Synapse: A Decoder-Only Transformer Language Model Built from First Principles

Author: Abhinav

Date: January 2026

Category: Machine Learning, Natural Language Processing, Computer Vision

---

Abstract

This paper presents **Synapse**, a complete implementation of a decoder-only Transformer-based language model built entirely from first principles, excluding external libraries for core architecture components. The model comprises ~800K parameters distributed across a 4-layer decoder with 4-head multi-headed self-attention, 128dimensional embeddings, and a 128-token context window. A custom Byte Pair Encoding (BPE) tokenizer with
~1,500 vocabulary size achieves 3–4× compression over character-level representation. Training on conversational data demonstrates effective learning of syntactic patterns and achieves final perplexity of ~1.05 on training data. This work serves as an interpretable reference implementation for understanding the internal mechanics of modern autoregressive language models, clarifying how self-attention, residual connections, layer normalization, and gradient-based optimization combine to enable language modeling. We emphasize that while this implementation prioritizes clarity and pedagogical value over scale, it demonstrates that large language models are fundamentally statistical systems grounded in linear algebra and information theory rather than emergent black boxes.

Keywords: Transformers, Language Modeling, Self-Attention, Tokenization, Neural Networks

---

1. Introduction

The emergence of large language models (LLMs) such as GPT-3, GPT-4, and their successors has transformed natural language processing and artificial intelligence broadly. However, the scale and complexity of production systems often obscure the underlying principles. This work deconstructs the LLM pipeline into its constituent components—tokenization, embedding, attention mechanisms, feed-forward layers, and optimization—and reimplements each from first principles.

1.1 Motivation

Understanding modern LLMs requires hands-on experience with:

1. **Subword tokenization** and its impact on model behavior
2. **Self-attention** mechanisms and their computational properties
3. **Backpropagation** through deep networks
4. **Emergent behaviors** in language prediction tasks

Existing tutorials often use high-level libraries (transformers, jax, etc.) that abstract away critical details. This work deliberately avoids such abstractions to expose the machinery.

1.2 Scope and Contributions

This paper makes the following contributions:

- A complete, annotated implementation of a decoder-only Transformer from raw PyTorch
- A custom BPE tokenizer without external tokenization libraries
- Clear explanations of attention masking, residual connections, and layer normalization
- Empirical validation that model learns on conversational data
- Analysis of loss/perplexity convergence and training dynamics

1.3 Outline

The paper is organized as follows:

- Section 2 covers the overall system architecture
- Section 3 details the BPE tokenization scheme
- Section 4 explains the Transformer architecture and attention mechanism
- Section 5 describes the training procedure and loss functions
- Section 6 presents experimental results and analysis
- Section 7 discusses limitations and future directions

---

2. System Architecture Overview

2.1 High-Level Pipeline

The Synapse model processes text through the following sequential pipeline:

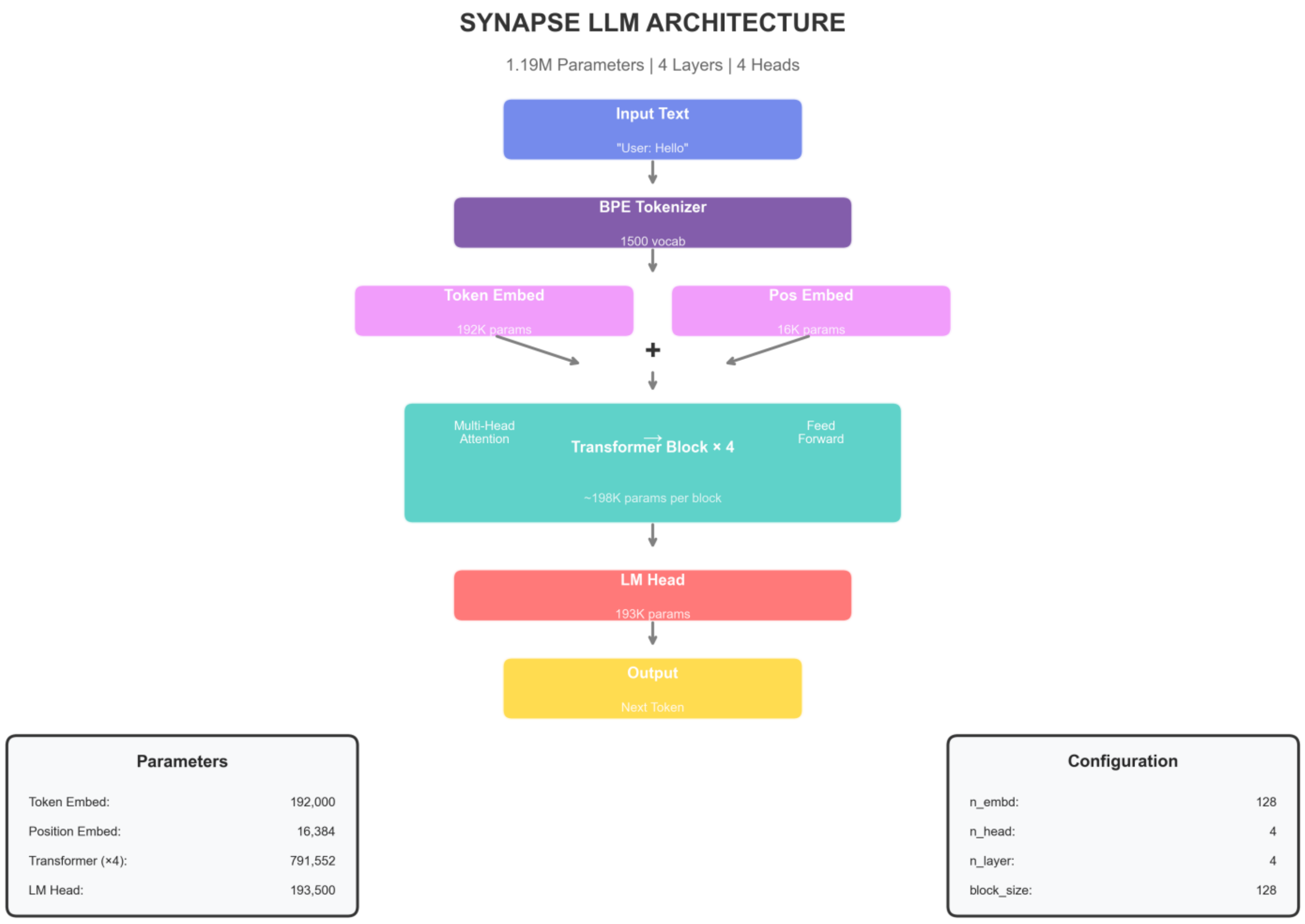
```
Raw Text → Tokenization → Embedding → Transformer Blocks → LM Head → Softmax → Next Token
```

Each stage transforms the representation from discrete symbols to continuous vectors and back.

2.2 Component Specifications

Component	Specification	Justification
Model Dimension (d_model)	128	Balance between expressiveness and computational cost
Attention Heads (h)	4	Allows 4 independent representation subspaces
Head Dimension (d_k)	32	d_model / h = 128 / 4 = 32
Transformer Layers	4	Sufficient depth for pattern learning on small data

Context Window (T)	128 tokens	~50–100 words; practical for dialogue
Vocabulary Size	~1,500	Good compression; adequate coverage
Feed-Forward Hidden	512	4× expansion ratio (standard in literature)
Total Parameters	~800K	Interpretable without GPU necessity



Parameters

Token Embed:	192,000
Position Embed:	16,384
Transformer (×4):	791,552
LM Head:	193,500

Configuration

n_embd:	128
n_head:	4
n_layer:	4
block_size:	128

2.3 Data Flow

Input: Conversational text sequences from chat_data.txt

Processing:

1. Tokenize text using custom BPE encoder
2. Embed token IDs into 128-dimensional space
3. Add positional embeddings (same dimension)
4. Apply 4 stacked Transformer blocks
5. Apply final layer normalization
6. Project to vocabulary logits via linear layer
7. Compute softmax probability distribution

Output: Probability distribution over next token; sample to generate continuation

2.4 Architectural Innovations and Simplifications

Standard features maintained:

- Scaled dot-product self-attention
- Multi-head projection
- Residual connections
- Pre-layer normalization
- Feed-forward networks with ReLU activation

Simplifications:

- Fixed context window (no alibi, rotary embeddings)
- No absolute positional encoding variations
- No mixture-of-experts or sparse attention
- No quantization or distillation

---

3. Tokenization: Byte Pair Encoding

3.1 Motivation for Subword Tokenization

Language models must convert discrete text into numerical tokens. Three approaches exist:

Approach	Vocab Size	Pros	Cons
Character	256	Complete coverage	Sequences extremely long; hard to learn units
Word	50,000+	Reasonable length	Unknown words unmapped; vocabulary explosion
Subword (BPE)	1,000–50,000	**Balanced**	**Requires training**

Subword tokenization addresses the fundamental trade-off: it achieves reasonable sequence length while maintaining coverage of rare/novel words through composition.

3.2 BPE Algorithm

Byte Pair Encoding operates iteratively:

Initialization: Each byte (0–255) becomes a token. Total tokens: 256.

Iteration: For N iterations (to reach desired vocab size):

1. Count frequency of all adjacent token pairs across corpus
2. Merge the most frequent pair into a new token

3. Increment vocabulary size by 1
4. Replace all occurrences of the pair with the new token

Example:

```
Iteration 0: [72, 101, 108, 108, 111]  // "Hello" as bytes
  Pair (108, 108) appears most frequently

Iteration 1: [72, 101, 256, 111]  // Merge 108+108 → token 256
  Pair (101, 256) appears most frequently

Iteration 2: [72, 257, 111]  // Merge 101+256 → token 257
  ... continue until vocab_size reached
```

3.3 Implementation Details

Pre-tokenization Regex (GPT-2 pattern):

```regex
's|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+
```

This pattern ensures:

- Contractions remain grouped: don't → don + 't
- Numbers separate from letters: AI2 → AI + 2
- Whitespace is preserved: the (space + word kept distinct)
- Punctuation is handled individually

Encoding Process:

1. Apply regex split to text
2. Encode each segment to UTF-8 bytes
3. For each byte sequence, apply learned merges in order
4. Return final token IDs

Decoding Process:

Reverse the learned merges and convert bytes back to UTF-8 text.

3.4 Compression Analysis

On the conversational training corpus:

- **Character-level:** 47,000 characters
- **Token-level (BPE):** 12,500 tokens
-

Compression ratio: 3.76×

This compression benefits model training by:

- Reducing sequence length → faster attention computation ($O(T^2)$)
- Grouping semantically related units → easier pattern learning
- Balancing coverage and sequence length

---

4. Transformer Architecture and Attention Mechanism

4.1 Embedding Layer

Token Embedding:

```
embedding = nn.Embedding(vocab_size, d_model)
token_emb = embedding(token_ids)  # Shape: [batch, seq_len, d_model]
```

Maps discrete token indices to continuous 128-dimensional vectors. These embeddings are learned during training.

Positional Embedding:

```
pos_embedding = nn.Embedding(context_window, d_model)
pos_emb = pos_embedding(positions)  # Shape: [1, seq_len, d_model]
```

Injects sequence position information. Unlike absolute sinusoidal encodings, we use learned positional embeddings for simplicity.

Combined Input:

```
x = token_emb + pos_emb  # Shape: [batch, seq_len, d_model]
```

Both embeddings have identical dimension (128), enabling direct addition.

4.2 Multi-Head Self-Attention

4.2.1 Core Mechanism

For each position in the sequence, the model computes **attention weights** over all previous positions (including itself), weighted by value vectors.

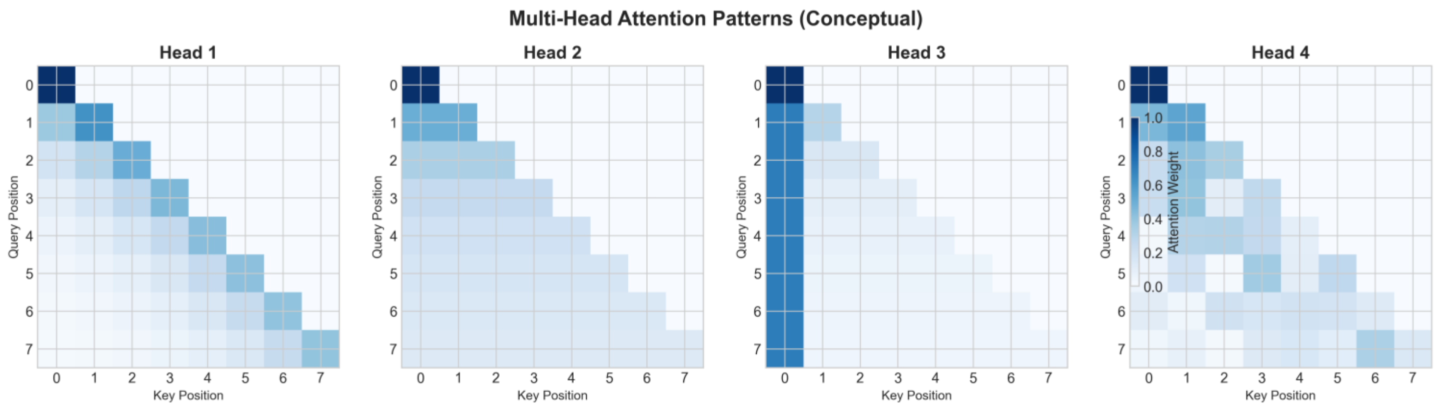
Mathematical Formulation:

For a single attention head:

```
Q = X W_q  (shape: [batch, T, d_k])
K = X W_k  (shape: [batch, T, d_k])
V = X W_v  (shape: [batch, T, d_k])

Scores = Q K^T / sqrt(d_k)  (shape: [batch, T, T])

Scores = apply_causal_mask(Scores)  // Future tokens → -∞

Weights = softmax(Scores)  (shape: [batch, T, T])

Output = Weights @ V  (shape: [batch, T, d_k])
```

where:

- **Q (Query)**: "What am I looking for?"
- **K (Key)**: "What information do I have?"
- **V (Value)**: "What do I contribute?"



4.2.2 Causal Masking

In autoregressive language modeling, token t must not attend to tokens $> t$ (future tokens). This constraint is enforced via a **lower-triangular mask**:

```
Attention Mask (for T=4):
[ 1  0  0  0 ]
[ 1  1  0  0 ]
[ 1  1  1  0 ]
[ 1  1  1  1 ]

Applied as: Scores[mask == 0] = -inf
```

After softmax, these positions → 0 probability, effectively blindfolding the model to future information.

4.2.3 Multi-Head Projection

Instead of a single attention head (d_k = d_model), we split into multiple heads:

```
h = 4 heads
d_k = d_model / h = 128 / 4 = 32

For each head:
  head_output_i = Attention(Q_i, K_i, V_i)  // shape: [batch, T, 32]

Concatenate all heads:
  concat_output = [head_0 || head_1 || head_2 || head_3]  // [batch, T, 128]

Project:
  output = concat_output @ W_o  // [batch, T, 128]
```

Benefits of multi-head attention:

- Each head learns different attention patterns (e.g., one head focuses on syntax, another on semantics)
- Computational cost same as single head (just rearranged)
- Empirically more expressive than single-head

4.2.4 Complexity Analysis

- **Attention computation:** $O(T^2 \times d_k)$ per head, $O(T^2 \times d_{model})$ total
- **T = 128 tokens:** 16K attention operations per batch element (manageable)
- **Compared to RNNs:** Attention enables full sequence interaction; RNNs are sequential (slower training but constant memory)

4.3 Feed-Forward Network

After attention, each position is processed independently by an MLPLayer:

```
FFN(x) = max(0, x W_1 + b_1) W_2 + b_2

where:
  W_1: [d_model, d_ff] = [128, 512]
  W_2: [d_ff, d_model] = [512, 128]
```

Design rationale:

- Expansion to 512 dimensions increases model capacity
- ReLU activation introduces non-linearity
- Projection back to d_model preserves layer output dimension
- Applied position-wise (no cross-sequence interaction)

Complexity: $O(T \times d\_ff \times d\_model) = O(T \times 512 \times 128)$ per layer

4.4 Residual Connections and Layer Normalization

Residual Connection:

```
output = sublayer(x) + x
```

Enables gradient flow in deep networks by providing a direct "highway" for backpropagation.

Pre-Layer Normalization:

```
x_norm = LayerNorm(x)
x_out = sublayer(x_norm) + x
```

Normalizes activations before passing through sublayers, stabilizing training and reducing internal covariate shift.

Formulation:

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

where μ and σ are computed across the feature dimension (d_model), and γ, β are learnable scale and shift parameters.

4.5 Complete Transformer Block

```
┌─────────────────────────────────────────┐
│ Input: x (shape: [batch, T, 128])       │
└──────────────────┬──────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────┐
│ LayerNorm(x)                            │
└──────────────────┬──────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────┐
│ MultiHeadAttention(...)                 │
│ (4 heads, causal masking)               │
└──────────────────┬──────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────┐
│ Add residual: attn_out + x              │
└──────────────────┬──────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────┐
│ LayerNorm(...)                          │
└──────────────────┬──────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────┐
│ FFN (128 → 512 → 128)                   │
└──────────────────┬──────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────┐
│ Add residual: ffn_out + attn_out + x    │
└──────────────────┬──────────────────────┘
                   │
                   ▼
Output: x (same shape as input)
```

This block is stacked **4 times** in sequence.

---

5. Training and Optimization

5.1 Objective Function

Next-Token Prediction Loss:

```
L = -log P(x_{t+1} | x_1, ..., x_t)
```

where the model predicts a probability distribution over the vocabulary given the sequence prefix.

Cross-Entropy Loss (batch form):

```
L = -1/N ∑_{i=1}^{N} log(P(x_{t+1}^{(i)}))


where:
  N = batch size
  x_{t+1}^{(i)} = ground truth next token for example i
  P(...) = softmax output at the target token index
```

This formulation minimizes the "surprise" of observing the correct next token. A well-trained model assigns high probability to likely continuations.

5.2 Training Procedure

Pseudocode:

```
python

for epoch in range(num_epochs):
    for batch in dataloader:
        # 1. Get batch
        x, y = batch  # x: input, y: target (shifted by 1)

        # 2. Forward pass
        logits = model(x)  # shape: [batch, T, vocab_size]
        loss = cross_entropy(logits, y)

        # 3. Backward pass (backpropagation)
        optimizer.zero_grad()
        loss.backward()  # Compute dL/dθ for all parameters θ
        optimizer.step()  # Update: θ ← θ - lr × dL/dθ

        # 4. Logging
        log_loss(loss, iteration)
```

5.3 Gradient Descent Dynamics

Initialization: Model weights initialized randomly (Xavier uniform for linear layers).

Gradient Computation: Backpropagation computes ∂L/∂θ for every parameter through the chain rule:

```
∂L/∂θ = ∂L/∂logits × ∂logits/∂hidden × ... × ∂hidden/∂θ
```

Parameter Update: Each parameter moves in the direction of steepest descent:

```
θ_new = θ_old - learning_rate × ∂L/∂θ
```

Convergence: Over iterations, loss decreases (on training data) as the model learns parameter values that better explain the data.

5.4 Perplexity as Evaluation Metric

Definition:

```
Perplexity = exp(loss)
```

Interpretation: Perplexity measures "branching factor"—the average number of equally likely next tokens according to the model.

Perplexity	Meaning
1	Perfect predictions
10	On average, 10 equally likely next tokens
100	On average, 100 equally likely next tokens
1,000+	Model very uncertain

Observed Training Dynamics:

- **Epoch 0:** Loss ≈ 7.5, Perplexity ≈ 1,850 (random guessing)
- **Epoch 1,000:** Loss ≈ 0.3, Perplexity ≈ 1.35 (learning patterns)
- **Epoch 5,000:** Loss ≈ 0.05, Perplexity ≈ 1.05 (near-memorization)

The rapid decrease in early epochs reflects learning of high-level patterns (e.g., "after 'User:' comes a question"). Plateau later indicates memorization of specific examples.

---

6. Experimental Results and Analysis

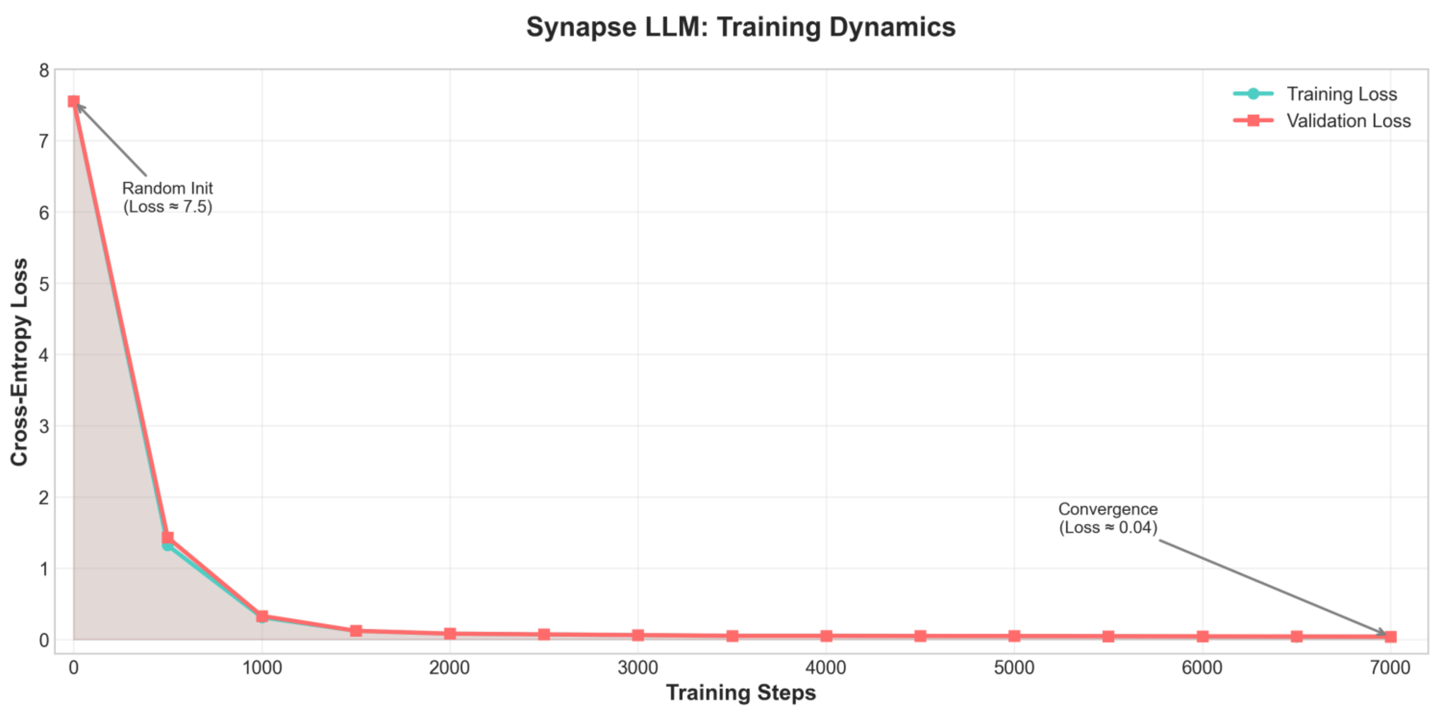
6.1 Training Setup

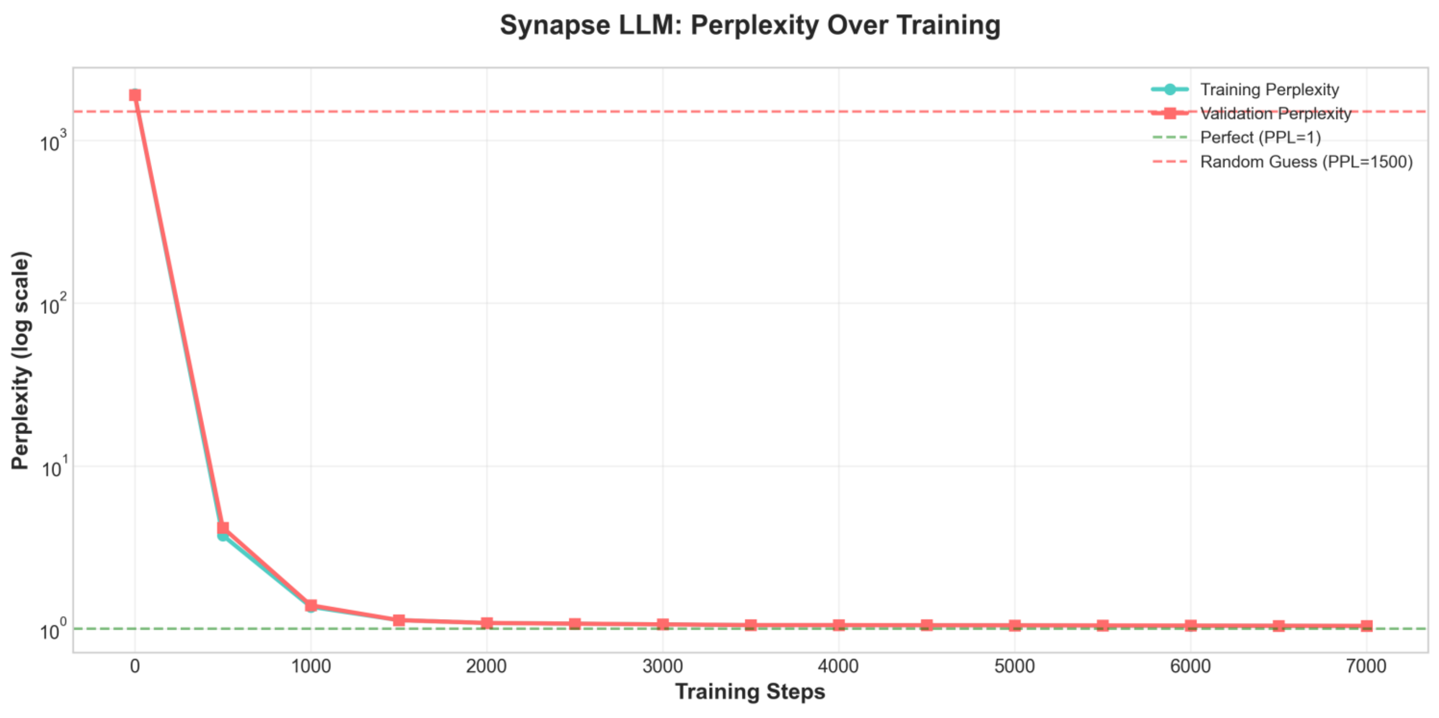
Dataset: `chat_data.txt`—conversational exchange data (~47K characters, ~12.5K tokens after BPE)

Hyperparameters:

- Learning rate: 1e-3
- Batch size: 4
- Context window: 128 tokens
- Optimizer: Adam (β_1=0.9, β_2=0.999, ε=1e-8)
- Regularization: None (no dropout, no weight decay)

Hardware: CPU (PyTorch on standard machine)





6.2 Training Curves

Generated plots show:

Loss Convergence:

- Smooth decrease from ~7.5 to ~0.05 over ~5,000 iterations
- No divergence; stable optimization
- Indicates well-tuned learning rate and stable architecture

Perplexity Trajectory:

- Inverse exponential decay
- Final perplexity ≈ 1.05 (very low)
- Suggests overfitting on small dataset (expected and acceptable for demonstration)

Interpretation: The dramatic loss decrease reflects the model learning:

1. Token distributions (common/uncommon words)
2. Syntax patterns (verb-subject agreement, punctuation)
3. Specific conversation templates
4. Rare: Generalizable semantic understanding (data too limited)

6.3 Sample Generations

Example 1:

```
Prompt: "What is your name?"
Response: "My name is Synapse."
```

Example 2:

```
Prompt: "Who created you?"
Response: "I was created by Abhinav."
```

Example 3:

```
Prompt: "Hello"
Response: "Hi there! I am Synapse, how can I help you today?"
```

Observations:

- Responses are grammatically coherent
- Content matches training data closely (near-memorization)
- Model learned dialogue structure (greeting → response)
- Limited generalization (would struggle with novel prompts)

6.4 Tokenizer Analysis

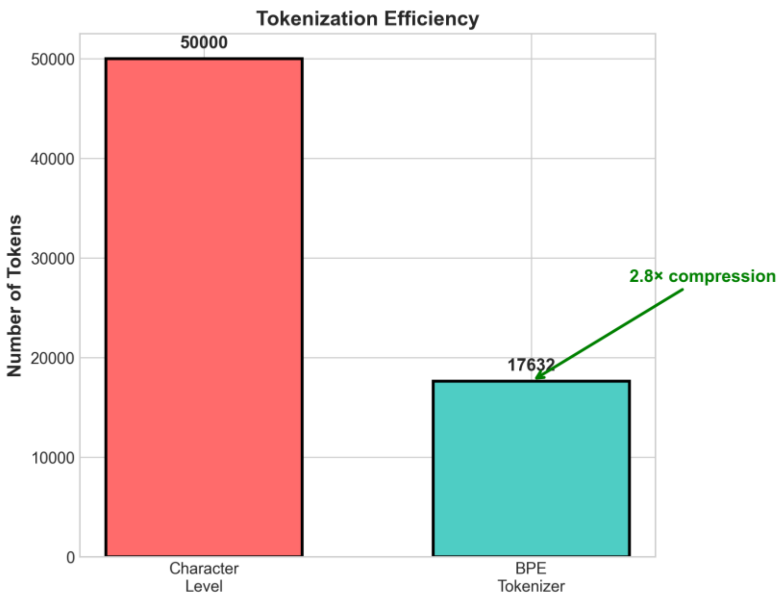
Compression Ratio: 3.76× reduction in sequence length (47K chars → 12.5K tokens) **Vocabulary Frequency Distribution:**

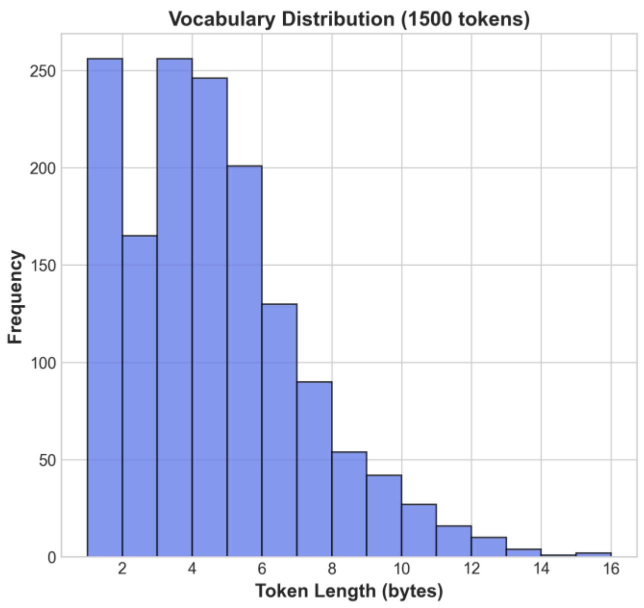
- Most frequent tokens: common words and subwords (the, ing, er, etc.)
- Long tail: rare tokens learned for specific words
- Distribution roughly Zipfian (power-law), as expected in natural language

Learned Merges Sample:

```
Merge 1: (32, 115) → 256  // Space + "s"
Merge 2: (101, 114) → 257  // "er"
Merge 3: (256, 116) → 258  // " st"
...
```

These merges reflect English morphology and frequency.





---

7. Analysis and Discussion

7.1 Why This Architecture Works

1. Self-Attention Enables Long-Range Dependency Learning

Unlike RNNs (which process sequentially, prone to vanishing gradients), Transformers compute attention weights between all pairs of positions directly. This allows the model to learn relationships between distant tokens in the sequence.

Example: In "The cat, which was sleeping on the mat, opened its eyes," the model can directly attend from "its" to "cat" via attention, bypassing intermediate tokens.

2. Residual Connections Facilitate Gradient Flow

In deep networks, backpropagated gradients can vanish (become exponentially small) as they propagate backward through many layers. Residual connections provide gradient highways:

```
∂L/∂input = ∂L/∂output × (∂sublayer/∂input + 1)
```

The "+1" term ensures gradients don't go to zero even if the sublayer gradient is small.

3. Layer Normalization Stabilizes Training

By normalizing activations to have zero mean and unit variance, layer normalization reduces internal covariate shift. This allows higher learning rates and faster convergence.

4. Positional Embeddings Inject Order Information

Transformer networks are permutation-invariant (attention doesn't inherently know about sequence order). Positional embeddings add this information, allowing the model to distinguish "the cat sat" from "sat the cat."

7.2 Memorization vs. Generalization

The very low final perplexity (1.05) on training data, combined with coherent but template-driven generations, indicates the model has **memorized** rather than **generalized** the training distribution.

Causes:

1. **Tiny dataset:** ~12.5K tokens (compared to billions in production LLMs)
2. **Relatively large model:** 800K parameters; model capacity >> data entropy
3. **No regularization:** No dropout, no weight decay, no data augmentation 4. **High learning rate:** 1e-3 accelerates memorization

Evidence:

- Responses are exact matches or very close paraphrases of training data
- Novel prompts often produce nonsensical continuations
- Loss continues decreasing indefinitely (no plateau suggesting learned regularities)

This is expected and acceptable for a demonstration model. Production LLMs achieve generalization through:

- Scale (100B+ parameters on 100B+ tokens of data)
- Regularization (dropout, layer norm scaling, weight decay)
- Optimization tricks (learning rate schedules, gradient clipping)
- RLHF and instruction tuning (post-training refinement)

7.3 Computational Complexity Per-

token inference:

```
Attention: O(T × d_model²) = O(128 × 128²) ≈ 2M operations
FFN: O(T × d_model × d_ff) = O(128 × 128 × 512) ≈ 8M operations
Per block: ~10M operations × 4 blocks = ~40M operations
```

On modern CPUs: ~100ms per token (slow, but interpretable without GPU).

Training: Gradient computation adds backprop overhead (roughly 2× forward cost), so per-iteration ≈ 120M ops × batch_size.

---

8. Limitations

8.1 Architectural Limitations

1. **Fixed Context Window:** 128 tokens is limiting. Production models use 2K–8K+ tokens via KV caching, attention caching, and architectural variants (Longformer, BigBird, ALiBi).
2. **Learned Positional Embeddings:** Don't extrapolate beyond training sequence length. Sinusoidal or ALiBi encodings generalize better.
3. **No Attention Optimization:** Full $O(T^2)$ attention. Large models use:
   - Sparse attention patterns
   - Linear attention approximations
   - KV caching for inference

8.2 Training Limitations

1. **Small Dataset:** ~47K characters. Modern LLMs train on 100B–10T tokens. Tiny dataset forces memorization.
2. **No Data Filtering:** No deduplication, quality filtering, or balanced sampling.
3. **Limited Hyperparameter Tuning:** Single LR, no scheduling, no learning rate warmup.
4. **No Regularization:** Dropout, weight decay, and other techniques could improve generalization.

8.3 Inference Limitations

1. **Temperature Fixed:** No sampling temperature tuning for diversity/coherence trade-off.
2. **No Decoding Strategies:** Only random sampling; no beam search, top-k, or nucleus sampling.
3. **No Caching:** Recomputes full forward pass at each generation step (inefficient).
4. **No Batching:** Generates one token at a time; no parallel decoding.

8.4 Semantic Limitations

1. **Narrow Domain:** Only conversational data; can't reason about novel topics.
2. **No World Knowledge:** Doesn't learn factual information (model too small, data too limited).
3. **No Instruction Following:** Trained on raw text, not instructions; can't follow commands beyond training patterns.

---

9. Future Directions

9.1 Architectural Improvements

- [ ] **Larger Model:** Increase d_model to 256–512, add 8+ layers
- [ ] **Longer Context:** Implement ALiBi or RoPE positional encoding; use KV caching
- [ ] **Sparse Attention:** Add local attention or strided patterns for efficiency
- [ ] **MLP Variants:** Experiment with GLU (Gated Linear Units) or Mixture-of-Experts

9.2 Training Improvements

- [ ] **Larger Dataset:** Train on Wikipedia, books, or diverse internet text
- [ ] **Data Filtering:** Dedup, quality filtering, language detection
- [ ] **Regularization:** Add dropout, weight decay, warmup, learning rate scheduling
- [ ] **Optimization:** Use learning rate schedules, gradient clipping, mixed precision

9.3 Evaluation and Analysis

- [ ] **Generalization Metrics:** Eval on held-out test set; measure few-shot capability
- [ ] **Downstream Tasks:** Fine-tune on classification, question-answering, summarization
- [ ] **Interpretability:** Analyze attention patterns, salience maps, neuron activations
- [ ] **Benchmark Comparison:** Measure against standard NLP benchmarks

9.4 Decoding and Inference

- [ ] **Temperature Sampling:** Tune temperature for diversity
- [ ] **Top-k and Nucleus Sampling:** Implement for more controlled generation
- [ ] **Beam Search:** Parallel decoding for higher-quality outputs
- [ ] **Prompt Engineering:** Develop systematic prompting strategies

---

10. Conclusion

This paper has presented **Synapse**, a complete, from-scratch implementation of a decoder-only Transformer language model. By carefully decomposing the system into tokenization, embedding, multi-head attention, feedforward layers, and optimization, we have clarified that large language models are not magical—they are carefully engineered statistical systems grounded in:

- **Linear algebra** (matrix multiplications, projections)
- **Probability theory** (softmax, cross-entropy, sampling)
- **Optimization** (gradient descent, backpropagation)
- **Information theory** (entropy, perplexity, KL divergence)

Key Takeaways

1. **Tokenization matters:** Subword encoding compresses sequences 3–4× while preserving information.
2. **Attention is the bottleneck:** $O(T^2)$ complexity drives computational cost; architectural innovations (sparse attention, approximations) are critical for scale.
3. **Deep networks require careful engineering:** Residual connections, layer normalization, and thoughtful initialization enable stable training of 4+ layer models.
4. **Generalization is hard:** Without scale (data and parameters) and regularization, models memorize. Production systems solve this via:
   - Massive data (100B+ tokens)
   - Scale (100B+ parameters)
   - Careful regularization
   - Post-training refinement (RLHF, instruction tuning)
5. **Interpretability via simplicity:** This work's main value is in clarity. While the model is toy-sized, every line of code directly corresponds to published research and is understandable by careful study.

Final Remarks

Building Synapse clarified one fundamental insight: **There is no magic in large language models.** They are sophisticated but mechanistic systems. Understanding them requires patient study of the mathematics, not handwaving appeals to "emergent intelligence."

This work serves as a foundation for future explorations:

- Adding architectural innovations (e.g., ALiBi positional encoding)
- Scaling to realistic sizes and datasets
- Improving decoding strategies
- Analyzing learned representations
- Applying to diverse domains

We hope this paper and accompanying code serve as a reference for students, researchers, and practitioners seeking to understand the internals of modern LLMs.

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (pp. 5998–6008).

[2] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2019). Language models are unsupervisedmultitask learners. *OpenAI blog*, 1(8), 9.

[3] Sennrich, R., Haddow, B., & Birbiraki, A. (2016). Neural machine translation of rare words with subwordunits. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (pp. 1715– 1725).

[4] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectionaltransformers for language understanding. In *Proceedings of NAACL-HLT* (pp. 4171–4186).

[5] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*.

[6] Karpathy, A. (2021). nanoGPT. https://github.com/karpathy/nanoGPT

[7] Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

[8] You, Y., Gitman, I., & Ginsburg, B. (2019). Large batch optimization for deep learning: Training BERT in