# 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 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. A lingering issue with existing diagramming methods is the inability to simultaneously express the detail of axes and the free arrangement of data, which neural circuit diagrams solve. Their compositional structure is analogous to code, creating a close correspondence between diagrams and implementation.

In this work, we introduce neural circuit diagrams for an audience of machine learning researchers. After introducing neural circuit diagrams, we cover a host of architectures to show their utility and breed familiarity. This includes the transformer architecture, convolution (and its difficult-to-explain extensions), residual networks, the U-Net, and the vision transformer. We include a Jupyter notebook that provides evidence for the close correspondence between diagrams and code. Finally, we examine backpropagation using neural circuit diagrams. We show their utility in providing mathematical insight and analyzing algorithms' time and space complexities.

## 1 Introduction

### 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).

Furthermore, 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 the 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. Value and key vectors come in pairs. 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).

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

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

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

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). Deter-

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

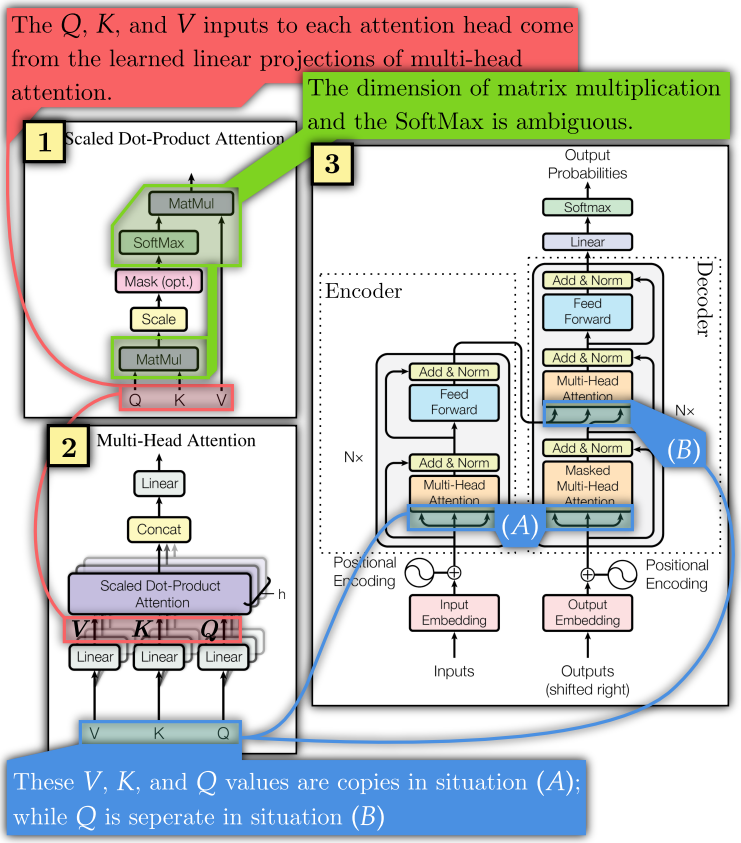



Figure 1: The diagrams from the original transformer model with our annotations. Critical information is missing regarding the origin of $Q$, $K$, and $V$ values (**red** and **blue**), and the axes over which operations act (**green**).

mining 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 & 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 simply 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 22, 23, and 38.

### 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.

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 in research and industry (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. Robust diagrams, therefore, can benefit both the communication and performance of architectures. 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 this approach 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.

### 1.4 The Philosophy of Our Approach

As we are introducing these diagrams, we have a burden to explain how we think they should be used, and to address criticisms of creating a diagramming standard in the first place. We will take a brief aside to address these points, which we believe will aid in the adoption of neural circuit diagrams.

These diagrams are intended to express sequential-tensor deep learning models. This is in contrast to machine learning or artificial intelligence systems more generally. Deep learning models are machine learning models with sequential data processing through neural network layers. We do not cover recursive or branching

models in this work. Furthermore, we assume data is always in the form of tuples of tensors. Generalizing diagrams to further contexts is an exciting avenue for future research.

By making these assumptions, we develop diagrams specialized for some of the most essential but difficult-to-explain systems in artificial intelligence research. Researchers outside the narrow scope of sequential-tensor deep learning models often rely on these tools. By more clearly communicating them, researchers who may not be up to date on the latest innovations or aware of their options stand to benefit an immense deal.

We do not expect two independent teams to diagram architectures the exact same way. Indeed, we do not believe the appropriate diagramming framework would have this property. Diagrams should have the flexibility to allow for innovations and to appeal to the audience's level of knowledge. The benefit of our framework, instead, is to have robust diagrams that include all the necessary details with clear correspondence to implementation and analysis, in contrast to ad-hoc diagrams, which often fail to include critical information.

Our diagrams can be decomposed into sections that allow for layered abstraction. The exact details of code can be abstracted into single-symbol components. Sections of diagrams can be highlighted for the reader's clarity, and repeated patterns can be defined as components. Diagrams have immense compositional structure. The horizontal axis represents sequential composition, and the vertical axis represents parallel composition. Sections and components can be joined like Lego bricks to construct models.

This sectioning allows for a close correspondence between diagrams and implementation. Every highlighted section becomes a module in code. Diagrams, therefore, provide a cross-platform blueprint for architectures. This allows implementations to be cross-checked to a reference, increasing reliability. Furthermore, which components are abstracted and the level of abstraction can vary depending on the audience, leading to clearer, specialized communication.

A common criticism is that the introduction of a new standard simply increases the number of standards, worsening the issue trying to be solved. We do not believe this is a relevant critique for deep learning diagrams. Currently, there are no standard diagramming methods. Every paper, in a sense, has its own ad-hoc diagramming scheme. Compared to this, neural circuit diagrams only need to be learned once, after which architectures can be clearly and explicitly explained. Furthermore, they build on existing research on robust monoidal string diagrams, which have been found to be a universal standard for various fields (Baez & Stay, 2010).

### 1.5 Our Contribution

To address the need for more robust communication and analysis of deep learning architectures, we introduce neural circuit diagrams. Neural circuit diagrams solve the lingering challenge of accommodating both the details of axes (the tensor approach) and the free arrangement of data (the Cartesian approach) in diagrams. They are specialized for sequential algorithms on memory states consisting of tuples of tensors.

Diagramming the details of axes means the shape of data is clear throughout a model. They easily show broadcasting and provide a graphical calculus to rearrange linear functions into equivalent forms. At the same time, they clearly represent tuples, copying, and deletion, processes that typical graphical methods struggle with. This makes them uniquely capable of accurately representing deep learning models.

Inspired by category theory and especially monoidal string diagrams (Selinger, 2009; Baez & Stay, 2010), this work builds on a literature of robust diagramming methods. However, the category theory details are omitted to maximize impact among machine learning researchers.

The benefits of neural circuit diagrams are many. They allow for clearer communication of new developments, making ideas more rapidly disseminated and understood. They offer robust blueprints for designing and implementing models, accelerating innovation and streamlining productivity. Furthermore, they allow for rigorous mathematical analysis of architectures, bringing us closer to a theoretical understanding of deep learning.

These points are evidenced by diagramming a host of architectures. We cover a basic multi-layer perceptron, the transformer architecture, convolution (and its difficult-to-explain permutations), the identity ResNet,

the U-Net, and the vision transformer. We provide a Jupyter notebook that implements these diagrams, which provides further evidence for the close relationship between diagrams and implementation. Finally, we offer a novel analysis of backpropagation, which shows the utility of neural circuit diagrams for rigorous analysis of architectures.

## 2 Neural Circuit Diagrams for Deep Learning

### 2.1 Commutative Diagrams

We aim to make diagrams that robustly represent deep-learning algorithms. While our diagrams will eventually be generalized, we will initially concentrate on common models. Specifically, we will explore models that successively process data of predictable types. To facilitate understanding, we will introduce diagrams of gradually increasing complexity. First, let's delve into an intuitive diagram, where symbols represent data types, and arrows signify the functions.

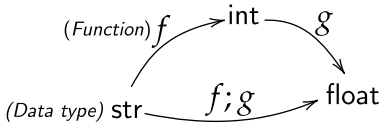



Figure 2: We have two functions: $f : \text{str} \to \text{int}$ and $g : \text{int} \to \text{float}$. These functions can be composed into a single function $f; g : \text{str} \to \text{float}$ (where we use forward ; instead of reverse $\circ$ composition notation, $x; f; g = g(f(x))$. We represent data types, such as str, int, and float, with floating symbols, while functions are denoted by arrows connecting them.

#### 2.1.1 Tuples and Memory

Algorithms are rarely composed of operations on a single variable. Instead, their steps involve operations on memory states composed of multiple variables. The data type of memory state is a tuple of the variables that compose it. Consider a single algorithmic step acting on memory, which may appear as $f : A \times B \times C \to A \times D$, even if $f$ solely manipulates $B \times C$ to produce $D$. In this step, the initial memory state includes variables of types $A$, $B$, and $C$, which are transformed into a modified memory state containing variables of types $A$ and $D$. We can represent an algorithm comprised of two such steps, $f : A \times B \times C \to A \times D$ and $g : A \times D \to E$, as illustrated below in Figure 3.

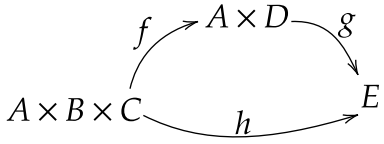



Figure 3: We have an algorithm $h : A \times B \times C \to E$ composed of two steps, $f : A \times B \times C \to A \times D$ and $g : A \times D \to E$. A memory state is a tuple of variables. For steps to compose, their input and output types must match. Every function, therefore, has to act on the entire memory state.

### 2.2 String Diagrams

However, these commuting arrow diagrams fall short. As algorithms scale, operations and memory states get more complex. Operations usually only act on some variables. However, it is not clear how these targeted functions can be easily represented. Compound data types and compound functions are better suited by reorienting diagrams as in Figure 4. We will have horizontal wires represent types, and symbols represent functions. Diagrams are forced to horizontally go left to right.

This reorientation allows us to separate compound types and functions easily. We can diagram tupled types $A \times B$ as a wire for $A$ and a wire for $B$ vertically stacked but separated by a dashed line. For increased clarity, we can draw boxes around functions.

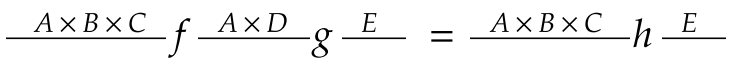



Figure 4: We reorient diagrams to go left to right. Wires represent data types, and symbols represent functions.

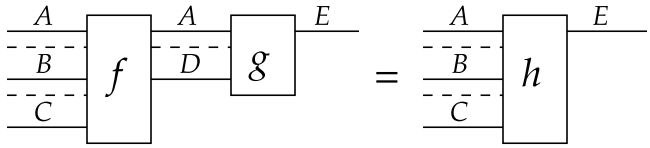



Figure 5: Tupled data types are diagrammed with wires separated by dashed lines. This is needed to show the precise actions of every algorithm step and ensure the steps can composed.

Functions often only act on some variables in memory, leaving the others alone. We represent this by having those functions consume their dependent variables and letting the other type-wires flow past them. This results in a function diagrammed by only acting on its dependent variables but with an overall vertical section that consumes and produces the entire memory state.

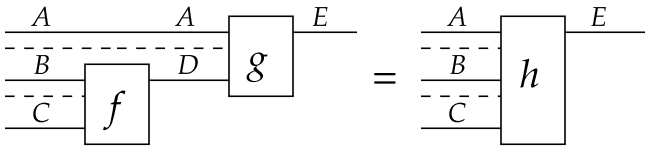



Figure 6: Functions only need to be shown acting on their dependent variables, and having the other wires pass through.

### 2.2.1 Interpreting Diagrams

Diagrams can be interpreted as having every vertical section represent something. Either the data types present in memory or a function acting on memory. A well-composed diagram has every vertical function section match adjacent memory states. This ensures that every step of an algorithm can be performed. Furthermore, diagrams can be split into constituent parts, each of which is composed with others. Vertical composition requires dashed lines, and horizontal composition requires wires matching like jigsaw indents.

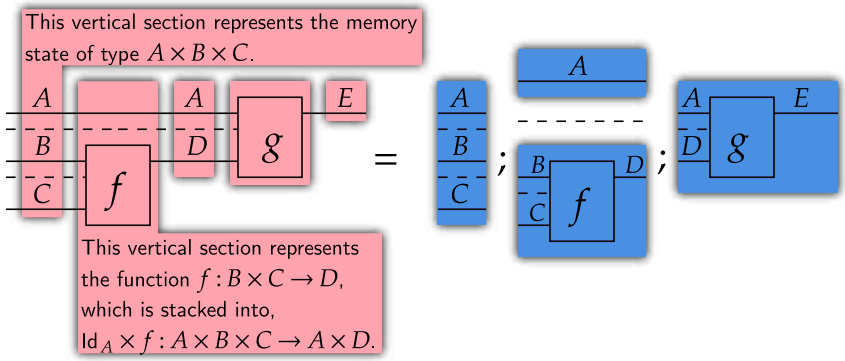



Figure 7: We can view diagrams as composed vertical sections, each representing a memory state or a function. Vertical sections can be further separated in the case of tuple-combined sections.

## 2.3 Tensors

We will specialize our diagrams for deep learning models whose memory states are tuples of tensors. We can generalize diagrams in the future. Tensors are numbers arranged along axes. So, a scalar $\mathbb{R}$ is a rank 0

tensor, a vector $\mathbb{R}^3$ is a rank 1 tensor, and a table $\mathbb{R}^{4\times3}$ is rank 2 tensor, and so on. With tensor data types, we may get something like Figure 8.

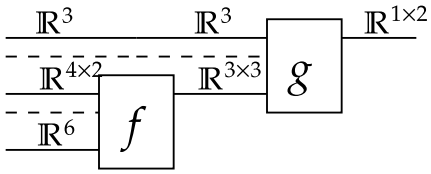



Figure 8: Similar to Figure 6, but with data types being tensors.

Just like we can represent $A \times B$ as either a single wire labeled $A \times B$ or dash-separated $A$ and $B$ wires, we have a specialized alternative representation of tensors $\mathbb{R}^{a\times b}$. We write a tensor $\mathbb{R}^{a\times b}$ as two vertically stacked wires labelled $a$ and $b$, without a dashed separation. For many deep learning models, assuming that data types are tensors is safe, so this does not limit expressiveness. This lets us rediagram Figure 8 as Figure 9. We also note how the diagram is composed of subdiagrams that accept and produce various memory states.

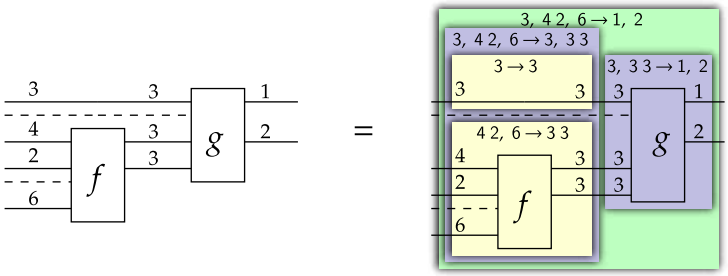



Figure 9: We can redraw Figure 8 into the form on the left. This diagram can be broken into many constituent parts, shown on the right. An exercise to better understand a diagram is to break it into these constituent parts. (*See cell 3, Jupyter notebook.*)

### 2.3.1 Indexes

Every tensor has indexes along each axis which access values. For example, $\mathbb{R}^3$ has three indexes, each extracting a value. $\mathbb{R}^{4\times2}$ has 4 indexes along the first axis, and 2 along the second, requiring one of each to access a value. For $\mathbb{R}^{4\times2}$, the indexes of the 4 axis have the form $\mathbb{R}^{4\times2} \to \mathbb{R}^2$. This represents extracting a $\mathbb{R}^2$-sized row from a $\mathbb{R}^{4\times2}$ table. We start indexing from 0, and diagram indexes by kets $|\_\rangle$. For tensors, they obey the regularity conditions of Figure 11. Note how indexes only act on one axis, so we let the others "pass-through".

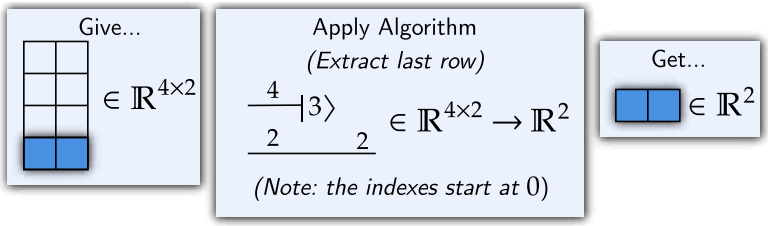



Figure 10: An example of how an index extracts a sub-tensor and is diagrammed by having the other axes "pass-through". Assigning one axis as rows and another as columns is arbitrary. (*See cell 5, Jupyter notebook.*)

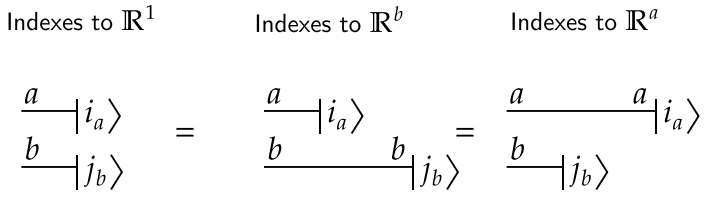



Figure 11: For a $\mathbb{R}^{a\times b}$ matrix, we have $a \times b$ separate indexes to $\mathbb{R}$. These are defined in relation to indexes on $\mathbb{R}^a$ and $\mathbb{R}^b$. The order of access does not change the values. We use the symbols $i_a$ to iterate over the indexes of $a$, and $j_b$ for those of $b$. Tensors are recursive, so the above diagram works for the indexes of $a = 4 \times 2$ and $b = 3$. The action of indexes on higher-order tensors is therefore well-defined. (*See cell 7, Jupyter notebook.*)

### 2.3.2 Broadcasting

Broadcasting a function over an axis applies it to each axis element. It is best to use an example. The vector norm is a function on vectors of some length. Over $\mathbb{R}^3$, it maps $\text{Norm} : \mathbb{R}^3 \to \mathbb{R}^1$. If we had 5 vectors $\mathbb{R}^3$ arranged into a table, we have a tensor $\mathbb{R}^{5\times3}$. Taking the norm of each vector in this table, we get a function $\text{Norms} : \mathbb{R}^{5\times3} \to \mathbb{R}^{5\times1}$.

However, this function's 5-length axis is special in that it is broadcast over. The operation $\text{Norms} : \mathbb{R}^{5\times3} \to \mathbb{R}^{5\times1}$ can be fully understood by the underlying function $\text{Norm} : \mathbb{R}^3 \to \mathbb{R}^1$, then simply applying it over the 5-length axis. So, we diagram Norms by drawing Norm and letting the 5-axis pass over it.

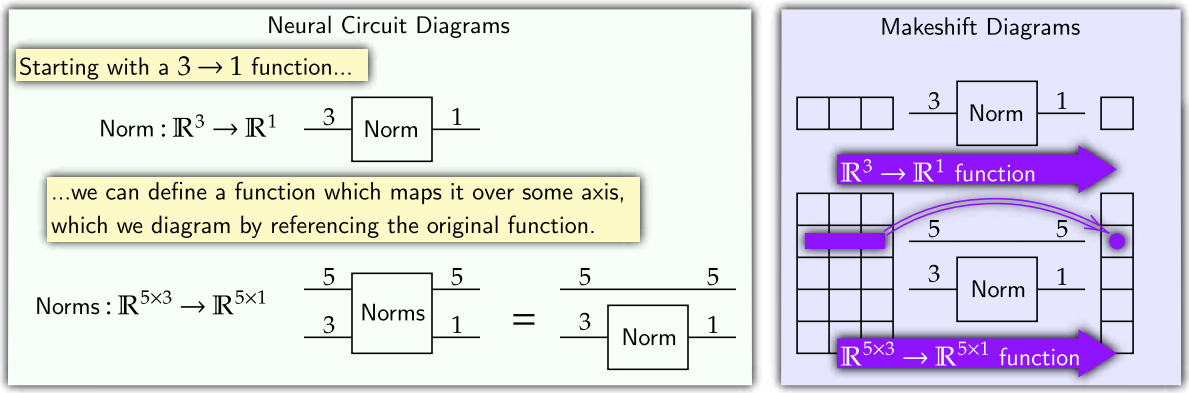



Figure 12: We can take an operation on rows and broadcast it over columns, giving an operation on an entire table. This can be diagrammed by having the column axis pass through. Thinking of one axis as rows and another as columns is arbitrary.

Formally, we define broadcasting with respect to the indexes. The indexes provide complete descriptions of functions. This means a function $\mathbb{R}^a \to \mathbb{R}^b$ can be uniquely identified by the $b$ index values it produces for each input. For example, if two functions $f, g : \mathbb{R}^4 \to \mathbb{R}^3$ always give the same $|0\rangle$, $|1\rangle$, and $|2\rangle$ index values, the functions are the same. This scales to the $a \times b$ indexes of tensors. So, if we define a broadcasted function for the indexes and a known function, it is uniquely defined. We do this with Figure 13. We also define inner broadcasting, which operates within a tuple segment in Figure 14.

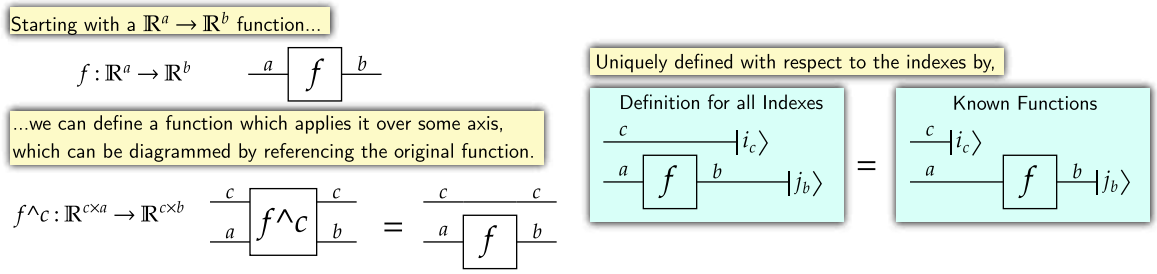



Figure 13: A generic presentation of broadcasting. Any function can be broadcast over some axis. This can be diagrammed with reference to the original function and an axis that passes through. We can define the broadcasted function by defining each of its output index values. (*See cell 9, Jupyter notebook.*)

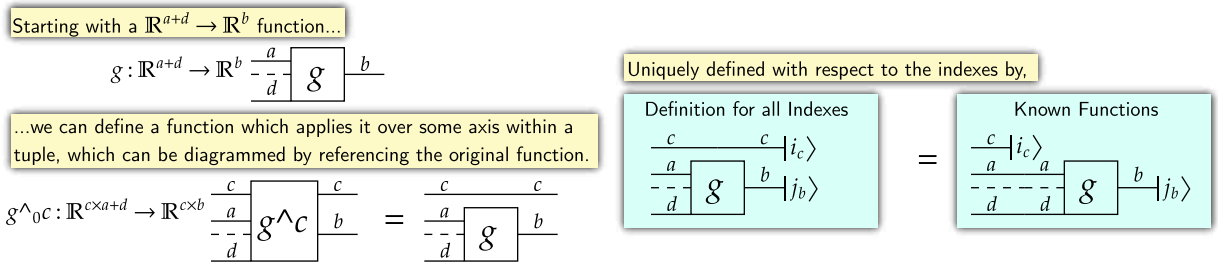



Figure 14: Inner broadcasting is defined within tuples. Similar to Figure 13, we define an inner broadcast by associating each index of an inner broadcasted function with a known function. (*See cell 11, Jupyter notebook.*)

### 2.3.3 Element-wise Operations

Tensors such as $\mathbb{R}^{1\times a}$ with a 1-length axis are accessed in the same ways as $\mathbb{R}^a$. We can freely introduce and remove 1-length axes. We do this explicitly by having entering or exiting arrows for wires. If a function is known to map to or from $\mathbb{R}^1$, we may leave out drawing the 1-length axis altogether. An application of this trick is with element-wise operations. Functions $f : \mathbb{R}^1 \to \mathbb{R}^1$ can be broadcast over a tensor $\mathbb{R}^a$ to get an operation $\mathbb{R}^{1\times a} \to \mathbb{R}^{1\times a}$, which corresponds to applying $f$ onto each element of the tensor. We diagram element-wise operations as in Figure 15.

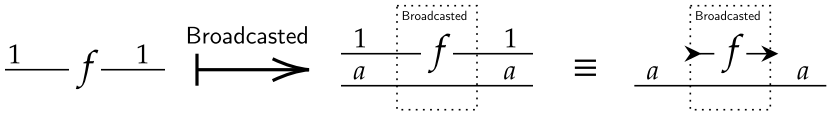



Figure 15: Broadcasting a $f : \mathbb{R} \to \mathbb{R}$ function over some shape gives an element-wise operation. *(See cell 13, Jupyter notebook.)*

## 2.4 Linearity

Linear functions are an important class of operations for deep learning. Linear functions can be highly parallelized, especially with GPUs. Previous works have shown how graphically modeling linear functions, and reimplementing algorithms can improve performance (Xu et al., 2023). Linear functions have immense regularity. Standard monoidal string diagrams rely on these properties to provide elegant graphical languages for various fields (Baez & Stay, 2010).

However, a pure monoidal string diagram has difficulty representing non-linear operations, a noted issue (Chiang et al., 2023). Our framework has Cartesian products and broadcasting, which are not generally analogous to how monoidal string diagrams combine linear functions. However, if we know functions are linear, we can use diagrams to efficiently reason about algorithms. By focusing on linear functions, we can take advantage of their parallelization properties.

Linear functions are required to obey additivity and homogeneity, as shown in Figure 16. These operations are closed under composition, so applying linear maps onto each other gives another linear map. Importantly to us, they are natural with respect to broadcasting. This means for any two linear functions $f$ and $g$, the equality in Figure 17 holds. This means they can be simultaneously broadcast. This lets a series of linear functions be efficiently parallelized and flexibly rearranged.

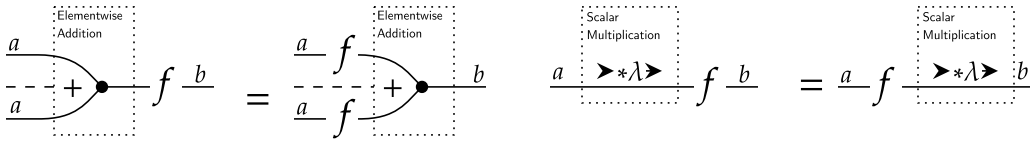



Figure 16: A subset of functions between $\mathbb{R}^a$ to $\mathbb{R}^b$ are linear, obeying additivity and homogeneity. This class of functions are closed under composition and has many important composition properties.

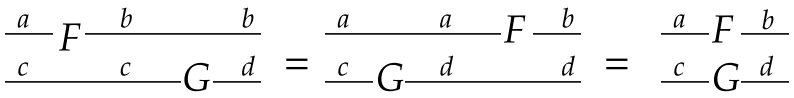



Figure 17: Linear functions are natural with respect to each other and broadcasting. This means the above equality holds, letting expressions be flexibly rearranged.

### 2.4.1 Multilinearity

It is important to note the distinction between linearity and multilinearity. Inner products, for example, are multilinear. The inner product $u(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} = \Sigma_i \, x_i y_i$ is linear with respect to each input. So, $u(\mathbf{x} + \mathbf{z}, \mathbf{y}) = u(\mathbf{x}, \mathbf{y}) + u(\mathbf{z}, \mathbf{y})$, and similarly for the second input. However, it is not linear with respect to element-wise addition over its entire input and output, as $u(\mathbf{x}_1 + \mathbf{x}_2, \mathbf{y}_1 + \mathbf{y}_2) \neq u(\mathbf{x}_1, \mathbf{y}_1) + u(\mathbf{x}_2, \mathbf{y}_2)$. Compare this to copying $\Delta$, which we can show is linear.

$$
\begin{aligned}
&\Delta : \mathbb{R}^a \to \mathbb{R}^{a \times a} \text{ and } x, y \in \mathbb{R}^a, \ \lambda \in \mathbb{R} \\
&\qquad \Delta(x)(x, x) \\
&\Delta(x + y) = (x + y, x + y) = (x, x) + (y + y) \\
&\qquad = \Delta(x) + \Delta(y) \\
&\Delta(\lambda \cdot x) = (\lambda \cdot x, \lambda \cdot x) = \lambda \cdot (x, x) = \lambda \\
&\qquad = \lambda \cdot \Delta(x)
\end{aligned}
$$

To simultaneously broadcast multilinear functions, we note how every multilinear operation equals an outer product followed by a linear function. The outer product is the ur-multilinear operation, taking a tuple input and returning a tensor, which takes the product of one element from each tuple segment. It is given by $\otimes : \mathbb{R}^a \times \mathbb{R}^b \to \mathbb{R}^{a \times b}$. All tuple-multilinear functions $f : \mathbb{R}^a \times \mathbb{R}^b \to \mathbb{R}^c$ have an associated tensor-linear form $f_\lambda : \mathbb{R}^{a \times b} \to \mathbb{R}^c$ such that $\otimes ; f_\lambda = f$. We diagram the outer product by simply having a tuple line ending, which will often occur before a host of linear operations are simultaneously applied.

### 2.4.2 Implementing Linearity and Common Operations

Key linear and multilinear operations can be implemented by the einops package, leading to elegant implementations of algorithms. Some key linear operations are **inner products**, which sum over an axis, **transposing**, which swaps axes, **views**, which rearranges axes, and **diagonalization**, which makes axes take the same index.

With neural circuit diagrams, we can clearly show these operations. We show inner products with cups, transposing by crossing wires, views by solid lines consuming and producing their respective shapes, and diagonalization by wires merging. As these operations are linear, they can be simultaneously applied.

The interaction of wires shows how incoming axes coordinate to produce outgoing axes. The einops package symbolically implements these operations by associate incoming and outgoing axes with symbols. The graphical notation we have chosen means that neural circuit diagrams correspond to einops symbolic expressions.

A good example that combines many of these operations is a section of multi-head attention shown in Figure 18. It employs an outer product, a transpose, a diagonalization, an inner product, and an element-wise operation. The input to this algorithm is a tuple of tensors. Axes with an overline are a width, representing the amount of rather than detail per thing.

Though a complex expression, we can break this figure up as in Figure 9 and implement the interaction of wires using einops, shown in Figure 19.

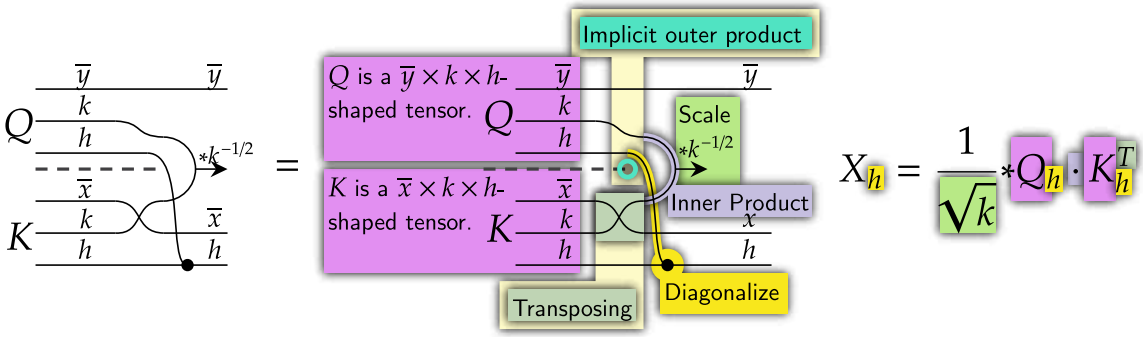
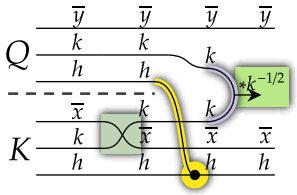



Figure 18: We can diagram a portion of multi-head attention, a sophisticated algorithm, with clarity using neural circuit diagrams.

Implementation using einsum

```
# Local memory contains,
# Q: y k h # K: x k h
# Transpose K,
Q, K = Q, einops.einsum(K, 'x k h -> k x h')
# Implicit outer product and diagonalize,
X = einops.einsum(Q, K, 'y k1 h, k2 x h \
                  -> y k1 k2 x h')
# Inner product,
X = einops.einsum(X, 'y k k x h -> y x h')
# Scale,
X = X / math.sqrt(k)
```



Implementation using einsum
(with simultaneous broadcasting of linear functions)

```
# Local memory contains,
# Q: y k h # K: x k h
X = einops.einsum(Q, K, 'y k h, x k h -> y x h')
X = X / math.sqrt(k)
```

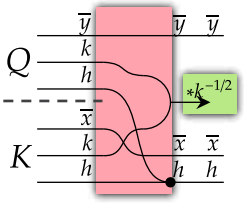


Figure 19: This section of multi-head attention can be implemented using the einsum operation. Here, we can see the close relationship between diagrams and implementation, and how neural circuit diagrams reflect the memory states and operations of algorithms. (*See cell 15, Jupyter notebook.*)

### 2.4.3 Linear Algebra

All linear functions $f : \mathbb{R}^a \to \mathbb{R}^b$ have an associated $\mathbb{R}^{a \times b}$ tensor that uniquely identifies them. This hints at the ability to transpose this associated tensor to get a new linear function, $f^T : \mathbb{R}^b \to \mathbb{R}^a$. To extract these associated transposes, we use the unit. The **unit** for a shape $a$, given by $\eta : \mathbb{R}^1 \to \mathbb{R}^{a \times a}$, is a linear map which returns $r$ times the $\mathbb{R}^{a \times a}$ identity matrix, for $r \in \mathbb{R}$.

Note that the associated transpose, which sends a linear function $f : \mathbb{R}^n \to \mathbb{R}^m$ to $f^T : \mathbb{R}^m \to \mathbb{R}^n$ by transposing the associated $\mathbb{R}^{n \times m}$ tensor, is different to a transpose operation which sends $\mathbb{R}^{n \times m}$ to $\mathbb{R}^{m \times n}$. Associated transposes are used for mathematical rearrangement and are not usually directly implemented in code, though we provide code examples in cell 17 of the Jupyter notebook.

The unit and the inner product can be arranged to give the identity map $\mathbb{R}^a \to \mathbb{R}^a$, as in Figure 20. This identity map can be freely introduced, split into a unit and the identity matrix, and then used to rearrange operations. For example, this allows us to convert the linear map $F : \mathbb{R}^a \to \mathbb{R}^{b \times c}$ into $F^T : \mathbb{R}^{b \times a} \to \mathbb{R}^c$. These associated tensors and transposes can be used to understand better convolution (Section 3.3) and backpropagation (Section 3.6).

Additionally, as the simultaneous broadcast of linear morphisms is their outer product, an outer product can be applied before or after a simultaneous broadcast to equivalent effect.

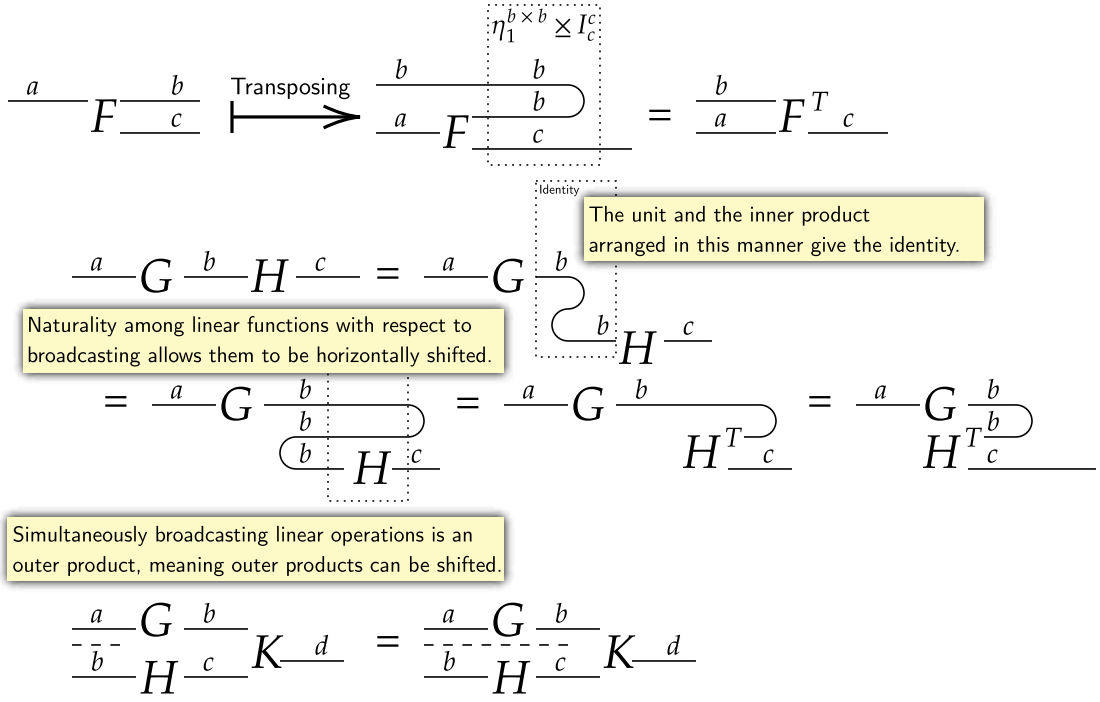



Figure 20: Linear operations have a flexible algebra. Simultaneous operations may increase efficiency (Xu et al., 2023). As the height of diagrams is related to the amount of data stored in independent segments, it gives a rough idea of memory usage. This is further explored in Section 3.6. (*See cell 17, Jupyter notebook.*)

These rearrangements can transpose specific axes. A linear operation $\mathbb{R}^{a \times b} \to \mathbb{R}^c$ has an associated $\mathbb{R}^{a \times b \times c}$ tensor. This tensor can be associated with various linear operations, such as $\mathbb{R}^{b \times a} \to \mathbb{R}^c$. These different forms are often of interest to us, as they can efficiently implement the reverse of operations (see Figure 26, 31). To extract these rearrangements, we can selectively apply units and the inner product to reorient the direction of wires for linear operations.

## 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 present this in Figure 21. 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, 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

```python
import torch.nn as nn
# Basic Image Recogniser
# This is a close copy of an introductory PyTorch tutorial:
# https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html
class BasicImageRecogniser(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
```

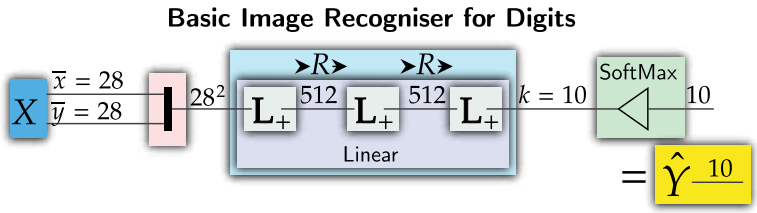



Figure 21: PyTorch code and a neural circuit diagram for a basic MNIST (digit recognition) neural network taken from an introductory PyTorch tutorial. Note the close correspondence between neural circuit diagrams and PyTorch code. (*See cell 19, Jupyter notebook.*)

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 ($\triangleleft$), 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.

### 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 22 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 23 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 38 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.

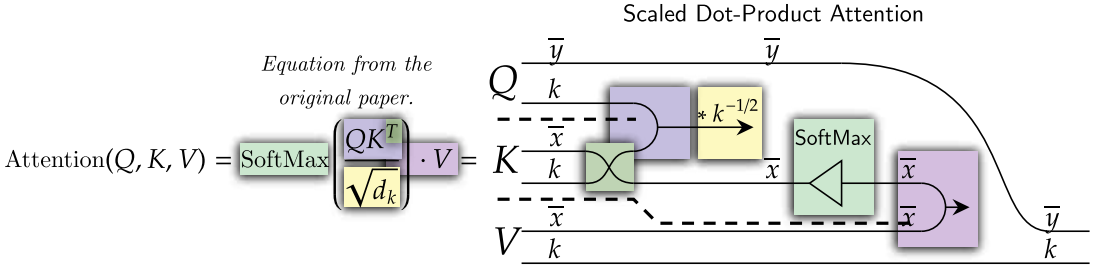



Figure 22: The original equation for attention against our diagram. The descriptions are unnecessary but clarify what is happening. Corresponds to Equation 1 and Figure 1.1. (*See cell 21, Jupyter notebook.*)

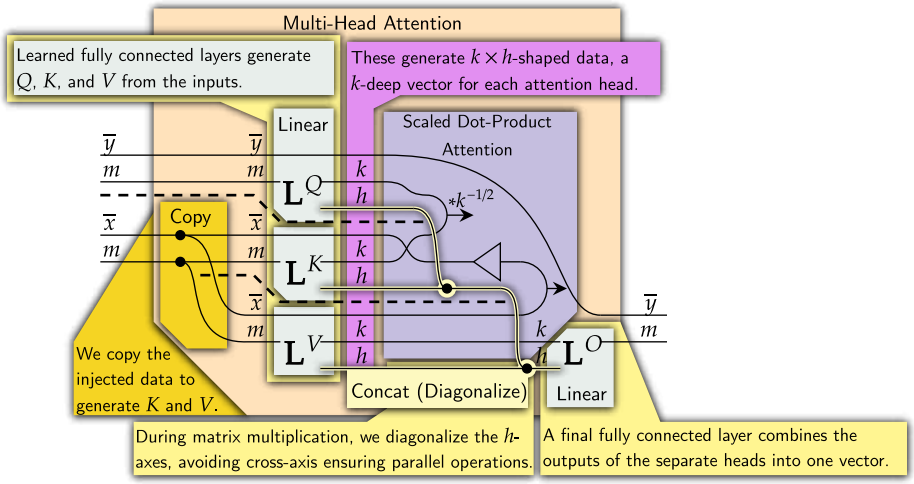



Figure 23: Neural circuit diagram for **multi-head attention**. Implementing matrix multiplication is clear with the cross-platform the einops package (Rogozhnikov, 2021). Corresponds to Equation 2 and 3 and Figure 1.2. (*See cell 23, Jupyter notebook.*)

### 3.3 Convolution

Convolutions are critical to understanding computer vision architectures. Different architectures extend and use convolution in various 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.)*

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

The maximum $i_{\overline{y}}$ value is such that it does not exceed the maximum index for $v[i_{\overline{y}} + j_k]$. Starting indexing at 0, we get $\overline{x} - 1 = i_{\max} + j_{\max} = \overline{y} + k - 2$, so the length of the output is therefore $\overline{y} = \overline{x} - k + 1$. Note how convolution is a multilinear operation; it is linear concerning each vector input $v$ and $w$. Therefore, it has a tensor-linear form with an associated tensor, the convolution tensor, that uniquely identifies it.

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

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

We diagram convolution with the below diagram, Figure 24. 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 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

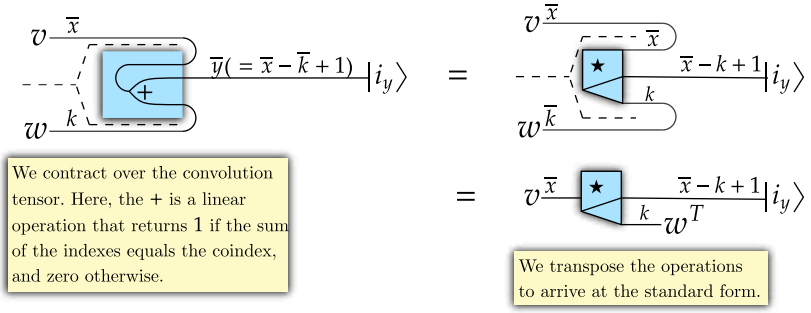



Figure 24: Convolution is a multilinear operation, with an associated tensor. This tensor is transposed into a standard form.

their length by one bolded symbol akin to a vector. In 2 dimensions 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 25 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, $1 \times 1$ convolution tensors give a linear operation $\mathbb{R}^{\overline{\mathbf{x}}} \to \mathbb{R}^{\overline{\mathbf{x}} \times 1}$, which we recognize to be the identity. Therefore, $1 \times 1$ kernels are the same as broadcasting over the input.

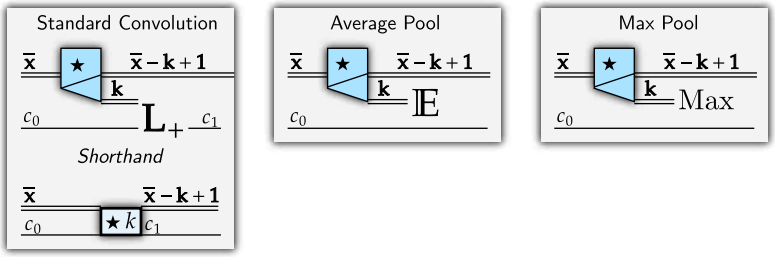



Figure 25: Convolution and related operations, clearly shown using neural circuit diagrams.

*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 26. 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.

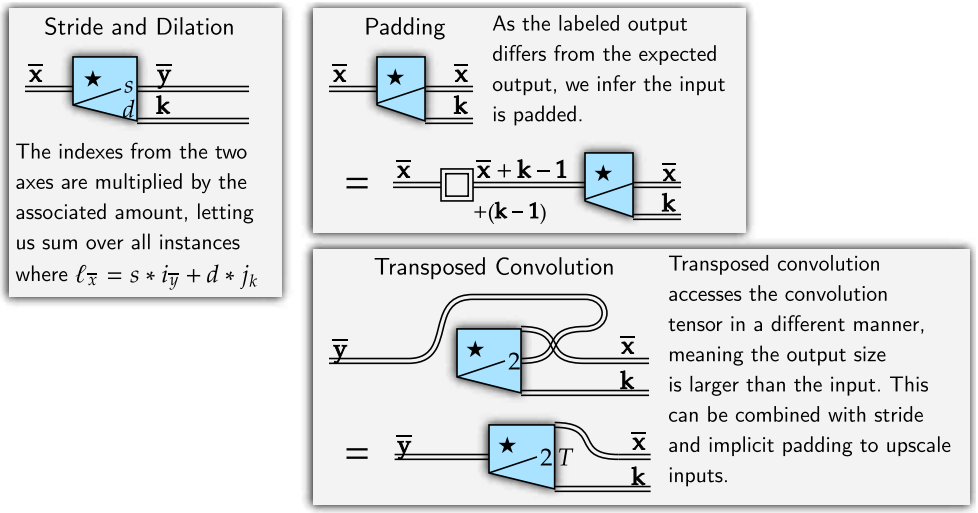



Figure 26: Stride, dilation, padding, and transposed convolution shown with neural circuit diagrams.

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.

### 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). Architectures can relate information at different scales (Luo et al., 2017), making architecture design task-dependant. Sophisticated architectures such as vision transformers combine the complexity of convolution and transformer architectures (Khan et al., 2022; Dehghani et al., 2023). These cases show why clear architecture design is promising for enhancing computer vision research. 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.

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 27, which shows many innovations of ResNets not included in common implementations, as well as the UNet architecture (Ronneberger et al., 2015) in Figure 28, which lets us show how saving and loading variables may be displayed. 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. The code associated with this work implements these algorithms guided by the blocks from the diagrams.

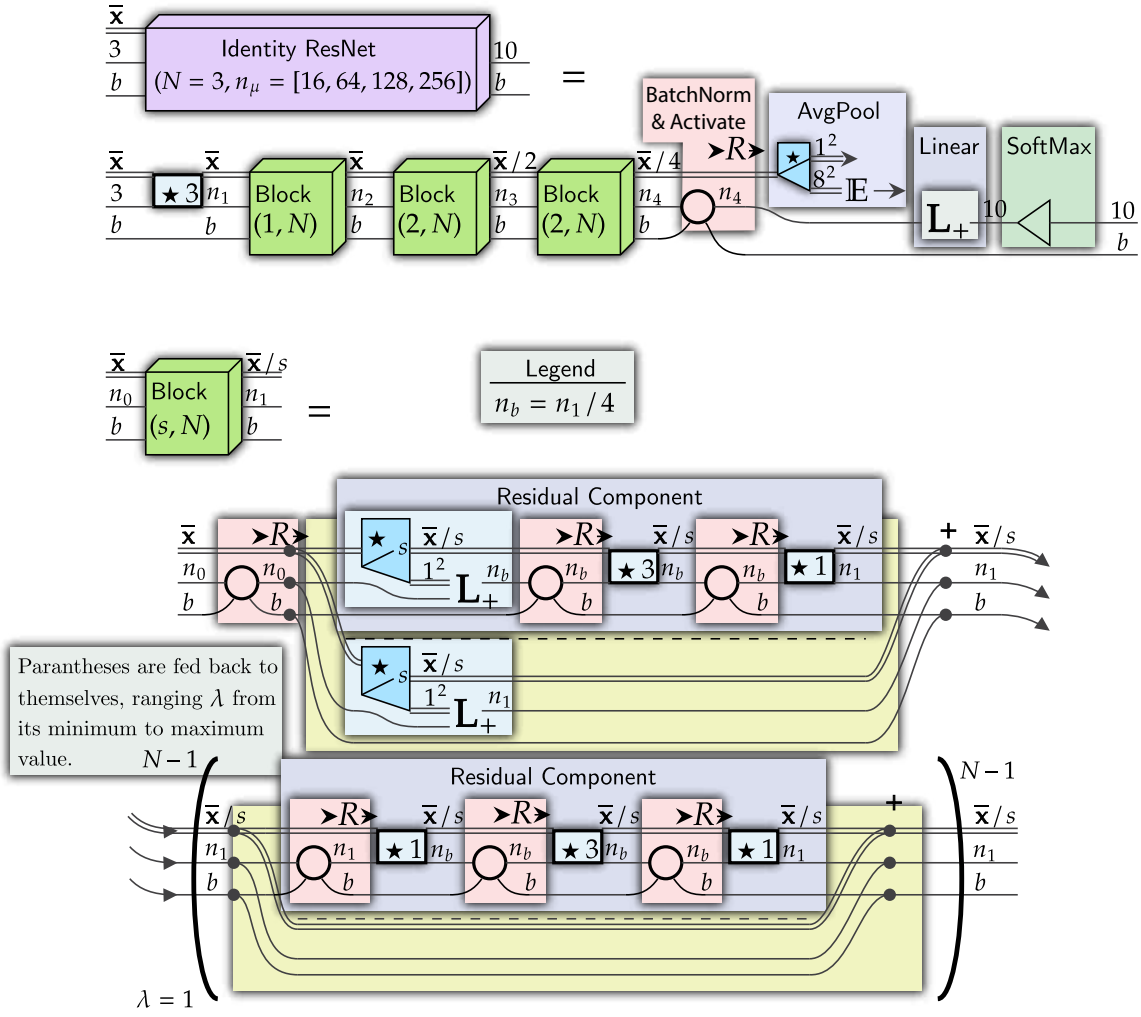



Figure 27: Residual networks with identity mappings and full pre-activation (IdResNet) (He et al., 2016) offered improvements over the original ResNet architecture. These improvements, however, are often missing from implementations. By making the design of the improved model clear, neural circuit diagrams can motivate common packages to be updated. (*See cell 25, 27, 28, Jupyter notebook.*)

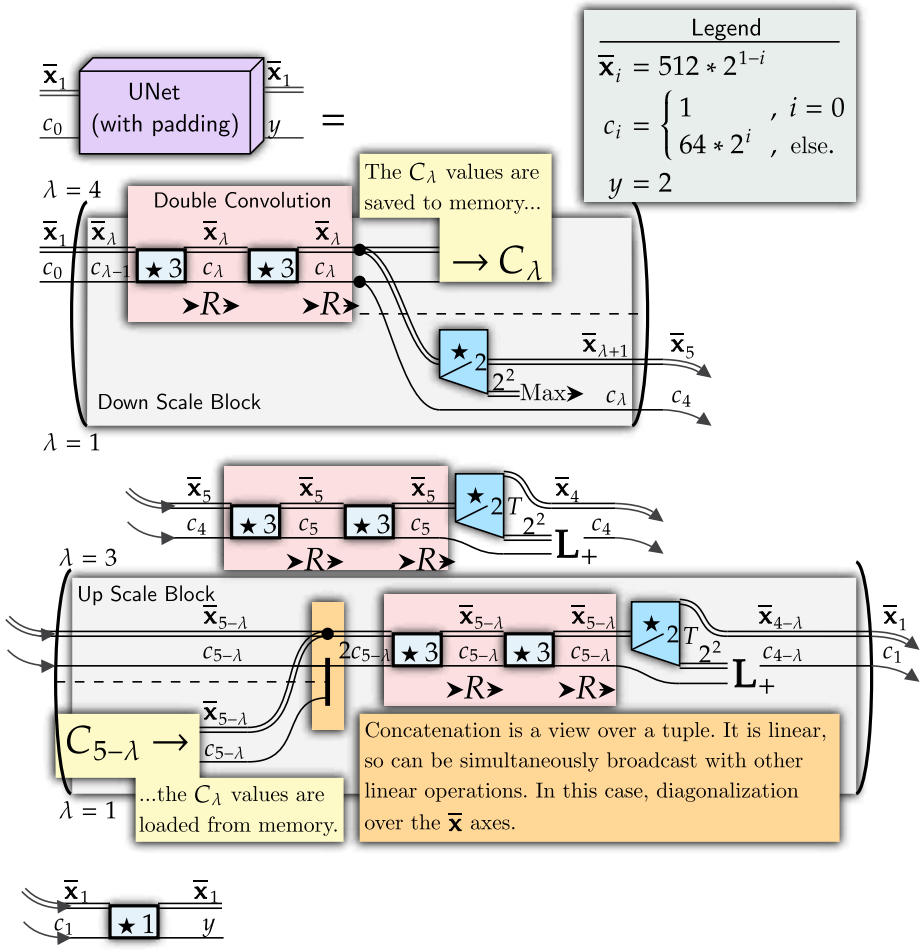



Figure 28: The UNet architecture (Ronneberger et al., 2015) forms the basis of probabilistic diffusion models, state-of-the-art image generation tools (Rombach et al., 2022). UNets rearrange data in intricate ways, which we can show with neural circuit diagrams. Note that in this diagram we have modified 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 30, 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 23 with tandem $\overline{\mathbf{y}}$ and $\overline{\mathbf{x}}$ axes and setting $h = 1$. As $1 \times 1$ convolutions are simply the identity map, $\text{Conv}(v, [1]) = v$, broadcasting a linear map $\mathbb{R}^c \to \mathbb{R}^k$ for each of $\overline{\mathbf{y}}$ pixels is a $1 \times 1$-convolution. This leaves us with Figure 29 for a visual attention mechanism.

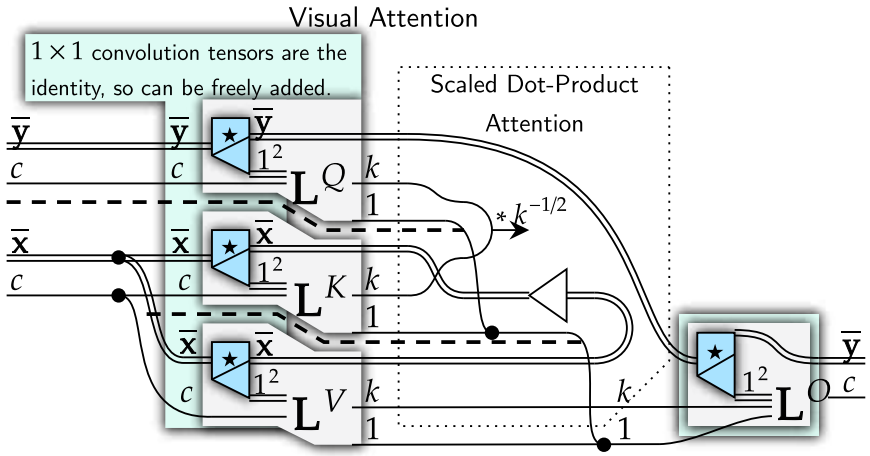



Figure 29: Using neural circuit diagrams, visual attention (Dosovitskiy et al., 2021) is shown to be a simple modification of multi-head attention (See Figure 23, Figure 18, cell 32, Jupyter notebook.)

This highly suggestive diagram calls 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 similar to Figure 18. 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 while being precise in how algorithms should be implemented.

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

We leave the most mathematically dense part of this work for last. Neural circuit diagrams intend to be used for the communication, implementation, tinkering, and analysis of architectures. These aims appeal to distinct audiences, and each should conceptualize neural circuit diagrams differently. The theoretical study of deep learning models requires understanding how individual components are composed into models and how properties scale during composition. Neural circuit diagrams are highly composed systems (see Figure 9) and thus provide a framework for studying composition. They have an underlying category, which is not the focus of this work.

Differentiation is an example of a property that is agreeable under composition. Differentiation is key to understanding information flows through architectures (He et al., 2016). The chain rule relates the derivative of composed functions to the composition of their derivatives and, therefore, provides a case study of how studying composition allows models to be understood. This analysis, however, is hampered by the fact that symbolically expressing the chain rule has quadratic length complexity relative to the number of composed

functions.

$$h'(x) = h'(x)$$
$$(g \circ h)'(x) = (g' \circ h)(x) \cdot h'(x)$$
$$(f \circ g \circ h)'(x) = (f' \circ g \circ h)(x) \cdot (g' \circ h)(x) \cdot h'(x)$$

This issue of symbolic methods proliferating symbols to keep track of relationships between objects was noted in the introduction. To understand how differentiation is composed and encourage more innovations like that of identity ResNets, which used differentiation to understand data flows (He et al., 2016), we need a graphical differentiation method.

Some graphical methods have been developed and applied to understanding differentiation in the context of deep learning, drawing on monoidal string diagrams from category theory (Shiebler et al., 2021; Cockett et al., 2019). As linearity cannot be completely ensured, these graphical methods are Cartesian, not expressing the details of axes. Other graphical approaches to neural networks could not incorporate differentiation, showing the significance of neural circuit diagrams being able to incorporate differentiation (Xu & Maruyama, 2022).

Differentiation, however, has key linear properties. Transposing differentiation is very important. These prior graphical methods require redefining differentiation for each transpose, making the relationships between these forms unclear. By detailing tensors and Cartesian products, our graphical presentation can show these linear relationships clearly. While drawing on their many theoretical contributions (Shiebler et al., 2021; Cockett et al., 2019), this work provides a significant advantage over these previous works.

In addition to theoretical understanding, clearly expressing differentiation is key to efficient implementations. Mathematically equivalent algorithms may have different time or memory complexities. The rules of linear algebra we have developed (see Figure 20) allow mathematically equivalent algorithms to be rearranged into more time or memory-efficient forms. To show the potential of neural circuit diagrams, we focus on backpropagation and analyze its time and memory complexity with neural circuit diagrams.

### 3.6.1 Modeling Differentiation

To model differentiation, consider a once differentiable function $F : \mathbb{R}^a \to \mathbb{R}^b$. It has a Jacobian which assigns to every point in $\mathbb{R}^a$ a $\mathbb{R}^{a \times b}$ tensor that describes its derivative, $\mathsf{J}F : \mathbb{R}^a \to \mathbb{R}^{b \times a}$. Functions answer questions, and $\mathsf{J}F$ answers how much a function responds to an infinitesimal change. The questions we ask $\mathsf{J}F$ are *where* is the change happening ($\mathbb{R}^a$ input), *how* much is it changing by ($\mathbb{R}^b$ output axis), and *which* direction are we moving in ($\mathbb{R}^a$ output axis). Inner products over the output axes "ask" these questions. The chain rule can be defined with respect to the Jacobian and is diagrammed in Figure 30.

$$\frac{\partial}{\partial x^a}(GF)^c\Big|_{\mathbf{x}} = \sum_b \left( \frac{\partial}{\partial x^b} G^c\Big|_{F(\mathbf{x})} \cdot \frac{\partial}{\partial x^a} F^b\Big|_{\mathbf{x}} \right) \qquad \mathsf{J}[F;G] = \quad \mathsf{J}F \quad \mathsf{J}G$$

Figure 30: The chain rule expressed symbolically with index notation, and with neural circuit diagrams.

This expression is convoluted, and will struggle to scale. Instead, we transpose $\mathsf{J}F$ into the forward derivative as per Cockett et al. (2019)'s definition 4. This form is more agreeable for the chain rule, and is the first transpose we employ.

$$DF = \quad \mathsf{J}F$$
$$D[F;G] = \quad F \quad \partial F \quad \partial G$$

Figure 31: Definition of the forward derivative, and how functions compose under it.

This naturally scales with depth. Furthermore, we can define a $(\_, D\_)$ functor, a composition preserving map, from once differentiable functions $F : \mathbb{R}^a \to \mathbb{R}^b$ to $(F, DF) : \mathbb{R}^a \times \mathbb{R}^a \to \mathbb{R}^b \times \mathbb{R}^b$. Per the chain rule, $(\_, D\_)[F; G] = (\_, D\_)F; (\_, D\_)G$. This is shown in Figure 32.

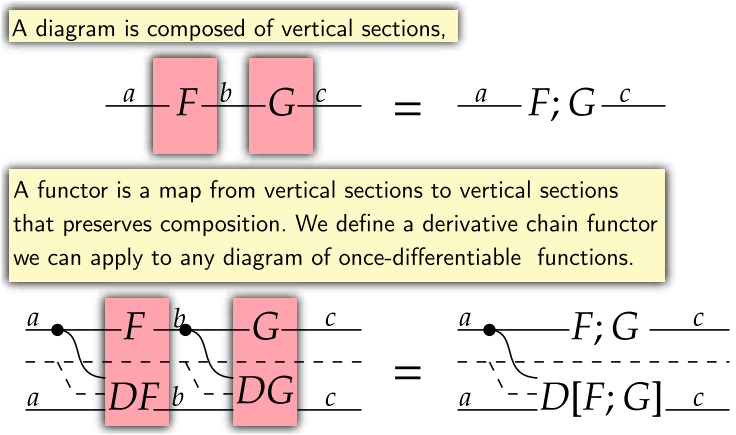



Figure 32: We have a composition preserving map, the $(\_, D\_)$ functor, that maps vertical sections to vertical sections, implementing the chain rule.

This composes elegantly. However, when optimizing an algorithm, we are not interested in how much a known infinitesimal change will alter an output. Rather, given some target change in output, we are interested in which direction will best achieve it. We can do this by calculating the change in the output for each element of $a$ in parallel, effectively running the algorithm multiple times. This is done by applying the unit and broadcasting. Furthermore, we sum the infinitesimal change over some target $\mathbb{R}^c$ value. The inner product does this. For an algorithm $F; \mathcal{L}$, where $\mathcal{L}$ is a loss function to $\mathbb{R}^1$, we can do optimization according to Figure 33.

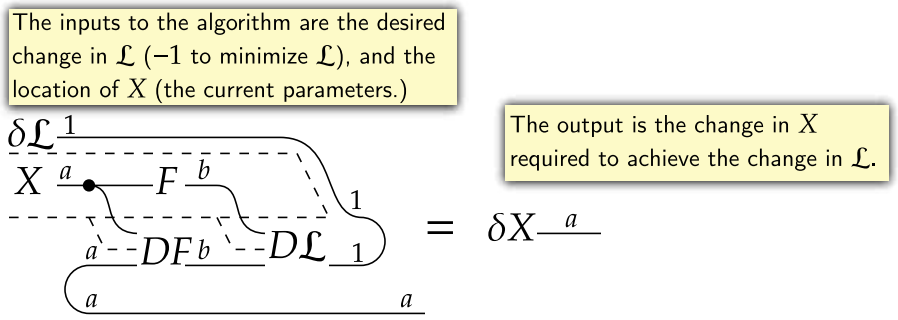



Figure 33: We turn a small chain rule expression into an optimization function by applying the inner product over the target direction and derivative output. An inner product over an axis of length 1 is just multiplication. Using the unit, we run this algorithm for every input degree of freedom, broadcasting over the $a$ axis.

However, the forward derivative has large time complexity. A linear function gives matrix multiplication. Therefore, a linear map $f : \mathbb{R}^a \to \mathbb{R}^b$ applied onto $\mathbb{R}^a$ will require $a \times b$ operations. In general, broadcasting multiplies time and memory complexity. The memory usage of an algorithm is related to the number of elements it stores at any step in the algorithm. We use these tricks to analyze the order of the time and space complexity for the above process.

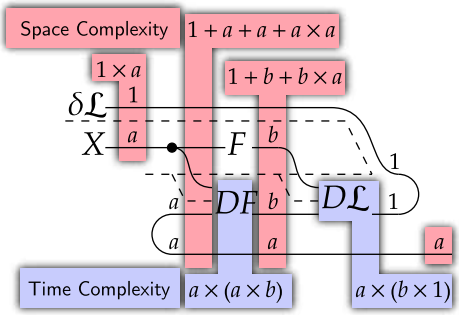



Figure 34: An analysis of the space and time complexity of the naive optimization algorithm.

We observe that this has a high time complexity, quadratic with respect to the size of $X$. In practice, we avoid the forward derivative, also called the Jacobian-vector product or JVP, in favor of the reverse derivative, or VJP, which more directly implements the above process. We define it in relation to the Jacobian and forward derivative in Figure 35. In Figure 37, we use our rules of linear algebra to re-express the optimization algorithm in terms of the forward derivative and show the far lower memory and time complexity required.

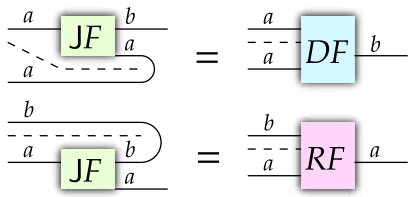



Figure 35: The definition of the forward and reverse derivative with respect to the Jacobian. This aligns with the Jacobian-vector product and the vector-Jacobian product, respectively.

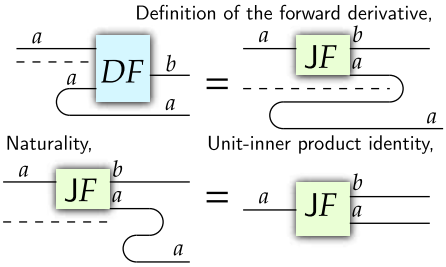



Figure 36: A full expression of how the unit and the forward derivative give the Jacobian. This demonstrates how linear algebra principles can illustrate the relationships between different forms of the derivative.

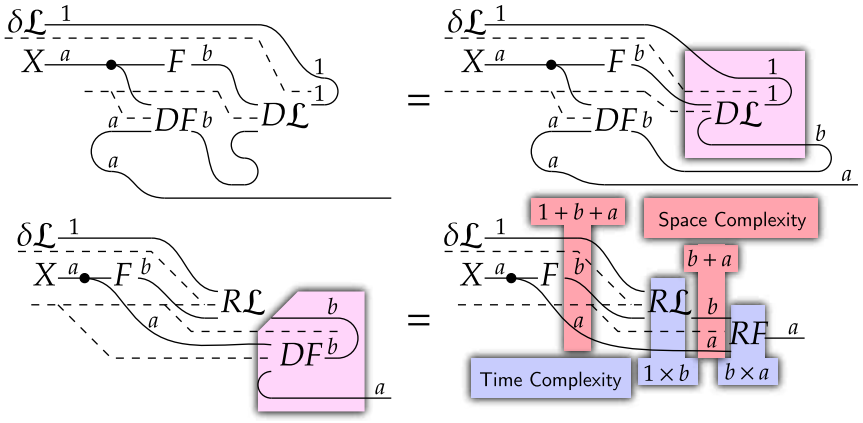



Figure 37: Using the previously developed linear algebra rules (see Figure 20, we rearrange our optimization algorithm to use the reverse instead of the forward derivative. The diagrams then reveal the computational advantages of the new algorithm, called backpropagation.

## 4 Conclusion

In this paper, we introduced neural circuit diagrams to solve the lingering problem of unclear communication in the deep learning community. Our introduction showed why this is a concern and argued why our approach is optimal, solving the challenge of reconciling the detail of tensor axes and the flexibility of tuples. We covered a host of architectures to breed familiarity, encourage adoption, and evidence the utility of neural circuit diagrams for understanding and implementing models.

We see neural circuit diagrams as appealing to diverse users, from students first learning neural networks to specialized theoretical researchers investigating their mathematical foundations. This leads to immense future potential. Future work can see neural circuit diagrams explained in a more concise and accessible manner that we, in our familiarity, have difficulty doing. A further range of models can be diagrammed and standards developed. Finally, their mathematical foundation can be more fully expressed. The underlying category theory structure can be fully investigated, allowing models to incorporate probabilistic functions (Perrone, 2022; Fritz et al., 2023), additional data types, or even quantum circuits.

### Acknowledgements

Mathcha was used to write equations and draw diagrams. The Harvard NLP annotated transformer was invaluable for drawing Figure 38.

### Funding

This work was supported by JST (JPMJMS2033).

## 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].

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

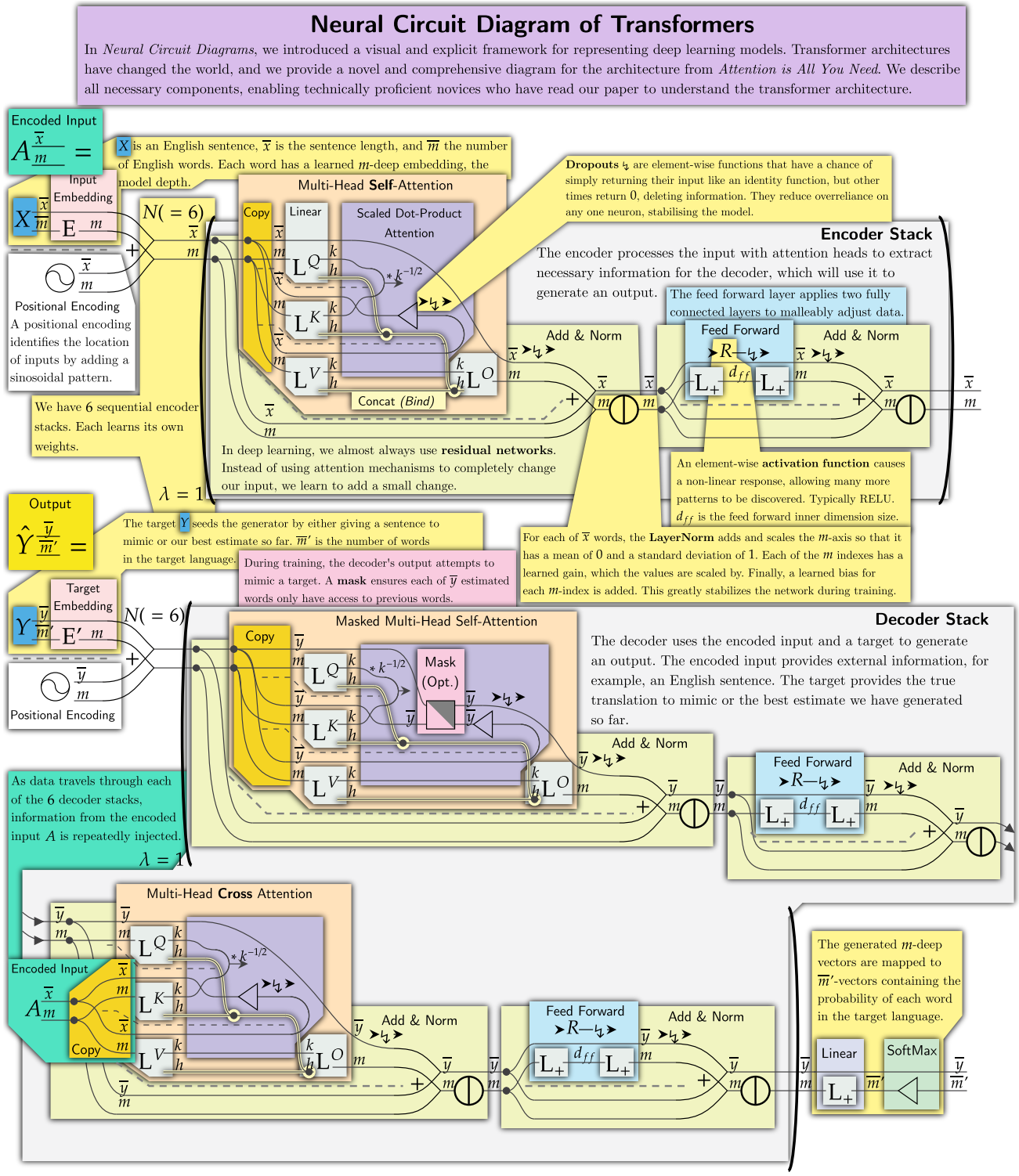



Figure 38: The fully diagrammed architecture from *Attention is All You Need* (Vaswani et al., 2017).

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/.