

# Kotlin ▽

A Shape Safe eDSL for Differentiable Functional Programming

Breandan Considine

Université de Montréal

*breandan.considine@umontreal.ca*

May 25, 2019

# Overview

- 1 A Short Lesson on Computing Derivatives
- 2 Introduction and motivation
- 3 Architectural Overview
- 4 Usage
- 5 Future exploration

If we have a function,  $P(x) : \mathbb{R} \rightarrow \mathbb{R}$ , recall the derivative is defined as:

$$P'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} = \frac{\Delta y}{\Delta x} = \frac{dP}{dx} \quad (1)$$

For  $P(x_0, x_1, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}$ , the gradient is a vector of derivatives:

$$\nabla P = \left[ \frac{\partial P}{\partial x_0}, \frac{\partial P}{\partial x_1}, \dots, \frac{\partial P}{\partial x_n} \right] \text{ where } \frac{\partial P}{\partial x_i} = \frac{dP}{dx_i} \quad (2)$$

For  $\mathbf{P}(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , the Jacobian is a vector of gradients:

$$\mathbf{J}_P = [\nabla P_0, \nabla P_1, \dots, \nabla P_n] \text{ or equivalently, } \mathbf{J}_{ij} = \frac{\partial P_i}{\partial x_j} \quad (3)$$

# Automatic differentiation

Suppose we have a scalar function  $P_k : \mathbb{R} \rightarrow \mathbb{R}$  such that:

$$P_k(x) = \begin{cases} p_0(x) = x & \text{if } k = 0 \\ (p_k \circ P_{k-1})(x) & \text{if } k > 0 \end{cases}$$

From the chain rule of calculus, we know that:

$$\frac{dP}{dp_0} = \frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \cdots \frac{dp_1}{dp_0} = \prod_{i=1}^k \frac{dp_i}{dp_{i-1}}$$

For a vector function  $\mathbf{P}_k(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , the chain rule still applies:

$$\mathbf{J}_{\mathbf{P}} = \prod_{i=1}^k \mathbf{J}_{p_i} = \underbrace{\left( \left( (\mathbf{J}_{p_k} \mathbf{J}_{p_{k-1}}) \cdots \mathbf{J}_{p_2} \right) \mathbf{J}_{p_1} \right)}_{\text{"Reverse accumulation"}} = \underbrace{\left( \mathbf{J}_{p_k} \left( \mathbf{J}_{p_{k-1}} \cdots (\mathbf{J}_{p_2} \mathbf{J}_{p_1}) \right) \right)}_{\text{"Forward accumulation"}}$$

If  $\mathbf{P}_k$  were a program, what would the type signature of  $\mathbf{p}_{0 < i < k}$  be?

$$\mathbf{p}_i : \mathcal{T}_{out}(\mathbf{p}_{i-1}) \rightarrow \mathcal{T}_{in}(\mathbf{p}_{i+1})$$

# Parameter learning and gradient descent

For parametric models, let us rewrite  $\mathbf{P}_k(\mathbf{x})$  as:

$$\hat{\mathbf{P}}_k(\mathbf{x}; \Theta = [\theta_0, \dots, \theta_k]) = \begin{cases} \mathbf{p}_0(\mathbf{x}; \theta_0) & \text{if } k = 0 \\ (\mathbf{p}_k(\theta_k) \circ \hat{\mathbf{P}}_{k-1}(\Theta))(\mathbf{x}) & \text{if } k > 0 \end{cases}$$

Where  $\Theta \in \mathbb{R}^{\max(n,m) \times k}$  are free parameters and  $\mathbf{x} \in \mathbb{R}^n$  is a single input. Given  $\mathbf{Y} = [\mathbf{y}^{(1)} = \mathbf{P}(\mathbf{x}^{(1)}), \dots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})]$  from an oracle, in order to approximate  $\mathbf{P}(\mathbf{x})$ , repeat the following procedure until  $\Theta$  converges:

$$\Theta \leftarrow \Theta - \frac{1}{z} \nabla_{\Theta} \sum_{i=0}^z \mathcal{L}(\hat{\mathbf{P}}_k(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})$$

If  $\hat{\mathbf{P}}_k$  were a program, what would the type signature of  $\mathbf{p}_{0 < i < k}$  be?

$$\mathbf{p}_i : \mathcal{T}_{out}(\mathbf{p}_{i-1}) \times \mathcal{T}(\theta_i) \rightarrow \mathcal{T}_{in}(\mathbf{p}_{i+1}(\theta_{i+1}))$$

# Shape checking and inference

- Scalar functions implicitly represent shape as arity  $f(1, 2) : \mathbb{R}^2 \rightarrow \mathbb{R}$
- To check array programs, we need a type-level encoding of shape
- Arbitrary operations (e.g. convolution) may require dependent types
- But parametric polymorphism will suffice for most tensor functions
- For most algebraic operations, we just need to check for equality. . .

Math	Derivative	Code	Type Signature
$a(b)$	$\mathbf{J}_a \mathbf{J}_b$	<code>a(b)</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\tau) \rightarrow (\mathbb{R}^\lambda \rightarrow \mathbb{R}^\pi)$
$a + b$	$\mathbf{J}_a + \mathbf{J}_b$	<code>a + b</code> <code>a.plus(b)</code> <code>plus(a, b)</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\pi) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R}^\pi)$
$ab$	$\mathbf{J}_a b + \mathbf{J}_b a$	<code>a * b</code> <code>a.times(b)</code> <code>times(a, b)</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times n}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^{n \times p}) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times p})$
$a^b$	$a^b(a' \frac{b}{a} + b' \ln a)$	<code>a.pow(b)</code> <code>pow(a, b)</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R})$

- Abstract algebra can be useful when generalizing to new structures
- Helps us to easily translate between mathematics and source code
- Fields are a useful concept when computing over real numbers
  - A field is a set  $\mathbb{F}$  with two operations  $+$  and  $\times$ , with the properties:
    - Associativity:  $\forall a, b, c \in \mathbb{F}, a + (b + c) = (a + b) + c$
    - Commutivity:  $\forall a, b \in \mathbb{F}, a + b = b + a$  and  $a \times b = b \times a$
    - Distributivity:  $\forall a, b, c \in \mathbb{F}, a \times (b + c) = (a \times b) + (a \times c)$
    - Identity:  $\forall a \in \mathbb{F}, \exists 0, 1 \in \mathbb{F}$  s.t.  $a + 0 = a$  and  $a \times 1 = a$
    - $+$  inverse:  $\forall a \in \mathbb{F}, \exists (-a)$  s.t.  $a + (-a) = 0$
    - $\times$  inverse:  $\forall a \neq 0 \in \mathbb{F}, \exists (a^{-1})$  s.t.  $a \times a^{-1} = 1$
- Extensible to other number systems (e.g. complex, dual numbers)
- What is a program, but a series of arithmetic operations?

# Why Kotlin?



- Goal: To implement automatic differentiation in Kotlin
- Kotlin is a language with strong static typing and null safety
- Supports first-class functions, higher order functions and lambdas
- Has support for algebraic data types, via tuples sealed classes
- Extension functions, operator overloading other syntax sugar
- Offers features for embedding domain specific languages (DSLs)
- Access to all libraries and frameworks in the JVM ecosystem
- Multi-platform and cross-platform (JVM, Android, iOS, JS, native)





- Type system
  - Strong type system based on algebraic principles
  - Leverage the compiler for static analysis
  - No implicit broadcasting or shape coercion
  - Parameterized numerical types and arbitrary-precision
- Design principles
  - Functional programming and lazy numerical evaluation
  - Eager algebraic simplification of expression trees
  - Operator overloading and tapeless reverse mode AD
- Usage desiderata
  - Generalized AD with functional array programming
  - Automatic differentiation with infix and Polish notation
  - Partial derivatives and higher order derivatives and gradients
- Testing and validation
  - Numerical gradient checking and property-based testing
  - Performance benchmarks and thorough regression testing

# Feature Comparison Matrix

Framework	Language	SD	AD	FP	TS	SS	DP	MP
Kotlin $\nabla$	Kotlin	✓	✓	✓	✓	✓		
DiffSharp	F#	✗	✓	✓	✓	✗	✓	✗
TensorFlow.FSharp	F#	✗	✓	✓	✓	✓	✓	✗
Myia	Python	✓	✓	✓	✓	✓	✓	✗
Deeplearning.scala	Scala	✗	✓	✓	✓	✗	✓	✗
Nexus	Scala	✗	✓	✓	✓	✓	✓	✗
Lantern	Scala	✗	✓	✓	✓	✗	✓	✗
Grenade	Haskell	✗	✓	✓	✓	✓	✗	✗
Eclipse DL4J	Java	✗	✓	✗	✓	✗	✗	✗
Halide	C++	✗	✓	✗	✓	✗	✓	✗
Stalin	Scheme	✗	✓	✓	✗	✗	✗	✗

SD: Symbolic Differentiation, AD: Automatic Differentiation, FP: Functional Program, TS: Type Safe, SS: Shape Safe, DP: Differentiable Programming, MP: Multiplatform

# How do we define algebraic types in Kotlin ▽?

// T: Group<T> is effectively a self type

```
interface Group<T: Group<T>> {  
    operator fun plus(f: T): T  
    operator fun unaryMinus(): X  
    operator fun minus(f: X): X = this + -f  
    operator fun times(f: T): T  
}
```

```
interface Field<X: Group<X>> {  
    val e: X  
    val one: X  
    val zero: X  
    operator fun div(f: X): X = this * f.pow(-one)  
    infix fun pow(f: X): X  
    fun ln(): X  
}
```

# Algebraic Data Types

```
class Var<X: Fun<X>>(label: String): Fun<X>()  
class Const<X: Fun<X>>(val num: Number): Fun<X>()  
class Sum<X: Fun<X>>(val f1: X, val f2: X): Fun<X>()  
class Prod<X: Fun<X>>(val f1: X, val f2: X): Fun<X>()
```

```
sealed class Fun<X: Fun<X>>: Field<Fun<X>>{  
    open fun diff(): Fun<X> = when(this) {  
        is Const -> Zero  
        is Sum -> f1.diff() + f2.diff()  
        is Prod -> f1.diff() * f2 + f1 * f2.diff()  
        is Var -> One  
    }  
}
```

```
operator fun plus(f: Fun<X>) = Sum(this, f)  
operator fun times(f: Fun<X>) = Prod(this, f)  
}
```

# Expression simplification

```
operator fun times(exp: Fun<X>): Fun<X> = when {  
    this is Const && num == 0.0 -> Const(0.0)  
    this is Const && num == 1.0 -> exp  
    exp is Const && exp.num == 0.0 -> exp  
    exp is Const && exp.num == 1.0 -> this  
    this is Const && exp is Const -> Const(num*exp.num)  
    else -> Prod(this, e)  
}
```

```
// Sum(Prod(Const(2.0), Var()), Const(6.0))  
val q = Const(2.0) * Sum(Var(), Const(3.0))
```

# Extension functions and contexts

```
object DoublePrecision {  
    operator fun Number.times(f: Fun<KDouble>) =  
        Const(toDouble()) * f  
}
```

```
class KDouble(num: Double): Const<KDouble>(num) {  
    override val e by lazy { KDouble(Math.E) }  
    override val one by lazy { KDouble(1.0) }  
    override val zero by lazy { KDouble(0.0) }  
    // Adapters for wrapping primitive Double...  
}
```

```
// Uses '*' operator in DoubleContext  
fun Fun<KDouble>.multiplyByTwo() =  
    with(DoublePrecision) { 2 * this }
```

# Automatic test case generation

```
val x = Var("x")
val y = Var("y")

val z = y * (sin(x * y) - x) // Function under test
val dz_dx = d(z) / d(x)      // Automatic derivative
val manualDx = y * (cos(x * y) * y - 1)

"dz/dx should be y * (cos(x * y) * y - 1)" {
    assertAll (DoubleGenerator) { cx, cy ->
        // Evaluate the results at a given seed
        val autoEval = dz_dx(x to cx, y to cy)
        val symbEval = manualDx(x to cx, y to cy)
        // Should pass if |adEval - manualEval| < eps
        autoEval shouldBeApproximately symbEval
    }
}
```

## Usage: plotting higher derivatives of nested functions

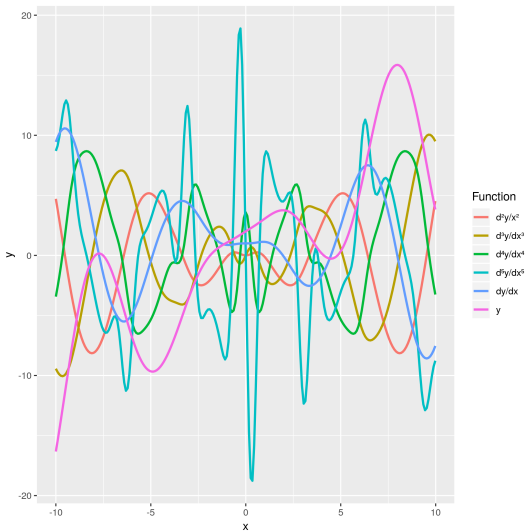
```
with(DoublePrecision) { // Use double-precision
    val x = variable() // Declare an immutable variable
    val y = sin(sin(sin(x)))/x + sin(x) * x + cos(x) + x

    // Lazily compute reverse-mode automatic derivatives
    val dy_dx = d(y) / d(x)
    val d2y_dx = d(dy_dx) / d(x)
    val d3y_dx = d(d2y_dx) / d(x)
    val d4y_dx = d(d3y_dx) / d(x)
    val d5y_dx = d(d4y_dx) / d(x)

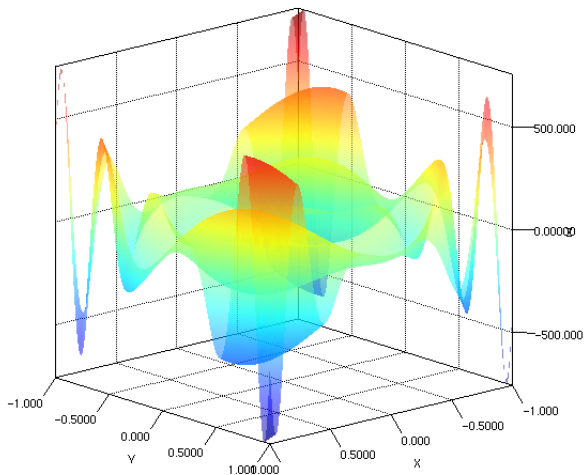
    plot(-10..10, dy_dx, dy2_dx, d3y_dx, d4y_dx, d5y_dx)
}
```



$$y = \frac{\sin \sin \sin x}{x} + x \sin x + \cos x + x, \quad \frac{dy}{dx}, \quad \frac{d^2y}{dx^2}, \quad \frac{d^3y}{dx^3}, \quad \frac{d^4y}{dx^4}, \quad \frac{d^5y}{dx^5}$$



$$z = \sin(10(x^2 + y^2))/10, \quad \frac{\partial^3 z}{\partial^2 y \partial x}$$



# Further directions to explore

## • Theory Directions

- Generalization of types to higher order functions, vector spaces
- Dependent types via code generation to type-check convolution
- General programming operators and data structures
- Imperative define-by-run array programming syntax
- Program induction and synthesis, cf.
  - The Derivative of a Regular Type is its Type of One-Hole Contexts
  - The Differential Lambda Calculus (2003)
- Asynchronous gradient descent (cf. HogWild, YellowFin, et al.)

## • Implementation Details

- Closer integration with Kotlin/Java standard library
- Encode additional structure, i.e. function arity into type system
- Vectorized optimizations for matrices with certain properties
- Configurable forward and backward AD modes based on dimension
- Automatic expression refactoring for numerical stability
- Primitive type specialization, i.e. `FloatVector <: Vector<T>?`

Learn more at:

<http://kg.ndan.co>

Liam Paull  
Michalis Famelis



Université  de Montréal

