Native Training in hanzo-ml/hanzo-engine: One Engine, One Quant Format, One Backend

Zach Kelling

Hanzo AI

June 2026

Abstract

We describe the architecture for training language models natively inside the hanzo Rust inference stack (`hanzo-ml` / `hanzo-engine`), eliminating the conversion boundary between a Python training stack and a separate serving engine. The existing engine already ships a reverse-mode autograd (`hanzo-ml`, a fork of HuggingFace `candle`; 809 lines, full matmul/conv/gather rules), bias-corrected AdamW with decoupled weight decay, safetensors checkpoint I/O, and a unified quantization kernel that serves `Q8_0` decode at 24.5 tokens/s, equal to 0.98× llama.cpp (25.05 tokens/s) on a gfx1151 / Radeon 8060S (Strix Halo) device. Two trainers prove the substrate end-to-end: an algorithm-complete, tested GRPO reinforcement-learning loop ported from `zen/gym`, and an in-engine AnyMoE (the MoE-distillation trainer inherited from `mistral.rs`) gate trainer that performs forward, cross-entropy, backward, optimizer step, and checkpoint save over frozen quantized expert weights. We show that the QLoRA primitive—a detached `dequantize()` of the quantized base plus a trainable `bf16` adapter pair—is already present in the code, so a model can be quantized once, served by the unified kernel, and trained by attaching adapters over the exact bytes that serve it, with zero dequant-to-train and zero re-quant-to-serve. We formalize the parameter and memory accounting of rank-r QLoRA over a frozen quantized base, and prove that linear LoRA adapters over a shared base compose additively (the "LoRA-soup" property). We give an honest accounting of what is real, what needs building, and what is deliberately out of scope.

1 Introduction

Production deployment of a fine-tuned language model conventionally crosses an engine boundary. A model is trained in a thick Python stack and then *re-exported and re-quantized* before a separate inference engine can serve it. The result is two engines, two quantization formats, two device backends, and a conversion step at every boundary.

The hanzo stack is positioned to remove that boundary. The Rust inference engine already owns a reverse-mode autograd (`ml/hanzo-ml/src/backprop.rs`, 809 lines; `hanzo-ml` is a fork of HuggingFace `candle`), real optimizers (`ml/hanzo-nn/src/optim.rs`), and a unified quantization kernel core that serves `Q8_0` decode at 24.5 tokens/s, equal to 0.98× of llama.cpp on a gfx1151 / Radeon 8060S device [1]. If training is built *on the same `Tensor/Var/quant` substrate* that serves inference, then a model is trained and served by one engine, in one quant format, on one device backend, with **zero conversion**. The strategic payoff concentrates in QLoRA: `bf16` adapters can be trained directly over the *exact* quantized weights that the serving kernel reads.

This is partly real today and partly a build plan. The contribution of this paper is to state the split honestly, to identify the single load-bearing finding—that the autograd primitive QLoRA needs is already present in the quantization layer—and to formalize the parameter, memory, and composition properties that make the native-training architecture sound.

Provenance of claims. Every assertion marked Real cites a file in the hanzo tree. Every performance number is measured on real hardware and cited to the ROCm performance record [1], or else explicitly labelled an estimate. Techniques inherited from Python tooling are marked Ported; the genuinely novel pieces are marked Novel. Build status is marked Real, Needs-Building, or Aspirational.

Contributions.

- An architecture that unifies training and serving on a single Rust substrate: `loss.backward` → `GradStore` → `AdamW.step` → `Var.set` → safetensors, reusing the exact quantization kernel that serves inference.

- Identification of the QLoRA primitive already present in the code: `QTensor::dequantize()` emits a detached constant, so a frozen quantized base carries no gradient and a separate adapter pair carries the entire learnable signal.
- A formal parameter and training-memory accounting for rank-r QLoRA [2, 3] over a frozen quantized base (Proposition 1), with proof.
- A composition lemma showing that linear LoRA adapters over a shared base compose additively, justifying "LoRA-soup" [6] merging (Lemma 1), with proof.
- An ordered migration plan from the Python `gym` stack to the native Rust trainer, with risk and effort estimates, and an explicit list of stubs to replace and features deliberately left out of scope.

2 The Foundation That Already Works

The following components are not stubs; each is cited and several are covered by passing tests. Collectively they form the training *substrate*.

Autograd (Real). `ml/hanzo-ml/src/backprop.rs`. `Tensor::backward()` returns a `GradStore` via topological `sorted_nodes()`, with full reverse rules including matmul (lhs gradient = grad $\cdot$ rhs$^\top$, rhs gradient = lhs$^\top$ $\cdot$ grad, lines 457–467), convolution, gather/scatter, `index_add`, concatenation, broadcast, and `CustomOp1/2/3` backward dispatch (lines 662–687). `GradStore` is a `HashMap<TensorId, Tensor>` (line 733).

Var (Real). `ml/hanzo-ml/src/variable.rs`. `Var(Tensor)` is the leaf wrapper; `is_variable()` marks gradient leaves; `set()` performs an autograd-safe in-place update, which is what an optimizer step writes through.

Optimizers (Real). `ml/hanzo-nn/src/optim.rs`. An `Optimizer` trait with `step(&GradStore)` and `backward_step(&loss)` (lines 5–29); plain `SGD` and a full bias-corrected `AdamW` with decoupled weight decay (lines 152–182). Crucially, `new()` *filters to float dtypes* (`var.dtype().is_float()`, lines 44–47 and 122–124), so quantized or integer leaves are never updated. That filter is exactly the invariant a QLoRA trainer requires.

VarMap and checkpoint I/O (Real). `ml/hanzo-nn/src/var_map.rs`. `save()`/`load()` via safetensors (lines 31–45), `all_vars()`, and lazy `get(shape, path, init, dtype, dev)`. This is real persistence for trainable parameters today.

Losses (Real). `ml/hanzo-nn/src/loss.rs`. `nll`, `cross_entropy` (log_softmax+nll), `mse`, `binary_cross_entrop` and `huber`, all differentiable.

NN layers (Real). `hanzo-nn` provides linear, embedding, the norm family, convolution, RNN, MoE, rotary embedding, activations, and `ops.rs` (log_softmax/softmax/topk). All are forward-only structs that ride autograd; no hand-written backward is needed.

Verdict. The training substrate—`loss.backward` → `GradStore` → `AdamW.step` → `Var.set` → safetensors save—is REAL and end-to-end. What is missing is the trainer scaffolding on top of it (Sections 6 and 8).

3 Two Real Trainers

3.1 GRPO reinforcement learning (Ported, tested)

`ml/hanzo-training/src/grpo/` contains `trainer`, `math`, `policy`, `sampler`, `config`, `reward`, and `metrics`. The module is ported from the `zen/gym` GRPO [5] trainer. `GrpoTrainer::step` (`trainer.rs:129`) runs the full five-phase update: rollout (`Policy::sample_group`); reward (weighted multi-reward sum, `score()` at `trainer.rs:119`); group-relative advantages (`math::batch_advantages`, unbiased standard deviation with ddof = 1, `math.rs:45--52`); a differentiable, sign-correct, length-normalized policy-gradient loss (`math::policy_gradient_loss`, `math.rs:87--120`); an optional low-variance `k3` KL penalty to a frozen reference ($k3 = \exp(\Delta) - \Delta - 1$, `math.rs:134--166`); and `AdamW.backward_step(&loss)` (`trainer.rs:228`). It is tested: a smoke test learns a toy reward to ceiling (asserts mean reward rises past $0.6 \cdot$ `max_len`), a KL-branch test stays finite, and math unit tests verify advantage zero-mean, unit-scale scaling, and policy-gradient sign.

The gap, stated honestly. The `Policy` trait (`policy.rs:33`) requires exactly three methods: `trainable_vars()`, `sample_group()`, and `sequence_logprob()`. The only implementor is `ToyPolicy`, a `[max_len, vocab]` logits bandit in tests and `examples/grpo_toy.rs`. No hanzo-engine transformer implements `Policy` yet; wiring a real model to those three methods is NEEDS-BUILDING (Section 8).

3.2 AnyMoE gate trainer (Real, end-to-end in-engine)

(`AnyMoE` is a `mistral.rs` feature inherited by the engine.) `engine/hanzo-engine/src/pipeline/amoe.rs` and `engine/hanzo-engine/src/amoe/mod.rs` contain the strongest existing "train on top of an inference model" precedent. It builds trainable MoE gating `Linear`s as `Var`s, then runs a real per-layer loop (`amoe.rs:348--516`): build a per-layer `AdamW` (`amoe.rs:351`), forward the frozen base model (`target.forward_inputs`, `amoe.rs:483`), collect each gate's logits, take `hanzo_nn::loss::cross_entropy` (`amoe.rs:507`), `loss.backward()`, `optimizer.step(&gradstore)` (`amoe.rs:511--512`), log CSV loss, and finally detach the gates and save `gate.safetensors`. It trains *new* gating parameters on top of frozen quantized expert weights—the exact pattern QLoRA needs: trainable `Var`s layered over a frozen base, inside the inference engine.

4 The Quant Stack and the QLoRA Primitive

4.1 What exists for inference (Real)

- **candle quant** (`ml/hanzo-ml/src/quantized/`): `QTensor`/`QMatMul`, GGUF k-quants (`Q2K`–`Q6K`, `Q8_0`, `MXFP4`), plus 11 low-bit IQ/ternary/FP4 types added and validated bit-exact against ggml [1].
- **engine quant** (`engine/hanzo-quant`): a `QuantMethod` trait (`lib.rs:990`) with `forward`, `dequantize_w` (`lib.rs:995`), `add_delta_w` (`lib.rs:1056`), and `apply_isq`; methods for gguf/gptq/hqq/afq/b in-situ quantization (ISQ), imatrix, and UQFF serialization.

- **the unified ROCm core** (NOVEL, Section 7): one templated `qmatvec_core<WTYPE,XT>` decode and `qmmq_core<WTYPE>` prefill, one decode function per format, table-driven dispatch. Adding a quant costs one `qdec`, one traits row, and one macro line; *zero new kernels.*

4.2 The load-bearing finding

`QTensor::dequantize()` (`quantized/mod.rs:775--778`) constructs its output with `BackpropOp::none()`:

```
pub fn dequantize(&self, device: &Device) -> Result<Tensor> {
    let storage = self.storage
        .dequantize(self.shape.elem_count())?;
    let none = crate::op::BackpropOp::none();
    crate::tensor::from_storage(
        storage, self.shape.clone(), none, false)
        .to_device(device)
}
```

`dequantize_f16` (`mod.rs:781`) and the `QMatMul`/`indexed_moe_forward` forwards likewise emit `BackpropOp::none()`. A dequantized weight is therefore a *detached constant*: no gradient flows into quantized storage. This is not a limitation; it is exactly the correct QLoRA primitive. The frozen quant base is a constant, and a separate `bf16`/`f16` LoRA `Var` pair (A, B), whose matmul backpropagates normally (rules at `backprop.rs:457--467`), carries the entire learnable signal. The AdamW float-dtype filter (Section 2) guarantees the quant base is never touched even if it were handed in.

Consequently the *math substrate* for QLoRA exists today. The LoRA layer and the assembly that injects it do not (Section 6).

4.3 What is not a trainer today

The existing LoRA (`engine/hanzo-engine/src/lora/loralinear.rs`) loads `a_adapters`/`b_adapters` as frozen `Linear`s from a `ShardedVarBuilder` (`loralinear.rs:17--18, 34--48`) and supports `merge_weights()` via `QuantMethod::add_delta_w` (`loralinear.rs:145--156`). It is inference-and-merge only: the adapters are constants, not `Var`s, and there is no backward path into them. Likewise `hanzo-quant/src/lora/static_lora.rs` is static. There is no trainable-LoRA path anywhere in the tree yet.

5 Formal Properties of QLoRA over a Frozen Quantized Base

We now formalize the parameter and memory accounting that motivates QLoRA, and the composition property that justifies adapter merging. Throughout, a linear layer maps $x \in \mathbb{R}^{d_{\text{in}}}$ to $y \in \mathbb{R}^{d_{\text{out}}}$ via a weight matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$.

Definition 1 (LoRA adapter)**.** *A rank-r LoRA adapter for a linear layer with frozen base weight $W_0 \in \mathbb{R}^{d_{out} \times d_{in}}$ is a pair (A, B) with $A \in \mathbb{R}^{r \times d_{in}}$ and $B \in \mathbb{R}^{d_{out} \times r}$, together with a fixed scalar $s = \alpha / r$. The adapted layer computes*

$$y = W_0 x + s\, B(Ax) = \big(W_0 + s\, BA\big)x.$$

Only A and B are trainable; W_0 is held constant.

Definition 2 (Quantized base)**.** *A* quantized base *is a weight tensor W_0 stored in a low-bit format Q (e.g. `Q8_0`, `Q4_K`) and exposed to autograd only through a detaching dequantization* $\mathrm{deq}_Q(W_0)$*, i.e.*

the operation that produces W_0 in compute precision carries `BackpropOp::none()` *and contributes no entry to the* `GradStore`*.*

Proposition 1 (QLoRA trainable-parameter and memory accounting)**.** *Consider a model whose linear layers carry frozen quantized base weights as in Definition 2, with rank-r LoRA adapters as in Definition 1 attached to a set $\mathcal{L}$ of layers, where layer $\ell \in \mathcal{L}$ has dimensions $(d^\ell_{out}, d^\ell_{in})$. Then:*

1. *The number of trainable parameters contributed by layer ℓ is exactly $r\,(d^\ell_{out} + d^\ell_{in})$; in the square case $d^\ell_{in} = d^\ell_{out} = d$ this is $2rd$ per matrix.*
2. *No gradient is ever written to any quantized base weight; the set of optimizer-updated leaves is precisely $\bigcup_{\ell\in\mathcal{L}}\{A_\ell, B_\ell\}$.*
3. *The optimizer state and weight-gradient memory scale as $\Theta\big(\sum_{\ell\in\mathcal{L}} r(d^\ell_{out}+d^\ell_{in})\big)$, independent of the (much larger) base parameter count $\sum_\ell d^\ell_{out} d^\ell_{in}$. Total training memory is therefore the quantized base (read-only) plus adapter weights, adapter gradients, adapter optimizer moments, and the forward activations retained for backward—*not *a full-precision copy of the base weights, their gradients, or their optimizer state.*

Proof. (1) By Definition 1, layer ℓ adds the trainable matrices $A_\ell \in \mathbb{R}^{r\times d^\ell_{\text{in}}}$ and $B_\ell \in \mathbb{R}^{d^\ell_{\text{out}}\times r}$ and no other trainable tensor (s is a fixed scalar, not a parameter). Counting entries gives $r\,d^\ell_{\text{in}} + d^\ell_{\text{out}}\,r = r(d^\ell_{\text{out}} + d^\ell_{\text{in}})$, which is $2rd$ when $d^\ell_{\text{in}} = d^\ell_{\text{out}} = d$.

(2) The adapted output is $y = \text{deq}_Q(W_0)x + s\,B(Ax)$. The first term is detached by Definition 2: $\text{deq}_Q(W_0)$ carries `BackpropOp::none()` and so registers no producer in the backward graph; hence $\partial\mathcal{L}/\partial W_0$ is never accumulated into the `GradStore`, regardless of downstream loss $\mathcal{L}$. The second term is a composition of standard matmuls, whose reverse rules (`backprop.rs:457--467`) populate gradients for A_ℓ and B_ℓ. Thus the only leaves receiving gradients are the $\{A_\ell, B_\ell\}$. Independently, the optimizer's construction filter retains only float-dtype leaves (Section 2); since the quantized base is stored in a non-float quant dtype, it is excluded from the update set even if mistakenly registered. The two mechanisms are independent and each is sufficient, so the optimizer-updated set is exactly $\bigcup_\ell\{A_\ell, B_\ell\}$.

(3) A first-order adaptive optimizer such as AdamW stores, per trainable scalar, one gradient and a constant number of moment estimates (first and second moment), i.e. $\Theta(1)$ memory per trainable scalar. By (2) the trainable scalars number $\sum_\ell r(d^\ell_{\text{out}}+d^\ell_{\text{in}})$, giving optimizer-plus-gradient memory of that same order. The base weights are read-only and contribute no gradient or moment; they reside in their compressed quantized form (Definition 2). Hence the total is the read-only quantized base plus adapter weights, adapter gradients, adapter moments, and retained activations, with the adapter-related terms independent of $\sum_\ell d^\ell_{\text{out}} d^\ell_{\text{in}}$. □

Lemma 1 (Additive composition of linear LoRA adapters; the LoRA-soup property)**.** *Fix a frozen base weight W_0 for a linear layer and a common scale s. Let $\{(A_i, B_i)\}_{i=1}^n$ be n rank-r_i LoRA adapters trained over the* same *W_0, with merged increments $\Delta_i = s\,B_iA_i$. For any convex combination weights $\lambda_1, \ldots, \lambda_n \geq 0$ with $\sum_i \lambda_i = 1$, define the soup weight*

$$W_{soup} = W_0 + \sum_{i=1}^{n} \lambda_i \Delta_i.$$

Then on every input x,

$$W_{soup}\,x = \sum_{i=1}^{n} \lambda_i \big(W_0 + \Delta_i\big)x,$$

i.e. applying the merged adapter equals the convex combination of the individually adapted layer outputs. Moreover $W_{soup} - W_0 = \sum_i \lambda_i \Delta_i$ admits an exact rank-$\left(\sum_i r_i\right)$ factorization, so the soup is itself representable as a single LoRA adapter.

Proof. Each adapted layer is affine in x with the *shared* linear part anchored at W_0: by Definition 1, $(W_0 + \Delta_i)x = W_0 x + \Delta_i x$. Take the convex combination and use $\sum_i \lambda_i = 1$:

$$\sum_{i=1}^{n} \lambda_i (W_0 + \Delta_i)x = \Big(\sum_i \lambda_i\Big) W_0 x + \sum_i \lambda_i \Delta_i x = W_0 x + \Big(\sum_i \lambda_i \Delta_i\Big) x = W_{\text{soup}}\, x.$$

The first equality is linearity of matrix-vector multiplication in the weight; the identity holds for all x, establishing the stated equality of maps. For the factorization, define the stacked matrices

$$\widetilde{B} = \left[\, s\lambda_1 B_1 \;\middle|\; s\lambda_2 B_2 \;\middle|\; \cdots \;\middle|\; s\lambda_n B_n \,\right] \in \mathbb{R}^{d_{\text{out}} \times \sum_i r_i}, \qquad \widetilde{A} = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} \in \mathbb{R}^{\sum_i r_i \times d_{\text{in}}}.$$

Then $\widetilde{B}\widetilde{A} = \sum_i s\lambda_i B_i A_i = \sum_i \lambda_i \Delta_i = W_{\text{soup}} - W_0$, exhibiting the increment as a single LoRA factorization of rank at most $\sum_i r_i$ (with scale folded into $\widetilde{B}$). □

Remark on necessity of the shared base. The argument uses $\sum_i \lambda_i = 1$ precisely to collapse $\sum_i \lambda_i W_0$ back to W_0. If the adapters were trained over *different* bases $W_0^{(i)}$, the cross terms $\sum_i \lambda_i (W_0^{(i)} - W_0)$ would not vanish and additive composition would fail. This is the formal reason hanzo's soup and delta-soup routines (`model_delta/soup.rs`, Section 6) operate over a common base checkpoint. □

These two results capture why owning both the trainer and the quant kernel is structurally advantageous: Proposition 1 shows the trainable footprint is the adapter, not the base, even though the base is served at full quantized speed; Lemma 1 shows multiple such adapters merge exactly over the shared frozen base, enabling post-hoc model-soup composition without retraining.

6 The Native Trainer Catalogue

Table 1 maps each trainer onto the autograd engine and records its build status.

Full fine-tune (SFT). The substrate is Real; only the scaffolding is missing. The path: build the model with a `VarBuilder::from_varmap` so weights are `Var`s, run a teacher-forced forward, take per-token `cross_entropy`, `AdamW.backward_step`, and checkpoint with `VarMap.save`. The blocker is that the roughly 80 engine model definitions build via `VarBuilder` into *frozen* `Tensor`s; SFT needs a `Var`-backed construction path. The generic `Trainer` advertised for this is a stub (Section 9).

LoRA. A new `TrainableLoraLinear` holding a frozen base alongside trainable `Var`s A, B and scale s. Forward: `base.forward(x) + lora_b(lora_a(x)) * scale`, with A, B created through a `VarMap`. Backward flows only into A, B because the base is detached. It reuses the existing adapter math from `loralinear.rs`, swapping the frozen `Linear` for `Var`s.

Table 1: Native trainer catalogue and autograd mapping.

Trainer	Status	Mechanism
Full SFT	Needs-building (substrate real)	Weights as `Var`s; teacher-forced LM `cross_entropy`; `AdamW`; save.
LoRA	Needs-building	Frozen base + `Var`s A, B; $y = W_0x + s\,B(Ax)$.
QLoRA	Needs-building (primitive real)	As LoRA; base is `QTensor` via `qmatvec_core`; already detached.
DoRA	Needs-building	LoRA + trainable column-magnitude `Var`; norm op.
BitDelta	Real (compress) trainer pending	Sign-pack $(W_f - W_0)$, scale $\alpha = \text{mean}\lvert\Delta\rvert$; reconstruct.
Soup / delta-soup	Real	Elementwise mean of checkpoints; $W_0 + \text{mean}_i(W_{f,i} - W_0)$.

QLoRA—the synergy that justifies the effort. Identical to LoRA, but the base is a `QTensor` served by the unified `qmatvec_core` / `QMatMul` path. Because that path already emits `BackpropOp::none()` (Section 4), no extra detaching is needed. The model is quantized once, served at the performance of Section 7, and trained by attaching `bf16` adapter `Var`s over the very same bytes: zero dequant-to-train, zero re-quant-to-serve. This is the one place where owning both the trainer and the quant kernel yields something the Python stack structurally cannot: bitsandbytes/peft must dequantize a block to `f16`, run the `f16` matmul, and serve through a *different* code path; here it is literally the same kernel. Proposition 1 quantifies the resulting memory advantage.

DoRA. Weight-decomposed LoRA [4]: $W' = m \cdot (W_0 + \Delta W)/\|W_0 + \Delta W\|_{\text{col}}$, where m is a trainable per-column magnitude vector and $\Delta W = BA$. On the engine: add one magnitude `Var` of shape $[d_{\text{out}}]$, compute the column norm with existing ops, and scale. All differentiable; no new backward. The only cost is recomputing the directional norm each step, a known DoRA expense, not a hanzo-specific one.

BitDelta and soups (Real). `model_delta/bitdelta.rs` implements the BitDelta [7] encode/decode_apply: encode is sign-pack$(W_f - W_0)$ plus a per-tensor scale $\alpha = \text{mean}\lvert\Delta\rvert$; reconstruct is $W_0 + \alpha \cdot \text{sign}(\Delta)$. `model_delta/soup.rs` implements soup (elementwise mean of checkpoints) and delta_soup $(W_0 + \text{mean}_i(W_{f,i} - W_0))$, both tested. They operate on saved checkpoints, so they compose with any trainer above: train, save `Var` weights, then `bitdelta::encode` for 1-bit delta storage or `delta_soup` to average several fine-tunes. Lemma 1 is the formal justification for the LoRA case of delta-soup. For the patent record: neither BitDelta nor any model-soup/TIES/DARE routine exists anywhere in `zen/gym`; these are hanzo-native with no Axolotl analog. The DeltaSoup *federation* variant (trim-mean of LoRA deltas) lives only in the Python `zen/gym` and is not yet ported to Rust.

7 Quant-Aware Training and Inherited Performance

The reason to train on hanzo's quant core is that the core is fast and bit-exact, and an adapter trained over it serves with no conversion. Table 2 summarizes the measured state on gfx1151 / Radeon 8060S (Strix Halo), all numbers from the ROCm performance record [1]. The real-hardware roofline is 217 GB/s read bandwidth and 52 TOPS-i8.

Table 2: Measured performance on gfx1151 / 8060S (Strix Halo). Roofline: 217 GB/s read, 52 TOPS-i8. "× llama" is the ratio to llama.cpp on the same hardware. All quant numbers verified `nbad`= 0 against the CPU `to_float` oracle.

Path	**Before**	**After**	**× llama**
Decode matvec (GB/s)	103.5	245.8	0.97
(% of BW wall)	45%	97%	—
Decode end-to-end (t/s)	8.5	24.5	0.98
MoE decode (t/s)	0.93	13.6	—
Prefill end-to-end (t/s)	39	1015	0.86
WMMA GEMM isolated	—	1414	> 1

Decode (Real). The lane-strided unified core rewrote `qmatvec_core` from serial to lane-strided: the isolated matvec went from 103.5 GB/s (45% of the bandwidth wall) to 245.8 GB/s (97%), a 2.37× gain; end-to-end eager decode went from 8.5 to 24.5 tokens/s, equal to 0.98× llama.cpp (25.05 tokens/s). This is effective parity and the measured honest ceiling: matvec sits at the bandwidth wall, and batch-1 attention is not a winnable lever.

The unified abstraction (Novel). One templated `qmatvec_core<WTYPE,XT>` decode and `qmmq_core<WTYPE>` prefill, with `quant_format!` / `for_each_quant!` table-driven dispatch (`quantized/quant_format.rs`, `proof.rs`). Adding a quant costs one `qdec`, one traits row, and one macro line; zero new kernels. It is proven across `Q8_0`/`Q4_0`/`Q4_K`/`Q6_K`/`IQ4_XS`/`TQ2_0`, all with `nbad`= 0 against the CPU `to_float` oracle. A QLoRA base in any of these formats trains and serves through the same core.

MoE. The same core powers `indexed_moe_forward` per routed expert; MoE decode went from 0.93 to 13.6 tokens/s (about 2×, via a resident-bank cache plus the lane-strided core), with `Q4_K`+`Q8_0` MoE at `nbad`= 0. This is the path an AnyMoE-style or MoE-QLoRA trainer would ride.

Prefill. The WMMA int8 GEMM kernel reaches about 1414 tokens/s in isolation (above llama); end-to-end prefill went from 39 to 1015 tokens/s, equal to 0.86× llama.cpp (1176 tokens/s). The remaining gap is engine attention, not the matvec; a multi-wave WMMA flash-attention kernel is the open lever (ASPIRATIONAL, a large rewrite).

Honest framing. Decode is parity, prefill is 0.86× and improving, and MoE decode is roughly 2×. The training value is not "we are fastest"; it is that the trained artifact is *already* in the served format on the served kernel.

8 Migration Plan: gym (Python) → native (Rust)

Reusable as-is (Real, no rewrite). Autograd; SGD/AdamW; VarMap save/load; the loss module; the roughly 80 model architecture definitions (as forward graphs); the full quant inference stack with ISQ and UQFF; the GRPO algorithm core; AnyMoE gate training; BitDelta; soups.

Build (Needs-Building), ordered by value/effort.

1. **Trainable LoRA/QLoRA adapter layer** (highest value, low-medium effort). A `TrainableLoraLinear` with (A, B) `Var`s over a frozen base; QLoRA variant over `QTensor`. Risk: LOW—the AD primitive is proven (Section 4), AnyMoE proves the train-over-frozen pattern (Section 3), and the adapter math already exists. Estimate: about one week to a tested LoRA + QLoRA on one architecture (the 8B dense transformer), reusing `qmatvec_core` for the base.

2. **Real transformer `Policy` for GRPO** (high value, medium effort). Implement the three trait methods (`policy.rs:33`) against a hanzo-engine model: `sample_group` is the existing generate loop under no-grad; `sequence_logprob` is a teacher-forced forward returning $\sum \log p$. Risk: MEDIUM—needs the `Var`-backed model path (shared with SFT) and careful no-grad/grad separation. Estimate: one to two weeks; pairs naturally with item 1.

3. **Generic SFT trainer + real tokenized dataset** (high value, medium effort). Replace the stub `Trainer`/`ModelWrapper`/`OptimizerWrapper` (Section 9) with a real epoch/step loop calling `hanzo_nn::AdamW` directly, a teacher-forced LM `cross_entropy`, and a real `tokenizers`-backed dataset. Risk: MEDIUM, mostly plumbing. Estimate: one to two weeks.

4. **Preference losses (DPO [8]/KTO/ORPO/SimPO)** (medium value, low effort once item 2 exists). Closed-form differentiable losses over (chosen, rejected) log-probs, using the same `sequence_logprob`. Risk: LOW. Estimate: a few days each.

5. **Training ergonomics** (medium value, low-medium effort). Real LR schedulers (cosine/linear are toy approximations in the stub, `optimizer.rs:55--77`), gradient clipping, gradient accumulation, mixed precision, optimizer-state checkpoint/resume (only `Var` weights save today, not moments), and explicit gradient checkpointing (only the global `GRAD_DO_NOT_DETACH` toggle exists). Risk: LOW-MEDIUM, individually small.

6. **DoRA / BitDelta-trainer / federated DeltaSoup control plane** (ASPIRATIONAL-roadmap). DoRA is a small extension of item 1. The federated trim-mean DeltaSoup from `zen/gym` is Python-only today; porting its control plane to Rust over the `bf16`-safetensors VarMap seam is a separate project.

9 Stubs to Replace

These are advertised in `hanzo-training` but do not function; a reader must not mistake them for the working pieces above.

- `ml/hanzo-training/src/optimizer.rs`: `OptimizerWrapper::step()`/`zero_grad()` are no-ops—the comment reads "Simplified optimizer step - in practice, this would update parameters" (`optimizer.rs:82`); it only advances a step counter and an LR number, touching no params or grads. (GRPO and AnyMoE ignore this and use `hanzo_nn::AdamW` directly, which is why they work.)

- `ml/hanzo-training/src/model.rs`: `ModelWrapper.forward()` returns a constant `Tensor::new(&[1.0])`, `backward()` is a no-op, `parameters()` returns `vec![]`, and `save()` writes only JSON metadata.

- `ml/hanzo-training/src/trainer.rs`: the generic SFT `Trainer` is wired to those two stubs, so it "trains" nothing; `train_qwen.rs`/`train_llama.rs`/`bin/train.rs` all call it.

- `ml/hanzo-training/src/dataset.rs`: "simplified tokenization," not a real tokenizer.

The migration (items 1–3) replaces all four with the proven `hanzo_nn::AdamW` + `loss.backward` + `VarMap` substrate.

10 Limitations and Future Work

We state the boundaries of the present architecture plainly.

The trainers above the substrate are not yet built. Sections 3 and 4 establish that the substrate (autograd, AdamW, VarMap, losses, detached quant base) is real and that two trainers (GRPO, AnyMoE) exercise it end-to-end. But the headline deliverable—trainable LoRA/QLoRA over an arbitrary engine model—is NEEDS-BUILDING. The proof obligations of Section 5 hold for the design; they do not assert that the `TrainableLoraLinear` type exists today, because it does not.

Var-backed model construction is the shared blocker. Both full SFT and a real GRPO `Policy` require building the roughly 80 engine models with `Var` parameters rather than frozen `Tensor`s. This is a construction-path change, not a kernel change, but it is a prerequisite for two of the three highest-value items.

Performance is parity, not dominance. Decode is 0.98× llama.cpp and prefill 0.86× (Section 7); the matvec is at the bandwidth wall and is not a further lever, while the prefill gap is in engine attention and awaits a WMMA flash-attention kernel (ASPIRATIONAL). The training argument is the absence of a conversion boundary, not raw speed.

Single-device scope. We deliberately do not replicate FSDP/DeepSpeed-scale multi-node sharding, tensor/sequence/context parallelism, fused Liger/Unsloth kernels, or Mamba/diffusion trainers. These are `gym` breadth, not the personalization core. The first deliverable is single-device LoRA/QLoRA + SFT + GRPO, which covers the dominant fine-tune and RL use cases.

Optimizer-state durability. Only `Var` weights are checkpointed today, not optimizer moments; resumable training requires the ergonomics work in item 5 of Section 8.

Future work. In priority order: ship `TrainableLoraLinear` and its QLoRA variant; add the `Var`-backed model path; wire a real transformer `Policy`; replace the SFT stub; add preference losses; then DoRA, a BitDelta trainer, and a Rust port of the federated DeltaSoup control plane.

11 Conclusion

The hanzo training substrate is real: a `candle`-derived autograd, AdamW/SGD, VarMap checkpointing, a differentiable loss family, and a unified quant core that serves `Q8_0` decode at parity (24.5 tokens/s, 0.98× llama.cpp) and prefill at 0.86×. Two trainers prove it trains end-to-end: the GRPO RL loop (algorithm-complete, tested, ported from `zen/gym`) and the AnyMoE in-engine gate trainer. The QLoRA primitive—a detached `dequantize()` base plus trainable `bf16` adapter `Var`s—is present in the code; only the adapter *layer* is missing. We formalized the consequences: the trainable footprint is the adapter, not the base (Proposition 1), and linear adapters over a shared base compose additively (Lemma 1). The build order is clear: `TrainableLoraLinear` +

QLoRA over `qmatvec_core`; a real-transformer `Policy` to unlock GRPO; a real SFT loop with a tokenized dataset; then preference losses and ergonomics. The end state is to train and serve a model in one Rust engine, one quant format, one device backend—with QLoRA adapters learned over the exact kernel that serves them.

References

[1] Hanzo AI Research. (2026). *Native ROCm Inference on gfx1151 / Radeon 8060S: Unified Quantization Kernels and the Path to Bandwidth-Wall Decode*. Hanzo Industries Research. Performance record, workflows `wl14hz0o7` and `wsvtp8gru`.

[2] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv:2106.09685.

[3] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. NeurIPS 2023.

[4] Liu, S.-Y., Wang, C.-Y., Yin, H., Molchanov, P., Wang, Y.-C. F., Cheng, K.-T., & Chen, M.-H. (2024). *DoRA: Weight-Decomposed Low-Rank Adaptation*. ICML 2024.

[5] Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., et al. (2024). *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. arXiv:2402.03300. (Group Relative Policy Optimization.)

[6] Wortsman, M., Ilharco, G., Gadre, S. Y., Roelofs, R., Gontijo-Lopes, R., et al. (2022). *Model Soups: Averaging Weights of Multiple Fine-Tuned Models Improves Accuracy Without Increasing Inference Time*. ICML 2022.

[7] Liu, J., Xiao, G., Li, K., Lee, J. D., Han, S., Dao, T., & Cai, T. (2024). *BitDelta: Your Fine-Tune May Only Be Worth One Bit*. NeurIPS 2024.

[8] Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., & Finn, C. (2023). *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. NeurIPS 2023.