Compositional Machine Learning and AI on Distributed Systems

An introduction to higher categorical semantics and executable specifications

Anonymous

Abstract. This introduction develops machine learning and artificial intelligence through three kinds of composition. Parametric maps and reverse differentiation describe prediction and training. Stochastic kernels describe conditioning, decisions, and the probability rows of attention. Higher comparisons describe changes of parameters, coherent regroupings of shards, and descent for replicated state. We prove that synchronous data-parallel execution agrees with the specified serial batch update in exact arithmetic and derive a stable reduction for sequence-sharded attention. Worked examples connect these claims to PyTorch distributed collectives and JAX named meshes, automatic differentiation, and StableHLO. Companion programs reproduce training and attention on CPU and expose the operational assumptions needed to refine a mathematical specification into an executable one.

Keywords Parametric maps, reverse differentiation, Markov categories, descent, data parallelism, StableHLO

1 Introduction

A learning system combines several maps. A predictor takes data and weights to a prediction. A loss assigns a number to a prediction and its target. An optimizer consumes a derivative and produces new weights. A distributed runtime divides the work and exchanges messages. The choice of how to divide it becomes part of the correctness question as soon as a worker can see a different batch size, parameter version, or order of additions.

For example, start an affine predictor with weight and bias zero and fit the four samples $(x, y) = (i, i)$ for $i = 0, \dots, 3$. The mean gradient is $(-7/2, -3/2)$. If one worker receives the first record and a second receives the other three, averaging their two local means gives $(-7/3, -1)$. The discrepancy arises at the interface between the objective and the collective. The mathematical remedy is to reduce a pair consisting of an unnormalized gradient and a count, then divide once.

The paper follows that interface through several levels. Sections 2–4 develop models, comparisons, and derivatives. Sections 5–7 introduce stochastic composition, a planning operator, and attention. Sections 8–10 connect coherent layout changes and replica compatibility to legal program lowerings. The primary derivations use finite index sets and familiar array operations. Higher cells record which changes preserve an interpretation and how such changes compose.

We assume elementary linear algebra, multivariable differentiation, and finite probability. The first examples can be calculated by hand. Each execution claim identifies the arithmetic model, placement assumptions, and reference output used in the companion programs.

The present treatment follows earlier work on compositional learners and categorical architecture [6, 7, 3]. The venue has published broader accounts of learning semantics and heterogeneous compilation [30, 31]. Here the emphasis is on short proofs that readers can follow into a typed schedule, a process-group program, or a lowered array program.

2 From models to higher morphisms

A model can be described by a family $f_p \colon X \to Y$ of predictions indexed by a parameter $p \in P$. A training program adds a loss, a rule for producing parameter updates, and a source of observations. A distributed implementation adds choices about data layout, communication, and execution order. These descriptions support different comparisons. Parameter tying compares two families of predictions, differentiation compares their sensitivities, and a scheduling transformation compares two ways to evaluate the same specified program.

An ordinary category has objects, arrows, and composition. A monoidal category also has a product $\otimes$ for placing two computations side by side. String diagrams represent arrows by boxes and object types by wires. Connecting adjacent boxes represents composition, and independent wires represent the monoidal product. Tensor is a bifunctor, so for well-typed f, f', g, g' one has

$$(f' \otimes g') \circ (f \otimes g) = (f' \circ f) \otimes (g' \circ g). \tag{2.1}$$

The equation permits a compiler to interleave independent stages once their input and output types agree. A cost model and an execution environment determine whether this independence yields a speedup. See Selinger [1] for the graphical calculus and its coherence results.

Definition 2.1 (Parametric maps)**.** Let **Euc** have finite-dimensional Euclidean coordinate spaces as objects and smooth maps as arrows. A parametric arrow $X \to Y$ is a pair (P, f) with $f \colon P \times X \to Y$. If (Q, g) is an arrow $Y \to Z$, their composite has parameter $Q \times P$ and evaluation

$$((q, p), x) \longmapsto g(q, f(p, x)).$$

The identity uses the one-point parameter space and evaluates to x. A 2-morphism $(P, f) \Rightarrow (Q, g)$ is a smooth

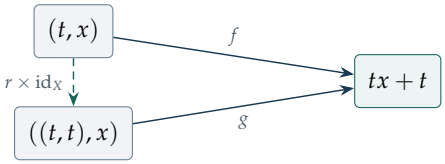


Figure 1. The parameter map $r(t) = (t, t)$ makes the two evaluations agree. Its reverse pass sends the two parameter cotangents to their sum.

map $r\colon P \to Q$ satisfying

$$f = g \circ (r \times \mathrm{id}_X). \tag{2.2}$$

Proposition 2.2. *These objects, arrows, and 2-morphisms form a bicategory* Para(**Euc**).

Proof. Vertical composition is composition of parameter maps. Horizontal composition sends $r\colon P \to P'$ and $s\colon Q \to Q'$ to $s \times r$ because substitution into the formula for composite evaluation preserves (2.2). Functoriality of the product gives interchange for vertical and horizontal composition. The canonical rebracketing and unit maps of products give invertible associators and unitors. Along each side of the pentagon and triangle, every resulting projection selects the same input coordinate, which proves coherence by the product universal property. □

The hom between X and Y is now a category whose objects are models and whose arrows are changes of parameter. Rebracketing a pipeline is an invertible comparison. Tying two independent parameters is typically a noninvertible comparison. The distinction is intrinsic to bicategories, whose associativity and unit constraints are coherent invertible 2-morphisms while general 2-morphisms may be noninvertible [2]. The use of parametric maps in a theory of network architecture is developed further in [3].

Example 2.3 (Parameter sharing)**.** The family $g((w, b), x) = wx + b$ has two independent parameters. The family $f(t, x) = tx + t$ uses one parameter in two places. The diagonal $r(t) = (t, t)$ defines a 2-morphism $f \Rightarrow g$ by (2.2), as in Figure 1. A compiler can realize the diagonal with two uses of the same parameter buffer. The reverse computation must add their contributions to that buffer's cotangent.

A further level of comparison. In an $(\infty, 1)$-category, morphisms between objects belong to mapping spaces. Paths compare morphisms, and higher simplices encode coherent comparisons between such paths [4]. A bicategory with arbitrary directed 2-morphisms and a mapping-space model with invertible higher comparisons describe different structures. Both can appear at once. For continuous parameter spaces P, Q and models f, g, take the compactly generated subspace

$$\mathcal{R}(f, g) = \{r \in C(P, Q) \mid f = g \circ (r \times \mathrm{id})\}$$

of the compactly generated function space and form its singular simplicial set. Its points are parameter comparisons and its paths are continuous families of such comparisons. Composition of parameter maps composes these families pointwise. For example, if $f(t, x) = tx$ and $g((u, v), x) = ux$, the family $r_s(t) = (t, st)$ for $0 \le s \le 1$ is a path from r_0 to r_1 inside $\mathcal{R}(f, g)$. For constant models with $P = \{*\}$ and $Q = S^1$, the comparison space itself is S^1 and its based loops have integer winding number. Pointwise equality of the associated predictions leaves this higher comparison data visible. The executable constructions below use chosen finite descriptions of morphisms and comparisons.

3 Reverse differentiation as composition

A reverse pass sends a sensitivity of the output back to a sensitivity of the input. A lens packages this pass with the forward map. More generally, for pairs (A, A') and (B, B'), a lens is specified by a forward map $f\colon A \to B$ and a backward map

$$f^{\leftarrow}\colon A \times B' \to A'.$$

All spaces and maps here can be taken in **Euc**. The backward map may depend on the original input. With a second lens $(g, g^{\leftarrow})$, composition has forward map $g \circ f$ and backward map

$$(a, c') \longmapsto f^{\leftarrow}\big(a, g^{\leftarrow}(f(a), c')\big). \tag{3.1}$$

The identity lens has forward identity and backward projection $(a, a') \mapsto a'$. For a chain of three lenses, either bracketing evaluates the same nested backward expression, inserting the respective intermediate forward values. This gives a category. Placing lenses side by side uses products of both pairs of objects and applies each backward map on its own inputs.

For a smooth map $f\colon A \to B$, let A^* and B^* denote the linear duals of the coordinate spaces, using translation to identify tangent spaces at different base points. The reverse derivative, also called a vector–Jacobian product in array software, is the pullback of a cotangent,

$$R[f](a, \bar{b}) = \bar{b} \circ Df(a).$$

Writing covectors as columns in Euclidean coordinates gives $R[f](a, \bar{b}) = Df(a)^{\mathsf{T}}\bar{b}$. In particular the backward action is linear in its second argument.

Proposition 3.1 (Compositional reverse differentiation)**.** *The assignment $A \mapsto (A, A^*)$ and $f \mapsto (f, R[f])$ is a strong symmetric monoidal functor from* **Euc** *with cartesian product to the lens category just defined.*

Proof. The chain rule yields

$$\begin{aligned} R[g \circ f](a, \bar{c}) &= Df(a)^{\mathsf{T}} Dg(f(a))^{\mathsf{T}} \bar{c} \\ &= R[f]\big(a, R[g](f(a), \bar{c})\big), \end{aligned}$$

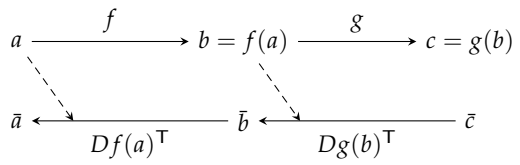


Figure 2. Forward values determine the evaluation points for the reverse pass. Dashed arrows indicate the forward values required by the respective pullbacks.

which is (3.1). The identity derivative gives backward projection. The differential of a product map is block diagonal. Its dual respects the canonical isomorphism $(A \times B)^* \cong A^* \times B^*$, which is compatible with rebracketing, the zero-dimensional unit, and permutation of factors. These isomorphisms make the functor strong symmetric monoidal. □

The same construction gives a reverse rule for fan-out. The diagonal $\Delta(a) = (a, a)$ pulls a pair of cotangents back to their sum,

$$R[\Delta](a, (\bar{a}_1, \bar{a}_2)) = \bar{a}_1 + \bar{a}_2.$$

The derivative of a discard into the unit space returns zero. An implementation can therefore represent a shared subexpression by one node and add all incoming cotangents before propagating to its inputs. Figure 2 shows the sequential case. Reverse derivative categories axiomatize this compositional structure [5]. The use of lenses, parametric maps, and reverse derivatives to express training algorithms is developed in [6, 7].

Reparameterization and the optimizer. For a scalar objective J on the weights and a smooth parameter change r, the preceding proposition gives

$$\nabla(J \circ r)(t) = Dr(t)^{\mathsf{T}} \nabla J(r(t)).$$

Here the gradient vector uses the chosen Euclidean metric. Pullback commutes with evaluation of the scalar objective by construction. Applying a discrete gradient update in parameter coordinates requires a further condition if one wants updates to commute with r. For $r(t) = (t, t)$ and $J(w, b) = \frac{1}{2}(2w + b - 1)^2$, a step of size $1/4$ from $t = 0$ gives $t = 3/4$, hence $(w, b) = (3/4, 3/4)$ under r. A step in the independent coordinates from $(0, 0)$ gives $(1/2, 1/4)$. A parameter-tying 2-morphism thus provides an exact forward relation and a pullback formula. A compiler that claims to preserve training trajectories must additionally preserve the parameterization and optimizer semantics used by the specification.

4 A supervised learner from its typed parts

A supervised prediction model $f\colon P \times X \to Y$ is trained from data in $X \times Y$ using a loss $\ell\colon Y \times Y \to \mathbb{R}$. For a population law μ on $X \times Y$, the population risk is

$$J_\mu(p) = \int \ell(f(p, x), y)\, d\mu(x, y)$$

Table 1. Per-observation reverse pass for affine regression at $(w, b) = (0, 0)$.

(x, y)	e	$\partial_w \ell$	$\partial_b \ell$
$(0, 0)$	0	0	0
$(1, 1)$	-1	-1	-1
$(2, 2)$	-2	-4	-2
$(3, 3)$	-3	-9	-3
Total		-14	-6

when the integral exists. For a nonempty finite multiset D of observations, write $J_D(p)$ for the average of the same integrand over D. The algorithm considers J_D, while a generalization claim concerns the relationship between J_D and J_μ and requires statistical assumptions.

The typed program for an empirical loss sends p and each observation (x, y) through f and ℓ, then averages the scalar losses. Seeding the reverse pass with 1 yields $\nabla J_D(p)$. For gradient descent with step size $\eta > 0$, the update is

$$p^+ = p - \eta \nabla J_D(p).$$

For a stateful optimizer one instead includes a state space S and a transition step: $S \times P \times P^* \to S \times P$. A run iterates this transition at successive parameter values. The choice of observations per iteration, the initial state, and the placement of the optimizer transition are part of the program. Stochastic gradient descent samples a minibatch, evaluates its mean gradient, and applies the same transition. The name refers to the stochastic selection of data, conditional on whatever randomness the model itself uses.

Example 4.1 (An exact gradient and update)**.** Take $p = (w, b)$, $f(p, x) = wx + b$, and $\ell(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$. For one training pair, let $e = wx + b - y$. The output seed 1 first pulls back to e through the loss and then pulls back to (xe, e) through the model. Table 1 gives these values for $p = (0, 0)$ and observations $(0, 0), (1, 1), (2, 2), (3, 3)$.

The gradient of the mean loss is $(-7/2, -3/2)$, and $\eta = 1/4$ gives $(w^+, b^+) = (7/8, 3/8)$. The four new errors, in input order, are $3/8, 2/8, 1/8, 0$. The mean loss falls from $7/4$ to $7/256$. Every number in this step has a terminating binary expansion, so this example provides an exact reference for implementations of the specified operations in standard binary floating point. For partitions of the observations, summing the local gradient totals and local counts produces the same mean as this table.

A larger model replaces the affine box by a composite with many tensors. The parameterization specifies which wires share weights, the reverse program applies the local pullbacks in reverse dependency order, and the optimizer consumes the resulting parameter cotangent. The passage from a single evaluation to a distributed training step uses an additional property of the empirical objective. Its finite sum allows each shard to contribute a gradient total at the same parameter value before the global update.

4.1 Classification and generalization

For k classes, a predictor can output logits $z \in \mathbb{R}^k$. The softmax probabilities are $\pi_j = e^{z_j} / \sum_i e^{z_i}$. With correct class t, the cross-entropy is

$$L(z,t) = \log\left(\sum_i e^{z_i}\right) - z_t.$$

The pullback of seed 1 to the logits has coordinates $\pi_j - \delta_{jt}$. This follows by differentiating the log-sum-exp term and the selected coordinate. If the logits are $(0,0)$ and the correct class is the second, then $L = \log 2$ and the pullback is $(1/2, -1/2)$. PyTorch's logit-level `cross_entropy` and its reverse pass reproduce these two values [20]. The same reduction in Theorem 9.1 applies to sums of such losses when each example's forward path is local.

For a classifier with decision function $\arg\max_j z_j$, fix a tie rule and measure prediction error on an independent labeled sample. Accuracy concerns the discrete decisions, while the smooth loss supplies a derivative for training. A comparison of shards can validate that the training step optimizes the specified empirical objective. Extrapolating error to future data requires conditions on the distribution and on how the held-out sample was used.

5 Probability as composable information flow

A deterministic predictor specifies one output for each input. A probabilistic model can instead specify a distribution for each input. For finite sets, write $K(y \mid x)$ for a nonnegative array with $\sum_y K(y \mid x) = 1$. Think of $K : X \to Y$ as a channel that draws a value y when supplied with x. The category FinStoch has finite sets as objects and these stochastic kernels as arrows. For $K : X \to Y$ and $L : Y \to Z$, the composite is

$$(L \circ K)(z \mid x) = \sum_{y \in Y} L(z \mid y) K(y \mid x). \tag{5.1}$$

The identity kernel is $\delta_{x,x'}$. Interchanging two finite sums proves associativity. A deterministic function $f : X \to Y$ embeds as $K_f(y \mid x) = \delta_{y,f(x)}$. The tensor product of objects is the set product. The tensor of two channels multiplies their probabilities,

$$(K \otimes L)(y, v \mid x, u) = K(y \mid x) L(v \mid u).$$

Thus the two draws are independent conditional on the given pair of inputs. A distribution on X is a state $I \to X$, where I is a singleton.

Copying and discarding. A Markov category is a symmetric monoidal category equipped with compatible commutative comonoids $\Delta_X : X \to X \otimes X$ and $!_X : X \to I$ on its objects, with I terminal [8]. In FinStoch, these maps copy a sampled value and discard it, respectively. Copying after a channel produces the joint law $K(y_1 \mid x)\delta_{y_1,y_2}$. Copying the input and applying the channel twice produces $K(y_1 \mid x)K(y_2 \mid x)$, which can assign positive probability to unequal pairs. The equation

$$\Delta_Y \circ K = (K \otimes K) \circ \Delta_X$$

characterizes deterministic kernels in this example. Indeed, if a row assigned positive probability to two distinct outcomes, the right-hand side would be positive off the diagonal and the left-hand side would vanish there.

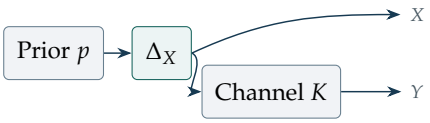


Figure 3. A prior and a signal channel produce a joint law. The upper wire retains the sampled source so that source and signal can be conditioned jointly.

Bayesian inversion. Given a prior $p : I \to X$ and a channel $K : X \to Y$, copy X before applying K to form the joint law in Figure 3,

$$j(x,y) = p(x)K(y \mid x).$$

Let $q(y) = \sum_x j(x,y)$. For $q(y) > 0$, the conditional $K_p^\dagger : Y \to X$ is given by

$$K_p^\dagger(x \mid y) = \frac{p(x)K(y \mid x)}{q(y)}. \tag{5.2}$$

At any y with $q(y) = 0$, a row may be chosen arbitrarily when X is nonempty. The conditional reconstructs j by drawing $y \sim q$, copying y, and applying the conditional to one copy. This is the finite case of a disintegration. The prior is an input to the construction, and the result is determined on the support of q. Existence of the relevant conditional in a general Markov category requires an additional hypothesis.

Example 5.1 (A noisy signal)**.** Let $X = \{H, L\}$ and $Y = \{+, -\}$, with

$$p(H) = \tfrac{1}{4}, \qquad K(+ \mid H) = \tfrac{4}{5}, \qquad K(+ \mid L) = \tfrac{1}{5}.$$

The joint masses for $(H,+), (H,-), (L,+), (L,-)$ are $4/20, 1/20, 3/20, 12/20$. Hence $q(+) = 7/20$ and $q(-) = 13/20$, giving

$$K_p^\dagger(H \mid +) = \tfrac{4}{7}, \qquad K_p^\dagger(H \mid -) = \tfrac{1}{13}.$$

The entries of the reverse kernel describe updated uncertainty about the source. They depend on the prior and the observed signal.

6 Decision kernels and value iteration

A finite Markov decision process specifies states S, a finite nonempty set A_s of admissible actions for each state, a transition kernel $P(s' \mid s, a)$, a bounded real reward $r(s,a,s')$, and $0 \le \gamma < 1$ [9]. Write $E = \{(s,a) : s \in S,\ a \in A_s\}$. A randomized stationary policy defines

a kernel $S \to E$ supported on actions at the supplied state, and the environment defines a kernel $E \to S$. Their composition has matrix

$$P_\pi(s' \mid s) = \sum_{a \in A_s} \pi(a \mid s) P(s' \mid s, a).$$

Its powers describe the successive state marginals for this policy. Retaining the intermediate states using the copy maps gives the joint law on finite trajectories. A learned policy may depend on parameters θ, so calibration of a transition model and optimization of a policy correspond to distinct parts of the specification.

Given a candidate value $V : S \to \mathbb{R}$, one planning step forms the action values and then reduces over the actions.

$$\begin{aligned} Q_V(s,a) &= \sum_{s' \in S} P(s' \mid s,a)\big[r(s,a,s') + \gamma V(s')\big], \\ (TV)(s) &= \max_{a \in A_s} Q_V(s,a). \end{aligned} \tag{6.1}$$

For a fixed policy, replacing the maximum by $\sum_a \pi(a \mid s) Q_V(s,a)$ gives its policy evaluation operator. One can compute the expectation for each action on separate workers before taking the statewise maximum. If next-state contributions are distributed, each worker sums its assigned contributions with their original probabilities. The sums for an action are combined before the maximum is taken.

Proposition 6.1. *The optimal-control Bellman operator T is a γ-contraction on $\mathbb{R}^S$ with the supremum norm. Iteration from any initial value converges to its unique fixed point.*

Proof. For each pair (s,a), the absolute difference $|Q_V(s,a) - Q_W(s,a)|$ is bounded by $\gamma \|V - W\|_\infty$, since the coefficients $P(s' \mid s,a)$ are nonnegative and sum to one. The difference of two finite maxima is bounded in absolute value by the maximum of the pointwise differences in absolute value. Taking the supremum over s proves the contraction bound. The finite-dimensional value space is complete, so the contraction mapping theorem applies. □

Example 6.2 (An immediate or delayed reward)**.** At state s, taking the immediate action yields reward 1 and enters an absorbing state t. The delayed action yields zero and enters c. The only action at c yields reward 3 and enters t, whose subsequent reward is zero. Figure 4 shows the transition graph. Take $\gamma = 1/2$ and start from $V_0 = 0$. Then $V_1(s) = 1$, $V_1(c) = 3$, and $V_1(t) = 0$. At the second iteration, the action values at s are 1 and $3/2$, so the maximizing action changes to the delayed choice. The two iterations differ because the second has propagated the reward at c back to s.

The companion program executes action-level Bellman backups in two worker processes with exact rational arithmetic and combines them on the coordinator. A sampled environment API, such as Gymnasium's `step`, instead supplies one observed transition at a time [29]. Exact planning as written above requires the model probabilities and rewards. Data from a simulator can be used to estimate those quantities or to drive a model-free learning algorithm.

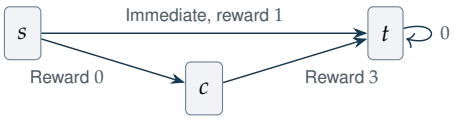


Figure 4. Value iteration propagates the delayed reward from c to s on its second step. State t is absorbing.

7 Attention as a stochastic kernel and a reduction

Let $q_i, k_j \in \mathbb{R}^{d_k}$ and $v_j \in \mathbb{R}^{d_v}$ be query, key, and value vectors. Let J_i be the nonempty set of eligible key indices for query i. With finite scores $s_{ij} = q_i^{\mathsf{T}} k_j / \sqrt{d_k}$, scaled dot-product attention without dropout is

$$\begin{aligned} A(j \mid i) &= \frac{\exp(s_{ij})}{\sum_{k \in J_i} \exp(s_{ik})}, \qquad j \in J_i, \\ o_i &= \sum_{j \in J_i} A(j \mid i) v_j. \end{aligned} \tag{7.1}$$

Set $A(j \mid i) = 0$ on excluded indices. For fixed query and key arrays and a fixed mask, A is a finite stochastic kernel from query indices to key indices. Given the value array, o_i is the expectation of the deterministic value map $j \mapsto v_j$ under its i-th row. In self-attention the three arrays are ordinarily projections of a shared input representation, so the kernel and the value map both vary as that input changes. The full layer is a deterministic tensor operation in this formulation [10]. PyTorch exposes this operation as `scaled_dot_product_attention`. For this specification select zero dropout, the intended mask, and the same scale in both implementations [11].

Shard summaries. Fix one query, suppress its index, and let $B \subseteq J_i$ be the eligible indices on a shard. A nonempty shard emits

$$\begin{aligned} m_B &= \max_{j \in B} s_j, \\ \ell_B &= \sum_{j \in B} e^{s_j - m_B}, \\ u_B &= \sum_{j \in B} e^{s_j - m_B} v_j. \end{aligned} \tag{7.2}$$

An empty shard emits a distinct identity element $\perp$. The centered normalizer is the online softmax summary of Milakov and Gimelshein [12]. The vector numerator permits the same rescaling to accumulate the attention output. FlashAttention uses blockwise rescaling and tiling within a GPU memory hierarchy [13]. Here the summary defines a reduction across disjoint shards.

Proposition 7.1 (Exact attention under sharding)**.** *Over the reals, $\{\perp\} \sqcup (\mathbb{R} \times \mathbb{R}_{>0} \times \mathbb{R}^{d_v})$ is a commutative monoid*

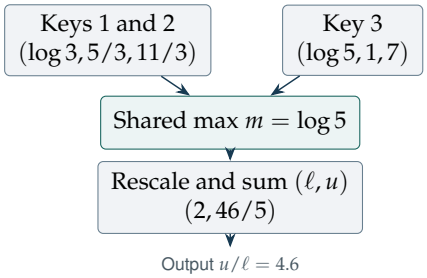


Figure 5. Stable sequence-sharded attention uses maximum and sum collectives. Each worker uses the shared maximum to express its contribution in a common scale.

for the merge below. If the shards partition J_i, the merged summary (m, ℓ, u) yields $o_i = u/\ell$.

Proof. For two nonempty summaries, set $m = \max(m_1, m_2)$ and $a_k = e^{m_k - m}$ for $k = 1, 2$. Define their merge as

$$\big(m,\ a_1\ell_1 + a_2\ell_2,\ a_1u_1 + a_2u_2\big).$$

Extend by declaring $\perp$ an identity. On nonempty summaries, the bijection

$$(m, \ell, u) \longmapsto (m, e^m\ell, e^m u)$$

carries merge to the product of the maximum operation on the first coordinate and addition on the two remaining coordinates. Their associativity and commutativity prove the claim. For an actual shard, the two rescaled coordinates are $\sum_{j\in B} e^{s_j}$ and $\sum_{j\in B} e^{s_j}v_j$. Their quotient after merging all shards is exactly (7.1). □

Example 7.2 (Two workers and one query)**.** Take scores $\log 2, \log 3, \log 5$ and scalar values $1, 3, 7$. The first worker holds the first two pairs and emits $(\log 3, 5/3, 11/3)$. The second worker emits $(\log 5, 1, 7)$. The merged summary is $(\log 5, 2, 46/5)$, giving $o_i = 23/5 = 4.6$. Direct evaluation yields the same result from the probability row $(2/10, 3/10, 5/10)$.

Collective implementation. A backend with maximum and sum collectives can realize the same specification with two reductions. First compute the global maximum m of all nonempty shard maxima. Then sum the concatenated vectors $e^{m_B - m}(\ell_B, u_B)$, letting empty shards contribute zeros. Each worker can divide the resulting numerator by the resulting normalizer. An implementation must declare its behavior for an entirely masked row, since the mathematical probability row presumes at least one eligible key. Query replication or routing to the shards is a separate placement decision. In floating point, the reduction tree, accumulation precision, and score implementation can change rounding. A comparison with dense attention can use the absolute output error $\|o_{\text{shard}} - o_{\text{dense}}\|_\infty$, together with a tolerance stated for the chosen dtype and inputs.

7.1 From attention to a Transformer block

An input $X \in \mathbb{R}^{n\times d}$ can produce a head by three parameterized maps $Q = XW_Q$, $K = XW_K$, and $V = XW_V$, with shared input implemented by fan-out. Position information can already be present in X, enter the projections, or enter the logits, depending on the model. With finite scores and an eligible key per row, the head sends X to the rowwise expectation described above. Several heads take the same X and run their own projections in parallel, followed by concatenation and an output projection. A residual step adds a compatible copy of X. A feed-forward block applies a shared nonlinear map to all positions. This illustrates how a head can have its own parallel schedule inside a compositional model [10].

An exact symmetry is available at the level of the head. Simultaneously relabeling the keys, their values, and the mask for a fixed query preserves the output. Indeed the numerator and denominator of the expectation reindex by the same bijection. Relabeling query rows as well permutes the output rows. This gives a simple semantic check on a layout conversion. For causal attention the allowed-index relation must move with the keys and queries. Restoring an old triangular mask after an arbitrary permutation can change the observed context, and positional scores must follow the intended logical positions.

In autoregressive use the resulting network can produce logits for the next token. A softmax over the vocabulary defines a kernel from each prefix to the next token. Repeatedly extending a prefix with a sampled token composes explicit prefix-state transitions. The stochastic step here is token selection, while the attention-row distribution describes how values are weighted at fixed model input. A training loop, a deterministic logits service, and a sampled generator therefore have distinct interfaces even when they share the same trained weights.

8 Coherent layouts and consistent snapshots

8.1 Refining a layout

Let I be a finite set of logical coordinates. These may be record identifiers, parameter indices, or blocks of a tensor. A layout is a map $\ell\colon I \to S$ to a finite set of shard labels. The fiber $\ell^{-1}(s)$ lists what worker s owns. A refinement of ℓ is a layout $\ell'\colon I \to S'$ together with a map $q\colon S' \to S$ such that $q\ell' = \ell$. This is an arrow from ℓ' to ℓ in the coslice category $I/\mathbf{FinSet}$. This convention permits idle workers, whose fibers are empty. For example, a four-record batch with identifiers $0, 1, 2, 3$ may be assigned to shards $\{0\}$, $\{1\}$, and $\{2, 3\}$. Merging the first two labels makes the coarse layout $\{0, 1\} \mid \{2, 3\}$.

To give the layout a semantics, let $A = (A_i)_{i\in I}$ be a family of sets containing the possible values at each coordinate. A map $f\colon I \to S$ gives a functor $f_*\colon \mathbf{Set}^I \to \mathbf{Set}^S$ by

$$(f_*A)_s = \prod_{\{i:f(i)=s\}} A_i.$$

On maps of families, f_* acts coordinatewise. The product over an empty fiber is a singleton. This is the right adjoint to pullback of families along f, since maps into a product are families of maps into its factors. Equivalently it is a right Kan extension along f with I and S viewed as discrete categories [14].

Proposition 8.1 (Coherent refinement)**.** *For a refinement $q\ell' = \ell$ there is a canonical natural isomorphism*

$$\alpha_{q,\ell'} : q_*\ell'_* \xRightarrow{\cong} \ell_*.$$

The isomorphisms for a chain of refinements agree under any parenthesization, and those for identity refinements give the canonical unit maps. Thus $I \mapsto \mathbf{Set}^I$ and $f \mapsto f_*$ form a pseudofunctor from **FinSet** *to* **Cat**.

Proof. At each label s, a value on the left is a tuple of tuples. Send it to the tuple whose ith entry is the projection with address $(\ell'(i), i)$. The inverse groups entries by $\ell'(i)$. Mapping each entry through a function commutes with this reindexing, giving naturality. Along any refinement chain, each leaf has a unique address ending in i. Both routes around the pseudofunctor associativity diagram take the projection with that address. The universal property of the product makes the routes equal. Likewise inserting or deleting an identity refinement preserves the projection at every leaf. □

The objects of **Cat** here are categories of layouts' values, the 1-cells are functors that group values, and the 2-cells are natural isomorphisms that compare regroupings. This supplies a compiler justification for changing the nesting of shard tuples while tracking each logical coordinate. A second justification is needed to replace tuples by summaries. For a commutative monoid $(M, \oplus, 0)$ and per-record contributions $m_i \in M$, the summary on shard s is

$$R_s = \bigoplus_{i \in \ell^{-1}(s)} m_i.$$

Every refinement tree yields $\bigoplus_{i \in I} m_i$ at its root if each leaf contributes exactly once. Apply associativity at internal nodes, the unit law at empty leaves, and commutativity when shard order changes. In a gradient update an appropriate m_i is the pair consisting of a gradient sum and a sample count, with normalization applied after the final reduction. For floating-point kernels the regrouping theorem applies to the exact model. A compiler can fix both operand order and tree shape for a specified floating-point implementation, or check its results within a stated error tolerance. Overlapping replicas also require a rule for selecting one contribution per identifier.

8.2 Gluing a versioned snapshot

Replication allows a coordinate to appear in more than one shard. Write $U_a \subseteq X$ for the coordinates visible to replica a, where X is a finite discrete space. Fix a version v of the model and let A_x be the set of possible values of coordinate x at that version. For an open set $U \subseteq X$ put $\mathcal{P}_v(U) = \prod_{x \in U} A_x$. Its restriction maps forget entries and leave the version fixed.

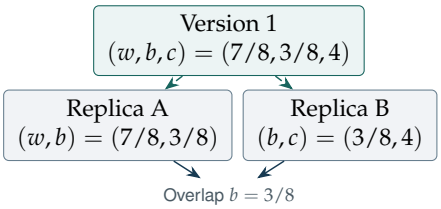


Figure 6. Two local views determine the versioned snapshot when their restrictions agree. The dashed arrows denote restrictions from the resulting global snapshot.

Proposition 8.2 (Snapshot descent)**.** *For a cover $U = \bigcup_a U_a$, the restriction map identifies $\mathcal{P}_v(U)$ with the equalizer of the two overlap restriction maps*

$$\prod_a \mathcal{P}_v(U_a) \rightrightarrows \prod_{a,b} \mathcal{P}_v(U_a \cap U_b).$$

In particular $\mathcal{P}_v$ is a sheaf in $\mathrm{Sh}(X)$.

Proof. Every global tuple restricts to a compatible family. Conversely, choose for each x any U_a containing x and take the value of the ath tuple there. Equality on intersections makes the independent choices agree. The result restricts to each local tuple, and the cover makes it unique. This is exactly the equalizer property. □

Example 8.3. Let the coordinates be weight w, bias b, and sample count c. The shard $\{w, b\}$ reports $(7/8, 3/8)$ and the shard $\{b, c\}$ reports $(3/8, 4)$ at version 1. The equalizer yields $(7/8, 3/8, 4)$ as in Figure 6. With $b = 1/2$ on the second shard the pair of restrictions fails the equalizer test. A snapshot assembler can therefore check versions and repeated coordinates before returning a model. How a system chooses a committed snapshot, waits for a missing read, or resolves an incompatible pair depends on its storage protocol.

8.3 Higher descent for changes of coordinates

The finite discrete topos has a transparent description. Restriction to singletons gives $\mathrm{Sh}(X) \simeq \prod_{x \in X} \mathbf{Set}$. With spaces as values one similarly has $\mathrm{Sh}(X; \mathcal{S}) \simeq \prod_{x \in X} \mathcal{S}$. In both cases a section on U is the product of its singleton sections because the singletons form a disjoint covering. For an introductory homotopical example, replace A_x by a groupoid B_x, whose objects are local presentations of a coordinate and whose arrows are invertible changes of presentation. Define $\mathcal{B}(U) = \prod_{x \in U} B_x$.

Descent data consist of local objects b_a and isomorphisms $g_{ab} : b_a|_{U_a \cap U_b} \to b_b|_{U_a \cap U_b}$. Require $g_{aa} = \mathrm{id}$ and $g_{bc} g_{ab} = g_{ac}$ on triple overlaps. Morphisms of descent data are families of isomorphisms intertwining the g_{ab}. A global object induces such data with identity transitions. This comparison is an equivalence of groupoids. Indeed, for each x choose one index $a(x)$ containing it and take the corresponding object of B_x as the global object at

x. The arrows $g_{a(x)b}(x)$ identify it with the object from each U_b. The triple condition makes these identifications compatible. A morphism between two global objects is determined coordinatewise, so compatible local morphisms glue uniquely. This proves essential surjectivity and full faithfulness and computes the homotopy limit of the cover diagram in this 1-truncated setting [15, 16].

For instance let all three replicas contain the same coordinate and let its groupoid have one object with automorphism group $C_2 = \{1, \tau\}$. Here τ can represent a swap of two permitted labels for the local view. The transitions $g_{12} = g_{23} = \tau$ and $g_{13} = 1$ satisfy the triple test, since $\tau^2 = 1$. Changing g_{13} to τ violates the test. The descent condition expresses compatibility of comparisons even when each pair admits an isomorphism. In the example every valid datum glues, and the resulting global object retains its automorphism group. The same construction gives a specification for checking declared transport maps when a runtime allows coordinate aliases.

9 The synchronous training theorem

Consider a global batch indexed by a nonempty finite set I. Let $D_i = (x_i, y_i)$, let p be the parameter value shared by all workers at the start of the step, and write $L_i(p) = \ell(f(p, x_i), y_i)$. Each occurrence in a batch has its own identifier, so repeated observations can have distinct indices. With $P = \mathbb{R}^d$, the record summary is

$$\phi_p(i) = \big(\nabla L_i(p), L_i(p), 1\big) \in \mathbb{R}^d \times \mathbb{R} \times \mathbb{N}. \tag{9.1}$$

The summary type has coordinatewise addition and a zero. It carries all quantities needed by the example's update and loss report.

Theorem 9.1 (Synchronous data parallelism)**.** *Suppose every L_i is differentiable at a common p. Assign each index to exactly one worker, and let worker s sum $\phi_p(i)$ over its own indices. Suppose every worker receives the global sum (G, L, N) and applies the same deterministic optimizer transition to $(p, G/N)$ and a common optimizer state. In exact arithmetic the resulting weights, optimizer state, and mean loss L/N agree with a serial step on the full batch.*

Proof. The layout fibers partition I. Their summary sum equals $\sum_{i \in I} \phi_p(i)$ by associativity, commutativity, and the unit law. Its count is $|I| > 0$. Linearity of differentiation for finite sums gives

$$\frac{G}{N} = \nabla \left(\frac{1}{|I|} \sum_{i \in I} L_i \right)(p).$$

The loss coordinate gives the same mean by the definition of J_D. Each optimizer receives the serial state, parameter, and cotangent, hence returns the serial result. For several steps, induction applies when every step begins with the common state and uses the same indexed batch as the serial reference. □

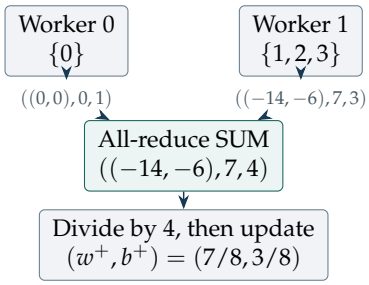


Figure 7. Uneven batches yield equal global updates when workers add totals and normalize by the global count. The parameter starts at $(0, 0)$ on both workers.

A useful boundary. An empty worker contributes zero but still participates in the collective. A batch-dependent operation can violate the hypothesis that each record has a local loss $L_i(p)$. For instance, local batch normalization uses each shard's own statistics. A global implementation must first compute the specified statistics with their own communication and differentiate through the resulting global operation. Likewise, a contrastive loss with cross-example negatives requires the defined cross-shard dependencies. A regularizer of the form $J_D(p) + \lambda R(p)$ contributes $\lambda \nabla R(p)$ once to the final normalized cotangent.

Randomness and versioning are further inputs to the theorem. To compare a serial and distributed stochastic step with identical examples and draws, choose a common per-example random stream keyed by the step and record identifier, and specify any randomness shared across records separately. Rank-local optimizer steps on different states, stale gradients, and changing worker membership define other transition systems. With fixed membership, failed collectives require a run-level failure or recovery rule that preserves the step epoch and collective order.

Arithmetic. The reduction theorem identifies real-arithmetic results and admits arbitrary regrouping in its commutative monoid. Binary floating-point addition can violate that equivalence. For instance in binary64 with rounding to nearest, grouping $(2^{54}, -2^{54}, 1)$ from the left gives 1, while grouping from the right gives 0. A reproducibility contract can fix the dtype, rounding and arithmetic policies, and reduction tree, or give an error budget for permitted implementations. MPI explicitly discusses the floating-point consequence of changing reduction evaluation order [17]. The dyadic quantities in Figure 7 are small enough to represent exactly in the demonstrated floating-point computation.

9.1 Beyond data parallelism

A matrix product illustrates a different source of collectives. If the contracted dimension is split across workers, write $X = [X_0\ X_1]$ and $W = [W_0^{\mathsf{T}}\ W_1^{\mathsf{T}}]^{\mathsf{T}}$. Then

$$XW = X_0 W_0 + X_1 W_1.$$

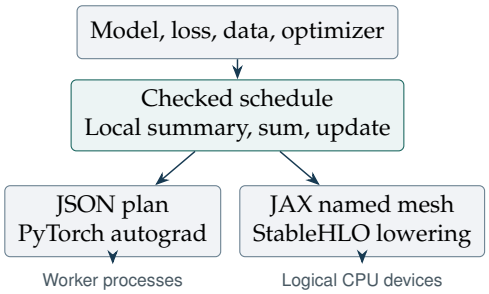


Figure 8. Two realizations of the same checked plan. The PyTorch runner interprets its instructions, and the JAX runner lowers the operations to a compiled mesh program.

Each worker produces a partial result of the full output shape. An all-reduce materializes a replica of the sum, while reduce-scatter produces shards of the sum. With column-sharded outputs and compatible inputs, concatenation instead reconstructs the result. Pipelines split consecutive layers and send their boundary activations forward and cotangents back. Real compiler strategies combine these decompositions and choose schedules using costs for memory, compute, communication, and device topology. Alpa is an example of automatic optimization at both intra-operator and inter-operator levels [18].

The reverse semantics of a collective. With the ordinary sum pairing across physical buffers, the map $A(g_1, \dots, g_s) = (\sum_j g_j, \dots, \sum_j g_j)$ is self-adjoint. Its pullback sums all output cotangents and replicates the result. A logical replicated value can instead represent one point with several presentations. For example, the diagonal $\Delta(p) = (p, \dots, p)$ is pulled back by summation, so using a seed of 1 on every physical copy of one logical scalar objective can multiply its cotangent. This is a reason to specify logical replication and cotangent ownership before lowering. In Theorem 9.1 the local derivatives are evaluated before the global sum, and the sum aggregates one contribution per record.

10 From a specification to an executed plan

A useful specification distinguishes the meaning of a program from the details a backend chooses. The source of the regression example declares a model, a loss, an optimizer, and indexed records. The accompanying layout assigns each record to one worker. A plan then identifies typed buffers, their data dependencies, and the collective calls each participant must enter. Figure 8 shows where the two framework examples make these choices.

10.1 A small lowering with explicit schedules

The companion file `affine.json` specifies an affine scalar model and half squared error. The `train.py` frontend checks finite parameters and records, positive step size and step count, unique record identifiers, and exactly one assignment per record. The target schedule uses a `Float64[2]` parameter row and shard buffers of shape $n_s \times 2$. A summary separates three floating-point totals from one integer count. An initialization broadcasts the parameter row. Each step then runs the operations in Table 2.

Table 2. Per-step abstract instructions of the regression plan.

Instruction	Semantics
`loss_and_vjp`	Evaluate local summed loss and its derivative, and count the local records.
`sum_collectives`	Produce replicated totals and the global count.
`mean_sgd`	Divide the gradient by the count and apply the specified update.

The interpreter gives this fixed instruction set a concrete semantics. PyTorch's `autograd.grad` evaluates the local pullback [19]. In the tested local transport, each worker process sends its summary to the coordinator, which sums the payloads and returns the results to all ranks. A second transport maps the sums to `all_reduce` on a Gloo process group [21]. That transport uses separate calls for the floating-point buffer and the integer count. Its world size and ordered calls are explicit, and ranks check that they interpret the same plan. The report records the loss sum and count before the update.

The following commands lower the example and execute the plan with PyTorch on two worker processes. The archive also includes a standard-library backend and the Gloo launch command.

```
python train.py compile affine.json plan.json
python train.py run plan.json \
  --backend torch-pipe
```

Proposition 10.1 (Correctness of the lowering in the exact model)**.** *For accepted inputs, take the real values of the parsed binary64 inputs. Interpret the subsequent arithmetic and differentiation in Table 2 over the reals. If the broadcast and sums implement their declared contracts, a completed interpretation returns the serial parameter value after the same number of steps on the declared batch.*

Proof. The frontend's partition check gives exactly one local occurrence of each record. Broadcast establishes equality of initial weights. The local instruction computes the shard sum of (9.1) by linearity of differentiation. The remaining instructions implement the sum and optimizer of Theorem 9.1. Replication of the result supplies the common weight required by the next iteration. Induct on the specified step count. □

This argument supplies a compact compiler proof because each instruction has a single interpretation contract. It applies to the small, fixed catalogue implemented in the archive. Backend code is tested by independently evaluating the serial expressions. Extending the instruction set to new model primitives requires their shape rules and

derivative semantics, while adding collectives requires specifying participation, placement, and communication behavior.

10.2 JAX, named meshes, and StableHLO

JAX allows the same calculation to be written directly against a named device mesh. A `NamedSharding` combines a `Mesh` with a `PartitionSpec`. The regression example places rows along the axis `data` and replicates the parameters. An explicit `shard_map` runs the local function over each device, and `lax.psum` sums the loss, parameter cotangent, and integer count along `data` [22, 23].

The demonstrated mesh has equal static shapes on two logical CPU devices. A validity mask makes the effective batch sizes 1 and 3. Padded records are finite, excluded from the loss, and excluded from the count. The program obtains a varying parameter view with `lax.pcast` outside `value_and_grad`. The local differentiated function then sees the ordinary local parameter and loss. This placement matters because an invariant-to-varying cast can transpose to an inter-device sum under autodiff [24]. The program explicitly sums local gradients once.

The stand-alone walkthrough lowers to three `all_reduce` operations in StableHLO with addition bodies. The runner `jax_train.py` consumes the checked JSON plan and lowers a packed variant to two such operations, one for the floating-point totals and one for the integer count. Each operation represents a collective over a group, and the StableHLO specification leaves the reduction-tree shape implementation-defined [25]. Shardy represents mesh and sharding annotations separately [26]. The captured module documents the expressed collectives before backend optimization. The two logical CPUs exercise partition and replication semantics on one host.

Sequence-sharded attention uses the same pattern with different local summaries. The companion JAX program first uses `lax.pmax` to obtain the global maximum and then uses `lax.psum` for the packed rescaled numerator and denominator. The lowered module contains two all-reduces, with maximum and addition bodies in that dependency order. The dense reference uses the softmax expectation from (7.1). In the empty-row case the implementation returns `NaN`, documenting a separate convention for the zero denominator.

10.3 Relating logical layouts to framework operations

PyTorch DTensor names common placement states explicitly [27]. In that vocabulary, a partial sum can be resolved to replicas by an all-reduce or to a sharded result by reduce-scatter. A shard can be materialized as a replica by all-gather. Table 3 lists these as implementation patterns. The choices follow from which logical value a consumer expects.

PyTorch DistributedDataParallel integrates gradient communication with autograd hooks and averages gradients across its participating replicas [28]. Uneven batches require objective-aware scaling or the explicit total-and-count construction used here. For a general data pipeline, a compiler must also track which mesh axis is independent, where a value is available, and how cross-axis collectives are ordered. These are operational obligations in addition to the equations for regrouping.

Table 3. Typical placement translations for a single mesh axis.

Placement change	Operation
Partial sum → replicated	All-reduce SUM
Partial sum → sharded	Reduce-scatter SUM
Sharded → replicated	All-gather
Replicated → sharded	Local slice

Table 4. Selected results from the companion programs.

Execution	Observed result
PyTorch regression on two processes, shards $1, 3$	Sum $(-14, -6)$, count 4, two matching replicas $(7/8, 3/8)$.
JAX regression on two logical devices	Mean loss $7/4 \to 7/256$, same updated parameters.
JAX sharded attention and PyTorch dense attention	Both return 4.6. JAX dense and sharded outputs differ by 0 in the test.
Rational Bellman backup on two processes	Value at s changes from 1 to $3/2$, and the preferred action changes.

10.4 Executable checks

The checks in Table 4 used Python 3.12, PyTorch 2.9.1 CPU, and JAX and jaxlib 0.12.1. The PyTorch worker runs used the local pipe transport. Gloo initialization was blocked by a host interface-permission error. The archive describes the conditions for running that backend on a compatible host and includes the checked plan and captured lowered modules.

The regression checks include an uneven layout, and the plan interpreter also checks an empty worker across consecutive updates against an independently computed serial reference. The attention demonstration checks a partially empty layout and an entirely masked row. The rational examples check Bayes inversion, reject two incompatible snapshots, and evaluate both a valid and an invalid triple of transitions. Together these tests exercise the numerical and structural obligations proved above on inspectable inputs.

11 What the specifications guarantee

Each layer contributes a different correctness condition. Parametric 2-morphisms compare forward evaluations and induce pullbacks of sensitivity. The reverse chain rule makes local derivatives compose. A reduction law justifies a change of shard grouping. The synchronization theorem combines that law with a common step state. Snapshot descent checks that multiple views encode a compatible value at one version. These conditions can be inspected independently in a compiler or runtime trace.

For regression, exact serial agreement is a property of a specified update on a specified finite batch. A statistical claim about prediction would require assumptions about how that batch and the future observations were obtained. For planning, the Bellman contraction concerns a known finite transition model, bounded rewards, and $\gamma < 1$. For attention, the exact merge law concerns eligible keys with finite scores and a fixed query. The experiments compare concrete floating-point implementations on small inputs and expose their communication instructions.

The archive contains the complete LaTeX sources, the declarative training input, the plan compiler, framework programs, and reproduction commands. The supplied examples use one machine. They test local processes and logical CPU devices, with a scope suitable for inspecting schedules and checking numerical results. Throughput, network faults, cross-host deployment, and model accuracy at scale require separate evidence.

Preparation and authorship. Generative AI assisted drafting, typesetting, and preparation of the executable examples. The draft uses an anonymous byline for editorial consideration. Responsibility for submission, attribution, and the final claims belongs to the submitting author.

References

[1] P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, ed., *New Structures for Physics*, pp. 289–355, Springer, 2011. arXiv:0908.3347.

[2] T. Leinster. *Basic bicategories*. 1998. arXiv:math/9810017.

[3] B. Gavranović, P. Lessard, A. J. Dudzik, T. von Glehn, J. G. M. Araújo, and P. Veličković. Position: Categorical deep learning is an algebraic theory of all architectures. *Proc. ICML*, PMLR 235, pp. 15209–15241, 2024. Proceedings.

[4] E. Riehl and D. Verity. *Elements of ∞-Category Theory*. Cambridge Studies in Advanced Mathematics 194, Cambridge University Press, 2022. Book website.

[5] R. Cockett, G. Cruttwell, J. Gallagher, J.-S. P. Lemay, B. MacAdam, G. Plotkin, and D. Pronk. Reverse derivative categories. *Proc. CSL*, LIPIcs 152, art. 18, 2020. DOI.

[6] B. Fong, D. I. Spivak, and R. Tuyéras. Backprop as functor: A compositional perspective on supervised learning. *Proc. LICS*, 2019. arXiv:1711.10455.

[7] G. S. H. Cruttwell, B. Gavranović, N. Ghani, P. Wilson, and F. Zanasi. Categorical foundations of gradient-based learning. *Proc. ESOP*, LNCS 13240, pp. 1–28, 2022. arXiv:2103.01931.

[8] T. Fritz. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Adv. Math.* 370, 107239, 2020. arXiv:1908.07021.

[9] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed., MIT Press, 2018. Publisher.

[10] A. Vaswani et al. Attention is all you need. *Adv. Neural Inf. Process. Syst.* 30, 2017. arXiv:1706.03762.

[11] PyTorch contributors. Scaled dot product attention. API reference.

[12] M. Milakov and N. Gimelshein. *Online normalizer calculation for softmax*. 2018. arXiv:1805.02867.

[13] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Adv. Neural Inf. Process. Syst.* 35, 2022. arXiv:2205.14135.

[14] S. Mac Lane. *Categories for the Working Mathematician*. 2nd ed., Springer, 1998. Publisher.

[15] The Stacks Project Authors. Stacks in groupoids, §8.5, tag 02ZH. The Stacks Project.

[16] J. Lurie. *Higher Topos Theory*. Annals of Mathematics Studies 170, Princeton University Press, 2009. Author's copy.

[17] MPI Forum. *MPI: A Message-Passing Interface Standard*, v4.1, 2023. Reduce and All-reduce.

[18] L. Zheng et al. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *Proc. OSDI*, pp. 559–578, 2022. Proceedings.

[19] PyTorch contributors. Automatic differentiation package. API reference.

[20] PyTorch contributors. Cross entropy. API reference.

[21] PyTorch contributors. Distributed communication package. API reference.

[22] JAX contributors. Distributed arrays and automatic parallelization. Documentation.

[23] JAX contributors. Manual parallelism with shard_map. Documentation.

[24] JAX contributors. Autodiff and sharding. Documentation.

[25] OpenXLA. StableHLO specification. All-reduce operation.

[26] OpenXLA. Shardy sharding representation. Documentation.

[27] PyTorch contributors. Distributed tensor. API reference.

[28] PyTorch contributors. DistributedDataParallel. API reference.

[29] Farama Foundation. Gymnasium environment API. Documentation.

[30] Anonymous. Higher-categorical semantics of machine learning and artificial intelligence. *Higher Categorical Semantics*, 4 November 2026. Journal article.

[31] Anonymous. Compiling tensor graphs and their adjoints for heterogeneous clusters. *Higher Categorical Semantics*, 2026. Journal article.