AxonML: A Modular Pure-Rust Deep Learning Framework with CUDA Acceleration, Distributed Training, and End-to-End Production Tooling

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

***Abstract*—We present AxonML (v0.6.1), an open-source deep learning framework implemented entirely in Rust, spanning 22 crates and 226,373 lines of code. The framework provides a complete machine learning stack: N-dimensional tensor operations with CUDA GPU acceleration via 15 PTX kernel modules, reverse-mode automatic differentiation with gradient checkpointing and automatic mixed precision, 41 neural network layer types including state-of-the-art mechanisms (rotary embeddings, grouped-query attention, mixture-of-experts, differential attention, ternary quantized layers), 5 optimizers with 7 learning rate schedulers, and 7 loss functions. AxonML includes nine large language model architectures (GPT-2, LLaMA, Mistral, Phi, BERT, SSM/Mamba, Hydra, Trident, Chimera), six model quantization formats (Q8_0, Q4_0, Q4_1, Q5_0, Q5_1, F16) with 1.58-bit ternary support, distributed training (DDP, FSDP, pipeline parallelism), ONNX import/export with 39 operators, model serialization to bincode/JSON/SafeTensors, a built-in HTTP training monitor with live browser dashboards, and a 33-subcommand CLI. The workspace contains 2,285 tests. AxonML has been used to build complete end-to-end projects in machine translation (two contrasting Akkadian-English translation systems built on identical underlying primitives), face recognition, and acoustic species identification, validating its completeness as a research and production framework. AxonML is dual-licensed under MIT and Apache-2.0.**

***Index Terms*—deep learning framework, Rust, CUDA acceleration, automatic differentiation, neural networks, distributed training, model quantization, ONNX, mixture of experts**

I. Introduction

The dominant deep learning frameworks—PyTorch [3], TensorFlow [4], and JAX [5]—are predominantly Python-based with C++ and CUDA backends. While extraordinarily capable for research, this architecture introduces friction in production deployment: a Python runtime is required, the Global Interpreter Lock constrains concurrent workloads, dependency management becomes complex, and shipping models to edge devices requires either a complete Python environment or model conversion to a separate inference runtime.

Rust offers compelling alternatives: memory safety without garbage collection, zero-cost abstractions, fearless concurrency through its ownership model, and a strong type system that eliminates entire classes of runtime errors. These properties are attractive for ML systems, particularly when deploying models on edge devices, embedded systems, or in security-sensitive environments where Python's runtime characteristics are undesirable.

We present AxonML, a complete deep learning framework implemented entirely in Rust, designed for both research experimentation and production deployment. Our contributions are:

1) **A complete ML stack in 22 composable crates** (226,373 lines of code), spanning low-level tensor operations through high-level model architectures, with no Python dependencies and no C++ wrapper layers.
2) **Native CUDA GPU acceleration** via 15 PTX kernel modules implementing element-wise operations, broadcasting, activations, reductions, normalization, convolution, recurrent gates, attention, and fused optimizer updates. The framework integrates cuBLAS for matrix multiplication and provides a CUDA memory pool to reduce allocator overhead during training.
3) **Modern architectural building blocks**: rotary positional embeddings, grouped-query attention, mixture-of-experts routing with load balancing, differential attention, sparse linear layers, ternary quantized layers, and graph neural network operators.
4) **Domain-specific model libraries** for computer vision (ResNet, VGG, ViT, BlazeFace, DETR, NanoDet, RetinaFace, biometric encoders), audio processing (mel spectrograms, speaker verification, dataset loaders), text processing (tokenization, embeddings, sequence-to-sequence transformers), and large language models (nine architectures from GPT-2 through state-of-the-art ternary-weight Trident).
5) **Production tooling**: model serialization with checkpoint/resume support, ONNX interoperability, INT8 and sub-byte quantization, distributed training primitives (DDP, FSDP, pipeline parallelism), a built-in training monitor with live browser dashboards, and a 33-subcommand CLI for training, inference, conversion, profiling, and deployment.

Scope of this paper. This paper is a system architecture and API design document, not a performance benchmarking paper. We focus on the architectural design, API ergonomics, modularity, and breadth of components required to build a

TABLE I
AXONML WORKSPACE CRATES.

Crate	Purpose
axonml-core	Device management, CUDA backend, memory pools
axonml-tensor	N-dimensional tensor operations
axonml-autograd	Reverse-mode automatic differentiation
axonml-nn	Neural network layers (41 types)
axonml-optim	Optimizers and LR schedulers
axonml-data	Datasets, data loaders, transforms
axonml-vision	Vision models and datasets
axonml-audio	Audio transforms and datasets
axonml-text	Text processing and tokenization
axonml-llm	LLM models (9 architectures)
axonml-serialize	Checkpoint and model serialization
axonml-onnx	ONNX import/export (39 operators)
axonml-quant	Model quantization (Q4, Q5, Q8)
axonml-fusion	Tensor fusion utilities
axonml-distributed	DDP, FSDP, pipeline parallelism
axonml-profile	Performance profiling
axonml-jit	JIT compilation primitives
axonml-cli	33-subcommand command-line interface
axonml-tui	Terminal UI for training
axonml-server	REST API server for inference and training
axonml-dashboard	Web dashboard (publish = false)
axonml	Umbrella crate with re-exports and TrainingMonitor

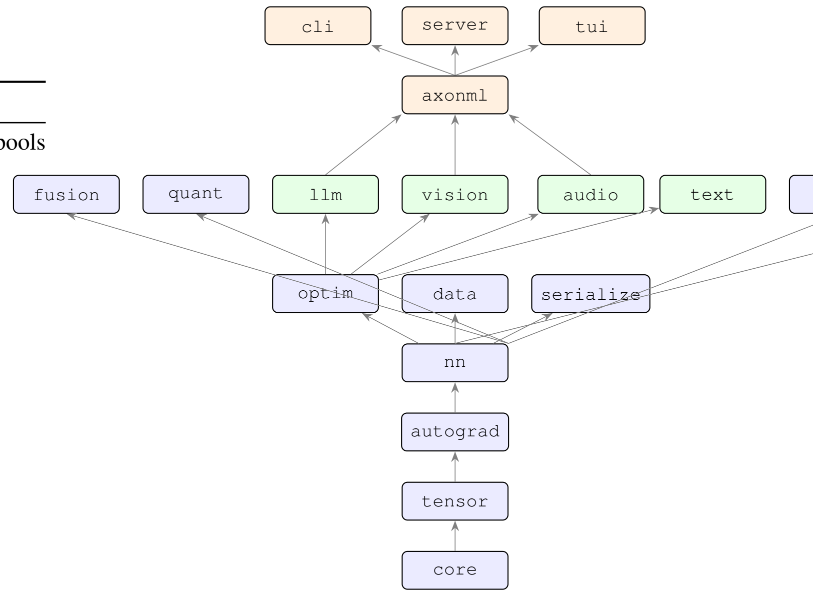

Fig. 1. AxonML crate dependency graph. Foundation crates (blue) provide tensor, autograd, and neural-network primitives; domain libraries (green) build vision, audio, text, and LLM models on the shared core; tooling crates (orange) wrap everything for end-user workflows. The umbrella `axonml` crate re-exports the full stack with feature gates. Arrows point in the direction of compilation dependency.

complete pure-Rust ML ecosystem from scratch. Rigorous head-to-head throughput and memory benchmarks against PyTorch baselines on standardized workloads are deferred to future work, as discussed in Section XII. The contribution is the engineering of a complete, type-safe, GPU-accelerated training and inference stack in a single language, demonstrated by the diverse end-to-end applications of Section X.

Code and artifacts. AxonML is open source at https://github.com/AutomataNexus/AxonML under the MIT and Apache-2.0 licenses, with crates published to crates.io. The framework has been used to train models across diverse domains, validating its completeness. Section II describes the overall workspace architecture. Section III details the tensor system. Section IV describes the automatic differentiation engine. Section V enumerates the neural network layers. Section VI describes the CUDA acceleration layer. Section VII describes training infrastructure. Section VIII describes domain-specific libraries. Section IX describes deployment tooling. Section X presents applications. Section XI surveys related work. Section XII discusses limitations and ongoing work. Section XIII describes availability. Section XIV concludes.

II. WORKSPACE ARCHITECTURE

AxonML is organized as a Cargo workspace with 22 crates (Table I), each with a single, well-defined responsibility. This modular structure enables selective compilation, minimal dependency footprints for deployment scenarios, and clear separation between research-oriented and production-oriented components.

The workspace is built with Rust edition 2024 and a minimum supported Rust version of 1.85. The workspace contains 2,285 tests across all crates. AxonML is dual-licensed under MIT and Apache-2.0.

The dependency graph forms a directed acyclic structure (Figure 1): `axonml-core` provides device abstractions and the CUDA backend; `axonml-tensor` depends only on core; `axonml-autograd` depends on tensor; `axonml-nn` depends on autograd; `axonml-optim`, `axonml-data`, and `axonml-serialize` depend on `axonml-nn`; domain crates (`axonml-vision`, `axonml-audio`, `axonml-text`, `axonml-llm`) depend on the core neural network stack; the umbrella `axonml` crate re-exports everything with feature gates.

III. TENSOR SYSTEM

The `axonml-tensor` crate provides the foundational `Tensor<f32>` type, an N-dimensional array supporting both CPU and CUDA GPU execution with transparent device transfer. The crate contains 112 tests.

A. Storage Model

A tensor consists of three components: a contiguous storage buffer (CPU `Vec<f32>` or CUDA device pointer), a shape vector specifying the dimensions, and a stride vector enabling non-contiguous views without copying. The storage type is dispatched at runtime via an enum, allowing the same operation API regardless of device:

```
let cpu_tensor = Tensor::<f32>::zeros(&[32, 128]);
let gpu_tensor = cpu_tensor.to_device(Device::Cuda
    (0))?;
let result = gpu_tensor.matmul(&weights);
```

Operations dispatch at the tensor method level: when both operands reside on a CUDA device, the GPU implementation is invoked; when on CPU, the BLAS-backed implementation runs via the `matrixmultiply` crate.

B. Operation Categories

The tensor crate provides several operation categories:

- **Creation**: `zeros`, `ones`, `full`, `arange`, `linspace`, `rand`, `randn`, `from_vec`, `eye`.
- **Shape manipulation**: `reshape`, `squeeze`, `unsqueeze`, `transpose`, `permute`, `flatten`, `contiguous`.
- **Views and slicing**: `slice`, `narrow`, `select`, `chunk`, `stack`, `cat`, `split`.
- **Element-wise arithmetic**: `add`, `sub`, `mul`, `div`, with scalar variants and full broadcasting support.
- **Reductions**: `sum`, `sum_dim`, `mean`, `mean_dim`, `var_dim`, `max`, `min`, `argmax`, `argmin`.
- **Linear algebra**: `matmul`, `dot`, batched matrix multiply.
- **Activation functions**: `relu`, `sigmoid`, `tanh`, `gelu`, `leaky_relu`, `elu`, `silu`, `softmax`, `log_softmax`.

C. Sparse and Lazy Tensors

In addition to dense tensors, the crate provides sparse tensor support via COO and CSR formats with operations including `coalesce`, `to_dense`, sparse-dense matrix multiply, and `eye` for sparse identity construction. A `LazyTensor` type defers computation through a `LazyOp` graph, enabling fusion opportunities for the JIT crate.

IV. Automatic Differentiation

The `axonml-autograd` crate implements reverse-mode automatic differentiation via a dynamic computational graph, matching PyTorch's eager execution semantics. The crate contains 132 tests.

A. Variable Type and Graph Construction

The `Variable` type wraps a tensor with optional gradient tracking. Each operation that consumes `Variable` inputs records a backward closure (a `GradFn`) that computes gradients during the backward pass:

```
let x = Variable::new(tensor, true); //
    requires_grad=true
let w = Variable::new(weights, true);
let y = x.matmul(&w).relu();
let loss = y.mean();
loss.backward();
let dx = x.grad().unwrap();
```

B. GradientFunction Trait

The core abstraction is the `GradientFunction` trait:

```
pub trait GradientFunction: Debug + Send + Sync {
    fn apply(&self, grad_output: &Tensor<f32>)
        -> Vec<Option<Tensor<f32>>>;
    fn name(&self) -> &'static str;
    fn next_functions(&self) -> &[Option<GradFn>];
    fn as_any(&self) -> &dyn Any;
}
```

Each operation provides a backward implementation that computes input gradients given output gradients. The `next_functions` field links to upstream operations, forming the reverse graph used during backpropagation. Leaf variables use a special `AccumulateGrad` function that stores gradients into the leaf's gradient buffer.

C. Graph Inspection and Debugging

The autograd crate includes debugging utilities not commonly found in other Rust ML frameworks:

- `ComputationGraph`: Traces the backward graph from a root variable, enabling structural analysis.
- `GraphSnapshot`: Captures graph structure with depth, node count, and operation type histograms for debugging.
- `no_grad` context manager: Disables gradient tracking for inference-only computations, reducing memory overhead.

D. Gradient Checkpointing and Mixed Precision

For memory-constrained training, the crate provides gradient checkpointing via the `checkpoint.rs` module. Activations are recomputed during the backward pass instead of stored, trading compute for memory. This is particularly useful for transformer training where attention activations dominate memory usage.

Automatic Mixed Precision (AMP) is supported via the `amp.rs` module, which provides an `autocast` context that automatically casts operations to lower precision where safe.

V. Neural Network Layers

The `axonml-nn` crate provides 41 layer implementations spanning classical and modern architectures (Table II). Every layer implements the `Module` trait:

```
pub trait Module: Send + Sync {
    fn forward(&self, input: &Variable) -> Variable;
    fn parameters(&self) -> Vec<Parameter> { ... }
    fn named_parameters(&self) -> HashMap<String,
        Parameter> { ... }
    fn train(&mut self) { ... }
    fn eval(&mut self) { ... }
}
```

TABLE II
Neural network layer categories (41 total).

Category	Count
Convolution (Conv1d, Conv2d, ConvTranspose2d)	3
Recurrent (LSTM, GRU, RNN, +Cell variants)	6
Attention (MultiHead, Cross, Differential)	3
Transformer (Encoder, Decoder, Seq2Seq)	5
Normalization (Batch1d/2d, Layer, Group, Instance)	5
Pooling (Max1d/2d, Avg1d/2d, AdaptiveAvg2d)	5
Regularization (Dropout, Dropout2d, AlphaDropout)	3
Linear, Embedding	2
Specialized (Residual, MoE, FFT, STFT, GNN, Sparse, Ternary)	9
Total	**41**

A. Modern Attention Mechanisms

AxonML provides three attention implementations:

MultiHeadAttention is the standard scaled dot-product attention with multi-head projections and optional batch-first input layout. CUDA acceleration uses the fused attention kernel from `ATTENTION_PTX`.

CrossAttention computes attention between separate query and key-value sequences, used in encoder-decoder architectures.

DifferentialAttention [6] computes the difference between two softmax attention maps with a learnable lambda parameter ($A_1 - \lambda A_2$), reducing attention noise. This mechanism is used in the Chimera LLM architecture.

B. Mixture of Experts

The `MoELayer` implements sparse mixture-of-experts routing for scaling model capacity without proportional compute increase. Each expert is a SwiGLU MLP (gate projection $\odot$ SiLU(up projection) $\rightarrow$ down projection). A learned router produces softmax routing probabilities, and the top-k experts are selected per token.

The MoE layer includes a load balancing auxiliary loss to prevent expert collapse:

$$\mathcal{L}_{\text{bal}} = N_{\text{experts}} \sum_i f_i P_i \tag{1}$$

where f_i is the fraction of tokens routed to expert i and P_i is the mean routing probability for expert i. Configuration: 8 experts with top-2 routing yields 25% of full-capacity compute per token while maintaining the parameter capacity of all experts.

C. Quantized Layers

`TernaryLinear` restricts weights to $\{-1, 0, +1\}$, enabling inference via addition and subtraction without floating-point multiplications. Training uses the Straight-Through Estimator to backpropagate through the discretization. This layer is used by the Trident LLM (Section VIII-B) to achieve 16$\times$ memory compression.

`SparseLinear` stores weights in CSR format, dispatching sparse matrix-vector products instead of dense GEMM. This is useful for pruned models with high sparsity.

D. Specialized Layers

`GATConv` and `GCNConv` implement graph attention and graph convolution layers for graph neural network workloads. `FFT1d` and `STFT` provide differentiable frequency-domain transforms for audio and signal processing. `ResidualBlock` provides a generic residual connection wrapper.

VI. CUDA Acceleration

GPU acceleration in AxonML is implemented via 15 PTX (Parallel Thread Execution) kernel modules, compiled at build time and loaded into the CUDA context via the `cudarc` crate. Table III enumerates the modules.

TABLE III
PTX KERNEL MODULES IN AXONML-CORE.

Module	Operations
ELEMENTWISE	add, sub, mul, div, scalar variants
BROADCAST	broadcast variants of binary ops
ACTIVATIONS	relu, sigmoid, tanh, gelu, silu, exp, log + backward
REDUCTION	sum, mean, max, argmax (global)
SUM_DIM	per-dimension sum and mean
LAYERNORM	fused LayerNorm forward and backward
CROSS_ENTROPY	fused softmax cross-entropy with reduction
EMBEDDING_SCATTER	embedding lookup and gradient scatter
ADAM	fused Adam parameter update
STRIDED_COPY	non-contiguous tensor copy
MASK	masked operations for attention
CONV	im2col-based Conv2d
LSTM	fused LSTM gate computation
POOLING	max and average pooling forward + backward
ATTENTION	fused scaled dot-product attention

A. Backend and Memory Management

The CUDA backend (`axonml-core/src/backends/cuda.rs`) provides over 100 public functions wrapping CUDA operations. Each kernel takes pointer-based input and output buffers and a launch configuration specifying block and grid dimensions.

A CUDA memory pool (`cuda_pool.rs`) caches and reuses freed device allocations, reducing the overhead of `cudaMalloc` and `cudaFree` during training. Without pooling, allocator overhead can dominate iteration time for small models with frequent allocations.

B. cuBLAS Integration

For matrix multiplication, AxonML wraps cuBLAS through the `CudaBlas` type, providing `gemm_f32`, `gemm_batched_f32`, and `gemm_strided_batched_f32`. The strided batched variant is used in attention computation where the same operation must be applied across multiple heads with shared memory layout.

C. Convolution via cuDNN

`Conv2d` dispatches to cuDNN through the optional `Cudnn` backend when present, using the cuDNN convolution descriptor and tensor descriptor APIs. `Conv1d` reuses the Conv2d CUDA implementation by reshaping its 3D input ($[B, C, L]$) to 4D ($[B, C, L, 1]$) and treating the spatial dimension as height-1, avoiding the need for a separate Conv1d kernel.

D. Fused Optimizer Updates

The `ADAM_PTX` kernel performs the complete Adam parameter update in a single kernel launch, fusing the moment estimates, bias correction, and parameter update steps. This avoids the round-trips between distinct kernels that would otherwise be required.

VII. Training Infrastructure

A. Optimizers

AxonML provides five optimizers in `axonml-optim`:

Adam [7]: First and second moment estimates with bias correction. Configurable via `Adam::new(params, lr)` or `Adam::with_options(params, lr, betas, eps, weight_decay, amsgrad)`. Supports the AMSGrad variant which tracks the maximum of past second moment estimates for stability.

AdamW [8]: Adam with decoupled weight decay applied directly to parameters rather than through gradients, providing better generalization on transformer training.

SGD: Stochastic gradient descent with optional momentum, weight decay, dampening, and Nesterov acceleration. Configurable via `SGD::with_options(params, lr, momentum, weight_decay, dampening, nesterov)`.

RMSprop: Moving average of squared gradients with optional centering and momentum.

LAMB [9]: Layer-wise Adaptive Moments for Batch training, computing per-layer trust ratios $\|\text{param}\|/\|\text{update}\|$ to enable stable training with very large batch sizes (up to tens of thousands).

All optimizers maintain per-parameter state on the same device as the parameters themselves, avoiding CPU–GPU round-trips during training.

B. Learning Rate Schedulers

Seven learning rate schedulers are available in `lr_scheduler.rs`:

1) `StepLR`: Decay by factor γ every N epochs.
2) `MultiStepLR`: Decay at specified milestone epochs.
3) `ExponentialLR`: Geometric decay each epoch.
4) `CosineAnnealingLR`: Cosine annealing from initial LR to minimum.
5) `ReduceLROnPlateau`: Adaptive reduction when a monitored metric plateaus.
6) `OneCycleLR` [10]: Single-cycle warmup-then-cooldown for super-convergence.
7) `WarmupLR`: Linear warmup followed by a base schedule.

C. Loss Functions

The `axonml-nn` crate provides 7 loss functions: MSELoss, L1Loss, CrossEntropyLoss, NLLLoss, BCELoss, BCEWithLogitsLoss, and SmoothL1Loss. Each provides both `compute()` (raw `f32` for inference) and `compute_var()` (graph-tracked `Variable` for training) interfaces, with configurable reduction modes (None, Mean, Sum). CrossEntropyLoss uses a fused softmax-and-loss CUDA kernel for efficiency.

D. Serialization and Checkpointing

The `axonml-serialize` crate provides three serialization formats:

- **Axonml binary**: Bincode-serialized state dictionaries, compact and fast.
- **JSON**: Human-readable, useful for debugging and interoperability.
- **SafeTensors**: Safe, layout-stable format for cross-framework compatibility (optional feature).

A `Checkpoint` bundles a `StateDict` (model parameters), an optimizer state dictionary, and a `TrainingState` (epoch, step counter, loss history, best metric, custom metric histories). This enables resume from arbitrary checkpoints, essential for long training runs.

E. Training Monitor

A unique feature of AxonML is the built-in `TrainingMonitor`, a small HTTP server that serves a live browser dashboard on a configurable port. The monitor exposes a JSON metrics endpoint and an HTML dashboard with no external dependencies:

```
let monitor = TrainingMonitor::new("MyModel",
    param_count)
    .total_epochs(50)
    .batch_size(32)
    .launch();

for epoch in 0..50 {
    let train_loss = train_one_epoch(...);
    let val_loss = validate(...);
    monitor.log_epoch(
        epoch + 1,
        train_loss,
        Some(val_loss),
        vec![("acc", accuracy), ("lr", current_lr)],
    );
}
monitor.set_status("complete");
```

The monitor tracks training loss, optional validation loss, arbitrary named extra metrics, and best loss seen. The dashboard updates live without requiring a separate web framework or external services.

F. Distributed Training

The `axonml-distributed` crate provides three distributed training paradigms:

DistributedDataParallel (DDP): Replicates the model across processes, partitions the input batch, and synchronizes gradients via all-reduce after each backward pass. Synchronization strategies include synchronous all-reduce and bucketed gradient communication for overlap with computation.

FullyShardedDataParallel (FSDP): Shards model parameters across processes, gathering them temporarily during forward and backward passes. Supports CPU offload for additional memory savings. Sharding strategies are configurable via `ShardingStrategy`.

Pipeline Parallelism: Splits the model across processes by layer, with micro-batch scheduling to maintain pipeline utilization. Includes `PipelineStage`, `PipelineSchedule`, and `PipelineMemoryStats` for performance tuning.

Collective operations include `all_reduce` (sum, mean, min, max, product), `all_gather`, `reduce_scatter`,

broadcast, barrier, gather, and scatter. The crate supports an optional NCCL backend (NcclBackend) and a mock backend for testing.

VIII. Domain Libraries

A. Vision (axonml-vision)

The vision crate (741 tests) provides over 20 model architectures across classification, detection, anomaly detection, depth estimation, and biometrics:

- **Classification**: LeNet, SimpleCNN, MLP, ResNet (ResNet18, ResNet34 with BasicBlock and Bottleneck variants), VGG (VGG11, VGG13, VGG16, VGG19), Vision Transformer (ViT-Base, ViT-Large with sinusoidal positional encoding).
- **Object Detection**: BlazeFace (lightweight face detection), DETR (Detection Transformer with set prediction), NanoDet (anchor-free lightweight detector), RetinaFace (face detection with landmarks), Helios (novel detection with TaskAlignedAssigner and CIoU loss).
- **Anomaly Detection**: PatchCore (memory bank approach), StudentTeacher (knowledge distillation for unsupervised anomaly detection).
- **Depth Estimation**: DPT (Dense Prediction Transformer), FastDepth (lightweight monocular depth).
- **Visual QA**: VQAModel with image-question fusion.
- **3D Reconstruction**: Aegis3D.
- **Specialized**: NightVision (thermal imaging), Phantom (proprietary detector).
- **Biometric**: Mnemosyne (face), Argus (iris), Echo (voice), Ariadne (fingerprint), Themis (multimodal fusion), AegisIdentity (unified API), IdentityBank.
- **Infrastructure**: FPN (Feature Pyramid Network), shared utilities.

The crate also provides built-in dataset loaders: MNIST, FashionMNIST, CIFAR-10, CIFAR-100, COCO, and WIDER FACE.

B. LLM (axonml-llm)

The LLM crate (109 tests) provides nine large language model architectures:

GPT-2 [11]: Decoder-only transformer with token and positional embeddings, configurable depth and width. Supports KV cache for efficient generation.

LLaMA [12]: Decoder-only with Rotary Positional Embeddings (RoPE), Grouped-Query Attention (GQA), RMSNorm, and SwiGLU MLPs. Configuration includes hidden_size, intermediate_size, num_hidden_layers, num_attention_heads, num_key_value_heads, max_position_embeddings, rope_theta, and rms_norm_eps.

Mistral [13]: LLaMA-style architecture with sliding-window attention for context-length efficiency.

Phi [14]: Microsoft's efficient small LLM with GQA and rotary embeddings.

BERT [15]: Encoder-only bidirectional transformer with token type embeddings, optional pooler for sequence classification.

SSM/Mamba [16]: Selective State Space Model with selective scan for long-context efficiency. Configuration: d_model, d_state, d_inner, d_conv, dt_rank.

Hydra: Hybrid SSM and sparse attention small language model with alternating SSM and attention layers, providing the long-context benefits of state space models with the flexibility of attention.

Trident: 1.58-bit ternary weight quantization (BitNet b1.58 [17] style), with weights restricted to $\{-1, 0, +1\}$ for transformer blocks while keeping embeddings and the LM head in full precision. Inference uses only addition and subtraction, achieving $16\times$ memory compression. Trained with the Straight-Through Estimator.

Chimera: Mixture of Experts combined with Differential Attention. Each layer contains 8 expert SwiGLU MLPs with top-2 routing (25% activation per token) and differential attention for noise cancellation. Trained with auxiliary load balancing loss to prevent expert collapse.

C. Audio (axonml-audio)

The audio crate provides differentiable audio transforms:

- **MelSpectrogram**: Computes mel-scaled spectrograms via STFT and mel filterbank. Configurable sample rate, FFT size, hop length, and mel bin count.
- **MFCC**: Mel-frequency cepstral coefficients via DCT-II of the log mel spectrogram.
- **Resample**: Linear interpolation resampling between sample rates.
- **TimeStretch** and **PitchShift**: Tempo and pitch modification for data augmentation.
- **AddNoise**: Gaussian noise injection at a specified SNR.
- **NormalizeAudio** and **TrimSilence**: Pre-processing utilities.

Built-in synthetic datasets enable testing without external data: SyntheticCommandDataset (speech commands), SyntheticMusicDataset (genre classification), SyntheticSpeakerDataset (speaker identification), and AudioSeq2SeqDataset (sequence-to-sequence audio tasks).

D. Text (axonml-text)

Text processing utilities include tokenizers, vocabulary management, BPE training, and embedding lookups. The crate is used by the LLM crate for input preprocessing.

E. Quantization (axonml-quant)

The quantization crate supports multiple quantization formats inspired by GGML/llama.cpp:

- **Q8_0**: 8-bit per-block quantization with FP16 scale, $\sim$3.8$\times$ effective memory reduction (34 bytes per 32-element block vs. 128 bytes FP32).
- **Q4_0**: 4-bit per-block with FP16 scale, $\sim$7.1$\times$ effective reduction (18 bytes per 32-element block).

- **Q4_1**: 4-bit with per-block scale and minimum (additional FP16 minimum).
- **Q5_0**: 5-bit per-block with FP16 scale, ∼6.0× effective reduction.
- **Q5_1**: 5-bit with per-block scale and minimum.
- **F16**: Half-precision floating point (2× reduction).

The default block size is 32 elements; effective compression ratios include the per-block scale and minimum overhead. The crate provides `quantize_tensor`, `dequantize_tensor`, `quantize_model`, calibration data structures for INT8 calibration, and `QuantizedLinear` layers for inference. The Trident LLM (Section VIII-B) implements 1.58-bit ternary quantization as a specialized layer beyond what the quant crate exposes.

F. ONNX (`axonml-onnx`)

The ONNX crate provides bidirectional interoperability with the Open Neural Network Exchange format, supporting opset 17 and IR version 8. The operator factory implements 39 operators: Add, Sub, Mul, Div, MatMul, Gemm, Pow, Sqrt, Exp, Log, Relu, Sigmoid, Tanh, LeakyRelu, Gelu, Softmax, Clip, ReduceSum, ReduceMean, ReduceMax, Reshape, Transpose, Flatten, Squeeze, Unsqueeze, Gather, Concat, Shape, Cast, Identity, Constant, Conv, MaxPool, AveragePool, BatchNormalization, Dropout, Equal, Greater, and Less.

The crate exposes `import_onnx`, `import_onnx_bytes`, `export_onnx`, and an `OnnxModel` representation with associated error types.

IX. Tooling and Deployment

A. Command-Line Interface

The `axonml-cli` crate provides 33 subcommands for typical ML workflows. Project lifecycle: `new`, `init`, `scaffold`, `rename`, `zip`. Training and inference: `train`, `resume`, `eval`, `predict`, `bench`, `analyze`. Model conversion and quantization: `convert`, `export`, `quant`, `load`, `inspect`. Data and datasets: `data`, `dataset`, `kaggle`. Reporting: `report`, `wandb`. GPU and profiling: `gpu`. Server lifecycle: `serve`, `start`, `stop`, `status`, `logs`. Authentication: `login`, `logout`, `sync`, `upload`, `hub`. Interactive interfaces: `tui`. The CLI integrates with Kaggle for dataset downloads and with Weights & Biases for experiment tracking.

B. Server

The `axonml-server` crate exposes a REST API for model inference, training job management, dataset operations, and system monitoring. API modules include authentication (JWT), inference endpoints, model management, training control, data pipelines, datasets, model hub integration, notebook support, system monitoring, training metrics, and Kaggle integration.

C. Terminal UI

The `axonml-tui` crate provides a terminal-based training interface for environments without a browser, useful for headless server training and remote monitoring over SSH.

TABLE IV
Transcendence ensemble configurations.

Model	Enc/Dec layers	d_{model}	Heads	Params
transcendence-a	3/3	192	6	∼4.85M
transcendence-b	2/2	192	6	∼2.85M
transcendence-c	3/3	256	8	∼5.50M

X. Applications

AxonML has been used to build complete, end-to-end machine learning projects across diverse domains. We highlight two machine translation projects below, both built entirely on AxonML with full training pipelines, beam-search decoding, evaluation metrics, and submission outputs. A multimodal biometric system (Aegis) is presented in a companion paper [2].

A. Transcendence: Akkadian-to-English Translation Ensemble

Transcendence is an encoder-decoder Transformer ensemble for the Kaggle Deep Past Initiative Machine Translation competition, translating Akkadian cuneiform transliteration into English. The competition metric is the geometric mean of BLEU and chrF++.

Architecture. Transcendence uses three independently-trained models for ensemble inference, configured via separate TOML files (Table IV). Each model is a standard pre-norm Transformer encoder-decoder with separate source and target embeddings, sinusoidal positional encoding, and a final output projection.

Tokenization. Transcendence uses separate BPE tokenizers for source and target languages, motivated by their fundamentally different character sets. The source (Akkadian) tokenizer uses syllable-aware splitting on hyphens (Akkadian transliteration is hyphen-delimited, e.g. `ma-nu-ba-lúm-a-šur`) followed by BPE with a 1,000-token vocabulary. The target (English) tokenizer uses standard whitespace pre-tokenization with a 4,000-token BPE vocabulary. Both tokenizers support BPE dropout [1] ($p = 0.05$) for subword regularization during training.

Training data. The training corpus combines three sources for a total of 10,419 parallel pairs: 1,562 pairs from the original ORACC training data, 4,439 pairs from the AICC parallel corpus, and 5,981 pairs of augmented data. Training uses CrossEntropyLoss with label smoothing 0.1, the Adam optimizer with cosine annealing and 400-step warmup, peak learning rate 10^{-3}, batch size 64, gradient clipping at norm 1.0, and early stopping with patience 40.

Inference. Each model decodes via beam search with beam width 5, max length 128, and length-normalized scoring. Ensemble inference averages the per-step logits from all three models before beam expansion, exploiting model diversity for improved generalization on the small training corpus. The complete project spans 6,006 lines of code (3,464 lines of Rust + 15 Python data-processing files) across 21 files.

B. Nabu: Character-Level Akkadian-English Translation

Nabu is a single-model character-level Transformer for the same Deep Past Initiative competition, exploring an alternative tokenization strategy. Where Transcendence uses linguistically-motivated BPE with separate vocabularies, Nabu uses a single shared character-level vocabulary with weight tying between input and output embeddings, eliminating the BPE training step entirely.

Architecture. Nabu is a 4-layer encoder, 4-layer decoder Transformer with $d_{\text{model}} = 384$, 8 attention heads, FFN dimension 1,536, dropout 0.25, and a maximum sequence length of 256 tokens. Pre-norm transformer blocks are used for improved gradient flow. The model file size is 66.7 MB (FP32 checkpoint).

Tokenization. A `CharTokenizer` assigns each unique character in the corpus its own token ID, with four reserved special tokens (`<pad>`, `<bos>`, `<eos>`, `<unk>`). The shared vocabulary spans both Akkadian and English characters, and weight tying initializes the output projection from the embedding weights. This approach trades off the linguistic structure encoded by syllable-aware BPE for simplicity, robustness to unseen tokens, and a smaller hyperparameter search space.

Training data. Nabu trains on the union of `train_updated.csv` and `aicc_parallel.csv` with deduplication via `HashSet`. Training uses CrossEntropyLoss with label smoothing 0.1, the Adam optimizer with cosine annealing, 2,000 warmup steps, peak learning rate 4×10^{-4}, batch size 16, character-level dropout ($p = 0.05$) and character-swap augmentation ($p = 0.03$), and early stopping with patience 20.

Inference. Beam search with beam width 5, max decode length 256, and length penalty $\alpha = 0.8$ (score normalized by len^{α}). The complete project spans 1,987 lines of Rust code across 7 files.

C. Comparison and Discussion

Transcendence and Nabu represent two contrasting approaches to the same translation task built on identical AxonML primitives. The shared infrastructure (Conv-free Transformer encoder/decoder, multi-head attention, sinusoidal positional encoding, fused CUDA kernels for GEMM, embedding lookup, cross-entropy loss, and Adam updates) is reused without modification across both projects, demonstrating that the framework's `Module` trait composition supports rapid architectural experimentation. The two projects differ entirely in tokenization strategy, vocabulary structure, and ensemble approach, yet share the same core neural network primitives, training loop scaffolding, and inference machinery.

D. Other Applications

AxonML has additionally been used to build a multimodal biometric identity suite (Aegis [2]) and a Sound Event Detection model targeting the Kaggle BirdCLEF acoustic species identification series (SED-Net, 2,885,482 parameters, 234-class multi-label classification over 5-second mel spectrogram windows from continuous soundscape recordings). These projects exercise the vision and audio domain crates respectively. Detailed evaluation of these systems is presented elsewhere or remains as future work.

XI. Related Work

Python frameworks. PyTorch [3] provides eager execution with dynamic graphs and is the dominant research framework. TensorFlow [4] offers static graphs (1.x) and eager mode (2.x). JAX [5] uses tracing-based functional transformations. All require Python and have substantial runtime footprints unsuitable for many deployment contexts.

Rust ML frameworks. `tch-rs` [18] provides Rust bindings to libtorch but requires the C++ PyTorch runtime as a dependency. `burn` [19] is a native Rust framework with multiple backends (NDArray, WGPU, Tch, LibTorch, Candle), targeting both training and inference. `candle` [20] from Hugging Face focuses on model inference with CUDA and Metal backends, optimized for LLM deployment. `dfdx` [21] uses Rust's type system for compile-time tensor shape checking. AxonML differs from these projects in providing a complete training stack with custom CUDA PTX kernels (rather than only relying on cuBLAS or cuDNN abstractions), nine LLM architectures, distributed training primitives, and integrated production tooling (CLI, server, training monitor).

Non-Python frameworks in other languages. Julia's Flux [22] provides differentiable programming with composable layers. Swift for TensorFlow [23] integrated automatic differentiation into the Swift compiler before being discontinued. MLX [24] from Apple targets Apple Silicon with automatic differentiation and GPU acceleration via Metal. AxonML targets NVIDIA GPU deployment with CUDA, in environments where Python is unavailable, undesirable, or inappropriate.

XII. Limitations and Ongoing Work

This paper describes the framework's design and capabilities; rigorous comparative throughput and memory benchmarks against PyTorch baselines are deferred to future work. We have observed that AxonML successfully trains the model architectures described in Section X on consumer NVIDIA hardware (RTX 5070 class), but we have not yet conducted controlled head-to-head benchmarks measuring iteration time, peak memory, or end-to-end training time against PyTorch implementations of the same architectures. Reporting such numbers without controlled experimental conditions would be misleading.

The Aegis biometric models referenced in Section X have only been preliminarily evaluated on small grayscale datasets; the BirdCLEF SED-Net has not yet been evaluated against a competition test set. We mention these systems as evidence that the framework supports diverse architectures end-to-end, not as evidence of state-of-the-art accuracy on any particular benchmark.

Future work includes: (1) controlled performance benchmarks against PyTorch baselines on representative workloads (ResNet-50 on ImageNet, BERT-Base on GLUE, LLaMA

inference throughput), (2) expanding ONNX operator coverage to opset 20+, (3) implementing Flash Attention for long-context training, (4) adding support for AMD ROCm in addition to CUDA, (5) extending distributed training primitives to support tensor parallelism, and (6) a WebGPU backend for browser deployment.

XIII. Availability

AxonML is available at https://github.com/AutomataNexus/AxonML under dual MIT/Apache-2.0 license. The crates are published to crates.io. The framework requires Rust 1.85+ (edition 2024). CUDA support requires an NVIDIA GPU with the `cuda` feature flag enabled at build time:

```
cargo build --release --features cuda
```

XIV. Conclusion

AxonML demonstrates that a complete, practical deep learning framework can be implemented in Rust without compromising on capability. The 22-crate workspace provides everything from low-level tensor operations through high-level model architectures, with native CUDA GPU acceleration via 15 PTX kernel modules, automatic differentiation with gradient checkpointing and AMP, distributed training (DDP, FSDP, pipeline parallelism), ONNX interoperability with 39 operators, six quantization formats, and production tooling including a training monitor with live browser dashboards, a REST API server, and a 33-subcommand CLI.

The framework's nine LLM architectures span the modern landscape from GPT-2 through ternary-weight Trident and mixture-of-experts Chimera. The vision library covers classification, detection, anomaly detection, depth estimation, and biometrics. Two end-to-end machine translation projects (Transcendence and Nabu) for the Kaggle Deep Past Initiative competition validate the framework's correctness and composability across contrasting tokenization and ensemble strategies built on identical underlying primitives.

References

[1] I. Provilkov, D. Emelianenko, and E. Voita, "BPE-Dropout: Simple and effective subword regularization," in *Proc. ACL*, 2020, pp. 1882–1892.

[2] A. Jewell Sr., "Aegis: A lightweight multimodal biometric suite via temporal crystallization, predictive residuals, polar phase encoding, and uncertainty-aware belief propagation fusion," Manuscript in preparation, 2025. Available: https://github.com/AutomataNexus/AxonML

[3] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. NeurIPS*, 2019.

[4] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI*, 2016, pp. 265–283.

[5] J. Bradbury *et al.*, "JAX: Composable transformations of Python+NumPy programs," 2018.

[6] T. Ye, L. Dong, Y. Xia, Y. Sun, Y. Zhu, G. Huang, and F. Wei, "Differential Transformer," *arXiv preprint arXiv:2410.05258*, 2024.

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

[8] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. ICLR*, 2019.

[9] Y. You *et al.*, "Large batch optimization for deep learning: Training BERT in 76 minutes," in *Proc. ICLR*, 2020.

[10] L. N. Smith and N. Topin, "Super-convergence: Very fast training of neural networks using large learning rates," *arXiv preprint arXiv:1708.07120*, 2017.

[11] A. Radford *et al.*, "Language models are unsupervised multitask learners," *OpenAI Blog*, 2019.

[12] H. Touvron *et al.*, "LLaMA: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[13] A. Q. Jiang *et al.*, "Mistral 7B," *arXiv preprint arXiv:2310.06825*, 2023.

[14] Y. Li *et al.*, "Textbooks are all you need II: Phi-1.5 technical report," *arXiv preprint arXiv:2309.05463*, 2023.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL*, 2019, pp. 4171–4186.

[16] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023.

[17] 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.

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

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

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

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

[22] M. Innes, "Flux: Elegant machine learning with Julia," *Journal of Open Source Software*, vol. 3, no. 25, p. 602, 2018.

[23] R. Wei *et al.*, "Swift for TensorFlow: A portable, flexible platform for deep learning," 2020.

[24] A. Hannun *et al.*, "MLX: An efficient machine learning framework for Apple Silicon," https://github.com/ml-explore/mlx, 2023.