

Dioptics: a Common Generalization of Open Games and Gradient-Based Learners

David “davidad” Dalrymple¹

¹ Protocol Labs, Palo Alto, USA

Correspondence to: davidad(at)alum.mit.edu

Version accepted for presentation at SYCO 5, 5th September 2019

Abstract

Compositional semantics have been shown for machine-learning algorithms [FST18] and open games [Hed18]; at SYCO 1, remarks were made noting the high degree of overlap in character and analogy between the constructions, and that there is known to be a monoidal embedding from the category of learners to the category of games, but it remained unclear exactly what kind of structure they both are. This is work in progress toward showing that both categories embed faithfully and bijectively-on-objects into instances of a pattern we call *categories of dioptics*, whose name and definition both build heavily on [Ril18]. Using a generalization of the reverse-mode automatic differentiation functor of [Ell18] to arbitrary diffeological spaces with trivializable tangent bundles, we also construct a *category of gradient-based learners* which generalizes gradient-based learning beyond Euclidean parameter spaces. We aim to show that this category embeds naturally into the category of learners (with a choice of update rule and loss function), and that composing this embedding with reverse-mode automatic differentiation (and the inclusion of Euclidean spaces into trivializable diffeological spaces) recovers the backpropagation functor L of [FST18].

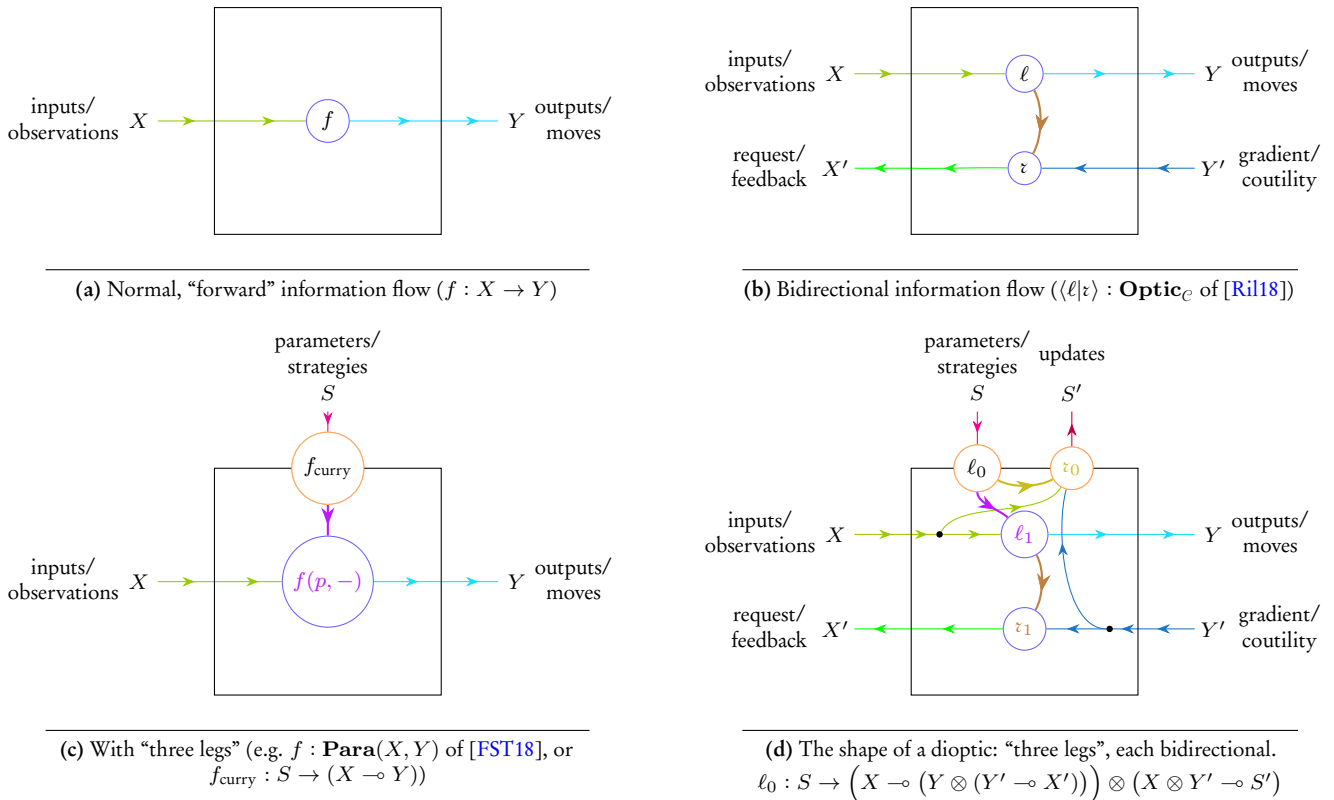


Figure 1. The ‘three-legged’ shape of components in deep learning or open games.

1 INTRODUCTION

Deep learning and open games both have a curious information-flow structure: at run-time, components accept “inputs” from other components and send “outputs” to other components, which eventually become “actions” (or “moves,” or classification “labels”); in response to the actions, reward signals (or “loss” or “utility”) are fed back into the system, and propagated backwards as feedback (or “couthility” or “gradient”) signals; yet underlying all these forward-and-backward interactions is *another* forward-and-backward interaction in a more distant-seeming frame (“training” or “equilibrium selection”), where the behavior of all components is controlled behind the scenes by a set of “parameters” (or “strategies”), and depending on their performance, some updates to these are iteratively calculated. (See fig. 1 for a graphical representation of how these information flows are built up.)

Although it may seem that this three-legged shape would lend itself to an awkward calculus of composition, in fact there is a natural way in which we can compose these objects, either vertically or horizontally: by applying the appropriate operation inside the “run-time” frame, simply “holding onto the loose threads” for parameters and updates, and collecting them in a monoidal product (regardless of whether the composition is sequential or parallel).

If we ignore the “third leg”, as in fig. 1b, then the curiosity of bidirectional flow—and the appropriate composition laws—are well described by the structure of a *category of optics* [Ril18]. But the role of parameters and updates has generally been handled by resorting to the external language of set theory. As such, the formal sense in which open games and deep learners manifest instances of the same kind of structure has been somewhat obscured.

In our view, informally, the forward flows of these three-legged objects can be thought of as maps from the parameter space S into the space of optics from inputs X to outputs Y , while the backward flow is from the context of the lens $X \rightsquigarrow Y$ to some “backward version” of S . In particular, the “backward version” of a vector space (as parameter spaces in deep learning nearly always are) is its dual space (the proper type of gradient vectors). We make use of the reverse-mode automatic-differentiation functor of [Ell18], whose codomain can be considered a category of optics of diffeological spaces.

2 BACKGROUND

In this section we collect the most relevant definitions from the four primary lines of work we build upon, for easy reference and with some suggestive uniformity of notational convention.

2.0 Notation

In this paper we use the following somewhat uncommon notations:

Definitions We will often use the format

$$\underbrace{\text{eval}}_{\text{name}} \underbrace{X, Y}_{\text{parameters}} : \underbrace{((X \multimap Y) \otimes X) \rightarrow Y}_{\text{type}} := \underbrace{\langle f, x \rangle}_{\text{bindings}} \mapsto \underbrace{f(x)}_{\text{expression}}$$

which should be familiar to users of some proof assistants (e.g. Coq, Lean), but perhaps not to others. The unfolding into a more broadly familiar format would be

$$\begin{aligned} \text{eval}_{X, Y} &: ((X \multimap Y) \otimes X) \rightarrow Y \\ \text{eval}_{X, Y} \langle f, x \rangle &= f(x) \end{aligned}$$

Composition To avoid ambiguity between Leibniz-order and diagrammatic-order composition, following a suggestion of Brendan Fong, we will write $f \circledast g$ for diagrammatic-order composition, i.e. $(f \circledast g)(x) \equiv g(f(x))$. In a slight abuse¹, we will sometimes use this operator for application as well where convenient, so that $x \circledast f \circledast g$ also denotes $g(f(x))$.

2.1 Optics [Ril18; dPai91]

A *concrete lens* from $X \rightsquigarrow Y$ is a pair of maps **get** : $X \rightarrow Y$ and **put** : $X \times Y \rightarrow X$. Lenses of this form are often assumed to obey certain laws, but here we are only concerned with what Cezar Ionescu termed “*outlaw lenses*”: roughly, a concrete outlaw lens from $(X, X') \rightsquigarrow (Y, Y')$ is a pair of maps **get** : $X \rightarrow Y$ and **put** : $X \times Y' \rightarrow X'$. In more explicitly categorical terms:

Definition 2.1.1. *Given a monoidal category \mathcal{C} , we define a function on objects*

$$\text{Lens}_{\mathcal{C}} : (|\mathcal{C}| \times |\mathcal{C}|) \times (|\mathcal{C}| \times |\mathcal{C}|) \rightarrow \mathbf{Set} := \left((X, X'), (Y, Y') \right) \mapsto \underbrace{\mathcal{C}(X, Y)}_{\text{get}} \times \underbrace{\mathcal{C}(X \otimes Y', X')}_{\text{put}}$$

Under these weak conditions on \mathcal{C} , $\text{Lens}_{\mathcal{C}}$ need not form a category; the identity lens from $(X, X') \rightsquigarrow (X, X')$ requires an element of $\mathcal{C}(X \otimes X', X')$, which is not guaranteed to exist in a non-cartesian monoidal category. There are three ways forward from here: (1) we can use coends to define *optics*, assuming only that \mathcal{C} is symmetric monoidal, (2) with *linear lenses*, we can avoid explicit manipulation of coends but require \mathcal{C} to be symmetric monoidal closed (i.e. $\forall X, - \otimes X$ has a right adjoint $X \multimap -$), or (3) we can restrict to the cartesian case (or even stronger conditions).

If \mathcal{C} is merely symmetric monoidal, we can define a version of lenses (termed *optics*) by using a coend, which is a certain kind of colimit that roughly corresponds to an existential type, or a dependent sum quotiented by observational equivalence (see [Lor17] for more on ends and coends).

¹This can perhaps be justified by considering “elements” or terms $x : X$ as actually morphisms from the ambient context of bindings $\Gamma \rightarrow X$ (i.e. from the monoidal unit $1 \rightarrow X$ if the context is empty), but this point does not seem worth belaboring.

Definition 2.1.2 (compare [Ril18, definition 2.0.1]). *Given a symmetric monoidal category \mathcal{C} , we define*

$$\mathbf{Optic}_{\mathcal{C}} : (\mathcal{C} \times \mathcal{C}^{\text{op}})^{\text{op}} \times (\mathcal{C} \times \mathcal{C}^{\text{op}}) \rightarrow \mathbf{Set} := \left((X, X'), (Y, Y') \right) \mapsto \int^{M:\mathcal{C}} \mathcal{C}(X, M \otimes Y) \times \mathcal{C}(M \otimes Y', X')$$

Theorem 2.1.3 ([Ril18, proposition 2.0.3 + theorem 2.0.12]). *$\mathbf{Optic}_{\mathcal{C}}$ forms a symmetric monoidal category with objects of $\mathcal{C} \times \mathcal{C}^{\text{op}}$.*

Proposition 2.1.4. *If \mathcal{C} is cartesian monoidal, then $\mathbf{Lens}_{\mathcal{C}}$ forms (the hom-functor of) a monoidal category, and $\mathbf{Lens}_{\mathcal{C}} \cong \mathbf{Optic}_{\mathcal{C}}$.*

Definition 2.1.5 (compare [Ril18, section 4.8]). *Given a symmetric monoidal closed category \mathcal{C} , we can define $\mathbf{LinOptic}_{\mathcal{C}}$ as*

$$\mathbf{LinOptic}_{\mathcal{C}} : (\mathcal{C} \times \mathcal{C}^{\text{op}})^{\text{op}} \times (\mathcal{C} \times \mathcal{C}^{\text{op}}) \rightarrow \mathbf{Set} := \left((X, X'), (Y, Y') \right) \mapsto \mathcal{C} \left(X, Y \otimes (Y' \multimap X') \right)$$

Proposition 2.1.6. *For any symmetric monoidal closed category \mathcal{C} , $\mathbf{LinOptic}_{\mathcal{C}} \cong \mathbf{Optic}_{\mathcal{C}}$.*

Definition 2.1.7. *If \mathcal{C} is symmetric monoidal closed, we define the functor*

$$\mathbf{InternalLinOptic}_{\mathcal{C}} : (\mathcal{C} \times \mathcal{C}^{\text{op}})^{\text{op}} \times (\mathcal{C} \times \mathcal{C}^{\text{op}}) \rightarrow \mathcal{C} := \left((X, X'), (Y, Y') \right) \mapsto X \multimap (Y \otimes (Y' \multimap X'))$$

If we accept more conditions on \mathcal{C} , we can go even further:

Theorem 2.1.8 (compare [dPai91, section 1.2] and [Ril18, remark 4.1.2]). *If \mathcal{C} is locally cartesian closed and has a terminal object, then $\mathbf{Optic}_{\mathcal{C}}$ forms a symmetric monoidal closed category, with internal hom defined as*

$$(X, X') \multimap_{\mathbf{Optic}_{\mathcal{C}}} (Y, Y') := \left(\mathbf{InternalLinOptic}_{\mathcal{C}} \left((X, X'), (Y, Y') \right), X \times Y' \right)$$

This fact will be very useful for us, so for the rest of this paper we will assume \mathcal{C} is locally cartesian closed and has a terminal object, unless otherwise stated. We will also use the notation $\rightsquigarrow_{\mathcal{C}}$ for $\multimap_{\mathbf{Optic}_{\mathcal{C}}}$.

2.2 Open games [Hed18; BHW18]

Open games [Hed18] formalize composition in game theory, where the “input” of one game emerges from the “output” of another.

Definition 2.2.1 ([Hed18, definition 3]). *Let X, X', Y, Y' be sets. An open game $\mathcal{G} : (X, X') \rightarrow (Y, Y')$ is defined by:*

- $S_{\mathcal{G}} : \mathbf{Set}$, the strategy profiles of \mathcal{G}
- $P_{\mathcal{G}} : S_{\mathcal{G}} \times X \rightarrow Y$, the play function of \mathcal{G}
- $C_{\mathcal{G}} : S_{\mathcal{G}} \times X \times Y' \rightarrow X'$, the coplay function of \mathcal{G}
- $B_{\mathcal{G}} : X \times (Y \rightarrow Y') \rightarrow (S_{\mathcal{G}} \times S_{\mathcal{G}}) \rightarrow \mathbf{2}$, the best response function of \mathcal{G}

In the same paper, the connection to lenses is shown:

Proposition 2.2.2 ([Hed18, lemma 1]). *An open game $\mathcal{G} : (X, X') \rightarrow (Y, Y')$ is exactly*

- a family of lenses, i.e. a set $S_{\mathcal{G}}$ and for each $s : S_{\mathcal{G}}$ a lens $\mathcal{G}_s : (X, X') \rightsquigarrow_{\mathbf{Set}} (Y, Y')$
- $B_{\mathcal{G}} : X \times (Y \rightarrow Y') \rightarrow (S_{\mathcal{G}} \times S_{\mathcal{G}}) \rightarrow \mathbf{2}$, just as in definition 2.2.1

Open games are also shown to form a category:

Theorem 2.2.3 ([Hed18, proposition 3, theorem 1]). *There is a symmetric monoidal category \mathbf{Game} with objects of $\mathbf{Set} \times \mathbf{Set}^{\text{op}}$ and whose morphisms are equivalence classes of open games.*

An alternative (presumably not equivalent) definition of open games defines an *equilibrium predicate* rather than a best-response relation:

Definition 2.2.4 ([BHW18, definition 1]). *Let X, X', Y, Y' be sets. An open game with equilibria $\mathcal{G} : (X, X') \rightarrow (Y, Y')$ is*

- a family of lenses, i.e. a set $S_{\mathcal{G}}$ and for each $s : S_{\mathcal{G}}$ a lens $\mathcal{G}_s : (X, X') \rightsquigarrow_{\mathbf{Set}} (Y, Y')$
- $E_{\mathcal{G}} : X \times (Y \rightarrow Y') \rightarrow S_{\mathcal{G}} \rightarrow \mathbf{2}$, the equilibrium predicate of \mathcal{G}

This definition also induces a symmetric monoidal category:

Theorem 2.2.5 (corollary to [BHW18, theorem 1]). *There is a symmetric monoidal category \mathbf{Game}_E with objects of $\mathbf{Set} \times \mathbf{Set}^{\text{op}}$ and whose morphisms are equivalence classes of open games with equilibria.*

Remark 2.2.6. In the intended game-theoretic interpretation, the equilibrium predicate $E_{\mathcal{G}}(x, k)(s) := B_{\mathcal{G}}(x, k)(s, s)$; this definition induces a symmetric monoidal functor $\mathbf{Game} \rightarrow \mathbf{Game}_E$.

2.3 Backpropagation as a functor [FST18]

Here we reproduce (and, where helpful, renotate) the definitions for the category **Para** of parametrised functions, the category **Learn** of learners, and the backpropagation functor $L_{e,\eta} : \mathbf{Para} \rightarrow \mathbf{Learn}$.

Definition 2.3.1 ([FST18, definition 3.1]). *Let **Para** be the strict symmetric monoidal category whose*

- objects X, Y are Euclidean spaces $\mathbb{R}^m, \mathbb{R}^n$
- morphisms $X \rightarrow Y$ are
 - pairs (S, f) , where
 - * $S = \mathbb{R}^k$ is an arbitrary Euclidean space (intuitively, of parameters upon which f depends) and
 - * $f : S \times X \rightarrow Y$ is a differentiable function,
 - quotiented by equivalence, where
 - * an equivalence $(S, f) \sim (S', f')$ is a differentiable isomorphism $\alpha : S \cong S'$ s.t. $f(-, \cdot) = f'(\alpha(-), \cdot)$
- sequential composition of $f : S \times X \rightarrow Y$ and $g : T \times Y \rightarrow Z$ is given by

$$(f \circ g) ((s, t), x) := g(t, f(s, x))$$

- parallel composition of $f : S \times X \rightarrow W$ and $g : T \times Y \rightarrow Z$ is given by

$$(f \otimes g) ((s, t), (x, y)) := (f(s, x), g(t, y))$$

- symmetry $\sigma_{X,Y} : \mathbb{R}^0 \times (X \times Y) \rightarrow (Y \times X) := (*, x, y) \mapsto (y, x)$.
- identity $\text{id}_X : \mathbb{R}^0 \times X \rightarrow X := (*, x) \mapsto x$.

Definition 2.3.2 ([FST18, definition 2.1]). *Let X, Y be sets. A learner ℓ from $X \rightarrow Y$ is defined by:*

- $S_\ell : \mathbf{Set}$, the parameter space of ℓ
- $I_\ell : S_\ell \times X \rightarrow Y$, the implementation function of ℓ
- $U_\ell : S_\ell \times X \times Y \rightarrow S_\ell$, the update function of ℓ
- $r_\ell : S_\ell \times X \times Y \rightarrow X$, the request function of ℓ

It is easy to see how this definition can be rephrased in terms of lenses, analogously to proposition 2.2.2:

Proposition 2.3.3. *A learner $\ell : X \rightarrow Y$ is exactly*

- a family of lenses, i.e. a set S_ℓ and for each $s : S_\ell$ a lens $\ell_s : (X, X) \rightsquigarrow_{\mathbf{Set}} (Y, Y)$
- $U_\ell : S_\ell \times X \times Y \rightarrow S_\ell$, the update function of ℓ , as in definition 2.3.2

Proposition 2.3.4 ([FST18, proposition 2.4]). *There is a symmetric monoidal category **Learn** whose objects are sets and whose morphisms are equivalence classes of learners.*

Theorem 2.3.5 ([FST18, theorem 3.2]). *Given a positive number $\eta : \mathbb{R}$ (the step size) and a differentiable function $e(x, y) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ (the loss function) such that $\frac{\partial e}{\partial x}(z, -) : \mathbb{R} \rightarrow \mathbb{R}$ is invertible $\forall z : \mathbb{R}$, we can define a faithful, injective-on-objects, symmetric monoidal functor $L_{e,\eta} : \mathbf{Para} \rightarrow \mathbf{Learn}$ that sends each parametrised function $f : S \times X \rightarrow Y$ to the learner (S, f, U_f, r_f) defined by*

$$\begin{aligned} U_f(s, x, y) &:= s - \eta \nabla_s \sum_j e(f(s, x)_j, y_j) \\ r_f(s, x, y) &:= f_x \left(\nabla_x \sum_j e(f(s, x)_j, y_j) \right) \end{aligned}$$

where f_x is componentwise application of the inverse to $\frac{\partial e}{\partial x}(x_i, -)$ for each i .

Proposition 2.3.6 ([FST18, proposition 5.1]). *The symmetric monoidal category **FinVect** of linear maps between Euclidean spaces sits inside **Para** by considering each linear maps as parametrised by the trivial parameter space \mathbb{R}^0 , we have an inclusion $\mathbf{FinVect} \hookrightarrow \mathbf{Para}$. This provides a Hopf monoid structure on every object of **Para**; for a given choice of e and η , these Hopf monoids can be transported by $L_{e,\eta}$ into **Learn**.*

Proposition 2.3.7 ([FST18, example 5.4]). *The morphisms of **Para** are generated by*

- differentiable functions $f : X \rightarrow Y$, considered as trivially parametrised functions $\mathbb{R}^0 \times X \rightarrow Y$, and
- parameter spaces $S = \mathbb{R}^n$, considered as parametrised constants $S \times \mathbb{R}^0 \rightarrow S$

The structures of propositions 2.3.6 and 2.3.7 can be used to define a string-diagram calculus for neural network architectures in general, including weight-tying; see [FST18, sections 5–6] for details.

2.4 Reverse-mode automatic differentiation as a functor [Ell18]

From the perspective of machine learning, reverse-mode automatic differentiation is often considered just a fancy name for backpropagation, so one might wonder why there should be any difference. However, reverse-mode automatic differentiation is more general [BPRS18]: for example, it can be used to compute Jacobians and Hessians for second-order methods, not just gradients.

2.4.1 Differentiation in general

We begin with the single-variable derivative of $f : \mathbb{R} \rightarrow \mathbb{R}$,

$$f'(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon}$$

then take the first generalization, from $\mathbf{End}_{\mathbf{Smooth}}(\mathbb{R})$ (the one-object category of smooth endofunctions on \mathbb{R}) to \mathbf{Euc} (the category of Euclidean spaces and smooth maps) by defining the *Fréchet derivative* of $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ as a function $f' : \mathbb{R}^m \rightarrow (\mathbb{R}^m \multimap \mathbb{R}^n)$ such that

$$\lim_{\varepsilon \rightarrow 0} \frac{\|f(x + \varepsilon) - [f(x) + f'(x)(\varepsilon)]\|}{\|\varepsilon\|} = 0.$$

In [Ell18], the compelling phrase *local linear approximation* is used to describe this idea of what a derivative represents, and [Spi65, chapter 2] is cited for the uniqueness result.

2.4.2 Automatic differentiation

Rewriting f' as the application of an operator D to f , we have a first definition of derivative

Definition 2.4.1 (compare [Ell18, page 3]). *For $X, Y : \mathbf{Euc}$, $f : X \rightarrow Y$, we define*

$$Df : \left(\overset{x}{X} \rightarrow \overset{f'(x) := g}{(X \multimap Y)} \right) := x \mapsto \text{the unique linear } g \text{ s.t. } \lim_{\varepsilon \rightarrow 0} \frac{\|f(x + \varepsilon) - [f(x) + g(\varepsilon)]\|}{\|\varepsilon\|} = 0$$

Theorem 2.4.2 (chain rule, [Spi65, theorem 2-2]).

$$D(f \circ g)(x) = Df(x) \circ Dg(f(x))$$

However, the problem is that this composition rule is not functorial: $D(f \circ g)$ is not determined only by Df and Dg , because it also depends on f . Therefore we use the following alternative definition.

Definition 2.4.3 (compare [Ell18, page 4]). *For $X, Y : \mathbf{Euc}$, $f : X \rightarrow Y$, we define*

$$D^+f : X \rightarrow (Y \times (X \multimap Y)) := \langle f, Df \rangle \quad \left(\text{or equivalently } x \mapsto \langle f(x), Df(x) \rangle \right)$$

Proposition 2.4.4 ([Ell18, corollaries 1.1–3.1]). *D^+ is a symmetric monoidal functor $\mathbf{Euc} \rightarrow \mathbf{CartDeriv}$, where $\mathbf{CartDeriv}$ is a symmetric monoidal category with objects of \mathbf{Euc} and morphisms $X \rightarrow Y$ of the form $X \rightarrow_{\mathbf{Smooth}} (Y \times (X \multimap Y))$.*

2.4.3 Reverse-mode automatic differentiation

The insight of reverse-mode automatic differentiation is that if we only care to compute derivatives of some ultimate “answer” $z : Z := f_n(f_{n-1}(\dots f_0(x)\dots))$ (e.g. a utility/loss function) with respect to each variable, then at each step of the computation we can represent derivatives by their maps into Z , accepting a “continuation” k for the rest of the derivative computation and precomposing the current step’s $Df(x)$:

Definition 2.4.5 (compare [Ell18, section 12]). *For $X, Y, Z : \mathbf{Euc}$ and $f : X \rightarrow Y$, we define the reverse-mode derivative of f towards Z as*

$$D^{\Leftarrow}f : X \rightarrow \left(Y \times ((Y \multimap Z) \multimap (X \multimap Z)) \right) := x \mapsto \langle f(x), k \mapsto Df(x) \circ k \rangle$$

3 DIOPTICS

Here we present our central proposal, the notion of a *category of dioptics*. At its core, a dioptic is like a “nested optic” in which **get** (or ℓ) of the “outer” optic yields an “inner” optic $(X, X') \rightsquigarrow (Y, Y')$ for every strategy/parametrisation in S , and **put** (or τ) of the outer optic yields an “update” (or piece of feedback) in S' for every context $X \times Y'$ of the inner optic. (Refer back to fig. 1d, or the elaborated version fig. 2 below, for visual intuition about this nesting.)

The additional intricacies are:

- The pairs (X, X') and (Y, Y') of objects of \mathcal{C} should be generated “automatically” by some functor F from single objects \check{X} of a potentially different category \mathcal{T} (the *target* category).
- The pair (S, S') of objects of \mathcal{C} should similarly be generated “automatically”, but a different notion of “backwards version” may be applicable on the level of parameter updates², so we use a different functor G , which may also have a different source category \mathcal{S} .
- Finally, the object \check{S} is bound by a coend $\int^{\check{S}:\mathcal{S}}$. This internalizes the usual quotienting-by-equivalence which is necessary to obtain associative composition of dioptic-like morphisms.

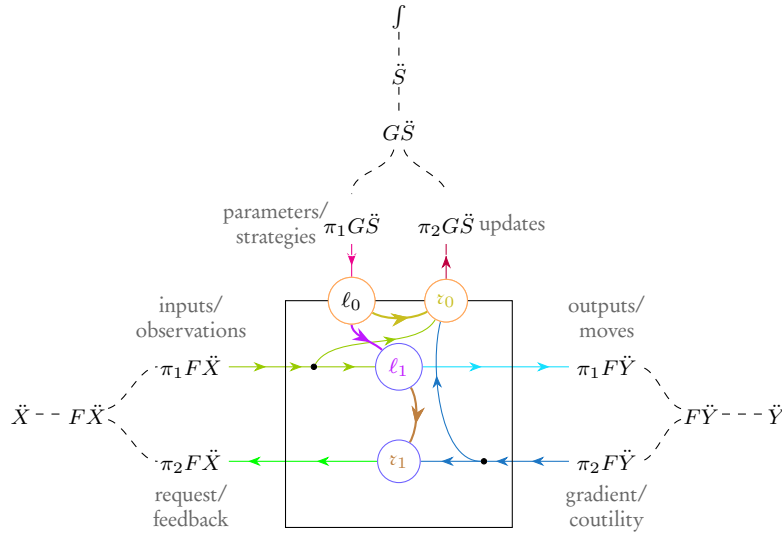


Figure 2. Dioptics with “automatically generated” forwards/backwards pairs of objects from functor F . *Note:* the “backwards version” of the entire object-level lens $(X, X') \rightsquigarrow (Y, Y')$ is $X \otimes Y'$ because of theorem 2.1.8.

Definition 3.0.1. *Given a cartesian closed and locally cartesian closed category \mathcal{C} , symmetric monoidal categories \mathcal{S}, \mathcal{T} , and symmetric oplax monoidal functors $F : \mathcal{T} \rightarrow \mathbf{Optic}_{\mathcal{C}}$ and $G : \mathcal{S} \rightarrow \mathbf{Optic}_{\mathcal{C}}$ (note that a functor into $\mathcal{C} \times \mathcal{C}^{\text{op}}$ can be considered a functor into $\mathbf{Optic}_{\mathcal{C}}$ via a canonical embedding), we define*

$$\mathbf{Dioptic}_{F,G} : \mathcal{T}^{\text{op}} \times \mathcal{T} \rightarrow \mathbf{Set} := (\check{X}, \check{Y}) \mapsto \int^{\check{S}:\mathcal{S}} \mathbf{Optic}_{\mathcal{C}}(G\check{S}, F\check{X} \rightsquigarrow_{\mathcal{C}} F\check{Y})$$

Conjecture 3.0.2. *$\mathbf{Dioptic}_{F,G}$ forms a category with objects of \mathcal{T} . If F is a symmetric monoidal functor, $\mathbf{Dioptic}_{F,G}$ is a symmetric monoidal category.*

Both sequential and parallel composition in $\mathbf{Dioptic}_{F,G}$ involve taking monoidal products of the objects $\check{S}, \check{T} : \mathcal{S}$ that are quotiented out by the coend, then relying on the internal composition rules of the symmetric monoidal closed category $\mathbf{Optic}_{\mathcal{C}}$.

Note. The use of a coend is inspired by [Ril18], but may not be the best abstraction in this context. One alternative we are exploring is to replace the coend by a normal coproduct and show that the resulting notion of dioptics forms a symmetric monoidal bicategory (with 2-cells representing reparametrisations). Another possibility may be to define a double category of dioptics, similar to the *Dbl* construction of [SK19], but where the vertical dimension represents parameter-passing rather than state-passing. However, it is not immediately clear how to collapse such a double category into a bicategory in the desired way: that is, with 1-morphisms being the double category’s 2-cells, and 2-morphisms being 2-cells that transform one 2-cell to another by vertical precomposition. Finding an elegant way to manage such reparametrisations is currently the main bottleneck in this work.

²For example, in open games, instead of returning an updated strategy profile on the backward pass, we return a Boolean result about whether the strategy profile is an equilibrium.

3.1 Learners as dioptics

We conjecture that the category **Learn** is equivalent to a category of dioptics.

Definition 3.1.1. *Given a symmetric monoidal category \mathcal{C} , we define the following “bivariant diagonal functor”:*

$$\Delta_{\mathcal{C}}^{\overleftarrow{=}} : \text{Core}(\mathcal{C}) \rightarrow \mathcal{C} \times \mathcal{C}^{\text{op}} := X \mapsto (X, X)$$

Conjecture 3.1.2.

$$\mathbf{Learn} \cong \mathbf{Dioptic}_{\Delta_{\text{Set}}^{\overleftarrow{=}}, \Delta_{\text{Set}}^{\overleftarrow{=}}}$$

Para also seems equivalent to a category of dioptics, but where the “backward” objects are trivial (the monoidal unit).

Definition 3.1.3. *Given a monoidal category \mathcal{C} , we define:*

$$\text{Fwd}_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C} \times \mathcal{C}^{\text{op}} := X \mapsto (X, 1)$$

Conjecture 3.1.4.

$$\mathbf{Para} \cong \mathbf{Dioptic}_{\text{Fwd}_{\mathbf{Euc}}, \text{Fwd}_{\mathbf{Euc}}}$$

3.2 Open games as dioptics

We conjecture that the categories **Game** and **Game_E** each have a faithful³ identity-on-objects functor into a category of dioptics.

Definition 3.2.1. *We define the following auxiliary symmetric oplax monoidal functors:*

$$E^+ : \mathbf{Set} \rightarrow \mathbf{Set} \times \mathbf{Set}^{\text{op}} := S \mapsto (S, \mathbf{2}) \quad C^+ : \mathbf{Set} \times \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Set} \times \mathbf{Set}^{\text{op}} := (X, X') \mapsto (X, X \multimap X')$$

$$B^+ : \mathbf{Set} \rightarrow \mathbf{Set} \times \mathbf{Set}^{\text{op}} := E^+ \ ; \ C^+ = S \mapsto (S, S \multimap \mathbf{2})$$

The oplaxator of E^+ is defined using conjunction:

$$E^+ \cdot \Delta_{S,T} : E^+(S \times T) \xrightarrow{(S \times T, \mathbf{2})} \mathbf{Set} \times \mathbf{Set}^{\text{op}} \xrightarrow{(S \times T, \mathbf{2} \times \mathbf{2})} E^+S \otimes E^+T := ((s, t) \mapsto (s, t), (a \wedge b) \leftarrow (a, b))$$

Conjecture 3.2.2.

$$\mathbf{Game} \hookrightarrow \mathbf{Dioptic}_{C^+, B^+}$$

Proof sketch:

$$\begin{aligned} & \mathbf{Dioptic}_{C^+, B^+} \left((X, X'), (Y, Y') \right) \\ &= \int^{S: \mathbf{Set}} \mathbf{Optic}_{\mathbf{Set}} \left(B^+ \check{S}, C^+(X, X') \rightsquigarrow_{\mathbf{Set}} C^+(Y, Y') \right) \\ &= \int^{S: \mathbf{Set}} \mathbf{Optic}_{\mathbf{Set}} \left((S, S \rightarrow \mathbf{2}), (X, X \rightarrow X') \rightsquigarrow_{\mathbf{Set}} (Y, Y \rightarrow Y') \right) \\ &\cong \int^{S: \mathbf{Set}} \mathbf{Optic}_{\mathbf{Set}} \left((S, S \rightarrow \mathbf{2}), \left(X \rightarrow \left(Y \times \left((Y \rightarrow Y') \rightarrow (X \rightarrow X') \right) \right), X \times (Y \rightarrow Y') \right) \right) \\ &\cong \int^{S: \mathbf{Set}} \mathbf{Set} \left(S, X \rightarrow \left(Y \times \left((Y \rightarrow Y') \rightarrow (X \rightarrow X') \right) \right) \right) \times \mathbf{Set} \left(S \times X \times (Y \rightarrow Y'), (S \rightarrow \mathbf{2}) \right) \\ &\cong \int^{S: \mathbf{Set}} \mathbf{Set} (S \times X, Y) \times \mathbf{Set} \left(S \times X \times (Y \rightarrow Y'), (X \rightarrow X') \right) \times \mathbf{Set} \left(X \times (Y \rightarrow Y'), (S \times S \rightarrow \mathbf{2}) \right) \\ &\leftarrow \coprod_{S: \mathbf{Set}} \mathbf{Set} (S \times X, Y) \times \mathbf{Set} \left(S \times X \times (Y \rightarrow Y'), (X \rightarrow X') \right) \times \mathbf{Set} \left(X \times (Y \rightarrow Y'), (S \times S \rightarrow \mathbf{2}) \right) \\ &\stackrel{\phi}{\leftarrow} \coprod_{S: \mathbf{Set}} \overbrace{\mathbf{Set} (S \times X, Y)}^{\text{play function } P} \times \overbrace{\mathbf{Set} (S \times X \times Y', X')}^{\text{coplay function } C} \times \overbrace{\mathbf{Set} (X \times (Y \rightarrow Y'), (S \times S) \rightarrow \mathbf{2})}^{\text{best-response function } B} \\ &\leftarrow \mathbf{Game} \left((X, X'), (Y, Y') \right) \end{aligned}$$

³Why not *fully* faithful? The dioptics construction doesn't place enough constraints on the backward flow of information: classical game theory implicitly assumes players have magical true knowledge of strategies' hypothetical consequences, and **Game** reflects this, whereas the dioptical formulation of games models consequences as being received “on the grapevine” from future players—who might not be truthful about hypotheticals! “Truthful” players (or player-ensembles) are the morphisms in the wide subcategory that is equivalent to **Game**.

where

$$\phi := (S, P, C, B) \mapsto \left(S, P, (s, \mathfrak{X}, k) \mapsto x \mapsto C(s, x, k(P(s, x))), B \right)$$

Faithfulness can be shown using a retraction of ϕ ,

$$\phi^\leftarrow := (S, P, K, B) \mapsto \left(S, P, (s, x, y') \mapsto K(s, x, (y \mapsto y'))(x), B \right)$$

Note. Because C^+ is not strong monoidal, $\mathbf{Diop}^{\mathbf{C}^+, B^+}$ fails to be a monoidal category (so of course the embedding functor is not a monoidal functor). Although C^+ is bilax monoidal and even Frobenius monoidal, this is not enough; in particular, in $\mathbf{Diop}^{\mathbf{C}^+, B^+}$, $\text{id}_X \otimes \text{id}_Y \not\cong \text{id}_{X \times Y}$. This is essentially because counterfactuals of the type $X \times Y \rightarrow X' \times Y'$ (“joint” counterfactuals) cannot be *reversibly* decomposed into a pair of separate counterfactuals with types $X \rightarrow X', Y \rightarrow Y'$. Future work is to explore richer notions of counterfactuals, e.g. something along the lines of $(X \rightarrow \Omega) \times (\Omega \rightarrow X')$ instead of $(X \rightarrow X')$, in hopes of finding one which *could* be reversibly decomposed (by some kind of join operation on Ω).

Conjecture 3.2.3. *Similarly,*

$$\mathbf{Game}_E \hookrightarrow \mathbf{Diop}^{\mathbf{C}^+, E^+}$$

4 GENERALIZED BACKPROPAGATION

As a category of dioptics, by itself, \mathbf{Learn} is somewhat unsatisfying: it just has the same types going forward and backward. What makes the category of learners interesting is the functor $L : \mathbf{Para} \rightarrow \mathbf{Learn}$ (theorem 2.3.5) which implements gradient-based learning compositionally. In this section, we will reproduce the backpropagation functor as a composition of functors

$$\overbrace{\mathbf{Diop}^{\mathbf{C}^+, \mathbf{C}^+}_{\mathbf{FwdEuc}, \mathbf{FwdEuc}}}^{\mathbf{Para} \cong} \cong \int^{S : \mathbf{Euc}} \mathbf{Euc}(S \times X, Y) \rightarrow \overbrace{\mathbf{Diop}^{\mathbf{C}^+, \mathbf{C}^+}_{T_{\mathbb{R}}^{\triangleleft}, T_{\mathbb{R}}^{\triangleleft}}}^{\mathbf{GradLearn} :=} \rightarrow \overbrace{\mathbf{Diop}^{\mathbf{C}^+, \mathbf{C}^+}_{\Delta_{\mathbf{Set}}^{\triangleleft}, \Delta_{\mathbf{Set}}^{\triangleleft}}}^{\mathbf{Learn} \cong}$$

where $T_{\mathbb{R}}^{\triangleleft}$ is a generalization of D_Z^{\triangleleft} from definition 2.4.5, with Z set to \mathbb{R} so as to compute gradient (co)vectors.

4.1 Generalizing reverse-mode automatic differentiation

In this subsection, we will use the differential geometry of diffeological spaces (as developed in [Sou80; Lau06; Vin08]) to abstract the development of reverse-mode automatic differentiation to the category of diffeological spaces, \mathbf{Diffeo} . The motivation for this generalization is that \mathbf{Diffeo} is locally cartesian closed [BH11], which we will need in order to apply theorem 2.1.8 (and thus definition 3.0.1).

4.1.1 Generalizing derivatives

In section 2.4.1, we generalized derivatives to Fréchet derivatives in \mathbf{Euc} . The Fréchet derivative can actually be extended to an endofunctor on $\mathbf{Banach}_{\mathbf{Smooth}}$, a category of Banach spaces and (Fréchet-)smooth maps. The natural next stop on this journey of generalization would be \mathbf{Smooth} (a.k.a. \mathbf{Mfd}), the category of smooth manifolds, but we will skip straight over that to \mathbf{Diffeo} , the category of diffeological spaces [Lau06; Vin08], as \mathbf{Smooth} is not cartesian closed.

In any diffeological space X , at every point x there exists a diffeological vector space $T_x X$, called the *tangent space* at x (of X) [Vin08, definition 3.3.1]. And given a morphism of diffeological spaces $f : X \rightarrow Y$ and a point $x : X$, there exists a *linear* morphism $T_x f : T_x X \rightarrow T_{f(x)} Y$ [Vin08, proposition 3.4.4], which can be thought of as a morphism of \mathbf{DVect} , the category of diffeological vector spaces [CW17].

We can then define

$$T(f : X \rightarrow Y) : \left(\coprod_{x : X} T_x X \right) \rightarrow \left(\coprod_{y : Y} T_y Y \right)$$

which maps a tangent vector t at a point x to a tangent vector t' at a point y , such that $y := f(x)$ and $t' : T_{f(x)} Y := T_x f(x)$.

Proposition 4.1.1 ([Lau06, corollary 5.13]). *T is an endofunctor $\mathbf{Diffeo} \rightarrow \mathbf{Diffeo}$.*

4.1.2 Staging

The first step towards reverse-mode automatic differentiation is to give the affordance to perform the computation of $f : X \rightarrow Y$ in a first pass, and defer the computation of $T_x f : T_x X \rightarrow T_{f(x)} Y$ to a later stage. Informally, we want a function like

$T'f : X \rightarrow Y \times (T_x X \multimap T_{f(x)} Y)$. As it stands, this is not a meaningful type because of the dependency on x in the codomain; the full (dependent) type would be

$$T'(f : X \rightarrow Y) : \prod_{x:X} \prod_{y:Y} T_x X \multimap T_y Y$$

For diffeological spaces X, Y such that their tangent bundles are trivializable (e.g. open subsets of Euclidean spaces, affine spaces, framed manifolds, Lie groups), we can remove the dependencies on points.

Definition 4.1.2. *Given diffeological spaces X, Y such that TX, TY are trivializable as $\gamma_X : TX \cong X \times X_D$, $\gamma_Y : TY \cong Y \times Y_D$,*

$$T^+(f : X \rightarrow Y) : X \rightarrow (Y \times X_D \multimap Y_D) := x \mapsto \langle f(x), d \mapsto \langle x, d \rangle \circ \gamma_X^{-1} \circ Tf \circ \gamma_Y \circ \pi_2 \rangle$$

X_D can be thought of as a space of “directions”—or perhaps more precisely, velocities—that is globally valid in X . For Euclidean spaces, \mathbb{R}_D^n can be chosen simply as \mathbb{R}^n : the space of velocities is isomorphic to the space of positions (in a canonical way that does not depend on where in \mathbb{R}^n one is). Alternatively, X_D can be thought of simply as some space which is isomorphic to $T_x X$ for all $x : X$.

For the remainder of section 4.1, we will assume that our spaces X are each equipped with such a trivialization of the tangent bundle TX . We refer to the full subcategory of such spaces as **TrivDiffo**, and observe that **TrivDiffo** is cartesian as T is product-preserving [CW16].

4.1.3 Reverse-mode

The insight of reverse-mode automatic differentiation is that if we only care to compute derivatives of some ultimate “answer” $z : Z := f_n(f_{n-1}(\dots f_0(x)\dots))$ (e.g. a utility/loss function) with respect to each variable, then at each step of the computation we can represent derivatives by their maps into Z_D .

Definition 4.1.3.

$$T_Z^\triangleleft(f : X \rightarrow Y) : X \rightarrow \left(Y \times \left(\overbrace{(Y_D \multimap Z_D)}^k \rightarrow \overbrace{(X_D \multimap Z_D)}^d \right) \right) := x \mapsto \langle f(x), k \mapsto d \mapsto \langle x, d \rangle \circ \gamma_X^{-1} \circ Tf \circ \gamma_Y \circ \pi_2 \circ k \rangle$$

where \multimap denotes the hom-functor of **DVect** viewed as **Diffo**-enriched.

4.1.4 The generalized reverse-mode automatic-differentiation functor

We observe that the type of $T_Z^\triangleleft(f)$ is equivalent to a hom-set of optics:

$$X \rightarrow \left(Y \times \left((Y_D \multimap Z_D) \multimap (X_D \multimap Z_D) \right) \right) \cong \mathbf{Optic}_{\mathbf{Diffo}} \left((X, X_D \multimap Z_D), (Y, Y_D \multimap Z_D) \right)$$

which motivates the following definition:

Definition 4.1.4. *Given $Z : \mathbf{TrivDiffo}$,*

$$T_Z^\triangleleft : \mathbf{TrivDiffo} \rightarrow \mathbf{Optic}_{\mathbf{Diffo}} := X \mapsto (X, X_D \multimap Z_D)$$

Conjecture 4.1.5. T_Z^\triangleleft is a symmetric monoidal functor, with the oplaxator defined using addition $+$: $Z_D \times Z_D \multimap Z_D$ in the vector space Z_D , like so,

$$\begin{aligned} T_Z^\triangleleft . \Delta_{X,Y} : \overbrace{T_Z^\triangleleft(X \times Y)}^{(X \times Y, (X \times Y)_D \multimap Z_D)} &\rightsquigarrow \overbrace{(T_Z^\triangleleft X \times T_Z^\triangleleft Y)}^{(X \times Y, (X_D \multimap Z_D) \times (Y_D \multimap Z_D))} \\ &:= \left((x, y) \mapsto \left((x, y), (z_x, z_y) \mapsto d_{xy} \mapsto \left(\gamma_{X \times Y}^{-1} \left((x, y), d_{xy} \right) \circ \langle T\pi_1, T\pi_2 \rangle \circ (\gamma_X \times \gamma_Y) \circ (\pi_2 \times \pi_2) \circ (z_x \times z_y) \circ + \right) \right) \right) \end{aligned}$$

and its inverse definable due to the linearity constraint on $(X \times Y)_D \multimap Z_D$.

4.1.5 The category of gradient-based learners

Definition 4.1.6. *We define the category of gradient-based learners to be*

$$\mathbf{GradLearn} := \mathbf{Dioptric}_{T_{\mathbb{R}}^\triangleleft, T_{\mathbb{R}}^\triangleleft}$$

4.2 Automatic differentiation with parameters: from Para \rightarrow GradLearn

Conjecture 4.2.1. *We can construct a functor $T^* : \mathbf{Para} \rightarrow \mathbf{GradLearn}$ as follows:*

$$\begin{aligned}
\mathbf{Para}(X, Y) &\cong \int^{S:\mathbf{Euc}} \mathbf{Euc}(S \times X, Y) \\
\downarrow \iota:\mathbf{Euc} \hookrightarrow \mathbf{TrivDiffeo} &\rightarrow \int^{S:\mathbf{Euc}} \mathbf{TrivDiffeo}(\iota S \times \iota X, \iota Y) \\
\downarrow T_{\mathbb{R}}^{\triangleleft} &\rightarrow \int^{S:\mathbf{Euc}} \mathbf{Optic}_{\mathbf{Diffeo}}(T_{\mathbb{R}}^{\triangleleft} \iota S \otimes T_{\mathbb{R}}^{\triangleleft} \iota X, T_{\mathbb{R}}^{\triangleleft} \iota Y) \\
\downarrow \text{hom-tensor adjunction} &\rightarrow \int^{S:\mathbf{Euc}} \mathbf{Optic}_{\mathbf{Diffeo}}(T_{\mathbb{R}}^{\triangleleft} \iota S, T_{\mathbb{R}}^{\triangleleft} \iota X \rightsquigarrow T_{\mathbb{R}}^{\triangleleft} \iota Y) \\
&=: \mathbf{Dioptric}_{T_{\mathbb{R}}^{\triangleleft}, T_{\mathbb{R}}^{\triangleleft}}(\iota X, \iota Y) \\
&\hookrightarrow \mathbf{Dioptric}_{T_{\mathbb{R}}^{\triangleleft}, T_{\mathbb{R}}^{\triangleleft}}(\iota X, \iota Y) =: \mathbf{GradLearn}(\iota X, \iota Y)
\end{aligned}$$

4.3 Gradient descent: from GradLearn \rightarrow Learn

To go from $\mathbf{Dioptric}_{T_{\mathbb{R}}^{\triangleleft}, T_{\mathbb{R}}^{\triangleleft}}$ to $\mathbf{Dioptric}_{\Delta_{\mathbf{Set}}^{\rightleftharpoons}, \Delta_{\mathbf{Set}}^{\rightleftharpoons}}$, completing the analogy with theorem 2.3.5, we must choose a (global) loss function $e(x, y)$ and a step size η .

Conjecture 4.3.1. *Given a positive number $\eta : \mathbb{R}$ and a differentiable function $e(x, y) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ such that $\frac{\partial e}{\partial x}(z, -) : \mathbb{R} \rightarrow \mathbb{R}$ is invertible $\forall z : \mathbb{R}$, we can define a faithful, injective-on-objects, symmetric monoidal functor $L_{e, \eta}^*$ from the image of T^* in $\mathbf{GradLearn}$ to \mathbf{Learn} .*

Another interpretation of gradient descent is to only generate concrete updates for parameters, and still pass cotangent vectors on inputs and outputs, i.e. to form a functor from $\mathbf{Dioptric}_{T_{\mathbb{R}}^{\triangleleft}, T_{\mathbb{R}}^{\triangleleft}} \rightarrow \mathbf{Dioptric}_{T_{\mathbb{R}}^{\triangleleft}, \Delta_{\mathbf{RM}}^{\rightleftharpoons}}$; this is possible on \mathbf{RM} , the subcategory of \mathbf{Diffeo} consisting of spaces X equipped with a Riemmanian metric (inner product on the tangent space X_D), by using the duality of type $(X_D \multimap \mathbb{R}) \cong X_D$ and the exponential map of type $X \times X_D \rightarrow X$.

5 FUTURE WORK

Aside from the obvious (settling the definition of $\mathbf{Dioptric}_{F, G}$ as either a symmetric monoidal category, symmetric monoidal bicategory, or some variety of double category, finding a strong monoidal version of C^+ , and then checking formally that the relevant coherence laws hold and equivalences are respected), here are some other directions for further exploration:

5.1 Characterizing truthfulness

It would be nice to give a dioptical construction that is fully equivalent to \mathbf{Game} (and/or \mathbf{Game}_E), rather than just one that they faithfully embed into. In an extremely informal sense, one might expect restricting to a subcategory of “truthful” optics to have some connection to the notion of “lawful optics”, but even the highly abstract notion of lawfulness given in [Ril18, section 3] is only defined when forward and backward types are equal (“for optics of the form $p : (S, S) \rightsquigarrow (A, A)$ ”).

5.2 Synthesizing functors between categories of dioptics

The (alleged) functors

$$T^* : \mathbf{Para} \cong \mathbf{Dioptric}_{\mathbf{FwdEuc}, \mathbf{FwdEuc}} \rightarrow \mathbf{Dioptric}_{T_{\mathbb{R}}^{\triangleleft}, T_{\mathbb{R}}^{\triangleleft}} =: \mathbf{GradLearn}$$

and

$$L_{e, \eta}^* : \mathbf{GradLearn} := \mathbf{Dioptric}_{T_{\mathbb{R}}^{\triangleleft}, T_{\mathbb{R}}^{\triangleleft}} \rightarrow \mathbf{Dioptric}_{\Delta_{\mathbf{Set}}^{\rightleftharpoons}, \Delta_{\mathbf{Set}}^{\rightleftharpoons}} \cong \mathbf{Learn}$$

both go from one category of dioptics to another. It may be possible to construct such functors using a generic ‘recipe’ that takes as ingredients relationships (functors, natural transformations) between the ingredients of the respective dioptical constructions.

5.3 Discontinuous activation functions

One of the most popular primitives used in machine learning (aside from addition, scaling, and duplication) is the “rectified linear unit” or ReLU, defined simply as

$$\text{ReLU} : \mathbb{R} \rightarrow \mathbb{R} := x \mapsto \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Of course, this function is not smooth—at least not in the usual sense. In practice, it is considered smooth ad-hoc, with its derivative being taken piecewise. There are at least three potential ways to give this meaning without giving up totality:

1. we can consider ReLU to be *smooth almost everywhere* (infinitely differentiable at every point except for a set of measure zero, namely $\{0\}$),
2. we can consider ReLU to be *subdifferentiable* (at every point, having a “subtangent line” which touches the graph at that point and is below or touching the graph everywhere)
3. we can consider ReLU to be *semismooth from the right* (at every point, infinitely differentiable in a one-sided sense from the right)

A future direction is to investigate which, if any of these, establishes ReLU as a morphism in some appropriate category \mathcal{C} (cartesian closed, locally cartesian closed, and ideally also cocomplete) in which to define dioptics.

It should be noted that yet another approach to handling nonsmooth functions generated from conditionals is to make use of the wealth of techniques in programming language semantics for handling partiality (adapted from their original purpose of handling nontermination). Recent work in this direction includes [VSK18a; VSK18b; GMC19; GCM19]. However, this approach does not seem likely to automatically yield a way of handling ReLU that agrees with its typical handling in practice.

ACKNOWLEDGMENTS

Thanks to Eliana Lorch for key insights that led to the development of section 3, and for substantial comments on earlier draft versions. Thanks as well to Brendan Fong, Jules Hedges, and David Spivak for many illuminating conversations.

REFERENCES

- [BH11] BAEZ, John C. and Alexander HOFFNUNG. ‘Convenient categories of smooth spaces’, *Transactions of the American Mathematical Society* 363(11) (6th June 2011), pp. 5789–5825, DOI: [10.1090/S0002-9947-2011-05107-X](https://doi.org/10.1090/S0002-9947-2011-05107-X) (cited on p. 8); preliminary versions at arXiv: [0807.1704v4](https://arxiv.org/abs/0807.1704v4) [math.DG] (8th Oct. 2009); related blog post URL: https://golem.ph.utexas.edu/category/2008/05/convenient_categories_of_smooth.html (17th May 2008).
- [BPRS18] BAYDIN, Atılım G., Barak A. PEARLMUTTER, Alexey A. RADUL and Jeffrey M. SISKIND. ‘Automatic differentiation in machine learning: A survey’, *Journal of Machine Learning Research* 18(153) (Apr. 2018), URL: <http://jmlr.org/papers/v18/17-468.html> (cited on p. 5); preliminary version on arXiv: [1502.05767v4](https://arxiv.org/abs/1502.05767v4) [cs.SC] (5th Feb. 2018).
- [BHW18] BOLT, Joe, Jules HEDGES and Viktor WINSCHERL. *The algebra of predicting agents*, 27th Mar. 2018, arXiv: [1803.10131v1](https://arxiv.org/abs/1803.10131v1) [cs.GT] (cited on p. 3).
- [CW16] CHRISTENSEN, J. Daniel and Enxin WU. ‘Tangent spaces and tangent bundles for diffeological spaces’, *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 57(1) (2016), pp. 3–50, ISSN: 1245-530X, URL: <http://cahierstgdc.com/wp-content/uploads/2017/11/ChristensenWu.pdf> (cited on p. 9).
- [CW17] CHRISTENSEN, J. Daniel and Enxin WU. *Diffeological vector spaces*, 22nd Mar. 2017, arXiv: [1703.07564v1](https://arxiv.org/abs/1703.07564v1) [math.DG] (cited on p. 8).
- [dPai91] De PAIVA, Valeria C. V. *The Dialectica categories*, tech. rep. UCAM-CL-TR-213, University of Cambridge Computer Laboratory, Jan. 1991, URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-213.pdf> (cited on pp. 2, 3).
- [Ell18] ELLIOTT, Conal. ‘The simple essence of automatic differentiation’, *Proc. ACM on Programming Languages*, ICFP 2018 (St. Louis, MO, USA), vol. 2.70, 24th–26th Sept. 2018, DOI: [10.1145/3236765](https://doi.org/10.1145/3236765) (cited on pp. 1, 2, 5); extended version on arXiv: [1804.00746v4](https://arxiv.org/abs/1804.00746v4) [cs.PL]; URL: <http://conal.net/papers/essence-of-ad/> (Mar. 2018).

- [FST18] FONG, Brendan, David I. SPIVAK and Rémy TUYÉRAS. ‘Backprop as Functor’, SYCO 1 (Birmingham, UK), 20th–21st Sept. 2018, URL: <http://events.cs.bham.ac.uk/syco/1/slides/fong.pdf> (cited on pp. 1, 4); based on full paper: *Backprop as Functor: A compositional perspective on supervised learning*; 13th Dec. 2017; arXiv: [1711.10455v2](https://arxiv.org/abs/1711.10455v2) [math.CT].
- [GCM19] GALLAGHER, Jonathan, Geoff CRUTTWELL and Ben MACADAM. ‘Towards formalizing and extending differential programming using tangent categories’, *Applied Category Theory*, ACT 2019 (Oxford, UK, 15th–19th July 2019), 1st July 2019, URL: <http://www.cs.ox.ac.uk/ACT2019/preproceedings/Jonathan%20Gallagher,%20Geoff%20Cruttwell%20and%20Ben%20MacAdam.pdf> (cited on p. 11).
- [GMC19] GALLAGHER, Jonathan, Ben MACADAM and Geoff CRUTTWELL. ‘Towards formalizing and extending differential programming via tangent categories’, Presentation at SYCO 4 (Chapman University, USA), 22nd–23rd May 2019 (cited on p. 11).
- [Hed18] HEDGES, Jules. ‘Compositional game theory’, SYCO 1 (Birmingham, UK), 20th–21st Sept. 2018 (cited on pp. 1, 3); based on full paper: GHANI, Neil, Jules HEDGES, Viktor WINSCHERL and Philipp ZAHN. ‘Compositional game theory’, *Logic in Computer Science*; LICS ’18 (Oxford, UK); 9th–12th July 2018; DOI: [10.1145/3209108.3209165](https://doi.org/10.1145/3209108.3209165); preliminary version on arXiv: [1603.04641v3](https://arxiv.org/abs/1603.04641v3) [cs.GT] (15th Feb. 2018).
- [Lau06] LAUBINGER, Martin. ‘Diffeological spaces’, *Proyecciones Journal of Mathematics* 25(2) (Aug. 2006), pp. 151–178, DOI: [10.4067/S0716-09172006000200003](https://doi.org/10.4067/S0716-09172006000200003), URL: <https://scielo.conicyt.cl/pdf/proy/v25n2/art03.pdf> (cited on p. 8).
- [Lor17] LOREGIAN, Fosco. *This is the (co)end, my only (co)friend*, 4th Dec. 2017, arXiv: [1501.02503v4](https://arxiv.org/abs/1501.02503v4) [math.CT] (cited on p. 2).
- [Ril18] RILEY, Mitchell. *Categories of Optics*, 7th Sept. 2018, arXiv: [1809.00738v2](https://arxiv.org/abs/1809.00738v2) [math.CT] (cited on pp. 1–3, 6, 10).
- [Sou80] SOURIAU, Jean-Marie. ‘Groupes différentiels’, *Differential Geometrical Methods in Mathematical Physics* (Aix-en-Provence, 3rd–7th Sept. 1979), Lecture Notes in Mathematics 836, Springer-Verlag, 1980, chap. II, pp. 91–128, DOI: [10.1007/BFb0089728](https://doi.org/10.1007/BFb0089728) (cited on p. 8).
- [Spi65] SPIVAK, Michael D. *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus*, Mathematics monograph series, Benjamin/Cummings, 26th Oct. 1965, ISBN: [9780805390216](https://www.amazon.com/dp/9780805390216) (cited on p. 5).
- [SK19] SPRUNGER, David and Shin-ya KATSUMATA. *Differentiable Causal Computations via Delayed Trace*, 4th Mar. 2019, arXiv: [1903.01093v1](https://arxiv.org/abs/1903.01093v1) [cs.LO] (cited on p. 6).
- [VSK18a] VÁKÁR, Matthijs, Sam STATON and Ohad KAMMAR. ‘Diffeological Spaces and Denotational Semantics for Differential Programming’, Presentation at a special session of MFPS 2018, 8th June 2018, URL: <http://www.columbia.edu/~mv2745/research/08-06-2018-Halifax.pdf> (cited on p. 11).
- [VSK18b] VÁKÁR, Matthijs, Sam STATON and Ohad KAMMAR. ‘Diffeological Spaces and Denotational Semantics for Differential Programming’, Presentation at Domains workshop of FLoC 2018, 8th July 2018, URL: <https://andrejbauer.github.io/domains-floc-2018/slides/Matthijs-Kammar-Staton.pdf> (cited on p. 11).
- [Vin08] VINCENT, Martin. *Diffeological differential geometry*, MS thesis, University of Copenhagen, 28th May 2008, URL: <https://www.math.ku.dk/english/research/tfa/top/paststudents/ms-theses/martinvincent.msthesis.pdf> (cited on p. 8).
- [Wie18] WIEGLEY, John. *Category Theory in Coq*, GitHub repository, 2018, URL: <https://github.com/jwiegley/category-theory/> (visited on 26th Oct. 2018) (cited on p. 13).

A1 COQ FORMALIZATION: PRELIMINARIES

These appendices mirror the structure of the paper. They are still very much a work in progress, but the goal is to formalize every definition, proposition and theorem.

Set Universe Polymorphism.

Set Primitive Projections.

Set Implicit Arguments.

Generalizable All Variables.

Require Import Coq.Program.Util.

Require Import Coq.Program.Equality.

A1.1 Category theory

There appears to be no widely agreed-upon Coq formalization of basic categorical concepts, although of course there are several excellent libraries. We have elected to make this formalization self-contained, but these definitions are heavily based on those of [Wie18]. However, we have removed the convention that all categories are quotient categories, since the question of “associativity on the nose” is relevant here. To compensate for this (and be able to derive more strict equalities), we use two axioms from Coq’s standard library, which are not included in its default logic: functional extensionality (functions are equal if they agree at every point in their domain) and proof irrelevance (proofs of equal propositions are equal).

```
Require Import Coq.Logic.FunctionalExtensionality.
Require Import Coq.Logic.ProofIrrelevance.
```

Rest assured that the logic, thus extended, is still intuitionistic, and that these axioms are certainly admissible in mainstream (classical, ZFC) mathematics.

We use this tactic to prove lemmas showing that equality of entity does not depend on their properties (only their structure), due to proof irrelevance.

```
Ltac ignore_properties u v :=
  intros; destruct u; destruct v; simpl in *; repeat subst; f_equal; apply proof_irrelevance.
```

This lemma helps to prove heterogenous equalities.

```
Lemma eq_heq (A : Type) (x y : A) : x = y → x ~ y.
Proof. intro H; rewrite H; constructor. Defined.
```

A1.1.1 Definition of a category

```
Record Category := {
  obj : Type;
  hom_universe := Type : Type;
  hom : obj → obj → hom_universe;

  dom {A B} (f : hom A B) := A;
  cod {A B} (f : hom A B) := B;

  id (A : obj) : hom A A;
  compose {A B C} : hom A B → hom B C → hom A C;

  id_left {A B} (f : hom A B) : compose (id A) f = f;
  id_right {A B} (f : hom A B) : compose f (id B) = f;
  comp_assoc {A B C D} (f : hom A B) (g : hom B C) (h : hom C D) :
    compose (compose f g) h = compose f (compose g h);
}.
Coercion obj : Category → Sortclass.
Coercion hom : Category → Funclass.
```

```
Notation "f ; g" := (compose _ f g) (at level 97, right associativity).
Notation "f ;[ C ] g" := (compose C f g) (at level 97).
```

A1.1.2 Definition of a functor

```
Record Functor (C D : Category) := {
  fobj : C → D;
  fmap {x y : C} (f : C x y) : D (fobj x) (fobj y);
  functorial_nullary {x : C} : fmap (id C x) = id D (fobj x);
  functorial_binary {x y z : C} (f : C x y) (g : C y z) :
    fmap (compose C f g) = compose D (fmap f) (fmap g);
}.
Coercion fobj : Functor → Funclass.
Notation "C → D" := (@Functor C D) (at level 91, right associativity).
```

Lemma *functor_eq* {C D : Category} (F G : C → D) :
 (fobj F = fobj G) → ((@fmap C D F) ~ (@fmap C D G)) → F = G.

Proof. *ignore_properties F G. Qed.*

A1.1.3 Definition of Cat, the (1-)category of (1-)categories

Program Definition *Id* {C : Category} : C → C := { |
 fobj x := x;
 fmap _ f := f;
 | }.

Program Definition *Compose* {C D E : Category} (F : C → D) (G : D → E) : (C → E) := { |
 fobj x := (fobj G (fobj F x));
 fmap _ f := (fmap G (fmap F f));
 | }.

Program Definition *Cat* : Category := { |
 obj := Category;
 hom := @Functor;
 id := @Id;
 compose := @Compose;
 | }.

Coq < Check Cat.

Cat
 : Category

A1.1.4 Definition of an opposite category

Program Definition *Opposite* (C : Category) : Category := { |
 obj := obj C;
 hom := fun x y ↦ hom C y x;
 id := fun x ↦ id C x;
 compose x y z := fun f g ↦ compose C g f;
 | }.

Coq < Check Opposite.

Opposite
 : Category -> Category

Notation "C ^op" := (Opposite C) (at level 7, format "C ^op").

Definition *op* {C : Category} {x y : C} (f : hom C y x) : hom C ^op x y := f.

Definition *unop* {C : Category} {x y : C} (f : hom C ^op x y) : hom C y x := f.

A1.1.5 Definition of a product category

Program Definition *Product* (C1 C2 : Category) : Category := { |
 obj := obj C1 × obj C2;
 hom := fun x y ↦ ((hom C1 (fst x) (fst y)) × (hom C2 (snd x) (snd y)))%type;
 id := fun x ↦ (id C1 (fst x), id C2 (snd x));
 compose := fun _ _ f g ↦
 (compose C1 (fst f) (fst g), compose C2 (snd f) (snd g));
 | }.

Coq < Check Product.

Product
 : Category -> Category -> Category

Notation "C × D" := (Product C D) (at level 90, right associativity, format "C × D").

Program Definition *morphism_pairing* {C D : Category} {c1 c2 : C} {d1 d2 : D}
 (f : C c1 c2) (g : D d1 d2) : (C × D) (c1, d1) (c2, d2) := _.

Notation "×(f , g)" := (morphism_pairing f g).

Program Definition *functor_pairing* {C1 C2 C3 C4 : Category}

 $(F : C1 \rightarrow C3) (G : C2 \rightarrow C4) : (C1 \times C2 \rightarrow C3 \times C4) := \{ |$

 $\text{fobj } x := (F (\text{fst } x), G (\text{snd } x));$

 $\text{fmap } - - f := (\text{fmap } F (\text{fst } f), \text{fmap } G (\text{snd } f));$

 $| \}$.

Notation "[| F , G |]" := (*functor_pairing* F G) (at level 92).

A1.1.6 Diagonal and constant functors

Program Definition $\Delta 2$ {C : Category} : C \rightarrow C \times C := { |

 $\text{fobj } x := (x, x);$

 $\text{fmap } - - f := (f, f);$

 $| \}$.

Program Definition *const* {C D : Category} (d : D) : C \rightarrow D := { |

 $\text{fobj } _ := d;$

 $\text{fmap } - - - := \text{id } D \text{ } d;$

 $| \}$.

Notation "const[d]" := (*const* d) (at level 91).

A1.1.7 Definition of a natural transform

Record *NatTrans* {C D : Category} (F G : C \rightarrow D) : Type := {

 $\text{transform } (x : C) : \text{hom } D (F x) (G x);$

 $\text{naturality } (x y : C) (f : \text{hom } C x y) :$

 $\text{compose } D (\text{fmap } F f) (\text{transform } y) = \text{compose } D (\text{transform } x) (\text{fmap } G f);$

 $\}$.

Coercion *transform*: *NatTrans* \rightarrow *Funclass*.

Notation "F \Longrightarrow G" := (*NatTrans* F G) (at level 94).

Lemma *nat_trans_eq* {C D : Category} {F G : C \rightarrow D} (N M : F \Longrightarrow G):

 $(N.(\text{transform}) = M.(\text{transform})) \rightarrow N = M.$

Proof. *ignore_properties* N M. **Qed.**

A1.1.8 Definition of a functor category

Program Definition *FunctorCategory* (C D : Category) : Category := { |

 $\text{obj} := C \rightarrow D;$

 $\text{hom } F G := F \Longrightarrow G;$

 $\text{id } F := \{ | \text{transform } x := \text{id } D (F x) | \};$

 $\text{compose } F G H \alpha \beta := \{ | \text{transform } x := \text{compose } D (\alpha x) (\beta x) | \};$

 $| \}$.

Coq < Check *FunctorCategory*.

FunctorCategory

 : Category \rightarrow Category \rightarrow Category

Notation "[| C , D]" := (*FunctorCategory* C D) (at level 89, right associativity).

A1.1.9 Definition of isomorphism

Definition *section* {C : Category} {x y : C} (f : hom C x y) (g : hom C y x) :=

 $\text{compose } C g f = \text{id } C y.$

Hint *Unfold* *section*.

Record *iso* {C : Category} (x y : C) := {

 $\text{fwd} : \text{hom } C x y;$

 $\text{bwd} : \text{hom } C y x;$

 $\text{left_inv} : \text{section } \text{bwd } \text{fwd};$

 $\text{right_inv} : \text{section } \text{fwd } \text{bwd};$

 $\}$.

Lemma *iso_eq* {C : Category} {x y : C} (f g : iso x y):

$$f.(fwd) = g.(fwd) \rightarrow f = g.$$

Program Definition *flip* {C : Category} {x y : C} (i : iso x y) := { |
 fwd := bwd i;
 bwd := fwd i;
 | }.

Solve All Obligations with `destruct i; assumption.`

A1.1.10 Core of a category, defined as an endofunctor of Cat

Program Definition *Core* : Cat \rightarrow Cat := { |
 fobj C := { |
 obj := obj C;
 hom x y := iso x y;
 | };
 | }.

Coq < Check Core.

Core
 : Cat \rightarrow Cat

A1.1.11 Natural isomorphism, as core of functor category

Definition *NatIso* {C D : Category} := Core ([C,D]).

Ltac *apply_nat_section* H a :=
 apply (f_equal (fun t \mapsto t.(transform) a)) in H; cbn in H; try rewrite H.

Program Definition *nat_iso_pairing* {C D : Category} {F1 F2 F3 F4 : [C,D]}
 (N : NatIso F1 F3) (M : NatIso F2 F4) : NatIso ([F1,F2|]) ([F3,F4|]) := { |
 fwd := { | transform x := (fwd N (fst x), fwd M (snd x)) | };
 bwd := { | transform x := (bwd N (fst x), bwd M (snd x)) | };
 | }.

Coq < Check @nat_iso_pairing.

@nat_iso_pairing
 : forall (C D : Category) (F1 F2 F3 F4 : [C, D]),
 NatIso F1 F3 -> NatIso F2 F4 -> NatIso ([F1, F2|]) ([F3, F4|])

Notation "[[| N, M |]]" := (*nat_iso_pairing* N M) (at level 92).

A1.1.12 Definition of a monoidal category

Program Definition *prod_cat_assoc* {C D E : Category} :
 @iso Cat (C \times (D \times E)) ((C \times D) \times E) := { |
 fwd := { | fobj x := ((fst x, fst (snd x)), snd (snd x)); | };
 bwd := { | fobj x := (fst (fst x), (snd (fst x), snd x)); | };
 | }.

Solve All Obligations with `functor_eq.`

Program Definition *prod_cat_symm* {C D : Category} :
 @iso Cat (C \times D) (D \times C) := { |
 fwd := { | fobj x := (snd x, fst x); | };
 bwd := { | fobj x := (snd x, fst x); | };
 | }.

Solve All Obligations with `functor_eq.`

Class *Monoidal* (C : Category) := {
 monoidal_unit: C;
 tensor: C \times C \rightarrow C;
 associator: NatIso
 (fwd prod_cat_assoc §[Cat] [| tensor, Id |] §[Cat] tensor)
 ([Id, tensor |] §[Cat] tensor);
 left_unitor: NatIso
 (Δ 2 §[Cat] [| const[monoidal_unit], Id |] § tensor) Id;

```

right_unity: NatIso
  (Δ2 §[Cat] [| Id, const[monoidal_unit] |] § tensor) Id;
triangle_eq (x y : C):
  ((fwd associator (x,(monoidal_unit,y))) §
   fmap tensor ×(id C x, fwd left_unity y)) =
  fmap tensor ×(fwd right_unity x, id C y);
pentagon_eq (w x y z : C):
  ((fwd associator (tensor (w,x),(y,z))) §
   (fwd associator (w,(x,tensor (y,z)))) =
  (fmap tensor ×(fwd associator (w,(x,y)), id C z) §
   (fwd associator (w,(tensor (x,y),z))) §
   fmap tensor ×(id C w, fwd associator (x,(y,z))));
}.

```

Notation "x y" := (tensor (x,y)) (at level 70, right associativity).

Notation "(| f, g |)" := (fmap tensor ×(f,g)).

Lemma interchange '@Monoidal C' (u v w x y z : C)
 (f1 : C u v) (f2 : C v w) (g1 : C x y) (g2 : C y z):
 (| f1 § f2, g1 § g2 |) = ((| f1, g1 |) § (| f2, g2 |)).

Proof. autorewrite with functorial; reflexivity. **Qed.**

Hint Rewrite @interchange : isotopy.

Lemma interchange1 '@Monoidal C' (u v w x y : C)
 (f1 : C u v) (f2 : C v w) (g : C x y):
 (| f1 § f2, g |) = ((| f1, g |) § (| f2, id C y |)).

Proof. rewrite ← interchange; autorewrite with category; reflexivity. **Qed.**

Hint Rewrite @interchange1 : isotopy.

Lemma shift_bwd_associator '@Monoidal C' (u v w x y z : C)
 (f1 : C u x) (f2 : C v y) (f3 : C w z):
 ((bwd associator (u,(v,w))) § (| (| f1, f2 |), f3 |))
 = ((| f1, (| f2, f3 |) |) § (bwd associator (x,(y,z)))).

Proof.
 symmetry; apply (naturality (bwd associator) (u,(v,w)) (x,(y,z)) (f1,(f2,f3))).

Qed.

Hint Rewrite @shift_bwd_associator : isotopy.

Class SymmetricMonoidal (C : Category) := {
 smc_is_monoidal : > Monoidal C;
 braiding : NatIso (tensor) (fwd prod_cat_symm § tensor);
 braiding_involutive (x y : C):
 (fwd braiding (x,y) § fwd braiding (y,x)) = id C -;
 hexagon_eq (x y z : C):
 (fwd associator (x,(y,z)) §
 fwd braiding (x,(tensor (y,z))) §
 fwd associator (y,(z,x))) =
 (fmap tensor ×(fwd braiding (x,y), id C z) §
 fwd associator (y,(x,z)) §
 fmap tensor ×(id C y, fwd braiding (x,z)));
}.

A1.1.13 Definition of a bicategory

```

Record Bicategory := {
  bi_obj: Type;
  bi_hom: bi_obj → bi_obj → Category;
  bi_id a : bi_hom a a;
  bi_comp a b c : (bi_hom a b) × (bi_hom b c) → (bi_hom a c);
  bi_associator {a b c d} : NatIso
    (fwd prod_cat_assoc §[Cat] [| bi_comp a b c, Id |] §[Cat] bi_comp a c d)
    ([| Id, bi_comp b c d |] §[Cat] bi_comp a b d);
  bi_left_unity {a b} : NatIso
    (Δ2 §[Cat] [| const[bi_id a], Id |] §[Cat] bi_comp a a b) Id;

```

```

bi_right_untor {a b} : NatIso
  (Δ2 §[Cat] [| Id, const[bi_id b]|] §[Cat] bi_comp a b b) Id;
bi_triangle_eq {a b c : bi_obj} (x: bi_hom a b) (y: bi_hom b c):
  (fwd bi_associator (x,(bi_id b,y))) §
  fmap (bi_comp a b c) ×(id - x, fwd bi_left_untor y) =
  fmap (bi_comp a b c) ×(fwd bi_right_untor x, id - y);
bi_pentagon_eq {a b c d e : bi_obj}
  (w: bi_hom a b) (x: bi_hom b c) (y: bi_hom c d) (z: bi_hom d e) :
  ((fwd bi_associator (bi_comp a b c (w,x),(y,z))) §
  (fwd bi_associator (w,(x,bi_comp c d e (y,z)))))) =
  (fmap (bi_comp a d e) ×(fwd bi_associator (w,(x,y)), id - z) §
  (fwd bi_associator (w,(bi_comp b c d (x,y),z))) §
  fmap (bi_comp a b e) ×(id - w, fwd bi_associator (x,(y,z)))));
}.

```

A monoidal category can be considered a bicategory with one object.

```

Definition monoidal_as_bicat '@Monoidal C} : Bicategory := { |
  bi_obj := unit;
  bi_hom _ _ := C;
  bi_id _ := monoidal_unit;
  bi_comp _ _ _ := tensor;
  bi_associator _ _ _ := associator;
  bi_left_untor _ _ := left_untor;
  bi_right_untor _ _ := right_untor;
  bi_triangle_eq _ _ _ := triangle_eq;
  bi_pentagon_eq _ _ _ _ := pentagon_eq;
| }.

```

```

Coq < Check @monoidal_as_bicat.
@monoidal_as_bicat
  : forall C : Category, Monoidal C -> Bicategory

```

A2 COQ FORMALIZATION: BACKGROUND

```

Definition Lens '@Monoidal C} : ((C × C) × (C × C)) → Set :=
  fun '(X,X')(Y,Y') => (C X Y × C (X ⊗ Y) X')%type.

```

Section CategoryOfOptics.

```
Context '@SymmetricMonoidal C}.
```

```
Record ConcreteOptic (X X' Y Y' : obj C) := {
  M : obj C;
  optic_fwd : C X (M ⊗ Y);
  optic_bwd : C (M ⊗ Y) X';
}.

```

```
Record ConcreteOpticMorphism {X X' Y Y'} (L1 L2 : ConcreteOptic X X' Y Y') := {
  optic_convert : C (M L1) (M L2);
  optic_commute_fwd : (optic_fwd L1 § (| optic_convert, id C Y |)) = (optic_fwd L2);
  optic_commute_bwd : ((| optic_convert, id C Y' |) § optic_bwd L2) = (optic_bwd L1);
}.

```

```
Lemma ConcreteOpticMorphism_eq {X X' Y Y'} {L1 L2: ConcreteOptic X X' Y Y'}
  (M1 M2: ConcreteOpticMorphism L1 L2) :
  (optic_convert M1 = optic_convert M2) → M1 = M2.

```

Proof. ignore_properties M1 M2. Qed.

```
Program Definition Optic_bicat : Bicategory := { |
  bi_obj := obj (C × C);
  bi_hom '(X,X')(Y,Y') := { |
    obj := ConcreteOptic X X' Y Y';
    hom L1 L2 := ConcreteOpticMorphism L1 L2;
  |

```

```

|};
bi_id '(X,X') := {
  M := monoidal_unit;
  optic_fwd := bwd left_unity X;
  optic_bwd := fwd left_unity X';
|};
bi_comp '(X,X') '(Y,Y') '(Z,Z') := {
  fobj '(L1,L2) := {
    M := M L1 ⊗ M L2;
    optic_fwd := (optic_fwd L1) ;
                  (| id C -, optic_fwd L2 |) ;
                  (bwd associator (-,(-, Z)));
    optic_bwd := (fwd associator (-,(-, Z'))) ;
                  (| id C -, optic_bwd L2 |) ;
                  (optic_bwd L1);
  };
|};
|};
|}.

```

Next Obligation.

unshelve esplit.

1: exact (id C -).

all: autorewrite with functorial category; easy.

Defined.

Next Obligation.

unshelve esplit.

1: exact (optic_convert X0 ; optic_convert X1).

all: destruct X0, X1; cbn; rewrite interchange1.

- rewrite ← comp_assoc, optic_commute_fwd0, optic_commute_fwd1; reflexivity.

- rewrite comp_assoc, optic_commute_bwd1, optic_commute_bwd0; reflexivity.

Defined.

Next Obligation.

destruct Heq_anonymous, Heq_anonymous0;

apply ConcreteOpticMorphism_eq; lazy; easy.

Defined.

Next Obligation.

destruct Heq_anonymous, Heq_anonymous0;

apply ConcreteOpticMorphism_eq; lazy; easy.

Defined.

Next Obligation.

destruct Heq_anonymous, Heq_anonymous0;

apply ConcreteOpticMorphism_eq; lazy; easy.

Defined.

Next Obligation.

destruct c, c0; cbn in *; unshelve esplit.

- exact ((| optic_convert0, optic_convert1 |)).

- cbn; autorewrite with functorial category isotopy.

The Coq development only goes this far for now.

End CategoryOfOptics.

A3 COQ FORMALIZATION: DIOPTICS

nothing here yet

A4 COQ FORMALIZATION: GENERALIZED BACKPROPAGATION

nothing here yet