Assignment 1: Understanding & Implementing Transformers

Alvaro Ibarra

September 24, 2025

Contents

1 Part A - Theory

1.1 Attention Mechanism

1.1.1 Question 1: Derive the Scaled Dot-Product Attention formula step by step

The attention mechanism basically lets each position in a sequence look at all other positions and decide what's important.

Starting with three matrices: queries (Q) - what we're looking for, keys (K) - what's available, and values (V) - the actual content.

First, we compute QK^T to get compatibility scores between every query and key. This gives us a matrix where each element shows how much one position should attend to another.

Next, we scale by $\sqrt{d_k}$ to prevent the scores from getting too large: $QK^T/\sqrt{d_k}$.

Then we apply softmax to turn these into probabilities: $\text{softmax}(QK^T/\sqrt{d_k})$. Each row now sums to 1.

Finally, we multiply by V to get the weighted output:

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

1.1.2 Question 2: Explain why we divide by $\sqrt{d_k}$

Without the scaling factor, the dot products get really large when d_k is big. Large values fed into softmax create nearly one-hot distributions (all weight goes to one position). This kills gradients during training.

The math behind it: if q and k components are random variables with variance 1, then $q \cdot k$ has variance d_k. So dividing by $\sqrt{d_k}$ keeps the variance at 1, maintaining reasonable softmax distributions that actually allow the model to train properly.

1.1.3 Question 3: What does the SoftMax achieve in this context?

Softmax does three main things here. It normalizes scores into probabilities that sum to 1, so we get a proper distribution over positions. It amplifies differences - higher scores get much more weight than lower ones. And it keeps everything differentiable, which we need for training.

Basically, it turns raw compatibility scores into interpretable attention weights that tell us how much each position matters.

1.2 Multi-Head Attention

1.2.1 Question 4: Why do we use multiple heads instead of one?

Like a multi-core processor where each core can handle different tasks simultaneously, multiple attention heads let the model focus on different types of relationships at the same time. One head might specialize in short-range dependencies (like adjective-noun pairs), another in long-range dependencies (like subject-verb agreement across a long sentence), and another in semantic relationships.

Using just one head would be like having a single-core processor - it could theoretically do everything, but it would have to choose one focus at a time instead of processing multiple types of patterns in parallel.

1.2.2 Question 5: Explain mathematically how queries, keys, and values are projected per head

Each head gets its own set of learned projection matrices: W_i^Q, W_i^K, and W_i^V for head i. Instead of using the original Q, K, V directly, each head computes:

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

These projections are what allow each head to learn different patterns - they transform the same input into different "subspaces" where different types of relationships become more apparent.

Finally, all heads are concatenated and projected through a final matrix W^O: $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$

1.2.3 Question 6: What advantage does this give in terms of representational power?

Just like how multiple CPU cores can run different programs optimally, multiple heads can specialize in different linguistic patterns without interfering with each other. This gives us much richer representations than a single head trying to capture everything at once.

1.3 Positional Encoding

1.3.1 Question 7: Why is positional information necessary in Transformers?

Without positional information, the Transformer would treat "The cat chased the dog" and "The dog chased the cat" as identical inputs since attention is permutation-invariant. This would lead to completely wrong outputs for tasks that depend on word order - like translation, parsing, or any language understanding task.

The model would waste computational resources trying to learn from inconsistent training data where the same word combinations could mean opposite things depending on their order. In many cases, this would make training impossible or produce unreliable results.

1.3.2 Question 8: Write down the sinusoidal encoding formulas and explain what property they provide

The sinusoidal encoding formulas are: $PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$ $PE(pos, 2i{+}1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$

These functions are good because they create a unique "fingerprint" for each position that the model can learn to recognize. The key property is that they provide **relative position information** - the model can learn that positions with similar sine/cosine patterns are close to each other, while positions with very different patterns are far apart.

The different frequencies (controlled by the $10000^{2i/d_{model}}$ term) ensure that each position gets a unique encoding while maintaining smooth relationships between nearby positions.

1.4 Feed-Forward Networks (MLP block)

1.4.1 Question 9: What role does the two-layer MLP play in each Transformer block?

After attention gathers and mixes information from different positions, the MLP processes that combined information to extract useful patterns and features. While attention is good at figuring out *what* to pay attention to, the MLP is where the actual *transformation* and *reasoning* happens on that gathered information.

The MLP adds non-linearity through the ReLU activation, which lets the model learn complex patterns that simple linear combinations can't capture. It's like having a processing unit that takes the mixed information from attention and decides what to do with it.

1.4.2 Question 10: Why do we apply it identically at every position?

We apply the same MLP transformation at every position because each position now contains a rich mixture of information from the attention step. By using identical parameters, we ensure consistent processing - the same type of pattern gets processed the same way regardless of where it appears in the sequence.

This also makes the model more efficient and helps with generalization. Rather than learning separate processing rules for each position, the model learns one set of transformation rules that work well everywhere.

1.5 Normalization and Residuals

1.5.1 Question 11: What is LayerNorm (or RMSNorm) doing mathematically?

LayerNorm stabilizes training by ensuring that the inputs to each layer have consistent statistics (mean 0, variance 1).

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

This prevents the "internal covariate shift" problem where layer inputs change dramatically during training, making it hard for the model to learn effectively. It's like giving each layer a consistent, normalized view of the data so it can focus on learning patterns rather than adapting to changing input scales.

1.5.2 Question 12: Why are residual connections important for training deep networks?

The main problem residual connections solve is **vanishing gradients**. In deep networks, gradients can become extremely small as they backpropagate through many layers, making training very slow or impossible.

Residual connections create "shortcuts" for gradients to flow directly backward: output $= F(x) + x$. This means gradients can take the direct path through the $+x$ part, ensuring they don't vanish even in very deep networks. It's like having express lanes on a highway - information can flow quickly without getting stuck in traffic.

This allows us to train much deeper networks that would otherwise be impossible to optimize.

1.6 Paper Results

1.6.1 Question 13: What main tasks did the authors evaluate on?

The authors evaluated on machine translation tasks - specifically WMT 2014 English-to-German and English-to-French translation. They also tested on English constituency parsing to show the model could generalize beyond translation.

They achieved state-of-the-art results, with their big model getting 28.4 BLEU on English-German (beating previous best by over 2 BLEU) and 41.8 BLEU on English-French. They did this while using significantly less training time and computational resources than previous models.

1.6.2 Question 14: Compare Transformers to RNNs/LSTMs in terms of parallelism and performance

Transformers have major advantages over RNNs/LSTMs:

Parallelism: RNNs process sequences step-by-step, so you have to wait for position t-1 before computing position t. Transformers can attend to all positions simultaneously, making them much more parallelizable during training.

Training Speed: Because of the parallelization, Transformers train much faster on modern hardware like GPUs that excel at parallel computation.

Long Sequences: RNNs struggle with long-range dependencies due to vanishing gradients over many time steps. Transformers can directly connect any two positions with constant path length, making them better at capturing long-range relationships.

Performance: The results speak for themselves - Transformers achieved better quality translations while requiring less training time.

2 Part B - Implementation

2.1 Implementation Results

The Transformer model was successfully implemented and trained on the Tiny Shakespeare dataset. The implementation includes all required components:

- Token and positional embeddings
- Multi-head self-attention with causal masking
- Feed-forward MLP blocks
- LayerNorm and residual connections
- Final linear projection to vocabulary

2.1.1 Training Configuration

The model was trained with the following hyperparameters:

- Layers: 6
- Hidden size (d_model): 384
- Attention heads: 6
- Sequence length: 256
- Batch size: 64
- Parameters: 10.8M

2.1.2 Training Results

- Training time: 11.3 minutes (680 seconds)
- Final training loss: 1.0564
- Final validation loss: 1.2575
- The model converged smoothly from initial loss of 4.2 to final loss of 1.06

2.1.3 Text Generation

The trained model successfully generates Shakespeare-like text with:

- Proper character dialogue format
- Period-appropriate vocabulary and phrasing
- Coherent conversational structure
- Recognition of Shakespeare character names

Sample generation with prompt "ROMEO:":

> *"The gods that do resolve me not and leave.*
> *ROMEO:*
> *I will not be long in blood of the mother."*

3 Part C - Computer Architecture & System Analysis

3.1 System Specification

3.1.1 Hardware Configuration

The training system specifications are:

- **CPU**: 16 logical cores (8 physical cores)
- **GPU**: NVIDIA GeForce RTX 4060 Laptop GPU
- **GPU Memory**: 7.6 GB GDDR6
- **System Memory**: 14.9 GB RAM
- **CUDA Version**: 12.1
- **PyTorch Version**: 2.5.1+cu121

3.2 Theoretical Peak Performance

3.2.1 GPU Performance Analysis

The RTX 4060 Laptop GPU specifications:

- CUDA Cores: 3072
- Base Clock: 1470 MHz
- Boost Clock: 1890 MHz
- Memory Bandwidth: 288 GB/s (theoretical)
- Tensor Performance: 15 TOPS (INT8)

Theoretical FP32 Performance: Peak FLOPs = CUDA Cores$\times$Clock Speed$\times$Operations per Clock $= 3072 \times 1.89$ GHz $\times 2 = 11.6$ TFLOPS

3.3 Practical Performance Analysis

3.3.1 Model Memory Analysis

For our 10.8M parameter Transformer model:

- **Parameters**: 41.2 MB (FP32)
- **Gradients**: 41.2 MB (same as parameters)
- **Optimizer State**: 82.4 MB (AdamW stores momentum terms)
- **Activations**: Variable based on batch size and sequence length
- **Total Training Memory**: 165 MB + activations

3.3.2 Memory Utilization

With batch size 64 and sequence length 256:

- Used GPU memory during training: 6-7 GB out of 7.6 GB available
- Memory utilization: 85-90%
- Remaining headroom: 1 GB for system overhead

3.4 Transformer Workload Analysis

3.4.1 FLOP Estimation for Forward Pass

For one forward pass with our configuration:

- **Multi-Head Attention**: $4 \times \text{seq\_len}^2 \times d_{model} + 4 \times \text{seq\_len} \times d_{model}^2$
- **Feed-Forward**: $2 \times \text{seq\_len} \times d_{model} \times d_{ff}$
- **Total per layer**: 50M FLOPs per token
- **Total model**: 300M FLOPs per token (6 layers)
- **Per batch**: 5 GFLOPs (batch_size=64, seq_len=256)

3.4.2 Compute vs Memory Bound Analysis

Arithmetic Intensity Analysis: Arithmetic Intensity $= \frac{\text{FLOPs}}{\text{Bytes Accessed}}$

For attention mechanism: $AI = \frac{\text{seq\_len} \times d_{model}^2}{\text{seq\_len} \times d_{model} \times 4} = \frac{d_{model}}{4} = \frac{384}{4} = 96$ FLOPs/byte

Given GPU memory bandwidth 288 GB/s: Memory-bound performance = 288×96 = 27.6 TFLOPS

Since theoretical peak (11.6 TFLOPS) ¡ memory-bound performance (27.6 TFLOPS), our workload is **compute-bound**.

3.5 Training vs Inference Efficiency

3.5.1 FLOP Comparison

- **Forward pass**: 5 GFLOPs per batch
- **Backward pass**: 2× forward pass = 10 GFLOPs
- **Total training**: 15 GFLOPs per batch
- **Training overhead**: 3× more compute than inference

3.5.2 Hardware Utilization

- **Training**: High GPU utilization (85-90%) due to parallel forward/backward
- **Inference**: Lower utilization due to sequential generation
- **Memory efficiency**: Training uses full batch processing advantage

3.6 Scaling Considerations

3.6.1 Parameter Scaling

Doubling model dimensions (384 → 768):

- **Parameters**: 10.8M × 4 = 43.2M (quadratic scaling)
- **Memory**: 165MB × 4 = 660MB (still fits in 7.6GB)
- **FLOPs**: 5GFLOPs × 4 = 20GFLOPs per batch

3.6.2 Optimization Strategies

For larger models that exceed memory capacity:

- **Gradient Checkpointing**: Trade compute for memory
- **Mixed Precision (FP16)**: 50% memory reduction
- **Gradient Accumulation**: Simulate larger batches
- **Model Sharding**: Distribute across multiple GPUs

3.7 Performance Conclusions

3.7.1 Bottleneck Analysis

- **Current model**: Compute-bound on RTX 4060
- **Training efficiency**: 85-90% GPU utilization achieved
- **Memory utilization**: Optimal at 90% capacity
- **Scaling headroom**: Can handle 4× larger models with current memory

3.7.2 System Optimization

The RTX 4060 Laptop GPU performs well for this workload:

- Sufficient compute for transformer training
- Adequate memory bandwidth for model size
- Good utilization of available resources
- Reasonable training time (11 minutes for 5000 iterations)

4 Conclusion

This assignment successfully implemented and analyzed a Transformer model from scratch, achieving several key objectives:

4.1 Implementation Success

- Successfully implemented all Transformer components following the original "Attention Is All You Need" paper
- Achieved excellent training results with final validation loss of 1.26
- Generated coherent Shakespeare-like text demonstrating proper language modeling
- Training completed efficiently in 11 minutes on consumer GPU hardware

4.2 Technical Understanding

The theoretical analysis provided deep insights into:

- Mathematical foundations of scaled dot-product attention
- Benefits of multi-head attention for representational power
- Importance of positional encoding for sequence modeling
- Role of normalization and residual connections in training stability

4.3 System Analysis Insights

The computer architecture analysis revealed:

- Current workload is compute-bound rather than memory-bound
- RTX 4060 provides sufficient resources for medium-scale transformer training
- System achieves 85-90% GPU utilization during training
- Model can scale $4\times$ larger while fitting in available memory

4.4 Practical Impact

This implementation demonstrates that:

- Modern consumer hardware can train meaningful language models
- Proper implementation of research papers yields production-quality results
- Character-level modeling can achieve impressive text generation quality
- System analysis is crucial for understanding model performance characteristics

The successful completion of this assignment bridges theoretical understanding with practical implementation, providing a comprehensive foundation for working with transformer architectures in research and industry applications.