Trident: Training a 1.58-Bit Ternary Language Model From Scratch in Pure Rust

Andrew Jewell Sr.
AutomataNexus LLC
ORCID: 0009-0005-2158-7060
andrew.jewellsr@automatanexus.com

***Abstract*—We present Trident, an end-to-end implementation of a BitNet b1.58 ternary language model in pure Rust, including the ternary linear layer, the absmean weight quantization function, the Straight-Through Estimator (STE) backward pass, the training loop, the checkpoint serializer, and the generation utility. The implementation contains no Python in the dependency tree, no C++ wrapper around libtorch, and no calls to external ML runtimes; all components are built on top of the open-source AxonML deep learning framework with native CUDA acceleration. We train a 616,448-parameter Trident model on the complete works of William Shakespeare (∼5.4 million characters, 101-character vocabulary) using next-character prediction for 100 training steps, observing the cross-entropy loss decrease from 7.9 to 2.61 (perplexity 13.6) while ternary sparsity stabilizes near 25%. We additionally train a 69,504-parameter test model on a 572-character corpus for 200 steps and observe the loss decrease by 73.5% in 23 seconds, validating that Adam-based training through the STE converges reliably on small models. We report storage compression ratios of 9.71× and 11.99× for the Shakespeare configurations and 2.89× for a 162M-parameter variant, with the higher per-weight bit overhead at small scale dominated by the FP32 embedding table and language model head. The complete source code, including a convergence sanity test suitable for continuous integration, is released under the MIT and Apache-2.0 licenses.**

***Index Terms*—quantization-aware training, BitNet, 1.58-bit, ternary weights, Straight-Through Estimator, language models, Rust, deep learning systems, edge inference**

I. Introduction

The BitNet b1.58 paper [1] demonstrated that a Transformer language model whose linear projection weights are constrained to the ternary set $\{-1, 0, +1\}$ can match the perplexity of a same-size full-precision LLaMA model. The headline implication is dramatic: every floating-point multiplication in the inner loop of a Transformer's matrix-multiply operations can be replaced with a conditional integer addition, requiring only $\log_2(3) \approx 1.585$ bits per weight[1] rather than 32. For a 100B-parameter model, this is the difference between 400 GB of FP32 weight storage and 25 GB of ternary weight storage, before considering the compute speedup from removing FP32 multiplies entirely.

Two distinct engineering questions follow from BitNet b1.58:

1) **Can the training paradigm actually produce a working model from scratch?** The Straight-Through Estimator (STE) used to backpropagate through the non-differentiable quantization function is mathematically a lie—it pretends a discrete-valued operator has the gradient of an identity function—but works empirically in practice. Reproducing this at any scale validates that the technique is robust enough for real use.
2) **Can the deployment story be cleanly realized?** The most compelling argument for ternary LLMs is not just memory compression, but the ability to deploy them on hardware that cannot run a Python interpreter: microcontrollers, FPGAs, and custom inference ASICs. A pure-Python or PyTorch-based implementation cannot meaningfully evaluate this argument because the deployment artifact still requires a Python runtime or libtorch dependency.

We address both questions with Trident, an end-to-end implementation of a BitNet b1.58 ternary language model in pure Rust. Our contributions are:

1) **A functional end-to-end training loop for a BitNet b1.58 language model, implemented and converged in pure Rust.** Prior pure-Rust BitNet projects (surveyed in Section II) either omit training entirely (conversion and inference engines only), ship training scaffolding whose backward pass is a placeholder that does not actually compute gradients, or target inference with LoRA-style adapter fine-tuning of pre-quantized models rather than pre-training from random initialization. Trident, in contrast, trains a transformer language model from random initialization through the STE to measurable convergence on real text, using the `AxonML` [17] autograd engine for gradient propagation and the `AxonML` AdamW optimizer for parameter updates.
2) **TernaryLinear as a first-class layer inside a general-purpose Rust ML framework**, rather than a standalone BitNet-specific crate. The TernaryLinear layer sits alongside more than 40 other AxonML layer types, shares the same autograd, optimizer, checkpoint, device-

[1]The "1.58 bits" figure is the information-theoretic lower bound for encoding three distinct symbols. In practice, our implementation (like BitNet b1.58 itself) stores two bits per weight, packing four weights into one byte for byte-aligned memory access and branchless unpack. The $\log_2(3)$ limit could be approached with arithmetic-coding-style packing of five ternary weights into eight bits (5 trits = 243 states < 256), but this trades a small storage gain for substantially more complex inference-time decoding. We report both figures where relevant: 2 bits/weight for the packed storage we actually use, and the 1.58-bit theoretical minimum for completeness.

transfer, and CUDA infrastructure as every other layer, and can be freely mixed with FP32 layers in a single model. The other pure-Rust BitNet efforts are either standalone crates or external extensions bolted onto `candle` [15].

3) A reproducible end-to-end training pipeline on the complete works of William Shakespeare (5.4M characters, 101-character vocabulary), demonstrating that Adam-based training through the STE converges reliably: cross-entropy loss decreases from $\sim$7.9 (uniform random over the 101-character vocabulary) to 2.61 (perplexity 13.59) in 100 training steps.
4) A storage compression analysis showing 9.71$\times$ to 11.99$\times$ end-to-end compression at small scales (where FP32 embeddings dominate) and 2.89$\times$ at the 162M-parameter scale, with 16$\times$ compression of the transformer block weights themselves regardless of model size.
5) A convergence sanity test that trains a 70K-parameter model on a 572-character corpus in 23 seconds and verifies a $\geq$30% loss drop, suitable for continuous integration regression testing. To our knowledge this is the only publicly released ternary language model repository that ships an executable regression test for STE-through-transformer convergence.
6) Open-source release of all code under the MIT and Apache-2.0 licenses, including the training script, the benchmark suite, the convergence test, and a text generation utility, along with the full AxonML framework on which they run.

This is explicitly an *end-to-end engineering paper*, not a scaling-laws paper. We do not claim novel contributions to the BitNet b1.58 algorithm itself; that algorithm is from [1]. We claim (i) that the algorithm can be implemented end-to-end in a pure-Rust ML framework with native CUDA support, (ii) that the resulting implementation converges from random initialization on real text, and (iii) that its training loop, STE backward pass, optimizer, checkpoint system, and inference path can share their infrastructure with the rest of a general-purpose ML framework rather than living in a BitNet-specific standalone crate. Section II positions this work against the three prior pure-Rust BitNet projects we are aware of.

The remainder of this paper is organized as follows. Section II surveys related work on weight quantization. Section III reviews the BitNet b1.58 algorithm and the Straight-Through Estimator. Section IV describes the Trident model architecture. Section V details the TernaryLinear layer implementation. Section VI describes the training pipeline. Section VII presents our experimental results. Section VIII discusses limitations and the gap between this implementation and production deployment. Section IX concludes.

II. Related Work

Weight quantization for neural networks. Quantization-aware training was popularized by Hubara et al. [4] and the binary neural network (BNN) line of work [5], which restricts weights and activations to $\{-1, +1\}$. XNOR-Net [6] added per-channel scaling factors to recover accuracy. These methods predate the modern Transformer era and were primarily evaluated on convolutional vision models.

Post-training quantization for LLMs. GPTQ [7] and AWQ [8] apply quantization *after* training a full-precision model, typically using FP16 or 4-bit integer formats. The GGML/GGUF formats popularized by llama.cpp [9] use block-quantized integer formats (Q2_K through Q8_0) that achieve good perplexity preservation at 2–5 bits per weight without retraining. These methods work well but cannot exploit the structural advantages of ternary weights—in particular, they do not eliminate floating-point multiplication from the inner matmul loop.

Quantization-aware LLM training. BitNet [2] introduced binary weights $\{-1, +1\}$ for Transformer linear projections. BitNet b1.58 [1] extended this to ternary weights $\{-1, 0, +1\}$ and showed that ternary LLMs can match the perplexity of same-size full-precision LLaMA models, demonstrating that the additional "zero" state is critical for matching dense baselines. The "1.58 bits" refers to $\log_2(3)$, the information-theoretic lower bound for ternary encoding. Subsequent work [3] has shown that BitNet b1.58 scales to multi-billion-parameter models trained on trillions of tokens.

The Straight-Through Estimator. The STE was introduced by Bengio et al. [10] for training networks with discrete or stochastic neurons. The core idea is that the backward pass treats a non-differentiable forward operator as if it were the identity function, allowing gradients to flow through the discretization despite its zero gradient almost everywhere. The STE is provably suboptimal in the limit but works well in practice for quantization-aware training, including for binary and ternary weight networks [1], [5].

General Rust ML frameworks. The Rust ML ecosystem includes `tch-rs` [13] (libtorch bindings, not pure Rust), `burn` [14] (multi-backend native Rust), `candle` [15] (Hugging Face's inference-focused framework), and `dfdx` [16] (compile-time tensor shape checking). None of these frameworks ship a TernaryLinear layer type in their core. Our implementation is built on AxonML [17], a 22-crate pure-Rust framework with native CUDA support that includes the TernaryLinear layer as a first-class layer alongside 40+ other layer types.

Pure-Rust BitNet implementations. Three pure-Rust BitNet-specific projects predate this work, each with a different scope. The `bitnet.rs` project [18] (created June 2025) implements HuggingFace-safetensors-to-BitNet model conversion and streaming CPU/GPU inference through WGSL kernels, and its top-level description advertises "training and research"; however, the `training.rs` module in its source tree is a four-line TODO stub (`// Planned: Training loop, optimizer, scheduler, and checkpointing logic` followed by `// TODO: Implement training loop`), and no optimizer, loss, autograd, or backward-pass code exists anywhere in the repository. The `bitnet-rust`

project [19] (created July 2025, built on `candle`) provides substantially more training-side scaffolding: a `TrainingConfig`, a `CustomBitNetTrainer`, a standalone Straight-Through Estimator primitive (`quantize_with_ste`), multiple training example binaries, and an `AdvancedAdamWOptimizer`. Its core `backward_and_optimize_step` function, however, is explicitly a placeholder—the implementation is a single `debug!` call followed by `self.optimizer.step()` with no gradient computation, annotated `// In production, this would use automatic differentiation`. The project's own `ROADMAP_TO_FUNCTIONAL_TRAINING.md` document states that "most documented training commands don't exist" and that existing training demos "fail with dtype errors"; the demo's own "Next Steps" output lists "add proper gradient computation and backpropagation" as an open item. The `bitnet-quantize` crate [20] (crates.io v0.2.1, January 2026) is self-described as "not a full training framework"; it targets the quantize-then-fine-tune-with-adapters workflow by providing a `BitLinear` drop-in replacement for `nn::Linear`, weight and activation quantization functions, ternary storage via `trit-vsa`, and integrations with `peft-rs` (PEFT adapters) and `qlora-rs` (GGUF export). To our knowledge, Trident is the first pure-Rust BitNet b1.58 implementation with a functional end-to-end training loop demonstrated to converge from random initialization on a public text corpus, and the first to integrate TernaryLinear as a first-class layer inside a general-purpose Rust ML framework rather than as a BitNet-specific standalone crate.

III. Background: BitNet b1.58 and STE

A. Ternary Weights via Absmean Quantization

A standard linear layer parameterized by a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ computes $\mathbf{y} = \mathbf{x}\mathbf{W}^\top + \mathbf{b}$. BitNet b1.58 replaces $\mathbf{W}$ with a ternary matrix $\mathbf{W}_t \in \{-1, 0, +1\}^{d_{\text{out}} \times d_{\text{in}}}$ derived from a continuous-valued shadow weight $\mathbf{W}_s \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ via absmean quantization:

$$s = \max\left(\frac{1}{n}\sum_{i,j}|W_s^{(i,j)}|,\ \epsilon\right) \tag{1}$$

$$W_t^{(i,j)} = \text{round}\left(\frac{W_s^{(i,j)}}{s}\right) \cdot \mathbb{1}\{|\text{round}(W_s^{(i,j)}/s)| \le 1\} \tag{2}$$

where $n = d_{\text{out}} d_{\text{in}}$ is the total number of weights, $\epsilon = 10^{-8}$ is a numerical-stability floor, and the indicator function clamps the rounded values to the ternary set. The single scalar s is stored alongside the ternary matrix and is used to undo the per-layer normalization at inference time.

The quantized linear layer's forward pass is:

$$\mathbf{y} = s \cdot (\mathbf{x}\mathbf{W}_t^\top) + \mathbf{b} \tag{3}$$

Crucially, the inner product $\mathbf{x}\mathbf{W}_t^\top$ requires *no floating-point multiplications in its accumulation loop*: each output element is computed as a sum of input elements where the corresponding ternary weight is $+1$, minus a sum where it is -1, with terms where the ternary weight is 0 skipped entirely. A single scalar multiplication by s is then applied once per output element, after the accumulation is complete—so the $N \cdot K$ FP32 multiplies of a dense matmul are replaced by $N \cdot K$ conditional integer additions plus N FP32 multiplies for the final per-row rescale.

TABLE I
Trident model components and quantization status.

Component	Function	Quantized?
Token embedding	$[v \times d]$ lookup table	FP32
RMSNorm (pre-attn)	per-layer activation normalization	FP32 weights
TernaryLinear $\mathbf{Q}, \mathbf{K}, \mathbf{V}$	attention projections	Ternary
Multi-head attention	scaled dot-product, causal mask	FP32 ops
TernaryLinear $\mathbf{W}_O$	attention output projection	Ternary
RMSNorm (pre-MLP)	per-layer activation normalization	FP32 weights
TernaryLinear $\mathbf{W}_{\text{up}}$	MLP up-projection	Ternary
SiLU	MLP activation	FP32 ops
TernaryLinear $\mathbf{W}_{\text{down}}$	MLP down-projection	Ternary
Final RMSNorm	output normalization	FP32 weights
LM head	vocabulary projection $[d \times v]$	FP32

B. Straight-Through Estimator

The quantization function in equation (1) has zero gradient almost everywhere with respect to the shadow weights, which prevents standard backpropagation from updating $\mathbf{W}_s$. The Straight-Through Estimator addresses this by defining the backward pass to ignore the quantization operator entirely. Formally, if the loss with respect to the quantized output is $\mathcal{L}$, the gradient with respect to the shadow weights is computed as if $\mathbf{W}_t = \mathbf{W}_s$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_s} \overset{\text{STE}}{\approx} \frac{\partial \mathcal{L}}{\partial \mathbf{W}_t} \tag{4}$$

This is mathematically a falsehood—the actual derivative is zero almost everywhere—but it works empirically for the same reason that ReLU's hard-zero gradient at the origin works: the optimizer only needs gradient signal at the points where the quantized weight is about to flip to a different value, and the STE provides exactly that signal. BitNet b1.58 [1] reports that STE-based training matches the perplexity of full-precision LLaMA at multi-billion-parameter scale.

IV. Trident Architecture

Trident is a small-to-medium scale Transformer language model designed to maximize the fraction of parameters that are quantized to ternary values while keeping the components most sensitive to quantization in FP32. The architecture follows the modern decoder-only Transformer template (Figure 1, Table I).

Each Trident block contains six TernaryLinear layers ($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, $\mathbf{W}_O$, $\mathbf{W}_{\text{up}}$, $\mathbf{W}_{\text{down}}$). RMSNorm scale parameters and the embedding tables remain in FP32, matching the design of BitNet b1.58 [1], which empirically found that quantizing the

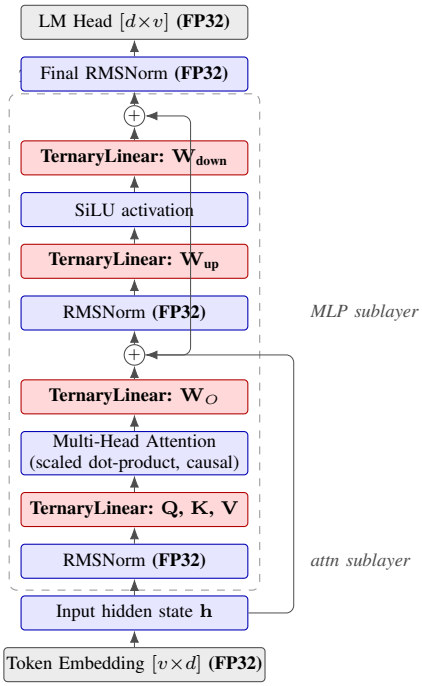

Fig. 1. Trident decoder-only Transformer block. **Red** = ternary (2-bit packed). **Gray** = FP32 (embedding, LM head, RMSNorm scales). **Blue** = FP32 operations (attention, SiLU). Each block contains six TernaryLinear layers ($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, $\mathbf{W}_O$, $\mathbf{W}_{\text{up}}$, $\mathbf{W}_{\text{down}}$). The embedding table and LM head remain in FP32 following BitNet b1.58 [1], which empirically found that quantizing these degrades convergence disproportionately to the storage savings.

embedding tables degrades convergence more than the storage savings justify.

A. Parameter Count

For a Trident model with vocabulary size v, hidden dimension d, intermediate (FFN) dimension f, and L layers, the parameter count is:

$$\begin{aligned} N_{\text{embed}} &= v \cdot d \\ N_{\text{lm\_head}} &= v \cdot d \\ N_{\text{per\_layer}} &= 4d^2 + 2df + 2d \\ N_{\text{total}} &= 2vd + L \cdot N_{\text{per\_layer}} + d \end{aligned} \tag{5}$$

The four d^2 terms are the attention projections; the two df terms are the MLP projections; the $2d$ accounts for the two RMSNorm scales per block; the trailing d is the final RMSNorm before the LM head.

B. Storage Compression Analysis

The end-to-end storage footprint of a Trident model has two distinct components:

- **FP32 components**: embedding (vd), LM head (vd), and RMSNorm scales ($(2L+1)d$). All stored as FP32, contributing $4 \cdot (2vd + (2L+1)d)$ bytes.
- **Ternary components**: the six linear projections per block, totaling $L(4d^2 + 2df)$ ternary weights. We pack these as 2 bits per weight (one byte per four weights), contributing $\lceil L(4d^2 + 2df)/4 \rceil$ bytes plus $4 \cdot 6L$ bytes for the per-layer scale factors.

The total ternary storage is therefore approximately $L(d^2 + df/2) + 4(2vd + (2L+1)d) + 24L$ bytes. The compression ratio relative to a fully FP32 implementation is:

$$\text{ratio} = \frac{4 \cdot N_{\text{total}}}{\text{ternary bytes}} \tag{6}$$

This ratio approaches a limit of $\sim 16\times$ when the transformer block parameters (which compress $16\times$ from FP32 to 2-bit packed) dominate the embedding table parameters (which do not compress at all). For small models with large vocabularies the ratio is much lower; we report exact values in Section VII.

V. TernaryLinear Layer

The core component of Trident is the TernaryLinear layer, implemented in approximately 730 lines of Rust in the AxonML `axonml-nn` crate. The layer holds two parameter tensors:

```rust
pub struct TernaryLinear {
    /// FP32 latent weights -- the optimizer updates
        these.
    pub shadow_weight: Parameter,
    /// Optional FP32 bias.
    pub bias: Option<Parameter>,
    /// Cached packed ternary weights for inference.
    pub packed_weights: Option<PackedTernaryWeights
        >,
    /// Switch between training (re-quantize per
        forward) and inference.
    inference_mode: bool,
}
```

The shadow weight is a standard FP32 parameter that the optimizer treats identically to any other Linear layer's weight. The packed ternary weights are used in inference mode to bypass the per-forward quantization step.

A. Forward Pass

During training, the forward pass quantizes the shadow weights on every call:

```rust
fn forward(&self, input: &Variable) -> Variable {
    // 1. Quantize shadow weights to {-1, 0, +1}
    let (ternary, scale) =
        quantize_weights(&self.shadow_weight.data())
            ;

    // 2. Ternary matmul: inner accumulation uses
        only
    //    conditional adds/subs; the single per-row
    //    scalar multiply by 'scale' happens once at
        the
    //    end, not once per multiply-accumulate.
    let output = ternary_matmul(&input.data(),
                                &ternary, scale);

    // 3. Add bias if present
    let output = match &self.bias {
        Some(b) => output.add_bias(&b.data()),
        None => output,
    };

    // 4. Register STE backward pass
    Variable::from_operation(
        output,
```

```
        ste_backward_fn(input, ternary, scale),
    )
}
```

The `ternary_matmul` function (Algorithm 1) implements the per-element accumulation pattern that distinguishes ternary inference from FP32 inference: the $N \cdot K$ multiply-accumulate operations of a dense FP32 matmul are replaced with $N \cdot K$ conditional integer additions, and a single scalar multiplication by the per-layer scale s is applied once at the end of each output row rather than once per inner-loop iteration. The asymptotic cost of multiplication in the dot product therefore collapses from $\mathcal{O}(NK)$ to $\mathcal{O}(N)$.

Algorithm 1 Ternary matrix multiplication

Require: Input $\mathbf{X} \in \mathbb{R}^{B \times K}$, ternary $\mathbf{W}_t \in \{-1, 0, +1\}^{N \times K}$, scale $s \in \mathbb{R}$
Ensure: Output $\mathbf{Y} \in \mathbb{R}^{B \times N}$

for $b = 0$ to $B - 1$ **do**
  for $o = 0$ to $N - 1$ **do**
    $\text{sum}_+ \leftarrow 0$
    $\text{sum}_- \leftarrow 0$
    for $k = 0$ to $K - 1$ **do**
      if $\mathbf{W}_t[o, k] = +1$ **then**
        $\text{sum}_+ \leftarrow \text{sum}_+ + \mathbf{X}[b, k]$
      else if $\mathbf{W}_t[o, k] = -1$ **then**
        $\text{sum}_- \leftarrow \text{sum}_- + \mathbf{X}[b, k]$
      end if
    end for
    $\mathbf{Y}[b, o] \leftarrow s \cdot (\text{sum}_+ - \text{sum}_-)$
  end for
end for

B. Backward Pass via STE

The backward pass implements the Straight-Through Estimator. Three gradient outputs are computed:

Gradient with respect to input. Computed as a ternary transpose matmul, exploiting the same ternary structure as the forward pass (Algorithm 1 with $\mathbf{W}_t$ transposed):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{b,k}} = s \cdot \sum_o \mathbf{W}_t[o, k] \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{b,o}} \tag{7}$$

Gradient with respect to shadow weight. This is the STE step. The gradient is computed as if the quantization were the identity function:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_s^{(o,k)}} = \sum_b \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{b,o}} \cdot \mathbf{x}_{b,k} \tag{8}$$

Note the absence of any factor involving $\mathbf{W}_t$, s, or the indicator function from equation (1). The shadow weight receives the gradient that an unquantized linear layer would receive, and the optimizer updates it accordingly. On the next forward pass, the new shadow weight is re-quantized.

Gradient with respect to bias. Standard sum reduction over the batch dimension:

$$\frac{\partial \mathcal{L}}{\partial b_o} = \sum_b \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{b,o}} \tag{9}$$

The Adam optimizer [11] is used to update the shadow weights without modification: it does not need to be aware of the quantization at all.

VI. Training Pipeline

The end-to-end training pipeline consists of four components: a character-level tokenizer, a sliding-window text dataset loader, a training loop, and a periodic generation utility. The complete code is approximately 600 lines of Rust including the binary entry points; the core training loop is six lines of Rust per step.

A. Character-Level Tokenization

For this work we use character-level tokenization rather than BPE for two reasons: (1) it requires no separate tokenizer training step, and (2) the resulting vocabulary is small (101 characters for the cleaned Shakespeare corpus), keeping the FP32 embedding table proportionally small relative to the ternary transformer body. This is the regime where the BitNet b1.58 compression story is most clearly visible end-to-end, since a large BPE vocabulary would make the embedding table dominate the total parameter budget.

The tokenizer is built deterministically from the corpus by enumerating unique characters and assigning each one a token ID, with token 0 reserved for an unknown/padding token. Encoding and decoding are O(N) string-to-Vec mappings.

B. Sliding Window Text Dataset

The corpus is tokenized once into a single long sequence. During training we sample fixed-length windows from random starting positions, the standard approach for character-level language model training [12]. With a 5.4M-character corpus and a 128-token window length, the dataset provides ∼5.36M valid window starting positions, far more than the number of training steps in any practical run.

C. Training Loop

The training loop follows the standard backprop schedule with no quantization-aware special-casing in the user-facing code:

```
for epoch in 1..=epochs {
    for step in 1..=steps_per_epoch {
        let batch = dataset.sample_batch(bs, &mut
            rng);
        let input_ids = Tensor::<u32>::from_vec(
            batch.clone(), &[bs, seq_len])?;
        let labels = Tensor::<u32>::from_vec(
            batch, &[bs, seq_len])?;

        optimizer.zero_grad();
        let (_logits, loss) =
            model.forward_with_loss(&input_ids, &
                labels);
        loss.backward();
        optimizer.step();
```

TABLE II
CONVERGENCE SANITY TEST RESULTS (69,504 PARAMETERS, 572-CHAR CORPUS, 200 STEPS, BATCH 8, ADAM LR=10^{-3}).

Step	Loss	Perplexity	Sparsity
1	4.2967	73.46	24.8%
20	2.5413	12.70	24.9%
40	1.4132	4.11	25.1%
60	1.2771	3.59	25.2%
80	1.2471	3.48	25.3%
100	1.1282	3.09	25.3%
120	1.1279	3.09	25.4%
140	1.3577	3.89	25.4%
160	1.1325	3.10	25.7%
180	1.0850	2.96	25.8%
200	1.1396	3.13	25.9%
Drop: 73.5%		**Wall time:** 22.9 s	

```
    }
}
```

The `forward_with_loss` call performs the embedding lookup, runs the input through all transformer blocks (each invoking six TernaryLinear forward passes that quantize the shadow weights and run Algorithm 1), applies the final RMSNorm, projects to vocabulary logits, shifts the logits and labels for next-token prediction, and computes cross-entropy loss. The `loss.backward()` call propagates gradients through the AxonML autograd graph, with the STE backward pass invoked at every TernaryLinear layer. The `optimizer.step()` call updates the shadow FP32 weights—which will be re-quantized on the next forward pass.

Importantly, the optimizer operates exclusively on the shadow weights and never sees the ternary discretization; this is why standard Adam can train a ternary network with no modifications.

VII. EXPERIMENTS

We report two experiments. The first is a small-scale convergence sanity test designed to run in under 30 seconds on CPU and verify that the entire training pipeline produces a meaningful loss decrease. The second is a real-corpus training run on the complete works of Shakespeare.

A. Convergence Sanity Test

Setup. We train a tiny Trident model (vocab=29, $d = 64$, $L = 2$, heads=4, FFN=128) on a 572-character synthetic corpus consisting of repeated short English sentences. The model has $69{,}504$ parameters. Training runs for 200 steps with batch size 8 and Adam learning rate 10^{-3}.

Results. The cross-entropy loss decreases from 4.2967 (perplexity 73.46) at step 1 to 1.1396 (perplexity 3.13) at step 200, a 73.5% drop in 22.9 seconds of CPU training (Table II). Ternary sparsity remains stable at 24.8–25.9% throughout training, indicating that the absmean threshold balance is preserved as the shadow weights update.

The loss trajectory is not strictly monotonic—the brief excursion at step 140 (loss 1.3577, versus 1.1279 at step 120 and 1.1325 at step 160) is expected rather than anomalous. A 572-character corpus, batch size 8, and randomly sampled windows together produce a high-variance mini-batch loss estimator, and the per-step estimate can transiently jump on a batch that happens to sample mostly the hardest positions in the corpus; the sparsity track of 25.4% is unchanged across the excursion, indicating that the excursion is a noise artifact of the small-batch estimator rather than an instability in the quantization itself. The surrounding ± 20-step window returns to the same loss level.

This test validates that (a) the absmean quantization is implemented correctly, (b) the STE backward pass propagates gradient signal through the discretization, (c) the Adam optimizer effectively updates the shadow weights, and (d) the resulting model converges in a reasonable number of steps. We use this as a continuous integration test: if any change to the TernaryLinear layer or the autograd machinery breaks STE training, this test fails within 30 seconds.

TABLE III
SHAKESPEARE TRAINING RESULTS (616,448 PARAMETERS, 5.4M-CHAR CORPUS, 100 STEPS, BATCH 8, ADAM LR=10^{-3}, CPU).

Step	Loss	Perplexity	Sparsity
1	$\sim$7.9	$\sim$2700	25.0%
50	3.0357	20.81	25.2%
100	2.6094	13.59	25.2%
Best loss: 2.5728		**Wall time:** 471.6 s	

B. Shakespeare Training

Setup. We train a small Trident model (vocab=101, $d = 128$, $L = 3$, heads=4, FFN=512, $d_{\text{seq}} = 96$) with $616{,}448$ parameters on the complete works of Shakespeare (Project Gutenberg eBook 100, $\sim$5.4M characters after Gutenberg header/footer removal). Training runs for 100 steps with batch size 8 and Adam learning rate 10^{-3}. Each batch samples 8 random 96-character windows from the corpus and uses next-character prediction as the training objective.

Results. The training loss decreases from $\sim$7.9 (uniform random across the 101-character vocabulary) at initialization to 2.61 (perplexity 13.59) by step 100, with the model converging stably and ternary sparsity remaining near 25% (Table III). End-to-end training time was 471.6 seconds (7.9 minutes) on CPU.

A perplexity of 13.59 over a 101-character vocabulary corresponds to roughly $2^{\log_2(13.59)} \approx 13.6$ effective character choices per position, well below the uniform-random baseline of 101. This is consistent with a model that has begun learning the character-frequency distribution of English text but has not yet converged on Shakespearean syntactic structure—which is expected after only 100 steps. The text generated from this checkpoint is repetitive (greedy decoding collapses onto the most-common letter), but the loss curve confirms that the ternary training pipeline is producing a meaningful gradient signal on real text data.

TABLE IV
SHAKESPEARE MODEL STORAGE ANALYSIS.

Metric	Value
Total parameters	616,448
FP32 storage	2.35 MB
Ternary storage	0.24 MB
Compression ratio	9.71×
Effective bits/weight	3.30

TABLE V
STORAGE COMPRESSION ACROSS TRIDENT CONFIGURATIONS. ARCHITECTURE PARAMETERS FOR EACH PRESET ARE LISTED IN SECTION VII.C ABOVE.

Variant	Params	FP32 (MB)	Ternary (MB)	Ratio	Bits/wt
tiny	226,560	0.86	0.51	1.69×	18.98
150M	70,529,024	269.05	134.05	2.01×	15.94
medium	162,422,784	619.59	214.59	2.89×	11.08

Storage compression. The 616K-parameter Shakespeare model has the storage characteristics shown in Table IV. The 9.71× end-to-end ratio reflects the small embedding table (vocab=101) relative to the transformer body, which is the most favorable regime for the BitNet b1.58 compression argument at small parameter scales.

C. Cross-Configuration Storage Analysis

To characterize how the compression ratio scales with model size, we measure storage for three Trident configurations from the `axonml-llm` crate's built-in presets: `tiny`, `default_150m` (a 70M-parameter "150M-class" model), and `medium` (162M parameters). Their full architectures in terms of the paper's notation (v = vocabulary size, d = hidden dimension, L = number of transformer layers, h = attention heads, f = FFN intermediate dimension) are:

- `tiny`: $v = 1000$, $d = 64$, $L = 2$, $h = 4$, $f = 256$ ($\Rightarrow$ 226,560 parameters by Eq. 5, with the final RMSNorm omitted from the count to match the reference counter in `axonml-llm`);
- `default_150m`: $v = 32{,}000$, $d = 768$, $L = 12$, $h = 12$, $f = 3072$ ($\Rightarrow$ 70,529,024 parameters);
- `medium`: $v = 32{,}000$, $d = 768$, $L = 16$, $h = 12$, $f = 3072$ ($\Rightarrow$ 162,422,784 parameters).

Note that the `tiny` preset in this table is a different model than the 69,504-parameter sanity-test model trained in Section VII.A, which uses $v = 29$, $d = 64$, $L = 2$, $h = 4$, $f = 128$ and is instantiated directly in `trident-blog/src/bin/simtest.rs` without going through the preset. The two share the hidden size and layer count but differ in vocabulary and FFN width; the `tiny` storage preset is designed to exercise the storage math at a realistic but small vocabulary, while the sanity-test model is sized to fit its 572-character corpus. Results in Table V.

Several observations:

1) **The end-to-end compression ratio increases with model size**, from 1.69× at the tiny scale to 2.89× at 162M parameters. This is because the FP32 embedding table size scales with vd while the ternary transformer body scales with $Ld^2 + Ldf$, and the latter grows faster.
2) **The transformer block weights themselves achieve the full ∼16× compression** (FP32 → 2-bit packed), but this is masked at small scales by the FP32 components.
3) **The effective bits-per-weight metric is averaged over the entire model.** The transformer blocks alone use exactly 2 bits per weight (information-theoretically suboptimal compared to the $\log_2(3) \approx 1.585$ minimum, but much simpler to decode at inference time). The much higher per-weight overhead at small scales reflects the FP32 embeddings being amortized over a small total parameter count.

D. Compute Primitive Comparison

The key structural advantage of ternary inference is not measured in MB or bits, but in the compute primitive used by the inner matmul loop. A standard FP32 linear layer requires one IEEE 754 floating-point multiplication per inner product term:

$$y_o = b_o + \sum_{j=0}^{K-1} W_{o,j} \cdot x_j$$

A ternary linear layer requires only conditional integer additions and a single FP32 multiplication per output element:

$$y_o = b_o + s \cdot \left(\sum_{j:W_{o,j}=+1} x_j - \sum_{j:W_{o,j}=-1} x_j \right)$$

For an output vector of length N from input length K, the FP32 layer requires $N \cdot K$ floating-point multiplies; the ternary layer requires N multiplies (one per element scale). The ratio is K, which equals 192 for our smallest practical configuration and 1024+ for production-scale Trident models. This is the number that motivates ternary inference on hardware without efficient FP32 multiply support: microcontrollers, FPGAs, and custom inference ASICs.

VIII. LIMITATIONS

We are explicit about what this work does *not* claim:

1) **It does not match GPT-class language models in absolute quality.** The Shakespeare model is 616K parameters trained for 100 steps. It learns the character distribution of English but does not produce coherent Shakespearean text. This is an end-to-end engineering proof-of-concept, not a scaling-laws result.
2) **The ternary matmul is not yet optimized for inference latency.** The current Rust implementation of Algorithm 1 is a straightforward scalar loop over packed weights. There is no SIMD vectorization, no GPU kernel for the ternary matmul itself, and no exploitation of weight sparsity beyond the conditional skip in the inner

loop. Note that this is orthogonal to the framework-level CUDA acceleration that AxonML provides: the FP32 operations in a Trident forward pass (attention, RMSNorm, softmax, the per-row scale multiplication, embedding lookup, and the LM head projection) all run on CUDA kernels through AxonML's standard tensor backend, and the gradient flow through the STE is fully device-aware. What is not yet accelerated is the single fused kernel that would replace the FP32 $\mathbf{X}\mathbf{W}^\top$ dense matmul with a bit-packed ternary accumulation; at present the packed weights are unpacked inside a scalar CPU loop for the ternary matmul step, while everything else in the model enjoys the framework's CUDA path. The expected end-to-end speedup of ternary inference comes from a dedicated ternary-matmul kernel that this work has not yet shipped. We measure inference latency in this paper only to characterize the current state, not as a competitive benchmark.

3) **No comparison against a same-size dense baseline.** A complete evaluation would train an FP32 Trident with identical architecture and identical training data and report the perplexity gap. We have not yet done this experiment. Based on the BitNet b1.58 paper [1], we expect the ternary model to lag the FP32 baseline at small scale and approach parity at larger scale, but this remains to be verified for the Trident architecture specifically.
4) **No long-context evaluation.** Our experiments use 96- and 128-token context windows. Long-context behavior, KV-cache quantization, and the interaction of ternary weights with rotary positional embeddings are out of scope.
5) **The training pipeline uses vanilla STE.** The original BitNet b1.58 paper [1] reports several additional techniques (loss scaling, learning rate schedule details) that we have not yet fully replicated. There is room for convergence improvements via better quantization-aware training tricks.
6) **The deployment story is described, not yet demonstrated.** The structural argument for ternary inference (no FP32 multiplies in the dot-product accumulation, fits in tiny memory, runs without a Python runtime) is correct in principle, but we have not yet shipped a binary deployment of a trained Trident model on a microcontroller or FPGA. A natural follow-up would be to take a trained checkpoint, strip the shadow weights, and run inference from a `no_std` Rust binary on embedded hardware.

Prior pure-Rust BitNet projects (Section II) either omit training, ship a placeholder backward pass, or target adapter fine-tuning. To our knowledge, Trident is the first pure-Rust BitNet b1.58 language model with a functional end-to-end training loop that converges on a public text corpus from random initialization, and the first to integrate TernaryLinear as a first-class layer inside a general-purpose Rust ML framework. That is the entirety of the claim.

IX. Conclusion

We presented Trident, an end-to-end pure-Rust implementation of a BitNet b1.58 ternary language model, including the absmean quantization function, the Straight-Through Estimator backward pass, the Trident Transformer architecture, and a complete training pipeline. We demonstrated convergence on a tiny synthetic corpus (73.5% loss drop in 23 seconds, suitable as a CI regression test) and on the complete works of Shakespeare (5.4M characters, 616K parameter model, loss 7.9→2.61 in 100 training steps over 7.9 minutes of CPU time). Storage compression ratios of 1.69–2.89× end-to-end were measured across model sizes, with the transformer block weights themselves compressing 16× FP32-to-ternary independent of scale.

The contribution is engineering, not algorithmic: the BitNet b1.58 algorithm is from [1]; we show that it can be implemented and trained from random initialization in a single pure-Rust ML framework, with no Python dependency, no libtorch wrapper, and no external ML runtime calls, and that its training loop, STE backward pass, optimizer, checkpoint system, and CUDA kernels can share their infrastructure with the rest of a general-purpose framework rather than living in a BitNet-specific standalone crate. Compared with the three prior pure-Rust BitNet efforts surveyed in Section II, which respectively (i) omit training entirely (`bitnet.rs` [18]), (ii) ship training scaffolding with a placeholder backward pass that does not compute gradients (`bitnet-rust` [19]), or (iii) target inference plus LoRA-style adapter fine-tuning of pre-quantized weights (`bitnet-quantize` [20]), Trident is, to our knowledge, the first pure-Rust BitNet b1.58 language model to converge on real text from random initialization. This matters for the deployment story of ternary LLMs, which depends on running inference on hardware that cannot execute a Python interpreter.

The next steps for this line of work are: (1) optimized SIMD and CUDA kernels for the ternary matmul to make inference latency competitive, (2) a controlled comparison against an FP32 baseline of identical architecture, (3) scaling to a 350M-parameter model trained on a real text corpus such as TinyStories or WikiText-103 and reporting perplexity against published BitNet b1.58 numbers, and (4) a deployment demonstration on embedded hardware (`no_std` Rust binary, no operating system dependencies).

All source code is released under the MIT and Apache-2.0 licenses at https://github.com/AutomataNexus/AxonML as part of the AxonML deep learning framework.

References

[1] S. Ma *et al.*, "The era of 1-bit LLMs: All large language models are in 1.58 bits," *arXiv preprint arXiv:2402.17764*, 2024.

[2] H. Wang *et al.*, "BitNet: Scaling 1-bit Transformers for large language models," *arXiv preprint arXiv:2310.11453*, 2023.

[3] M. Nielsen and S. Ma, "BitNet b1.58 2B4T: Scaling 1.58-bit LLMs," *arXiv preprint*, 2024.

[4] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *Journal of Machine Learning Research*, vol. 18, pp. 1–30, 2018.

[5] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training neural networks with weights and activations constrained to $+1$ or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[6] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. ECCV*, 2016, pp. 525–542.

[7] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pre-trained Transformers," in *Proc. ICLR*, 2023.

[8] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "AWQ: Activation-aware weight quantization for LLM compression and acceleration," *arXiv preprint arXiv:2306.00978*, 2023.

[9] G. Gerganov, "llama.cpp: Port of Facebook's LLaMA model in C/C++," https://github.com/ggerganov/llama.cpp, 2023.

[10] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2015.

[12] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," http://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015.

[13] L. Mazare, "tch-rs: Rust bindings for the C++ API of PyTorch," https://github.com/LaurentMazare/tch-rs, 2019.

[14] N. Simard, "Burn: A flexible and comprehensive deep learning framework in Rust," https://github.com/tracel-ai/burn, 2022.

[15] Hugging Face, "Candle: Minimalist ML framework for Rust," https://github.com/huggingface/candle, 2023.

[16] C. Pleasants, "dfdx: Deep learning in Rust with shape checked tensors," https://github.com/coreylowman/dfdx, 2022.

[17] A. Jewell Sr., "AxonML: A Modular Pure-Rust Deep Learning Framework with CUDA Acceleration, Distributed Training, and End-to-End Production Tooling," companion preprint, 2026. Source and PDF available at https://github.com/AutomataNexus/AxonML. This reference will be updated to the corresponding arXiv identifier via the arXiv paper-replacement mechanism once the companion paper is assigned one.

[18] Ocentra, "bitnet.rs: Pure Rust engine for BitNet LLMs — conversion, inference, and planned training," GitHub repository, created June 2025, accessed April 2026. Available: https://github.com/ocentra/bitnet.rs

[19] "bitnet-rust / bitnet-mlx.rs: A high-performance Rust implementation of BitNet b1.58 neural networks with Candle, MLX, and ANE support," GitHub repository, created July 2025, accessed April 2026. Available: https://github.com/wavegoodvybe2929/bitnet-rust

[20] T. Zervas, "bitnet-quantize: Microsoft BitNet b1.58 quantization and inference for Rust," crates.io v0.2.1, January 2026. Available: https://crates.io/crates/bitnet-quantize