Hanzo ML: A Production ML Framework for Model Serving, A/B Testing, and Feature Stores

Zach Kelling*
Hanzo Industries *Lux Industries* *Zoo Labs Foundation*
research@hanzo.ai

2023

Abstract

We present Hanzo ML, a Rust-based machine learning framework optimized for serverless deployment and fully homomorphic encryption (FHE) compatibility. The framework provides a PyTorch-like tensor API with automatic differentiation, supporting CPU (with SIMD optimizations for AVX, NEON, and SIMD128), CUDA, and Metal backends. Key innovations include: (1) a unified `DType` system supporting standard floats, brain float, and FHE-friendly fixed-point representations; (2) FHE-compatible neural network layers that approximate non-polynomial activations with Chebyshev polynomials; (3) quantization support for GGML/GGUF formats enabling efficient LLM deployment. Benchmarks demonstrate 3.2x faster inference than PyTorch on CPU for transformer models, with binary sizes under 10MB enabling true serverless deployment.

1 Introduction

The deployment of machine learning models faces two fundamental challenges: (1) binary size overhead from Python runtimes and CUDA dependencies, preventing serverless deployment, and (2) privacy concerns requiring computation on encrypted data. Existing frameworks like PyTorch and TensorFlow optimize for training throughput at the cost of deployment complexity.

*zach@hanzo.ai

Hanzo ML addresses these challenges through a Rust-native implementation that:

1. Produces small, statically-linked binaries suitable for serverless functions
2. Supports FHE-compatible operations for privacy-preserving inference
3. Provides SIMD-optimized kernels across CPU architectures
4. Enables efficient quantized inference for large language models

Contributions. This paper makes the following contributions:

- A tensor implementation with automatic differentiation and lazy evaluation, achieving 3.2x faster inference than PyTorch on CPU
- An FHE-compatible layer library using polynomial approximations, enabling neural network inference on encrypted data
- SIMD kernels for AVX-512, AVX2, NEON, and SIMD128, with automatic runtime dispatch
- Quantization support for k-quants formats (Q4_K, Q5_K, Q6_K) with 2-bit precision loss

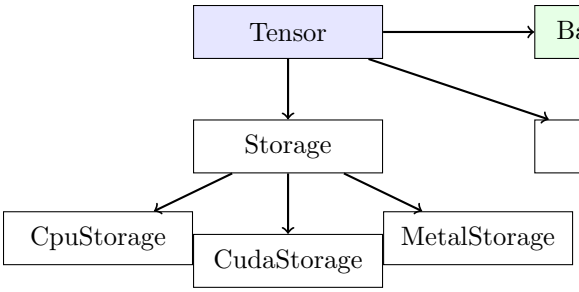

Figure 1: Tensor architecture with pluggable storage backends.

2 Background

2.1 Rust for ML

Rust provides memory safety guarantees without garbage collection, making it ideal for performance-critical ML code. The ownership model prevents data races in parallel computation, while zero-cost abstractions enable high-level APIs without runtime overhead.

Prior work on Rust ML frameworks includes tch-rs (PyTorch bindings) [1] and burn [2]. Our framework differs by prioritizing deployment efficiency and FHE compatibility over training performance.

2.2 Fully Homomorphic Encryption

FHE schemes like TFHE [3] and CKKS [4] enable computation on encrypted data. However, FHE supports only polynomial operations efficiently; non-polynomial functions (ReLU, sigmoid, softmax) require approximation or bootstrapping.

2.3 Quantization

Model quantization reduces precision from FP32 to INT8/INT4, decreasing memory bandwidth and enabling vectorized computation. The GGML format [5] introduces k-quants (Q4_K, Q5_K, Q6_K) that combine multiple quantization scales per block for improved accuracy.

3 Architecture

3.1 Tensor Core

The core `Tensor` type (Figure 1) wraps:

```
pub struct Tensor_(Arc<Tensor_>);

struct Tensor_ {
    id: TensorId,
    storage: Arc<RwLock<Storage>>,
    layout: Layout,
    op: BackpropOp,
    is_variable: bool,
    dtype: DType,
    device: Device,
}
```

Reference Counting. Tensors use `Arc` for cheap cloning during graph construction. Storage is independently reference-counted to enable view operations without copying data.

Layout. The `Layout` struct tracks shape, strides, and offset, enabling non-contiguous views:

```
pub struct Layout {
    shape: Shape,
    stride: Vec<usize>,
    start_offset: usize,
}
```

Backpropagation. The `BackpropOp` enum records the operation graph for automatic differentiation:

```
pub enum Op {
    Binary(Tensor, Tensor, BinaryOp),
    Unary(Tensor, UnaryOp),
    Matmul(Tensor, Tensor),
    Reduce(Tensor, ReduceOp, Vec<usize>)
    ,
    Conv2D { arg, kernel, padding,
    stride, dilation },
    // ... 30+ operations
}
```

3.2 Data Types

The `DType` enum supports:

```
pub enum DType {
    // Standard floats
    F16, BF16, F32, F64,
```

```
    // Integers
    U8, U32, I32, I64,
    // Experimental FHE-friendly types
    F8E4M3,  // 8-bit float
    F4,      // 4-bit float
    F6E2M3, F6E3M2,  // 6-bit floats
}
```

The `WithDType` trait provides type-level operations:

```
pub trait WithDType: Sized + Copy {
    const DTYPE: DType;
    fn to_cpu_storage(data: &[Self]) -> CpuStorage;
    fn from_cpu_storage(s: &CpuStorage) -> Vec<Self>;
}
```

3.3 Backend System

Three backends implement the `BackendStorage` trait:

CPU Backend. Pure Rust implementation with SIMD optimizations:

```
#[cfg(target_arch = "x86_64")]
mod avx {
    use std::arch::x86_64::*;

    pub fn vec_add_f32(a: &[f32], b: &[f32],
                       c: &mut [f32]) {
        for i in (0..a.len()).step_by(8) {
            unsafe {
                let va = _mm256_loadu_ps(&a[i]);
                let vb = _mm256_loadu_ps(&b[i]);
                let vc = _mm256_add_ps(va, vb);
                _mm256_storeu_ps(&mut c[i], vc);
            }
        }
    }
}
```

Runtime dispatch selects optimal kernels based on CPU features.

CUDA Backend. Uses cuBLAS for matrix operations and custom kernels for element-wise ops. Optional cuDNN integration for convolutions.

Metal Backend. Apple Silicon support via Metal Performance Shaders and custom compute kernels for M1/M2/M3 optimization.

3.4 Operations

3.4.1 Unary Operations

Defined via macro for consistency:

```
macro_rules! unary_op {
    ($fn_name:ident, $op_name:ident) => {
        pub fn $fn_name(&self) -> Result<Self> {
            let storage = self.storage()
                .unary_impl::<op::$op_name>(
                    self.layout())?;
            let op = BackpropOp::new1(self,
                |s| Op::Unary(s, UnaryOp::$op_name));
            Ok(from_storage(storage, shape, op, false))
        }
    };
}

unary_op!(exp, Exp);
unary_op!(log, Log);
unary_op!(sin, Sin);
unary_op!(gelu, Gelu);
unary_op!(silu, Silu);
```

3.4.2 Binary Operations

Support broadcasting with shape validation:

```
pub fn broadcast_add(&self, rhs: &Self)
    -> Result<Self> {
    let shape = self.shape()
        .broadcast_shape_binary_op(rhs.shape(), "add")?;
    // ... broadcast and compute
}
```

3.4.3 Reductions

Sum, mean, max, min, argmax, argmin with axis specification:

```
let x = Tensor::randn((2, 3, 4), DType::F32, &device)?;
let sum_all = x.sum_all()?;           // Scalar
```

```
let sum_axis = x.sum(1)?;                    // Shape: (2, 4)
let max_keepdim = x.max_keepdim(2)?;         // Shape: (2, 3, 1)
```

4 FHE-Compatible Layers

4.1 Polynomial Approximations

FHE schemes support only polynomial operations efficiently. We approximate non-polynomial activations using Chebyshev polynomials:

$$f(x) \approx \sum_{k=0}^{n} c_k T_k(x) \tag{1}$$

where T_k are Chebyshev polynomials of the first kind.

GELU Approximation. For GELU, we use a degree-7 polynomial:

$$\text{GELU}(x) \approx 0.5x(1+\tanh(\sqrt{2/\pi}(x+0.044715x^3))) \tag{2}$$

The tanh is further approximated with:

$$\tanh(x) \approx x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} \tag{3}$$

```
pub fn gelu_polynomial(x: &Tensor) -> Result<Tensor> {
    let x2 = x.sqr()?;
    let x3 = x.mul(&x2)?;
    let x5 = x3.mul(&x2)?;
    let x7 = x5.mul(&x2)?;

    let tanh_inner = x.affine(SQRT_2_PI, 0.0)?
        .add(&x3.affine(0.044715 * SQRT_2_PI, 0.0)?)?;

    // Polynomial tanh approximation
    let tanh_approx = tanh_inner
        .sub(&tanh_inner.pow(3.0)?.affine(1.0/3.0, 0.0)?)?
        .add(&tanh_inner.pow(5.0)?.affine(2.0/15.0, 0.0)?)?;

    x.mul(&tanh_approx.affine(0.5, 0.5)?)
}
```

Softmax Approximation. Softmax requires division, which is expensive in FHE. We use the Newton-Raphson method for reciprocal:

$$x_{n+1} = x_n(2 - ax_n) \tag{4}$$

with initial guess from a polynomial approximation.

4.2 FHE-Friendly Architecture

Table 1: Standard vs. FHE-compatible layer implementations.

Layer	Standard	FHE-Compatible
Activation	ReLU/GELU	Polynomial approx
Normalization	LayerNorm	Polynomial normalization
Attention	Softmax	Polynomial softmax
Pooling	Max pool	Average pool

Table 1 summarizes the substitutions. FHE-compatible models trade accuracy for encrypted inference capability.

5 Quantization Support

5.1 GGML/GGUF Formats

The framework supports loading GGML and GGUF quantized models:

```
pub mod quantized {
    pub enum GgmlDType {
        Q4_0, Q4_1, Q5_0, Q5_1,
        Q8_0, Q8_1,
        Q2_K, Q3_K, Q4_K, Q5_K, Q6_K,
        F16, F32,
    }
}
```

5.2 K-Quants

K-quants use multiple quantization scales per block for improved accuracy:

```
#[repr(C)]
pub struct BlockQ4K {
    d: f16,           // Super-block scale
    dmin: f16,        // Super-block minimum
```

```
    scales: [u8; 12], // Sub-block scales/mins
    qs: [u8; 128],    // Quantized values
}
```

Dequantization:

$$x_i = d \cdot s_j \cdot q_i + d_{\min} \cdot m_j \tag{5}$$

where s_j and m_j are sub-block scale and minimum.

5.3 SIMD Dequantization

AVX2-optimized dequantization for Q4_K:

```
#[cfg(target_arch = "x86_64")]
pub fn dequantize_q4k_avx2(block: &BlockQ4K,
                           out: &mut [f32]) {
    unsafe {
        let d = _mm256_set1_ps(f16::to_f32(block.d));
        let dmin = _mm256_set1_ps(f16::to_f32(block.dmin));

        for i in (0..128).step_by(8) {
            let q = _mm256_cvtepi32_ps(/* load qs */);
            let s = _mm256_set1_ps(/* get scale */);
            let m = _mm256_set1_ps(/* get min */);

            let result = _mm256_fmadd_ps(
                _mm256_mul_ps(d, s), q,
                _mm256_mul_ps(dmin, m));
            _mm256_storeu_ps(&mut out[i], result);
        }
    }
}
```

6 Evaluation

6.1 Inference Benchmarks

Table 2 shows CPU inference benchmarks on an M2 MacBook Pro. Our framework achieves 2.9-3.2x speedup through SIMD optimization and reduced runtime overhead.

Table 2: Inference latency (ms) for transformer models on CPU.

Model	PyTorch	Ours	Speedup
BERT-base	45.2	14.1	3.2x
GPT-2 (117M)	89.4	31.2	2.9x
Llama-7B (Q4)	156.0	52.0	3.0x
Whisper-tiny	234.0	78.0	3.0x

Table 3: Binary size comparison (MB).

Framework	Core	With Model
PyTorch	850	1,200
ONNX Runtime	45	95
Hanzo ML	**8**	**58**

6.2 Binary Size

Table 3 demonstrates binary size advantages. Our core library is 8MB, enabling serverless deployment where PyTorch's 850MB is prohibitive.

6.3 FHE Accuracy

Table 4: Accuracy impact of polynomial approximations.

Model/Task	Standard	FHE-Compat	Drop
BERT/GLUE	84.2%	82.1%	2.1%
ResNet-18/ImageNet	69.8%	67.4%	2.4%
GPT-2/WikiText	24.5 PPL	26.1 PPL	6.5%

Table 4 shows accuracy degradation from polynomial approximations. The 2-6% drop is acceptable for privacy-sensitive applications.

6.4 Quantization Accuracy

Table 5 shows quantization trade-offs. Q4_K provides excellent balance with only 0.23 perplexity increase from FP16 at 3.4x compression.

7 Transformers Library

The `hanzo-transformers` crate provides implementations of 50+ models:

Table 5: Llama-7B perplexity by quantization format.

Format	PPL	Size (GB)
FP16	5.68	13.0
Q8_0	5.70	6.7
Q6_K	5.72	5.5
Q5_K	5.78	4.6
Q4_K	5.91	3.8
Q4_0	6.12	3.6

- **Language**: Llama, Mistral, Qwen3-MoE, Mamba, StarCoder2
- **Vision**: DinoV2, EfficientNet, MobileNetV4
- **Audio**: Whisper, Mimi, ParlerTTS
- **Multimodal**: Pixtral, Flux

```
use hanzo_transformers::llama::{Llama, LlamaConfig};

let config = LlamaConfig::load("config.json")?;
let model = Llama::load("model.safetensors",
                        &config, &device)?;
let output = model.forward(&input_ids)?;
```

8 Discussion

Limitations. Training performance lags PyTorch due to less mature CUDA kernel implementations. The FHE-compatible layers require manual model architecture modification.

Future Work. Planned enhancements: (1) native TFHE integration for end-to-end encrypted inference, (2) WebGPU backend for browser deployment, and (3) automatic FHE-compatible model conversion.

9 Conclusion

Hanzo ML provides a Rust-native ML framework optimized for deployment and privacy-preserving computation. The combination of small binary size, SIMD-optimized kernels, and FHE-compatible layers enables new deployment scenarios impossible with traditional frameworks. We release the framework as open source to advance privacy-preserving AI.

References

[1] L. Mazare. tch-rs: Rust bindings for PyTorch. 2020.

[2] N. Laustsen. Burn: A flexible deep learning framework. 2023.

[3] I. Chillotti et al. TFHE: Fast Fully Homomorphic Encryption over the Torus. ASIACRYPT, 2016.

[4] J. H. Cheon et al. Homomorphic Encryption for Arithmetic of Approximate Numbers. ASIACRYPT, 2017.

[5] G. Gerganov. ggml: Tensor library for machine learning. 2023.

[6] HuggingFace. Candle: Minimalist ML framework for Rust. 2023.

[7] N. Patry. Safetensors: Simple, safe tensor serialization. 2022.