The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering, Computing and Cybernetics (CECC)

Robust Diagrams for Deep Learning Architectures: Applications and Theory

— 24 pt Honours project (S1/S2 2023)

A thesis submitted for the degree
Bachelor of Science (Honours)

By:
Vincent Thornton Abbott

Supervisor:
Dr. Yoshihiro Maruyama

November 2023

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

November, Vincent Thornton Abbott

Abstract

Deep learning models are at the forefront of human technology. However, we lack a systematic framework for understanding these systems as mathematically explicit composed structures. This hinders our ability to communicate, implement, and analyze models. This work resolves this shortfall. In this thesis, I present neural circuit diagrams, a comprehensive graphical language for deep learning architectures with a robust mathematical basis in category theory. This work is split into three chapters that identify gaps in the research and contribute solutions.

Chapter 1: ***The Problem and the Solution*** I assess the current state of deep learning research. I identify the importance of architectural innovations to the success of contemporary models. I contribute a critique that I believe is vital to address: we lack a robust graphical language for understanding architectures. I provide a case study of *Attention is All You Need*, showing how its presentation is unclear. More generally, we lack a compositional mathematical framework that encompasses contemporary models. Then, I argue why category theory is a promising approach to resolve these issues.

Chapter 2: ***Applications of Neural Circuit Diagrams*** I introduce neural circuit diagrams with a focus on practical applications. This chapter avoids category theory, proving that diagrams can achieve general adoption. I provide comprehensive diagrams for a host of architectures – from transformers to computer vision – contributing explanations for systems that are otherwise difficult to communicate. Finally, I prove the analytical utility of neural circuit diagrams by using them to analyze linear rearrangements and computational complexities of algorithms.

Chapter 3: ***Theory of Functor String Diagrams*** I focus on the robust theory underlying neural circuit diagrams. Using category theory, I build on the nascent field of functor string diagrams, contributing first-principles and novel tools such as family expressions. I reconcile neural circuit diagrams with functor string diagrams, showing how neural circuit diagrams have a robust mathematical basis. By providing an explicit model of deep learning architectures, this section contributes the foundation for future work that systematically analyzes the mathematical properties of deep learning architectures.

Table of Contents

Chapter 1

The Problem and the Solution

1.1 The Importance of Communicating Architectures

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 and 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 frequently 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 and 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 explaining models through 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 and 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 replicating 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 by allowing the *idea* of models to be detached from any specific implementation.

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

To highlight the problem of insufficient communication of deep learning architectures, I present a case study of *Attention is All You Need*, the paper that introduced transformer models (Vaswani et al., 2017). Introduced in 2017, transformer models have revolutionized machine learning, finding applications in natural language processing, image processing, and generative tasks (Phuong and 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 I refer to as the original transformer paper, explains these algorithms using diagrams (see Figure 1.1) and equations (see Equation 1.1,1.2,1.3) that hinder understandability (Chiang et al., 2023; Phuong and Hutter, 2022).

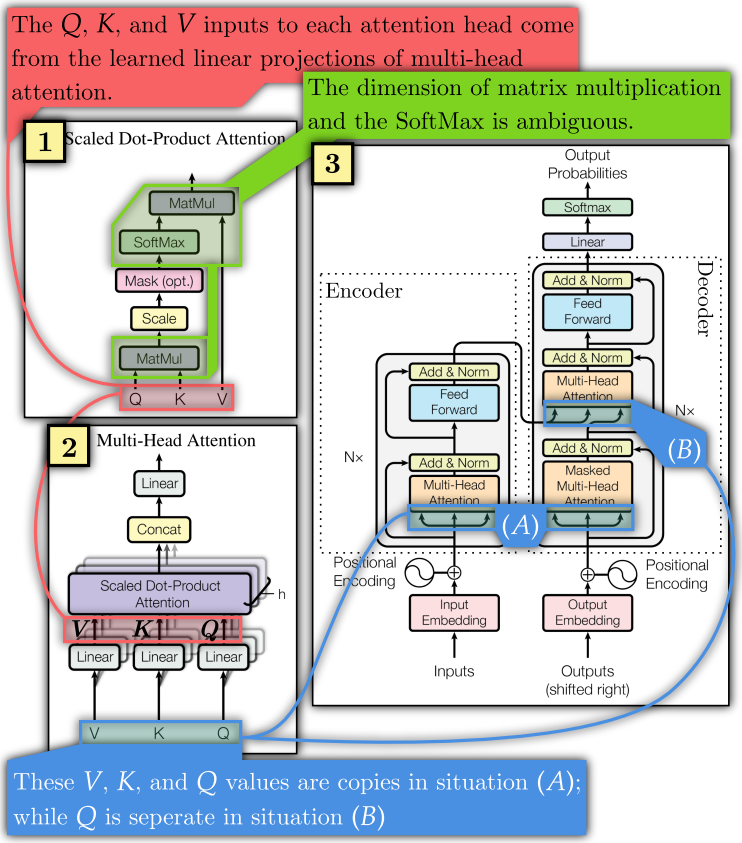


Figure 1.1: My annotations of the diagrams of the original transformer model. Critical information is missing regarding the origin of Q, K, and V values (**red** and **blue**), and the axes over which operations act (**green**).

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (d_k \text{ is the key depth}) \tag{1.1}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{...,head}_h)W^O \tag{1.2}$$

$$\text{where head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{1.3}$$

The original transformer paper obscures dimension sizes and their interactions. The dimensions over which SoftMax[1] and matrix multiplication operates is ambiguous (Figure 1.1.1, **green**; Equation 1.1, 1.2, 1.3).

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

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 1.1.1 and Equation 1.1, they represent separate values fed to each attention head. In Figure 1.1.2 and Equation 1.2 and 1.3, they are all copies of each other at location ***(A)*** of the overall model in Figure 1.1.3, 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 and Bajzek, 2008; Ross et al., 1977). In Figure 1.1, 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.

Therefore, we 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, I present alternative diagrams in Figures 2.18, 2.19, and 2.20.

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 and Hutter, 2022; Chiang et al., 2023; Xu et al., 2023; Xu and 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 and 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. Besides, a rigorously developed framework-independent graphical language for deep learning architectures would help to improve 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 and Bergholm, 2017; Xu et al., 2023; Xu and Maruyama, 2022). 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 (Baez and Stay, 2010; Awodey, 2010).

All these graphical methods have been found to represent an underlying category, a mathematical space with well-defined composition rules (Meseguer and Montanari, 1990; Baez and Stay, 2010). A category theory approach allows a common structure, monoidal products, to define an intuitive graphical language (Selinger, 2009; Fong and 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 and Zanasi, 2022; Cruttwell et al., 2021). However, Cartesian monoidal products do not automatically keep track of dimensionality and cannot easily represent broadcasting or linear operations. These works often rely on the most rudimentary model of deep-learning networks as sequential linear layers and activation functions, despite residual networks having become the norm (He et al., 2015, 2016). The graphical language generated by a pure Cartesian approach fails to show the details of architectures, limiting its ability to consider models as they appear in practice.

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 typically 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 anyway. The challenge of reconciling Cartesian and tensor approaches, therefore, remains.

1.4 The Promise of Category Theory

There is, however, a nascent field of category theory that can graphically reconcile the details of axes and products. In "*Category Theory Using String Diagrams*", Marsden (2014) used functors, structure-preserving maps, instead of objects as wires in diagrams. This generates diagrams that clearly show compositional structures at different levels of analysis. His work has garnered interest from other authors (Piedeleu and Zanasi, 2023; Braithwaite and Román, 2023; Román, 2021). However, only Nakahira (2023) has worked to extend Marsden's string diagram approach to new circumstances in "*Diagrammatic Category Theory*", a work from July this year.

I will call diagrams which use functor wires *functor string diagrams*, in contrast to traditional monoidal string diagrams (Selinger, 2009), which are specialized to show a category with a privileged product. I see them as an immensely promising framework to provide robust diagrams that show structure at different levels of analysis and can show both the details of axes and products. However, they have not been extensively developed for practical applications. Developing a general graphical language for deep learning architectures using functor string diagrams will require extensive work.

The nature of a problem determines which field of mathematics is appropriate to understand it. Deep learning models are highly composed systems, with commands forming components, which form layers, which form models at immense scales. Previously (see Section 1.2), we saw how standard mathematical approaches, such as linear algebra notation fall short of understanding these systems. Category theory is the mathematical study of composition and how composition is preserved between perspectives. Therefore, it is the appropriate mathematics to understand deep learning models.

In addition to offering comprehensive diagrams, a robust category theory-based graphical language for deep learning architectures would have additional benefits. The category of diagrams would correspond to a category of mathematical expressions. Deep learning models are mathematical expressions and should be understood as such. However, they are also implemented code and human-comprehensible engineered systems. Moving between these perspectives, a category theory approach allows the underlying structure to be preserved.

Category theory can place diagrams on a rigorous foundation that relates them to existing research combining graphical methods (Selinger, 2009), machine learning (Shiebler et al., 2021; Fong et al., 2019; Cruttwell et al., 2021), probability theory (Perrone, 2022; Fritz et al., 2023), and various other fields. Meanwhile, machine learning models often have vague theory (He et al., 2016). Robust diagrams are the key to connecting a field

with too few abstract foundations, deep learning, to a field with too many, category theory, satisfying a gap in the literature.

Previous works that have used category theory to understand deep learning only use rudimentary models. In Fong et al. (2019), only the most basic sequential linear layer-activation layer neural networks are covered by the category **NNet** being studied. In a survey of string diagram methods, Piedeleu and Zanasi (2023) identified that the models presented by Fong et al. (2019) are among the more diagrammatically rich. In their survey, Piedeleu and Zanasi (2023) notes that the current rudimentary models of deep learning systems are only a "starting point" for future work.

The issue of only looking at rudimentary, linear layer-activation layer models is pervasive in deep learning research (Zhang et al., 2017; Saxe et al., 2019; Li et al., 2022). There are uncountably many ways of relating inputs to outputs. Every theory or hypothesis about deep learning algorithms has to assume that we are working with some subset of all possible functions. However, specifying this subset means theoretical insights can only apply to that subset. This precludes us from using such theories to compare disparate architectures and make design choices.

The problem of only considering rudimentary models – I feel – is a consequence of us not having the tools to robustly represent more complex models, never mind the tools to confidently analyze them. Category theory-based diagrams can serve as models of intricate systems. Structure-preserving maps allow analyses to scale over entire models. Therefore, developing comprehensive diagrams that correspond to mathematical expressions can be the first step in a rigorous theory of deep learning architectures with clear practical applications.

1.5 Contributions

Aware of the need for comprehensive diagrams for deep learning models, and the immense benefits that would accompany a category theory-based approach, I present *neural circuit diagrams* - a robust, category theory-based graphical language developed to represent deep learning models. The main content of this thesis is split into two chapters – a practical chapter, introducing neural circuit diagrams in an accessible manner, and which contributes comprehensive diagrams for various architectures; and a theoretical chapter, contributing tools for the study of functor string diagrams and using them to underpin broadcasting and neural circuit diagrams.

Chapter 2: *Applications of Neural Circuit Diagrams*

In chapter 2, I present neural circuit diagrams for practical applications. To prove that neural circuit diagrams can achieve general adoption by artificial intelligence researchers without specialized knowledge, I provide a guide for reading neural circuit diagrams without category theory.

Then, I diagram various architectures that are otherwise difficult to explain. This includes the transformer model (see Section 2.3.2), convolutional neural networks (see Section 2.3.3), the residual network with identity mappings, the UNet architecture (see Section 2.3.4), and then the vision transformer (see Section 2.3.5). Even for many machine learning specialists reading this work, I believe these diagrams will be the first time that many of these architectures have been understandably explained to them.

The transformer was chosen as the original presentation is cumbersome, presenting a barrier to entry for researchers wishing to learn about state-of-the-art models. Convolution is difficult to explain, especially when transposed[2] or when dilation is used. The residual network with identity mappings is an improvement on the original residual networks (He et al., 2016), however, even the PyTorch implementation of residual networks failed to realize the subtle innovations that had occurred. Contrasting to existing presentations – therefore – neural circuit diagrams are a powerful and comprehensive tool that overcomes present limitations in deep learning research.

My investigation of the vision transformer is particularly interesting. There, I show how neural circuit diagrams naturally link to extending architectures to new modalities, motivating innovation. Furthermore, directly reading computational complexity from diagrams is a powerful tool for designing and improving algorithms. These analytical tools open up an exciting avenue of future research, where the development and refinement of models are accelerated by using a robust graphical framework.

This chapter addresses the open problem of improving the communication of architectures in a novel way that improves on previous works. A graphical approach is a significant advantage over symbolic approaches such as "*Named Tensor Notation*" by Chiang et al. (2023) or "*Formal Algorithms for Transformers*" by Phuong and Hutter (2022). Unlike "*Graph Tensor Networks*" by Xu et al. (2023), understanding neural circuit diagrams does not require specialized mathematical knowledge. By covering a broad range of architectures, my presentation has a wider scope than these works. Additionally, by providing an analysis of backpropagation and computational complexity, this chapter presents a technical contribution that goes beyond these previous approaches.

Chapter 3: *Theory of Functor String Diagrams*

As typical monoidal graphical approaches have difficulty showing both independent data and the details of axes, I use this chapter to extend the functor string diagrams developed by Marsden (2014) and Nakahira (2023). Each section of this chapter contributes to

[2] The PyTorch ConvTranspose2D describes it as;

> This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation as it does not compute a true inverse of convolution). For more information, see the visualizations here and the Deconvolutional Networks paper.

making functor string diagrams more generally applicable and accessible. *From Theory to Application: A Robust Basis for Neural Circuit Diagrams*, the final section, reconciles neural circuit diagrams with functor string diagrams. This relates neural circuit diagrams with a mathematically robust graphical framework. This allows us to generate category theory-based insights into algorithms.

3.1 Building Blocks

I introduce functor string diagrams from a first-principles perspective. I ensure that all diagrams can be decomposed into vertical sections, and that new features are equivalent expressions of prior constructs. These form the two key principles of diagrams. This ensures all diagrams can be decomposed and understood. In contrast, Marsden (2014) and Nakahira (2023) emphasize colored regions that I feel distract from the focus of diagrams, which should be clearly presenting composition.

3.2 Reasoning with Diagrams

To reason about diagrams, I contribute diagrammatic family expressions. These are powerful tools to conduct proofs using diagrams. Graphical proofs are addressed by Nakahira (2023). However, my family expressions allow new features to be graphically defined and are more explicit than the dotted boxes Nakahira (2023) employs.

Having developed these tools, I provide a graphically intuitive proof of the Yoneda lemma, a central result from category theory. This shows the utility of my approach and contributes to understanding this notoriously strange result, encouraging more widespread use of category theory-derived ideas.

3.3 The Product Extension

I extend functor string diagrams to consider products. This section sets up the infrastructure for both the details of axes and separate data to be considered. This topic is briefly covered by Marsden (2014) using subdiagrams, and by Nakahira (2023) using specialized notation. However, to represent the free construction of morphisms within an axis using products, I had to develop novel notation that employs a "pseudomorphism". Furthermore, relying on the principle of vertical section decomposition and equivalent expression instead of colored regions leads to a more natural expression of monoidal products and bifunctors than those previous works.

3.4 Broadcasting

I conduct an in-depth category theory-based analysis of broadcasting. Broadcasting is critical to understanding how deep learning networks scales, and confidently diagramming it is an explicit goal of my approach. The deep learning community needs a clear, general definition of broadcasting. Works like Chiang et al. (2023) had to develop bespoke infrastructure to consider it, and the PyTorch broadcasting semantics are difficult

to parse. PyTorch derives its broadcasting methodology from NumPy, indicating this is a widespread shortfall.

I present a general definition of broadcasting in. Then, I analyze what tools we need to consider broadcasting in neural circuit diagrams. This leads to the insight that a few key specifications – a monoidal index category – is sufficient for broadcasting to be well-defined. Furthermore, such index categories create a correspondence between *coprojections* – elements which identify functions – and indexes for broadcasted axes. Additionally, I prove that broadcasting is independent of the choice of indexes used to define it. These proofs let broadcasting be confidently used in diagrams.

3.5 From Theory to Application: A Robust Basis for Neural Circuit Diagrams

I show how neural circuit diagrams emerge from a specification of Cartesian monoidal index categories. This provides a solid mathematical foundation for neural circuit diagrams. Then, I show particular category theory insights regarding natural transformations that let us better understand algorithms. I show how they guide us from a diagram that describes multi-head attention to a diagram that implements it.

Chapter 4: *Future Work*

Neural circuit diagrams offer a comprehensive means of accelerating the development and analysis of models used in practice, while functor string diagrams are a robust graphical framework for compositional systems. This opens up many avenues for future research. In addition to the benefits of diagramming more architectures not covered in this work, tools could be developed to convert between diagrams and code automatically.

Additionally, the theory underpinning diagrams could be extended to consider compositional learning (Fong et al., 2019; Cruttwell et al., 2021) and probabilistic mechanics (Fritz and Rischel, 2020; Fritz et al., 2023; Perrone, 2022). This would allow the mathematical nature of all architectures that can be represented by neural circuit diagrams to be rigorously analyzed. This would provide a mathematical foundation for almost all contemporary deep learning models, contrasting with existing theoretical work that often focuses on toy models (Zhang et al., 2017; Saxe et al., 2019; Li et al., 2022; Cruttwell et al., 2021).

Theory developed from neural circuit diagrams would apply to all models that can be represented with them, encompassing an immense scope. In this way, neural circuit diagrams are vital for bridging the gap between the application and theory of deep learning architectures.

Chapter 2

Applications of Neural Circuit Diagrams

Neural circuit diagrams intend to be used by various parties, from students first learning the subject to deep learning researchers investigating their mathematical foundations. This makes their presentation delicate; different audiences will want to know different things. This chapter aims to give a general introduction emphasizing using neural circuit diagrams to understand applied models better.

Separating this chapter from my more dense mathematical investigations serves a few purposes. It shows that neural circuit diagrams can be accessible and serve as a genuine diagrammatic standard for the deep learning community. It shows they have clear value to the applied engineering of deep learning models. For some readers, this may be the first time that the transformer model, transposed convolution, or visual attention has been presented clearly. Finally, introducing neural circuit diagrams in an accessible way will ease the more mathematically dense next chapter and show the motivation for my abstractions.

2.1 The Philosophy of My Approach

As I am introducing these diagrams, I have a burden to explain how I think they should be used and to address criticisms of creating a diagramming standard in the first place. I will take a brief aside to address these points, which I 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. I do not cover recursive or branching models in this work. Furthermore, I assume data is always in the form of tuples of tensors. Generalizing diagrams to further contexts is an exciting avenue for future research.

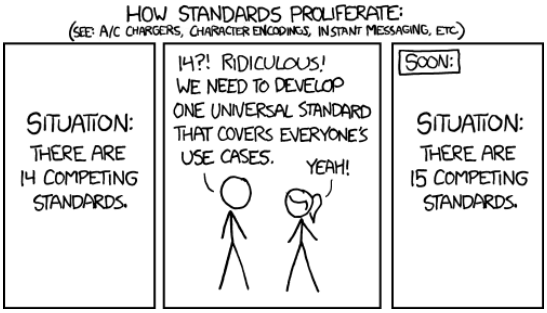
By making these assumptions, I develop diagrams specialized for some of the most essential but difficult-to-explain systems in artificial intelligence research. Researchers outside the narrow scope of sequential-tensor deep learning models often rely on these tools. By more clearly communicating them,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

I do not expect two independent teams to diagram architectures the exact same way. Indeed, I 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. Instead, the benefit of my framework is to have comprehensive, robust diagrams with clear correspondence to implementation and analysis, in contrast to ad-hoc diagrams, which often fail to include critical information.

Neural circuit 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 an 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 introducing a new standard simply increases the number of standards, worsening the issue trying to be solved (below). I 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 and Stay, 2010).



2.2 Reading Neural Circuit Diagrams

2.2.1 Commuting Diagrams

We aim to craft diagrams that precisely represent deep learning algorithms. While these 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. To begin, let's delve into an intuitive diagram, where symbols represent data types, and arrows signify the functions connecting them.

Note, I use forward composition with ";", meaning $f : \text{str} \to \text{int}$ composes with $g : \text{int} \to \text{float}$ by $(f; g) : \text{str} \to \text{float}$.

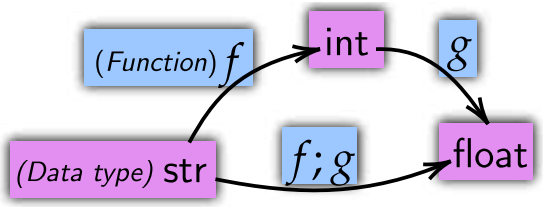


Figure 2.1: We have two functions: $f : \text{str} \to \text{int}$ and $g : \text{int} \to \text{float}$. These functions can be composed into a single function $(f; g) : \text{str} \to \text{float}$. In commuting diagrams, we represent data types, such as str, int, and float, with floating symbols, while functions are denoted by arrows connecting them.

2.2.2 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 a memory state is a tuple of the variables which compose it. So, a state containing an int and a str would have a type $\text{int} \times \text{str}$.

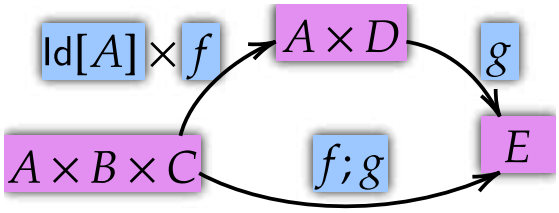


Figure 2.2: Here, I diagram two functions, $f : B \times C \to D$ and $g : A \times D \to E$, acting together. To represent the full memory states, we are required to amend f into $\text{Id}[A] \times f : A \times (B \times C) \to A \times D$

Consider a single algorithmic step acting on a compound memory state $A \times B \times C$. A function $f : B \times C \to D$ acting on this memory state would give an overall step with

shape $\mathrm{Id}[A] \times f : A \times (B \times C) \to A \times D$. Note, that $\mathrm{Id}[A]$ is the identity. We need to indicate A, even though f does not act on it, so that the initial and final memory states are properly shown. In Figure 2.2, I diagram f along another function $g : A \times D \to E$.

2.2.3 String Diagrams

These commuting diagrams fall short, however. As algorithms scale, operations and memory states get more complex. Usually, functions only act on some variables. However, it is not clear how to these targeted functions. Compound data types and compound functions are better suited by reorienting diagrams as in Figure 2.3. We will have horizontal wires represent types, and symbols represent functions. Diagrams are forced to horizontally go left to right.

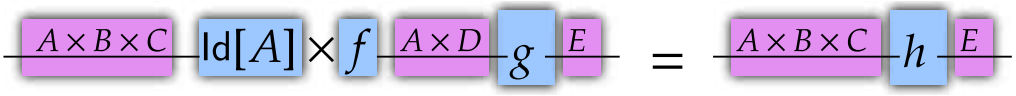


Figure 2.3: We reorient diagrams to go left to right. Wires represent data types, and symbols represent functions. This expression defines h.

This reorientation allows us to represent 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. In Figure 2.4, we see a clear reexpression of Figure 2.3. Here, we have the unchanged A variable untouched by f, which acts only on $B \times C$.

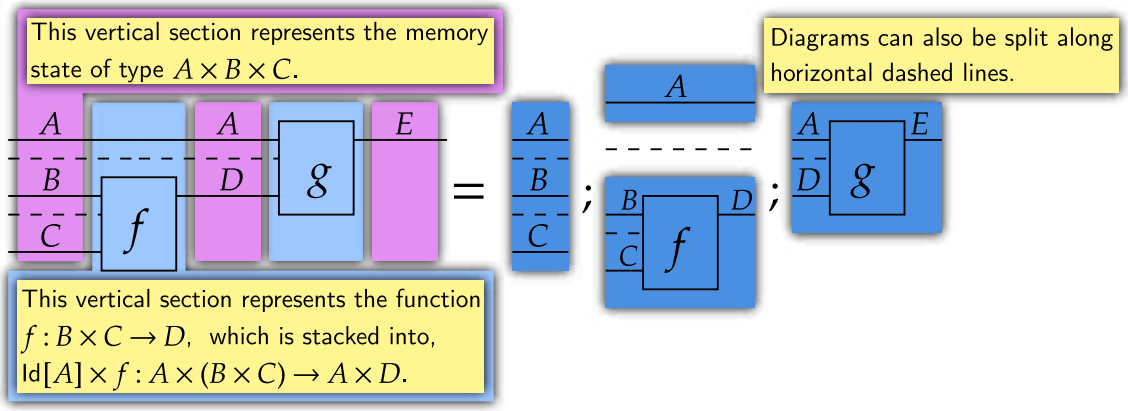


Figure 2.4: Tupled data types are diagrammed with wires separated by dashed lines. This clearly shows when functions act on only some variables.

Every vertical section of a diagram represents something. Either, it shows which data type is present in memory, or which function is applied at this step. Diagrams can always be decomposed into vertical sections, each of which must compose with adjacent sections to ensure algorithms are well-defined. Diagrams can also be split along dashed

lines. Diagrams are built from these vertically and horizontally composed sections, with wires acting like jigsaw indents.

2.2.4 Tensors

Diagrams will be specialized to represent tensors. Memory states will be tuples of tensors. 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, a table $\mathbb{R}^{4\times3}$ is a rank 2 tensor, and so on. If our diagram takes tensor data types, we get something like Figure 2.5.

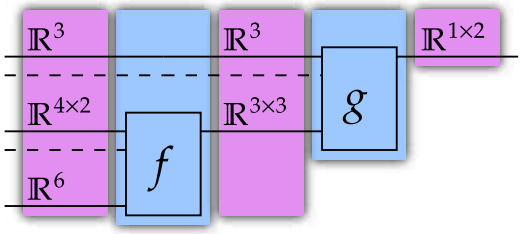


Figure 2.5: Similar to Figure 2.4, but with data types being tensors.

However, we benefit from diagramming the details of axes. Instead of diagramming a wire labeled $\mathbb{R}^{a\times b}$, we diagram a wire labeled a and a wire labeled b, without a dashed line separating them. This lets us diagram Figure 2.5 into the clear form of Figure 2.6.

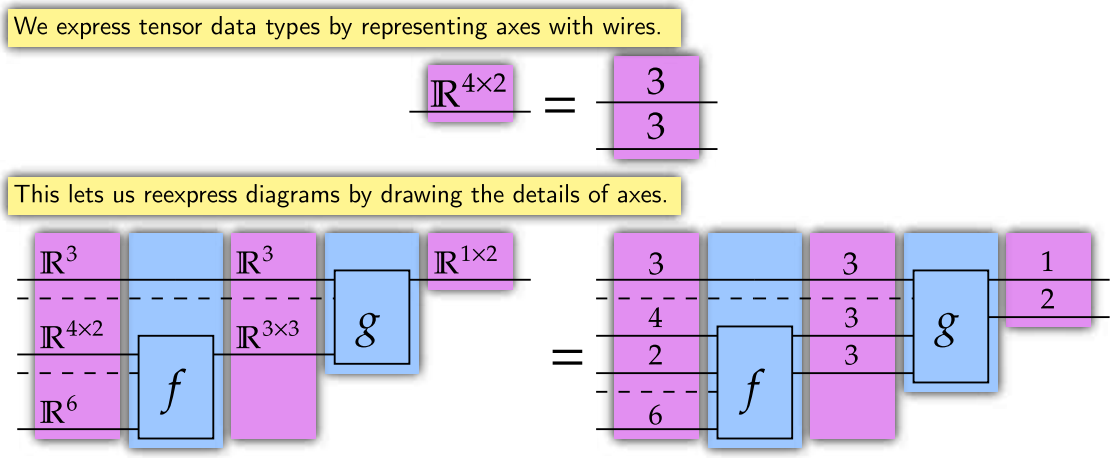


Figure 2.6: We can diagram types $\mathbb{R}^{a\times b}$ as two wires labeled a and b, without a dashed line separating them. (*See cell 2, Jupyter notebook.*)

2.2.5 Indexes

Values in tensors are accessed by indexes. A tensor $A \in \mathbb{R}^{4\times3}$, for example, has constituent values $A[i_4, j_3] \in \mathbb{R}$, where $i_4 \in \{0\ldots3\}$ and $j_3 \in \{0\ldots2\}$. Indexes can also be

used to access subtensors, so we have expressions $A[i_4, :] \in \mathbb{R}^3$. This subtensor extraction is therefore an operation $\mathbb{R}^{4\times3} \to \mathbb{R}^3$. We diagram it by having indexes act on the relevant axis. Indexes are diagrammed with pointed pentagons, or kets $|\_\rangle$. This type of subtensor extraction is diagrammed according to Figure 2.7.

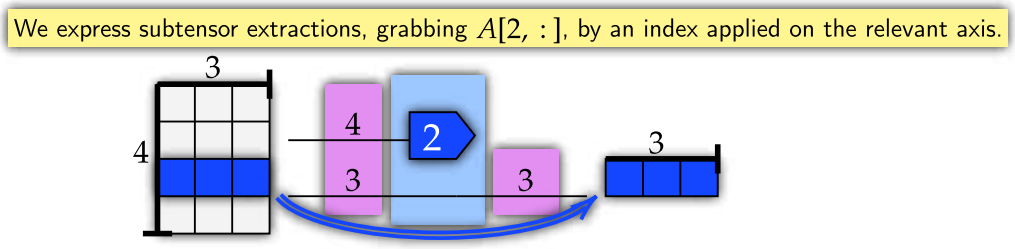


Figure 2.7: We diagram indexes with pointed pentagons labeled with the index being extracted. (*See cell 3, Jupyter notebook.*)

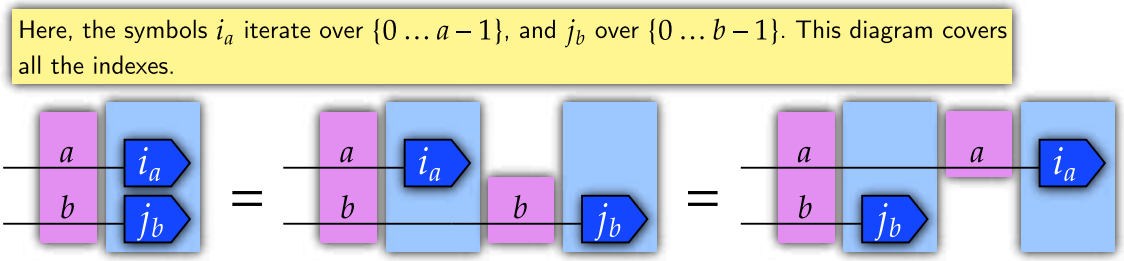


Figure 2.8: These subtensors are defined such that $A[i_4, :][j_3] = A[i_4, j_3]$. This expression is the same in the reverse order. (*See cell 4, Jupyter notebook.*)

2.2.6 Broadcasting

Broadcasting is critical to understanding deep learning models. It lifts an operation to act in parallel over additional axes. Here, we show an operation $G : \mathbb{R}^3 \to \mathbb{R}^2$ lifted to an operation $\mathbb{R}^{4\times3} \to \mathbb{R}^{4\times2}$. We diagram this broadcasting by having the 4-length wire pass over G, adding a 4-length axis to its input and output shapes. This is shown in Figure 2.9.

Inner broadcasting acts within tuple segments. A $\mathbb{R}^{4\times3} \times \mathbb{R}^4$ collection of data can be reduced to $\mathbb{R}^3 \times \mathbb{R}^4$ in 4 different ways. Therefore, there are 4 ways of applying an operation $H : \mathbb{R}^3 \times \mathbb{R}^4 \to \mathbb{R}^2$ to it. This gives a function lifted by "inner broadcasting", which has a shape $\mathbb{R}^{4\times3} \times \mathbb{R}^4 \to \mathbb{R}^{4\times2}$. We diagram this by drawing a wire from the source tuple segment over the function, as shown in Figure 2.10. This adds an axis of equal length to the target tuple segment and to the output, reflecting the shape of the lifted operation.

Broadcasting naturally represents element-wise operations. A function on values $f : \mathbb{R}^1 \to \mathbb{R}^1$, when broadcast, gives an operation $\mathbb{R}^{1\times a} \to \mathbb{R}^{1\times a}$. One length axes do

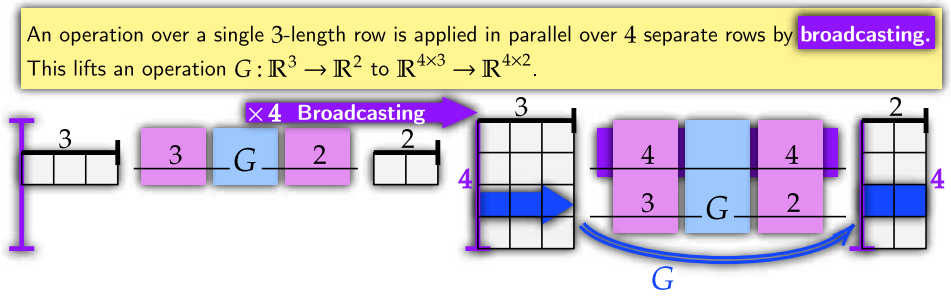


Figure 2.9: An operation is lifted over a 4-length axis by broadcasting. This applies G over corresponding subtensors. *(See cell 5, Jupyter notebook.)*

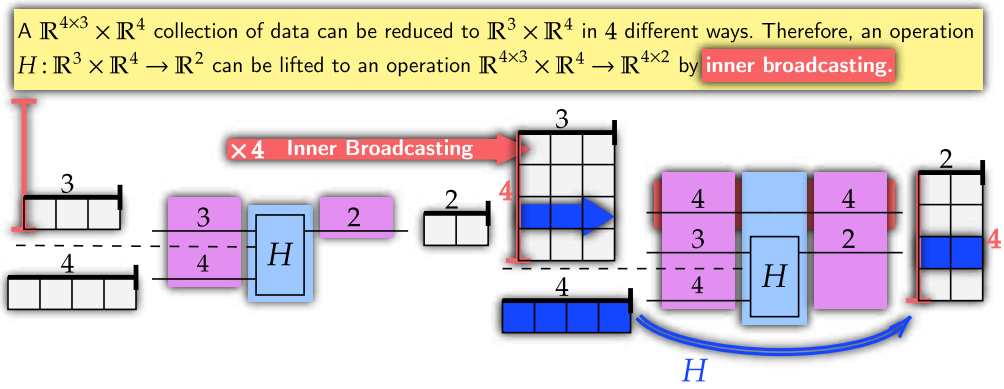
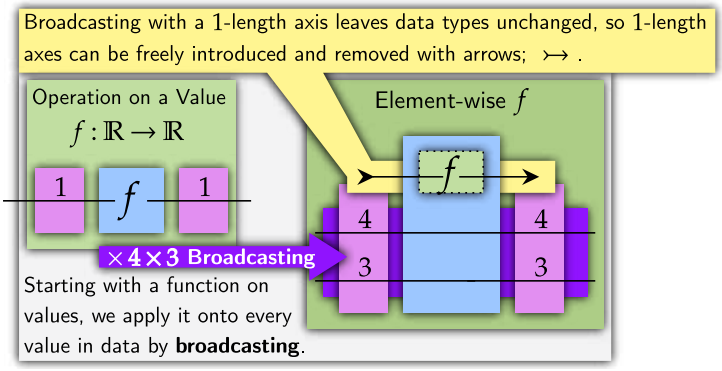


Figure 2.10: Lifting an operation within a tuple segment gives inner broadcasting. We diagram it by having a wire from the target tuple segment over the function, reflecting the shape of the lifted function. *(See cell 6, Jupyter notebook.)*

not change the shape of data, and can be freely amended or removed from pre-existing shapes by arrows. This means we diagram element-wise functions by drawing incoming and outgoing arrows, which represent the amendment and removal of a 1-length axis. This is shown in Figure 2.11.

Figure 2.11: Element-wise operations can be naturally shown with broadcasting. *(See cell 7, Jupyter notebook.)*



2.2.7 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 and Stay, 2010).

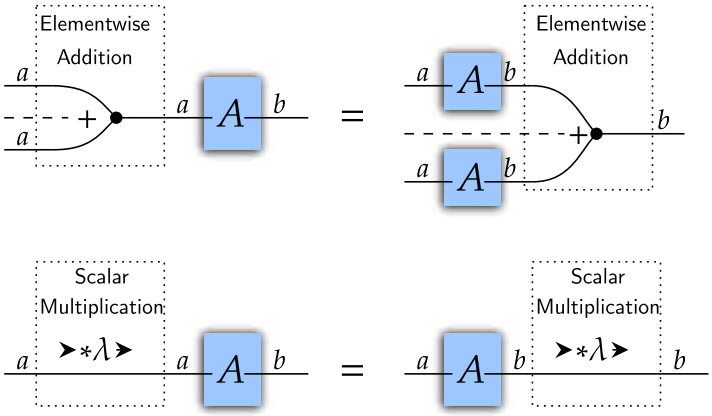


Figure 2.12: A subset of functions between $\mathbb{R}^a$ to $\mathbb{R}^b$ are linear, obeying additivity and homogeneity. This class of functions are closed under composition and has many important composition properties.

However, a pure monoidal string diagram has difficulty representing non-linear operations, a noted issue (Chiang et al., 2023). My 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.

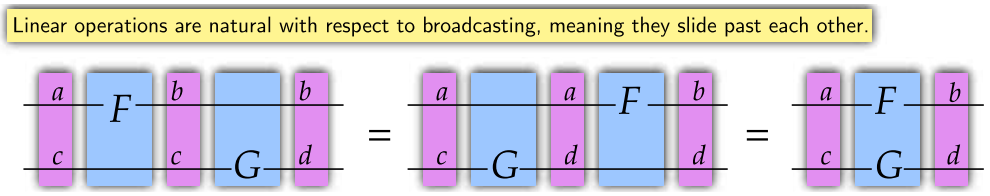


Figure 2.13: Linear functions are natural with respect to each other and broadcasting. This means the above equality holds, letting expressions be flexibly rearranged.

Linear functions are required to obey additivity and homogeneity, as shown in Figure 2.12. 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 2.13 holds. This means they can be simultaneously broadcast. This lets a series of linear functions be efficiently parallelized and flexibly rearranged.

2.2.8 Multilinearity

There is an important distinction between linear and multilinear operations. Inner products, for example, are multilinear. The inner product $u(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} = \Sigma_i \; x[i] \cdot y[i]$ 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 Δ, which we can show is linear.

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

To simultaneously broadcast multilinear functions, we note that 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 over 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 $M : \mathbb{R}^a \times \mathbb{R}^b \to \mathbb{R}^c$ have an associated tensor-linear form $M_\lambda : \mathbb{R}^{a \times b} \to \mathbb{R}^c$ such that $\otimes ; M_\lambda = M$. 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.

2.2.9 Implementing Linearity and Common Operations

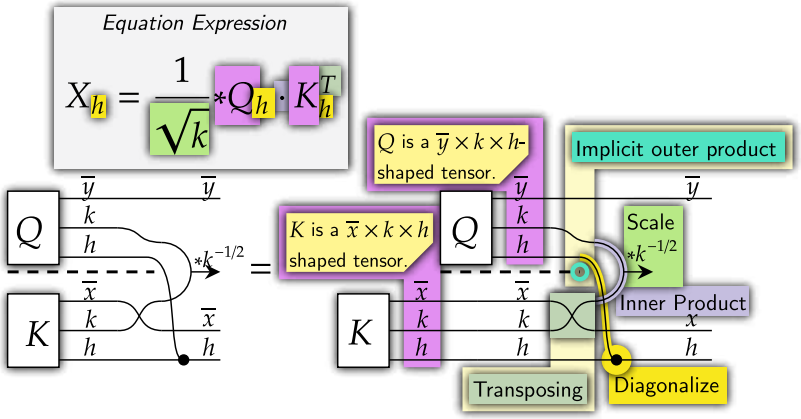
Key linear and multilinear operations can be implemented by the einops package, leading to elegant implementations of algorithms. Some key linear operations are **inner products**, which sum over an axis, **transposing**, which swaps axes, **views**, which rearranges axes, and **diagonalization**, which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

A good example that combines many of these operations is a section of multi-head attention shown in 2.14. 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

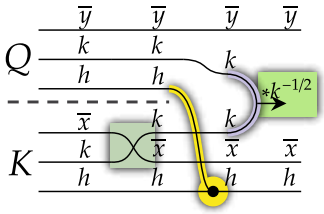
detail per thing. Though a complex expression, we can break this figure up as in Figure 2.6 and implement the interaction of wires using einops, shown in Figure 2.15.

Figure 2.14: 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)
```

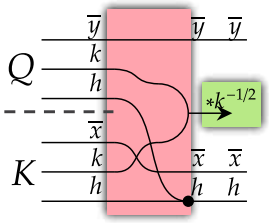


Figure 2.15: This section of multi-head attention can be implemented using the einsum operation. Note the close relationship between diagrams and implementation and how diagrams reflect the memory states and operations of algorithms. *(See cell 8, Jupyter notebook.)*

2.2.10 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 I provide code examples in cell 9 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 2.16. 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 better understand convolution (Section 2.3.3) and backpropagation (Section 2.3.6).

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 2.23, 2.28). To extract these rearrangements, we can selectively apply units and the inner product to reorient the direction of wires for linear operations.

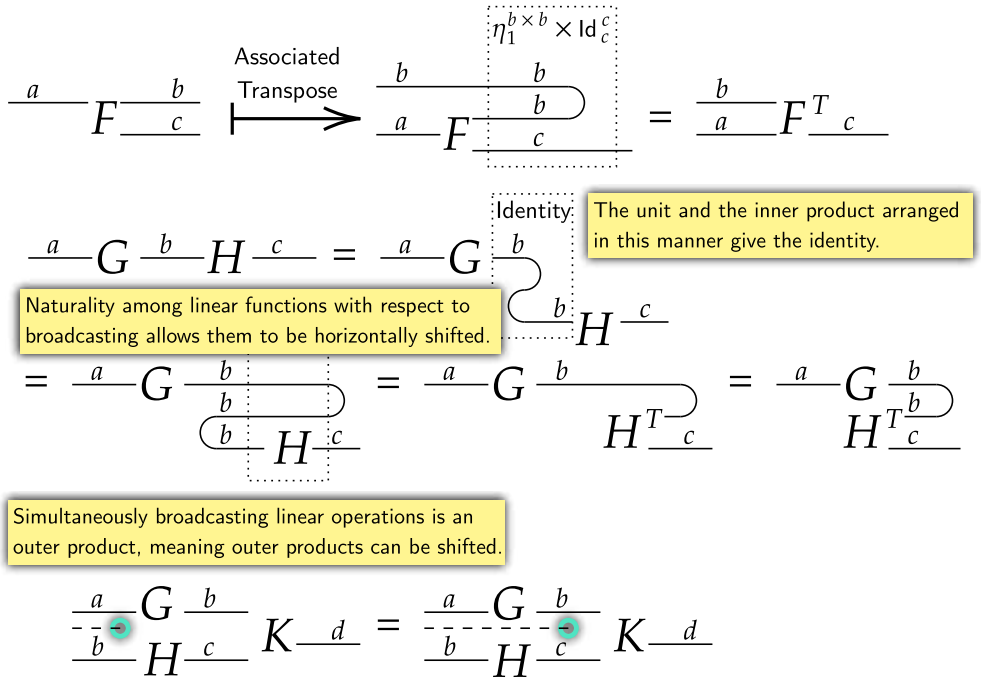


Figure 2.16: Linear operations have a flexible algebra. Simultaneous operations may increase efficiency (Xu et al., 2023). As the height of diagrams is related to the amount of data stored in independent segments, it gives a rough idea of memory usage. This is further explored in Section 2.3.6. *(See cell 9, Jupyter notebook.)*

2.3 Results: Key Applied Cases

2.3.1 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 2.17. We use pictograms to represent components analogous to traditional circuit diagrams and to create more memorable diagrams (Borkin et al., 2016).

```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
```

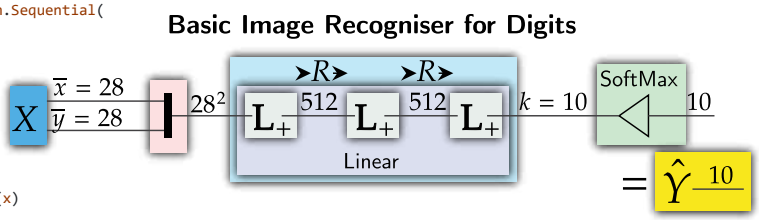


Figure 2.17: PyTorch code and a neural circuit diagram for a basic MNIST (digit recognition) neural network taken from an introductory PyTorch tutorial. Note the close correspondence between neural circuit diagrams and PyTorch code. *(See cell 10, Jupyter notebook.)*

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 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 since AlexNet(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, my method of displaying broadcasting makes it clear how SoftMax is applied.

2.3.2 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. 2.18 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. 2.19 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. 2.20 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.

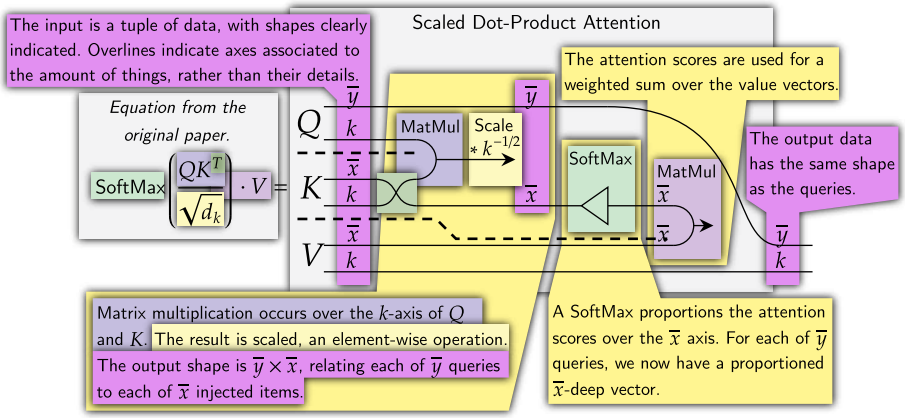
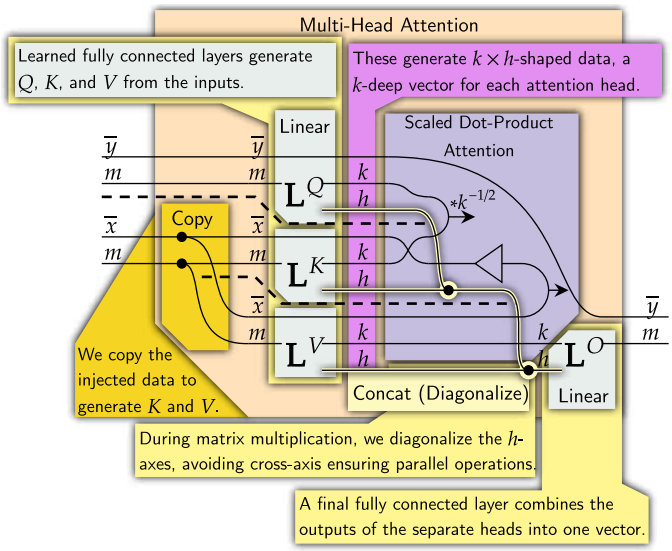


Figure 2.18: The original equation for attention against my diagram. The descriptions are unnecessary but clarify what is happening. Corresponds to Equation 1.1 and Figure 1.1.1. *(See cell 11, Jupyter notebook.)*

Figure 2.19: Neural circuit diagram for **multi-head attention**. Implementing matrix multiplication is clear with the cross-platform the einops package (Rogozhnikov, 2021). Corresponds to Equation 1.2 and 1.3 and Figure 1.1.2. *(See cell 12, Jupyter notebook.)*



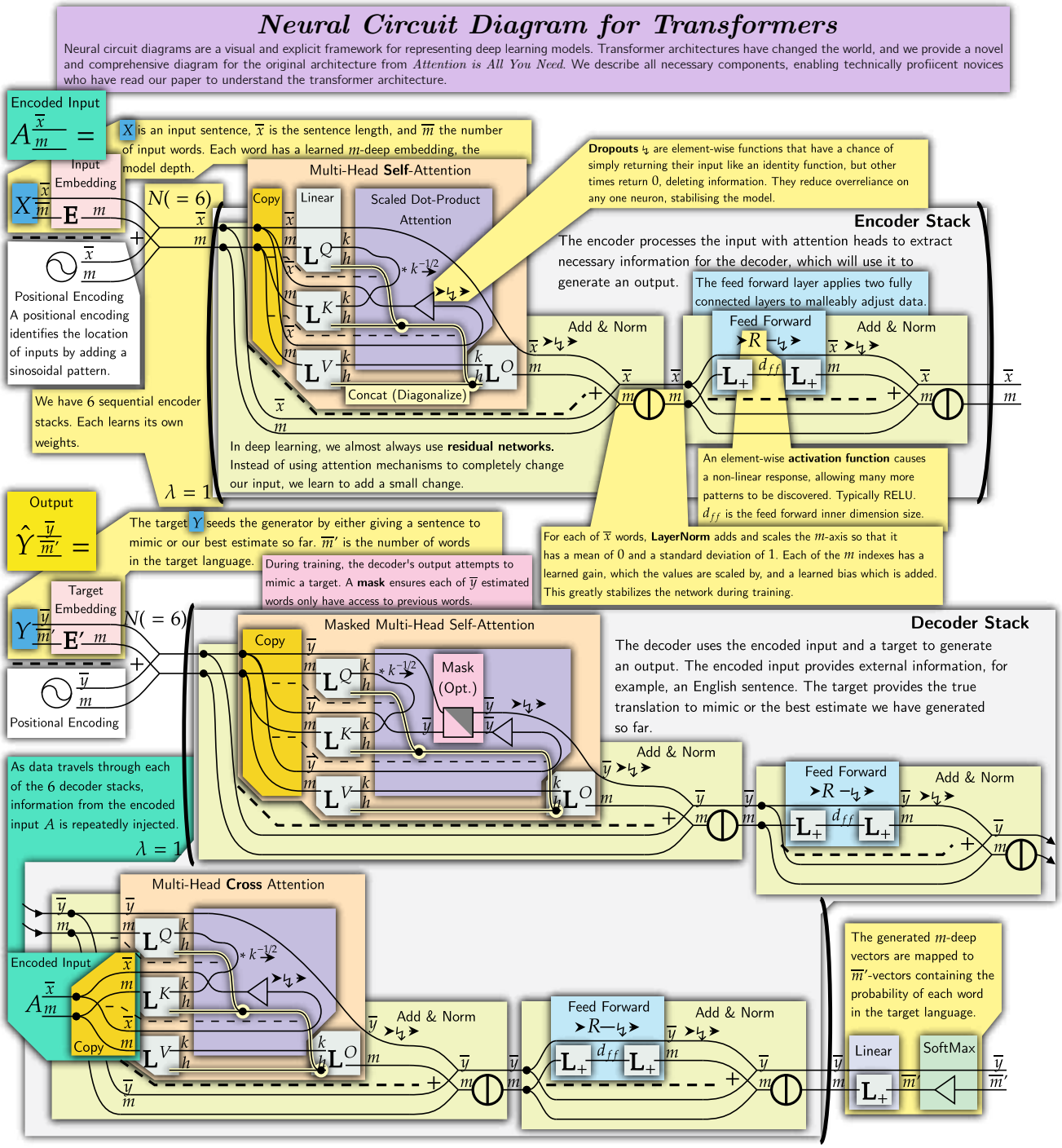


Figure 2.20: The entire encoder-decoder architecture from *Attention is All You Need* (Vaswani et al., 2017), expressed with neural circuit diagrams. Corresponds to Figure 1.1.3.

2.3.3 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 2.21. 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 their length by one bolded symbol akin to a vector. In 2 dimensions the convolution tensor becomes;

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

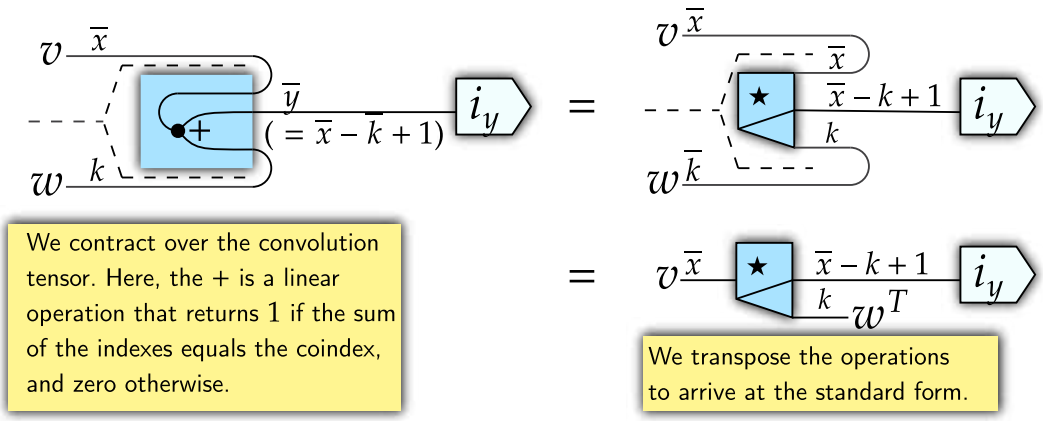


Figure 2.21: Convolution is a multilinear operation, with an associated tensor. This tensor is transposed into a standard form.

Figure 2.22 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, 1×1 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, 1×1 kernels are the same as broadcasting over the input.

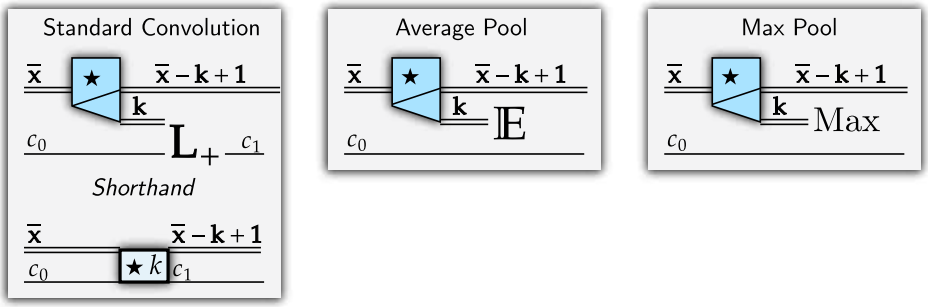


Figure 2.22: Convolution and related operations, clearly shown using neural circuit diagrams.

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 2.1. We diagram these changes by adding the s or

d multiplier where the axis meets the tensor as in Figure 2.23. These multipliers also change the size of the output, allowing for downscaling operations.

$$(\star\ s, d)[i_{\overline{y}}, j_k, \ell_{\overline{x}}] = \begin{cases} 1 & \text{, if } \ell_{\overline{x}} = s\ *\ i_{\overline{y}} + d\ *\ j_k. \\ 0 & \text{, else.} \end{cases} \tag{2.1}$$

$$\overline{y} = \left\lfloor \frac{\overline{x} - d\ *\ (k-1) - 1}{s} + 1 \right\rfloor \tag{2.2}$$

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 a high-dimensional output.

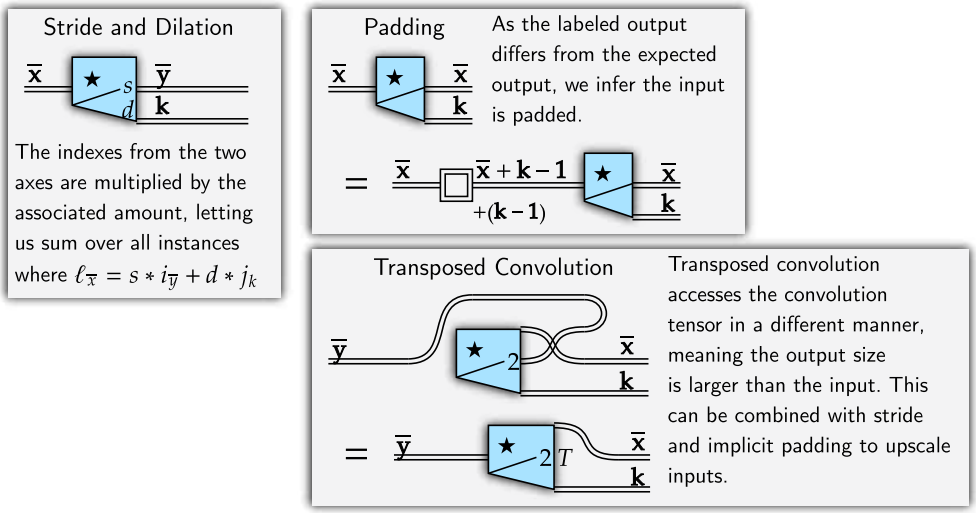


Figure 2.23: Stride, dilation, padding, and transposed convolution shown with neural circuit diagrams.

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.

2.3.4 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).

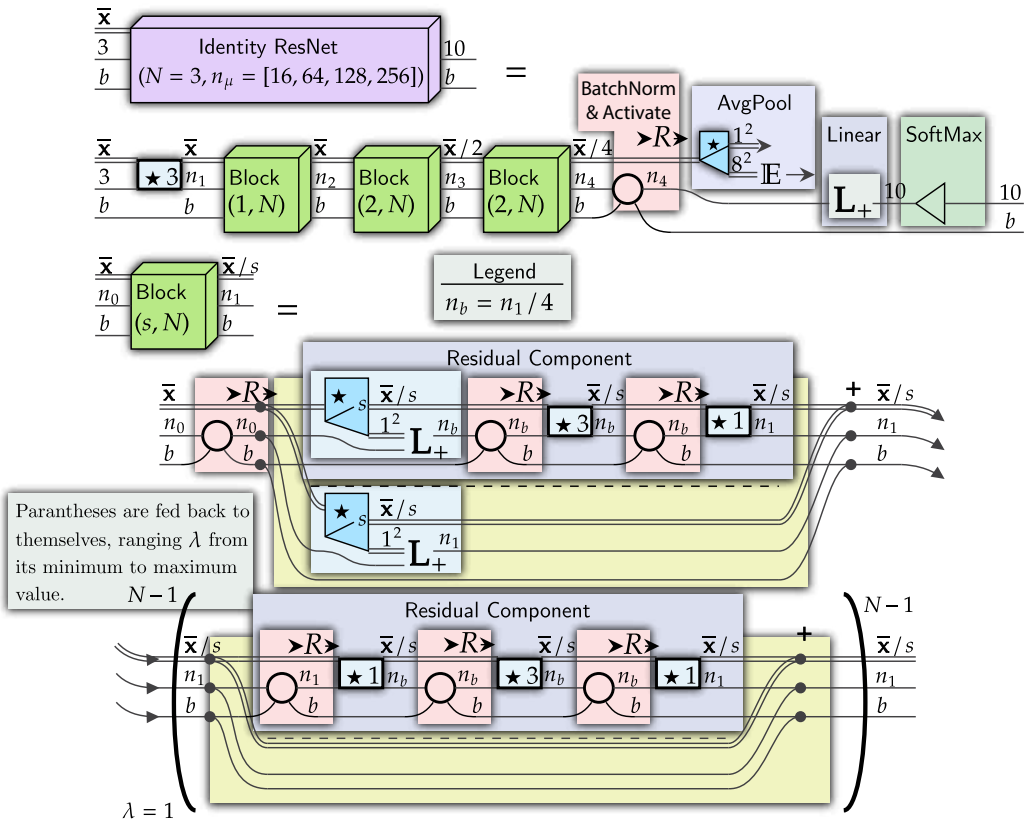


Figure 2.24: Residual networks with identity mappings and full pre-activation (IdResNet) (He et al., 2016) offered improvements over the original ResNet architecture. These improvements, however, are often missing from implementations. By making the design of the improved model clear, neural circuit diagrams can motivate common packages to be updated. *(See cell 13, Jupyter notebook.)*

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.

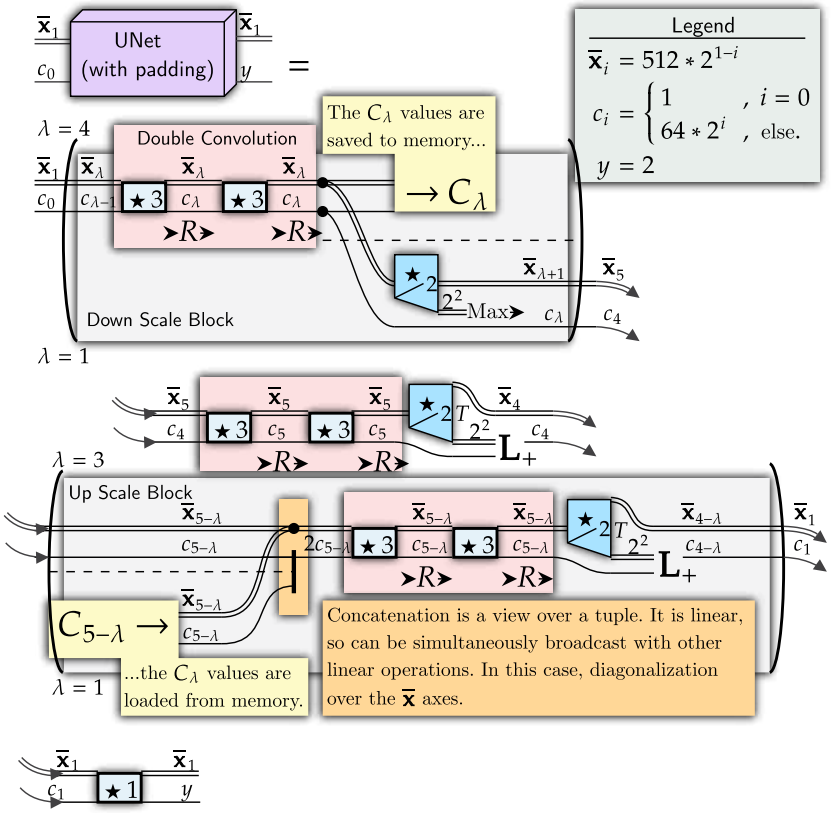


Figure 2.25: The UNet architecture (Ronneberger et al., 2015) forms the basis of probabilistic diffusion models, state-of-the-art image generation tools (Rombach et al., 2022). UNets rearrange data in intricate ways, which we can show with neural circuit diagrams. Note that in this diagram, I have modified 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.)*

As examples of neural circuit diagrams applied to computer vision architectures, I have diagrammed the identity residual network architecture (He et al., 2016) in Figure 2.24, which shows many innovations of ResNets not included in common implementations, as well as the UNet architecture (Ronneberger et al., 2015) in Figure 2.25, 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.

2.3.5 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 2.19 with tandem $\overline{\mathbf{y}}$ and $\overline{\mathbf{x}}$ axes and setting $h = 1$. As 1×1 convolutions are simply the identity, broadcasting a linear map over all of $\overline{\mathbf{y}}$ pixels is a 1×1-convolution. This leaves us with Figure 2.26 for a visual attention mechanism.

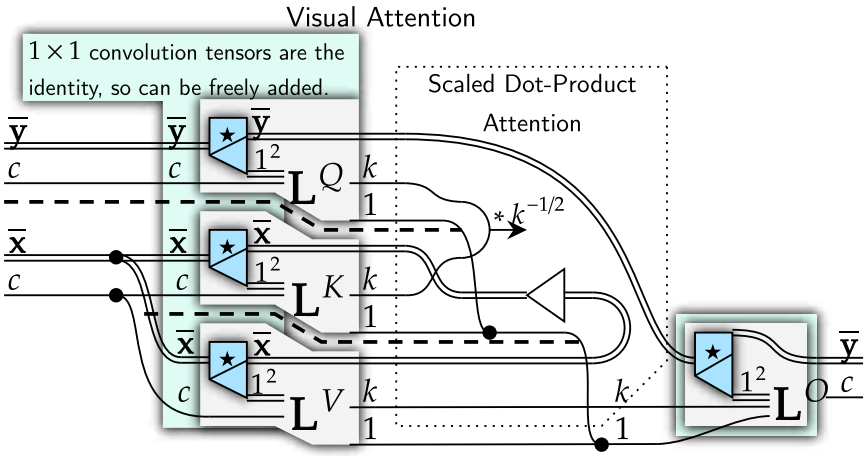


Figure 2.26: Using neural circuit diagrams, visual attention (Dosovitskiy et al., 2021) is shown to be a simple modification of multi-head attention (See Figure 2.19, Figure 2.14, cell 32, Jupyter notebook.)

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 2.14. 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.

2.3.6 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 2.6) 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.

$$\begin{aligned} h'(x) &= h'(x) \\ (g \circ h)'(x) &= (g' \circ h)(x) \cdot h'(x) \\ (f \circ g \circ h)'(x) &= (f' \circ g \circ h)(x) \cdot (g' \circ h)(x) \cdot h'(x) \end{aligned}$$

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 and 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, my 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 graphical linear algebra rules (see Figure 2.16) 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.

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 2.27.

$$\frac{\partial}{\partial x^a}(GF)^c\bigg|_{\mathbf{x}} = \sum_b \left(\frac{\partial}{\partial x^b} G^c \bigg|_{F(\mathbf{x})} \cdot \frac{\partial}{\partial x^a} F^b \bigg|_{\mathbf{x}} \right)$$

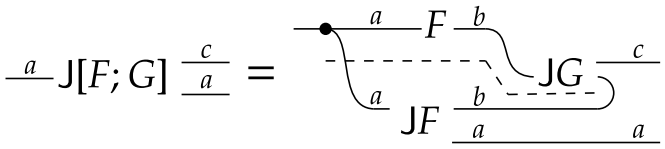


Figure 2.27: The chain rule expressed symbolically with index notation, and with neural circuit diagrams.

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.

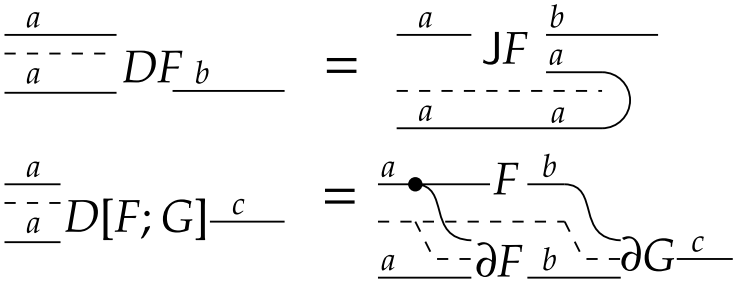


Figure 2.28: Definition of the forward derivative, and how functions compose under it.

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 2.29.

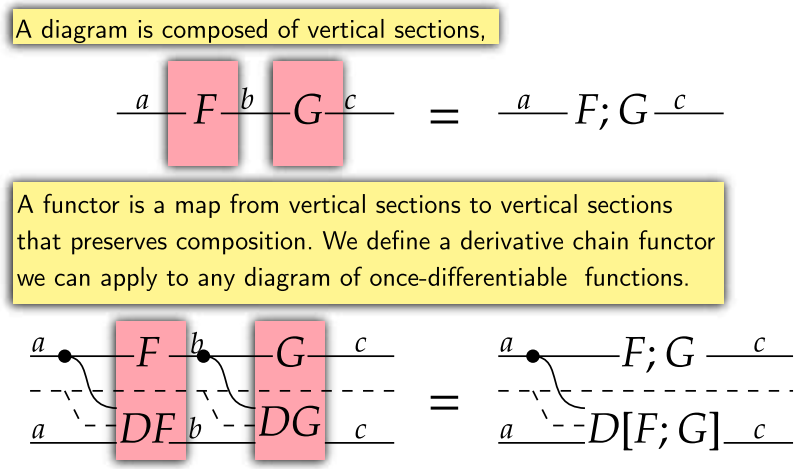


Figure 2.29: We have a composition preserving map, the $(\_, D\_)$ functor, that maps vertical sections to vertical sections, implementing the chain rule.

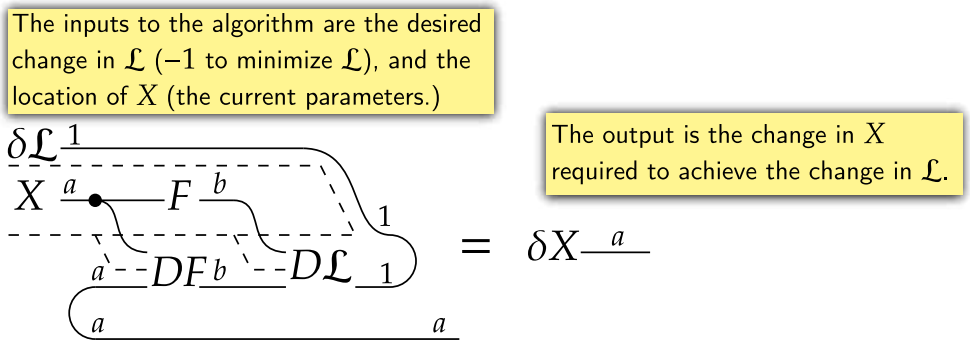
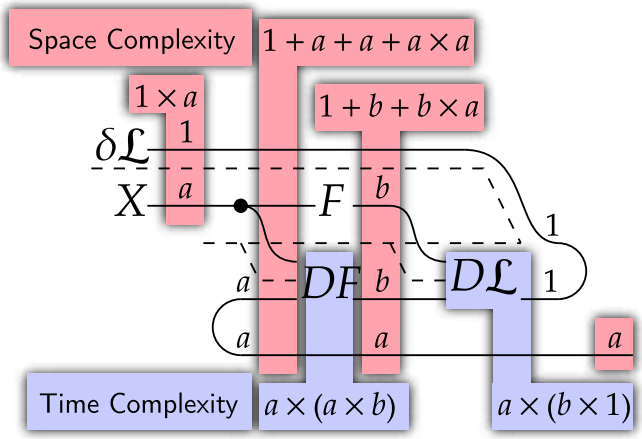


Figure 2.30: We turn a small chain rule expression into an optimization function by applying the inner product over the target direction and derivative output. An inner product over an axis of length 1 is just multiplication. Using the unit, we run this algorithm for every input degree of freedom, broadcasting over the a axis.

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 2.30.

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 2.31: 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 2.32. In Figure 2.34, we use the 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 2.32: 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.

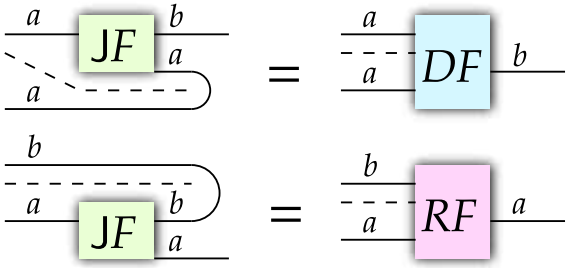
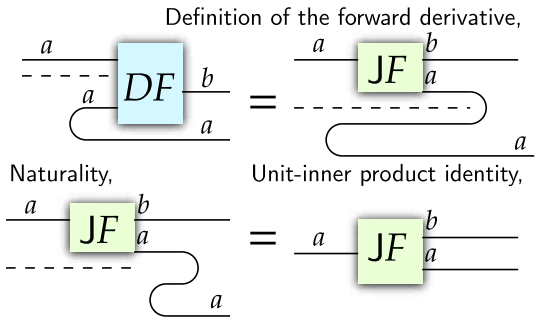


Figure 2.33: A full expression of how the unit and the forward derivative give the Jacobian. This demonstrates how linear algebra principles can illustrate the relationships between different forms of the derivative.



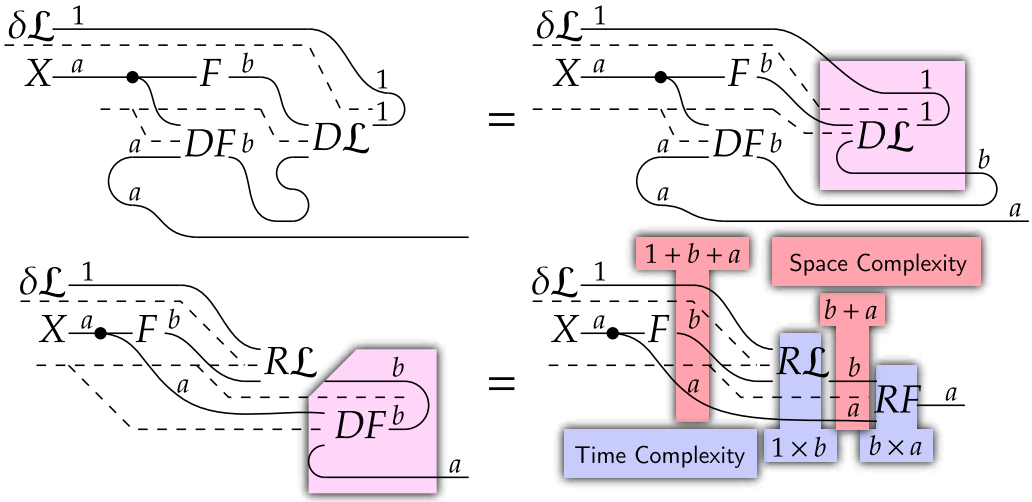


Figure 2.34: Using the previously developed linear algebra rules (see Figure 2.16, we rearrange the optimization algorithm to use the reverse instead of the forward derivative. This reveals the performance advantages of backpropagation.

2.4 Conclusion

Neural circuit diagrams improve the communication and analysis of deep learning architectures. Accessible to a wide range of users, they provide comprehensive descriptions of algorithms. Neural circuit diagrams have applied utility. Future work can see them extended to other architectures and have the initial family of symbols presented here improved and standardized.

Explicitly developing their mathematical foundation will associate every diagram with a precise mathematical expression to be analyzed and dissected. Extending neural circuit diagrams to consider the training of hidden parameters (Fong et al., 2019; Cruttwell et al., 2021) or the mechanics of information (Perrone, 2022) is possible within a category theory framework. However, we must first develop this foundation, which will be the focus of the next chapter.

Chapter 3

Theory of Functor String Diagrams

So far in this work, we have outlined the case for why deep learning requires improved communication and why category theory is a natural solution to this problem. Then, I presented neural circuit diagrams as an accessible graphical framework that can clearly communicate architectures. Now, I move onto establishing a robust foundation for neural circuit diagrams. This build on the diagrammatic scheme introduced by Marsden (2014) and extended by Nakahira (2023). Their approach has the most promise for providing a framework that can show both the details of axes and independent data. However, their approach needs to be further developed, and I spend this chapter contributing additional tools to reason about these functor string diagrams. Then, I do an in-depth study of broadcasting and, finally, reconcile functor string diagrams with neural circuit diagrams, placing them on a robust mathematical basis.

3.1 Building Blocks

First, we will lay out the basic elements of diagrams and two key principles that guide their design. Previous functor string diagram approaches have relied heavily on colors and symbols to express details (Marsden, 2014; Nakahira, 2023). My approach, on the other hand, centers on graphical intuition. Nonetheless, the diagrams, as laid out here, are able to prove theorems. In this section, I contribute this graphically intuitive perspective, which aims to improve the usefulness and adoption of functor string diagrams.

Contributions

In this section, I introduce functor string diagrams. Building on previous works, my presentation focuses on two key principles: the principle of vertical decomposition and the principle of equivalent expression. Centering these two principles allows even complex expressions to be decomposed and understood, and provides us a framework to consistently extend diagrams to new situations.

3.1.1 Categories

Staircase Notation We will often 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 Id^a_a. Additionally, we will use forward composition with ";" instead of backwards composition with "$\circ$" so that the direction of symbolic expressions align with the direction of diagrams.

Definition 1 (Categories, objects, morphisms)**.** *A category $\mathcal{C}$ consists of;*

- *A collection of objects, $a, b, c,$ etc. $\in Ob(\mathcal{C})$.*
- *Between any two objects $a, b \in Ob(\mathcal{C})$, a collection of morphisms $f^a_b, g^a_b, h^a_b,$ etc. $\in \mathcal{C}(a, b)$ between a and b. The first object is called the domain or source, and the latter is called the codomain or target.*
- *Composition, written as $f^a_b; g^b_c = (f; g)^a_c$, which maps $\mathcal{C}(a, b) \times \mathcal{C}(b, c) \to \mathcal{C}(a, c)$.*
  - *Which is closed, $f^a_b; g^b_c \in \mathcal{C}(b, c)$*
  - *Which is associative, $f^a_b; (g; h)^b_d = (f; g)^a_c; h^c_d$,*
- *An identity for every object, $Id^a_a \in \mathcal{C}(a, a)$ such that $Id^a_a; f^a_b = f^a_b = f^a_b; Id^b_b$;*

Anything which satisfies these conditions forms a category. The canonical example is **Set**, the category with sets as objects and functions between them as morphisms. However, we could just as easily have the category of relations **Rel**, which has sets as objects and *relations* between them as morphisms, allowing multiple outputs for each input. Other categories could express some specific property without direct analogy to sets and functions. For example, the *preorder on* $\mathbb{N}$ has natural numbers as objects and a morphism $\leqslant : a \to b$ whenever $a \leqslant b$. Composition, then, follows from $a \leqslant b$ and $b \leqslant c$ implying $a \leqslant c$.

Categories are composed of objects, morphisms, and equivalences. A category is seeded with morphisms that recursively define further morphisms through composition. This, however, may generate many long but useless expressions. So, categories need equivalence relations to be useful instead of just being an exercise in symbolic construction. Equivalence relations are, therefore, the "third component" of categories besides objects and morphisms.

Fundamentally, maps give a single output for each input. This means the amount of outputs is less than or equal to the amount of inputs. The composition map, $\mathcal{C}(a, b) \times \mathcal{C}(b, c) \to \mathcal{C}(a, c)$, is no different. In addition to implicitly indicating composition is closed, it notifies us composition may remove information about the inputs, imposing equivalence relations.

Categories form a type of symbolic grammar. Objects (syntax) indicate which morphisms (vocabulary) can be applied, and equivalences (meaning) indicate which expres-

sions are the same. This lets categories be perceived as purely symbolic constructs that almost coincidentally reflect the real world. This may seem needlessly abstract but it is about as good a description of mathematics generally as it is of category theory specifically.

This symbolic view grants us enormous flexibility to explore and question mathematical structures. For example, every category $\mathcal{C}$ has a dual category $\mathcal{C}^{\text{op}}$. In the dual category, morphisms $f : a \to b$ become morphisms $f^\dagger : b \to a$. Everything is reversed. What this category represents may not be clear, for example, what is $\mathbf{Set}^{\text{op}}$? Reverse functions? However, its objects, morphisms, and equivalence relations can be perfectly expressed. All composition and equivalence relations are simply reversed.

Because of cultural conventions, we are more used to thinking in terms of sets than categories. However, categories can give richer insight. They encompass sets by including the unit object 1 in **Set**, meaning that the elements of a set a correspond to the morphisms $\mathbf{Set}(1, a)$. Functions (morphisms in **Set**) on elements $x_a^1; f_b^a$ produce elements in the target set, $(x; f)_b^1 \in \mathbf{Set}(1, b)$. Shared patterns and important distinctions between elements, functions, and different categories can be carefully and consistently expressed using category theory, opening up powerful generalization tools.

3.1.2 Commutative Diagrams

The importance of equivalences to understanding categories gives rise to our first formal diagrams, *commutative diagrams*. With commutative diagrams, objects are symbols, anchoring morphisms which are arrows between them. If any two paths have the same origin and destination, and one path has more than one arrow, then the morphisms along those paths are equivalent (Abramsky and Tzevelekos, 2010, p. 11). The diagram "commutes". For example, Figure 3.1 contains a diagram that indicates that $f; g = h = i; j$, and another that shows $f; g = f; h$, but *not* $g = h$.

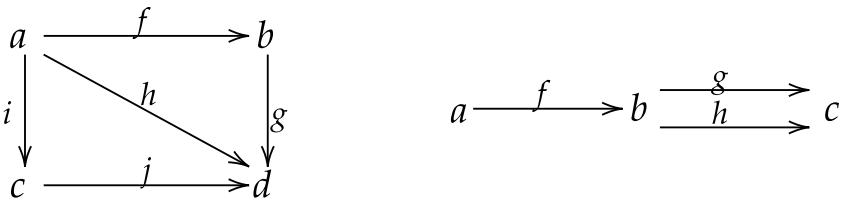


Figure 3.1: A commuting diagram showing $f; g = h = i; j$, and another where $f; g = f; h$, but *not* $g = h$.

3.1.3 String Diagrams

Just like in Chapter 2, typical commutative diagrams fall short and fail to express key insights clearly. They tend to assemble a host of symbols and have no obvious way of representing functors, composition-preserving maps between categories, or *natural transformations*, composition-preserving maps between functors. Exclusively using them, we

would miss out on many insights that category theory offers. Just as in the previous chapter, we move on to string diagrams.

Definition 2 (String diagrams)**.** *In string diagrams, the composition of morphisms in an arbitrary category are shown by arranging composed morphisms horizontally such that,*

- *Morphisms are represented by symbols,*
- *Morphisms are interspersed by representations of the current object,*
- *The current object must match the codomain of the morphism to the left, and the domain of the morphism to the right,*
- *Objects are represented by labeled horizontal wires,*
- *Identity morphisms are represented by labeled horizontal wires,*

Therefore, every vertical section must give an object or morphism expression which composes.

These diagrams are one-dimensional and have clear composition rules. Diagrams can be decomposed into vertical sections. Each vertical section can be understood in isolation, and a well-formed overall diagram follows as long as each isolated vertical section composes with adjacent ones.

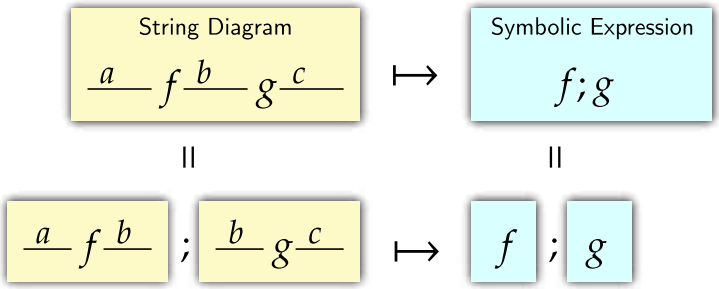


Figure 3.2: A basic string diagram emphasizing how diagrams can be decomposed into vertical sections and how these vertical sections relate to symbolic expressions.

I aim to develop a first-principles approach to understanding diagrams. This will allow them to be extended to new situations with ease, and understood even when they seem highly complex.

The fundamental structure of string diagrams gives us *the principle of vertical section decomposition.* Every diagram can be decomposed into vertical sections which must compose. Vertical sections ultimately decompose into alternating object and morphism expressions that can be clearly understood in isolation. This principle can be used to understand any string diagram, and we will preference perspectives that enable it.

Principle 1 (Vertical Section Decomposition)**.** *All string diagrams in a category $\mathcal{C}$ can be decomposed into vertical sections, each with incoming and outgoing wires. A diagram is well-formed if wires of adjacent sections match. Each vertical section maps to either a morphism in the category or an object in the category. Diagrams ultimately decompose into alternating object and morphism vertical sections.*

The principle has to be kept in mind when trying to understand diagrams. Aspects which might not seem to obey it, such as wires turning backwards in the case of linearity (see Section 2.2.7) can still be understood as one vertical section with two parallel wires and another vertical section merging them. Every diagram can be decomposed, and ultimately understood. Sticking to this principle will allow diagrams to handle a great deal of complexity.

3.1.4 Functors

The focus of category theory is composition. Functors are maps on objects and morphisms that preserve composition, and are therefore of immense interest to us. Using string diagrams and the principle of vertical section decomposition, we could diagram an expression employing functors such as $F[f;g] = F[f];F[g]$ as Figure 3.3. However, this gives little insight. Though the diagrams are built from well-composed vertical sections, the unique behavior of functors is unclear.

Definition 3 (Functors)**.** *A functor is a map between categories $F : \mathcal{C} \to \mathcal{D}$ that provides;*

- *A map between objects, $F : Ob(\mathcal{C}) \to Ob(\mathcal{D})$;*
- *A map between homsets, $F : \mathcal{C}(a,b) \to \mathcal{D}(Fa, Fb)$;*
  - *Which preserves composition, $F\left[f_b^a; g_c^b\right]_{Fc}^{Fa} = F\left[f_b^a\right]_{Fb}^{Fa}; F\left[g_c^b\right]_{Fc}^{Fb}$. This implies the identity is preserved, $F\left[Id_a^a\right]_{Fa}^{Fa} = Id_{Fa}^{Fa}$;*

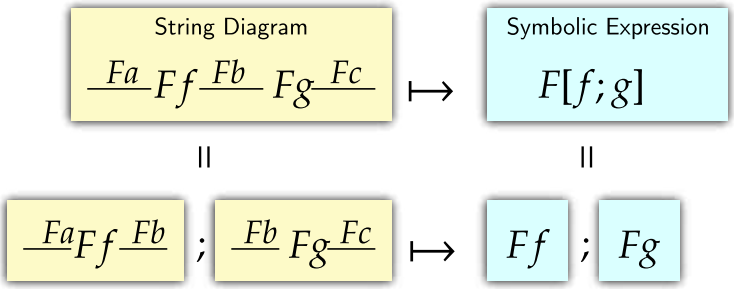


Figure 3.3: A basic string diagram, expressing the composition of functor-ed morphisms.

As functors act on both objects and morphisms, we will diagram them similarly for both. This is done with a labeled horizontal line passing over the target morphism, as in

Figure 3.4. When diagrams are vertically decomposed, we interpret an F functor wire above an expression f as equivalent to drawing Ff directly. Therefore, we see how the principle of vertical decomposition helps us understand any diagram as vertical sections that can be understood in isolation.

Functor wires build on the second principle of our diagrams, *equivalent expression*. Additional notation will mostly be equivalent ways of expressing something that could be done by one dimensional string diagrams. For example, functor wires F add an additional wire, a new feature. However, an F functor wire over f could have been written as Ff without an additional line. Therefore, the additional line can be understood as clarifying diagrams and their properties rather than fundamentally changing what they express and how they are read.

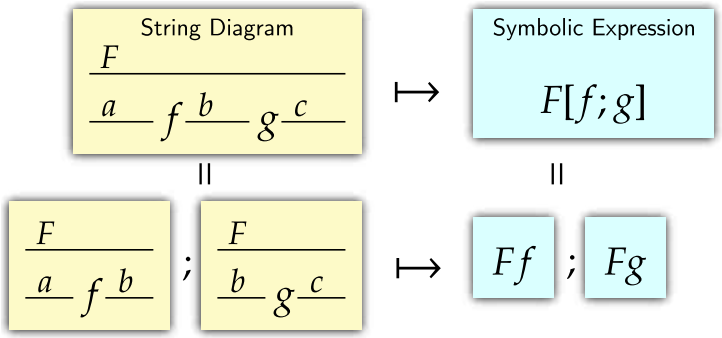


Figure 3.4: An equivalent expression of Figure 3.3 representing functors as wires. The feature of drawing functors as wires is an equivalent expression that aids graphical regularity.

Principle 2 (Equivalent Expression)**.** *Additional features in diagrams, from functor wires to arrows, are equivalent means of expressing pre-existing concepts. New features are designed to support graphical intuition, especially regarding equivalence relations.*

Applying the $F : \mathcal{C} \to \mathcal{D}$ functor to a diagram in $\mathcal{C}$ gives us a diagram in $\mathcal{D}$. Vertical sections that give objects and morphisms in $\mathcal{C}$ are mapped to vertical sections that give objects and morphisms in $\mathcal{D}$. We can then diagram morphisms from the objects $Fa, Fb \in \mathcal{D}$ which are not present in the image of the functor. Recall that as long as every vertical section indicates an object or morphism that composes, diagrams are well-defined. By the principle of equivalent expression, if F maps $a \in \mathcal{C}$ to $\alpha \in \mathcal{D}$, then a wire labeled α is the same as a wire labeled a with an F functor wire above it.

3.1.5 Natural Transformations

Functor wires are distinct from underlying object wires. We cannot directly apply morphisms from the base category onto them. However, they can accept natural transformations, which are composition-preserving maps between functors.

Natural transformations give a map between two functors $F, G : \mathcal{C} \to \mathcal{D}$. They allow an expression using one functor to be turned into an expression using the other. This is

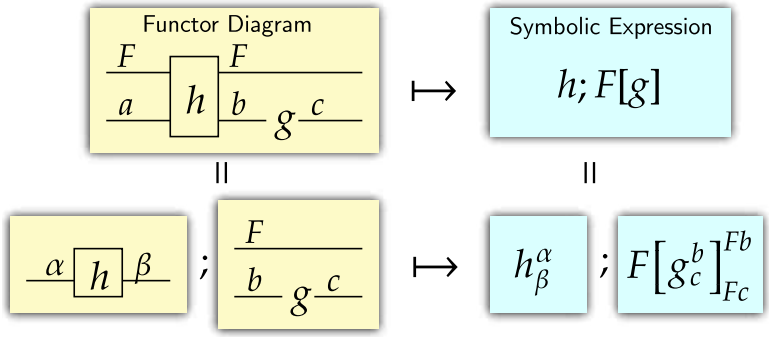
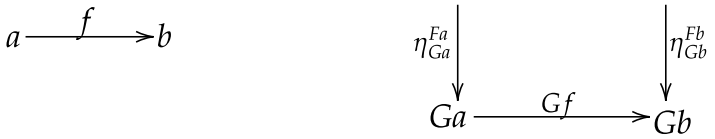


Figure 3.5: By the principle of equivalent expression, a morphism $h : \alpha \to \beta$ can be diagrammed as $h : Fa \to Fb$ if $Fa = \alpha$ and $Fb = \beta$. The Fa and Fb labeled wires can be equivalently expressed by wires labeled a and b with an F functor wire above. We can do this even if h^α_β is not in the image of F.

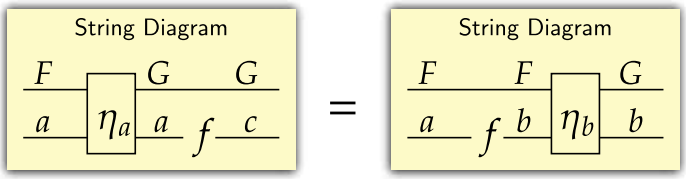
achieved by having morphisms called *components* in $\mathcal{D}$ between corresponding objects from $\mathcal{C}$.

Definition 4. *A natural transformation $\eta : F \to G$ between two functors $F : \mathcal{C} \to \mathcal{D}$, $G : \mathcal{C} \to \mathcal{D}$ provides;*

- *For each object in $\mathcal{C}$, a morphism η^{Fa}_{Ga} in $\mathcal{D}$ called a component,*
- *Such that for all morphisms f^a_b in $\mathcal{C}$, it holds that $Ff^{Fa}_{Fb}; \eta^{Fb}_{Gb} = \eta^{Fa}_{Ga}; Gf^{Ga}_{Gb}$. This is equivalent to the diagram commuting for each f^a_b,*



Or, using functor string diagrams,



Graphical isotopy (Selinger, 2009) is the idea that certain graphical transformations of diagrams give equivalence relations. The above diagram gives a sort of isotopy; components can "pass through" morphisms. Furthermore, components are defined for every object and so do not need to reference their underlying object. We develop an equivalent expression for natural transformations that clarifies diagrams, only drawing them on the

$F \to G$ wire without referencing the underlying object. Shifting this component symbol horizontally gives an equivalence relation.

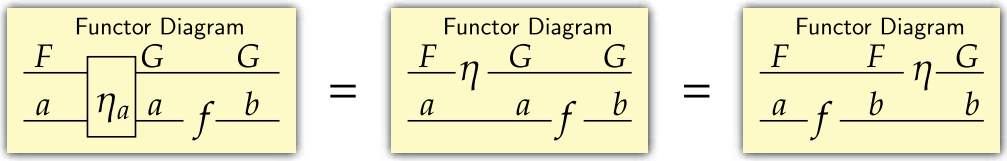


Figure 3.6: Natural transformations can act on the functor wire. The diagrams on the right can be deciphered by vertical section decomposition. Natural transformations acting on the functor wire $F \to G$ are an additional feature that are defined as an equivalent expression for a component $\eta[a] : Fa \to Ga$, where a is the underlying object.

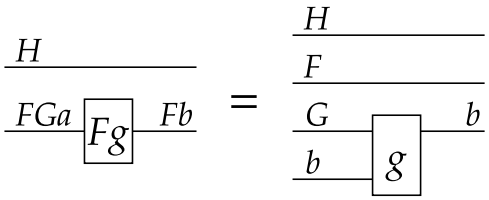
This framework shows how isotopy and natural transformations are closely related. Isotopy means movement gives equivalence relations. As natural transformations are slid along a functor wire, we get diagrams which are mapped to different expressions. As components are defined for every underlying object, these expressions exist. Naturality tells us there is an equivalence relation between these expressions.

We can now outline the broad behavior of functor string diagrams.

Definition 5 (Functor string diagrams)**.** *Functor string diagrams are string diagrams with the additional properties that,*

- *Functors are represented by wires above the objects or morphisms they are acting on,*
- *Natural transformation components are represented by the natural transformation acting on functor wires without referencing the underlying base object.*

Figure 3.7: Here, we have a morphism with multiple functor wires present. Using the principle of equivalent expression, this can be re-expressed into an easier-to-understand form.



In functor string diagrams, the case of multiple functors is easy to manage. Functors are shown by being placed above the objects or morphisms they act on. As we see below, this definition holds just as well for a functor placed directly on some $f : a \to b$, or for a functor placed over $Fg : FGa \to Fb$. This is shown in Figure 3.7. Functor wires are a new feature that abides by the principle of equivalent expression. This means complex diagrams can be understood by breaking them down into objects and morphisms in the underlying category.

3.2 Reasoning with Diagrams

My diagrams aim to provide clear, explicit graphical expressions to understand deep learning and other systems better. Therefore, we want diagrams to allow for graphical reasoning. They should intuitively suggest ways to rearrange expressions and allow us to explore properties that are difficult to intuit in a forest of symbols.

The previous works on string diagrams have developed some reasoning tools. The work by Marsden (2014) uses subdiagrams to consider Cartesian products and representable functors. However, the subdiagrams do not stem from a unified approach or provide much graphical intuition. Nakahira (2023) builds on this work, creating a comprehensive framework to consider a variety of category theory constructs.

My approach builds on these works by creating a general language that can consider families of expressions and then integrate them into diagrams. This lets us deal with certain ideas in an explicit manner by considering a type of lambda calculus using diagrams. This is a powerful framework that lets new operations be graphically defined and reasoned with.

I follow our established principles, including ensuring all diagrams can be decomposed into vertical sections and that new features graphically suggest equivalences. Reasoning with functor string diagrams is novel, and by contributing clearer graphical presentations, I provide a valuable new perspective to this nascent field.

As this section develops a method to reason with diagrams using category theory, there is potential for future work to integrate this approach into other works on logic and category theory (Awodey, 2010). This would be an interesting avenue for future research but is not the focus of this thesis.

Contributions

In this section, I build on the works of Marsden (2014) and Nakahira (2023) by letting diagrams reason about families of expressions. This lets new operations be graphically defined. This allows for more explicit reasoning with diagrams, letting them be robustly extended.

We use these tools to understand hom-functors and families of diagrams in the category of **Set**. Using them, I contribute a graphical derivation of the Yoneda lemma using my tools. The Yoneda lemma is a central result of category theory; letting it be intuitively understood contributes to a more widespread application of category theory tools.

3.2.1 Families

Morphisms in a category are expressions that derive meaning from how they behave when amended with other morphisms. To define a morphism, therefore, we need to consider its behavior under all possible amendments.

To define and reason about morphisms, we need the tools to discuss collections of expressions. This is done with families of expressions. An I-family is a map from a collection I, called the index collection, to some expression. In the context of category theory, this is usually morphism. So, an I-family expression in a category $\mathcal{C}$ would be a map $I \to \mathcal{C}(a, b)$. If two family expressions are equivalent, then they must be equivalent for every $i \in I$.

Each diagram gives a morphism from the object on the left to the object on the right. We can, therefore, represent families of expressions by families of diagrams. For example, we can define the natural transformation property by equating two families of diagrams, shown in Figure 3.8. Naturality has to hold over every underlying morphism, so the index collection are the morphisms $f : a \to b$ in the underlying category.

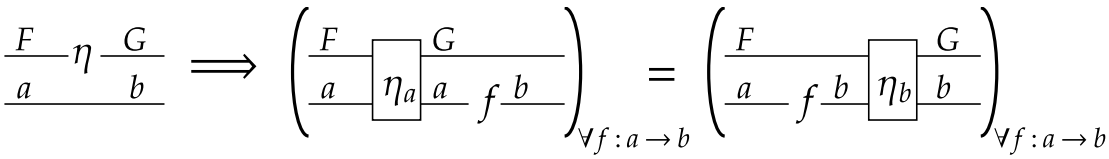


Figure 3.8: Here, natural transformations are defined by equating two families of diagrams. As families of diagrams are generally maps $I \to \mathcal{C}(\_, \_)$, in this case the families are maps $\mathcal{C}(a, b) \to \mathcal{D}(Fa, Gb)$. To be equal, the families must be equivalent for every $f \in \mathcal{C}(a, b)$.

3.2.2 Generating Objects

The category of **Set** can support particularly powerful logic, owing to the existence of a generating object. A generating object g in $\mathcal{C}$ uniquely identifies morphisms. So, if $e^g_a; f^a_b = e^g_a; h^a_b$ for all $e^g_a \in \mathcal{C}(g, a)$, then $f^a_b = h^a_b$. We recognize this as a family expression, shown with the diagram in Figure 3.9.

The generating object g is an object. However, morphisms from it are akin to how we typically think about elements. When diagramming generating objects, we benefit from not drawing their wires. This equivalent expression adds graphical intuition to diagrams by letting us think in terms of elements. Furthermore, this feature is clear. When we decompose a diagram into vertical sections, we know a generating object is present wherever the vertical sections are empty.

In a category with a generating object, morphisms $a \to b$ can be represented by family expressions from $\mathcal{C}(g, a)$ to $\mathcal{C}(g, b)$. Therefore, we can use these family expressions as morphisms in diagrams. In this case, the index object a becomes a wire on the left, and the b wire on the right of the diagrams is continued. The generic symbol λe is now a pending substitution. This gives a lambda calculus of diagrams, with the composition of family expression diagrams being a substitution.

As an exercise, we can show how lambda-calculus expressions compose. With a generating object, we can expand $f : a \to b$ over generating morphisms $e : g \to a$.

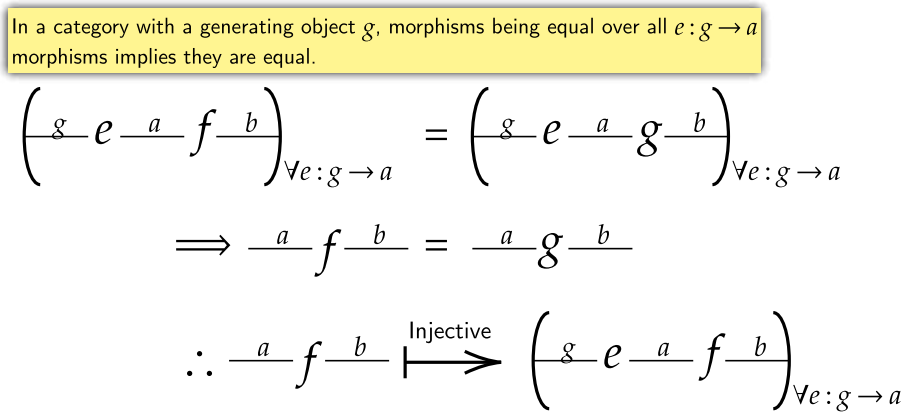
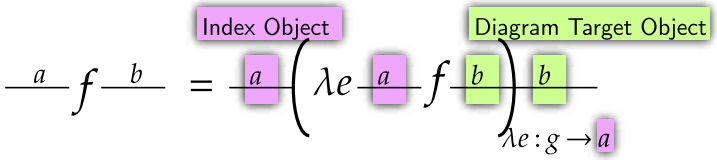
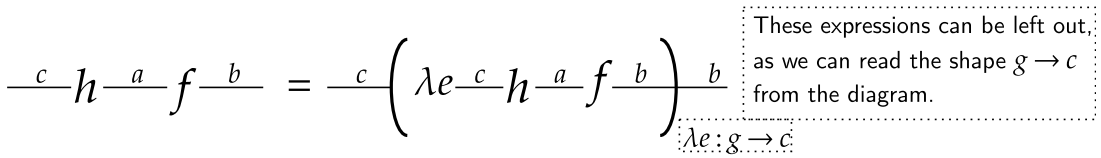


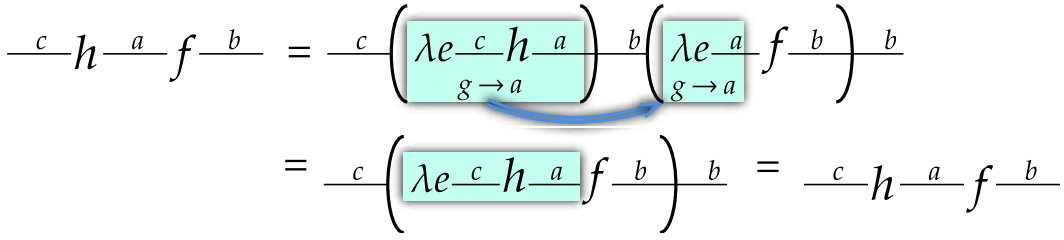
Figure 3.9: In a category with a generating object g, morphisms $f : a \to b$ are uniquely identified by their action on all the generating objects $e : g \to a$ of their source object. Therefore, if two morphisms generate the same $g \to b$ values for every $g \to a$ element, they are equal.



Composition moves inside diagrams, giving one expression for $h; f$.



Alternatively, we can compose $h; f$ using the lambda calculus rules.



In **Set**, this generating object is 1, the set with one element. The collection of morphisms $\mathbf{Set}(1, a)$ correspond to the elements of the set a. Functions are identified by how they map elements and are equal if they map elements the same way, meaning 1 is a generating object.

The generating object 1 in **Set** offers *free construction*, meaning a morphism $a \to b$ can be constructed to correspond to any family expression $\mathcal{C}(1,a) \to \mathcal{C}(1,b)$. In addition to identifying morphisms as generating objects usually do, in **Set** family expressions from the generating object can construct morphisms.

3.2.3 Hom-Functors

A category $\mathcal{C}$ is called "locally small" if the collection of morphisms between any two objects, $\mathcal{C}(a,b)$, forms a set. A family of diagrams in $\mathcal{C}$ indexed over morphisms $a \to b$, and producing morphisms $c \to d$, is a family $\mathcal{C}(a,b) \to \mathcal{C}(c,d)$. If $\mathcal{C}$ is locally small, then this family is a function in **Set**.

Families of diagrams in a locally small category $\mathcal{C}$ therefore define functions in **Set**. A particular application of this is for hom-functors. A hom-functor $\mathcal{C}(a,\_) : \mathcal{C} \to \mathbf{Set}$ built from $a \in \mathrm{Ob}(\mathcal{C})$ maps objects $\mathrm{Ob}(x) \in \mathcal{C}$ to $\mathcal{C}(a,x)$, their hom-set in **Set**.

To diagram the functor $\mathcal{C}(a,\_)$, instead of labeling a wire with $\mathcal{C}(a,\_)$, we label a wire with a and have an arrow going right to the left. This equivalent expression supports graphical intuition, as we will soon see.

For an object a and a morphism $g : x \to y$, we define a function in **Set** of the form $\mathcal{C}(a,g) : \mathcal{C}(a,x) \to \mathcal{C}(b,y)$ by the diagram in Figure 3.10.

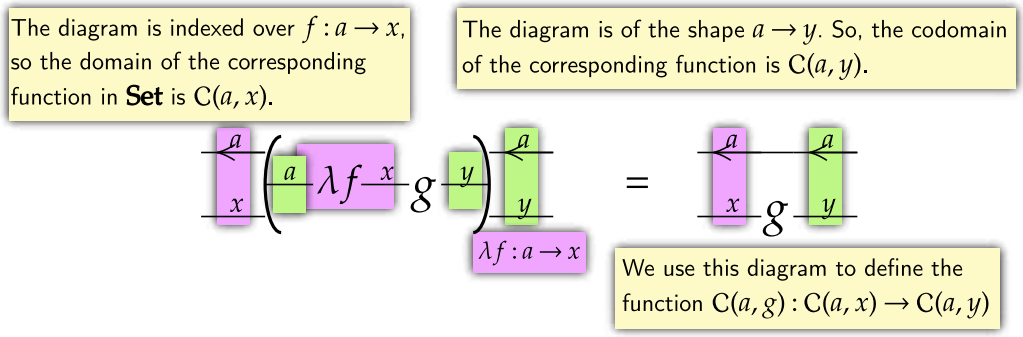
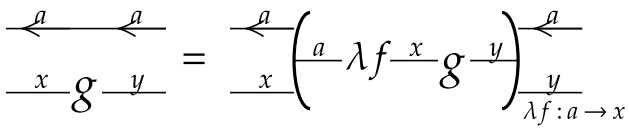
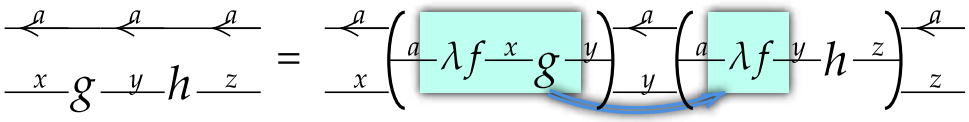


Figure 3.10: With $g : x \to y \in \mathcal{C}$, we have $\mathcal{C}(a,g) : \mathcal{C}(a,x) \to \mathcal{C}(b,y)$, a function between sets. The diagram on the right is a family expression $\mathcal{C}(a,x) \to \mathcal{C}(a,y)$. Therefore, it defines this function.

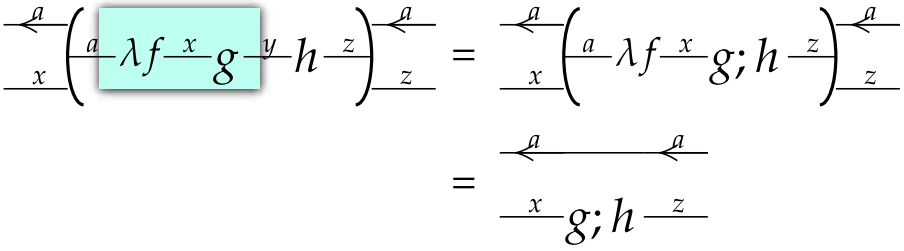
The question is whether this lifting is a functor. This would require $\mathcal{C}(a,f);\mathcal{C}(a,g) = \mathcal{C}(a,f;g)$. We can test this using the diagrams and the logical rules we have developed. To prove that $\mathcal{C}(a,\_)$ is a functor, we need to prove that it preserves composition. We expand the definition of $\mathcal{C}(a,g) : \mathcal{C}(a,x) \to \mathcal{C}(a,y)$.



We interrogate composition by expanding the lambda calculus expressions. Here, the second λf can be replaced by the $\lambda f; g$ from the first expansion.



The g and h terms can be internally composed. The λf term is still pending, meaning that the expression is a hom-functored expression. This proves that $\mathcal{C}(a, g); \mathcal{C}(a, h) = \mathcal{C}(a, g; h)$, meaning $\mathcal{C}(a, \_)$ is a hom-functor.



Consider pre-composing the definition of a hom-functor from Figure 3.10 with some $h : a \to x$ expressed as an element of $\mathcal{C}(a, x)$, $h : 1 \to \mathcal{C}(a, x)$. This is shown at the top of Figure 3.11. In the diagram on the right, $h : 1 \to \mathcal{C}(a, x)$ gets indexed to the morphism $h : a \to x$. The index has been chosen, so the diagram becomes a morphism from $1 \to \mathcal{C}(a, y)$. This diagram gives the *hom-rules* which we will often use.

3.2.4 Graphical Yoneda Lemma

The Yoneda lemma is a central result of category theory. Natural transformations are powerful but difficult to derive, and the Yoneda lemma offers a rare opportunity to enumerate and classify them. The Yoneda lemma makes a statement about the natural transformations between a hom-functor and any other functor to **Set**.

Therefore, proving and understanding it requires clear expressions of properties specific to hom-functors and general properties that hold for functors. Additionally, it requires understanding several correspondences, such as that between the identity morphism $\mathrm{Id}^a_a \in \mathcal{C}(a, a)$ and the element $\mathrm{Id}^1_{aa} \in \mathbf{Set}(1, \mathcal{C}(a, a))$.

To understand the Yoneda lemma, we need a range of tools. We need to clearly see functors and natural transformations, and how they interact with objects and morphisms. Correspondences need to be clear. Furthermore, we need to be able to define new functions in **Set** by referring to known expressions. Throughout this work, I have been presenting tools which allow for this understanding. Being able to derive and understand the Yoneda lemma clearly was an explicit goal of this work.

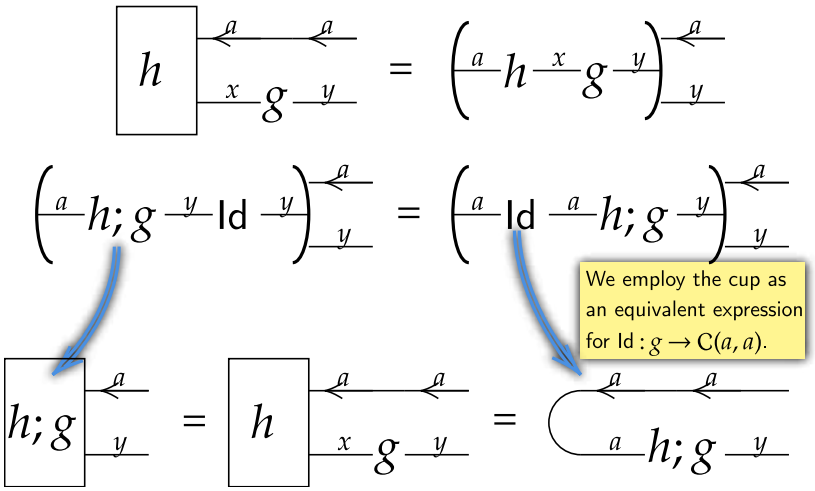


Figure 3.11: The rectangle around morphisms makes expressions clearer. It is purely aesthetic. We pre-compose Figure 3.10 with a specific value. Then, observe how the right-hand side is equivalent to various expressions. This means the corresponding expressions on the left must be equal. This derives the hom-rules, and they let us graphically intuit the behaviour of hom-functor wires.

Typical approaches involve symbolic or commutative diagram expressions. An example is from Awodey (2010), on page 195. The Yoneda lemma is proven over four and a half pages, has six commutative diagrams, and includes a nine-line symbolic equality expression block. Though such proofs are rigorous, symbols and commutative diagrams disrupt intuition about the proof and the result we ultimately derive.

A more diagrammatic approach is offered by Ochs (2020). That work develops a diagrammatic scheme. However, I feel vertical decomposition offers easier-to-understand expressions than networks of symbols and arrows. Furthermore, family expressions are often more clear and powerful when relating concepts than arrows between arrows.

The approach most similar to mine is from Nakahira (2023). My work builds on his. However, my principles of vertical decomposition and equivalent expression aid in understanding diagrams and building intuition. Furthermore, my family expressions allow new functions in **Set** to be clearly defined and related to others. The dotted boxes employed by Nakahira are less explicit than my family expression approach, while my symbolic correspondence is more liberal but aids intuition.

The Yoneda Lemma

Theorem 1 (The Yoneda Lemma)**.** *For a locally small category $\mathcal{C}$, a hom-functor $\mathcal{C}(a, \_) : \mathcal{C} \to$ **Set** and any other functor $\Phi : \mathcal{C} \to$ **Set**, there is a one-to-one correspondence between distinct natural transformations $\mathcal{C}(a, \_) \to \Phi$ and distinct elements of*

$\Phi(a)$, *the set that the object* a *is sent to by* Φ.

$$Nat(\mathcal{C}(a, \_), \Phi) \cong \Phi(a) = \mathbf{Set}(1, \Phi a)$$

Proof. I will prove this by constructing a map between elements $\varphi \in \Phi(a)$ and natural transformations $\eta : \mathcal{C}(a, \_) \to \Phi$ and vice-versa. Then, I will show these maps are inverses of each other, giving an isomorphism.

We begin by using an element $\varphi : 1 \to \Phi a$ to define a component for an arbitrary object x, $\varphi^*[x] : \mathcal{C}(a, x) \to \Phi x$. This component is a function in **Set**, so can be defined using a diagram indexing over all possible $f : 1 \to \mathcal{C}(a, x)$ values (Figure 3.12, middle). The simplest way that $\varphi : 1 \to \Phi a$ can interact with $f : a \to x$ is by pre-composition with $\Phi f : \Phi a \to \Phi x$ (Figure 3.12, left). We equate the definition for $\varphi^*[x]$ over all f with the simplest way that φ and f can interact.

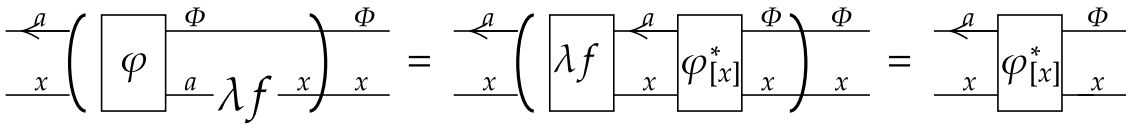
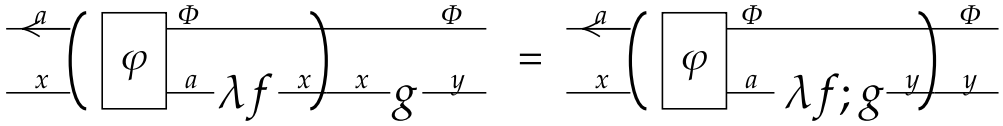
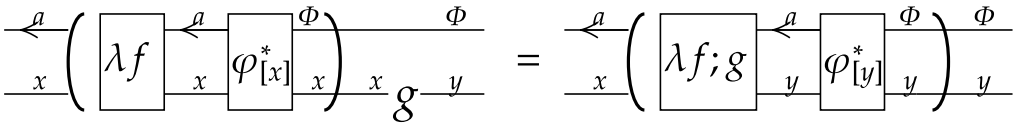


Figure 3.12: We wish to define a component $\varphi^*[x] : \mathcal{C}(a, x) \to \Phi x$. We do this by taking its generating expression in the middle, and equating it to the simplest interaction between $f : a \to x$ and $\varphi : 1 \to \Phi a$ on the left.

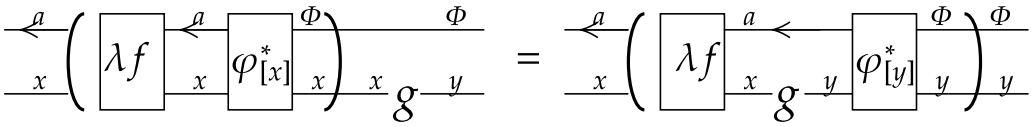
We see that λf gets moved from after to before the expression. This is exactly what a natural transformation should do. To take advantage of this, we post-compose the left side with a morphism $\Phi g : \Phi x \to \Phi y$.



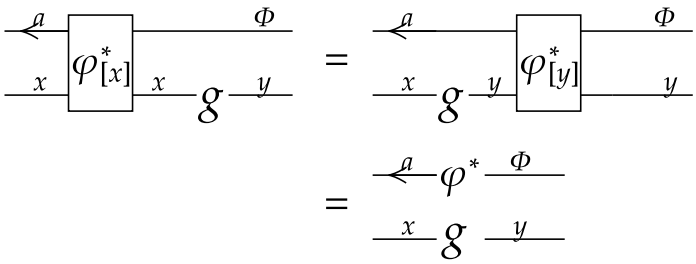
We recognize that the left-hand side corresponds to action with the $\varphi^*[x]$ component, and the right-hand side with the $\varphi^*[y]$ component.



Next, we use the hom-rules to expand the right-hand side.

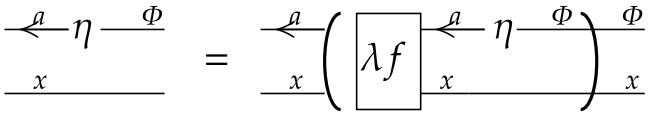


We recognize these as generating expressions in **Set**.

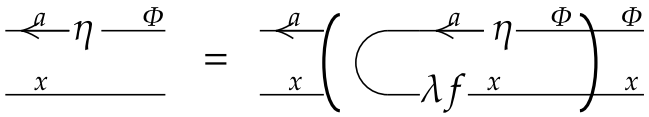


Therefore, we see how the freedom of definitions in **Set** and the hom-rule conspire to generate natural transformations $\mathcal{C}(a, \_) \to \Phi$ using the elements of Φa.

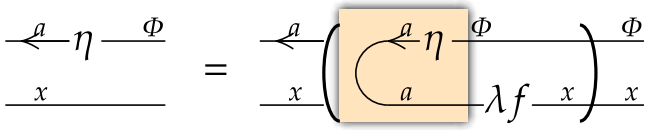
Now, we need a corresponding map from natural transformations to elements of Φa. We begin with a natural transformation η, and consider its definition over its set of inputs $\mathcal{C}(a, x)$. As we are in **Set**, we begin with the generating definition.



Next, we employ the hom-rules.



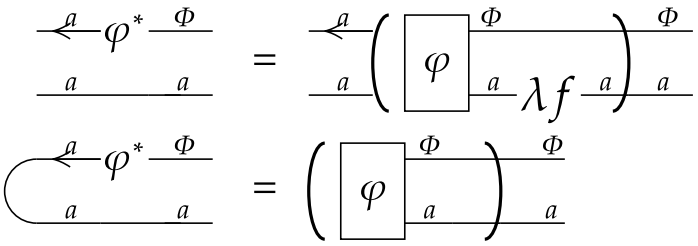
And follow with naturality.



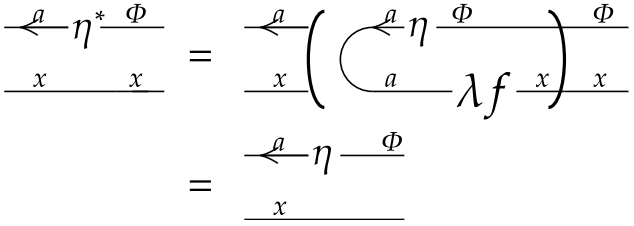
The highlighted section gives a morphism $1 \to \mathcal{C}(a, a)$, an element in $\Phi(a)$. Note how this element is independent of the arbitrary x object. Therefore, it is associated to this natural transformation for all components. So, we have just used the **Set** (which allowed the natural transformation component morphisms to be constructed), hom and naturality rules to derive an element of $\Phi(a)$ for every natural transformation $\eta : \mathcal{C}(a, \_) \to \Phi$.

Finally, we are required to show the correspondence is one-to-one. We take an element φ and use it to define a natural transformation. We then investigate the element corresponding to this natural transformation. Returning the original element makes the process invertible in this direction.

Now, we investigate the a natural transform η^*, defined from the element associated to the natural transform η. For this step, we follow the derivation of the associated element



in reverse. We find that we get back to the original natural transform. Therefore, the process is invertible in this direction as well.



Therefore, our map from elements to natural transformations is an isomorphism. This finishes the proof of the Yoneda lemma using our graphical calculus.

□

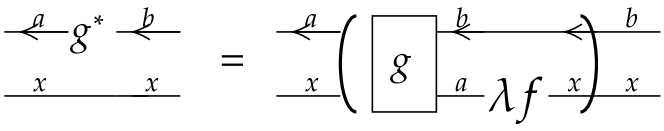
3.2.5 Hom-functor wires as Reverse Basewires

The Yoneda lemma is so called because of its utility in proving other theorems. A particular application is investigating the natural transformations between hom-functors. The lemma states that $\mathrm{Nat}(\mathcal{C}(a,\_),\Phi) \cong \Phi(a)$. So, if the target functor is $\Phi = \mathcal{C}(b,\_)$ then we have,

$$\mathrm{Nat}(\mathcal{C}(a,\_),\mathcal{C}(b,\_)) \cong \mathcal{C}(b,a)$$

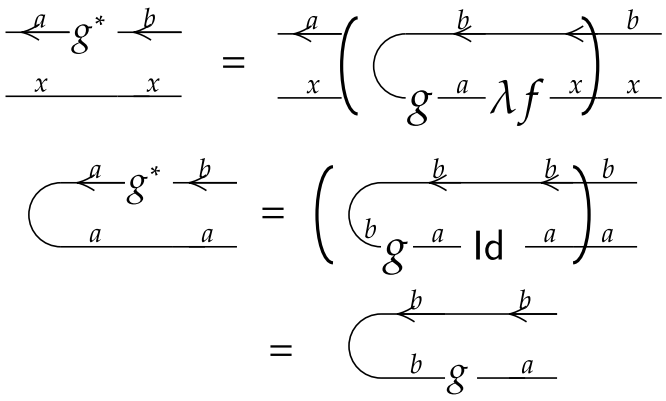
Therefore, the natural transformations between the a and b hom-functor wires are the morphisms $b \to a$. The functor wire, therefore, are the morphisms of the basewires, albeit reversed.

Starting with some $g \in \mathcal{C}(b,a)$, we will use the Yoneda lemma to find the corresponding $g^* : \mathcal{C}(a,\_) \to \mathcal{C}(b,\_)$ natural transformation. This gives us an expression that uses the corresponding $g : 1 \to \mathcal{C}(a,b)$ element of $\mathcal{C}(a,b)$.



The element g corresponds to a morphism, so we apply the hom-rule.

We can investigate this natural transformation by precomposing with the identity cup, revealing how it corresponds to the original morphism.



We see how the morphisms placed on hom-functor wires correspond to sliding morphisms through the identity cup, reversing their direction. This is a graphical isotopy that is intuitively shown through equivalently expressing all identity $1 \to \mathcal{C}(a, a)$ elements as cups and hom-functor wires with arrows. Noting the significance of this correspondence, we will diagram corresponding natural transformations without an asterisk.

3.3 The Product Extension

So far, I have introduced functor string diagrams, established key principles for developing new features, and shown how to reason with a family-expression-based graphical calculus. I will now extend diagrams with products. Products give a framework for the general description and construction of morphisms in a category. In addition, they allow diagrams to be placed on top of each other, representing parallel expressions.

Previously (see Section 1.1), we identified that previous diagrammatic methods (Selinger, 2009) struggle to consider both the details of axes (the tensor approach) and independent operations (the Cartesian approach). We have already carefully studied hom-functors, which express the details of axes. Now, by developing the tools to graphically consider products, functor string diagrams resolve this challenge, displaying both the details of axes and independent, parallel operations.

Furthermore, Cartesian products and especially projections serve as the basis of broadcasting. Broadcasting is critical to understanding deep learning, and in this section, I develop the tools to formally consider it in the next (see Section 3.4).

Combining products with functor string diagrams was done by both prior works on functor string diagrams. Marsden (2014) looked at products briefly, using colored subdiagrams in his preferred presentation. Nakahira (2023) looked at direct products. The first work had an awkward time with bifunctors, which the latter improved upon. Neither covered products in depth.

My approach builds on these works, and more clearly considers products in a unified framework. Family expressions can readily show how families of morphisms are con-

structed into a single morphism using products. Meanwhile, the principle of vertical section decomposition and equivalent expression means my diagrams have a disciplined approach to introducing new features, including products. This overcomes some of the awkwardness of previous approaches.

Contributions

In this section, I develop the tools needed to integrate products with functor string diagrams. Products are important because they are one of the few tools that can freely construct morphisms in a category. The previous functor string diagram works briefly looked at products. Here, I integrate Cartesian products with the family expression view, using a "pseudomorphism" presentation that is reminiscent of linear algebra notation. Then, I explore monoidal products and how traditional results relating copying, deletion, and Cartesian monoidal products can be shown using functor string diagrams.

3.3.1 Projections

Products combine things in some way. Cartesian products are flexible, allowing anything to be combined and subsequently deleted. In category theory, we can develop a general definition of Cartesian projections. This integrates well with our previous exploration of hom-functors and will allow us to define broadcasting clearly.

Definition 6 (Cartesian projections)**.** *For an object b, an I-family of projections from b to a family of objects $(Bi)_{i\in I}$ is a family of morphisms $(\pi[i]^b_{Bi})_{i\in I} \in Proj(b)$;*

- *Which gives a **complete description**, for any object a and two morphisms f^a_b and g^a_b, if for all $i \in I$, we have $f^a_b;\pi[i]^b_{Bi} = g^a_b;\pi[i]^b_{Bi}$, then $f^a_b = g^a_b$.*
- *Which **accepts free construction (the Cartesian property)**, for any family of morphisms $(f[i]^a_{Bi})_{i\in I}$, there exists a morphism f^a_b such that for all $i \in I$, we have $f^a_b;\pi^b_{Bi} = f[i]^a_{Bi}$.*

From this definition, we see that a morphism to b is completely described, and therefore defined, by its projections to $(Bi)_{i\in I}$. Furthermore, any family of morphisms to the projection objects $(Bi)_{i\in I}$ can be assembled into a morphism to b.

There is, therefore, a one-to-one correspondence between morphisms to b and morphisms to the individual degrees of freedom offered by the projections. We can diagram this correspondence by the diagram in Figure 3.13.

The right-hand expression in Figure 3.13 defines a morphism to b. We will, therefore, want to use it in diagrams as an equivalent expression of a morphism to b. However, we cannot unambiguously use these diagrams. The expression on the right could define a different morphism to b depending on the family of projections we use. Furthermore, the target object of the diagrams is not the same as the target object of the morphism we are defining with it.

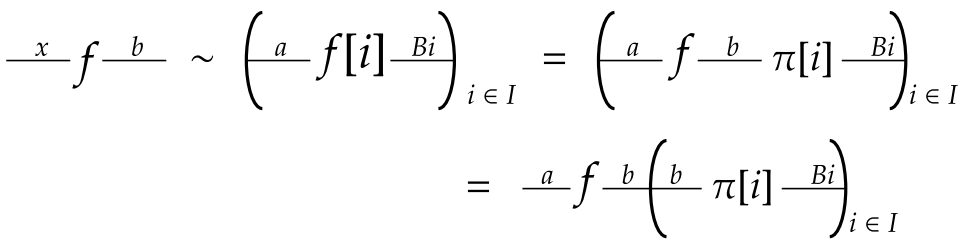


Figure 3.13: A morphism to an object b with a collection of (Cartesian) projections Bi can be expressed as a family of morphisms to Bi. I use "$\sim$" to show a correspondence, with the above expression stating there is a one-to-one correspondence between morphisms $f : a \to b$ and families of morphisms $(f[i]^a_{Bi})_{\,i \in I}$.

So, we introduce a pseudomorphism $\pi^*[i]^{Bi}_b$ that exists to clarify the set of projections we are defining f over, and to make the family of diagrams align with the overall diagrams. This is not a morphism. Instead, it is simply an equivalent expression that looks like one. We introduce and diagram this feature in Figure 3.14.

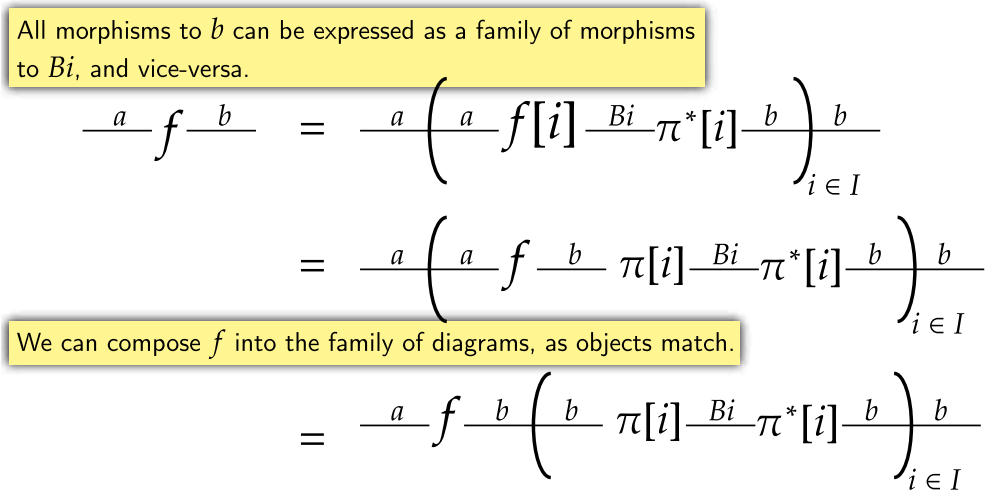


Figure 3.14: Morphisms to objects b with a family of projections $(Bi)_{i \in I}$ are in one-to-one correspondence with a family of morphisms to $(Bi)_{i \in I}$. We diagram this correspondence by a family expression and use pseudomorphisms to "contract" the family of morphisms to $(Bi)_{i \in I}$ towards a morphism to b. This family expression accepts precomposition.

Note how every morphism $f : a \to b$ from an object a to an object b which has an I-family of projections to $(Bi)_{i \in I}$ is in one-to-one correspondence with a family of morphisms $(f[i]^a_{Bi})_{i \in I}$. Therefore, the collection $\mathcal{C}(a, b)$ is in one-to-one correspondence with the collection $\Pi_{i \in I}\, \mathcal{C}(a, Bi)$. We can view the object b as being $\Pi_{i \in I} Bi$. Therefore, we can write this correspondence as,

$$\mathcal{C}(a, b) \cong \mathcal{C}(a, \Pi_{i \in I} Bi) \cong \Pi_{i \in I}\, \mathcal{C}(a, Bi) \tag{3.1}$$

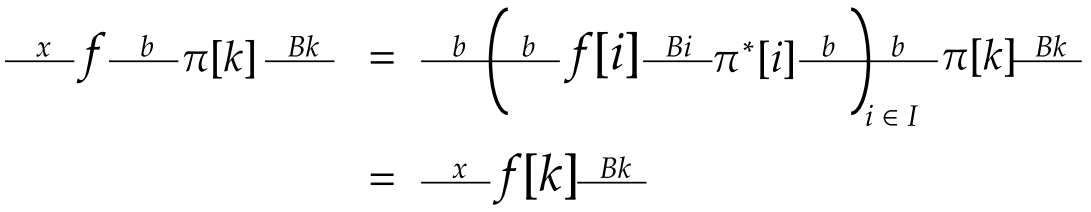


Figure 3.15: The projection $\pi[k] : b \to Bk$ extracts the k member of the family of morphisms which construct $f : x \to b$.

Furthermore, objects can also have families of *co*projections. For an object a, an I-family of coprojections $(p_a^{Ai})_{i \in I}$ provides a complete description that accepts free construction. So, if two morphisms from a are equal when pre-composed with each coprojection, the two morphisms must be equal. Additionally, any family of morphisms $(h[i]_y^{Ai})_{i \in I}$ can be assembled into a morphism h_y^a. Similarly to Equation 3.1, we can view coprojections as offering,

$$\mathcal{C}(a, b) \cong \mathcal{C}(\amalg_{i \in I} Ai, b) \cong \Pi_{i \in I}\ \mathcal{C}(Ai, b) \tag{3.2}$$

3.3.2 The Product of Categories

For any two categories $\mathcal{C}$ and $\mathcal{D}$, there exists a product category $\mathcal{C} \times \mathcal{D}$ (Awodey, 2010, p. 16). It has objects of the form (a, x), with a drawn from $\mathcal{C}$ and x from $\mathcal{D}$. Similarly, morphisms are of the form $\left(f_b^a,\ g_y^x\right) : (a,\ x) \to (b,\ y)$, with f_b^a from $\mathcal{C}$ and g_y^x from $\mathcal{D}$. Composition is done componentwise, so we have,

$$(f_b^a,\ g_y^x); (h_c^b,\ k_z^y) = (f_b^a; h_c^b,\ g_y^x; k_z^y).$$

We then define two projection functors, $\pi_{\mathcal{C}}^{\mathcal{C} \times \mathcal{D}}$ and $\pi_{\mathcal{D}}^{\mathcal{C} \times \mathcal{D}}$. They map objects (a, x) to a or x, respectively, and similarly extract the elements of morphism tuples, sending $\left(f_b^a,\ g_y^x\right)$ to f_b^a or g_y^x.

Every diagram is an expression in some category $\mathcal{C}$ ("expressions in a category" correspond to morphisms). So, we can take the product of any two diagrams. This is done by placing them above each other, and placing a double-dash line to show they are clearly separated. We reserve a single-dashed line for products *within* a category, that will be covered soon.

The principle of vertical section decomposition holds, every vertical section represents either objects or morphisms in the product category. Furthermore, componentwise composition lets us decompose diagrams into horizontal sections. With the projection functors, we can isolate horizontal sections from diagrams. Using both vertical and horizontal section decomposition is shown in Figure 3.16.

On the right of Figure 3.16, notice how we have an expression in $\mathcal{D}$ and place something above to get an overall expression in $\mathcal{C} \times \mathcal{D}$. The expression from $\mathcal{C}$ is a functor, moving us into a new category while preserving the underlying composition. Therefore, the

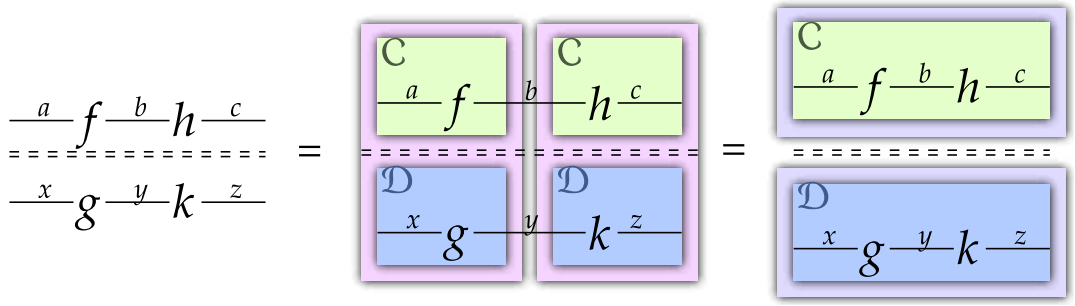


Figure 3.16: The product of two diagrams can be understood by decomposition into composing vertical sections with matching wires or as two horizontal sections joined together.

product of categories $\mathcal{C} \times \mathcal{D}$ can be seen as using expressions from $\mathcal{C}$ to define functors on $\mathcal{D}$. Objects in $\mathcal{C}$ behave like functors, and morphisms like natural transformations. We can explicitly see this by substituting identities into the above diagrams, as seen in Figure 3.17.

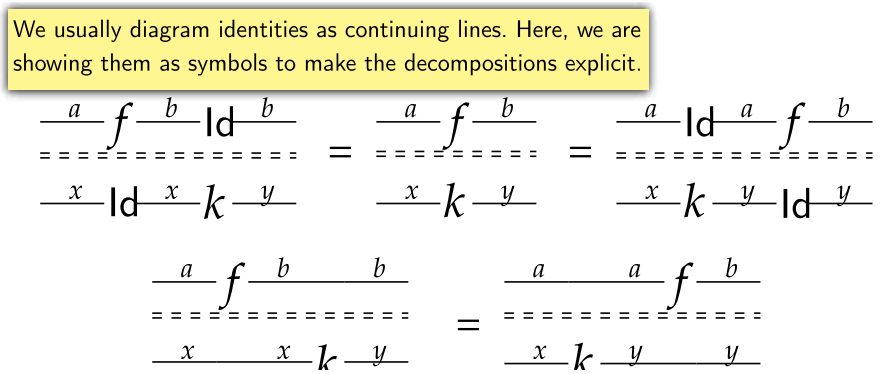


Figure 3.17: The morphisms involved in the product of categories behave like natural transformations.

3.3.3 Bifunctors

Functors preserve composition. So, functors on products of categories maintain the naturality of morphisms between the constituent categories. Functors on products of categories are called bifunctors. Bifunctors from a category to itself, of the form $\cdot : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, are of particular interest to us. For any two expressions f_b^a and g_y^x in $\mathcal{C}$, we have an expression $\left(f_b^a, g_y^x\right)$ in $\mathcal{C} \times \mathcal{C}$. A bifunctor maps this back to the category, giving us a vertically composed expression $f_b^a \cdot g_y^x : a \cdot x \to b \cdot y$ which is also in $\mathcal{C}$.

Bifunctors, where this vertical composition behaves like morphism composition, are particularly powerful. This requires vertical composition to be associative, $\left(f_b^a \cdot g_y^x\right) \cdot h_q^p \cong f_b^a \cdot \left(g_y^x \cdot h_q^p\right)$. Additionally, there has to be an identity, given by the identity of a unit

object u, $f_b^a \cdot \mathrm{Id}_u^u \cong f_b^a \cong \mathrm{Id}_u^u \cdot f_b^a$. These statements are isomorphisms, and they are provided by isomorphisms between objects such as $a \cdot u$ and a. If a bifunctor's vertical composition behaves like morphism composition, it is called a *monoidal product* (Awodey, 2010; Selinger, 2009).

Both Cartesian and tensor products are examples of monoidal products. Other constructs, such as joint distributions, are also monoidal products (Fritz et al., 2023). A category equipped with a monoidal product is called a *monoidal category.* By developing tools to deal with monoidal products, we can use functor string diagrams to consider a variety of constructs.

A monoidal category $(\mathcal{C}, \otimes, u)$ typically only has one monoidal product $\otimes$ with a unit object u. They inherit a rich graphical scheme (Selinger, 2009). However, we have already noted its limitations. Traditional monoidal graphical schemes have difficulty representing both functors and monoidal products, which precludes them from expressing both the details of axes (hom-functors) and the independent combination of morphisms (products).

We could diagram monoidal products in a direct way, with a functor wire $\otimes$ sending an expression $\mathcal{C} \times \mathcal{C}$ with a double dashed line to an expression in $\mathcal{C}$. This requires us to indicate that the $\otimes$ functor wire is dominant by thickening it, showing that it acts on the entire expression rather than just the top segment.

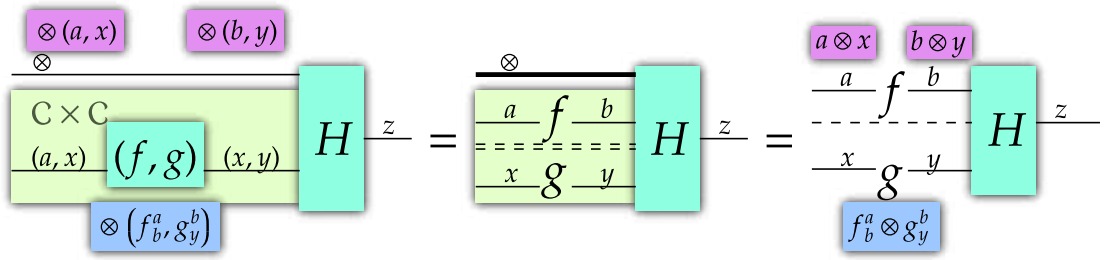


Figure 3.18: The monoidal product $\otimes$ is a functor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$. Here, we diagram an expression involving it in three different ways. In pink, we note the object, and in light blue the morphism. The direct $(\_, \_)$ expression on the left is an equivalent expression to a vertically stacked expression in the middle. In the middle, we thicken the $\otimes$ functor wire to show that it applies over the entire block, not just the first segment. Finally, we use a single dashed line on the right as an equivalent expression for monoidal products.

We develop an equivalent expression, diagramming monoidal products with a single dashed line, which we show in Figure 3.18. As monoidal products are associative, we do not need to indicate the order in which products are placed. Furthermore, we can always introduce or remove a segment that only consists of the unit object. Typically, we do not diagram unit object wires.

Lastly, recall that placing a functor wire F over an object a or a morphism f_b^a are equivalent expressions for Fa or Ff_{Fb}^{Fa}. We could diagram $Ff \otimes Gg$ directly, as on the left of Figure 3.19, but using equivalent expressions for functors, we get the clearer expression on the right.

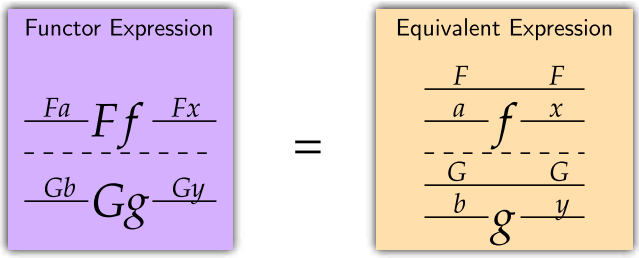


Figure 3.19: Using equivalent expressions, we can show functor expressions in a clearer manner.

3.3.4 Cartesian Monoidal Product

A Cartesian monoidal product is the most direct inclusion of $\mathcal{C} \times \mathcal{C}$ into $\mathcal{C}$. A Cartesian monoidal product combines objects a and b into a product object $a \times b$, and morphisms f_x^a and g_y^b into $f_x^a \times g_y^b : a \times b \to x \times y$. To exist, a category has to have an appropriate unit object and products between any two objects. For the products to be Cartesian, they must have projections which give complete descriptions and accept free construction.

The product of morphisms combine any morphisms in the natural way. Starting with f_x^a and g_y^b, we generate $\pi_a^{a\times b}; f_x^a$ and $\pi_b^{a\times b}; g_y^b$. These are morphisms from $a \times b$ to x and y respectively, so we can freely construct them into a morphism $f_x^a \times g_y^b$ as $x \times y$ is a product of x and y. They are diagrammed by placing morphisms above each other, separated by a dashed line. This is shown in Figure 3.20.

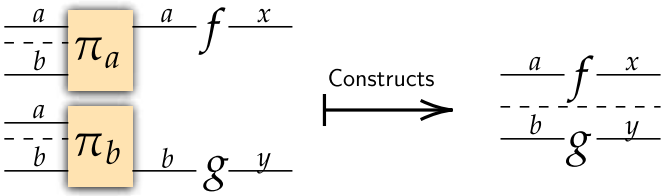


Figure 3.20: In a Cartesian monoidal category, we can take the product of any two morphisms.

Copying and Deletion

It is worthwhile to point out an alternative formulation of Cartesian monoidal products. Cartesian monoidal products are intimately tied to *copying* and *deletion*. This is a valuable perspective, as it tells us why classical computing, which allows copying and deletion, is necessarily Cartesian monoidal. Furthermore, it lets us know that perspectives that do not allow copying and deletion – such as quantum or probabilistic systems (Baez and Stay, 2010; Fritz et al., 2023) – must be thought of in a distinct way.

This is a standard result (Heunen and Vicary, 2012, p. 79). However, our approach is somewhat different. We do not assume the bifunctor is symmetric. Defining symmetry

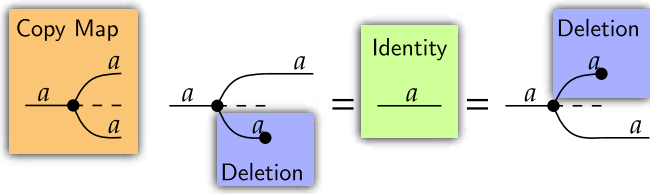
is somewhat problematic (Selinger, 2009). It is better to first prove the product is Cartesian so we can freely construct symmetry however we want. Alternatively, we could assert that what remains after deletion gives a product. This is not particularly useful. Products giving products is hardly insightful.

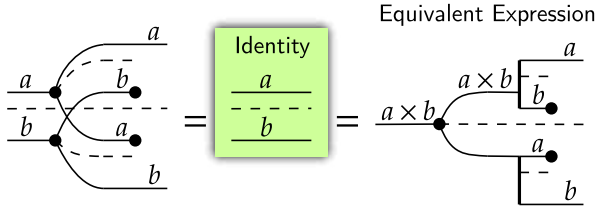
For some category $\mathcal{C}$ with a monoidal product $\times$ and unit object u, we begin with asserting that the unit object u is a terminal object, meaning there is a unique morphism del_u^x from every object to it. Therefore, any composition to u, such as $f_x^a ; \text{del}_u^x$, must equal del_u^a. Hence, it deletes all incoming information. We diagram deletion as a wire terminating in a black dot. Similar to generating objects, we do not diagram the unit u. If the unit object is terminal, the monoidal product is *semi-Cartesian.*

We define the relationship between deletion and a copy map using the following definitions. Then, we will prove this is sufficient for $a \times b$ to be the Cartesian product of a and b.

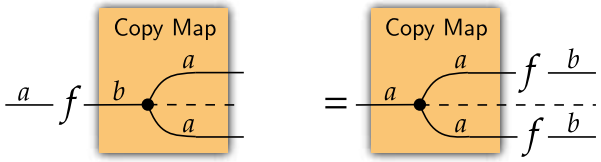
Definition 7 (Cartesian monoidal category)**.** *A Cartesian monoidal category $\mathcal{C}$ has a monoidal bifunctor $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ with a unit object u such that,*

- *(**Monoidal product**) It obeys monoidal regularity conditions. Vertical composition is associative and the unit object u can be freely introduced. The unit object wires are not diagrammed.*
- *(**Semi-Cartesian**) The unit object is terminal, meaning there is only one morphism, del_u^a, from every object to it. It is diagrammed by a wire terminating in a black dot.*
- *(**Copying comonoid**) For every object a, there is a copying map $\Delta : a \to a \times a$. It is diagrammed by a black dot, followed by diverging wires. Together with the deletion maps, it obeys the following properties.*





- *(**Duplicating property**) For any morphism f_b^a, the copy map duplicates it in the following way,*



Using just these conditions, we can prove that the objects $a \times b$ are products of a and b. The proof can be found in the Appendix A.1.

Proposition 1 (The Residues of Deletion gives Projections)**.** *A Cartesian monoidal category, as defined above, has Cartesian projections for any object $a \times b$ given by $\pi[a]_a^{a \times b} = Id_a^a \times del_u^b$ and $\pi[b]_b^{a \times b} = del_u^a \times Id_b^b$.*

3.4 Broadcasting

As we saw in *Applications of Neural Circuit Diagrams*, broadcasting is crucial to understanding deep learning models. Lifting a morphism f over an object a defines a new morphism $\mathcal{B}_a f$ that is a-fold parallelized. Hom-functors are an example of a broadcast, lifting $f : x \to y$ to $\mathcal{C}(a, f) : \mathcal{C}(a, x) \to \mathcal{C}(y, x)$. However, inner broadcasting is not perfectly analogous to hom-functors without a fair deal of additional definitions.

Besides, the deep learning community needs a general definition of broadcasting. Broadcasting is critical but rarely precise. The PyTorch broadcasting syntax is very finicky. It inherits from NumPy, another extremely common package, so is hardly exceptional in its treatment. Other works aiming to improve how we communicate deep learning models, such as Chiang et al. (2023), had to introduce bespoke definitions.

Instead of relying completely on hom-functors, we will contribute a general definition of broadcasting. Our definition allows morphisms to be lifted over complex data types while having clear behavior. Furthermore, it integrates well with diagrams and code implementations.

Contributions

In this section, I contribute a general definition of broadcasting, which is needed in the deep learning community. Additionally, I contribute the specifications for an *index category*, a minimal category to handle broadcasting. I prove that a few basic requirements imply the existence of indexes and hom-functors that stay within the category. Furthermore, I prove that a broadcasted morphism is equal for any choice of family of indexes. Then, I set up the framework for *monoidal index categories*, which incorporate index categories with monoidal products. This section defines and formalizes broadcast-

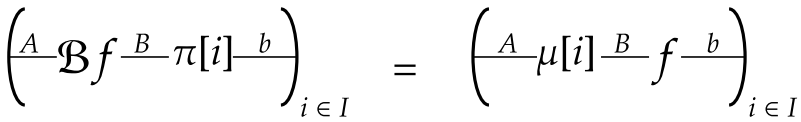
ing, providing the framework for broadcasting to be used in neural circuit diagrams and other systems.

3.4.1 A General Definition of Broadcasting

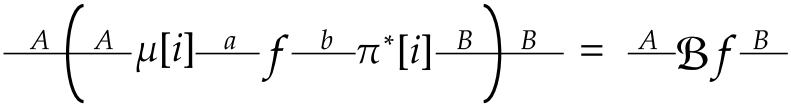
Definition 8 (General Broadcasting)**.** *Given a morphism $f : a \to b$, an I-fold family of morphisms $(\mu[i]^A_a)_{i\in I}$ from A to a, an I-fold family of projections $(\pi[i]^B_b)_{i\in I}$ from B to b, we define the broadcasted morphism $\mathcal{B}f : A \to B$ by,*

$$\mathcal{B}f^A_B ; \pi[i]^B_b = \mu[i]^A_a ; f^a_b$$

This definition can be given by the diagram family expression,



We can also define $\mathcal{B}f : A \to B$ as a contraction over the family of projections $(\pi[i]^B_b)_{i\in I}$,



In our general definition, broadcasting a morphism $f : a \to b$ requires identifying a family of morphisms to a and a family of projections to b of equal size. This yields a lifted morphism $\mathcal{B}f : A \to B$. This lifted morphism is defined for each of I projections from B to b and is, therefore, a uniquely defined morphism.

3.4.2 Broadcasting and Indexes

We see that broadcasting is dependent on the families of projections we choose. We want these families of projections to align with how we broadcast using diagrams. So, we will begin by diagramming how we "want" broadcasting to work, and will then work to formalize it by choosing appropriate families of morphisms and projections. The desired shapes of broadcasted morphisms are shown in Figure 3.21.

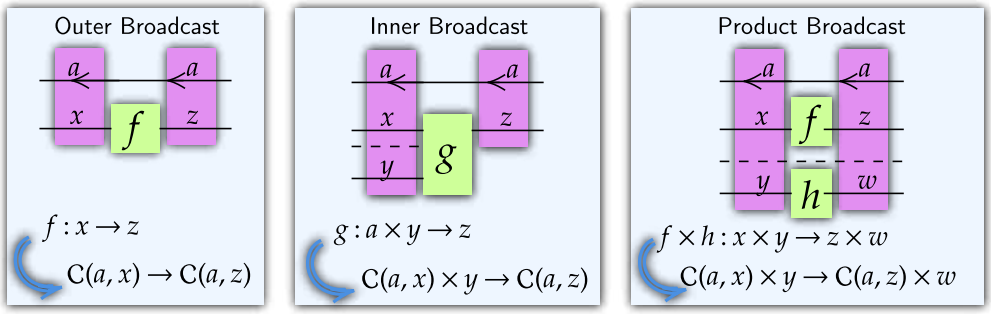
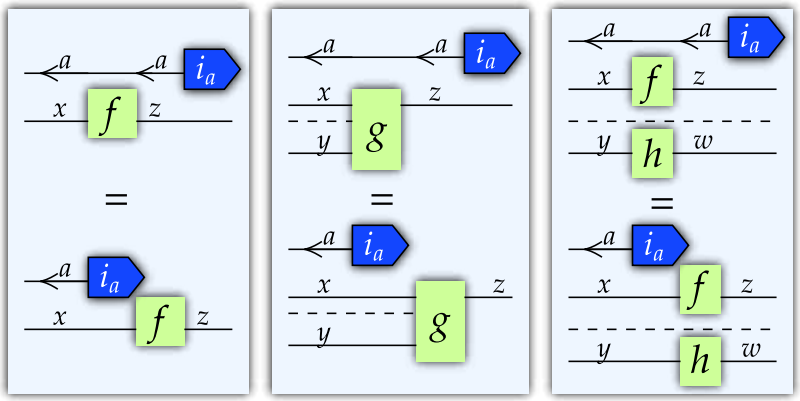


Figure 3.21: Different cases of broadcasting. We show how we want broadcasting to influence the shape of morphisms in each case.

Figure 3.21 shows, on the left, how the simplest form of broadcasting aligns with hom-functors. However, inner broadcasting does not align with functors. Furthermore, on the right, we see that even simple broadcasts do not align with hom-functors. The diagram on the right shows a case where we broadcast over the first segment and leave the other alone. This is not a direct application of a hom-functor, which would yield a morphism $\mathcal{C}(a, x \times y) \to \mathcal{C}(a, z \times w)$. Rather, we must separate f, generate $\mathcal{C}(a, f)$, and recombine the morphisms.

Our general definition of broadcasting can cover all these cases. To lift a morphism, we must identify an appropriate family of morphisms for the source and target objects. The appropriate family is indicated by how we diagram. We show the choice of appropriate projections in Figure 3.22, which aligns with how we defined broadcasting in Figure 2.8.

Figure 3.22: The appropriate projections to clearly define broadcasting in each case.

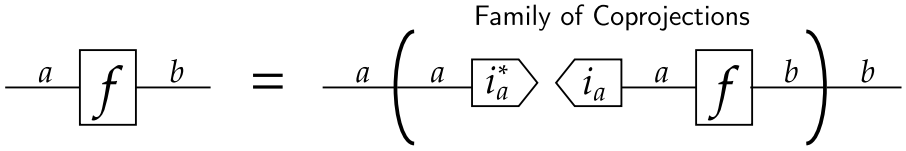


The existence of these index morphisms, however, requires further infrastructure. We want to relate the indexes on $\mathcal{C}(a, x)$ to the object a. Furthermore, we want to know the conditions under which broadcasting is possible. Doing this, we can produce a more general and powerful framework for analyzing deep learning models. It will put broadcasting on a solid mathematical foundation that can be reliably extended to new circumstances.

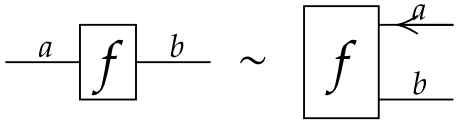
3.4.3 Index Categories

Definition 9 (Index Category)**.** *An index category $\mathcal{C}$ has,*

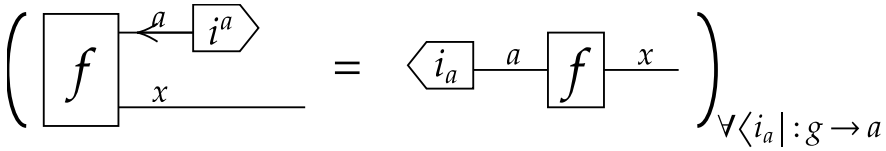
- ***(Coprojection rule)*** *A generating object g, such that for every object a, there is a family of coprojections from g.*
  - *We diagram these coprojections as bras, $\langle \_\_|$, or pointed pentagons. Iterating over $i \in I_a$, an object a has a family of coprojections given by $\langle i_a| : g \to a$. The dual morphism used for $\mathcal{C}(\langle i_a|, \_) : \mathcal{C}(a, \_) \to \mathcal{C}(g, \_)$ is diagrammed by a ket, $|i^a\rangle\_$. The associated pseudomorphisms are diagrammed by kets with an internal asterisk, $|i_a^*\rangle$;*
  - *Every morphism $\varphi : g \to a$ is in some family of coprojections for a;*
  - *Every morphism, therefore, has a coprojection expansion;*



- ***(Representation rule)*** *The hom-collections $\mathcal{C}(a, b)$ are objects in $\mathcal{C}$. We have isomorphisms $\mathcal{C}(g, a) \cong a$.*
  - *Therefore, there is a one-to-one correspondence between $\mathcal{C}(a, b)$ and $\mathcal{C}(g, \mathcal{C}(a, b))$. So, every morphism $f : a \to b$ has a corresponding morphism $f : g \to \mathcal{C}(a, b)$. We can show this by (using $\sim$ for corresponding expressions);*



- ***(Index interaction rule)*** *For any morphism $\langle i_a| : g \to a$, we have indexes $|i^a\rangle x_x^x = \mathcal{C}(\langle i_a|, x) : \mathcal{C}(a, x) \to \mathcal{C}(g, x)$, which interact with morphism representations in the following way;*



In **Set**, coprojections are given by the elements of a set. Any map between elements from the domain to codomain set can be constructed. In **Vect**, there is an underlying field (a space with addition, multiplication, and division) such as $\mathbb{R}$, and we can construct all vectors as maps $\mathbb{R} \to \mathbb{R}^n$, and linear maps $\lambda : \mathbb{R}^n \to \mathbb{R}^m$ as matrices $\mathbb{R}^{n \times m}$. Later on, if

we want to work with a subcategory of **Set**, we know which properties must remain for broadcasting to be well-defined.

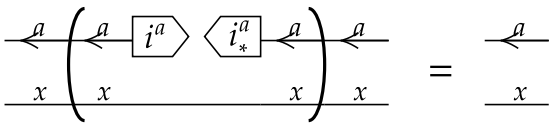
Our index category is intentionally minimal. We only define hom-objects and assert there is a correspondence between expressions $\mathcal{C}(a,b)$ and $\mathcal{C}(g,\mathcal{C}(a,b))$. The lifted coprojections are defined as interacting with these correspondences in a specific way according to the index interaction rule. These definitions are clearly satisfied in **Set**. For a space where all operations are linear, a coprojection family is a vector basis. These definitions are sufficient for broadcasting to exist in a category.

We require the following proposition related to index categories to construct the indexes for Figure 3.22. This proof can be found in Appendix A.2.

Proposition 2 (Indexes as Projections)**.** *In an index category $\mathcal{C}$, where:*

- *An object a has an I-family of coprojections $(\langle i_a|)_{i\in I}$,*

Then indexes $(|i^a\rangle x^x_x)_{i\in I}$ forms a family of projections for $\mathcal{C}(a,x)$. We can diagram this as:



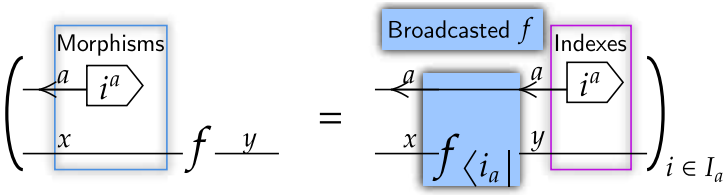
3.4.4 Broadcasting and the Hom-Functor

We defined index categories with only three principal properties: coprojections, representation, and index interaction. From just these, I derived that indexes serve as projections using the graphical calculus I previously introduced. This definition is minimal. Our index category does not have explicitly defined hom-functors, nevermind the tools for a generic lambda calculus. However, I will soon prove an internal hom-functor necessarily exists and that we have all the tools for inner and outer broadcasting.

There are various advantages to this minimal description. It means we can easily check whether a new construct is an index category without needing to resort to **Set**. The rules can be implemented in an imperative, sequential manner, which is easier to conceptualize than functional expressions that require a lambda calculus.

First, we need to define outer broadcasting. Broadcasting a morphism $f : x \to y$ requires an I-family of morphisms $\left(\mu[i]^X_x\right)_{i\in I}$ and projections $\left(\pi[i]^Y_y\right)_{i\in I}$ to define $\mathcal{B}_I f^X_Y$ (see Section 3.4.1). For outer broadcasting, we choose an object a and an I_a-family of coprojections $(\langle i_a|)_{i\in I_a}$ to lift f as follows;

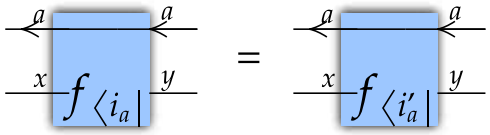
Definition 10 (Outer Broadcasting)**.** *In an index category $\mathcal{C}$, with an I_a-family of coprojections $(\langle i_a|)_{i\in I_a}$, we have an I_a-family of morphisms $(|i^a\rangle x^x_x)_{i\in I_a}$ to x, and an I_a-family of projections $\left(|i^a\rangle y^y_y\right)_{i\in I_a}$ to y. We define an a-fold outer broadcasting of a morphism $f : x \to y$ using $(\langle i_a|)_{i\in I_a}$ by,*



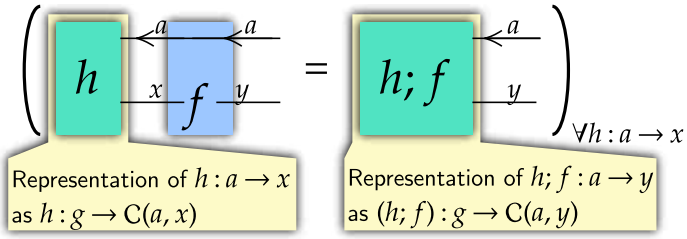
I indicate that f was broadcasted over the $\langle i_a|$-family of coprojections. The right-hand side are projections, meaning the highlighted morphism $f_{\langle i_a|} : \mathcal{C}(a, x) \to \mathcal{C}(a, y)$ is defined for all its degrees of freedom.

However, we come to an important realisation, which is that the choice of indexes we use to define broadcasting does not affect the morphism. This can be found in Appendix A.3.

Proposition 3 (Outer Broadcasting as the Internal Hom-Functor)**.** *An a-fold outer broadcast of a morphism $f : x \to y$ is equal for all choice of families of coprojections. For any two families of coprojections $(\langle i_a|)_{i\in I_a}$ and $\left(\langle i'_a|\right)_{i\in I_a}$, we have,*



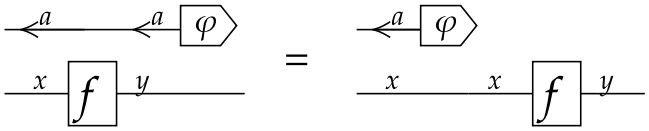
This means we can diagram an a-fold outer broadcast of a morphism $f : x \to y$ without reference to a family of coprojections. Furthermore, this expression behaves like the hom-functored $\mathcal{C}(a, f) : \mathcal{C}(a, x) \to \mathcal{C}(a, y)$, but internal to $\mathcal{C}$ without exiting to **Set**.



This means we can diagram an a-fold outer broadcast of a morphism $f : x \to y$ without reference to a family of coprojections.

This means that any morphism $g \to a$ which is a coprojection in any family also satisfies the broadcast rule. In the definition of index categories, we asserted that each morphism

$\varphi : g \to a$ is a morphism in some family of coprojections to a. Therefore, the broadcast rule generally holds.

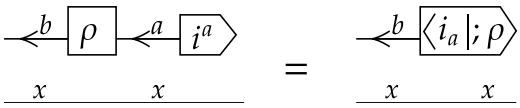


The above proposition implies any morphism $g \to a$, which is a coprojection in any family, also satisfies the broadcast rule. Previously, we asserted that each morphism $\langle\varphi| : g \to a$ is a morphism in some family. Therefore, the broadcast rule generally holds.

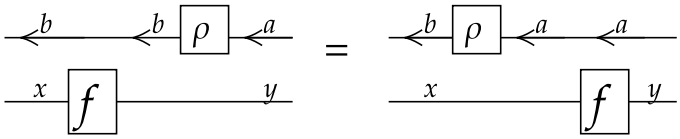
Note that in **Vect** the morphism $\langle 0| : \mathbb{R}^1 \to \mathbb{R}^n$ is *not* in a family of coprojections, meaning it is not a "true" index category. However, the below statement still holds for $\langle 0|$, as multiplying linear operations by zero before or after is the same.

This allows any morphism to be considered naturally on the hom-functor line, even though we have not defined hom-functors. This requires us to provide a quick definition of the natural correspondence of morphisms. I prove these are, in fact, natural transformations, and are equal for any choice of family of indexes in Appendix A.4.

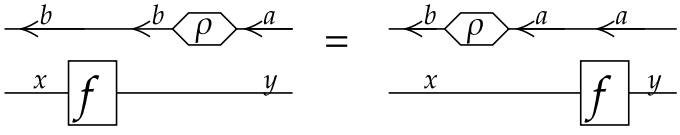
Definition 11 (Corresponding Natural Transformation)**.** *For every morphism $\rho : a \to b$ and object x, there is a "corresponding natural transformation" $\mathcal{C}(\rho, x) : \mathcal{C}(b, x) \to \mathcal{C}(a, x)$ defined over the indexes in the following way;*



Proposition 4 (Corresponding Natural Transformations)**.** *Corresponding natural transformations $\mathcal{C}(\rho, x) : \mathcal{C}(b, x) \to \mathcal{C}(a, x)$ are independent of the choice of family of indexes used to define them, and are indeed natural transformations. This implies that;*



We can equivalently express these using hexagons, to distinguish their naturality.



Therefore, we see that only using our three key index category properties we were able to derive the hom-functor properties. This has many advantages. The conditions I laid

out are minimal, meaning we can see how they extend to new domains. They hold for both **Set**, the category of sets and any function between them, and **Vect**, the category of exclusively linear operations (with very slight modifications).

Furthermore, unlike in *Reasoning with Diagrams*, we do not have to move into **Set**, nor did we have to assume the existence of hom-functors and an internal lambda calculus. Rather, hom-functors are derived from these minimal conditions. All operations can be interrogated by a combination of coprojections and indexes, meaning expressions in index categories can be readily decomposed.

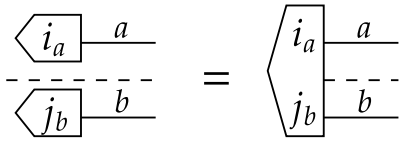
When developing and studying machine learning models, broadcasting is often critical. By contributing a general definition of broadcasting and laying out the minimal conditions for a broadcast category, I lay the foundation for more sophisticated models of machine learning systems to be developed, confident that a tool as critical as broadcasting is well-defined.

3.4.5 Broadcasting and Products

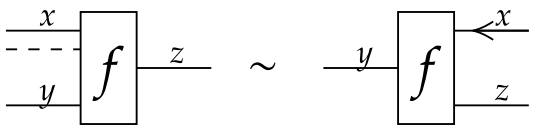
Previously, we approached products in a general manner. Monoidal products offer the framework to simultaneously consider Cartesian products, tensor products, joint probabilities, among other constructs. For inner broadcasting, we will define broadcasting with respect to any monoidal product. Being monoidal is necessary for products to be graphically unambiguous, as they let us ignore associativity. Here, we require the unit object to be the generating object.

Definition 12 (Monoidal Index Category)**.** *A monoidal index category $\mathcal{C}$ is an index category in addition to having,*

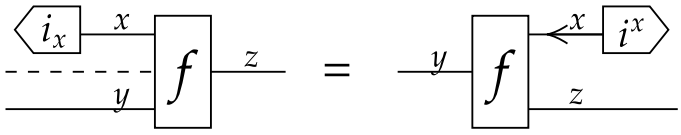
- ***(Monoidal product)*** *A monoidal product $\otimes$ with the generating object g as a unit. For a family of coprojections $(\langle i_a|)_{i\in I}$ for an object a and a family of coprojections $(\langle j_b|)_{j\in J}$ for an object b, we have a family of coprojections $(\langle i_a \otimes j_b|)_{i,j\in I,J}$ for $a \otimes b$;*



- ***(Product representation rule)*** *There is a correspondence between expressions $\mathcal{C}(x \otimes y, z) \cong \mathcal{C}(y, \mathcal{C}(x, z))$. We diagram this as,*

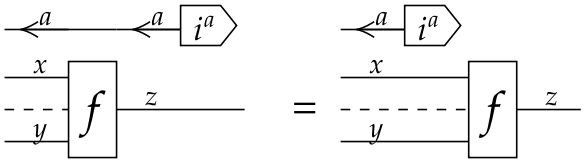


- ***(Product index interaction rule)*** *Coprojections* $\langle i_x|$ *generate indexes* $|i^x\rangle z^z_z$ *that interact with morphism representations in the following way,*



Similarly to the above, we define inner broadcasting in the following manner. By interrogating over the coprojections, we can prove that this expression is also independent of the choice of family of indexes. This is attached in Appendix A.5.

Definition 13 (Inner Broadcasting)**.** *In a monoidal index category, we lift a morphism* $f : x \otimes y \to z$ *to* $\mathcal{B}_a f : \mathcal{C}(a, x) \otimes y \to \mathcal{C}(a, z)$ *using a family of coprojections* $(\langle i_a|)_{i \in I_a}$ *which derive indexes that broadcast* f *according to;*



Proposition 5 (Inner Broadcasting)**.** *Inner broadcasted morphisms over* a *are equal for all choices of families of coprojections.*

This broadcasting is well-defined; we have an I-family of morphisms on $\mathcal{C}(a, x) \otimes y$ and an I-family of projections on $\mathcal{C}(a, z)$. The morphisms on $\mathcal{C}(a, x) \otimes y$ are not projections, meaning that our general definition of broadcasting is required. The general definition of inner broadcasting I present can be constructed in any monoidal index category, offering a flexible framework that can be extended to new situations.

3.5 From Theory to Application: A Robust Basis for Neural Circuit Diagrams

In *Applications of Neural Circuit Diagrams*, we looked at neural circuit diagrams as an applied framework to overcome a lingering challenge in deep learning research. Then, we investigated functor string diagrams, developing a graphical calculus based on family expressions to extend them to new situations. This allowed us to derive properties related to broadcasting and to specify an index category.

We now focus on using functor string diagrams to provide a robust basis for neural circuit diagrams. Neural circuit diagrams are an equivalent expression for a particular specification of a monoidal index category. By establishing this framework, I contribute a significant improvement over current category theory-based approaches to modeling deep neural networks, which can only represent the most basic models with the category

NNet (Fong et al., 2019), or rudimentary Cartesian graphical models (Cruttwell et al., 2021; Shiebler et al., 2021). Recall that Piedeleu and Zanasi (2023) noted that current category theory-based models for deep learning networks are mere "starting points".

This robust foundation opens up future work that can integrate neural circuit diagrams into the existing research on deep learning and category theory, including automatic differentiation (Fong et al., 2019; Cruttwell et al., 2021) and analyzing the mechanics of information (Perrone, 2022). Furthermore, this foundation motivates the application of category theory tools to models as they appear in practice. The approach I take is carefully specified. Extending it to new situations will involve refining these specifications or building additional tools on top of them. The significance of this section, therefore, is in providing the foundation for a rigorous analysis of deep learning architectures.

Contributions

In this section, I take the theoretical tools I have developed and use them to contribute a rigorous foundation for neural circuit diagrams and, hence, deep learning models more generally. I show how neural circuit diagrams are an equivalent expression for a subcategory of a Cartesian index category where all objects are sets of functions to $\mathbb{R}$. This mathematical model for deep learning architectures contributes a basis for future analysis that, unlike current models, represents models exactly as they appear in practice (Fong et al., 2019; Cruttwell et al., 2021; Shiebler et al., 2021).

Then, I explore how natural transformations can be represented in neural circuit diagrams and identify how they give us a unique perspective on algorithms. I use these tools to reason about multi-head Attention. I show how the idea of splitting and joining parallel heads can be conceptually presented with a diagram and how this diagram can be used to derive an implementation. Overall, this section reconciles neural circuit diagrams with robust theory and uses this theory to identify how categorical constructs, such as natural transformations, can be used to understand and implement algorithms.

3.5.1 Set as a Cartesian Index Category

Definition 14 (Cartesian Index Category)**.** *A Cartesian index category $\mathcal{C}$ is all of;*

- ***(Monoidal)*** *Has a monoidal product $\times$ with unit object u such that it is,*
- ***(Cartesian monoidal)*** *It is semi-Cartesian, with a terminal unit object that every object has only one morphism del_u^a towards, has a copying comonoid for every object $\Delta : x \to x \times x$ that interacts with deletion in the standard way, and copying obeys the duplicating property.*
- ***(An index category)*** *The unit object u is a generating objects from which every object x has a family of coprojections $(\langle i_x|)_{i \in I_x}$, families of coprojections $(\langle i_x|)_{i \in I_x}$ for x and $(\langle j_y|)_{j \in J_y}$ for y gives a family of coprojections $(\langle i_x| \otimes \langle j_y|)_{j \in J_y}$ for $x \otimes y$,*

we have access to $\mathcal{C}(a, x)$ objects which obey the product representation rule, and the index interaction rule holds.

A Cartesian index category encompasses **Set**. In **Set**, a monoidal Cartesian product $\times$ exists. Furthermore, the coprojections for a function $f : X \to Z$ are the elements of X, $x : 1 \to X$. The elements of $X \times Y$ are $(x, y) : 1 \to X \times Y$, which are constituted of an element from $x : 1 \to X$ and $y : 1 \to Y$. Representations exist as sets of functions exist within **Set**. Therefore, we see that **Set** is a Cartesian index category.

We could represent **Set** using functor string diagrams. In Figure 3.23, we are able to communicate complex dynamics of various data types broadcast and interacting in different ways. We see that functor string diagrams are a natural way to extend neural circuit diagrams to arbitrary data types.

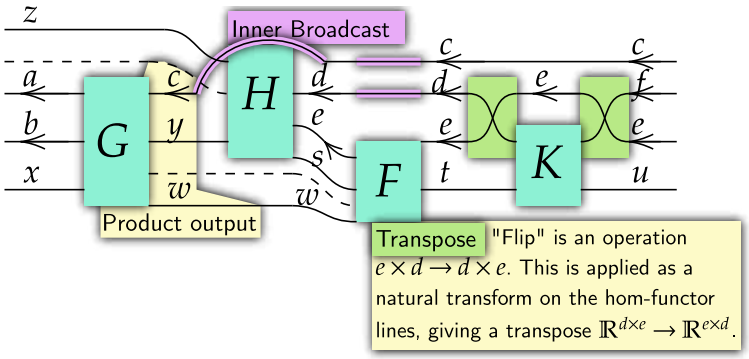


Figure 3.23: Here is an expression using functor string diagrams in a monoidal index category. As **Set** is a Cartesian index category, we could view the objects as sets, with hom-functor lines giving sets of functions such as $\mathbf{Set}(a \times b, x)$.

Starting with a Cartesian index category $\mathcal{C}$, we move into a subcategory. A subcategory may not be a Cartesian index category, rather, it is a category, closed under composition, drawn from one. This lets us keep any expression in the original category, as long as it is built from objects and morphisms contained in the subcategory. The subcategory therefore keeps many agreeable properties, without requiring strict specifications.

3.5.2 The $\mathbf{Set}(\_, \mathbb{R})$ Subcategory

We consider the subcategory $\mathbf{Set}(\_, \mathbb{R})$. This subcategory has objects $\mathbf{Set}(s, \mathbb{R})$, where s is any set. The unit object 1 is derived from $\mathbf{Set}(\emptyset, \mathbb{R})$, which we can also write $\mathbf{Set}(0, \mathbb{R})$. This means all objects in the subcategory have $\mathbb{R}$ as their underlying object, or are 1. Morphisms between objects $\mathbb{R}^a$ (or $\mathbf{Set}(a, \mathbb{R})$) and $\mathbb{R}^b$ in this subcategory are the functions between the sets represented by the objects $\mathbb{R}^a$ and $\mathbb{R}^b$.

As the underlying object in this subcategory are always the same, we do not need to diagram it. Instead, we introduce an equivalent expression where we only draw the hom-

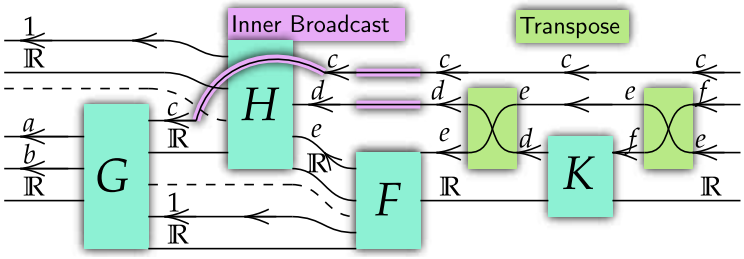


Figure 3.24: Here, all objects have $\mathbb{R}$ as their underlying object. This means the above expression, and all its constituent morphisms, are in the subcategory $\mathbf{Set}(\_, \mathbb{R})$.

functor wires, and do not drawn arrows. To diagram the object 1, we draw a 0 wire, which gives $\mathbb{R}^0$. This gives diagrams that begin to approach neural circuit diagrams.

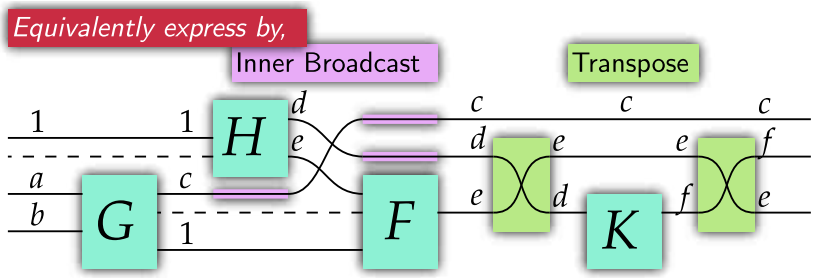


Figure 3.25: The expression in $\mathbf{Set}(\_, \mathbb{R})$ from Figure 3.24 can be diagrammed using an equivalent expression where we do not draw the underlying $\mathbb{R}$ object, keeping only the hom-functor wires. In this case, we do not draw arrows.

Next, we need to clearly define broadcasting on lines that are not on the lowest level. This is done by applying a transpose before and after a function. I show the transpose in Figure 3.26, and show lower broadcasting in Figure 3.27.

Figure 3.26: The transpose is an operation that swaps indexes. It can be used to define lower broadcasting.

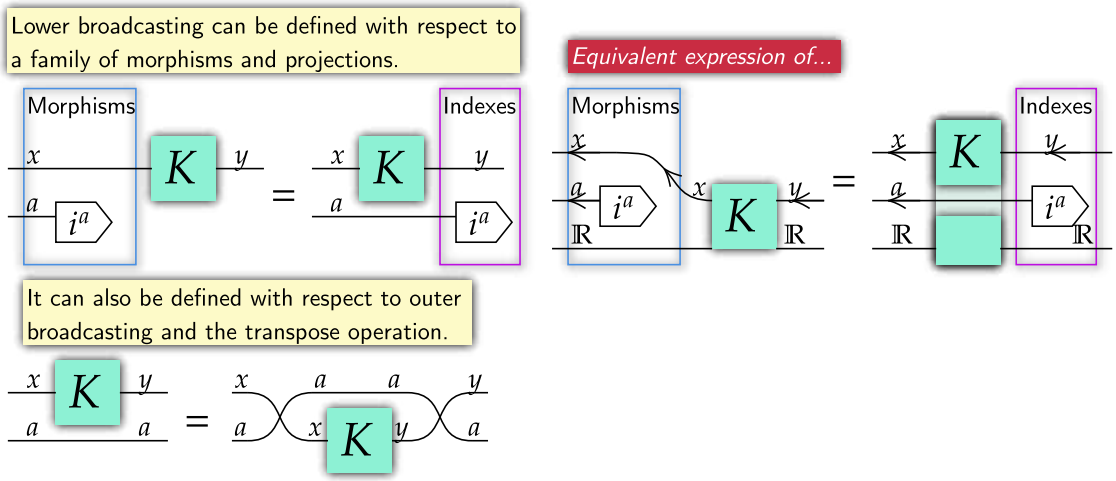
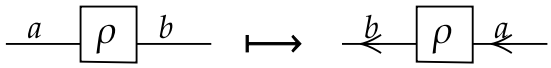


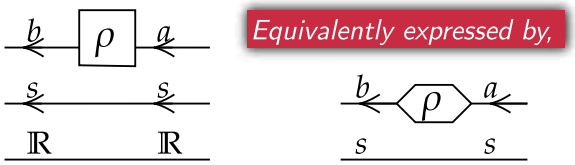
Figure 3.27: Often, in neural circuit diagrams we have wires pass below operations. This can be defined with a family of morphisms and projections, or by using two transpose operations.

3.5.3 Natural Transformation Components in Neural Circuit Diagrams

Indexes are akin to natural transformations applied onto functor wires, rather than functions on underlying objects. Operations $\rho : a \to b$ yield natural transformations $\mathbf{Set}(\rho, \_) : \mathbf{Set}(b, \_) \to \mathbf{Set}(a, \_)$ (see Proposition 4, or Subsection 3.2.5). We can diagram this as,



Applied onto an object $\mathbf{Set}(s, \mathbb{R})$ in our subcategory, we get a component;



Therefore, we see that neural circuit diagrams can express natural transformations between hom-wires by showing their components.

Functions derived from the components of natural transformations between hom-functor wires have special properties. They are computationally inexpensive and are natural with respect to functions on other wires. In a full functor string diagram expression, we can clearly see which functions are natural transformations and which are not.

If we want to maintain this information in neural circuit diagrams, we could diagram functions that are the components of some natural transformation between hom-functor

wires using hexagons or maintain the arrows on wires. We do not always need to indicate a function is a natural transformation component. However, it can, at times, lead to elegant graphical intuition for rearranging expressions.

Indexes are derived from coprojections $\langle i_a| : 1 \to a$ which yield natural transformations $\mathbf{Set}\left(\langle i_a|, \_\right) : \mathbf{Set}(a, \_) \to \mathbf{Set}(1, \_)$, and I diagram them with pointed pentagons. Transposes are a natural transform derived from the flip operation on tuples, and diagramming them like we do in Figure 3.26 shows the path that indexes follow.

Any operation defined solely from indexes is a natural transformation, and benefits from naturality and inexpensive evaluation. Natural transformations can also be derived from functions between index sets. This includes a host of operations, from diagonalization, which is the natural transform corresponding to copying, and convolution, which is the natural transform corresponding to addition, if we take addition on n and m to be the addition between $\{0, \ldots, n-1\} \subset \mathbb{N}$ and $\{0, \ldots, m-1\} \subset \mathbb{N}$, which gives $+ : n \times m \to (n+m-1)$.

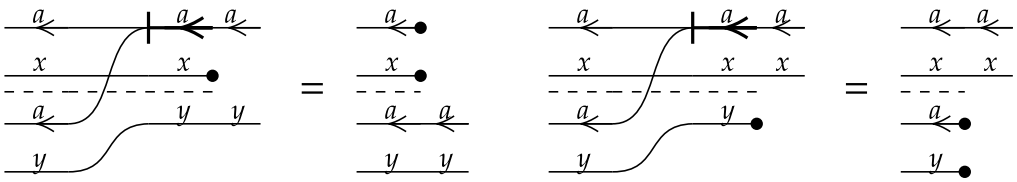
Neural circuit diagrams are uniquely positioned to identify natural transformations and to accommodate their unique properties. Natural transformations obey their own algebra. For instance, diagonalization duplicates natural transforms that come after it, as it is derived from copying. Natural transformations are also compatible with broadcasting, letting us place the axis with the lowest cardinality over the function we are broadcasting. This, and other manipulations, are revealed by neural circuit diagrams, granting us opportunities to understand existing algorithms, and to innovate new ones.

3.5.4 Dominant Axes

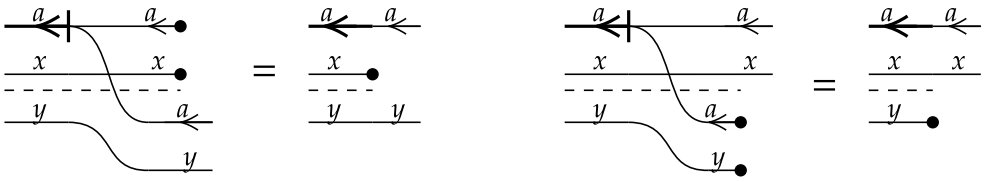
Previously, we briefly covered a case where we needed a "dominant" functor wire (see Figure 3.18). This was needed because we wanted to express $F(f \times g)$, rather than $Ff \times g$. Dominant functor wires are useful in index category expressions, where functor wires are hom-functors wires. Dominant functor wires let us express the relationship between $\mathbb{R}^{a \times (b+c)}$ as opposed to $\mathbb{R}^{a \times b + a \times c}$, which have a one-to-one correspondence of indexes but behave differently when broadcasting.

This correspondence is implemented by a distributor map, which is defined with respect to deletion in the following way. Observe that when a dominant functor wire is acting over only one object, it becomes a regular functor wire.

Definition 15 (Distributor)**.** *For a Cartesian monoidal category which is closed (the hom-objects $\mathcal{C}(a, x)$ are contained in the category), for objects a, x and y, the distributor is a morphism $\nabla_{a,xy} : \mathcal{C}(a, x) \times \mathcal{C}(a, y) \to \mathcal{C}(a, x \times y)$. We define it over the two projections of $\mathcal{C}(a, x) \times \mathcal{C}(a, y)$, which are the residue of deletion;*

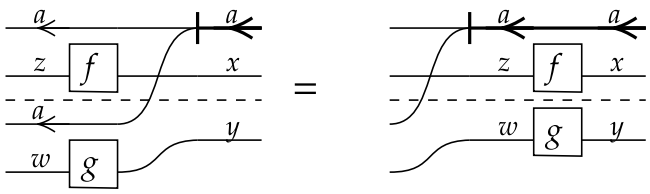


As this morphism is constructed from identities and deletion, it always exists in a closed Cartesian monoidal category. The reverse distributor is defined in the following way;



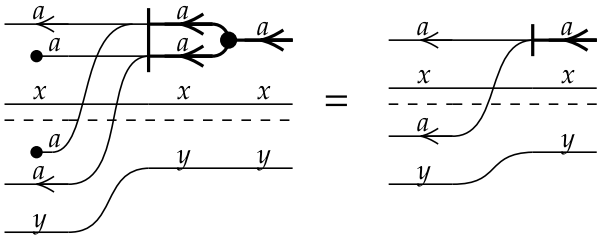
Significantly, the distributor is a natural transformation over independent morphisms, such as $\mathcal{C}(a, f) \times \mathcal{C}(a, g)$. Proof can be found in the appendix; see Appendix A.6.

Proposition 6 (Naturality of the Distributor)**.** *The distributor* $\nabla_{a,\_*} : \mathcal{C}(a, \_) \times \mathcal{C}(a, *) \to \mathcal{C}(a, \_ \times *)$ *acts like a natural transformation on independent morphisms,* $\mathcal{C}(a, f) \times \mathcal{C}(a, g)$*. This means the distributor is a natural transformation* $\nabla_a : \mathcal{C}(a, \_) \times \mathcal{C}(a, \_) \to \mathcal{C}(a, \_ \times \_)$*, which are functors of the form* $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$*.*

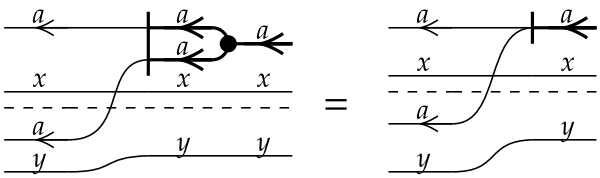


The distributor obeys another significant property that lets it give insight into how algorithms are structured. With respect to the natural transform associated to the copy map, the following proposition holds. I prove this in Appendix A.7.

Proposition 7 (The Distributor and Diagonalization)**.** *The distributor* $\nabla_{a,\_*} : \mathcal{C}(a, \_) \times \mathcal{C}(a, *) \to \mathcal{C}(a, \_ \times *)$ *and the natural transformation associated to the copy map,* $\mathcal{C}(\Delta, \_) : \mathcal{C}(a \times a, \_) \to \mathcal{C}(a, \_)$*, are related in the following way;*



We can also draw this expression as;

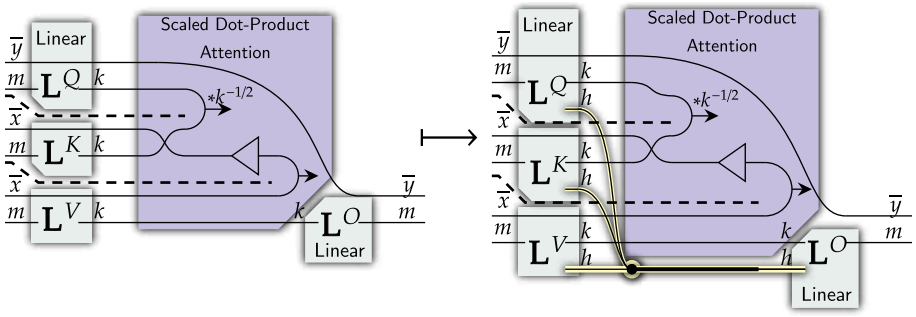


This has critical applications. As diagonalization, the natural transform of the copy map, is a natural transform, we can slide it along expressions. Whenever we have a tuple of independent operations, the distributor is also natural, meaning we can slide it along. These facts let us rearrange algorithms, easily adding additional features that take advantage of parallelization. Then, using the naturality of the distributor and diagonalization to find an implementation.

3.5.5 Natural Transformations and Multi-Head Attention

An example of applying these facts are with multi-head attention. Multi-head attention broadcasts scaled dot-product attention over an additional h-length axis (see Section 2.3.2). This efficiently uses parameters and is essential to the effectiveness of transformer architectures (Vaswani et al., 2017; Lin et al., 2021). However, it is not obvious how to implement or diagram the splitting and recombination of parallel heads. As I describe in Section 1.2, the typical presentation of splitting and concatenating these heads is difficult to understand.

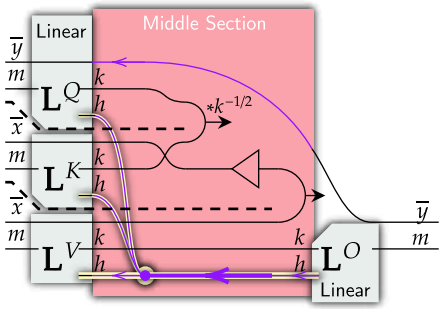
Using neural circuit diagrams, we can first reason that splitting and joining attention heads should have an overall structure that looks something like the following;



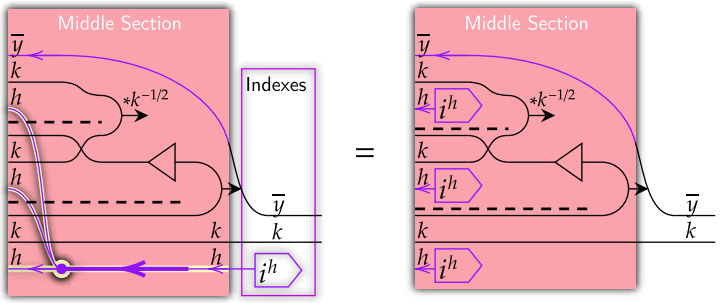
This has scaled dot-product attention act in parallel, as we have an additional h-axis for each linear projection. These are joined, and scaled dot-product attention broadcast over it. However, it is not obvious how we would go about implementing such an operation. An imperative loop is one option but it fails to take advantage of the parallelization benefits of linear operations and graphical processing units.

To find an implementation, we use an insight from category theory. We isolate the middle section. Within this section, the h-axis is purely natural; all operations on it are natural transformations. Natural transformations exist within expressions as component

functions, meaning they accompany vertical and horizontal section decompositions. I have highlighted the purely natural axes within the middle section in purple.

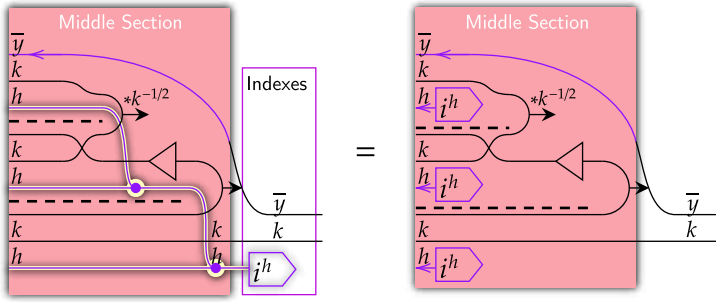


Then, we can interrogate this with a generic index over the h-axis. Using broadcasting and naturality rules, we can expand this expression, moving the indexes to the front. Here, we use an insight from the previous section. For index categories, the behaviour of hom-functors and natural transformations between them are completely derived from and hence encompassed by indexes and the broadcast rules.



Recognize that the left-hand side can be extended over a family of indexes. Therefore, the expression on the right-hand side defines it. Importantly, this implies that any other assembly of functions and natural transformations that reduces to the right-hand side is equivalent to the left-hand side.

This guides us toward a step-by-step implementation. This expression is equal to the expression is equal to the above over indexes, which form a family of projections, meaning the expressions are equivalent.



This rearrangement can be clearly implemented using efficient linear operations. Overall, this case study of natural transformations and multi-head attention show why the categorical structure of neural circuit diagrams offers unique insight. Firstly, we saw that multi-head attention, though conceptually difficult, can be clearly shown using neural circuit diagrams. However, this expression did not have a clear implementation. Then, I used naturality and a generic index to show that any assembly of natural transformations that yields the same relationship between initial and final indexes is equivalent. This was used to derive the implementation of multi-head attention from Figure 2.19.

3.6 Conclusion

Functor string diagrams are a powerful graphical language that can robustly communicate compositional structures. Compared to standard monoidal string diagrams (Selinger, 2009), functor string diagrams can readily show functors and natural transformations, structure preserving maps. For deep learning, this allows the details of axes to be simultaneously communicated with independent products. This is a novel field, with only two works having extensively studied this approach to diagrammatic category theory (Marsden, 2014; Nakahira, 2023).

In this chapter, I contributed an alternative approach to functor string diagrams that focuses on vertical section decomposition and equivalent expressions to efficiently scale diagrams to new situations. This contrasts with the previous approaches, which focus on colored regions that express which category we are working with and how we move between them. I feel this approach prevents diagrams from focusing on composition and graphical intuition. I show that my framework is able to prove the Yoneda lemma, as well as provide a rigorous foundation for broadcasting with indexes and neural circuit diagrams.

This rigorous foundation for broadcasting and neural circuit diagrams provides a mathematically explicit model of deep learning architectures as they are used in practice. This contrasts to the limited frameworks currently used (Shiebler et al., 2021; Cruttwell et al., 2021; Fong et al., 2019; Saxe et al., 2019). This opens up exciting avenues for future research, which will allow us to consider the theory of deep learning architectures in a manner that is relevant to applied models.

Chapter 4

Concluding Remarks

4.1 Future Work

The most exciting aspect of this thesis are the many avenues for future research it opens. Neural circuit diagrams can, of course, be used to diagram algorithms not covered in this paper. Complex models like the text-to-image diffusion architectures (Ho et al., 2020; Nichol and Dhariwal, 2021) would particularly benefit from such investigations. However, there are three approaches which go deeper that I believe would be immensely beneficial to the deep learning community, and address aspects overlooked in this thesis.

4.1.1 Implementations

The relationship between diagrams and implementation can be further developed. I have already shown that there is a close correspondence between diagrams and code. However, as diagrams *are* an explicit set of instructions it is possible to use them as code. Like any high-level programming language, the details of components are hidden. Interestingly, neural circuit diagrams are a high-level language with very explicit memory management. Diagrams can serve as a platform-agnostic visual programming language which can then be integrated into existing packages such as PyTorch or TensorFlow. Creating tools to automatically move between code and diagrams, or diagrams and code, is an exciting route for future research.

Furthermore, algorithms defined using neural circuit diagrams have the potential to be rearranged and have their performance improved using some of the tricks I have developed. For instance, linear operations can be reorganized to accelerate computation, or natural transforms can be rearranged to avoid calculations. Given that the possible rearrangements of linear operations and natural transformations changes under composition, an automated system may be able to find cases where $(A; B)$ is cheaper to evaluate than A followed by B. This may yield similar results and perhaps even subsume approaches such as Xu et al. (2023), where a graph tensor network perspective was used to reinterpret, rearrange, and improve various algorithms' performance.

4.1.2 Mathematics All the Way Down

In their survey, Shiebler et al. (2021) covers the main category theory perspectives on machine learning. They cover the parametric gradient-based learning perspective, which allows models to consider the accumulation of hidden parameters as a type of composition and to automatically produce associated learning algorithms that improve these parameters (Fong et al., 2019; Cruttwell et al., 2021). Additionally, the survey covers Markov categories which allow statistical characteristics of models to be graphically analyzed (Fritz and Rischel, 2020).

However, in the discussion, they note these approaches are disjoint and in need of unification. I believe functor string diagrams would allow these concepts to be understood in a graphically intuitive manner and then reconciled with neural circuit diagrams. By reconciling parametric-based and probabilistic-based neural circuit diagrams, we could specify when both perspectives are available to us. However, this would require a careful treatment of measure theory to ensure the functions generated by diagrams are probabilistically meaningful.

If successful, such an investigation would have many benefits. It would mean neural circuit diagrams more accurately represent models by considering compositional properties related to hidden parameters and internal randomness. It would allow deep learning models to be clearly understood as probabilistic learning mechanisms, randomly suggesting improvements on hidden parameters. They would be mathematically explicit all the way down, opening up robust analysis of training, convergence, and generalization in a rigorous measure theory manner.

Results drawn from such an analysis would be applicable to any model that can be represented by a neural circuit diagram. This is, of course, an immense scope, and would be in significant contrast to much of the existing theoretical work, which often derive specialized results about specialized toy models.

4.1.3 The Mechanics of Information

Neural circuit diagrams and Markov categories offer an exciting opportunity to analyze the mechanics of information throughout a model. Information theory is at the intersection of mechanics and statistics, assigning quantitative values to concepts such as certainty or relatedness. These values can be empirically studied, as was done by Saxe et al. (2019) to disprove old hypotheses about deep learning models (Tishby et al., 2000). Information theory offers a rigorous and empirically testable means of developing theory and intuition about models.

This promises a significant improvement over works such as He et al. (2016) which resort to vague and ultimately untestable theories about why certain features benefit architectures. Effectively, the only tool He et al. (2016) had available to analyze models in general was differentiation. Differentiation is one of the few operations with well-known compositional behavior and which can be used to analyze models in general.

However, a graphical category theory approach opens up additional compositional tools. A particularly promising approach is by Perrone (2022), who uses Markov categories to graphically analyze information theory concepts such as entropy and divergence. These concepts form the basis of most loss functions. Therefore, understanding their mechanics will allow model performance to be better understood.

Combining neural circuit diagrams with a probabilistic perspective will open up these tools. This would allow the minimum theoretical increase in the loss function from certain design choices to be assessed. By providing the means to graphically assess both the performance and computational complexity impacts of design choices, neural circuit diagrams could become essential to the model design process.

4.2 Conclusion

In this thesis, I identified an open problem in deep learning research and presented a solution with practical applications and robust underlying theory. In *The Problem and the Solution*, I identified that deep learning research suffers from imprecise models of architectures, and that category theory is a promising approach to take.

Moving onto the *Applications of Neural Circuit Diagrams*, I presented neural circuit diagrams as a robust diagramming scheme that, for the first time, lets models be systematically visually communicated and analyzed. I showed many practical applications of this framework, proving it can communicate contemporary architectures and elucidate opportunities for innovation more clearly than existing methods.

Finally, I developed the *Theory of Functor String Diagrams*. I extended a nascent graphical approach to category theory from Marsden (2014) and Nakahira (2023) into something that provides a flexible graphical calculus and that offers a robust foundation for neural circuit diagrams. The principled approach I present emphasizes the compositional character of diagrams.

These techniques worked extraordinarily well. A host of architectures can be clearly communicated by neural circuit diagrams, all the while having close correspondence to code and necessarily offering an explicit mathematical expression. Functor string diagrams proved to be a powerful way of approaching category theory, with mechanisms to present functors, naturality, family expressions, products, and other constructs.

This success, I believe, comes back to the nature of the problem we are dealing with. The nature of a problem hints at the relevant mathematics. Particle physics studies symmetry; hence group theory has remarkable applications. The same goes for deep learning architectures and category theory. They are highly composed systems with a compositional structure needing to be interpreted at multiple levels of analysis. Category theory studies composition and how it is preserved across perspectives, meaning it forms a natural foundation for a robust conceptualization of deep learning architectures, bridging the gap between theory and application.

Appendix: Accompanying Proofs

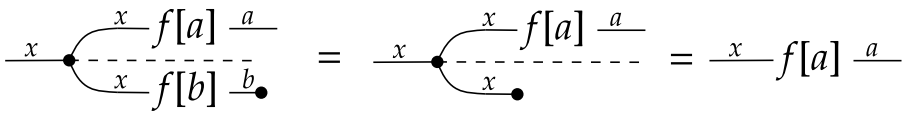
1 The Residues of Deletion gives Projections

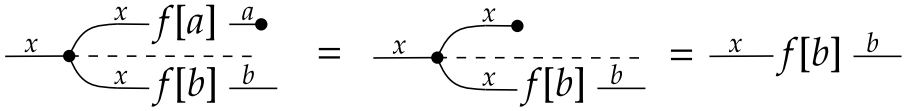
Proposition 1 (The Residues of Deletion gives Projections)**.** *A Cartesian monoidal category, as defined above, has Cartesian projections for any object $a \times b$ given by $\pi[a]_a^{a \times b} = Id_a^a \times del_u^b$ and $\pi[b]_b^{a \times b} = del_u^a \times Id_b^b$.*

The projections are, therefore, the following morphisms;

Proof. First, we prove the free construction property. This would require that any pair of morphisms $f[a]_a^x$ and $f[b]_b^x$ can be constructed into a morphism $f_{a\times b}^x$ such that,

We can do this using the copy map. Observe that,

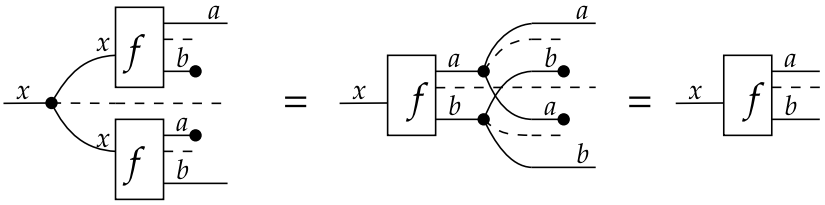




Therefore, using the semi-Cartesian properties, we can show that $a \times b$ accepts free construction.

Next, we prove the complete description property. This relies on the copy map's duplicating property. This requires showing that if $f^x_{a\times b};\pi[a]^{a\times b}_a = g^x_{a\times b};\pi[a]^{a\times b}_a$ and $f^x_{a\times b};\pi[b]^{a\times b}_b = g^x_{a\times b};\pi[b]^{a\times b}_b$, then $f^x_{a\times b} = g^x_{a\times b}$.

In other words, $f^x_{a\times b};\pi[a]^{a\times b}_a$ and $f^x_{a\times b};\pi[b]^{a\times b}_b$ are sufficient to uniquely identify $f^x_{a\times b}$. Therefore, we start with $f^x_{a\times b};\pi[a]^{a\times b}_a$ and $f^x_{a\times b};\pi[b]^{a\times b}_b$, and attempt to reconstruct $f^x_{a\times b}$.



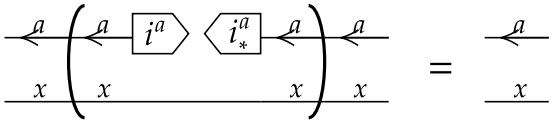
We see that the existence of a duplicating copy map implies a one-to-one correspondence between pairs of morphisms $(f[a]^x_a, f[b]^x_b)$ and morphisms $f^x_{a\times b}$. Therefore, $a \times b$ is a Cartesian product of a and b. □

2 Indexes as Projections

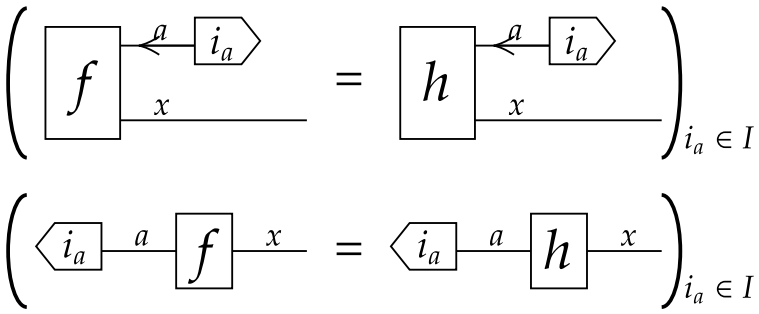
Proposition 2 (Indexes as Projections)**.** *In an index category $\mathcal{C}$, where:*

- *An object a has an I-family of coprojections $(\langle i_a|)_{i\in I}$,*

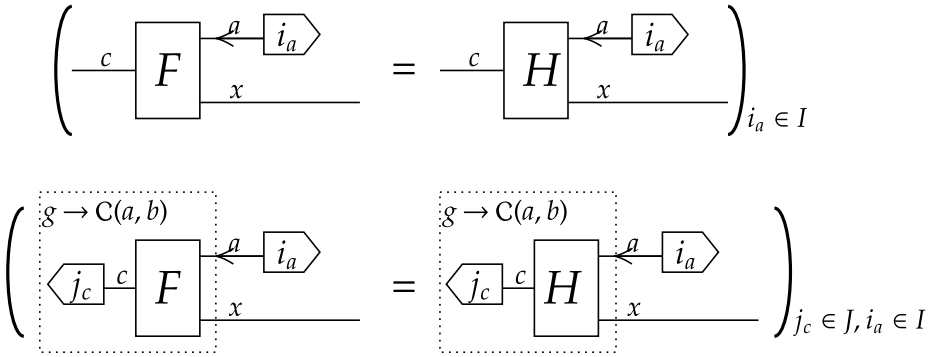
Then indexes $(|i^a\rangle x^x_x)_{i\in I}$ forms a family of projections for $\mathcal{C}(a,x)$. We can diagram this as:



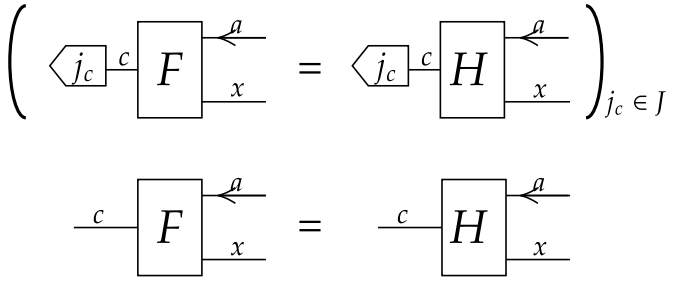
Proof. First, the complete description property. Assume for two morphisms f_{ax} and h_{ax}, the following are equal for all indexes,



Therefore, as $(\langle i_a|)_{\,i\in I}$ gives a family of coprojections, f must equal g. So, $(|i^a\rangle x^x_x)_{\,i\in I}$ gives a complete description of morphisms $g \to \mathcal{C}(a,b)$. For morphisms $F, H : c \to \mathcal{C}(a,b)$, we expand over the coprojections of c.



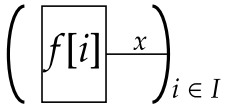
We use the complete description property we just derived to contract this expression over the i_a indexes of I.



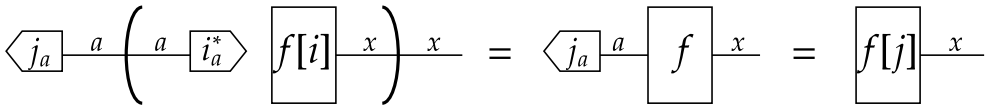
This final derivation means we have proved that our initial expression gives a complete description.

For free construction, we need to prove that for any I_a-family of morphisms $(h[i]^w_x)_{i\in I_a}$, there exists a morphism h^w_{ax} such that $h[i]^w_x = h^w_{ax};|i^a\rangle x^x_x$ for $i \in I_a$. We do this in two steps, first proving it for $(f[i]^g_x)_{i\in I_a}$, and then for the full expression.

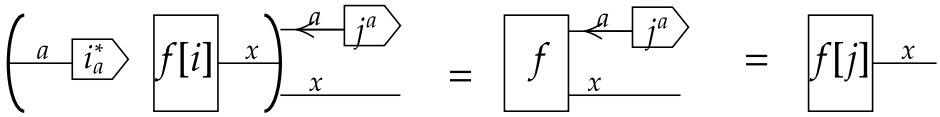
We start with a family of morphisms $(f[i]^g_x)_{i\in I_a}$,



By free construction, we have a morphism that gives these values when coprojected.

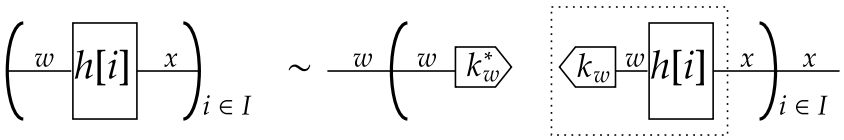


By the index interaction rule, the representation of the $a \to x$ morphisms as $g \to \mathcal{C}(a, x)$ accepts indexes in the following way.

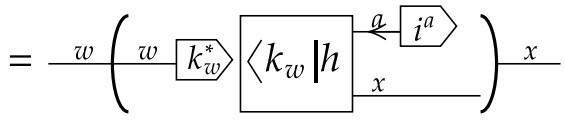


Therefore, for any family of morphisms $(f[i]^g_x)_{i \in I_a}$, there is a morphism $g \to \mathcal{C}(a, x)$ which returns those morphisms under the indexes.

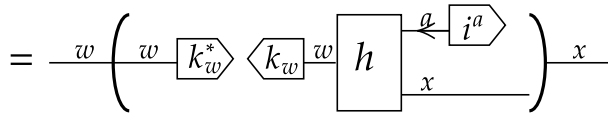
Having established this, we now move to a family of morphisms $(h[i]^w_x)_{i \in I_a}$. We expand over the coprojections of the w axis;



Composition is closed, meaning $\langle k_w |; h[i]^w_x$ is a morphism $g \to x$. This is a family of morphisms $(\langle k_w |; h[i]^w_x)_{i \in I_a}$, which we can expand according to the result we derived above. We have free construction over coprojections, meaning a morphism h^w_x that is



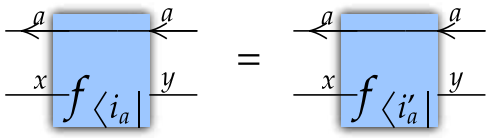
equal to $\langle k_w | h^w_x$ for all values of $k \in K_w$ must exist.



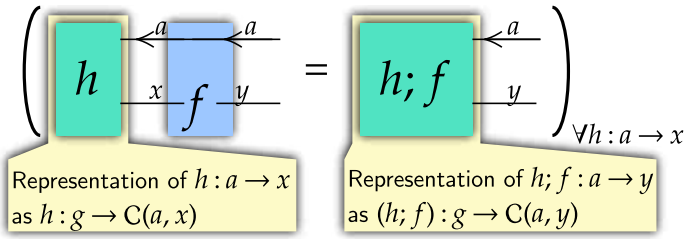
Constructing the appropriate expression. □

3 Outer Broadcasting as the Internal Hom-Functor

Proposition 3 (Outer Broadcasting as the Internal Hom-Functor)**.** *An a-fold outer broadcast of a morphism $f : x \to y$ is equal for all choice of families of coprojections. For any two families of coprojections $(\langle i_a|)_{i\in I_a}$ and $(\langle i'_a|)_{i\in I_a}$, we have,*

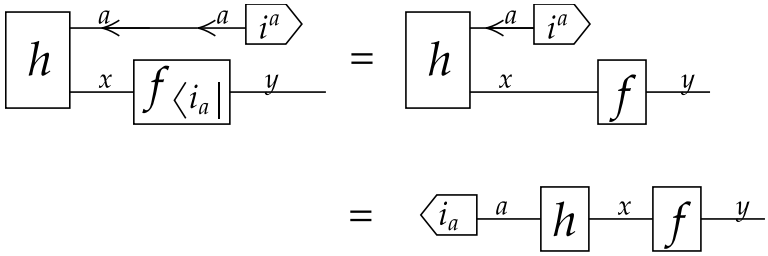


This means we can diagram an a-fold outer broadcast of a morphism $f : x \to y$ without reference to a family of coprojections. Furthermore, this expression behaves like the hom-functored $\mathcal{C}(a, f) : \mathcal{C}(a, x) \to \mathcal{C}(a, y)$, but internal to $\mathcal{C}$ without exiting to **Set**.

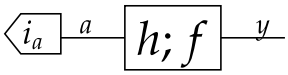


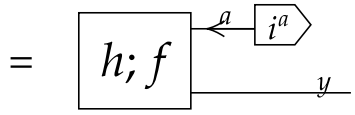
This means we can diagram an a-fold outer broadcast of a morphism $f : x \to y$ without reference to a family of coprojections.

Proof. We interrogate some $f_{\langle i_a|}$ over morphisms $h : g \to \mathcal{C}(a, x)$. This lets us make a statement over all possible coprojections. We use the index interaction rule and compose the expressions.

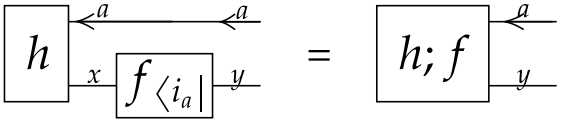


We again use the index interaction rule.





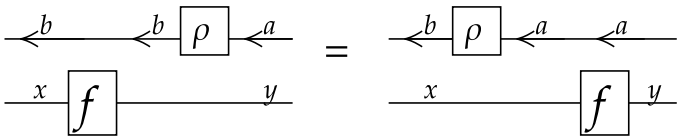
The expression holds for a family of indexes, which we previously proved form a family of projections.



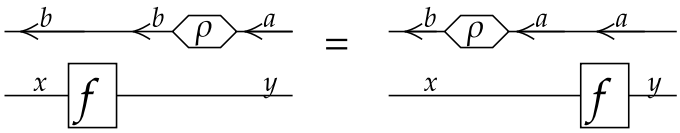
Finally, we see that over all $h : g \to \mathcal{C}(a, x)$, which covers the coprojections of $\mathcal{C}(a, x)$, that $f_{\langle i_a|}$ is equal to an expression that makes no reference to the family $(\langle i_a|)_{i \in I}$. Therefore, $f_{\langle i_a|}$ is independent of the choice of the family of coprojections or corresponding indexes. □

4 Corresponding Natural Transformations

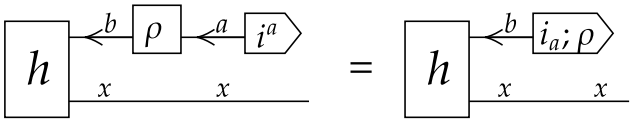
Proposition 4 (Corresponding Natural Transformations)**.** *Corresponding natural transformations $\mathcal{C}(\rho, x) : \mathcal{C}(b, x) \to \mathcal{C}(a, x)$ are independent of the choice of family of indexes used to define them, and are indeed natural transformations. This implies that;*

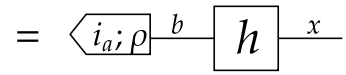


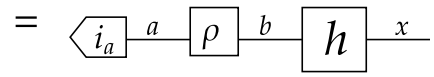
We can equivalently express these using hexagons, to distinguish their naturality.

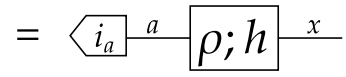


Proof. Proof. First, we prove that the expression is independent of the chosen family of coprojections.

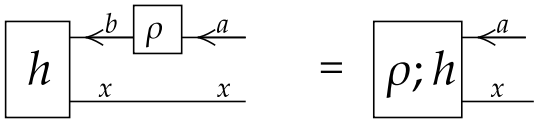






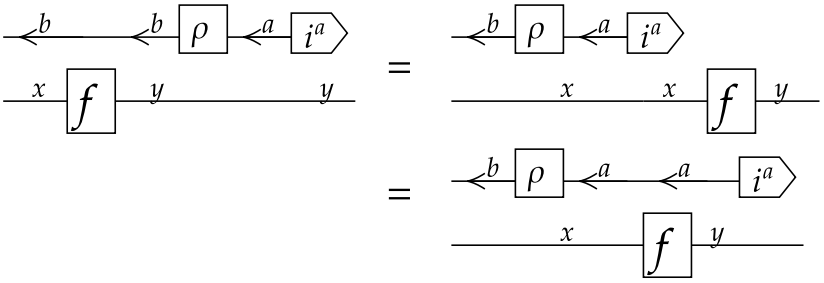


As these are equal over a family of indexes, we get that;

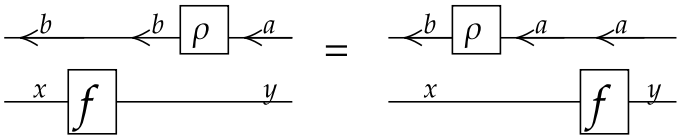


Now, we expand $\langle i_a|; \rho^a_b$ into a morphism and the index. This lets us get two expressions that are equal over all indexes.

To show naturality, consider that we have $\langle i_a|; \rho^a_b : g \to b$ pass through in the manner of broadcasting, as broadcasting is defined for any family of coprojections, and every morphism $g \to b$ is in a family of coprojections.



The above expression holds over the entire $(|i^a\rangle x^x_x)_{\,i \in I}$ family of projections. Therefore, we get;

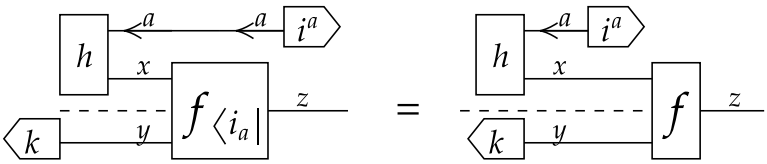


□

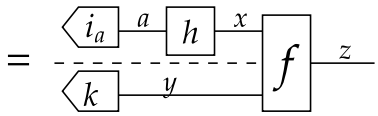
5 Inner Broadcasting

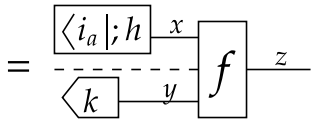
Proposition 5 (Inner Broadcasting)**.** *Inner broadcasted morphisms over a are equal for all choices of families of coprojections.*

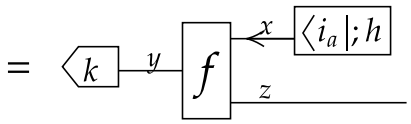
Proof. We interrogate an expression over the coprojections of $\mathcal{C}(a, x)$, k and indexes of a, and use the definition of the inner broadcast on the right-hand side. We use the



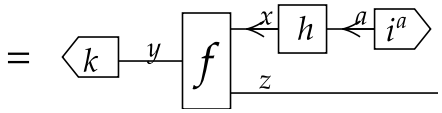
index representation rule, Then, the index interaction rule. This gives us an expression







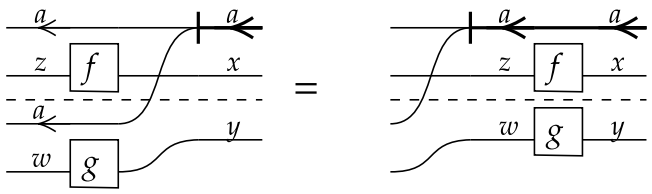
over the indexes, a family of projections. Meaning we get; This equates the inner-



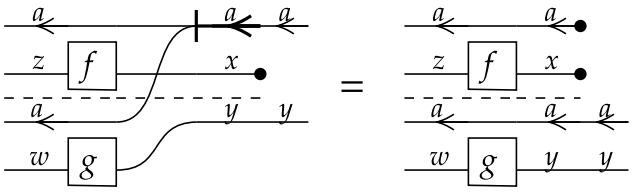
broadcasted f with the right-hand-side, an expression independent of the choice of family of coprojections. □

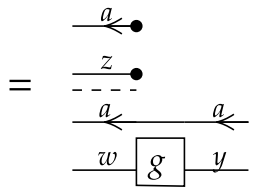
6 Naturality of the Distributor

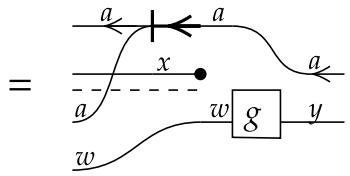
Proposition 6 (Naturality of the Distributor)**.** *The distributor $\nabla_{a,\_*} : \mathcal{C}(a, \_) \times \mathcal{C}(a, *) \to \mathcal{C}(a, \_ \times *)$ acts like a natural transformation on independent morphisms, $\mathcal{C}(a, f) \times \mathcal{C}(a, g)$. This means the distributor is a natural transformation $\nabla_a : \mathcal{C}(a, \_) \times \mathcal{C}(a, \_) \to \mathcal{C}(a, \_ \times \_)$, which are functors of the form $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$.*



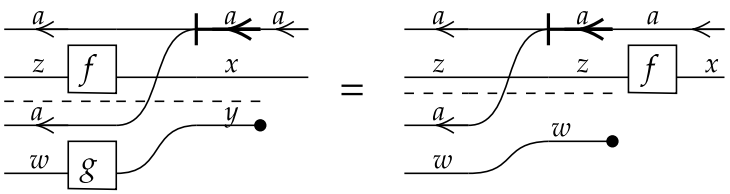
Proof. We expand the left-hand hand-side, using the definition of the distributor interacting with a deletion.







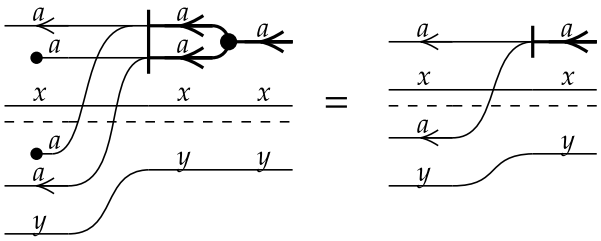
Similarly, we show that,



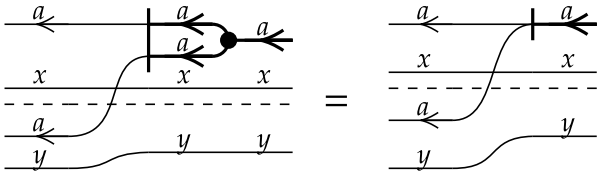
As the residue of deletion gives projections, this means applying a distributor before or after an expression is the same, proving the proposition. □

7 The Distributor and Diagonalization

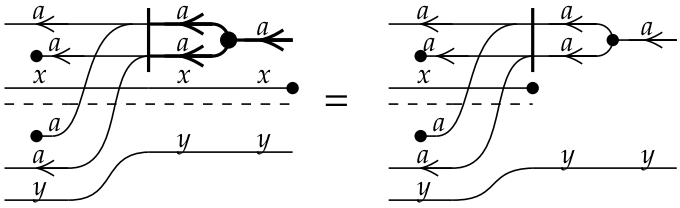
Proposition 7 (The Distributor and Diagonalization)**.** *The distributor* $\nabla_{a,\_*} : \mathcal{C}(a,\_) \times \mathcal{C}(a,*) \to \mathcal{C}(a,\_\times *)$ *and the natural transformation associated to the copy map,* $\mathcal{C}(\Delta,\_) : \mathcal{C}(a\times a,\_) \to \mathcal{C}(a,\_)$*, are related in the following way;*



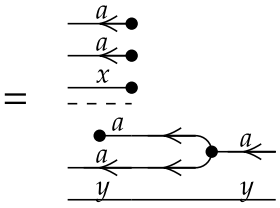
We can also draw this expression as;



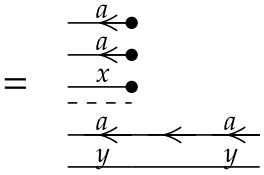
Proof. To prove this proposition, we interrogate the expression over the residues of deletion, which act as projections. On the right, we use the fact that $\mathcal{C}(\Delta, \_) : \mathcal{C}(a \times a, \_) \to \mathcal{C}(a, \_)$ is a natural transformation to move the copy map backwards.



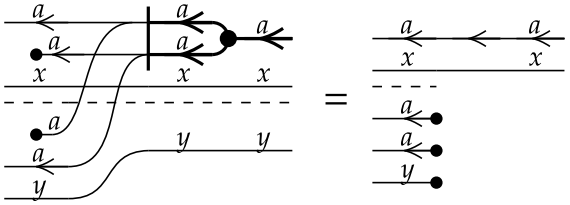
We then use the definition of the distributor with respect to the delete map.



We then use the copy-delete property to recover the identity.



We similarly prove it for deletion on y.



As the two expressions are equal over both projections, the proposition holds. □

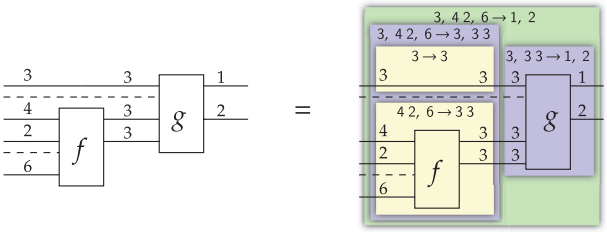
Appendix B

Appendix: Accompanying Code

```
import torch
import typing
import functorch
import itertools
```

2.2.4 Tensors

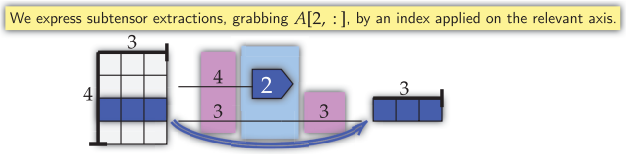
We diagrams tensors, which can be vertically and horizontally decomposed.




```
# This diagram shows a function h : 3, 4 2, 6 -> 1 2
constructed out of f: 4 2, 6 -> 3 3 and g: 3, 3 3 -> 1
2
# We use assertions and random outputs to represent
generic functions, and how diagrams relate to code.
T = torch.Tensor
def f(x0 : T, x1 : T):
    """ f: 4 2, 6 -> 3 3 """
    assert x0.size() == torch.Size([4,2])
    assert x1.size() == torch.Size([6])
    return torch.rand([3,3])
def g(x0 : T, x1: T):
    """ g: 3, 3 3 -> 1 2 """
    assert x0.size() == torch.Size([3])
    assert x1.size() == torch.Size([3, 3])
    return torch.rand([1,2])
def h(x0 : T, x1 : T, x2 : T):
    """ h: 3, 4 2, 6 -> 1 2"""
    assert x0.size() == torch.Size([3])
    assert x1.size() == torch.Size([4, 2])
    assert x2.size() == torch.Size([6])
    return g(x0, f(x1,x2))

h(torch.rand([3]), torch.rand([4, 2]), torch.rand([6]))
```

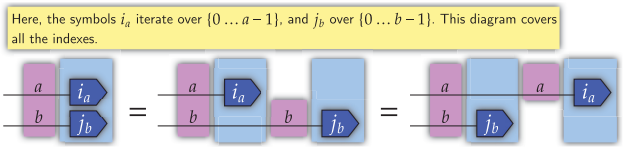
2.2.5 Indexes

Figure 2.7: Indexes




```
# Extracting a subtensor is a process we are familiar
with. Consider,
# A (4 3) tensor
table = torch.arange(0,12).view(4,3)
row = table[2,:]
row
```

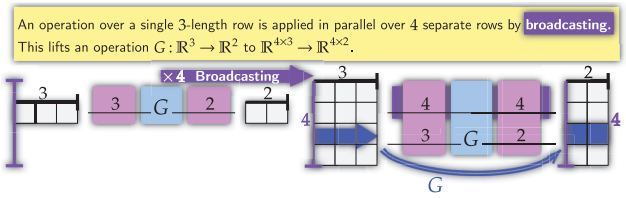
Figure 2.8: Subtensors




```
# Different orders of access give the same result.
# Set up a random (5 7) tensor
a, b = 5, 7
Xab = torch.rand([a] + [b])
# Show that all pairs of indexes give the same result
for ia, jb in itertools.product(range(a), range(b)):
    assert Xab[ia, jb] == Xab[ia, :][jb]
    assert Xab[ia, jb] == Xab[:, jb][ia]
```

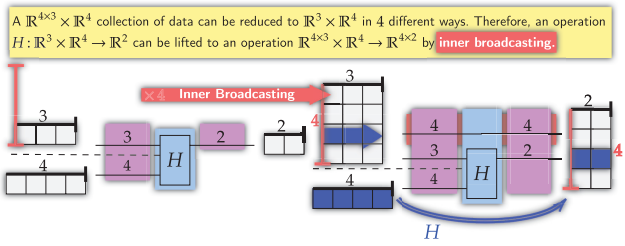
2.2.6 Broadcasting

Figure 2.9: Broadcasting



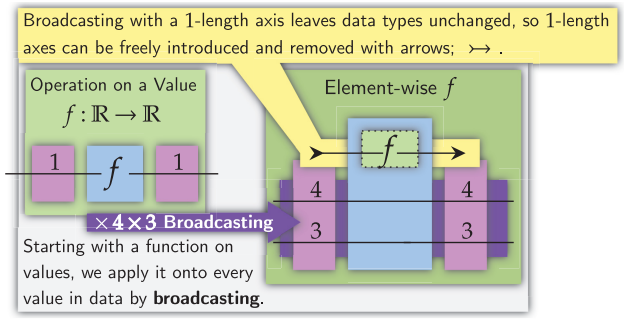

```
a, b, c, d = [3], [4], [5], [3]
T = torch.Tensor

# We have some function from a to b;
def G(Xa: T) -> T:
    """ G: a -> b """
    return sum(Xa**2) + torch.ones(b)

# We could bootstrap a definition of broadcasting,
# Note that we are using spaces to indicate tensoring.
# We will use commas for tupling, which is in line with
standard notation while writing code.
def Gc(Xac: T) -> T:
    """ G c : a c -> b c """
    Ybc = torch.zeros(b + c)
    for j in range(c[0]):
        Ybc[:,jc] = G(Xac[:,jc])
    return Ybc

# Or use a PyTorch command,
# G *: a * -> b *
Gs = torch.vmap(G, -1, -1)

# We feed a random input, and see whether applying an
index before or after
# gives the same result.
Xac = torch.rand(a + c)
for jc in range(c[0]):
    assert torch.allclose(G(Xac[:,jc]), Gc(Xac)[:,jc])
    assert torch.allclose(G(Xac[:,jc]), Gs(Xac)[:,jc])

# This shows how our definition of broadcasting lines
up with that used by PyTorch vmap.
```

Figure 2.10: Inner Broadcasting




```
# We have some function which can be inner broadcast,
def H(Xa: T, Xd: T) -> T:
    """ H: a, d -> b """
    return torch.sum(torch.sqrt(Xa**2)) +
torch.sum(torch.sqrt(Xd ** 2))  + torch.ones(b)

# We can bootstrap inner broadcasting,
def Hc0(Xca: T, Xd : T) -> T:
    """ c0 H: c a, d -> c d """
     # Recall that we defined a, b, c, d in [_] arrays.
    Ycb = torch.zeros(c + b)
    for ic in range(c[0]):
        Ycb[ic,  :] = H(Xca[ic, :], Xd)
    return Ycb

# But vmap offers a clear way of doing it,
# *0 H: * a, d -> * c
Hs0 = torch.vmap(H, (0, None), 0)

# We can show this satisfies Definition 2.14 by,
Xca = torch.rand(c + a)
Xd = torch.rand(d)
for ic in range(c[0]):
    assert torch.allclose(Hc0(Xca, Xd)[ic, :],
H(Xca[ic, :], Xd))
    assert torch.allclose(Hs0(Xca, Xd)[ic, :],
H(Xca[ic, :], Xd))
```

Figure 2.11 Elementwise operations

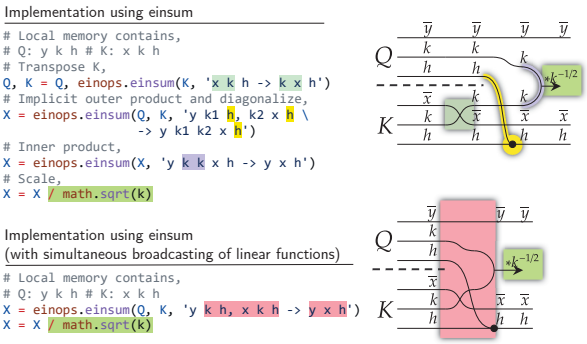



```
# Elementwise operations are implemented as usual ie
def f(x):
    "f : 1 -> 1"
    return x ** 2

# We broadcast an elementwise operation,
# f *: * -> *
fs = torch.vmap(f)

Xa = torch.rand(a)
for i in range(a[0]):
    # And see that it aligns with the index before =
index after framework.
    assert torch.allclose(f(Xa[i]), fs(Xa)[i])
    # But, elementwise operations are implied, so no
special implementation is needed.
    assert torch.allclose(f(Xa[i]), f(Xa)[i])
```

2.4 Linearity

Implementing Linearity and Common Operations

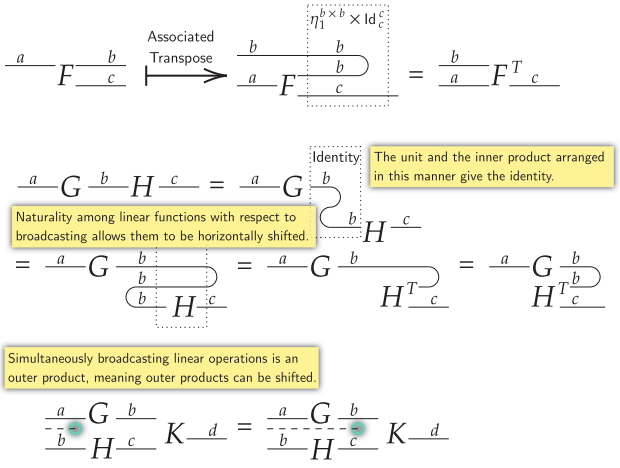
Figure 2.15: Multi-head Attention and Einsum




```
import math
import einops
x, y, k, h = 5, 3, 4, 2
Q = torch.rand([y, k, h])
K = torch.rand([x, k, h])

# Local memory contains,
# Q: y k h # K: x k h
# Outer products, transposes, inner products, and
# diagonalization reduce to einops expressions.
# Transpose K,
K = einops.einsum(K, 'x k h -> k x h')
# 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)

Q = torch.rand([y, k, h])
K = torch.rand([x, k, h])

# 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)
```

2.2.10 Linear Algebra

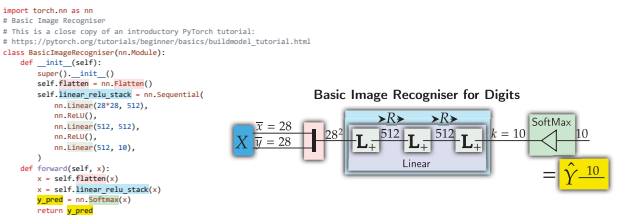
Figure 2.16: Graphical Linear Algebra



```
# We will do an exercise implementing some of these
equivalences.
# The reader can follow this exercise to get a better
sense of how linear functions can be implemented,
# and how different forms are equivalent.

a, b, c, d = [3], [4], [5], [3]

# We will be using this function *a lot*
es = einops.einsum

# F: a b c
F_matrix = torch.rand(a + b + c)

# As an exericse we will show that the linear map F: a
-> b c can be transposed in two ways.
# Either, we can broadcast, or take an outer product.
We will show these are the same.

# Transposing by broadcasting
#
def F_func(Xa: T):
    """ F: a -> b c """
    return es(Xa,F_matrix,'a,a b c->b c',)
# * F: * a -> * b c
F_broadcast = torch.vmap(F_func, 0, 0)

# We then reduce it, as in the diagram,
# b a -> b b c -> c
def F_broadcast_transpose(Xba: T):
    """ (b F) (.b c): b a -> c """
    Xbbc = F_broadcast(Xba)
    return es(Xbbc, 'b b c -> c')

# Transpoing by linearity
#
# We take the outer product of Id(b) and F, and follow
up with a inner product.
# This gives us,
F_outerproduct = es(torch.eye(b[0]), F_matrix,'b0 b1, a
b2 c->b0 b1 a b2 c',)
# Think of this as Id(b) F: b0 a -> b1 b2 c arranged
into an associated b0 b1 a b2 c tensor.
# We then take the inner product. This gives a (b a c)
matrix, which can be used for a (b a -> c) map.
F_linear_transpose = es(F_outerproduct,'b B a B c->b a
c',)

# We contend that these are the same.
#
Xba = torch.rand(b + a)
assert torch.allclose(
    F_broadcast_transpose(Xba),
    es(Xba,F_linear_transpose, 'b a, b a c -> c'))

# Furthermore, lets prove the unit-inner product
identity.
#
# The first step is an outer product with the unit,
outerUnit = lambda Xb: es(Xb, torch.eye(b[0]), 'b0, b1
b2 -> b0 b1 b2')
# The next is a inner product over the first two axes,
dotOuter = lambda Xbbb: es(Xbbb, 'b0 b0 b1 -> b1')
# Applying both of these *should* be the identity, and
hence leave any input unchanged.
Xb = torch.rand(b)
assert torch.allclose(
    Xb,
    dotOuter(outerUnit(Xb)))

# Therefore, we can confidently use the expressions in
Figure 8 to manipulate expressions.
```

2.3.1 Basic Multi-Layer Perceptron

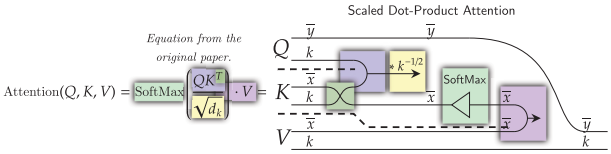
Figure 2.17: Implementing a Basic Multi-Layer Perceptron




```
import torch.nn as nn
# Basic Image Recogniser
# This is a close copy of an introductory PyTorch
tutorial:
#
https://pytorch.org/tutorials/beginner/basics/buildmode
l_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

my_BasicImageRecogniser = BasicImageRecogniser()
my_BasicImageRecogniser.forward(torch.rand([1,28,28]))
```

2.3.2 Neural Circuit Diagrams for the Transformer Architecture

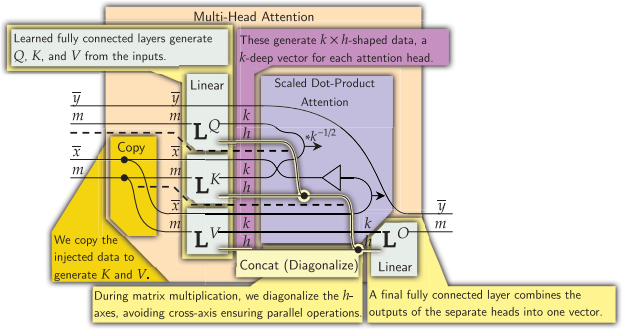
Figure 2.18: Scaled Dot-Product Attention




```
# Note, that we need to accomodate batches, hence the
... to capture additional axes.

# We can do the algorithm step by step,
def ScaledDotProductAttention(q: T, k: T, v: T) -> T:
    ''' yk, xk, xk -> yk '''
    klength = k.size()[-1]
    # Transpose
    k = einops.einsum(k,    '... x k -> ... k x')
    # Matrix Multiply / Inner Product
    x = einops.einsum(q, k, '... y k, ... k x -> ... y
x')
    # Scale
    x = x / math.sqrt(klength)
    # SoftMax
    x = torch.nn.Softmax(-1)(x)
    # Matrix Multiply / Inner Product
    x = einops.einsum(x, v, '... y x, ... x k -> ... y
k')
    return x
```

```
# Alternatively, we can simultaneously broadcast linear
functions.
def ScaledDotProductAttention(q: T, k: T, v: T) -> T:
    ''' yk, xk, xk -> yk '''
    klength = k.size()[-1]
    # Inner Product and Scale
    x = einops.einsum(q, k, '... y k, ... x k -> ... y
x')
    # Scale and SoftMax
    x = torch.nn.Softmax(-1)(x / math.sqrt(klength))
    # Final Inner Product
    x = einops.einsum(x, v, '... y x, ... x k -> ... y
k')
    return x
```

Figure 2.19: Multi-Head Attention



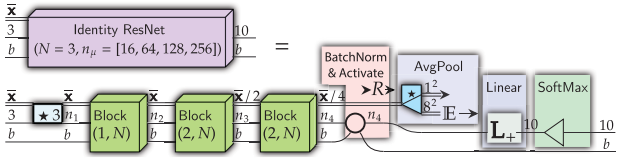
We will be implementing this algorithm. This shows us how we go from diagrams to implementations, and begins to give an idea of how organized diagrams leads to organized code.

```
def MultiHeadDotProductAttention(q: T, k: T, v: T) ->
T:
    ''' ykh, xkh, xkh -> ykh '''
    klength = k.size()[-2]
    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')
    return x

# We implement this component as a neural network
model.
# This is necessary when there are bold, learned
components that need to be initialized.
class MultiHeadAttention(nn.Module):
    # Multi-Head attention has various settings, which
become variables
    # for the initializer.
    def __init__(self, m, k, h):
        super().__init__()
        self.m, self.k, self.h = m, k, h
        # Set up all the boldface, learned components
        # Note how they bind axes we want to split,
which we do later with einops.
        self.Lq = nn.Linear(m, k*h, False)
        self.Lk = nn.Linear(m, k*h, False)
        self.Lv = nn.Linear(m, k*h, False)
        self.Lo = nn.Linear(k*h, m, False)


    # We have endogenous data (Eym) and external /
injected data (Xxm)
    def forward(self, Eym, Xxm):
        """ y m, x m -> y m """
        # We first generate query, key, and value
vectors.
        # Linear layers are automatically broadcast.

        # However, the k and h axes are bound. We
define an unbinder to handle the outputs,
        unbind = lambda x: einops.rearrange(x, '... (k
h)->... k h', h=self.h)
        q = unbind(self.Lq(Eym))
        k = unbind(self.Lk(Xxm))
        v = unbind(self.Lv(Xxm))

        # We feed q, k, and v to standard Multi-Head
inner product Attention
        o = MultiHeadDotProductAttention(q, k, v)

        # Rebind to feed to the final learned layer,
        o = einops.rearrange(o, '... k h-> ... (k h)',
h=self.h)
        return self.Lo(o)

# Now we can run it on fake data;
y, x, m, jc, heads = [20], [22], [128], [16], 4
# Internal Data
Eym = torch.rand(y + m)
# External Data
Xxm = torch.rand(x + m)

mha = MultiHeadAttention(m[0],jc[0],heads)
assert list(mha.forward(Eym, Xxm).size()) == y + m
```

2.3.4 Computer Vision

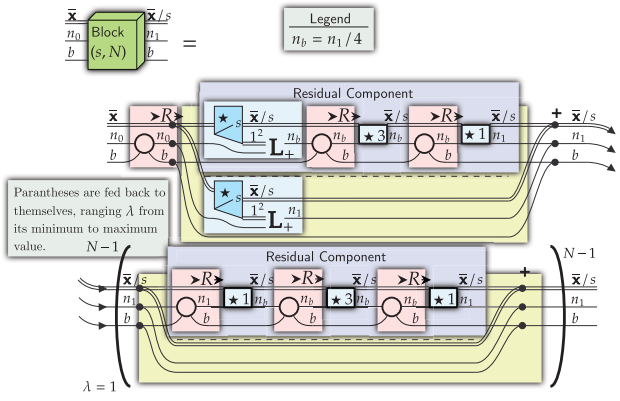
Here, we really start to understand why splitting diagrams into ``fenced off'' blocks aids implementation. In addition to making diagrams easier to understand and patterns more clearn, blocks indicate how code can structured and organized.

Figure 2.24: Identity Residual Network




```
# For Figure 2.24, every fenced off region is its own
module.

# Batch norm and then activate is a repeated motif,
class NormActivate(nn.Sequential):
    def __init__(self, nf, Norm=nn.BatchNorm2d,
Activation=nn.ReLU):
        super().__init__(Norm(nf), Activation())

def size_to_string(size):
    return " ".join(map(str,list(size)))

# The Identity ResNet block breaks down into a
manageable sequence of components.
class IdentityResNet(nn.Sequential):
    def __init__(self, N=3, n_mu=[16,64,128,256],
y=10):
        super().__init__(
            nn.Conv2d(3, n_mu[0], 3, padding=1),
            Block(1, N, n_mu[0], n_mu[1]),
            Block(2, N, n_mu[1], n_mu[2]),
            Block(2, N, n_mu[2], n_mu[3]),
            NormActivate(n_mu[3]),
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Linear(n_mu[3], y),
            nn.Softmax(-1),
            )
```

The Block can be defined in a seperate model, keeping the code manageable and closely connected to the diagram.

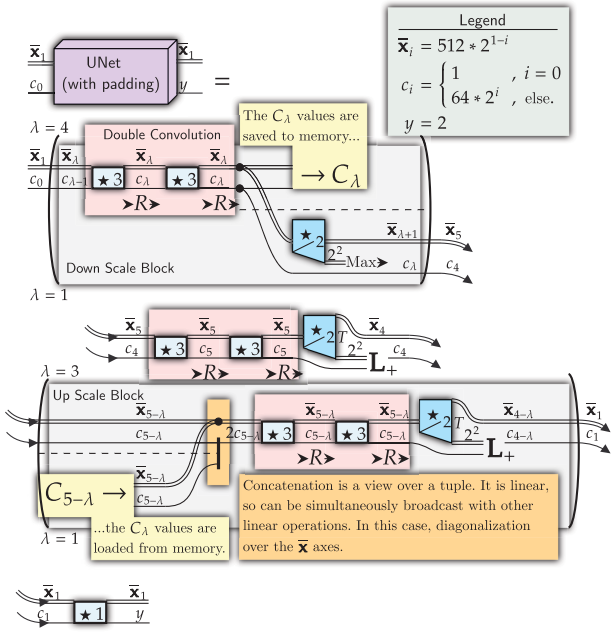



```
# We then follow how diagrams define each ``block''
class Block(nn.Sequential):
    def __init__(self, s, N, n0, n1):
        """ n0 and n1 as inputs to the initializer are
implicit from having them in the domain and codomain in
the diagram. """
        nb = n1 // 4
        super().__init__(
            *[
            NormActivate(n0),
            ResidualConnection(
                nn.Sequential(
                    nn.Conv2d(n0, nb, 1, s),
                    NormActivate(nb),
                    nn.Conv2d(nb, nb, 3, padding=1),
                    NormActivate(nb),
                    nn.Conv2d(nb, n1, 1),
                ),
                nn.Conv2d(n0, n1, 1, s),
            )
            ] + [
            ResidualConnection(
                nn.Sequential(
                    NormActivate(n1),
                    nn.Conv2d(n1, nb, 1),
                    NormActivate(nb),
                    nn.Conv2d(nb, nb, 3, padding=1),
                    NormActivate(nb),
                    nn.Conv2d(nb, n1, 1)
                ),
                )
            ] * N

            )
# Residual connections are a repeated pattern in the
diagram. So, we are motivated to encapsulate them
# as a seperate module.
class ResidualConnection(nn.Module):
    def __init__(self, mainline : nn.Module, connection
: nn.Module | None = None) -> None:
        super().__init__()
        self.main = mainline
        self.secondary = nn.Identity() if connection ==
None else connection
    def forward(self, x):
        return self.main(x) + self.secondary(x)
```

```
# A standard image processing algorithm has inputs
shaped b c h w.
b, c, hw = [3], [3], [16, 16]

idresnet = IdentityResNet()
Xbchw = torch.rand(b + c + hw)
# And we see if the overall size is maintained,
assert list(idresnet.forward(Xbchw).size()) == b + [10]
```

The UNet is a more complicated algorithm than residual networks. The ``fenced off'' sections help keep our code organized. Diagrams streamline implementation, and helps keep code organized.

Figure 2.25: The UNet architecture



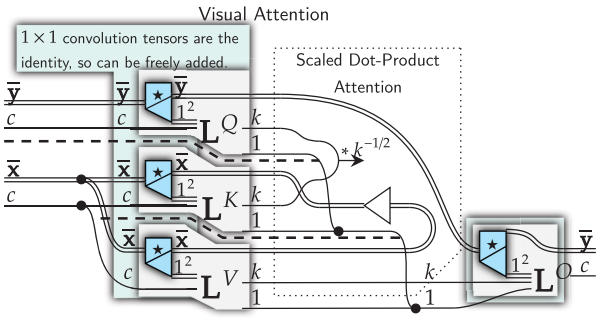

```
# We notice that double convolution where the numbers
of channels change is a repeated motif.
# We denote the input with c0 and output with c1.
# This can also be done for subsequent members of an
iteration.
# When we go down an iteration eg. 5, 4, etc. we may
have the input be c1 and the output c0.
class DoubleConvolution(nn.Sequential):
    def __init__(self, c0, c1, Activation=nn.ReLU):
        super().__init__(
            nn.Conv2d(c0, c1, 3, padding=1),
            Activation(),
            nn.Conv2d(c0, c1, 3, padding=1),
            Activation(),
            )

# The model is specified for a very specific number of
layers,
# so we will not make it very flexible.
class UNet(nn.Module):
    def __init__(self, y=2):
        super().__init__()
        # Set up the channel sizes;
        c = [1 if i == 0 else 64 * 2 ** i for i in
range(6)]

        # Saving and loading from memory means we can
not use a single,
        # sequential chain.

        # Set up and initialize the components;
        self.DownScaleBlocks = [
            DownScaleBlock(c[i],c[i+1])
            for i in range(0,4)
        ] # Note how this imitates the lambda operators
in the diagram.
        self.middleDoubleConvolution =
DoubleConvolution(c[4], c[5])
        self.middleUpscale = nn.ConvTranspose2d(c[5],
c[4], 2, 2, 1)
```

```
        self.upScaleBlocks = [
            UpScaleBlock(c[5-i],c[4-i])
            for i in range(1,4)
        ]
        self.finalConvolution = nn.Conv2d(c[1], y)

    def forward(self, x):
        cLambdas = []
        for dsb in self.DownScaleBlocks:
            x, cLambda = dsb(x)
            cLambdas.append(cLambda)
        x = self.middleDoubleConvolution(x)
        x = self.middleUpscale(x)
        for usb in self.upScaleBlocks:
            cLambda = cLambdas.pop()
            x = usb(x, cLambda)
        x = self.finalConvolution(x)

class DownScaleBlock(nn.Module):
    def __init__(self, c0, c1) -> None:
        super().__init__()
        self.doubleConvolution = DoubleConvolution(c0,
c1)
        self.downScaler = nn.MaxPool2d(2, 2, 1)
    def forward(self, x):
        cLambda = self.doubleConvolution(x)
        x = self.downScaler(cLambda)
        return x, cLambda

class UpScaleBlock(nn.Module):
    def __init__(self, c1, c0) -> None:
        super().__init__()
        self.doubleConvolution =
DoubleConvolution(2*c1, c1)
        self.upScaler = nn.ConvTranspose2d(c1,c0,2,2,1)
    def forward(self, x, cLambda):
        # Concatenation occurs over the C channel axis
(dim=1)
        x = torch.concat(x, cLambda, 1)
        x = self.doubleConvolution(x)
        x = self.upScaler(x)
        return x
```

2.3.5 Vision Transformer

We adapt our code for Multi-Head Attention to apply it to the vision case. This is a good exercise in how neural circuit diagrams allow code to be easily adapted for new modalities.

Figure 2.26: Visual Attention




```
class VisualAttention(nn.Module):
    def __init__(self, c, k, heads = 1, kernel = 1,
stride = 1):
        super().__init__()

        # w gives the kernel size, which we make
adjustable.
        self.c, self.k, self.h, self.w = c, k, heads,
kernel
        # Set up all the boldface, learned components
        # Note how standard components may not have
axes bound in
        # the same way as diagrams. This requires us to
rearrange
        # using the einops package.

        # The learned layers form convolutions
        self.Cq = nn.Conv2d(c, k * heads, kernel,
stride)
        self.Ck = nn.Conv2d(c, k * heads, kernel,
stride)
        self.Cv = nn.Conv2d(c, k * heads, kernel,
stride)
        self.Co = nn.ConvTranspose2d(
                            k * heads, c, kernel,
stride)

    # Defined previously, closely follows the diagram.
    def MultiHeadDotProductAttention(self, q: T, k: T,
v: T) -> T:
        ''' ykh, xkh, xkh -> ykh '''
        klength = k.size()[-2]
        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')
        return x

    # We have endogenous data (EYc) and external /
injected data (XXc)
    def forward(self, EcY, XcX):
        """ cY, cX -> cY
        The visual attention algorithm. Injects
information from Xc into Yc. """
        # query, key, and value vectors.
        # We unbind the k h axes which were produced by
the convolutions, and feed them
        # in the normal manner to
MultiHeadDotProductAttention.
        unbind = lambda x: einops.rearrange(x, 'N (k h)
H W -> N (H W) k h', h=self.h)
        # Save size to recover it later
        q = self.Cq(EcY)
        W = q.size()[-1]

        # By appropriately managing the axes, minimal
changes to our previous code
        # is necessary.
        q = unbind(q)
        k = unbind(self.Ck(XcX))
        v = unbind(self.Cv(XcX))
        o = self.MultiHeadDotProductAttention(q, k, v)

        # Rebind to feed to the transposed convolution
layer.
        o = einops.rearrange(o, 'N (H W) k h -> N (k h)
H W',
                                h=self.h, W=W)
        return self.Co(o)

# Single batch element,
b = [1]
Y, X, c, k = [16, 16], [16, 16], [33], 8
# The additional configurations,
heads, kernel, stride = 4, 3, 3

# Internal Data,
EYc = torch.rand(b + c + Y)
# External Data,
XXc = torch.rand(b + c + X)

# We can now run the algorithm,
visualAttention = VisualAttention(c[0], k, heads,
kernel, stride)
```

```
# Interestingly, the height/width reduces by 1 for
stride
# values above 1. Otherwise, it stays the same.
visualAttention.forward(EYc, XXc).size()
```

Appendix

```
# A container to track the size of modules,
# Replace a module definition eg.
# > self.Cq = nn.Conv2d(c, k * heads, kernel, stride)
# With;
# > self.Cq = Tracker(nn.Conv2d(c, k * heads, kernel,
stride), "Query convolution")
# And the input / output sizes (to check diagrams) will
be printed.
class Tracker(nn.Module):
    def __init__(self, module: nn.Module, name : str =
""):
        super().__init__()
        self.module = module
        if name:
            self.name = name
        else:
            self.name = self.module._get_name()
    def forward(self, x):
        x_size = size_to_string(x.size())
        x = self.module.forward(x)
        y_size = size_to_string(x.size())
        print(f"{self.name}: \t {x_size} -> {y_size}")
        return x
```

Bibliography

Abramsky, S. and Tzevelekos, N., 2010. Introduction to categories and categorical logic. vol. 813, 3–94. doi:10.1007/978-3-642-12821-9_1. http://arxiv.org/abs/1102.1313. [Cited on page 39.]

Awodey, S., 2010. *Category Theory*. Oxford University Press, Inc., 2nd edn. ISBN 978-0-19-923718-0. [Cited on pages 5, 45, 50, 57, and 59.]

Ba, J. L.; Kiros, J. R.; and Hinton, G. E., 2016. Layer normalization. doi:10.48550/arXiv.1607.06450. http://arxiv.org/abs/1607.06450. [Cited on page 1.]

Baez, J. C. and Stay, M., 2010. Physics, topology, logic and computation: A rosetta stone. vol. 813, 95–172. doi:10.1007/978-3-642-12821-9_2. http://arxiv.org/abs/0903.0340. [Cited on pages 5, 12, 18, and 60.]

Biamonte, J. and Bergholm, V., 2017. Tensor networks in a nutshell. doi:10.48550/arXiv.1708.00006. http://arxiv.org/abs/1708.00006. [Cited on page 5.]

Borkin, M. A.; Bylinskii, Z.; Kim, N. W.; Bainbridge, C. M.; Yeh, C. S.; Borkin, D.; Pfister, H.; and Oliva, A., 2016. Beyond memorability: Visualization recognition and recall. 22, 1 (01 2016), 519–528. doi:10.1109/TVCG.2015.2467732. Conference Name: IEEE Transactions on Visualization and Computer Graphics. [Cited on pages 4 and 22.]

Braithwaite, D. and Román, M., 2023. Collages of string diagrams. doi:10.48550/arXiv.2305.02675. http://arxiv.org/abs/2305.02675. [Cited on page 6.]

Chiang, D.; Rush, A. M.; and Barak, B., 2023. Named tensor notation. http://arxiv.org/abs/2102.13196. [Cited on pages 1, 3, 4, 5, 8, 9, 18, and 62.]

Cockett, R.; Cruttwell, G.; Gallagher, J.; Lemay, J.-S. P.; MacAdam, B.; Plotkin, G.; and Pronk, D., 2019. Reverse derivative categories. doi:10.48550/arXiv.1910.07065. http://arxiv.org/abs/1910.07065. [Cited on pages 31 and 32.]

Cruttwell, G. S. H.; Gavranović, B.; Ghani, N.; Wilson, P.; and Zanasi, F., 2021. Categorical foundations of gradient-based learning. doi:10.48550/arXiv.2103.01931. http://arxiv.org/abs/2103.01931. [Cited on pages 5, 6, 10, 35, 71, 79, and 82.]

Dehghani, M.; Mustafa, B.; Djolonga, J.; Heek, J.; Minderer, M.; Caron, M.; Steiner, A.; Puigcerver, J.; Geirhos, R.; Alabdulmohsin, I.; Oliver, A.; Padlewski, P.; Gritsenko, A.; Lučić, M.; and Houlsby, N., 2023. Patch n' pack: NaViT, a vision transformer for any aspect ratio and resolution. doi:10.48550/arXiv.2307.06304. http://arxiv.org/abs/2307.06304. [Cited on page 28.]

Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Dehghani, M.; Minderer, M.; Heigold, G.; Gelly, S.; Uszkoreit, J.; and Houlsby, N., 2021. An image is worth 16x16 words: Transformers for image recognition at scale. doi:10.48550/arXiv.2010.11929. http://arxiv.org/abs/2010.11929. [Cited on pages 25 and 30.]

Drummond, C., 2009. Replicability is not reproducibility: Nor is it good science. (01 2009). [Cited on page 2.]

Fong, B. and Spivak, D. I., 2019. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press, 1 edn. ISBN 978-1-108-66880-4 978-1-108-48229-5 978-1-108-71182-1. doi:10.1017/9781108668804. https://www.cambridge.org/core/product/identifier/9781108668804/type/book. [Cited on page 5.]

Fong, B.; Spivak, D. I.; and Tuyéras, R., 2019. Backprop as functor: A compositional perspective on supervised learning. doi:10.48550/arXiv.1711.10455. http://arxiv.org/abs/1711.10455. [Cited on pages 5, 6, 7, 10, 35, 71, 79, and 82.]

Fritz, T.; Gonda, T.; Perrone, P.; and Rischel, E. F., 2023. Representable markov categories and comparison of statistical experiments in categorical probability. 961 (06 2023), 113896. doi:10.1016/j.tcs.2023.113896. http://arxiv.org/abs/2010.07416. [Cited on pages 6, 10, 59, and 60.]

Fritz, T. and Rischel, E. F., 2020. Infinite products and zero-one laws in categorical probability. 2 (08 2020), 3. doi:10.32408/compositionality-2-3. http://arxiv.org/abs/1912.02769. [Cited on pages 10 and 82.]

Goodfellow, I.; Bengio, Y.; Courville, A.; and Bengio, Y., 2016. *Deep learning*, vol. 1. MIT Press. [Cited on page 1.]

Hayes, J. and Bajzek, D., 2008. Understanding and reducing the knowledge effect: Implications for writers. 25 (01 2008), 104–118. [Cited on page 4.]

He, K.; Zhang, X.; Ren, S.; and Sun, J., 2015. Deep residual learning for image recognition. http://arxiv.org/abs/1512.03385. [Cited on pages 1 and 5.]

He, K.; Zhang, X.; Ren, S.; and Sun, J., 2016. Identity mappings in deep residual networks. doi:10.48550/arXiv.1603.05027. http://arxiv.org/abs/1603.05027. [Cited on pages 1, 5, 6, 8, 28, 29, 31, and 82.]

Heunen, C. and Vicary, J., 2012. Lectures on categorical quantum mechanics. (2012). [Cited on page 60.]

Ho, J.; Jain, A.; and Abbeel, P., 2020. Denoising diffusion probabilistic models. http://arxiv.org/abs/2006.11239. [Cited on pages 1 and 81.]

Ioffe, S. and Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. http://arxiv.org/abs/1502.03167. [Cited on page 1.]

Kapoor, S. and Narayanan, A., 2022. Leakage and the reproducibility crisis in ML-based science. doi:10.48550/arXiv.2207.07048. http://arxiv.org/abs/2207.07048. [Cited on page 2.]

Khan, S.; Naseer, M.; Hayat, M.; Zamir, S. W.; Khan, F. S.; and Shah, M., 2022. Transformers in vision: A survey. 54, 10 (01 2022), 1–41. doi:10.1145/3505244. https://dl.acm.org/doi/10.1145/3505244. [Cited on pages 25, 28, and 30.]

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E., 2017. ImageNet classification with deep convolutional neural networks. 60, 6 (05 2017), 84–90. doi:10.1145/3065386. https://dl.acm.org/doi/10.1145/3065386. [Cited on pages 1, 22, and 28.]

Lee, M., 2023. GELU activation function in deep learning: A comprehensive mathematical analysis and performance. doi:10.48550/arXiv.2305.12073. http://arxiv.org/abs/2305.12073. [Cited on pages 1 and 22.]

Li, Y.; Luo, T.; and Yip, N. K., 2022. Towards an understanding of residual networks using neural tangent hierarchy (NTH). 3, 4 (06 2022), 692–760. doi:10.4208/csiam-am.SO-2021-0053. http://arxiv.org/abs/2007.03714. [Cited on pages 7 and 10.]

Lin, T.; Wang, Y.; Liu, X.; and Qiu, X., 2021. A survey of transformers. doi:10.48550/arXiv.2106.04554. http://arxiv.org/abs/2106.04554. [Cited on pages 2 and 77.]

Liu, H.; Dai, Z.; So, D. R.; and Le, Q. V., 2021. Pay attention to MLPs. doi:10.48550/arXiv.2105.08050. http://arxiv.org/abs/2105.08050. [Cited on page 1.]

Luo, W.; Li, Y.; Urtasun, R.; and Zemel, R., 2017. Understanding the effective receptive field in deep convolutional neural networks. https://arxiv.org/abs/1701.04128v2. [Cited on pages 1 and 28.]

Marsden, D., 2014. Category theory using string diagrams. https://arxiv.org/abs/1401.7220v2. [Cited on pages 6, 8, 9, 37, 45, 54, 79, and 83.]

Meseguer, J. and Montanari, U., 1990. Petri nets are monoids. 88, 2 (10 1990), 105–155. doi:10.1016/0890-5401(90)90013-8. https://www.sciencedirect.com/science/article/pii/0890540190900138. [Cited on page 5.]

Murata, T., 1989. Petri nets: Properties, analysis and applications. 77, 4 (04 1989), 541–580. doi:10.1109/5.24143. Conference Name: Proceedings of the IEEE. [Cited on page 5.]

Nakahira, K., 2023. Diagrammatic category theory. doi:10.48550/arXiv.2307.08891. http://arxiv.org/abs/2307.08891. [Cited on pages 6, 8, 9, 37, 45, 50, 54, 79, and 83.]

Nichol, A. and Dhariwal, P., 2021. Improved denoising diffusion probabilistic models. http://arxiv.org/abs/2102.09672. [Cited on pages 1 and 81.]

Ochs, E., 2020. On my favorite conventions for drawing the missing diagrams in category theory. doi:10.48550/arXiv.2006.15836. http://arxiv.org/abs/2006.15836. [Cited on page 50.]

Perrone, P., 2022. Markov categories and entropy. http://arxiv.org/abs/2212.11719. [Cited on pages 6, 10, 35, 71, and 83.]

Phuong, M. and Hutter, M., 2022. Formal algorithms for transformers. doi:10.48550/arXiv.2207.09238. http://arxiv.org/abs/2207.09238. [Cited on pages 2, 3, 4, and 8.]

Piedeleu, R. and Zanasi, F., 2023. An introduction to string diagrams for computer scientists. doi:10.48550/arXiv.2305.08768. http://arxiv.org/abs/2305.08768. [Cited on pages 6, 7, and 71.]

Pinker, S., 2014. *The sense of style: The thinking person's guide to writing in the 21st century*. Penguin Publishing Group. ISBN 978-0-698-17030-8. https://books.google.com.au/books?id=FzRBAwAAQBAJ. [Cited on page 4.]

Raff, E., 2019. A step toward quantifying independently reproducible machine learning research. In *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/hash/c429429bf1f2af051f2021dc92a8ebea-Abstract.html. [Cited on page 1.]

Rogozhnikov, A., 2021. Einops: Clear and reliable tensor manipulations with einstein-like notation. https://openreview.net/forum?id=oapKSVM2bcj. [Cited on page 23.]

Rombach, R.; Blattmann, A.; Lorenz, D.; Esser, P.; and Ommer, B., 2022. High-resolution image synthesis with latent diffusion models. http://arxiv.org/abs/2112.10752. [Cited on pages 1 and 29.]

Román, M., 2021. Open diagrams via coend calculus. 333 (02 2021), 65–78. doi:10.4204/EPTCS.333.5. http://arxiv.org/abs/2004.04526. [Cited on page 6.]

Ronneberger, O.; Fischer, P.; and Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. doi:10.48550/arXiv.1505.04597. http://arxiv.org/abs/1505.04597. [Cited on pages 1 and 29.]

Ross, L.; Greene, D.; and House, P., 1977. The "false consensus effect": An egocentric bias in social perception and attribution processes. 13, 3 (1977), 279–301. [Cited on page 4.]

Sadoski, 1993. Impact of concreteness on comprehensibility, interest. 85, 2 (1993), 291–304. [Cited on page 4.]

Saxe, A. M.; Bansal, Y.; Dapello, J.; Advani, M.; Kolchinsky, A.; Tracey, B. D.; and Cox, D. D., 2019. On the information bottleneck theory of deep learning. 2019, 12 (12 2019), 124020. doi:10.1088/1742-5468/ab3985. https://iopscience.iop.org/article/10.1088/1742-5468/ab3985. [Cited on pages 7, 10, 79, and 82.]

Selinger, P., 2009. A survey of graphical languages for monoidal categories. doi:10.1007/978-3-642-12821-9_4. https://arxiv.org/abs/0908.3347v1. [Cited on pages 5, 6, 43, 54, 59, 61, and 79.]

Shiebler, D.; Gavranović, B.; and Wilson, P., 2021. Category theory in machine learning. doi:10.48550/arXiv.2106.07032. http://arxiv.org/abs/2106.07032. [Cited on pages 5, 6, 31, 71, 79, and 82.]

Srivastava, R. K.; Greff, K.; and Schmidhuber, J., 2015. Highway networks. http://arxiv.org/abs/1505.00387. [Cited on page 1.]

Sun, Y.; Dong, L.; Huang, S.; Ma, S.; Xia, Y.; Xue, J.; Wang, J.; and Wei, F., 2023. Retentive network: A successor to transformer for large language models. doi:10.48550/arXiv.2307.08621. http://arxiv.org/abs/2307.08621. [Cited on page 1.]

Tishby, N.; Pereira, F. C.; and Bialek, W., 2000. The information bottleneck method. http://arxiv.org/abs/physics/0004057. [Cited on page 82.]

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I., 2017. Attention is all you need. http://arxiv.org/abs/1706.03762. [Cited on pages 1, 2, 22, 24, and 77.]

Wilson, P. and Zanasi, F., 2022. Categories of differentiable polynomial circuits for machine learning. doi:10.48550/arXiv.2203.06430. http://arxiv.org/abs/2203.06430. [Cited on page 5.]

Xu, T. and Maruyama, Y., 2022. Neural string diagrams: A universal modelling language for categorical deep learning. In *Artificial General Intelligence*, Lecture Notes in Computer Science (Cham, 2022), 306–315. Springer International Publishing. doi:10.1007/978-3-030-93758-4_32. [Cited on pages 4, 5, and 31.]

Xu, Y. L.; Konstantinidis, K.; and Mandic, D. P., 2023. Graph tensor networks: An intuitive framework for designing large-scale neural learning systems on multiple domains. doi:10.48550/arXiv.2303.13565. http://arxiv.org/abs/2303.13565. [Cited on pages 1, 4, 5, 8, 18, 21, and 81.]

Zeiler, M. D.; Krishnan, D.; Taylor, G. W.; and Fergus, R., 2010. Deconvolutional networks. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (San Francisco, CA, USA, 06 2010), 2528–2535. IEEE. doi: 10.1109/CVPR.2010.5539957. http://ieeexplore.ieee.org/document/5539957/. [Cited on page 25.]

Zhang, C.; Bengio, S.; Hardt, M.; Recht, B.; and Vinyals, O., 2017. Understanding deep learning requires rethinking generalization. http://arxiv.org/abs/1611.03530. [Cited on pages 7 and 10.]