STATICALLY TYPED TENSOR PROGRAMMING WITH SEMANTIC DIMENSION NAMES

Anonymous authors
Paper under double-blind review

ABSTRACT

In mainstream programming languages, types can state what values mean and constrain the operations permitted on them, making code easier to comprehend and enabling automated verification of logical consistency. However, common numerical and machine learning libraries forgo these benefits by representing all multidimensional data with generic tensor types. We present *DimWit*, a numerical and machine learning library in Scala 3 that encodes tensor dimension names as programmer-defined types, such as `Batch`, allowing tensor types like `Tensor2[Batch, Feature, Float32]` for a batch of feature vectors, and extending the benefits of static typing to expressing, tracking, and verifying dimension names and data types in tensors and across their operations. DimWit leverages JAX as its computational engine, inheriting hardware acceleration, just-in-time compilation, and automatic differentiation. To explore DimWit's semantic tensor types in practice, we implement diverse applications, ranging from linear regression to variational autoencoders, as well as clustering algorithms and simulations. We highlight excerpts of these implementations as examples, showing how semantic tensor types give component interfaces precise meaning and can make programs more comprehensible. To demonstrate the type-safety benefits, we reconstruct twelve selected axis- and shape-related defects from widely used open-source projects, each of which silently produced incorrect results, and show that DimWit rejects them at compile time. To confirm that the approach adds negligible run-time cost at scale, we train a 124M-parameter GPT-2 from scratch with model and training code written in Scala 3 and execution delegated to JAX, reaching 97 % of the throughput of an equivalent JAX implementation in Python. DimWit illustrates how semantic tensor types can make tensor programs more comprehensible and reject axis- and shape-related defects at compile time, without sacrificing run-time performance. Code available at `https://anonymous.4open.science/r/dimwit-9234` and `https://anonymous.4open.science/r/deepwit-DC06/`.

1 INTRODUCTION

Tensors are multi-dimensional arrays that organize data along dimensions. These dimensions carry context-specific meaning. For example, a two-dimensional tensor can represent a batch of feature vectors, a sequence of embeddings, or a linear map. The meaning is in the dimensions: a batch dimension represents independent samples, a feature dimension encodes distinct attributes, and an embedding dimension describes an underlying vector space. These meanings are shared across tensors. A batch of feature vectors and a linear map, for instance, can carry the same feature dimension.

Developers naturally reason about tensors in these terms, yet mainstream machine learning libraries represent semantically different tensors using the same generic `Tensor` type and address dimensions only by position. Even a basic operation like `t.sum(dim=0)` requires deducing what `t` represents to know what is being reduced: are we summing over samples or features? A binary operation like the dot product is only meaningful along a shared dimension, but positional contraction checks sizes, not meaning, and often forces a transpose (`.T`) just to move the axis into place: `features.dot(weights.T)`. Function signatures merely map `Tensor` to `Tensor`, so a function's behavior must be deduced from its name, documentation, implementation, or by trial and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

```python
def cross_entropy(
  logits: Tensor,
  targets: Tensor
) -> Tensor
```

(a) Opaque signature (Python): the targets' encoding and the reduction behavior remain unspecified.

```python
def crossEntropy(
  logits: Tensor2[Batch, Class, Float32],
  targets: Tensor1[Batch, Int32]
): Tensor1[Batch, Float32]
```

(b) Named-dimension signature (DimWit): the types specify discrete targets and an unreduced output.

Figure 1: Named tensor-dimension types put operation semantics into the type signature: the same cross-entropy function without them in Python (a) and with them in DimWit (b). Dimension names such as `Batch` are programmer-defined types, statically tracked and verified throughout the program.

error (fig. 1a). In practice, developers resort to tracking dimension semantics in comments, e.g., `#(B, nh, T, hs)`, but comments are unverified and easily outdated.

To enable developers to express these dimension roles directly in code, we present *DimWit* (short for "Dimension Wit"), a numerical and machine learning library in Scala 3 that delegates value-level execution to JAX. DimWit encodes tensor dimension names as programmer-defined types, such as `Batch`, allowing tensor types like `Tensor2[Batch, Feature, Float32]` for a batch of feature vectors, or `Tensor3[Height, Width, Channel, UInt8]` for an image; the final type parameter tracks the element data type. Dimensions are addressed by name rather than position: the opaque `t.sum(dim=0)` becomes `t.sum(Axis[Batch])`, and the positional contraction `features.dot(weights.T)` becomes `features.dot(Axis[Feature])(weights)`. The transpose disappears because the axis is found by name.

We refer to the dimension names and the data type carried by a tensor's type as *data semantics*. Scala's static type system automatically tracks and verifies these data semantics across all operations, including user-defined functions, so a reader never has to infer what a tensor represents: the data semantics of every tensor are known statically and visible in an IDE without running the code. The data semantics of a function's input and output tensors collectively define *operation semantics*. An `argmax` over a dimension, for instance, yields a tensor without that dimension and with data type `Int32`. In DimWit, the type signature captures exactly that, forming a strict interface between caller and implementation. Figure 1b exemplifies this on a `crossEntropy` signature. It reveals, without exposing its implementation, that the targets are discrete class indices rather than one-hot vectors, and that the returned loss is per-sample rather than reduced. Figure 2 provides further examples of data and operation semantics expressed in DimWit.

We support DimWit's central claim — that semantic tensor types can make tensor programs more comprehensible and improve type safety, without costing run-time performance — in three ways. To explore DimWit's semantic tensor types, we implement a diverse range of applications spanning linear regression, convolutional neural networks, autoencoders, GPT-2, clustering algorithms, and simulations. In section 4, we highlight excerpts from these implementations as examples, showing how DimWit can enforce strict, minimal conceptual boundaries, power a theory-aligned design, enhance local reasoning, and capture domain concepts as distinct dimension names. To demonstrate the type-safety benefits, we reconstruct twelve selected axis- and shape-related defects reported in widely used open-source projects, including PyTorch, Hugging Face Transformers, Keras, and vLLM. Each silently produced incorrect results in the original codebase, whereas in DimWit, each is rejected at compile time (section 5). To confirm that the approach adds negligible run-time cost at scale, we implement a 124M-parameter GPT-2 model entirely in DimWit and train it from scratch, achieving 97 % of throughput of an equivalent JAX implementation in Python (appendix B).

Our main contributions are as follows: (1) We propose a strategy for static tensor programming that encodes tensor dimension names as types to express, track, and verify data and operation semantics; (2) We identify JAX, with its functional design and execution-based tracing, as a natural computational engine for this strategy; (3) We provide a design and implementation of DimWit, a high-performance Scala 3 library that realizes this strategy; (4) We showcase and discuss the advantages of this approach across diverse applications, highlighting how semantic tensor types enforce component boundaries and can aid program comprehension; (5) We demonstrate the type-safety benefits on twelve selected axis- and shape-related defects from open-source projects, each rejected

by DimWit at compile time; (6) We confirm that the approach adds negligible run-time cost at scale by implementing a 124M-parameter GPT-2 model entirely in Scala and training it from scratch, with throughput on par with an equivalent JAX implementation in Python.

```
// --- Computer Vision Tasks ---
def classification(img: Tensor3[Height, Width, Channel, Float32]): Tensor0[Int32]

def segmentation(img: Tensor3[Height, Width, Channel, Float32]): Tensor2[Height, Width, Int32]

// --- Architectural Components ---
def encoder(sample: Tensor1[Feature, Float32]): Tensor1[Latent, Float32]

def crossAttention(
  source: Tensor2[Source, Embedding, Float32],
  target: Tensor2[Target, Embedding, Float32]
): Tensor2[Target, Embedding, Float32]

// --- Mathematics ---
def pca(samples: Tensor2[Sample, Feature, Float32]): Tensor2[Sample, PC, Float32]

def svd(matrix: Tensor2[Row, Col, Float32]): (
  u: Tensor2[Row, Rank, Float32], s: Tensor1[Rank, Float32], v: Tensor2[Rank, Col, Float32]
)
```

(a) **Operation Semantics**. Output types define the characteristics of computer vision tasks. Type signatures create strict boundaries between architectural components to facilitate composition. A change in dimension name marks a change in what the tensor represents, such as `pca` mapping a `Feature` dimension to `PC`.

```
// --- Unscaled Covariance Matrix, Kernel (Gram) Matrix ---
val X: Tensor2[Sample, Feature, Float32] = ??? // Assuming centered data
val cov: Tensor2[Feature, Feature, Float32] = X.dot(Axis[Sample])(X) // X^T X
val gram: Tensor2[Sample, Sample, Float32] = X.dot(Axis[Feature])(X) // X X^T

// --- Unscaled Cross-Attention Scores ---
val Q: Tensor2[Target, Query, Float32] = ???
val K: Tensor2[Source, Key, Float32] = ???
val scores: Tensor2[Target, Source, Float32] = Q.dot(Axis[Query] -> Axis[Key])(K) // Q K^T
```

(b) **Data Semantics.** The compiler tracks declared dimension information across operations, clarifying what tensors represent: A covariance matrix represents pairwise feature similarities (`Feature`×`Feature`). A kernel (Gram) matrix represents pairwise sample similarities (`Sample`×`Sample`). A cross-attention score matrix represents pairwise alignment scores between target and source positions (`Target`×`Source`). The operations themselves are equally explicit: naming the dot-product axis, rather than relying on transposition, states intent directly. `???` marks a dummy implementation.

Figure 2: Examples of operation and data semantics expressed in DimWit.

2 Related Work

The idea of named tensor dimensions has existed for several years and has even been formalized as a named tensor notation for mathematics (Chiang et al., 2023). In Python, one line of libraries, exemplified by xarray (Hoyer & Hamman, 2017; Rush, 2018; Harvard NLP, 2018; PyTorch Contributors, 2019; Domke, 2024), stores dimension names as run-time values in the tensor object. This enables referencing dimensions by name, but the names remain invisible to the type checker. A complementary line, exemplified by jaxtyping (Kidger, 2022; Sinha, 2018), declares dimension names in type annotations. This allows expressing a function's operation semantics, yet the tensors themselves carry no dimension names. The declared names in signatures therefore cannot be verified against arguments, and what is checked is extent consistency at function boundaries: equal names must have equal sizes at run time. Haliax began in the first line (Hall et al., 2023) and later added named tensor types to signatures (Marin Community, 2024). However, the names carried by the tensors are not checked against those declared in the signatures. In einops (Rogozhnikov, 2022), axis names are local to a single call's pattern string and are not carried by the tensor, so they are neither tracked across operations nor checked against signatures.

Beyond Python, Julia libraries (Schouten, 2026; Invenia Technical Computing Corporation, 2019) do check names against signatures: dimension names are part of both the tensor objects and the

Table 1: Representative libraries for tensor dimension names, compared to DimWit. Cells describe the treatment of dimension names; other checks, such as data type, may differ.

Library	Language	Names in tensors	Axis by name	Signature names checked
xarray	Python	run-time	✓	—
jaxtyping	Python	—	✗	partially run-time
Haliax	Python	run-time	✓	partially run-time
DimensionalData	Julia	run-time	✓	run-time
tensor_annotations	Python	partially static	✗	partially static
DimWit	Scala	static	✓	static

Legend: — absent; *run-time* only while the code runs; *partially run-time* only extent consistency checked, not the names themselves; *static* before the code runs; *partially static* as *static*, but only across a fixed set of operations; *axis by name* axes referenced by name, not position.

signatures, and are checked at each call, making them a run-time counterpart to DimWit. These libraries, however, track data semantics only at run time: a reader inspecting the code must still infer them, and the checks cover only the paths a run happens to execute.

Static checking of dimension names has been attempted within Python's type system: tensor_annotations (TensorAnnotations Contributors, 2020) verifies names before execution, but only for a fixed set of operations, since each requires overloads per argument combination (Rahtz, 2023). Moreover, in our testing, higher-order transformations like `grad` and `vmap` erase type information. Data semantics are therefore tracked and verified only in fragments of a program.

In DimWit, dimension names are types within a static type system: automatically tracked across all operations and higher-order transformations like `grad` and `vmap`, and strictly verified at compile time (table 1). Chen (2017) explored this direction in Scala 2, encoding dimension names as types, with type safety as the primary goal. DimWit instead makes program comprehension the primary goal, while retaining type safety, and develops the idea into a full-fledged library, with JAX as a high-performance computational engine. Other machine learning libraries in Scala (e.g., Pauls et al., 2022; Platanios, 2018; Meyer et al., 2024) use generic tensor types without dimension names. Storch (Storch Contributors, 2023) tracks the data type in the tensor type, but not dimension names.

A related approach is to track a tensor's shape within its type, such as `Tensor2[5, 10]` (e.g., Trojahner & Grelck, 2009; Maclaurin et al., 2019; Suwa & Igarashi, 2026). This allows the compiler to detect shape mismatches, again targeting type safety. However, a size at best hints at a dimension's meaning, and two dimensions of equal size are indistinguishable to the compiler, whereas names state the meaning (e.g., `Embedding` vs. `512`) and can stay distinct. Names are also known statically, while sizes often depend on run-time configuration, and type-level size arithmetic requires user-defined functions to prove output sizes follow from input sizes (Suwa & Igarashi, 2026).

3 Design of DimWit

DimWit pairs Scala's type system with JAX as its computational engine. Its type-level implementation encodes dimension names as types in Scala (section 3.1). Its value-level implementation delegates tensor operations to JAX (section 3.2).

3.1 Type-level implementation

DimWit's tensor type abstraction requires a tuple type `T` (all dimension names) and a data type `V` (e.g., `Float32`). A dimension name can be any Scala type such as a `trait` defined by the programmer. The data type is similarly flexible; however, tensor operations are restricted to base data types, e.g., floating-point operations require a floating-point data type. Type aliases exist to define tensors with common ranks, e.g., `type Tensor2[L1, L2, V] = Tensor[(L1, L2), V]`. To reference a dimension by name, DimWit requires a value argument, e.g., `t.sum(Axis[Batch])` rather than the type application `t.sum[Batch]`. Passing a value leaves the remaining type parameters such as the data type `V` to inference and turns axis mistakes into ordinary argument errors at the call site. Figure 3 illustrates these concepts.

The automatic tracking of data semantics across operations relies on type-level functions that run at compile time to infer types from types, for example, `Tensor1[Feature, Float32]` from `Tensor2[Batch, Feature, Float32]` and `Axis[Batch]`. This tracking across basic tensor operations (e.g., reductions, contractions) thus reduces to implementing such type-level functions, for which Scala 3 offers match types (Blanvillain et al., 2022) and implicit resolution. We refer readers interested in the specific type logic to the source code.

```
// Tensor abstraction
class Tensor[T <: Tuple, V]

// Define dimension names as types
trait Batch
trait Feature

def example(in: Tensor2[Batch, Feature, Float32]): Tensor1[Feature, Float32] =
  // Types are automatically tracked. The "out" tensor type is only written for reader clarity.
  val out: Tensor1[Feature, Float32] = in.sum(Axis[Batch])
  out
```

Figure 3: DimWit's tensor type abstraction and basic concepts.

By leveraging general Scala types for both dimension names and the data type, DimWit naturally supports type-based features such as type hierarchies, type aliases, and generic types. Figure 4 exemplifies this, showing a type hierarchy for named tensor dimensions and a generic function. Reusable components in our implementations are likewise generic in their dimension names. The listings in this paper fix concrete names only for readability.

```
// Named dimension hierarchy expressed through Scala type system
trait Sample
trait TrainSample extends Sample
trait TestSample extends Sample
trait NoSample // NoSample does not extend Sample

// Define a function for any subtype of Sample using generic type S
def evalSamples[S <: Sample](samples: Tensor1[S, Float32]): Tensor0[Float32] = ???
val train: Tensor1[TrainSample, Float32] = ???
evalSamples(train) // compiles for TrainSample
val test: Tensor1[TestSample, Float32] = ???
evalSamples(test)  // compiles for TestSample
val no: Tensor1[NoSample, Float32] = ???
evalSamples(no)    // compilation error for NoSample
```

Figure 4: Using Scala types as dimension names enables type-based language features.

3.2 Value-level implementation

A key design decision of DimWit is to use JAX as its computational engine. DimWit maps all basic tensor operations to tensor routines implemented in JAX by calling a Python environment via the ScalaPy language bridge (Laddad & Sen, 2020). This mirrors what Python libraries do themselves, as they also map all basic operations to high-performance routines implemented in low-level languages like C. Just as they wrap native code, DimWit wraps JAX in semantic tensor types, inheriting hardware acceleration, just-in-time compilation, and automatic differentiation.

Crucially, for compiled functions this wrapping costs nothing at execution time. For compilation, JAX traces a function into a computational graph by running it with tracer values in place of concrete tensors. This graph is then lowered to the intermediate representation StableHLO (OpenXLA Project, 2024) and compiled by XLA to efficient native code. The language bridge is thus crossed only during tracing. The compiled code executes without it. We confirm this: with a compiled training step, our GPT-2 training run performs on par with plain JAX (appendix B). This execution-based tracing is why the choice of JAX matters for run-time performance. PyTorch 2's primary compilation path, by contrast, captures graphs by analyzing Python bytecode (Ansel et al., 2024), which DimWit does not produce. A PyTorch-based DimWit would thus be cut off from this path.

Beyond run-time performance, JAX's functional design makes it a natural match: pure operations on immutable tensors, tensor trees, explicit randomness, and transformations such as `vmap` and `grad` carry over to DimWit while gaining semantic tensor types (appendix C). Python serves DimWit only as an execution vehicle, while the type level requires capabilities Python lacks. We selected Scala 3 (Odersky et al., 2021) for its type-level functions, its static type system, and its brace-less syntax, which stays close to Python's.

4 Semantic tensor types for comprehensible tensor programs

To show how semantic tensor types can make tensor programs more comprehensible, we implement diverse applications, from linear regression to GPT-2, clustering, and simulations. This section presents excerpts from these implementations[1], each illustrating an advantage: enforcing strict, minimal conceptual boundaries (section 4.1), powering a theory-aligned design (section 4.2), enhancing local reasoning (section 4.3), and capturing domain concepts as distinct dimension names (section 4.4). More examples are in appendix D.

4.1 Affine transformation: Strict, minimal conceptual scope

Consider an affine transformation, $y = xA + b$, which maps an input vector $x \in \mathbb{R}^n$ to an output vector $y \in \mathbb{R}^m$. Figure 5 (left) illustrates how we can express this concept in DimWit. The `affine` function strictly requires an `In` vector `x`, an `In×Out` projection matrix `A`, and an `Out` bias vector `b`. This signature is also structurally minimal: it takes a single vector, not a batch, and applying it per sample must use `vmap`. Monolithic tensor types lack this strictness and encourage permissive implementations that exceed the concept's scope, such as directly applying `affine` to a batched input `X` (fig. 5, right). Such overly general implementations are common; for example, standard linear layers typically accept arbitrary high-dimensional tensors by implicitly operating on the last dimension. Strict type signatures eliminate these opaque positional assumptions. A minimal scope keeps the concept's core logic free of other concerns and clarifies the call site: `vmap` applies `affine` to each sample independently, whereas `affine(X, A, b)` leaves open whether samples interact.

```
def affine( // Strict conceptual scope
  x: Tensor1[In, Float32],
  A: Tensor2[In, Out, Float32],
  b: Tensor1[Out, Float32]
): Tensor1[Out, Float32] =
  x.dot(Axis[In])(A) + b

// Applied to each sample x independently
val Y: Tensor2[Batch, Out, Float32] =
  X.vmap(Axis[Batch])(x => affine(x, A, b))
```

```
def affine( # Permissive conceptual scope
  x: Tensor,
  A: Tensor,
  b: Tensor
) -> Tensor:
  return x @ A + b

# Unclear whether samples in X interact
Y = affine(X, A, b)
```

Figure 5: Affine function applied to batch of inputs `X` in DimWit (left) and Python (right).

4.2 Linear regression: Semantic types power theory-aligned design

In our implementations, we follow a theory-aligned design, making key concepts like the cost function, parameters, gradients and all their dependencies explicit. For example, we bundle a model's parameters into a `Params` object and make them an explicit argument to the model. This mirrors standard machine learning theory, where a model f is expressed as a parameterized function $f(x; \theta)$, and decouples the model's architecture from parameter lifecycle management. Consequently, parameter initialization is a function returning a `Params` object, checkpointing is serializing these objects, and parameter updating is an optimizer mapping current to new parameters. Furthermore, explicit parameters allow expressing a theory-aligned cost function $\mathcal{L}(\theta; X, y)$ that maps the parameters θ to

[1]Listings are lightly simplified for presentation, e.g., omitting `derives Label`, using concrete dimension names where the implementation is generic (cf. fig. 4), and showing duplicate dimension names unlifted (appendix C.6); exact code is in the repository.

a scalar cost given a dataset X, y. Figure 6 illustrates this for a linear regression model. This functional style is established in the JAX ecosystem (Heek et al., 2024; Kidger & Garcia, 2021); DimWit adopts it and adds semantic types to the signatures. Semantic types power this theory-aligned design by mirroring the theoretical objects and verifying their data flow and composition.

```
case class Params(A: Tensor1[In, Float32], b: Tensor0[Float32])

// Explicit parameters θ, matching the theoretical formulation y = f(x; θ)
case class LinearRegression(params: Params):
  def apply(x: Tensor1[In, Float32]): Tensor0[Float32] =
    params.A.dot(Axis[In])(x) + params.b

// Parameter initialization decoupled from model
LinearRegression(params=initParameters(Axis[In] -> 10, Random.Key.fromTime()))

// Checkpointing decoupled from model
LinearRegression(params=loadCheckpoint[Params](path))

// Parameter update decoupled from model with theory-aligned cost, differentiation, gradients
def costFnFor(                                          // ℒ(θ; X, y)
  X: Tensor2[Sample, In, Float32], y: Tensor1[Sample, Float32]
)(params: Params): Tensor0[Float32] =
  val model = LinearRegression(params)
  val yHat: Tensor1[Sample, Float32] = X.vmap(Axis[Sample])(model)
  (y - yHat).pow(2).mean

// Types mirror the theoretical objects (θ → ℝ for costFn) and verify their composition
val costFn: Params => Tensor0[Float32] = costFnFor(X, y)  // ℒ_{X,y}(θ) = ℒ(θ; X, y)
val dCost: Params => Params = grad(costFn)                // g(θ) = ∇_θ ℒ_{X,y}(θ)
val grads: Params = dCost(params)                         // g_t = g(θ_t)
val nextParams: Params = params -- alpha * grads          // θ_{t+1} = θ_t − α g_t
```

Figure 6: Theory-aligned linear regression in DimWit. Semantic types mirror the theoretical objects.

4.3 GPT-2: Local reasoning and understanding within a code block

Figure 7 illustrates the `logits` function of our GPT-2 implementation with dependencies as abstract tensor functions. Each dependency's role is clear from operation semantics alone. Data semantics and referencing dimensions by name reveal that `embedder` and `outputHead` are applied at each context position, whereas the `transformer` operates across the context and embeddings. The `logits` function is thus understandable locally, without reading any dependency's implementation.

```
trait GPT:
  // Dependency abstractions: role understandable from signature alone
  def embedder(token: Tensor0[Int32]): Tensor1[Embedding, Float32]
  def transformer(context: Tensor2[Context, Embedding, Float32]): Tensor2[Context, Embedding, Float32]
  def outputHead(vector: Tensor1[Embedding, Float32]): Tensor1[Vocab, Float32]

  // Implementation: code understandable locally, without dependency implementation
  def logits(tokens: Tensor1[Context, Int32]): Tensor2[Context, Vocab, Float32] =
    val embeddings: Tensor2[Context, Embedding, Float32] = tokens.vmap(Axis[Context])(embedder)
    val mixed: Tensor2[Context, Embedding, Float32] = transformer(embeddings)
    mixed.vmap(Axis[Context])(outputHead)
```

Figure 7: GPT `logits` function with abstract dependencies.

4.4 N-body simulation: Domain concepts as distinct dimension names

Our n-body simulation treats `Position`, `Velocity`, and `Acceleration` as strictly distinct types, although all three are vectors in the same space (fig. 8). Operations across these types then require explicit relabeling: the time-scaled `Velocity` must be relabeled to `Position` before it can be added to the current position, reflecting that integrating velocity over time yields a displacement. Likewise, time-scaled `Acceleration` is relabeled to `Velocity`, and the direction vectors toward a body are relabeled from `Position` to `Acceleration`, clarifying their role. We found that this strictness pays off in comprehension: each relabeling marks a shift between kinematic concepts that

untyped code performs silently. A coarser design would also be sound: a single `Spatial` type for all three. However, this finer vocabulary makes the kinematics explicit in the code. DimWit makes such vocabularies part of the type system, at whatever granularity the use case demands.

```
trait Body
trait Position  // Domain concepts as distinct types: position, velocity, acceleration
trait Velocity
trait Acceleration

// -- Single body update --
case class BodyState(
  val mass: Tensor0[Float32],
  val position: Tensor1[Position, Float32],
  val velocity: Tensor1[Velocity, Float32]
)
def bodyStepFn(
  masses: Tensor1[Body, Float32],
  positions: Tensor2[Body, Position, Float32]
)(state: BodyState): BodyState =
  val position: Tensor1[Position, Float32] = state.position
  val velocity: Tensor1[Velocity, Float32] = state.velocity
  val acceleration: Tensor1[Acceleration, Float32] =
    val relativeOffsets = positions -! state.position
    val accelerationDir = (relativeOffsets /! (relativeOffsets.norm(Axis[Position]) +! eps))
      .relabel(Axis[Position] -> Axis[Acceleration])  // Clarify intent of direction vectors
    val relativeSquaredDistances = relativeOffsets.pow(2).sum(Axis[Position]) +! eps
    val accelerationMagnitudes = masses / relativeSquaredDistances
    (accelerationDir *! accelerationMagnitudes).sum(Axis[Body]) *! G
  val deltaP = (velocity *! dt).relabelTo(Axis[Position])  // Integration yields displacement
  val deltaV = (acceleration *! dt).relabelTo(Axis[Velocity])  // Integration yields velocity
  BodyState(state.mass, position + deltaP, velocity + deltaV)

// -- N-body update --
case class SystemState(
  val masses: Tensor1[Body, Float32],
  val positions: Tensor2[Body, Position, Float32],
  val velocities: Tensor2[Body, Velocity, Float32]
)
def systemStep(systemState: SystemState): SystemState =
  val (ms, ps, vs) = (systemState.masses, systemState.positions, systemState.velocities)
  val (newMs, newPs, newVs) = zipvmap(Axis[Body])((ms, ps, vs)): (m, p, v) =>
    val newBodyState = bodyStepFn(ms, ps)(BodyState(m, p, v))
    (newBodyState.mass, newBodyState.position, newBodyState.velocity)
  SystemState(newMs, newPs, newVs)
```

Figure 8: N-body simulation with distinct types for position, velocity, and acceleration. Operators suffixed with `!` broadcast their operands; others require matching dimension names (appendix C.3).

5 Rejecting axis- and shape-related defects at compile time

We show twelve axis- and shape-related defects from the issue trackers and pull requests of widely used open-source projects, including PyTorch, Hugging Face Transformers, Keras, and vLLM, that DimWit rejects at compile time. They come from a single LLM prompt asking for reported defects that semantic tensor types would reject. The cases are therefore selected to show the type-safety benefit on defects that occur in practice, not sampled from shape defects in general, and we make no claim about what share of such defects DimWit rejects. The prompt yielded fifteen candidates. We excluded two that were not upstream defect reports, and we report one separately (case 04), as it lies outside what dimension types check. Appendix A details the selection protocol and each case.

Most defects share one pattern: the code confuses which axis is which. A height is read where a width is expected, a batched model is fed an unbatched sequence, or image axes arrive in an unexpected order. Such mistakes can stay silent because reductions, reshapes, and broadcasts ignore what the axes mean. We reconstruct each defect in plain NumPy/PyTorch and in DimWit. In Python, all cases run silently with incorrect results. In DimWit, the twelve do not compile: DimWit tracks data semantics and verifies them against each operation's expected inputs. Names also enable stricter broadcasting (Hall et al., 2023), extending an operand only along dimensions it lacks (appendix C.3).

6 DISCUSSION

A vocabulary of dimension names can draw the conceptual distinctions the use case requires. From this vocabulary, data semantics state what a tensor is and operation semantics what a function does. Conventionally, this knowledge lives in comments, documentation, and developers' heads, where it is unverified, easily outdated, or never written down. Semantic tensor types express this knowledge within code, where it is shared, verified, and cannot go stale. Expressed this way, this knowledge can help developers understand the code and the domain, directly within executable tensor programs. In the terms of Naur (1985), a program is not primarily its text but the theory its developers hold of it; the dimension vocabulary externalizes part of that theory into the program, where the compiler keeps it consistent with the rest of the code.

6.1 IMPACT ON AGENTIC CODING

Whether the comprehension advantages carry into agentic coding depends on what code becomes. If code remains a shared medium between humans and agents, the case is direct: the developer's role shifts from writing code toward directing agents and reviewing their output (Dhanorkar et al., 2026). Reviewing is a comprehension task, so the advantages illustrated in section 4 carry over. The dimension vocabulary further serves as a shared language: developers instruct agents in domain terms and verify output in the same terms. It also structures the reverse direction: developers can ask the agent what a concept covers, whether it could be split, or why it is absent from a signature. If code instead becomes a medium among agents only, the comprehension advantages lose their human audience, and whether semantic tensor types help AI agents remains to be shown.

The type-safety benefits of section 5 sharpen the developer's oversight role. Type signatures define strict boundaries, and the compiler enforces them. Verified boundaries shrink what can go wrong and what must be checked, for agent-produced code no less than for human-written code. Beyond oversight, the compiler is available to the agent itself, providing immediate, machine-readable feedback an agentic workflow can incorporate.

6.2 LIMITATIONS AND FUTURE WORK

Adopting DimWit requires learning Scala and its toolchain, a real barrier for the Python-native machine learning community. A further challenge is the maturity of Scala's data science ecosystem. Python provides established, end-to-end workflows where libraries for plotting, modeling, and logging seamlessly interoperate. Bridging these libraries into Scala via ScalaPy is a viable workaround (appendix E), yet a natively integrated, statically typed ecosystem remains preferable. DimWit also relies on a development environment encompassing the JVM, ScalaPy, Python, and JAX; deployment overhead is mitigated through computational-graph export, enabling language-agnostic inference. In eager execution, every operation crosses the language bridge, adding overhead relative to eager JAX (appendix B); compiled functions avoid this cost. Finally, our evidence is illustrative: a user study could test whether the comprehension advantages of section 4 hold for developers in practice, and whether semantic tensor types sharpen their understanding of the underlying domain as well. A systematic corpus of axis- and shape-related defects could determine what share of them the type safety shown in section 5 rejects.

7 CONCLUSION

We introduced DimWit, a library that encodes dimension names and the data type directly into the tensor type, tracking data semantics and capturing operation semantics in strict type signatures. Across diverse applications, these types enforced conceptual boundaries, powered a theory-aligned design, enhanced local reasoning, and captured domain concepts as distinct dimension names. Reconstructed in DimWit, twelve selected axis- and shape-related defects from open-source projects were rejected at compile time. A 124M-parameter GPT-2, implemented entirely in Scala and trained from scratch, performed on par with plain JAX. Without sacrificing run-time performance, DimWit extends the benefits of static typing to tensor programming and can make tensor programs more comprehensible. What tensors mean becomes part of the program itself.

AI USE STATEMENT

In this work, we used generative AI tools as supportive development assistants, for assistance during the implementation of the library's type logic, for surfacing candidate related work and candidate defects, for drafting the initial reconstructions in the defect study (appendix A), and for editing and refining the manuscript's text. We have not used generative AI tools for formulating our core research ideas or designing DimWit. All AI-assisted work was rigorously reviewed: code suggestions were manually tested, refined, and integrated by the authors. Each related-work candidate was read and assessed by the authors. Each defect was manually verified against its upstream issue and its reconstructions revised for correctness and cross-version consistency. All text revisions were verified to ensure accuracy. We take full responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ACKNOWLEDGMENTS

This section has been removed for the double-blind review process.

REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM international conference on architectural support for programming languages and operating systems, volume 2*, pp. 929–947, 2024.

Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proceedings of the ACM on Programming Languages*, 6 (POPL):1–24, 2022.

Edwin Brady. *Type-Driven Development with Idris*. Manning Publications, Shelter Island, NY, 2017. ISBN 9781617293023.

Tongfei Chen. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pp. 45–50, 2017.

David Chiang, Alexander M. Rush, and Boaz Barak. Named tensor notation. *Transactions on Machine Learning Research*, 2023. URL https://openreview.net/forum?id=hVT7SHlilx.

Shipi Dhanorkar, Samir Passi, and Mihaela Vorvoreanu. Human oversight of agentic systems in practice: Examining the oversight work, challenges, and heuristics of developers using software agents. In *The 2026 ACM Conference on Fairness, Accountability, and Transparency*, pp. 6438–6465, 2026.

Justin Domke. Numbat: NumPy+JAX with named axes and an uncompromising attitude, 2024. URL https://github.com/justindomke/numbat.

David Hall, Ivan Zhou, and Percy Liang. Levanter — legible, scalable, reproducible foundation models with JAX. Stanford CRFM Blog, June 2023. URL https://crfm.stanford.edu/2023/06/16/levanter-1_0-release.html. Accessed: 2026-07-01.

Harvard NLP. NamedTensor. https://github.com/harvardnlp/NamedTensor, 2018.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2024. URL http://github.com/google/flax.

Stephan Hoyer and Joseph J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017. doi: 10.5334/jors.148. URL https://doi.org/10.5334/jors.148.

Invenia Technical Computing Corporation. NamedDims.jl: For working with dimensions of arrays by name, 2019. URL https://github.com/invenia/NamedDims.jl.

Patrick Kidger. jaxtyping: Type annotations and runtime checking for shape and dtype of arrays, 2022. URL https://github.com/patrick-kidger/jaxtyping.

Patrick Kidger and Cristian Garcia. Equinox: neural networks in JAX via callable PyTrees and filtered transformations. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021.

Shadaj Laddad and Koushik Sen. ScalaPy: seamless Python interoperability for cross-platform Scala programs. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala*, pp. 2–13, 2020.

Dougal Maclaurin, Alexey Radul, Matthew J Johnson, and Dimitrios Vytiniotis. Dex: array programming with typed indices. In *Program Transformations for ML Workshop at NeurIPS 2019*, 2019.

Marin Community. Haliax: Neural networks with named axes, 2024. URL https://github.com/marin-community/haliax.

Benjamin Meyer, Thilo Stadelmann, and Marcel Lüthi. ScalaGrad: a statically typed automatic differentiation library for safer data science. In *2024 11th IEEE Swiss Conference on Data Science (SDS)*, pp. 229–232. IEEE, 2024.

Peter Naur. Programming as theory building. *Microprocessing and microprogramming*, 15(5):253–261, 1985.

Martin Odersky, Lex Spoon, Bill Venners, and Frank Sommers. *Programming in Scala*. Artima Inc, 5th edition, 2021.

OpenXLA Project. StableHLO: Backward compatible ML compute operation set. https://openxla.org/stablehlo, 2024. URL https://github.com/openxla/stablehlo.

Adam Pauls, David Hall, Theo Lanman, Alex Kyte, Hao Fang, Anthony Platanios, and Dmitrij Peters. scala_torch: Scala bindings for LibTorch. https://github.com/microsoft/scala_torch, 2022.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale. In *Advances in Neural Information Processing Systems*, volume 37, pp. 30811–30849. Curran Associates, Inc., 2024. doi: 10.52202/079017-0970.

Emmanouil Antonios Platanios. TensorFlow Scala. https://github.com/eaplatanios/tensorflow_scala, 2018.

PyTorch Contributors. Named tensors. https://pytorch.org/docs/stable/named_tensor.html, 2019. Prototype feature, introduced in PyTorch 1.3.

Matthew Rahtz. Why TensorAnnotations is being deprecated. https://docs.google.com/document/d/1AAP-wq06j1TQwJPtrlky4lfyPHyl7-itgN5S47oZO98, June 2023. Accessed: 2026-09-13. Archived at the Internet Archive.

Alex Rogozhnikov. Einops: Clear and reliable tensor manipulations with einstein-like notation. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=oapKSVM2bcj.

Alexander Rush. Tensor considered harmful. https://nlp.seas.harvard.edu/NamedTensor, 2018. Harvard NLP Blog.

John K Salmon, Mark A Moraes, Ron O Dror, and David E Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pp. 1–12, 2011.

Rafael Schouten. DimensionalData.jl: Datasets with named dimensions, 2026. URL https://github.com/rafaqz/DimensionalData.jl.

Nishant Sinha. tsalib: Tensor shape annotation library. https://github.com/ofnote/tsalib, 2018.

Storch Contributors. Storch: GPU accelerated deep learning and numeric computing for Scala 3. https://github.com/bytedeco/storch, 2023.

Takashi Suwa and Atsushi Igarashi. Compile-time tensor shape checking via staged shape-dependent types. In *40th European Conference on Object-Oriented Programming (ECOOP 2026)*, volume 372 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 28:1–28:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi: 10.4230/LIPIcs.ECOOP.2026.28.

TensorAnnotations Contributors. TensorAnnotations: Annotating tensor shapes using Python types, 2020. URL https://github.com/google-deepmind/tensor_annotations. DeepMind. Accessed: 2026-09-13.

Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009.

Dangwei Wu, Beijun Shen, and Yuting Chen. An empirical study on tensor shape faults in deep learning systems. *arXiv preprint arXiv:2106.02887*, 2021.

A RECONSTRUCTED AXIS- AND SHAPE-RELATED DEFECTS FROM PUBLIC ISSUE TRACKERS

This appendix documents the twelve defects of section 5 and how they were selected. The candidates come from a single prompt to Claude Opus 5 in an IDE session that had the DimWit and DeepWit repositories and prior project context available. The prompt asked for axis- and shape-related defects from established repositories that DimWit's semantic tensor types would catch at compile time. The candidates are therefore specifically selected to show type-safety benefits, not sampled from shape defects in general. The prompt yielded fifteen candidates. We verified each candidate manually against its upstream source and excluded two: a feature extension rather than a defect report (whisper#839), and a dataset rather than an upstream report (Wu et al., 2021). Table A.1 lists all thirteen remaining defects with their outcomes: twelve are rejected at compile time, and one (case 04) lies outside what dimension types check and is included to mark that boundary. The remainder of this section details each case. The full reconstructions are available at `https://anonymous.4open.science/r/dimwit-evidence-F7F5/`.

Table A.1: Thirteen axis- and shape-related defects from open-source issue trackers and pull requests, each reconstructed in plain NumPy/PyTorch and in DimWit. *compile-time*: compiler rejects program. *missed*: runs with incorrect results. Case 04 lies outside what dimension types check.

#	Defect	Plain	**DimWit**
01	attention mask on the query axis	missed	**compile-time**
02	MSE over $[n, 1]$ vs $[n]$	missed	**compile-time**
03	`cross` product on the first length-3 axis	missed	**compile-time**
05	NCHW height/width swapped	missed	**compile-time**
06	softmax reduction dropped its axis	missed	**compile-time**
07	stacked dim name collided	missed	**compile-time**
08	reshape with pre-transpose sizes	missed	**compile-time**
09	single G-vector broadcast	missed	**compile-time**
10	model called without a batch axis	missed	**compile-time**
11	NCHW frames into an NHWC pipeline	missed	**compile-time**
12	batched contraction over the wrong axis	missed	**compile-time**
13	(w, h) extents into an (h, w) API	missed	**compile-time**
04	displacement normalized by the wrong side	missed	missed

A.1 CASE 01 — ATTENTION MASK ON THE QUERY AXIS

Pix2Struct adds a padding mask to its attention scores. NumPy requires the mask to be padded with singleton axes first, to match the rank. The code placed them wrongly, so broadcasting applied the mask across queries rather than keys (transformers#23974). DimWit broadcasts by axis name, extending an operand only along the axes it lacks. The padding step does not exist, so it cannot go wrong.

A.2 CASE 02 — MSE OVER [N,1] VS [N]

A model returning `[n, 1]` predictions was compared against `[n]` targets. NumPy aligns shapes from the right and stretches any axis of length 1, so the subtraction became an `[n, n]` matrix of every pairwise residual. Its mean is a plausible loss, so the model trained on the wrong objective (pytorch#16045). DimWit broadcasts only by adding an axis an operand does not have, and never stretches one it does, so `[n, 1]` against `[n]` cannot become `[n, n]`. The mistake is caught earlier still, by the signature: a loss scoped to `Tensor1[Batch]` on both sides refuses a prediction carrying an extra axis before any arithmetic happens.

A.3 CASE 03 — CROSS PRODUCT ON THE FIRST LENGTH-3 AXIS

`torch.cross` defaults to the first axis of length 3, while NumPy, JAX and TensorFlow use the last. Keras forwarded that default through to its torch backend, so the same call gave different results on

different backends (keras#23219). Following the minimal conceptual scope of section 4.1, DimWit defines `cross` on two vectors sharing one axis, so there is no axis argument to default. Higher-rank inputs are handled by `vmap`.

A.4 Case 04 — Displacement normalized by the wrong side

For an elastic-transform augmentation the two divisors of height and width were swapped (vision#9299). DimWit does not catch this at compile time as height and width are integers, not named tensors. However, by naming the tensor shape by `shape(Axis[Height])` rather than positional `shape[0]` the error becomes easier to spot.

```
// compiles despite semantic error, dx is normalized by the height, dy by the width
val dx = noise.sample(kx) *! alpha._1 /! shape(Axis[Height])
val dy = noise.sample(ky) *! alpha._2 /! shape(Axis[Width])
```

A.5 Case 05 — NCHW height/width swapped

ONNX model inputs are NCHW, so the height comes before the width. A YOLO example read the two the wrong way round, and every letterboxed image and rescaled box came out skewed by the aspect ratio (ultralytics#23126). In DimWit, the dimension names are assigned once for the data and once for the model; these are the only steps the compiler cannot check. From there on, every match between data and model is checked: swapping height and width results in a `Tensor3[Width, Height, Channel]` where a `Tensor3[Height, Width, Channel]` is declared, which are different types, so a compile error. The model's interface is in the types, not in a comment.

A.6 Case 06 — Softmax reduction dropped its axis

Triton reduced dimensions within its `softmax` method, leading to wrong broadcasts and results (triton#11406). DimWit defines `softmax` on a vector, where a reduced value has only one axis to return to. Furthermore, broadcasting by name expands the missing axis correctly.

A.7 Case 07 — Stacked dim name collided

ArviZ names dimensions with an internal `sample` name potentially colliding with existing names (arviz#1647). DimWit's dimension names are Scala types, so two libraries can both use the name `Sample` without colliding: they are distinct types, not equal strings.

A.8 Case 08 — Reshape with pre-transpose sizes

PyMC moves two axes to the front and then reshapes to flatten, yet the reshape read the target sizes from the original array, leading to a scrambled result (pymc#7178). DimWit has no `reshape` precisely because it mixes across concepts in this error-prone way. In DimWit we use `flatten` for this.

```
// flatten records which axes were fused, and in what order using type-level |*| as the label.
val t2: Tensor2[Draw |*| Chain, Team, Float32] =
  t1.transpose((Axis[Draw], Axis[Chain], Axis[Team])).flatten((Axis[Draw], Axis[Chain]))
```

A.9 Case 09 — Single G-vector broadcast

A PySCF routine expects a grid of vectors, but its signature accepts any tensor. Passing a vector led to internal broadcasting and a wrong result (pyscf#2961). DimWit puts the grid-of-vectors expectation in the signature as `Tensor2`, so passing a single vector, a `Tensor1`, does not compile.

A.10 Case 10 — Model called without a batch axis

The text-generation script called a model written for batched input with a single unbatched sequence. Every operation inside accepted it, so a step meant to average over the batch averaged over the sequence instead (lit-llama#166). DimWit enforces a clear named boundary, catching wrong data semantics.

A.11 Case 11 — NCHW frames into an NHWC pipeline

Two video decoders behind one interface disagreed about where the channel axis goes. Swapping one for the other changed the meaning of every axis index, with no error (vllm#51076). In DimWit, the two layouts are distinct types, e.g., `Tensor4[Frame, Channel, Height, Width]` and `Tensor4[Frame, Height, Width, Channel]`. The shared interface declares one of them, so a decoder returning the other does not compile. The types admit only valid compositions of components.

A.12 Case 12 — Batched contraction over the wrong axis

Due to implicit size broadcasting a contraction was silently applied to different axes (mlx#4125). DimWit names the contracted axis, so no silent behavior change is possible.

A.13 Case 13 — (W,H) extents into an (H,W) API

`cv2.resize` takes its target size as width then height, while the array it returns is height then width. Gymnasium declared its output shape in the first order and produced the second, which stayed invisible for as long as the size was square (Gymnasium#1312). A resize keeps the axes it was given rather than renaming them. In DimWit, the output's extents carry the input's labels, and the input pins them. Passing them in the wrong order fails even when the function is fully generic.

```
def resize[H: Label, W: Label](
   frame: Tensor2[H, W, Float32],
   width: AxisExtent[W], // W and H are bound to frame's dimension labels
   height: AxisExtent[H]
): Tensor2[H, W, Float32]
```

B GPT-2: Run-time performance sanity check

To confirm that DimWit's abstractions do not introduce meaningful run-time overhead, we ran a performance sanity check by training a 124M-parameter GPT-2 model on a DGX Spark node. We compared our DimWit implementation's throughput against a plain JAX implementation in Python, and included nanoGPT[2] (PyTorch) as a reference to confirm that our numbers were within the expected performance range for this hardware. All setups followed the standard GPT-2 (124M) training configuration of nanoGPT with identical hyperparameters: GPT-2 BPE tokenizer, context length 1024, an effective batch size of $524,288$ tokens via gradient accumulation, AdamW with linear warm-up and cosine decay, mixed-precision training, and flash attention. Throughput was measured over 100 iterations after a standard warm-up phase, and JIT overhead was averaged across three compilations. Our DimWit implementation maintained its throughput across $20,000$ iterations ($\sim$3 days of training) and converged with a final validation loss of 3.35 on FineWeb-Edu (Penedo et al., 2024). Sampled completions from the trained model were coherent.

As shown in table B.1, although the DimWit and JAX implementations are distinct, they perform on par. The negligible difference in throughput and JIT overhead demonstrates that the DimWit abstraction layer introduces no meaningful run-time overhead.

The slight difference between DimWit and JAX is attributed to gradient accumulation happening eagerly, i.e., not compiled. For eager execution, the language bridge is active. We measured the eager execution overhead for a six-layer fully-connected autoencoder and a three-layer convolutional

[2] https://github.com/karpathy/nanoGPT

	tokens/s	JIT overhead (s)
JAX	$44,134 \pm 117$	188.7 ± 1.0
DimWit	$42,956 \pm 417$	181.2 ± 0.3
nanoGPT	$51,049 \pm 97$	—

Table B.1: Performance comparison on GPT-2 (124M) on a DGX Spark node. Throughput over 100 iterations after warm-up; JIT overhead averaged over three compilations. nanoGPT (PyTorch) included as a hardware sanity reference.

classifier, both trained on MNIST at batch size 512. On a single NVIDIA RTX 3060, they run $2.4\times$ and $1.4\times$ slower than equivalent eager JAX implementations. On an Apple M1 Pro CPU the same two models run $4.7\times$ and $1.5\times$ slower than equivalent eager JAX implementations. We consider this overhead acceptable: eager execution serves development and debugging, where it remains interactive, while training at scale is JIT-compiled and thus unaffected.

C DimWit: Supplementary design details

C.1 Explicit stochasticity (and determinism)

DimWit adopts JAX's approach by restricting all stochasticity to a single argument of type `Key`. This key serves as the only source of randomness (Salmon et al., 2011). The absence of this argument expresses determinism. Consequently, randomness becomes an explicit dependency and thus can be managed, e.g., reproducing results requires an identical key.

```
def createRandomTensor(key: Key): Tensor2[In, Out,
    Float32] = ??? // Stochasticity indicated by key

// We can control randomness by using the same key...
val key = Key(42)
val t1 = createRandomTensor(key)
val t2 = createRandomTensor(key) // t1 and t2 are identical

// ... or using different keys (by splitting a key)
val (t3Key, t4Key) = key.split(2)
val t3 = createRandomTensor(t3Key)
val t4 = createRandomTensor(t4Key) // t3 and t4 are different
```

Figure C.1: Explicit stochasticity: a parameter of type `Key` signals stochastic capabilities.

C.2 Explicit mapping across dimensions

DimWit adopts JAX's strategy for mapping functions across higher-dimensional tensors using a vectorized map (`vmap`). This allows for conceptually minimal type signatures, e.g., a neural network requiring a single sample rather than a batched input. The XLA compiler then seamlessly transforms this into efficient execution (fig. C.2).

```
def classificationModel(vector: Tensor1[Feature, Float32]): Tensor0[Int32]

val dataBatch: Tensor2[Batch, Feature, Float32] = ???
val output: Tensor1[Batch, Int32] = dataBatch.vmap(Axis[Batch])(classificationModel)
```

Figure C.2: Minimal structure in functions, applied to higher-dimensional tensors with `vmap`.

C.3 Explicit, name-based broadcasting

Named tensor dimensions allow for a smarter broadcasting strategy, where broadcasting is only extending the shape of a lower-rank tensor to match a higher-rank tensor based on the missing di-

mension names (Hall et al., 2023). This permits more shape broadcasting, for example, both tensors `Tensor1[A, Float32]` and `Tensor1[B, Float32]` can be broadcast to match a tensor `Tensor2[A, B, Float32]`—both times broadcasting the lower-rank tensor over the missing dimension. Yet, DimWit rejects size broadcasting, meaning an existing dimension's size is never altered as this leads to unexpected behavior (Hall et al., 2023). Furthermore, DimWit requires a programmer to make broadcasting explicit within code to clarify intent using a `broadcastTo` method or with broadcasting variants of operations, marked by `!`, e.g., `+!` or `where_!`.

C.4 Explicit data-type casting

In DimWit, operations require a tensor to have a valid data type. For example, `sum` is not defined for boolean tensors as the definition of summing boolean values is unclear. Similarly, division and the `mean` operation are only available for floating-point tensors, as the integers are not closed under division. Thus applying such operations requires an explicit cast to a more general data type. Furthermore, binary operations require both tensors to have the same data type. For example, adding an `Int16` tensor with an `Int32` tensor requires an explicit cast of one tensor to fit the precision of the other. Figure C.3 illustrates that we can implement functions that generalize across precisions using an evidence parameter for the underlying kind of data (i.e. `IsFloating`).

```
def standardize[V: IsFloating](vector: Tensor1[Feature,
    V]) = (vector - vector.mean) /! vector.std
```

Figure C.3: Implementation generalizing for different floating-point precisions.

C.5 Extensions to tensor trees

Tensors often appear grouped within tree structures (e.g., lists, tuples, custom objects). DimWit automatically derives a `TensorTree` type class to support type-checked operations like `map` over these hierarchical structures. Additionally, it overloads arithmetic operations to allow combining trees, e.g., adding two trees together. To distinguish these tree-level operations from standard tensor operations, DimWit utilizes double-character operator syntax, e.g., `++` instead of `+`. Figure C.4 shows a minimal example.

```
case class Params(w: Tensor2[In, Out, Float32], b: Tensor1[Out, Float32])
val p1: Params = ???
val p2: Params = ???
p1 ++ p2 // compiles, adds each tensor within the tensor tree of Params
```

Figure C.4: Arithmetic operations across tensor trees like `Params` objects.

C.6 Duplicate dimension names

Some operations yield a tensor with the same dimension name twice, e.g., the covariance matrix `X.dot(Axis[Sample])(X)` in fig. 2b. To keep both axes addressable by name, DimWit lifts the second occurrence to `Prime[Feature]`: the inferred type is `Tensor2[Feature, Prime[Feature], Float32]`, and `cov.sum(Axis[Prime[Feature]])` is unambiguous.

C.7 Named shapes

In DimWit, a tensor's shape carries its dimension names: `Shape2[Batch, Feature]`. Shape extents are read by name rather than by position: `t.shape(Axis[Batch])` instead of `t.shape[0]`. The extent of a single axis keeps its name as well: `AxisExtent[Batch]` is a size tagged with the dimension it belongs to.

C.8 Explicit sharding across devices

DimWit lifts parts of JAX's sharding mechanism to support data parallelism for multi-device training. Sharding follows the same named discipline as tensor dimensions with a mesh carrying axis names (e.g., `X`). Crucially, the sharding of a tensor is reflected in its type, e.g., sharding the `Batch` dimension across mesh axis `X` yields `Tensor2[Batch |@| X, Feature, Float32]` (fig. C.5). A sharded and an unsharded tensor are therefore distinct types, so accidentally combining them is a compile error rather than an implicit re-layout. As a sanity check at scale, sharding the batch dimension of our GPT-2 training (appendix B) across four NVIDIA A100 (40 GB) GPUs sustains a throughput of over 130,000 tokens/s. Sharding model parameters, and with it further parallelism strategies, is future work.

```
val mesh: Mesh1[X] = Mesh(MeshAxis[X] -> 4) // 4 devices along mesh axis X

val x: Tensor2[Batch, Feature, Float32] = ???
val shardedX: Tensor2[Batch |@| X, Feature, Float32] =
  x.shard(mesh, Axis[Batch] -> MeshAxis[X]) // placement is part of the type
```

Figure C.5: Sharding the `Batch` dimension across a device mesh.

C.9 Readable compile errors

Type-level programming may raise the concern that type errors become unreadable. Scala 3 allows libraries to customize error messages, which DimWit uses to state the mistake in the user's vocabulary. For example, summing over an axis the tensor does not have (fig. C.6) fails with the message "`Axis[C] not found in Tensor[(A, B)]`", followed by the type-level search trace. The first line states the semantic mistake; the trace below is rarely needed.

```
val t: Tensor2[A, B, Float32] = ???
val res = t.sum(Axis[C]) // compilation error:  Axis[C] not found
```

Figure C.6: Compile error for summing over an axis the tensor does not have.

D Supplementary examples of comprehensible tensor programs

D.1 Variational autoencoder: Semantic types for probabilistic objects

DimWit provides probability distributions such as `Normal`, typed by the dimensions of the space they range over. Figure D.1 exemplifies this on a variational autoencoder over a flat pixel vector: the prior $p(z)$, posterior $q(z \mid x)$, and decoder $p(x \mid z)$ appear as three signatures, so the graphical model is readable from the types alone. The prior and posterior share one type `Latent`, making them directly comparable. Sampling is the only stochastic step, indicated by `Key` (appendix C.1).

D.2 k-means and k-medians: Semantic tensor types for better abstractions

The k-means algorithm has a structurally analogous counterpart called k-medians. Figure D.2 illustrates how we abstract these two algorithms with DimWit, highlighting that they differ only in their distance metric and cluster center calculation. Writing the signatures first, in the spirit of type-driven development (Brady, 2017), helped identify that distance is best defined per-point rather than over all points, keeping all shared logic in the base trait.

```
trait Pixel
trait Latent

trait VariationalAutoencoder:
  val prior: Normal[Tuple1[Latent], Float32]                                    // p(z)
  def posterior(x: Tensor1[Pixel, Float32]): Normal[Tuple1[Latent], Float32]    // q(z | x)
  def decoderLogits(z: Tensor1[Latent, Float32]): Tensor1[Pixel, Float32]       // p(x | z)

  def reconstruct(x: Tensor1[Pixel, Float32], key: Key): Tensor1[Pixel, Float32] =
    val z = posterior(x).sample(key)   // z ~ q(z | x)
    sigmoid(decoderLogits(z))          // x̂ = 𝔼[x | z]

  def generate(key: Key): Tensor1[Pixel, Float32] =
    val z = prior.sample(key)          // z ~ p(z)
    sigmoid(decoderLogits(z))          // x̂ = 𝔼[x | z]
```

Figure D.1: Variational autoencoder over a flat pixel vector in DimWit.

```
trait CenterBasedClustering(val nClusters: Int):
  // Algorithm-specific operations. Scoping type signatures guide abstraction design.
  def distance(point: Tensor1[Dim, Float32], center: Tensor1[Dim, Float32]): Tensor0[Float32]
  def calcCenter(clusterPoints: Tensor2[Point, Dim, Float32]): Tensor1[Dim, Float32]

  // Shared tensor logic and control flow.
  def step(
    points: Tensor2[Point, Dim, Float32], centers: Tensor2[Cluster, Dim, Float32]
  ): Tensor2[Cluster, Dim, Float32] =
    val dists = points.vmap(Axis[Point]): p =>
      centers.vmap(Axis[Cluster]): c =>
        distance(p, c)
    val assignments: Tensor1[Point, Int32] = dists.argmin(Axis[Cluster])
    val newCenters = (0 until nClusters).map: clusterIndex =>
      val mask = assignments.elementEquals(clusterIndex)
      calcCenter(where_!(mask, points, Float.NaN))
    stack(newCenters, Axis[Cluster])

class KMeans(nClusters: Int) extends CenterBasedClustering(nClusters):
  def distance(point: Tensor1[Dim, Float32], center: Tensor1[Dim, Float32]): Tensor0[Float32] =
    (point - center).pow(2).sum           // Squared L₂ distance
  def calcCenter(clusterPoints: Tensor2[Point, Dim, Float32]): Tensor1[Dim, Float32] =
    clusterPoints.nanmean(Axis[Point])    // mean for new center

class KMedians(nClusters: Int) extends CenterBasedClustering(nClusters):
  def distance(point: Tensor1[Dim, Float32], center: Tensor1[Dim, Float32]): Tensor0[Float32] =
    (point - center).abs.sum              // L₁ distance
  def calcCenter(clusterPoints: Tensor2[Point, Dim, Float32]): Tensor1[Dim, Float32] =
    clusterPoints.nanmedian(Axis[Point])  // median for new center
```

Figure D.2: `CenterBasedClustering` abstracts the k-means and the k-medians algorithms.

E Ecosystem workaround: Lift Python libraries to Scala

ScalaPy enables the direct integration of Python code within Scala. This allows Python libraries to be used from Scala, seamlessly yet untyped. Figure E.1 shows an example of how to integrate Weights & Biases logging within DimWit using the existing Python SDK. This workaround is available for any Python library.

```
val wandb = py.module("wandb")
py.`with`(wandb.init("<entity>", "<project>")): run =>
   val metrics = global.dict(Map("accuracy" -> 0.9, "loss" -> 0.1))
   run.log(metrics)
```

Figure E.1: Lifting the Weights & Biases Python SDK into Scala via ScalaPy.