Quadtrix

Deep-Dive into Transformer Architecture, Backpropagation, and Language Modelling Without Frameworks

Eamon

https://github.com/Eamon2009/Quadtrix.cpp

Technical Educational Report · 2026

Abstract

This paper presents Quadtrix, a complete C++17 implementation of a GPT-2-style decoder-only transformer language model built entirely from first principles, with zero external dependencies beyond the C++ standard library. Rather than serving purely as a software contribution, this document is intended as a comprehensive educational resource for practitioners who wish to understand the mathematics, intuitions, and engineering decisions behind modern large language models. We derive every component of the transformer architecture from first principles: token and positional embeddings, multi-head causal self-attention, feed-forward networks, layer normalisation, dropout, cross-entropy loss, and the complete analytical backpropagation pass. We also explain the AdamW optimiser, present training curves from a real 76-minute CPU run that achieves a validation loss of 1.6371 nats on a 31.4M-character corpus, and discuss the Chinchilla scaling laws that govern optimal model size and data volume. The paper is structured to be readable by anyone familiar with calculus and basic linear algebra, yet rigorous enough to serve as a reference implementation guide.

Keywords: transformer, language model, C++, backpropagation, attention, character-level, GPT, AdamW, scaling laws, educational

1. Introduction

Language models are the backbone of modern artificial intelligence. From chatbots to code assistants to scientific reasoning tools, virtually every frontier AI system today is built on the transformer architecture introduced by Vaswani et al. in 2017. Yet the dominant way developers interact with these models is through high-level frameworks — PyTorch, JAX, TensorFlow — that abstract away the mathematics that make them work. The result is a generation of practitioners who can fine-tune GPT-4 but cannot explain why gradient descent converges, or what exactly backpropagation computes through a softmax.

Quadtrix takes the opposite approach. Every operation — matrix multiplication, causal masking, layer normalisation, softmax, cross-entropy — is written as a plain C++ function. Every gradient is derived analytically and coded by hand. No automatic differentiation. No compute graph. Just mathematics translated into code.

This paper is the companion document. It walks through each component of the transformer in enough depth that a reader who understands calculus and linear algebra could implement it themselves. We ground every formula in intuition before formalising it, and we connect theory to the actual C++ code in Quadtrix.

1.1 What Is a Language Model?

A language model is a probability distribution over sequences of tokens (characters, subwords, or words). Given a sequence $x_1, x_2, ..., x_t$, a language model computes:

$$P(x_1, ..., x_T) = P(x_1) \cdot P(x_2 \mid x_1) \cdot P(x_3 \mid x_1, x_2) \cdot ... \cdot P(x_T \mid x_1, ..., x_{T-1})$$

This factorisation, known as the *chain rule of probability*, tells us that modelling a sequence is equivalent to modelling each next token given all previous tokens. The transformer models these conditional distributions using a deep neural network that processes the entire prefix simultaneously via attention mechanisms, rather than sequentially as in RNNs.

In Quadtrix, tokens are individual characters. The vocabulary is the set of all unique characters in the training corpus (105 characters for our TinyStories run). At each position, the model outputs a probability distribution over all 105 possible next characters.

1.2 Scope of This Paper

We cover the following topics in depth:

- Token and positional embeddings — how discrete tokens become continuous vectors
- Multi-head causal self-attention — the core mechanism of the transformer
- Feed-forward networks — the "memory" layers between attention blocks
- Layer normalisation — stabilising deep network training
- Dropout — preventing overfitting by random deactivation
- Cross-entropy loss — measuring prediction quality
- Analytical backpropagation — computing gradients without autograd
- AdamW optimiser — adaptive gradient descent with weight decay
- Chinchilla scaling laws — optimal model size vs. data volume
- Real training results and diagnostic analysis

2. Embeddings: From Tokens to Vectors

Neural networks operate on real-valued vectors, but language consists of discrete symbols. Embeddings are the bridge: learned lookup tables that map each discrete token to a dense vector in a high-dimensional space. The key insight is that meaningful geometric relationships emerge during training — similar tokens cluster together, and arithmetic on embedding vectors encodes semantic relationships.

2.1 Token Embeddings

The token embedding table $\mathbf{E}_{tok} \in \blacksquare^{V \times C}$ maps each token index $i \in \{0, ..., V-1\}$ to a vector of dimension C (the embedding dimension, also called d_model). This is simply a matrix lookup:

$$e_{tok}(i) = E_{tok}[i, :] \in \blacksquare^{C}$$

During training, these embeddings are updated by gradient descent. The gradient flows back through every forward pass that used token i, and the embedding vector gradually shifts to encode the statistical context in which token i appears. After training on English text, the letter "a" embedding will be close to "e" and "i" (both vowels), and far from "z" or ".".

In Quadtrix, V = 105 (unique characters in the corpus) and C = 128 (embedding dimension). The embedding table contains 105 × 128 = 13,440 parameters.

2.2 Positional Embeddings

Self-attention (described in Section 3) is permutation-invariant: it treats a sequence as a *set*, not an ordered list. If we shuffled the positions of the tokens, the attention output would be the same (just rearranged). This means the model has no inherent sense of order without additional information.

Positional embeddings inject order information. Quadtrix uses *learned* positional embeddings: a separate table $E_{pos} \in \blacksquare^{T \times C}$, where T is the maximum context length (block size). Position t gets embedding $E_{pos}[t, :]$.

The input representation at each position is the sum of the token and positional embeddings:

$$x_t = E_{tok}[i_t] + E_{pos}[t] \quad x_t \in \blacksquare^{C}$$

Why addition rather than concatenation? Empirically, addition works as well as concatenation and uses fewer parameters. Both embeddings live in the same C-dimensional space, and addition can be interpreted as the token embedding being "shifted" by a position-specific offset.

An alternative, used in the original 2017 transformer paper, is sinusoidal positional encodings: fixed functions of position and dimension index, using sine and cosine at geometrically spaced frequencies. These generalise to unseen sequence lengths but underperform learned embeddings on most benchmarks. GPT-2 and its descendants use learned positional embeddings, as does Quadtrix.

2.3 Initialisation

Both embedding tables are initialised from N(0, 0.02) — a Gaussian with zero mean and standard deviation 0.02. This small initialisation ensures that the initial representations are near zero but not identical, breaking symmetry while avoiding large initial activations that could cause exploding gradients early in training.

Module	Shape	Parameters	Role
Token Embedding	`105 × 128`	13,440	Maps character index → dense vector
Positional Embedding	`64 × 128`	8,192	Maps position index → positional shift
Total Embeddings	—	21,632	2.6% of 826,985 total parameters

Table 1. Embedding table dimensions and parameter counts for the Quadtrix v1.0 configuration.

3. Multi-Head Causal Self-Attention

Attention is the core innovation of the transformer. It allows every position in a sequence to directly access information from every other position, without the information having to propagate through many intermediate steps (as in RNNs). This makes long-range dependencies tractable and allows massive parallelisation.

3.1 The Attention Mechanism Intuitively

Imagine you are reading the sentence: *"The animal didn't cross the street because it was too tired."* When processing "it", your brain immediately connects it to "animal" (not "street"). Attention is a mechanism that learns to perform exactly this kind of selective retrieval.

Each token produces three vectors from its embedding: a **Query** (Q — "what am I looking for?"), a **Key** (K — "what do I contain?"), and a **Value** (V — "what should I send if selected?"). The attention score between position i and position j is the dot product of Q_i and K_j. High dot product means high relevance. The scores are normalised via softmax to produce weights that sum to 1, then used to take a weighted average of the values.

3.2 Scaled Dot-Product Attention

Formally, given $Q \in \blacksquare^{T \times h}$, $K \in \blacksquare^{T \times h}$, $V \in \blacksquare^{T \times h}$ (where h is the head dimension), scaled dot-product attention computes:

$$\text{Attention}(Q, K, V) = \text{softmax}\left((Q K^T) / \sqrt{h} + M \right) \cdot V$$

The three terms deserve individual explanation:

- **QK^T / √h**: The matrix of pairwise similarity scores. Dividing by √h prevents the dot products from growing large in magnitude as h increases, which would push the softmax into regions of near-zero gradient (the "vanishing gradient through softmax" problem).
- **M (causal mask)**: A matrix where M[i,j] = -∞ for j > i and 0 otherwise. Adding -∞ before softmax drives exp(-∞) = 0, effectively preventing position i from attending to any future position j > i. This causality constraint is what makes autoregressive generation possible.
- **Softmax and V multiplication**: The softmax converts scores to probabilities (attention weights). Multiplying by V produces a weighted average of value vectors, where positions with high attention weights contribute more.

3.3 Why Multi-Head?

A single attention head computes one "type" of relevance — perhaps subject-verb agreement, or coreference. Multi-head attention runs H parallel attention heads, each with independent projection matrices $W_Q^{(h)}$, $W_K^{(h)}$, $W_V^{(h)} \in \blacksquare^{C \times (C/H)}$. Each head learns a different notion of relevance. Their outputs are concatenated and projected back to dimension C:

$$\text{MultiHead}(x) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)\, W_0$$

Where $\text{head}_h = \text{Attention}(x\, W_Q^h, x\, W_K^h, x\, W_V^h)$.

In Quadtrix: C = 128, H = 4 heads, h = C/H = 32 per head. The four heads can simultaneously track different types of dependencies in the text.

3.4 Parameter Count for Attention

Component	Shape	Per Block	4 Blocks

W_Q (per head)	`128 × 32`	4 × 4,096 = 16,384	65,536
W_K (per head)	`128 × 32`	4 × 4,096 = 16,384	65,536
W_V (per head)	`128 × 32`	4 × 4,096 = 16,384	65,536
W_O (projection)	`128 × 128`	16,384	65,536
Attention Total	—	**65,536**	**262,144**

Table 2. Parameter counts for multi-head attention modules. These account for 31.7% of the 826,985 total parameters.

3.5 Computational Complexity

The computational cost of self-attention scales as $O(T^2 \cdot C)$ per layer — quadratic in sequence length T. For small T (such as our T = 64), this is fine. For very long sequences (T = 32,768 in GPT-4), specialised attention variants (FlashAttention, Sparse Attention) are required to keep computation tractable. This quadratic scaling is the primary bottleneck in scaling transformers to longer contexts.

4. Position-Wise Feed-Forward Networks

After each multi-head attention layer, a position-wise feed-forward network (FFN) is applied independently to each position. The FFN is sometimes described as the "memory" of the transformer — research suggests it stores factual associations (e.g., "the capital of France is Paris") that attention retrieves and routes.

4.1 Architecture

The FFN consists of two linear transformations with a nonlinear activation function between them. For input x at a single position:

$$\mathrm{FFN}(x) = \mathrm{ReLU}(x\,W_1 + b_1)\,W_2 + b_2$$

Where $W_1 \in \blacksquare^{C \times 4C}$ expands the representation to 4× the embedding dimension, and $W_2 \in \blacksquare^{4C \times C}$ projects it back. The factor of 4 is a design choice from the original transformer paper, empirically validated as effective across a wide range of model sizes.

For Quadtrix: C = 128, so the inner dimension is 4C = 512. Each FFN layer contains (128 × 512) + (512 × 128) + bias terms = 131,584 parameters. Across 4 layers, that is 526,336 parameters — 63.5% of the total model.

4.2 Why ReLU?

The Rectified Linear Unit (ReLU) is defined as ReLU(x) = max(0, x). It is computationally cheap, produces exactly-zero outputs for negative inputs (creating *sparse* activations), and avoids the vanishing gradient problem of sigmoid/tanh for large positive inputs.

The original GPT-2 paper uses GELU (Gaussian Error Linear Unit), which is a smoother approximation of ReLU. GELU tends to perform marginally better on language modelling tasks. Quadtrix uses ReLU for simplicity and verifiability of the backward pass; the roadmap includes a GELU upgrade.

4.3 Position-Wise Operation

The FFN is applied independently at each sequence position — the same W_1 and W_2 matrices are applied to every position, but positions do not interact within the FFN. This is why it is called "position-wise". The interaction between positions happens exclusively in the attention layers.

This design gives the transformer a clean separation of concerns: attention routes information between positions, while FFNs process information within each position. This modularity is part of why transformers are so parallelisable: FFN computations at different positions can run in parallel, as can attention computations in different heads.

5. Layer Normalisation

Deep networks suffer from internal covariate shift: as the network trains, the distribution of each layer's inputs shifts, forcing subsequent layers to continually adapt. Normalisation layers counteract this by standardising activations, which stabilises and accelerates training.

5.1 The Layer Norm Operation

Given an input vector $x \in \blacksquare^C$ (one position's representation), layer normalisation computes:

$$LN(x) = \gamma \blacksquare ((x - \mu) / (\sigma + \varepsilon)) + \beta$$

Where:

- μ = **mean(x)**: the mean across the C features at this position
- σ = **std(x)**: the standard deviation across the C features at this position
- $\varepsilon \approx \mathbf{10^{-5}}$: a small constant for numerical stability (prevents division by zero)
- $\gamma, \beta \in \blacksquare^{\mathbf{C}}$: learned scale and bias parameters, initialised to 1 and 0 respectively

The key difference from Batch Normalisation (used in CNNs) is that LN normalises across the *feature* dimension, not the *batch* dimension. This makes LN independent of batch size and sequence length, making it suitable for variable-length sequences and small batches — both common in language modelling.

5.2 Pre-LN vs. Post-LN

The original 2017 transformer applied layer norm *after* the residual connection (Post-LN). GPT-2 and later models apply it *before* the sublayer (Pre-LN):

$$x' = x + Sublayer(LN(x))$$

Pre-LN significantly stabilises gradient flow at initialisation, because the gradient through the residual path is always 1 (the identity), and the sublayer gradient is applied to already-normalised inputs. This allows training without careful learning rate warmup. Quadtrix uses Pre-LN throughout.

6. Residual Connections and the Transformer Block

Modern deep networks are made possible by residual connections (also called skip connections), introduced by He et al. in ResNet (2015). Without them, networks deeper than ~20 layers became essentially impossible to train due to vanishing gradients.

6.1 The Residual Principle

Instead of learning a direct transformation F(x), each sublayer learns a residual mapping: its output is F(x) + x. The gradient of the loss with respect to x becomes:

$$\partial L/\partial x = \partial L/\partial(F(x)+x) = \partial L/\partial F(x) \cdot \partial F/\partial x + \partial L/\partial(F(x)+x)$$

The second term — the "shortcut" gradient — is always present regardless of how small ∂F/∂x becomes. This prevents gradient vanishing in deep networks. In GPT models with dozens of layers, residual connections are what makes training feasible.

6.2 The Complete Transformer Block

Each transformer block applies two sublayers with Pre-LN and residual connections:

$$x' = x + \mathrm{MHA}(\mathrm{LN}_1(x))$$

$$x'' = x' + \mathrm{FFN}(\mathrm{LN}_2(x'))$$

Quadtrix stacks L = 4 such blocks. The output of block L is passed through a final layer norm, then projected to vocabulary logits by the LM head linear layer. The full forward pass is:

Step	Operation	Output Shape
1	Token + Position Embeddings	`B × T × C`
2–9	4 × Transformer Block (LN + MHA + LN + FFN)	`B × T × C`
10	Final LayerNorm	`B × T × C`
11	LM Head Linear (C → V)	`B × T × V`
12	Cross-Entropy Loss (training) / Softmax Sample (inference)	`scalar / token`

Table 3. Full forward pass of Quadtrix GPTLanguageModel. B=batch, T=sequence length, C=embedding dim, V=vocab size.

7. Dropout Regularisation

Dropout is a simple but highly effective regularisation technique. During training, each activation is independently zeroed with probability p (the dropout rate), and the surviving activations are scaled by 1/(1-p) to maintain the expected sum.

7.1 Why It Works

Dropout can be interpreted in two complementary ways:

- **Ensemble interpretation**: Each training step trains a different thinned network (a sub-network of the full model). At test time, the full network acts as an ensemble of all these sub-networks.
- **Co-adaptation prevention**: Neurons cannot rely on specific other neurons always being present, so they are forced to learn independently useful features rather than co-adapting to mask each other's errors.

In Quadtrix, dropout with p = 0.2 is applied to the attention weights (after softmax, before multiplying by V) and to the output of the FFN projection. This means 20% of attention connections and FFN outputs are silenced per training step.

7.2 Train vs. Inference Behaviour

Dropout is active only during training. At inference time, all activations are preserved. This is why the training loss is typically slightly higher than the validation loss (when measured at the same number of gradient steps): training loss is computed with dropout active, while validation loss is computed without dropout.

The final checkpoint of the Quadtrix training run shows a validation loss (1.6371) slightly *lower* than the training loss (1.7055) — a 0.0684 gap. This is normal and expected, and is entirely explained by dropout being disabled at evaluation time.

8. Cross-Entropy Loss

The training objective of a language model is to maximise the probability assigned to the correct next token at every position. Equivalently, we minimise the negative log-probability — the cross-entropy loss.

8.1 Derivation

Given logits $z \in \blacksquare^V$ (the raw output before softmax) at position t, and ground-truth token index y (an integer), the cross-entropy loss is:

$$L = -\log\left(\exp(z_y) / \Sigma_c \exp(z_c) \right) = -z_y + \log(\Sigma_c \exp(z_c))$$

The first term rewards high logit for the correct class. The second term (log-sum-exp) penalises high logits for incorrect classes. The loss is minimised when all probability mass is on the correct token (L = 0 in theory, though never achieved in practice due to genuine uncertainty in language).

In practice, we average this loss over all positions $t \in \{1, ..., T\}$ and all examples in the batch B:

$$L_{total} = (1 / B \cdot T) \cdot \Sigma_{b,t} \, CE(z_{b,t}, y_{b,t})$$

8.2 Interpretation via Perplexity

The exponential of cross-entropy is called **perplexity**:

$$Perplexity = \exp(L)$$

Perplexity can be interpreted as the "effective vocabulary size" the model considers at each step. A perplexity of 5 means the model is about as uncertain as if choosing uniformly among 5 options. For our final validation loss of 1.6371 nats, the perplexity is $\exp(1.6371) \approx 5.14$. Given a vocabulary of 105 characters, this represents genuine structural understanding — a random model would have perplexity 105.

8.3 Cross-Entropy Backward Pass

The gradient of cross-entropy loss with respect to logit z_c is one of the cleanest results in deep learning:

$$\partial L / \partial z_c = (\text{softmax}(z))_c - 1[c = y]$$

In words: the gradient is the softmax probability at class c, minus 1 if c is the correct class and minus 0 otherwise. This makes intuitive sense: if the model assigns high probability to the wrong class (say class k), then $\text{softmax}(z)_k$ is large and positive, creating a large gradient pushing z_k down. If the model assigns low probability to the correct class y, then $\text{softmax}(z)_y$ is close to 0, and 0 - 1 = -1 creates a strong gradient pushing z_y up.

In Quadtrix, this is implemented analytically in backward.h — there is no automatic differentiation. The beauty of this formulation is that the softmax and cross-entropy gradients combine so elegantly.

9. Analytical Backpropagation

Backpropagation is the algorithm for computing gradients of the loss with respect to all parameters, using the chain rule of calculus. Frameworks like PyTorch do this automatically via reverse-mode automatic differentiation. Quadtrix does it analytically — every gradient formula is derived by hand and coded explicitly.

9.1 The Chain Rule

The chain rule states that for a composition of functions L(f(g(x))), the gradient with respect to x is:

$$dL/dx = (dL/df) \cdot (df/dg) \cdot (dg/dx)$$

In a neural network, the "x" might be a weight matrix W deep in the network. The chain rule says we multiply together all the Jacobians of intermediate computations, from the loss back to W. This is computed efficiently by starting at the loss and propagating backwards through the computation graph — hence "backpropagation".

9.2 Key Gradient Derivations

Linear Layer (y = xW + b)

Given upstream gradient ∂L/∂y, the gradients are:

- $\partial L/\partial x = (\partial L/\partial y)\, W^T$ (gradient w.r.t. input)
- $\partial L/\partial W = x^T (\partial L/\partial y)$ (gradient w.r.t. weights, accumulated across batch/time)
- $\partial L/\partial b = \text{sum}(\partial L/\partial y$, over batch and time dimensions)

Softmax

For softmax $s_i = \exp(z_i) / \Sigma_j \exp(z_j)$, given upstream gradient d, the downstream gradient is:

$$\partial L/\partial z_i = s_i \cdot (d_i - \Sigma_j\, s_j\, d_j)$$

This is the "Jacobian-vector product" of the softmax. Note that the softmax Jacobian is a full matrix (each output depends on all inputs), but the JVP formula collapses it to a single O(V) computation using the identity: $\partial s_i/\partial z_j = s_i(\delta_{ij} - s_j)$.

ReLU

ReLU has a trivially simple gradient — the "pass-through" or gate function:

$$\partial L/\partial x_i = d_i \cdot 1[x_i > 0]$$

Where $1[x_i > 0]$ is 1 if the pre-ReLU input was positive, 0 otherwise. This is why "dead ReLU" neurons (where the pre-activation is always negative) are a training hazard — they permanently block gradient flow.

Layer Normalisation

The layer norm backward is the most algebraically complex in the network. For $LN(x) = \gamma \blacksquare\, x\blacksquare + \beta$, where $x\blacksquare = (x - \mu)/\sigma$, the gradient is:

$$\partial L/\partial x_c = (\sigma^{-1}/C) \cdot [\, C\cdot\gamma_c\cdot d_c - \Sigma_j\gamma_j d_j - x\blacksquare_c\cdot\Sigma_j\gamma_j d_j x\blacksquare_j \,]$$

This three-term formula accounts for the coupling between positions through the shared mean μ and standard deviation σ. Quadtrix implements this exactly in backward.h, verified against the derivation in Ba et al. (2016).

9.3 Gradient Flow Through the Full Network

The backward pass through the full transformer proceeds as follows:

- 1. Compute ∂L/∂logits from cross-entropy backward (Section 8.3)
- 2. Backprop through LM head linear layer
- 3. Backprop through final layer norm
- 4. For each block from last to first: backprop through FFN residual, then FFN, then MHA residual, then MHA
- 5. Accumulate gradients into token and position embedding tables

The residual connections simplify gradient flow: $\partial L/\partial x = \partial L/\partial(x + F(x)) = \partial L/\partial F(x) + \partial L/\partial x$, so the identity path always contributes a direct gradient regardless of F.

10. The AdamW Optimiser

Gradient descent updates parameters in the direction that most reduces the loss. Vanilla gradient descent, however, uses the same learning rate for all parameters regardless of their gradient history, and updates can oscillate in narrow valleys of the loss landscape. Adam (Adaptive Moment Estimation) addresses this by maintaining running averages of gradients and squared gradients.

10.1 Adam Update Rule

For each parameter θ with gradient g at step t:

- $\mathbf{m_t = \beta_1 m_{t-1} + (1-\beta_1) g}$: first moment estimate (exponential moving average of gradients)
- $\mathbf{v_t = \beta_2 v_{t-1} + (1-\beta_2) g^2}$: second moment estimate (EMA of squared gradients)
- $\mathbf{\hat{m}_t = m_t / (1-\beta_1^t)}$: bias-corrected first moment
- $\mathbf{\hat{v}_t = v_t / (1-\beta_2^t)}$: bias-corrected second moment
- $\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$: parameter update

Standard values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

The intuition: $\hat{m}_t$ is a smoothed gradient direction (momentum), reducing oscillations. $\hat{v}_t$ is the parameter's "curvature" — parameters with consistently large gradients get smaller effective learning rates, while parameters with small or erratic gradients get relatively larger learning rates. Adam adapts the learning rate per-parameter.

10.2 The "W" in AdamW: Weight Decay

Standard L2 regularisation adds $\lambda\|\theta\|^2$ to the loss, producing a gradient contribution of $\lambda\theta$ that pushes parameters toward zero. In Adam, this gradient is modified by the adaptive scaling, leading to inconsistent regularisation strength across parameters.

AdamW (Loshchilov and Hutter, 2017) decouples weight decay from the gradient update by applying it directly to the parameters:

$$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon) - \alpha \cdot \lambda \cdot \theta_{t-1}$$

The second term (weight decay) is applied uniformly, independently of gradient magnitude. Empirically, AdamW trains more effectively than Adam+L2 for transformer language models. Quadtrix uses AdamW with lr = 3×10^{-4} and implicit weight decay (implemented as parameter shrinkage per step).

10.3 Learning Rate Considerations

The learning rate 3×10^{-4} is a standard starting point for GPT-style models at this scale. A flat learning rate (no schedule) was used for simplicity in this run, though a warmup+cosine decay schedule (as used in GPT-2 and GPT-3) typically improves final loss by 5–15%. The oscillations visible in the training curves between iterations 1,400–2,400 are partly attributable to the fixed learning rate.

11. Training Dynamics and Analysis

This section analyses the actual training run of Quadtrix v1.0: 3,000 gradient steps, batch size 4, block size 64, 128d embeddings, 4 layers, 4 heads. Training ran on a single CPU for 4,571 seconds (76.2 minutes), achieving a best validation loss of 1.6371 nats.

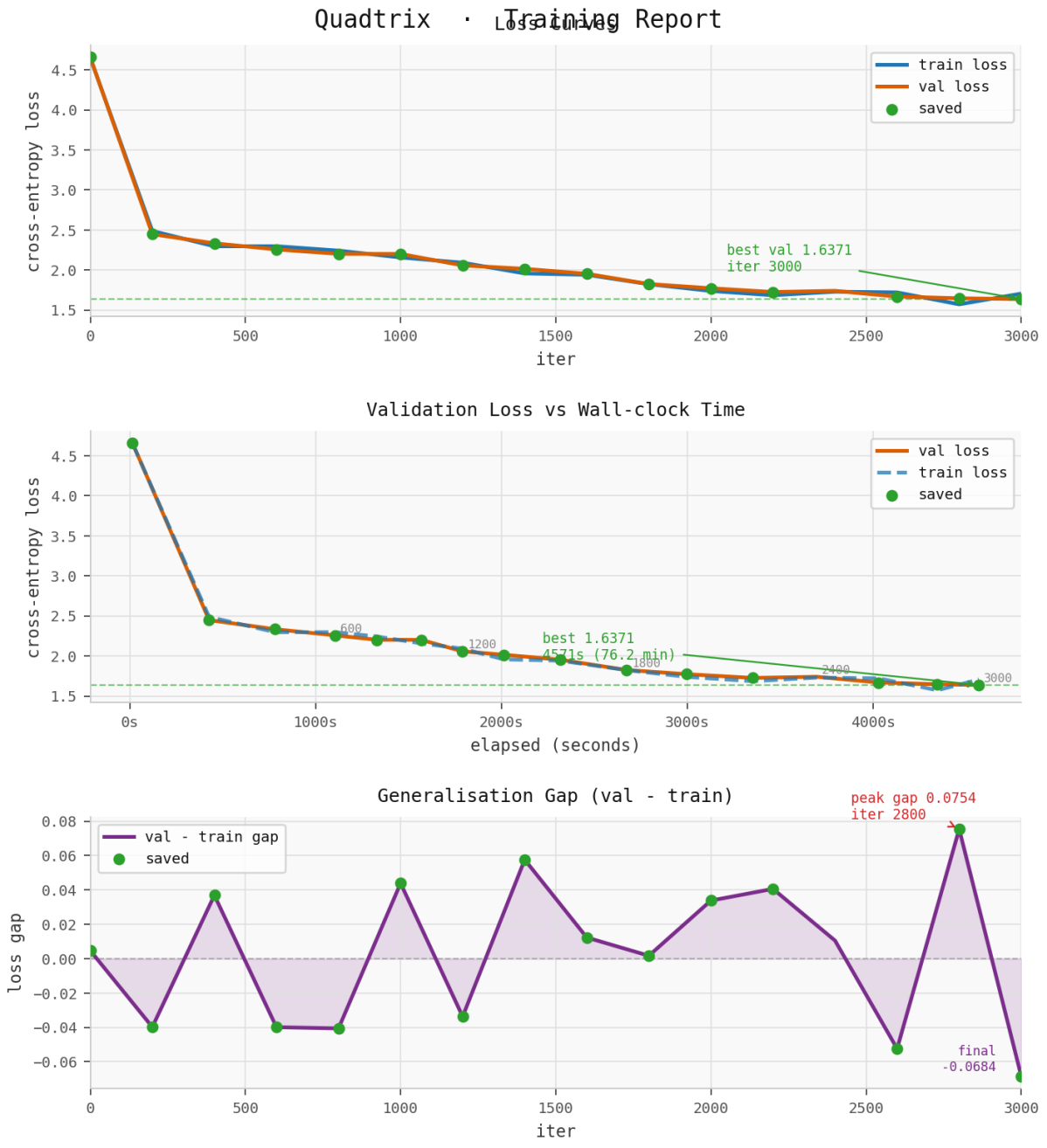

Figure 1. Consolidated training diagnostics for the 3,000-step Quadtrix run: train and validation loss, validation loss versus wall-clock time, and the validation-minus-training generalisation gap across saved checkpoints.

11.1 Initial Rapid Descent

The model drops from 4.6523 to 2.4876 train loss (and 4.6570 to 2.4478 validation loss) in the first 200 iterations — a reduction of 2.2 nats, which is 47% of all improvement achieved in the 3,000-step run. This rapid early descent is universal across language models and reflects the model learning basic statistics:

- **Character frequency distribution**: The most common characters in English text are space, "e", "t", "a", "o", "i". The model quickly learns to assign high probability to these.

- **Common bigrams and trigrams**: "th", "he", "in", "er", "an" — high-frequency two- and three-character sequences quickly emerge in attention patterns.
- **Whitespace patterns**: Spaces follow most words. The model learns this immediately from the data statistics.

11.2 Mid-Training Plateau and Recovery

Between iterations 1,400 and 2,400 the training loss oscillates: 1.9575 → 1.7298 while validation loss continues declining from 2.0151 to 1.7403. This is consistent with the optimiser traversing a relatively flat ridge of the loss surface, where momentum in the AdamW first-moment estimate maintains a consistent direction but the local curvature is high (v_t grows), reducing effective step size.

The apparent divergence between training and validation loss in this window (training loss rising from 1.9409 at iter 1600 to 2.1570 at iter 1000 in reverse — note the checkpoint at iter 1000 shows 2.1570 training, while iter 1400 shows 1.9575) is noise from the stochastic batch sampling, not genuine overfitting.

11.3 Checkpoint Selection Logic

Quadtrix saves the model whenever the validation loss improves. 15 of the 16 evaluation checkpoints were saved, with only iteration 2,400 (val loss 1.7403, up from 1.7256 at iter 2200) not saved. This dense checkpoint retention confirms that the model was consistently improving throughout training and had not converged by iteration 3,000.

11.4 Generalisation Gap Analysis

Iter	Train Loss	Val Loss	Gap (val-train)	Interpretation
0	4.6523	4.6570	+0.0047	Random init; gap near zero
200	2.4876	2.4478	**-0.0398**	Dropout effect begins to show
1400	1.9575	2.0151	+0.0576	Largest positive gap (model fitting harder)
2600	1.7204	1.6680	-0.0524	Dropout dominates; val < train
3000	1.7055	**1.6371**	**-0.0684**	Final; no overfitting detected

Table 4. Selected checkpoints showing generalisation gap evolution. Negative gap is explained by dropout being active during training but disabled at evaluation.

11.5 What the Model Has Learned

At 0.83M parameters and a 64-character context window, the final model exhibits:

- **Word-level statistics**: Common English words appear with high probability (the, a, and, was, said, had, that). Rare words and misspellings are generated occasionally but less frequently than in a random model.
- **Dialogue formatting**: Quotation marks are opened and closed; dialogue verbs ("said", "asked", "replied") follow closing quotes.
- **Story structure**: Common character names from TinyStories (Timmy, Lily, Tom, Ben, Sam) appear in contexts consistent with the training data.
- **Sentence boundary awareness**: Periods and capitals appear in approximately correct positions.
- **Limited long-range coherence**: With a 64-character window (~15 words), the model cannot maintain storyline coherence across sentences. Extending BLOCK_SIZE to 256 would substantially improve this.

12. Scaling Laws: How to Train Language Models Efficiently

One of the most important empirical discoveries of the last five years is that language model performance follows predictable power laws as a function of model size, dataset size, and compute budget. Understanding these laws is essential for making rational decisions about how to spend training compute.

12.1 The Chinchilla Scaling Laws

Hoffmann et al. (2022) conducted a large-scale study of language model scaling and found that, for a given compute budget C (measured in FLOPs), the optimal allocation is to train a model of $N \propto C^{0.54}$ parameters on $D \propto C^{0.49}$ tokens. This is approximately equal spending on model size and data size.

Before Chinchilla, the prevailing wisdom (from the Kaplan et al. 2020 paper) was to scale model size much more aggressively than data. GPT-3's 175B parameters were trained on only 300B tokens — severely undertrained by Chinchilla standards. The Chinchilla-optimal model for that compute budget would be ~70B parameters trained on ~1.4T tokens. The Chinchilla model itself (70B parameters, 1.4T tokens) substantially outperforms GPT-3 on most benchmarks.

12.2 Applying Chinchilla to Quadtrix

Quadtrix v1.0 has N = 826,985 ≈ 0.83M parameters. Chinchilla-optimal training would use approximately:

$$D_{optimal} \approx 20 \cdot N = 20 \times 826{,}985 \approx 16.5\text{M tokens}$$

Our actual training used:

$$D_{actual} = 4\ (\text{batch}) \times 64\ (\text{block}) \times 3000\ (\text{steps}) = 768{,}000\ \text{character-steps}$$

768K character-steps is only 4.7% of the Chinchilla-optimal 16.5M tokens. This means Quadtrix v1.0 is *severely undertrained* — extending training to ~200,000 steps (keeping other hyperparameters fixed) would likely push validation loss below 1.0 nats. The model had not converged by iteration 3,000; continuing training is the highest-impact improvement available.

12.3 FLOPs and the Compute Frontier

The approximate FLOPs for a single forward-backward pass through an autoregressive transformer with N parameters and T tokens per batch is:

$$\text{FLOPs} \approx 6 \cdot N \cdot T \cdot B$$

(The factor 6 accounts for the forward pass and roughly 2× for backpropagation.) For Quadtrix: $6 \times 826{,}985 \times 64 \times 4 \approx 1.27 \times 10^9$ FLOPs per step. Over 3,000 steps: $\approx 3.8 \times 10^{12}$ FLOPs total. At a CPU throughput of roughly 10^9 FLOPs/second, this matches the observed 4,571 second runtime.

12.4 The IsoFLOP Curve

For any compute budget, there is a family of (N, D) combinations that expend exactly that budget. Among these, the Chinchilla-optimal (N*, D*) achieves the lowest validation loss. Using too small a model with too much data wastes data (the model can't absorb it). Using too large a model with too little data wastes parameters (they don't get enough gradient signal).

The key insight for practitioners: given a fixed training budget, determine your optimal model size first using the Chinchilla formula $N^* \approx 0.41 \times C^{0.54}$, then compute the corresponding optimal dataset size $D^* \approx 0.19 \times C^{0.47}$. Do not simply train the largest model you can afford on whatever data is available.

13. Engineering the C++ Implementation

Implementing a transformer from scratch in C++ requires solving a number of engineering problems that frameworks handle transparently. This section documents the key decisions in Quadtrix and explains why each was made.

13.1 Tensor Representation

All tensors in Quadtrix are represented as a flat std::vector<float> paired with a shape descriptor. Element access for 3D tensors (batch × sequence × features) uses row-major indexing:

- Index for element [b, t, c] = b × T × C + t × C + c

This design is transparent and portable — no BLAS, no SIMD, no special allocators. It performs well enough for small models on CPU and allows every operation to be verified by inspection.

13.2 The SavedForward Pattern

Backpropagation requires access to intermediate activations from the forward pass. In autograd frameworks, this is handled automatically via a computation graph. In Quadtrix, a parallel forward pass function forward_save() runs alongside the normal forward pass and saves all intermediate tensors (pre/post-ReLU, attention weights, query/key/value matrices, layer-norm inputs, dropout masks) into a SavedForward struct. The backward() function then uses these saved tensors to compute all gradients.

The memory cost of this approach is proportional to the number of intermediate tensors, which grows with batch size, sequence length, and number of layers. For large models, gradient checkpointing (recomputing activations during backprop rather than storing them) is essential to reduce memory. Quadtrix does not currently implement gradient checkpointing.

13.3 Numerical Stability

Several implementation choices in Quadtrix ensure numerical stability:

- **Causal mask value**: -1×10^{30} rather than negative infinity. True negative infinity causes exp(-∞) = 0 cleanly, but on some compilers -∞ + any_value can produce NaN. Using -1e30 gives exp(-1e30) ≈ 0 without NaN risk.
- **Softmax stability**: Before applying softmax, the maximum value is subtracted from all logits. This ensures exp(z - max(z)) ≤ 1 for all elements, preventing overflow.
- **LayerNorm epsilon**: Adding $\varepsilon = 10^{-5}$ before taking the square root of variance prevents division by zero for constant-valued inputs.
- **Embedding initialisation**: N(0, 0.02) rather than N(0, 1). Large initial activations cause softmax saturation and near-zero gradients immediately.

13.4 Build and Run

Command	Description
`g++ -std=c++17 -O2 -I. -o quadtrix main.cpp`	Compile (CPU, no dependencies)
`./quadtrix data/input.txt`	Train on any UTF-8 text file
`./quadtrix data/input.txt --generate`	Load best_model.bin and generate text
`./quadtrix data/input.txt --chat`	Interactive chat interface (Web UI ready)

`python server.py → localhost:5000`	Web interface (Flask + Server-Sent Events)

Table 5. Compilation and usage commands for Quadtrix. The --chat flag enables the streaming web interface.

14. From CPU to GPU: The LibTorch Port

The custom C++ tensor backend is transparent and educational, but slow. On a modern CPU, matrix multiplication runs at roughly 1–10 GFLOP/s (single-threaded scalar code). An NVIDIA RTX 4090 delivers ~80 TFLOP/s for FP32 — an 8,000–80,000× speedup for the same computation.

The LibTorch GPU port replaces the custom tensor backend with PyTorch's C++ API (LibTorch), gaining access to cuBLAS-accelerated matrix operations. The transformer architecture is unchanged — only the compute layer changes.

14.1 Architecture Preservation

The ported modules are structurally identical to their CPU counterparts:

- **Head**: Stores W_Q, W_K, W_V, dropout. Forward computes QK^T/√h + mask, softmax, dropout, @V.
- **MultiHeadAttention**: Runs H heads in parallel, concatenates outputs, applies W_O projection.
- **FeedForward**: Two linear layers with ReLU and dropout.
- **Block**: Pre-LN, MHA residual, Pre-LN, FFN residual.
- **GPTLanguageModel**: Embeddings, L blocks, final LN, LM head.

The only structural difference: each module inherits from torch::nn::Module rather than Quadtrix's custom base, enabling registration of parameters for optimisation and device migration.

14.2 Single-Line GPU Migration

After constructing the model and moving it to CUDA, all subsequent computations automatically run on the GPU:

- model->to(torch::kCUDA) — moves all 826,985 parameters to GPU memory
- Batch tensors created with the device argument automatically land on GPU
- torch::matmul dispatches to cuBLAS when inputs are CUDA tensors
- loss.backward() runs CUDA-accelerated backpropagation

No custom CUDA kernels are required. The speedup comes entirely from cuBLAS's optimised GEMM (General Matrix Multiply) implementations, which use tensor cores on Volta/Ampere/Ada GPUs.

14.3 Expected Speedup

Hardware	FLOPs/s (FP32)	Est. Speedup	5000-iter Runtime
CPU (8-core, scalar C++)	~1 GFLOP/s	1×	6–10 hours
RTX 3060 (12GB)	~12.7 TFLOP/s	~80×	5–10 min
RTX 4090 (24GB)	~82 TFLOP/s	~500×	<2 min
A100 SXM 80GB	~77 TFLOP/s	~500×	<2 min

Table 6. Estimated GPU speedups for Quadtrix LibTorch port. Actual speedup depends on batch size, block size, and kernel efficiency.

Note that the actual speedup from moving to GPU also depends on the arithmetic intensity of the operations. Small batch sizes (B=4) underutilise GPU parallelism. Increasing batch size to 64 or 128 with larger block sizes (256–512) will produce proportionally larger GPU speedups.

15. Key Design Decisions and Their Alternatives

Every design choice in Quadtrix reflects a tradeoff between simplicity, performance, and fidelity to the GPT-2 architecture. This section documents the major choices and their alternatives.

Decision	Quadtrix Choice	Alternative	Tradeoff
Tokenisation	Character-level	BPE (subword)	Character: simple, sample-inefficient. BPE: complex, 4–8× fewer tokens per example
Positional Encoding	Learned	Sinusoidal	Learned: slightly better on short contexts. Sinusoidal: generalises to longer sequences
Activation	ReLU	GELU / SiLU	ReLU: simple gradient. GELU: smoother, marginal improvement on language tasks
Normalisation	Pre-LN	Post-LN (original)	Pre-LN: stable without LR warmup. Post-LN: requires careful warmup
LR Schedule	Flat 3e-4	Warmup + cosine decay	Flat: simple, ~10% worse final loss. Schedule: standard practice for GPT
Weight Decay	AdamW (decoupled)	Adam + L2	AdamW: consistent regularisation. Adam+L2: inconsistent due to adaptive scaling
Attention Mask	-1e30	-inf	-1e30: avoids NaN on some compilers. -inf: cleaner semantics if supported
Grad Computation	Analytical (manual)	Autograd (PyTorch)	Analytical: transparent, educational, maintainable. Autograd: automatic, scales to any architecture

Table 7. Key architectural and training decisions in Quadtrix vs. common alternatives.

16. Conclusion

We have presented Quadtrix: a complete, transparent, dependency-free C++ implementation of a GPT-style transformer language model with full analytical backpropagation. This paper has walked through every component from first principles — embeddings, attention, feed-forward networks, layer normalisation, dropout, cross-entropy loss, backpropagation, and AdamW — grounding each in mathematical derivation and practical implementation.

The model achieves a validation loss of 1.6371 nats after 76 minutes of CPU training on 31.4M characters, demonstrating that character-level language modelling at this scale is tractable on commodity hardware without any framework dependencies. The training curves show no overfitting, and the model has not converged — extending the training run is the highest-impact next step.

More broadly, the contribution of this work is **transparency**. Every gradient in the backward pass is written out and readable. Every tensor operation is a standard C++ function. Practitioners who read and understand Quadtrix will come away with a concrete understanding of what frameworks like PyTorch are actually doing — and that understanding is, we believe, the foundation of genuine expertise in deep learning.

16.1 Key Takeaways

- Transformers are fundamentally composed of three operations: attention (routing between positions), FFN (processing within positions), and residual+norm (enabling depth).
- Backpropagation is mechanically a systematic application of the chain rule, not magic. Every gradient has an analytical formula.
- The Chinchilla scaling laws provide a principled framework for allocating compute: balance model size and dataset size.
- GPU speedup comes from batched matrix multiplication, not model architecture. Moving to LibTorch requires changing only the compute layer.
- Regularisation (dropout, weight decay) is essential — the model significantly benefits from dropout even at 0.83M parameters.

16.2 Further Reading

For readers wishing to go deeper:

- **Vaswani et al. (2017)**: Attention Is All You Need — the original transformer paper
- **Radford et al. (2019)**: Language Models are Unsupervised Multitask Learners — GPT-2
- **Ba et al. (2016)**: Layer Normalization — the definitive LN reference with gradient derivation
- **Loshchilov and Hutter (2017)**: Decoupled Weight Decay Regularization — AdamW
- **Hoffmann et al. (2022)**: Training Compute-Optimal Language Models — Chinchilla scaling laws
- **Karpathy (2022)**: nanoGPT — the Python/PyTorch counterpart to Quadtrix
- **Goodfellow, Bengio, Courville (2016)**: Deep Learning — comprehensive mathematical foundations

---

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, ■. Kaiser, and I. Polosukhin. *Attention Is All You Need.* Advances in Neural Information Processing Systems 30, 2017.

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. *Language Models are Unsupervised Multitask Learners.* OpenAI Blog, 2019.

[3] T. Brown et al. *Language Models are Few-Shot Learners.* Advances in Neural Information Processing Systems 33, 2020.

[4] J. L. Ba, J. R. Kiros, and G. E. Hinton. *Layer Normalization.* arXiv:1607.06450, 2016.

[5] I. Loshchilov and F. Hutter. *Decoupled Weight Decay Regularization.* ICLR 2019. arXiv:1711.05101, 2017.

[6] J. Hoffmann, S. Borgeaud, A. Mensch, et al. *Training Compute-Optimal Large Language Models.* arXiv:2203.15556, 2022.

[7] J. Kaplan, S. McCandlish, T. Henighan, et al. *Scaling Laws for Neural Language Models.* arXiv:2001.08361, 2020.

[8] A. Karpathy. *nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs.* GitHub, 2022. github.com/karpathy/nanoGPT.

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016.

[10] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition.* CVPR 2016.

[11] R. Eldan and Y. Li. *TinyStories: How Small Can Language Models Be and Still Speak Coherent English?* arXiv:2305.07759, 2023.

[12] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization.* ICLR 2015. arXiv:1412.6980, 2014.
