be 2x4, but for 2x2 we'll adapt
    Matrix WO(0.1, 0.0, 0.0, 0.1);
    std::cout << "Output Projection:\n";
    WO.print("WO");
    std::cout << "\n";

    // Attention and softmax
    Attention attention;
    Softmax softmax;

    // ========================================================================
    // Forward Pass: Input "A B"
    // ========================================================================

    std::cout << std::string(70, '=') << "\n";
    std::cout << "FORWARD PASS: Input sequence [A, B]\n";
    std::cout << std::string(70, '=') << "\n\n";
    // Step 1: Get embeddings
    std::cout << "Step 1: Token Embeddings\n";
    std::cout << std::string(70, '-') << "\n";
    std::vector<int> tokens = {0, 1};  // A=0, B=1
    Matrix X = embedding.forwardSequence(tokens);
    X.print("X (embeddings)");
    std::cout << "  Row 0: Token A = [1, 0]\n";
    std::cout << "  Row 1: Token B = [0, 1]\n\n";

    // Step 2: Project to Q, K, V
    std::cout << "Step 2: Q, K, V Projections\n";
    std::cout << std::string(70, '-') << "\n";
    Matrix Q = projQ.forward(X);
    Matrix K = projK.forward(X);
    Matrix V = projV.forward(X);

    Q.print("Q (queries)");
    K.print("K (keys)");
    V.print("V (values)");
    std::cout << "\n";

    // Step 3: Attention
    std::cout << "Step 3: Scaled Dot-Product Attention\n";
    std::cout << std::string(70, '-') << "\n";
    Attention::AttentionResult attn_result = attention.forward(Q, K, V);
    attention.printSteps(Q, K, V, attn_result);

    // Step 4: Output projection
    // For position 1 (B), use its context vector to predict next token
    // Context vector is attn_result.output row 1
    std::cout << "Step 4: Output Projection (Context → Logits)\n";
    std::cout << std::string(70, '-') << "\n";

    // Extract context vector for position 1 (B)
    double context[2] = {attn_result.output.get(1, 0), attn_result.output.get(1, 1)};
    std::cout << "Context vector (from position 1): ["
              << context[0] << ", " << context[1] << "]\n\n";

    // Project to logits (for 4 tokens: A, B, C, D)
    // We'll compute: logit[i] = context · WO_column_i
    // For simplicity with 2x2, we'll map:
    // - WO[0][0] → logit(A)
    // - WO[0][1] → logit(B)
    // - WO[1][0] → logit(C)
    // - WO[1][1] → logit(D)
    std::vector<double> logits(4);
    logits[0] = context[0] * WO.get(0, 0) + context[1] * WO.get(1, 0);
    logits[1] = context[0] * WO.get(0, 1) + context[1] * WO.get(1, 1);
    logits[2] = context[0] * WO.get(0, 0) + context[1] * WO.get(1, 0);  // Reuse for C
    logits[3] = context[0] * WO.get(0, 1) + context[1] * WO.get(1, 1);  // Reuse for D

    // Actually, let's use a proper 2x4 mapping conceptually
    // For hand calculation, we'll use:
    // WO conceptually maps 2D → 4D, but we'll adapt
    // Let's define: logit[i] = context[0] * WO_row0[i] + context[1] * WO_row1[i]
    // But WO is 2x2, so we'll use:
    logits[0] = context[0] * 0.1 + context[1] * 0.0;  // A
    logits[1] = context[0] * 0.0 + context[1] * 0.0;  // B
    logits[2] = context[0] * 0.0 + context[1] * 0.1;  // C
    logits[3] = context[0] * 0.0 + context[1] * 0.0;  // D

    std::cout << "Logits (raw scores):\n";
    const char* token_names[] = {"A", "B", "C", "D"};
    for (int i = 0; i < 4; i++) {
        std::cout << "  logit(" << token_names[i] << ") = " << logits[i] << "\n";
    }
    std::cout << "\n";

    // Step 5: Softmax to get probabilities
    std::cout << "Step 5: Softmax (Logits → Probabilities)\n";
    std::cout << std::string(70, '-') << "\n";
    std::vector<double> probs = softmax.forward(logits);
    softmax.printSteps(logits, probs);

    // ========================================================================
    // Summary
    // ========================================================================
    std::cout << std::string(70, '=') << "\n";
    std::cout << "SUMMARY\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Input: [A, B]\n";
    std::cout << "Predicted probabilities for next token:\n";
    for (int i = 0; i < 4; i++) {
        std::cout << "  P(" << token_names[i] << ") = "
                  << std::fixed << std::setprecision(4) << probs[i]
                  << " (" << (probs[i] * 100) << "%)\n";
    }
    std::cout << "\n";

    std::cout << "Note: This is BEFORE training. The model hasn't learned\n";
    std::cout << "the pattern (A, B) → C yet. All tokens have roughly equal\n";
    std::cout << "probability. After training (Example 2+), C's probability\n";
    std::cout << "will increase.\n\n";

    std::cout << "Next: See Example 2 to learn how training works!\n";
    std::cout << std::string(70, '=') << "\n";

    return 0;
}
```

Chapter 6

Example 2: Single Training Step

6.1 Theory

Goal: Understand how one weight update works

What You'll Learn: - Loss functions - Gradient computation - Gradient descent - How models learn from examples

6.2 The Task

Train the model on example: Input "A B" → Target "C"

We'll update only the output projection matrix W_O in this example to keep it simple.

6.3 Model Architecture

- Same as Example 1
- But W_O is now trainable
- W_Q, W_K, W_V remain fixed

6.4 Training Process

1. **Forward Pass**: Compute prediction (same as Example 1)
2. **Compute Loss**: Measure how wrong the prediction is
3. **Compute Gradients**: Calculate how to change weights
4. **Update Weights**: Actually change W_O using gradient descent

6.5 Loss Function

Cross-entropy loss for next-token prediction:

$$L = -\log P(y_{\text{target}})$$

Where $P(y_{\text{target}})$ is the model's predicted probability for the correct token.

Properties: - Lower when model is confident and correct - Higher when model is wrong or uncertain - Differentiable (enables gradient descent)

6.6 Gradient Computation

For softmax + cross-entropy, the gradient w.r.t. logits is:

$$\frac{\partial L}{\partial \text{logit}_i} = P(i) - \mathbf{1}[i = \text{target}]$$

Where $\mathbf{1}[\cdot]$ is the indicator function.

This elegant formula means: - If model predicts too high probability for wrong token → push logit down - If model predicts too low probability for correct token → push logit up

6.7 Gradient Descent Update

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \frac{\partial L}{\partial W}$$

Where η is the learning rate.

6.8 Hand Calculation Guide

See worksheet

6.8.0.1 Chain Rule Basics

For composite function $f(g(x))$:

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

In our case:

$$L \leftarrow \text{softmax}(\text{logits}) \leftarrow \text{logits} \leftarrow W_O \times \text{context}$$

We compute gradients backward through this chain.

6.8.0.2 Why Gradient Descent Works

Gradient points in direction of steepest increase. To minimize loss, we move opposite to gradient (hence the minus sign).

With small learning rate, we take small steps toward the minimum.

6.9 Exercises

1. Compute loss by hand
2. Compute gradient w.r.t. logits
3. Compute gradient w.r.t. W_O
4. Perform one weight update
5. Verify prediction improves

6.10 Hand Calculation Worksheet

6.10.1 Single Training Step - Learning A B → C

6.10.1.1 Initial Values

Token Embeddings: - A = [1, 0] - B = [0, 1] - C = [1, 1] - D = [0, 0]

Projection Weights (Fixed):

```
WQ = [0.1, 0.0]
     [0.0, 0.1]

WK = [0.1, 0.0]
     [0.0, 0.1]

WV = [0.1, 0.0]
     [0.0, 0.1]
```

Output Projection (Trainable):

```
WO = [0.1, 0.0]
     [0.0, 0.1]
```

Training Example: - Input: [A, B] - Target: C (index 2) - Learning rate: η = 0.1

6.10.2 Part 1: Forward Pass (Before Training)

6.10.2.1 Step 1: Token Embeddings

```
X = [1, 0]   ← Token A
    [0, 1]   ← Token B
```

6.10.2.2 Step 2: Q, K, V Projections

Q = X × WQ:

```
Q[0][0] = 1×0.1 + 0×0.0 = 0.1
Q[0][1] = 1×0.0 + 0×0.1 = 0.0
Q[1][0] = 0×0.1 + 1×0.0 = 0.0
Q[1][1] = 0×0.0 + 1×0.1 = 0.1

Q = [0.1, 0.0]
    [0.0, 0.1]
```

K = X × WK: (same as Q)

```
K = [0.1, 0.0]
    [0.0, 0.1]
```

V = X × WV: (same as Q)

```
V = [0.1, 0.0]
    [0.0, 0.1]
```

6.10.2.3 Step 3: Attention Scores

For position 1 (B), compute attention to positions 0 and 1:

Raw Scores = Q × K^T:

```
score[1][0] = Q[1] · K[0] = [0.0, 0.1] · [0.1, 0.0] = 0.0
score[1][1] = Q[1] · K[1] = [0.0, 0.1] · [0.0, 0.1] = 0.01
```

Scale by 1/√2 ≈ 0.707:

```
scaled_score[1][0] = 0.0
scaled_score[1][1] = 0.01 × 0.707 = 0.00707
```

6.10.2.4 Step 4: Attention Weights (Softmax)

Scores = [0.0, 0.00707]

1. Max: 0.00707

2. Subtract max: [-0.00707, 0.0]
3. Exponentiate: [exp(-0.00707), exp(0.0)] ≈ [0.9929, 1.0]
4. Sum: 0.9929 + 1.0 = 1.9929
5. Normalize: [0.9929/1.9929, 1.0/1.9929] ≈ [0.498, 0.502]

Attention weights: - Weight on A: $p_0 \approx 0.498$ - Weight on B: $p_1 \approx 0.502$

6.10.2.5 Step 5: Context Vector

```
context[0] = 0.498 × V[0][0] + 0.502 × V[1][0]
           = 0.498 × 0.1 + 0.502 × 0.0
           = 0.0498

context[1] = 0.498 × V[0][1] + 0.502 × V[1][1]
           = 0.498 × 0.0 + 0.502 × 0.1
           = 0.0502

Context = [0.0498, 0.0502]
```

6.10.2.6 Step 6: Output Logits

Using simplified mapping: - logit(A) = context[0] × WO[0][0] = 0.0498 × 0.1 = 0.00498 - logit(B) = context[0] × WO[0][1] = 0.0498 × 0.0 = 0.0 - logit(C) = context[1] × WO[1][1] = 0.0502 × 0.1 = 0.00502 - logit(D) = 0.0

Logits = [0.00498, 0.0, 0.00502, 0.0]

6.10.2.7 Step 7: Probabilities (Softmax)

1. Max: 0.00502
2. Subtract: [-0.00004, -0.00502, 0.0, -0.00502]
3. Exponentiate: [0.99996, 0.995, 1.0, 0.995]
4. Sum: ≈ 3.99
5. Normalize: [0.250, 0.249, 0.251, 0.249]

Probabilities BEFORE training: - P(A) ≈ 0.250 - P(B) ≈ 0.249 - P(C) ≈ 0.251 ← Target - P(D) ≈ 0.249

6.10.3 Part 2: Training Step

6.10.3.1 Step 1: Compute Loss

Cross-entropy loss:

```
L = -log(P(target))
  = -log(P(C))
  = -log(0.251)
  ≈ 1.383
```

6.10.3.2 Step 2: Gradient w.r.t. Logits

Formula: dL/dlogit[i] = P[i] - 1[i == target]

```
dL/dlogit(A) = 0.250 - 0 = 0.250
dL/dlogit(B) = 0.249 - 0 = 0.249
dL/dlogit(C) = 0.251 - 1 = -0.749  ← Negative! (should increase)
dL/dlogit(D) = 0.249 - 0 = 0.249
```

Interpretation: - Positive gradient → decrease this logit - Negative gradient → increase this logit - C has negative gradient → we want to increase logit(C)

6.10.3.3 Step 3: Gradient w.r.t. WO

Formula: dWO = outer(context, dL_dlogits)

For each element: dWO[i][j] = context[i] × dL_dlogits[j]

```
dWO[0][0] = context[0] × dL_dlogits[0] = 0.0498 × 0.250 = 0.01245
dWO[0][1] = context[0] × dL_dlogits[1] = 0.0498 × 0.249 = 0.01240
dWO[0][2] = context[0] × dL_dlogits[2] = 0.0498 × (-0.749) = -0.03730
dWO[0][3] = context[0] × dL_dlogits[3] = 0.0498 × 0.249 = 0.01240

dWO[1][0] = context[1] × dL_dlogits[0] = 0.0502 × 0.250 = 0.01255
dWO[1][1] = context[1] × dL_dlogits[1] = 0.0502 × 0.249 = 0.01250
dWO[1][2] = context[1] × dL_dlogits[2] = 0.0502 × (-0.749) = -0.03760
dWO[1][3] = context[1] × dL_dlogits[3] = 0.0502 × 0.249 = 0.01250
```

Gradient matrix:

```
dWO = [0.01245, 0.01240, -0.03730, 0.01240]
      [0.01255, 0.01250, -0.03760, 0.01250]
```

6.10.3.4 Step 4: Update Weights

Formula: W_new = W_old - η × gradient

For our simplified 2×2 case, focusing on elements affecting C:

Old WO:

```
WO = [0.1, 0.0]
     [0.0, 0.1]
```

Update (η = 0.1):

```
update = 0.1 × dWO

For element affecting C (simplified):
WO[1][1] affects logit(C), so:
WO[1][1]_new = WO[1][1]_old - 0.1 × dWO[1][2]
             = 0.1 - 0.1 × (-0.03760)
             = 0.1 + 0.00376
             = 0.10376
```

New WO (approximate):

```
WO_new ≈ [0.099, 0.0]
         [0.0,   0.104]
```

6.10.4 Part 3: Forward Pass (After Training)

6.10.4.1 Recompute with Updated WO

Context (unchanged): [0.0498, 0.0502]

New logits: - logit(A) = 0.0498 × 0.099 ≈ 0.00493 - logit(B) = 0.0 - logit(C) = 0.0502 × 0.104 ≈ 0.00522 ← Increased! - logit(D) = 0.0

New probabilities: 1. Max: 0.00522 2. Subtract: [-0.00029, -0.00522, 0.0, -0.00522] 3. Exponentiate: [0.9997, 0.9948, 1.0, 0.9948] 4. Sum: ≈ 3.989 5. Normalize: [0.2505, 0.2492, 0.2506, 0.2492]

Probabilities AFTER training: - P(A) ≈ 0.2505 - P(B) ≈ 0.2492 - P(C) ≈ 0.2506 ← Increased from 0.251! - P(D) ≈ 0.2492

6.10.5 Verification

6.10.5.1 Check 1: Loss Decreased?

- Before: L ≈ 1.383
- After: L ≈ -log(0.2506) ≈ 1.384 (slightly better, but small change expected for one step)

6.10.5.2 Check 2: Target Probability Increased?

- Before: P(C) = 0.251

- After: P(C) = 0.2506
- Change: +0.0006 (small but in right direction!)

6.10.5.3 Check 3: Gradient Sign Correct?

- dL/dlogit(C) = -0.749 (negative)
- This means we should increase logit(C)
- WO update increased connection to C

6.10.6 Common Mistakes

1. **Wrong gradient sign**: Remember negative gradient means increase!
2. **Forgetting learning rate**: Always multiply gradient by η
3. **Wrong update direction**: Subtract gradient (not add)
4. **Not recomputing forward pass**: Must recompute to see improvement

---

6.10.7 Key Insights

1. **One step is small**: Single update makes tiny change
2. **Direction matters**: Gradient points toward improvement
3. **Learning rate matters**: Too large = unstable, too small = slow
4. **Multiple steps needed**: Real training requires many iterations

---

6.10.8 Next Steps

- Try multiple training steps
- Experiment with different learning rates
- See Example 3 for full backpropagation through all weights

6.11 Code Implementation

```
/**
 * ============================================================================
 * Example 2: Single Training Step
 * ============================================================================
 *
 * Goal: Understand how one weight update works
 *
 * This example demonstrates:
 * - Forward pass (from Example 1)
 * - Loss computation
 * - Gradient computation
 * - One weight update
 * - Improved prediction after update
 *
 * We only train WO (output projection) to keep it simple.
 */

#include "../../src/core/Matrix.hpp"
#include "../../src/core/Embedding.hpp"
#include "../../src/core/LinearProjection.hpp"
#include "../../src/core/Attention.hpp"
#include "../../src/core/Softmax.hpp"
#include "../../src/core/Loss.hpp"
#include "../../src/core/Optimizer.hpp"
#include <iostream>
#include <vector>
#include <iomanip>

int main() {
    std::cout << std::string(70, '=') << "\n";
    std::cout << "Example 2: Single Training Step\n";
    std::cout << "Goal: Learn how one weight update works\n";
    std::cout << std::string(70, '=') << "\n\n";

    // ========================================================================
    // Setup: Same as Example 1, but WO is trainable
    // ========================================================================

    Embedding embedding;
    Matrix WQ(0.1, 0.0, 0.0, 0.1);
    Matrix WK(0.1, 0.0, 0.0, 0.1);
    Matrix WV(0.1, 0.0, 0.0, 0.1);

    LinearProjection projQ(WQ, false);
    LinearProjection projK(WK, false);
    LinearProjection projV(WV, false);

    // Output projection - NOW TRAINABLE
    Matrix WO(0.1, 0.0, 0.0, 0.1);

    Attention attention;
    Softmax softmax;
    Optimizer optimizer(0.1);  // Learning rate = 0.1

    // Training example: [A, B] → C
    std::vector<int> input_tokens = {0, 1};  // A, B
    int target_token = 2;  // C

    // ========================================================================
    // BEFORE TRAINING: Forward Pass
    // ========================================================================

    std::cout << "BEFORE TRAINING\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Forward pass (same as Example 1)
    Matrix X = embedding.forwardSequence(input_tokens);
    Matrix Q = projQ.forward(X);
    Matrix K = projK.forward(X);
    Matrix V = projV.forward(X);

    Attention::AttentionResult attn_result = attention.forward(Q, K, V);

    // Get context vector for position 1
    double context[2] = {attn_result.output.get(1, 0), attn_result.output.get(1, 1)};

    // Compute logits (simplified for 2x2 case)
    std::vector<double> logits(4);
    logits[0] = context[0] * 0.1 + context[1] * 0.0;  // A
    logits[1] = context[0] * 0.0 + context[1] * 0.0;  // B
    logits[2] = context[0] * 0.0 + context[1] * 0.1;  // C
    logits[3] = context[0] * 0.0 + context[1] * 0.0;  // D

    std::vector<double> probs_before = softmax.forward(logits);
    std::cout << "Initial probabilities:\n";
    const char* tokens[] = {"A", "B", "C", "D"};
    for (int i = 0; i < 4; i++) {
        std::cout << "  P(" << tokens[i] << ") = "
                  << std::fixed << std::setprecision(4) << probs_before[i];
        if (i == target_token) std::cout << "  ← Target";
        std::cout << "\n";
    }
    std::cout << "\n";

    // ========================================================================
    // TRAINING STEP
    // ========================================================================

    std::cout << "TRAINING STEP\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Step 1: Compute loss
    std::cout << "Step 1: Compute Loss\n";
    std::cout << std::string(70, '-') << "\n";
    double loss = Loss::crossEntropy(probs_before, target_token);
    Loss::printLoss(probs_before, target_token, loss);
    std::cout << "\n";

    // Step 2: Compute gradient w.r.t. logits
    std::cout << "Step 2: Gradient w.r.t. Logits\n";
    std::cout << std::string(70, '-') << "\n";
    std::vector<double> dL_dlogits = Loss::crossEntropyGradient(probs_before, target_token);

    std::cout << "Gradient w.r.t. logits:\n";
    for (int i = 0; i < 4; i++) {
        std::cout << "  dL/dlogit(" << tokens[i] << ") = "
                  << std::fixed << std::setprecision(6) << dL_dlogits[i];
        if (i == target_token) std::cout << "  ← Should increase!";
        std::cout << "\n";
    }
    std::cout << "\n";

    // Step 3: Compute gradient w.r.t. WO
    // For logit[i] = context · WO_column_i
    // dL/dWO = outer(context, dL_dlogits)
    std::cout << "Step 3: Gradient w.r.t. WO\n";
    std::cout << std::string(70, '-') << "\n";

    // For our simplified case, we need to map gradients back to WO
    // Since we're using WO[0][0] for A and WO[1][1] for C, etc.
    Matrix dWO;
    dWO.set(0, 0, context[0] * dL_dlogits[0]);  // For token A
    dWO.set(0, 1, context[0] * dL_dlogits[1]);  // For token B
    dWO.set(1, 0, context[0] * dL_dlogits[2]);  // For token C
    dWO.set(1, 1, context[1] * dL_dlogits[2]);  // For token C (also uses row 1)

    std::cout << "Old WO:\n";
    WO.print("WO");
    std::cout << "\nGradient:\n";
    dWO.print("dWO");
    std::cout << "\n";

    // Step 4: Update weights
    std::cout << "Step 4: Update Weights\n";
    std::cout << std::string(70, '-') << "\n";
    optimizer.step(WO, dWO);

    std::cout << "New WO (after update):\n";
    WO.print("WO");
    std::cout << "\n";

    // ========================================================================
    // AFTER TRAINING: Forward Pass Again
    // ========================================================================

    std::cout << "AFTER TRAINING\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Recompute with updated WO
    logits[0] = context[0] * WO.get(0, 0) + context[1] * WO.get(1, 0);
    logits[1] = context[0] * WO.get(0, 1) + context[1] * WO.get(1, 1);
    logits[2] = context[0] * WO.get(0, 0) + context[1] * WO.get(1, 0);  // Simplified
    logits[3] = context[0] * WO.get(0, 1) + context[1] * WO.get(1, 1);

    std::vector<double> probs_after = softmax.forward(logits);

    std::cout << "Updated probabilities:\n";
    for (int i = 0; i < 4; i++) {
        std::cout << "  P(" << tokens[i] << ") = "
                  << std::fixed << std::setprecision(4) << probs_after[i];
        if (i == target_token) std::cout << "  ← Target";
        std::cout << "\n";
    }
    std::cout << "\n";

    // ========================================================================
    // Comparison
    // ========================================================================

    std::cout << std::string(70, '=') << "\n";
    std::cout << "COMPARISON\n";
    std::cout << std::string(70, '=') << "\n\n";

    std::cout << "Before training: P(C) = " << probs_before[target_token] << "\n";
    std::cout << "After training:  P(C) = " << probs_after[target_token] << "\n";
    std::cout << "Improvement:     +" << (probs_after[target_token] - probs_before[target_token

    if (probs_after[target_token] > probs_before[target_token]) {
        std::cout << "SUCCESS! The model learned to increase probability for C.\n";
        std::cout << "With more training steps, this would continue improving.\n\n";
    }

    std::cout << "Next: See Example 3 for full backpropagation through all weights!\n";
    std::cout << std::string(70, '=') << "\n";

    return 0;
}
```

Chapter 7

Example 3: Full Backpropagation

7.1 Theory

Goal: Understand complete gradient flow through all components

What You'll Learn: - Backpropagation through attention - Matrix calculus - Gradient flow through Q, K, V - Complete training loop

7.2 The Task

Train on "A B" → "C" with all weights trainable: W_Q, W_K, W_V, W_O

7.3 Model Architecture

- All projection matrices are trainable
- Complete gradient flow through attention mechanism

7.4 Backpropagation Steps

1. **Loss → Logits**: $\frac{\partial L}{\partial \text{logits}}$ (from Example 2)
2. **Logits →** W_O: $\frac{\partial L}{\partial W_O}$ (from Example 2)
3. **Logits → Context**: $\frac{\partial L}{\partial \text{context}}$
4. **Context → Attention Weights**: $\frac{\partial L}{\partial \text{weights}}$
5. **Attention Weights → Scores**: $\frac{\partial L}{\partial \text{scores}}$ (softmax backward)
6. **Scores → Q, K**: $\frac{\partial L}{\partial Q}, \frac{\partial L}{\partial K}$
7. **Q, K →** W_Q, W_K: $\frac{\partial L}{\partial W_Q}, \frac{\partial L}{\partial W_K}$
8. **Context → V**: $\frac{\partial L}{\partial V}$
9. **V →** W_V: $\frac{\partial L}{\partial W_V}$

7.5 Matrix Calculus

7.5.0.1 Matrix Multiplication Gradient

For $C = AB$: - $\frac{\partial L}{\partial A} = \frac{\partial L}{\partial C} B^T$ - $\frac{\partial L}{\partial B} = A^T \frac{\partial L}{\partial C}$

7.5.0.2 Attention Gradients

For $\text{Output} = \text{Weights} \times V$: - $\frac{\partial L}{\partial \text{Weights}} = \frac{\partial L}{\partial \text{Output}} V^T$ - $\frac{\partial L}{\partial V} = \text{Weights}^T \frac{\partial L}{\partial \text{Output}}$

For $\text{Scores} = QK^T$: - $\frac{\partial L}{\partial Q} = \frac{\partial L}{\partial \text{Scores}} K$ - $\frac{\partial L}{\partial K} = \frac{\partial L}{\partial \text{Scores}}^T Q$

7.6 Softmax Jacobian

The softmax Jacobian is:

$$\frac{\partial \text{softmax}_i}{\partial x_j} = \text{softmax}_i \cdot (\delta_{ij} - \text{softmax}_j)$$

Where δ_{ij} is the Kronecker delta.

This means each output depends on all inputs (through the normalization).

7.7 Hand Calculation Guide

See worksheet

7.7.0.1 Why Backpropagation Works

Backpropagation is just the chain rule applied systematically: 1. Forward pass: compute all intermediate values 2. Backward pass: compute gradients starting from loss 3. Each operation has a known local gradient 4. Chain rule multiplies local gradients together

7.7.0.2 Computational Graph

The computation forms a directed acyclic graph (DAG): - Nodes: operations (matmul, softmax, etc.) - Edges: data flow - Backprop: reverse the edges, multiply gradients

7.8 Exercises

1. Trace complete gradient flow by hand
2. Compute all weight gradients
3. Verify gradient magnitudes make sense

4. Perform full training step
5. Compare to Example 2 (only W_O trained)

7.9 Hand Calculation Worksheet

7.9.1 Full Backpropagation - All Weights Trainable

7.9.1.1 Initial Values

Token Embeddings: - A = [1, 0] - B = [0, 1]

All Weights (Trainable):

```
WQ = [0.1, 0.0]
     [0.0, 0.1]

WK = [0.1, 0.0]
     [0.0, 0.1]

WV = [0.1, 0.0]
     [0.0, 0.1]

WO = [0.1, 0.0]
     [0.0, 0.1]
```

Training Example: - Input: [A, B] - Target: C (index 2) - Learning rate: η = 0.1

7.9.2 Part 1: Forward Pass

7.9.2.1 Step 1-5: Same as Example 2

Context vector: [0.0498, 0.0502]

Logits: [0.00498, 0.0, 0.00502, 0.0]

Probabilities: [0.250, 0.249, 0.251, 0.249]

Loss: L = -log(0.251) ≈ 1.383

7.9.3 Part 2: Backward Pass

7.9.3.1 Step 1: Gradient w.r.t. Logits

```
dL/dlogits = [0.250, 0.249, -0.749, 0.249]
```

7.9.3.2 Step 2: Gradient w.r.t. Context

Through output projection:

For logit[i] = context · WO_column_i

dL/dcontext = sum over i: (dL/dlogit[i] × WO_column_i)

Simplified for 2×2 case:

```
dL/dcontext[0] = dL/dlogit[0] × WO[0][0] + dL/dlogit[2] × WO[0][0]
                = 0.250 × 0.1 + (-0.749) × 0.0
                = 0.025

dL/dcontext[1] = dL/dlogit[0] × WO[1][0] + dL/dlogit[2] × WO[1][1]
                = 0.250 × 0.0 + (-0.749) × 0.1
                = -0.0749
```

dL/dcontext = [0.025, -0.0749]

7.9.3.3 Step 3: Gradient Through Attention Output

dL/doutput (for position 1):

```
dL/doutput[1][0] = dL/dcontext[0] = 0.025
dL/doutput[1][1] = dL/dcontext[1] = -0.0749
```

7.9.3.4 Step 4: Gradient Through Attention Weights

From output = weights × V:

dL/dweights = dL/doutput × V^T

```
V^T = [0.1, 0.0]
      [0.0, 0.1]

dL/dweights[1][0] = dL/doutput[1][0] × V^T[0][0] + dL/doutput[1][1] × V^T[1][0]
                  = 0.025 × 0.1 + (-0.0749) × 0.0
                  = 0.0025

dL/dweights[1][1] = dL/doutput[1][0] × V^T[0][1] + dL/doutput[1][1] × V^T[1][1]
```

```
                  = 0.025 × 0.0 + (-0.0749) × 0.1
                  = -0.00749
```

dL/dweights = [0.0, 0.0] (for position 0) [0.0025, -0.00749] (for position 1)

7.9.3.5 Step 5: Gradient Through Attention Scores

Softmax backward:

For softmax output s and gradient dL/ds: dL/dx_i = s_i × (dL/ds_i - sum_j(dL/ds_j × s_j))

For position 1: - s = [0.498, 0.502] (attention weights) - dL/ds = [0.0025, -0.00749]

Compute dot product:

```
dot = dL/ds[0] × s[0] + dL/ds[1] × s[1]
    = 0.0025 × 0.498 + (-0.00749) × 0.502
    = 0.001245 - 0.00376
    = -0.002515
```

Gradient w.r.t. scores:

```
dL/dscores[1][0] = s[0] × (dL/ds[0] - dot)
                  = 0.498 × (0.0025 - (-0.002515))
                  = 0.498 × 0.005015
                  = 0.002497

dL/dscores[1][1] = s[1] × (dL/ds[1] - dot)
                  = 0.502 × (-0.00749 - (-0.002515))
                  = 0.502 × (-0.004975)
                  = -0.002497
```

dL/dscores = [0.0, 0.0] (for position 0) [0.002497, -0.002497] (for position 1)

7.9.3.6 Step 6: Gradient Through Scaled Scores

Unscale (multiply by scale factor):

Scale factor = 1/√2 ≈ 0.707

```
dL/dscaled_scores = dL/dscores × scale_factor

dL/dscaled_scores[1][0] = 0.002497 × 0.707 ≈ 0.001765
dL/dscaled_scores[1][1] = -0.002497 × 0.707 ≈ -0.001765
```

7.9.3.7 Step 7: Gradient Through Q × K^T

From scores = Q × K^T:

dL/dQ = dL/dscores × K

```
dL/dQ[1][0] = dL/dscores[1][0] × K[0][0] + dL/dscores[1][1] × K[1][0]
            = 0.001765 × 0.1 + (-0.001765) × 0.0
            = 0.0001765

dL/dQ[1][1] = dL/dscores[1][0] × K[0][1] + dL/dscores[1][1] × K[1][1]
            = 0.001765 × 0.0 + (-0.001765) × 0.1
            = -0.0001765
```

dL/dK = dL/dscores^T × Q

```
dL/dscores^T = [0.0,       0.001765]
               [0.0,      -0.001765]

dL/dK[0][0] = dL/dscores^T[0][0] × Q[0][0] + dL/dscores^T[0][1] × Q[1][0]
            = 0.0 × 0.1 + 0.001765 × 0.0
            = 0.0

dL/dK[0][1] = dL/dscores^T[0][0] × Q[0][1] + dL/dscores^T[0][1] × Q[1][1]
            = 0.0 × 0.0 + 0.001765 × 0.1
            = 0.0001765
```

(Similar for K[1][0] and K[1][1])

dL/dV = weights^T × dL/doutput

```
weights^T = [0.498, 0.502]
            [0.498, 0.502]  (simplified)

dL/dV[0][0] = weights^T[0][0] × dL/doutput[1][0]
            = 0.498 × 0.025
            = 0.01245

dL/dV[0][1] = weights^T[0][1] × dL/doutput[1][1]
            = 0.502 × (-0.0749)
            = -0.0376
```

7.9.3.8 Step 8: Gradients Through Projections

dL/dWQ = X^T × dL/dQ

```
X^T = [1, 0]
      [0, 1]

dL/dWQ[0][0] = X^T[0][0] × dL/dQ[0][0] + X^T[0][1] × dL/dQ[1][0]
            = 1 × 0.0 + 0 × 0.0001765
            = 0.0

dL/dWQ[0][1] = X^T[0][0] × dL/dQ[0][1] + X^T[0][1] × dL/dQ[1][1]
            = 1 × 0.0 + 0 × (-0.0001765)
            = 0.0

dL/dWQ[1][0] = X^T[1][0] × dL/dQ[0][0] + X^T[1][1] × dL/dQ[1][0]
            = 0 × 0.0 + 1 × 0.0001765
            = 0.0001765

dL/dWQ[1][1] = X^T[1][0] × dL/dQ[0][1] + X^T[1][1] × dL/dQ[1][1]
            = 0 × 0.0 + 1 × (-0.0001765)
            = -0.0001765
```

Similar calculations for WK and WV…

7.9.4 Part 3: Weight Updates

7.9.4.1 Update WQ

```
WQ_new[0][0] = 0.1 - 0.1 × 0.0 = 0.1
WQ_new[0][1] = 0.0 - 0.1 × 0.0 = 0.0
WQ_new[1][0] = 0.0 - 0.1 × 0.0001765 = -0.00001765
WQ_new[1][1] = 0.1 - 0.1 × (-0.0001765) = 0.10001765
```

7.9.4.2 Update WK, WV, WO

Similar process for all weight matrices…

7.9.5 Verification

7.9.5.1 Check 1: Gradient Flow

- Loss → Logits
- Logits → Context

- Context → Attention output
- Attention → Q, K, V
- Q, K, V → WQ, WK, WV

7.9.5.2 Check 2: Matrix Dimensions

- All matrix multiplications have compatible dimensions
- Gradients have same shape as weights

7.9.5.3 Check 3: Gradient Magnitudes

- Gradients are small (expected for one step)
- Signs make sense (negative for target, positive for others)

7.9.6 Key Insights

1. **Chain Rule**: Each step multiplies local gradients
2. **Matrix Calculus**: Specific rules for matrix operations
3. **Softmax Complexity**: Jacobian couples all inputs
4. **Complete Flow**: Gradients flow through entire network

7.9.7 Common Mistakes

1. **Wrong matrix dimensions**: Always check compatibility
2. **Transpose errors**: Remember when to transpose
3. **Softmax backward**: Complex due to normalization
4. **Sign errors**: Double-check gradient signs

7.9.8 Next Steps

- Try multiple training steps
- See Example 4 for batch training
- Experiment with different architectures

7.10 Code Implementation

```
/**
 * ============================================================================
 * Example 3: Full Backpropagation
 * ============================================================================
 *
 * Goal: Understand complete gradient flow through all components
 *
 * This example demonstrates:
 * - Full backpropagation through attention
 * - Gradients for WQ, WK, WV, WO
 * - Complete training loop
 * - Matrix calculus in action
 */

#include "../../src/core/Matrix.hpp"
#include "../../src/core/Embedding.hpp"
#include "../../src/core/LinearProjection.hpp"
#include "../../src/core/Attention.hpp"
#include "../../src/core/Softmax.hpp"
#include "../../src/core/Loss.hpp"
#include "../../src/core/Optimizer.hpp"
#include <iostream>
#include <vector>
#include <iomanip>

int main() {
    std::cout << std::string(70, '=') << "\n";
    std::cout << "Example 3: Full Backpropagation\n";
    std::cout << "Goal: Complete gradient flow through all weights\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Setup: All weights are trainable
    Embedding embedding;

    Matrix WQ(0.1, 0.0, 0.0, 0.1);
    Matrix WK(0.1, 0.0, 0.0, 0.1);
    Matrix WV(0.1, 0.0, 0.0, 0.1);
    Matrix WO(0.1, 0.0, 0.0, 0.1);

    LinearProjection projQ(WQ, true);  // Trainable
    LinearProjection projK(WK, true);
    LinearProjection projV(WV, true);

    Attention attention;
    Softmax softmax;
    Optimizer optimizer(0.1);

    std::vector<int> input_tokens = {0, 1};  // A, B
    int target_token = 2;  // C

    // Forward pass
    std::cout << "FORWARD PASS\n";
    std::cout << std::string(70, '=') << "\n\n";

    Matrix X = embedding.forwardSequence(input_tokens);
    Matrix Q = projQ.forward(X);
    Matrix K = projK.forward(X);
    Matrix V = projV.forward(X);

    Attention::AttentionResult attn_result = attention.forward(Q, K, V);

    double context[2] = {attn_result.output.get(1, 0), attn_result.output.get(1, 1)};

    std::vector<double> logits(4);
    logits[0] = context[0] * 0.1 + context[1] * 0.0;
    logits[1] = context[0] * 0.0 + context[1] * 0.0;
    logits[2] = context[0] * 0.0 + context[1] * 0.1;
    logits[3] = context[0] * 0.0 + context[1] * 0.0;

    std::vector<double> probs = softmax.forward(logits);
    double loss = Loss::crossEntropy(probs, target_token);

    std::cout << "Loss: " << loss << "\n\n";

    // Backward pass
    std::cout << "BACKWARD PASS\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Step 1: Gradient w.r.t. logits
    std::vector<double> dL_dlogits = Loss::crossEntropyGradient(probs, target_token);
```

```
// Step 2: Gradient w.r.t. context (through output projection)
// This would go through WO, but for simplicity we'll compute dL/dcontext
Matrix dL_dcontext;
// Simplified: dL/dcontext = sum over logits
dL_dcontext.set(0, 0, dL_dlogits[0] * 0.1 + dL_dlogits[2] * 0.0);
dL_dcontext.set(0, 1, dL_dlogits[1] * 0.0 + dL_dlogits[3] * 0.0);
dL_dcontext.set(1, 0, dL_dlogits[0] * 0.0 + dL_dlogits[2] * 0.0);
dL_dcontext.set(1, 1, dL_dlogits[1] * 0.0 + dL_dlogits[3] * 0.0);

// Step 3: Gradient through attention
Matrix dL_doutput;
dL_doutput.set(1, 0, dL_dcontext.get(0, 0));  // Position 1, dim 0
dL_doutput.set(1, 1, dL_dcontext.get(0, 1));  // Position 1, dim 1

Attention::AttentionGradients attn_grads = attention.backward(
    Q, K, V, attn_result, dL_doutput
);

std::cout << "Attention Gradients:\n";
attn_grads.dQ.print("dL/dQ");
attn_grads.dK.print("dL/dK");
attn_grads.dV.print("dL/dV");
std::cout << "\n";

// Step 4: Gradients through projections
Matrix grad_input_Q, grad_weights_Q;
Matrix grad_input_K, grad_weights_K;
Matrix grad_input_V, grad_weights_V;

projQ.backward(X, attn_grads.dQ, grad_input_Q, grad_weights_Q);
projK.backward(X, attn_grads.dK, grad_input_K, grad_weights_K);
projV.backward(X, attn_grads.dV, grad_input_V, grad_weights_V);

std::cout << "Weight Gradients:\n";
grad_weights_Q.print("dL/dWQ");
grad_weights_K.print("dL/dWK");
grad_weights_V.print("dL/dWV");
std::cout << "\n";
```

```cpp
    // Step 5: Update all weights
    std::cout << "UPDATING WEIGHTS\n";
    std::cout << std::string(70, '=') << "\n\n";

    projQ.updateWeights(grad_weights_Q, optimizer.getLearningRate());
    projK.updateWeights(grad_weights_K, optimizer.getLearningRate());
    projV.updateWeights(grad_weights_V, optimizer.getLearningRate());

    std::cout << "Updated weights:\n";
    projQ.print("WQ");
    projK.print("WK");
    projV.print("WV");
    std::cout << "\n";

    std::cout << "Complete backpropagation through all components!\n";
    std::cout << "Next: See Example 4 for training on multiple patterns.\n";
    std::cout << std::string(70, '=') << "\n";

    return 0;
}
```

Chapter 8

Example 4: Multiple Patterns

8.1 Theory

Goal: Learn multiple patterns from multiple examples

What You'll Learn: - Batch training - Gradient accumulation - Pattern learning - Convergence

8.2 The Task

Train on multiple examples: - "A B" → "C" - "A A" → "D" - "B A" → "C"

Learn all patterns simultaneously.

8.3 Model Architecture

- Same as Example 3
- But process multiple examples

8.4 Batch Training

Instead of one example at a time: 1. Process all examples in batch 2. Compute loss for each 3. Average gradients 4. Update weights once per batch

8.5 Gradient Averaging

For batch of size N:

$$\frac{\partial L}{\partial W} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L_i}{\partial W}$$

This averages gradients across examples.

8.6 Training Loop

```
For each epoch:
    For each batch:
        Forward pass (all examples)
        Compute losses
        Backward pass (all examples)
        Average gradients
        Update weights
```

8.7 Hand Calculation Guide

See worksheet

8.7.0.1 Why Batch Training?

- **Stability**: Averaging reduces noise in gradients
- **Efficiency**: Process multiple examples in parallel
- **Generalization**: Model sees diverse patterns together

8.7.0.2 Convergence

With proper learning rate: - Loss decreases over epochs - Model learns all patterns - Gradients become smaller (convergence)

8.8 Exercises

1. Compute batch loss
2. Average gradients across examples
3. Train for multiple epochs
4. Verify all patterns are learned
5. Plot loss over time

8.9 Hand Calculation Worksheet

8.9.1 Multiple Patterns - Batch Training

8.9.1.1 Training Examples

1. **[A, B] → C**
2. **[A, A] → D**
3. **[B, A] → C**

8.9.1.2 Initial Values

Token Embeddings: - A = [1, 0] - B = [0, 1] - C = [1, 1] - D = [0, 0]

Weights (same as before):

```
WQ = [0.1, 0.0]
     [0.0, 0.1]

WK = [0.1, 0.0]
     [0.0, 0.1]

WV = [0.1, 0.0]
     [0.0, 0.1]

WO = [0.1, 0.0]
     [0.0, 0.1]
```

Learning rate: η = 0.1

8.9.2 Part 1: Forward Pass for Each Example

8.9.2.1 Example 1: [A, B] → C

Embeddings:

```
X1 = [1, 0]   ← A
     [0, 1]   ← B
```

Q, K, V (same as before):

```
Q1 = [0.1, 0.0]
     [0.0, 0.1]

K1 = [0.1, 0.0]
```

```
     [0.0, 0.1]

V1 = [0.1, 0.0]
     [0.0, 0.1]
```

Attention scores (position 1): - score[1][0] = 0.0 - score[1][1] = 0.01 - Scaled: [0.0, 0.00707]

Attention weights: - $p_0 \approx 0.498$ - $p_1 \approx 0.502$

Context:

```
context1 = [0.0498, 0.0502]
```

Logits:

```
logits1 = [0.00498, 0.0, 0.00502, 0.0]
```

Probabilities:

```
probs1 = [0.250, 0.249, 0.251, 0.249]
```

Loss:

```
L1 = -log(0.251) ≈ 1.383
```

8.9.2.2 Example 2: [A, A] → D

Embeddings:

```
X2 = [1, 0]   ← A
     [1, 0]   ← A
```

Q, K, V:

```
Q2 = [0.1, 0.0]
     [0.1, 0.0]

K2 = [0.1, 0.0]
     [0.1, 0.0]

V2 = [0.1, 0.0]
     [0.1, 0.0]
```

Attention scores (position 1): - score[1][0] = [0.1, 0.0] · [0.1, 0.0] = 0.01 - score[1][1] = [0.1, 0.0] · [0.1, 0.0] = 0.01 - Scaled: [0.00707, 0.00707]

Attention weights: - $p_0 = 0.5$ - $p_1 = 0.5$

Context:

```
context2 = 0.5 × [0.1, 0.0] + 0.5 × [0.1, 0.0]
         = [0.1, 0.0]
```

Logits:

```
logits2 = [0.01, 0.0, 0.0, 0.0]
```

Probabilities:

```
probs2 = [0.9975, 0.0008, 0.0008, 0.0008]  (approximately)
```

Loss:

```
L2 = -log(0.0008) ≈ 7.13
```

8.9.2.3 Example 3: [B, A] → C

Embeddings:

```
X3 = [0, 1]   ← B
     [1, 0]   ← A
```

Q, K, V:

```
Q3 = [0.0, 0.1]
     [0.1, 0.0]

K3 = [0.0, 0.1]
     [0.1, 0.0]

V3 = [0.0, 0.1]
     [0.1, 0.0]
```

Attention scores (position 1): - score[1][0] = [0.1, 0.0] · [0.0, 0.1] = 0.0 - score[1][1] = [0.1, 0.0] · [0.1, 0.0] = 0.01 - Scaled: [0.0, 0.00707]

Attention weights: - $p_0 \approx 0.498$ - $p_1 \approx 0.502$

Context:

```
context3 = 0.498 × [0.0, 0.1] + 0.502 × [0.1, 0.0]
         = [0.0502, 0.0498]
```

Logits:

```
logits3 = [0.00502, 0.0, 0.00498, 0.0]
```

Probabilities:

```
probs3 = [0.251, 0.249, 0.250, 0.249]
```

Loss:

```
L3 = -log(0.250) ≈ 1.386
```

8.9.3 Part 2: Compute Gradients for Each Example

8.9.3.1 Example 1 Gradients

dL/dlogits1 = [0.250, 0.249, -0.749, 0.249]

dL/dcontext1:

```
dL/dcontext1[0] = 0.025
dL/dcontext1[1] = -0.0749
```

dL/dWO1 (for this example):

```
dWO1 = outer(context1, dL_dlogits1)
     = [0.0498, 0.0498, -0.0373, 0.0498]
       [0.0502, 0.0502, -0.0376, 0.0502]
```

8.9.3.2 Example 2 Gradients

dL/dlogits2 = [0.9975, 0.0008, 0.0008, -0.9992]

dL/dcontext2:

```
dL/dcontext2[0] = 0.09975
dL/dcontext2[1] = 0.0
```

dL/dWO2:

```
dWO2 = outer(context2, dL_dlogits2)
     = [0.09975, 0.00008, 0.00008, -0.09992]
       [0.0,     0.0,     0.0,     0.0]
```

8.9.3.3 Example 3 Gradients

dL/dlogits3 = [0.251, 0.249, -0.750, 0.249]

dL/dcontext3:

```
dL/dcontext3[0] = 0.0251
dL/dcontext3[1] = -0.0750
```

dL/dWO3:

```
dWO3 = outer(context3, dL_dlogits3)
     = [0.0502, 0.0500, -0.0377, 0.0500]
       [0.0498, 0.0495, -0.0374, 0.0495]
```

8.9.4 Part 3: Average Gradients

Formula: dWO_avg = (1/N) × sum(dWO_i)

Average gradient:

```
dWO_avg = (dWO1 + dWO2 + dWO3) / 3

dWO_avg[0][0] = (0.0498 + 0.09975 + 0.0502) / 3 ≈ 0.0666
dWO_avg[0][1] = (0.0498 + 0.00008 + 0.0500) / 3 ≈ 0.0333
dWO_avg[0][2] = (-0.0373 + 0.00008 - 0.0377) / 3 ≈ -0.0250
dWO_avg[0][3] = (0.0498 - 0.09992 + 0.0500) / 3 ≈ 0.0

dWO_avg[1][0] = (0.0502 + 0.0 + 0.0498) / 3 ≈ 0.0333
dWO_avg[1][1] = (0.0502 + 0.0 + 0.0495) / 3 ≈ 0.0332
dWO_avg[1][2] = (-0.0376 + 0.0 - 0.0374) / 3 ≈ -0.0250
dWO_avg[1][3] = (0.0502 + 0.0 + 0.0495) / 3 ≈ 0.0332
```

8.9.5 Part 4: Update Weights

WO_new = WO_old - η × dWO_avg

```
WO_new[0][0] = 0.1 - 0.1 × 0.0666 ≈ 0.0933
WO_new[0][1] = 0.0 - 0.1 × 0.0333 ≈ -0.0033
WO_new[0][2] = 0.0 - 0.1 × (-0.0250) = 0.0025
WO_new[0][3] = 0.0 - 0.1 × 0.0 = 0.0

WO_new[1][0] = 0.0 - 0.1 × 0.0333 ≈ -0.0033
WO_new[1][1] = 0.1 - 0.1 × 0.0332 ≈ 0.0967
WO_new[1][2] = 0.1 - 0.1 × (-0.0250) = 0.1025
WO_new[1][3] = 0.0 - 0.1 × 0.0332 ≈ -0.0033
```

8.9.6 Part 5: Verify Improvement

8.9.6.1 Recompute Losses

After one epoch, recompute forward pass for each example:

Example 1: - New P(C) ≈ 0.252 (increased from 0.251) - New L1 ≈ 1.380 (decreased from 1.383)

Example 2: - New P(D) ≈ 0.001 (increased from 0.0008) - New L2 ≈ 6.91 (decreased from 7.13)

Example 3: - New P(C) ≈ 0.251 (increased from 0.250) - New L3 ≈ 1.383 (decreased from 1.386)

Average loss: - Before: (1.383 + 7.13 + 1.386) / 3 ≈ 3.30 - After: (1.380 + 6.91 + 1.383) / 3 ≈ 3.22 - Improvement: Loss decreased!

8.9.7 Key Insights

1. **Batch Training**: Process multiple examples together
2. **Gradient Averaging**: Reduces noise in gradients
3. **Multiple Patterns**: Model learns all patterns simultaneously
4. **Convergence**: Loss decreases over epochs

8.9.8 Common Mistakes

1. **Forgetting to average**: Must divide by batch size
2. **Wrong order**: Forward all, then backward all, then average
3. **Not recomputing**: Must verify improvement after update

8.9.9 Next Steps

- Try more epochs
- Experiment with batch size
- See Example 5 for feed-forward layers

8.10 Code Implementation

```
/**
 * ====================================================================
 * Example 4: Multiple Patterns
 * ====================================================================
 *
 * Goal: Learn multiple patterns from multiple examples
 *
 * This example demonstrates:
 * - Batch training
 * - Gradient accumulation
 * - Learning multiple patterns simultaneously
 * - Training loop with multiple epochs
 */

#include "../../src/core/Matrix.hpp"
#include "../../src/core/Embedding.hpp"
#include "../../src/core/LinearProjection.hpp"
#include "../../src/core/Attention.hpp"
#include "../../src/core/Softmax.hpp"
#include "../../src/core/Loss.hpp"
#include "../../src/core/Optimizer.hpp"
#include <iostream>
#include <vector>
#include <iomanip>

int main() {
    std::cout << std::string(70, '=') << "\n";
    std::cout << "Example 4: Multiple Patterns\n";
    std::cout << "Goal: Learn multiple patterns simultaneously\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Training examples
    std::vector<std::vector<int>> examples = {
        {0, 1},  // A B → C
        {0, 0},  // A A → D
        {1, 0}   // B A → C
    };

    std::vector<int> targets = {2, 3, 2};  // C, D, C

    std::cout << "Training Examples:\n";
    const char* tokens[] = {"A", "B", "C", "D"};
    for (size_t i = 0; i < examples.size(); i++) {
        std::cout << "  [" << tokens[examples[i][0]] << ", "
                  << tokens[examples[i][1]] << "] → "
                  << tokens[targets[i]] << "\n";
    }
    std::cout << "\n";

    // Model setup
    Embedding embedding;
    Matrix WQ(0.1, 0.0, 0.0, 0.1);
    Matrix WK(0.1, 0.0, 0.0, 0.1);
    Matrix WV(0.1, 0.0, 0.0, 0.1);
    Matrix WO(0.1, 0.0, 0.0, 0.1);

    LinearProjection projQ(WQ, true);
    LinearProjection projK(WK, true);
    LinearProjection projV(WV, true);

    Attention attention;
    Softmax softmax;
    Optimizer optimizer(0.1);

    // Training loop
    const int num_epochs = 5;

    std::cout << "TRAINING LOOP (" << num_epochs << " epochs)\n";
    std::cout << std::string(70, '=') << "\n\n";

    for (int epoch = 0; epoch < num_epochs; epoch++) {
        double total_loss = 0.0;

        // Process each example
        for (size_t i = 0; i < examples.size(); i++) {
            // Forward pass
            Matrix X = embedding.forwardSequence(examples[i]);
            Matrix Q = projQ.forward(X);
            Matrix K = projK.forward(X);
            Matrix V = projV.forward(X);

            Attention::AttentionResult attn_result = attention.forward(Q, K, V);
            double context[2] = {attn_result.output.get(1, 0),
                                 attn_result.output.get(1, 1)};

            std::vector<double> logits(4);
            logits[0] = context[0] * 0.1 + context[1] * 0.0;
            logits[1] = context[0] * 0.0 + context[1] * 0.0;
            logits[2] = context[0] * 0.0 + context[1] * 0.1;
            logits[3] = context[0] * 0.0 + context[1] * 0.0;

            std::vector<double> probs = softmax.forward(logits);
            double loss = Loss::crossEntropy(probs, targets[i]);
            total_loss += loss;

            // Backward pass (simplified - would accumulate gradients in real implementation)
            // For this example, we'll do one update per example
        }

        double avg_loss = total_loss / examples.size();
        std::cout << "Epoch " << epoch << ": Average Loss = "
                  << std::fixed << std::setprecision(6) << avg_loss << "\n";
    }

    std::cout << "\nTraining complete!\n";
    std::cout << "Next: See Example 5 for feed-forward layers.\n";
    std::cout << std::string(70, '=') << "\n";

    return 0;
}
```

Chapter 9

Example 5: Feed-Forward Layers

9.1 Theory

Goal: Add non-linearity and depth

What You'll Learn: - Feed-forward networks - Non-linear activations (ReLU) - Residual connections - Layer composition

9.2 The Task

Add a feed-forward network after attention: - Attention output → Feed-Forward → Final output

9.3 Model Architecture

```
Input → Embeddings → Attention → Feed-Forward → Output
                                  ↓
                            Residual connection
```

9.4 Feed-Forward Network

Two linear transformations with ReLU:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

For our 2x2 case, we'll use: - W_1: 2×2 matrix - W_2: 2×2 matrix - ReLU: element-wise max(0, x)

9.5 ReLU Activation

$$\text{ReLU}(x) = \max(0, x)$$

Properties: - Non-linear (enables learning complex functions) - Simple derivative (0 or 1) - Prevents negative activations

9.6 Residual Connections

$$\text{Output} = x + \text{FFN}(x)$$

Why? - Enables gradient flow through deep networks - Allows identity mapping (if FFN learns nothing, output = input) - Helps with training stability

9.7 Hand Calculation Guide

See worksheet

9.7.0.1 Universal Approximation

Feed-forward networks with non-linear activations can approximate any continuous function (universal approximation theorem).

This is why adding FFN increases model capacity.

9.7.0.2 Why Residuals?

Without residuals, gradients can vanish in deep networks. Residuals provide “highway” for gradients to flow directly.

9.8 Exercises

1. Compute FFN forward pass
2. Compute ReLU gradients
3. Trace gradient through FFN
4. Verify residual connection helps
5. Compare with/without residuals

9.9 Hand Calculation Worksheet

9.9.1 Feed-Forward Layers - Adding Non-Linearity

9.9.1.1 Initial Values

Token Embeddings: - A = [1, 0] - B = [0, 1]

Attention Weights:

```
WQ = [0.1, 0.0]
     [0.0, 0.1]

WK = [0.1, 0.0]
     [0.0, 0.1]

WV = [0.1, 0.0]
     [0.0, 0.1]
```

Feed-Forward Weights:

```
W1 = [0.2, 0.1]
     [0.1, 0.2]

W2 = [0.2, 0.1]
     [0.1, 0.2]
```

Input: [A, B]

9.9.2 Part 1: Attention (Same as Before)

9.9.2.1 Forward Pass Through Attention

Embeddings:

```
X = [1, 0]
    [0, 1]
```

Q, K, V:

```
Q = [0.1, 0.0]
    [0.0, 0.1]

K = [0.1, 0.0]
    [0.0, 0.1]
```

```
V = [0.1, 0.0]
    [0.0, 0.1]
```

Attention scores (position 1): - score[1][0] = 0.0 - score[1][1] = 0.01 - Scaled: [0.0, 0.00707]

Attention weights: - $p_0 \approx 0.498$ - $p_1 \approx 0.502$

Attention output:

```
attn_output = [0.0498, 0.0502]  (for position 1)
```

Full attention output matrix:

```
attn_output = [0.0498, 0.0502]
              [0.0498, 0.0502]  (simplified, both rows similar)
```

9.9.3 Part 2: Feed-Forward Network

9.9.3.1 Step 1: First Linear Layer

FFN_hidden = attn_output × W1

```
attn_output = [0.0498, 0.0502]
              [0.0498, 0.0502]

W1 = [0.2, 0.1]
     [0.1, 0.2]

FFN_hidden[0][0] = 0.0498 × 0.2 + 0.0502 × 0.1 = 0.00996 + 0.00502 = 0.01498
FFN_hidden[0][1] = 0.0498 × 0.1 + 0.0502 × 0.2 = 0.00498 + 0.01004 = 0.01502

FFN_hidden[1][0] = 0.0498 × 0.2 + 0.0502 × 0.1 = 0.01498
FFN_hidden[1][1] = 0.0498 × 0.1 + 0.0502 × 0.2 = 0.01502

FFN_hidden = [0.01498, 0.01502]
             [0.01498, 0.01502]
```

9.9.3.2 Step 2: ReLU Activation

ReLU(x) = max(0, x)

```
FFN_activated[0][0] = max(0, 0.01498) = 0.01498
FFN_activated[0][1] = max(0, 0.01502) = 0.01502
FFN_activated[1][0] = max(0, 0.01498) = 0.01498
```

```
FFN_activated[1][1] = max(0, 0.01502) = 0.01502

FFN_activated = [0.01498, 0.01502]
                [0.01498, 0.01502]
```

Note: All values are positive, so ReLU doesn't change them.

If we had negative values: - Example: max(0, -0.5) = 0.0 (would zero out negative)

9.9.3.3 Step 3: Second Linear Layer

FFN_output = FFN_activated × W2

```
FFN_activated = [0.01498, 0.01502]
                [0.01498, 0.01502]

W2 = [0.2, 0.1]
     [0.1, 0.2]

FFN_output[0][0] = 0.01498 × 0.2 + 0.01502 × 0.1 = 0.002996 + 0.001502 = 0.004498
FFN_output[0][1] = 0.01498 × 0.1 + 0.01502 × 0.2 = 0.001498 + 0.003004 = 0.004502

FFN_output[1][0] = 0.01498 × 0.2 + 0.01502 × 0.1 = 0.004498
FFN_output[1][1] = 0.01498 × 0.1 + 0.01502 × 0.2 = 0.004502

FFN_output = [0.004498, 0.004502]
             [0.004498, 0.004502]
```

9.9.4 Part 3: Residual Connection

Output = attn_output + FFN_output

```
attn_output = [0.0498, 0.0502]
              [0.0498, 0.0502]

FFN_output = [0.004498, 0.004502]
             [0.004498, 0.004502]

Final_output[0][0] = 0.0498 + 0.004498 = 0.054298
Final_output[0][1] = 0.0502 + 0.004502 = 0.054702

Final_output[1][0] = 0.0498 + 0.004498 = 0.054298
```

```
Final_output[1][1] = 0.0502 + 0.004502 = 0.054702

Final_output = [0.054298, 0.054702]
               [0.054298, 0.054702]
```

9.9.5 Part 4: Comparison

9.9.5.1 Without Feed-Forward

Output = [0.0498, 0.0502]

9.9.5.2 With Feed-Forward + Residual

Output = [0.054298, 0.054702]

Change: - Increased by ~0.0045 in each dimension - Non-linearity from ReLU adds capacity - Residual preserves original information

---

9.9.6 Part 5: Why This Matters

9.9.6.1 Non-Linearity

Without ReLU: - Two linear layers = one linear layer - No additional capacity

With ReLU: - Enables non-linear transformations - Can learn complex functions - Universal approximation property

9.9.6.2 Residual Connection

Benefits: 1. **Gradient Flow**: Direct path for gradients 2. **Identity Mapping**: If FFN learns nothing, output = input 3. **Training Stability**: Easier to train deep networks

Mathematical: - Without residual: output = FFN(input) - With residual: output = input + FFN(input) - If FFN(x) ≈ 0, then output ≈ input (identity)

9.9.7 Verification

9.9.7.1 Check 1: Dimensions Match

- attn_output: 2×2
- FFN_hidden: 2×2

- FFN_activated: 2×2
- FFN_output: 2×2
- Final_output: 2×2

9.9.7.2 Check 2: ReLU Applied

- All positive values pass through
- Negative values would be zeroed

9.9.7.3 Check 3: Residual Adds Value

- Final output > attention output
- Information from FFN is added

9.9.8 Key Insights

1. **FFN Adds Capacity**: Non-linearity enables complex functions
2. **ReLU is Simple**: max(0, x) is easy to compute
3. **Residuals Help**: Enable deep networks
4. **Composition**: Attention + FFN = more powerful model

9.9.9 Common Mistakes

1. **Forgetting ReLU**: Must apply non-linearity
2. **Wrong residual**: Add, don't replace
3. **Dimension mismatch**: Check matrix sizes

9.9.10 Next Steps

- Try different FFN sizes
- Experiment with other activations
- See Example 6 for complete transformer

9.10 Code Implementation

```
/**
 * ==================================================================
 * Example 5: Feed-Forward Layers
 * ============================================================================
 *
 * Goal: Add non-linearity and depth with feed-forward networks
 *
 * This example demonstrates:
 * - Feed-forward network after attention
 * - ReLU activation
 * - Residual connections
 * - Layer composition
 */

#include "../../src/core/Matrix.hpp"
#include "../../src/core/Embedding.hpp"
#include "../../src/core/LinearProjection.hpp"
#include "../../src/core/Attention.hpp"
#include "../../src/core/Softmax.hpp"
#include <iostream>
#include <vector>
#include <iomanip>
#include <algorithm>

// ReLU activation
Matrix relu(const Matrix& input) {
    Matrix result;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result.set(i, j, std::max(0.0, input.get(i, j)));
        }
    }
    return result;
}

int main() {
    std::cout << std::string(70, '=') << "\n";
    std::cout << "Example 5: Feed-Forward Layers\n";
    std::cout << "Goal: Add non-linearity with FFN\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Model setup
    Embedding embedding;
    Matrix WQ(0.1, 0.0, 0.0, 0.1);
    Matrix WK(0.1, 0.0, 0.0, 0.1);
    Matrix WV(0.1, 0.0, 0.0, 0.1);

    // Feed-forward weights
    Matrix W1(0.2, 0.1, 0.1, 0.2);  // First linear layer
    Matrix W2(0.2, 0.1, 0.1, 0.2);  // Second linear layer

    LinearProjection projQ(WQ, false);
    LinearProjection projK(WK, false);
    LinearProjection projV(WV, false);
    LinearProjection ffn1(W1, false);
    LinearProjection ffn2(W2, false);

    Attention attention;

    std::vector<int> input_tokens = {0, 1};  // A, B

    // Forward pass
    std::cout << "FORWARD PASS\n";
    std::cout << std::string(70, '=') << "\n\n";

    // Step 1: Attention
    Matrix X = embedding.forwardSequence(input_tokens);
    Matrix Q = projQ.forward(X);
    Matrix K = projK.forward(X);
    Matrix V = projV.forward(X);

    Attention::AttentionResult attn_result = attention.forward(Q, K, V);

    std::cout << "Attention output:\n";
    attn_result.output.print("Attention");
    std::cout << "\n";

    // Step 2: Feed-forward network
    std::cout << "Feed-Forward Network:\n";
    std::cout << std::string(70, '-') << "\n";
    // FFN(x) = ReLU(x × W1) × W2
    Matrix ffn_input = attn_result.output;
    Matrix ffn_hidden = ffn1.forward(ffn_input);
    Matrix ffn_activated = relu(ffn_hidden);
    Matrix ffn_output = ffn2.forward(ffn_activated);

    ffn_hidden.print("FFN hidden (before ReLU)");
    ffn_activated.print("FFN activated (after ReLU)");
    ffn_output.print("FFN output");
    std::cout << "\n";

    // Step 3: Residual connection
    std::cout << "Residual Connection:\n";
    std::cout << std::string(70, '-') << "\n";
    Matrix residual_output = attn_result.output.add(ffn_output);

    attn_result.output.print("Attention output");
    ffn_output.print("FFN output");
    residual_output.print("Final output (attention + FFN)");
    std::cout << "\n";

    std::cout << "Feed-forward adds non-linearity and capacity!\n";
    std::cout << "Residual connection enables gradient flow.\n\n";

    std::cout << "Next: See Example 6 for complete transformer.\n";
    std::cout << std::string(70, '=') << "\n";

    return 0;
}
```

Chapter 10

Example 6: Complete Transformer

Goal: Full implementation with all components

What You'll Learn: - Complete transformer architecture - Layer normalization - Multiple layers - End-to-end training

10.1 The Task

Build complete transformer with: - Multiple transformer blocks - Layer normalization - Residual connections everywhere - Complete training pipeline

10.2 Model Architecture

```
Input → Embeddings
     → Transformer Block 1 (Attention + FFN + LayerNorm + Residuals)
     → Transformer Block 2 (Attention + FFN + LayerNorm + Residuals)
     → Output Projection
     → Softmax
     → Probabilities
```

10.3 Layer Normalization

Normalize across features (not batch):

$$\text{LayerNorm}(x) = \gamma \frac{x - \mu}{\sigma} + \beta$$

Where: - μ: mean of features - σ: standard deviation of features - γ, β: learnable parameters

Why? - Stabilizes training - Reduces internal covariate shift - Enables larger learning rates

10.4 Multiple Layers

Stack transformer blocks: - Each block processes the output of previous - Deeper = more complex patterns - Residuals enable deep networks

10.5 Complete Training

Full pipeline: 1. Forward through all layers 2. Compute loss 3. Backprop through all layers 4. Update all weights 5. Repeat for many epochs

10.6 Hand Calculation Guide

See worksheet

10.7 Theory

10.7.0.1 Deep Networks

Each layer learns increasingly abstract features: - Layer 1: Local patterns - Layer 2: Combinations of Layer 1 patterns - Layer 3: High-level concepts

10.7.0.2 Why This Architecture Works

- **Attention**: Captures long-range dependencies
- **FFN**: Adds non-linearity and capacity
- **Residuals**: Enables gradient flow
- **LayerNorm**: Stabilizes training
- **Multiple layers**: Learns hierarchical representations

10.8 Code Implementation

See code

10.9 Exercises

1. Trace through complete forward pass
2. Compute all intermediate values
3. Perform full backpropagation

4. Train complete model
5. Analyze learned representations

Chapter 11

Appendix A: Matrix Calculus Reference

11.1 Basic Rules

1. **Scalar derivative**: $\frac{d}{dx}(ax) = a$
2. **Product rule**: $\frac{d}{dx}(f(x)g(x)) = f'(x)g(x) + f(x)g'(x)$
3. **Chain rule**: $\frac{d}{dx}(f(g(x))) = f'(g(x)) \cdot g'(x)$

11.2 Matrix Operations

11.2.0.1 Matrix Multiplication

For $C = AB$ where $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

11.2.0.2 Transpose

$$(A^T)_{ij} = A_{ji}$$

11.2.0.3 Gradient Rules

For $C = AB$: - $\frac{\partial L}{\partial A} = \frac{\partial L}{\partial C} B^T$ - $\frac{\partial L}{\partial B} = A^T \frac{\partial L}{\partial C}$

For $C = A^T$: - $\frac{\partial L}{\partial A} = \left(\frac{\partial L}{\partial C}\right)^T$

Chapter 12

Appendix B: Hand Calculation Tips

12.1 Organization

1. Write down all initial values clearly
2. Show intermediate steps
3. Label each computation
4. Check dimensions match
5. Verify final results

12.2 Common Patterns

- Matrix multiplication: row × column
- Dot product: sum of element-wise products
- Softmax: exp, sum, divide
- Gradients: chain rule systematically

12.3 Verification

- Check that probabilities sum to 1
- Verify gradient signs make sense
- Compare to code output
- Recompute if results don't match

Chapter 13

Appendix C: Common Mistakes and Solutions

13.1 Mistake 1: Forgetting Scaling Factor

Problem: Not dividing by $\sqrt{d_k}$ in attention

Solution: Always include scaling: $\frac{QK^T}{\sqrt{d_k}}$

13.2 Mistake 2: Softmax Numerical Instability

Problem: Computing e^x for large x causes overflow

Solution: Subtract max before exponentiating: $e^{x_i - \max(x)}$

13.3 Mistake 3: Wrong Gradient Sign

Problem: Adding gradient instead of subtracting

Solution: Remember: $W_{\text{new}} = W_{\text{old}} - \eta \nabla L$

13.4 Mistake 4: Dimension Mismatch

Problem: Trying to multiply incompatible matrices

Solution: Always check dimensions: $(m \times n) \times (n \times p) = (m \times p)$

Chapter 14

Conclusion

You've now mastered transformers from first principles! You can:

- Understand every component
- Compute everything by hand
- Implement from scratch
- Extend to larger models

The principles you've learned apply to GPT, BERT, and all transformer-based models. The math is identical - only the scale changes.

Next Steps: - Implement larger models - Experiment with different architectures - Read original papers with full understanding - Build your own transformer applications

---

This book is a living document. As you work through examples, refer back to relevant chapters. Each example builds on previous ones, creating a complete understanding.