Kotlin∇

A Shape Safe eDSL for Differentiable Functional Programming

Breandan Considine

Université de Montréal

breandan.considine@umontreal.ca

May 22, 2019

Overview

Differentiation

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

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

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

$$\nabla P = \left[\frac{\partial P}{\partial x_0}, \frac{\partial P}{\partial x_1}, \ldots, \frac{\partial P}{\partial x_n}\right] \text{ where } \frac{\partial P}{\partial x_i} = \frac{dP}{dx_i} \tag{2}$$

For $\mathbf{P}(x_0, x_1, \ldots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the Jacobian is a vector of gradients:

$$\mathbf{J_P} = [\nabla P_0, \nabla P_1, \ldots, \nabla P_n] \text{ or equivalently, } \mathbf{J}_{ij} = \frac{\partial P_i}{\partial x_j} \tag{3}$$

Type checking automatic differentiation

Suppose we have a program $P : \mathbb{R} \rightarrow \mathbb{R}$ where:

$$\mathbf{P}(p_0) = p_q \circ p_{q-1} \circ p_{q-2} \circ \cdots \circ p_1 \circ p_0 \tag{4}$$

From the chain rule of calculus, we know that:

$$\frac{dP}{dp_0} = \frac{dp_q}{dp_{q-1}} \frac{dp_{q-1}}{dp_{q-2}} \cdots \frac{dp_1}{dp_0} = \prod_{i=1}^{n} \frac{dp_i}{dp_{i-1}} \tag{5}$$

More generally, for $P : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the chain rule also applies:

$$\mathbf{J_P} = \prod_{i=1}^{q} \mathbf{J}_{p_i} = \left(\left((\mathbf{J}_{p_q}\mathbf{J}_{p_{q-1}}) \ldots \mathbf{J}_{p_2}\right)\mathbf{J}_{p_1}\right) = \left(\mathbf{J}_{p_q}\left(\mathbf{J}_{p_{n-1}} \ldots (\mathbf{J}_{p_2}\mathbf{J}_{p_1})\right)\right) \tag{6}$$

In order for **P** to type check, what is the type signature of $p_{0<i<n}$?

$$p_i : T_{out}(p_{i-1}) \rightarrow T_{in}(p_{i+1}) \tag{7}$$

Shape checking and inference

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

Math	Derivative	Code	Type Signature
$a(b)$	$\mathbf{J}_a\mathbf{J}_b$	`a(b)`	$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^\pi, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}^\tau) \to (\mathbb{R}^\lambda \to \mathbb{R}^\pi)$

$a + b$	$\mathbf{J}_a + \mathbf{J}_b$	`a + b`
		`a.plus(b)`
		`plus(a, b)`
			$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^\pi, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}^\pi) \to (\mathbb{R}^? \to \mathbb{R}^\pi)$

ab	$\mathbf{J}_a b + \mathbf{J}_b b$	`a * b`
		`a.times(b)`
		`times(a, b)`
			$(\mathtt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m \times n}, \mathtt{b} : \mathbb{R}^\lambda \to \mathbb{R}^{n \times p}) \to (\mathbb{R}^? \to \mathbb{R}^{m \times p})$

Numeric tower

- 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 F with two operations $+$ and $\times$, with the properties:
    - Associativity: $\forall a, b, c \in F, a + (b + c) = (a + b) + c$
    - Commutivity: $\forall a, b \in F, a + b = b + a$ and $a \times b = b \times a$
    - Distributivity: $\forall a, b, c \in F, a \times (b \times c) = (a \times b) \times c$
    - Identity: $\forall a \in F, \exists 0, 1 \in F$ s.t. $a + 0 = a$ and $a \times 1 = a$
    - $+$ inverse: $\forall a \in F, \exists(-a)$ s.t. $a + (-a) = 0$
    - $\times$ inverse: $\forall a \neq 0 \in F, \exists(a^{-1})$ 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)

Kotlin∇ Priorities

- 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 arbitary-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
  - Partials and higher order derivatives and gradients
- Testing and validation
  - Numerical gradient checking and property-based testing
  - Performance benchmarks and thorough regression testing

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 times(f: T): T
}

// Inherits from Group, default methods
interface Field<T: Field<T>>: Group<T> {
  operator fun unaryMinus(): T
  operator fun minus(f: T): T = this + -f
  fun inverse(): T
  operator fun div(f: T): T = this * f.inverse()
}
```

Algebraic Data Types

```
class Var: Expr()
class Const(val num: Number): Expr()
class Sum(val e1: Expr, val e2: Expr): Expr()
class Prod(val e1: Expr, val e2: Expr): Expr()

sealed class Expr: Group {
  fun diff() = when(expr) {
    is Const -> Zero
    is Sum -> e1.diff() + e2.diff()
    is Prod -> e1.diff() * e2 + e1 * e2.diff()
    is Var -> One
  }

  operator fun plus(e: Expr) = Sum(this, e)
  operator fun times(e: Expr) = Prod(this, e)
}
```

Expression simplification

```
operator fun Expr.times(exp: Expr) = 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

```
class Expr<T: Group<T>>: Group<Expr<T>> {
  //...
  operator fun plus(exp: Expr<T>) = Sum(this, exp)
  operator fun times(exp: Expr<T>) = Prod(this, exp)
}

object DoubleContext {
  operator fun Number.times(exp: Expr<DoubleReal>) =
    Const(toDouble()) * exp
}

// Uses `*` operator in DoubleContext
fun Expr<DoubleReal>.multiplyByTwo() =
  with(DoubleContext) { 2 * this }
```

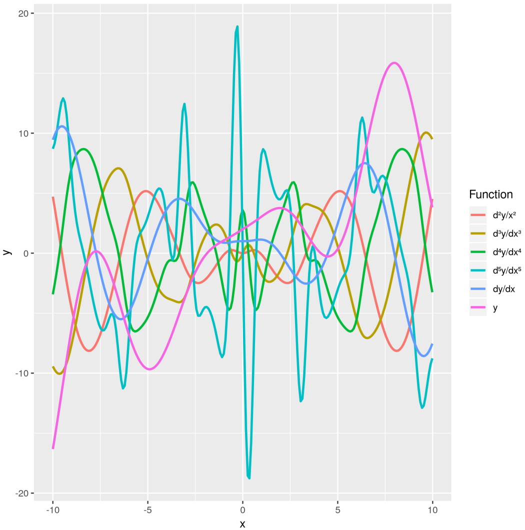
Automatic test case generation

```
val x = variable("x")
val y = variable("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 (NumGen) { 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
  }
}
```

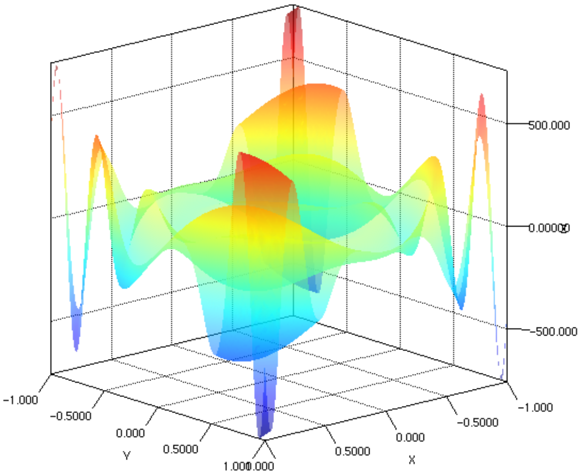
Usage: plotting higher derivatives of nested functions

```
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_dx2) / d(x)
 val d4y_dx = d(d3y_dx3) / d(x)
 val d5y_dx = d(d4y_dx4) / d(x)

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

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



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



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`

Special thanks

Liam Paull

Michalis Famelis

Université de Montréal

Mila