Synapse v2: A Decoder-Only Transformer Language Model Built from First Principles

An Evolution in Scale, Architecture, and Instruction Understanding

Author: Abhinav Tyagi
Date: January 2026
Category: Machine Learning, Natural Language Processing, Deep Learning

---

Abstract

This paper presents **Synapse v2**, a complete implementation of a decoder-only Transformer-based language model representing a significant architectural evolution from toy-scale to production-oriented design. The model comprises **~3.67M parameters** distributed across a **12-layer decoder** with **4-head multi-headed self-attention**, **128-dimensional embeddings**, and a **64-token context window**. A professional-grade **Turbo BPE tokenizer** implementing the **GPT-2 regex pre-tokenization pattern** with **~1,037 vocabulary size** achieves efficient compression while maintaining clean token boundaries. Training on an instruction-tuned hybrid dataset demonstrates effective learning of conversational patterns, technical reasoning, and structured response generation. The model achieves a **final training loss of 2.04** and **validation loss of 3.26** after **5,000 iterations**, starting from an initial cross-entropy loss of **8.54**. This work represents a **4.6× parameter scaling** and **3× depth increase** from the original Synapse v1, demonstrating that systematic architectural improvements, professional tokenization, and instruction-aware training enable the transition from basic language modeling to functional assistant capabilities. We emphasize that this implementation prioritizes educational clarity and demonstrates how proper scaling, regularization, and data curation transform model behavior from memorization to generalization.

Keywords: Transformers, Instruction Tuning, Multi-Head Attention, BPE Tokenization, Deep Learning, Neural Networks

---

1. Introduction

The field of large language models has evolved from basic autoregressive text prediction to sophisticated instruction-following systems. While production models like GPT-4 and Claude operate at hundreds of billions of parameters, understanding the fundamental principles requires building scaled systems from first principles. Synapse v2 represents this educational journey—bridging the gap between toy implementations and production architectures.

1.1 Motivation

The original Synapse v1 successfully demonstrated core concepts:

- Self-attention mechanisms

- Byte Pair Encoding tokenization
- Next-token prediction training
- Autoregressive generation

However, it revealed several critical limitations:

1. **Insufficient Scale:** 800K parameters proved too small for learning beyond memorization
2. **Shallow Architecture:** 4 layers limited hierarchical feature learning
3. **Basic Tokenization:** Simple BPE without production-standard pre-tokenization
4. **Overfitting:** Perplexity of 1.05 indicated pure memorization, not generalization
5. **Limited Capability:** Could only continue text, not follow instructions

Synapse v2 addresses these limitations through:

- **Systematic Scaling:** 3.67M parameters (4.6× increase)
- **Architectural Depth:** 12 layers (3× increase) enabling multi-level abstraction
- **Professional Tokenization:** GPT-2 regex-based Turbo BPE
- **Regularization:** Dropout (0.1) preventing overfitting
- **Instruction Tuning:** Hybrid dataset teaching task structure

1.2 Scope and Contributions

This paper makes the following contributions:

Architectural:

- 12-layer decoder-only Transformer with optimized depth-to-width ratio
- Dropout regularization in attention and feed-forward layers
- Pre-layer normalization for training stability
- Learned positional embeddings with 64-token context

Tokenization:

- Production-grade Turbo BPE with GPT-2 regex pattern
- Rank-based encoding for deterministic tokenization
- Doubly-linked list optimization for training efficiency
- Special token support for extensibility

Training:

- Instruction-tuned hybrid dataset (Dolly-15k logic + domain knowledge)
- Stable convergence from loss 8.54 → 2.04 over 5,000 iterations
- Generalization evidence (val loss 3.26 vs train loss 2.04)
- Complete training dynamics documentation

Analysis:

- Comparison of v1 vs v2 architectural decisions
- Ablation insights on dropout and layer depth
- Tokenizer compression analysis
- Sample generations demonstrating capabilities

1.3 Evolution from Synapse v1

Metric	Synapse v1	Synapse v2	Factor
Parameters	800,000	3,672,832	4.6×
Layers	4	12	3×
Vocabulary	1,500 (basic BPE)	1,037 (Turbo BPE)	Professional
Context	128 tokens	64 tokens	Optimized
Regularization	None	Dropout 0.1	Added
Training Loss	0.05	2.04	Better generalization
Validation Loss	Not measured	3.26	Generalization tracking
Perplexity	1.05 (memorization)	7.7 train / 26.1 val	Learned patterns
Training Data	Conversational text	Instruction-tuned hybrid	Task-aware
Capabilities	Text continuation	Instruction following	Functional

The evolution demonstrates that **systematic scaling + regularization + quality data = generalization**.

1.4 Outline

- **Section 2:** System architecture and design decisions
- **Section 3:** Professional Turbo BPE tokenization

- **Section 4:** Enhanced Transformer architecture
- **Section 5:** Instruction-tuned training methodology
- **Section 6:** Experimental results and training dynamics
- **Section 7:** Analysis, ablations, and insights
- **Section 8:** Limitations and lessons learned
- **Section 9:** Future directions
- **Section 10:** Conclusion

---

2. System Architecture Overview

2.1 High-Level Pipeline

Synapse v2 processes text through a refined pipeline optimized for instruction understanding:

Raw Text → Turbo BPE Tokenization (GPT-2 Regex) → Token Embeddings (1037→128)

→ Positional Embeddings (64→128) → Add → 12× Transformer Blocks

→ Final LayerNorm → LM Head (128→1037) → Softmax → Next Token

2.2 Component Specifications

Component	Specification	Justification
Model Dimension (n_embd)	128	Sweet spot for 3–5M parameter range
Attention Heads (n_head)	4	32-dim per head (optimal for 128-dim model)
Head Dimension (d_k)	32	n_embd / n_head = 128 / 4
Transformer Layers (n_layer)	12	Deep enough for hierarchical features
Context Window (block_size)	64	Optimized for dialogue efficiency
Vocabulary Size	1,037	Turbo BPE professional compression
Feed-Forward Hidden	512	4× expansion (standard ratio)
Dropout Rate	0.1	Regularization without over-damping
Total Parameters	**3,672,832**	Scalable on CPU, interpretable

Parameter distribution:

- Token Embeddings: 132,736 (1,037 × 128)
- Position Embeddings: 8,192 (64 × 128)
- Transformer Blocks: ~3,480,000 (12 layers × ~290K per layer)
- Final LayerNorm: 256
- LM Head: 132,736 (128 × 1,037)

2.3 Data Flow

Input: Instruction-tuned text (questions, commands, technical queries)

Processing:

1. **Tokenization:** Apply GPT-2 regex → UTF-8 bytes → BPE merges → token IDs
2. **Embedding:** Map tokens to 128-dim vectors + add 64-position embeddings
3. **Transformer Stack:** Process through 12 blocks of (LayerNorm → Attention → Residual → LayerNorm → FFN → Residual)
4. **Output Head:** Final LayerNorm → Linear projection to vocabulary → Softmax
5. **Sampling:** Select next token via sampling/argmax

Output: Token ID → decode via BPE → UTF-8 text

2.4 Design Philosophy

Why 12 layers?

- Enables hierarchical learning: syntax (lower) → semantics (middle) → reasoning (upper)
- Empirically validated as sweet spot for ~4M parameter models
- Deeper than v1 (4 layers) but shallower than GPT-2 small (12 layers at 117M params)

Why 64 tokens?

- Most conversational turns fit in 50–80 tokens
- Reduces $O(T^2)$ attention cost by 4× vs v1's 128 tokens
- Faster training and inference

Why dropout 0.1?

- Prevents overfitting without over-regularizing
- Standard in Transformer literature (BERT, GPT-2 use 0.1)
- Validation gap (3.26 vs 2.04) shows effective regularization

---

3. Tokenization: Professional Turbo BPE

3.1 Why Professional Tokenization Matters

Tokenization is the **foundation** of language model performance. Poor tokenization leads to:

- Inefficient sequence lengths (higher compute)
- Ambiguous token boundaries (harder learning)
- Poor handling of numbers, punctuation, contractions
- Inconsistent encoding/decoding

Synapse v1's basic BPE worked but lacked production standards. Synapse v2 implements **Turbo BPE** with:

1. **GPT-2 regex pre-tokenization** (industry standard)
2. **Rank-based encoding** (deterministic, production-quality)
3. **Optimized data structures** (doubly-linked lists, hash maps)
4. **Special token support** (extensible for future features)

3.2 GPT-2 Regex Pre-Tokenization Pattern

The core innovation is the pre-tokenization regex (used by GPT-2, GPT-3, GPT-4, Llama):

regex

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

Pattern breakdown:

Pattern	Matches	Example
's|'t|'re|'ve|'m|'ll|'d	Contractions	don't → don + 't
?\p{L}+	Optional space + letters	" the" vs "the"
?\p{N}+	Optional space + numbers	" 123" vs "123"
?[^\s\p{L}\p{N}]+	Optional space + punctuation	" !" vs "!"
\s+(?!\S)|\s+	Whitespace handling	Multiple spaces

Why this matters:

python

 # Without regex:

"don't" → ['d', 'o', 'n', "'", 't'] *# 5 tokens, loses structure*

 # With GPT-2 regex:

"don't" → ['don', "'t"] *# 2 tokens, preserves contraction*

 # Number handling:

"AI100" → ['AI', '100'] *# Prevents letter-number merge*

 # Whitespace tracking:

"the cat" → [' the', ' cat'] *# Leading space preserved*

```

This ensures consistent tokenization regardless of context.

*### 3.3 Turbo BPE Algorithm (Optimized)*

**Training process:**

```

1. Pre-tokenize corpus using GPT-2 regex

2. Convert each chunk to UTF-8 bytes (0-255)

3. Build doubly-linked list:

  - prev_ptrs[i] = index of previous token

  - next_ptrs[i] = index of next token

  - values[i] = token ID at position i

4. Initialize pair statistics:

  - pair_pos[(t1, t2)] = set of positions where pair appears

5. For (target_vocab_size - 256) iterations:
   a. Find most frequent pair: max(pair_pos, key=len)
   b. Create new token: new_id = len(vocab)
   c. Record merge: merges[(t1, t2)] = new_id
   d. Update occurrences:
      - Remove old pairs from statistics
      - Merge tokens in linked list (O(1) updates)
      - Add new pairs from neighbors
   e. vocab[new_id] = vocab[t1] + vocab[t2]

6. Save: merges, vocab, special_tokens
```

**Key optimizations:**

| Optimization | Benefit |
|---|---|
| Doubly-linked list | O(1) merge updates instead of O(n) list rebuilding |
| Hash map for pairs | O(1) pair lookup instead of O(n) scanning |
| Position sets | Track all occurrences efficiently |
| Chunk boundaries | Prevent merging across pre-tokenized chunks |

**Example training:**
```
Input: "Hello Hello"

Iteration 0: [72, 101, 108, 108, 111, 32, 72, 101, 108, 108, 111]
             (UTF-8 bytes of "Hello Hello")

Pair frequencies: (108, 108)=2, (101, 108)=2, (72, 101)=2, ...

Iteration 1: Merge (108, 108) → 256

[72, 101, 256, 111, 32, 72, 101, 256, 111]

Iteration 2: Merge (101, 256) → 257

[72, 257, 111, 32, 72, 257, 111]

... continue to vocab_size = 1,037

```

*### 3.4 Rank-Based Encoding (Production Standard)*

**Problem with frequency-based encoding:**

```

Text: "AAA"

Merges: (A,A)→X, (A,X)→Y, (X,A)→Z

Frequency-based encoding is ambiguous:

- Could encode as [A, X, A] then [A, Z]

- Could encode as [X, A, A] then [Z, A]

- Result depends on scan direction!

Solution: Rank-based encoding

python

def encode(text):

    chunks = regex.findall(GPT2_PATTERN, text)

    all_tokens = []

```
    for chunk in chunks:
        chunk_ids = list(chunk.encode("utf-8"))

        while len(chunk_ids) >= 2:
            # Find all pairs that exist in learned merges
            stats = {
                pair: merges[pair]
                for pair in zip(chunk_ids, chunk_ids[1:])
                if pair in merges
            }

            if not stats: break

            # CRITICAL: Merge pair with LOWEST rank (earliest learned)
            best_pair = min(stats, key=stats.get)
            chunk_ids = merge_chunk(chunk_ids, best_pair, merges[best_pair])

        all_tokens.extend(chunk_ids)
    return all_tokens
```

Why lowest rank?

- Merges are assigned ranks during training (0, 1, 2, ...)
- Earliest-learned merges (lowest rank) take priority
- This matches GPT-2/GPT-4 behavior
- Encoding becomes **deterministic** and **reversible**

3.5 Vocabulary Analysis

Learned merge examples:

python

 # Common English morphology

Merge 15: (32, 116) → 271 *# " t" (the, that, to)*

Merge 28: (101, 114) → 284 *# "er" (after, under, parameter)*

Merge 42: (105, 110) → 298 *# "in" (in, training, embedding)*

Merge 67: (97, 116) → 323 *# "at" (at, attention, data)*

 # Technical vocabulary

Merge 89: (116, 111) → 345 *# "to" (tokenizer, token)*

Merge 134: (109, 111) → 390 *# "mo" (model, more)*

Merge 201: (108, 97) → 457 *# "la" (layer, language)*

 # Conversational markers

Merge 12: (32, 72) → 268 *# " H" (Hello, Hi, How)*

Merge 45: (63, 32) → 301 *# "? " (question marker)*

Merge 78: (33, 32) → 334 *# "! " (exclamation marker)*

Compression statistics:

- Raw characters: Variable (depends on text)
- Average compression: **3–4× reduction** in sequence length
- Example: "Hello, how are you?"
    - Characters: 19
    - Tokens (Turbo BPE): ~5–6
    - Compression: ~3.2–3.8×

Special tokens:

python

```
special_tokens = {
   "<|endoftext|>": 100000  # Document separator (future scaling)
}
```

This enables future extensions:

- Multiple document training
- Task-specific control codes
- Mixture-of-experts routing
- Instruction prefixes

---

4. Enhanced Transformer Architecture

4.1 Embedding Layers

Token Embedding:

python

self.token_embedding_table = nn.Embedding(vocab_size=1037, n_embd=128)

token_emb = self.token_embedding_table(idx) *# [B, T, 128]*

Maps discrete token IDs to continuous 128-dimensional vectors. These embeddings are **learned** during training to capture semantic relationships.

Positional Embedding:

python

self.position_embedding_table = nn.Embedding(block_size=64, n_embd=128)

pos_emb = self.position_embedding_table(torch.arange(T, device=device)) *# [T, 128]*

Injects sequence order information. Transformer attention is **permutation-invariant** (doesn't inherently know order), so positional embeddings are critical for understanding "The cat sat" vs "sat the cat".

Combined representation:

python

x = token_emb + pos_emb *# [B, T, 128]*

Element-wise addition combines "what" (token) with "where" (position).

4.2 Single Attention Head (Core Mechanism)

python

```
class Head(nn.Module):
    def __init__(self, head_size, n_embd, block_size, dropout):
        super().__init__()
```

```
    self.key = nn.Linear(n_embd, head_size, bias=False)
    self.query = nn.Linear(n_embd, head_size, bias=False)
    self.value = nn.Linear(n_embd, head_size, bias=False)

    # Causal mask (lower triangular)
    self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
    self.dropout = nn.Dropout(dropout)

def forward(self, x):
    B, T, C = x.shape  # Batch, Time, Channels

    # Project to Q, K, V
    k = self.key(x)    # [B, T, head_size=32]
    q = self.query(x)  # [B, T, 32]
    v = self.value(x)  # [B, T, 32]

    # Scaled dot-product attention
    wei = q @ k.transpose(-2, -1) * (C ** -0.5)  # [B, T, T]

    # Apply causal mask (prevent attending to future)
    wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))

    # Softmax → attention weights
    wei = F.softmax(wei, dim=-1)  # [B, T, T]
    wei = self.dropout(wei)  # Regularization

    # Weighted sum of values
    out = wei @ v  # [B, T, 32]
```

return out

```

**Mathematical formulation:**

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

Where:

- $Q, K, V$ = Query, Key, Value matrices
- $d_k$ = head dimension (32)
- $M$ = causal mask (lower triangular)
- Scaling by $\sqrt{d_k}$ prevents softmax saturation

**Causal mask visualization (T=4):**

```
[[1, 0, 0, 0],   Token 0 can only attend to itself
 [1, 1, 0, 0],   Token 1 attends to 0, 1
 [1, 1, 1, 0],   Token 2 attends to 0, 1, 2
 [1, 1, 1, 1]]   Token 3 attends to all previous
```

After masking: 0 → -inf → softmax → 0 probability

This ensures **autoregressive** generation: future tokens don't leak information to past.

**4.3 Multi-Head Attention**

python

```
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, head_size, n_embd, block_size, dropout):
        super().__init__()
        self.heads = nn.ModuleList([
```
```

```
            Head(head_size, n_embd, block_size, dropout)
            for _ in range(num_heads)
        ])
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # Run 4 heads in parallel
        out = torch.cat([h(x) for h in self.heads], dim=-1)  # [B, T, 128]

        # Output projection + dropout
        out = self.dropout(self.proj(out))
        return out
```

Why multiple heads?

- Each head learns different attention patterns:
    - Head 1: Syntactic dependencies (subject-verb)
    - Head 2: Semantic relationships (entity-attribute)
    - Head 3: Positional patterns (beginning/end of sentence)
    - Head 4: Long-range dependencies (pronouns to antecedents)

Computational efficiency:

- 4 heads × 32 dims = 128 total (same as single 128-dim head)
- Cost: Same as single head (just rearranged computation)
- Benefit: Richer representations through independent subspaces

4.4 Feed-Forward Network

python

```
class FeedForward(nn.Module):
    def __init__(self, n_embd, dropout):
        super().__init__()
```

```
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),  # 128 → 512
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),  # 512 → 128
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)
```

Design rationale:

- **4× expansion:** Standard in Transformer literature (GPT-2, BERT)
- **ReLU activation:** Introduces non-linearity (allows learning complex functions)
- **Position-wise:** Applied independently to each token (no cross-sequence interaction)
- **Dropout:** Regularization on output

Role in architecture:

- Attention gathers information (communication)
- FFN processes information (computation)
- Together: "Communicate then compute"

4.5 Complete Transformer Block

python

```
class Block(nn.Module):
    def __init__(self, n_embd, n_head, block_size, dropout):
        super().__init__()
        head_size = n_embd // n_head  # 128 // 4 = 32

        self.sa = MultiHeadAttention(n_head, head_size, n_embd, block_size, dropout)
        self.ffwd = FeedForward(n_embd, dropout)
        self.ln1 = nn.LayerNorm(n_embd)
```

```
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        # Pre-norm architecture (more stable than post-norm)
        x = x + self.sa(self.ln1(x))      # Attention path + residual
        x = x + self.ffwd(self.ln2(x))    # FFN path + residual
        return x
```

Visual flow:

```
Input [B, T, 128]
  ↓
LayerNorm (normalize activations)
  ↓
Multi-Head Attention (gather information)
  ↓
Dropout + Residual Add (x + attn_out)
  ↓
LayerNorm
  ↓
Feed-Forward (process information)
  ↓
Dropout + Residual Add (x + ffn_out)
  ↓
Output [B, T, 128]
```

Why pre-norm?

- More stable training (gradients don't explode/vanish as easily)

- Allows higher learning rates
- Standard in modern architectures (GPT-3, Claude)

Residual connections:

y=x+sublayer(x)*y*=*x*+sublayer(*x*)

Benefits:

- Gradient highway during backpropagation
- Enables training of very deep networks (50+ layers)
- Original ResNet innovation (He et al., 2015)

4.6 Complete Model

python

```
class TinyGPT(nn.Module):
    def __init__(self, vocab_size=1037, n_embd=128, n_head=4,
                 n_layer=12, block_size=64, dropout=0.1, device='cpu'):
        super().__init__()
        self.block_size = block_size
        self.device = device

        # Embedding layers
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)

        # Stack 12 Transformer blocks
        self.blocks = nn.Sequential(*[
            Block(n_embd, n_head, block_size, dropout)
            for _ in range(n_layer)
        ])

        # Output layers
```

```
        self.ln_f = nn.LayerNorm(n_embd)  # Final normalization
        self.lm_head = nn.Linear(n_embd, vocab_size)  # Project to vocabulary

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # Embeddings
        tok_emb = self.token_embedding_table(idx)  # [B, T, 128]
        pos_emb = self.position_embedding_table(torch.arange(T, device=self.device))  # [T, 128]
        x = tok_emb + pos_emb  # [B, T, 128]

        # Transformer blocks
        x = self.blocks(x)  # [B, T, 128]

        # Output
        x = self.ln_f(x)  # [B, T, 128]
        logits = self.lm_head(x)  # [B, T, vocab_size]

        # Loss computation (if targets provided)
        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)
```

```
        return logits, loss
```

Parameter count:

Component	Shape	Parameters
Token Embedding	[1037, 128]	132,736
Position Embedding	[64, 128]	8,192
Block (×12)	Various	~290,000 each
- Self-Attention	[128, 32]×3×4 + [128,128]	~49,000
- Feed-Forward	[128,512] + [512,128]	~131,000
- LayerNorms	128×2×2	512
Final LayerNorm	[128]×2	256
LM Head	[128, 1037]	132,736
Total		**3,672,832**

5. Instruction-Tuned Training Methodology

5.1 Dataset Composition

Unlike v1's simple conversational data, v2 uses a **hybrid instruction-tuned corpus**:

Component	Size	Purpose	Example

| **Dolly-15k** | ~15K instructions | Reasoning, QA, summarization | "Explain photosynthesis in simple terms" |

| **Creator Profile** | ~500 examples | Personal identification | "Who created you?" → "Abhinav Tyagi" |

| **GenAI Knowledge** | ~2K examples | Technical expertise | "What is QLoRA?" → Technical explanation |

| **Domain Facts** | ~1K examples | Grounded knowledge | "Capital of India?" → "New Delhi" |

Total estimate: ~18–20K instruction-response pairs, ~200–500K tokens after BPE encoding

Data format:

```
Instruction: What is a Transformer model?

Response: A Transformer is a neural network architecture that uses self-attention mechanisms to process sequences in parallel, unlike RNNs which process sequentially.

Instruction: Who created Synapse?

Response: Synapse was created by Abhinav Tyagi.

Instruction: Explain QLoRA

Response: QLoRA (Quantized Low-Rank Adaptation) is a technique for efficiently fine-tuning large language models using 4-bit quantization combined with LoRA adapters.
```

5.2 Training Objective

Next-token prediction via cross-entropy loss:

L=−1NT∑i=1N∑t=1Tlog⁡P(xt+1(i)|x1(i),…,xt(i))*L*=−*NT*1*i*=1∑*Nt*=1∑*T*log*P*(*xt*+1(*i*)|*x*1(*i*),…,*xt*(*i*))

Where:

- N*N* = batch size
- T*T* = sequence length

- x_t = token at position t
- $P(\cdot)$ = model's predicted probability distribution (softmax over vocabulary)

Implementation:

python

```
# Forward pass
logits, loss = model(input_ids, targets)


# logits: [B, T, vocab_size] - raw scores
# targets: [B, T] - ground truth next tokens


# Cross-entropy loss
loss = F.cross_entropy(
    logits.view(-1, vocab_size),  # Flatten to [B*T, vocab_size]
    targets.view(-1)              # Flatten to [B*T]
)
```

5.3 Training Configuration

Hyperparameters:

Parameter	**Value**	**Rationale**
Optimizer	Adam	Standard for Transformers; adaptive learning rates
Learning Rate	1e-3 (0.001)	Stable for Adam; allows fast convergence
Beta 1	0.9	Momentum term (standard)
Beta 2	0.999	Second moment (standard)
Epsilon	1e-8	Numerical stability
Batch Size	4	Memory-efficient for CPU training
Context Length	64 tokens	Optimized for dialogue
Dropout	0.1	Regularization
Training Steps	5,000	Observe