

Categories as a Foundation for both Learning and Reasoning

by

Nolan Peter Shaw

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Nolan Peter Shaw 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Geoff Cruttwell
Associate Professor
Dept. of Mathematics and Computer Science,
Mount Allison University

Supervisor(s): Jeff Orchard
Associate Professor
Cheriton School of Computer Science,
University of Waterloo

Internal Member: Richard Treffler
Associate Professor
Cheriton School of Computer Science,
University of Waterloo

Internal Member: Grant Weddell
Associate Professor
Cheriton School of Computer Science,
University of Waterloo

Internal-External Member: Britt Anderson
Associate Professor
Dept. of Psychology,
University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis explores two distinct research topics, both applying category theory to machine learning.

The first topic discusses Vector Symbolic Architectures (VSAs). I present the first attempt at formalising VSAs with category theory. VSAs are built to perform symbolic reasoning in high-dimensional vector spaces. I present a brief literature survey demonstrating that the topic is currently completely unexplored. I discuss some desiderata for VSA models, then describe an initial formalisation that covers two of the three desiderata. My formalisation focuses on two of the three primary components of a VSA: binding and bundling, and presents a proof of why element-wise operations constitute the ideal means of performing binding and bundling. The work extends beyond vectors, to any co-presheaves with the desired properties. For example, GHRR representations are captured by this generalisation.

The second line of work discusses, and expands upon, recent work by Milewski in the construction of “pre-lenses.” This work is motivated by pre-established formalisations of supervised machine learning. From the perspective of category theory, pre-lenses are interesting because they unify the category **Para**, or **Learn**, with its dual **co-Para**, or **Learn^{op}**. From a computer science perspective, pre-lenses are interesting because they enable programmers to build neural networks with vanilla function composition, and they unify various network features by leveraging the fact that they are profunctors. I replicate Milewski’s code, extend it to the non-synthetic data, MNIST, implement re-parameterisations, and describe generative models as dual to discriminative models by way of pre-lenses. This work involved creating a simple dataloader to read in external files, randomising the order that inputs are presented during learning, and fixing some bugs that didn’t manifest when training occurred on the very small dataset used by Milewski.

Acknowledgements

I would like to thank the following people:

- Clifton Cunningham for first introducing me to category theory.
- Britt Anderson for convincing me to switch research topics when I was feeling quite lost and discouraged in my work.
- My supervisor, Jeff Orchard, who has been endlessly supportive, and the model of how I wish to live as an academic.
- Bartosz Milewski for being a fantastic educator and mentor.
- Michael Furlong, who possesses an abundance of both competence and kindness.
- The broader applied category theory and passionate math communities. In particular, Nathan Hayden and Priyaa Varshinee Srinivasan.
- My family. First and foremost, my spouse, Katie Pita, whose constant love has made this thesis possible and enjoyable. Also, my parents, Gail, Patrick, and Sophia, who have only ever shown confidence in me and my work. Also, our cat, Harriet.
- My friends and “found family.” There are too many to list, but I’m particularly grateful for Ben Anderson-Sackaney, Vijay Bhattiprolu, Matthew Chiang, Gabrielle Geenen, Ryan Hancock, Neil Hester, Adam Humeniuk, Josh Jung, Nicole Kitt, Paul Lawrence, Tim Miller, Joaco Prandi, Hayley Reid, John Sawatzky, Jenny Wang, and Di Xiao.

Territorial Acknowledgement

This thesis was completed on land known as the Haldimand Tract. This region is the traditional home of the Neutral, Anishinaabeg, and Haudenosaunee people. During my studies, my social and political philosophies have changed as much as my mathematical perspectives, due to the work, knowledge, and friendship of Indigenous people in Kitchener and Waterloo. In particular, groups such as the various “Land Back” collectives have had a deep impact on my views regarding land, labour, equity, community, and governance¹.

I’ve seen deep inconsistencies in how Truth and Reconciliation is treated institutionally. To date, the University of Waterloo continues to operate on this land. While it coordinates with the Office of Indigenous Relations, achieving non-trivial steps towards improving the well-being of some Indigenous community members, it nevertheless exercises its land rights in various ways that rely on its sole title to the land—a title that I consider inconsistent with acknowledgements of its rightful, ancestral keepers. Further, I’ve seen how the university resists any meaningful steps towards Truth and Reconciliation through collective bargaining. Nevertheless, the university is composed of people who are actively working towards Reconciliation, and I remain optimistic that this living process will continue to improve. I am also encouraged by actions such as waiving tuition for members of the Mississaugas of the Credit or Six Nations of the Grand River (along with other tuition reductions for out-of-province Indigenous students).

While it may not happen in my lifetime, I hope that Truth and Reconciliation develops towards a more comprehensive picture of what we want our collective utopia to be. Land acknowledgements are a first step, but I hope that we achieve more material improvements. That we decolonise our ideas of land “ownership” and move towards universal expropriation, recognising that land is meant to be shared. That Indigenous people can increasingly exercise opportunities to be neighbours, leaders, and knowledge keepers. That we see the growth of new Indigenous communities, rather than exclusively inviting Indigenous people to become parts of institutions that were built as, and largely remain, colonial institutions. Most relevantly to academia, I hope that we see a shift away from the increasingly exclusive funding models of post-secondary education, and move towards a university that is truly built by, and to the benefit of, the commons. Most immediately, in the Haldimand Tract, I hope that the Neutral, Anishinaabeg, and Haudenosaunee people can freely use the land that is their home.

Personally, I plan to advance Reconciliation primarily through education. I hope to contribute to closing educational gaps between Indigenous and non-Indigenous people. This work

¹My mathematical perspectives have also been shaped by scholars such as Leeroy Little Bear, who discusses dynamic/verb-centered metaphysics as opposed to static/noun-centered ones—clearly appealing to someone studying category theory. “Systems thinking,” practiced by non-Indigenous folks, but immediately natural in Indigenous stories and worldviews, also holds obvious value to the category theorist.

will occur on my past and future home of Treaty 7 land, though I want to develop open-access educational resources for students more broadly. I also hope to push my home institution to change its funding model from providing scholarships to Indigenous student towards waiving tuition for those students, as this seems to be a more equitable system. I also wish to decolonise the means by which education is provided, incorporating means of learning and assessment that suit the broader ways in which people learn and demonstrate their knowledge. Part of this will require debunking the view that mathematics is only for the “elite.” I firmly believe that mathematics is for everyone, and have seen how few Indigenous people have had the chance to become mathematicians.

Dedication

This thesis is dedicated to our union, CUPE 5524, where I met my amazing spouse, Katie. I hope it continues to be a strong, growing community that fights for the well-being of graduate students at Waterloo. Graduate students do important work; they deserve security and respect.

Table of Contents

| | |
|-------------------------------------------------------------------------------------------------------|-------------|
| Examining Committee Membership | ii |
| Author’s Declaration | iii |
| Abstract | iv |
| Acknowledgements | v |
| Territorial Acknowledgement | vi |
| Dedication | viii |
| List of Figures | xiv |
| List of Tables | xvii |
| 1 Introduction | 1 |
| 1.1 Motivation | 2 |
| 1.2 Summary of Contributions | 4 |
| 1.2.1 Formalising Vector Symbolic Architectures | 5 |
| 1.2.2 Presenting “Pre-lenses” as a Generalisation of Supervised Learning Ar- chitectures | 5 |
| 1.3 Conventions and Notation | 6 |

| | | |
|----------|-------------------------------------------------------------------|-----------|
| 2 | Background | 9 |
| 2.1 | Neural Networks | 9 |
| 2.2 | Backpropagation | 11 |
| 2.2.1 | Other Error-propagating Algorithms | 12 |
| 2.2.2 | Learning Pipeline | 12 |
| 2.3 | Modern Deep Learning | 13 |
| 2.3.1 | Local Variations | 14 |
| 2.3.2 | Structural Variations | 17 |
| 2.3.3 | Closing Remarks on Machine Learning | 22 |
| 2.4 | Category Theory | 22 |
| 2.4.1 | What is a Category? | 22 |
| 2.4.2 | Duality | 24 |
| 2.4.3 | Universal Properties | 25 |
| 2.4.4 | Functors | 27 |
| 2.4.5 | Natural Transformations | 29 |
| 2.4.6 | 2-Categories and Bicategories | 30 |
| 2.4.7 | Monoids and Monoidal Categories | 31 |
| 2.5 | Applied Category Theory | 32 |
| 2.5.1 | Algebraic Data Types | 33 |
| 2.5.2 | Cryptography | 34 |
| 2.5.3 | Databases | 36 |
| 3 | A Categorical Foundation for Vector Symbolic Architectures | 38 |
| 3.1 | Introduction | 38 |
| 3.2 | Background | 39 |
| 3.2.1 | VSA Definition | 40 |
| 3.2.2 | VSA Operations | 41 |
| 3.2.3 | Survey of Commonly Used VSAs | 42 |

| | | |
|----------|----------------------------------------------------------------------------------------------------|-----------|
| 3.2.4 | Example Problem: Composing Functions | 44 |
| 3.3 | Desiderata | 44 |
| 3.4 | Formalising VSAs Using Category Theory | 45 |
| 3.4.1 | Generalising to Co-presheaves | 46 |
| 3.4.2 | External Tensor Product and Sum | 47 |
| 3.4.3 | Kan Extensions | 47 |
| 3.4.4 | Examples | 51 |
| 3.5 | Discussion and Future Work | 52 |
| 4 | How Pre-lenses Reconcile Backpropagation and Autodifferentiation (and Other Symmetries) | 54 |
| 4.1 | Introduction | 54 |
| 4.2 | Background | 55 |
| 4.2.1 | Autodifferentiation | 55 |
| 4.2.2 | Parametric Lenses | 57 |
| 4.2.3 | Lenses in Related Domains | 60 |
| 4.2.4 | Profunctors | 61 |
| 4.2.5 | Tambara Modules | 64 |
| 4.2.6 | Existential Lenses | 66 |
| 4.3 | Pre-lenses | 67 |
| 4.3.1 | Tri-Profunctors | 71 |
| 4.3.2 | Triple Tambara Modules | 72 |
| 4.3.3 | Tri-lenses | 74 |
| 4.4 | Building an ANN with Pre-Lenses | 75 |
| 4.4.1 | Single Neurons | 75 |
| 4.4.2 | Layers | 76 |
| 4.4.3 | Batching | 78 |
| 4.4.4 | Re-parametrisations | 78 |

| | | |
|----------|--------------------------------------------------------------|------------|
| 4.5 | Running on MNIST | 79 |
| 4.5.1 | Changes Made to Existing Model | 79 |
| 4.5.2 | Details of Training | 80 |
| 4.5.3 | Changes Needed in Future Work | 80 |
| 4.6 | Generative Models | 82 |
| 4.7 | Discussion and Future Work | 83 |
| 5 | Conclusion | 85 |
| 5.1 | Closing Summary on VSAs | 85 |
| 5.2 | Closing Summary on Pre-lenses | 86 |
| 5.3 | Speculations | 86 |
| 5.3.1 | Attention as Another Axis for Pre-Lens Composition | 86 |
| 5.3.2 | Modelling Predictive Coding with Pre-lenses | 86 |
| 5.3.3 | Unifying Symbolic Reasoning and Learning | 87 |
| 5.4 | Closing Statement | 88 |
| | References | 89 |
| | APPENDICES | 104 |
| A | A Critique of Machine Learning Research | 105 |
| A.1 | Methodological Issues | 105 |
| A.1.1 | Systemic P-Hacking | 105 |
| A.1.2 | Replication | 107 |
| A.2 | Institutional Issues | 107 |
| A.2.1 | Obsession with Performance | 108 |
| A.2.2 | Strength of Claims | 109 |
| A.2.3 | Transparency | 109 |
| A.3 | Moral Issues | 110 |
| A.4 | Societal Issues | 111 |

| | | |
|----------|------------------------------------------------------------------------------|------------|
| B | Other Paradigms of Machine Learning | 113 |
| B.1 | Unsupervised Learning | 113 |
| B.2 | Reinforcement Learning | 115 |
| C | Literature Survey of Category Theory for VSAs | 117 |
| D | Additional Examples of VSA Algorithms | 119 |
| D.1 | Constructing and Deconstructing Tuples | 119 |
| D.2 | An Argument for Commutative Binding: Representing Numbers | 120 |
| D.3 | An Argument Against Commutative Binding: Composing Data Structures | 120 |
| E | Verifying that Lenses are a Profunctor by Type-checking | 122 |

List of Figures

| | | |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1 | A visual comparison of traditional computation (top) versus machine learning (bottom). When represented this way, it is easier to see that machine learning isn't really some "better" way to build algorithms—both models are kind of like an algebraic equation solving for one unknown, and it is simply which unknown that varies. | 10 |
| 2.2 | An illustration of a typical recurrent neural network (top), and how it can be "unfolded" to produce a completely acyclic representation (bottom). Here, n represents the final value of t . Since gradients depend on upstream errors, parameter updates early in training sequences can become exponentially small if activities in the hidden layer are small for large values of t | 17 |
| 2.3 | An illustration of a typical convolutional layer. The blue box on the left represents a given receptive field, while the box on the right represents the corresponding target neurons in the subsequent layer. The depth of the box on the right represents that there can be multiple nodes that map from this receptive field. | 19 |
| 2.4 | The general structure of a residual connection in a neural network. Here, $a^{(i)}$ is the input, while f is the computation performed by a given layer (or layers) of a neural network. The hooked arrows represent the inclusion mapping to some product that is then reduced using a given \oplus binary operator. Typically, this operator is simple addition, however other choices, such as concatenation, are possible as well. The final object is $a^{(j)}$ —the input to subsequent layers. | 20 |
| 2.5 | A simple, three object category. All arrows are made explicit in the left diagram. Conventionally, identity morphisms are omitted to reduce clutter, resulting in the right diagram. | 23 |
| 2.6 | A picture of how all objects in Set map <i>uniquely</i> to any singleton, $*$. The other objects are kept as unlabelled dots in this diagram to emphasise that the sizes and elements of these sets are irrelevant. | 26 |

| | | |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.7 | Two examples of a product, categorically. On the left is the Cartesian product of sets A and B , written as $A \times B$. The Cartesian product is defined as the set with functions, π_0 and π_1 , to A and B respectively (think of these as the “projections” from $A \times B$ to each part of the product) such that any object, X that has morphisms f and g mapping to A and B respectively, maps uniquely through $!$ such that $!; \pi_0 = f$ and $!; \pi_1 = g$. On the right, we have a product from the category with Natural numbers as objects and morphisms representing the “divides” relation. Here, every hom-set contains one morphism at most, so the uniqueness of $!$ is guaranteed. The diagram expresses that any number, x , which divides both a and b must also divide $\text{gcd}(a, b)$ | 27 |
| 2.8 | Various properties of how 2-cells interact in 2-categories. Technically, it’s redundant to define both horizontal composition and the pair of whiskering operations, because one can be derived from the other. However, it is useful to visually see how whiskering is the result of “collapsing” either side of a horizontal composition (technically composing with an identity on either side). | 31 |
| 4.1 | A representation of a single network layer that uses autodifferentiation. Subsequent network layers will have an identical shape and fit together by horizontally composing them. Note how this means that da from a subsequent layer “fulfills the promise” that there will be an input to db in the current layer, while simultaneously creating a new requested input upstream. | 56 |
| 4.2 | String diagram of morphisms and composition in $\text{Lens}(C)$. The upper left diagram represents the morphism, (f, f^*) , which is then “unboxed” in the upper right diagram. The bottom diagram illustrates how two morphisms compose. Note that this visualisation of a lens might look a bit different from the one described when talking about databases. Nevertheless, it is very much the case that these two things are equivalent! | 59 |
| 4.3 | A commuting diagram demonstrating the relationship between all the inputs, a , a' , b , and b' ; f and g ; P , and; dimap . In the case where P is \rightarrow , we can simply remove the labels. What I love about this diagram is that it also explains why we are contravariant in our input. Remember with functors that fmap is only needed because our compiler needs to distinguish between how we map objects and how we map morphisms. The same is true for dimap and if we reuse P we can see that $P \circ f = g$ would hold. For the arrow, \rightarrow , this composition wouldn’t type-check if f were the opposite direction (<i>i.e.</i> if we were covariant in a). | 64 |

| | | |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 4.4 | A comparison of classical, get/set lenses compared to existential lenses. It's important to note that m is quantified over <i>all</i> possible values, meaning that it is “internal” to the lens in a way. Also notice that setting $m := a$ recovers the classic lens representation. | 66 |
| 4.5 | From existential lenses in 4.5a, we can add parameters as in 4.5b. At this point, an almost-symmetry is obvious in the Type signature of our lens. All inputs and outputs of f , have a partner in f^* , except m which is present in both. The idea of pre-lenses is very simple: simply break m up into a pair m and dm . The result is 4.5c. Note that this means we no longer quantify over m | 68 |
| 4.6 | Using the same jigsaw representation as Figure 4.1, I show how pre-lenses highlight the known equivalence between learning via traditional backprop and learning with autodifferentiation. In the top diagram, the top half is meant to be computed first, with the outward puzzle inserts corresponding to the messages, m , that pre-lenses leave for the backwards pass. The rightmost “end-piece” is the terminal parametric lens in [24]. In contrast, the bottom diagram builds layers in lock-step. Also note how, though they are meant to be separate diagrams, each dp_i can connect with a new p_i . This vertical tiling corresponds to learning, and will be discussed more in Section 4.6. | 70 |
| 4.7 | An example input to the network, along with the classification made by the network after training. | 81 |
| 4.8 | Learning as an iterative process of using training data, (a, b) , to create new parameters, p . Note that the role of inputs and outputs, typically called a and b respectively in this document, is now flipped with the parameter role. | 82 |
| 4.9 | A picture of how generative mappings relate to their discriminative counterparts. | 83 |
| 5.1 | A very simplified picture of a predictive coding “unit.” Here, v represents the value node, while e represents the error node. The two-in/two-out nature of the model makes it very tempting to formulate this picture as a pre-lens, however the directions of the arrows don't seem to line up—at least, not directly out of the box. | 87 |
| B.1 | A possible output of a clustering algorithm. Originally, the data points, represented as squares, would be unlabelled. | 114 |
| B.2 | An illustration of the generic RL setup. Here, an interpreter is distinguished from the environment itself, as it is possible that the details of the environment may be more sophisticated than an agent can perceive. | 115 |

List of Tables

| | | |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1 | A list of the considered VSAs, clarifying their naming conventions, and providing references to important early papers. | 40 |
| 3.2 | A modified version of Table 1 from Schlegel <i>et al.</i> [134]. I adopt the notation of Schlegel <i>et al.</i> , annotating the MAP VSAs to indicate the different domains vectors are permitted to exist in. Note that these different spaces come equipped with slightly different bundling operations, to ensure the produced vectors stay in the desired space. Similarly, I use Schlegel <i>et al.</i> 's notation to distinguish BSDC VSAs that use different binding operations. The unwieldy size of the table, being unable to contain within the margins of this document, is a happy coincidence that conveys the abundant variety of VSAs, and the need for unifying principles. | 43 |

Chapter 1

Introduction

In the summer of 2022, I was returning to my PhD research in a fairly discouraged state. I had been studying machine learning (ML) for a few years and became increasingly convinced that ML research is often “systematised p-hacking”—form a theory, build a model, train it for days, then tune hyperparameters when your theory doesn’t work. Who knows? Maybe it didn’t work because of a bug in the code. Better to “fix” things and try again than to throw away all your efforts, right? I was also discouraged by the proportion of research that was dedicated to topics such as optimising advertisements or music recommendation algorithms. This cynicism coincided with the explosion of large language models (LLMs), such as ChatGPT. My jaded attitude worsened as I saw growing research on topics such as prompt engineering¹.

Meanwhile, I was running a category theory reading group as a hobby with friends. I enjoyed the day-to-day work of studying mathematics a lot more than machine learning. My supervisor, Jeff Orchard, was supportive and mentioned this reading group to a colleague in Psychology, Britt Anderson. Britt invited me to join their workshop on *Category Theory for Cognitive Science* at CogSci 2022. My two reactions were “applying category theory sounds very cool!” but also “do you really need the full abstract toolkit of category theory to discuss things like cognition (surely tools from algebra, calculus, *etc.* are enough)?”

I’m sharing this background because these two thoughts drive this work. Despite my initial skepticism, I was struck by how simple and *natural* it felt discussing perception, reason, and learning in terms of morphisms and functors. In particular, Geoff Cruttwell’s demonstration at CogSci 2022 of how *performing* a task becomes *learning* a task via a simple functor was very compelling [13]. *Categorical Foundations of Gradient-Based Learning* [24] has become a

¹Though outside the main scope of this thesis, my full thoughts on the state of machine learning research are provided in Appendix A.

favourite paper of mine, largely due to the strong connection it makes between the compositional nature of category theory and the innate compositionality of deep learning. Furthermore, seeing the bridge between “abstract nonsense” and the foundations of programming has helped me appreciate both topics more thoroughly.

The goal of this thesis is obviously to present the results of my work. However, I wish to simultaneously impart upon you my passion for mathematics and theoretical computer/data science. In an era that is dominated by asking “what can AI *do for us*,” I want to highlight the reasons why my area of research is interesting regardless of its utility. Ultimately, my hope is that you, the reader, enjoy reading about this work as much as I have enjoyed studying it².

1.1 Motivation

I want to elaborate on the motivations for this work via two broad veins: my personal attachment to the material, and arguing that the larger community should also care about applied category theory. The reasons in one vein are also applicable to the other, so maybe it’s better to catalogue these motivations along the lines of conceptual/abstract versus practical. In either case, my reasons for study are as follows:

Aesthetics: First, this research has allowed me to return to my background in pure mathematics. My original motivation in academia was that the objects of study were just so *beautiful*. A bit of this was lost when I started studying the types of machine learning questions that seem to arise from asking about the statistical nature of the training data, as is the case in reinforcement learning or graph neural networks. The high-level questions posed by machine learning and AI are certainly fascinating, but it has been through category theory that I have most often felt that sense of aesthetics return to my studies.

Global vs. local thinking: Consider set theory, which asks us to understand objects by “looking inside them.” After all, the sole operator of set theory, \in , can be read as “is contained within,” or “is an element of.” In contrast, when studying category theory, objects are vacuous. All the information is in the morphisms. We might *pretend* that objects can be unpacked, as is the case when studying **Set**, but this is really a conceptual crutch. It’s possible to recover elements in a set simply by looking at how that set relates to *all other* sets.

²I have chosen to take a more conversational tone in this manuscript to try to facilitate this goal. I understand this may not be to everyone’s taste, but I only ever get to write one PhD thesis.

I view the categorical perspective as a shift from “local” thinking to “global” thinking, and I think it translates well to computer science. Consider a program from a typical first-year course: A set of instructions that looks inside registers and manipulates data that goes back into registers. You execute the first line of code, then the second, and so on. This imperative model strikes me as akin to the local, set-theoretic way of thinking. The alternative model of computation, functional programming, asks us to think about functions/mappings from one Type (Strings, Integers, *etc.*) to another. In this view, programs are simply the composition of these functions, and individual data points can be recovered from their data Types by observing how the functions behave. While neither model is computationally “better” than the other, I do appreciate the big picture perspective that functional programming encourages.

Enjoyment: As mentioned earlier, I also find that the moment-to-moment work of studying applied category theory is extremely *fun*. Although I enjoy programming, the software engineering side of research has been of little interest to me. Applied category theory has allowed me to struggle less with my package manager and spend more time sketching proofs and drawing diagrams to better understand topics.

Recognising that the above reasons are largely subjective and personal, I also have the following reasons that this work is broadly valuable:

Looking inside the “black box”: Across all the different local, structural, and paradigmatic variations of machine learning, a huge amount of cross-pollination and mixing has happened. A wide swath of machine learning research is conducted by tackling problems with this mix-and-match approach, but is largely *ad hoc* and unprincipled. As a result, the field has often been described as “data rich, but theory poor,” or (somewhat more critically) “alchemy” [123]. This is a big motivation for the line of research that I’ve chosen: I wish to use category theory as a means of understanding machine learning—or, more broadly, cognitive/intelligent function—in some “richer” sense. Though category theory doesn’t immediately reveal the complexities of emergent, learned behaviour, I think a good starting place is to at least understand what types of fundamental mathematical objects we’re studying.

Transplanting results to and from other topics: One of the most significant achievements of category theory has been describing mappings from disparate areas of math, then using these mappings to solve problems in one area using long-standing results from the other. In one sense, no new work is done: the proof was already formulated. However, it is valuable to strip away cosmetic differences and explicitly state connections that help people develop

general, abstract models. As an example that will appear in this thesis, vector symbolic architectures/algebras (VSAs) are really just Hilbert spaces with restrictions that arise from storage constraints. Because Hilbert spaces are so central to quantum research, my hope is that my research on VSAs may have value outside my field while also benefitting from the popularity of quantum information theory, quantum computing, *etc.*

Functional programming as a “compiler” for category theory: The above two motivations focus primarily on the benefits that category theory provides when doing applied research. However, I believe that category theorists can gain value from computer science as well. When studying category theory, it can be quite difficult to keep track of what layer of abstraction we are working within. For instance, the arrows of a given category, C , become the objects of its “arrow category”, $\text{Arr}(C)$, with morphisms becoming commutative squares in the original C . The wonderful thing about working with code is that it has to *compile* at the end of the day. In particular, typed languages force users to ensure that types are consistent at execution. Enforcing type-matching, even across functions (using the arrow $(->)$ constructor), means that programmers are forced to be intentional about exactly what our objects, morphisms, and any additional structure really are. This has motivated me to promote computer science and data science topics, such as machine learning and VSAs, as valuable tools for teaching category theory.

1.2 Summary of Contributions

To summarise in one sentence, my work uses applied category theory to understand models in data science. This work has had two main thrusts. The first, presented in Chapter 3, discusses VSAs, where I’ve constructed a unifying framework using category theory. The second, in Chapter 4, presents a generalisation of the current categorical foundations of supervised machine learning. In contrast to areas like interpretability, my focus will be on the underlying properties of models, rather than trying to formalise high-level behaviour. Though originally unintended, I’ve come to see an elegant “duality” in my research: both topics use high-dimensional vectors as the fundamental unit of data; VSAs are built for symbolic reasoning and computation with these vectors, while traditional machine learning is interested in incrementally adjusting such vectors by learning from training data.

1.2.1 Formalising Vector Symbolic Architectures

Vector symbolic architectures, or “algebras” (VSAs), are a family of algebras on high-dimensional vector representations. They arose in cognitive science from the need to unify neural processing and the kind of symbolic reasoning that humans perform. Chapter 3 contains my work on providing a categorical foundation of VSAs, similar to what [24] achieves for supervised learning. Within this broad objective, my specific contributions are as follows:

1. A literature survey in Appendix C demonstrates the novelty of the work.
2. A list of desiderata for VSAs is provided in Section 3.3.
3. A generalisation from vectors to co-presheaves (read: “functors,” though the reasons for choosing “co-presheaves” will be clarified in Chapter 3). Intuitively, you can understand the rationale for this generalisation as “unboxing” vectors into index-value pairs, $[\mathcal{I}, \mathcal{V}]$. Though it may initially seem that this generalisation is obfuscating, since vectors are so familiar in mathematics and computer science, it means that VSAs can be developed with a larger variety of fundamental operands. In particular, this formalisation is inclusive of recently developed GHRRs [172], which use hypervectors composed of d instances of $m \times m$ matrices. This generalisation also enables the following, most significant, contribution.
4. A principled way to determine optimal VSA operations by way of Kan extensions—a fairly ubiquitous categorical construction. Specifically, I present the two primary VSA operations, bind and bundle, as right Kan extensions of the external tensor product and external direct sum. The required properties of \mathcal{I} and \mathcal{V} are specified and end up being rather inclusive. Notably, this construction provides the properties required for our indexing category, \mathcal{I} , that allow dimension-preservation, which is a critical storage-saving consideration when building VSAs.
5. A collection of worked examples, demonstrating how to apply the above formalisation to understanding current VSA models, and suggesting new ways to build VSAs.

1.2.2 Presenting “Pre-lenses” as a Generalisation of Supervised Learning Architectures

Chapter 4 discusses supervised machine learning through the categorical notion of lenses and bicategories. A proper introduction to lenses and bicategories will be provided later, though

a quick summary is that lenses describe the bidirectional nature of backpropagation (computation vs. learning), while bicategories capture networks being able to compose in two distinct ways: “horizontally” along the layers of a network (formally 1-cells), and “vertically” as re-parameterisations (formally 2-cells). I noticed that this formalisation had some conceptual clunkiness; while vertical compositions always resulted in mappings from the network’s input, A , horizontal compositions would concatenate each set of parameters, $(P_0 \times P_1 \times \dots \times P_i \times \dots \times P_{n-1})$, where n is the number of layers in the network and i is the index of a given layer.

I invited Bartosz Milewski to discuss this problem in the Fall of 2023. Over the course of a week, we tried to find a generalisation of lenses that would reinforce a symmetry between composing network layers and re-parameterising a network. We were unsuccessful by the end of the week. I continued to think of it in the background but mostly shifted my focus to my VSA work. During this time, Milewski continued to work on the problem independently. By March 2024, they had come up with a new notion that achieved our goal: “pre-lenses” [102]. For this reason, Chapter 4 is based largely on Milewski’s work.

In spite of this, I’m including work on pre-lenses in my thesis for two reasons. First, the work remains unpublished in a peer-reviewed setting, appearing only on Milewski’s blog. Second, this thesis contains novel contributions of my own. Those are:

1. Replicating the work done by Milewski that builds neural networks using pre-lenses in Haskell.
2. Extending the above model to non-synthetic datasets, namely MNIST. This involved fixing a number of issues with the original codebase that didn’t manifest on such a small dataset.
3. Observing that pre-lenses emphasise the equivalence of traditional backpropagation with auto-differentiation. I also note some of the other symmetries that arise when implementing other ANN features, such as batching and re-parameterisations.
4. Characterising generative models as dual to discriminative models, coaching this observation in the language of pre-lenses.

1.3 Conventions and Notation

I’ve chosen to follow these conventions throughout the thesis:

- Lower-case letters from the start of the Latin alphabet will typically denote single objects, or inputs/output (read alternatively: the source and target of mappings). For instance, a may be a particular object in a given category.

- Lower case letters from the middle of the Latin alphabet will denote mappings between objects. For example, f could be a mapping from set a to set b
- Lower-case letters from the end of the Latin alphabet, such as x , will typically denote arbitrary values, or free variables.
- Upper-case Latin letters will follow similar conventions, though for “larger” kinds of things. For instance, C may denote a category instead of an object in a category; F may denote a functor.
- Particular, well-studied categories will be denoted with bold font. For example, the above-mentioned **Set** is the category whose objects are sets and whose arrows are functions on sets.
- The “blackboard” font will be used for sets of numbers, $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$. The Naturals will include zero.
- I use capital-T “Types” to refer to data types. I also capitalise the names of particular Types such as Strings, Booleans, Naturals, Lists, *etc.* This is partially to emphasise their importance, but also to disambiguate from using the word “type” when meaning “a kind of.”
- When code is discussed, the `code` font will be used. Generally, pseudocode will be as specific as this document gets, though some of the following conventions will be taken from Haskell:
 - Double colons (`:`) will be used to denote the Type of a given function. For example, `len :: Str -> Nat` means that `len` is a function that takes Strings as inputs and outputs Naturals.
 - The single equal sign (`=`) we be used to define a function. For example, `add1 x = x + 1.`
 - The list constructor will be a single colon (`:`) used as an infix operator. For example, `x = h:t` denotes that `x` is a list (of Type `List a`) with `h` as its head (Type `a`) and `t` as its tail (Type `List a`). Square brackets (`[_]`) will be used as an alternative notation when isolating the head or tail isn’t needed. The empty list is denoted `[]`.
- When composing two mappings, the composition of “ g after f ” is often written as $g \circ f$, which composes as it reads right-to-left. Some researchers have preferred switching to a left-to-right scheme and express the same composition as $f;g$. Since the semi-colon operator is unused elsewhere, I consider both to be useful. Typically, when relating an

equation to code, I prefer the \circ operator, as it is equivalent to the dot (.) operator in many languages. When working solely with mathematical expressions, I may use the semi-colon.

Every research area has its own common language and colloquialisms. Because my work is innately interdisciplinary, it can be challenging to balance the trade-off between using internally consistent notation and remaining consistent with pre-existing standards. My hope is that context is enough to clarify. When reusing symbols, I've tried to ensure that there is at least some thematic connection. In fact, there are places where this reuse is intentional to reinforce that—while we may be talking about different areas of research—the mathematical objects that underlie these topics are the same.

Chapter 2

Background

Here I provide a summary of the knowledge and literature that is relevant to my research. I assume that the reader has a background in undergraduate linear algebra, calculus, set theory, and general computer science. Some of the background that follows has been adapted from the background found in my masters thesis [138] and Comprehensive-II exam. This chapter covers the broad areas of machine learning and category theory, while subsequent chapters contain more topic-specific background.

2.1 Neural Networks

In recent years, machine learning, or ML, has become ubiquitous. Its applications are numerous, from self-driving cars [14] to nearly human-level chatbots [106]. The foundational idea of (supervised) machine learning is that an algorithm will be given input/output pairs for some task and learn the function that maps these inputs to outputs. This contrasts to classical computing, where the function and inputs are known and the algorithm solves for the outputs. These two paradigms are compared pictorially in Figure 2.1. Currently, artificial neural networks, or ANNs, constitute a considerable share of the algorithms used for machine learning purposes. Loosely inspired by the brain, these networks are composed of nodes (neurons) arranged in hierarchical layers, with connection weights (synapses) propagating information through the network.

A very early type of ANN was the perceptron [97]—a computational unit based loosely on the function of a single neuron. Unfortunately, it had significant limitations in its computational breadth. The most canonical example of this limit is that perceptrons are unable to compute the XOR function, as shown by Minsky and Papert [104]. This is not particularly surprising—the

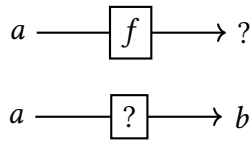


Figure 2.1: A visual comparison of traditional computation (top) versus machine learning (bottom). When represented this way, it is easier to see that machine learning isn’t really some “better” way to build algorithms—both models are kind of like an algebraic equation solving for one unknown, and it is simply which unknown that varies.

human brain doesn’t get its computational strength from how well a single neuron functions, but from the use of billions of neurons. For this reason, the perceptron fell out of favour for decades, with researchers favouring methods such as statistical optimisation and decision trees.

However, researchers soon demonstrated that perceptrons could be stacked in multiple layers to greatly expand the class of functions that they can learn¹. These ANNs are called multi-layered perceptrons (MLPs) or feed-forward neural networks. They are composed of an input layer, an output layer, and some number of intermediary layers between the two, called hidden layers. These layers are arranged hierarchically, with weights connecting each layer of the network to the next. The area of research that studies these multi-layer neural networks has been dubbed “deep learning” and its impact on computer science and, more broadly, modern life has been considerable.

Formally, these networks can be described as:

1. A set of nodes, organised in layers, with inputs, a , and outputs, b ;
2. A set of connection weights and biases, W and h respectively—often grouped together as parameters, p —and;
3. A propagation function σ , such that $b = \sigma(a)$ and, for the subsequent layer, $a = W \cdot b + h$.

With just these components, one can define any measurable function. However, without a means of learning a desired function, these networks are restricted to modelling predefined functions. Thus, we need one additional component:

4. A learning rule that specifies how to modify W and h .

¹In fact, a three-layer perceptron can learn any measurable function [69], though the number of neurons/nodes needed might be intractable.

2.2 Backpropagation

Of the various learning algorithms implemented in feed-forward neural networks, the most widely studied and implemented is the error backpropagation (*alt*: “backprop” or “BP”) algorithm [130]. Simply put, error back propagation works by providing a dataset of input-output pairs, applying the inputs to the neural network, propagating that input forward through the weights and activations of the network, and using this output along with the target output to generate an error signal with some specified loss function. That is, $E = L(b, t)$, where b is the predicted output and t is a given target. Once this error signal has been computed, the gradient of the error with respect to each weight is calculated using the chain rule. These errors indicate which updates should be made to the weights to reduce the loss of the network on the given task. Thus, back propagation is essentially performing gradient descent on the parameter space of the network.

In detail, the gradient of the error, E , can be computed for any given connection weight in the network, $w_{ji} \in W$, connecting the i -th neuron in the previous layer to the j -th neuron in the current layer. Applying the chain rule, this gradient is expressed as

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial b_j} \cdot \frac{\partial b_j}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}}, \quad (2.1)$$

where b_j is the output of the neuron, and $a_j = \sum_k w_{kj} b_k$ is the input of the neuron, computed by taking the linear sum of the outputs from the previous layer times the associated weights.

Each term in Equation 2.1 is fairly straightforward to compute; For the output layer, the combined $\frac{\partial E}{\partial b_j} \cdot \frac{\partial b_j}{\partial a_j} = \frac{\partial E}{\partial a_j}$ is just the gradient of the loss (often $(b - t)$, where t is the target prediction). For internal layers, $\frac{\partial E}{\partial b_j}$ can be computed recursively to yield

$$\frac{\partial E}{\partial b_j} = \sum_k \left(\frac{\partial E}{\partial b_k} \cdot \frac{\partial b_k}{\partial a_k} \cdot w_{kj} \right) \quad (2.2)$$

for all neurons k in the subsequent layer. The term $\frac{\partial b_j}{\partial a_j}$ is simply the slope of the activation function, $b = \sigma(a)$, alternatively denoted as $\sigma'(a)$. The final term, $\frac{\partial a_j}{\partial w_{ji}}$, is less immediate, but remember that $a_j = \sum_k (w_{jk} \cdot b_k)$. So the only term containing w_{ji} is exactly $(w_{ji} \cdot b_i)$ whose gradient is b_i . Biases are computed in a similar manner through recursion. With all the values for $\frac{\partial E}{\partial b_j}$ and $\frac{\partial E}{\partial w_{ji}}$ found for the current layer, the errors for the previous layer can be found in the same manner. This is done recursively for every layer until all the gradients for each weight have been found.

Having obtained all the values of $\frac{\partial E}{\partial w_{ji}}$, the goal is to then use these gradients to improve the performance of the network. Since the loss function provides the measure of performance, and the gradients of that error have been found, a simple update can be made to adjust all parameters of the network such that the loss decreases. This method is called “gradient descent” and gives weight updates of

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \frac{\partial E}{\partial w_{\text{old}}} \quad (2.3)$$

for some learning rate, α . Biases are updated in the same way.

In general, when discussing any loss function, L , parameterised by all network parameters, p (which consists of all W and h), we denote its gradient as $\nabla_p L(p \times (b \times t)) = \nabla_p L(p)$, since b and t are fixed for each iteration.

2.2.1 Other Error-propagating Algorithms

Back propagation is not the only error propagating algorithm. While this work does not focus on other error propagation algorithms directly, they are very common in applications, as they typically address many of the issues with backprop (sensitivity to α and the order samples are provided, obtaining local rather than global minima, *etc.*). Noteworthy examples are Nesterov momentum [30], Adam [78], and error-entropy minimisation [38]. These other optimisers are usually modified versions of standard BP that work by reparameterising p in some way. In other words, they are usually a remapping of p to some pair $(p \times s)$, where s is some persistent state. This will come up briefly later when discussing how category theory unifies the various components of a machine learning model.

2.2.2 Learning Pipeline

With the learning process described in detail, I now present the typical learning process, start to finish, for a feed-forward neural network. Consider two sets, S and T , where S is the set of input data, and T is a set of classes. This process uses sample input-output pairs, $(s, t) \in (S \times T)$, to teach the network how to associate the two (this notation hopefully reinforces that it is learning some *function* between S and T). Practitioners call this paradigm “supervised” learning. I will use the superscript notation $-^{(i)}$ to refer to the i -th layer of a network. For example, $a^{(i)}$ would refer to the *input* of layer i .

1. **Initialisation:** The network is constructed with its parameters initialised randomly.

2. **Feed-forward pass:** Input $a^{(0)} = s$ is fed into the network’s input layer, and the activation function of each neuron is applied

$$b^{(0)} = \sigma(a^{(0)}). \quad (2.4)$$

These values are then multiplied by the weight matrix, $W^{(0)}$, with dimensions $m \times n$, where n is the dimension of the input layer and m is the dimension of the subsequent layer, then biased by $h^{(0)}$, yielding

$$a^{(1)} = W^{(0)} \cdot b^{(0)} + h^{(0)}. \quad (2.5)$$

This is then repeated for every layer until the output layer. The output of the network will be referred to as b^{out} .

3. **Compute error:** The output of the network, b^{out} , and the target output, t , are then used to compute the error for some given loss function. Typically, the L_2 (Euclidean distance) or cross entropy error are used.
4. **Propagate error backwards:** This error signal is then propagated back through the network as specified at the start of Section 2.2.
5. **Update parameters:** With all gradients now computed, the weights and biases of the network are updated in the manner shown in Equation 2.3.
6. **Loop or halt:** This process is then repeated from Step 2 for each training sample. One full pass of all the training samples is called an “epoch.” Then, if the halting condition is reached (either by having sufficiently low loss or running for enough epochs), the algorithm terminates. Otherwise, the algorithm re-shuffles the data set and returns to Step 2 again.

This process is quite straightforward and, by itself, has achieved significant performance in various classification and regression tasks. However, “improvements” to this simple framework have catapulted deep learning models to further heights and are discussed now.

2.3 Modern Deep Learning

Variations of backprop are not the only developments in deep learning that have contributed to its success and widespread adoption. Changes to deep learning models can be roughly separated into local modifications and structural modifications.

Local modifications are changes made to the computations performed at a given step of the learning process. The variations of backprop mentioned above are of the local variety. However, these changes need not focus on just the backward phase of computation. Other examples include changes to our choice of activation functions, or mechanisms that modify input distributions to layers by incorporating statistical information. All these developments are characterised by focusing on a component of the learning process that is modular in some way and can be applied to a broad range of problem domains.

In contrast, structural developments are ones which modify the network structure in some manner. For example, there are applications where engineers may wish to use a network to make a sequence of predictions, with each one dependent on previous predictions in some ways. To tackle this type of problem, a vanilla feed-forward network can have loops from a layer to itself, creating *recurrence* within the network. Structural changes are often more “radical” in some sense, as they may require new specialised learning rules, *etc.* Compared to local changes, they are often the result of trying to address more domain-specific problems.

2.3.1 Local Variations

As stated, local changes to the learning process are characterised by being modular and domain-independent. In addition to being applicable to most structural variants of neural networks, local changes can often be combined with one another as well. Variations of learning rules were presented with sufficient detail in Section 2.2.1, so I will instead focus on other forms of local modifications to networks in this section.

A different way in which networks can be locally changed is in the choice of activation function, σ , which was never specified in Section 2.2.2. For a long time, the canonical choice of activation function was the logistic function,

$$\text{logistic}(x) = \frac{1}{1 + e^{-x}},$$

as it shares some number of properties that biological neurons exhibit. First, it demonstrates a sharp slope at the origin, resembling the kind of “thresholding” seen in real neurons. Intuitively, this makes it a good candidate for tasks where the goal is to discriminate its inputs into two classes (or form some hyperplane at the network level). This thresholding is paired with saturation behaviour at the extremes—even though the logistic function is defined on all of \mathbb{R} , its image is $(0, 1)$, with these two boundaries being approached very quickly. As a result, a learning model will not be unduly affected by distant outliers in its inputs. The logistic function is also “odd” around the point $(0, 1/2)$, endowing it with an elegant rotational symmetry.

From a practical perspective, the logistic function is nice to work with because its derivative is simply $\text{logistic}'(x) = \text{logistic}(x)(1 - \text{logistic}(x))$. Since $\text{logistic}'$ is critical to computing the gradients in our network, it is extremely convenient that it can be computed using simple arithmetic on logistic, which was already calculated during the forward pass. The logistic function is an instance of the broader class of “sigmoid” functions (functions that demonstrate an S-shaped curve). Another such function is the tanh function. In fact, one is simply an affine transformation of the other. Despite this, practitioners have seen that tanh often converges faster than logistic in practice [89], possibly because its range includes negative values.

Unfortunately, both the tanh and logistic function have issues when learning. First, saturation at the extreme ends of the function lead to gradients that are very small, since the $\frac{\partial b}{\partial a}$ in Equation 2.1 is close to zero. This results in a phenomenon called the “vanishing gradient” problem, where activities in the saturated ends of the function for higher layers of the network prevent error signals from propagating back to the lower levels of the network. Second, as networks scale in size to millions, or possibly billions, of nodes, even the simple computations of the derivatives for these functions can add up to be expensive.

To address these issues, practitioners have widely adopted the use of “rectified linear units,” or ReLUs, for their choice of activation function. ReLUs were first proposed in [46] and are defined as

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

These functions have the advantage of being less sensitive to vanishing gradients for positive inputs. Furthermore, they are very efficient to compute, as their derivative is either zero or one. This results in very sparse representations when both producing predictions and when backpropagating gradients via matrix multiplication.

These benefits don’t come entirely for free though. ReLUs are not differentiable at the origin. In practice, engineers can simply choose to make its derivative at the origin either zero or one, but this solution is not exactly rigorous. Also, they are not symmetric about the origin, and have a bias towards positive values. In practice, this is mitigated by inputs being biased by the parameters from the previous layer, but this is not always easily solved since ReLUs also have their own form of vanishing gradients; when a ReLU returns a zero value, all parameters below that activation end up with a zero-valued gradient. This issue is has been dubbed the “dying ReLU” problem.

In addition to choosing different activation functions, researchers have also begun to modify network inputs before passing them into activation functions. The most well-known method for doing so is called “batch normalisation,” or batch norm [71]. While training networks on large databases, it is faster to process many predictions in parallel, rather than one at a time. Thus,

inputs will be organised into batches of some given size. The underlying principle of batch normalisation is to use the statistical mean and variance of each batch to scale and re-center the data. This normalises each dimension in the input to be as close to a standard Gaussian as possible.

Since batch norm operates at the level of the node, and doesn't interact with the weights of a network, it can be thought of as an "intrinsic" mechanism. This notion, sometimes referred to as "intrinsic plasticity" (contrasting with weight-change mechanisms that fall under "synaptic plasticity") includes other processes. Researchers, such as Jochen Triesch, have demonstrated that neurons can effectively learn the activation function that maximises a neuron's "entropy", or "information potential,"² for a fixed mean firing rate (or "energy budget"). In biological neurons, this energy consideration is important for reducing the caloric requirements of the brain. However, artificial networks are unconcerned with maintaining a low firing rate, as energy consumption is not an issue the same way it is for biological brains. Thus, a rule can be used that strictly maximises information potential. Bell and Sejnowski's Infomax rule does exactly this [17]. The ideal activation function turns out to be one that results in an output distribution as close to uniform as possible, as the uniform distribution possesses the highest entropy.

For my previous work I developed another such mechanism, dubbed the IP rule [139]. This rule was built to address stability issues with the Infomax rule. Its other benefits were very similar to those seen with batch norm—it addressed the vanishing gradient problem and accelerated learning considerably. However, unlike batch normalisation, it also resulted in neuronal entropy levels comparable to the Infomax rule.

Intuitively, it may seem that these mechanisms would homogenise network inputs, however the impact of homogeneity seems to largely be outweighed by the other benefits that these mechanisms provide. In fact, since each batch is scaled to have a variance of approximately one, there is still plenty of room to discriminate inputs. The benefits are considerable, with batch norm being particularly well-documented as improving network performance. However, the underlying reasons that batch norm works are still not well understood. The original claim that it worked by "reducing internal covariate shift" has since been brought into question [132].

²The technical background of information potential is founded in information theory [137], which is largely unrelated to this work. In brief, it can be seen as a measure of how efficiently neurons/nodes communicate data to upstream neurons.

2.3.2 Structural Variations

One major structural iteration on feed-forward neural networks is introducing additional connections from a given layer to itself³. This means that network outputs are not only dependent on inputs, but also on the previous state of the layers where these “recurrent” connections exist. These recurrent neural networks, or RNNs, have a form of ad hoc *state*, or *memory*, that makes them well-suited to tasks where temporal information is important. Examples are speech recognition [60], and translation [151]. While there are more recurrent architectures than can be summarised here, the general principle is that these self connections can be handled computationally by “unrolling” them into a computational graph that closely resembles the simpler feed-forward networks. An illustration of this unfolding is provided in Figure 2.2.

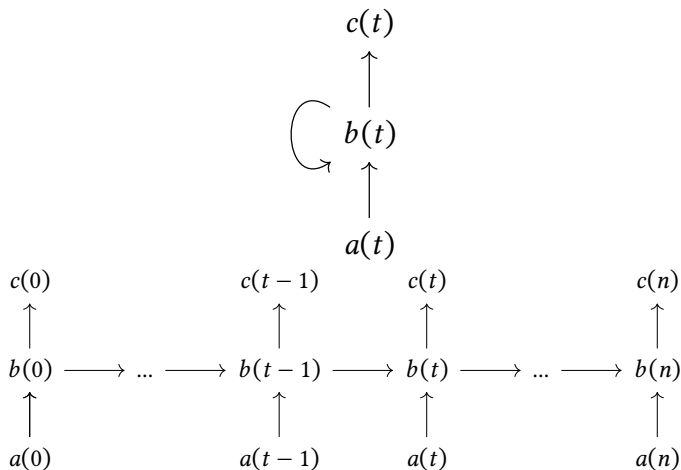


Figure 2.2: An illustration of a typical recurrent neural network (top), and how it can be “unrolled” to produce a completely acyclic representation (bottom). Here, n represents the final value of t . Since gradients depend on upstream errors, parameter updates early in training sequences can become exponentially small if activities in the hidden layer are small for large values of t .

This new recurrent structure requires a modified learning rule to ensure that the recurrent weights appropriately handle all types of inputs (or, more precisely, all possible *sequences* of inputs). Backprop can be generalised in a fairly natural manner, resulting in the “back propagation through time,” or BPTT, algorithm [131], which is considered a standard method of training. However, this method is particularly sensitive to vanishing gradients, as each step through time

³We will later refer to an operation from an object to itself as an “endo”-morphism, once we are equipped with the language of category theory.

introduces another (potentially) near-zero scaling factor on the error [66]. In some domains, practitioners will instead use global optimisation methods, such as genetic algorithms [152].

Another form of recurrent networks is the “long short-term memory” (LSTM) network [67]. These have been widely adopted for practical use, as they largely avoid the types of vanishing gradients that standard RNNs face. In short, LSTM networks are stacks of LSTM units, which are composed of a “cell” that stores information, an input gate, output gate, and “forget” gate. Since these cells can pass information through the network without being modified, error signals can also propagate back through the network without needing to pass through the recurrently connected layers. LSTM networks have been immensely successful in a variety of tasks, including robotic dexterity [96] and game playing (most notably, Dota 2 [18] and Starcraft 2 [51]).

Where recurrence in networks is typically used to tackle problems with some *temporal* component, researchers have also modified network architectures to better process *spatial* information. The most well-known of these architectures are “convolutional” neural networks (CNNs), which were first introduced in [47]. Inspired by the sort of architecture seen in the visual cortex [70], CNNs forgo the fully-connected layers of MLPs for convolutional layers that have a limited receptive field. This is illustrated in Figure 2.3. These layers can be seen as *filters* that learn how to detect particular features within their receptive field and propagate those features forward. Since we can stack convolutional layers on top of one another, hierarchical CNNs can be seen as learning progressively abstract features by combining simpler features from layers closer to the input (for example, starting out by filtering for line orientations in low-level layers, and working up towards high-level concepts like faces).

CNNs have a few advantages within their typical problem domains. The first is weight reuse. Rather than having the fully connected layers of MLPs, a single filter can be applied to all receptive fields from the previous layer. Assuming that we wish to extract the same sorts of features across the entire input, this single filter is the only set of parameters that needs to be learned. As a result, learning is much faster. Another related advantage is that using the same filter in a limited receptive field naturally leads to spatial information being encoded. Since the convolution only applies locally, global information doesn’t affect the information propagating forward through the network.

It is also common practice to introduce “pooling” layers to CNNs, which reduce the dimensionality of a given layer by combining the information from multiple neurons and storing them in a single value. This improves the speed it takes to train a network by compressing the data in a systematic way. Two common types of pooling are “max” pooling, which stores the largest value across a given set of neurons, while “average” pooling returns the average value.

CNNs have been very successful in a few applications. In particular, CNNs have been used to achieve extremely low error rates on MNIST [90], ImageNet [28], and CIFAR-10 [82] classi-

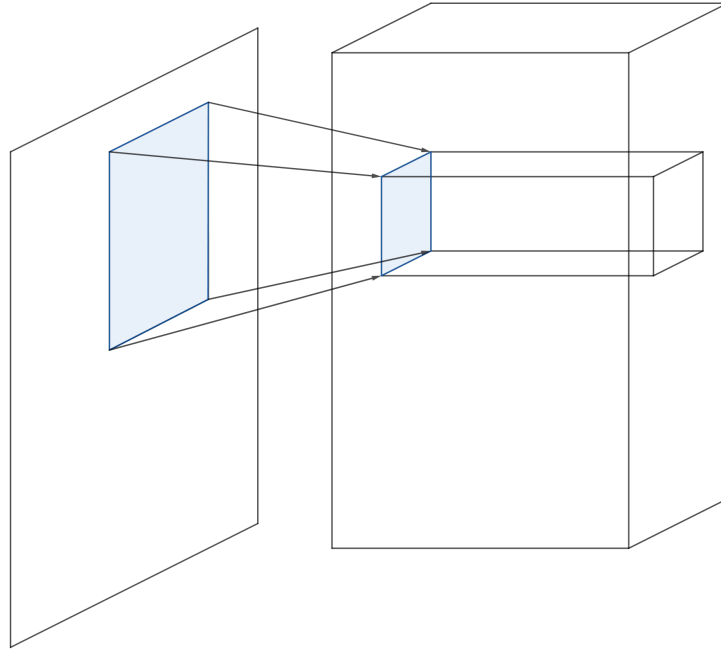


Figure 2.3: An illustration of a typical convolutional layer. The blue box on the left represents a given receptive field, while the box on the right represents the corresponding target neurons in the subsequent layer. The depth of the box on the right represents that there can be multiple nodes that map from this receptive field.

fication datasets. However, applications are not limited to just image recognition; they have also been successfully applied to playing board games [22] and drug discovery [162]. CNNs have even been successful in time series forecasting [158]—a domain thought to be better-suited to RNNs for its sequential nature—by arranging temporally-ordered data in a spatial order.

Another key development in modern deep learning has been the introduction of “skip” connections, or “residuals,” between layers. These residuals work by adding connections from low levels of a network to much higher levels, skipping over one or more hidden layers. An illustration of this process is given in Figure 2.4. First introduced by Kaiming *et al.* in [62], residual networks (or “ResNets”) gained popularity very quickly after winning the 2015 ImageNet competition.

Residual connections are very beneficial to learning in deep networks, as the strength of input signals can be independent of the depth of the network. In the forward direction, discrimination can be performed with data that comes directly from the input, as well as whatever function has

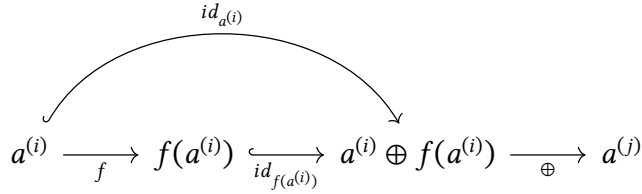


Figure 2.4: The general structure of a residual connection in a neural network. Here, $a^{(i)}$ is the input, while f is the computation performed by a given layer (or layers) of a neural network. The hooked arrows represent the inclusion mapping to some product that is then reduced using a given \oplus binary operator. Typically, this operator is simple addition, however other choices, such as concatenation, are possible as well. The final object is $a^{(j)}$ —the input to subsequent layers.

processed that input. More specifically, the input into a higher layer, $a^{(j)}$, changes from $f(a^{(i)})$ to $f(a^{(i)}) \oplus a^{(i)}$, where $a^{(i)}$ is the input to some lower layer and \oplus is a chosen combination operator similar to one described later.

In the reverse direction, remember that our error signal can be expressed by

$$\frac{\partial E}{\partial a^{(i)}} = \frac{\partial E}{\partial a^{(j)}} \cdot \frac{\partial a^{(j)}}{\partial a^{(i)}}. \tag{2.6}$$

Expanding the second term on the right, we obtain

$$\frac{\partial E}{\partial a^{(i)}} = \frac{\partial E}{\partial a^{(j)}} \cdot \left(1 \oplus \frac{\partial f}{\partial a^{(i)}} \right) \tag{2.7}$$

$$= \frac{\partial E}{\partial a^{(j)}} \oplus \frac{\partial E}{\partial a^{(j)}} \cdot \frac{\partial f}{\partial a^{(i)}}. \tag{2.8}$$

Thus, regardless of how small $\frac{\partial E}{\partial a^{(j)}} \cdot \frac{\partial f}{\partial a^{(i)}}$ becomes due to vanishing gradients, there is always the $\frac{\partial E}{\partial a^{(j)}}$ term to propagate error backwards. Because of this benefit, residual connections are now present in most modern neural network architectures.

Another structural variety of neural network models are “graph neural networks,” or GNNs. Graph neural networks are somewhat distinct from the previous examples, in that they deal with a particular type of data—graph data. This type of data is characterised by having an adjacency matrix that is $n \times n$, where n is the number of data points in our dataset. Graph neural networks capitalise on this additional, structural information by pre-processing the data using a convolution, or “message-passing,” operation prior to passing the output to another neural network architecture. In a sense, CNNs are just one instance of GNNs where our adjacency matrix is the grid of neighbouring pixels.

Typically, these message-passing algorithms are some variety of neighbourhood “averaging.” A given node, x , with neighbourhood N_x is updated to be h , where h is the output of some activation function whose input is

$$\bigoplus_{v \in N_x} \phi(x, v, e_{xv}),$$

where e_{xv} is the edge from v to x . Note that this formalisation does not forbid x from being its own neighbour—other formalisations sometimes choose to omit x from its neighbourhood and explicitly include it as an argument to the activation function. The operation \oplus is characterised primarily by being “permutation invariant”. This simply means that the output, h , should be independent of the order that each ϕ is computed (i.e. \oplus is commutative).

Graph neural networks have seen use in a wide number of modern applications. Protein folding [27], combinatorial optimisation [20], and recommender systems [39] are just some of the domains where graph structure is an integral part of the data being examined.

However, graph networks aren’t strictly beneficial. The convolution operation can be seen as a type of “smoothing” of the input data. Over-smoothing can result in datapoints that are too close together to be easily distinguished by some discriminative model. GNNs are also known to abide by the same restrictions as the Weisfeiler-Lehman Graph Isomorphism Test [168]. This means that there are certain forms of graph data that are known to be indistinguishable by a GNN. It is currently unclear whether GNNs with augmented datasets can distinguish more inputs than simpler message-passing algorithms.

The final structural variant I wish to briefly discuss are “transformer” networks [161], as they are largely the state-of-the-art for machine learning models. Transformers are neural computational “units” that are composed of many previously-developed learning components (normalisation, skip connections, MLP layers, *etc.*), along with the innovation of a “multi-head attention” mechanism. In short, a given input is broken into “tokens” (ex. text being split into words), which are then weighted by a “context window”, with the resulting contextualisation promoting more “attention” being given to important tokens.

Originally, these transformers were primarily used for natural language processing (NLP) and computer vision [29]. They have gained favour compared to LSTMs and other recurrent models due to their quick training. However, now they have also been applied to audio and multi-modal data [121]. Famously, transformers are the architecture that underlie GPT models [106].

Even though these structural variants aren’t local to a single layer of a network, they can still be seen as modular at the network level. This means they can largely be mixed and matched. For example, the original ResNet was also a convolutional neural network, since its task was image recognition. Another example is that video data can be processed by passing the individual frames through a convolutional network, then composing this with some recurrent model.

2.3.3 Closing Remarks on Machine Learning

There are other paradigms besides supervised learning. I've provided an overview of unsupervised learning and reinforcement learning in Appendix B. A common theme throughout all models of learning, though supervised learning in particular, is that they are in some way modular, or *composable*. Neural networks rose to prominence during a period when stacking network layers was used to solve more sophisticated tasks, with the hypothesis being that more “abstract” representations could be learned in higher layers of a network. Nevertheless, the compositional nature of networks, and learning more broadly, has been overshadowed by pursuits of optimising performance and engineering new models. The compositional nature of these models is precisely why I, and many other researchers, have just recently turned to category theory to describe universal models of learning.

2.4 Category Theory

By mathematical standards, category theory is a very modern field. It was first introduced in 1945 by Eilenberg and Mac Lane while doing work in algebraic topology [35]. Given this origin, it is unsurprising that category theory is highly abstract. While this has caused some people to doubt its usefulness (even some mathematicians), category theory has quickly become a common language in many areas of research due to its ability to describe many objects and concepts in highly general and unified terms. For example, notions of products, duality, and limits can all be understood in simple terms, and across various fields, when viewed from a categorical perspective.

2.4.1 What is a Category?

From the outset, I want to emphasise that categories are concerned with understanding mathematical concepts or objects by how they *relate* to one another, rather than by examining their intrinsic properties. Philosophically, I view this as a shift from expressing concepts using “local” properties to “global” ones. The following definition of a category uses notions of objects and morphisms. I encourage you to think of the objects as vacuous, while the morphisms contain all the information about the category at hand.

Definition 2.4.1. *A category, C , is built from three things: objects, morphisms, and composition. “Objects” are atomic and serve only as the start/endpoints of morphisms. The class of all objects in a category is written $\text{ob}(C)$.*

“Morphisms,” (or, interchangeably, “arrows” or “maps”) connect pairs of objects, often called the “source” and “target”. A morphism, f , can be written as $f : a \rightarrow b$, where a and b are the source and target respectively. The class of all morphisms between a and b is written $\text{hom}(a, b)$ or $C(a, b)$ (often called the “hom-set” of a and b), with the class of all morphisms in the category being written $\text{hom}(C)$.

Finally, there is the composition operator, \circ , which is binary and must obey the following three axioms:

1. *Composition:* for all pairs of morphisms, $f : a \rightarrow b$ and $g : b \rightarrow c$, there must exist a third morphism, $g \circ f^4 : a \rightarrow c$.
2. *Identity:* every object, x , in a category, C , must have an identity morphism, $1_x : x \rightarrow x$, such that for every morphism, $f : a \rightarrow b$, we have

$$1_b \circ f = f \circ 1_a.$$

3. *Associativity:* For any three morphisms, f , g , and h , if $f : a \rightarrow b$, $g : b \rightarrow c$, and $h : c \rightarrow d$ then

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

In summary, categories are essentially directed graphs with the fewest constraints required to model composition and the various structures that arise from coherently composed morphisms. For this reason, it is very common to represent categories and subcategories with diagrams such as the one in Figure 2.5. In fact, category theory is one of the few fields where “proof by picture”, or “diagram chasing,” is considered a legitimate proof technique!



Figure 2.5: A simple, three object category. All arrows are made explicit in the left diagram. Conventionally, identity morphisms are omitted to reduce clutter, resulting in the right diagram.

⁴Alternatively: $f; g$.

Once we understand what categories are, they start to appear all over the place. For example, sets can be reinterpreted as a category with sets as objects and functions as morphisms. We call this category **Set**. Groups also form a category, **Grp**, where each group is an object and each group homomorphism is a morphism.

The list goes on, and even simpler structures can be viewed as categories—for example, the ordering on the natural numbers, \mathbb{N} , becomes a category where each number is an object and morphisms between numbers represent \leq . In fact, this is one instance of a *partial order*, which can be understood categorically and is an important concept appearing in Chapter 3.

Definition 2.4.2. *A partial order is a category where there exists at most one morphism between any two objects (This morphism is often written as \leq). If there is a morphism from a to b , and a morphism from b to a , then we can say that $a = b$.*

A special category that we can define is the category of a programming language, L . Here, the objects are Types, such as `Int`, `Bool`, and `Str`, while the arrows are functions between these Types. For instance, we may have a function, `len :: List a -> Nat`, that maps from a list of Type a and returns its length as a Natural number. From a functional perspective, if a and b are both Types, then $a \rightarrow b$ is also a Type⁵. Because computation is generally language-independent, I often think of the category of a given programming language, L , as the more general “category of Types,” **Type**.

2.4.2 Duality

A question that immediately arises when looking at categories is “what happens if we just reverse the arrows?” This seemingly naive question has a simple answer but reveals a philosophically important aspect of category theory.

Definition 2.4.3. *Every category, C , has a corresponding category, C^{op} , that is the result of flipping the arrows of C while still using the composition from C . More formally:*

- *Each object, $c \in \text{ob}(C)$, appears in C^{op} .*
- *For every pair of objects, $a, b \in \text{ob}(C)$, each morphism, $f \in \text{hom}(a, b)$ has a corresponding morphism in $\text{hom}(b, a)$ of C^{op} .*
- *Whenever two morphisms, $f, g \in \text{hom}(C)$ compose as $f; g$, their corresponding morphisms in C^{op} compose as $g; f$.*

⁵In categorical terms, this means that the category is “closed,” and this property will be important later.

Though it is easy to conceptualise taking the dual of a category, it can sometimes be difficult to parse the “meaning” of our new arrows in C^{op} . For example, in **Set** we know that our morphisms are functions between two given objects. Let’s call these a and b . In \mathbf{Set}^{op} a morphism going in the reverse direction from b to a can still be imagined as a function from a to b . However, arrows indicate some notion of starting at the source and going to the head. For this reason, it would be nice to have a name for a mapping from b to a . Such a mapping is equivalent to the important, though less familiar, “pullback” of a function [160, Theorem 2.4]. This example also highlights that duals are not simply taking the inverse mapping that is represented by a morphism (pullbacks can be seen as including one-to-many mappings that are prohibited by inverses).

In general, all concepts and results in category theory are generally “buy one, get one free” by way of duality. In the next section, I’ll introduce universal properties and whenever a concept is introduced (such as “terminal objects”), trust that there is a dual notion that corresponds to that concept (like “initial objects”). While some pairs of notions—such as initial and terminal—may have unique terms, many are simply captured with the prefix “co-”, such as products and coproducts, limits and colimits, and ends and coends, *etc.*

2.4.3 Universal Properties

As mentioned above, the objects of a given category don’t really contain any information (except for acting as the source and target of morphisms). Everything is in the arrows, so to speak. However, for examples such as **Set** and **Type**, we clearly see that there are *supposed* to be things inside our vacuous objects. For instance, the object \mathbf{Bool} in **Type** should contain both \mathbf{T} and \mathbf{F} , and sets generally contain many elements. So where are these internal elements in category theory? It turns out that we can recover everything we need by only examining the structure and number of morphisms between collections of objects.

To develop an intuition of how this works, consider **Set**. How could we count the number of elements in a set? Let’s start by seeing if we can pick out singleton sets without “looking inside” any sets. We can ask: how many functions map from *any* set to a set with just one element? More formally, let $\{*\}$, or $*$ be a singleton set. For any set, $x \in \text{ob}(\mathbf{Set})$, how many morphisms are there in $\text{hom}(x, *)$? The answer is only one, no matter the size our domain, x ! If we cheat for a moment and phrase this in terms of elements, we know that it is because we are forced to map every element to the sole element of our singleton set. However, we only need to inspect our hom-set to see that there’s a unique morphism to $*$. This is represented pictorially in Figure 2.6.

In categorical terms, we say that $*$ is a “terminal” object and is defined formally as follows:

Definition 2.4.4. *A terminal object in a category, C , is an object, $*$, such that for every other*

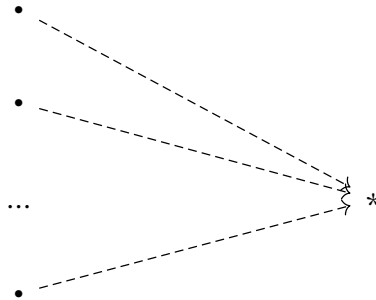


Figure 2.6: A picture of how all objects in **Set** map *uniquely* to any singleton, $*$. The other objects are kept as unlabelled dots in this diagram to emphasise that the sizes and elements of these sets are irrelevant.

*object, $x \in \text{ob}(C)$, there is a unique morphism, $! : x \rightarrow *$. A terminal object, if it exists, is unique up to isomorphism, and may thus be called “the” terminal object of C .*

A few things to note. First, we know that there are many singleton sets, such as $\{1\}$, $\{a\}$, or $\{\heartsuit\}$. In fact, there are infinitely many singleton sets. This demonstrates that, in general, terminal objects are not unique—hence the “unique *up to isomorphism*” language in our definition. However this raises an interesting question: what’s really the difference between any singleton set? In some sense, because they are all isomorphic, there isn’t much that really distinguishes them from one another. Second, as discussed in Section 2.4.2, we know that we can obtain a dual concept by flipping the direction of the arrows in Figure 2.6. This gives us the idea of an “initial” object, and for **Set** that is simply the empty set, \emptyset (which in this case is truly unique). Finally, it’s not necessarily guaranteed that initial and terminal objects exist for all categories. For instance, any partially ordered set, or poset, without a “smallest” object lacks an initial object, and the same can be said of terminal objects if it lacks a “largest” object.

The language of Definition 2.4.4 is written in such a way that we can say that $*$ satisfies a described “universal property.” That is, what we’re saying is that $*$ *is* a terminal object *because* it has the unique corresponding mappings from every object, x in C . However, there are myriad possible structures that boil down to some universal property. Figure 2.7 presents a diagrammatic representation of a “product” using two familiar examples.

More examples of universal properties will arise throughout this thesis. In general, the “interesting” properties of objects or morphisms of a category are universal properties. From a learning perspective, it can sometimes be helpful to “peek inside” objects for categories that are

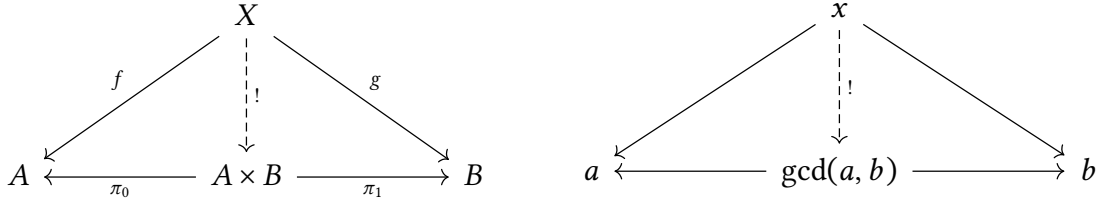


Figure 2.7: Two examples of a product, categorically. On the left is the Cartesian product of sets A and B , written as $A \times B$. The Cartesian product is defined as the set with functions, π_0 and π_1 , to A and B respectively (think of these as the “projections” from $A \times B$ to each part of the product) such that any object, X that has morphisms f and g mapping to A and B respectively, maps uniquely through $!$ such that $!; \pi_0 = f$ and $!; \pi_1 = g$. On the right, we have a product from the category with Natural numbers as objects and morphisms representing the “divides” relation. Here, every hom-set contains one morphism at most, so the uniqueness of $!$ is guaranteed. The diagram expresses that any number, x , which divides both a and b must also divide $\text{gcd}(a, b)$.

familiar to us. However, this is largely a conceptual crutch and unnecessary for understanding the nature of a given category. A high-level way of expressing this perspective is “we can understand objects by studying how they relate to all other things rather than looking at the objects themselves” and it is generally considered the core tenet of category theory, expressed through the Yoneda and co-Yoneda lemmas⁶.

2.4.4 Functors

By themselves, categories are quite expressive. As seen above, common mathematical notions, such as products and coproducts, are easily expressed as universal properties. However, to see the full expressive power of category theory, it is important to understand both “functors,” which are *structure-preserving* maps between categories, and “natural transformations,” which are structure-preserving mappings between functors.

Definition 2.4.5. *If C and D are two categories, then a functor, $F : C \rightarrow D$ is a mapping that assigns each object a in C to an object $F(a)$ in D , and each morphism $f : a \rightarrow b$ to a morphism $F(f) : F(a) \rightarrow F(b)$ such that:*

1. $F(1_x) = 1_{F(x)}$ for all objects x in C , and;

⁶These lemmas, while central to category theory, only appear as a tool for the proof in Section 3.4.3. For a complete statement and proof you can refer to [93].

2. $F(g \circ f) = F(g) \circ F(f)$ for all morphisms, $f : a \rightarrow b$ and $g : b \rightarrow c$ in C .

By duality, functors come in two breeds. “Co-variant” functors are exactly as described above and preserve the directions of morphisms. In contrast, “Contra-variant” functors reverse the directions of arrows in our codomain, but behave exactly like regular, co-variant functors in all other respects.

At this point, you may notice that these functors seem to behave quite similarly to the morphisms already defined above. This is because of the restriction that they must preserve structure, i.e. *composition*. Indeed, given a category, C , one can always define a functor $1_C : C \rightarrow C$ that maps every object and morphism in C to itself. It’s also quite straightforward to show that functors necessarily compose:

Lemma 2.4.1. *Let $F : C \rightarrow D$ and $G : D \rightarrow E$ be any two functors. Then there exists a functor $G \circ F : C \rightarrow E$*

Proof. To show that $G \circ F$ exists, all objects in C must be mapped to some object in E . This follows directly from definition by taking $G(F(a))$ for any a in C .

Demonstrating that all morphisms are preserved is the more interesting part, as both conditions in Definition 2.4.5 must be satisfied. First, we note that $G(F(1_a)) = G(1_{F_a}) = 1_{G(F(a))}$, by the functorality of both F and G . Similarly, $(G \circ F)(g \circ f) = G(F(g \circ f)) = G(F(g) \circ F(f)) = G(F(g)) \circ G(F(f))$, as desired. \square

Putting this all together, we can see that it’s possible to construct a category with categories as objects and functors as morphisms. This “category of all (small)⁷ categories” is written as **Cat**⁸, and its existence is the first example of how category theory can become very “meta” very quickly.

For a more grounded example of a functor, let’s consider `List`—the class of lists in some arbitrary programming language. More formally, let L be the category that we use to represent our language, with `Type`s as objects and functions as arrows. Then there exists a functor, `List` :: $L \rightarrow L$ that maps each `Type`, `a`, to a new `Type`, `List a` (i.e. `[a]`) (as a quick note, functors, and in fact all morphisms that have the same source and target, are called endofunctors and endomorphisms). But what about the morphisms? Well, morphisms in L are functions,

⁷At this point I feel the need to point out that there’s some technical details about what is meant by “all” categories. For our purposes, when talking about categories, I will only be discussing “small” categories, where `ob(C)` and `hom(C)` are both sets. “Large” categories do exist, where `ob(C)` and `hom(C)` are *classes*, but understanding this distinction is unnecessary for my area of work.

⁸PS: take that, Russell!

so we need an operation that takes a function $f :: a \rightarrow b$ to some new function with Type $[a] \rightarrow [b]$. Namely, some operation $fmap :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$ that takes f and maps it to $fmap f :: [a] \rightarrow [b]$ ⁹. This familiar operation is more commonly known as `map`.

Intuitively, (endo)functors in programming behave like containers for our Types. We use a special Type constructor (in the above case, `List`) to place some arbitrary Type (a) into a “box” (`List a`). This constructor is special because it requires a polymorphic function, `fmap`, that describes how to pass any function on any Type into our container. Further, since functors can be composed, we immediately see that we can have nested containers without issue.

2.4.5 Natural Transformations

Extending this analogy of the container, it is natural to ask if there are ways to move the contents from one container to another *without violating the integrity of our containers*. This is not only possible, but happens frequently. These special mappings are called “natural transformations,” and are defined as follows:

Definition 2.4.6. *A natural transformation $\eta : F \rightarrow G$ is a collection of morphisms in D such that, if F and G are both functors that map from categories C to D , then:*

1. *every object, a in C , has a corresponding morphism, $\eta_a : F(a) \rightarrow G(a)$ in D . This is called the “component” of η at a .*
2. *every morphism, $f : a \rightarrow b$ in C , satisfies $G(f) \circ \eta_a = \eta_b \circ F(f)$.*

Natural transformations appear everywhere in mathematics and computer science. The determinant of invertible matrices, the `len` function for lists, and the abelianisation of a group are all instances of natural transformations. Here’s a simple example of a natural transformation that could appear in a first-year coding course. Let η be the polymorphic function

$$\text{rev} :: [a] \rightarrow [a]$$

that takes in a list, $[a]$, and produces the list in reverse order. We can check that `rev` is a natural transformation. First, we know that any singleton list can be reversed. Thus, `rev` always has a component for any a . Second, for every function, $f :: a \rightarrow b$, we have

$$(fmap f) . \text{rev} :: [a] \rightarrow [b]$$

⁹If we weren’t dealing with finicky computers, we’d simply reuse `List` as the name for this mapping of functions, rather than `fmap`.

which reverses the elements in our list of Type `[a]`, then maps each element in that list to an element of Type `b`, producing a list of Type `[b]` with the elements reversed from how they would be if we had only applied `(fmap f)`. However, if we decided to map over `[a]` first, then reverse afterwards, we would have the exact same thing, *i.e.*

$$(fmap f) . rev = rev . (fmap f)$$

as desired.

It probably won't surprise you at this point that natural transformations also compose, and pulling this thread leads to the topic of 2-categories, then n -categories, then ∞ -categories—which are generally outside the scope of this report. However, 2-categories, or more specifically a generalisation of 2-categories called *bicategories*, are fundamental to lenses and “learners,” which are the topic of Chapter 4.

2.4.6 2-Categories and Bicategories

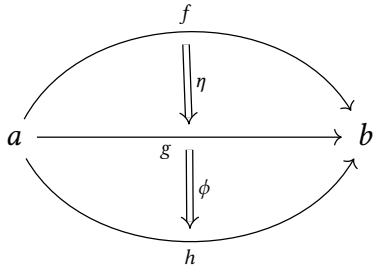
As the name suggests, a 2-category is an idea that extends the notion of a category. Where categories have objects and morphisms, 2-categories have:

- 0-cells (objects),
- 1-cells (mappings between 0-cells/objects) with arrows written as \rightarrow , and;
- 2-cells (mappings between 1-cells/morphisms) with arrows written as \Rightarrow .

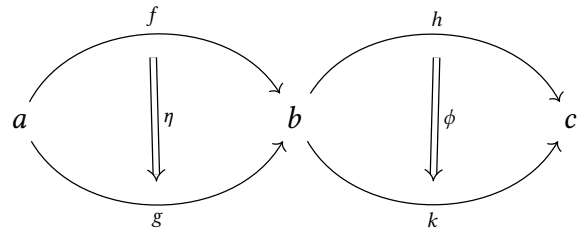
We've already seen the classic example of a (strict) 2-category: **Cat**. Its k -cells are each of the fundamental concepts covered above—0-cells are categories, 1-cells are functors between categories, and 2-cells are natural transformations between those functors.

Typically, we refer to the composition of 2-cells along objects as “horizontal” composition and the composition of 2-cells along morphisms as “vertical” composition. As you'd expect, there are a variety of rules to ensure that 1-cells and 2-cells properly compose. Whether the laws for composition hold up to equality or only “coherent isomorphism” dictate whether the 2-category is “strict” or “weak” with weak 2-categories being called bicategories¹⁰. The full conditions for being a (strict) 2-category and bicategory can be found in [2] and [1] respectively, though I've included some diagrams in Figure 2.8 to illustrate some of the key properties (particularly of the less familiar 2-cells).

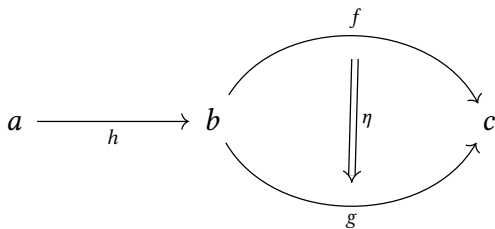
¹⁰“2-category” can typically refer to either a strict or weak 2-category, however the existence of the term “bicategory” means that “2-category” usually refers to the strict kind.



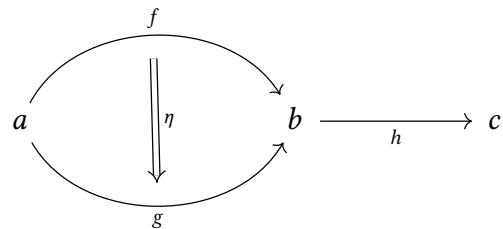
(a) Vertical composition of 2-cells, which is expressed as $(\phi \cdot \eta) : f \Rightarrow h$. Like with composition of 1-cells, vertical composition is associative and has identities.



(b) Horizontal composition, which is subject to the “interchange law” with vertical composition. This would be written as $(\phi \circ \eta) : (h \circ f) \Rightarrow (k \circ g)$.



(c) Left “whiskering” expressed as $(\eta \triangleleft h) : (f \circ h) \Rightarrow (g \circ h)$.



(d) Right “whiskering” expressed as $(h \triangleright \eta) : (h \circ f) \Rightarrow (h \circ g)$.

Figure 2.8: Various properties of how 2-cells interact in 2-categories. Technically, it’s redundant to define both horizontal composition and the pair of whiskering operations, because one can be derived from the other. However, it is useful to visually see how whiskering is the result of “collapsing” either side of a horizontal composition (technically composing with an identity on either side).

2.4.7 Monoids and Monoidal Categories

The last concept I want to cover are monoids and, more specifically, monoidal categories. For those familiar with algebraic groups, a monoid is simply a group where elements may not necessarily have inverses. They appear all over the place. A few examples are

- the Naturals, \mathbb{N} , which form a monoid under addition (because of the absence of negative numbers);

- the powerset of a given set, A , is a monoid under both set intersection and set union, and;
- list construction, $:$, with $[\]$ as the identity.

In category theory, monoids continue to be ubiquitous. They appear via the broad class of monoidal categories.

Definition 2.4.7. *A monoidal category is a category $(C, \otimes, I, \alpha, \lambda, \rho)$ where*

- $\otimes : C \times C \rightarrow C$ is the monoidal, or tensor product,
- I is the monoidal unit, or identity for \otimes ,
- α is a natural isomorphism for any a, b, c that ensures \otimes is associative, and;
- λ and ρ are two natural isomorphisms that ensure I is both a left and right identity for \otimes .

Coherence conditions, or diagrams, also ensure that these natural isomorphisms are “well-behaved” when interacting with other operations. These conditions can be found in [8].

For brevity, I will typically write (C, \otimes, I) , leaving the natural isomorphisms as contextually obvious.

Note that, by this definition, we only have associativity and identity up to isomorphism. That is, rather than $a \otimes (b \otimes c) = (a \otimes b) \otimes c$, we have $a \otimes (b \otimes c) \cong (a \otimes b) \otimes c$. This can be seen in the following example.

Example 2.4.1. Set is a monoidal category, where \otimes is the Cartesian product, \times , and *any* singleton set can act as the identity.

More examples of monoidal (and bimonoidal) categories will feature in both Chapter 3 and Chapter 4.

2.5 Applied Category Theory

While category theory is extremely abstract, many researchers have found a great deal of utility in applying category theory to more concrete domains. The benefit in doing so is largely twofold. First, as mentioned previously, category theory provides a great language for generalising and simplifying many structural relationships found within an area of study. Second, the capacity of

category theory to make statements about itself, via functors and natural transformations, lends itself to formalising the idea of *analogy* in research. Being able to draw upon inspiration from one domain, and use results from that domain to solve problems in a new domain, has been a growing method of solving problems in mathematics, and the same approach can work in more applied areas of research. Applied category theory (ACT) has benefitted physics [11], control theory [15], and probability theory [12, 111].

As some of the earlier examples highlight, computer scientists have also noticed the applicability of category theory. This really shouldn't be too surprising though—the nature of category theory is to break complex structures into their smallest pieces, then assemble them such that the target of one mapping can be used as the input to another mapping. This understanding of code is very functional (pun intended), and indeed many notions in functional programming, such as monads, lenses, and algebraic data types, are categorical in nature.

Here I provide three concrete examples of how applied category theory has influenced thinking in computer science. First, I give another example of how category theory is useful in programming by describing algebraic data types. Then, I describe how researchers in cryptography have managed to apply category theory to improving the security of cryptographic schemes. Finally, I describe how category theory is used to imagine databases in a new manner that doesn't require side-effects/mutability. This last example will informally introduce the notion of a “lens”.

2.5.1 Algebraic Data Types

Let's revisit `List`. We discussed how we can view `List` as a functor, but many programmers are probably more comfortable with the following definition of `List`:

```
data List a = [] | a:(List a).
```

Here, `data` simply indicates `List` is a new `Type` constructor on `a`. But what about the bar? This can be read as “or,” and means that `List` is something called a “sum Type”. Read in plain English, the data definition for `data List a` says that a list is either the empty list *or* it is a list that is the result of appending an element of `Type a` to a pre-existing list. We can generalise sum Types as being combinations of pre-existing Types that work as a kind of “disjoint union,” however what these Types really are is an instance of a coproduct—the dual notion of a product in category theory. Applying functions to sum Types means that those function definitions will be expected to contain pattern-matching (“do *x* if the input looks like *A*; do *y* if it looks like *B*”), hence the reading of the bar operator as “or.”¹¹

¹¹What's *really* happening here is that logical or is another instance of a coproduct in the category of `Bool`.

If sum Types are just coproducts constructed with `|`, then it is natural to ask if there are product Types, and indeed there are—products! Any Type of the form `pair a b = (a,b)` can be viewed as a product of `a` and `b`. With this in mind, we can see with the example of `List` that the list constructor, `:`, is acting as nothing more than syntactic sugar for a pair constructor, where the first element of the pair is an object of Type `a`, and the second object of the pair is an element of Type `List a`.

So, “algebraic data types” are essentially sum Types of product Types (note the order here corresponds to typical order of operations in arithmetic). To reinforce this connection, it is common to use `+` to refer to sum Types, and `×` to refer to product Types. However, we can take this idea even further. Remember that we have special objects in arithmetic that correspond to our two operators: the additive and multiplicative identities, `0` and `1`, respectively. In our category of Types, these identities correspond to our initial and final objects. To see how these relate, consider `List`: the number of ways to map any list to the empty list; there’s only one, and it’s by mapping the whole list to `[]`. Thus, `[]` is a final object in our category of Types and we can express it as `1` (we could also define an initial object called `Void` that can be written as `0`).

Putting this all together, we can re-express `List` in the following way:

```
data List a = 1 + a × List a.12
```

What’s more exciting is that these objects in our Type system behave exactly as one would expect `0` and `1` to behave in arithmetic (ex. $1 \times a \approx a$). Even more sophisticated notions in arithmetic, such as exponentiation, can also be generalised to Types. Actually, functions themselves are these exponential objects in our Type algebra.

2.5.2 Cryptography

To pivot out of theoretical computer science, I want to discuss cryptography since its real-world applications are very clear. However, I want to start this discussion from a slightly different perspective. Imagine a group, `g`, in the classic mathematical sense—a collection of objects with some operation, such that one object, `1` or `ε`, is an identity, and every object, `a ∈ g`, has an inverse `a-1`. As stated earlier, there exists a category called **Grp**, that contains all such groups.

It is possible to describe a mapping from **Grp** to **Set**, that “forgets” the group structure of any `g` in **Grp** so that all that remains is an unstructured set of objects. This mapping, $U : \mathbf{Grp} \rightarrow \mathbf{Set}$

¹²As a fun exercise, you can try rearranging this expression and using the geometric series formula to get `List a = 1 + a + aa + ...` which is saying that a list can be the empty list *or* a singleton list *or* a list with two elements, and so on.

is a functor (U is the canonical name, and stands for *underlying*) and functors of this kind appear all over in mathematics. However, is it possible to describe a mapping in the opposite direction? That is, can we take a set, s , and map it into **Grp** such that there's some sort of (“weak”) inversion of U ?

As it turns out, there is, and this functor is called the “free” functor, $F : \mathbf{Set} \rightarrow \mathbf{Grp}$. It takes the elements of a set, s , and uses them as “generators” to create new elements. First, for each element, $x \in s$, there is an associated x^{-1} . Next, we add an identity, ϵ . All these elements are “joined together” by the group operation to create new elements (*i.e.* if a and b are elements then we can write ab or ba as another element. This can't be expressed as another element, c , as this would imply some additional structure beyond the “bare minimum”). Intuitively, we can think of the free group of a set, s , denoted $F(s)$ to be the set of “words” generated by s , with ϵ being the empty word. What is interesting about these free groups is that they act as the “least structured” version of a group that contains elements appearing in s .

Interpreting free groups in this way, it is clear that there are no finite free groups (as one can always find a new element in $F(s)$ by appending any element in s to a pre-existing word). This illuminates a problem with thinking about F as being the “opposite” of U , in that the two aren't actually inverses of one another. Indeed, it isn't the case that for any group, $g \in \mathbf{Grp}$, that there exists an $s \in \mathbf{Set}$ such that $F(s) = g$. Finite-sized groups are just one instance of the class of groups that have no pre-image in **Set**. However, F and U do exist in a relationship known as “adjunction,” which can be seen as a weak form of equivalence.

An adjunction between two functors, $F : D \rightarrow C$ and $U : C \rightarrow D$, exists when there exists a natural isomorphism (a natural transformation that is invertible), $\Phi : \text{hom}_C(F-, -) \rightarrow \text{hom}_D(-, U-)$ where specific morphisms in Φ look like

$$\Phi_{ab} : \text{hom}_C(Fb, a) \rightarrow \text{hom}_D(b, Ua)$$

for all objects $a \in C$ and $b \in D$. Here, F is called the *left* adjoint, while U is the *right* adjoint¹³. In the example of our free and forgetful functors, this means that the set of homomorphisms from the free group, $F(b)$ to any other fixed group, a , is isomorphic to the set of functions from our generator of the free group to the underlying set of a , $U(a)$.

I prefer to think of adjunctions as the basis for rigorously understanding analogies. However, there's an intuition for adjunctions that is couched in more computational language. This intu-

¹³As an aside, one of my favourite philosophical things about category theory is how rigorously it answers the question: “What does it mean for two things to be the same?” To be the exact same (*i.e.* equal) means that the mapping between them is the identity, 1_x , implying the strictest version of $x = x$. To be the same *in all important ways* (*i.e.* equal up to isomorphism) means that there's two mappings such that the composition on both sides gives the identity. Now, with adjunctions, we see a new way of being “the same”: *almost* the same, or the same *by way of analogy*.

ition comes from asking two related questions. First, *what is the most efficient way to solve a given problem in a structured/algorithmic way* (you can see how this might loosely corresponds to asking what is the “best” way to make a group out of a set)? The corresponding second question is, *for a given algorithm is there a problem for which that algorithm is the best solution?* To continue the analogy to the free and forgetful functors, you might imagine a problem as being the *underlying* input that an algorithm is running on.

So how does this relate to cryptography? Well, groups are often used for cryptographic schemes. Ideally, we would like to construct cryptographic schemes that are difficult to break. Often this corresponds to selecting groups such that there is as little structure as possible for an adversary to use to find solutions to non-trivial equations (think equations of the form $m^e \equiv c \pmod n$ in RSA encryption [129] where m is a message, e is our encryption key, and c is the resulting cypher in \mathbb{Z}_n). Minimising the amount of group structure should then reduce the ways that an attacker can attack a given scheme.

This sounds exactly like what a free group could be used for, since they’re the *least* structured groups! Unfortunately, infinitely-sized groups aren’t much use in a computational sense. However, recent researchers have proposed the idea of “pseudo-free” groups [128] as a way of constructing cryptographic schemes that are strong. These are finite groups that are indistinguishable from a free group by any polynomial time adversary.

2.5.3 Databases

The final example I wish to discuss is how one might implement a database in a functional language. A database is somewhere that a user can store information in a persistent manner that can then be accessed at a later point in time. However, functional languages lack the notion of some persistent state. Further, when storing and referencing large amounts of data, being able to modify single entries in a database is extremely useful. However, since side effects—thus mutability—are prohibited, it may seem challenging to use functional languages to build a database. Category theory provides an elegant solution to this challenge with “lenses” (and more broadly, “optics”).

First, let’s consider what operations are fundamental to the workings of databases. First, we wish to obtain particular entries from a database so that lookups are possible. That is, for a whole database, d , and entry, e in d , we wish to have a function, $\text{read} :: d \rightarrow e$. Second, we want to be able to modify some value, e' , in d to obtain a new database, d' , with the selected entry changed. This is a pure function, $\text{write} :: d \rightarrow e' \rightarrow d'$ ¹⁴.

¹⁴We will later see `read` and `write` appear as `fwd` and `bwd`, respectively.

These two operations provide a high-level view of how to peek inside databases, and modify parts of them, but it is useful to think of the properties that must be satisfied for our database to be “well-behaved.” Thus, we define “lenses” as pairs, `read` and `write`, that satisfy the following three laws.

First, we wish to ensure that a given value, `a`, is returned if you try to read it immediately after writing it in the database. So, `read (write d a) = a`. Second, writing a value as `a`, then immediately writing it as `b` should be the same thing as if you had just written `b` initially. This is expressed as `write (write d a) b = write d b`. Finally, reading a particular value from a database and writing that value should do nothing. Using our category theoretic language, we have that `d = write d (read d)`, implying that $1_d = \text{write} \cdot \text{read}$. With these properties being satisfied, we can ensure that these lenses satisfy the identity, associativity, and composability laws of a category. A tutorial for how to implement a database using lenses can be found in [7], along with a more detailed explanation of their value. In Section 4.2.2 of Chapter 4 we will define lenses formally for any given category.

Chapter 3

A Categorical Foundation for Vector Symbolic Architectures

3.1 Introduction

The rapid adoption of machine learning in software engineering has largely outpaced the rate at which we understand many learning algorithms. To better understand these algorithms, two main approaches have been taken. The first is to make complex models more “interpretable” to human users. The second, more recent approach, has been to improve our understanding of the mathematical foundations of machine learning. While the mechanics of differentiation and matrix multiplication have been well-understood for a long time, these provide little perspective on the *emergent* behaviour and structure of large-scale algorithms.

In the past few years, considerable strides have been made in developing a category-theoretic perspective on machine learning and deep learning [24,52,141]. A lot of this work has focused on supervised learning, where models learn through exposure to input/output training pairs. Here, lenses have been identified as providing the right structure for explaining the bi-directional flow of forward-propagating computations and backward-propagating errors. Parameterising these lenses yields a means to modify the computation of a model.

Less progress has been made on understanding learning (and, more generally, cognition) from the perspective of information storage and association. Artificial feed-forward neural networks do a good job of capturing the hierarchical structures in brains, but brains are very complicated, and neural connectivity is often highly recurrent and seemingly chaotic. This complexity has prompted some researchers to look at neuronal computation at the level of “pop-

ulations” of neurons and the dynamics contained within. Examples include reservoir computing [135, 153], neural engineering frameworks [36], and the focus of this paper: vector symbolic architectures/algebras, or “VSAs”.

As the name implies, VSAs are fundamentally vector spaces that aim to encode symbols with an associated “meaning” as vectors. They possess a means of comparing how similar vectors are, and two operations—binding/unbinding and bundling—that produce new vectors corresponding to more sophisticated symbols. Maintaining (near-)orthogonality is critical to ensuring that symbols can still be distinguished from one another. The advantages of VSAs are evident in their ability to store information: given a vector space of dimension d , 2^d near-orthogonal vectors can be encoded with a high degree of noise-resistance [63]. Further, for researchers who want to study learning as it occurs in biological brains, VSAs have been implemented in spiking neuron models [32, 33, 36, 37, 42–44, 49, 58, 80, 107, 125, 149], demonstrating their biological plausibility.

VSAs also aim to describe cognitive features such as *compositionality* that are absent from some other connectionist models [41]. In spite of this ambition, discussion of VSAs in categorical terms is largely absent from the literature; I provide a literature review in Appendix C. This thesis is an initial step in understanding VSAs from a category-theoretic perspective. I provide a list of desiderata for properties of VSAs, similar to the work done in [24] for supervised learning. My primary contribution is generalising from vectors to co-presheaves, and then deriving “optimal” VSA operations using Kan extensions. This formalisation determines the proper choice of bind and bundle operations.

3.2 Background

VSAs¹ address the problem of how human behaviour can be characterised by rules and symbolic reasoning, while being implemented in neural networks, which rely on manipulating vector representations and engage in similarity-based reasoning [146]. VSAs use randomly generated vectors to represent variables and the values they take on (also called “slots” and “fillers”). These base symbols are composed into higher-level representations using a fixed set of algebraic operations.

¹Terminology attributed to [54], VSAs are also known as Hyperdimensional Computing (HDC) [74], and more recently, Vector Symbolic Algebras [48].

3.2.1 VSA Definition

Individual architectures can be described by the types of vectors that they operate on and their corresponding operations: similarity, (un-)bundling, (un-)binding, and braiding [53]. Frady *et al.* note that bundling and binding “are dyadic operations that form a ring like structure” [44]. Additionally, there are operations which may assist in implementing the algebra, namely the (pseudo-) inverse of vectors, and “cleanup” operations for de-noising. While not commonly emphasised in VSA literature, vectors can be multiplied by scalars, including negative numbers.

For any particular VSA, the vector space, or domain manifold within, determines the choice of operators and the space in which the base vectors are generated. This can include $\{-1, 1\}^d$ [53], $\{0, 1\}^d$ [44, 73, 83, 120], \mathbb{R}^d [50, 59, 116], and \mathbb{C}^d [43, 44, 115], where d represents the dimensionality of our space, sometimes in the order of 10,000. It is also not unusual for there to be further constraints placed on base vectors. For example, the Fourier Holographic Reduced Representations (FHRR; [116]) algebra (see Table 3.1 for a list of VSA names and abbreviations) requires that the complex numbers comprising the vector have magnitude one, forcing all datapoints to live on a hypertorus, and HRR vectors must be unitary. The Binary Spatter Code (BSC; [73]) and Multiply-Add-Permute (MAP) VSAs require atomic vectors that live on the vertices of hypercubes. Other VSAs enforce sparsity [120] or a specific block structure [44, 83] in their atomic vectors.

| Initialism | VSA Name | Citation(s) |
|------------|------------------------------------------------------|--------------------|
| TPR | Tensor Product Representations | [147] |
| MAP | Multiply Add Permute | [53] |
| HRR | Holographic Reduced Representations | [115, 117] |
| FHRR | Fourier Holographic Reduced Representations | [115, 117] |
| BSC | Binary Spatter Code | [73] |
| BSDC | Binary Sparse Distributed Codes (or Representations) | [44, 83, 119, 120] |
| VTB | Vector-derived Tensor Binding | [59] |
| MBAT | Matrix Binding of Additive Terms | [50, 155] |

Table 3.1: A list of the considered VSAs, clarifying their naming conventions, and providing references to important early papers.

3.2.2 VSA Operations

The following operations are the fundamental components of a VSA. While not all need to be present in every architecture (for instance, braiding), these describe the functions and properties that most practitioners consider useful.

Similarity: $\text{sim} : X \times X \rightarrow \mathbb{R}$. This function takes two vectors and returns a number that measures similarity. Quite commonly, VSA similarity is provided by cosine similarity. In the case of binary vectors, Hamming distance or overlap are used. In all cases, similarity is bounded, with extremum (usually one) indicating equivalence between vectors. While we may not always be working with exact values, for the purposes of this document I will denote equivalence between vectors as $\text{sim}(x, y) = 1$, and indicate maximum dissimilarity between vectors as $\text{sim}(x, y) = 0$. I use $x \sim y$ to denote two vectors that are similar.

Bundling: $\oplus : X \times X \rightarrow X$. The bundling operation takes two vectors and produces a new vector that, importantly, maintains some degree of similarity to its constituent parts. Given a bundle $z = x \oplus y$, we expect that both $\text{sim}(x, z)$ and $\text{sim}(y, z)$ to be high, although possibly not as high as $\text{sim}(x, x)$ or $\text{sim}(y, y)$. Note that, for some VSAs, the bundling operation comes with some additional normalisation term which ensures that the magnitude of the bundles never exceed some specified range (*e.g.* the unit hypersphere, vertices of the unit hypercube).

Binding: $\otimes : X \times X \rightarrow X$. Binding transforms two vectors into a new vector that is not similar to either constituent element. Given a bound vector $z = x \otimes y$, we expect that $\text{sim}(x, z) \sim 0 \sim \text{sim}(y, z)$. While it is commonly the case, binding is not always commutative or associative in all VSAs (*ex.* Smolensky’s tensor product [44, 53, 116]).

Unbinding: $\oslash : X \times X \rightarrow X$. The purpose of unbinding is to undo the binding operation, *i.e.*, if $z = x \otimes y$, then $x \oslash z = y$ and $z \oslash y = x$. Despite some VSAs distinguishing this operation from binding, it’s also acceptable to have some value $k = x^{-1}$ such that $k \otimes z = y$ (or some right inverse for the other case). Depending on the chosen algebra and vector space, the vectors can be *self-inverse* or not, with respect to binding [134]. In practice, “lossy” binding may mean that only a pseudo-inverse can be found. In this case, then $x^{-1} \otimes z \sim y$, whereas for exact inverses $x^{-1} \otimes z = y$ holds. This problem is often addressed using an additional “cleanup” operation. I expand on the relationship between binding, unbinding, and inverses in Section 3.4.

Braiding: $\text{orth} : X \rightarrow X$. Braiding (alternatively, *hiding*) is an invertible unitary operation that produces a new vector that is dissimilar to the input vector, *i.e.*, $\text{sim}(x, \text{orth}(x)) \sim 0$. Braiding was introduced by [53] to deal with a number of problems in VSAs that only supplied binding and bundling. Specifically, if the binding operation is commutative, or if a given vector is its own inverse, then it is not possible to impose hierarchical structure on encoded representations. As I will discuss below, the need for a braiding operation may be an artefact of choosing a commutative binding operator.

3.2.3 Survey of Commonly Used VSAs

Table 3.1 provides a list of VSA acronyms, and references that describe them. Smolensky’s Tensor Product Representations [147] is the first VSA. It uses the tensor product as a binding operation that is not commutative, but is self-inverse. However, the tensor product has the practical limitation that the dimensionality of the vector representation grows exponentially with the number of applications of the binding operator. Consequently, when Gayler [54] defined vector symbolic architectures, he focused on algebras with dimensionality-preserving operations. Schlegel *et al.* [134] surveyed different VSAs, comparing different operations, and I provide a modified version in Table 3.2.

Note in Table 3.2 that not all VSAs require a braiding operator. Typically, braiding is required when the binding operation is commutative. Without being able to distinguish between $x \otimes y$ and $y \otimes x$ using the similarity operator, it becomes difficult to implement more complex structures like trees or graphs, as I discuss below. This is further compounded if the binding operation admits a self-inverse. In the case of the (F)HRR, VTB, and MBAT algebras, one way to impose some structure is to repeatedly bind a vector with itself; in the case of self-inverting representations this would not be possible.

The binding operator for the TPR, MAP, (F)HRR, BSC, and MBAT VSAs are fairly straightforward, but it is worth explaining binding for the VTB and the BSDC VSAs. For VTB, when two vectors, x, y , are bound, one vector is used to construct a matrix, $V(x)$, with a block diagonal structure, and then the bound vector is constructed $z = V(x)y$. For BSDC-S vectors are bound by circularly shifting one vector by an integer number of steps derived from the other vector. The BSDC-CDT (Context-Dependent Thinning) [119, 120] binding operator is unique in that it produces new vectors that are not orthogonal to their inputs. For the BSDC-SEG VSA, the vectors are divided up into blocks, and the shifting process is applied block-wise [83], which is equivalent to circular convolution applied block-wise for maximally sparse blocks [44]. Frady *et al.* [44] proposed other sparsity-preserving binding operations for sparse block codes.

| Algebra | Base Vector | Domain | Similarity | Bundling | Binding | Inverse | Braiding |
|----------|------------------------------------------|----------------|--------------------|-------------------|----------------------|------------------------------------|----------------------------------------|
| TPR | $\{-1, 1\}^d$ | \mathbb{R}^d | cosine similarity | Vector addition | Tensor product | $x^{-1} = x$ | – |
| MAP-I | $\{-1, 1\}^d$ | \mathbb{Z}^d | ” | Elem. addition | Hadamard prod. | $x^{-1} = x$ | permutation |
| MAP-B | ” | $\{-1, 1\}^d$ | ” | ” + threshold | ” | ” | ” |
| MAP-C | $[-1, 1]^d$ | \mathbb{R}^d | ” | ” + cutting | ” | ” | ” |
| FHRR | e^{ia} $(a_i) \in [-\pi, \pi]$ | \mathbb{C}^d | inner product | ” | ” | $(x^{-1})_i = 1/x_i$ | permutation or bind with random vector |
| HRR | $(x_i) \sim \mathcal{N}(0, \frac{1}{d})$ | \mathbb{R}^d | cosine similarity | ” + normalization | circular convolution | $(x^{-1})_i = x_{(d-i+1) \bmod d}$ | ” |
| MBAT | ” | \mathbb{R}^d | ” | ” + normalization | Matrix mul. | Matrix Inv. | – |
| VTB | ” | \mathbb{R}^d | ” | ” + normalization | VTB | transpose VTB | – |
| BSC | $\{0, 1\}^d$ | $\{0, 1\}^d$ | Hamming distance | AND | XOR | $x^{-1} = x$ | permutation |
| BSDC-S | ” | ” | Normalized Overlap | OR | Shifting | Reverse shift | – |
| BSDC-SEG | ” | ” | ” | ” | Segment Shift | ” | – |
| BSDC-CDT | ” | ” | ” | ” | OR + CDT | – | – |

Table 3.2: A modified version of Table 1 from Schlegel *et al.* [134]. I adopt the notation of Schlegel *et al.*, annotating the MAP VSAs to indicate the different domains vectors are permitted to exist in. Note that these different spaces come equipped with slightly different bundling operations, to ensure the produced vectors stay in the desired space. Similarly, I use Schlegel *et al.*’s notation to distinguish BSDC VSAs that use different binding operations. The unwieldy size of the table, being unable to contain within the margins of this document, is a happy coincidence that conveys the abundant variety of VSAs, and the need for unifying principles.

3.2.4 Example Problem: Composing Functions

There are a number of ways to compute functions in a VSA. One way is to take the set definition of a function, $F = \{(x, f(x)) \mid x \in X\}$ and encode it as a bundle over the domain and range of the function:

$$F = \bigoplus_{x \in X} (x \otimes f(x)), \quad (3.1)$$

$$\Leftrightarrow f(x) \sim x \oslash F. \quad (3.2)$$

Remember that \oslash refers to the unbinding operation, so the second expression is essentially querying what the output ($f(x)$) of the function F is for input x . For computer scientists, this equivalence is really just the idea that functions can be curried and uncurried. To compose two functions, f and g , we would write:

$$(f; g)(x) \sim ((x \oslash F) \oslash G). \quad (3.3)$$

In this case, the function value should be approximately equivalent to the value we would have obtained had we just implemented $(f; g)(x)$ directly.

Because some unbinding operations are only approximate and we're manipulating (potentially) large bundles of only pseudo-orthogonal vectors, there is always a risk of cross-talk noise. Consequently, it is sometimes the case that clean-up operations are interjected between stages, *i.e.*,

$$(f; g)(x) = \text{cleanup}(\text{cleanup}(x \oslash F) \oslash G). \quad (3.4)$$

Here, the cleanup operation projects a noisy vector back to the closest point that is one of the original $(x \otimes f(x))$ that were bundled together. More examples of VSA computations can be found in Appendix D.

3.3 Desiderata

Here I characterise the fundamental properties that enable VSAs to represent concepts and perform computations, and which any categorical formalisation should respect. The first two are strictly necessary. The third arises from practical considerations, but any formalisation based on category theory should also capture the cases where it is needed.

- 1. Similarity:** The first necessary property of a VSA is the notion of how “similar” two vectors are (or aren’t). We wish to quantify how closely related concepts are to one another. We also need a means of ensuring that a “collision” hasn’t occurred—that is that two vectors aren’t so similar that they are difficult to distinguish.
- 2. Two binary operations:** We’d like to have an operation that produces a vector that is similar to its operands, and one that creates a vector that is dissimilar. We’ve been referring to these operations as bundling and binding, respectively, and denote them as \oplus and \otimes . Further, we require that it is possible to “undo” both operations.
- 3. Dimension Preservation:** An important practical concern when building VSAs is that the dimension of our vector space remain constant. It is understood that two vectors can be bound and bundled without losing any information in Smolensky’s TPR however, as mentioned in Section 3.2.3, this results in vectors from a d^2 -dimensional space, given operand vectors from a d -dimensional space. To alleviate space constraints, practitioners need bind and bundle operations that always produce vectors with a fixed size.

Ultimately, this work manages to describe a principled way of determining binding and bundling operations, while still being able preserve the dimensionality of our underlying vector space. Similarity, unbinding, and unbundling are left as future work.

3.4 Formalising VSAs Using Category Theory

A VSA requires two binary operations. One must be reversible, and the other must distribute over the first. This requires our objects to be at least a ring. Because we want to perform unbinding and unbundling, this ring should be a *division* ring. Note that this formalisation necessitates that binding, and bundling, are invertible. Unfortunately, describing division rings categorically is highly non-trivial (see [4] for a discussion of how to describe a (commutative) division ring—*i.e.* field—as a category).

At first glance, I assumed that describing separate and distinct operations for unbinding and unbundling was a form of conceptual obfuscation. This assumption was motivated by observing that operations like subtraction and division don’t *really* exist in division rings; they are simply addition and multiplication of inverses, respectively. However, the difficulty of describing inverse operations for bind and bundle categorically suggests that there may be interesting reasons to distinguish bind and bundle from the operations that undo them.

My strategy here is to decouple our vectors into indices and values. This generalises from vectors to co-presheaves². With this, I isolate the rig structure in our VSAs as arising from the rig operations of the category from which we draw our values (a definition of a rig and a rig category are provided in the next section). This decoupling gives us the capacity to include both dimension preserving and non-dimension preserving VSAs. I formalise the bind and bundle operations as right Kan extensions to the external tensor product and sum, respectively.

3.4.1 Generalising to Co-presheaves

The core concept here is splitting the “vectors” up into their indices and values, only requiring that the indexing category is Cartesian monoidal, and the values category being a rig, or bimonoidal category. Loosely, a rig is mathematical object that has two operations. One operation, say plus (+), needs to be a commutative monoid, while the other, times (\cdot), only needs to be a monoid. A rig requires that times distributes over plus, but neither operation needs to be invertible. In short, a rig is a ring without the “n” (additive negation). Categorifying this notion gives us the following:

Definition 3.4.1. *A rig, or bimonoidal, category is a category, \mathcal{C} , equipped with a commutative monoid, \oplus with identity 0, and a monoid, \otimes with identity 1, that satisfy (via natural isomorphisms) distributivity:*

$$d_\ell : x \otimes (y \oplus z) \rightarrow (x \otimes y) \oplus (x \otimes z)$$

$$d_r : (y \oplus z) \otimes x \rightarrow (y \otimes x) \oplus (z \otimes x),$$

and annihilation:

$$a_\ell : 0 \otimes x \rightarrow x$$

$$a_r : x \otimes 0 \rightarrow x.$$

These natural isomorphisms must satisfy the appropriate coherence conditions. This definition, and those coherence conditions can be found at [10; 87, Section 1; 75, pp. 323-346].

Returning to VSAs, I generalise from vectors to functors $v : \mathcal{I} \rightarrow \mathcal{V}$ from a (closed) Cartesian monoidal category, $(\mathcal{I}, \otimes, I)$, to a rig category, $(\mathcal{V}, \cdot, +, 1, 0)$. The action of v on an object $i : \mathcal{I}$ is written as v_i . Since the fundamental “vectors” of a VSA are my primary focus, I refer to these functors as co-presheaves to highlight my interest in the arrow category, $[\mathcal{I}, \mathcal{V}]$. For example, we could describe vectors in a three-dimensional vector space, where \mathcal{I} would be the discrete

²“Co-presheaves” are conceptually identical to covariant functors. I prefer the term co-presheaves to highlight that, even though we need access to both indices and values, our ultimate interest is the objects in $[\mathcal{I}, \mathcal{V}]$: “vectors.”

category whose objects are the natural numbers, 1, 2, 3, while \mathcal{V} could be the non-negative Reals, with objects as the corresponding numbers, \leq as our morphisms, and addition and multiplication acting as our two monoids, with multiplication commuting over addition. We could write such a vector as $v = [v_0, v_1, v_2]$, reconciling conventional vector representations with this generalisation.

3.4.2 External Tensor Product and Sum

In the remaining work, I will generally restrict my discussion to the bind operation, \otimes , since all the following arguments apply equivalently to bundle, \oplus . With this new generalisation to co-presheaves, remember that our goal is to have two operations (bind and bundle) of type

$$\otimes, \oplus : [\mathcal{I}, \mathcal{V}] \times [\mathcal{I}, \mathcal{V}] \rightarrow [\mathcal{I}, \mathcal{V}].$$

We can get part way there using the external tensor product. The external tensor product of two co-presheaves has type:

$$\bar{\otimes} : [\mathcal{I}, \mathcal{V}] \times [\mathcal{I}, \mathcal{V}] \rightarrow [\mathcal{I} \times \mathcal{I}, \mathcal{V}]$$

for operands, $v, w : [\mathcal{I}, \mathcal{V}]$. Its action on objects, $i, j : \mathcal{I} \times \mathcal{I}$, is given by:

$$(v \bar{\otimes} w)(i, j) = v_i \cdot w_j,$$

where (\cdot) comes from the multiplication operation of \mathcal{V} . The exact same types and notation apply to the external sum $\bar{\oplus}$ using $(+)$.

The external tensor product and sum possess almost all the properties we want out of our bind and bundle operations, respectively. However, the results of these operations do not have the same dimensionality as their operands, so they are only well-suited for algebras like Smolensky's TPA. If we want bind and bundle to produce outputs that preserve dimensionality, we'll need an additional tool to capture the ideal "compression" of $\bar{\otimes}/\bar{\oplus}$.

3.4.3 Kan Extensions

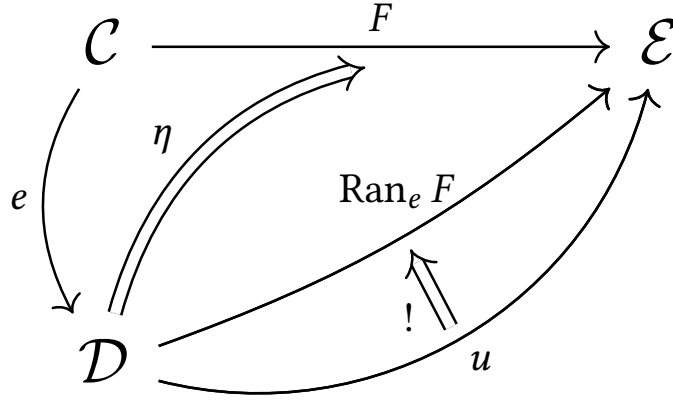
The last task is to find a mapping $\mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$. Applying this map inside $[\mathcal{I} \times \mathcal{I}, \mathcal{V}]$ would give us the desired output of type $[\mathcal{I}, \mathcal{V}]$. For this, remember that we've specified that \mathcal{I} must be

monoidal. Thus, we can use \otimes from \mathcal{I} to induce the following diagram:

$$\begin{array}{ccc}
 \mathcal{I} \times \mathcal{I} & \xrightarrow{v \bar{\otimes} w} & \mathcal{V} \\
 \downarrow \otimes & & \nearrow v \otimes w \\
 \mathcal{I} & &
 \end{array}$$

However, there are many possible ways that $v \otimes w$ could map from \mathcal{I} to \mathcal{V} , such that the diagram still commutes. What we really want is some “optimal” choice that preserves as much information as possible from $v \bar{\otimes} w$. Questions of optimisation are often expressed categorically using (co-)limits, and the triangular diagram above prompts us to express this limit as a Kan extension. While the left Kan extension would give us the “freest,” or most “liberal,” version of \otimes w.r.t. $\bar{\otimes}$, we desire the version of \otimes that is as *similar* to $\bar{\otimes}$ as possible. Such a notion is captured by the right Kan (Ran) extension, which can be intuitively understood as the most “cautious,” or “conservative,” version of $\bar{\otimes}$ when extended along \otimes .

For review: let \mathcal{C}, \mathcal{D} , and \mathcal{E} be categories. The right Kan extension of a functor, $F : \mathcal{C} \rightarrow \mathcal{E}$, when extended along a functor $e : \mathcal{C} \rightarrow \mathcal{D}$ is the (unique up to isomorphism) functor $\text{Ran}_e F : \mathcal{D} \rightarrow \mathcal{E}$ along with a natural transformation, $\eta : e; (\text{Ran}_e F) \Rightarrow F$. This natural transformation comes with the guarantee that any other $u : \mathcal{D} \rightarrow \mathcal{E}$ with its own natural transformation of type $(e; u \Rightarrow F)$ has a unique mapping to $\text{Ran}_e F$. This relationship is captured by the following diagram:



The right (and left) Kan extensions do not always exist for any pair of functors, F and e . Though as luck would have it, Kan extensions for the external tensor product do exist! An expression for this extension can be adopted from the general case in [76, Equation 4.24] as

$$(\text{Ran}_e v \overline{\otimes} w)_i = \int_{jk} \mathcal{I}(i, e(j, k)) \pitchfork (v_j \cdot w_k).$$

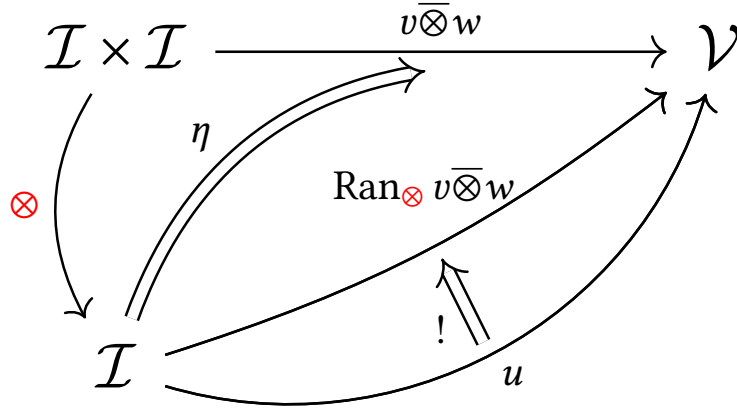
This expression introduces some new notation. First, $A \pitchfork B$ is the power operator and is simply the set of maps between A and B .³ Second, the integral symbol represents something called an “end.” Most authors introduce (co-)ends by way of (co-)limits, however I find the treatment given by Milewski in [3] to be most informative. The rough intuition is that an end can be thought of loosely as a type of product. Here, it is acting like a quantifier over all j and k . Dually, a co-end in this context would act like a trace (i.e. sum) across all j and k .

When e is \otimes we have

$$(\text{Ran}_{\otimes} v \overline{\otimes} w)_i = \int_{jk} \mathcal{I}(i, j \otimes k) \pitchfork (v_j \cdot w_k). \tag{3.5}$$

³In the case of non-enriched category theory. See [9] for a treatment of powering in the enriched context.

Merging the two previous diagrams gives us



and indicates that $v \otimes w := \text{Ran}_{\otimes} v \bar{\otimes} w$ is the bind operation we're looking for. To this end, let's evaluate the mapping into $\text{Ran}_{\otimes} v \bar{\otimes} w$ from an arbitrary $u : [\mathcal{I}, \mathcal{V}]$ (I use component-wise notation for clarity):

$$\begin{aligned}
 & \int_i \mathcal{V}(u_i, (\text{Ran}_{\otimes} v \bar{\otimes} w)_i) \\
 \cong & \int_i \mathcal{V}(u_i, \int_{jk} \mathcal{I}(i, j \otimes k) \pitchfork (v_j \cdot w_k)), && \text{from 3.5} \\
 \cong & \int_{ijk} \mathcal{V}(u_i, \mathcal{I}(i, j \otimes k) \pitchfork (v_j \cdot w_k)), && \text{by continuity} \\
 \cong & \int_{ijk} \mathbf{Set}(\mathcal{I}(i, j \otimes k), \mathcal{V}(u_i, v_j \cdot w_k)), && \text{by definition of power operator} \\
 \cong & \int_{ijk} \mathbf{Set}(\mathcal{I}(i, j) \times \mathcal{I}(i, k), \mathcal{V}(u_i, v_j \cdot w_k)), && \text{since } \otimes \text{ is Cartesian} \\
 \cong & \int_{ijk} \mathbf{Set}(\mathcal{I}(i, j), \mathbf{Set}(\mathcal{I}(i, k), \mathcal{V}(u_i, v_j \cdot w_k))), && \text{by currying} \\
 \cong & \int_{ik} \mathbf{Set}(\mathcal{I}(i, k), \mathcal{V}(u_i, v_i \cdot w_k)), && \text{by (ninja) Yoneda [92]} \\
 \cong & \int_i \mathcal{V}(u_i, v_i \cdot w_i), && \text{by Yoneda again} \\
 \Rightarrow & (\text{Ran}_{\otimes} v \bar{\otimes} w)_i \cong v_i \cdot w_i.
 \end{aligned}$$

Thus, we have

$$v \otimes w = \text{Ran}_{\otimes} v \overline{\otimes} w = \int_i v_i \cdot w_i \quad (3.6)$$

and

$$v \oplus w = \text{Ran}_{\oplus} v \overline{\oplus} w = \int_i v_i + w_i. \quad (3.7)$$

Some exciting observations immediately follow from this derivation. First, it shows that the rig structure of VSAs is inherited from the category of values, \mathcal{V} , being a rig. Second, decoupling indices from values means that we can isolate the matter of “compression” to the mapping of $\mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ along \otimes when dealing with co-presheaves on a fixed, finite domain. Finally, this result validates that the widespread practice of using element-wise operations in VSAs is optimal. In fact, many of the alterations to element-wise multiplication/division (i.e. normalisation, thresholding, *etc.*) are imposed to model certain behavioural effects, such as neuronal saturation, rather than endowing computational advantages.

3.4.4 Examples

First, let’s consider a relatively canonical example of how to describe a VSA with this framework.

Example 3.4.1. Let \mathcal{I} be the discrete category whose objects are natural numbers, with \otimes being the tensor product. Also, let $\mathcal{V} = (\mathbb{R}^{\geq 0}, \leq, \cdot, +, 1, 0)$ be the category of non-negative real numbers described earlier. Then $[\mathcal{I}, \mathcal{V}]$ are non-negative, real-valued vectors, with optimal bind and bundle being element-wise multiplication and addition, respectively, within the resulting d^2 -dimensional tensor of the original d -dimensional v and w vectors, as in TPR models.

Next, consider how we might describe a setting where we are restricted to a fixed, finite dimension. In these cases, \mathcal{I} must have finitely many objects. Fortunately, the only requirement for \otimes in my derivation is that it is Cartesian, and any finite poset with a corresponding “meet” as its \otimes has this property. Consider the following indexing category:

Example 3.4.2. Let \mathcal{I} be the category representing the poset of natural numbers between 0 and 10 (inclusive), with morphisms being \leq like before. Its monoidal action, \otimes , is the min operation for two given objects. Then the mapping $\otimes : \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ takes two 10-dimensional inputs and still has a 10-dimensional output.

Note that this example doesn’t mention \mathcal{V} . This is because we are free to choose any rig/bimonoidal category. In fact, decoupling vectors into \mathcal{I} and \mathcal{V} allows engineers to mix-and-match index and value categories as desired. Our two restrictions, that \mathcal{I} be Cartesian and that \mathcal{V}

be a rig, are extremely broad. For instance, binary values are used in BSC, with XOR and AND acting as our multiply and divide, respectively.

The minimal restriction on \mathcal{I} is loose enough that we can even consider indexing categories that are slightly more “exotic” than total orders, like the Reals and Naturals. Consider:

Example 3.4.3. Let \mathcal{I} be the category representing the powerset of three elements, with morphisms being set inclusion. Its objects are the eight subsets of the underlying set and \otimes is set intersection. Then the mapping $\otimes : \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ produces the original eight-object lattice structure of the powerset.

3.5 Discussion and Future Work

There’s still clearly work that needs to be done on this topic. In particular, similarity, unbinding, and unbundling aren’t captured by my description. Originally, my hope was that unbinding and unbundling could arise naturally provided that the category of values, \mathcal{V} behaved like a division ring. However, division rings, as a category, are not a standard notion, despite how fundamental rig categories are.

I have attempted to describe similarity using enrichment. Enriched categories are a generalisation of categories that extend hom-sets to hom-*objects*—specifically, rather than the morphisms between two objects in a category being a set, they are an object from a monoidal category, and inherit the structure of that monoid between morphisms. My intuition is that such structure would be well-suited to describing the similarity measure in a VSA. For one, enrichment is already used to describe things such as Lawvere metric spaces [6]. However, thus far this effort has not yet resulted in a rigorous formalisation.

Another important thing to note is that some of the binding and bundling operations in Table 3.2 are actually inconsistent with this formalisation. In particular, it’s clear that the various modifications to bind and bundle (thresholding, normalisation, *etc.*) don’t arise from bimonoidal structures. In these cases, such modifications are historically for *modelling*, rather than *computational*, purposes. For example, thresholding is often used to model the kind of “saturation” that neurons exhibit. This means that engineers add extra mechanics to VSA implementations when the goal is modelling cognitive biases/phenomena. I hope that this formalisation helps engineers design VSAs when computation is their only concern.

Second, the worked examples in Section 3.4.4 largely validate current VSA models. Example 3.4.3 acknowledges that such an indexing set is well-behaved with any division ring. Finding new possible VSAs using my formalisation is left for future work. This could be as simple as picking a new combination of \mathcal{I} and \mathcal{V} that satisfy the described properties.

This work describes how to view vectors as distinct indices and values, and suggests that it may be possible to “mix-and-match” any combination of a Cartesian indexing category with a rig values category, but I leave it to others to explore the various combinations of these. Future work can also explore the relationship that the similarity function has with bind and bundle.

Another result I wish to work towards is axiomatising a category, **VSA**, in a manner similar to the work on Hilbert spaces in [64, 65, 98]. If possible, I would then be interested to see if it is possible to express an adjunction between these two categories. Doing so would formalise the analogy between VSAs and Hilbert spaces that has long been observed [48, 49]. It may be that there’s no meaningful difference between the two, but even if VSAs are just finite-dimensional Hilbert spaces (with similarity arising from being an inner product space, and binding and bundling corresponding to the tensor and direct sum, respectively), current axiomatisations of finite-dimensional Hilbert spaces are limited to ones with contractive mappings.

Finally, this work has largely ignored how most of the models built using VSAs are deeply compositional in nature. Appendix D includes instances of compositional data structures for the interested reader (in particular, Appendix D.3). This clearly indicates that there is a wealth of work that can be done to ground VSA applications in category theory. For instance, modelling *analogy* is a recurring application in the VSA literature [74, provides the canonical “what is the dollar of mexico?” example], and there is a clear opportunity to understand this notion more rigorously. Future work will directly explore these kinds of topics more precisely in categorical terms. From an engineering perspective, a significant achievement towards this end would be the construction of a VSA that is purely functional⁴ in nature.

⁴“Functional” in the programming sense, rather than set theoretic.

Chapter 4

How Pre-lenses Reconcile Backpropagation and Autodifferentiation (and Other Symmetries)

4.1 Introduction

This work focuses on using pre-lenses to describe supervised learning models. It is a direct extension of the work done in [24] that describes learners as an object in a bicategory of “parametric lenses.” The value of pre-lenses is that they unify this bicategory with its dual. This also enables a “triple” profunctor (and Tambara module) representation of pre-lenses. On a more practical (and aesthetic) note, these pre-lenses demonstrate an elegant symmetry of pairs of two-in/two-out operations that allow us to model ANN features as a variety of parallel compositions combined with some form of “zipping” these pieces together.

As mentioned in Section 1.2.2, much of this work is based on a conjecture I made in the Fall of 2023. Existing formalisations of neural networks described them as bicategories, however composing network layers results in iteratively taking a product of parameters. It would be nice to take this bicategory (which some have called **Para**, and others have called **Learn**) and resolve the asymmetry between the inputs and parameters.

During a week-long visit with Bartosz Milewski, we tried to address this problem, but were unable to come up with a nice formalisation during that period. After their visit, I worked on the material presented in Chapter 3. During that period, Milewski came up with the pre-lens construction, which is published on their blog [102]. This work coincides with work that ob-

serves similar properties (though from a different perspective that will be discussed briefly in Section 4.7).

My contributions are largely in the form of various observations and incremental extensions of Milewski’s work. Specifically, I clarify how unifying **Para** with its dual, **co-Para**, equates to a reconciliation between traditional backpropagation and autodifferentiation. I have also implemented the methods by Milewski on a non-synthetic dataset (namely MNIST), and described how to implement re-parameterisations. Finally, I provide an insight enabled by pre-lenses: that generative models can be seen as the `set` operation of a pre-lens that corresponds to the `put` operation of a lens described in [24].

4.2 Background

The background material discussed here will be more extensive than seen in prior parts of this thesis. There are two reasons for this. First, the material is considerably less broadly known, and resources for learning the material are less accessible. Second, I believe that appreciating pre-lenses is easier when having a clear picture of the material that they directly build on.

4.2.1 Autodifferentiation

Nowadays, the algorithm for backprop described in Section 2.2 is usually implemented in an alternative manner. This method is called automatic differentiation¹ and arises from the observation that lower layers of a neural network don’t depend directly on the original error signal/loss; they only depend on the error signal *w.r.t.* the immediately subsequent layer. Once you know what the upstream loss is, you can easily compute the loss for the preceding layer.

With this in mind, network components can now be constructed as pairs of operations, `fwd` and `bwd`, with the following type signatures:

```
fwd :: a -> b
bwd :: (a, db) -> da
```

¹There is both “forward” and “backward” automatic differentiation, but the distinction isn’t particularly important to this work.

Here, a is the layer’s input, b is the layer’s output, and the prime versions of each are the respective error derivatives². When considered together, they can be seen as a function of two inputs (a and db) and two outputs (b and da). In fact, in Section 4.2.6 we will see precisely how these pairs of operations are a single function. Adding parameters can be done easily by tupling a with a parameter, p (and same for da with dp). A simplified picture of the result is provided in Figure 4.1. These composing operations can then be chained together to form a “computational graph” that computes all the derivatives³ at once when the graph is capped with an error signal.

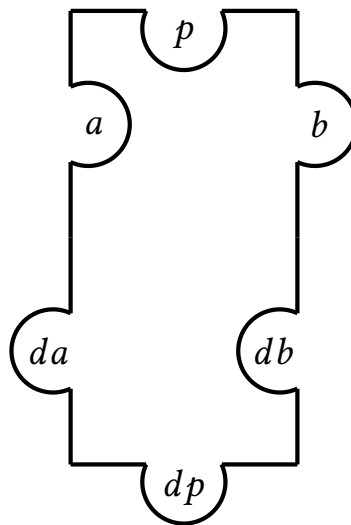


Figure 4.1: A representation of a single network layer that uses autodifferentiation. Subsequent network layers will have an identical shape and fit together by horizontally composing them. Note how this means that da from a subsequent layer “fulfills the promise” that there will be an input to db in the current layer, while simultaneously creating a new requested input upstream.

For engineers, automatic differentiation almost always includes the practice of associating each operation ($+$, \cdot , $\sin(x)$, e^x , *etc.*) with its corresponding (reverse) derivative without the engineer needing to provide that operation themselves. Then, each variable, x , is associated with an adjoining gradient value, $x.\text{grad}()$, that is initially empty, but updated once an error signal is computed (in PyTorch this is simply `loss.backward()`). For the network parameters, Equation 2.3 can then be easily implemented by simply adding the grad values to the originally defined variables.

²Note that I’m using dx instead of x' when discussing lenses in the context of ANNs. This is to free up the prime notation for when discussing a variety of functors later on, that change the types of the inputs.

³I’m being imprecise with my language here. *Which* derivatives: the input or weights? The answer is both and more precision will be introduced in the subsequent section.

Automatic differentiation is possible partly because the derivatives of the error signals are subject to the “chain rule”. That is, for two composing functions, f and g , the derivative of $(g \circ f)(x)$ is $(g \circ f)'(x) = g'(f(x)) \cdot f'(x)$. As long as multiplication is associative and has an identity, the composing forward functions of a network always have a corresponding expression for the derivatives of the backward error signals that also compose in a manner that satisfies our categorical axioms. These pairs of mappings, just like `read` and `write` from Section 2.5.3, constitute a morphism in a corresponding lens category.

4.2.2 Parametric Lenses

In contrast to the examples in Section 2.5, machine learning has received little interest from category theorists until recently. Some connections were drawn between compositional models in cognitive science and neural architecture [113, 144], but it is only quite recently that Cruttwell *et al.* proposed a categorical foundation for gradient-based learning [24]. This foundation relies primarily on the construction of “parametric lenses.” In brief terms, parametric lenses are the categorical objects that describe automatic differentiation.

A parametric lens is just the composition of two simpler constructions: parametrisations and lenses. Its motivation comes from the following observations of the learning process. First, we want learning to be parametric. That is, we wish to model functions, $f' : (a, p) \rightarrow b$, such that p can be modified until $f(-, p) = f_p : a \rightarrow b$ is the ideal function from a to b . Learning is then reduced to choosing the ideal p . Second, we wish for computations to be bidirectional. Specifically, we wish to model some task as a *forward* operation that can be passed through successive layers of a network, while *backward* computations can carry error signals back through layers of a network. This way, errors can propagate through the entirety of a network and be used for choosing all our parameters. Cruttwell *et al.* formalise these two notions in categorical terms.

Definition 4.2.1. *Let C be a strict, symmetric monoidal category with tensor product, \otimes , and I as its unit object. Then the parametric category, $\text{Para}(C)$, has the same objects as C , and has morphisms of the form (p, f) , where p is an object of C and $f : a \otimes p \rightarrow b$. Two maps, $(p, f) : a \rightarrow b$ and $(q, g) : b \rightarrow c$ compose via $(q \otimes p, g \circ (1_q \otimes f)) : a \rightarrow c$. Identities for objects, a , in this category are the pairs $(I, 1_a)$.*

The restrictions on C in the definition above simply ensure that the tensor product, \otimes , exists and is symmetric, while strictness—which is simply the additional structure of an $=$ relationship—is present here just to ensure that $\text{Para}(C)$ is a category rather than a bicategory. In practice, these restrictions don’t constrain any typical learning application.

Let’s consider the following example: take the category **Smooth**, whose objects are natural numbers, and whose morphisms are all the smooth mappings of the form $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ for two natural numbers m and n . Then $\text{Para}(\mathbf{Smooth})$ exists and can be interpreted as the category of neural networks. Morphisms in this category have the form $f : \mathbb{R}^m \times \mathbb{R}^p \rightarrow \mathbb{R}^n$, where \mathbb{R}^m is the dimension of our input space, \mathbb{R}^n is the dimension of our output space, and \mathbb{R}^p represents our network weights. Note that this formalisation naturally expresses the hierarchical properties of neural networks, since composition of layers is just composition of morphisms.

While $\text{Para}(\mathbf{Smooth})$ contains all possible neural networks, it doesn’t express how networks can learn. For vector spaces, the canonical way neural networks learn is through backpropagation or one of its variants. We need to generalise this idea of “backwards” computation that is paired with “forwards” computation. This generalisation is called a “lens.” We saw lenses appear in Section 2.5.3, and now formally define them categorically.

Definition 4.2.2. *Let C be any Cartesian category. Then $\text{Lens}(C)$ is a category whose objects are pairs of objects from C . Specifically, (a, da) is an object of $\text{Lens}(C)$ where a and da are objects of C . Morphisms in $\text{Lens}(C)$ are pairs of mappings (f, f^*) from (a, da) to (b, db) where $f : a \rightarrow b$ and $f^* : a \times db \rightarrow da$. These pairs of mappings have many names, with f often called “get,” “read,” or “forward,” and f^* called “put,” “set,” “write,” or “backward,” respectively.*

Let π_i be the i -th projections of a Cartesian product. Two morphisms, $(f, f^) : (a, da) \rightarrow (b, db)$ and $(g, g^*) : (b, db) \rightarrow (c, dc)$, in $\text{Lens}(C)$ compose to form the pair*

$$(g \circ f, f^* \circ (\pi_0, g^* \circ (f \circ \pi_0, \pi_1))),$$

from (a, da) to (c, dc) . The identity of an object, (a, da) , is the pair $(1_a, \pi_1)$.

Remark. Here da is written to maintain consistency with the other sections of this document. While a and da are related by virtue of being paired together as an object in a lens category, d is not meant to relate a to da as some sort of operator.

Type-checking the morphisms of these lenses becomes fairly tricky—particularly for the put component. For this reason, it is convenient to represent these morphisms using string diagrams. The validity of such diagrams is justified in [136]. Figure 4.2 provides a diagrammatic representation of the composition described in Definition 4.2.2.

Composing the two constructions of parametrisations and lenses, we now have everything required to understand how learning parameters occurs in category theoretic terms.

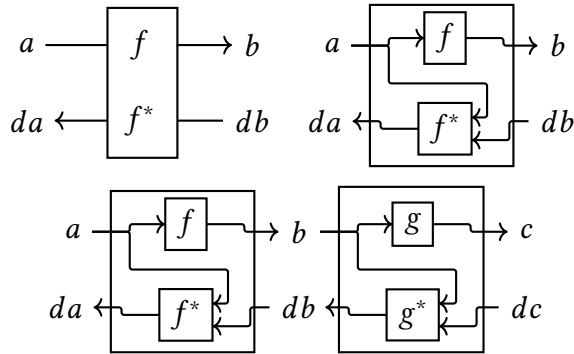


Figure 4.2: String diagram of morphisms and composition in $\text{Lens}(C)$. The upper left diagram represents the morphism, (f, f^*) , which is then “unboxed” in the upper right diagram. The bottom diagram illustrates how two morphisms compose. Note that this visualisation of a lens might look a bit different from the one described when talking about databases. Nevertheless, it is very much the case that these two things are equivalent!

Definition 4.2.3. $\text{Para}(\text{Lens}(C))$ is the category of parametric lenses on C . Its objects are the pairs (a, da) of objects in C . Its morphisms from (a, da) to (b, db) —called *parametric lenses*—are a choice of parameter pairs, (p, dp) , along with the mappings $(f, f^*) : (a, da) \times (p, dp) \rightarrow (b, db)$, where $f : a \times p \rightarrow b$ and $f^* : (a \times p) \times db \rightarrow (da \times dp)$ ⁴.

I would like to quickly note that even though $\text{Para}(-)$ isn’t strictly a functor (since we have to choose the objects, p , that form our parameters), it is still *functorial*. That is, composition is preserved when morphisms are mapped from C to morphisms in $\text{Para}(C)$, regardless of the choice of p . Further, $\text{Lens}(-)$ is a functor, in the strict sense. Since we know that functors compose, it isn’t really problematic to think of $\text{Para}(\text{Lens}(-))$ like a functor acting on some symmetric monoidal category.

We’ve already seen what a type signature would look like for a fwd/bwd ⁵ lens when discussing autodifferentiation. The type signature for a parametric lens is similar, but replacing the inputs with the Cartesian pair of inputs and parameters:

$$\begin{aligned} \text{fwd} &:: (a, p) \rightarrow b \\ \text{bwd} &:: ((a, p), db) \rightarrow (da, dp). \end{aligned}$$

⁴Both sets of additional parentheses in this signature are actually unnecessary, but included to highlight that f^* is still “backwards” *w.r.t.* f just as before, only with the appropriate parameters being accounted for.

⁵Alternatively, “get/set.”

Now that learning has been described, there is a small issue that needs to be addressed. While we can certainly update our parameters with f^* , this doesn't guarantee that f^* will do a *good* job of updating them. For this, we need to ensure that our categories have a gradient, so that some extremum can be found incrementally. However, this gradient is more general than what is seen in calculus. In [24], Cruttwell *et al.* describe learning on any category that is a “Cartesian reverse differential category,” or CRDC [23]. This means that it is a “Cartesian left additive category,” along with an operation that associates each morphism $f : a \rightarrow b$ to a reverse map $R[f] : a \times b \rightarrow a$. Note that this function, $R[f]$ (almost) conveniently type matches with f^* ! What $R[f]$ really does is take a *tangent* object (read: vector, for all intents and purposes) from db and map it to a tangent object in da . Hence, we can use $f^* = R[f] : a \times db \rightarrow da$.

Cruttwell *et al.* then describe the entire supervised learning process in these terms. Models, losses, learning rates, and optimisers can all be made into lenses (that is, *every* component of a feed-forward network). What's even more exciting is that we can reinterpret the entire model as the put component of a mapping that takes as input some parameters, p , together with a chosen training pair $a \times b$ to obtain a new set of parameters for p . This morphism, $\text{put} : p \times (a \times b) \rightarrow p$, is nothing more than yet another parametric morphism!

Since [24], authors have noted that parametric lenses form a bicategory [141]. In this bicategory, 0-cells and 1-cells are as defined above, while 2-cells are the re-parametrisations. We can call this bicategory **Para**, and the main theoretical contribution of Milewski when constructing pre-lenses is unifying **Para** with its dual, **co-Para**.

4.2.3 Lenses in Related Domains

Since lenses formalise bidirectional operations, it isn't too surprising that researchers have noticed lenses in other domains of intelligence. An example is in [145], where St. Clere Smithe introduces “Bayesian lenses” to characterise Bayesian inference. The goal of Bayesian inference is to work backwards from some statistical model to obtain the probability of some hypothesis using Bayes' theorem. For example, working backwards from some observations of the world to obtain the causes of those observations. We can picture this process as an agent taking some perceptions and using those to modify its *internal* state.

However, if we wish to go beyond Bayesian inference to deal with “active” inference, then we also need to describe how an agent can take information from its internal state, or beliefs, and use those to motivate actions that modify some *external* state (i.e. environment) to achieve those beliefs. These duals of action and perception are argued to both be in service of optimising one quantity—the free energy [45] of some system. The process of modelling such bidirectional

operations locally in a way that composes is called “predictive coding,” and it can be well-formalised using these Bayesian lenses.

In [25], de Felice also applies lenses to a stochastic domain, but this time focuses on the category **Prob** [12]—the category whose objects are probability spaces and whose morphisms are measurable functions that satisfy an extra property of being “absolutely continuous.” They develop **Lens(Prob)** by examining a special form of composition of these lenses with some “context,” or environment. This construction leads them to discuss Markov Decision Processes, which de Felice uses to describe games. Ultimately, the main application here is actually for generative and discriminative NLP models.

4.2.4 Profunctors

What makes parametric lenses a nice formalisation is that they are fairly straightforward and don’t require many complex constructions from category theory. Generalising to pre-lenses simplifies the process of building network layers using ordinary function composition, but comes at the trade-off of requiring more background in category theory. The rest of this section is dedicated to providing that background. For the time being, we will forget about parametric lenses and go back to what is known about non-parametric lenses.

The first step is introducing profunctors. However, before getting into profunctors, it’s helpful to see the more general “bifunctors”:

Definition 4.2.4. *A bifunctor, F , is simply a functor whose domain is a product category. That is,*

$$F : C_0 \times C_1 \rightarrow D$$

for categories C_0 , C_1 , and D .

Note that, while bifunctors sound like they could be functors between bicategories, this is not the case. Those would typically be called 2-functors. A very familiar example of a bifunctor arises in every monoidal category, (C, \otimes, I) where \otimes is a bifunctor, $\otimes : C \times C \rightarrow C$.

In programming, we can think of a bifunctor, B , as having two arguments, a and b , where it is covariant in both. Explicitly,

$$B :: \text{Any} \rightarrow \text{Any} \rightarrow \text{Any}.$$

However, B is equipped with the mapping,

$$\text{bimap} :: (a \rightarrow a') \rightarrow (b \rightarrow b') \rightarrow B\ a\ b \rightarrow B\ a'\ b'$$

such that for the appropriate functions, $f :: a \rightarrow a'$ and $g :: b \rightarrow b'$, we have

$$\text{bimap } f \ g :: B \ a \ b \rightarrow B \ a' \ b'.$$

It's also important to ensure that $\text{bimap } \text{id} \ \text{id} \equiv \text{id}$ ⁶.

Setting the codomain of a bifunctor to be **Set**, and the first component of the product to be an opposite category yields a “profunctor”:

Definition 4.2.5. *A profunctor, F , is a bifunctor*

$$\text{hom}_F : D^{\text{op}} \times C \rightarrow \mathbf{Set}$$

for categories C and D ⁷. It can also be expressed as $F : C \nrightarrow D$.

But what are these profunctors intuitively? First, let's address a couple oddities in the definition. First, why is it important that our mapping takes an input from D^{op} rather than D ? The intuition is that a profunctor is like an intermediary operation that “sucks in” from one side and “spits out” the other. A better illustration of this appears shortly in Figure 4.3.

Second, we're defining F as the profunctor, but the common notational practice uses hom_F . I believe that this can be better understood by looking at relations. Consider a relation, R , that asks whether some $x \in X$ relates to some $y \in Y$, written xRy . Unlike functions, relations can be one-to-many mappings. Thus, R can be considered a subset of $Y \times X$ ⁸. However, this subset of $Y \times X$ is equivalent to taking pairs (y, x) and mapping them to \mathbb{B} , where outputs of 0/False indicate that $xRy \notin R \subseteq Y \times X$ and outputs of 1/True indicate that $xRy \in R$. Thus, a relation can be written as

$$R : Y \times X \rightarrow \mathbb{B}.$$

In this way, profunctors are similar to relations, except that we're mapping from a pair of categories rather than a pair of sets and we're mapping onto **Set** instead of \mathbb{B} . This similarity to relations also manifests through profunctors generalising functors in the same way that relations generalise functions⁹. In fact, given any functor, $F : C \rightarrow D$, one can define a profunctor by composing F with the “Yoneda” functor, $Y_D : D \rightarrow [D^{\text{op}}, \mathbf{Set}]$.

Because functions and relations are so common in programming, it isn't surprising that profunctors can be found everywhere. In fact we've been using a particular profunctor throughout

⁶Instead of `bimap`, it would be fine to define two separate functions, `lmap` and `rmap`, separately, where each applies a function to the first and second input respectively.

⁷Typically in programming, and for this thesis, C and D are the same category.

⁸Here we write $Y \times X$ instead of the more usual $X \times Y$ to reinforce the connection to Definition 4.2.5.

⁹Some people even call profunctors “relators” for this reason, however this is rare.

this entire document: our Type function arrow, \rightarrow ! We can see exactly how this arrow is a profunctor by starting with the general definition. A profunctor, P , has two arguments, a and b , where it is contravariant in a covariant in b . Explicitly,

$$P :: \text{Any} \rightarrow \text{Any} \rightarrow \text{Any}.$$

However, P is equipped with the mapping,

$$\text{dimap} :: (a' \rightarrow a) \rightarrow (b \rightarrow b') \rightarrow P\ a\ b \rightarrow P\ a'\ b'$$

such that for the appropriate functions, $f :: a' \rightarrow a$ and $g :: b \rightarrow b'$, we have

$$\text{dimap}\ f\ g :: P\ a\ b \rightarrow P\ a'\ b'.$$

This definition is almost identical to a bifunctor, except that f reverses the domain and codomain. Like bifunctors, we ensure that $\text{dimap}\ \text{id}\ \text{id} \equiv \text{id}$, and we can also separate dimap into a pair of mappings, lmap and rmap .

Replacing P with \rightarrow (as an infix operator) in the Type expression for dimap yields

$$\text{dimap} :: (a' \rightarrow a) \rightarrow (b \rightarrow b') \rightarrow (a \rightarrow b) \rightarrow (a' \rightarrow b').$$

We can make the connection to function composition more clear by rearranging as

$$\text{dimap} :: (a' \rightarrow a) \rightarrow (a \rightarrow b) \rightarrow (b \rightarrow b') \rightarrow (a' \rightarrow b'),$$

though I like the diagram in Figure 4.3 best. A helpful intuition for why we are contravariant in the first argument is that f is somehow “sucking in” inputs, in contrast to g “spitting out” outputs.

We can also demonstrate that lenses are instances of profunctors. Let $f :: a' \rightarrow a$ and $g :: da \rightarrow da'$. Also, let $\text{Lens}\ b\ db\ a\ da$ be the type signature of our lens¹⁰. Then, $(\text{Lens}\ b\ db)$ is a profunctor with

$$\text{dimap}\ f\ g\ (\text{Lens}\ \text{fwd}\ \text{bwd}) = (\text{Lens}\ \text{fwd}'\ \text{bwd}')$$

where

$$\begin{aligned} \text{fwd}' &= \text{fwd} \cdot f \\ \text{bwd}'\ (a',\ db) &= g\ (\text{bwd}\ (f\ a',\ db)). \end{aligned}$$

While I will generally leave the type-checking to you for the remainder of this document, I’ve included a demonstration that this expression type-checks in Appendix E.

¹⁰Notice that the outputs are written first. This is simply to make partial function application easier, as we generally want to express profunctoriality *w.r.t.* our input side, (a, da) .

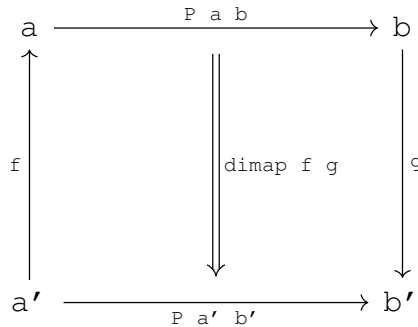


Figure 4.3: A commuting diagram demonstrating the relationship between all the inputs, a , a' , b , and b' ; f and g ; P , and; dimap . In the case where P is \rightarrow , we can simply remove the labels. What I love about this diagram is that it also explains why we are contravariant in our input. Remember with functors that fmap is only needed because our compiler needs to distinguish between how we map objects and how we map morphisms. The same is true for dimap and if we reuse P we can see that $P \circ f = g$ would hold. For the arrow, \rightarrow , this composition wouldn't type-check if f were the opposite direction (*i.e.* if we were covariant in a).

4.2.5 Tambara Modules

Recall that profunctors are like relations. This raises the question “which subclass of profunctors are the ones that behave like functions?” To address this we need to review the notion of an “action”. For those comfortable with group or ring theory, these actions will be familiar. We can translate this concept into categorical terms as follows:

Definition 4.2.6. Let (M, \otimes, I) be a monoidal category. A (left) M -actegory¹¹ is a category, C , equipped with an “action” functor

$$\cdot : M \times C \rightarrow C$$

and natural isomorphisms that ensure the unitors and associator of M are coherent with the action on C . We may also say that M “acts” on C .

Actegories have a corresponding definition for actions on the right side (*i.e.* actions of type $C \times M \rightarrow C$). When M is symmetric, we may drop the left and right distinctions and simply call C an M -actegory, or “bi-module.” For the purposes of this thesis, we will always assume that M

¹¹Not a typo.

is symmetric, as this is the case for the Cartesian product and we are already imposing symmetry on our parameters.

Definition 4.2.7. *Let (M, \otimes, I) be a symmetric monoidal category, and let C and D be M -categories. A Tambara module is a profunctor, $T : C^{\text{op}} \times D \rightarrow \mathbf{Set}$ equipped with a collection of morphisms,¹² called the “strength” of T , whose components are*

$$s_{c,d,m} : T(c, d) \rightarrow T(m \cdot c, m \cdot d).$$

These morphisms are natural in c and d , and dinatural (natural but direction-reversing) in m .

It’s very easy to express Tambara modules in Haskell, using the Cartesian product as our monoidal operator. All that’s required is specifying that a Tambara module, t , is a (strong¹³) profunctor that satisfies

```
strength :: forall c d m. t c d -> t (m, c) (m, d).
```

As with profunctors, it’s fairly straightforward to show that $(\text{Lens } b \text{ } db)$ is a Tambara module:

```
strength (Lens fwd bwd) = (Lens fwd' bwd')
```

where

```
fwd' (m, a) = fwd a
bwd' ((m, a), db) = (m, bwd (a, db)).
```

With Tambara modules, the type signature of a lens is simple as

```
type Lens b db a da = forall t. Tambara t => t b db -> t a da
```

Profunctors as Functions

As stated in the introduction, the point of all this additional structure is to enable one simple feature: for two lenses, `lens1` and `lens2`, we would like to compose them as

```
lens1 . lens2,
```

¹²These would simply be a natural transformation if they were all in the same direction.

¹³you can read “strong” as being a monoid that behaves like a product. In this work, we will always be treating it as a Cartesian product during implementation.

i.e. with simple function composition. Indeed for any pair of lenses that are represented as Tambara modules,

```
lens1 :: forall t. t b db -> t a da
lens2 :: forall t. t c dc -> t b db,
```

we can compose them as

```
lens1.lens2 :: forall t. t c dc -> t a da,
```

as desired.

4.2.6 Existential Lenses

An interesting thing about lenses is that, by virtue of being Tambara modules, they have an existential representation. By “existential” what I mean is that we are quantifying over all values m from our monoidal category M that is acting on a given C . I think this existential representation can be best understood using Figure 4.4.

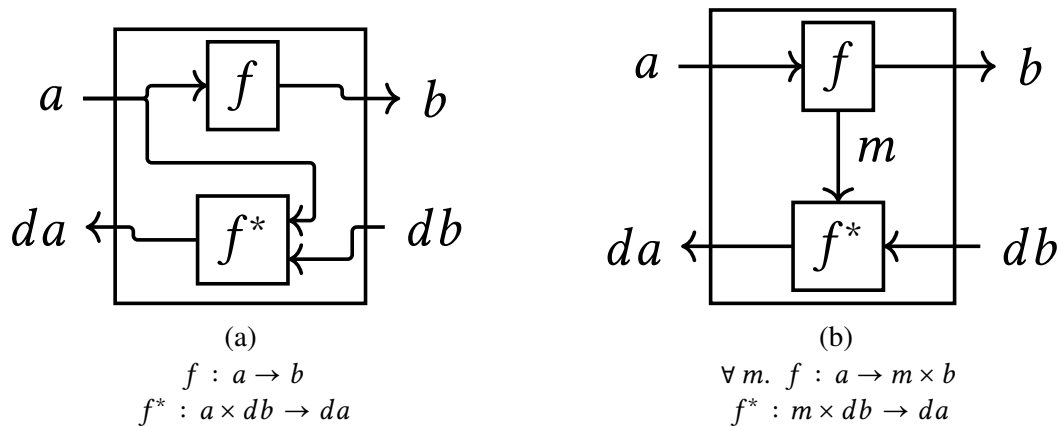


Figure 4.4: A comparison of classical, get/set lenses compared to existential lenses. It’s important to note that m is quantified over *all* possible values, meaning that it is “internal” to the lens in a way. Also notice that setting $m := a$ recovers the classic lens representation.

A helpful intuition for these “existential lenses” is that, rather than feeding a directly into f^* , we can ask f to create a “package” that f^* accepts. Because we don’t see these packages outside the lens, we quantify that the lens needs to function regardless of the Type of m .

Recall that $C(a, b)$ represents the mappings from a to b in a category C . For an underlying category, C , we express an existential lens as

$$\text{Lens}(b, db, a, da) = \int^m C(a, (m, b)) \times C((m, db), da),$$

or (in Haskell)

```
forall m. fwd :: (a, p) -> (m, b)
         bwd :: (m, db) -> (da, dp).
```

Figure 4.4 visually explains how we may recover classical lenses from the above expression, but we can also prove equivalence with a short derivation:

$$\begin{aligned} & \int^m C(a, (m, b)) \times C((m, db), da) \\ \cong & \int^m C(a, m) \times C(a, b) \times C((m, db), da), && \text{by } C \text{ being Cartesian} \\ \cong & C(a, b) \times C((a, db), da), && \text{by Yoneda (replace every } m \text{ with } a). \end{aligned}$$

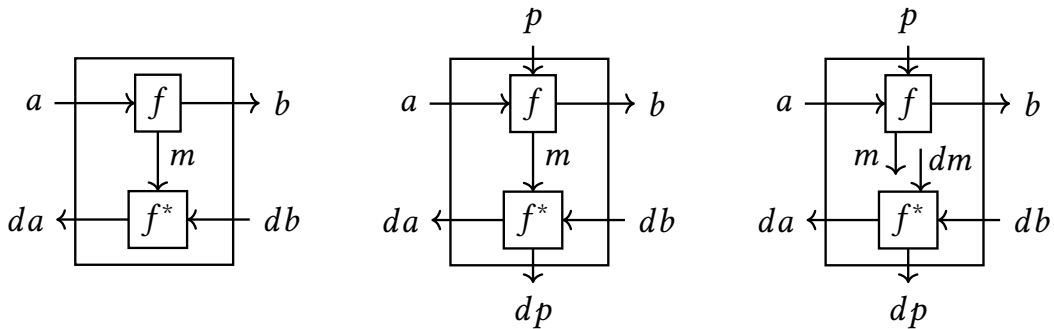
4.3 Pre-lenses

At this point we can appreciate the main motivation of pre-lenses. Given that existential lenses compose using ordinary function composition, is the same true of parametric lenses? The answer is yes (up to some associative laws), though we'll need some new machinery: pre-lenses. I'll start by showing what pre-lenses *are*, then demonstrate why they work by discussing tri-profunctors and triple Tambara modules.

I think it's easier to appreciate pre-lenses visually, rather than introducing them outright. Figure 4.5 demonstrates how pre-lenses can be constructed from existential lenses. That process can be explained as follows.

Consider our existential representation for a lens,

$$\int^m C(a, (m, b)) \times C((m, db), da).$$



(a) Existential Lens:

$$\forall m. f : a \rightarrow m \times b$$

$$f^* : m \times db \rightarrow da$$

(b) Existential Para-Lens:

$$\forall m. f : p \times a \rightarrow m \times b$$

$$f^* : m \times db \rightarrow dp \times da$$

(c) Pre-lens:

$$f : p \times a \rightarrow m \times b$$

$$f^* : dm \times db \rightarrow dp \times da$$

Figure 4.5: From existential lenses in 4.5a, we can add parameters as in 4.5b. At this point, an almost-symmetry is obvious in the Type signature of our lens. All inputs and outputs of f , have a partner in f^* , except m which is present in both. The idea of pre-lenses is very simple: simply break m up into a pair m and dm . The result is 4.5c. Note that this means we no longer quantify over m .

It's simple to add the parameters back in, as expressed in Figure 4.5b, yielding

$$\int^m C((p, a), (m, b)) \times C((m, db), (dp, da)).^{14}$$

At this point there's an opportunity to make both parts of our Cartesian product similar by decoupling at m . Doing so, by splitting m into m and dm , results in

$$C((p, a), (m, b)) \times C((dm, db), (dp, da)).$$

In Haskell, a pre-lens of Type, `PreLens b db m dm p dp a da` is simply

$$\text{fwd} :: (p, a) \rightarrow (m, b)$$

$$\text{bwd} :: (dm, db) \rightarrow (dp, da).$$

¹⁴Note that I've reversed the order of a and p , which is allowed because we're insisting that C is symmetric. This is to emphasise that p is "acting on" a , which will be important very soon. I could have done so while writing Section 4.2.2 as well, but chose to preserve the notation used therein.

Our existential representation for parametric lenses can then be recovered by simply setting `m` and `dm` to be the same type, (`m` for notational consistency, though it doesn't matter), then quantifying over all possible values.

Pre-lenses also have a very simple identity:

```
identityPreLens :: PreLens a da () () () () a da
identityPreLens = PreLens id id.
```

Their composition can also be fairly easily expressed using nothing more than a few projections, associators, and symmetrizers (see [102]). However, I think being slightly more explicit makes it a bit easier to trace.

```
prelensCompose :: PreLens b db m dm p dp a da
                -> PreLens c dc n dn q dq b db
                -> PreLens c dc (m, n) (dm, dn) (p, q) (dp, dq) a da

prelensCompose (PreLens fwd1 bwd1) (PreLens fwd2 bwd2)
              = (PreLens fwd3 bwd3)
```

with `fwd3 ((p, q), a)` defined as

```
let (m, b) = fwd1 (p, a)
    (n, c) = fwd2 (q, b)
in ((m, n), c)
```

and `bwd3 ((dm, dn), dc)` defined as

```
let (dq, db) = bwd2 (dn, dc)
    (dp, da) = bwd1 (dm, db)
in ((dp, dq), da)
```

One nice feature about pre-lenses is that they accumulate parameters and messages in tandem, meaning that they can be built in lockstep, as with autodifferentiation, or composing all the `fwd` and `bwd` operations separately, then tracing after-the-fact. This equivalence is something that has always been obvious for ANN engineers, but is made explicit and central in pre-lenses. I've drawn a simplified figure in Figure 4.6, which serves as a visual partner to Figure 4.1.

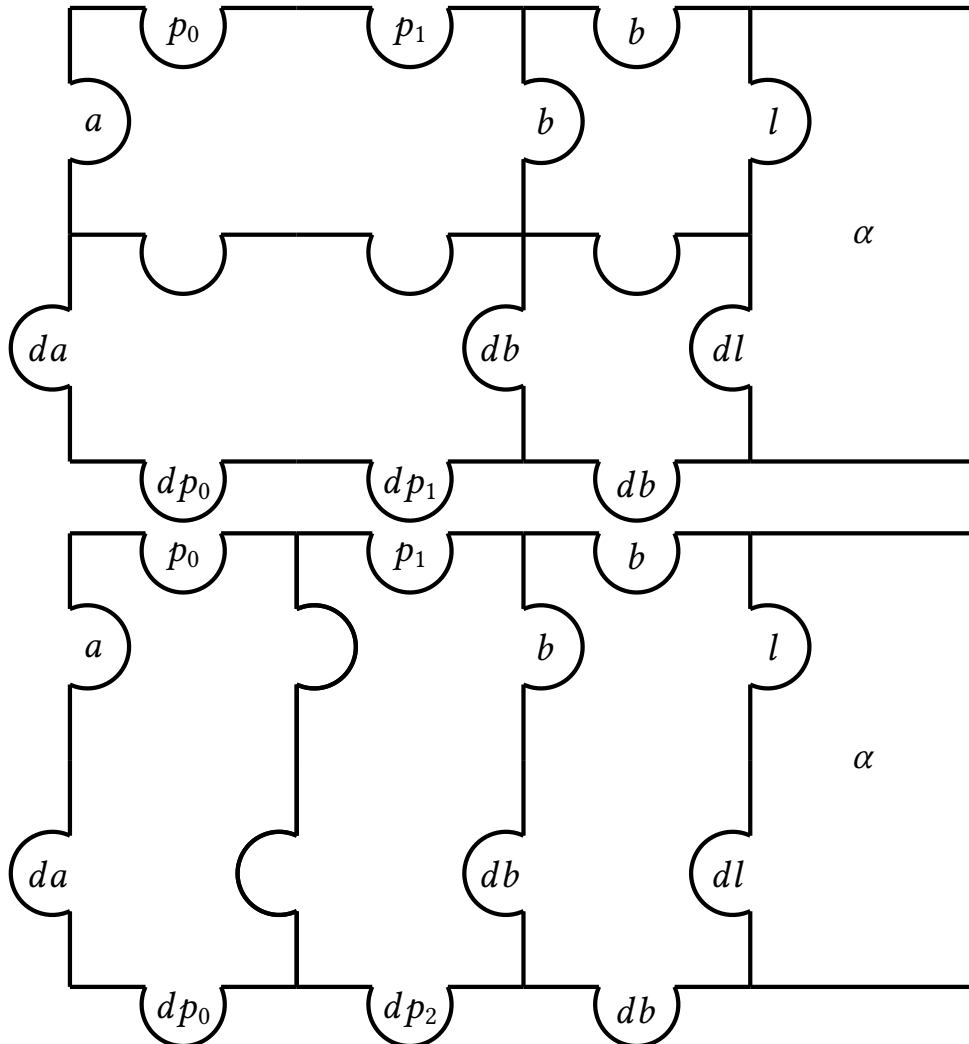


Figure 4.6: Using the same jigsaw representation as Figure 4.1, I show how pre-lenses highlight the known equivalence between learning via traditional backprop and learning with autodifferentiation. In the top diagram, the top half is meant to be computed first, with the outward puzzle inserts corresponding to the messages, m , that pre-lenses leave for the backwards pass. The right-most “end-piece” is the terminal parametric lens in [24]. In contrast, the bottom diagram builds layers in lock-step. Also note how, though they are meant to be separate diagrams, each dp_i can connect with a new p_i . This vertical tiling corresponds to learning, and will be discussed more in Section 4.6.

4.3.1 Tri-Profunctors

When discussing profunctors and Tambara modules, our examples were limited to non-parametric lenses. Strictly speaking, parametric lenses don't have a possible profunctor representation. However, pre-lenses are profunctorial in their three pairs of arguments, a, da ; p, dp ; and m, dm . This results in three instances of `dimap` rather than just one. For a, da the Type signature is

```
dimapA :: (a' -> a) -> (da -> da') -> t m dm p dp a da
        -> t m dm p dp a' da'.
```

The signature for `dimapP` is the same, *mutatis mutandis*. However, `dimapM` reverses the direction of the functions, f and g , to give

```
dimapM :: (m -> m') -> (dm' -> dm) -> t m dm p dp a da
        -> t m' dm' p dp a da.
```

In Haskell we can show that a pre-lens is a tri-profunctor (`PreLens b db`) by implementing the following three versions of `dimap` for each of (a, da) , (p, dp) , and (m, dm) :

dimapA

```
dimapA f g (PreLens fwd bwd) = PreLens fwd' bwd'
```

where

```
fwd' (p, a') = fwd (p, f a')
```

and

```
bwd' (dm, db) = let (dp, da) = bwd (dm, db)
                  in (dp, g da).
```

dimapP

```
dimapP f g (PreLens fwd bwd) = PreLens fwd' bwd'
```

where

```
fwd' (p', a) = fwd (f p', a)
```

and

```
bwd' (dm, db) = let (dp, da) = bwd (dm, db)
                in (g dp, da).
```

dimapM

```
dimapM f g (PreLens fwd bwd) = PreLens fwd' bwd'
```

where

```
fwd' (p, a) = let (m, b) = fwd (p, a)
               in (f m, b)
```

and

```
bwd' (dm', db) = bwd (g dm', db).
```

4.3.2 Triple Tambara Modules

A “triple” Tambara module, or “Trimbara” module as Milewski calls it in [102], is a tri-profunctor with two natural transformations, α and β .

```
alpha :: t m dm p dp a da
       -> t (m1, m) (dm1, dm) p dp (m1, a) (dm1, da)
```

```
beta :: t m dm p dp (a, p1) (da, dp1)
      -> t m dm (p1, p) (dp1, dp) a da
```

I like to think of `alpha` as expressing how a monoid, `m` plays nicely with our input, while `beta` expresses how pre-existing parameters play nicely with subsequent parameters. As with tri-profunctors, we can show that any pre-lens, `PreLens b db`, is a Trimbara module by showing what the `alpha` and `beta` functions are.

alpha

Our Type signature is

```
alpha :: PreLens b db m dm p dp a da
      -> PreLens b db (m1, m) (dm1, dm) p dp (m1, a) (dm1, da)
```

and the function can be written as

```
alpha (PreLens fwd bwd) = PreLens fwd' bwd'
```

where

```
fwd' (p, (m1, a)) = let (m, b) = fwd (p, a)
                    in ((m1, m), b).
```

and

```
bwd' ((dm1, dm), db) = let (dp, da) = bwd (dm, db)
                        in (dp, (dm1, da)).
```

beta

Our Type signature is

```
beta :: PreLens b db m dm p dp (p1, a) (dp1, da)
      -> PreLens b db m dm (p1, p) (dp1, dp) a da
```

and the function can be written as

```
beta (PreLens fwd bwd) = PreLens fwd' bwd'
```

where

```
fwd' ((p1, p), a) = fwd (p, (p1, a))
```

and

```
bwd' (dm, db) = let (dp, (dp1, da)) = bwd (dm, db)
                 in ((dp1, dp), da).
```

4.3.3 Tri-lenses

Pre-lenses can be directly represented as Trimbara modules. Such a representation,

```
TriLens b db m dm p dp a da,
```

can be written

```
forall t. Trimbara t => forall m' dm' p' dp'.
  t m' dm' p' dp' b db
  -> t (m, m') (dm, dm') (p, p') (dp, dp') a da.
```

What's nice about this representation is that, up to the usual identity, associativity, and commutativity laws for m , dm , p , dp ¹⁵, we have that the identity is the usual `id` and that two tri-lenses,

```
t1 :: TriLens b db m dm p dp a da
```

and

```
t2 :: TriLens c dc n dn q dq b db,
```

compose as

```
t1 . t2.
```

This “pseudocode” is largely ignoring the details that would ensure proper type-matching, which can be achieved using `dimapP` and `dimapM` with the appropriate unit and associativity laws.

For ease of use, we don't actually need to define our lenses as these tri-lenses. We can always define them with our `fwd` and `bwd` operations, then convert them to and from Trimbara modules via

```
toTrim :: PreLens b db m dm p dp a da
  -> TriLens b db m dm p dp a da
```

with

```
toTrim (PreLens fwd bwd) = beta . dimapA fwd bwd . alpha,
```

and

```
toFwdBwd :: TriLens b db m dm p dp a da
  -> PreLens b db m dm p dp a da
```

¹⁵We can rely on these laws because we're always dealing with a *symmetric* monoidal category and a strong profunctor, where $\emptyset \times A \cong A$.

with

```
toFwdBwd t = t identityPreLens.16
```

4.4 Building an ANN with Pre-Lenses

With this machinery, we have everything needed to build a neural network using pre-lenses throughout. In the same spirit as [24], I want to draw attention to how remarkably similar all the various components of the model appear. Where Cruttwell *et al.* discuss layers, losses, learning rates, and re-parametrisations, I want to extend this to batching, and point out how pre-lenses present a picture of learning as an n -ary composition, where n arises from how many “orthogonal” views we have of our data.

4.4.1 Single Neurons

I often find it easiest to conceptualise parametric lenses and pre-lenses as layers of a neural network—or as an entire network even—but components of a model can be atomised right down to individual operations. Here’s how the pre-lens for a single neuron can be built from weights, biases, and an activation function. A weight pre-lens is

```
weightLens :: TriLens C C -- b db
             (Vect C, Vect C) (Vect C, Vect C) -- m dm
             (Vect C) (Vect C) -- p dp
             (Vect C) (Vect C) -- a da.
```

Biases are simple because no message needs to be passed (*i.e.* no residues) due to the shifting happening irrespective of the input values. Their type is

```
biasLens :: TriLens C C () () C C C C
```

where `()` represents “Nil” or “Nothing.”

On the flip side, activation functions don’t require parameterisation and have a signature of

```
activationLens :: TriLens C C C C () () C C
```

¹⁶Again, up to unit laws.

With this formulation, a neuron is simply

```
neuronLens = activationLens.biasLens.weightLens,
```

with Type signature:

```
neuronLens :: TriLens C C
              ((Vect C, Vect C), C) ((Vect C, Vect C), C)
              (Para C) (Para C)
              (Vect C) (Vect C).
```

which results from renaming $((Vect\ C, C))$ as our parameters, $Para\ C$, and compressing $()$ as needed using our unit laws. If written in full, m and dm become increasingly nested tuples exactly as p and dp do (though also have a and da along for the ride).

4.4.2 Layers

Layers can then be built by composing neurons in parallel, via a product or vectorisation. Since the inputs will be identical, we can also “flatten” along the inputs to obtain

```
nnLayer :: TriLens (Vect C) (Vect C)
           ((Matr C, Vect C), Vect C) ((Matr C, Vect C), Vect C)
           (Vect (Para C)) (Vect (Para C))
           (Vect C) (Vect C)
```

where $(Matr\ C)$ is used as shorthand for $(Vect\ Vect\ C)$. Also note that we could apply the vectorisation inside $Para$ instead, resulting in the canonical $(Matr\ C, Vect\ C)$ parameters. Letting C be $Reals$ (or $Type\ Double$ in Haskell) means that our lenses are morphisms in $PreLens(\mathbf{Smooth})$. For simplicity, I will use R to express $Double$, Vec to express $Vect\ R$, and Mat to express $Matr\ R$.

However, if we go back to ignoring the details of implementing $PreLens(\mathbf{Smooth})$, and use our x/dx notation, we could write a single neuron as

```
neuronLens :: TriLens b db m dm p dp a da
```

and a layer as

```
nnLayer :: TriLens [b] [db] [m] [dm] [p] [dp] a da.17
```

Losses and learning rates can be expressed as

```
lossLayer :: TriLens l dl m dm b db b db
```

and

```
rateLayer :: TriLens () () () () r dr l dl
```

respectively, where `l` and `dl` represents the values and reverse derivatives for our losses, while `r` and `dr` correspond to rates.

However, we start to see some flexibility in our representation here. Eventually we know that we need `dp` to be used as an update for the next iteration of parameters, `p`, so instead of using the learning rate as a lens that “caps off” the network, it could be used as a scaling factor when stitching together `dpi` and `pi+1`.

Notice how all but one pair of values in `nnLayer` get bundled together. This pattern will be present in the remaining sections. We start with an arbitrary piece of our network that is Type

```
TriLens b db m dm p dp a da
```

and either we

- apply one of our various `dimap` operations, depending on which of `a`, `p`, or `m` (and the corresponding reverse derivative) we are modifying, or;
- we apply `(replicate N)` first, which makes `N` identical copies of the given pre-lens, to get a pre-lens of Type

```
TriLens [b] [db] [m] [dm] [p] [dp] [a] [da].
```

Then, we `dimap` to perform whatever type of “zipping,” “compressing,” or “accumulating” we want.

In the example of building a layer, we have a function `toLayer` that has Type

```
toLayer :: Int -> TriLens b db m dm p dp a da  
-> TriLens [b] [db] [m] [dm] [p] [dp] a da
```

defined simply as

```
toLayer N lens = dimapA (replicate N) fst ((replicate N) lens).
```

¹⁷I use the square bracket notation here, in line with Milewski’s convention in [102], for clarity. When discussing a given lens, you can take the Types of any given `x` and `dx` as arbitrary, while `[x]` and `[dx]` will be *relative to the original Types*. Specifically, `x` and `dx` may be (and often are) already vectorised themselves.

4.4.3 Batching

The above two sections give us everything needed to do one step of basic supervised learning. However, there's many other features that we typically want to employ to improve performance or modify the behaviour of the network. One such feature is taking multiple samples at a time, called a "batch." This allows us to learn on multiple data points in parallel.

Using the same strategy as above, we can express batching as a function of Type

```
batch :: Int -> TriLens b db m dm p dp a da
      -> TriLens [b] [db] [m] [dm] p dp [a] [da].
```

Since we are using `p` and `dp` as our fixed Types this time, we use `dimapP` instead of `dimapA` as follows:

```
batch N lens = dimapP (replicate N) sum ((replicate N) lens).
```

Batch Normalisation

As a brief aside, I want to quickly point out that we can also perform batch normalisation when we are batching across inputs. Doing so can be done simply with

```
neuronBNLens = neuronLens.batchNormLens
```

where the Type for `batchNormLens` is

```
TriLens a da m dm p dp a da,
```

provided that `a` and `da` are already vectorised.

4.4.4 Re-parametrisations

As a final demonstration, let's take a look at how we might handle re-parameterisations (in this case, momentum). Doing so with the `fwd/bwd` formulation of a pre-lens is quite straightforward (though clunky) and exactly what you'd expect:

```
momentum :: R -> PreLens b db (p, a) (p, a) p dp a da
          -> PreLens b db
              ((p, s), a) ((p, s), a)
              (p, s) (dp, ds)
              a da,
```

implemented as

```
momentum gamma (PreLens fwd bwd) = (PreLens fwd' bwd')
```

where

```
fwd' ((p, s), a) = ((p, s), a), snd $ fwd (p, a)
```

and

```
bwd' (((p, s), a), db) =  
  let (dp', da) = bwd ((p, a), db)  
  let ds = (-gamma*s + dp')  
  let dp = p + ds  
  in ((dp, ds), da).
```

Note that, in Haskell, the above code won't compile as-is, because we're now dictating *specific* operations that our various Types need to Type check with (+, *, etc.). However, setting all the appropriate x/dx pairs to be Reals, \mathbb{R} , resolves this issue. At this point, we can simply use our converter defined in Section 4.3.3, `toTrim`, to convert the above pre-lens into it's Trimbara representation.

There is certainly a way to define re-parameterisations using only our various `dimap` or `alpha` and `beta` functions, rather than using the `fwd/bwd` representation and converting. However, I suspect that the result will mirror `toTrim`, modifying `fwd` and `bwd` there as necessary.

4.5 Running on MNIST

Milewski built a very simple neural network as a proof of concept for pre-lenses [101]. This code is tested on a dataset of four inputs that are four-dimensional. Though I think the value of this work is mainly theoretical, I have expanded upon that code so that it runs on a naturalised dataset rather than a synthetic one. All the code was written in Haskell, and is therefore purely functional.

4.5.1 Changes Made to Existing Model

Building a model for MNIST involved a few necessary changes to the code, and highlighted a few issues with the existing code that didn't manifest on such a small dataset. First, Milewski's

original code didn't contain a dataloader. I built some rudimentary code that reads in training data from a file, and can separately load in data for the purposes of testing. This code also randomises the order that inputs are presented across training epochs.

Second, the original code had a small bug that caused it to read in the same batch of inputs during training, even if additional inputs were passed in for training. Because the toy dataset only had four inputs, it was sufficient to have a single batch of size four. I tested the model on one hundred digits of MNIST. This was enough inputs that it was impossible to train on a single batch (attempting to do so results in a memory out-of-bounds error).

Third, the original code saved a checkpoint of the model parameters every iteration. While this would be ideal practice, the size of the larger model required for MNIST meant that saving the model parameters every iteration resulted in extreme slowdowns during training. My working memory (which was, admittedly, quite low at 16GB) quickly ran out, causing the code to read and write the data structure storing parameters into long-term memory every iteration. This slowed training from approximately fifteen seconds per iteration to approximately two *minutes* (an 8 times increase in duration).

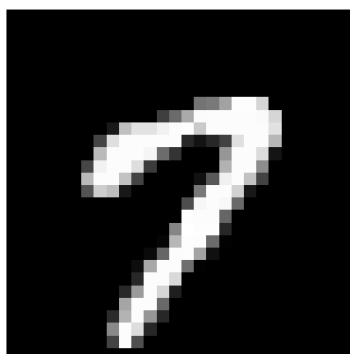
4.5.2 Details of Training

I trained a network built using pre-lenses on a subset of MNIST containing 100 digits. The network had an input layer size of 784, matching the size of the input images, two hidden layers of size 50, and an output layer of size 10. The output size corresponds to the classification vectors, which contained all negative ones except for a positive one in the position that corresponds to the relevant digit (ordered 0 – 9). Model parameters were initialised between values of -1 and 1 .

The model was trained for 120 epochs. The batch size was set to 20 inputs, thus five iterations were required to go through the whole dataset. The optimiser was simply stochastic gradient descent with a learning rate of 0.05. An example classification made by the trained network is provided in Figure 4.7. Since I am implementing simple backprop, the empirical results obtained are consistent with all other results that researchers have seen on MNIST.

4.5.3 Changes Needed in Future Work

The code leaves a lot to be desired in scalability and usability. It is very terse and readable in its current state, which is great for highlighting the value of pre-lenses. However, I suggest the following improvements as immediate next steps for creating better software.



`[-0.9996, -0.9997, -0.9992, -0.9999, -0.9908, -0.9981, -0.9998, 0.80645, -0.9980, -0.8465]`

Figure 4.7: An example input to the network, along with the classification made by the network after training.

The most obvious is parallelisation. The choice to train on 100 digits of MNIST rather than the original 60,000 was entirely practical, so that the network would train in a reasonable time. Haskell has far less support for GPU acceleration compared to languages like Python. Thus, the code ran on my CPU. There are currently some libraries that provide CUDA support for Haskell, but their maintenance varies. Future work (which I have partially implemented) will implement GPU acceleration, allowing for training on larger datasets. More work is also needed to improve the primitive dataloader that I've constructed to read in training data. My current code works for .csv files that are formatted in a particular manner. A proper dataloader will allow for different file formats, optimised streaming (currently the entire dataset is read into memory), and a simpler means of including training, testing, and validation data. Finally, the code would benefit from re-implementing checkpointing in a manner that is more efficient, and save the model parameters to a permanent location. This last item is fairly approachable but was unnecessary for my proof of concept.

I'm certain there are other design considerations that I have missed, by virtue of not being a machine learning engineer. In the long term, I hope to see a Haskell library akin to keras or PyTorch, that implements pre-lenses, or to contribute to a currently existing library, such as HLearn [72] or HaskTorch [5].

4.6 Generative Models

There’s one last symmetry to discuss. As mentioned in Section 4.2.2, Cruttwell *et al.* note that the very process of learning can be interpreted as the `put` operation of a lens, by reversing the roles of parameters and input/output pairs. Now parameters will be fed into our Learner, while training data (inputs and labels both) will act as the “parameters.” It may be easier to understand this by viewing Figure 4.8. This process can be repeated as many times as needed until some stop criteria is met—usually a maximum number of iterations, or reaching sufficiently small improvements in accuracy/loss. I’ve chosen to call this operation `disc` rather than `put` for increased clarity in the remaining text.

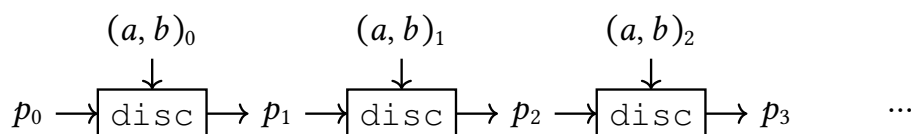


Figure 4.8: Learning as an iterative process of using training data, (a, b) , to create new parameters, p . Note that the role of inputs and outputs, typically called a and b respectively in this document, is now flipped with the parameter role.

I think this perspective is very cool and meta, showing just how general the framework of parametric lenses is. However, it leaves the natural question of what the corresponding `set` operation is, which goes unanswered in that paper. I think the answer is that it corresponds to a generative operation, `gen`¹⁸.

With pre-lenses, we can understand these operations as each half of a pre-lens. I’ve included a picture of this perspective in Figure 4.9. What’s also nice about this perspective is that it is an instance where a non-trivial mapping from m to dm is intuitive.

Let’s consider the Type signature of `gen`. It takes as input dm and dp . Thus far, to ensure that we could trace across m and dm , we’ve insisted that $m == dm := (p, a)$ which, for `disc` and `gen` is

$$dm/m :: ((a, b), p).$$

Thus, our Type signature for `gen` is

$$gen :: ((a, b), p), dp) \rightarrow ((da, db), dp).$$

¹⁸While mention of generative models is also present in [24], it is in the context of modifying the parametric lens that controls the loss function.

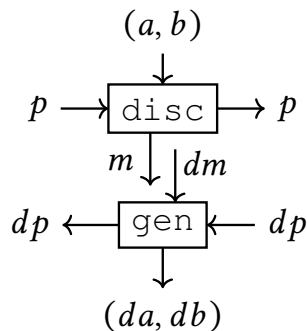


Figure 4.9: A picture of how generative mappings relate to their discriminative counterparts.

The ingredients needed to perform a generative task are all there. We have how well a model, parameterised by p , performs on training data (a, b) and can use that to create new training data, (da, db) . If we use a trivial mapping between m and dm , then we can recover the type of generation discussed in [24], namely “super stimulus” or “deep dreaming.” However, if we instead flip the value of m when deciding on the input, dm , then we can interpret gen as generating an adversarial training point instead.

There’s an interesting point here that dp seems to tag along the whole time, but doesn’t contribute meaningfully to gen . I think that this is actually natural. Consider that the stop condition for learning is typically when the loss is sufficiently low. In this case, we can expect that $dp \approx 0$.

4.7 Discussion and Future Work

What I find appealing about these pre-lenses is how much they lend themselves to all of the discussed symmetries. It makes me wonder if there are even more modifications to vanilla supervised learning that can be captured by pre-lenses (I mention attention briefly in Section 5.3.1).

The lack of scalability in the current implementation means that the code doesn’t lend itself to more rigorous empirical studies. Future work will implement a fully featured dataloader that includes test and validation sets, rather than only training sets. It will also include more modern optimisers, such as Adam. This will allow users to evaluate models on generalisability, rather than just training loss.

Recent work by Riley also discusses how Learners (parametric lenses) are *almost* compact closed [127]. In their work, they note that there is a duality in Learners. I believe this coincides

with the equivalence of **Para** and **co-Para** that Milewski notes in pre-lenses but more work is required to make this equivalence rigorous. The work by Riley defines an involution that doesn't quite recover an original Learner when that involution is applied twice. Their observation of almost compact closedness requires a quotienting operation that doesn't seem to be necessary for pre-lenses (though ensuring Type matching in m and d_m may be related to this quotienting).

I'm also intrigued by other ways to break the equivalence of m and d_m , beyond generative models. There's some immediate possibilities that aren't very interesting (implementing learning rates as an operation from m to d_m , rather than as a pre-lens at the end of a network, for instance), but I'm hopeful that there may be ways to see the full expressive power of this decoupling.

Chapter 5

Conclusion

In this thesis I've reported two distinct projects that I worked on. Broadly, all the work falls under applied category theory, specifically applying it to data science. The first project studied VSAs, which describe how to store and manipulate distributed data that is represented by high-dimensional vectors. The second studied the more familiar landscape of supervised learning, building on the recent wave of work done that describes the categories **Para**, or **Learn**. In this final chapter, I want to summarise these contributions in a slightly more detail, making clear what I think the main points of the work are. Then, I want to provide a few, less formal, speculations on the nature of my current work as well as possible future directions.

5.1 Closing Summary on VSAs

To the best of my knowledge, my colleagues and I provide the first description of VSAs explicitly using categorical language. My contribution generalised from vectors to co-presheaves, and used Kan extensions to determine the optimal bind and bundle operations. I suspect that the category theory community will refine this formalisation, specifically by describing unbinding, unbundling, and similarity; I plan to work on refining this formalisation myself. The proposed directions for future work will hopefully lead towards a more comprehensive categorical foundation of VSAs.

5.2 Closing Summary on Pre-lenses

Pre-lenses generalise the bicategory of parametric learners, unifying them with their dual. The machinery of pre-lenses also enables a number of elegant ways to build/modify networks by taking advantage of their triple Profunctor and triple Tambara module representation. Though following directly off the work done by Milewski, I validated the method on MNIST, implemented re-parameterisations and batch normalisation, and describe how pre-lenses can be used to understand generative models as duals to discriminative models. My hope is to continue improving the codebase, making it more scalable and more user friendly. I also intend to do further research on optics (and possibly pre-optics), which are an important generalisation of lenses.

5.3 Speculations

I want to briefly mention some of the half-baked ideas that I haven't yet been able to rigorously pursue, but have interested me during my studies (and that I hope interest you too). For the sake of brevity, I will be far more loose with the level of background and detail provided.

5.3.1 Attention as Another Axis for Pre-Lens Composition

The novel component of a transformer unit, which are famously used in LLMs, is the attention mechanism (or, more specifically, the *multi-head* attention mechanism). At a high level, this attention mechanism tries to learn how to modify a given embedding depending on some surrounding context. I'm curious if—like batching, layering, and re-parameterisations—attention can be captured as a `dimap` operation on pre-lenses, or products of pre-lenses.

5.3.2 Modelling Predictive Coding with Pre-lenses

In addition to learning across discrete time steps, researchers have described continuous-time models, such as predictive coding [103]. Such models implement learning as the solution to some dynamical system. In predictive coding, this happens by clamping the input and output ends of a network, then letting a collection of connective weights update until the system reaches some internal equilibrium. Researchers have even noted that this process can approximate back-propagation [163].

What makes this model interesting to me is that each “layer” or “unit” of a predictive coding model is composed of two nodes: a value node, and an error node. Each of these nodes has

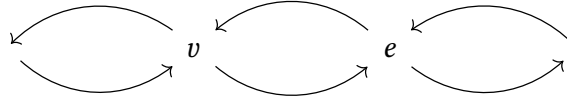


Figure 5.1: A very simplified picture of a predictive coding “unit.” Here, v represents the value node, while e represents the error node. The two-in/two-out nature of the model makes it very tempting to formulate this picture as a pre-lens, however the directions of the arrows don’t seem to line up—at least, not directly out of the box.

two weights going into them and two going out, with one pair going to/from the corresponding node and the other pair going to/from the next layer (see Figure 5.1). I’ve made some extremely preliminary attempts to connect this to the pairs of two-in/two-out mappings that constitute pre-lenses, but haven’t been able to make a one-to-one connection between all of the pieces yet. Still, I’m interested in exploring this further.

5.3.3 Unifying Symbolic Reasoning and Learning

This thesis has contained two distinct lines of research, connected only through category theory. However, something has sat uncomfortably in the back of my mind as I’ve jumped between studying VSAs and pre-lenses. Both VSAs and modern supervised architectures (particularly large models like LLMs) perform tasks by manipulating vectors in a high-dimensional vector space. Does this mean the two can be related in some way?

We could ask if they are equivalent in some way. Though the answer may be “yes” at some level of abstraction, I don’t see the relationship. Then the question for me is whether there is an elegant way to combine the two. This seems more promising, and I think it would be interesting to see some hybridisation of the two models. Unfortunately, efforts to combine learning and symbolic reasoning/logic typically struggle to determine when each method should be used over the other. It’s hard to imagine a “controller” for the two that isn’t heavily engineered or contrived. Thus, I worry the nuances of combining the two systems would be filled with many confounding subtleties. Further, I can easily imagine such lines of working contributing to the continued “AI-ification” of research, which I’ve already bemoaned at length.

5.4 Closing Statement

There's some clear next steps in my research. As mentioned already, the pre-lens codebase needs a bit of work to make it fully featured and scalable. For VSAs, there's also determining how to best describe the connection between similarity, and bind and bundle. My immediate goals are those two items.

However, in the longer term, I intend to move away from machine learning and towards theoretical computer science and mathematics. I have a long way to go before I feel like I will be a fully mature category theorist, but I have found that studying category theory is what I enjoy most. Working towards that goal by way of computer science has been extremely helpful. As mentioned at the start of this document, working with a compiler really forces you to be rigorous about what you're describing in a way that complements the abstract nature of categories.

On a related note, I've become quite keen on exploring this relationship between category theory and computer science from a pedagogical perspective. I am quite convinced that earlier exposure to abstraction helps with learning computer science, and view categories as embodying the two main pillars of CS: composition and abstraction. I also think that CS is a great way to introduce young mathematicians to categories much earlier than graduate school. I plan on building an educational programme centered on these ideas throughout the remainder of my career.

To that end, I hope that this document has served you, the reader, in learning about category theory and machine learning. I'm always impressed by how categories can be described with so few axioms, yet be so expressive. That said, you don't get any knowledge for free: category theory is rife with jargon, so understanding simple expressions usually requires extensive knowledge of the vocabulary used in those expressions. For this thesis, I tried to strike the right balance of being detailed enough that all relevant concepts would be clear, while still being concise enough not to distract from what's important. Ultimately, I hope that you may appreciate categories as I have, and that this thesis might inspire those new to the area to study it further with me.

-- the end

References

- [1] 2-category. <https://ncatlab.org/nlab/show/2-category>. Accessed: 2025-07-24.
- [2] Bicategories. <https://ncatlab.org/nlab/show/bicategory>. Accessed: 2025-07-24.
- [3] Ends and coends. <https://bartoszmilewski.com/2017/03/29/ends-and-coends/>. Accessed: 2025-12-28.
- [4] field - constructive notions. <https://ncatlab.org/nlab/show/field#constructive>. Accessed: 2025-12-29.
- [5] Hasktorch. <http://hasktorch.org/>. Accessed: 2025-08-28.
- [6] Lawvere metric spaces. <https://ncatlab.org/nlab/show/metric+space#LawvereMetricSpace>. Accessed: 2025-12-29.
- [7] Lenses, prisms, and optics in haskell: Mastering data manipulation. <https://softwarepatternslexicon.com/haskell/functional-design-patterns/lenses-prisms-and-optics/>. Accessed: 2025-12-22.
- [8] Monoidal category. <https://ncatlab.org/nlab/show/monoidal+category>. Accessed: 2025-11-05.
- [9] powering. <https://ncatlab.org/nlab/show/powering>. Accessed: 2025-12-28.
- [10] rig categories. <https://ncatlab.org/nlab/show/rig+category>. Accessed: 2025-12-23.

- [11] Samson Abramsky and Bob Coecke. Categorical quantum mechanics. *Handbook of quantum logic and quantum structures*, 2:261–325, 2009.
- [12] Takanori Adachi and Yoshihiro Ryu. A category of probability spaces. *arXiv preprint arXiv:1611.03630*, 2016.
- [13] Britt Anderson. Category theory for cognitive science workshop/tutorial. <https://cognitivesciencesociety.org/wp-content/uploads/2022/05/cat-theory-cog-sci-workshop.pdf>. Accessed: 2025-07-24.
- [14] Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago M Paixao, Filipe Mutz, et al. Self-driving cars: A survey. *Expert Systems with Applications*, 165:113816, 2021.
- [15] John C Baez and Blake S Pollard. A compositional framework for reaction networks. *Reviews in Mathematical Physics*, 29(09):1750028, 2017.
- [16] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7(48):1–13, 2014.
- [17] Anthony J Bell and Terrence J Sejnowski. The “independent components” of natural scenes are edge filters. *Vision research*, 37(23):3327–3338, 1997.
- [18] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębniak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.
- [19] Bernd Bischl, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, Theresa Ullmann, Marc Becker, Anne-Laure Boulesteix, et al. Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(2):e1484, 2023.
- [20] Quentin Cappart, Didier Chételat, Elias B Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. *J. Mach. Learn. Res.*, 24:130–1, 2023.
- [21] Miguel A Carreira-Perpinan and Geoffrey Hinton. On contrastive divergence learning. In *International workshop on artificial intelligence and statistics*, pages 33–40. PMLR, 2005.

- [22] Christopher Clark and Amos J. Storkey. Teaching deep convolutional neural networks to play Go. *arXiv preprint*, abs/1412.3409, 2014.
- [23] Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin, and Dorette Pronk. Reverse derivative categories. *arXiv preprint arXiv:1910.07065*, 2019.
- [24] Geoffrey SH Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning. In *European Symposium on Programming*, pages 1–28. Springer International Publishing Cham, 2022.
- [25] Giovanni de Felice. Categorical tools for natural language processing. *arXiv preprint arXiv:2212.06636*, 2022.
- [26] Lance De Vine and Peter Bruza. Semantic oscillations: Encoding context and structure in complex valued holographic vectors. In *2010 AAAI fall symposium series*, 2010.
- [27] Deepmind. Alphafold. <https://www.deepmind.com/research/highlighted-research/alphafold>, 2018. Accessed: 2023-09-21.
- [28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.
- [29] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [30] Timothy Dozat. Incorporating Nesterov momentum into Adam. *International Conference on Learning Representations*, 2016.
- [31] Harold Edson Driver and Alfred Louis Kroeber. *Quantitative expression of cultural relationships*, volume 31. Berkeley: University of California Press, 1932.
- [32] Nicole Sandra-Yaffa Dumont and Chris Eliasmith. Accurate representation for spatial cognition using grid cells. In *Joint International Conference on Cognitive Science*, 2019.
- [33] Nicole Sandra-Yaffa Dumont, P Michael Furlong, Jeff Orchard, and Chris Eliasmith. Exploiting semantic information in a spiking neural slam system. *Frontiers in Neuroscience*, 17, 2023.

- [34] Francis Ysidro Edgeworth. On the probable errors of frequency-constants. *Journal of the Royal Statistical Society*, 71(2):381–397, 1908.
- [35] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58:231–294, 1945.
- [36] Chris Eliasmith. *How to build a brain: A neural architecture for biological cognition*. Oxford University Press, 2013.
- [37] Chris Eliasmith, Terrence C Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *science*, 338(6111):1202–1205, 2012.
- [38] Deniz Erdogmus and Jose C Principe. An error-entropy minimization algorithm for supervised training of nonlinear adaptive systems. *IEEE Transactions on Signal Processing*, 50(7):1780–1786, 2002.
- [39] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426, 2019.
- [40] Daniele Fanelli. Negative results are disappearing from most disciplines and countries. *Scientometrics*, 90(3):891–904, 2012.
- [41] Jerry A Fodor and Zenon W Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988.
- [42] E. Paxon Frady, Denis Kleyko, Christopher J. Kymn, Bruno A. Olshausen, and Friedrich T. Sommer. Computing on functions using randomized vector representations (in brief). In *ACM International Conference Proceeding Series*, pages 115–122, 3 2022.
- [43] E Paxon Frady and Friedrich T Sommer. Robust computation with rhythmic spike patterns. *Proceedings of the National Academy of Sciences*, 116(36):18050–18059, 2019.
- [44] Edward Paxon Frady, Denis Kleyko, and Friedrich T Sommer. Variable binding for sparse distributed representations: Theory and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 34(5):2191–2204, 2021.
- [45] Karl Friston. The free-energy principle: a unified brain theory? *Nature reviews neuroscience*, 11(2):127–138, 2010.

- [46] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):322–333, 1969.
- [47] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.
- [48] P Michael Furlong and Chris Eliasmith. Bridging cognitive architectures and generative models with vector symbolic algebras. In *Proceedings of the AAAI Symposium Series*, volume 2, pages 262–271, 2023.
- [49] P Michael Furlong and Chris Eliasmith. Modelling neural probabilistic computation using vector symbolic architectures. *Cognitive Neurodynamics*, 18(6):1–24, 2024.
- [50] Stephen I Gallant and T Wendy Okaywe. Representing objects, relations, and sequences. *Neural computation*, 25(8):2038–2078, 2013.
- [51] Dan Garisto. Google AI beats top human players at strategy game StarCraft II. *Nature*, 2019.
- [52] Bruno Gavranović. Fundamental components of deep learning: A category-theoretic approach, 2024.
- [53] Ross W Gayler. Multiplicative binding, representation operators & analogy (workshop poster). 1998.
- [54] Ross W Gayler. Vector symbolic architectures answer Jackendoff’s challenges for cognitive neuroscience. In *Joint International Conference on Cognitive Science*, pages 133–138, 2003.
- [55] Ross W Gayler. Vector symbolic architectures answer Jackendoff’s challenges for cognitive neuroscience. *arXiv preprint cs/0412059*, 2004.
- [56] Ross W Gayler and Simon D Levy. A distributed basis for analogical mapping. In *New Frontiers in Analogy Research; Proc. of 2nd Intern. Analogy Conf*, volume 9, 2009.
- [57] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L^AT_EX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.
- [58] Jan Gosmann and Chris Eliasmith. Optimizing semantic pointer representations for symbol-like processing in spiking neural networks. *PloS One*, 11(2):e0149928, 2016.

- [59] Jan Gosmann and Chris Eliasmith. Vector-derived transformation binding: An improved binding operation for deep symbol-like processing in neural networks. *Neural computation*, 31(5):849–869, 2019.
- [60] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.
- [61] Sander Greenland, Stephen J Senn, Kenneth J Rothman, John B Carlin, Charles Poole, Steven N Goodman, and Douglas G Altman. Statistical tests, p values, confidence intervals, and power: a guide to misinterpretations. *European journal of epidemiology*, 31(4):337–350, 2016.
- [62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [63] Robert Hecht-Nielsen et al. Context vectors: general purpose approximate meaning representations self-organized from raw data. *Computational intelligence: Imitating life*, 3(11):43–56, 1994.
- [64] Chris Heunen and Andre Kornell. Axioms for the category of hilbert spaces. *Proceedings of the National Academy of Sciences*, 119(9):e2117024119, 2022.
- [65] Chris Heunen, Andre Kornell, and NESTA Van Der Schaaf. Axioms for the category of hilbert spaces and linear contractions. *arXiv preprint arXiv:2211.02688*, 2022.
- [66] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1):31, 1991.
- [67] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [68] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.
- [69] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
- [70] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat’s striate cortex. *The Journal of physiology*, 148(3):574, 1959.

- [71] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.
- [72] Mike Izbicki. Github: Hlearn. <https://github.com/mikeizbicki/HLearn>. Accessed: 2025-08-28.
- [73] Pentti Kanerva. Binary spatter-coding of ordered k-tuples. In *Artificial Neural Networks—ICANN 96: 1996 International Conference Bochum, Germany, July 16–19, 1996 Proceedings 6*, pages 869–873. Springer, 1996.
- [74] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1:139–159, 2009.
- [75] G Maxwell Kelly. Coherence theorems for lax algebras and for distributive laws. In *Category Seminar: Proceedings Sydney Category Theory Seminar 1972/1973*, pages 281–375. Springer, 2006.
- [76] Gregory Maxwell Kelly. *Basic concepts of enriched category theory*, volume 64. CUP Archive, 1982.
- [77] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. ieee, 1995.
- [78] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [79] Donald Knuth. *The T_EXbook*. Addison-Wesley, Reading, Massachusetts, 1986.
- [80] Brent Komer, Terrence C Stewart, Aaron Voelker, and Chris Eliasmith. A neural representation of continuous space using fractional binding. In *CogSci*, pages 2038–2043, 2019.
- [81] Hans-Peter Kriegel, Peer Kröger, Jörg Sander, and Arthur Zimek. Density-based clustering. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(3):231–240, 2011.
- [82] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 and CIFAR-100 datasets. <https://www.cs.toronto.edu/~kriz/cifar.html>, 2009. Accessed: 2023-09-21.

- [83] Mika Laiho, Jussi H. Poikonen, Pentti Kanerva, and Eero Lehtonen. High-dimensional computing with sparse vectors. In *IEEE Biomedical Circuits and Systems Conference: Engineering for Healthy Minds and Able Bodies, BioCAS 2015 - Proceedings*, 12 2015.
- [84] Nick Lally. “it makes almost no difference which algorithm you use”: on the modularity of predictive policing. *Urban geography*, 43(9):1437–1455, 2022.
- [85] Leslie Lamport. *TEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.
- [86] Janet M Lang, Kenneth J Rothman, and Cristina I Cann. That confounded p-value, 1998.
- [87] Miguel L Laplaza. Coherence for distributivity. In *Coherence in categories*, pages 29–65. Springer, 2006.
- [88] F William Lawvere. Metric spaces, generalized logic, and closed categories. *Rendiconti del seminario matematico e fisico di Milano*, 43:135–166, 1973.
- [89] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient back-prop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.
- [90] Yann LeCun and Corinna Cortes. The MNIST database of handwritten digits. <http://yann.lecun.com/exdb/mnist/>, 1998. Accessed: 2023-09-21.
- [91] Mena Leemhuis and Özgür L Özçep. Analogical proportions and betweenness. 2023.
- [92] Fosco Loregian. This is the (co) end, my only (co) friend. *arXiv preprint arXiv:1501.02503*, 643, 2015.
- [93] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1998.
- [94] Ujjwal Maulik and Sanghamitra Bandyopadhyay. Genetic algorithm-based clustering technique. *Pattern recognition*, 33(9):1455–1465, 2000.
- [95] Harald Maurer. *Cognitive science: Integrative synchronization mechanisms in cognitive neuroarchitectures of modern connectionism*. CRC Press, 2021.
- [96] Hermann Mayer, Faustino Gomez, Daan Wierstra, Istvan Nagy, Alois Knoll, and Jürgen Schmidhuber. A system for robotic heart surgery that learns to tie knots using recurrent neural networks. *Advanced Robotics*, 22(13-14):1521–1537, 2008.

- [97] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.
- [98] Matthew Di Meglio and Chris Heunen. Dagger categories and the complex numbers: Axioms for the category of finite-dimensional hilbert spaces and linear contractions, 2024.
- [99] Cade Metz. ‘the godfather of a.i.’ leaves google and warns of danger ahead. <https://www.nytimes.com/2023/05/01/technology/ai-google-chatbot-engineer-quits-hinton.html>. Accessed: 2023-05-01.
- [100] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.
- [101] Bartosz Milewski. Github: Deeplearning/prelens. <https://github.com/BartoszMilewski/DeepLearning/tree/main/PreLens>. Accessed: 2025-08-28.
- [102] Bartosz Milewski. Neural networks, pre-lenses, and triple tambara modules. <https://bartoszmilewski.com/2024/03/22/neural-networks-pre-lenses-and-triple-tambara-modules/>. Accessed: 2025-05-22.
- [103] Beren Millidge, Anil Seth, and Christopher L Buckley. Predictive coding: a theoretical and experimental review, 2022.
- [104] Marvin L Minsky and Seymour A Papert. Perceptrons: expanded edition, 1988.
- [105] National Academies of Sciences, Medicine, Policy, Global Affairs, Board on Research Data, Division on Engineering, Physical Sciences, Committee on Applied, Theoretical Statistics, Board on Mathematical Sciences, et al. *Reproducibility and replicability in science*. National Academies Press, 2019.
- [106] OpenAI. Introducing ChatGPT. <https://openai.com/blog/chatgpt>, 2022. Accessed: 2023-09-21.
- [107] Jeff Orchard and Russell Jarvis. Hyperdimensional computing with spiking-phasor neurons, 2023.

- [108] Alexander G Ororbia and Mary Alexandria Kelly. A neuro-mimetic realization of the common model of cognition via hebbian learning and free energy minimization. In *Proceedings of the AAAI Symposium Series*, volume 2, pages 369–378, 2023.
- [109] Judea Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *Probabilistic and Causal Inference: The Works of Judea Pearl*, pages 129–138. ACM, 2022.
- [110] Karl Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.
- [111] Paolo Perrone. Notes on category theory with examples from basic mathematics. *arXiv preprint arXiv:1912.10642*, 2019.
- [112] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Reinforcement learning for humanoid robotics. In *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pages 1–20, 2003.
- [113] Steven Phillips and William H Wilson. Categorical compositionality: A category theory explanation for the systematicity of human cognition. *PLoS computational biology*, 6(7):e1000858, 2010.
- [114] Tony A Plate. Holographic recurrent networks. *Advances in neural information processing systems*, 5, 1992.
- [115] Tony A Plate. *Distributed representations and nested compositional structure*. PhD thesis, University of Toronto, Toronto, ON, Canada, 1994.
- [116] Tony A Plate. Holographic reduced representations. *IEEE Transactions on Neural networks*, 6(3):623–641, 1995.
- [117] Tony A Plate. Holographic reduced representation: Distributed representation for cognitive structures. 2003.
- [118] Frank Qiu. *Graph Embeddings, Disentanglement, and Algorithm Maps*. University of California, Berkeley, 2023.
- [119] Dmitri A. Rachkovskij. Representation and processing of structures with binary sparse distributed codes. *IEEE transactions on Knowledge and Data Engineering*, 13(2):261–276, 2001.

- [120] Dmitri A Rachkovskij and Ernst M Kussul. Binding and normalization of binary sparse distributed representations by context-dependent thinning. *Neural Computation*, 13(2):411–452, 2001.
- [121] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*, pages 28492–28518. PMLR, 2023.
- [122] Mohammad Raeini. The golden era of mathematics: From computer science to data science. Available at SSRN 4686564, 2024.
- [123] Ali Rahimi. Machine learning has become alchemy. https://www.youtube.com/watch?v=x7psGHgatGM&ab_channel=PreserveKnowledge, 2017. NIPS 2017 Test of time award.
- [124] Seema Rawat and Sanjay Meena. Publish or perish: Where are we heading? *Journal of research in medical sciences: the official journal of Isfahan University of Medical Sciences*, 19(2):87, 2014.
- [125] Alpha Renner, Yulia Sandamirskaya, Friedrich Sommer, and E Paxon Frady. Sparse vector binding on spiking neuromorphic hardware using synaptic delays. In *Proceedings of the International Conference on Neuromorphic Systems 2022*, pages 1–5, 2022.
- [126] Douglas A Reynolds et al. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.
- [127] Mitchell Riley. Learners are almost free compact closed. 2024.
- [128] Ronald L Rivest. On the notion of pseudo-free groups. In *Theory of Cryptography Conference*, pages 505–521. Springer, 2004.
- [129] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
- [130] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
- [131] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning internal representations by error propagation, 1985.

- [132] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *Advances in neural information processing systems*, 31, 2018.
- [133] Tom Schaul and Jürgen Schmidhuber. Metalearning. *Scholarpedia*, 5(6):4650, 2010.
- [134] Kenny Schlegel, Peer Neubert, and Peter Protzel. A comparison of vector symbolic architectures. *Artificial Intelligence Review*, 55(6):4523–4555, 2022.
- [135] Benjamin Schrauwen, David Verstraeten, and Jan Van Campenhout. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th european symposium on artificial neural networks*. p. 471-482 2007, pages 471–482, 2007.
- [136] Peter Selinger. A survey of graphical languages for monoidal categories. *New structures for physics*, pages 289–355, 2011.
- [137] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.
- [138] Nolan Shaw. The computational advantages of intrinsic plasticity in neural networks. Master’s thesis, University of Waterloo, 2019.
- [139] Nolan Peter Shaw, Tyler Jackson, and Jeff Orchard. Biological batch normalisation: How intrinsic plasticity improves learning in deep neural networks. *Plos one*, 15(9):e0238454, 2020.
- [140] David Sherrington and Scott Kirkpatrick. Solvable model of a spin-glass. *Physical review letters*, 35(26):1792, 1975.
- [141] Dan Shiebler, Bruno Gavranović, and Paul Wilson. Category theory in machine learning, 2021.
- [142] Patrick E ShROUT and Joseph L Rodgers. Psychology, science, and knowledge construction: Broadening perspectives from the replication crisis. *Annual review of psychology*, 69(1):487–510, 2018.
- [143] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

- [144] Toby B Smithe. Radically compositional cognitive concepts. *arXiv preprint arXiv:1911.06602*, 2019.
- [145] Toby St Clere Smithe. Compositional active inference i: Bayesian lenses. statistical games. *arXiv preprint arXiv:2109.04461*, 2021.
- [146] Paul Smolensky. On the proper treatment of connectionism. *Behavioral and brain sciences*, 11(1):1–23, 1988.
- [147] Paul Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, 46(1-2):159–216, 1990.
- [148] Hugo Steinhaus et al. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci*, 1(804):801, 1956.
- [149] Terrence Stewart, Xuan Choo, and Chris Eliasmith. Symbolic reasoning in spiking neurons: A model of the cortex/basal ganglia/thalamus loop. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 32, 2010.
- [150] Douglas Summers-Stay. Propositional deductive inference by semantic vectors. In *Intelligent Systems and Applications: Proceedings of the 2019 Intelligent Systems Conference (IntelliSys) Volume 1*, pages 810–820. Springer, 2020.
- [151] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.
- [152] Omar Syed. Applying genetic algorithms to recurrent neural networks for learning network parameters and architecture. Master’s thesis, Case Western Reserve University, 1995.
- [153] Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019.
- [154] Sverrir Thorgeirsson, Tracy Ewen, and Zhendong Su. What can computer science educators learn from the failures of top-down pedagogy? In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, pages 1127–1133, 2025.
- [155] Migel D Tissera and Mark D McDonnell. Enabling ‘question answering’ in the mbat vector symbolic architecture by exploiting orthogonal random matrices. In *2014 IEEE International Conference on Semantic Computing*, pages 171–174. IEEE, 2014.

- [156] Michel Tokic. Adaptive ε -greedy exploration in reinforcement learning based on value differences. In *Annual Conference on Artificial Intelligence*, pages 203–210. Springer, 2010.
- [157] Alexis Toumi. Category theory for quantum natural language processing. *arXiv preprint arXiv:2212.06615*, 2022.
- [158] Avraam Tsantekidis, Nikolaos Passalis, Anastasios Tefas, Juho Kannianen, Moncef Gabbouj, and Alexandros Iosifidis. Forecasting stock prices from the limit order book using convolutional neural networks. In *2017 IEEE 19th conference on business informatics (CBI)*, volume 1, pages 7–12. IEEE, 2017.
- [159] Hendrik P Van Dalen. How the publish-or-perish principle divides a science: The case of economists. *Scientometrics*, 126(2):1675–1694, 2021.
- [160] Jaap Van Oosten. Basic category theory and topos theory. *Lecture course notes. Available electronically at <http://www.staff.science.uu.nl/ooste110/syllabi/cattop16.pdf>*, 2016.
- [161] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [162] Izhar Wallach, Michael Dzamba, and Abraham Heifets. Atomnet: a deep convolutional neural network for bioactivity prediction in structure-based drug discovery. *arXiv preprint arXiv:1510.02855*, 2015.
- [163] James CR Whittington and Rafal Bogacz. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural computation*, 29(5):1229–1262, 2017.
- [164] Dominic Widdows and Trevor Cohen. Reasoning with vectors: A continuous model for fast robust inference. *Logic Journal of the IGPL*, 23(2):141–173, 2015.
- [165] Dominic Widdows, Kristen Howell, and Trevor Cohen. Should semantic vector composition be explicit? can it be linear? *arXiv preprint arXiv:2104.06555*, 2021.
- [166] Dominic Widdows, Kirsty Kitto, and Trevor Cohen. Quantum mathematics in artificial intelligence. *Journal of Artificial Intelligence Research*, 72:1307–1341, 2021.
- [167] Hanzhou Wu, Gen Liu, Yuwei Yao, and Xinpeng Zhang. Watermarking neural networks with watermarked images. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(7):2591–2601, 2020.

- [168] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.
- [169] Gang Xue, Shuiguang Deng, Di Liu, and Zeming Yan. Reaching consensus in decentralized coordination of distributed microservices. *Computer Networks*, 187:107786, 2021.
- [170] Daniel LK Yamins, Ha Hong, Charles F Cadieu, Ethan A Solomon, Darren Seibert, and James J DiCarlo. Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the national academy of sciences*, 111(23):8619–8624, 2014.
- [171] Xin-She Yang. Firefly algorithms for multimodal optimization. In *International symposium on stochastic algorithms*, pages 169–178. Springer, 2009.
- [172] Calvin Yeung, Zhuowen Zou, and Mohsen Imani. Generalized holographic reduced representations. *arXiv preprint arXiv:2405.09689*, 2024.
- [173] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. *ACM sigmod record*, 25(2):103–114, 1996.
- [174] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

APPENDICES

Appendix A

A Critique of Machine Learning Research

I think there are serious ills being done to people by AI models, and the companies and institutions that employ them. However, this appendix will focus mostly on the issues with AI research specifically. With that said, it's hard to ignore the ways in which societal influences and costs interact with academics and industry researchers, so I'll abide only loosely by this restriction.

A.1 Methodological Issues

When starting my Master's degree, I was extremely optimistic about AI, both in the progress it would make and the insights we could gain from that research. The first seed of cynicism wasn't due to the moral failings of the applications (at the time there were only whispers of models used for things such as predictive policing, amidst obvious concerns [84]), but rather the methodological issues with the research. Everyone knows that academia frequently fails to uphold the basic standards of the scientific method—reproduction and replication are almost non-existent [105], negative results rarely see publication [40], and $p = 0.05$ is taken as some gold standard across datasets with no consideration of noise-to-signal ratios or effect sizes [61, 86]—however, some of these issues seem particularly problematic in machine learning. I discuss two: p-hacking and replication.

A.1.1 Systemic P-Hacking

For review, “p-hacking” is a practice in data collection where the data collector impacts the results of the experiment to obtain a desired result (usually one that negates the null hypothesis).

Though they do not directly falsify any data point, they will take stock of the datapoints they've seen thus far and choose the stopping point of data collection dynamically, rather than deciding beforehand how much data to collect. Does the data already confirm what you wanted to see? Stop collecting early. Things aren't looking so good? Maybe a few more samples will nudge it in the right direction...

One of the key practical observations in machine learning is how sensitive models are to the state of their hyper-parameters; number of layers, learning rate, weight initialisation, *etc.* While there has been some work trying to better understand the causal effects of various hyper-parameter design decisions [19], many researchers seem to select hyper-parameters based on past experience and intuition.

This interacts with the black-box nature of ML models in an insidious way. Imagine the following scenario. You're a machine learning researcher. You believe that you can "improve" some pre-existing, state-of-the-art model. So you build your baseline and train it for a week (takes a long time to learn after all). 98.4% test accuracy. Then you build your new model. Again, another week of training. Uh-oh. Your method only got 98.1% test accuracy.

It's so close, but did something go wrong? You are pretty sure that your idea should have a positive impact on classification. There's many possibilities:

1. Maybe there's simply a bug in the code (this has certainly happened to me). Even a sign error somewhere might not be catastrophic in learning, since parameters can change to offset such things, but it might be enough to weaken performance.
2. Maybe there's some interaction between the chosen hyperparameters that favour the baseline over your method.
3. Or maybe the idea just doesn't work.

There's good reasons not to default to option 3 right away. Ignoring the pressure to publish, there very well may be a bug, or there might be an opportunity to *learn* a bit yourself by poking the model in various way to see how it wiggles. However, without knowing what the problem is, you default to options 1 or 2.

This is where the problem starts. The right thing to do would be to save that version of the experiment, model and outcome both, then modify exactly one part of the code and re-run the experiment. The problem with this is that things take so long to properly train that you don't have time to do this iteratively for each issue. So you go through your code and maybe you *do* find a bug, but you also change some hyperparameters. Then you run the experiment again.

This can happen over and over. Maybe at some point you get 98.9% test accuracy. Do you keep going? Most researchers I’ve seen do not. Your chance to publish at a good conference is at stake, and you’ve found the right concoction of variables and initial conditions that give you a positive result. Conveniently, there’s little page space in the proceedings of your prospective venue, so it’s not worth elaborating all the attempts that *didn’t* work (besides, those old experiments contained bugs anyway).

The very nature of the research lends itself to p-hacking. It’s so prevalent, but so hard to enforce any strict rules to prevent it. Models *are* sensitive to small changes in hyperparameters. Establishing the exact details of the experiment beforehand isn’t necessarily feasible. Still, I’m discouraged by how little this issue has been acknowledged by the ML research community.

A.1.2 Replication

The second main methodological problem with ML research is how extraordinary difficult it can even be to repeat another researcher’s experiments. In the previous scenario I described, it was taken for granted that you could build the baseline against which you’ll compare your method. However, even this isn’t always possible for a few reasons. First, the code might not even be available (which relates to Section [A.2.3](#)). Second, even if the code is available, it might require infeasible computational resources to replicate the same results.

In addition to the barriers that prevent replicating experiments, the rewards for doing so are non-existent. Conferences and most journals simply will not publish the same results twice. Replication is seen as work to be done within labs, only for the purposes of making a baseline, rather than valuable work itself. In ML, there’s at least some opportunity to publish based on a new application (“We took ResNet, but dropped it on top of text data rather than images!”) but even these feel somewhat cheap and dishonest compared to doing the unexciting work of repeating the exact same thing twice.

I believe that if AI research wasn’t progressing at such a blistering pace, where state-of-the-art status is supplanted constantly and outdated methods become forgotten, ML researchers would be reckoning with a replication crisis of a comparable (or perhaps even larger) scope to that seen in the social sciences [[142](#)].

A.2 Institutional Issues

While I think the methodological issues raised above are fairly uncontroversial, academia—concretely, universities, conferences, and publications—bears responsibility for poor quality re-

search as well. What follows are the institutional pressures and failings that I have noticed. However, I won't go so far as to deconstruct, for example, the "publish or perish" virus, whose infection continues to afflict academia [124, 159].

A.2.1 Obsession with Performance

The first institutional issue is fairly simple. The cultural obsession with "performance" as the sole measure of success by ML conferences and venues has directed machine learning research along a reductive path. In general, "good" research papers will state something like "Our method achieved higher test accuracy than other methods." Secondary status is reserved for papers that state "Our method ran faster than other methods." Everything else is considered unimportant.

For clarity, *these are important results!* Computer science is obviously computational in nature, so the outcomes are almost always clearly measurable and simple in nature. Further, results that improve computational speed or lower resource requirements directly contribute to addressing the replication issues described above.

The problem is how little recognition goes to research that doesn't fit these two formats. Machine learning models are certainly complicated, and there have been efforts to "open up" black boxes (I feel skeptical of "white box" approaches, but respect that they are at least trying), but they have been quite limited. Obviously, this claim is difficult to validate, because it is describing the absence of something from the literature. My hope is that readers in academia can ask themselves whether they see this bias present in their lives. I also believe that seeing more results that are exploratory, qualitative, or even just first steps towards something ambitious would have a positive impact on the research landscape.

I want to provide a concrete example by describing just the *writing* of the particle swarm paper [77]. That paper outlines a pivotal optimisation method, and is therefore very performance-centric. However, the document largely explores the *steps* towards developing particle swarm, including many of the errors that contributed to the theory for why particle swarm works as well as it does. Seeing more of this narrative style, in contrast to simply presenting final products, is something I find appealing for the purposes of enriching understanding (and is also more enjoyable to read).

Another example of this is a paper that explored how internal layers of a convolutional network do a better job of modelling visual-stream neuronal firing patterns than models that try to explicitly model these patterns this [170]. Though the the network was originally designed to classify images, the focus of the research is on *how* this simple objective causes interesting emergent effects within networks.

A.2.2 Strength of Claims

I also find it somewhat jarring how “over-sold” many modern AI papers feel in their writing. Not only does every method have to “beat” all other methods, but it also needs to be maximally novel and open the way for benefitting all applications. Though such claims are sometimes merited, not every paper needs to be the next ResNet or AlphaGo.

Take, as an example, the Firefly algorithm paper [171]. Ignoring the fact that it is egregiously misleading about its runtime (using “number of steps” as the measure of efficiency, without reporting how expensive those steps are), it also frames itself as this revolutionary method that can supplant particle swarm. This, and many other lines of research falling under “biologically-inspired optimisation methods,” are constantly claiming to be the best method, but at this point there’s a comically large number of methods all making the same claim. I especially resent when such claims are made of models that aren’t shared, or can’t easily be re-trained by the typical user. This relates to the next topic.

A.2.3 Transparency

I am extremely disappointed with how tolerant academia has been of industry researchers being allowed to publish at venues while keeping their work behind a proprietary wall. Companies such as Google, Microsoft, Huawei, Facebook, Amazon, and OpenAI¹ frequently present results at major conferences, while only disclosing a fraction of the work that contributes to their results. If companies wish to maintain proprietary ownership of the code they build, then their published results should receive little respect and recognition.

Further, these same companies have extremely tight connections directly with universities that lead to inaccessible research. Graduate students are frequently given research funding from companies to pursue projects that may not even be published. Such research projects is met with celebration by university administrators, who see revenue coming into their campuses, but ignores that copyright and NDAs often rob their students of any rights to this work. There’s also the issue of how distinguished professors will receive considerable compensation for industry consultation that is actually done by their students. The public, who I believe also have a right to our work, have no hope of seeing any knowledge or benefits from such work.

Finally, though it isn’t specific to machine learning, I think it’s important to continue the refrain of decrying paid research publications. These publishers profit immensely off of knowledge they don’t produce (and that cost comparative pennies to disseminate—their stated reason

¹The irony kills me.

for existing), and I think it's shameful that these profits are enabled by academics whose service requirements can be met by volunteering as reviewers or editors for such publications.

A.3 Moral Issues

I also think that researchers have failed to take personal responsibility for the projects they choose to pursue. I largely accept the line of thinking that “knowledge is amoral; its use dictates its utility.” I also don't think we should let the fear of deleterious secondary research projects prevent us from pursuing primary research on fundamental truths. Where I think criticism is rightly earned is in the number of academics who pursue projects that are morally questionable, at best.

One such example is in the amount of research that goes towards applying AI models to everything from the banal offenses of targeted advertisements, to the truly dystopian applications of police and company surveillance. Whether it's needing money to live, or feeling that fire must be fought with fire in the face of some existential threat, I hear and understand the practical reasons people do such research. However, it is still tiring to always hear some combination of “better that we do it ourselves *the right way*, than someone worse” and “doing this work is justified by the hypotheticals of what happens if we don't,” as trump cards for dismissing the moral qualms of our work. Researchers are spending too much time making such algorithms “more equitable,” when it would have been better to not make the inequitable algorithm in the first place.

Another example can be seen in areas such as “watermarking” [167]. Here, the immoral nature of the work doesn't arise solely from the application, but is woven into the nature of the work. In this case, developing methods to protect wealthy companies that build AI models, enabling them to seek legal retribution. It may be controversial to say that companies don't have a right to the models they build, but I think it is far less controversial to say that such methods are more likely to entrench current technological monopolies than protect any sort of “small-time” AI researcher/hobbyist. We see this exact same scenario play out when small business owners are propped up as the justification for implementing economic policies that overwhelmingly benefit larger corporations.

It's discouraging to hear the rising chorus of researchers, such as Geoff Hinton (who seems *more* moral than most, so I almost feel bad using them as an example), regret part of their life's work because of the harms that it has, or will, cause [99]. However, it's really hard to feel sympathetic when most of these recollections of conscience come after various large paydays from, say, Google.

A.4 Societal Issues

Though I wanted to avoid letting this critique of research open up to complaining about all of societies problems, there's a few social issues that I believe are relevant to academics. My examples are largely in the lens of the current wave of LLMs, however I want to state that I don't think this technology has created *new* problems. Many of the criticisms I make apply just as much to search algorithms, social media algorithms, *etc.* I've also tried my best to put aside criticisms of the "boiling the ocean" variety, that require discussion of huge issues related to ecology, consumption, and economics (though those remain important conversations to have).

My first, and largest, problem is one that I don't hear frequently: LLMs are continuing the trend to homogenise our world. Whether it's a turbocharged version of the "front page effect" that gave rise to search engine optimisation, or the "Writeful" grammar checker that has been integrated into the editor where this very document was written, AI tools are encouraging people to write the same, to get the same answers to questions, and (more nebulously) to think the same. Perhaps this problem wouldn't exist if there were multitudes of possible models, all trained in slightly different ways or on slightly different data. However, the reality is that very few organisations have the resources to produce high-quality LLMs.

I also think that accepting LLMs as some form of ground truth, rather than the black boxes they are, creates *bad* research. Maybe I'm becoming the "old man yells at cloud" stereotype, but are we really willing to acknowledge that "prompt engineering" is a valid area of research? What happens when these models change or fall out of use? What knowledge do we gain by not even attempting to look inside the black box? This line of research seems obviously brittle, and I hope that it goes the way of NFTs, being quickly discredited.

Being in academia, it's important to reckon with how these tools impact not just research, but education as well. Right off the bat, I'll say that I am not taking the ridiculous position that "AI will ruin education because now everyone will cheat!" This recent hysteria feels far too similar to every previous iteration of this phenomenon (computers, the internet, smart phones; contemporaries even decried the printing press). Frankly, the problem is the faculty—not the students. *Some* students have always cheated, and always will. Long before ChatGPT, educators have used cheating as the scapegoat for watering down education, when the real concern has always been the reliability of accreditation. Rather, my concern is with academics who are floating ideas like using ChatGPT to teach programming in a top-down manner, despite this method clearly failing in various other domains, such as language acquisition [154].

This final point is where the elephant in the room is impossible to ignore. The private ownership of LLMs should be enough reason to resist their integration with academia. Given that universities are largely publicly funded, I think it is immensely hypocritical to overlook how

LLMs have become as powerful as they are by pillaging the commons, or outright stealing the digital copyright of others, then turning this around to enrich the owners of the models. The narrative of piracy as stealing from hard workers falls apart when seeing how little the same laws apply to the wealthy.

Appendix B

Other Paradigms of Machine Learning

Supervised learning is not the only manner by which algorithms can “learn.” Here, I will quickly describe two other paradigms: unsupervised learning and reinforcement learning. Other paradigms, such as meta-learning [133] and transfer learning [174] also exist, but are well outside the scope of this work.

B.1 Unsupervised Learning

In contrast to supervised learning, unsupervised learning describes algorithms that perform tasks and identify patterns from *unlabelled* data. Rather than relying heavily on backpropagation, unsupervised learning often tackles problems with other tools, such as the Hopfield [68] or Boltzmann [140] learning rules, contrastive divergence [21], and maximum likelihood [34]. This often comes in the form of trying to optimise some “cost,” or “energy.” The most important of problem domains in unsupervised learning are principal component analysis (PCA) [110] and cluster analysis [173].

In PCA, the goal is to take very high-dimensional data and project onto some lower-dimensional manifold such that we preserve as much information from the original dataset as possible. Reducing the number of features is extremely useful for a number of problems, including exploratory data analysis and accelerating learning (since learning on high dimensional data usually requires larger numbers of parameters). While it is a fairly old method in statistics, obtaining more than two principal components was fairly difficult and time-consuming until the advent of computers.

Clustering algorithms aim to group a set of given data points into “clusters” (read: labels) such that two things are generally true: data points in the same cluster tend to be similar, while

data points from differing clusters are dissimilar. Like PCA, it is a fairly old method and doesn't have its roots in computer science [31]. Formally, it can be seen as an iterative, multi-objective optimisation method, where the objectives arise from some variation of the two outlined above. The iteration comes from applying some labels or parameters, then evaluating how well these labels satisfy the objectives given. An example of clustering is given in Figure B.1.

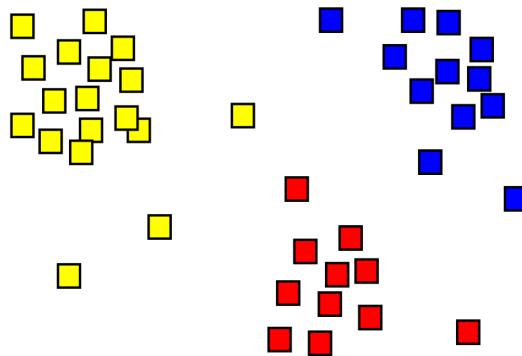


Figure B.1: A possible output of a clustering algorithm. Originally, the data points, represented as squares, would be unlabelled.

Some examples of clustering algorithms are k-means clustering [148], Gaussian mixture models [126], and density-based clustering [81]. In computer science, information theory has been employed to develop “belief propagation” [109], which performs inference on graph models such as Bayesian networks. Genetic algorithms have also been used to perform clustering [94].

Finally, I want to review a specific family of unsupervised learning algorithms, called word2vec [100] since these have some similarities to VSAs, which are the focus of Chapter 3. word2vec is a type of NLP algorithm that takes words from a corpus of text and populates a high-dimensional vector space with vectors associated to those words. The goal is to apply some objective function such that words are close together in the vector space if they are closely related both semantically and syntactically.

B.2 Reinforcement Learning

The second learning paradigm I wish to review is reinforcement learning, or RL. Inspired by psychological principles, RL is concerned with modelling the process of an agent interacting with a given “environment,” such that some reward is maximised. Environments are expressed as a set of possible states. Each state encodes the properties of the environment at a given time step. Agents select “actions” within the environment, usually causing the state of the environment to be updated (sometimes in a probabilistic manner). The environment, or some external interpreter, then gives the agent a reward corresponding to its action. Ultimately, the agent needs to learn an “action policy” that accrues the highest reward. Formally, this can be seen as learning within a “Markov decision process (MDP).” A diagram is given in Figure B.2.

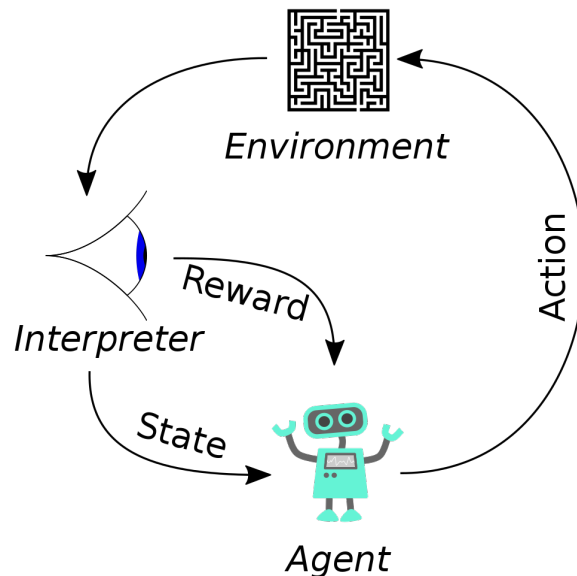


Figure B.2: An illustration of the generic RL setup. Here, an interpreter is distinguished from the environment itself, as it is possible that the details of the environment may be more sophisticated than an agent can perceive.

A strength of reinforcement learning is that agents can associate rewards that are far into the future with actions taken in the present. This means that RL algorithms are often capable of learning long-term strategies for tackling problems. For example, in game-playing, we might choose to provide a reward of one only when an agent reaches a winning state, and give a reward of zero otherwise. Even in longer games, agents can still be capable of approaching a decent playing strategy. As mentioned already, machine learning algorithms—or reinforcement learning

algorithms specifically—have attained great success even in Go [143], with precisely this type of reward system.

However, one consideration in RL is that there is often a trade-off between exploiting actions that already give the agent some reward, versus exploring new possible actions that could lead to even greater rewards. Various exploration strategies have been suggested that usually rely on randomly selecting actions from a given distribution some fraction of the time [156].

Due to the breadth of what can be expressed as an MDP, reinforcement learning has been successfully applied to a wide variety of problems. Some notable examples, in addition to playing Go, are real-time strategy games [51], robotics [112], and self-driving cars [14].

Appendix C

Literature Survey of Category Theory for VSAs

I conducted a brief literature survey to verify there is almost no research on category theory applied to VSAs. Using a search string of

```
(``vector symbolic architecture`` OR ``vector symbolic  
algebra`` OR ``hyperdimensional computing``)  
AND  
``category theory``
```

on Google scholar yielded only twelve results as of 2025-03-06. Of these, one was simply indexing all terms that appeared in IEEE Signal Processing Letters for the year 2014. Another was a job posting for a research position. The remaining ten results were proper academic work that I catalogue below. I list these works roughly in order of increasing relevance.

1. A textbook by Maurer that discussed modern connectionism in cognitive science [95]. VSAs were mentioned as one model for neuroarchitectures. Category theory appeared only in a reference title and was completely absent from the non-paywalled parts of the document¹.
2. An extremely broad survey of mathematics by Raeini that mentions the two topics completely separately [122].

¹I did my best to discern whether category theory was meaningful to the text by reading a provided abstract for every chapter. There was no mention of the topic therein.

3. A paper by Xue *et al.* on distributed microservices discusses category theory at length, but VSAs only appear as a citation when discussing background work [169]. The same thing occurs in a paper by Leemhuis and Özçep about analogical proportions and “betweenness” [91].
4. Three works discussed VSAs at length, but no substantial connection to category theory is made. First, there’s a demonstration by Summers-Stay that VSAs can perform deductive inference [150]. Category theory is mentioned once in the closing paragraph, with the author invoking it to argue that conjunctive normal form (CNF) is a necessary part of their model (I am unsure of the connection, since this is not expanded upon). Second, a work by Widdows and Cohen also discusses inference using VSAs [164]. Last, a work by Gayler and Levy argues against localist representations for analogical mappings in neuronal models [56]. This work argues that the distributed representations of VSAs make them better suited to doing analogical reasoning than their localist counterparts. The sole mention of category theory is an aside to justify the representational power of graphs.
5. One work discusses both in detail, but in separate contexts, rather than connecting the two. This is a PhD thesis by Qiu that is primarily concerned with graph embeddings and disentanglement [118]. Here, VSAs serve as inspiration for methods used to perform graph embedding, while category theory is mentioned in a separate speculative chapter on algorithm transfer. No direct connection is made.
6. Two works by Widdows *et al.* are quite closely related. The first is a broad overview of how various operations used in quantum mechanics and quantum computing can be applied to AI [166]. There, a direct connection is made between VSAs and quantum mechanics, through tensors/binding. A second paper discusses the use of vector representations in more detail [165]. Though category theory is only mentioned briefly in that work, it directly acknowledges how category theory relates quantum models to vector representations of semantics and grammar.

Appendix D

Additional Examples of VSA Algorithms

Here, I provide some more examples of how VSAs can be used to perform computations and solve problems. Credit for these examples goes to Michael Furlong. First, we consider a very simple example of how to construct and deconstruct tuples. Then, I present two more examples that highlight reasons when practitioners may wish to use commutative binding versus non-commutative binding.

D.1 Constructing and Deconstructing Tuples

Sometimes, individuals construct bundles that have more structure than an unordered set. We can, for example, create a tuple with two roles, `first` and `second`. To construct that tuple from two vectors, v_1, v_2 , we would write

$$w = (\text{first} \otimes v_1) \oplus (\text{second} \otimes v_2). \tag{D.1}$$

where binding takes precedence over bundling for order of operations (we've added parentheses simply for clarity).

Then, to recover any element, we would unbind the corresponding role vector:

$$\begin{aligned} v_1 &\sim \text{first} \oslash w \\ v_2 &\sim \text{second} \oslash w \end{aligned}$$

D.2 An Argument for Commutative Binding: Representing Numbers

In MAP and (F)HRR, integers and real-valued data can be represented through iterative applications of operations, not unlike using a successor operation to construct the naturals. However, they may be more easily represented by simply binding a given vector iteratively. For example, using an initial vector, x , one can represent an integer, $n \in \mathbb{Z}^+$, in the (F)HRR algebras by binding x with itself n times, denoted:

$$\phi[n] := x \otimes \dots \otimes x \tag{D.2}$$

$$= \bigotimes_{i=1}^n x, \tag{D.3}$$

where $\phi[n]$ denotes the vector representation of n .

For VSAs where vectors are their own inverse (*e.g.*, BSC, MAP), the successor encoding is required. Integers are represented by applying the braiding operation an integer number of times to the initial vector:

$$\phi[n] := \text{orth}(\text{orth}(\dots x \dots)) \tag{D.4}$$

$$= \text{orth}^n x. \tag{D.5}$$

In commutative algebras, like the first example, it is the case that $\phi[n + m] = \phi[n] \otimes \phi[m] = \phi[m] \otimes \phi[n] = \phi[m + n]$. In non-commutative algebras this relationship may not hold for all elements. In theory, one could exploit superposition and the pseudo-orthogonality to define a bundle that is the equivalence class of all additions, but this would place constraints on the lower bound on the vector dimension to ensure that bundle capacity is not reached. This problem would be further complicated in the case of *fractional* binding [80, 114], where integer binding is naturally extended to the reals through element-wise exponentiation (computed in the Fourier domain in the case of circular convolution binding), although it is not immediately clear what the appropriate translation of fractional binding would be to the non-commutative algebras.

D.3 An Argument Against Commutative Binding: Composing Data Structures

Data structures can be defined in VSAs. I previously discussed slot-filler relationships, like the example of constructing a tuple in Section D.1, but it is also possible to construct data structures that have more detailed ordering, such as lists or trees. A list of items (x_1, \dots, x_n) can be

constructed using bundling and repeated application of a braiding operation, $\rho x := \text{orth}(x)$ as follows:

$$\text{list}(x_1, \dots, x_n) = x_1 \oplus \rho x_2 \oplus \rho \rho x_3 \oplus \dots \oplus \rho^{n-1} x_n \quad (\text{D.6})$$

where ρ^k indicates $k \in \mathbb{Z}^+$ applications of the braiding operation. Individual elements can be selected. This construction is made without relying on the binding operation, although such guarding is possible in VSAs with commutative binding operations. This is accomplished by binding with a “guard vector”, h , that has been iteratively bound with itself:

$$\text{list}(x_1, \dots, x_n) = x_1 \oplus h \otimes x_2 \oplus \dots \oplus h^{n-1} \otimes x_{n-1}, \quad (\text{D.7})$$

where h^k is shorthand for $\bigotimes_{i=1}^k h$. Problems with this approach become more apparent when constructing, for example, a binary tree. In this case, one would require two braiding operations, ρ_L and ρ_R , which are typically implemented as two different permutation matrices. Storing data elements in the leaves of a tree with depth 2 would be implemented:

$$\text{tree}(x_1, x_2, x_3, x_4) = \rho_L \rho_L x_1 \oplus \rho_L \rho_R x_2 \oplus \rho_R \rho_L x_3 \oplus \rho_R \rho_R x_4. \quad (\text{D.8})$$

To construct a tree using guard vectors we would similarly define h_L and h_R , and construct the tree:

$$\text{tree}(x_1, x_2, x_3, x_4) = h_L \otimes h_L \otimes x_1 \oplus h_L \otimes h_R \otimes x_2 \oplus h_R \otimes h_L \otimes x_3 \oplus h_R \otimes h_R \otimes x_4. \quad (\text{D.9})$$

But if \otimes commutes, then $h_L \otimes h_R = h_R \otimes h_L$, making it impossible to disambiguate the two middle leaves of the tree. Gayler [53] proposed a braiding operator for precisely this purpose. Plate suggested that multiple non-commutative, similarity preserving binding operators may exist, noting systematically permuting one of the arguments of a commutative binding operation as one example [117, §3.6.7]. Gosmann and Eliasmith [59] propose the VTB as one non-commutative binding operation, simplifying the number of operations required in the VSA to impose structure.

Appendix E

Verifying that Lenses are a Profunctor by Type-checking

Recall that `(Lens fwd bwd)` has the following Types:

```
fwd :: a -> b
bwd :: (a, db) -> da
```

and that our `dimap` permits functions `f :: a' -> a` and `g :: da -> da'`. Then, we can define how `dimap` transforms `(Lens fwd bwd)` into a new `(Lens fwd' bwd')` by considering

```
dimap f g (Lens fwd bwd) = (Lens fwd' bwd')
```

where we ultimately want `fwd' :: a' -> b` and `bwd' :: (a', db) -> da'`.

It's fairly easy to find an expression that type-matches for `fwd'`. Since we have `f :: a' -> a` and `fwd :: a -> b`, we can simply compose the two to get

```
fwd' = fwd . f :: (a' -> a) . (a -> b) ≡ (a' -> b).
```

Finding an expression for `bwd'` is slightly trickier, but still pretty easy when using Types to direct us. `bwd'` already takes inputs `(a', db)`, and we'll have access to both of these post-dimapping. Now let us look at our expression

```
bwd' (a', db) = g (bwd (f a', db)).
```

Since `bwd` requires an input of `(a, db)`, we need to apply `f` to `a'` to get an `a`. However, `bwd` only spits out a `da`—not the required `da'`. Luckily we can now use `g` to obtain the final result, namely that the output of `bwd'` is `da'`.