Weaves, Wires, and Morphisms: Formalizing and Implementing the Algebra of Deep Learning

Anonymous authors
Paper under double-blind review

Abstract

Despite deep learning models running well-defined mathematical functions, we lack a formal mathematical framework for describing model architectures. Ad-hoc notation, diagrams, and pseudocode poorly handle nonlinear broadcasting and the relationship between individual components and composed models. This paper introduces a categorical framework for deep learning models that formalizes broadcasting through the novel axis-stride and array-broadcasted categories. This allows the mathematical function underlying architectures to be precisely expressed and manipulated in a compositional manner. These mathematical definitions are translated into human manageable diagrams and machine manageable data structures. We provide a mirrored implementation in Python and TypeScript to show the universal aspect of our framework, along with features including algebraic construction, graph conversion, PyTorch compilation and diagram rendering. This lays the foundation for a systematic, formal approach to deep learning model design and analysis.

1 Introduction

Deep learning models implement precisely defined mathematical computations, yet the way we describe model architectures remains surprisingly informal. In practice, architectures are communicated through a mixture of tensor notation, framework-specific code, and hand-drawn diagrams. These representations are useful locally, but they do not provide a single formal object on which one can systematically reason. As a result, properties that should in principle be derivable from a model's mathematical definition, such as equivalent formulations, efficient low-level implementations, performance models, or realizations in multiple software frameworks, are often discovered through manual derivation and engineering intuition rather than obtained procedurally. A more explicit mathematical language for architectures would make these relationships easier to analyze, compare, and eventually automate.

This need is especially acute for broadcasting. In modern tensor programs, the meaning of an operation is determined not only by the underlying local function, but also by how that function is lifted over additional axes. Broadcasting governs how computation is replicated across batches, tokens, heads, channels, or spatial positions, and in realistic models these lifts are not limited to simple batching. Axes may be rearranged, duplicated, or removed. Broadcasting therefore lies at the interface between the mathematical definition of a model and its eventual parallel execution on hardware. Recent work suggests that important optimizations can arise directly from this structure. For example, *FlashAttention* (Dao et al., 2022) can be viewed through the broadcasting pattern of attention, and *FlashAttention on a Napkin* (Abbott & Zardini, 2025) shows that this pattern can be analyzed diagrammatically and used to derive efficient implementations. What is still missing is a general formalism that makes such derivations systematic rather than exceptional.

Existing descriptions do not fully solve this problem. Standard deep learning notation is concise, but it often leaves shape transformations and nonlinear broadcasting implicit. Framework code is executable, but it entangles the mathematical content of a model with the idiosyncrasies of a particular software stack. Formal accounts of tensor algorithms and named dimensions provide valuable partial tools (Chiang et al., 2023; Phuong & Hutter, 2022), but they are not, on their own, a compositional framework for building, transforming, and analyzing whole models. To support systematic reasoning, we need a representation that

can express individual components, composition between components, and the way broadcasting changes the shape and semantics of those components.

Category theory is a natural candidate because it is, at its core, a mathematics of composition and abstraction (Censi et al., 2024). It provides a language for describing how fundamental units called *morphisms* are built up through sequential composition and parallel products. Morphisms can represent a range of constructs in different categories, such as linear operations in **Vect**, functions in **Set**, backpropagated algorithms in **Para** Fong et al. (2019); Cruttwell et al. (2022; 2024); Gavranović (2024); Shiebler et al. (2021) and even concrete concepts such as resource/functionality requirements in **DP**, the context of system design optimization Zardini (2023). As these constructs share a basic underlying structure, we can convert between them, meaning a system built with category theory has access to a host of systematic, mathematical manipulations. For instance, the common symmetric monoidal category (see Section 3) structure can be translated to hypergraphs Piedeleu & Zanasi (2025) without loss of generality.

In machine learning, categorical methods have already been used to study backpropagation through **Para** (Fong et al., 2019; Cruttwell et al., 2022; 2024; Gavranović, 2024), as well as symmetries and equivariances in geometric deep learning (Veličković et al., 2018; Bronstein et al., 2021). However, most of this literature treats model components at a relatively abstract level: as functions, parameterized morphisms, or equivariant maps. The broadcasted tensor structure that dominates practical deep learning architectures has received much less direct categorical treatment.

Broadcasting itself has a strongly compositional character. Lifting an operation over an additional axis behaves like a systematic transformation of that operation, and broadcasts over shared axes compose in a predictable way. This observation underlies *Neural Circuit Diagrams* (NCDs) (Abbott, 2024; Abbott & Zardini, 2025; Abbott et al., 2025), where array axes are represented as wires and broadcasting is expressed by weaving those wires around an expression. NCDs make the structure of tensor programs visually explicit and have already been used to derive attention optimizations. Yet, in their current form, they remain primarily a diagrammatic interface. Without a full formalization of the underlying categorical structure, it is difficult to subject them to automated algebraic analysis or to connect them cleanly to machine-manageable representations.

In this paper, we develop such a formalization. We present a categorical framework for deep learning models that makes broadcasting explicit and compositional. Our starting point is a general construction for translating mathematical definitions into structured representations that can be realized as symbols, diagrams, or code (Section 2). We then instantiate this construction categorically (Section 3), introducing two categories tailored to tensor computation: the axis-stride category **St**, which captures axes and reindexings, and the array-broadcasted category **Br**, which captures arrays and broadcasted operations. These definitions yield a precise mathematical account of how model components are composed and how broadcasts act on them.

2 Encoding Mathematics

A formal framework for deep learning is only useful if the resulting objects can be manipulated by both humans and machines. In practice, we want to move between at least three views of the same object: a mathematical definition, a human-facing representation such as notation or a diagram, and a machine-facing representation such as code. This section introduces a general framework for doing so. The key idea is to separate the mathematical objects we wish to describe from the concrete terms used to represent them.

We begin with a set Γ of mathematical entities. These entities are equipped with a family of basic structure maps, which we call **core properties**. For each $k \in K$, a core property has the form $\pi_k : \Gamma_{k,i} \to \Gamma_{k,f}$, where $\Gamma_{k,i}, \Gamma_{k,f} \subseteq \Gamma$ specify the domain on which the property is defined and the codomain in which it lands. We also assume that finite products of entities are again entities, so that $\prod_{i \in I} \Gamma \subseteq \Gamma$. This lets us express multi-input structure without leaving the ambient space of entities.

Definition 1 (Mathematical Entities)**.** *A system of mathematical entities consists of:*

- *A set of* ***entities*** Γ*. We include finite products of entities, so that* $\prod_{i\in I}\Gamma \subseteq \Gamma$*. For any index* $\square$*, we write* $\Gamma_\square \subseteq \Gamma$ *for a relevant subset.*
- *A* K*-family of* ***core properties*** $\pi_k : \Gamma_{k,i} \to \Gamma_{k,f}$*.*

A **constructed term system** is a representation layer for Γ. Its terms may be symbolic expressions, diagrams, or code objects, but in every case they must faithfully represent the same underlying entities. Formally, we introduce a set of terms G together with an interpretation function $V_G : G \to \Gamma$. The map V_G tells us which mathematical entity a term denotes. As with entities, we include finite products of terms and interpret them componentwise:

$$\left(\prod_{i\in I} g_i\right) \fatsemi V_G = \prod_{i\in I} (g_i \fatsemi V_G)\,.$$

For each core property π_k, the term system should provide an internal counterpart

$$p_k : G_{k,i} \to G_{k,f}, \qquad G_\square := V_G^{-1}(\Gamma_\square),$$

so that evaluating inside the term system agrees with evaluating in the mathematical space after interpretation. This is the basic soundness condition of the framework.

Definition 2 (Constructed Term System)**.** *A constructed term system consists of:*

- *A set of* ***terms*** G*, again closed under finite products:* $\prod_{i\in I} G \subseteq G$*. For any subset* $\Gamma_\square \subseteq \Gamma$*, we write* $G_\square := V_G^{-1}(\Gamma_\square)$ *for the terms whose interpretations lie in* $\Gamma_\square$*.*
- *An* ***interpretation function*** $V_G : G \to \Gamma$*. On elements, we may write* $g \fatsemi V_G = [\![g]\!]$*.*
- *For each* $k \in K$*, an internal* ***core property*** $p_k : G_{k,i} \to G_{k,f}$*, satisfying:*
  - ***Applicability.*** *If* $g \fatsemi V_G \in \Gamma_{k,i}$*, then* $p_k(g)$ *is defined.*
  - ***Internal evaluation.*** *We have* $p_k \fatsemi V_G = V_G \fatsemi \pi_k$*.*

Operationally, terms can carry information in two complementary ways. In one case, a term remembers the inputs from which it was built. In the other, it stores the data needed to expose certain properties directly. We treat these two cases using **construction rules** and **root terms**, respectively. This data may either have a covariant character, so that $G_{k,i}$ indicates the value of $G_{k,f}$, or a contravariant character, where $G_{k,f}$ is embedded with $G_{k,i}$.

We start with the contravariant case. We choose a subset of core properties $C \subseteq K$ so that p_c for $c \in C$ are implemented as a **construction rules** $T_c : G_{c,i} \to G_{c,f}$ wherein we have a recovery function $\hat{T}_c : \mathrm{img}(T_c) \to G_{c,i}$ which obeys $\hat{T}_c \fatsemi T_c = \mathrm{Id}_{\mathrm{img}(T_C)}$. This means that the *inputs* $g_{c,i} \in G_{c,i}$ to the core property are embedded in the *output* $g_{c,f} \in \mathrm{img}(T_c)$. These are data wrappers, ensuring applicability. We define interpretation over these wrappers as $V_{\mathrm{img}(T_c)} = \hat{T}_c \fatsemi V_G \fatsemi \pi_k$, which implies internal evaluation.

Definition 3 (Construction Rules)**.** *For a chosen subset* $C \subseteq K$ *of core properties, we implement* p_c *for* $c \in C$ *as a construction rule* $T_c : G_{c,i} \to G_{c,f}$ *so that:*

- *There exists a recovery function* $\hat{T}_c : \mathrm{img}(T_c) \to G_{c,i}$ *so that* $\hat{T}_c \fatsemi T_c = Id_{\mathrm{img}(T_C)}$ *(i.e., for every* $g \in \mathrm{img}(T_c)$*, we have* $g \fatsemi \hat{T}_c \fatsemi T_c = g$*). This recovers data from the term.*
- *Interpretation is defined by* $V_{\mathrm{img}(T_c)} = \hat{T}_c \fatsemi V_G \fatsemi \pi_c$*.*

Compound construction rules are built from placing rules in parallel or composing them. For parallel rules $T_c \times T_d$, we have a recovery

$$\hat{T}_c \times T_d : \mathrm{img}(T_c \times T_d) \to G_{c,i} \times G_{d,i}.$$

For composed rules, $T_c \fatsemi T_d$, we have a recovery

$$\hat{T}_d \fatsemi \hat{T}_c : \mathrm{img}(T_c \fatsemi T_d) \to G_{c,i}.$$

In both cases, recovery holds. As compound rules are built from these elemental combinations, complex expressions built from construction rules can be decomposed and interpreted as non-constructed root terms.

An additional consequence of defining construction rules in this manner are **axioms**. If a term g may be created by (possibly compound) construction rules c or d, then we have

$$g \subseteq \mathrm{img}(T_c) \cap \mathrm{img}(T_d).$$

This corresponds to an equality in the entity space, as we have:

$$\begin{aligned} g \fatsemi V_G = (g \fatsemi \hat{T}_c \fatsemi V_G) \fatsemi \pi_c &= g \fatsemi \hat{T}_d \fatsemi V_G \fatsemi \pi_d \\ \underbrace{(g_{c,i} \fatsemi V_G)}_{\text{Entities in } \Gamma} \fatsemi \pi_c &= \underbrace{(g_{d,i} \fatsemi V_G)}_{\text{Entities in } \Gamma} \fatsemi \pi_d \end{aligned}$$

For instance, consider the case of a construction rules $T_{x+y} : \mathbb{R}^2 \to \mathbb{R}$ which generates the term "$x + y$", and then we set the compound construction rules $T_{(x+y)+z} : \mathbb{R}^3 \to \mathbb{R}$ and $T_{x+(y+z)} : \mathbb{R}^3 \to \mathbb{R}$ to both generate the term "$x + y + z$" without brackets. The term now lives in the image of both. It then follows that $\pi_+(\pi_+([\![x]\!], [\![y]\!]), [\![z]\!]) = \pi_+([\![x]\!], \pi_+([\![y]\!], [\![z]\!]))$ within Γ. Axioms have utility in diagrams, as seen in Section 3.

For non-constructed rules $\ell \in K \backslash C$, we do not use simple wrappers. Rather, we either embed $G_{\ell,f}$ in non-constructed root terms or define a method over applicable constructed terms. Non-constructed root terms are separated into **root types**. A root type corresponds to a subset $r \subseteq K \backslash C$ of construction rules for which only those are applicable. Terms in root types $g_r \in G_r$ are embedded with data which can derive all $\ell \in r$.

Definition 4 (Root Terms)**.** *For a subset $r \in K \backslash C$, the root term type is:*

$$G_r = \{g \in G \mid (\forall \ell \in K \backslash C. [\![g]\!] \in \Gamma_\ell \leftrightarrow \ell \in r) \wedge (\forall \ell \in C. g \notin \mathrm{img}(T_c))\}.$$

For the root term type, we define $p_{\ell,r} : G_r \to G_{\ell,f}$ for $\ell \in r$. This has to be done via extracting data from $p_{\ell,r}$, so that each $g_r \in G_r$ is akin to a tag for the subset r accompanied by each $g_{\ell,f}$.

Interpretation of a root term $g_r \in G_r$ results in an entity $\gamma_r \in \Gamma_r$ so that all properties are equivalent. As we construct the data of terms by assigning $g_{\ell,f}$ properties, we use **templates** to restrict the creation of this data only to terms which indeed correspond to an entity. Entities may differ on some non-core property which is not the focus of the representation. In this case, the root terms are equipped with **metadata** tags which can be used by some alternative representation. For non-constructed properties of constructed terms, we must define **methods**. If $\mathrm{img}(T_c) \subseteq G_{\ell,i}$ for $\ell \in K \backslash C$, we must define a method $m_{c,\ell} : G_{c,i} \to G_{\ell,i}$ so that $\hat{T}_c \fatsemi m_{c,\ell} \fatsemi V_G = \pi_\ell$. Overall, this leads to a table of applicability and interpretation over all terms (Table 2).

2.1 Placeholder Terms

Many useful expressions are only partially instantiated. For example, the expression "$x + y + z$" does not denote a concrete real number; instead, it denotes a construction pattern with three open inputs. In the present framework, this corresponds to a compound construction rule $T_{x+y+z} : \mathbb{R}^3 \to \mathbb{R}$ whose inputs have not yet been fixed. Such partially constructed terms are important because they preserve the degrees of freedom in an expression while still exposing its compositional structure.

In symbolic notation and diagrams, these open slots appear as free symbols. In code, they appear as terms equipped with unique identifiers (UIDs). This lets us manipulate expressions symbolically before committing to concrete inputs. For instance, imposing the substitution $y := z$ transforms the expression "$x + y + z$" into "$x + y + y$", and correspondingly transforms $T_{x+y+z} : \mathbb{R}^3 \to \mathbb{R}$ into $T_{x+y+y} : \mathbb{R}^2 \to \mathbb{R}$. Equivalently, the

Term Type	**Associated Data**	**Constructed Property**	**Non-constructed Property**	**Interpretation**
Product	$g_{\Pi j} = \Pi_{j \in J} g_j$	*Wrapping*	Method $m_{\Pi j,\ell} : \Pi_{j \in J} G_j \to G_{\ell,i}$ $\pi_\ell = m_{c,\ell} \fatsemi V_G$	$(\Pi_{j \in J} g_j) \fatsemi V_G = \Pi_{j \in J}(g_j \fatsemi V_G)$
Constructed Term	$g_{c,f} \mapsto g_{c,i}$ via $\hat{T}_c$	*Wrapping*	Method $m_{c,\ell} : G_{c,i} \to G_{\ell,f}$ $\pi_\ell = \hat{T}_c \fatsemi m_{c,\ell} \fatsemi V_G$	$g_{c,f} \fatsemi V_G = g_{c,f} \fatsemi \hat{T}_c \fatsemi V_G \fatsemi \pi_c$
Root Term	$g_r \mapsto \mu_r \times \Pi_{\ell \in r} g_{\ell,f}$ *Where m_r is non-core metadata.*	*Wrapping*	*Data Extraction* $p_{r,\ell} : G_r \to G_{r,\ell}$ $\pi_\ell = p_{r,\ell} \fatsemi V_G$	$g_r \fatsemi V_G = \gamma_r$ $\forall \ell \in r. g_r \fatsemi p_{r,\ell} \fatsemi V_G = \gamma_r \fatsemi \pi_\ell$

Table 1: Applicability and interpretation for the main term types.

substitution may be viewed as a meta-level rearrangement $[0, 1, 1] : \mathbb{R}^2 \to \mathbb{R}^3$, anticipating the rearrangement structure introduced in Section 3.

By equipping the term system with placeholder terms, we obtain a uniform account of symbolic manipulation across notation, diagrams, and code. We can scan an expression for free symbols or UID-tagged terms, generate the corresponding configuration space automatically, and impose canonical substitutions when performing algebraic simplifications.

3 Categories

Section 2 introduced a general framework for representing mathematical objects by symbolic terms, diagrams, and code. We now instantiate that framework with category theory (Censi et al., 2024). The motivation is practical as much as conceptual: deep learning models are built by composing operations sequentially and in parallel, and category theory provides a precise language for exactly these patterns. Informally, objects play the role of interfaces or types, while morphisms play the role of composable operations between them. This gives us a common structural language in which different views of the same model can be expressed and related.

For instance, consider two closely connected ideas: working with sets and functions, and working with measurable spaces and Markov processes. These are closely related, we can associatively compose functions and Markov processes if inputs and outputs have the same structure, and we can place functions or Markov processes in parallel, generating a product expression which acts on the Cartesian product of sets or parallel measurable spaces. These have a close relationship, we can map from sets to measurable spaces over them and from functions to Kronecker Markov processes. Category theory makes these similarities explicit by treating both settings as instances of a shared compositional template, i.e., a "symmetric monoidal product category". This template allows us to think primarily in terms of compositions and products, clearly identifying shared structure and relationships between them. Sets and functions are described by the category **Set**, while measurable spaces and Markov processes are described by the category **Stoch** (Fritz et al., 2023). Sets and measurable spaces are called the "objects" while functions and Markov processes are called the "morphisms".

These collections of objects and morphisms each form a category, which is further equipped with a notion of associative composition. Extending the base categorical template we have a monoidal product, which allows us to place objects and morphisms in parallel. This same template-based language can instruct us how the categories differ. Monoidal products have rearrangements, morphisms which manage data without altering the content. **Set** and **Stoch** differ in the critical respect that copying data is natural in **Set** but not in **Stoch**. Consider that copying the output of a function is equivalent to copying the input and applying the function twice, while copying a dice roll is not the same as two individual dice rolls. We can start to develop more elaborate categories, such as parametric functions **Para** which support backpropagation, vector spaces and linear maps **Vect**, or even novel categories to describe algorithmic resource usage. The key insight of

category theory is that these different perspectives are closely related and we can move between them with structure-preserving maps called functors. For formalizing deep learning, utilizing these multiple perspective is critical. Furthermore, for a categorical analysis to have applied utility, we must ensure that mathematical definitions can be implemented per the term-construction framework above. In Definition 5, we provide a definition of a monoidal product category with clearly delineated terms.

Definition 5 (Product Category)**.** *A product category $\mathcal{C} = \mathbf{Prod}[L, M]$ with lone objects L and root morphisms M consists of:*

- ***Objects.*** *Objects $A \in Ob\mathcal{C}$ of the category are finite products of lone objects $A = \Pi_{i\in I} A_i$ for $A_i \in L$. These products are associative. The empty product is the* ***unit object*** *(of length 0) $\mathbb{1} = \prod_{i\in\emptyset} A_i$. An object has a flat length, $L[A] \in \mathbb{N}$, indicating its number of lone objects.*
- ***Morphisms.*** *Morphisms $f : A \to B \in Mo\mathcal{C}$ have a domain $A \in Ob\mathcal{C}$ and a codomain $B \in Ob\mathcal{C}$. We indicate the collection of morphisms with shape $A \to B$ by $\mathcal{C}(A, B)$. Morphisms are generated by:*
  - ***Root Morphisms.*** *A collection of morphisms $m \in M$, equipped with metadata for key properties. The universal key properties are the domain and codomain, which are exposed by methods in the term system.*
  - ***Composition.*** *For a family of objects $(A_i)_{i\in I+1}$, there is an associative operation $\fatsemi_{(A_i)_{i\in I}} : \Pi_{i\in I}\mathcal{C}(A_i, A_{i+1}) \to \mathcal{C}(A_0, A_I)$ so that $(f \fatsemi g) \fatsemi h \equiv f \fatsemi (g \fatsemi h)$ ($\equiv$ indicates mathematical equivalence, or equivalent interpretation)*
  - ***Products.*** *There is an associative operation $\otimes : \Pi_{i\in I}\mathcal{C}(A_i, B_i) \to \mathcal{C}(\Pi_{i\in I} A_i, \Pi_{i\in I} B_i)$ so that we have $(f \otimes g) \otimes h \equiv f \otimes (g \otimes h)$ and* ***bifunctoriality****, $(f \fatsemi h) \otimes (g \fatsemi k) \equiv (f \otimes g) \fatsemi (h \otimes k)$.*
  - ***Rearrangements.*** *A product category has a collection of allowed remappings, $\mu : J \to I$ between discrete sets J and I. A subset of these are considered natural. Allowed remappings generate rearrangements $[\mu]_{(A_i)_{i\in I}} : \Pi_{i\in I} A_i \to \Pi_{j\in J} A_{\mu(i)}$ over I-families of objects $(A_i)_{i\in I}$. These morphisms obey;*
    - ***Identity Generation.*** *If $\mu = Id_I$, then $[Id_I] \equiv Id_{\Pi_{i\in I} A_i} : \Pi_{i\in I} A_i \to \Pi_{i\in I} A_i$ so that for $f : \Pi_{i\in I} A_i \to \Pi_{j\in J} B_j$ we have $[Id_I] \fatsemi f \equiv f \equiv f \fatsemi [Id_J]$.*
    - ***Composition.*** *For $\mu : J \to I, \nu : K \to J$, we have*

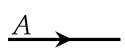
$$[\mu]_{(A_i)_{i\in I}} \fatsemi [\nu]_{(A_{\mu(j)})_{j\in J}} = [\nu \fatsemi \mu]_{(A_i)_{i\in I}}$$

    - ***Products.*** *For $\rho : K \to L$, we have $[\mu] \otimes [\rho] \equiv [\mu \oplus \rho]$ where; $(\mu \oplus \rho)(\ell) = (\mu(\ell)$ if $\ell \in J$ else $I + \mu(\ell - J))$ (see Appendix 6)*
    - ***Naturality.*** *If μ is considered natural, then for a family of morphisms $(f_i)_{i\in I}$ where $f_i : A_i \to B_i$, then $(\Pi_{i\in I} f_i) \fatsemi [\mu] \equiv [\mu] \fatsemi (\Pi_{j\in J} f_{\mu(j)})$. For this to work, we have associativity compatibility, outlined in Appendix 15.*
  - ***Blocks.*** *A block is a function which preserves domains and codomains, $B : \mathcal{C}(A, B) \to \mathcal{C}(A, B)$. This corresponds to loops, or aesthetic tags.*

3.1 Implementing Categories

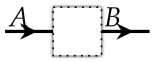
We now realize Definition 5 in diagrams and code using the construction scheme of Section 2. Product objects, composition, products of morphisms, and blocks are construction rules. Root morphisms and rearrangements are root terms. Domains, codomains, and rearrangement data are exposed by metadata and methods. The high-level scaffold is shared by all product categories; specific categories arise by choosing the root morphisms and the class of allowed and natural remappings.

- **Objects** $Ob\mathcal{C}$**.** Objects are terms which anchor composition. Every object is represented as a tuple of *lone objects* $A = \Pi_{i\in I} A_i$, together with the unit object $\mathbb{1} = \Pi_{i\in\emptyset}$. In formal diagrams, objects are drawn as wires with arrows.



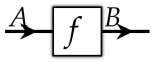

```
class ProductObject[L]:
    content: Prod[L]
```

- **Morphisms** Mo$\mathcal{C}$. Morphisms are composable terms with a *domain* and *codomain* object. These interfaces determine when sequential composition is valid, and in diagrams they appear to the left and right of a morphism.

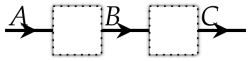



```
abstract class Morphism[L]:
    def dom(self) -> ProdObject[L]
    def cod(self) -> ProdObject[L]
```

Compound morphisms are built from composition and products, ultimately terminating in seed morphisms equipped with metadata. At minimum, that metadata supplies the domain, codomain, and a symbolic or pictorial description of how the morphism acts.

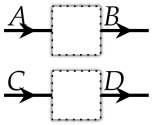


- **(Sequential) Composition.** Composition combines morphisms horizontally whenever the codomain of one matches the domain of the next. The resulting term is again a morphism, and associativity means that only the order of morphisms matters, not the parenthesization.



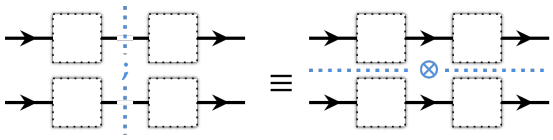

```
class Composed[L, M: Morphism[L]]
    (Morphism[L]):
    content: Prod[M]
    def dom(self):
        return self.content[0].dom()
    def cod(self):
        return self.content[-1].cod()
```

- **(Parallel) Product.** The product of objects concatenates their lone object contents, while the product of morphisms concatenates the corresponding domains and codomains. In diagrams, this is vertical stacking.

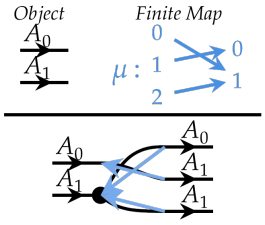



```
class ProductOfMorphisms
    [L, M: Morphism[L]]
    (Morphism[L]):
    content: Prod[M]
    def dom(self):
      return ProdObject(concat(
        m.dom() for m self.content))
    def cod(self):
      return ProdObject(concat(
        m.cod() for m self.content))
```

Products place morphisms in parallel, so bifunctoriality allows horizontal and vertical composition to be interchanged when shapes match. Because diagrammatic terms are built from $\fatsemi$ and $\otimes$, this law is enforced by construction.

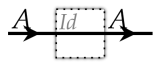


- **Rearrangements.** Given a domain $A = \Pi_{i \in I} A_i$ and a mapping $\mu : J \to I$, we generate a rearrangement morphism $[\mu] : \Pi_{i \in I} A_i \to \Pi_{j \in J} A_{\mu(j)}$ so that the jth output comes from the $\mu(j)^{\text{th}}$ input.

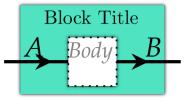



```python
class Rearrangement[L](Morphism[L]):
    mapping: Prod[int]
    _dom: Prod[L]
    def dom(self):
        return ProdObject(self._dom)
    def cod(self):
        return ProdObject(
            self._dom[mu_j]
            for mu_j in self.mapping)
```

When the mapping is the identity, we generate an identity morphism.



- **Blocks.** Blocks serve an organizational role in diagrams and code. They can distinguish functionality, group subexpressions, or indicate repeated structure without changing the underlying type. They are diagrammed by placing a backdrop behind an expression.




```python
class Block[L, M: Morphism[L]](Morphism[L]):
    body: M
    # Indicates the block's function.
    block_tag: BlockTag = BlockTag()
    def dom(self): return self.body.dom()
    def cod(self): return self.body.cod()
```

3.2 Adding Structure

The definition above gives a generic template for product categories, but by itself it does not say how morphisms are distinguished or how they can be probed. To support deep learning, we want as little additional structure as possible while still covering deterministic, stochastic, and linear viewpoints. The next ingredient is therefore a notion of *elements*: enough "points" of an object to characterize morphisms by their action.

3.2.1 Elemental Categories

The first structural element we will introduce are **elements**. In the category **Set**, elements of a set X correspond one-to-one with the morphisms $\mathbf{Set}(\mathbb{1}, X)$, that is, constant functions with no input. In other categories this correspondence need not hold literally, but the same idea can still be used: we select a distinguished family of morphisms from the unit object that is rich enough to separate morphisms. This leads to the notion of an elemental category.

Definition 6 (Elemental Category)**.** *An* elemental category *is a product category $\mathcal{C}$ together with, for each object X, a distinguished collection of elements $\mathrm{El}(X) \subseteq \mathcal{C}(\mathbb{1}, X)$ such that:*

- ***Unique specification.*** *For morphisms $f, g : X \to Y$, if $x \fatsemi f = x \fatsemi g$ for all $x \in \mathrm{El}(X)$, then $f = g$. Equivalently, evaluation on elements gives an injective map*

$$\mathcal{C}(X, Y) \hookrightarrow \mathbf{Set}\big(\mathrm{El}(X), \mathcal{C}(\mathbb{1}, Y)\big).$$

  A morphism $f : X \to Y$ which maps from elements to elements, so that for all $x \in \mathrm{El}(X)$ we have $x \fatsemi f \in \mathrm{El}(Y)$, is called ***deterministic****.*

- ***Product construction.*** *Elements of product objects are tuples of elements of the factors:*

$$\mathrm{El}\left(\prod_{i \in I} A_i\right) = \left\{ \prod_{i \in I} a_i \;\middle|\; a_i \in \mathrm{El}(A_i) \text{ for all } i \in I \right\}.$$

- ***Elemental naturality.*** *The action of allowed rearrangements* $[\mu]_{(A_i)_{i\in I}} : \Pi_{i\in I} A_i \to \Pi_{j\in J} A_{\mu(j)}$ *on elements* $\Pi_{i\in I} a_i \in \mathrm{El}(\Pi_{i\in I} A_i)$ *is given by;*

$$\left(\prod_{i\in I} a_i\right) \fatsemi [\mu]_{(A_i)_i\in I} = \prod_{j\in J} a_{\mu(j)}$$

- ***(Optional) Completeness.*** *If the minimal valid choice of* $\mathrm{El}(X)$ *is the full hom-set* $\mathcal{C}(\mathbb{1}, X)$ *for every* $X \in Ob(\mathcal{C})$*, then we call the category* ***complete elemental****.*

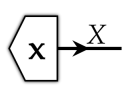
Consequently, products act pointwise on elements. For morphisms $f : X \to Y$ *and* $h : Z \to W$*,*

$$(x \otimes z) \fatsemi (f \otimes h) = (x \fatsemi f) \otimes (z \fatsemi h) \qquad \textit{for all } x \in \mathrm{El}(X),\ z \in \mathrm{El}(Z).$$

This definition captures several familiar settings. In **Set**, elements are ordinary set elements, so **Set** is complete elemental. In **Vect** (vector spaces and linear maps), one may take a basis such as the unit vectors; these separate linear maps even though arbitrary vectors are not themselves singled out as chosen elements. In **Stoch** (measurable sets and Markov processes), the analogous notion is the set of Kronecker distributions on the underlying measurable space. These are sufficient to determine a Markov process by its action on point masses, even though $\mathbf{Stoch}(\mathbb{1}, X)$ contains all probability distributions on X. More generally, the same template can describe restricted classes of set maps, provided the chosen elements still separate morphisms.

We diagram elements as left-pointing pentagons:

- **Elements.** Elements of a lone object X_i are morphisms $x_i : \mathbb{1} \to X_i$. Elements of a product object $\prod_{i\in I} X_i$ are tuples generated from elements of the constituent objects. Elements are a subset of seed morphisms.



3.2.2 Types of Products

The main freedom left in Definition 5 lies in the choice of allowed and natural remappings. These determine which structural manipulations of data are available and which commute with ordinary morphisms. In practice, this is what distinguishes functional, probabilistic, and linear settings.

We classify a remapping $\mu : J \to I$ by its **count**

$$\mathrm{Count}[\mu](i) = \big|\{j \in J \mid \mu(j) = i\}\big|.$$

A count of 0 means that the ith input is discarded, while a count greater than 1 means that it is duplicated. Allowed remappings are those for which the corresponding rearrangement exists. Natural remappings are those whose rearrangements can be moved through ordinary morphisms:

$$\left(\prod_{i\in I} f_i\right) \fatsemi [\mu]_{(B_i)_{i\in I}} = [\mu]_{(A_i)_{i\in I}} \fatsemi \left(\prod_{j\in J} f_{\mu(j)}\right).$$

Several standard categorical regimes arise from this distinction. If remappings with count exactly one are allowed and natural, we obtain a **symmetric monoidal category**. For example, swapping two outputs is the same as swapping the two inputs and the order of the functions: $(f \otimes g) \fatsemi [1, 0] = [1, 0] \fatsemi (g \otimes f)$. If arbitrary remappings are allowed, we obtain a **copy-discard category**. This covers functions, algorithms, and Markov processes, where data can be routed, duplicated, or discarded at the structural level. By

contrast, in **Vect** copying and discarding are not linear operations in general, so they do not belong to the same categorical structure.

Among copy-discard categories, further distinctions depend on which remappings are natural. If count reduction is natural, then deletion commutes with morphisms. This models, for instance, ignoring an output of a function or taking a marginal of a random process. If arbitrary counts are natural, then copying also commutes with morphisms, and we recover a **Cartesian category**. This is appropriate for deterministic functions but not for stochastic kernels: copying a single dice roll is not the same as generating two independent dice rolls.

Definition 7 (Product Templates)**.** *A product category has a class of* ***allowed*** *remappings and, among them, a class of* ***natural*** *remappings. An allowed remapping $\mu : J \to I$ is a function between finite discrete sets. Over a family of objects $(A_i)_{i\in I}$, it induces a rearrangement*

$$[\mu]_{(A_i)_{i\in I}} : \prod_{i\in I} A_i \to \prod_{j\in J} A_{\mu(j)}.$$

If μ is natural, then for every family of morphisms $(f_i)_{i\in I}$ with $f_i : A_i \to B_i$:

$$\left(\prod_{i\in I} f_i\right) \fatsemi [\mu]_{(B_i)_{i\in I}} = [\mu]_{(A_i)_{i\in I}} \fatsemi \left(\prod_{j\in J} f_{\mu(j)}\right).$$

The count of a remapping $\mu : J \to I$ is $\mathrm{Count}[\mu](i) = \left|\{j \in J \mid \mu(j) = i\}\right|$. The standard templates are:

- ***Symmetric monoidal.*** *Remappings with $\mathrm{Count}[\mu](i) = 1$ are allowed and natural.*
- ***Copy-discard.*** *Arbitrary remappings are allowed, but not necessarily natural.*
- ***Deletion.*** *Remappings with $\mathrm{Count}[\mu](i) \leq 1$ are natural. By Fox's theorem (Fox, 1976) (Appendix Thm. 1), this yields an injective map*

  $$\prod_{i\in I} \mathcal{C}(A, B_i) \to \mathcal{C}\left(A, \prod_{i\in I} B_i\right).$$

  Deletion is typically provided by taking an element $\langle a| : \mathbf{1} \to I$ for $a \in I$, providing a projection rearrangement $[a]_{(A_i)_{i\in I}} : \Pi_{i\in I} A_i \to A_a$.
- ***Cartesian.*** *All remappings are natural. By Fox's theorem (Appendix 1), this yields a bijective map*

  $$\prod_{i\in I} \mathcal{C}(A, B_i) \to \mathcal{C}\left(A, \prod_{i\in I} B_i\right).$$

  Copying is typically provided by taking the remapping $\delta^I : I \to \mathbf{1}$, mapping all inputs to the unique element $0 \in \mathbf{1}$, providing a copying rearrangement $[\delta^I]_A : A \to \Pi_{i\in I} A$.

These levels of structure are precisely what we need later. By working at the level of elemental copy-discard categories, we retain enough generality to describe functional, probabilistic, and algorithmic views of deep learning models, while still allowing stronger assumptions such as Cartesianity when a particular application supports them.

4 Array-Broadcasted Category

The structure outlined in Section 3 can be used as a template to describe parallel, compositional structures. These templates can be filled by providing different lone objects and root morphisms. We will use

this structure to describe deep learning models with the axis-stride category **St** and the array-broadcasted category **Br**.

These address the challenge of describing broadcasting, a key aspect of deep learning models which existing frameworks fail to capture. Broadcasting describes how operations are parallelized over additional axes. Deep learning models can be fundamentally thought of as composed and parallel broadcasted operations. This broadcasting closely relates to parallel GPU execution, and therefore clearly understanding and mathematically manipulating broadcasting is essential to relating the mathematics of deep learning models to efficient execution. PyTorch, the standard framework for expressing deep learning models, has convoluted broadcasting semantics, borrowed from NumPy. Linear algebra notation (Goodfellow et al., 2016) fails to describe non-linear broadcasting, leading to unclear expressions of models (Abbott, 2024).

The axis-stride category **St** supplies the ability to describe the shape of array via its objects and the relationship between array coordinates via its morphisms. Its lone objects are **axes**, equipped with a numeric size indicating allowed coordinate values. An axis A has a size $|A| \in \mathbb{N}$ indicating its number of elements. In implementations, axes and their size are assigned UIDs, to allow automatic alignment of expressions. The product of lone object axes naturally form shapes, describing the possible coordinates of arrays via elements. The morphisms of the axis-stride category **St** are finite affine transforms. These describe reliable and compressible expressions for relating coordinates of arrays.

Definition 8 (Axis-Stride Category)**.** *The axis-stride category* **St** *is a complete elemental Cartesian product category with;*

- ***Axes*** *as lone objects. Each axis A has a size $|A| \in \mathbb{N}$. An axis has an element $|i_A\rangle \in El(A)$ for each $i \in |A|$. Objects $\Pi_{i\in I} A_i \in Ob\mathbf{St}$ are products of axes, representing the* ***shapes*** *of arrays. Elements correspond to coordinates within the shape.*

- ***Finite Affine Transformations*** *as root morphisms. A root morphism $\eta : \Pi_{i\in I} A_i \to \Pi_{j\in J} B_j$ is identified by an $\mathbb{N}^{I\times J}$ linear matrix Λ^η and an $\mathbb{N}^J$ vector v^η. The action on elements $\Pi_{i\in I} a_i \in El(\Pi_{i\in I} A_i)$ performs;*

$$\Pi_{i\in I} |a_{A_i}\rangle \fatsemi \eta = \Pi_{j\in J} |v^\eta_j + \Sigma_{i\in I} a_i \cdot \Lambda^\eta_{ij}\rangle$$

  Under the restriction that the image of the function is captured by the codomain.

The array-broadcasted category **Br** represents deep learning models. The array-broadcasted category is a deletion product category, allowing it to capture both a deterministic **Set** and probabilistic **Stoch** perspective. Lone objects are arrays $[a, A]$, which have a base datatype set $a \in \mathbf{Dt}$, and a shape $A = \Pi_{i\in I} A_i \in \mathrm{Ob}\mathbf{St}$. Datatypes are measurable sets equipped with auxiliary information. The elements of array objects $\mathbf{x} \in \mathrm{El}([a, A])$ have a value in a for each coordinate along A, and thus correspond to $\mathrm{El}(A)$-families $(x_{i_A})_{|i_A\rangle \in \mathrm{El}A}$ of values $x_{i_A} \in a$. Root morphisms are broadcasted operations. As an elemental category, they are defined by action on array product elements. Their specific behaviour with respect to broadcasting is developed in Section 4.1.

Definition 9 (Array-Broadcasted Category)**.** *The array-broadcasted category* **Br** *is a product category with;*

- ***Arrays*** *as lone objects. Arrays $[a, A]$ consist of a base datatype $a \in \mathbf{Dt}$ and a product object $A \in Ob\mathbf{St}$. Elements $\mathbf{x} \in El([a, A])$ of array objects correspond to families $(x_i)_{|i_A\rangle \in El(A)}$ of values $x_i \in a$.*

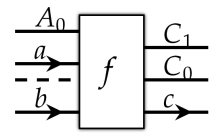
- ***Array-Product Correspondence.*** *Each array object $[a, A]$ has access to an $El(A)$-family of* ***indexes****, morphisms $[a, i] : [a, A] \to [a, \mathbb{1}]$ for elements $\langle i| : \mathbb{1} \to A$. The collective structure of these indexes matches the product category. Morphisms $x, y : \mathbb{1} \to [a, A]$ are equivalent if they are equivalent over the dependent separator, given by;*

$$[\delta^A]_{[a,A]} \fatsemi \prod_{i\in El(A)} [a, i]$$

- ***Broadcasted Operations*** *as root morphisms. As an elemental category, these are distinguished by their action on tuples of arrays. The precise definition and implementation of broadcasted operations is outlined in Section 4.1.*

The array-broadcasted category is diagrammed in a unique manner. We want to clearly diagram the axes of arrays, but want to distinguish this notion of stacking from products. Therefore, we use dashed lines to indicate products of objects and morphisms. Arrays are diagrammed by stacking wires representing axes without arrows on top of a wire representing the base datatype. The base datatype wire has an arrow. We may have an implicit base datatype, usually $\mathbb{R}$, allowing us to skip drawing the arrowed wire. This results in a diagram akin to Figure 1.

Figure 1: Here, we have diagrammed an array morphism $f : [a, A_0] \times [b, \mathbb{1}] \to [c, C_0 C_1]$. The dashed line indicates an array product (tupling).



4.0.1 Note on Specification

This outline leaves open multiple directions for further exploration, which fall outside the scope of this paper. We leave open the variety of allowed datatypes, and which exact morphisms are allowed. Common datatypes will include numeric values $n \in \mathbb{N}$ with a discrete measure, real numbers $\mathbb{R}$ over a Lebesgue measure, and quantized values represented by $\mathbb{R} \times \{q\}$, where $q \in Q$ is a tag representing the quantization used. Representing quantized values in this manner rather than a set of all possible, say, $FP16$ values is critical for differentiability to be maintained and there to be a relationship between the same mathematical operation defined at different levels of quantization.

A key advantage of defining arrays and broadcasting with finitely sized shapes is that we maintain measurability. Allowing any set to define the base and exponent of an array would result in arrays such as $[\mathbb{R}, \mathbb{R}]$ or $[\mathbb{R}^{\mathbb{R}}, \mathbb{1}]$, which would not be measurable, computable, or differentiable in any meaningful sense.

Regarding allowed morphisms, for now we assert that they are differentiable almost everywhere with respect to continuous inputs. This restricts them to measurable, computable, and differentiable operations. These foundations allow our system to be extended to a backpropagated **Para** Fong et al. (2019); Cruttwell et al. (2022; 2024); Gavranović (2024); Shiebler et al. (2021) framework and to accommodate the effects of quantization in a precise manner that describes models as they occur in practice.

4.1 Broadcasting

To formally describe broadcasting in a manner that can be implemented, we will first build up constructs to describe reindexing morphisms, which manipulate arrays without changing the content of data. These rely on the axis-stride **St** category to describe the relationship between values located at certain output coordinates with respect to values located at input coordinates. These reindexing morphisms smother a batch-lifted base operation, allowing for intricate broadcasting dynamics to be precisely defined, diagrammed, and represented.

Definition 10 (Reindexing)**.** *Given a stride morphism $\eta : P \to Q$ from* **St** *and a base datatype a, we have an identity reindexing morphism $[a, \eta] : [a, Q] \to [a, P]$ in* **Br** *so that action on elements $(a_i)_{i \in Q}$ is given by;*

$$(a_i)_{i \in El(Q)} \fatsemi [a, \eta] = (a_{\eta(j)})_{j \in El(P)} \tag{1}$$

In the case that the reindexing is an element $p : \mathbb{1} \to P$ of P, then we recover an **index** $[a, p] : [a, P] \to [a, \mathbb{1}]$, mapping $(a_i)_{i \in \mathrm{El}(Q)}$ to $(a_q)_{k \in \mathbb{1}}$ for $q \in \mathrm{El}(Q)$. We diagram reindexings with hexagons, allowing us to express Equation 1 by Figure 2. Furthermore, in the case that the stride morphism consists of identities and elements, we we have a slice, corresponding to Pythonic `__getitem__` operations such as `x[i,:,j]`, which are realized as morphisms $[a, i_{A0} \otimes A_1 \otimes j_{A2}] : [a, A_0 \otimes A_1 \otimes A_2] \to [a, A_1]$, shown in Figure 3. We use a notational shorthand that the product of an object and morphism (in this case, $i_{A0} : \mathbb{1} \to A_0$) treats the object as its own identity. We can use a range of compatible shorthands for reindexing lifts. These include;

- **Object-Object Lift.** For an array product $X = \Pi_{i \in I}[a_i, A_i] \in \text{Ob}\mathbf{Br}$ and a shape $P \in \text{Ob}\mathbf{St}$, we have $[X, P] = \Pi_{i \in I}[a_i, A_i \otimes P]$.
- **Object-Morphism Lift.** For an array product $X = \Pi_{i \in I}[a_i, A_i] \in \text{Ob}\mathbf{Br}$ and a stride morphism $\eta : P \to Q \in \text{Mo}\mathbf{St}$, we have $[X, \eta] = \Pi_{i \in I}[a_i, A_i \otimes \eta]$ with form $[X, \eta] : [X, Q] \to [X, P]$.

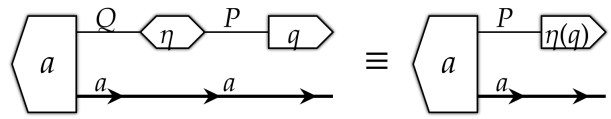


Figure 2: Reindexings are represented with hexagons passing over base operations. Their action "absorbs" indexes into themselves.

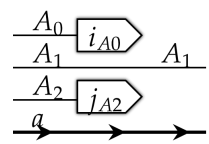


Figure 3: Slices are reindexings built from elements and identities, and correspond to Pythonic slices.

Next, we will define batch lifting, running an array-broadcasted morphism $f : X \to Y$ in P-fold parallel over a shape P. A deletion monoidal category provides infrastructure to naturally convert a family of morphisms $\Pi_{i \in I}\mathcal{C}(A, B_i)$ to a morphism $\mathcal{C}(A, \Pi_{i \in I} B_i)$ (see Theorem 1). This natural map builds on the structure of the category. For example, in **Stoch** this map generates independent distributions, while in **Set** we generate a function which provides the result of each f_i in each of the outputs. When defining a batch lift, we want to borrow this natural structure. As a result, each index along the batch lifted axis can be independently calculated, reflecting GPU execution. While reindexing describes $[X, \eta]$ for a product array object and stride morphism, batch lifting describes $[f, P]$ for a broadcasted morphism and shape object.

Definition 11 (Batch Lifting)**.** *Given a base morphism $f : X \to Y \in Ob\mathbf{Br}$ in the array-broadcasted category* **Br** *and a shape $P \in Ob\mathbf{St}$ the batch lift $[f, P] : [X, P] \to [Y, P]$ is defined so that;*

$$(\textit{copy remapping})\ \ \delta^P : P \to \mathbf{1}$$

$$[f, P] \fatsemi [\delta^P]_{[Y,P]} \fatsemi \prod_{p \in El(P)} [Y, p] = [\delta^P]_{[X,P]} \fatsemi (\prod_{p \in El(P)} [X, p] \fatsemi f) \tag{2}$$

In a deletion product category, we can extract $[q]_{(Y)_{p \in El(P)}}$ for some $q : \mathbf{1} \to |P|$, to obtain;

$$[f, P] \fatsemi [Y, q] = [X, q] \fatsemi f \tag{3}$$

There are two key insights from this definition. Firstly, from Equation 3, a batch lift is defined so that the pth output slice along the lifted shape is defined by the base morphism acting on the pth input slice. This reflects, for instance, a parallel row-wise operation where the pth output row is generated by the pth input row. This action is shown in Figures 4 and 5. As a result of this expression, we can conceptualize broadcasting as weaving a shape over an underlying expression and letting the slice pass over. Secondly, the full definition provided by Equation 2 indicates that the separate indexes are generated in the natural, independent manner of the underlying category. Each output slice is generated by a separate instantiation of f, describing true, parallel evaluation. This concept is shown in Figure 6, where we see how copying interacts with collectively defining the values along each slice.

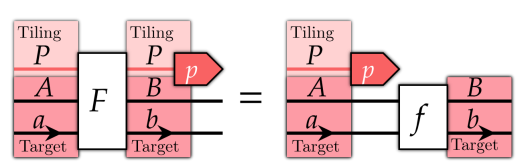


Figure 4: Equation 3 defines F on the left, so that slices over the output P-axis correspond to f acting over slices of the input P-axis.

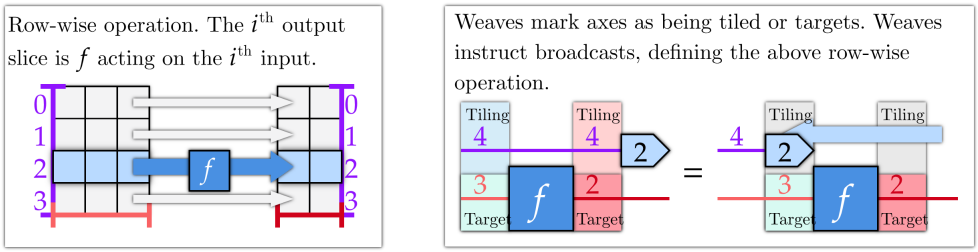


Figure 5: On the left, we visualize a row-wise operation conducted over every row-slice. We see how this corresponds to the 2 slice of the output being generated by the 2 slice of the input. This can be diagrammed in an alternative manner, where we weave the tiled row axis around the expression, showing how the indexes transfer.

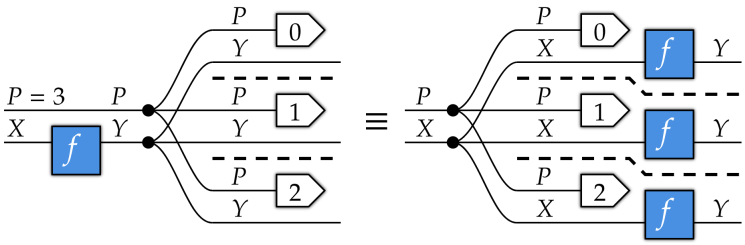


Figure 6: Equation 2 describes a collective broadcast over each weaved slice. The value of the broadcasted operation $[f, P]$ over each index is defined by running f on each input slice independently. This independence is indicated by the multiple instances of f on the right hand side. In this diagram, we make the underlying datatype implicit and have $P = 3$.

To handle more complex broadcasts, such as those that often occur with matrix multiplication over selected axes, we must provide a standard means of selecting key axes and considering the relationship between output and input indexes. Selecting axes is provided by constructs called weaves, which select some inputs to be placed towards the front and others towards the back of a product. They are formally described in Definition 12, but the diagram in Figure 7 provides a more intuitive explanation where axes are split into target and tiled segments.

Definition 12 (Weave)**.** *A family of booleans $(w_i)_{i \in I}$ where $w_i \in \mathbf{2}$ generates a **weave**, a function $\Omega_w : I \to I$ given by;*

$$\Omega_w(i) = \begin{cases} |j \in i.w(j)| & , w(i) \\ \Sigma_{i \in I} w_i + |\{j \in i | \neg w(j)\}| & , \neg w(i) \end{cases}$$

*Note that $j \in i$ indicates that $j \in \{0, 1, \ldots, i-1\}$. $w_i = 1$ indicates that the input is moved to the front, forming part of the **target**, while $w_i = 0$ indicates the input is moved to the back, forming part of the **tiling**. Weaves are symmetric, and therefore we have access to an inverse weave $\hat{\Omega}_w : I \to I$ so that $\hat{\Omega}_w \fatsemi \Omega_w = Id_I$.*

A rearrangement in the axis-stride category **St** generated by a weave over a family of axes $(A_i)_{i \in I}$ provides us with the affine morphism $[\Omega_w]_{(A_i)_{i \in I}} : \Pi_{i \in I} A_i \to \Pi_{i \in I} A_{\Omega_w(i)}$. This can be used to generate a reindexing $[a, [\Omega_w]_{(A_i)_{i \in I}}] : [a, \Pi_{i \in I} A_{\Omega_w(i)}] \to [a, \Pi_{i \in I} A_i]$ in the array-broadcasted category **Br**, mapping from an organized target-tiling arrangement to a mixed, weaved arrangement. We can further specify this using an I-family of axes $(B_i)_{i \in \Sigma_{i \in I} w_i} \times (P_i)_{i \in \Sigma_{i \in I} \neg w_i}$, indicating the target and tiled axes separately, and use the notation $[a, [\Omega_w]_{B \times P}]$. We diagram the weaved rearrangement by the diagram in Figure 7, and see that indexes placed along the tiled axes get shifted to the top.

We can now move onto defining a complex broadcasted operation. Mathematically, a complex broadcasted operation from Definition 13 involves weaves surrounding a core batch lifted morphism. This definition is more intuitively realized by the diagrams in Figures 8 and 9, where we see how slices along output tilings

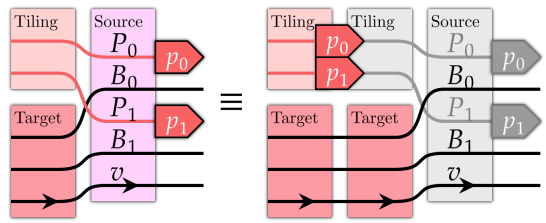


Figure 7: Definition 12 describes a remapping which arranges targets to the front of an expression (bottom in the diagram). In this figure, we show the reindexing $[\Omega_w]_{(A_i)_{i\in I}} : \Pi_{i\in I} A_i \to \Pi_{i\in I} A_{\Omega_w(i)}$ and how it relates slices along the output target axes to slices along the target input axes.

are weaved around the base operation, through reindexings, and through reversed input weaves to define sophisticated relationships between output and input slices.

In code, however, we want to bake weaves into the operation to distinguish between types of broadcasts which require different memory access patterns. The weaves are entities which supply the target datatype and axes, and tag the tiled axes. The degree and input tiled sizes are drawn from the reindexings. The base operation is provided by a metadata tag which describes a polymorphic function between the target arrays. This arrangement is shown in the diagram of Figure 8.

Definition 13 (Broadcasted Operation)**.** *To construct a broadcast, we require;*

- *A **base operation** $f : \Pi_{i\in I}[a_i, A_i] \to \Pi_{j\in J}[b_j, B_j]$ in the array-broadcasted category* **Br**. *The flat length of A_i is $L[A_i]$, and of B_j is $L[B_j]$.*
- *An I-family of stride morphisms $(\eta_i)_{i\in I}$ called **reindexings** from* **St** *with shape $\eta_i : P \to Q_i$. The shape $P \in Ob$**St** *is the **degree** of the broadcast.*
- *An I-family of **input weaves** $(s_i)_{i\in I}$ where the length of each s_i is $L[A_i] + L[Q_i]$ and where s_i has $L[A_i]$ weaved segments, $\Sigma_{\ell\in L[s_i]} s_{i\ell} = L[A_i]$.*
- *A J-family of **output weaves** $(t_j)_{j\in J}$ where the length of each t_j is $L[B_j] + L[P]$ and where t_j has $L[B_j]$ weaved segments, $\Sigma_{\ell\in L[t_j]} t_{j\ell} = L[B_j]$.*

The shape of the broadcasted operation F is given by;

$$F : \Pi_{i\in I}\left[a_i, dom([\Omega_{si}]_{A_i\otimes Q_i})\right] \to \Pi_{j\in J}\left[b_j, dom([\Omega_{tj}]_{B_j\otimes P})\right]$$

And is defined by;

$$F = (\prod_{i\in I}[a_i, [\hat{\Omega}_{si}]] \fatsemi [a_i, A_i \otimes \eta_i]) \fatsemi [f, P] \fatsemi (\prod_{j\in J}[b_j, [\Omega_{tj}]])$$

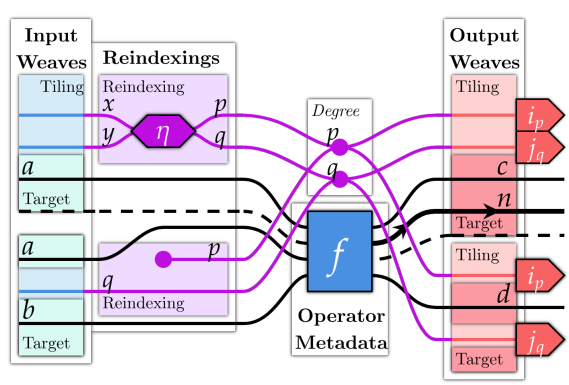


Figure 8: An example of a broadcasted operation equipped with all necessary metadata for an implementation. We provide information for the location and target of input and output weaves. As in the first output weave, the weave constructs provide information regarding the target datatype, which is drawn if it is not the default datatype such as $\mathbb{R}$. The reindexings for each input weave are provided, and derive the degree of the expression. The operator is provided by metadata indicating a polymorphic operation.

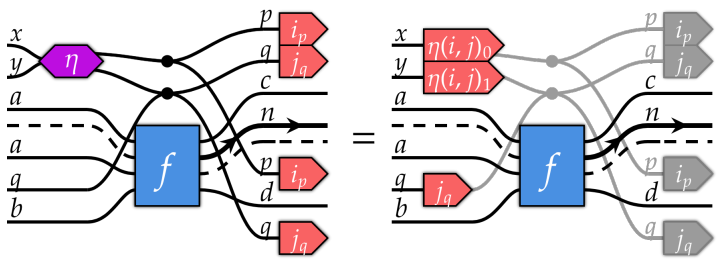


Figure 9: The broadcasted operation is functionally defined so that, for each index $\mathbf{i}_P \in P$ of the degree P, the corresponding slices placed on the output tilings corresponds to the underlying function acting on slices determined by each input's reindexing operations. In the form above, reindexings with deleted degree axes (via a rearrangement, as in the p axis of the second input) have that axis simply not drawn.

4.2 Key Operations

To maximize the utility of diagrams, we assign pictograms for operations. This allows for diagrams to be understood at a glance, and in some cases captures the "meaning" of operations.

4.2.1 Rearrangement Broadcasts

Though Mo**St** allows us to support arbitrary affine reindexings, usually operations will be broadcasted by simple rearrangements of axes. This covers the case of batched, row-wise, and column-wise operations. It is noteworthy that an advantage of *Neural Circuit Diagrams* is the lack of need to think in terms of rows- or columns- when dealing with multidimensional constructs where these concepts lose meaning. For rearrangement broadcasts, we can draw a broadcasted operation without hexagons, and merely need to show axes becoming rearranged. For example, SoftMax, pictogrammed as an expanding triangle, over the second-last dimension can be shown as in Figure 10.

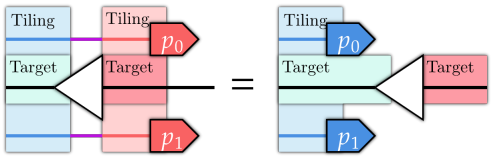


Figure 10: SoftMax over the second-last dimension of an array can be shown with rearranging wires. The weaving of the operation indicates that it supplies $F(\mathbf{x})[p_0, :, p_1] = \mathrm{SoftMax}(\mathbf{x}[p_0, :, p_1])$.

Rearrangement broadcasts may have more complex forms. They may indicate diagonalization along some axes, whereby $\mathbf{y}[p, :] = \mathbf{x}[p, p, :]$ using $\eta(p) = (p, p)$, or repetition whereby $\mathbf{y}[p, :] = \mathbf{x}[:]$ using $\eta(p) = ()$ ie $\eta : P \to \mathbb{1}$ is deletion. These forms are diagrammed in Figures 11 and 12.

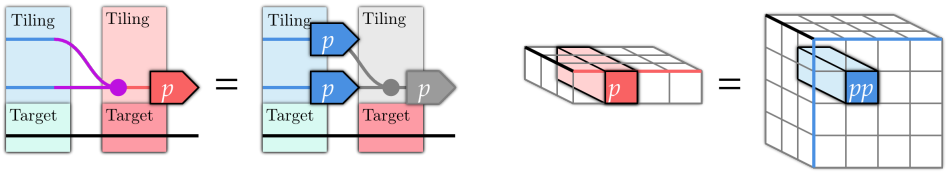


Figure 11: Diagonalization corresponds to the equation $\mathbf{y}[p, :] = \mathbf{x}[p, p, :]$. This can be expressed using the rearrangement reindexing $p \mapsto (p, p)$.

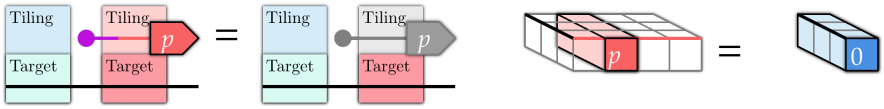


Figure 12: Repetition corresponds to the equation $\mathbf{y}[p, :] = \mathbf{x}[:]$. This can be expressed using the rearrangement reindexing $p \mapsto ()$.

4.2.2 Also Operations

Einstein operations are those which can be readily shown with the Einstein summation convention, and include transposes, summations, outer products, and inner products. We use two shortcuts to diagram these operations. The termination of a dashed product line automatically indicates multiplication, and thereby yields an outer product, as shown in Figure 13. Summation is indicated by a terminating wire or, in the case where it is immediately preceded by a diagonalization, a curved cup. The curved cup mimics Penrose graphical notation and provides visual distinctiveness to the critical dot product operation. This is shown in Figure 14

Figure 13: Multiplication can be expresed by having a dashed wire come to an end. The underlying operation is $(\cdot) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and therefore the target arrays $[\mathbb{R}, \mathbb{1}]$ have no axes. Here, with degree $P = P_0 P_1 P_2$, our reindexings are the rearrangements $p_0 p_1 p_2 \mapsto p_0 p_1$ and $p_0 p_1 p_2 \mapsto p_2$ respectively. This means the operation provides $\mathbf{z}[p_0, p_1, p_2] = \mathbf{x}[p_0, p_1] \cdot \mathbf{y}[p_2]$, or an **outer product**.

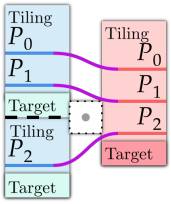
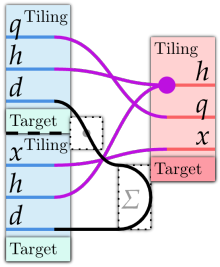


Figure 14: Weaved multiplication followed by summation yields an Einstein operation. Here, we show the operation `'q h d, x h d → h q x'` which forms the query-key multiplication of multi-head attention. Note how the parallel calculation of heads is expressed by broadcasting.



4.2.3 Learned Operations

Learned operations use stored weights to learn patterns within data. They can be categorically described using the **Para** construct. To indicate that an operation has a learned component, we bold some aspect of it. Diagramming linear layers and norming operations is shown in Figures 15 and 16. Note that our framework allows multidimensional learned layers to be clearly expressed, in contrast to PyTorch where multidimensional linear operations often require exogenous stride manipulation to properly manage.

Figure 15: A learned linear operation can be expressed by a chipped rectangle labeled **L**. The number and size of input/output axes is appropriately displayed according to the broadcasting framework. Here, we show the shape of the final heads aggregator layer of multi-head attention, with $\overline{x}$-indicating the number of tokens.

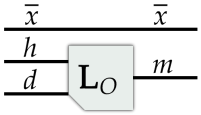
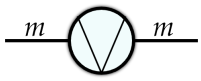


Figure 16: Norms are expressed as a bold (*learned*) circle with the inner space being occupied by a pictogram. The circle aims to be evocative of the idea of fitting values along a certain size. The "V" indicates RMSNorm.



Embeddings are noteworthy in that they are operations $\mathbf{E} : V \to [\mathbb{R}, m]$, where $V \in \mathbb{N}$ is the size of the vocabulary and m is the token size. Therefore, the underlying datatype of their input array is an integer, and not a continuous value. Given a V-sized datatype and a V-sized axis, we override the cupping notation to indicate the selection operation $V \times [_, V] \to _$. This allows us to show the internals of embedding as a selection operation in Figure 17, and provides the infrastructure to later deal with Mixture-of-Expert gating selection.

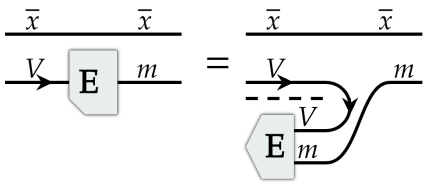


Figure 17: The underlying operation of an embedding has shape $V \to [\mathbb{R}, m]$, where $V \in \mathbb{N}$. If we have made the real datatype implicit, then we need to explicitly label V with an arrow. We can use selection to express the internals of embedding.

4.3 Convolution

Under our framework, convolution can be defined by a reindexing followed by a learned linear layer. Convolution is defined as $\mathbf{y}[i_{\overline{x}'}, :] = \Sigma_{j_k \in k} \mathbf{L}[j_k] \cdot \mathbf{x}[i_{\overline{x}'} + j_k, :]$. This can be split into a convolution operation which first maps $(*\mathbf{x})[i_{\overline{x}'}, j_k, \ell_c] = \mathbf{x}[i_{\overline{x}'} + j_k, \ell_c]$ followed by a kc to c' linear layer $\mathbf{L}$. The convolution operator can be represented by a reindexing using addition, $+ : \overline{x}'k \to \overline{x}$. Because the convolution matrix can be supplied by a reindexing, it is computationally distinct from the general class of all matrices, and this distinction is clearly shown by our framework. Convolution expressed in our framework is shown in Figure 18.

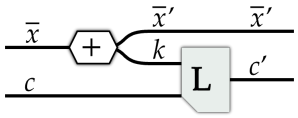


Figure 18: Convolution can be expressed as a two-step process of an addition reindexing (*convolution matrix*) followed by a learned linear layer.

A noteworthy benefit of our framework is the ability to reason about models. In the case of convolution, we are able to observe the translational equivariance by propagating indexes through the $\overline{x}'$ axis. We can define translation as a reindexing which shifts an element i to $i + t$. We can "slide" this translation through the expression using the broadcasting rules and the fact that $(+)(i + t, k) = (+)(i, k) + t$. This sliding is derived from the Yoneda sliding, elaborated in Appendix A.4. This is shown in Figure 19, revealing the translational equivariance of convolution across the $\overline{x}/\overline{x}'$ axis.

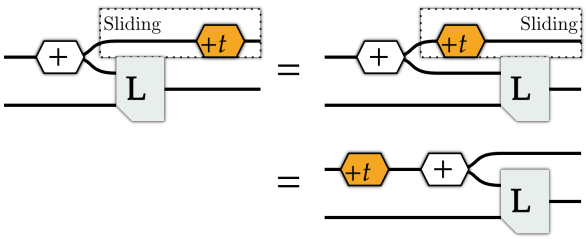


Figure 19: The equivariance of convolution can be shown diagrammatically by sliding an index-wise translation over the operation.

5 Results

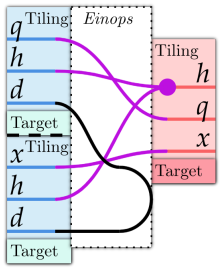
The contribution of this paper is a formal representational layer for deep learning models rather than a new benchmark architecture or a new optimized kernel. Accordingly, the relevant notion of "results" is constructive rather than benchmark-driven. The question is whether the categorical objects introduced in Sections 2–4.1 are operational: can the same formal term support symbolic model construction, deferred configuration, executable lowering, rewrite-oriented transformation, serialization across implementations, and automatic visualization? This section answers that question affirmatively through mirrored implementations in Python and TypeScript.

The two implementations play complementary roles. The Python backend supports machine-facing uses of the formalism, including algebraic construction, configuration, compilation, and graph conversion. The TypeScript backend supports human-facing uses by rendering the same serialized terms as diagrams. Together, they test the central claim of the paper: that the framework provides a shared representation from which both executable and diagrammatic views of a model can be derived. This section is accompanied by a results notebook in Appendix B, which provides corresponding code and diagrams.

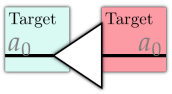
5.1 Python

5.1.1 Algebraic construction via autoalignment

Python is uniquely suited to provide algebraic manipulation and integration with existing deep learning packages such as PyTorch. Python provides support for functional data structures via `@dataclass(frozen=True)`, and has extensive operator overloading. This allows for components to be constituted into a full model via algebraic operations with backend processing. Following the construction rules of Section 2, we implement axes with UIDs in expressions. When we compose expressions with '@' overloading, we can assign aligned axes to be the same. When the number of axes in components mismatch, we perform the necessary batch broadcast. When the number of tuple segments in components mismatch, we add identity morphisms towards the bottom of expressions. This allows operations such as attention to be cleanly, algebraically expressed while constructing a full expression with all broadcasted explicitly expressed at the backend. This overcomes the challenge of ensuring that the size of tensors throughout an expression match.

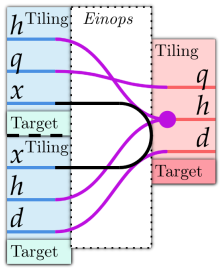



```
qk_matmul = ops.Einops
  .template(
    'q h d, x h d -> h q d'
    )
```

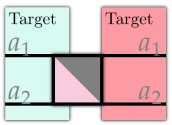



```
softmax = ops.SoftMax.template()
```

As we did not provide the axes, an unnamed axis is generated. Autoalignment will then rename it.




```
sv_matmul = ops.Einops
  .template(
    'h q x, x h d -> q h d'
    )
```




```
mask = ops.
  WeightedTriangularLower
  .template()
```

5.1.2 Configuration Generation

Constructed terms – such as the one expressed in Figure 20 – have free terms associated with UIDs. Above, we showed how axes can be automatically aligned during composition. The remaining UIDs reveals the overall degrees of freedom for configuration, and an expression can be scanned to find these terms. From the list of UIDs, we generate an assignment to set axes to desired sizes. In Figure 21, we provide the example of setting the axis sizes of multi-head attention placed inside a ResNet He et al. (2015), making the expression ready for constructing a PyTorch model.

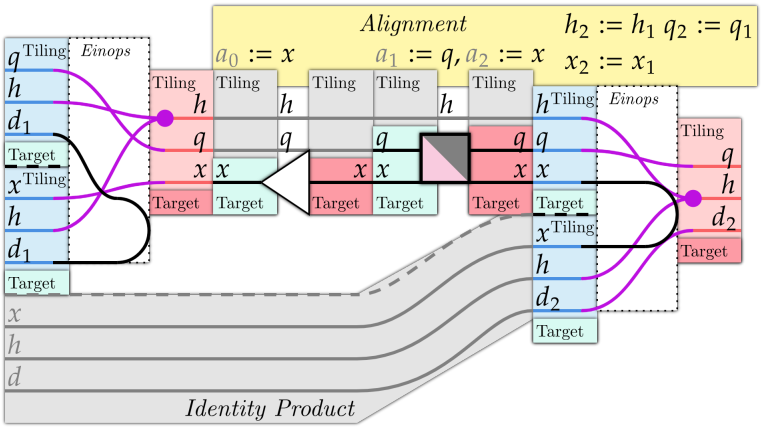


Figure 20: Applying `qk_matmul @ softmax @ mask @ sv_matmul`, we perform autoalignment operations at each step. Axes are aligned to be the same. Operations are batch broadcasted when the number of axes mismatch. In the case of `sv_matmul`, we take the product with an identity rearrangement with the array $[\mathbb{R}, xhd]$. Note that the h, q, x axes of `qk_matmul` and `sv_matmul` are separately generated, meaning their equivalence is only realized after composition. The d_1 and d_2 axes are never aligned together, meaning that the final configuration of the expression takes $\langle q, h, x, d_1, d_2 \rangle$.

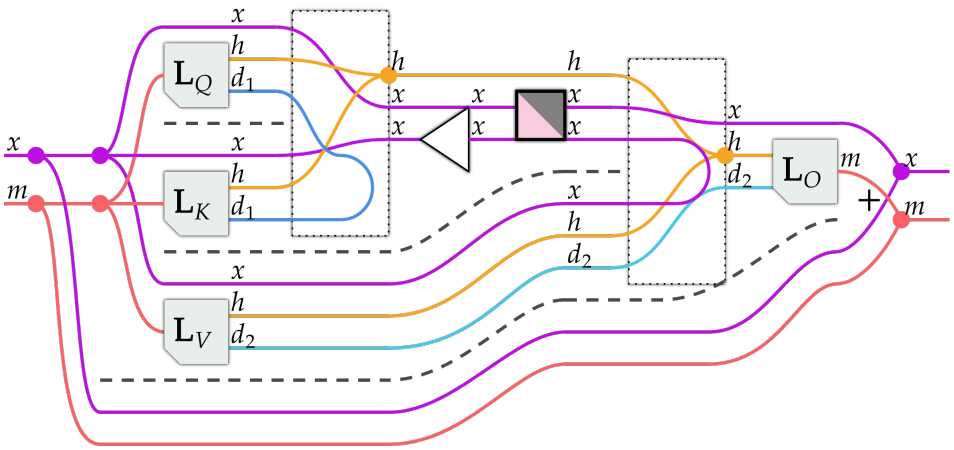


Figure 21: A full ResNet attention block expression constructed through autoalignment has a number of numeric sizes with UIDs. By assigning values to these, we have a configured expression ready for PyTorch compilation.

5.1.3 Compositional compilation to PyTorch

Algebraically defined models indicate execution instructions from start to end and, therefore, we have sufficient information to supply a corresponding PyTorch module. This can be done in a compositional manner by converting `Composed` constructs to sequential PyTorch modules, and `ProductOfMorphisms` to parallel PyTorch modules. Root morphisms in the array-broadcasted category **Br** contain broadcasting information and execution metadata, which can be converted into PyTorch modules. Due to PyTorch's elaborate broadcasting semantics, this conversion requires some infrastructure. The end result of this process allows us to take a configured constructed term, as above, and generate a runnable PyTorch module. Compilation to PyTorch is an incidental feature of the constructed terms. No aspect of the algebraic framework is explicitly defined for PyTorch. Compilation to any other framework – TensorFlow, Triton, or others – would all be done in a similar manner. Algebraic terms serve as a universal framework for compilation.

5.1.4 Algebraic Manipulation with Hypergraphs

A key feature of algebraic terms is algebraic manipulation, the ability to define general algebraic rules and apply these to models. The composed-product approach we outline above is tailored to constructing and diagramming terms. However, bifunctoriality and symmetry (see Section 3) yield redundancy. Certain algebraic properties are better captured by hypergraphs (Piedeleu & Zanasi, 2025) which discard immaterial information about morphisms' location among composed, product, and rearrangement structures. We can define an algorithm to convert from a "ProdCategory[L,M]" to a "Hypergraph[L,M]", defining the conversion from mathematical first-principles and therefore having it applicable to any category, including the axis-stride **St** or array-broadcasted **Br** categories. A hypergraph form of Figure 21 is shown in Figure 22.

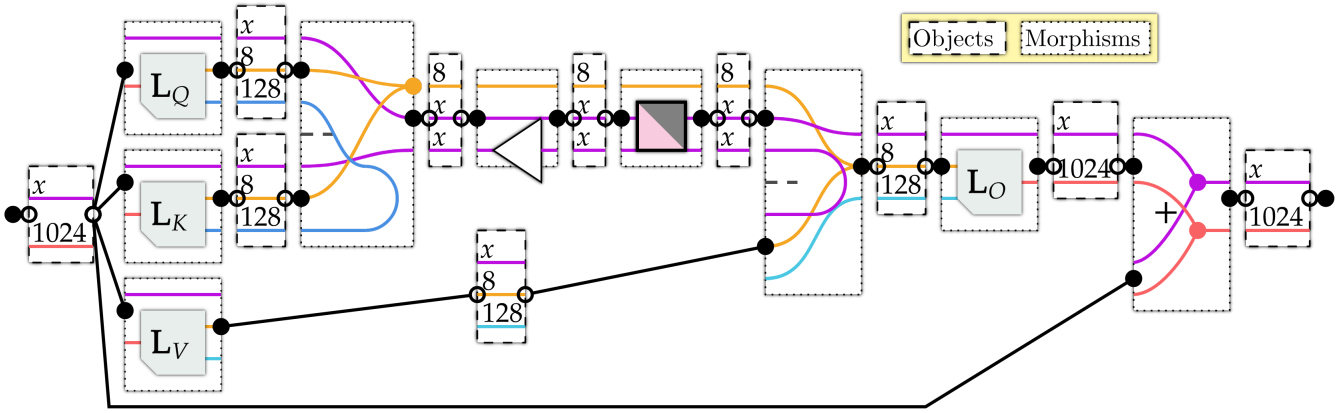


Figure 22: By converting from a "ProdCategory[L,M]" to a "Hypergraph[L,M]", we can perform algebraic manipulation with hypergraph rewrite rules. This allows us to apply general algebraic properties such as associativity, bifunctoriality, and symmetry, which are not easily captured by the composed-product approach.

5.2 Interoperability

The constructed term framework means expressions are functional data classes without reference to an external state. These can be packaged into JSON files and sent to mirrored implementations. We assert that terms with the same UID are equivalent in all ways, and therefore the JSON packaging can be compressed by having an accompanying UID repository. The JSON file can be loaded by a framework with a mirrored implementation. This interoperability supports the "framework agnostic" philosophy of formalized deep learning models, and allows us to leverage various frameworks for their specific utility, such as PyTorch via Python and diagrams via TypeScript.

5.3 Diagram Generation

Within TypeScript, we can implement the diagramming procedure. We convert `Composed` constructs into horizontally placed blocks, and `Product` constructs into vertically placed blocks. This rendering is done by realizing objects as a series of anchors which sequential expressions link together, matching the manual diagramming process. We supply a general framework for rendering categories in general, and then specific instructions for the objects and morphisms of specific categories. In this manner, category theory's templating approach to mathematical structure allows code to be reused in different contexts. An example of a diagram for a basic transformer model is shown in Figure 23.

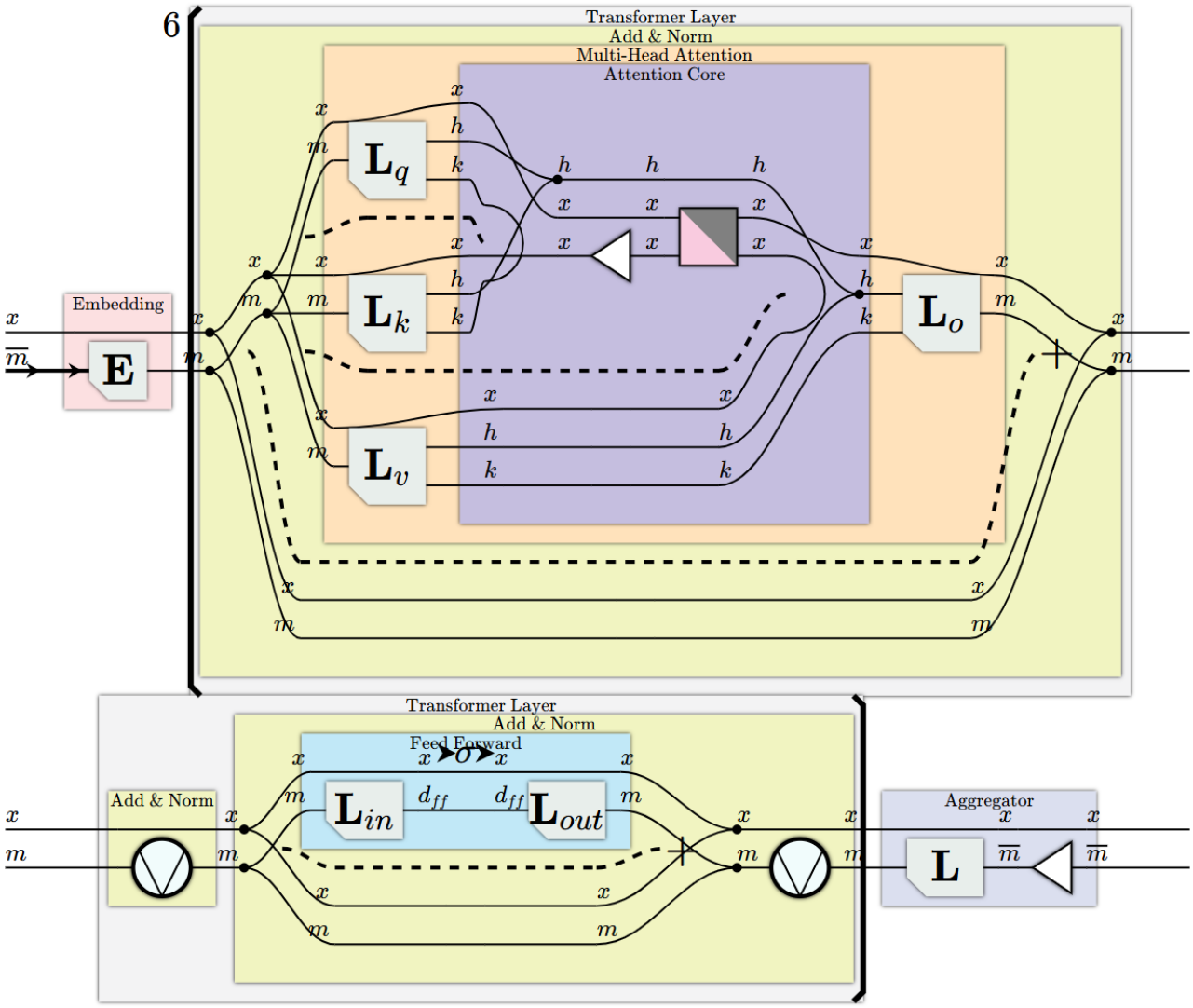


Figure 23: This diagram is generated by the TypeScript implementation of constructed terms. The constructed term is transferred via JSON and WebSockets.

6 Future Work and Conclusion

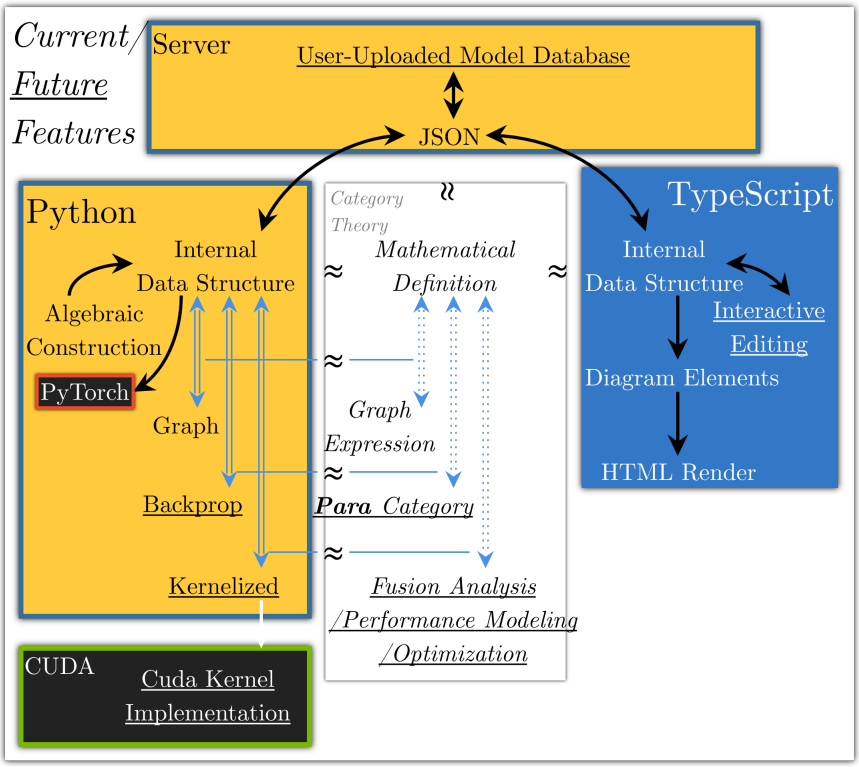


Figure 24: The modularity of the framework allows for a web of features to be developed and integrated. At any point, new features can be "hooked" into the system, allowing for a powerful and extensible framework for future AI research.

The formal mathematical framework established above and implementation following the construction rules paradigm allows us to process deep learning models algebraically. The dependency relations of implementations only have to follow the dependency relations of underlying mathematical definitions, allowing for modular code. This allows new features to be readily added.

The most impactful novel feature would be automatic low-level kernel derivation. This follows from the techniques outlined in ***F**lash**A**ttention on a **N**apkin (FAN)* Abbott & Zardini (2025). That work would go beyond the scope of this paper, which focuses on the foundations of a formal approach. FAN is uniquely positioned to derive critical low-level optimizations which are inaccessible to standard compilations found in PyTorch. Tiled matrix multiplication and attention can be derived from first-principles. Furthermore, FAN allows for hardware-aware performance models, which guide improved future design. By integrating model analysis into mathematics, PyTorch, and pollable tools, AI-accelerated development of future algorithms will be within reach.

More broadly, various analyses dependent on compositionality will be possible within the categorical framework. The **Para** approach allows for backpropagated algorithms to be derived algebraically and piecewise. This offers an alternative to PyTorch's backpropagation tools, circumventing the need for its infrastructure almost entirely allowing for a true universal model of deep learning architectures which can be directly optimized and compiled into low-level code. Additionally, we can develop novel compositional analyses. A subject of particular interest is quantization error composition. Computational costs are linearly dependent on quantization size, while bandwidth costs – often the more pressing concern for models – are superlinearly dependent. Determining where to utilize lower quantizations is a byproduct of how effectively random

rounding errors will propagate. This is a compositional property, and therefore category theory's tools are of particular interest.

The diagramming tool can be developed into a suite of tools which allow for models to be developed and reasoned about diagrammatically, then shared to other uses through the natural JSON encoding resulting from the term construction system. Such a toolset will allow models to be intuitively manipulated and shared, enhancing research productivity.

This collection of modularity, mathematics, and tools leads to a "web" of features which can be extended into the future. At any point we can "hook" into the system (see Figure 24), creating a powerful framework for future AI research.

Finally, we can integrate the categorical and optimization tools of this framework into categorical codesign Zardini (2023) to optimize various stages of the artificial intelligence stack together. Models generate performance models, indicating the available memory-bandwidth-compute requirements for specified levels of performance, and this requirements-functionality relationship can be fit into the requirements-functionality relationships of hardware, power supply, and other components. This allows for a holistic approach to AI development, where the design of models, algorithms, and hardware are co-optimized.

Overall, the mathematical procedures, formal descriptions, algebraic implementation, and automated diagramming provided in this work lay the foundation for a robust, formalized, and systematic approach to the design of deep learning algorithms. This addresses a key challenge in the deep learning community, and opens the possibility of using artificial intelligence to design itself.

References

Vincent Abbott. Neural Circuit Diagrams: Robust Diagrams for the Communication, Implementation, and Analysis of Deep Learning Architectures. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=RyZB4qXEgt.

Vincent Abbott and Gioele Zardini. FlashAttention on a Napkin: A Diagrammatic Approach to Deep Learning IO-Awareness. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=pF2ukh7HxA.

Vincent Abbott, Kotaro Kamiya, Gerard Glowacki, et al. Accelerating Machine Learning Systems via Category Theory: Applications to Spherical Attention for Gene Regulatory Networks. In *Artificial General Intelligence: 18th International Conference, AGI 2025, Reykjavic, Iceland, August 10–13, 2025, Proceedings, Part I*, pp. 1–11, Berlin, Heidelberg, 2025. Springer-Verlag. ISBN 978-3-032-00685-1. doi: 10.1007/978-3-032-00686-8_1. URL https://doi.org/10.1007/978-3-032-00686-8_1.

Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. *arXiv preprint arXiv:2104.13478*, 2021.

Andrea Censi, Jonathan Lorand, and Gioele Zardini. *Applied Compositional Thinking for Engineering*. 2024. URL https://bit.ly/3qQNrdR. work-in-progress book.

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

Geoffrey S. H. Cruttwell, Bruno Gavranović, Neil Ghani, et al. Categorical Foundations of Gradient-Based Learning. In *European Symposium on Programming*, pp. 1–28. Springer, 2022.

Geoffrey S. H. Cruttwell, Bruno Gavranović, Neil Ghani, et al. Deep Learning with Parametric Lenses. *arXiv preprint arXiv:2404.00408*, 2024.

Tri Dao, Dan Fu, Stefano Ermon, et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Brendan Fong, David I. Spivak, and Rémy Tuyéras. Backprop as Functor: A compositional perspective on supervised learning. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 2019.

Thomas Fox. Coalgebras and cartesian categories. *Communications in Algebra*, 4(7):665–667, 1976. doi: 10.1080/00927877608822127. URL https://doi.org/10.1080/00927877608822127.

Tobias Fritz, Tomáš Gonda, Paolo Perrone, and Eigil Fjeldgren Rischel. Representable Markov categories and comparison of statistical experiments in categorical probability. *Theoretical Computer Science*, 961:113896, 2023. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2023.113896. URL https://www.sciencedirect.com/science/article/pii/S0304397523002098.

Bruno Gavranović. *Fundamental Components of Deep Learning: A Category-Theoretic Approach*. PhD thesis, University of Strathclyde, 2024.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, 2015.

Mary Phuong and Marcus Hutter. Formal Algorithms for Transformers. *arXiv preprint arXiv:2207.09238*, 2022.

Robin Piedeleu and Fabio Zanasi. *An Introduction to String Diagrams for Computer Scientists*. Cambridge University Press, 2025. ISBN 9781009625708.

Dan Shiebler, Bruno Gavranović, and Paul Wilson. Category Theory in Machine Learning. In *Applied Category Theory*, 2021.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, et al. Graph Attention Networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJXMpikCZ.

Gioele Zardini. *Co-Design of Complex Systems: From Autonomy to Future Mobility Systems*. PhD thesis, ETH Zurich, 2023.

A Appendix

A.1 Fox's Theorem

Fox's theorem relates the naturality of a product category to the algebraic properties and degrees of freedom of its morphisms. The classical result from Fox (1976) relates Cartesian to monoidal categories. We split the result into two sections, relating it to the properties of copying (unique identification) and deletion (free construction).

Theorem 1. *[Fox's Theorem] We split Fox's theorem into two parts, relating naturality properties to aspects of Cartesian products. For a copy-discard category where all rearrangements are allowed we have;*

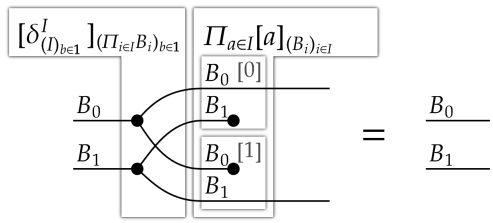
- ***Unique Identification*** *If copying (count increases) are natural, then for morphisms $f : A \to \Pi_{i\in I} B_i$ and $g : A \to \Pi_{i\in I} B_i$ if for all remappings $i : \mathbf{1} \to i$ we have $f \fatsemi [i] = g \fatsemi [i]$, then $f = g$. This supplies an injective map $\mathcal{C}(A, \Pi_{i\in I} B_i) \to \Pi_{i\in I}\mathcal{C}(A, B_i)$.*
- ***Free Construction*** *If deletion (count decreases) are natural, then for an I-family of morphisms $(f_i)_{i\in I}$ where $f_i : A \to B_i$, there exists a morphism $F : A \to \Pi_{i\in I} B_i$ so that for remappings $i : \mathbf{1} \to i$ we have $F \fatsemi [i] = f_i$. This supplies an injective map $\Pi_{i\in I}\mathcal{C}(A, B_i) \to \mathcal{C}(A, \Pi_{i\in I} B_i)$.*

Together, these properties describe a Cartesian product and indicate a bijective map $\mathcal{C}(A, \Pi_{i\in I} B_i) \sim \Pi_{i\in I}\mathcal{C}(A, B_i)$.

For the proof, the high-level idea is to first construct an identity using a copy followed by selective projections, given by;

$$[\delta^I]_{(\Pi_{i\in I}B_i)_{b\in\mathbf{1}}} \fatsemi \Pi_{a\in I}[a]_{(B_i)_{i\in I}} = \mathrm{Id}[\Pi_{i\in I}B_i] \tag{4}$$

The case of $I = \mathbf{2}$ can be diagrammed by;



As a morphism followed by an identity is itself, applying this construct provides a bijective map on morphisms.

Proof. We will show that;

$$[\delta^I]_{(\Pi_{i\in I}B_i)_{b\in\mathbf{1}}} \fatsemi \prod_{a\in I}[a]_{(B_i)_{i\in I}} = \mathrm{Id}[\Pi_{i\in I}B_i]$$

We have our rearrangements supplied by;

$$\begin{aligned} \delta^I &: I \to \mathbf{1} & [\delta^I]_{\Pi_{i\in I}B_i} &: \Pi_{i\in I}B_i \to \Pi_{a\in I}\Pi_{i\in I}B_i \\ \langle a| &: \mathbf{1} \to I & [a]_{(B_i)_{i\in I}} &: \Pi_{i\in I}B_i \to B_a \\ & & \prod_{a\in I}[a]_{(B_i)_{i\in I}} &: \Pi_{a\in I}\Pi_{i\in I}B_i \to \Pi_{a\in I}B_a \end{aligned}$$

The product and composition exclusively of rearrangements is defined by their underlying remapping, and therefore is the same in all product categories where the rearrangements are allowed. This allows us to use product naturality rules. Acting on this expression with an element $\langle c| : \mathbf{1} \to I$ as a projection $[c]_{(B_a)_{a\in I}} : \Pi_{a\in I}B_a \to B_c$ we get;

$$\begin{aligned} [\delta^I]_{\Pi_{i\in I}B_i} \fatsemi \underbrace{\left(\prod_{a\in I}[a]_{(B_i)_{i\in I}}\right) \fatsemi [c]_{(B_a)_{a\in I}}}_{\text{Extract } a=c} &= \underbrace{[\delta^I]_{\Pi_{i\in I}B_i} \fatsemi [c]_{(\Pi_{i\in I}B_i)_{a\in I}}}_{\text{Compatible}} \fatsemi [c]_{(B_i)_{i\in I}} \\ &= [c \fatsemi \delta^I]_{\Pi_{i\in I}B_i} \fatsemi [c]_{(B_i)_{i\in I}} \\ &= [\mathrm{Id}_{\mathbf{1}}]_{\Pi_{i\in I}B_i} \fatsemi [c]_{(B_i)_{i\in I}} \\ &= [c]_{(B_i)_{i\in I}} \end{aligned}$$

Therefore, the rearrangement $[\delta^I]_{(\Pi_{i\in I}B_i)_{b\in\mathbf{1}}} \fatsemi \Pi_{a\in I}[a]_{(B_i)_{i\in I}}$ has a remapping which maps each $\langle c| : \mathbf{1} \to I$ to $\langle c| : \mathbf{1} \to I$, and is therefore the identity.

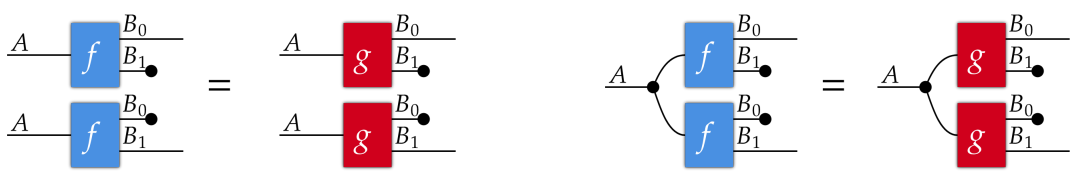
□

We can now prove unique identification. We will correspond equations with diagrams for the case of $I = 2$.

Proof. We take two morphisms $f, g : A \to \Pi_{i \in I} B_i$ where for all elements $\langle a| : \mathbf{1} \to I$ we have $f \fatsemi [a]_{(B_i)_{i \in I}} = g \fatsemi [a]_{(B_i)_{i \in I}}$. Therefore, we have;

$$\left(\prod_{a \in I} f \fatsemi [a]_{(B_i)_{i \in I}} \right) = \left(\prod_{a \in I} g \fatsemi [a]_{(B_i)_{i \in I}} \right)$$
$$[\delta^I]_A \fatsemi \prod_{a \in I} f \fatsemi \prod_{a \in I} [a]_{(B_i)_{i \in I}} = [\delta^I]_A \fatsemi \prod_{a \in I} g \fatsemi \prod_{a \in I} [a]_{(B_i)_{i \in I}}$$

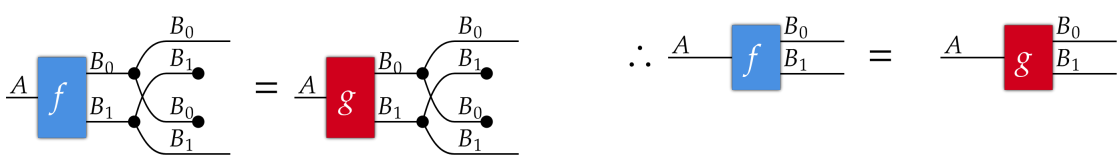
This corresponds to the diagrams;



If count increases are natural, then we apply;

$$f \fatsemi [\delta^I]_{\Pi_{i \in I} B_i} \fatsemi \prod_{a \in I} [a]_{(B_i)_{i \in I}} = g \fatsemi [\delta^I]_{\Pi_{i \in I} B_i} \fatsemi \prod_{a \in I} [a]_{(B_i)_{i \in I}}$$
$$f \fatsemi \mathrm{Id}[\Pi_{i \in I} B_i] = g \fatsemi \mathrm{Id}[\Pi_{i \in I} B_i] \quad \text{(Eq. 4)}$$
$$f = g$$

Corresponding to the diagrams;



Therefore, if for all elements $\langle a| : \mathbf{1} \to I$ we have $f \fatsemi [a]_{(B_i)_{i \in I}} = g \fatsemi [a]_{(B_i)_{i \in I}}$, in a natural copying category, then $f = g$. This provides an injective map from $\mathcal{C}(A, \Pi_{i \in I} B_i)$ to a family of morphisms $\Pi_{i \in I} \mathcal{C}(A, B_i)$.

□

Finally, we derive free construction from natural deletion / count decreases.

Proof. Given a family of morphisms $(f_i)_{i \in I}$ where $f_i : A \to B_i$, we define;

$$F = [\delta^I]_A \fatsemi \left(\prod_{a \in I} f_a \right)$$

If count decreases (deletion) is natural, then we can apply a rearrangement generated by an element $\langle c| : \mathbf{1} \to I$ to get;

$$\begin{aligned} F \fatsemi [c]_{(B_a)_{a\in I}} &= [\delta^I]_A \fatsemi \left(\prod_{a\in I} f_a\right) \fatsemi [c]_{(B_a)_{a\in I}} \\ &= [\delta^I]_A \fatsemi [c]_{(A)_{a\in I}} \fatsemi f_c \\ &= [\mathrm{Id}_{\mathbf{1}}]_A \fatsemi f_c \\ &= f_c \end{aligned}$$

Therefore, we can construct a morphism $F : A \to \Pi_{i\in I} B_i$ so that $F \fatsemi [c]_{(B_a)_{a\in I}} = f_c$ for all $\langle c| : \mathbf{1} \to I$. This provides an injective map from families of morphisms $\Pi_{i\in I}\mathcal{C}(A, B_i)$ to $\mathcal{C}(A, \Pi_{i\in I} B_i)$.

□

A.2 Remapping Algebra

Parallelism with respect to discrete functions is offered by the **direct sum**, which concatenates maps $\mu : J \to I$ and $\nu : K \to L$ into a map $\mu \oplus \nu : J \oplus K \to L \oplus I$. This satisfies the conditions of a monoidal product, though our lone objects approach would require objects n to be expressed as $\oplus_{i\in n}\mathbf{1}$ to not have redundancy in objects. If we express discrete functions as tuples $\mu \sim I^J$, then the discrete sum is given by tuple concatenation. Direct sums of remappings gives the product of rearrangements. This can be provided by Definition 5, or follow from the elemental category element properties of Definition 6. The direct sum of discrete functions is shown in Definition 14, diagrammed in Figure 25, along with correspondence to rearrangements.

Definition 14. *[Direct Sum of Discrete Functions] Given a family of discrete functions $(\mu_i)_{i\in I}$ where $\mu_i : P_i \to Q_i$, the **direct sum**;*

$$(\oplus_{i\in I}\mu_i) : \Sigma_{i\in I} P_i \to \Sigma_{i\in I} Q_i \tag{5}$$

Is defined so that for $s \in I$ (segment choice) and $\ell \in P_s$ (offset) we have;

$$(\oplus_{i\in I}\mu_i)(\ell + \Sigma_{i\in s} P_i) = \mu_s(\ell) + \Sigma_{i\in s} Q_i \tag{6}$$

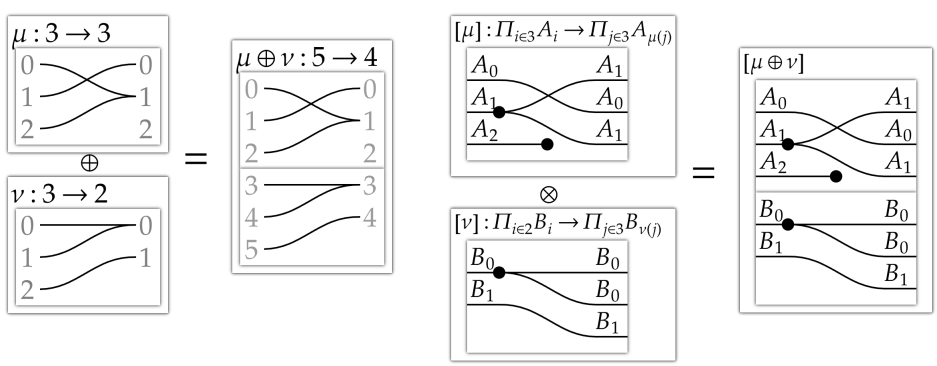


Figure 25: The direct sum of discrete functions concatenates their mappings, and is given by offsetting latter functions. This corresponds to the product of rearrangements.

Naturality applies to products of morphisms in general. For $f_i : A_i \to B_i$ where naturality provides $(\Pi_{i\in I} f_i) \fatsemi [\mu]_{(B_i)_{i\in I}} = [\mu]_{(A_i)_{i\in I}} \fatsemi (\Pi_{j\in J} f_{\mu(j)})$ applies just as well to B_i being singular or product objects. Therefore,

we need a mechanism for expressing this form of naturality, and we provide this by offsets provided in Definition 15.

Definition 15 (Associativity Compatibility)**.** *A remapping can be made flat. If we have a remapping $\mu : J \to I$ and target sizes $(L_i)_{i\in I}$, then we define;*

$$\mu_{(L_i)_{i\in I}} : \Sigma_{j\in J} L_{\mu(j)} \to \Sigma_{i\in I} L_i \tag{7}$$

So that for $s \in J$ (segment choice) and $\ell \in L_{\mu(s)}$ (offset) we have (using bra-notation $\langle\Box|$ to indicate elements clearly);

$$\langle \ell + \Sigma_{j\in s} L_{\mu(j)} | \fatsemi \mu_{(L_i)_{i\in I}} = \langle \ell + \Sigma_{i\in\mu(s)} L_i | \tag{8}$$

A remapping $\mu : J \to I$ over a family of flat objects $(A_i)_{i\in I}$ so that $A_i = \Pi_{k\in L_i} A_{ik}$ is defined so that;

$$[\mu]_{(\Pi_{k\in L[A_i]} A_{ik})_{i\in I}} = [\mu_{(L[A_i])_{i\in I}}]_{(A_{ik})_{i\in I, k\in L[A_i]}}$$

Naturality for an I-family of morphisms $(f_i)_{i\in I}$ where $f_i : A_i \to B_i$ (A_i and B_i are flat objects with some length) can then be more precisely expressed as;

$$\begin{aligned}(\Pi_{i\in I} f_i) \fatsemi [\mu]_{(B_i)_{i\in I}} &= [\mu]_{(A_i)_{i\in I}} \fatsemi (\Pi_{j\in J} f_{\mu(j)}) \\ (\Pi_{i\in I} f_i) \fatsemi [\mu_{(L[B_i])_{i\in I}}]_{(B_{ik})_{i\in I, k\in L_i}} &= [\mu_{(L[A_i])_{i\in I}}]_{(A_{ik})_{i\in I, k\in L_i}} \fatsemi (\Pi_{j\in J} f_{\mu(j)})\end{aligned}$$

In implementations, we will be working with flat remappings.

A.3 Deterministic Naturality

In an elemental category, deterministic morphisms which map elements to elements are universally natural, owing to the naturality of elements. Rearrangements on elements $(a_i)_{i\in I}$ so that $a_i \in \mathrm{El}(A_i)$ are defined so that;

$$\left(\prod_{i\in I} a_i\right) \fatsemi [\mu]_{(A_i)_{i\in I}} = \left(\prod_{j\in J} a_{\mu(j)}\right)$$

Consider a family of deterministic morphisms $(f_i)_{i\in I}$ so that $f_i : A_i \to B_i$. Determinism (see Definition 6) maps elements to elements, so for an element $a_i \in \mathrm{El}(A_i)$ we have $a_i \fatsemi f_i \in \mathrm{El}(B_i)$. Therefore, we have;

$$\left(\prod_{i\in I} f_i\right) \fatsemi [\mu]_{(B_i)_{i\in I}} = [\mu]_{(A_i)_{i\in I}} \fatsemi \left(\prod_{j\in J} f_{\mu(j)}\right) \tag{9}$$

As we have for the left-hand side;

$$\begin{aligned}\left(\prod_{i\in I} a_i\right) \fatsemi (LHS) &= \left(\prod_{i\in I} a_i\right) \fatsemi \left(\prod_{i\in I} f_i\right) \fatsemi [\mu]_{(B_i)_{i\in I}} \\ &= \left(\prod_{i\in I} (a_i \fatsemi f_i)\right) \fatsemi [\mu]_{(B_i)_{i\in I}} \\ &= \left(\prod_{j\in J} (a_{\mu(j)} \fatsemi f_{\mu(j)})\right)\end{aligned}$$

And the right-hand side;

$$\left(\prod_{i\in I} a_i\right) \fatsemi (RHS) = \left(\prod_{i\in I} a_i\right) \fatsemi [\mu]_{(A_i)_{i\in I}} \fatsemi \left(\prod_{j\in J} f_{\mu(j)}\right)$$
$$= \left(\prod_{j\in J} a_{\mu(j)}\right) \fatsemi \left(\prod_{j\in J} f_{\mu(j)}\right) = \left(\prod_{j\in J} a_{\mu(j)} \fatsemi f_{\mu(j)}\right)$$

Therefore, the left- and right-hand sides are equal as they have the same action on elements, and deterministic morphisms are natural over all rearrangements.

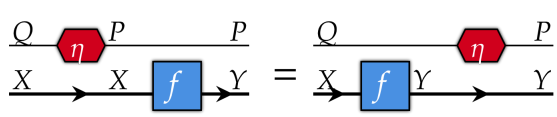
A.4 Yoneda Sliding

The Yoneda lemma describes how we can naturally transform between functors, composition preserving maps between morphisms. In our case, it corresponds to reindexing slides as seen in Figure 19. As we are not strictly enforcing the Cartesian property, we are working outside **Set** and therefore the Yoneda lemma only applies under certain cases.

Theorem 2 (Yoneda Sliding)**.** *In the array-broadcasted category* **Br***, given a morphism* $f : X \to Y \in Mo\mathbf{Br}$ *and a stride morphism* $\eta : P \to Q \in Mo\mathbf{St}$*, where either (a) the morphism is deterministic, or (b) the stride morphism provides a natural remapping, we have Yoneda sliding given by;*

$$[X, \eta] \fatsemi [f, P] = [f, Q] \fatsemi [Y, \eta] \tag{10}$$

This can be diagrammed by;



Proof. We consider the extracted case, applying $[\delta^P]_{[Y,P]} \fatsemi \Pi_{p\in \mathrm{El}(P)}[Y, p]$ to both sides. This gives us;

$$(LHS) = [X, \eta] \fatsemi [f, P] \fatsemi [\delta^P]_{[Y,P]} \fatsemi \left(\prod_{p\in \mathrm{El}(P)} [Y, p]\right)$$
$$= [X, \eta] \fatsemi [\delta^P]_{[X,P]} \fatsemi \left(\prod_{p\in \mathrm{El}(P)} [X, p] \fatsemi f\right) \quad \text{(Eq. 2)}$$
$$= [\delta^P]_{[X,Q]} \fatsemi \left(\prod_{p\in \mathrm{El}(P)} [X, \eta] \fatsemi [X, p] \fatsemi f\right) \quad \text{(Eq. 9)}$$
$$= [\delta^P]_{[X,Q]} \fatsemi \left(\prod_{p\in \mathrm{El}(P)} [X, \eta(p)] \fatsemi f\right) \quad \text{(Eq. 1)}$$

For the right-hand-side, we use the fact that;

$$[\delta^P]_{[Y,P]} \fatsemi \left(\prod_{p\in \mathrm{El}(P)} [Y, \eta(p)]\right) = [\delta^Q]_{[Y,P]} \fatsemi \left(\prod_{q\in \mathrm{El}(Q)} [Y, q]\right) \fatsemi [\eta]_{(Y)_{q\in Q}} \tag{11}$$

Which we can show by taking an element $y \in \mathrm{El}[Y, P]$,

$$
\begin{aligned}
y \fatsemi [\delta^P]_{[Y,P]} \fatsemi \left(\prod_{p \in \mathrm{El}(P)} [Y, \eta(p)] \right) &= \left(\prod_{p \in \mathrm{El}(P)} y \fatsemi [Y, \eta(p)] \right) \\
y \fatsemi [\delta^Q]_{[Y,P]} \fatsemi \left(\prod_{q \in \mathrm{El}(Q)} [Y, q] \right) \fatsemi [\eta]_{(Y)_{q \in Q}} &= \left(\prod_{q \in \mathrm{El}(Q)} y \fatsemi [Y, q] \right) \fatsemi [\eta]_{(Y)_{q \in Q}} \\
&= \left(\prod_{p \in \mathrm{El}(P)} y \fatsemi [Y, \eta(p)] \right)
\end{aligned}
$$

Therefore, we have;

$$
\begin{aligned}
(RHS) &= [f, Q] \fatsemi [Y, \eta] \fatsemi [\delta^P]_{[Y,P]} \fatsemi \left(\prod_{p \in \mathrm{El}(P)} [Y, p] \right) \\
&= [f, Q] \fatsemi [\delta^P]_{[Y,P]} \fatsemi \left(\prod_{p \in \mathrm{El}(P)} [Y, \eta(p)] \right) \quad \text{(Eq. 9, 1)} \\
&= [f, Q] \fatsemi [\delta^Q]_{[Y,P]} \fatsemi \left(\prod_{q \in \mathrm{El}(Q)} [Y, q] \right) \fatsemi [\eta]_{(Y)_{q \in Q}} \quad \text{(Eq. 11)} \\
&= [\delta^Q]_{[X,P]} \fatsemi \left(\prod_{q \in \mathrm{El}(Q)} [X, q] \fatsemi f \right) \fatsemi [\eta]_{(Y)_{q \in Q}}
\end{aligned}
$$

This shows how the two expressions differ. Using $[f, P]$ as opposed to $[f, Q]$ determines how many times the underlying operation is run in parallel. This has implications for the degrees of independence of the output, computational cost, or other "non-functional" factors which are overlooked in an all-natural **Set** based approach.

If we now assume that either f is deterministic, and therefore is natural with respect to all remappings, or that $\eta : P \to Q$ provides a natural remapping, then the RHS further simplifies;

$$
\begin{aligned}
(RHS) &= [\delta^Q]_{[X,P]} \fatsemi [\eta]_{(Y)_{q \in Q}} \fatsemi \left(\prod_{p \in \mathrm{El}(P)} [X, \eta(p)] \fatsemi f \right) \\
&= [\delta^P]_{[X,P]} \fatsemi \left(\prod_{p \in \mathrm{El}(P)} [X, \eta(p)] \fatsemi f \right) \\
&= (LHS)
\end{aligned}
$$

This provides index sliding. As the two sides are equal over an injective composition, we derive Equation 10. □

B Results Notebook

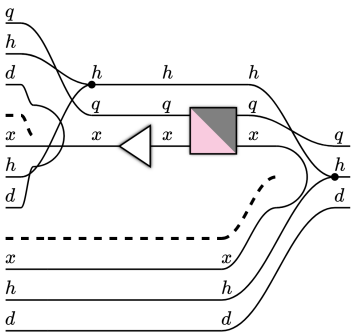
```
import asyncio
# Use server OR display locally, inline
USE_SERVER = True
import websocket_transfer.websockets_transfer as wst
import display as dsp
async def print_term(term):
    if USE_SERVER:
        await wst.send_term(term)
    else:
        dsp.print_category(term)
```

1.1.1 Autoalignment

```
import construction_helpers as ch # Required for
overloaded operators
import data_structure.Category as cat
import data_structure.Operators as ops

qk_matmul = ops.Einops.template('q h d, x h d -> h q
x')
softmax = ops.SoftMax.template()
mask = ops.WeightedTriangularLower.template()
sv_matmul = ops.Einops.template('h q x, x h d -> q h
d')
attention_core = qk_matmul @ softmax @ mask @
sv_matmul
await print_term(attention_core)
```

```
Received from server: {"msgType": "Connected"}
Received from server: {"msgType": "DataReceived"}
```

Diagram from TypeScript,

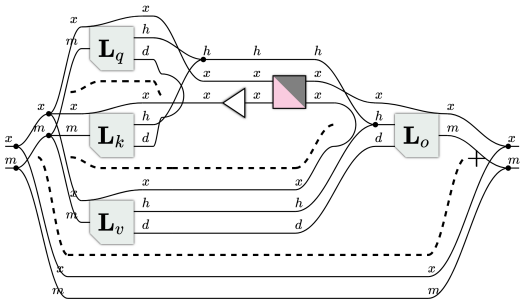


1.1.2 Configuration Generation

```
import term_utilities.generate_config as gc
import display.display_config as dpc

## Making the multi-head attention block
## (0, 0) becomes a Rearrangement with mapping \mu:
0 ↦ 0, 1 ↦ 0
def resnet[B: cat.Datatype, A: cat.Axis](target:
cat.BroadcastedCategory[B, A]):
    return (0, 0) @ target @
ops.AdditionOp.template()

Lq, Lk, Lv = [ops.Linear.template('m', 2, name) for
name in ['q', 'k', 'v']]
Lo = ops.Linear.template(2, 'm', 'o')
attention_linears = (0, 0, 0) @ (Lq * Lk * Lv) @
attention_core @ Lo
attention_block = resnet(attention_linears)
await print_term(attention_block)
```

```
Received from server: {"msgType": "Connected"}
Received from server: {"msgType": "DataReceived"}
```

Diagram from TypeScript,

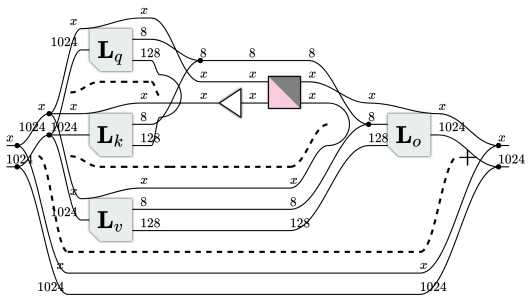


1.1.2 Applying the configuration;

```
print('Scanning for Free Numerics,')
config = gc.NumericConfig.template(attention_block)
print(dpc.display_config(config))

print('\nApplying Configuration,')
ATTN_INNER, ATTN_HEAD, MODEL_DIM = (128, 8, 1024)
config.assign_values(d=ATTN_INNER, h=ATTN_HEAD,
m=MODEL_DIM)
print(dpc.display_config(config))
configured_attention =
config.apply_context(attention_block)

await print_term(configured_attention)
```

```
Scanning for Free Numerics,
Name|Type       |Bucket  |Assignment
x   |FreeNumeric|        |
m   |FreeNumeric|        |
h   |FreeNumeric|        |
d   |FreeNumeric|        |
d   |FreeNumeric|        |

Applying Configuration,
Name|Type       |Bucket  |Assignment
x   |FreeNumeric|        |
m   |FreeNumeric|2       |Integer(_value=1024)
h   |FreeNumeric|1       |Integer(_value=8)
d   |FreeNumeric|0       |Integer(_value=128)
d   |FreeNumeric|0       |Integer(_value=128)
Received from server: {"msgType": "Connected"}
Received from server: {"msgType": "DataReceived"}
```

Diagram from TypeScript,



1.1.3 PyTorch compilation

```
import torch
import torch_compile.torch_compile as tc
import einops
from torch_compile.torch_utilities import
Multilinear

## An implemented torch module
def weighted_triangular_lower(x: torch.Tensor) ->
torch.Tensor:
    trilled = torch.tril(x)
    return trilled / (torch.sum(trilled, dim=-1,
keepdim=True) + 1e-8)

class MultiHeadAttention(torch.nn.Module):
    def __init__(self, d: int, h: int, m: int):
        super().__init__()
        self.d, self.h, self.m = d, h, m
        self.Lq, self.Lk, self.Lv = [Multilinear(m,
(h, d), bias=False) for _ in range(3)]
        self.Lo = Multilinear((h, d), m, bias=False)

    def forward(self, x):
        q, k, v = (self.Lq(x), self.Lk(x),
self.Lv(x))
        qk = einops.einsum(q, k, '... q h d, ... x h
d -> ... h q x')
        qk = torch.softmax(qk, dim=-1)
        qk = weighted_triangular_lower(qk)
        qkv = einops.einsum(qk, v, '... h q x, ... x
h d -> ... q h d')
        return self.Lo(qkv) + x

module_nn = MultiHeadAttention(ATTN_INNER,
ATTN_HEAD, MODEL_DIM)
module_cat =
tc.ConstructedModule.construct(configured_attention)

with torch.inference_mode():
    for p0, p1 in zip(module_nn.parameters(),
module_cat.parameters()):
        p0.data = p1.data.clone()

    BATCH_SIZE, SEQ_LEN = 8, 16
    dummy_data = torch.randn(BATCH_SIZE, SEQ_LEN,
MODEL_DIM)

    y_cat = module_cat(dummy_data)[0]
    y_nn = module_nn(dummy_data)
    print(torch.equal(y_cat, y_nn))
```

```
True
```

1.1.4 Graph Processing

```
import graphs.UIDHypergraph as uhg
graph = uhg.morphism2hypergraph(attention_block)
reconstructed = uhg.hypergraph2morphism(graph)
await print_term(reconstructed)
```

```
Received from server: {"msgType": "Connected"}
Received from server: {"msgType": "DataReceived"}
```

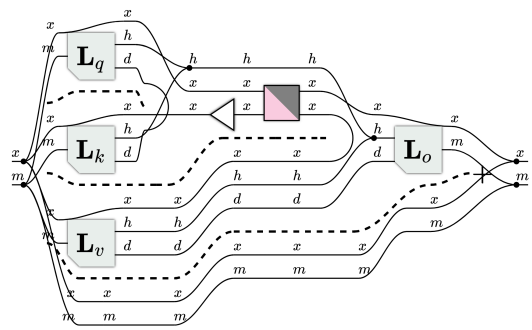



Diagram from TypeScript,