# Neural Circuit Diagrams: Robust Diagrams for the Communication, Implementation, and Analysis of Deep Learning Architectures

**Anonymous authors**
**Paper under double-blind review**

## Abstract

Diagrams matter. Unfortunately, the deep learning community has no standard method for diagramming architectures. The current combination of linear algebra notation and ad-hoc diagrams fails to offer the necessary precision to understand architectures in all their detail. However, this detail is critical for faithful implementation, mathematical analysis, further innovation, and ethical assurances. We present neural circuit diagrams, a graphical language based on category theory tailored to the needs of communicating deep learning architectures. Neural circuit diagrams naturally keep track of the changing arrangement of data, precisely show how operations are broadcast over axes, and display the critical parallel behavior of linear operations.

In this paper, we introduce neural circuit diagrams for an audience of machine learning researchers. Our introduction discusses the necessity of improved communication of architectures. Looking at related works, we establish why diagrams based on category theory offer the most promising approach. We then cover the mathematical foundations of neural circuit diagrams by introducing the category of *shaped data*. This section is accessible, given familiarity with sets and linear algebra, and contributes new perspectives on broadcasting, linear operations, and multilinearity. Finally, our results section justifies the utility of neural diagrams by explaining in quick succession a host of deep-learning architectures and features that are otherwise difficult to communicate. This includes convolution and its extensions with stride, dilation, and transposing; the transformer model; the U-Net; residual networks; and the vision transformer. We briefly discuss differentiation and graphically derive the memory cost of backpropagation, showing the potential of neural circuit diagrams to provide mathematical insight.

## 1 Introduction

**Target audience.** Neural circuit diagrams aim to appeal to a wide range of users. From software developers who just need reliable diagrams to design and implement architectures, to foundational mathematical researchers who require formal, systematic, descriptions of architectures. This paper targets machine learning researchers, who are between these extremes. We hope that this paper sets the foundation for future work disseminating neural circuit diagrams to a broader audience, as well as inspiring further mathematical analysis of deep learning architectures. Basic category theory will be used. In this work, categories can be thought of as a collection of sets (*objects*) with functions (*morphisms*) between them. Category theory is used as it shows the mathematical robustness of neural circuit diagrams and relates their development to the broader literature. We focus on *deep learning* architectures, rather than machine learning architectures generally, as it lets us assume that data is numeric and goes through a consistent procedure.

### 1.1 Necessity of Improved Communication in Deep Learning

Deep learning models are immense statistical engines. They rely on components connected in intricate ways to slowly nudge input data toward some target. Deep learning models convert big data into usable predictions, forming the core of many AI systems. The design of a model - its architecture - can significantly impact performance (Krizhevsky et al., 2017), ease of training (He et al., 2015; Srivastava et al., 2015), generalization (Ioffe & Szegedy, 2015; Ba et al., 2016), and ability to efficiently tackle certain classes of data (Vaswani et al., 2017; Ho et al., 2020). Architectures can have subtle impacts, such as different image models recognizing patterns at various scales (Ronneberger et al., 2015; Luo et al., 2017). Many significant innovations in deep learning have resulted from architecture design, often from frighteningly simple modifications (He et al., 2015). Furthermore, architecture design is in constant flux. New developments constantly improve on state-of-the-art methods (He et al., 2016; Lee, 2023), often showing that the most common designs are just one of many approaches worth investigating (Liu et al., 2021; Sun et al., 2023).

However, these critical innovations are presented using ad-hoc diagrams and linear algebra notation (Vaswani et al., 2017; Goodfellow et al., 2016). These methods are ill-equipped for the non-linear operations and actions on multi-axis tensors that constitute deep learning models (Xu et al., 2023; Chiang et al., 2023). These tools are insufficient for papers to present their models in full detail. Subtle details such as the order of normalization or activation components can be missing, despite their impact on performance (He et al., 2016). Works with immense theoretical contributions can fail to communicate equally insightful architectural developments (Rombach et al., 2022; Nichol & Dhariwal, 2021). Many papers cannot be reproduced without reference to accompanying code. This was quantified by (Raff, 2019), where only 63.5% of 255 machine learning papers from 1984 to 2017 could be independently reproduced without reference to the author's code. Interestingly, the number of equations present was *negatively* correlated with reproduction, further highlighting the deficits of how models are currently communicated. The year that papers were published had no correlation to reproducibility, indicating that this problem is not resolving on its own.

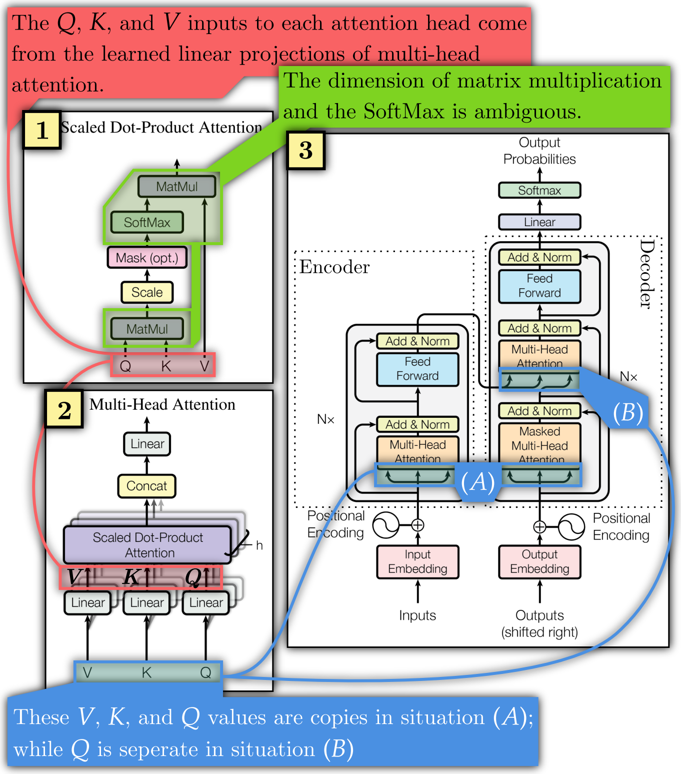
Relying on code raises many issues. The reader must understand a specific programming framework, and there is a burden to dissect and reimplement the code if frameworks mismatch. Without reference to a blueprint, mistakes in code cannot be cross-checked. The overall structure of algorithms is obfuscated, raising ethical risks about how data is managed (Kapoor & Narayanan, 2022). Furthermore, papers that clearly explain their models without resorting to code provide stronger scientific insight. As argued by Drummond (2009), replicating the code associated with experiments leads to weaker scientific results than reproducing a procedure. After all, replicating an experiment perfectly controls *all* variables, including irrelevant ones, making it difficult to link any independent variable to the observed outcome. However, in machine learning, papers often cannot be independently reproduced without referencing their accompanying code. As a result, the machine learning community misses out on experiments that provide general insight independent of specific implementations. Improved communication of architectures, therefore, will offer clear scientific value.

### 1.2 Case Study: Shortfalls of *Attention is All You Need*

To highlight the problem of insufficient communication of architectures, we present a case study of *Attention is All You Need*, the paper that introduced transformer models (Vaswani et al., 2017). Since being introduced in 2017, transformer models have revolutionized machine learning, finding applications in natural language processing, image processing, and generative tasks (Phuong & Hutter, 2022; Lin et al., 2021). Transformers' effectiveness stems partly from their ability to inject external data of arbitrary width into base data. We refer to axes representing the number of items in data as a **width**, and axes indicating information per item as a **depth**.

An **attention head** gives a weighted sum of the injected data's value vectors, $V$. The weights depend on the attention score the base data's query vectors, $Q$, assign to each key vector, $K$, of the injected data. Fully connected layers, consisting of learned matrix multiplication, generate $Q$, $K$, and $V$ vectors from the original base and injected data. **Multi-head attention** uses multiple attention heads in parallel, enabling efficient parallel operations and the simultaneous learning of distinct attributes.

*Attention is All You Need*, which we refer to as the original transformer paper, explains these algorithms using diagrams (see Figure 1) and equations (see Equation 1,2,3) that hinder understandability (Chiang et al., 2023; Phuong & Hutter, 2022).


The $Q$, $K$, and $V$ inputs to each attention head come from the learned linear projections of multi-head attention.

The dimension of matrix multiplication and the SoftMax is ambiguous.

These $V$, $K$, and $Q$ values are copies in situation $(A)$; while $Q$ is seperate in situation $(B)$

Figure 1: The diagrams from the original transformer model with our annotations. Critical information is missing regarding the origin of $Q$, $K$, and $V$ values (**red** and **blue**) along with the axes over which operations act (**green**).

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \;\; (d_k \text{ is the key depth}) \tag{1}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2}$$

$$\text{where head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{3}$$

The original transformer paper obscures dimension sizes and their interactions. The dimensions over which SoftMax[1] and matrix multiplication operates is ambiguous (Figure 1.1, **green**; Equation 1, 2, 3). Determining the initial and final matrix dimensions is left to the reader. This obscures key facts required to understand transformers. For instance, $K$ and $V$ can have a different width to $Q$, allowing them to inject external information of arbitrary width. This fact is not made clear in the original diagrams or equations. Yet, it is necessary to understand why transformers are so effective at tasks with variable input widths, such as language processing.

The original transformer paper also has uncertainty regarding $Q$, $K$, and $V$. In Figure 1.1 and Equation 1, they represent separate values fed to each attention head. In Figure 1.2 and Equation 2 and 3, they are all copies of each other at location $(A)$ of the overall model in Figure 1.3, while $Q$ is separate in situation $(B)$.

Annotating makeshift diagrams does not resolve the issue of low interpretability. As they are constructed for a specific purpose by their author, they carry the author's curse of knowledge (Pinker, 2014; Hayes &

---

[1] Using $i$ and $k$ to index over data, we have $\text{SoftMax}(\mathbf{v})[i] = \exp(\mathbf{v}[i])/\Sigma_k \exp(\mathbf{v}[k])$.

Bajzek, 2008; Ross et al., 1977). In Figure 1, low interpretability arises from missing critical information, not from insufficiently annotating the information present. The information about which axes are matrix multiplied or are operated on with the SoftMax is not present. We, therefore, need to develop a framework for diagramming architectures that ensures key information, such as the axes over which operations occur, is automatically shown. Taking full advantage of annotating the critical information already present in neural circuit diagrams, we present alternative diagrams in Figures 10, 11, and 19.

### 1.3 Current Approaches and Related Works

These issues with the current ad-hoc approaches to communicating architectures have been identified in prior works, which have proposed their own solutions (Phuong & Hutter, 2022; Chiang et al., 2023; Xu et al., 2023; Xu & Maruyama, 2022). This shows that this is a known issue of interest to the deep learning community. Non-graphical approaches focus on enumerating all the variables and operations explicitly, whether by extending linear algebra notation (Chiang et al., 2023) or explicitly describing every step with pseudocode (Phuong & Hutter, 2022). Visualization, however, is essential to human comprehension (Pinker, 2014; Borkin et al., 2016; Sadoski, 1993). Standard non-graphical methods are essential to pursue, and the community will benefit significantly from their adoption; however, a standardized graphical language is still needed.

Deep neural networks are typically composed of layers of abstraction. Individual commands assemble into components arranged into blocks and composed into architectures. Communicating architectures need to operate at these different scales. Expressing the details of implementing components common to most packages is redundant. Non-graphical methods have difficulty scaling up as the relationship between symbols becomes ever harder to parse. Either symbols are proliferated, or information is easily lost. This is observed in Equations 1, 2, 3 from *Attention is All You Need*, where a failure to scale leads to ill-formed expressions. Furthermore, the audience's experience level changes the appropriate level of abstraction. Specialized texts may want to communicate entire self-attention blocks with a single symbol or pictogram. A non-graphical approach may want to show them concisely; however, symbols will nonetheless proliferate if they have many inputs and outputs.

Graphical methods overcome this issue of abstraction. Architectures often scale by repeating some pattern, which can be delightfully expressed with graphical methods. Standard components can be shown with pictograms, abstracting away their precise details just as one would during implementation. This allows the focus to be placed on the study of architecture design. Repeated patterns can be "fenced off", clearly indicating them to the audience. They can be referenced and used without reiterating the precise detail, even across papers. Their proliferation is less painful if inputs and outputs are lines instead of symbols. Both how and why components are connected to the rest of the architecture become clear when the entire architecture can be observed at a glance.

The inclination towards visualizing complex systems has led to many tools being developed for industrial applications. Labview, MATLAB's Simulink, and Modelica are used in academia and industry to model various systems. For deep learning, TensorBoard and Torchview have become convenient ways to graph architectures. These tools, however, do not offer sufficient detail to implement architectures. They are often dedicated to one programming language or framework, meaning they cannot serve as a general means of communicating new developments. Developing a framework-independent graphical language for deep learning architectures would aid in improving these tools. This requires diagrams equipped with a mathematical framework that captures the changing structure of data, along with key operations such as broadcasting and linear transformations.

Many mathematically rigorous graphical methods exist for a variety of fields. This includes Petri nets, which have been used to model several processes (Murata, 1989). Tensor networks were developed for quantum physics and have been successfully extended to deep learning (Biamonte & Bergholm, 2017; Xu et al., 2023). Xu et al. (2023) showed that re-implementing models after making them graphically explicit can improve performance by letting parallelized tensor algorithms be employed. Formal graphical methods have also been developed in physics, logic, and topology. All these graphical methods have been found to represent an underlying category, a mathematical space with well-defined composition rules (Meseguer & Montanari, 1990; Baez & Stay, 2010). A category theory approach allows a common structure, monoidal products, to define an intuitive graphical language (Selinger, 2009; Fong & Spivak, 2019). Category theory, therefore, provides a robust framework to understand and develop new graphical methods.

However, a noted issue (Chiang et al., 2023) of previous graphical approaches is they have difficulty expressing non-linear operations. This arises from a tensor approach to monoidal products. Data brought together cannot necessarily be copied or deleted. This represents, for instance, axes brought together to form a matrix, and makes linear operations elegantly manageable. It, however, makes expressing copying and deletion impossible. The alternative Cartesian approach allows copying and deletion, reflecting the mechanics of classical computing. The Cartesian approach has been used to develop a mathematical understanding of deep learning (Shiebler et al., 2021; Fong et al., 2019; Wilson & Zanasi, 2022). However, Cartesian monoidal products do not automatically keep track of dimensionality and cannot easily represent broadcasting or linear operations. Therefore, the graphical language generated by a pure Cartesian approach fails to show the details of architectures, limiting its utility outside of pure analysis.

The literature reveals a combination of problems that need to be solved. Deep learning suffers from poor communication and needs a graphical language to understand and analyze architectures. Category theory can provide a rigorous graphical language but forces a choice between tensor or Cartesian approaches. The elegance of tensor products and the flexibility of Cartesian products must both be available to properly represent architectures. A category arises when a system has sufficient compositional structure, meaning a non-category theory approach to diagramming architectures will likely yield a category. The challenge of reconciling Cartesian and tensor approaches, therefore, remains.

**Our contributions.** To address the need for more robust communication of deep learning architectures, we develop the category of shaped data that combines Cartesian and tensor approaches. We do this by having a global Cartesian product called *tupling* that represents independent data and operations, separated by dashed lines. Within tuple segments, axes are represented by solid lines, *tensored* together. In this way, we can easily show broadcasted operations, and we contribute a general definition of broadcasting.

We then restrict ourselves to the linear subcategory and find that broadcasting becomes the tensor product. Furthermore, by noting the difference between the linear and multilinear forms of operations, we create an especially powerful linear subcategory that includes many operations typically reserved for the Cartesian approach. This mathematical foundation makes a standardized graphical approach possible and invites further mathematical analysis of architectures.

We then diagram several components and architectures to motivate the adoption of neural circuit diagrams and to prove their utility. This includes a basic multi-layer perceptron with accompanying code; the transformer architecture; convolution (and its difficult-to-explain permutations); the identity ResNet; the U-Net; and the vision transformer. We also provide a Jupyter notebook that implements diagrams in PyTorch, showing how diagrams are related to implementation. Lastly, we show differentiation using neural circuit diagrams, relating our work to existing research at the intersection of category theory and deep learning, and putting our technical contributions on par with similar work (Chiang et al., 2023).

## 2 The Category of Shaped Data

### 2.1 Building Blocks: Categories and Products

**Definition 2.1** (Categories, objects and morphisms)**.** A *category* $\mathcal{C}$ consists of a collection of symbols called objects, and for every ordered pair of objects $a$, $b$, a collection $\mathcal{C}(a,b)$ of *morphisms* from *a to b,* such that for any pair of morphisms $f \in \mathcal{C}(a,b)$ and $g \in \mathcal{C}(b,c)$, a composite $f;g \in \mathcal{C}(a,c)$ exists. Furthermore, each object has an identity morphism $\mathrm{Id}^a_a \in \mathcal{C}(a,a)$, and composition is associative.

We will be working in close analogy to the category of sets and functions between them, **Set**, meaning objects can be considered as sets and morphisms as functions between them. Often, we will talk about subcategories that exclude many morphisms so that further assumptions about structure can be made. Subcategories are categories, so they must be closed under composition. Furthermore, we will be using *staircase* notation. Morphisms $f \in \mathcal{C}(a,b)$ or $f : a \to b$ may be written $f^a_b$, which conveniently keeps track of objects and lets us distinguish between morphisms similarly defined for many objects. For example, identities are written as $\mathrm{Id}_a^a$. Finally, we will use forward composition with ";" instead of backward composition with "$\circ$" so that the direction of symbolic expressions aligns with the direction of diagrams.

**Definition 2.2** (Cartesian projections)**.** For an object $b$, an $|I|$-sized family of projections from $b$ to a family of objects $(Bi)_{i \in I}$ is a family of morphisms $\left(\pi_{Bi}^{b}\right)_{i \in I} \in \mathrm{Proj}(a)$ (we typically work with $n$-sized sets of projections);

- Which gives a **complete description**, for any object $a$ and two morphisms $f_b^a$ and $g_b^a$, if for all $i \in I$, we have $f_b^a ; \pi_{Bi}^b = g_b^a ; \pi_{Bi}^b$, then $f_b^a = g_b^a$
- Which **accepts free construction** *(the Cartesian property)*, for any family of morphisms $\left(f_{Bi}^a\right)_{i \in I}$, there exists a morphism $f_b^a = \Pi_b^{i \in I} f_{Bi}^a$ such that for all $i \in I$, we have $f_b^a ; \pi_{Bi}^b = f_{Bi}^a$.

**Definition 2.3** (Monoidal products)**.** A Category $\mathcal{C}$ with a monoidal product $*$ equipped with a unit object $I$ has a means of combining objects, $* : \mathrm{Ob}(\mathcal{C}) \times \mathrm{Ob}(\mathcal{C}) \to \mathrm{Ob}(\mathcal{C})$, and combining morphisms, $f_b^a \ * \ g_d^c = (f \ * \ g)_{b \, * \, d}^{a \, * \, c}$. It is then equipped with isomorphisms - associators and unitors - along with the pentagon and triangle axioms, ensuring regularity (Selinger, 2009, p.9). Monoidal products are ***bifunctors***, meaning that combined morphisms exhibit $(f_b^a ; k_e^b) * (g_d^c ; h_f^d) = (f * g)_{b*d}^{a*c} ; (k * h)_{e*f}^{b*d}$ and $\mathrm{Id}_a^a * \mathrm{Id}_b^b = \mathrm{Id}_{a \, * \, b}^{a \, * \, b}$.

We desire specific forms of isomorphisms, which we will introduce along with tupling, the primary product of our novel category. The benefit of monoidal products is they naturally lead to a graphical language (Fong & Spivak, 2019; Khan et al., 2022) and allow us to easily access further category theory-based analysis (Fritz et al., 2023; Cockett et al., 2019; Shiebler et al., 2021). Graphically, we show objects as stacked lines and morphisms with their domain to the left and codomain to the right. For example, we can show the bifunctor property as Figure 2.

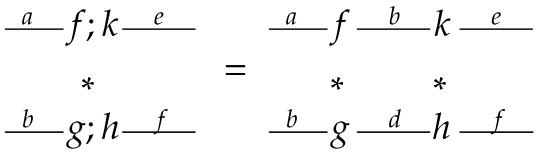


Figure 2: The bifunctor property for some generic monoidal product $*$, shown with monoidal string diagrams.

### 2.2 Tuples: Shaped Data as a Standard Cartesian Category

**Definition 2.4** (Shapes and indexes)**.** The category of shaped data over a field $V$ (*a set with addition and multiplication*), $\mathbf{ShD}(V)$, has as its objects *shapes,* $\mathbf{Sh}$. Every shape has a standard set of projections, called *indexes,* to object $\underline{1}$, which corresponds to the field, $V^1$. For a shape $a$, we write its indexes as $|i_a\rangle_1^a$, with $i_a$ enumerating over the indexes of $a$. The indexes form a set of projections, meaning shapes correspond to some $V^n$. Morphisms are functions between these sets. In addition to the field object $\underline{1}$, we have the terminal object $\underline{0}$ which corresponds to the singleton. Elements of shaped data of some shape $a$ correspond to the morphisms $\underline{0} \to a$. The field $V$, for example, corresponds to the morphisms $\underline{0} \to \underline{1}$.

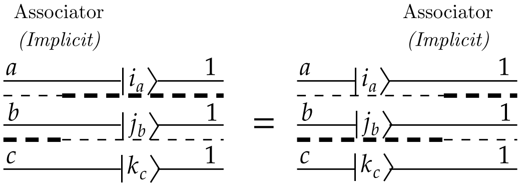
We start populating the shapes from the *lone axes* of natural number length, given by $\mathrm{Axis} : \{0\} \cup \mathbb{N} \to \mathbf{Sh}$. We write $\mathrm{Axis}(n)$ as $\underline{n}$, and it is the lone axis with $n$ indexes to $\underline{1}$.

**Definition 2.5** (Tupling)**.** Shaped data over a field $V$ has a left-additive symmetric Cartesian monoidal product called tupling denoted by "$\underline{+}$" with unit object $\underline{0}$. We generate new shapes and morphisms by $\underline{+} : \mathbf{ShD}(V) \times \mathbf{ShD}(V) \to \mathbf{ShD}(V)$.

**Definition 2.6** (Tupled shapes)**.** As $\underline{+}$ is a *Cartesian* monoidal product, all shapes constructed by $a \,\underline{+}\, b$, tupled shapes, has a set of projections $\{\pi_a^{a+b}, \pi_b^{a+b}\}$. We define their set of indexes $|i_{a+b}\rangle_1^{a+b}$, which forms a set of projections all to $\underline{1}$, as the union of the set $\pi_a^{a+b} ; |i_a\rangle_1^a$ and the set $\pi_b^{a+b} ; |i_b\rangle_1^b$. Additionally, a Cartesian product implies the existence of a copy map for each object, ${}_{a+a}^{a}$, and an erase map, $\bullet_0^a$ (Selinger, 2009, p. 42).

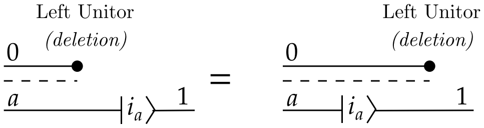
As tupling is a monoidal product, it is a bifunctor with a series of monoidal isomorphisms. We define these isomorphisms in a standard manner with respect to the indexes.

**Definition 2.7** (Associator)**.** This operation ensures that the order in which axes are tupled does not affect how they are treated. As a result, we will ignore the distinction between $(a \underline{+} b) \underline{+} c$ and $a \underline{+} (b \underline{+} c)$. ($\xrightarrow{\cong}$ represents an isomorphism.)



$$\alpha_{a+(b+c)}^{(a+b)+c} : (a \underline{+} b) \underline{+} c \xrightarrow{\cong} a \underline{+} (b \underline{+} c)$$

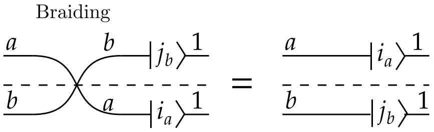
**Definition 2.8** (Left and right unitors)**.** These isomorphic operations allow the unit objects to be freely introduced and removed.



Left unitor; $\lambda_a^{0+a} : \underline{0} \underline{+} a \xrightarrow{\cong} a$

Right unitor; $\rho_a^{a+0} : a \underline{+} \underline{0} \xrightarrow{\cong} a$

**Definition 2.9** (Symmetric braiding)**.** As our monoidal product is symmetric, we additionally have ***symmetric braiding.*** This enforces symmetry and allows axes to cross in diagrams.

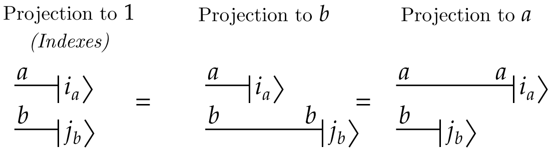


$$B_{b+a}^{a+b} : a \underline{+} b \xrightarrow{\cong} b \underline{+} a$$

**Definition 2.10** (Left-additive)**.** This means that there is a “commutative monoid” such that for every object $a$, there is a morphism $+_a^{a+a} : a \underline{+} a \to a$, and a zero map $0_a : \underline{0} \to a$. Respectively, these are considered index-wise additions between the tuple segments and a shape filled with zeroes. We only have to define the operation on singular values, given by $+ : \underline{1} \underline{+} \underline{1} \to \underline{1}$ over the field, which is then extended to other objects by broadcasting, which will be defined soon. We note the left-additive property as it integrates our work into other works on analyzing deep-learning models with category theory (Wilson & Zanasi, 2022; Cockett et al., 2019).

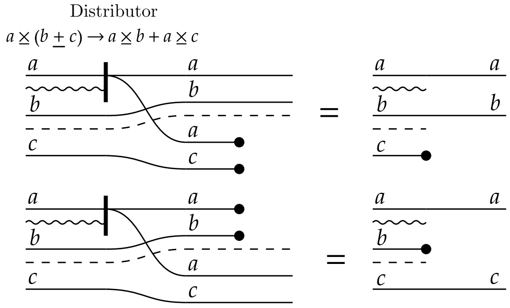
## 2.3 Tensored Shapes: Higher-Order Shapes

**Definition 2.11** (Tensored shapes)**.** We have an injective function $\underline{\times} : \mathbf{Sh} \times \mathbf{Sh} \to \mathbf{Sh}$ that generates tensored shapes. A tensored shape $a \underline{\times} b$ has an index $|i_a, j_b\rangle_1^{a\times b} = (|i_a\rangle \underline{\times} |j_b\rangle)_1^{a \times b}$ for each pair of indexes $|i_a\rangle_1^a$ and $|j_b\rangle_1^b$ of its constituent shapes. Furthermore, we have two valid sets of Cartesian projections using the constituent shapes. We have a set of projections to $b$ given by $|i_a\rangle^a \underline{\times} I_b^b$, and a set of projections to $a$ given by $I_a^a \underline{\times} |i_b\rangle^b$. We have not defined “$\underline{\times}$” between arbitrary morphisms, and the diagram below only defines $|i_a\rangle^a \underline{\times} I_b^b$ and $I_a^a \underline{\times} |i_b\rangle^b$.With no dashed line separation indicating tensoring shapes or morphisms, the indexes graphically satisfy the following condition; (*See cell 2 of the Jupyter notebook.*)



Furthermore, tensored shapes have the symmetric monoidal isomorphisms meaning they are associative, commutative, and have a unit object, which is $\underline{1}$. This means tensoring with $\underline{1}$ can be freely introduced and removed. We define index isomorphisms similar to tupling (Definitions 2.7–2.9). By drawing tensored shapes without dashed lines separating them, we conform to typical graphical notation (Fong & Spivak, 2019; Wilson & Zanasi, 2022). *However, tensoring is not a universal bifunctor.* We have not defined how arbitrary morphisms can be tensored.

We also define a distributor. The rules for generating objects allow for the construction of higher-order shapes such as $a \underline{\times} (b \underline{+} c)$, which we want to manage easily. We use a wavey line to graphically represent a priority "$\underline{\times}$" operation as in that expression. We have a distributor as an isomorphism $a \underline{\times} (b \underline{+} c) \xrightarrow{\cong} a \underline{\times} b + a \underline{\times} c$, which we define for projections as follows (*note that terminating a tuple with • erases it*);


Distributor
$a \underline{\times} (b \underline{+} c) \to a \underline{\times} b + a \underline{\times} c$
a
b
c
a
b
a
c
=
a
a
b
b
c
a
b
c
a
b
a
c
=
a
a
b
c
c

## 2.4 Broadcasting

**Definition 2.12** (Broadcasting)**.** Given an object $a$, a set of $n$ morphisms from $a$ to $b$, $m(i)^a_b$ for $i \in \{0 \dots n-1\}$, a morphism $f^b_c$, and an $n$-sized set of projections for $d$ to $c$ given by $\pi(i)^d_c$ for $i \in \{0 \dots n-1\}$, the *broadcasted morphism* $f'^a_d$ is such that, for all $i \in \{0 \dots n-1\}$, the following diagram commutes (ie. $f'^a_b; \pi(i)^d_c = m(i)^a_b; f^b_c$);

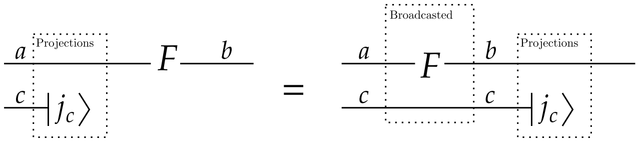
$$\begin{tikzcd}
a \arrow[r, "f'"] \arrow[d, "m(i)"'] & d \arrow[d, "\pi(i)"] \\
b \arrow[r, "f"] & c
\end{tikzcd}$$

**Lemma.** *The broadcasted morphism $f'^a_b$ exists and is unique.*

The broadcasted morphism, $f'^a_d$, is defined for a set of projections on $d$. Therefore, by the completeness of a set of projections, it is unique. As a set of projections accepts free construction, the broadcasted morphism $f'^a_d$ as defined above, constructed from the morphisms $m(i)^a_b; f^b_c$, exists. We call the set of morphisms $m(i)^a_b$ the pre-morphisms and the set of projections which follow, $\pi(i)^d_c$, the post-projections.

Note how in our definition of broadcasting, the broadcasting depends on the choice of morphisms to $b$ and the projections on $d$. Different sets of pre-morphisms or post-projections may yield different broadcasted morphisms. Furthermore, not using projections at the end may result in a $f'$ which is not unique. This definition of broadcasting is why we emphasize a standard choice of index morphisms. It is more general than other approaches (Chiang et al., 2023) and can help answer how to extend broadcasting to new circumstances (Perrone, 2022).

**Definition 2.13** (Broadcasting with indexes)**.** For a tensor $a \underline{\times} c$, we have pre-morphisms to $a$ according to the indexes of $c$, $I^a_a \underline{\times} |j_c\rangle^c$. For the tensor $b \underline{\times} c$ we have post-projections to $b$ according to the indexes of $c$, $I^b_b \underline{\times} |j_c\rangle^c$. Therefore, we can broadcast a morphism $F : a \to b$ into a morphism $a \underline{\times} c \to b \underline{\times} c$ according to; (*See cell 3 of the Jupyter notebook.*)



Note how the ability to broadcast follows from tensors having projections into their subshapes. It is an application of our lemma, rather than a completely novel construct. We can vertically reflect the diagram with symmetric braiding to get broadcasting above a morphism.

**Definition 2.14** (Inner broadcasting)**.** Inner broadcasting takes the same idea of delaying the application of an index to act *within* tuple segments. We have pre-morphisms from $a \;\underline{+}\; (b \;\underline{\boxtimes}\; d)$ to $a \;\underline{+}\; b$ given by $I_a^a \;\underline{+}\; \left(I_b^b \boxtimes |k_d\rangle^{d}\right)$, iterating over the indexes of $d$. We have post-projections from $c \;\underline{\boxtimes}\; d$ to $c$ given by $I_c^c \;\underline{\boxtimes}\; |k_d\rangle^{d}$. This choice lets us lift an operation $G : a \;\underline{+}\; b \to c$ into a morphism $a \;\underline{+}\; (b \;\underline{\boxtimes}\; d) \to c \;\underline{\boxtimes}\; d$. This is shown in Figure 3. (*See also cell 4 of the Jupyter notebook.*)

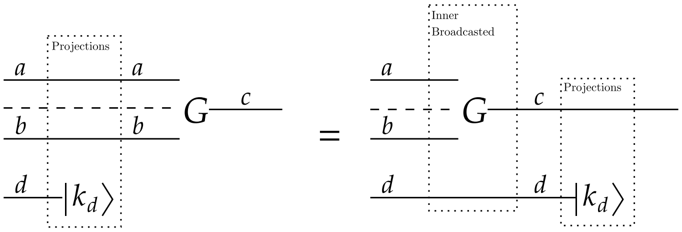


Figure 3: *Inner broadcasting.* We assert the below expression for all indexes of $d$. The left-hand side is fully defined, and we use it to infer the indexes of the inner broadcast on the right-hand side. This leaves us with a morphism defined for all its indexes and hence is uniquely defined.

### 2.5 Elementwise Operations

Functions on the field $V$ (see Figure 4), in our framework, correspond to morphisms $f : \underline{1} \to \underline{1}$. When broadcast over shapes of any size, we get elementwise operations. The broadcasted operations have the form $\underline{1} \boxtimes a \to \underline{1} \boxtimes a$, which we reduce to $a \to a$ by the unitors for tensoring managing the unit object $\underline{1}$. We represent the unitor introduction or removal of $\underline{1}$ with arrows, representing them appearing or disappearing as needed. Note how broadcasting rules require that broadcasted forms of $f : \underline{1} \to \underline{1}$ always have matching input and output shapes. (*See cell 5, Jupyter notebook.*)

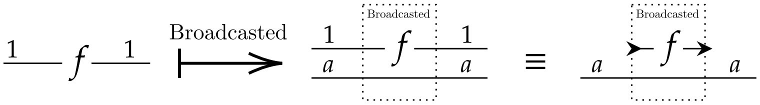


Figure 4: Broadcasting a $f : \underline{1} \to \underline{1}$ morphism over some shape gives an elementwise operation.

### 2.6 Addition and Copying

Addition is, fundamentally, an operation between two numbers producing another number, $+ : \underline{1} \;\underline{+}\; \underline{1} \to \underline{1}$. This is the only form of addition that needs to be defined - it can then be broadcast and rearranged into many higher-order forms. The different forms of addition naturally follow from the rules of neural circuit diagrams (see Figure 5), including broadcasting, inner broadcasting, and the application of unitors. Copying is an operation $\Delta^{1}_{1+1}$ that acts like the natural counterpart to addition. It can be broadcast to $\Delta^{a}_{a+a}$ for any shape. However, inner broadcasting is asymmetric, so copying does not have the same flexibility as addition. (*See cell 6, 7, Jupyter notebook.*)

### 2.7 Uniting Cartesian and Tensor Perspectives: The Linear Subcategory

Currently, our category contains all functions between shapes, restricted only by the requirement composition with the indexes uniquely define morphisms to a shape. We now consider the subcategory of linear morphisms, Lin**ShD**$(V)$. In this subcategory, broadcasting becomes a universal tensor product, allowing complex linear interactions to be expressed. All operations in the subcategory are inherited by our main category through the inclusion functor, $\iota : \mathrm{Lin}\mathbf{ShD}(V) \hookrightarrow \mathbf{ShD}(V)$, meaning diagrams inherit the various means of manipulating linear morphisms. We will first cover a brief review of linear maps.

**Definition 2.15** (Linear map)**.** A map between vector spaces over a field $V$, $L : A \to B$, is a *linear map* if it obeys additivity, $L(a) + L(a') = L(a + a')$ for $a, a' \in A$, and homogeneity, $L(v \cdot a) = v \cdot L(a)$ for $v \in V$ (Cockett et al., 2019).

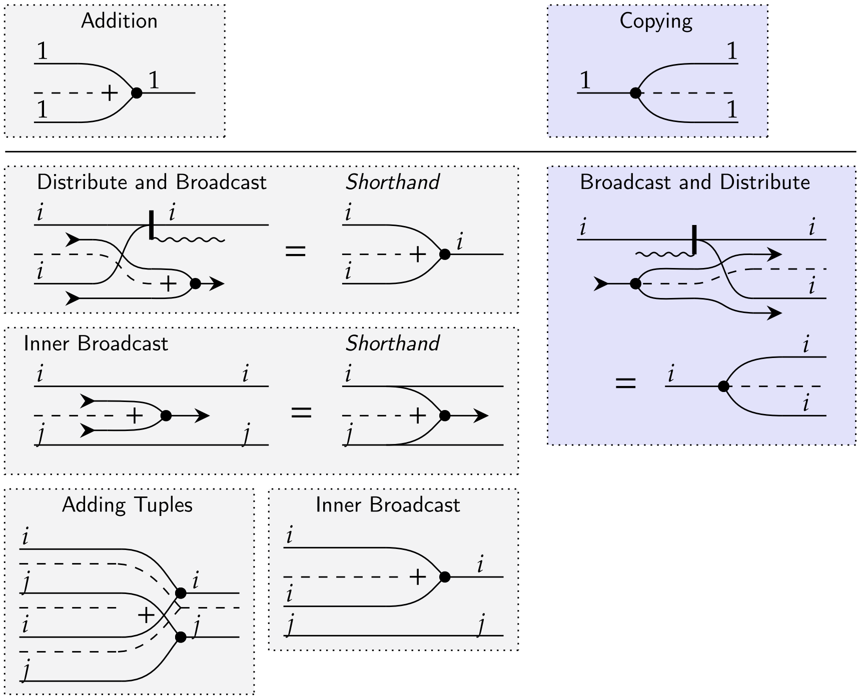


Figure 5: Different forms of addition are generated from broadcasting addition over the field to other shapes. Copying can be broadcast to any shape. However, inner broadcasting is asymmetric, so copying does not have the same flexibility as addition.

**Lemma.** *If $A$, a vector space over a field $V$, has basis vectors $\{\mathbf{a}_i\}$, then all linear maps $L : A \to B$ to some other vector space are uniquely identified by the values $L(\mathbf{a}_i)$.*

When we have $L$ act on some $\mathbf{w} = \Sigma_i \mathbf{a}_i w^i$, by linearity we get $L(\mathbf{w}) = \Sigma_i w^i L(\mathbf{a}_i)$. Therefore, if we have $L(\mathbf{a}_i) = M(\mathbf{a}_i)$ for all $\{\mathbf{a}_i\}$, then $L(\mathbf{w}) = M(\mathbf{w})$ for all $\mathbf{w} \in A$. Therefore, $L = M$ (Cockett et al., 2019, Proposition 11.).

**Definition 2.16** (The linear subcategory)**.** The subcategory of linear morphisms, Lin**ShD**$(V)$, only includes morphisms between shapes that obey *additivity* and *homogeneity*. Using neural circuit diagrams, we show these properties with Figure 6.

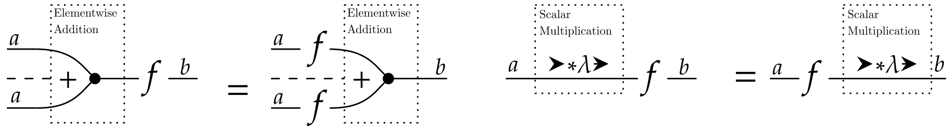
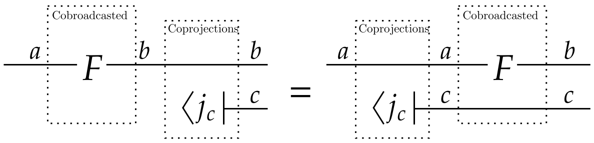


Figure 6: Additivity and homogeneity shown using neural circuit diagrams.

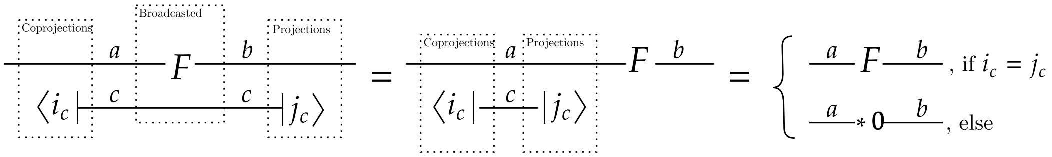
**Definition 2.17** (Coindexes and direct sums)**.** This restriction enforces regularity in our linear subcategory. By the properties of linear maps, these morphisms are uniquely identified by their action on the basis elements. For every object $a$, we construct a basis of coindexes $\langle i_a|^{1}_{a}$ such that $\langle i_a|^{1}_{a}; |j_a\rangle^{a}_{1}$ is scalar multiplication ($V^1 \to V^1$) of times 1, the identity on $\underline{1}$, when $i_a = j_a$, and of times 0 otherwise. Shapes, therefore, are *co*products in addition to being products. Following from linearity, full addition $+^{a+a}_{a}$ acts as *co*copying, acting in an equal but opposite manner. Note that raw data morphisms $A : \underline{0} \to a$ are *not* linear as they disobey additivity and homogeneity. The only linear raw data morphisms from $A : \underline{0} \to a$ are those that return zeros. In our linear subcategory, $\underline{0}$ is the *zero object*. To introduce or remove data, operations of the form $B : \underline{1} \to b$ must be used.

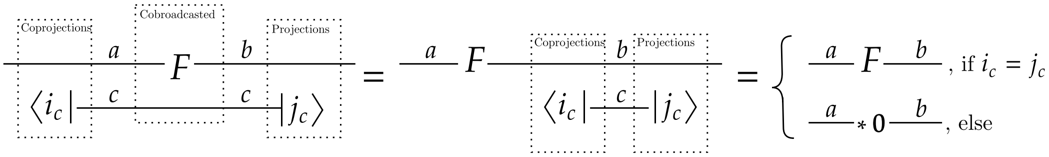
## 2.8 Cobroadcasting and the Tensor Product

Furthermore, coindexes allow for cobroadcasting. Similarly to Definition 2.13, we can uniquely define a cobroadcasted operation by having the following hold for the $j_c$ enumerated morphisms of $b \rightarrow b \underline{\times} c$ and coprojections $a \rightarrow a \underline{\times} c$;



We can analyze whether this is equal to regular broadcasting by considering all coindexes and indexes of the $c$ axis. This covers all the necessary cases needed to identify whether the broadcasted and cobroadcasted morphisms are the same.





Therefore, in the linear subcategory, we are justified in considering broadcasting and cobroadcasting to be the same operation, which we take to be $F_b^a \underline{\times} I_c^c$. Observe how this gives a tensor product between the linear map $F$ and the identity on $c$. The tensor product is a bifunctor on linear maps, meaning that $(F_b^a \otimes I_c^c); (I_b^b \otimes G_d^c) = F_b^a \otimes G_d^c$. In the linear subcategory, we take tensoring "$\underline{\times}$" to be the tensor product, letting us define simultaneous broadcasting of linear operations by $F_b^a \underline{\times} G_d^c = (F_b^a \underline{\times} I_c^c); (I_b^b \underline{\times} G_d^c)$. In the linear subcategory, $\underline{\times}$ is a full symmetric monoidal product, inheriting all the associated graphical regularity (Wilson & Zanasi, 2022; Fong et al., 2019).

Our main category contains the linear subcategory, and thus valid tensored linear morphisms also exist in our main category. The useful properties and means of manipulating linear operations are maintained. Our approach shows that broadcasting and tensor products are closely related, justifying our system similarly representing them. Furthermore, our framework for linear operations can consider operations involving tuples, such as copying, deletion, and addition, as long as they obey additivity and homogeneity. This makes our framework more flexible than previous approaches and allows us to capture the operations which occur in practice.

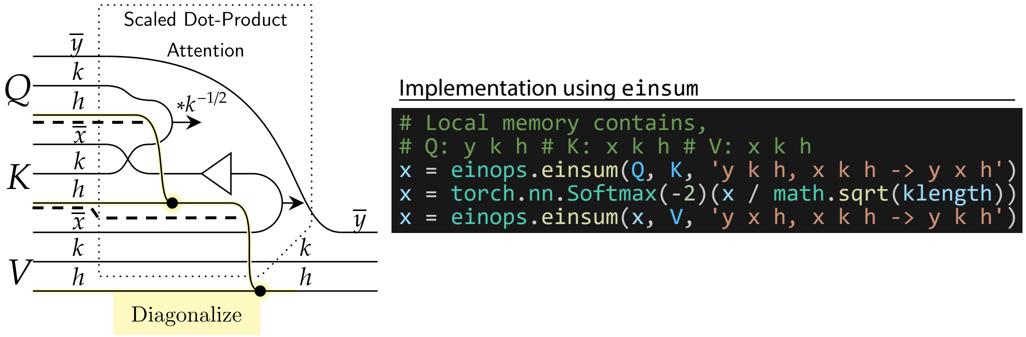
## 2.9 A Common Roadblock: A Note on Multilinearity

The expressiveness of our linear subcategory hinges on associativity and homogeneity being closed under composition. This loose definition of linearity allows for copying, tuple addition, and other operations to be included, atypical of many tensor-based approaches. However, this comes at the cost of abandoning multilinearity. These operations have a tuple input and are linear with respect to each *individually*. An example is the dot product. Scaling both inputs by the same scalar results in multiplying the output by the scalar square. Therefore, the dot product from a tuple to a singular value cannot be contained in our linear subcategory.

Every tuple-multilinear operation has an associated tensor-linear form. We include the tensor-linear form in our linear subcategory and let the outer product map from tuples to tensors in our main category. The outer product is the ur-multilinear operation that takes a tuple input and generates a tensor output, with the tensor indexes selecting an element from each tuple segment to be multiplied together. Applying the outer product, however, requires us to leave and reenter our linear subcategory. The outer product is implied by having a dashed tuple separation end, naturally implying a tuple becoming a tensor. Distinguishing between an operation's tuple-multilinear and tensor-linear forms allows the latter to be present in our linear subcategory.

### 2.10 Summing Over and Rearranging Data

Sums over axes and data rearrangement are essential operations that must be carefully expressed. Functions such as einsum and packages such as einops naturally express these operations by symbolically matching the input to desired output shapes (Rogozhnikov, 2021). We similarly express these operations by drawing lines between input and output dimensions, as shown in Figure 7. Note we first take an implicit outer product so that these operations are all considered tensor-linear and can be simultaneously broadcast. Summing over two dimensions of similar length, **dot products**, are shown with a cup. **Views** rearrange their input tensors into the shape of their output and are shown with a horizontal black bar which consumes the input dimensions and produces the output shape. **Transposing** rearranges the order of axes and is inherited from symmetric braiding. **Diagonalization** takes two tensored axes of the same length and makes them take the same index by joining them, which has the same effect as element-wise multiplication. (*See cell 8, 9, Jupyter notebook.*)



```
# Local memory contains,
# Q: y k h # K: x k h # V: x k h
x = einops.einsum(Q, K, 'y k h, x k h -> y x h')
x = torch.nn.Softmax(-2)(x / math.sqrt(klength))
x = einops.einsum(x, V, 'y x h, x k h -> y k h')
```

Figure 7: A section of scaled dot-product attention displays an implied outer product, dot products, diagonalization, and the natural relationship between neural circuit diagrams and implementation using the convenient einsum function or the einops package (Rogozhnikov, 2021).

### 2.11 Associated Tensors and Linear Algebra

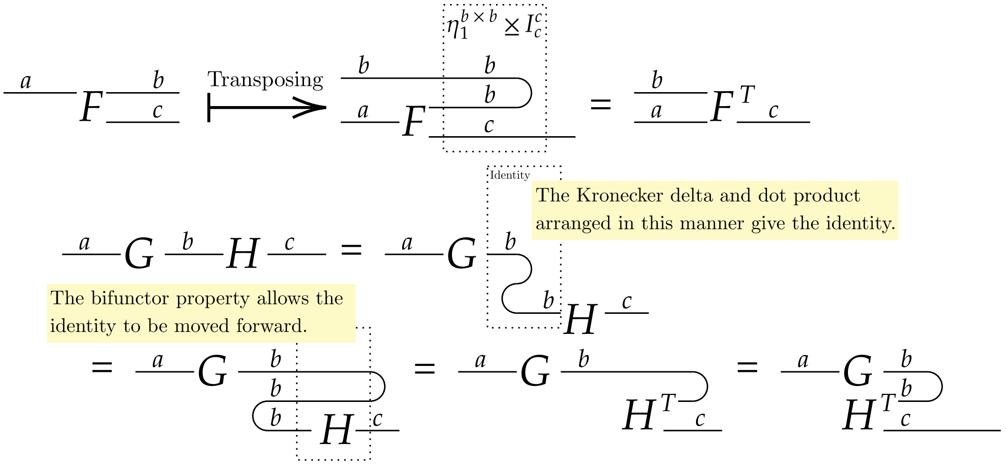
Linear operations $F^{a}_{b \times c}$ have an associated $a \underline{\times} (b \underline{\times} c)$ shaped tensor. We can therefore construct another linear operation $F^{a \times b}_{c}$ by viewing the tensor as $(a \underline{\times} b) \underline{\times} c$. This transposes the operation, offering flexibility in exchanging input and output shapes. These transposes are themselves linear operations $a \underline{\times} (b \underline{\times} c) \rightarrow (a \underline{\times} b) \underline{\times} c$. They can be represented by applying dot products, $\eta^{a \times a}_{1}$, or the Kronecker delta, $\delta^{1}_{a \times a}$, together with $\underline{\times} I^{a}_{a}$ as seen in Figure 8. Horizontally composed linear operations sum over the axes which join them, so we can think of the dot product as having a tensor that returns 1 when the coindexes match and zero otherwise, over which we sum. The Kronecker delta tensor behaves similarly for indexes. (*See cell 10, Jupyter notebook.*)

## 3 Results: Key Applied Cases

### 3.1 Basic Multi-Layer Perceptron

Diagramming a basic multi-layer perceptron will help consolidate knowledge of neural circuit diagrams and show their value as a teaching and implementation tool. We use pictograms to represent components analogous to traditional circuit diagrams and to create more memorable diagrams (Borkin et al., 2016).

Fully connected layers are shown as boldface **L**, with boldface indicating a component with internal learned weights. Their input and output sizes are inferred from the diagrams. If a fully connected layer is biased,



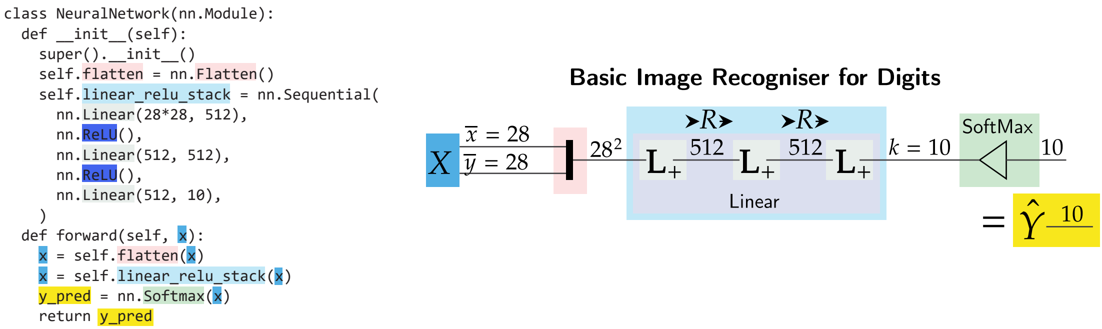
The Kronecker delta and dot product arranged in this manner give the identity.

The bifunctor property allows the identity to be moved forward.

Figure 8: Linear operations have a flexible algebra. Simultaneous operations may increase efficiency (Xu et al., 2023). As the height of diagrams is related to the amount of data stored in independent segments, it gives a rough idea of memory usage.

we add a "+" in the bottom right. Traditional presentations easily miss this detail. For example, many implementations of the transformer, including those from PyTorch and Harvard NLP, have a bias in the query, key, and value fully-connected layers despite *Attention is All You Need* (Vaswani et al., 2017) not indicating the presence of bias.

Activation functions are just element-wise operations. Though traditionally ReLU (Krizhevsky et al., 2017), other choices may yield superior performance (Lee, 2023). With neural circuit diagrams, the activation function employed can be checked at a glance. SoftMax is a common operation that converts scores into probabilities, and we represent it with a left-facing triangle (◁), indicating values being "spread" to sum to 1. As mentioned in Section 1.2, how operations such as SoftMax are broadcast can be ambiguous in traditional presentations. This is especially worrisome as SoftMax can be applied to shapes of arbitrary size. On the other hand, our method of displaying broadcasting makes it clear how SoftMax is applied.



```
class NeuralNetwork(nn.Module):
  def __init__(self):
    super().__init__()
    self.flatten = nn.Flatten()
    self.linear_relu_stack = nn.Sequential(
      nn.Linear(28*28, 512),
      nn.ReLU(),
      nn.Linear(512, 512),
      nn.ReLU(),
      nn.Linear(512, 10),
    )
  def forward(self, x):
    x = self.flatten(x)
    x = self.linear_relu_stack(x)
    y_pred = nn.Softmax(x)
    return y_pred
```

Figure 9: PyTorch code and a neural circuit diagram for a basic MNIST (digit recognition) neural network taken from an introductory PyTorch tutorial. Note the close correspondence between neural circuit diagrams and PyTorch code. (*See cell 11, Jupyter notebook.*)

### 3.2 Neural Circuit Diagrams for the Transformer Architecture

In Section. 1.2, we identified the shortfalls in *Attention is All You Need*. We now have the tools to address these shortcomings using neural circuit diagrams. Figure. 10 shows scaled-dot product attention. Unlike the approach from *Attention is All You Need*, the size of variables and the axes over which matrix multiplication and broadcasting occur is clearly shown. Figure. 11 shows multi-head attention. The origin of queries, keys, and values are clear, and concatenating the separate attention heads using einsum naturally follows. Finally, we show the full transformer model in Figure. 19 using neural circuit diagrams. Introducing such a large architecture requires an unavoidable level of description, and we take some artistic license and notate all the additional details.

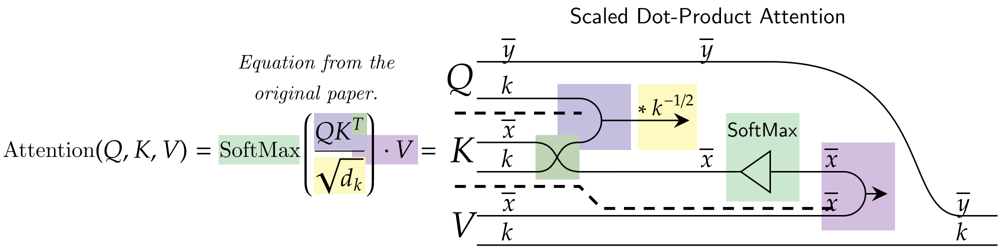

Scaled Dot-Product Attention
Equation from the original paper.
Attention(Q, K, V) = SoftMax(QK^T / √d_k) · V =
Q
K
V
y
k
x
k
x
k
y
* k^{-1/2}
SoftMax
x
x
x
y
k

Figure 10: The original equation for attention against our diagram. The descriptions are unnecessary but clarify what is happening. Corresponds to Equation 1 and Figure 1.1. (*See cell 12, Jupyter notebook.*)

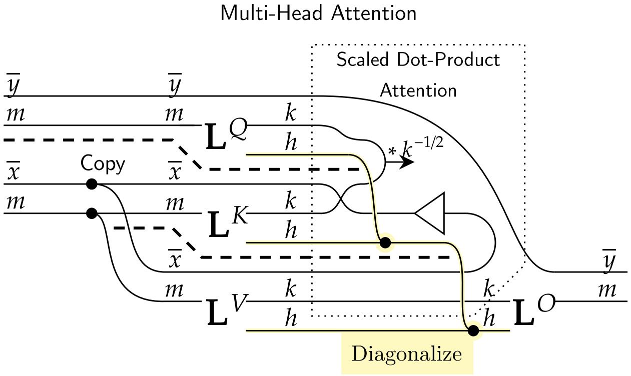


Figure 11: Neural circuit diagram for **multi-head attention**. In PyTorch, the implementation of matrix multiplication and dot products are clear with the torch.einsum function. Corresponds to Equation 2 and 3 and Figure 1.2. (*See cell 12, Jupyter notebook.*)

### 3.3 Convolution

Convolutions are critical to understanding computer vision architectures. Different architectures extend and use convolution in a variety of ways, so implementing and understanding these architectures requires convolution and its variations to be accurately expressed. However, these extensions are often hard to explain. For example, PyTorch concedes that dilation is "harder to describe". Transposed convolution is similarly challenging to communicate (Zeiler et al., 2010). A standardized means of notating convolution and its variations would aid in communicating the ideas already developed by the machine learning community and encourage more innovation of sophisticated architectures such as vision transformers (Dosovitskiy et al., 2021; Khan et al., 2022).

In deep learning, convolutions alter a tensor by taking weighted sums over nearby values. With standard bracket notation to access values, a convolution over vector $v$ of length $\overline{x}$ by a kernel $w$ of length $k$ is given by, *(Note: we subscript indexes by the axis over which they act.)*

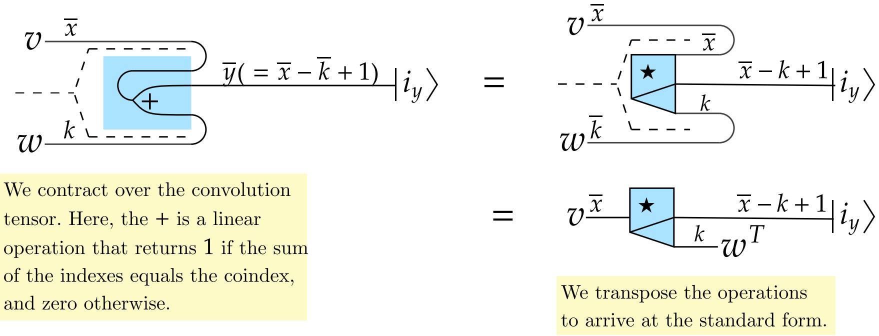
$$\mathrm{Conv}(v, w)[i_{\overline{y}}] = \sum_{j_k} v[i_{\overline{y}} + j_k] \cdot w[j_{\overline{k}}]$$

We start indexing from 0, meaning the maximum $i_{\overline{y}}$ value is given by $i_y + k - 1 = \overline{x} - 1$, and the length of the output is therefore $\overline{y} = \overline{x} - k + 1$. Note how convolution is a multilinear operation; it is linear with respect to each vector input $v$ and $w$. Therefore, it has a tensor-linear has an associated tensor, the convolution tensor, that uniquely identifies it.

$$\mathrm{Conv}(v,w)[i_{\overline{y}}] = \sum_{j_k}\sum_{\ell_x} (\star)[i_{\overline{y}}, j_k, \ell_{\overline{x}}]\cdot v[\ell_{\overline{x}}]\cdot w[j_k]$$

$$(\star)[i_{\overline{y}}, j_k, \ell_{\overline{x}}] = \begin{cases}1 & \text{, if } \ell_{\overline{x}} = i_{\overline{y}} + j_k. \\ 0 & \text{, else.}\end{cases}$$

We diagram convolution with the below diagram, Figure 12. We then transpose the linear operation into a more standard form, letting the input be to the left, and the kernel be to the right.



We contract over the convolution tensor. Here, the + is a linear operation that returns 1 if the sum of the indexes equals the coindex, and zero otherwise.

We transpose the operations to arrive at the standard form.

Figure 12: Convolution is a multilinear operation, with an associated tensor. This tensor is transposed into a standard form.

We typically work with higher dimensional convolutions, in which case the indexes act like tuples of indexes. We diagram axes that act in this tandem manner by placing them especially close to each other and labeling their length by one bolded symbol akin to a vector. In $2D$ the convolution tensor becomes;

$$(\star\ 2D)[i_{\overline{y0}}, i_{\overline{y1}}, j_{k0}, j_{k1}, \ell_{\overline{x0}}, \ell_{\overline{x1}}] = \begin{cases}1 & \text{, if } (\ell_{\overline{x0}}, \ell_{\overline{x1}}) = (i_{\overline{y0}}, i_{\overline{y1}}) + (j_{k0}, j_{k1}). \\ 0 & \text{, else.}\end{cases}$$

Figure 13 shows what convolution does. It takes an input, uses a linear operation to separate it into overlapping blocks, and then broadcasts an operation over each block. Using neural circuit diagrams, we now easily show the extensions of convolution. A standard convolution operation tensors the input with a channel depth axis, and feeds each block and the channel axis through a learned linear map. Additionally, we can take an average, maximum, or some other operation rather than a linear map on each block. This lets us naturally display average or max pooling, among other operations. Displaying convolutions like this has further benefits for understanding. For example, $1\times1$ convolution tensors give a linear operation $\overline{\mathbf{x}} \to \overline{\mathbf{x}} \underline{\times} 1$, which we recognize to be the identity. Therefore, $1\times1$ kernels are the same as broadcasting over the input.

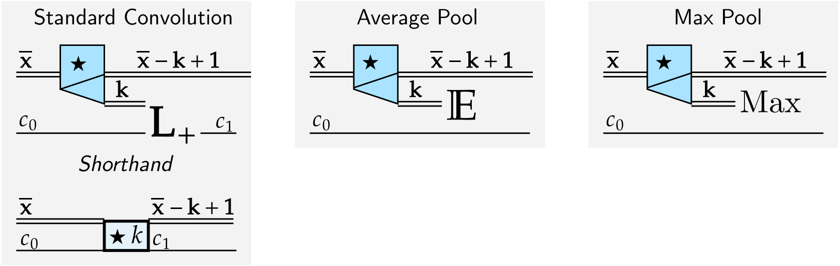


Figure 13: Convolution and related operations, clearly shown using neural circuit diagrams.

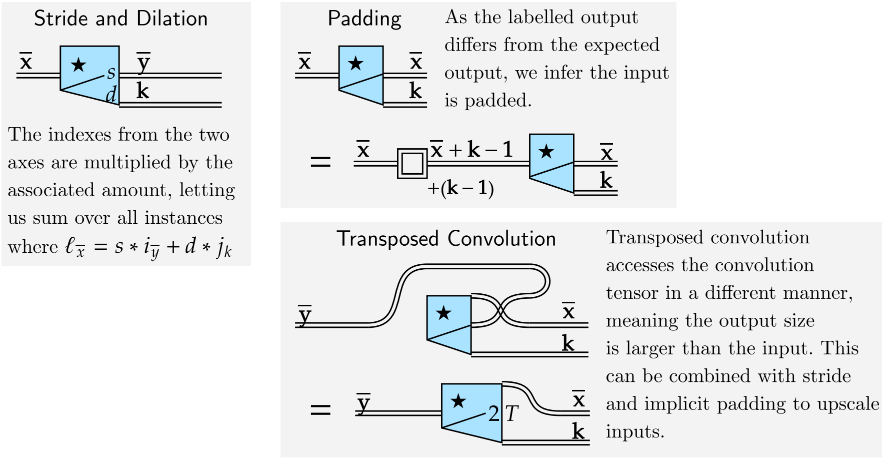
*Stride* and *dilation* scale the contribution of $i_y$ or $j_k$ in the convolution tensor, increasing the speed at which the convolution scans over its inputs. This changes the convolution tensor into the form of Equation 4. We diagram these changes by adding the $s$ or $d$ multiplier where the axis meets the tensor as in Figure 14. These multipliers also change the size of the output, allowing for downscaling operations.

$$(\star\ s, d)[i_{\overline{y}}, j_k, \ell_{\overline{x}}] = \begin{cases} 1 & \text{, if } \ell_{\overline{x}} = s\ *\ i_{\overline{y}} + d\ *\ j_k. \\ 0 & \text{, else.} \end{cases} \tag{4}$$

$$\overline{y} = \left\lfloor \frac{\overline{x} - d\ *\ (k-1) - 1}{s} + 1 \right\rfloor \tag{5}$$

We often want to make slight adjustments to the output size. This is done by **padding** the input with zeros around its borders. We can explicitly show the padding operation, but we make it implicit when the output dimension does not match the expectation given the input dimension, kernel dimension, stride, and dilation used.

Stride can make the output axis have a far lower dimension than the input axis. This is perfect for downscaling. However, it does not allow for upscaling. We implement upscaling by transposing strided convolution, resulting in an operation with many more output blocks than actual inputs. We broadcast over these blocks to get our high-dimensional output. Transposed convolution is challenging to intuit in the typical approach to convolutions, which focuses on visualizing the scanning action rather than the decomposition of an image's data structure into overlapping blocks. The blocks generated by transposed convolution can be broadcast with linear maps, maximum, average, or other operations, all easily shown using neural circuit diagrams.



The indexes from the two axes are multiplied by the associated amount, letting us sum over all instances where $\ell_{\overline{x}} = s * i_{\overline{y}} + d * j_k$

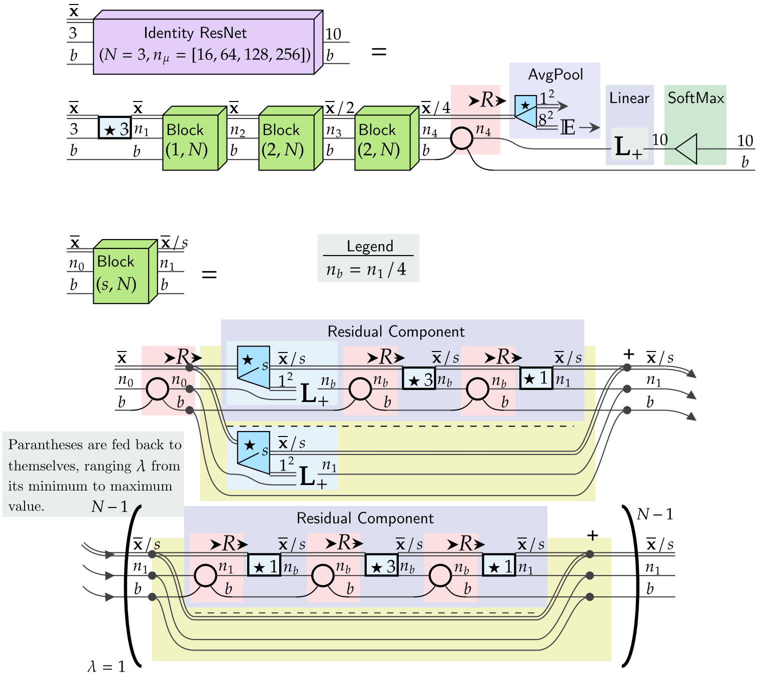
As the labelled output differs from the expected output, we infer the input is padded.

Transposed convolution accesses the convolution tensor in a different manner, meaning the output size is larger than the input. This can be combined with stride and implicit padding to upscale inputs.

Figure 14: Stride, dilation, padding, and transposed convolution shown with neural circuit diagrams.

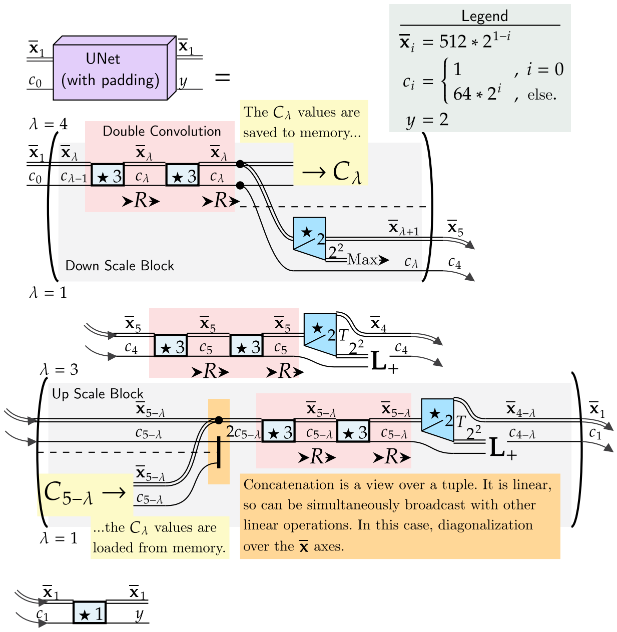
### 3.4 Computer Vision

In computer vision, the design of deep learning architectures is critical. Computer vision tasks often have enormous inputs that are only tractable with a high degree of parallelization (Krizhevsky et al., 2017). Additionally, different architectures can relate information at different scales (Luo et al., 2017). Sophisticated architectures such as vision transformers combine the complexity of convolution and transformer architectures (Khan et al., 2022; Dehghani et al., 2023). Neural circuit diagrams, therefore, are in a unique position to accelerate computer vision research, motivating parallelization, task-appropriate architecture design, and further innovation of sophisticated architectures. Architectures often comprise sub-components, which we show as blocks that accept configurations. This is analogous to classes or functions that may appear in code. As examples of neural circuit diagrams applied to computer vision architectures, we have diagrammed the identity residual network architecture (He et al., 2016) in Figure 15, which shows many innovations of ResNets not included in common implementations, as well as the UNet architecture (Ronneberger et al., 2015) in Figure 16, which lets us show how saving and loading variables may be displayed.



Parantheses are fed back to themselves, ranging $\lambda$ from its minimum to maximum value.

Figure 15: Residual networks with identity mappings and full pre-activation (IdResNet) (He et al., 2016) offered improvements over the original ResNet architecture. These improvements, however, are often missing from implementations. By making the design of the improved model clear, neural circuit diagrams can motivate common packages to be updated. (*See cell 13, 14, 15, Jupyter notebook.*)



The $C_\lambda$ values are saved to memory...

...the $C_\lambda$ values are loaded from memory.

Concatenation is a view over a tuple. It is linear, so can be simultaneously broadcast with other linear operations. In this case, diagonalization over the $\overline{\mathbf{x}}$ axes.

Figure 16: The UNet architecture (Ronneberger et al., 2015) forms the basis of probabilistic diffusion models, state-of-the-art image generation tools (Rombach et al., 2022). UNets rearrange data in intricate ways, which we can show with neural circuit diagrams. Note that in this diagram we have modified the UNet architecture to pad the input of convolution layers. To get the original UNet architecture, the $\overline{\mathbf{x}}_\lambda$ values can be further distinguished as $\overline{\mathbf{x}}_{\lambda,j}$, the sizes of which can be added to the legend. (*See cell 16, Jupyter notebook.*)

### 3.5 Vision Transformer

Neural circuit diagrams reveal the degrees of freedom of architectures, motivating experimentation and innovation. A case study that reveals this is the vision transformer, which brings together many of the cases we have already covered. Its explanations (Khan et al., 2022, See Figure 2) suffer from the same issues as explanations of the original transformer (See Section 1.2), made worse by even more axes being present. With neural circuit diagrams, visual attention mechanisms are as simple as replacing the $\overline{y}$ and $\overline{x}$ axes in Figure 11 with tandem $\overline{\mathbf{y}}$ and $\overline{\mathbf{x}}$ axes and setting $h = 1$. As $1 \times 1$ convolutions are simply the identity, broadcasting a linear map over all of $\overline{\mathbf{y}}$ pixels is a $1 \times 1$-convolution. This leaves us with Figure 17 for a visual attention mechanism. It is highly suggestive, calling us to experiment with the convolutions' stride, dilation, and kernel sizes, potentially streamlining models. The diagram clarifies how to implement multi-head visual attention with $h \neq 1$, especially using einsum similarly to Figure 7. Additionally, $\overline{\mathbf{y}}$ does not need to match $\overline{\mathbf{x}}$. We could have $\overline{\mathbf{y}}$ be image data, and $\overline{x}$ be textual data without convolutions. This case study shows how neural circuit diagrams reveal the degrees of freedom of architectures, and therefore motivate innovation, all while being precise in how algorithms should be implemented.

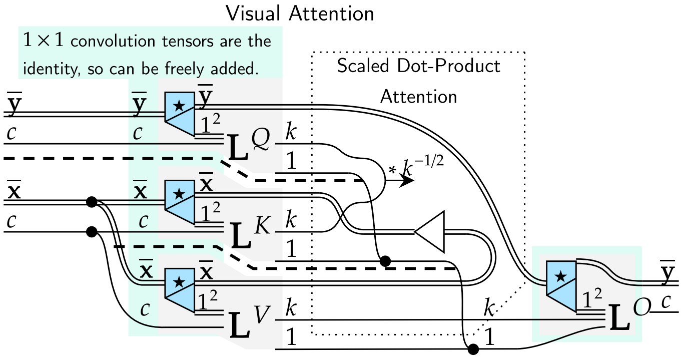

Visual Attention
1 × 1 convolution tensors are the identity, so can be freely added.
Scaled Dot-Product Attention
ȳ
c
x̄
c
$1^2$
L Q
L K
L V
L O
k
1
$*k^{-1/2}$

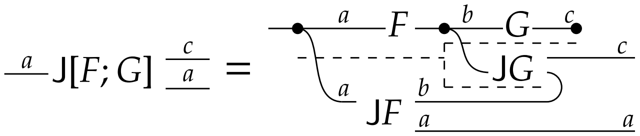
Figure 17: Using neural circuit diagrams, visual attention (Dosovitskiy et al., 2021) is shown to be a simple modification of multi-head attention (See Figure 11, Figure 7, Cell 17, Jupyter notebook.)

### 3.6 Differentiation: A Clear Improvement over Prior Methods

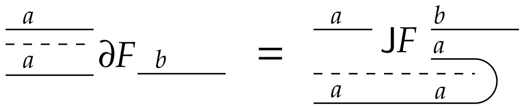
In addition to more clearly communicating architectures, neural circuit diagrams can be used for robust mathematical analysis. An example of such a use is for differentiation, where the combination of tensor and Cartesian products makes for a particularly pleasant presentation of the chain rule. This is important to deep learning, where differentiation is critical to understanding information flows through architectures (He et al., 2016). Our graphical diagrams of differentiation are more clear and compact than prior methods (Shiebler et al., 2021; Cockett et al., 2019) and does not require the introduction of concepts unfamiliar to the reader (Chiang et al., 2023). Previous graphical approaches, furthermore, have been unable to represent differentiation (Xu & Maruyama, 2022). The chain rule is particularly interesting to us, given its close tie to backpropagation. Using neural circuit diagrams, we can derive the linear memory cost of backpropagation, a simple result that hints at the utility of neural circuit diagrams as an analytical tool over other methods of blueprinting architectures.

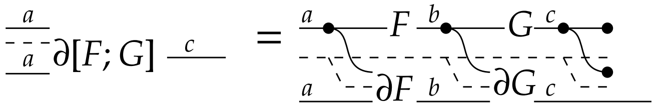
We consider shaped data over a field that accepts differentiation, such as $\mathbb{R}$. We consider the subcategory of once-differentiable functions, $\mathbf{C}^1\mathbf{ShD}(\mathbb{R})$. As they are once-differentiable, the Jacobian of each morphism, of the form $\mathsf{J}F : a \to b \underline{\times} a$, exists in $\mathbf{ShD}(\mathbb{R})$. Composing $F_b^a$ with $G_c^b$, we could draw the chain rule for $\mathsf{J}[F;G]$ as follows;

$$\frac{\partial}{\partial x_a}(GF)\bigg|_{\mathbf{x}} = \frac{\partial}{\partial x_b}G\bigg|_{F(\mathbf{x})} \cdot \frac{\partial}{\partial x_a}F^b\bigg|_{\mathbf{x}}$$

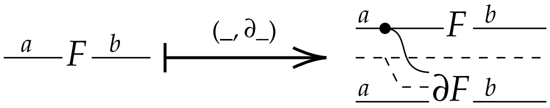


This is clearly unacceptable. Instead, we transpose $\mathsf{J}F : a \to b \underline{\times} a$ to get the (forward) derivative $\partial F : a \underline{+} a \to b$ . We then get an elegant form of the chain rule that efficiently scales with more operations. This form of the derivative aligns with Cockett et al. (2019)'s definition 4.

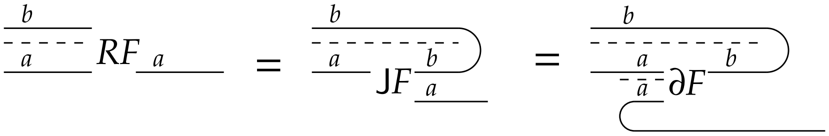




This naturally scales with depth. Over the subcategory of once differentiable morphisms, $\mathbf{C^1ShD}(\mathbb{R})$, we define the derivation functor $(\_, \partial\_) : \mathbf{C^1ShD}(\mathbb{R}) \to \mathbf{ShD}(\mathbb{R})$ such that $(\_, \partial\_)$ maps shapes $a \mapsto a \underline{+} a$ and once-differentiable morphisms $F_b^a \mapsto (F, \partial F)_{b \underline{+} b}^{a \underline{+} a}$ such that;



Functors must respect composition, which is ensured by the chain rule. In machine learning, we are often interested in backpropagation, which employs the reverse derivative (Shiebler et al., 2021; Cockett et al., 2019), another transpose of the Jacobian. As the Jacobian, reverse derivative, and forward derivatives differ by transposes, neural circuit diagrams are in a unique position to represent the relationship between them. Using the rules of linear algebra we have developed, the reverse derivative can be defined as;



Backpropagation is of particular interest to us. Backpropagation uses the reverse derivative, so we need to rearrange chained forward derivatives into their reverse form. This can be achieved using the rules of linear algebra we have developed, including the bifunctor property of tensor products and the Kronecker delta-dot product identity. This is done in Figure 18, where we see that the reverse derivative rearrangement of a derivative chain requires a number of tuple segments linear to the number of layers. This derives the memory cost of backpropagation as linear to the number of layers.

We see that neural circuit diagrams reflect both underlying mathematical equivalences as well as algorithmic properties, distinguishing between algorithms that yield the same results with different complexity. This is a distinct advantage over symbolic methods (Goodfellow et al., 2016) or code (Phuong & Hutter, 2022) where these properties require separate derivation. Additionally, reformulations may improve the performance of algorithms (Xu et al., 2023), further motivating the use of neural circuit diagrams during the design and analysis process.

## 4 Conclusion

In this paper, we advocated for using neural circuit diagrams to communicate deep learning research. We discussed the need for improved communication, why category theory is the most promising approach to defining a graphical language, and the issues that had to be resolved. Then, we constructed a category that synthesized Cartesian and tensor approaches. This category combines the flexibility of Cartesian products with the tools of linear tensor products. We then diagrammed several architectures in quick succession, showing the utility of neural circuit diagrams for efficiently and precisely communicating models. In the case of vision transformers, we showed how neural circuit diagrams invite further innovation, a distinct

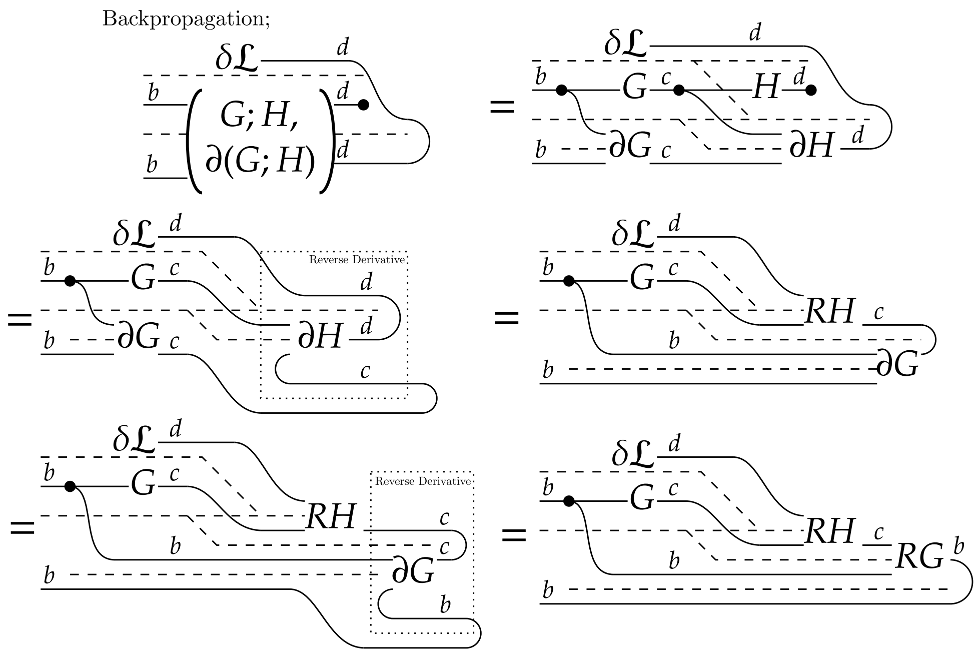


Figure 18: The derivative of a chain of functions summed over some change gives backpropagation. After rearranging the expression, every layer in the chain requires its own tuple. The number of simultaneous tuple segments, and hence memory consumption, increases linearly with the number of layers. Hence, backpropagation has memory consumption linear to the number of layers.

advantage over ad-hoc approaches that obfuscate the connection between architectures. In addition to communicating architectures, we showed how neural circuit diagrams can be used to analyze processes' mathematical foundations and derive algorithms' properties.

This paper naturally invites further work. Neural circuit diagrams can be further standardized and applied to new situations. Neural circuit diagrams are based on category theory, meaning each diagram is a robust mathematical expression. This bridges practical models with their mathematical foundations, motivating these to be further explored. A particularly promising avenue is combining deep learning with probabilistic models using Markov categories (Perrone, 2022; Fritz et al., 2023), or generalizing broadcasting further. This paper targets machine learning researchers, and we feel it is in the sweet spot between a purely practical guide and a treatise on mathematical foundations. We hope neural circuit diagrams can motivate work on both of these fronts and become a valuable tool for the deep learning community.

### 4.0.1 Acknowledgements

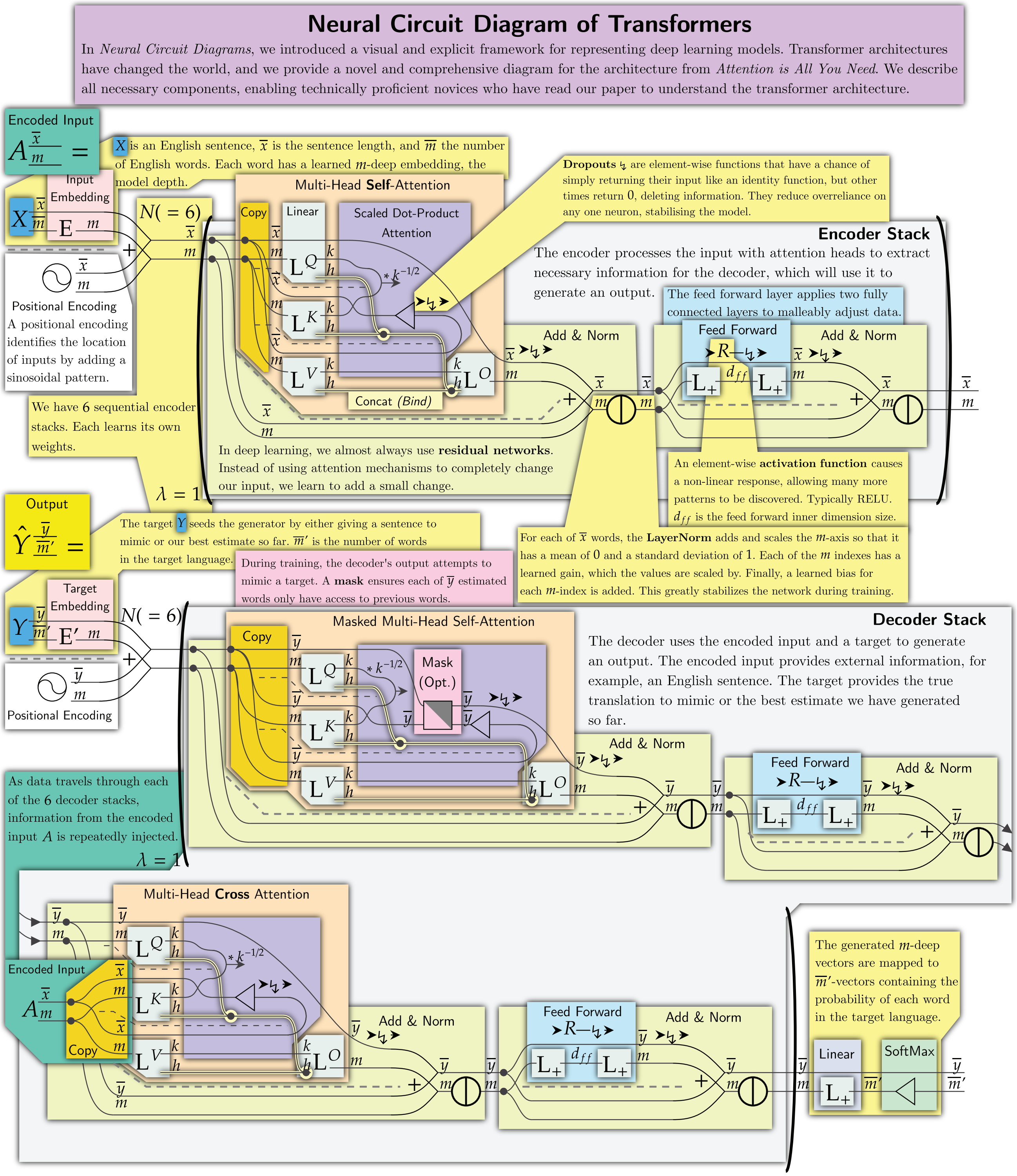
Mathcha was used to write equations and draw diagrams. The Harvard NLP annotated transformer was invaluable for drawing Figure 19.

## References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization, July 2016. URL http://arxiv.org/abs/1607.06450. arXiv:1607.06450 [cs, stat].

John C. Baez and Mike Stay. Physics, Topology, Logic and Computation: A Rosetta Stone. volume 813, pp. 95–172. 2010. doi: 10.1007/978-3-642-12821-9_2. URL http://arxiv.org/abs/0903.0340. arXiv:0903.0340 [quant-ph].

Jacob Biamonte and Ville Bergholm. Tensor Networks in a Nutshell, July 2017. URL http://arxiv.org/abs/1708.00006. arXiv:1708.00006 [cond-mat, physics:gr-qc, physics:hep-th, physics:math-ph, physics:quant-ph].



**Neural Circuit Diagram of Transformers**

In *Neural Circuit Diagrams*, we introduced a visual and explicit framework for representing deep learning models. Transformer architectures have changed the world, and we provide a novel and comprehensive diagram for the architecture from *Attention is All You Need.* We describe all necessary components, enabling technically proficient novices who have read our paper to understand the transformer architecture.

$X$ is an English sentence, $\overline{x}$ is the sentence length, and $\overline{m}$ the number of English words. Each word has a learned $m$-deep embedding, the model depth.

**Dropouts** ↯ are element-wise functions that have a chance of simply returning their input like an identity function, but other times return 0, deleting information. They reduce overreliance on any one neuron, stabilising the model.

**Encoder Stack**

The encoder processes the input with attention heads to extract necessary information for the decoder, which will use it to generate an output.

The feed forward layer applies two fully connected layers to malleably adjust data.

Positional Encoding

A positional encoding identifies the location of inputs by adding a sinosoidal pattern.

We have 6 sequential encoder stacks. Each learns its own weights.

In deep learning, we almost always use **residual networks**. Instead of using attention mechanisms to completely change our input, we learn to add a small change.

An element-wise **activation function** causes a non-linear response, allowing many more patterns to be discovered. Typically RELU. $d_{ff}$ is the feed forward inner dimension size.

For each of $\overline{x}$ words, the **LayerNorm** adds and scales the $m$-axis so that it has a mean of 0 and a standard deviation of 1. Each of the $m$ indexes has a learned gain, which the values are scaled by. Finally, a learned bias for each $m$-index is added. This greatly stabilizes the network during training.

The target $Y$ seeds the generator by either giving a sentence to mimic or our best estimate so far. $\overline{m}'$ is the number of words in the target language.

During training, the decoder's output attempts to mimic a target. A **mask** ensures each of $\overline{y}$ estimated words only have access to previous words.

**Decoder Stack**

The decoder uses the encoded input and a target to generate an output. The encoded input provides external information, for example, an English sentence. The target provides the true translation to mimic or the best estimate we have generated so far.

As data travels through each of the 6 decoder stacks, information from the encoded input $A$ is repeatedly injected.

The generated $m$-deep vectors are mapped to $\overline{m}'$-vectors containing the probability of each word in the target language.

Figure 19: The fully diagrammed architecture from *Attention is All You Need* (Vaswani et al., 2017).

Michelle A. Borkin, Zoya Bylinskii, Nam Wook Kim, Constance May Bainbridge, Chelsea S. Yeh, Daniel Borkin, Hanspeter Pfister, and Aude Oliva. Beyond Memorability: Visualization Recognition and Recall. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):519–528, January 2016. ISSN 1941-0506. doi: 10.1109/TVCG.2015.2467732. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

David Chiang, Alexander M. Rush, and Boaz Barak. Named Tensor Notation, January 2023. URL http://arxiv.org/abs/2102.13196. arXiv:2102.13196 [cs].

Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin, and Dorette Pronk. Reverse derivative categories, October 2019. URL http://arxiv.org/abs/1910.07065. arXiv:1910.07065 [cs, math].

Mostafa Dehghani, Basil Mustafa, Josip Djolonga, Jonathan Heek, Matthias Minderer, Mathilde Caron, Andreas Steiner, Joan Puigcerver, Robert Geirhos, Ibrahim Alabdulmohsin, Avital Oliver, Piotr Padlewski, Alexey Gritsenko, Mario Lučić, and Neil Houlsby. Patch n' Pack: NaViT, a Vision Transformer for any Aspect Ratio and Resolution, July 2023. URL http://arxiv.org/abs/2307.06304. arXiv:2307.06304 [cs].

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, June 2021. URL http://arxiv.org/abs/2010.11929. arXiv:2010.11929 [cs].

Chris Drummond. Replicability is not reproducibility: Nor is it good science. *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, January 2009.

Brendan Fong and David I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press, 1 edition, July 2019. ISBN 978-1-108-66880-4 978-1-108-48229-5 978-1-108-71182-1. doi: 10.1017/9781108668804. URL https://www.cambridge.org/core/product/identifier/9781108668804/type/book.

Brendan Fong, David I. Spivak, and Rémy Tuyéras. Backprop as Functor: A compositional perspective on supervised learning, May 2019. URL http://arxiv.org/abs/1711.10455. arXiv:1711.10455 [cs, math].

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, June 2023. ISSN 03043975. doi: 10.1016/j.tcs.2023.113896. URL http://arxiv.org/abs/2010.07416. arXiv:2010.07416 [cs, math, stat].

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

John Hayes and Diana Bajzek. Understanding and Reducing the Knowledge Effect: Implications for Writers. *Written Communication*, 25:104–118, January 2008.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. URL http://arxiv.org/abs/1512.03385. arXiv:1512.03385 [cs].

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks, July 2016. URL http://arxiv.org/abs/1603.05027. arXiv:1603.05027 [cs].

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models, December 2020. URL http://arxiv.org/abs/2006.11239. arXiv:2006.11239 [cs, stat].

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, March 2015. URL http://arxiv.org/abs/1502.03167. arXiv:1502.03167 [cs].

Sayash Kapoor and Arvind Narayanan. Leakage and the Reproducibility Crisis in ML-based Science, July 2022. URL http://arxiv.org/abs/2207.07048. arXiv:2207.07048 [cs, stat].

Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in Vision: A Survey. *ACM Computing Surveys*, 54(10s):1–41, January 2022. ISSN 0360-0300, 1557-7341. doi: 10.1145/3505244. URL https://dl.acm.org/doi/10.1145/3505244.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017. ISSN 0001-0782, 1557-7317. doi: 10.1145/3065386. URL https://dl.acm.org/doi/10.1145/3065386.

Minhyeok Lee. GELU Activation Function in Deep Learning: A Comprehensive Mathematical Analysis and Performance, May 2023. URL http://arxiv.org/abs/2305.12073. arXiv:2305.12073 [cs].

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A Survey of Transformers, June 2021. URL http://arxiv.org/abs/2106.04554. arXiv:2106.04554 [cs].

Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. Pay Attention to MLPs, June 2021. URL http://arxiv.org/abs/2105.08050. arXiv:2105.08050 [cs].

Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the Effective Receptive Field in Deep Convolutional Neural Networks, January 2017. URL https://arxiv.org/abs/1701.04128v2.

José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, October 1990. ISSN 0890-5401. doi: 10.1016/0890-5401(90)90013-8. URL https://www.sciencedirect.com/science/article/pii/0890540190900138.

T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. ISSN 1558-2256. doi: 10.1109/5.24143. Conference Name: Proceedings of the IEEE.

Alex Nichol and Prafulla Dhariwal. Improved Denoising Diffusion Probabilistic Models, February 2021. URL http://arxiv.org/abs/2102.09672. arXiv:2102.09672 [cs, stat].

Paolo Perrone. Markov Categories and Entropy, December 2022. URL http://arxiv.org/abs/2212.11719. arXiv:2212.11719 [cs, math, stat].

Mary Phuong and Marcus Hutter. Formal Algorithms for Transformers, July 2022. URL http://arxiv.org/abs/2207.09238. arXiv:2207.09238 [cs].

S. Pinker. *The sense of style: The thinking person's guide to writing in the 21st century*. Penguin Publishing Group, 2014. ISBN 978-0-698-17030-8. URL https://books.google.com.au/books?id=FzRBAwAAQBAJ.

Edward Raff. A Step Toward Quantifying Independently Reproducible Machine Learning Research. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/c429429bf1f2af051f2021dc92a8ebea-Abstract.html.

Alex Rogozhnikov. Einops: Clear and Reliable Tensor Manipulations with Einstein-like Notation. October 2021. URL https://openreview.net/forum?id=oapKSVM2bcj.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models, April 2022. URL http://arxiv.org/abs/2112.10752. arXiv:2112.10752 [cs].

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015. URL http://arxiv.org/abs/1505.04597. arXiv:1505.04597 [cs].

Lee Ross, David Greene, and Pamela House. The "false consensus effect": An egocentric bias in social perception and attribution processes. *Journal of Experimental Social Psychology*, 13(3):279–301, 1977.

Sadoski. Impact of concreteness on comprehensibility, interest. *Journal of Educational Psychology*, 85(2): 291–304, 1993.

Peter Selinger. A survey of graphical languages for monoidal categories, August 2009. URL https://arxiv.org/abs/0908.3347v1.

Dan Shiebler, Bruno Gavranović, and Paul Wilson. Category Theory in Machine Learning, June 2021. URL http://arxiv.org/abs/2106.07032. arXiv:2106.07032 [cs].

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway Networks, November 2015. URL http://arxiv.org/abs/1505.00387. arXiv:1505.00387 [cs].

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive Network: A Successor to Transformer for Large Language Models, August 2023. URL http://arxiv.org/abs/2307.08621. arXiv:2307.08621 [cs].

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, December 2017. URL http://arxiv.org/abs/1706.03762. arXiv:1706.03762 [cs].

Paul Wilson and Fabio Zanasi. Categories of Differentiable Polynomial Circuits for Machine Learning, May 2022. URL http://arxiv.org/abs/2203.06430. arXiv:2203.06430 [cs, math].

Tom Xu and Yoshihiro Maruyama. Neural String Diagrams: A Universal Modelling Language for Categorical Deep Learning. In Ben Goertzel, Matthew Iklé, and Alexey Potapov (eds.), *Artificial General Intelligence*, Lecture Notes in Computer Science, pp. 306–315, Cham, 2022. Springer International Publishing. ISBN 978-3-030-93758-4. doi: 10.1007/978-3-030-93758-4_32.

Yao Lei Xu, Kriton Konstantinidis, and Danilo P. Mandic. Graph Tensor Networks: An Intuitive Framework for Designing Large-Scale Neural Learning Systems on Multiple Domains, March 2023. URL http://arxiv.org/abs/2303.13565. arXiv:2303.13565 [cs].

Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2528–2535, San Francisco, CA, USA, June 2010. IEEE. ISBN 978-1-4244-6984-0. doi: 10.1109/CVPR.2010.5539957. URL http://ieeexplore.ieee.org/document/5539957/.