...Machine Learning...

Artur Ezequiel Nelson

Universidade do Minho

26 de Abril

Index

Definition of Derivative

Definition

Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function. The derivative of f at point $x \in \mathbb{R}$ is defined the following way:

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

This definition will also work with functions of types $\mathbb{C} \rightarrow \mathbb{C}$ and $\mathbb{R} \rightarrow \mathbb{R}^n$.

Definition of Derivative

Definition

Let $f : \mathbb{R} \to \mathbb{R}$ be a function. The derivative of f at point $x \in \mathbb{R}$ is defined the following way:

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

This definition will also work with functions of types $\mathbb{C} \to \mathbb{C}$ and $\mathbb{R} \to \mathbb{R}^n$.

Definition of Derivative

For functions F of types $\mathbb{R}^m \to \mathbb{R}$ and $\mathbb{R}^m \to \mathbb{R}^n$ (with $n > 1$), we need a different definition.

- For functions of type $\mathbb{R}^m \to \mathbb{R}$, it is necessary the introduction of the notion of parcial derivatives, $\frac{\partial F}{\partial x_j}$, with $j \in \{1, ..., m\}$.
- For functions of type $\mathbb{R}^m \to \mathbb{R}^n$ (with $n > 1$), apart from the use of parcial derivatives, it is necessary the use of Jacobian matrices $\mathbf{J}_{i,j} = \frac{\partial F_i}{\partial x_j}$, where $i \in \{1, ..., n\}$ and F_i is a function $\mathbb{R}^m \to \mathbb{R}$.

Definition of Derivative

For functions F of types $\mathbb{R}^m \to \mathbb{R}$ and $\mathbb{R}^m \to \mathbb{R}^n$ (with $n > 1$), we need a different definition.

- For functions of type $\mathbb{R}^m \to \mathbb{R}$, it is necessary the introduction of the notion of parcial derivatives, $\frac{\partial F}{\partial x_j}$, with $j \in \{1, ..., m\}$.
- For functions of type $\mathbb{R}^m \to \mathbb{R}^n$ (with $n > 1$), apart from the use of parcial derivatives, it is necessary the use of Jacobian matrices $\mathbf{J}_{i,j} = \frac{\partial F_i}{\partial x_j}$, where $i \in \{1, ..., n\}$ and F_i is a function $\mathbb{R}^m \to \mathbb{R}$.

Definition of Derivative

For functions F of types $\mathbb{R}^m \to \mathbb{R}$ and $\mathbb{R}^m \to \mathbb{R}^n$ (with $n > 1$), we need a different definition.

- For functions of type $\mathbb{R}^m \to \mathbb{R}$, it is necessary the introduction of the notion of parcial derivatives, $\frac{\partial F}{\partial x_j}$, with $j \in \{1, ..., m\}$.
- For functions of type $\mathbb{R}^m \to \mathbb{R}^n$ (with $n > 1$), apart from the use of parcial derivatives, it is necessary the use of Jacobian matrices $\mathbf{J}_{i,j} = \frac{\partial F_i}{\partial x_j}$, where $i \in \{1, ..., n\}$ and F_i is a function $\mathbb{R}^m \to \mathbb{R}$.

Generalization and Chain Rule

Let **A** and **B** be two Jacobian matrices.
The chain rule in $\mathbb{R}^m \to \mathbb{R}^n$ is:

$$(\mathbf{A} \cdot \mathbf{B})_{i,j} = \sum_{k=1}^{m} \mathbf{A}_{i,k} \cdot \mathbf{B}_{k,j}$$

Generalization and Chain Rule

Assuming that the notion of derivates that we need matches with a linear map, where it is accepted the chain rule previously seen, we will define a new generalization:

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

$$\Leftrightarrow \lim_{\varepsilon \to 0} \frac{\|f(x+\varepsilon) - (f(x)) + \varepsilon \cdot f'(x)\|}{\|\varepsilon\|} =$$

Generalization and Chain Rule

Assuming that the notion of derivates that we need matches with a linear map, where it is accepted the chain rule previously seen, we will define a new generalization:

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

$$\Leftrightarrow \lim_{\varepsilon \to 0} \frac{\|f(x+\varepsilon) - (f(x)) + \varepsilon \cdot f'(x)\|}{\|\varepsilon\|} =$$

Derivate as a linear map

Definition

Let $f :: a \to b$ be a function, where a and b are vetorial spaces that share a common underlying field. The first derivate definition is the following:

$$\mathcal{D} :: (a \to b) \to (a \to (a \multimap b))$$

If we differentiate two times vezes, we have:

$$\mathcal{D}^2 = \mathcal{D} \circ \mathcal{D} :: (a \to b) \to (a \to (a \multimap a \multimap b))$$

Rules for Differentiation - Sequential Composition

Theorem

Let $f :: a \rightarrow b$ and $g :: b \rightarrow c$ be two functions. Then the derivative of the composition of f e g is:

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a$$

Rules for Differentiation - Sequential Composition

Unfortunately the previous theorem isn't a efficient recipe for composition, and now we will introduce a second derivate definition:

$$\mathcal{D}_0^+ :: (a \to b) \to ((a \to b) \times (a \to (a \multimap b)))$$
$$\mathcal{D}_0^+ \, f = (f, \mathcal{D} f)$$

With this, the chain rule will have the following expression:
$\mathcal{D}_0^+ (g \circ f) =$
$= (g \circ f, \mathcal{D} (g \circ f))$ (definition of $\mathcal{D}_0^+$)
$= (\lambda a \to g(f\, a), \lambda a \to \mathcal{D}\, g\, (f\, a) \circ \mathcal{D}\, f\, a)$ (theorem and definition of $g \circ f$)

Rules for Differentiation - Sequential Composition

Unfortunately the previous theorem isn't a efficient recipe for composition, and now we will introduce a second derivate definition:

$$\mathcal{D}_0^+ :: (a \to b) \to ((a \to b) \times (a \to (a \multimap b)))$$
$$\mathcal{D}_0^+\, f = (f, \mathcal{D} f)$$

With this, the chain rule will have the following expression:
$\mathcal{D}_0^+ (g \circ f) =$
$= (g \circ f, \mathcal{D}(g \circ f))$ (definition of $\mathcal{D}_0^+$)
$= (\lambda a \to g(f\, a), \lambda a \to \mathcal{D}\, g\, (f\, a) \circ \mathcal{D}\, f\, a)$ (theorem and definition of $g \circ f$)

Rules for Differentiation - Sequential Composition

Having in mind optimizations, we introduce the third and last derivate definition:

$$\mathcal{D}^+ :: (a \to b) \to (a \to (b \times (a \multimap b)))$$
$$\mathcal{D}^+\ f\ a = (f\ a, \mathcal{D}\ f\ a)$$

As $\times$ has more priority than $\to$ e $\multimap$, we can rewrite $\mathcal{D}^+$ as:

$$\mathcal{D}^+ :: (a \to b) \to (a \to b \times (a \multimap b))$$
$$\mathcal{D}^+\ f\ a = (f\ a, \mathcal{D}\ f\ a)$$

Rules for Differentiation - Sequential Composition

Having in mind optimizations, we introduce the third and last derivate definition:

$$\mathcal{D}^{+} :: (a \to b) \to (a \to (b \times (a \multimap b)))$$
$$\mathcal{D}^{+}\, f\, a = (f\, a, \mathcal{D}\, f\, a)$$

As $\times$ has more priority than $\to$ e $\multimap$, we can rewrite $\mathcal{D}^{+}$ as:

$$\mathcal{D}^{+} :: (a \to b) \to (a \to b \times (a \multimap b))$$
$$\mathcal{D}^{+}\, f\, a = (f\, a, \mathcal{D}\, f\, a)$$

Rules for Differentiation - Sequential Composition

Corollary

$\mathcal{D}^+$ is efficiently compositional in relation to $(\circ)$, that is, in Haskell:

$$\mathcal{D}^+ \, (g \circ f) \; a = \textbf{let} \{(b, f') = \mathcal{D}^+ \, f \; a; \, (c, g') = \mathcal{D}^+ \, g \; b\} \; \textbf{in} \, (c, g' \circ f')$$

Rules for Differentiation - Parallel Composition

Another important way of combining functions is the operation cross, that combines two functions in parallel:

$$(\times) :: (a \to c) \to (b \to d) \to (a \times b \to c \times d)$$
$$f \times g = \lambda(a, b) \to (f\,a, g\,b)$$

Theorem

Let $f :: a \to c$ and $g :: b \to d$ be two function. Then the cross rule is the following:

$$\mathcal{D}\,(f \times g)\,(a, b) = \mathcal{D}\,f\,a \times \mathcal{D}\,g\,b$$

Rules for Differentiation - Parallel Composition

Another important way of combining functions is the operation cross, that combines two functions in parallel:

$$(\times) :: (a \to c) \to (b \to d) \to (a \times b \to c \times d)$$
$$f \times g = \lambda(a, b) \to (f\, a, g\, b)$$

Theorem

Let $f :: a \to c$ and $g :: b \to d$ be two function. Then the cross rule is the following:

$$\mathcal{D}\,(f \times g)\,(a, b) = \mathcal{D}\, f\; a \times \mathcal{D}\, g\; b$$

Rules for Differentiation - Parallel Composition

Corollary

The function $\mathcal{D}^+$ is compositional in relation to $(\times)$

$$\mathcal{D}^+\,(f \times g)\,(a, b) = \textbf{\textit{let}}\,\{(c, f') = \mathcal{D}^+\,f\,a;\,(d, g') = \mathcal{D}^+\,g\,b\}\;\textbf{\textit{in}}\,((c, d), f'$$

Derivative e Linear Functions

Definition

A function f is said to be linear when preserves addition and scalar multiplication.

$$f(a + a') = f\,a + f\,a'$$
$$f(s \cdot a) = s \cdot f\,a$$

Theorem

For all linear functions f, $\mathcal{D}\,f\,a = f$.

Corollary

For all linear functions f, $\mathcal{D}^+\,f = \lambda a \rightarrow (fa, f)$.

Derivative e Linear Functions

Definition

A function f is said to be linear when preserves addition and scalar multiplication.

$$f(a + a') = f\,a + f\,a'$$
$$f(s \cdot a) = s \cdot f\,a$$

Theorem

For all linear functions f, $\mathcal{D}\,f\,a = f$.

Corollary

For all linear functions f, $\mathcal{D}^{+}\,f = \lambda a \rightarrow (fa, f)$.

Derivative e Linear Functions

Definition

A function f is said to be linear when preserves addition and scalar multiplication.

$$f(a + a') = f\,a + f\,a'$$
$$f(s \cdot a) = s \cdot f\,a$$

Theorem

For all linear functions f, $\mathcal{D}\,f\;a = f$.

Corollary

For all linear functions f, $\mathcal{D}^{+}\,f = \lambda a \rightarrow (fa,\; f)$.

A short introdution

- We want to calculate $\mathcal{D}$.
- However, $\mathcal{D}$ is not computable.
- Solution: reimplement corollaries using category theory

A short introdution

- We want to calculate $\mathcal{D}$.
- However, $\mathcal{D}$ is not computable.
- Solution: reimplement corollaries using category theory

A short introdution

- We want to calculate $\mathcal{D}$.
- However, $\mathcal{D}$ is not computable.
- Solution: reimplement corollaries using category theory

A short introdution

Corollary 1.1

NOTA: adicionar definição do corolário 1.1 aqui

Corollary 2.1

NOTA: adicionar definição do corolário 2.1 aqui

Corollary 3.1

NOTA: adicionar definição do corolário 3.1 aqui

Categories

A category is a collection of objects(sets and types) and morphisms(operation between objects), with 2 basic operations(identity and composition) of morfisms, and 2 laws:

- (C.1) $id \circ f = id \circ f = f$
- (C.2) $f \circ (g \circ h) = (f \circ g) \circ h$

Note

For this paper, objects are data types and morfisms are functions

$$
\begin{array}{ll}
\textbf{class}\ \mathit{Category}\ k\ \textbf{where} & \textbf{instance}\ \mathit{Category}\ (\rightarrow)\ \textbf{where} \\
\quad id :: (a'k'a) & \quad id = \lambda a \rightarrow a \\
\quad (\circ) :: (b'k'c) \rightarrow (a'k'b) \rightarrow (a'k'c) & \quad g \circ f = \lambda a \rightarrow g\ (f\ a)
\end{array}
$$

Functors

A functor F between 2 categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

- given any object t $\in \mathcal{U}$ there exists an object F t $\in \mathcal{V}$
- given any morphism m :: a $\rightarrow$ b $\in \mathcal{U}$ there exists a morphism F m :: F a $\rightarrow$ F b $\in \mathcal{V}$
- F id ($\in \mathcal{U}$) = id ($\in \mathcal{V}$)
- F (f $\circ$ g) = F f $\circ$ F g

Note

Given this papers category properties(objects are data types) functors map types to themselves

Objective

Let's start by defining a new data type:
newtype $\mathcal{D}$ a b = $\mathcal{D}$ (a $\rightarrow$ b $\times$ (a $\multimap$ b))
,and adapting $\mathcal{D}^+$ to use it:

Adapted definition

$$\hat{\mathcal{D}} :: (a \rightarrow b) \rightarrow \mathcal{D}\ a\ b$$
$$\hat{\mathcal{D}}\ f = \mathcal{D}\ (\mathcal{D}^+\ f)$$

Our objective is to deduce an instance of a Category for $\mathcal{D}$ where $\hat{\mathcal{D}}$ is a functor.

Instance deduction

Before deducing our instance we must first note that using corollaries 3.1 and 1.1 we can determine that

- (DP.1) $\mathcal{D}^+$ id = λ a -> (id a,id)
- (DP.2) $\mathcal{D}^+$(g $\circ$ f) = λ a -> let(b,f') = $\mathcal{D}^+$ f a; (c,g') = $\mathcal{D}^+$ g b in (c,g' $\circ$ f')

Saying that $\hat{\mathcal{D}}$ is a functor is equivalent to, for all f and g functions of apropriate types:

$id = \hat{\mathcal{D}}\ id = \mathcal{D}\ (\mathcal{D}^+\ id)$

$\hat{\mathcal{D}}$ g $\circ$ $\hat{\mathcal{D}}$ f = $\hat{\mathcal{D}}$ (g $\circ$ f) = $\mathcal{D}$ ($\hat{\mathcal{D}}$ (g $\circ$ f))

Instance deduction

Based on (DP.1) and (DP.2) we'll rewrite the above into the following definition:
id = $\mathcal{D}$ (λ a -> (id a,id))
$\hat{\mathcal{D}}$ g $\circ$ $\hat{\mathcal{D}}$ f = $\mathcal{D}$ (λ a -> let(b,f') = $\mathcal{D}^+$ f a; (c,g') = $\mathcal{D}^+$ g b in (c,g' $\circ$ f'))
The first equation shown above has a trivial solution(define id of instance as $\mathcal{D}$(λ a -> (id a,id)))
To solve the second we'll first solve a more general one:
$\mathcal{D}$ g $\circ$ $\mathcal{D}$ f = $\mathcal{D}$(λ a -> let(b,f') = f a; (c,g') = g b in(c,g' $\circ$ f'))
This condition also leads us to a trivial solution inside our instance.

Instance deduction

$\hat{\mathcal{D}}$ definition for linear functions

$$\mathit{linearD} :: (a \to b) \to \mathcal{D}\ a\ b$$
$$\mathit{linearD}\ f = \mathcal{D}\ (\lambda a \to (f\ a, f))$$

Categorical instance we've deduced

$$\begin{array}{l}\textbf{instance}\ \mathit{Category}\ \mathcal{D}\ \textbf{where} \\ \quad \mathit{id} = \mathit{linearD}\ \mathit{id} \\ \quad \mathcal{D}\ g \circ \mathcal{D}\ f = \\ \quad \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))\end{array}$$

Instance proof

In order to prove that the instance is correct we must check if it follows laws (C.1) and (C.2).
First we must make a concession: that we only use morfisms arising from $\mathcal{D}^+$(we can force this by transforming $\mathcal{D}$ into an abstract type). If we do, then $\mathcal{D}^+$ is a functor.

(C.1) proof

id $\circ$ $\hat{\mathcal{D}}$
= $\hat{\mathcal{D}}$ id $\circ$ $\hat{\mathcal{D}}$ f - functor law for id (specification of $\hat{\mathcal{D}}$)
= $\hat{\mathcal{D}}$ (id $\circ$ f) - functor law for ($\circ$)
= $\hat{\mathcal{D}}$ f - categorical law

Instance proof

(C.2) proof

$\hat{\mathcal{D}}$ h $\circ$ ($\hat{\mathcal{D}}$ g $\circ$ $\hat{\mathcal{D}}$ f)
= $\hat{\mathcal{D}}$ h $\circ$ $\hat{\mathcal{D}}$ (g $\circ$ f) - functor law for ($\circ$)
= $\hat{\mathcal{D}}$ (h $\circ$ (g $\circ$ f)) - functor law for ($\circ$)
= $\hat{\mathcal{D}}$ ((h $\circ$ g) $\circ$ f) - categorical law
= $\hat{\mathcal{D}}$ (h $\circ$ g) $\circ$ $\hat{\mathcal{D}}$ f - functor law for ($\circ$)
= ($\hat{\mathcal{D}}$ h $\circ$ $\hat{\mathcal{D}}$ g) $\circ$ $\hat{\mathcal{D}}$ f - functor law for ($\circ$)

Note

This proofs don't require anything from $\mathcal{D}$ *and* $\hat{\mathcal{D}}$ aside from functor laws. As such, all other instances of categories created from a functor won't require further proving like this onr did.

Monoidal categories and functors

Generalized parallel composition shall be defined using a monoidal category:

$$\textbf{class } \mathit{Category}\ k \Rightarrow \mathit{Monoidal}\ k\ \textbf{where}$$
$$(x) :: (a\ '\mathtt{k}'\ \ c) \to (b\ '\mathtt{k}'\ \ d) \to ((a\ x\ b)\ '\mathtt{k}'\ \ (c\ x\ d))$$

$$\textbf{instance } \mathit{Monoidal}\ (\to)\ \textbf{where}$$
$$f\ x\ g = \lambda(a, b) \to (f\ a, g\ b)$$

Monoidal Functor definition

A monoidal functor F between categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

- F is a functor
- F (f $\times$ g) = F f $\times$ F g

Instance deduction

From corollary 2.1 we can deduce that:
$\mathcal{D}^+$ (f $\times$ g) = λ (a,b) -> let(c,f')=$\mathcal{D}^+$ f a;(d,g')= $\mathcal{D}^+$ g b
in ((c,d),f' $\times$ g')
Deriving F from $\hat{\mathcal{D}}$ leaves us with the following definition:
$\mathcal{D}$ ($\mathcal{D}^+$ f) $\times$ $\mathcal{D}$ ($\mathcal{D}^+$ g) = $\mathcal{D}$ ($\mathcal{D}^+$ (f $\times$ g))
Using the same method as before, we replace $\mathcal{D}^+$ with it's definition and generalize the condition:
$\mathcal{D}$ f $\times$ $\mathcal{D}$ g =
$\mathcal{D}$ (λ (a,b) -> let(c,f') = f a; (d,g') = g b in ((c,d),f' $\times$ g'))
and this is enough for our new instance.

Instance deduction

Categorical instance we've deduced

$$
\begin{array}{l}
\textbf{instance } \mathit{Monoidal}\ \mathcal{D}\ \textbf{where} \\
\quad \mathcal{D}\ f\ x\ \mathcal{D}\ g = \mathcal{D}\ (\lambda(a, b) \rightarrow \textbf{let } \{(c, f') = f\ a; (d, g') = g\ b\} \\
\qquad \textbf{in } ((c, d), f'\ x\ g'))
\end{array}
$$

Cartesian categories and functors

class *Monoidal k* $\Rightarrow$ *Cartesian k* **where**
  $exl :: (a, b)\ \texttt{'k'}\ a$
  $exr :: (a, b)\ \texttt{'k'}\ b$
  $dup :: a'k'\ (a, a)$

instance *Cartesian* $(\rightarrow)$ **where**
  $exl = \lambda(a, b) \rightarrow a$
  $exr = \lambda(a, b) \rightarrow b$
  $dup = \lambda a \rightarrow (a, a)$

A cartesian functor F between categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

- F is a monoidal functor
- F exl = exl
- F exp = exp

Instance deduction

From corollary 3.1 and from exl,exr and dup being linear functions we can deduce that:
$\mathcal{D}^{+}$ exl = $\lambda p \rightarrow$ (exl p,exl)
$\mathcal{D}^{+}$ exr = $\lambda p \rightarrow$ (exr p,exr)
$\mathcal{D}^{+}$ dup = $\lambda p \rightarrow$ (dup a,dup)
With this in mind we can arrive at our instance:
exl = $\mathcal{D}(\mathcal{D}^{+}$ exl)
exr = $\mathcal{D}(\mathcal{D}^{+}$ exr)
dup = $\mathcal{D}(\mathcal{D}^{+}$ dup)

Instance deduction

Replacing $\mathcal{D}^+$ with it's definition and remembering linearD's definition we can obtain:
exl = linearD exl
exr = linearD exr
dup = linearD dup
and convert this directly into a new instance:

Categorical instance we've deduced

instance *Cartesian D* **where**
$exl = linearD\ exl$
$exr = linearD\ exr$
$dup = linearD\ dup$

Cocartesian category

This type of categories is the dual of the cartesian type of categories.

Note

In this paper coproducts are categorical products, i.e., biproducts

Definition

$$\begin{aligned}
&\textbf{class } \mathit{Category}\ k \Rightarrow \mathit{Cocartesian}\ k \textbf{ where}:\\
&\quad \mathit{inl} :: a' k' \ (a, b)\\
&\quad \mathit{inr} :: b' k' \ (a, b)\\
&\quad \mathit{jam} :: (a, a)\ '\mathtt{k}'\ \ a
\end{aligned}$$

Cocartesian functors

Cocartesian functor definition

A cocartesian functor F between categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

- F is a functor
- F inl = inl
- F inr = inr
- F jam = jam

Fork and Join

- $\Delta :: \mathit{Cartesian}\ k \Rightarrow (a \mathbin{'\mathrm{k}'} c) \rightarrow (a \mathbin{'\mathrm{k}'} d) \rightarrow (a \mathbin{'\mathrm{k}'} (c \times d))$
- $\nabla :: \mathit{Cartesian}\ k \Rightarrow (c \mathbin{'\mathrm{k}'} a) \rightarrow (d \mathbin{'\mathrm{k}'} a) \rightarrow ((c \times d) \mathbin{'\mathrm{k}'} a)$

Instance of $\rightarrow^{+}$

$$
\begin{array}{l}
\textbf{newtype}\ a \rightarrow^{+} b = \mathit{AddFun}\ (a \rightarrow b) \\
\textbf{instance}\ \mathit{Category}\ (\rightarrow^{+})\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Obj}\ (\rightarrow^{+}) = \mathit{Additive} \\
\quad \mathit{id} = \mathit{AddFun}\ \mathit{id} \\
\quad \mathit{AddFun}\ g \circ \mathit{AddFun}\ f = \mathit{AddFun}\ (g \circ f) \\
\textbf{instance}\ \mathit{Monoidal}\ (\rightarrow^{+})\ \textbf{where} \\
\quad \mathit{AddFun}\ f \times \mathit{AddFun}\ g = \mathit{AddFun}\ (f \times g) \\
\textbf{instance}\ \mathit{Cartesian}\ (\rightarrow^{+})\ \textbf{where} \\
\quad \mathit{exl} = \mathit{AddFun}\ \mathit{exl} \\
\quad \mathit{exr} = \mathit{AddFun}\ \mathit{exr} \\
\quad \mathit{dup} = \mathit{AddFun}\ \mathit{dup}
\end{array}
$$

Instance of $\rightarrow^{+}$

instance *Cocartesian* $(\rightarrow^{+})$ **where**
  $inl = AddFun\ inlF$
  $inr = AddFun\ inrF$
  $jam = AddFun\ jamF$

$$inlF :: Additive\ b \Rightarrow a \rightarrow a \times b$$
$$inrF :: Additive\ a \Rightarrow b \rightarrow a \times b$$
$$jamF :: Additive\ a \Rightarrow a \times a \rightarrow a$$

$$inlF = \lambda a \rightarrow (a, 0)$$
$$inrF = \lambda b \rightarrow (0, b)$$
$$jamF = \lambda(a, b) \rightarrow a + b$$

NumCat definition

class *NumCat k a* **where**
  $negateC :: a \text{ `} k \text{ `} a$
  $addC :: (a \times a) \text{ `} k \text{ `} a$
  $mulC :: (a \times a) \text{ `} k \text{ `} a$
  $\ldots$
instance $Num\ a \Rightarrow NumCat\ (\rightarrow)\ a$ **where**
  $negateC = negate$
  $addC = uncurry\ (+)$
  $mulC = uncurry\ (*)$
  $\ldots$

$$\mathcal{D}\ (negate\ u) = negate\ (\mathcal{D}\ u)$$
$$\mathcal{D}\ (u + v) = \mathcal{D}\ u + \mathcal{D}\ v$$
$$\mathcal{D}\ (u * v) = u * \mathcal{D}\ v + v * \mathcal{D}\ u$$

- Imprecise on the nature of u and v.
- A precise and simpler definition would be to differentiate the operations themselves.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Scalable}\ k\ a\ \textbf{where} \\
\quad \mathit{scale} :: a \rightarrow (a\ `\ k\ `\ a) \\
\textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Scalable}\ (\rightarrow^{+})\ a\ \textbf{where} \\
\quad \mathit{scale}\ a = \mathit{AddFun}\ (\lambda da \rightarrow a * da) \\
\textbf{instance}\ \mathit{NumCat}\ D\ \textbf{where} \\
\quad \mathit{negateC} = \mathit{linearD}\ \mathit{negateC} \\
\quad \mathit{addC} = \mathit{linearD}\ \mathit{addC} \\
\quad \mathit{mulC} = D\ (\lambda(a, b) \rightarrow (a * b, \mathit{scale}\ b \triangledown \mathit{scale}\ a)) \\
\textbf{instance}\ \mathit{FloatingCat}\ D\ \textbf{where} \\
\quad \mathit{sinC} = D\ (\lambda a \rightarrow (\sin a, \mathit{scale}\ (\cos a))) \\
\quad \mathit{cosC} = D\ (\lambda a \rightarrow (\cos a, \mathit{scale}\ (-\sin a))) \\
\quad \mathit{expC} = D\ (\lambda a \rightarrow \textbf{let}\ e = \exp a\ \textbf{in}\ (e, \mathit{scale}\ e)) \\
\quad \ldots
\end{array}
$$

Examples

$$
\begin{aligned}
&sqr :: Num\ a \Rightarrow a \rightarrow a\\
&sqr\ a = a * a\\
&magSqr :: Num\ a \Rightarrow a \times a \rightarrow a\\
&magSqr\ (a, b) = sqr\ a + sqr\ b\\
&cosSinProd :: Floating\ a \Rightarrow a \times a \rightarrow a \times a\\
&cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \textbf{where}\ z = x * y
\end{aligned}
$$

With a compiler plugin we can obtain

$$
\begin{aligned}
&sqr = mulC \circ (id \vartriangle id)\\
&magSqr = addC \circ (mulC \circ (exl \vartriangle exl) \vartriangle mulC \circ (exr \vartriangle exr))\\
&cosSinProd = (cosC \vartriangle sinC) \circ mulC
\end{aligned}
$$

Generalizing Automatic Differentiation

$$\textbf{newtype}\ D_k\ a\ b = D\ (a \to b \times (a\ \text{`}\ k\ \text{`}\ b))$$

$$linearD :: (a \to b) \to (a\ \text{`}\ k\ \text{`}\ b) \to D_k\ a\ b$$
$$linearD\ f\ f' = D\ (\lambda a \to (f\ a, f'))$$

$$\textbf{instance}\ Category\ k \Rightarrow Category\ D_k\ \textbf{where}$$
$$\quad \textbf{type}\ Obj\ D_k = Additive \wedge Obj\ k\ \ldots$$

$$\textbf{instance}\ Monoidal\ k \Rightarrow Monoidal\ D_k\ \textbf{where}\ \ldots$$

$$\textbf{instance}\ Cartesian\ k \Rightarrow Cartesian\ D_k\ \textbf{where}\ \ldots$$

$$\textbf{instance}\ Cocartesian\ k \Rightarrow Cocartesian\ D_k\ \textbf{where}$$
$$\quad inl = linearD\ inlF\ inl$$
$$\quad inr = linearD\ inrF\ inr$$
$$\quad jam = linearD\ jamF\ jam$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Scalable}\ k\ s \Rightarrow \mathit{NumCat}\ D_k\ s \textbf{ where} \\
\quad \mathit{negateC} = \mathit{linearD}\ \mathit{negateC}\ \mathit{negateC} \\
\quad \mathit{addC} = \mathit{linearD}\ \mathit{addC}\ \mathit{addC} \\
\quad \mathit{mulC} = D\ (\lambda(a, b) \rightarrow (a * b, \mathit{scale}\ b \triangledown \mathit{scale}\ a))
\end{array}
$$

Matrices

There exists three, non-exclusive, possibilities for a nonempty matrix W:

- width W = height W = 1;
- W is the horizontal juxtaposition of two matrices U e V, where height W = height U = height V and width W = width U + width V;
- W is the vertical juxtaposition of two matrices U e V, where width W = width U = width V and height W = height U + height V.

Extracting a Data Representation

In machine learning, a Gradient-based optimization works by searching for local minima in the domain of a differentiable function $f :: a \to s$. Each step in the search is in the direction opposite of the gradient of f, which is a vector form of $\mathcal{D}f$.

Given a linear map $f' :: U \multimap V$ represented as a function, it is possible to extract a Jacobian matrix by applying f to every vector in a basis of U.

Extracting a Data Representation

In machine learning, a Gradient-based optimization works by searching for local minima in the domain of a differentiable function $f :: a \to s$. Each step in the search is in the direction opposite of the gradient of f, which is a vector form of $\mathcal{D}f$.

Given a linear map $f' :: U \multimap V$ represented as a function, it is possible to extract a Jacobian matrix by applying f to every vector in a basis of U.

Generalized Matrices

Given a scalar field s, a free vector space has the form $p \to s$ for some p, where the cardinality of p is the dimension of the vector space and there exists a finite number of values for p.

In particular, we can represent vector spaces over a given field as a representable functor, i.e., a functor F such that $\exists p\, \forall s\; F\ s \cong p \to s$.

Generalized Matrices

Given a scalar field s, a free vector space has the form $p \to s$ for some p, where the cardinality of p is the dimension of the vector space and there exists a finite number of values for p.

In particular, we can represent vector spaces over a given field as a representable functor, i.e., a functor F such that $\exists p\, \forall s\; F\ s \cong p \to s$.

A short introdution

- In chapter 4 we've derived an AD algorithm that was generalized in figure 6 of the document
- With fully right-associated compositions this algorithm becomes a foward-mode AD and with fully left-associated becomes a reverse-mode AD
- We want to obtain generalized FAD and RAD algorithms
- How do we describe this in Categorical notation?

A short introdution

- In chapter 4 we've derived an AD algorithm that was generalized in figure 6 of the document
- With fully right-associated compositions this algorithm becomes a foward-mode AD and with fully left-associated becomes a reverse-mode AD
- We want to obtain generalized FAD and RAD algorithms
- How do we describe this in Categorical notation?

A short introdution

- In chapter 4 we've derived an AD algorithm that was generalized in figure 6 of the document
- With fully right-associated compositions this algorithm becomes a foward-mode AD and with fully left-associated becomes a reverse-mode AD
- We want to obtain generalized FAD and RAD algorithms
- How do we describe this in Categorical notation?

A short introdution

- In chapter 4 we've derived an AD algorithm that was generalized in figure 6 of the document
- With fully right-associated compositions this algorithm becomes a foward-mode AD and with fully left-associated becomes a reverse-mode AD
- We want to obtain generalized FAD and RAD algorithms
- How do we describe this in Categorical notation?

Converting morfisms

Given a category k we can represent its morfisms the following way:

Left-Compose functions

f :: a'k'b $\Rightarrow$ ($\circ$ f) :: (b'k'r) $\rightarrow$ (a'k'r) where r is any object of k.

If h is the morfism we'll compose with f then h is the continuation of f.

Instance deduction

Defining new type

$$\textbf{newtype}\ Cont\ (k, r)\ a\ b = Cont\ ((b'k'r) \rightarrow (a'k'r))$$

Functor derived from type

$$cont :: Category\ k \Rightarrow (a\ '\mathrm{k}'\ \ b) \rightarrow Cont\ (k, r)\ a\ b$$
$$cont\ f = Cont\ (\circ f)$$

Instance deduction

$$
\begin{aligned}
&\textbf{instance } \mathit{Category}\ k \Rightarrow \mathit{Category}\ \mathit{Cont}\ (k,r)\ \textbf{where}\\
&\quad \mathit{id} = \mathit{Cont}\ \mathit{id}\\
&\quad \mathit{Cont}\ g \circ \mathit{Cont}\ f = \mathit{Cont}\ (f \circ g)\\
&\textbf{instance } \mathit{Monoidal}\ k \Rightarrow \mathit{Monoidal}\ \mathit{Cont}\ (k,r)\ \textbf{where}\\
&\quad \mathit{Conf}\ f\ x\ \mathit{Cont}\ g = \mathit{Cont}\ (\nabla \circ (f\ x\ g) \circ \mathit{unjoin})\\
&\textbf{instance } \mathit{Cartesian}\ k \Rightarrow \mathit{Cartesian}\ \mathit{Cont}\ (k,r)\ \textbf{where}\\
&\quad \mathit{exl} = \mathit{Cont}\ (\nabla \circ \mathit{inl});\ \mathit{exr} = \mathit{Cont}\ (\nabla \circ \mathit{inr})\\
&\quad \mathit{dup} = \mathit{Cont}\ (\mathit{jam} \circ \mathit{unjoin})\\
&\textbf{instance } \mathit{Cocartesian}\ k \Rightarrow \mathit{Cocartesian}\ \mathit{Cont}\ (k,r)\ \textbf{where}\\
&\quad \mathit{inl} = \mathit{Cont}\ (\mathit{exl} \circ \mathit{unjoin});\ \mathit{inr} = \mathit{Cont}\ (\mathit{exr} \circ \mathit{unjoin})\\
&\quad \mathit{jam} = \mathit{Cont}\ (\nabla \circ \mathit{dup})\\
&\textbf{instance } \mathit{Scalable}\ k\ a \Rightarrow \mathit{Scalable}\ \mathit{Cont}\ (k,r)\ a\ \textbf{where}\\
&\quad \mathit{scale}\ s = \mathit{Cont}\ (\mathit{scale}\ s)
\end{aligned}
$$

A short introdution

Due to it's widespread use in ML we'll talk about a specific case of RAD: computing gradients(derivatives of functions with scalar codomains)
A vector space A over a scalar field has A $\multimap$ s as its dual.
Each linear map in A $\multimap$ s can be represented in the form of dot u for some u :: A where

Definition and instanciation

$$\textbf{class } \mathit{HasDot}\ (S)\ u \textbf{ where } \mathit{dot} :: u \rightarrow (u - \circ\ s)$$

$$\textbf{instance } \mathit{HasDot}\ (\mathit{IR})\ \mathit{IR} \textbf{ where } \mathit{dot} = \mathit{scale}$$

$$\textbf{instance } (\mathit{HasDot}\ (S)\ a, \mathit{HasDot}\ (S)\ b) \Rightarrow \mathit{HasDot}\ (S)\ (a\ x\ b) \textbf{ where } \mathit{dot}\ (u, v) = \mathit{dot}\ u\ \Delta\ \mathit{dot}\ v$$

Instance deduction

The internal representation of $Cont^{s}_{\multimap}$ a b is (b $\multimap$ s) -> (a $\multimap$ s) which is isomorfic to (a -> b).

Type definition for duality

$$\textbf{newtype}\ Dual\ (K)\ a\ b = Dual\ (b'k'a)$$

Instance deduction

All we need to do to create dual representations of linear maps is to convert from $Cont_k^S$ to $Dual_k$ using a functor:

Functor definition

$$
\begin{aligned}
&asDual :: (HasDot\ (S)\ a, HasDot\ (S)\ b) \Rightarrow \\
&\quad ((b -\circ s) \to (a -\circ s)) \to (b -\circ a) \\
&asDual\ (Cont\ f) = Dual\ (onDot\ f)
\end{aligned}
$$

where

$$
\begin{aligned}
&onDot :: (HasDot\ (S)\ a, HasDot\ (S)\ b) \Rightarrow \\
&\quad ((b -\circ s) \to (a -\circ s)) \to (b -\circ a) \\
&onDot\ f = dot\ \hat{}- 1 \circ f \circ dot
\end{aligned}
$$

Instance deduction

instance $Category\ k \Rightarrow Category\ Dual\ (k)$ **where**
$id = Dual\ id$
$Dual\ g \circ Dual\ f = Dual\ (f \circ g)$
instance $Monoidal\ k \Rightarrow Monoidal\ Dual\ (k)$ **where**
$Dual\ f\ x\ Dual\ g = Dual\ (f\ x\ g)$
instance $Cartesian\ k \Rightarrow Cartesian\ Dual\ (k)$ **where**
$exl = Dual\ inl; exr = Dual\ inr; dup = Dual\ jam$
instance $Cocartesian\ k \Rightarrow CocartesianDual\ (k)$ **where**
$inl = Dual\ exl; inr = Dual\ exr; jam = Dual\ dup$
instance $Scalable\ k \Rightarrow Scalable\ Dual\ (k)$ **where**
$scale\ s = Dual\ (scale\ s)$

Final notes

- ∇ *and* Δ mutually dualize ($Dual\ f \nabla Dual\ g = Dual\ (f \Delta g)$ *and* $Dual\ f \Delta Dual\ g = Dual\ (f \nabla g)$)
- Using the definition from chapter 8 we can determine that the duality of a matrix corresponds to it's transposition

Fowards-mode Automatic Differentiation(FAD)

We can use the same deductions we've done in Cont and Dual to derive a category with full right-side association, thus creating a generized FAD algorithm.
This algorithm is far more apropriated for low dimention domains.

Type definition and functor from type

$$\textbf{newtype}\ \mathit{Begin}\ (k, r)\ a\ b = \mathit{Begin}\ ((r'k'a) \rightarrow (r'k'b))$$
$$\mathit{begin} :: \mathit{Category}\ k \Rightarrow (a\ '\mathrm{k}'\ \ b) \rightarrow \mathit{Begin}\ (k, r)\ a\ b$$
$$\mathit{begin}\ f = \mathit{Begin}\ (f\circ)$$

We can derive categorical instances from the functor above and we can choose r to be the scalar field s, noting that s $\multimap$ a is isomorfic to a.

- Practical applications often involves high-dimensional spaces.
- Binary products are a very inefficient and unwieldy way of encoding high-dimensional spaces.
- A practical alternative is to consider n-ary products as representable functors(?)

$$
\begin{array}{l}
\textbf{class}\ \mathit{Category}\ k \Rightarrow \mathit{MonoidalI}\ k\ h\ \textbf{where} \\
\quad \mathit{crossI} :: h\ (a\ \text{`}\ k\ \text{`}\ b) \rightarrow (h\ a\ \text{`}\ k\ \text{`}\ h\ b) \\
\textbf{instance}\ \mathit{Zip}\ h \Rightarrow \mathit{MonoidalI}\ (\rightarrow)\ h\ \textbf{where} \\
\quad \mathit{crossI} = \mathit{zipWith}\ \mathit{id}
\end{array}
$$

$$\begin{aligned}
&\textbf{class } \mathit{MonoidalI}\ k\ h \Rightarrow \mathit{CartesianI}\ k\ h \textbf{ where}\\
&\quad \mathit{exl}\ \ :: h\ (h\ a\ `k`\ a)\\
&\quad \mathit{replI} :: a\ `k`\ h\ a
\end{aligned}$$

$$\begin{aligned}
&\textbf{class } (\mathit{Representable}\ h, \mathit{Zip}\ h, \mathit{Pointed}\ h) \Rightarrow\\
&\quad \mathit{CartesianI}\ (\to)\ h \textbf{ where}\\
&\quad \mathit{exl} = \mathit{tabulate}\ (\mathit{flip}\ \mathit{index})\\
&\quad \mathit{replI} = \mathit{point}
\end{aligned}$$

- The following is not the class the author was thinking

$$\begin{aligned}
&\textbf{class } \mathit{Representable}\ h \textbf{ where}\\
&\quad \textbf{type } \mathit{Rep}\ h :: *\\
&\quad \mathit{tabulate} :: (\mathit{Rep}\ h \to a) \to h\ a\\
&\quad \mathit{index} :: h\ a \to \mathit{Rep}\ h \to a
\end{aligned}$$

$$
\begin{array}{l}
\textbf{class } \mathit{MonoidalI}\ k\ h \Rightarrow \mathit{CocartesianI}\ k\ h \textbf{ where} \\
\quad \mathit{inI} :: h\ (a\ `k`\ h\ a) \\
\quad \mathit{jamI} :: h\ a\ `k`\ a \\
\textbf{instance } (\mathit{MonoidalI}\ k\ h, \mathit{Zip}\ h) \Rightarrow \mathit{MonoidalI}\ D_k\ h \textbf{ where} \\
\quad \mathit{crossI}\ \mathit{fs} = D\ ((\mathit{id} \times \mathit{crossI}) \circ \mathit{unzip} \circ \mathit{crossI}\ (\mathit{fmap}\ \mathit{unD}\ \mathit{fs})) \\
\textbf{instance } (\mathit{CocartesianI}\ (\rightarrow)\ h, \mathit{CartesianI}\ k\ h, \mathit{Zip}\ h) \Rightarrow \\
\quad \mathit{CartesianI}\ D_k\ h \textbf{ where} \\
\quad \mathit{exI} = \mathit{linearD}\ \mathit{exI}\ \mathit{exI} \\
\quad \mathit{replI} = \mathit{zipWith}\ \mathit{linearD}\ \mathit{replI}\ \mathit{replI} \\
\textbf{instance } (\mathit{CocartesianI}\ k\ h, \mathit{Zip}\ h) \Rightarrow \mathit{CocartesianI}\ D_k\ h \textbf{ where} \\
\quad \mathit{inI} = \mathit{zipWith}\ \mathit{linearD}\ \mathit{inIF}\ \mathit{inI} \\
\quad \mathit{jamI} = \mathit{linearD}\ \mathit{sum}\ \mathit{jamI}
\end{array}
$$

- Suggests that some of the work refered does just a part of this paper.
- This paper was a continuation of the [Elliot 2017]
- Suggests that this implementation is simple, efficient, it can free memory dinamically (RAD) and is naturally parallel.
- Future work are detailed performace analysis; higher-order differentiation and automatic incrementation (continuing previous work [Elliot 2017])