Differentiating stateful processes

David Sprunger
NII, Japan & Birmingham, UK

LIPN
12 November 2020

(My) animating philosophy

Theoreticians have important skills needed to advance our understanding of machine learning and neural networks.

(My) animating philosophy

Theoreticians have important skills needed to advance our understanding of machine learning and neural networks.

The kinds of questions different researchers ask affects the kinds of results we see in the fields they work in.

(My) animating philosophy

Theoreticians have important skills needed to advance our understanding of machine learning and neural networks.

The kinds of questions different researchers ask affects the kinds of results we see in the fields they work in. In deep learning:

- Engineers answer questions like
  - Can we do this better with different hardware? (GPUs, CUDA and TPUs)
  - Can we build software to handle common use cases? (Tensorflow, PyTorch)

(My) animating philosophy

Theoreticians have important skills needed to advance our understanding of machine learning and neural networks.

The kinds of questions different researchers ask affects the kinds of results we see in the fields they work in. In deep learning:

- Engineers answer questions like
  - Can we do this better with different hardware? (GPUs, CUDA and TPUs)
  - Can we build software to handle common use cases? (Tensorflow, PyTorch)
- Experimentalists answer questions like
  - Is there a more performant network architecture for this task? (CNNs)
  - Are there ways to speed up training processes? (Adagrad, Adam)
  - What other areas can we use these in? (many examples)

(My) animating philosophy

Theoreticians have important skills needed to advance our understanding of machine learning and neural networks.

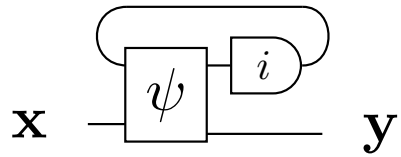
The kinds of questions different researchers ask affects the kinds of results we see in the fields they work in. In deep learning:

- Engineers answer questions like
  - Can we do this better with different hardware? (GPUs, CUDA and TPUs)
  - Can we build software to handle common use cases? (Tensorflow, PyTorch)
- Experimentalists answer questions like
  - Is there a more performant network architecture for this task? (CNNs)
  - Are there ways to speed up training processes? (Adagrad, Adam)
  - What other areas can we use these in? (many examples)
- What can theoreticians answer?

Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

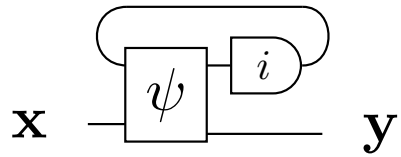
$\mathbf{\Psi}$:



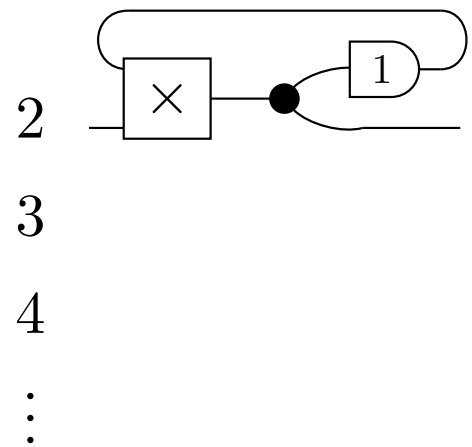
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



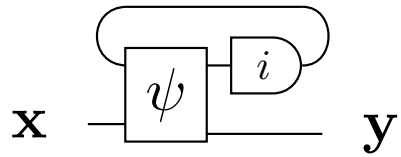
Operationally, these work as you expect:



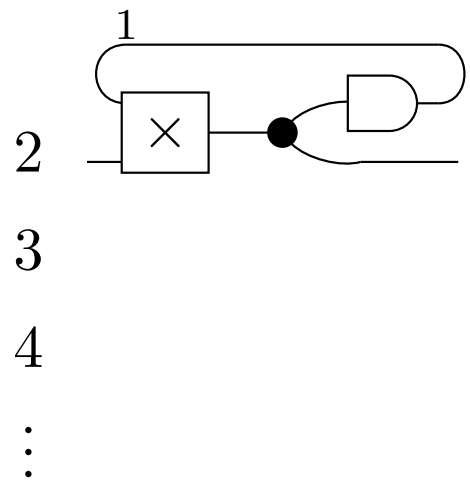
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\boldsymbol{\Psi}$:



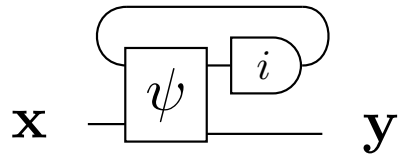
Operationally, these work as you expect:



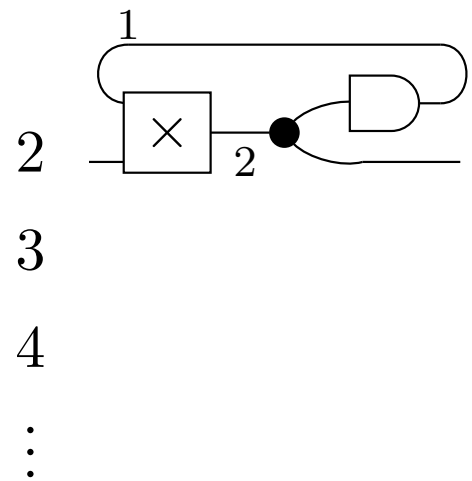
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



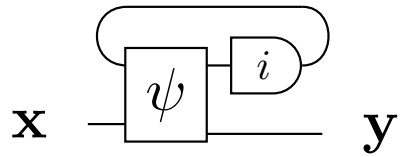
Operationally, these work as you expect:



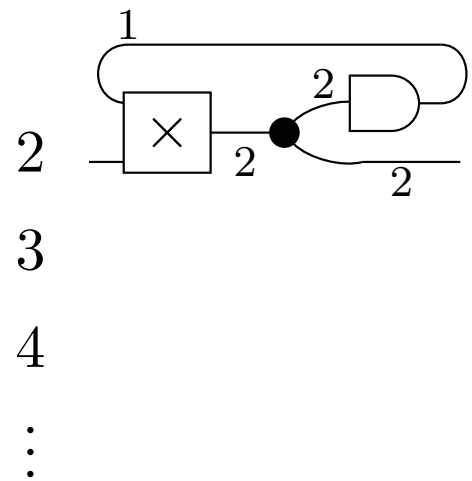
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



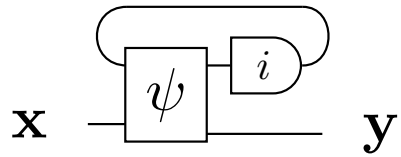
Operationally, these work as you expect:



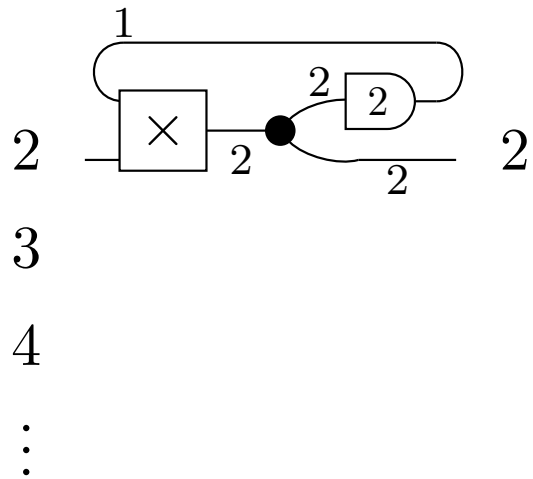
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



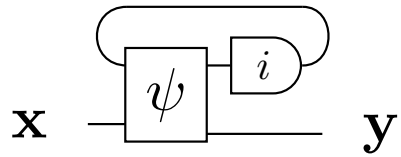
Operationally, these work as you expect:



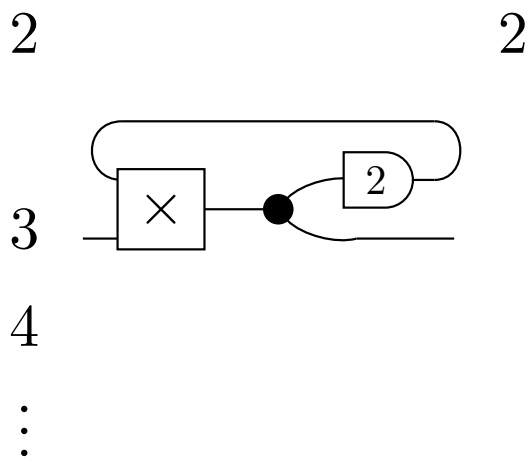
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



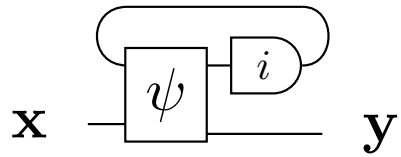
Operationally, these work as you expect:



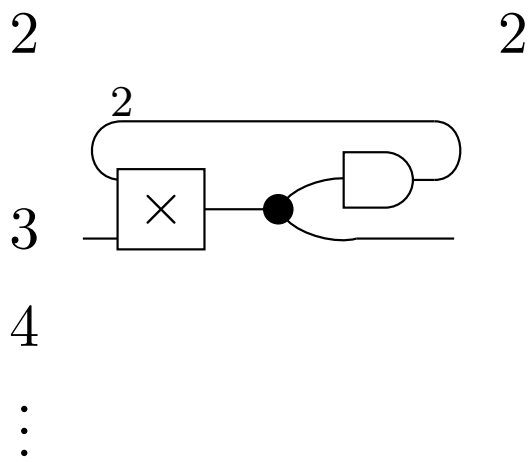
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



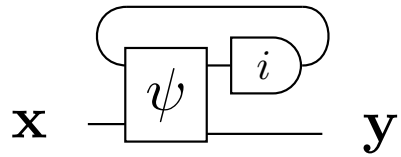
Operationally, these work as you expect:



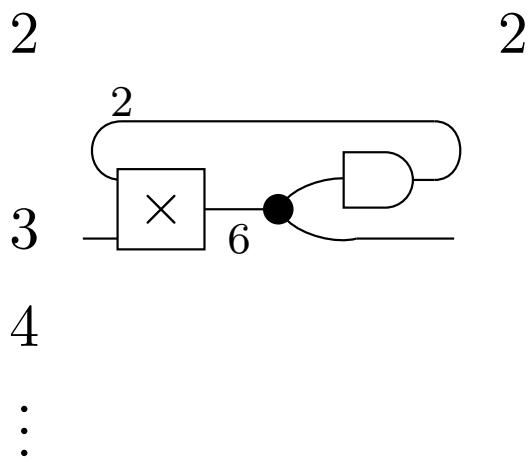
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



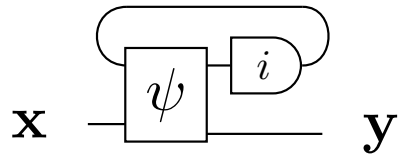
Operationally, these work as you expect:



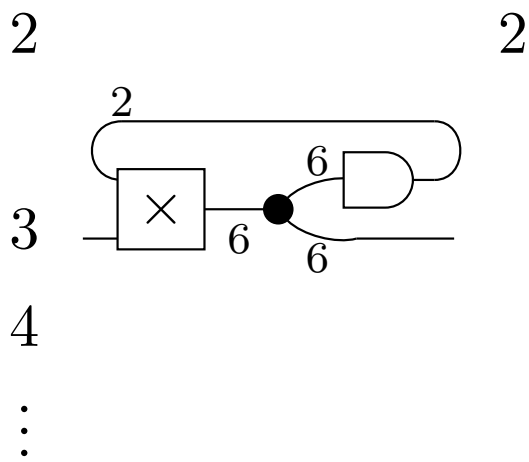
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



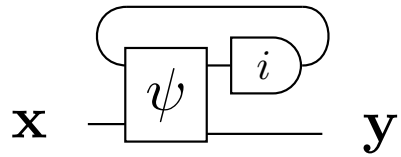
Operationally, these work as you expect:



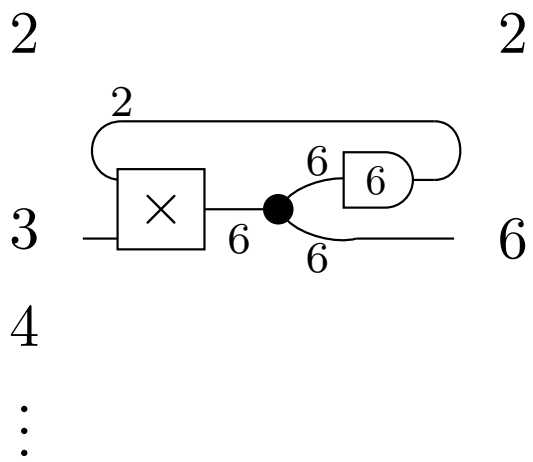
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



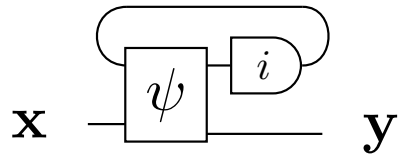
Operationally, these work as you expect:



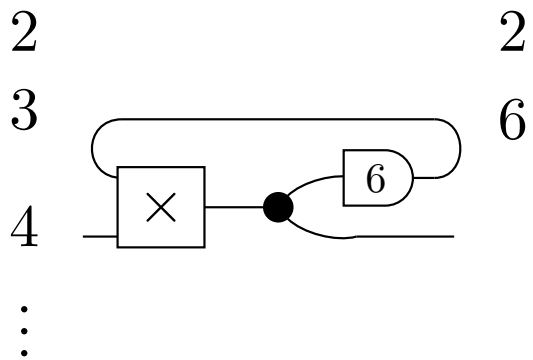
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\mathbf{\Psi}$:



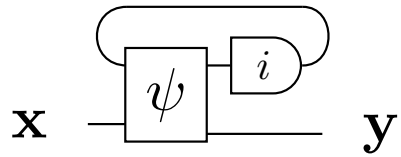
Operationally, these work as you expect:



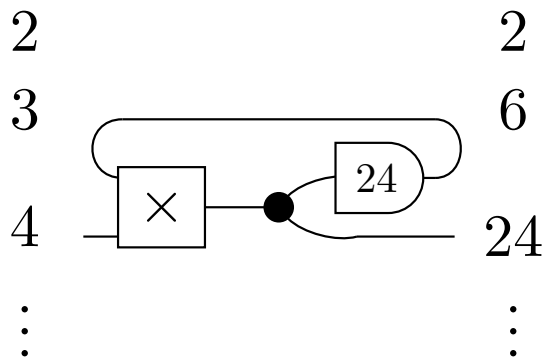
Recurrent neural networks

Recurrent neural networks (RNNs) process lists of inputs using *state*, which is stored in *registers*:

$\boldsymbol{\Psi}$:

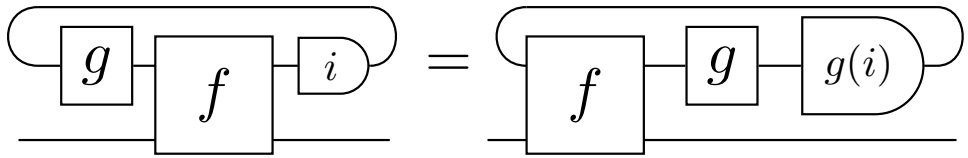


Operationally, these work as you expect:



Natural questions for theoreticians

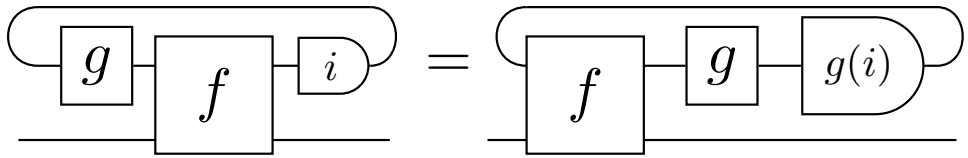
- How do we reason about equivalence of RNNs? For example, it seems like



ought to hold.

Natural questions for theoreticians

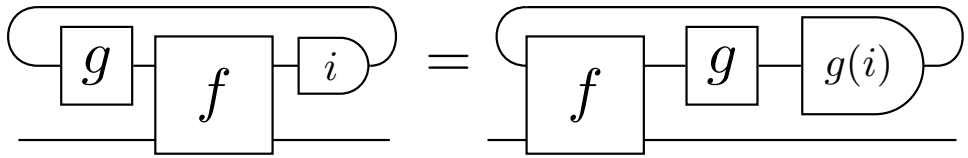
- How do we reason about equivalence of RNNs? For example, it seems like



  ought to hold.
- What are the differences between RNNs and Mealy machines?

Natural questions for theoreticians

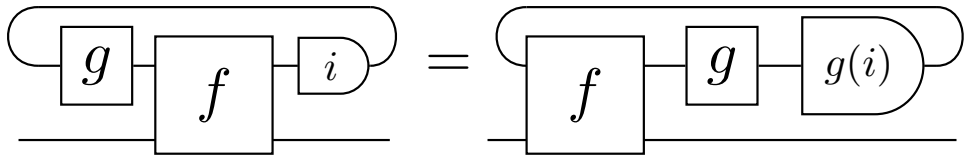
- How do we reason about equivalence of RNNs? For example, it seems like



  ought to hold.
- What are the differences between RNNs and Mealy machines?
- RNNs are not pure functions—their output depends on the current state. What does it mean to train RNNs by "following the gradient" then?

Natural questions for theoreticians

- How do we reason about equivalence of RNNs? For example, it seems like



  ought to hold.
- What are the differences between RNNs and Mealy machines?
- RNNs are not pure functions—their output depends on the current state. What does it mean to train RNNs by "following the gradient" then?
- What compositional properties does RNN training have? (Can we give the training process for sequentially composed RNNs in terms of the component RNNs?)

Outline

Cartesian differential categories: overview

We treat differentiation synthetically, following Blute, Cockett and Seely '09. (Interesting Q: can we get similar results with Fréchet derivatives in infinite-dimensional spaces?)

Cartesian differential categories: overview

We treat differentiation synthetically, following Blute, Cockett and Seely '09. (Interesting Q: can we get similar results with Fréchet derivatives in infinite-dimensional spaces?)

A **Cartesian differential category** (CDC) is a category *with certain structure* that has a differential operator *with certain properties*.

Cartesian differential categories: overview

We treat differentiation synthetically, following Blute, Cockett and Seely '09. (Interesting Q: can we get similar results with Fréchet derivatives in infinite-dimensional spaces?)

A **Cartesian differential category** (CDC) is a category *with certain structure* that has a differential operator *with certain properties*.

The derivative from multivariable calculus has the following typing:

$$\frac{f : \mathbb{R}^n \to \mathbb{R}^m}{Df : \mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)}$$

where $(Df)(x) : \mathbb{R}^n \to \mathbb{R}^m$ is a *linear* map, the Jacobian matrix.

Cartesian differential categories: overview

We treat differentiation synthetically, following Blute, Cockett and Seely '09. (Interesting Q: can we get similar results with Fréchet derivatives in infinite-dimensional spaces?)

A **Cartesian differential category** (CDC) is a category *with certain structure* that has a differential operator *with certain properties*.

The derivative from multivariable calculus has the following typing:

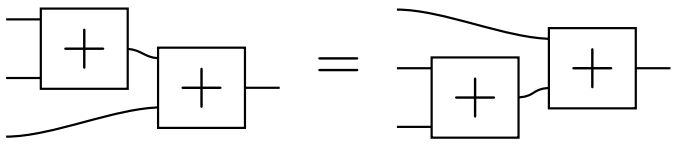
$$\frac{f : \mathbb{R}^n \to \mathbb{R}^m}{Df : \mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)}$$

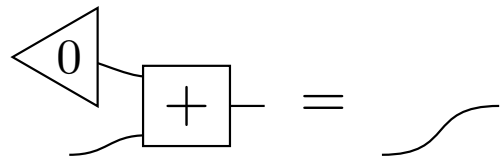
where $(Df)(x) : \mathbb{R}^n \to \mathbb{R}^m$ is a *linear* map, the Jacobian matrix. To avoid assumptions about closedness, differential operators in CDCs are typed as follows:

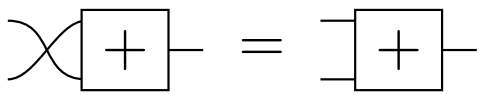
$$\frac{f : X \to Y}{Df : X \times X \to Y}$$

Cartesian differential categories: categorical structure

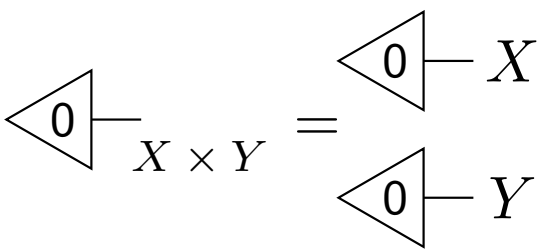
A **left additive Cartesian category** is a Cartesian category with a designated commutative monoid structure on each object, $0_X : 1 \to X$ and $+_X : X \times X \to X$:

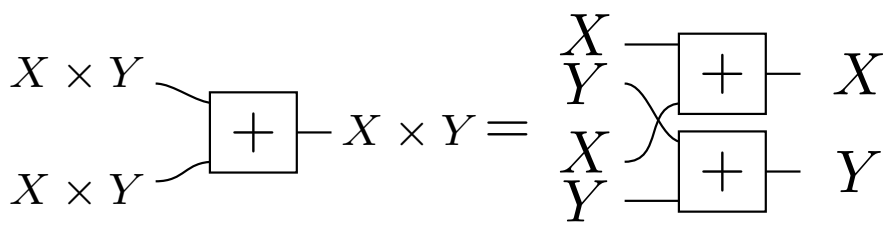






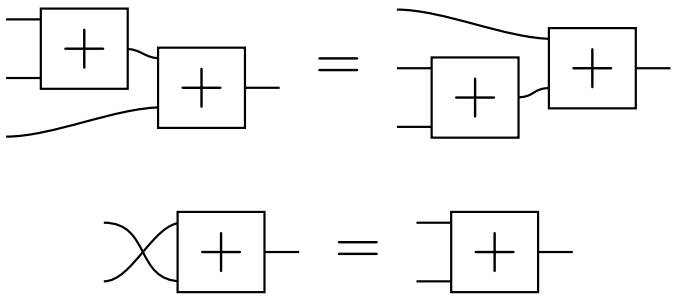
that is compatible with the Cartesian structure:



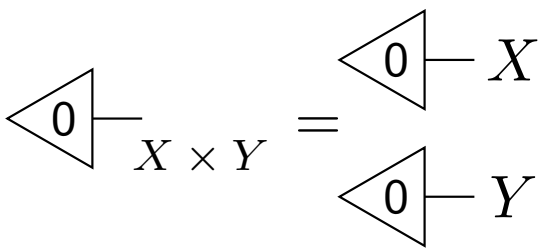


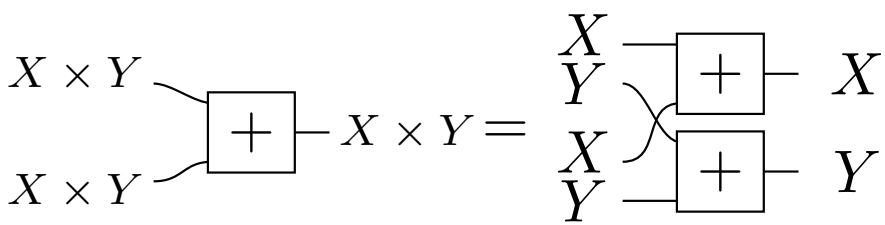
Cartesian differential categories: categorical structure

A **left additive Cartesian category** is a Cartesian category with a designated commutative monoid structure on each object, $0_X : 1 \to X$ and $+_X : X \times X \to X$:



that is compatible with the Cartesian structure:

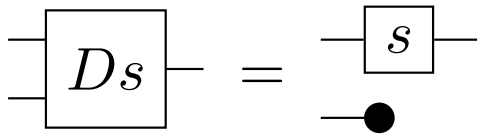




(Think of this as a weak analogue of vector space structure.)

Cartesian differential categories: differential operator

A **Cartesian differential category** is a left additive category with a differential operator satisfying seven axioms, including:

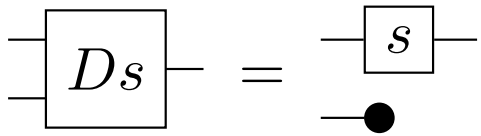
CD1. 

for $s \in \{\mathsf{id}, \sigma, !, \Delta, +, 0\}$

Idea: s is its own derivative at every point for these basic functions. (Arrows with this property are "linear" in differential category parlance.)

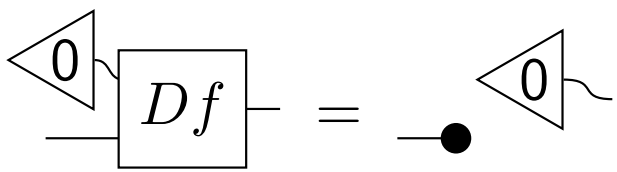
Cartesian differential categories: differential operator

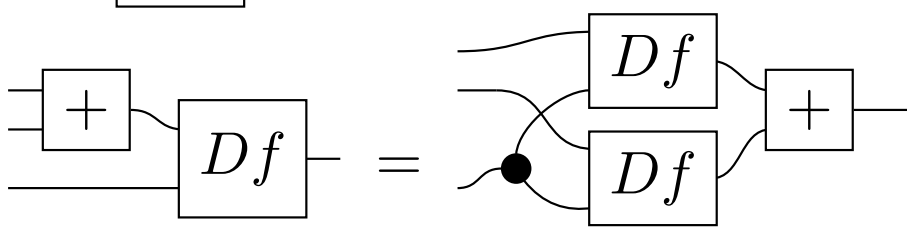
A **Cartesian differential category** is a left additive category with a differential operator satisfying seven axioms, including:

CD1. 

for $s \in \{\mathsf{id}, \sigma, !, \Delta, +, 0\}$

Idea: s is its own derivative at every point for these basic functions. (Arrows with this property are "linear" in differential category parlance.)

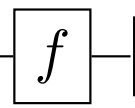
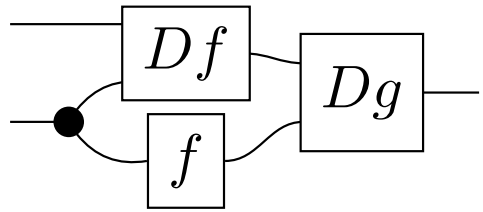
CD2. 

CD3. 

Idea: $Df(\Delta x, x)$ is linear ("additive") in the Δx argument, so $Df(0, x) = 0$ and $Df(\Delta x + \Delta x', x) = Df(\Delta x, x) + Df(\Delta x', x)$.

Cartesian differential categories: differential operator

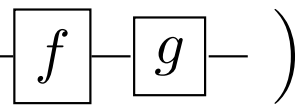
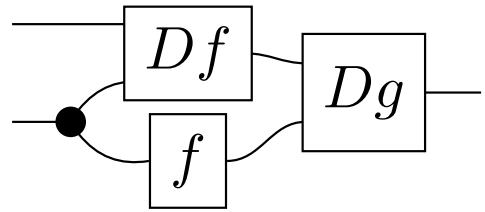
A **Cartesian differential category** is a left additive category with a differential operator satisfying seven axioms, including:

CD4. $D($  $) =$ 

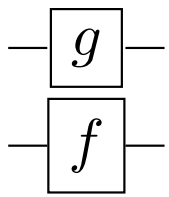
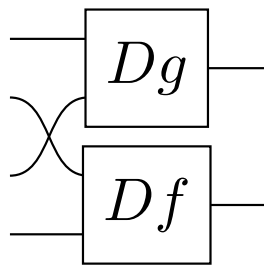
Idea: The chain rule, $D(g \circ f)(\Delta x, x) = Dg(Df(\Delta x, x), f(x))$

Cartesian differential categories: differential operator

A **Cartesian differential category** is a left additive category with a differential operator satisfying seven axioms, including:

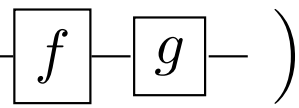
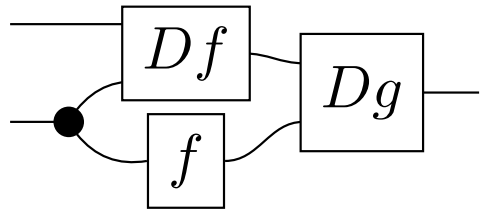
CD4. $D\Big($  $\Big) =$ 

Idea: The chain rule, $D(g \circ f)(\Delta x, x) = Dg(Df(\Delta x, x), f(x))$

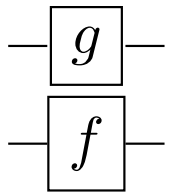
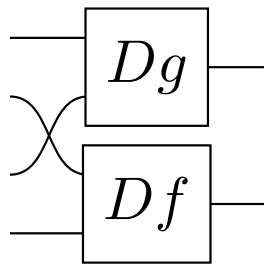
CD5. $D\Big($  $\Big) =$ 

Cartesian differential categories: differential operator

A **Cartesian differential category** is a left additive category with a differential operator satisfying seven axioms, including:

CD4. $D($  $) =$ 

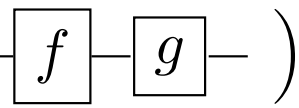
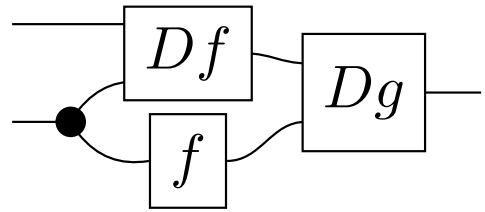
Idea: The chain rule, $D(g \circ f)(\Delta x, x) = Dg(Df(\Delta x, x), f(x))$

CD5. $D($  $) =$ 

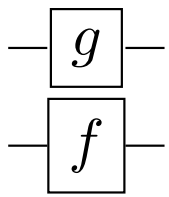
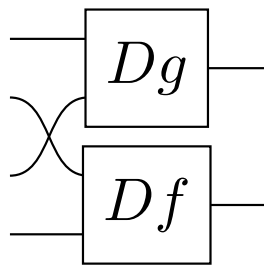
(And two axioms related to second derivatives.)

Cartesian differential categories: differential operator

A **Cartesian differential category** is a left additive category with a differential operator satisfying seven axioms, including:

CD4. $D($  $) =$ 

Idea: The chain rule, $D(g \circ f)(\Delta x, x) = Dg(Df(\Delta x, x), f(x))$

CD5. $D($  $) =$ 

(And two axioms related to second derivatives.)

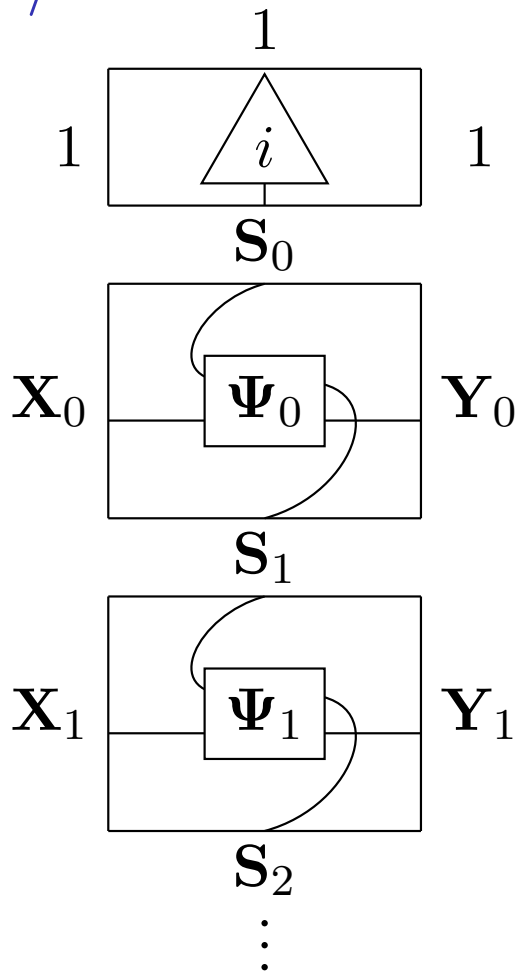
Goal: Start from a CDC of transition functions and build a CDC of Mealy machines.

Outline

Stateful **computations**—LICS'19 w/ S. Katsumata

Let $(\mathbb{C}, \times, 1)$ be a strict Cartesian category, whose morphisms we think of as stateless functions. A stateful sequence computation looks like this:

$$\boldsymbol{\Psi}_i \in \mathbb{C}(\mathbf{S}_i \times \mathbf{X}_i, \mathbf{S}_{i+1} \times \mathbf{Y}_i)$$

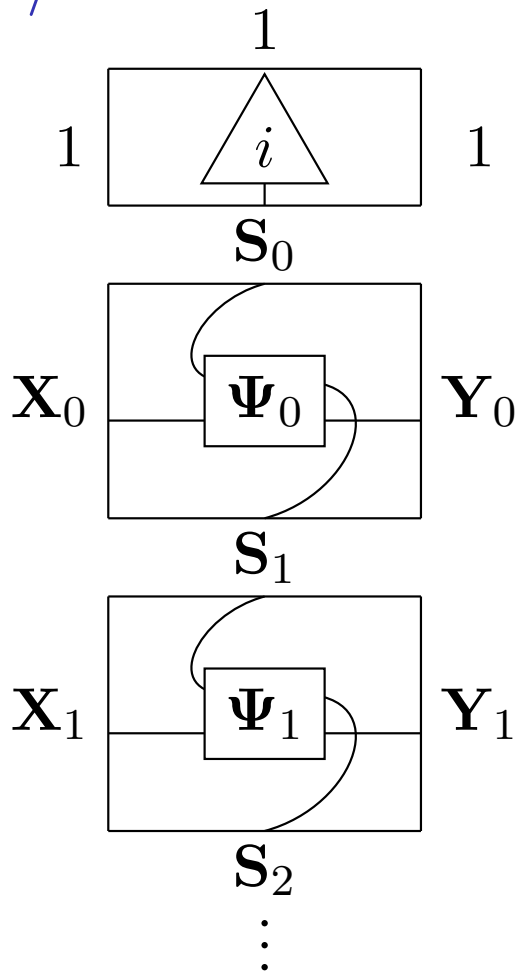


Stateful **computations**—LICS'19 w/ S. Katsumata

Let $(\mathbb{C}, \times, 1)$ be a strict Cartesian category, whose morphisms we think of as stateless functions. A stateful sequence computation looks like this:

$$\mathbf{\Psi}_i \in \mathbb{C}(\mathbf{S}_i \times \mathbf{X}_i, \mathbf{S}_{i+1} \times \mathbf{Y}_i)$$

This is a sequence of 2-cells in a double category based on $\mathbb{C}$, with a restriction on the first 2-cell.

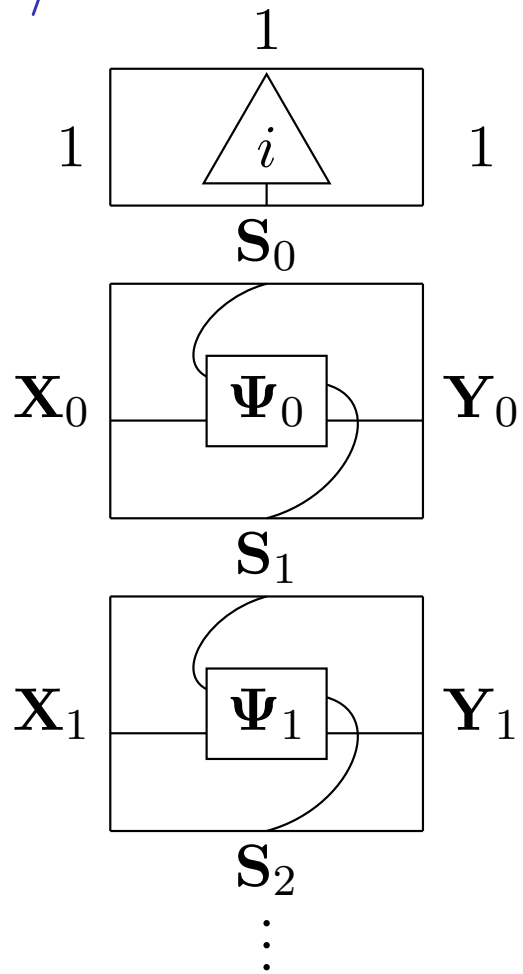


Stateful **computations**—LICS'19 w/ S. Katsumata

Let $(\mathbb{C}, \times, 1)$ be a strict Cartesian category, whose morphisms we think of as stateless functions. A stateful sequence computation looks like this:

$$\mathbf{\Psi}_i \in \mathbb{C}(\mathbf{S}_i \times \mathbf{X}_i, \mathbf{S}_{i+1} \times \mathbf{Y}_i)$$

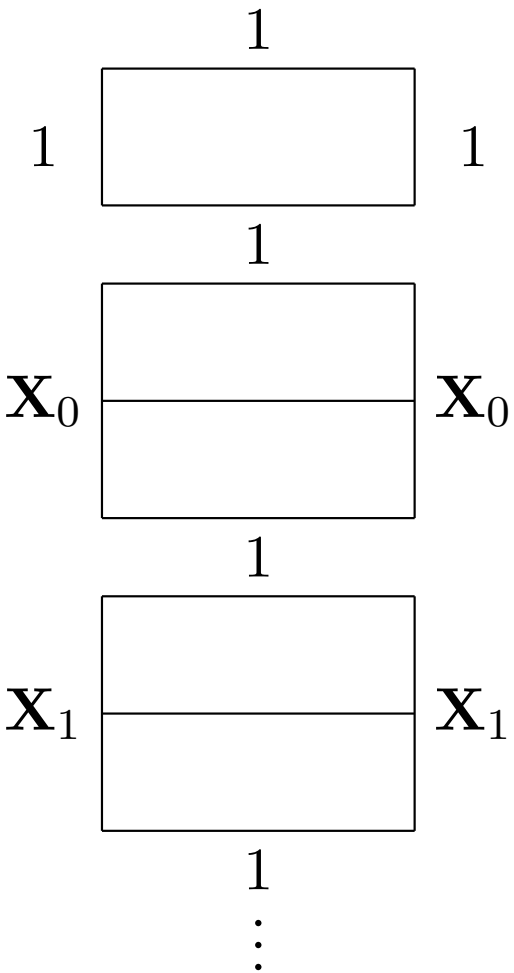
This is a sequence of 2-cells in a double category based on $\mathbb{C}$, with a restriction on the first 2-cell.

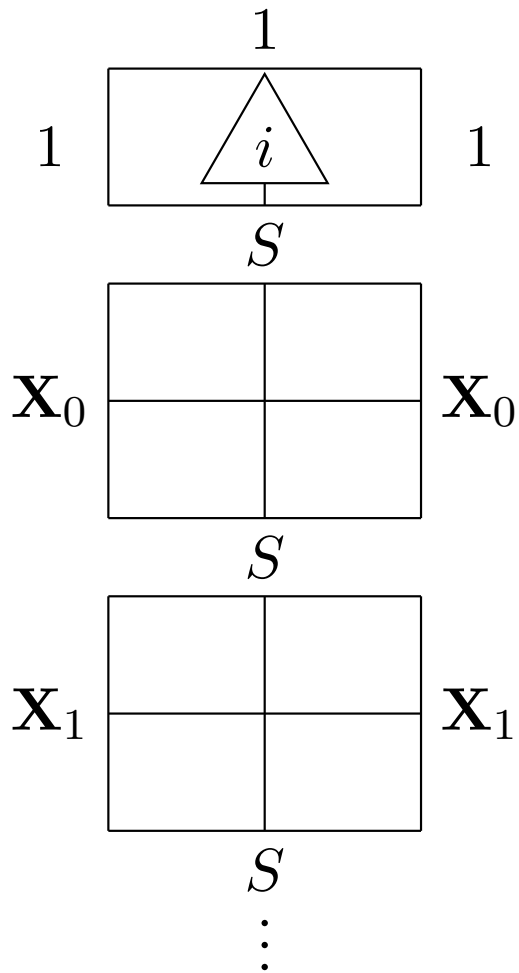


At time i, the computation executes function $\mathbf{\Psi}_i$, which takes a *value* of type $\mathbf{X}_i$ from the environment and a *state* of type $\mathbf{S}_i$ prepared by the previous step and returns a value of type $\mathbf{Y}_i$ and a state of type $\mathbf{S}_{i+1}$.

Stateful **functions**—LICS'19 w/ S. Katsumata

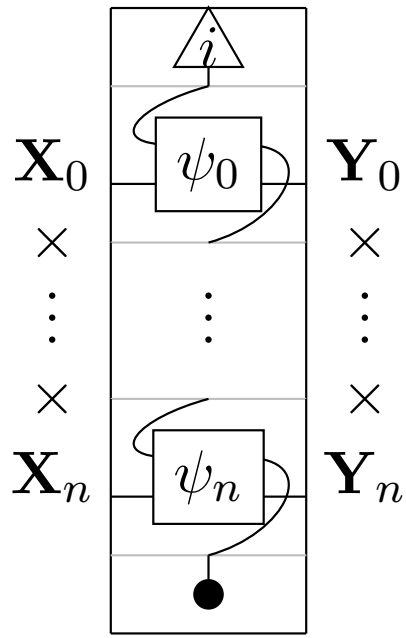
Two computation sequences might have different state spaces and still compute the same function. For example:





Stateful **functions**—LICS'19 w/ S. Katsumata

The n*th truncation* of a computation sequence is the morphism of the vertical composite of the first $n+1$ steps:



Definition

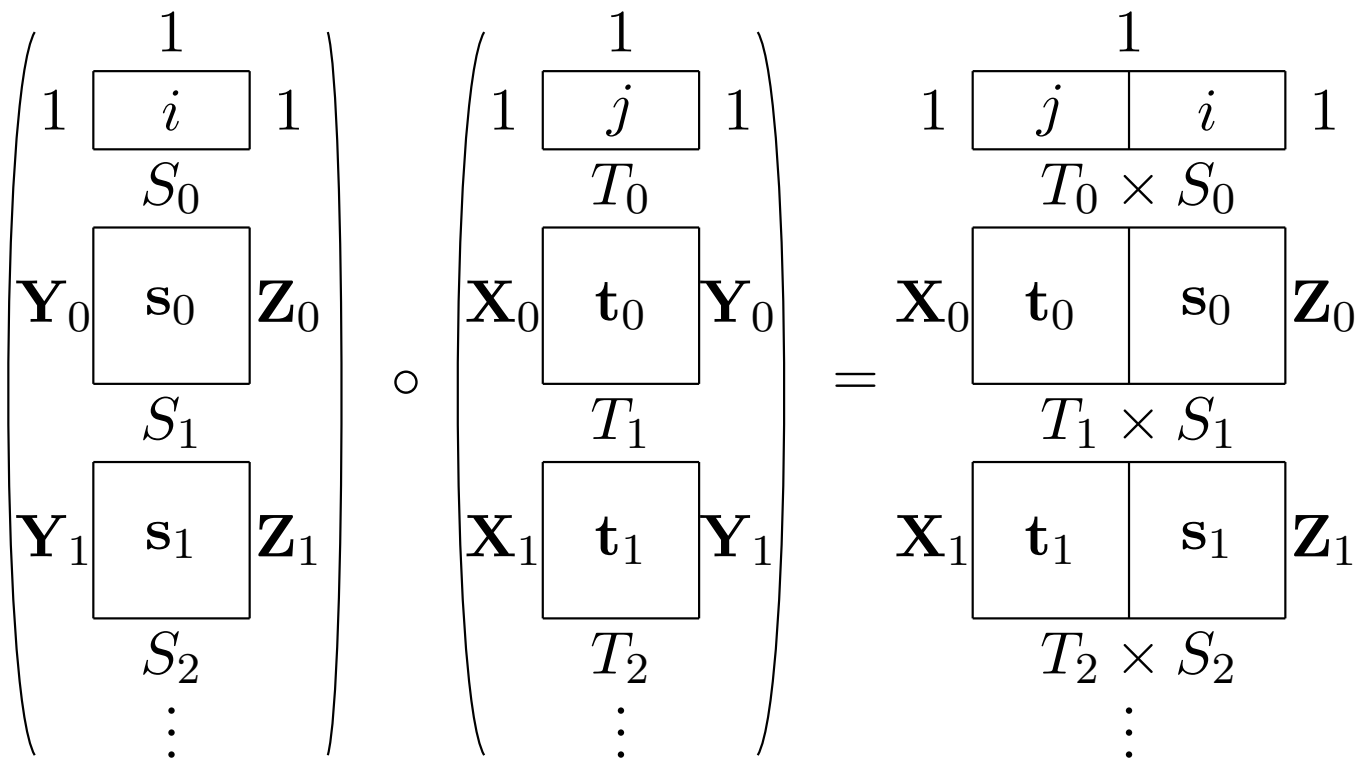
Two computation sequences are *extensionally equivalent* means they have the same nth truncation for all $n \in \mathbb{N}$. A *stateful (sequence) function* is an extensional equivalence class of computation sequences.

Stateful **functions**—LICS'19 w/ S. Katsumata

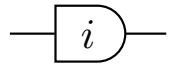
Definition

If $\mathbb{C}$ is a strict Cartesian category, then its *stateful sequence extension* is a category $\mathrm{St}(\mathbb{C})$ where

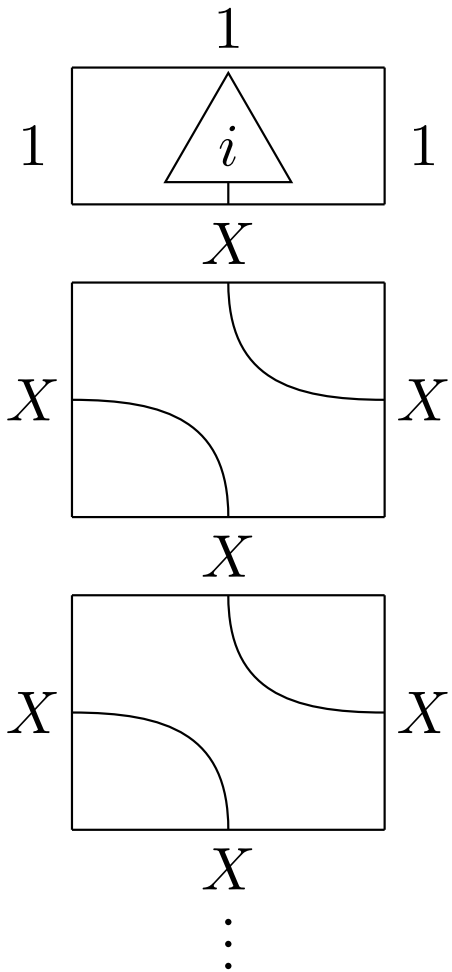
- objects are infinite sequences of objects in $\mathbb{C}$ and
- morphisms are stateful functions $\mathbf{\Psi} : \mathbf{X} \to \mathbf{Y}$.



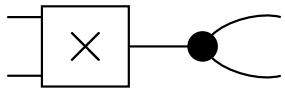
Example computation sequences

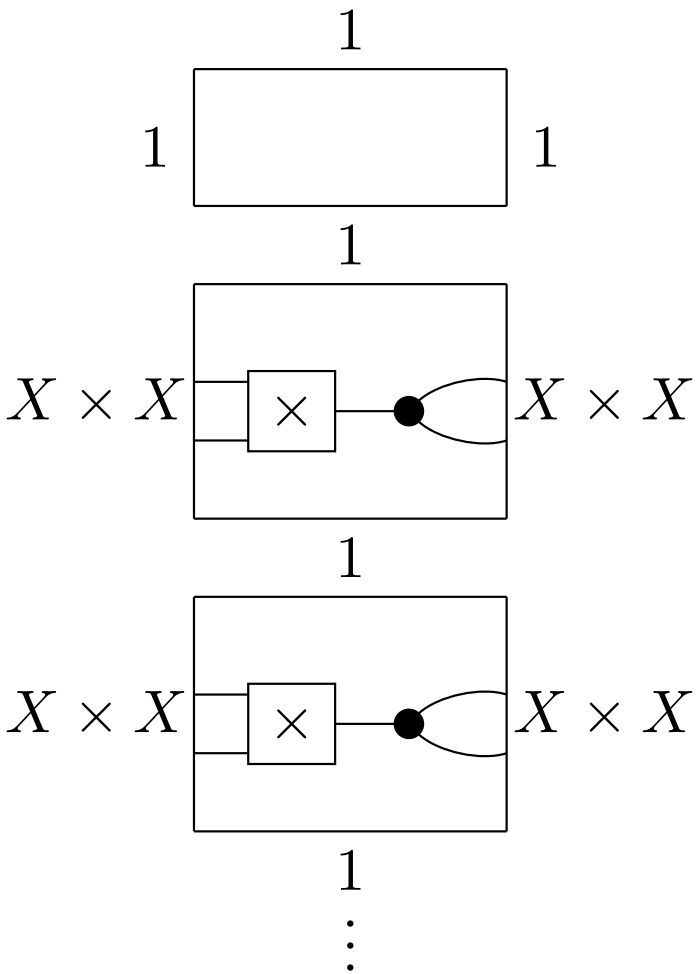
Here is 

as a computation sequence:

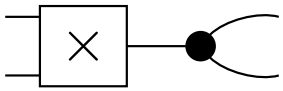


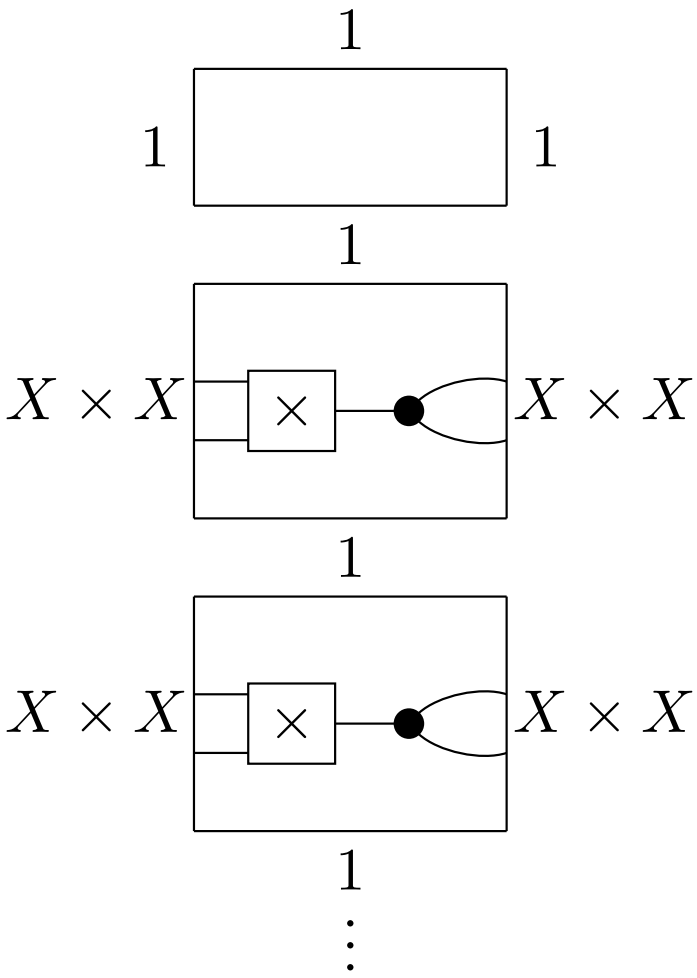
Example computation sequences

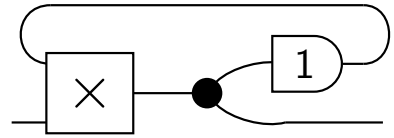
Here is  as a computation sequence:

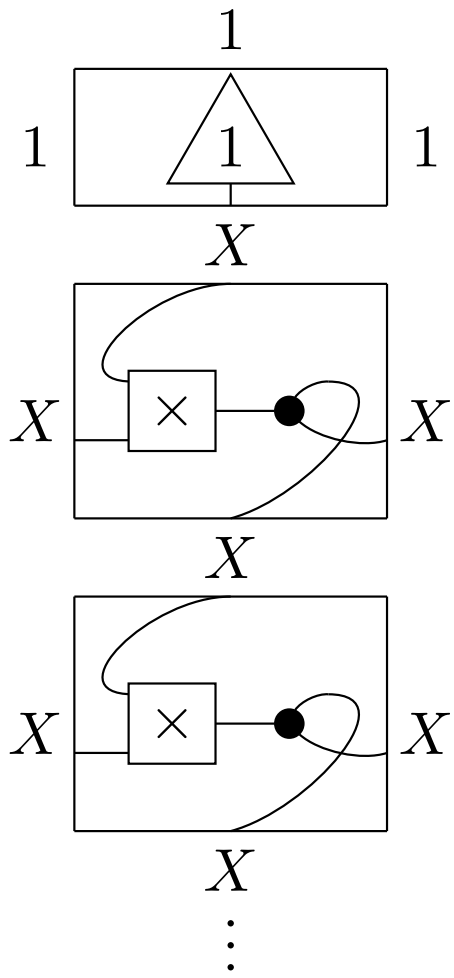


Example computation sequences

Here is  as a computation sequence:



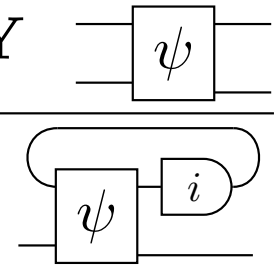
Here is  as a computation sequence:



Delayed trace

This loop-with-register is a like a trace (Joyal+ '96)—we call it a *delayed trace*. It satisfies several properties of an ordinary trace:

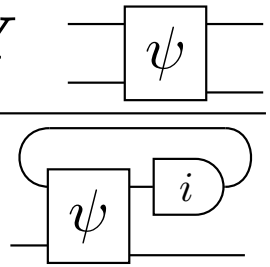
$$\frac{\psi : \mathbf{S} \times \mathbf{X} \to \mathbf{S} \times \mathbf{Y}}{dtr_i^{\mathbf{S}}(\psi) : \mathbf{X} \to \mathbf{Y}}$$



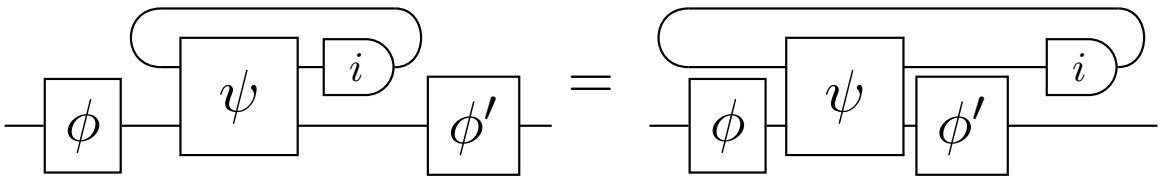
Delayed trace

This loop-with-register is a like a trace (Joyal+ '96)—we call it a *delayed trace*. It satisfies several properties of an ordinary trace:

$$\frac{\psi : \mathbf{S} \times \mathbf{X} \to \mathbf{S} \times \mathbf{Y}}{dtr_i^{\mathbf{S}}(\psi) : \mathbf{X} \to \mathbf{Y}}$$



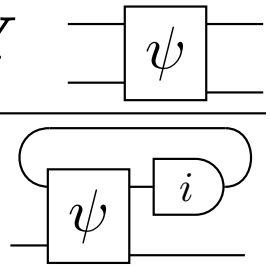
S/T naturality:



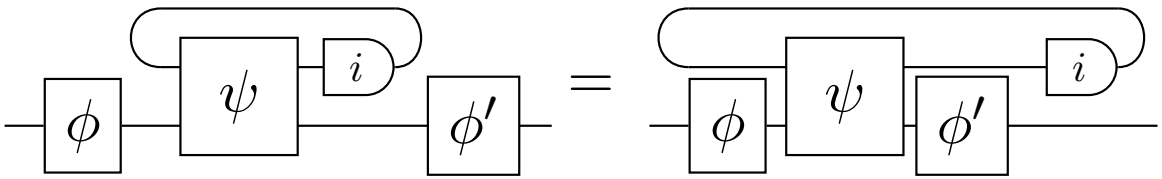
Delayed trace

This loop-with-register is a like a trace (Joyal+ '96)—we call it a *delayed trace*. It satisfies several properties of an ordinary trace:

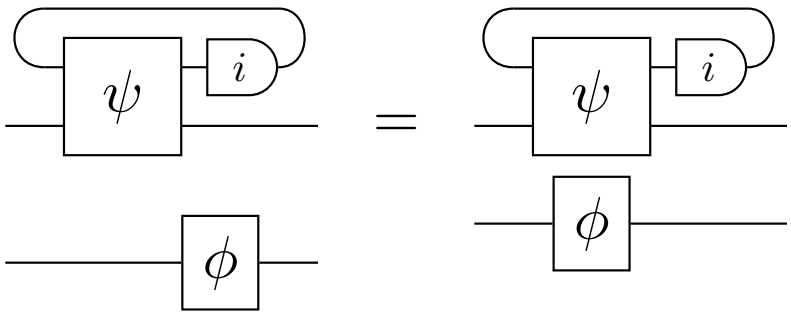
$$\frac{\psi : \mathbf{S} \times \mathbf{X} \to \mathbf{S} \times \mathbf{Y}}{dtr_i^{\mathbf{S}}(\psi) : \mathbf{X} \to \mathbf{Y}}$$



S/T naturality:



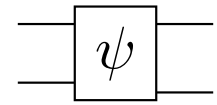
Superposition:

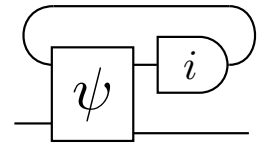


Delayed trace

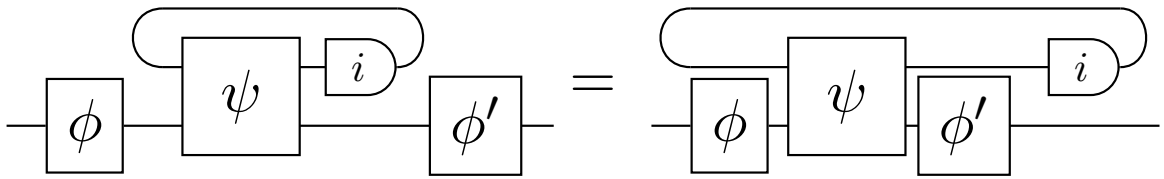
This loop-with-register is a like a trace (Joyal+ '96)—we call it a *delayed trace*. It satisfies several properties of an ordinary trace:

$$\frac{\psi : \mathbf{S} \times \mathbf{X} \to \mathbf{S} \times \mathbf{Y}}{dtr_i^{\mathbf{S}}(\psi) : \mathbf{X} \to \mathbf{Y}}$$

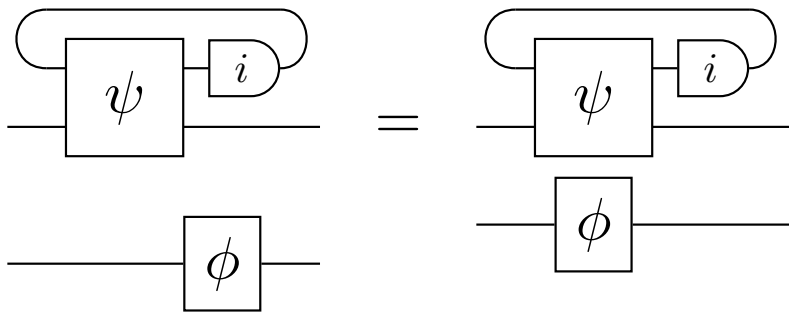




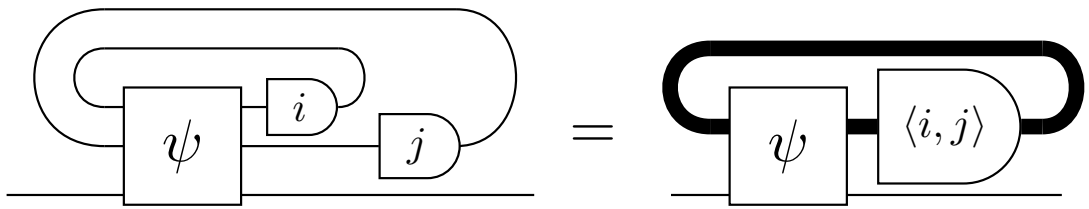
S/T naturality:



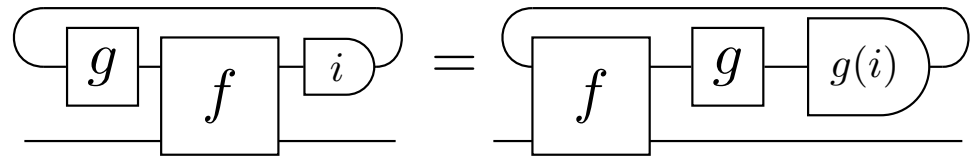
Superposition:

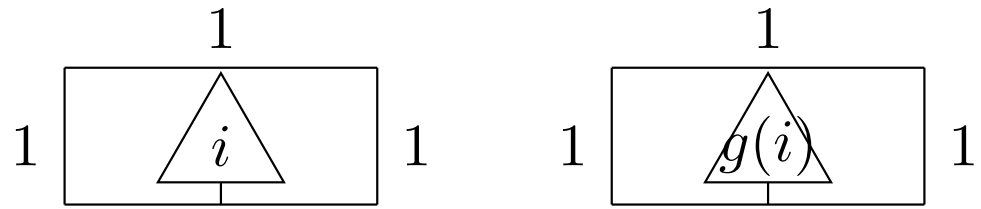


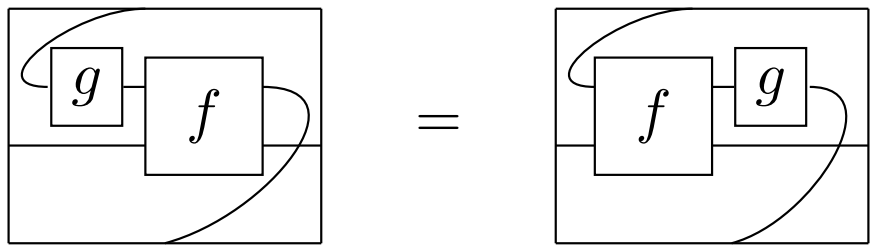
Vanishing $\times$:

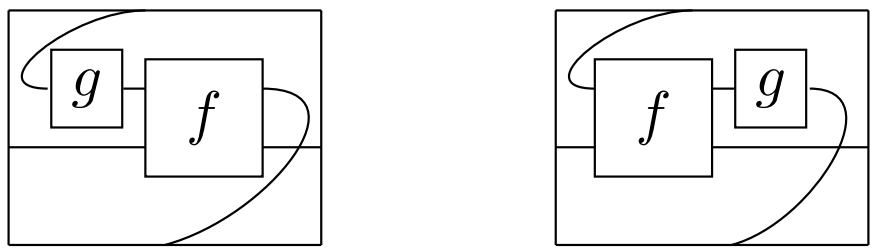


~~Dinaturality~~ → retiming



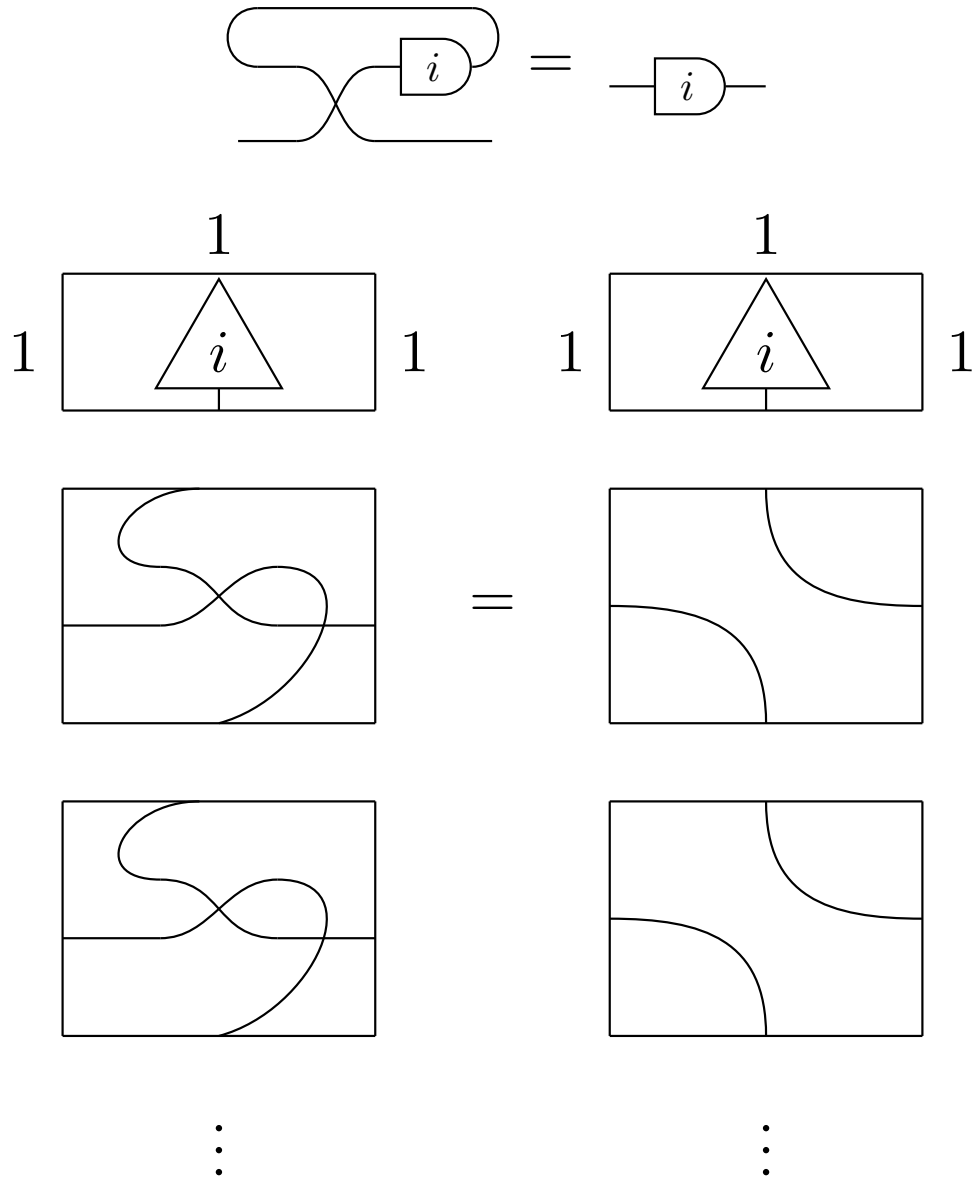






⋮ ⋮

~~Yanking~~ → delay

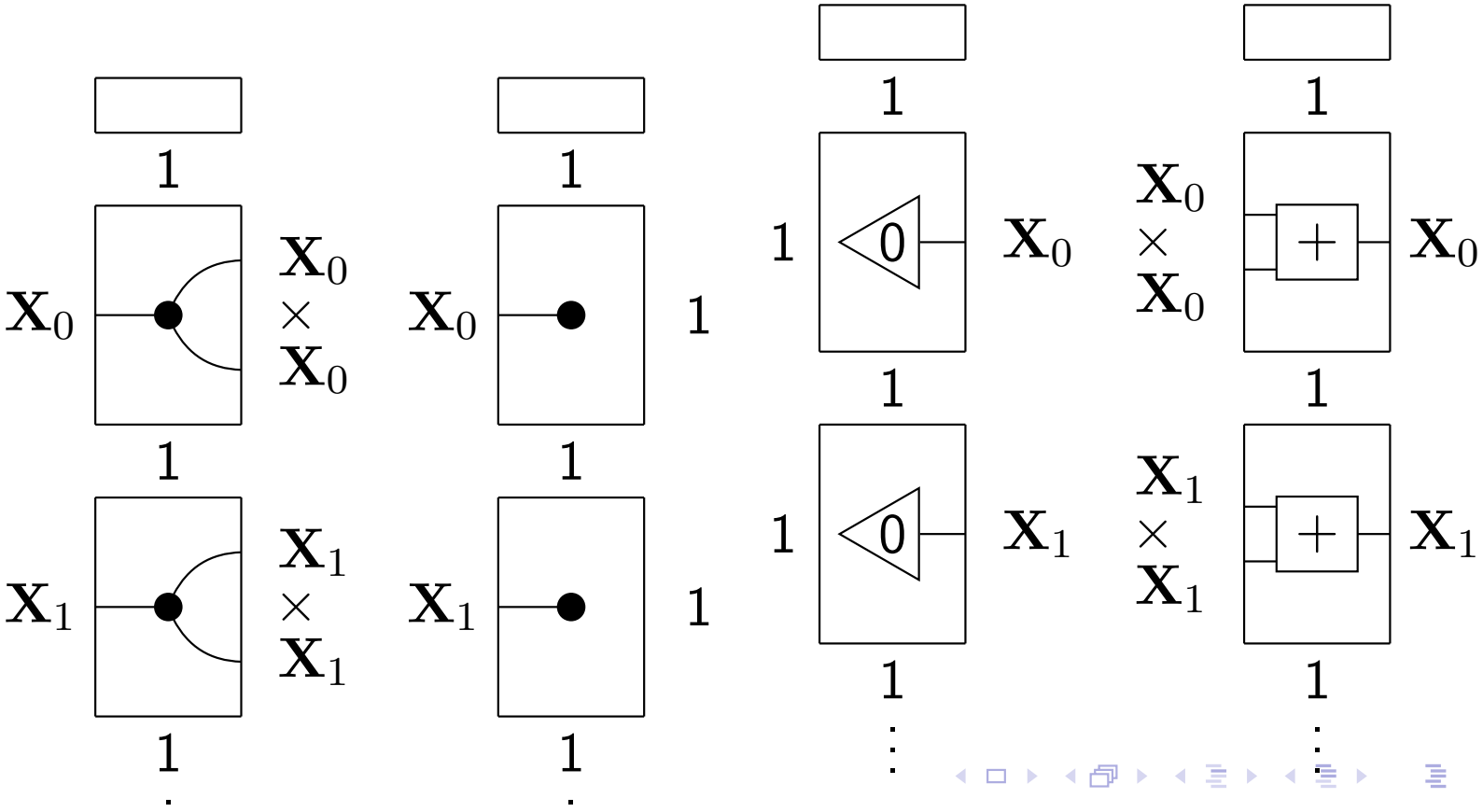


$\mathrm{St}(-)$ preserves a lot of structure

Theorem

$\mathrm{St}(\mathbb{C})$ is a Cartesian category.
If $\mathbb{C}$ is a Cartesian left additive category, so is $\mathrm{St}(\mathbb{C})$.
If $\mathbb{C}$ is a Cartesian differential category, so is $\mathrm{St}(\mathbb{C})$.

The first two are easy:



Sanity checks

Proposition: $\mathbb{C}$ embeds in $\mathrm{St}(\mathbb{C})$

There is a finite-product preserving functor $H : \mathbb{C} \to \mathrm{St}(\mathbb{C})$. It takes A to $(A, A, A, \ldots)$ and f to the stateless sequence executing f at each step.

Sanity checks

Proposition: $\mathbb{C}$ embeds in $\mathrm{St}(\mathbb{C})$

There is a finite-product preserving functor $H : \mathbb{C} \to \mathrm{St}(\mathbb{C})$. It takes A to $(A, A, A, \ldots)$ and f to the stateless sequence executing f at each step.

Proposition: Normal forms for $\mathrm{St}(\mathbb{C})$

Every $\Psi \in \mathrm{St}(\mathbb{C})(\mathbf{A}, \mathbf{B})$ can be written as $\Psi = dtr_i^{\mathbf{S}}(\psi)$ where ψ has a stateless representative and $i : 1 \to S$.

Sanity checks

Proposition: $\mathbb{C}$ embeds in $\mathrm{St}(\mathbb{C})$

There is a finite-product preserving functor $H : \mathbb{C} \to \mathrm{St}(\mathbb{C})$. It takes A to $(A, A, A, \ldots)$ and f to the stateless sequence executing f at each step.

Proposition: Normal forms for $\mathrm{St}(\mathbb{C})$

Every $\Psi \in \mathrm{St}(\mathbb{C})(\mathbf{A}, \mathbf{B})$ can be written as $\Psi = dtr_i^{\mathbf{S}}(\psi)$ where ψ has a stateless representative and $i : 1 \to S$.

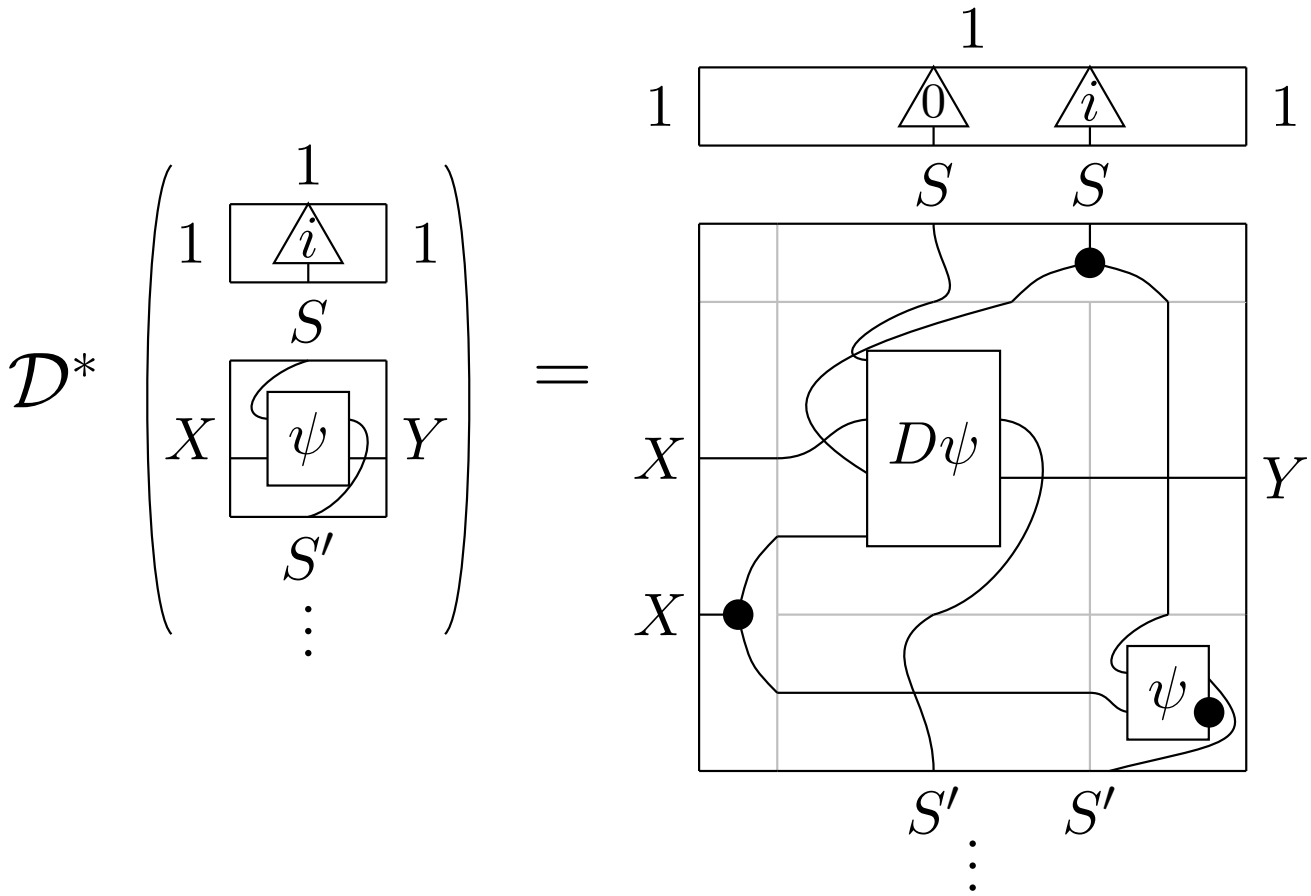
Adding state to computations is common:

1. Bicategorically—Katis, Sabadini, & Walters '97
2. Digital circuits—Ghica & Jung, '16
3. Signal flow graphs—Bonchi, Sobociński, & Zanasi, '14

Outline

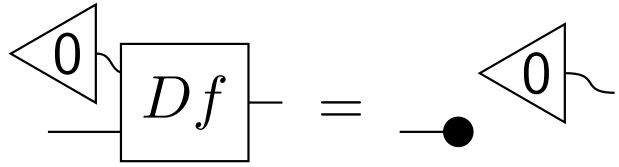
Differentiation for stateful functions

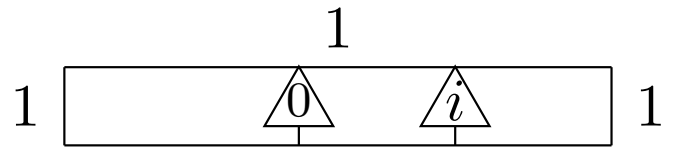
Let $\mathbb{C}$ be Cartesian differential with differential operator D. The following is a Cartesian differential operator on $\mathrm{St}(\mathbb{C})$:

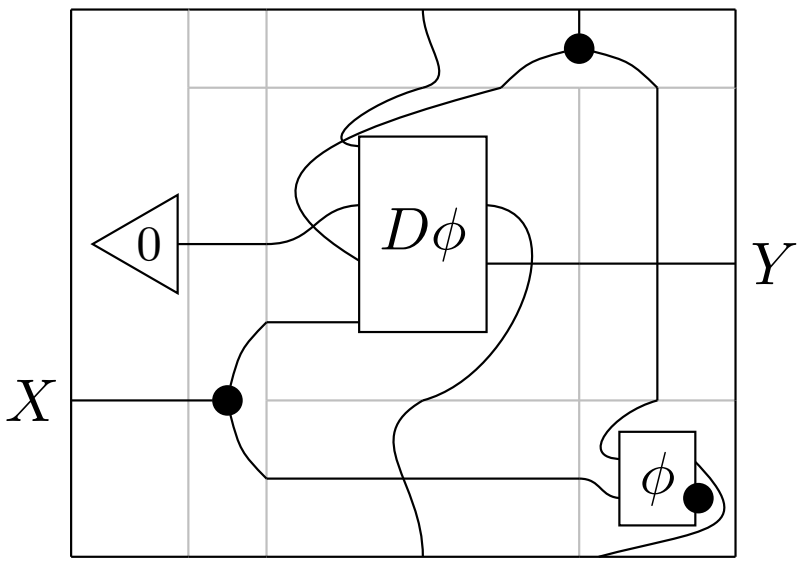


$\mathcal{D}^*$ is a Cartesian differential operator

Proof idea. For CD2:

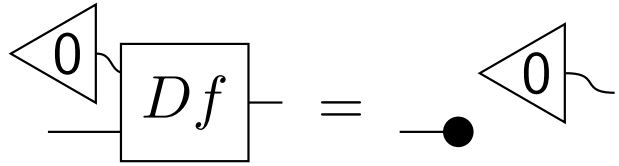


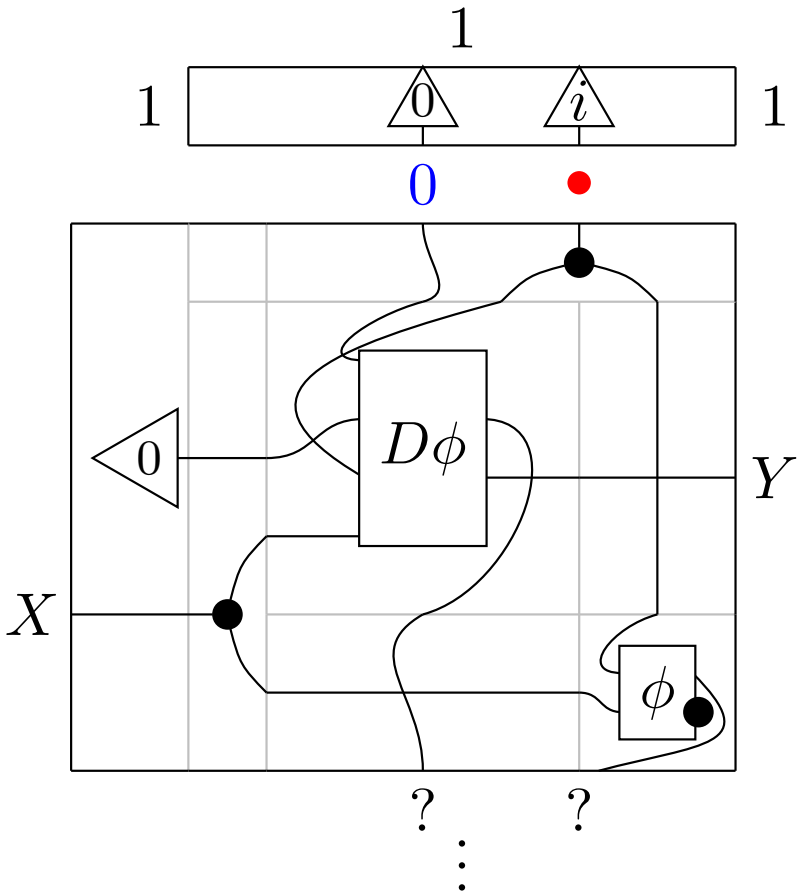




$\mathcal{D}^*$ is a Cartesian differential operator

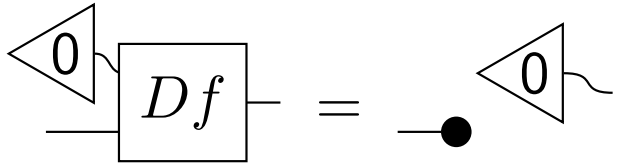
Proof idea. For CD2:

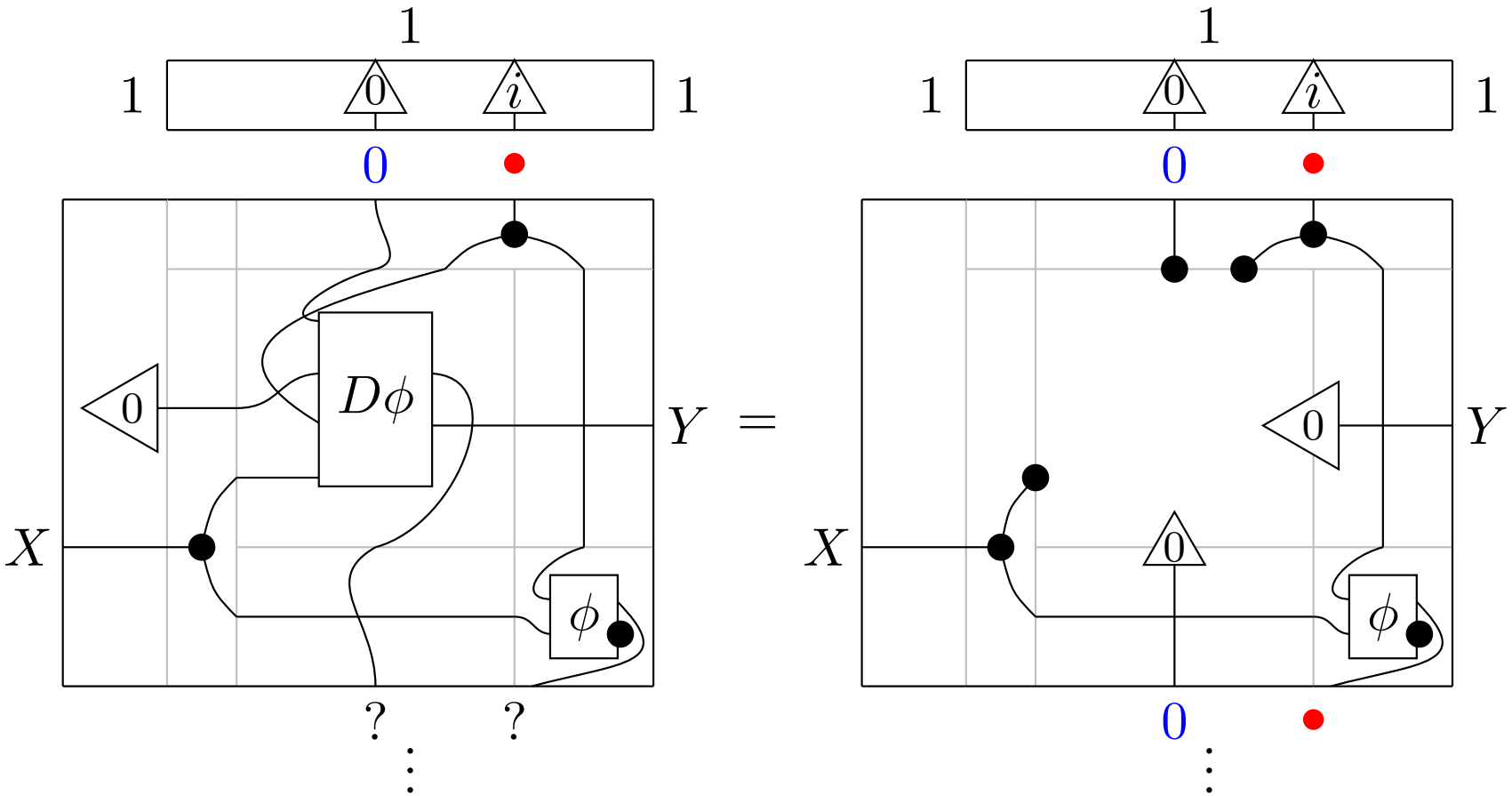




$\mathcal{D}^*$ is a Cartesian differential operator

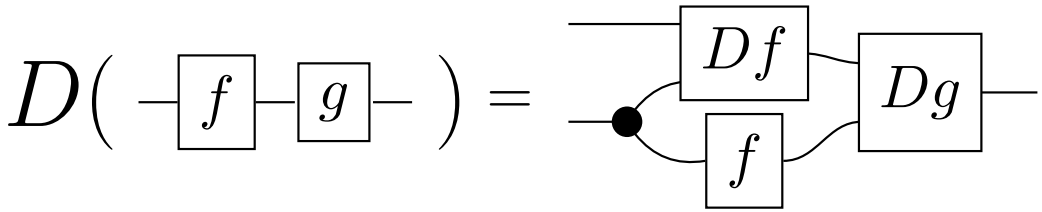
Proof idea. For CD2:

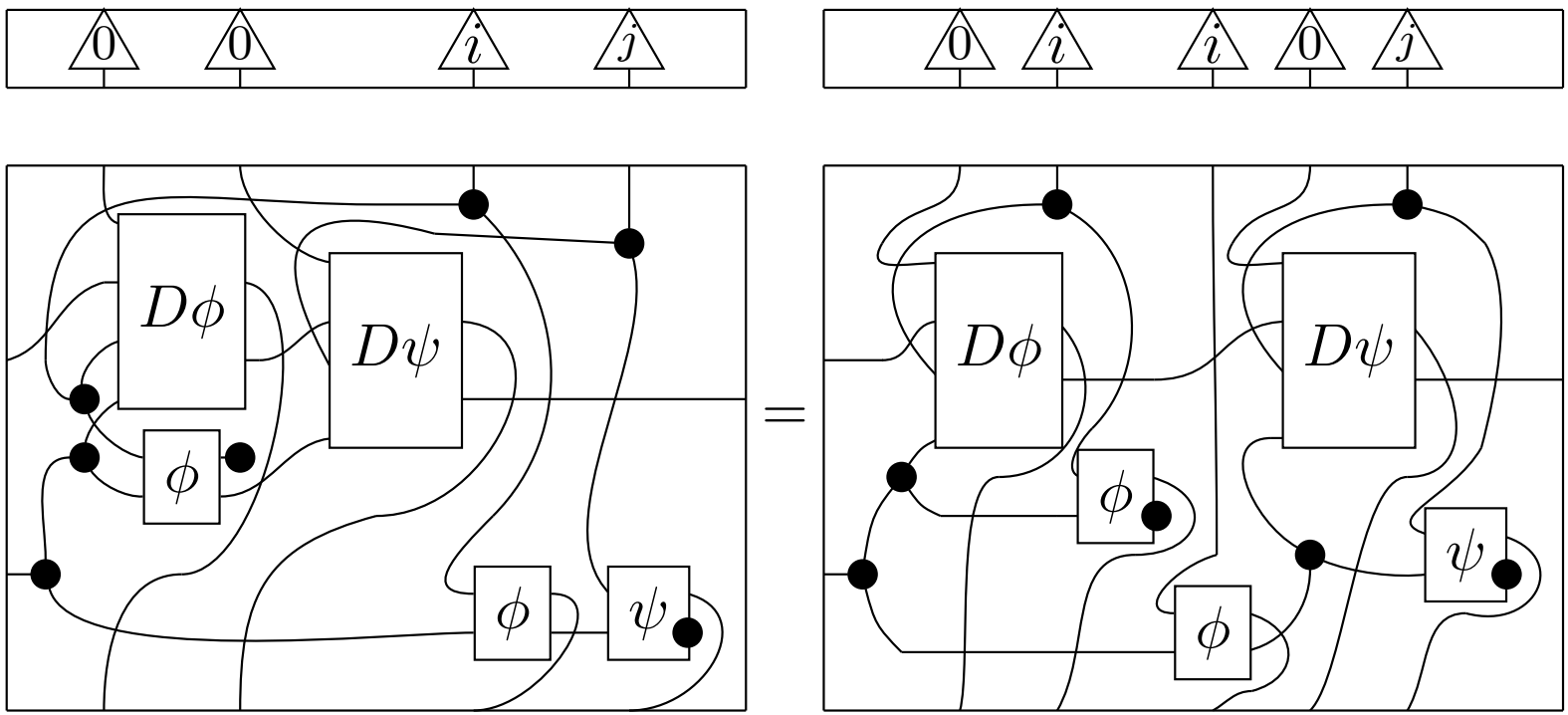




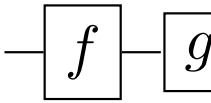
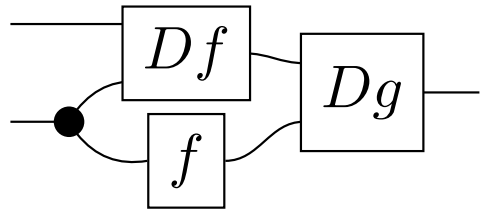
$\mathcal{D}^*$ is a Cartesian differential operator

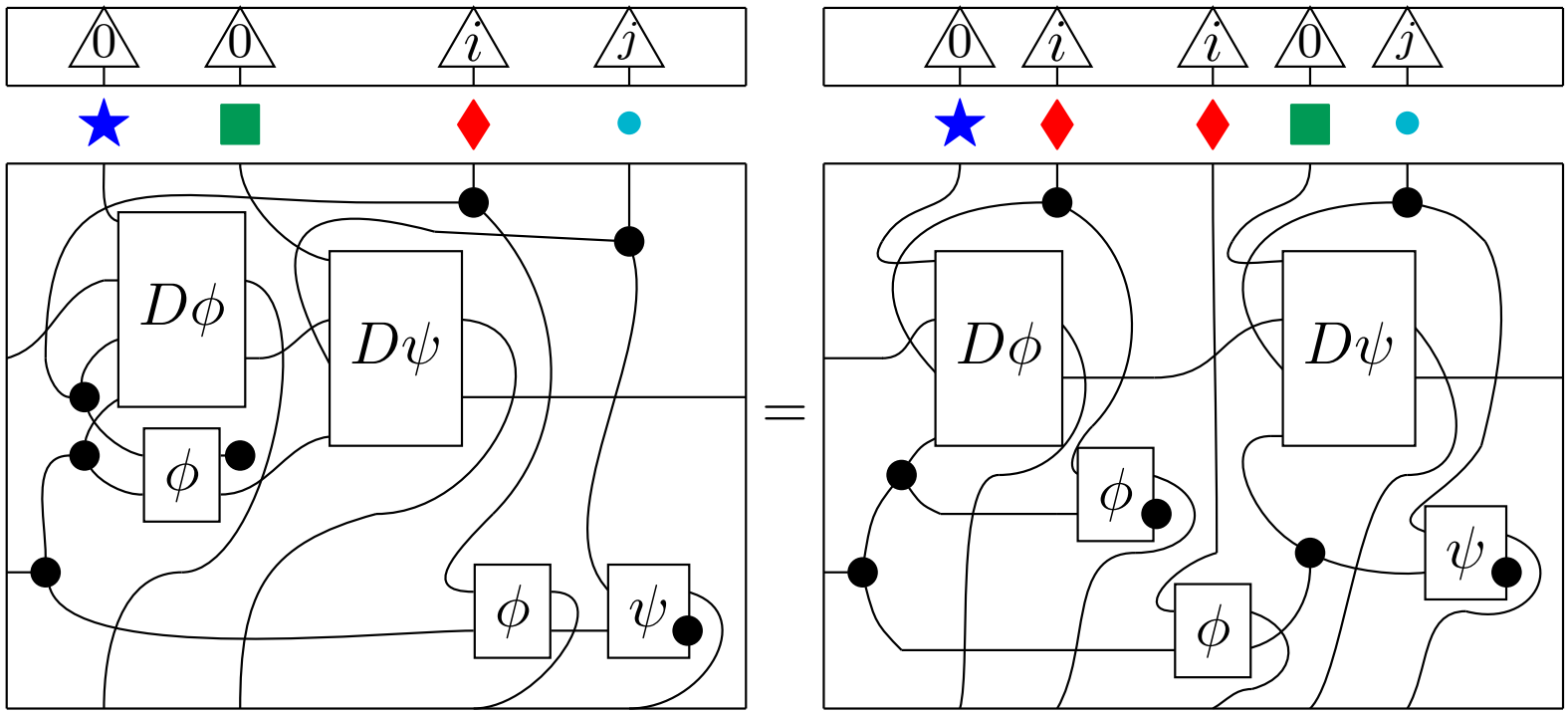
Proof idea. For CD4:



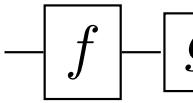
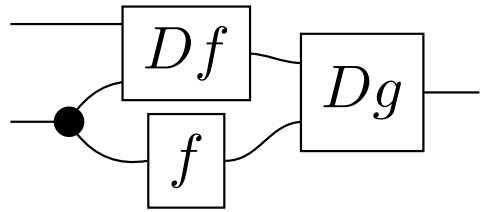


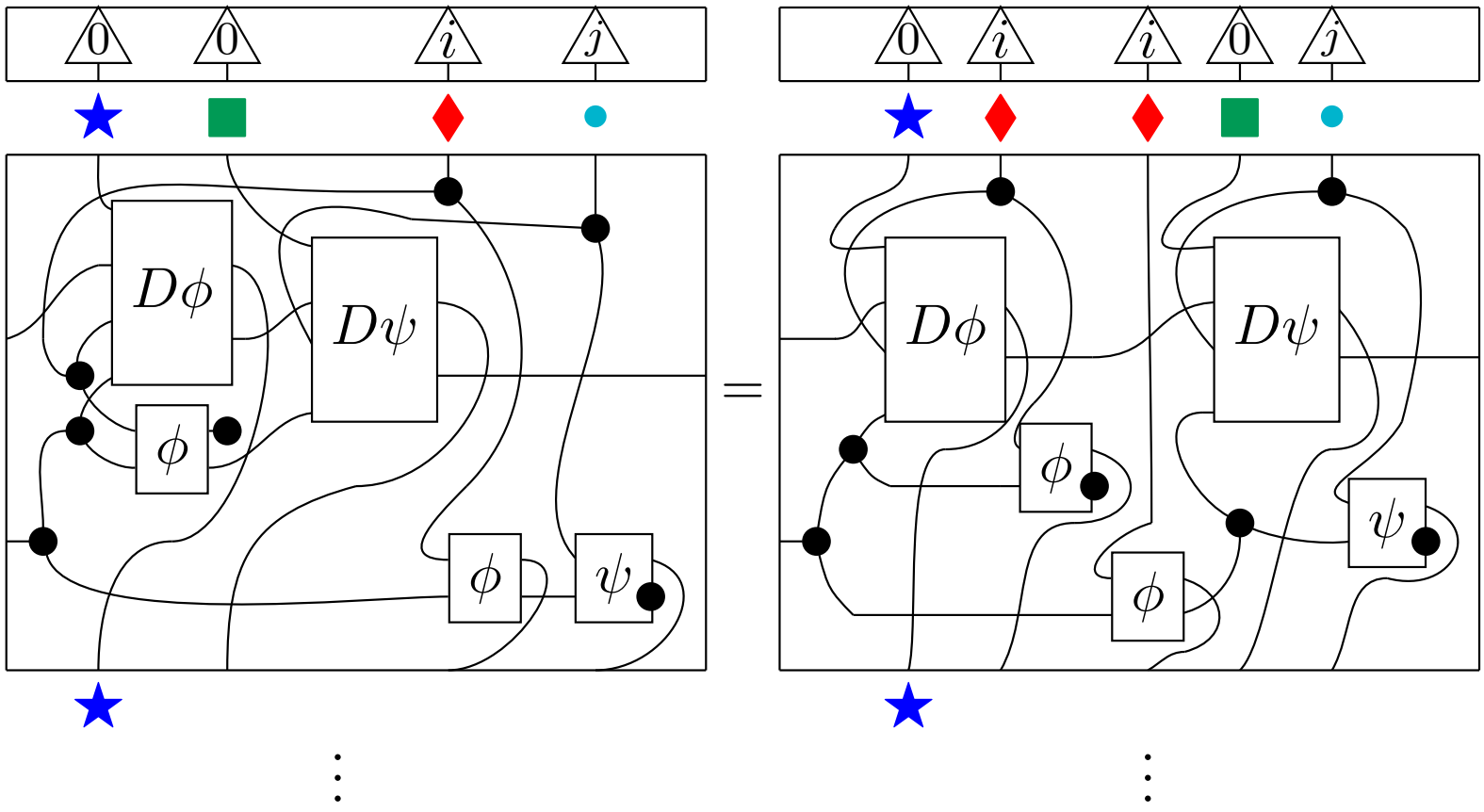
$\mathcal{D}^*$ is a Cartesian differential operator

Proof idea. For CD4: $D($  $) =$ 

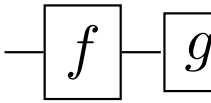
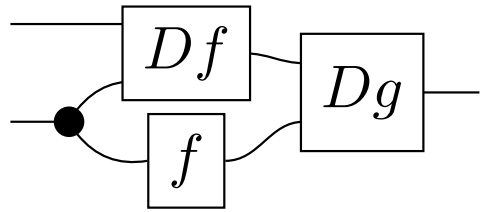


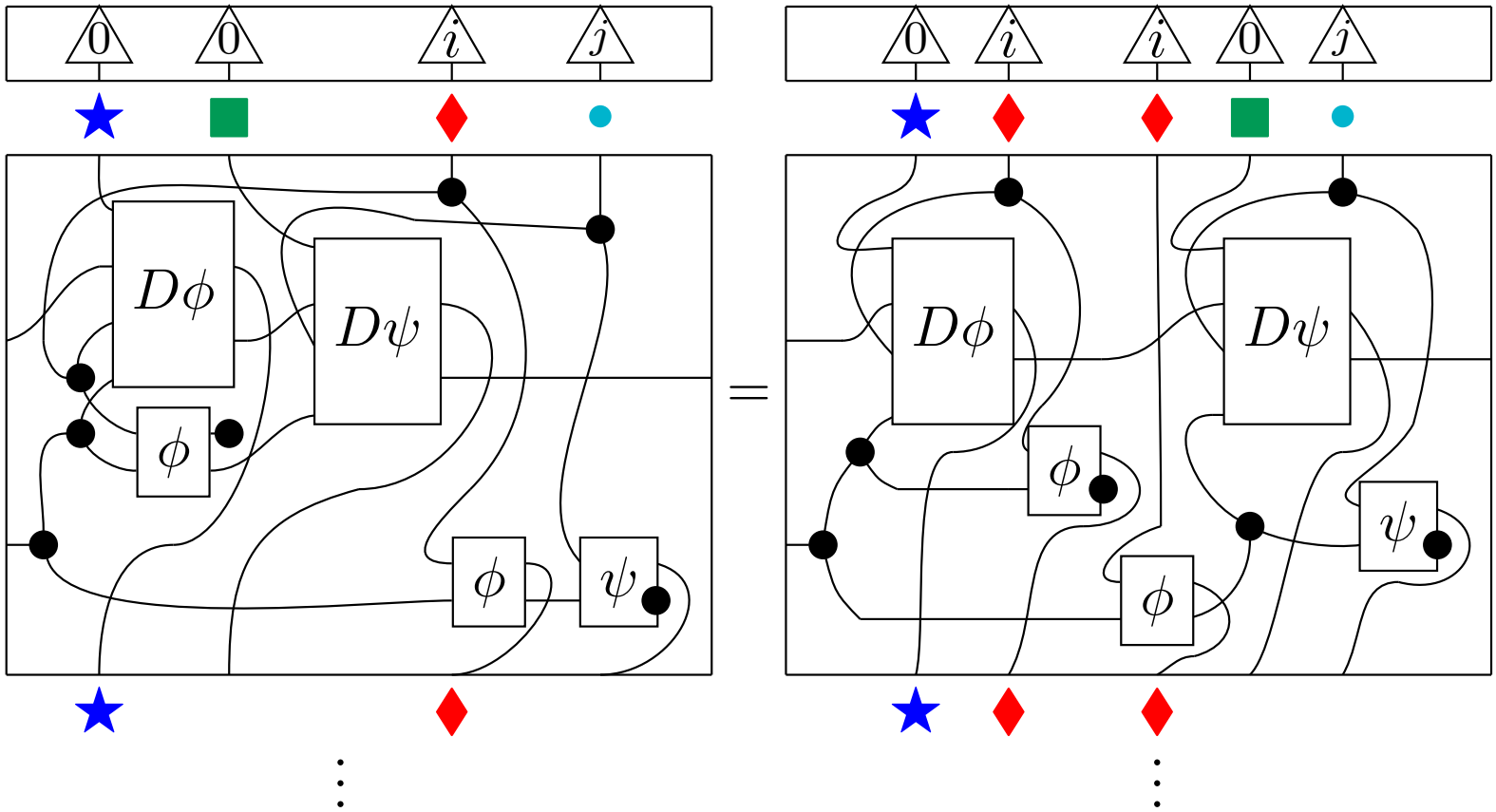
$\mathcal{D}^*$ is a Cartesian differential operator

Proof idea. For CD4: $D($  $) =$ 

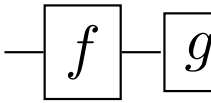
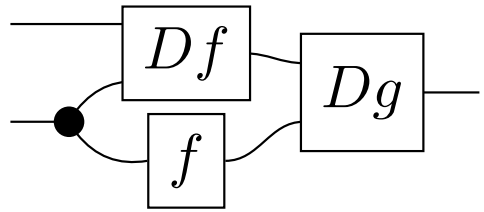


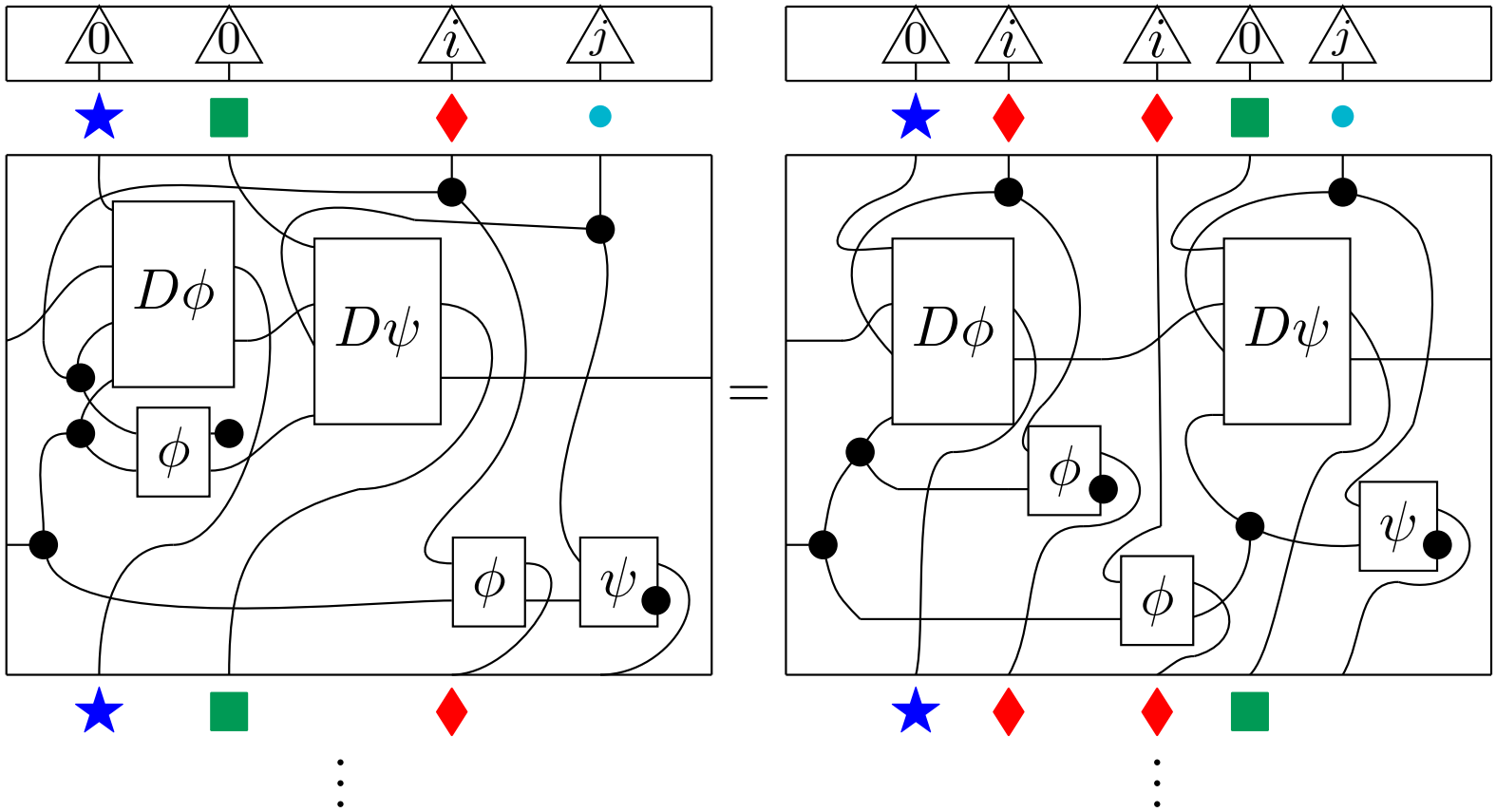
$\mathcal{D}^*$ is a Cartesian differential operator

Proof idea. For CD4: $D($  $) =$ 

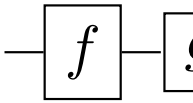
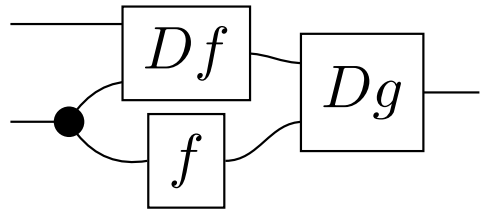


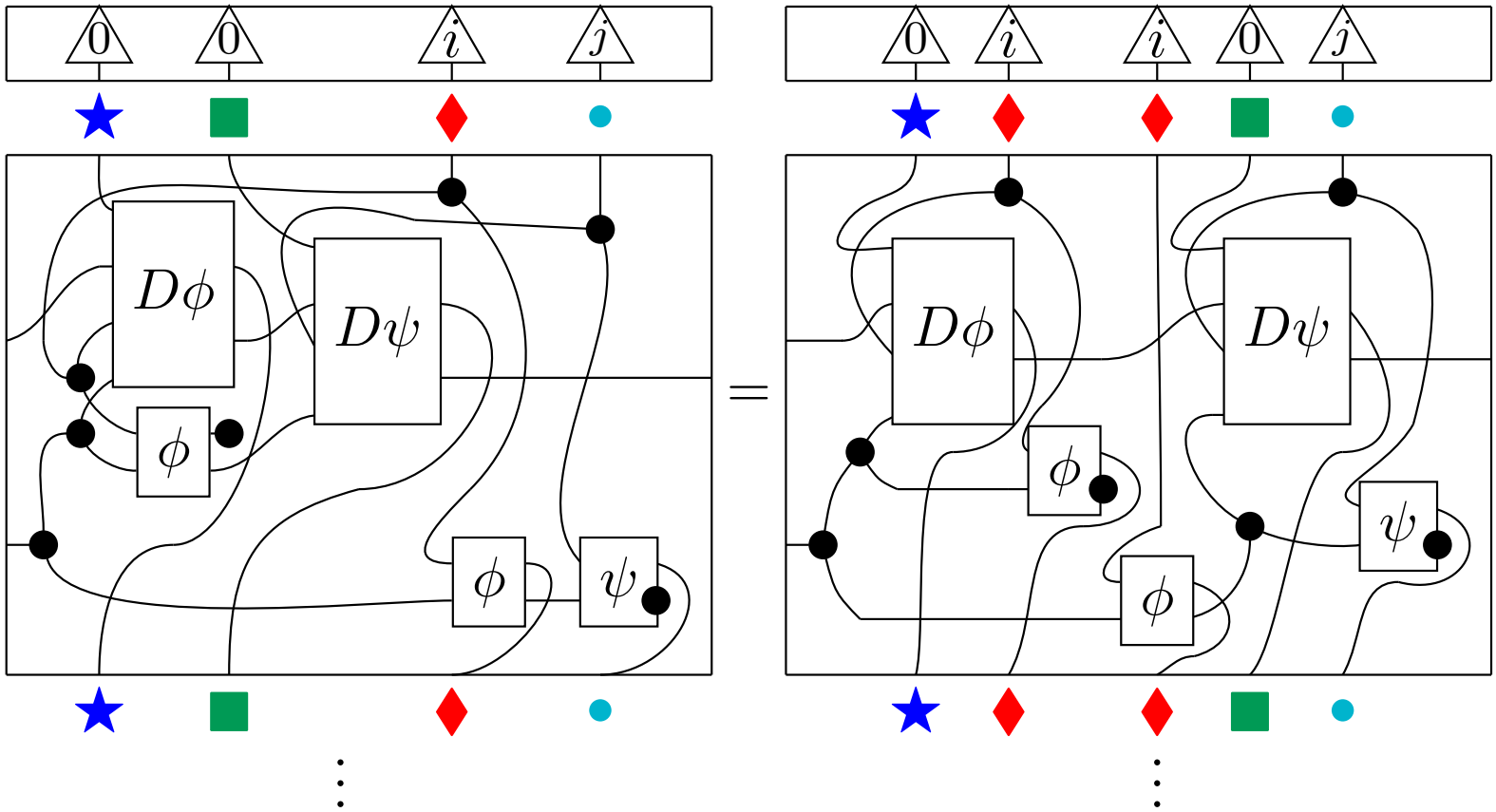
$\mathcal{D}^*$ is a Cartesian differential operator

Proof idea. For CD4: $D($  $) =$ 



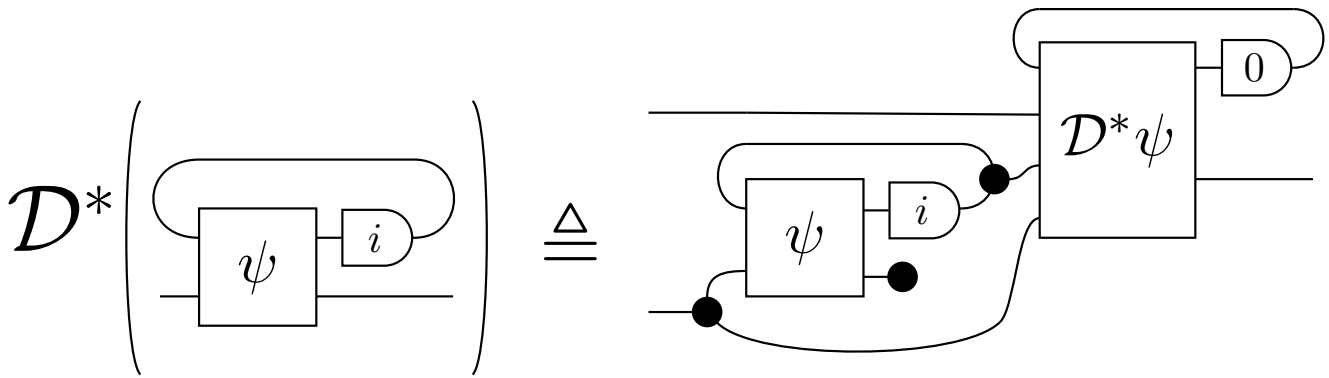
$\mathcal{D}^*$ is a Cartesian differential operator

Proof idea. For CD4: $D($  $) =$ 



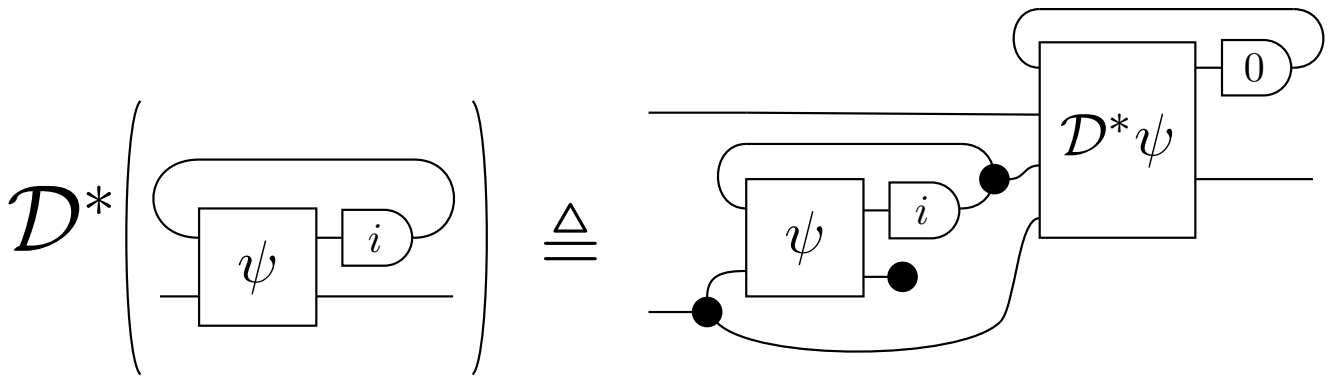
Derivatives of RNNs

When the transition function is the same at each step, we get



Derivatives of RNNs

When the transition function is the same at each step, we get



Future directions

Several obstacles prevent us from applying these ideas in practice right away:

1. Need non-smooth, partly differentiable, or partial functions
2. Need a transpose for computational efficiency

Future directions

Several obstacles prevent us from applying these ideas in practice right away:

1. Need non-smooth, partly differentiable, or partial functions
2. Need a transpose for computational efficiency

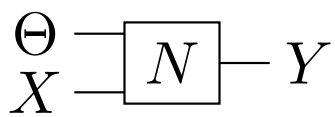
There are some questions related to the theory that we would like to understand better:

1. Can we add more advanced differential category structure (restriction, join, tangent, reverse, ...)?
2. Categorical properties of $\mathrm{St}(-)$?
3. Basic results for delayed trace categories?
4. Other data shapes (trees, distributions, ...)?

Outline

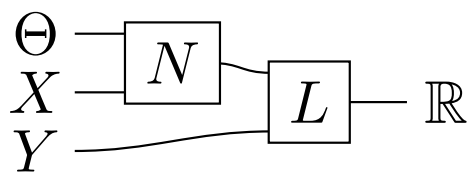
An introduction to training neural networks

A *feedforward neural network* (N) is a function with two kinds of inputs: data inputs (from X) and parameters (from Θ).



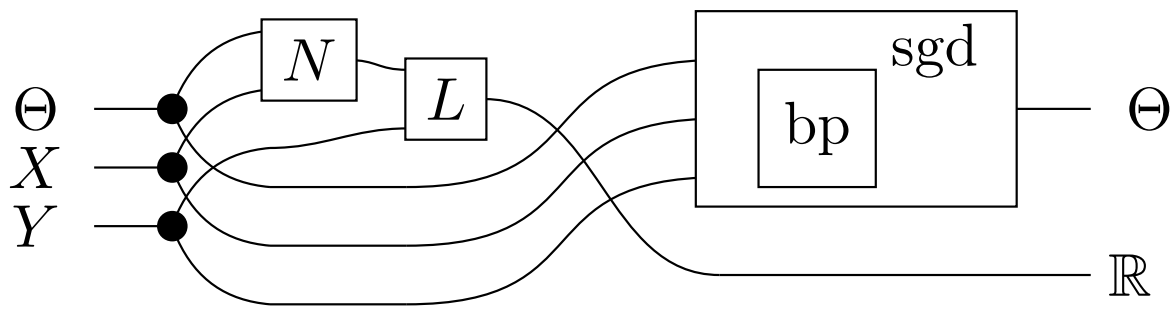
An introduction to training neural networks

A *feedforward neural network* (N) is a function with two kinds of inputs: data inputs (from X) and parameters (from Θ). The network's performance is measured by a loss function L using data from $X \times Y$.



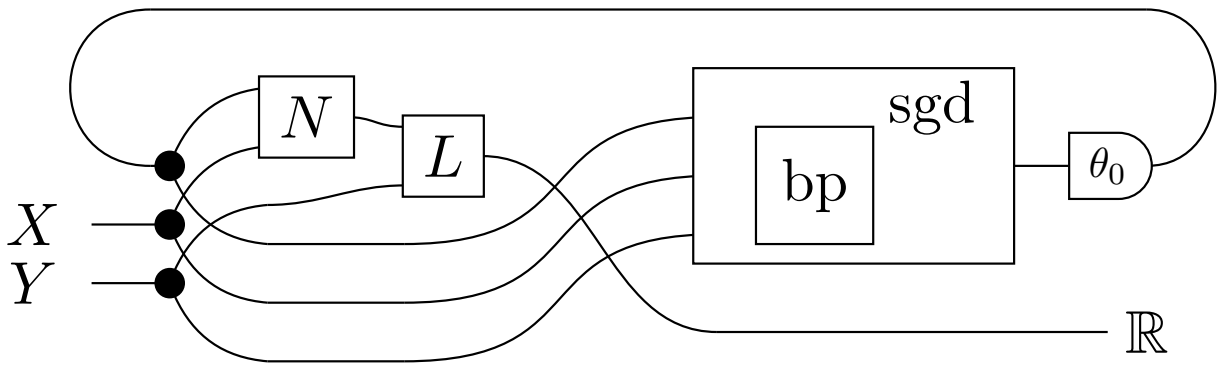
An introduction to training neural networks

A *feedforward neural network* (N) is a function with two kinds of inputs: data inputs (from X) and parameters (from Θ). The network's performance is measured by a loss function L using data from $X \times Y$. A *training algorithm* updates the parameter value, often using "backpropagation" to find the gradient of $L \circ N$ wrt Θ.



An introduction to training neural networks

A *feedforward neural network* (N) is a function with two kinds of inputs: data inputs (from X) and parameters (from Θ). The network's performance is measured by a loss function L using data from $X \times Y$. A *training algorithm* updates the parameter value, often using "backpropagation" to find the gradient of $L \circ N$ wrt Θ.



Where are we going?

Simple training scenario

Here's a "hello world"-type training scenario. We start with a random dense layer $\mathbb{R}^{10} \to \mathbb{R}^4$, given by

$$N(x; A^*, b^*) = \tanh(A^* x + b^*)$$

Simple training scenario

Here's a "hello world"-type training scenario. We start with a random dense layer $\mathbb{R}^{10} \to \mathbb{R}^4$, given by

$$N(x; A^*, b^*) = \tanh(A^* x + b^*)$$

We then randomly sample 200 vectors $x_i \in \mathbb{R}^{10}$ and find the corresponding $y_i \triangleq N(x_i; A^*, b^*) \in \mathbb{R}^4$.

Simple training scenario

Here's a "hello world"-type training scenario. We start with a random dense layer $\mathbb{R}^{10} \to \mathbb{R}^4$, given by

$$N(x; A^*, b^*) = \tanh(A^* x + b^*)$$

We then randomly sample 200 vectors $x_i \in \mathbb{R}^{10}$ and find the corresponding $y_i \triangleq N(x_i; A^*, b^*) \in \mathbb{R}^4$.

The goal is to recover A^* and b^* by training a dense layer on half the data points. Loss is measured by the L^2 distance between the network's predictions and the actual values on the **unseen** data points.

Simple training scenario

Here's a "hello world"-type training scenario. We start with a random dense layer $\mathbb{R}^{10} \to \mathbb{R}^4$, given by

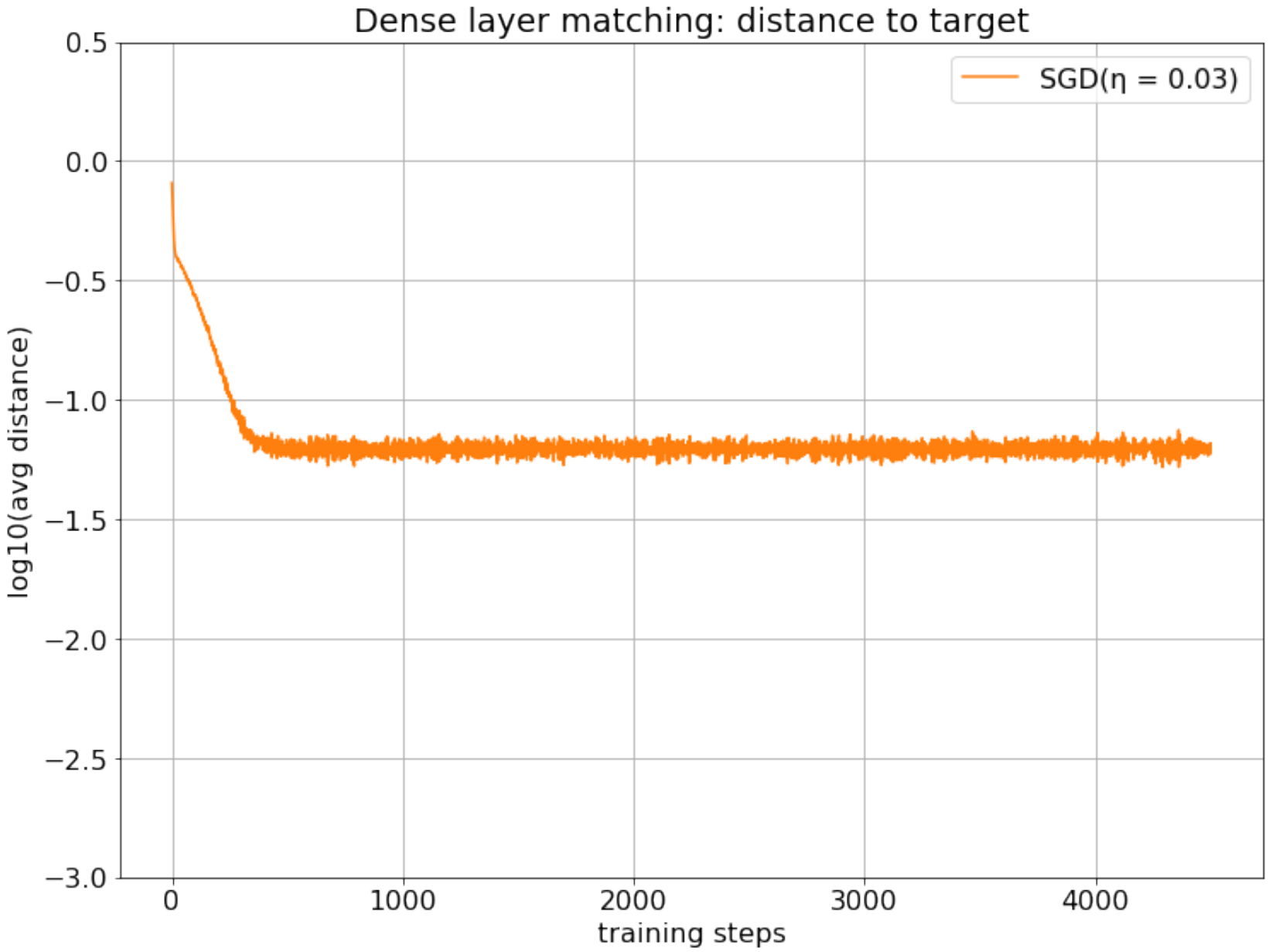
$$N(x; A^*, b^*) = \tanh(A^* x + b^*)$$

We then randomly sample 200 vectors $x_i \in \mathbb{R}^{10}$ and find the corresponding $y_i \triangleq N(x_i; A^*, b^*) \in \mathbb{R}^4$.

The goal is to recover A^* and b^* by training a dense layer on half the data points. Loss is measured by the L^2 distance between the network's predictions and the actual values on the **unseen** data points.

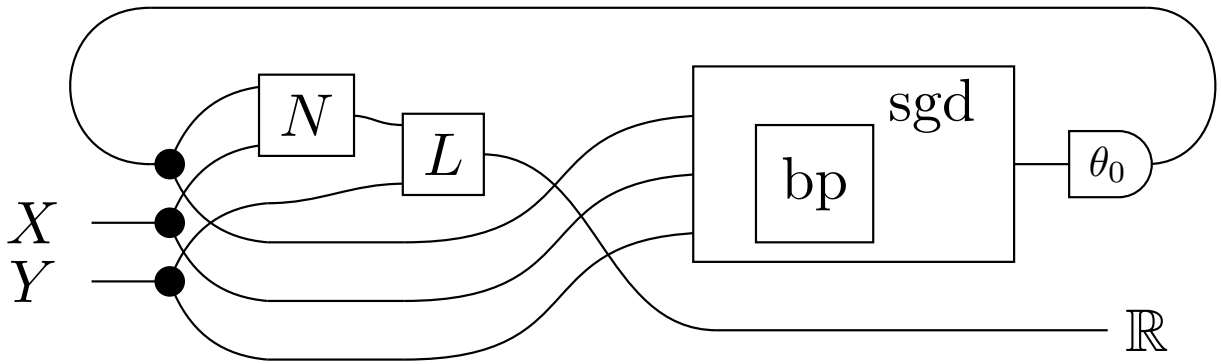
In the plots following, we will present the average results obtained over 50 sample training runs on this problem.

Training results



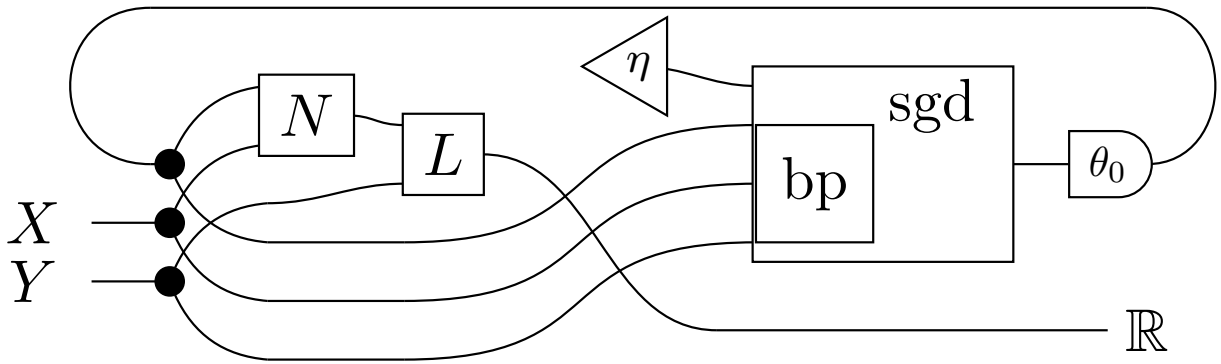
Hyperparameters

A *training algorithm* updates the parameter value, often using "backpropagation" to find the gradient of $L \circ N$ wrt Θ.

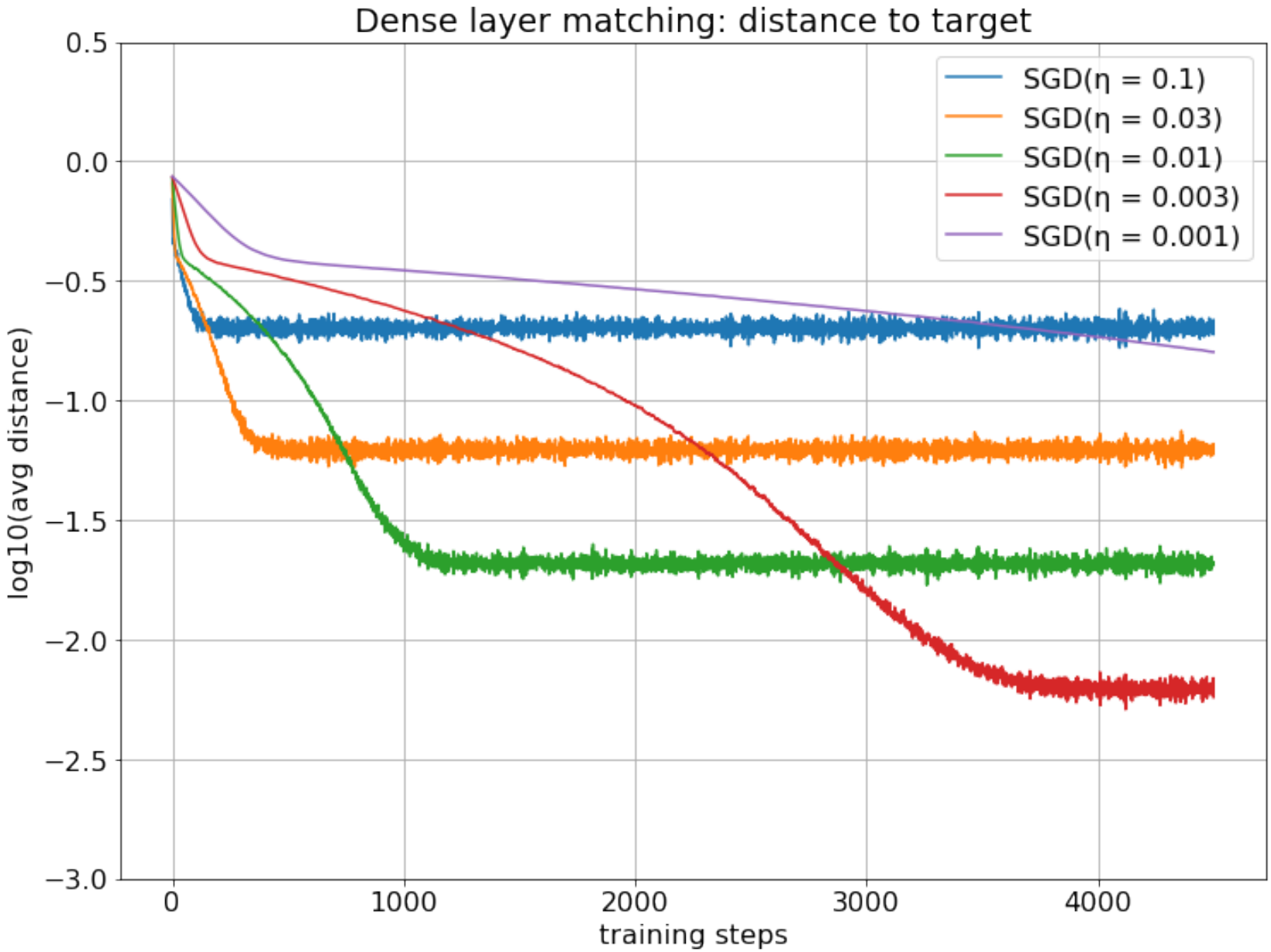


Hyperparameters

A *training algorithm* updates the parameter value, often using "backpropagation" to find the gradient of $L \circ N$ wrt Θ. Most training algorithms can be adjusted via so-called *hyperparameters*, the most famous of which is the "learning rate".



Training results for different learning rates



Getting the best of both worlds

Two common techniques for adapting the learning rate:

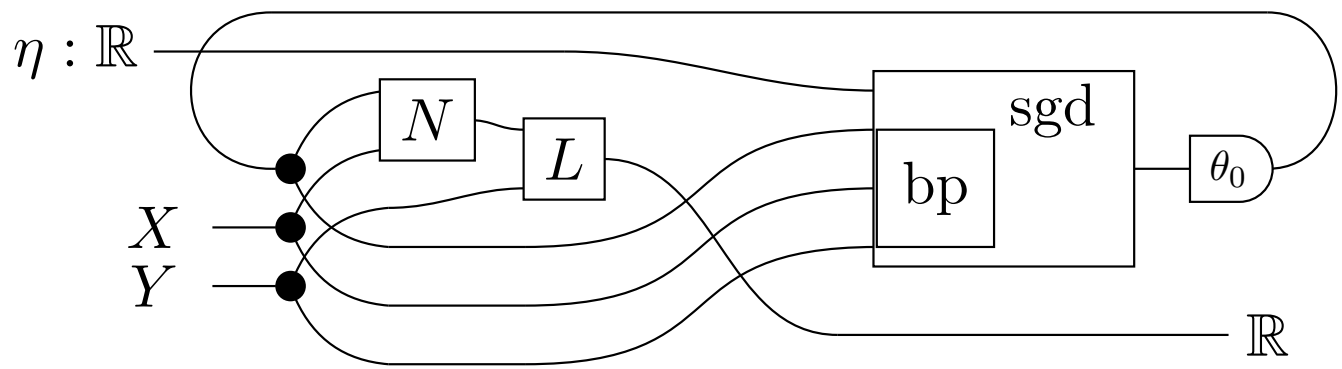
- Learning rate scheduling: $\eta_t = \eta_0 \cdot b^{-t}$
- Loss monitoring: η multiplied by b when triggering condition on loss is met

Getting the best of both worlds

Two common techniques for adapting the learning rate:

- Learning rate scheduling: $\eta_t = \eta_0 \cdot b^{-t}$
- Loss monitoring: η multiplied by b when triggering condition on loss is met

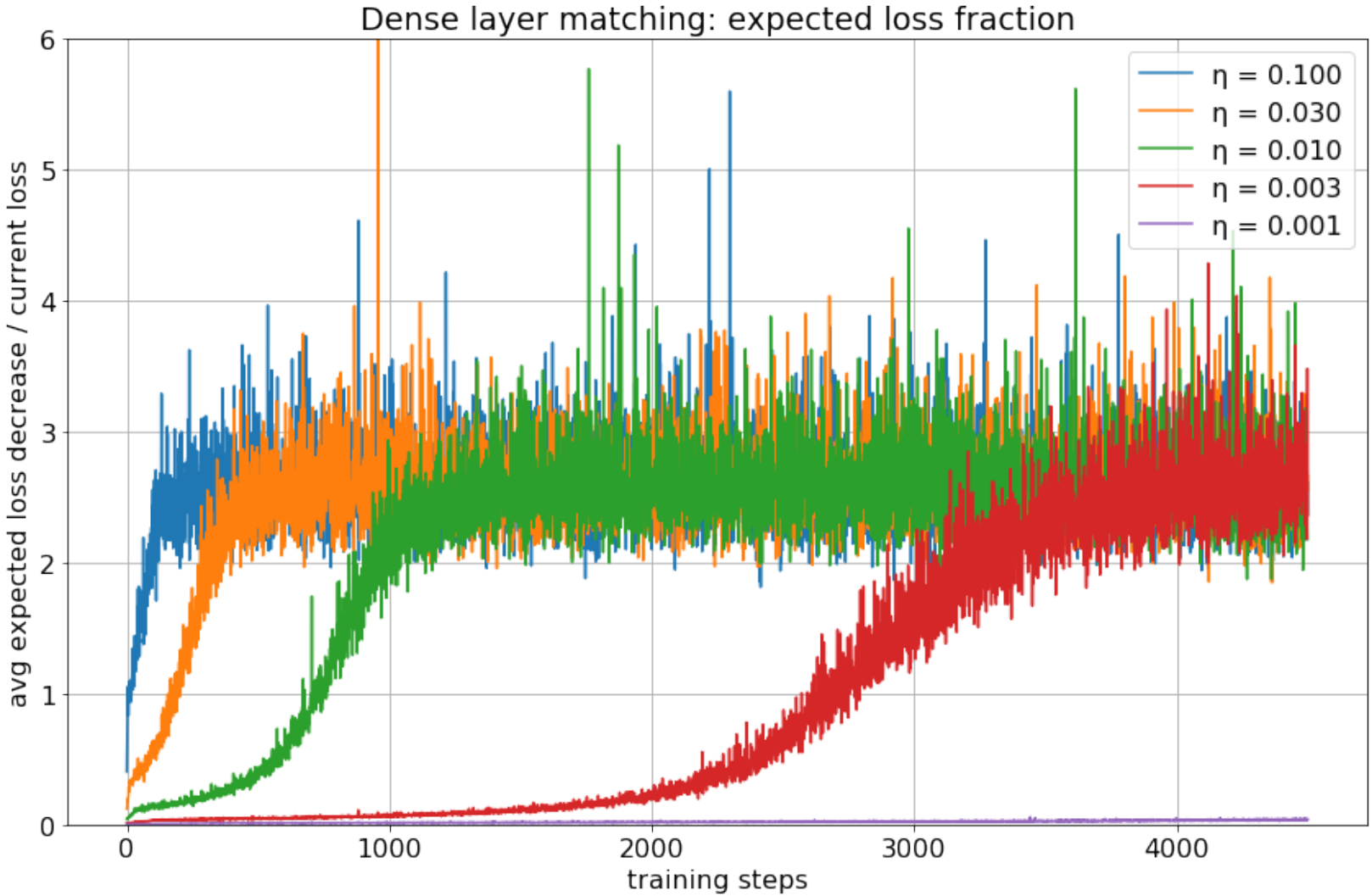
Since we can differentiate the training network with respect to η,



we build a first-order approximation to understand the effect of η.

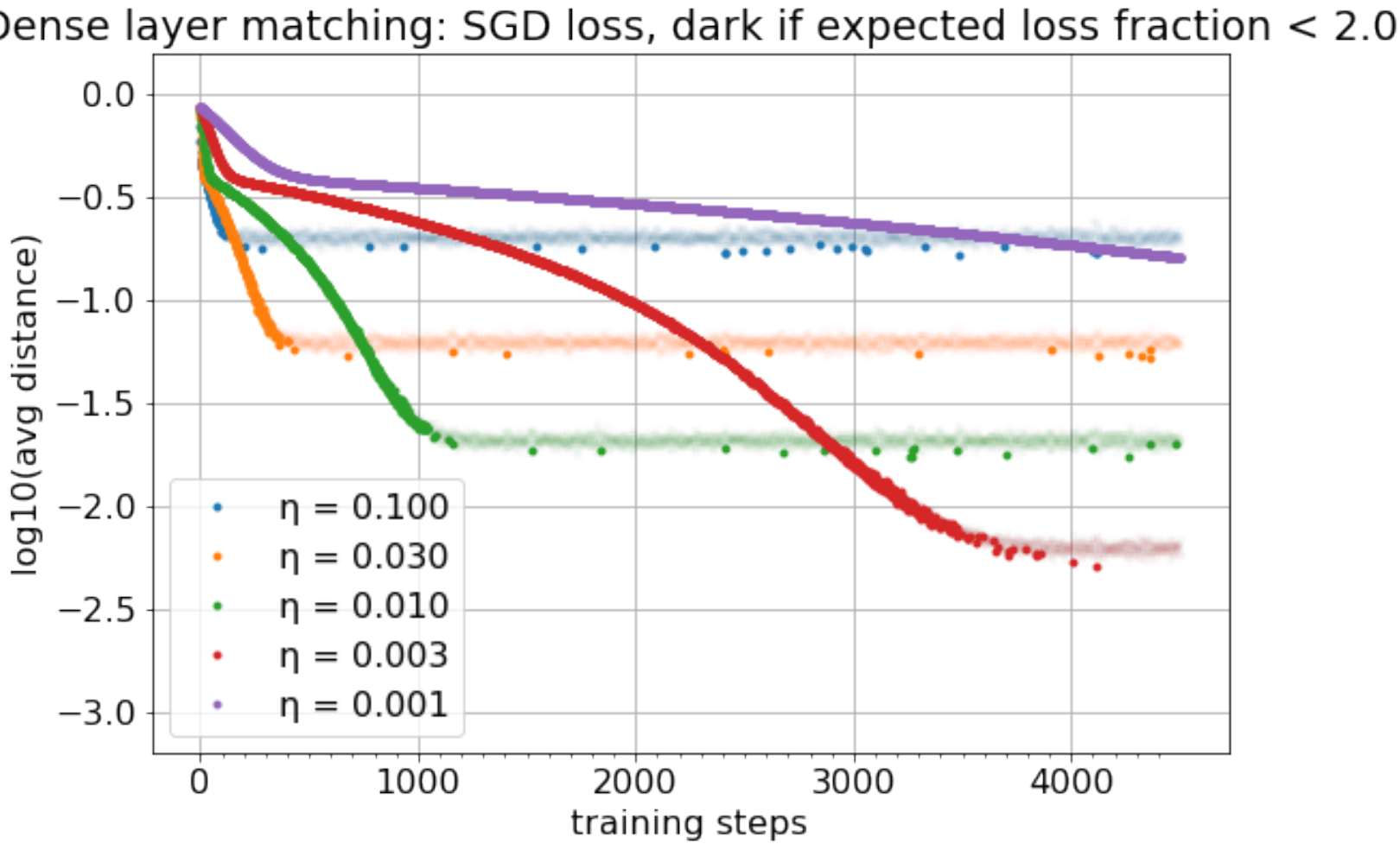
Expected loss fraction

Our first-order approximation tells us how much the loss should decrease at each step, for each learning rate.



Expected loss fraction gives insight into performance

This is the same loss-vs-step chart, except the points are solid dark when the expected loss fraction is < 2, and translucent when ≥ 2.

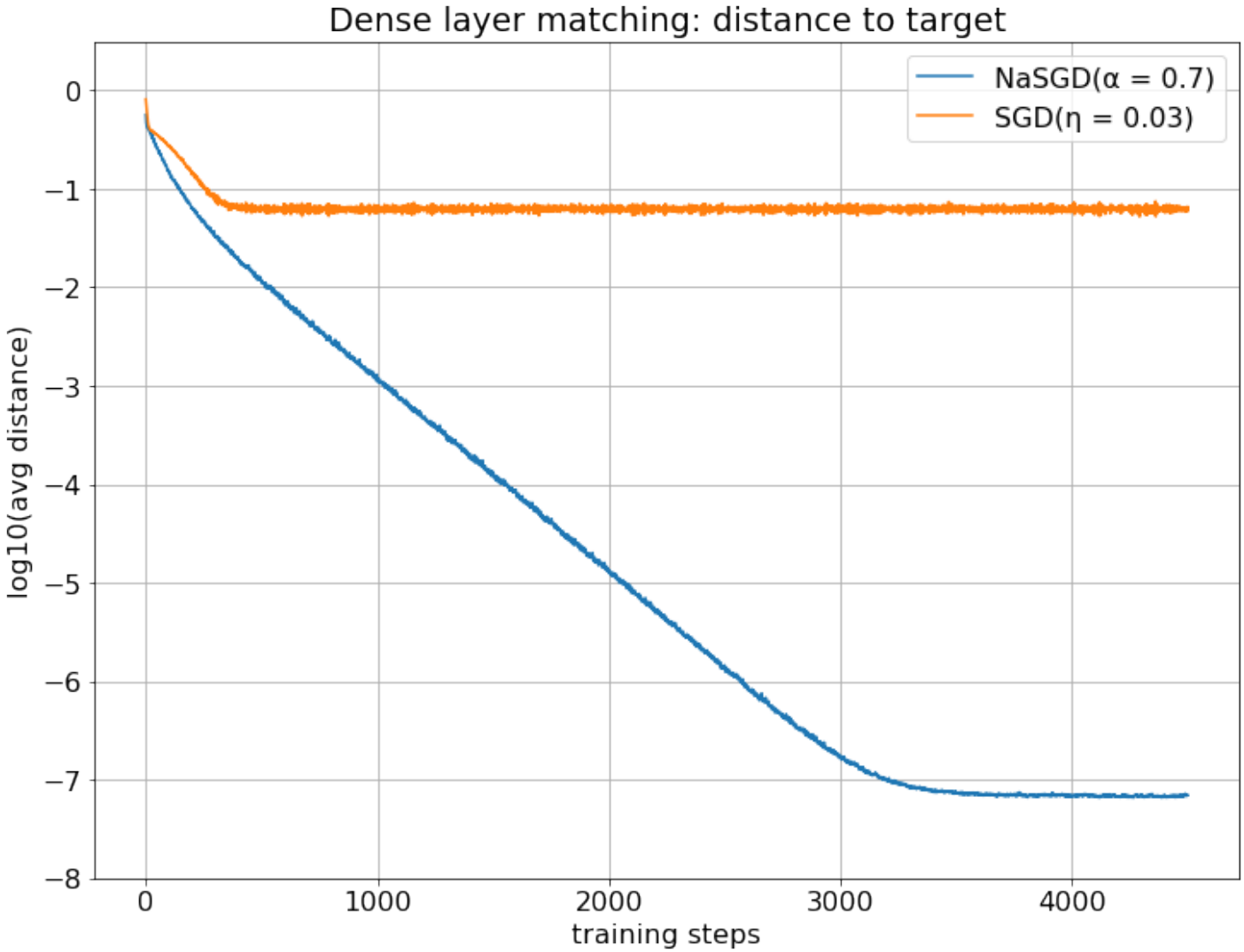


New optimizer: norm-adapted gradient descent

What if we adjusted the learning rate so that the expected loss fraction was constant, say 0.7?

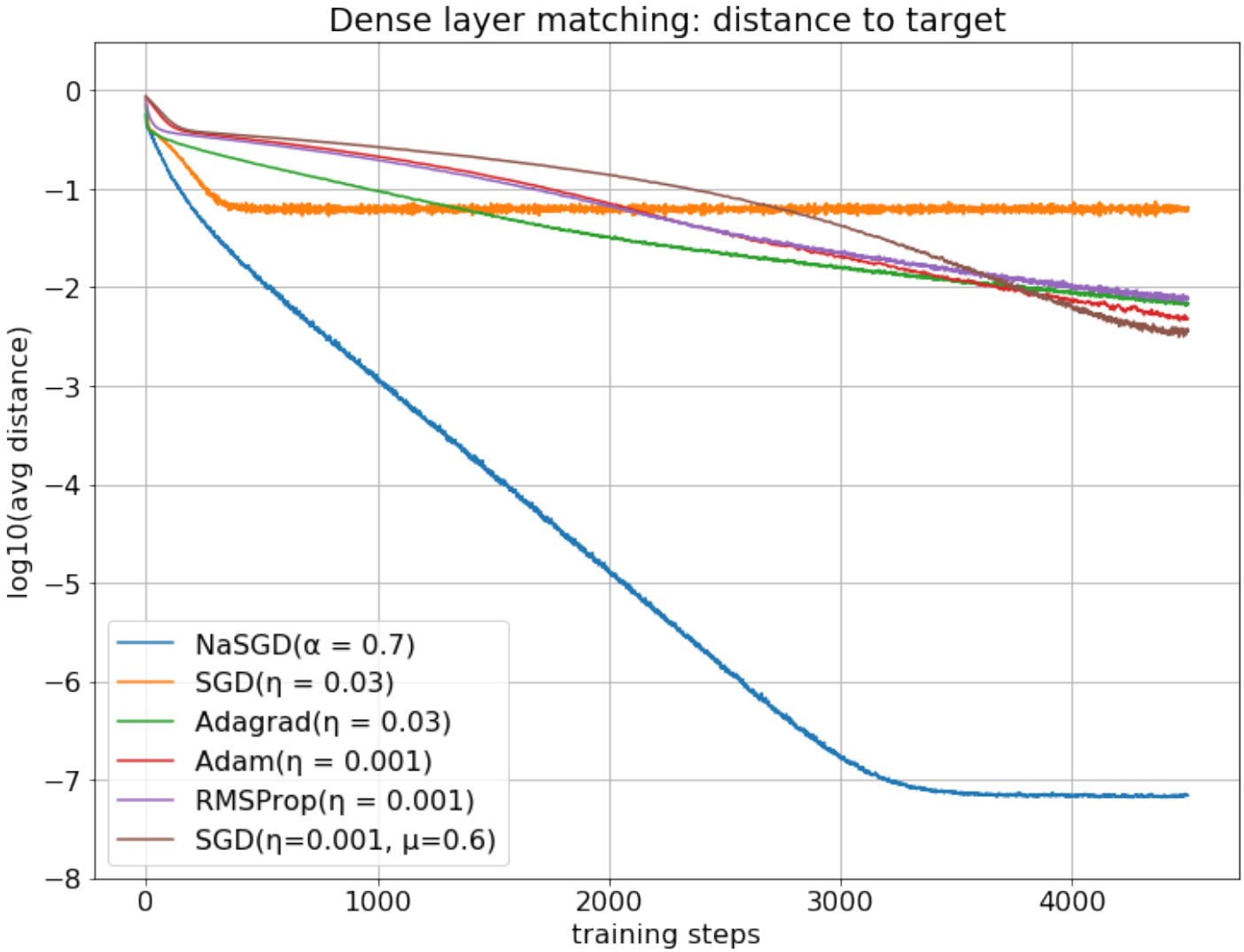
New optimizer: norm-adapted gradient descent

What if we adjusted the learning rate so that the expected loss fraction was constant, say 0.7?



New optimizer: norm-adapted gradient descent

What if we adjusted the learning rate so that the expected loss fraction was constant, say 0.7?



Words of caution

- This is a special situation: no noise and a model guarantee
- In practical benchmarks, NaSGD performs well, but not always to the degree here
- We haven't tried differentiating other optimizers

Main messages

- Cartesian differential categories are a synthetic, categorical treatment of differentiation

Main messages

- Cartesian differential categories are a synthetic, categorical treatment of differentiation
- It's possible to give a meaningful notion of derivative to Mealy machines

Main messages

- Cartesian differential categories are a synthetic, categorical treatment of differentiation
- It's possible to give a meaningful notion of derivative to Mealy machines
- Understanding RNNs better allows us to understand NN training algorithms better

Main messages

- Cartesian differential categories are a synthetic, categorical treatment of differentiation
- It's possible to give a meaningful notion of derivative to Mealy machines
- Understanding RNNs better allows us to understand NN training algorithms better
- Your skills can be useful in machine learning

Main messages

- Cartesian differential categories are a synthetic, categorical treatment of differentiation
- It's possible to give a meaningful notion of derivative to Mealy machines
- Understanding RNNs better allows us to understand NN training algorithms better
- Your skills can be useful in machine learning

Thanks!

References


R.F. Blute, J.R.B. Cockett, and R.A.G. Seely.
Cartesian differential categories.
Theory and Applications of Categories, 22(23):622–672, 2009.


F. Bonchi, P. Sobociński, and F. Zanasi.
A categorical semantics of signal flow graphs.
In *CONCUR 2014*, 2014.


C. Elliott.
The simple essence of automatic differentiation.
PACMPL, 2(ICFP):70:1–70:29, 2018.


B. Fong, D. Spivak, and R. Tuyéras.
Backprop as functor: A compositional perspective on supervised learning.
See `arxiv.org/abs/1711.10455`, 2017.


D.R. Ghica and A. Jung.
Categorical semantics of digital circuits.
FMCAD '16, Austin, TX, 2016.


P. Katis, N. Sabadini, and R.F.C. Walters.
Bicategories of processes.
Journal of Pure and Applied Algebra, 115(2):141–178, Feb 1997.


David Sprunger and Shin-ya Katsumata.
Differentiable causal computations via delayed trace.
CoRR, abs/1903.01093, 2019.