Kotlin∇

A Shape Safe eDSL for Differentiable Functional Programming

Breandan Considine

McGill University

breandan.considine@mcgill.ca

September 5, 2019

Overview

Differentiation

If we have a function, $P(x) : \mathbb{R} \to \mathbb{R}$, 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 $P(x_0, x_1, \ldots, x_n) : \mathbb{R}^n \to \mathbb{R}$, 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 $\mathbf{P}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian is a vector of gradients:

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

Automatic differentiation

Suppose we have a scalar function $P_k : \mathbb{R} \to \mathbb{R}$ 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:

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

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

$$\mathcal{J}_{\mathbf{P_k}} = \prod_{i=1}^{k} \mathcal{J}_{p_i} = \underbrace{\left(\left((\mathcal{J}_{p_k}\mathcal{J}_{p_{k-1}}) \dots \mathcal{J}_{p_2}\right)\mathcal{J}_{p_1}\right)}_{\text{“Reverse accumulation”}} = \underbrace{\left(\mathcal{J}_{p_k}\left(\mathcal{J}_{p_{k-1}} \dots (\mathcal{J}_{p_2}\mathcal{J}_{p_1})\right)\right)}_{\text{“Forward accumulation”}}$$

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

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

Parameter learning and gradient descent

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

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

Where $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_k\}$ are free parameters and $\mathbf{x} \in \mathbb{R}^n$ is a single input. Given $\mathbf{Y} = \{\mathbf{y}^{(1)} = \mathbf{P}(\mathbf{x}^{(1)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$ from an oracle, in order to approximate $\mathbf{P}(\mathbf{x})$, repeat the following procedure until $\boldsymbol{\Theta}$ converges:

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

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

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

Why differentiable programming?

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

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

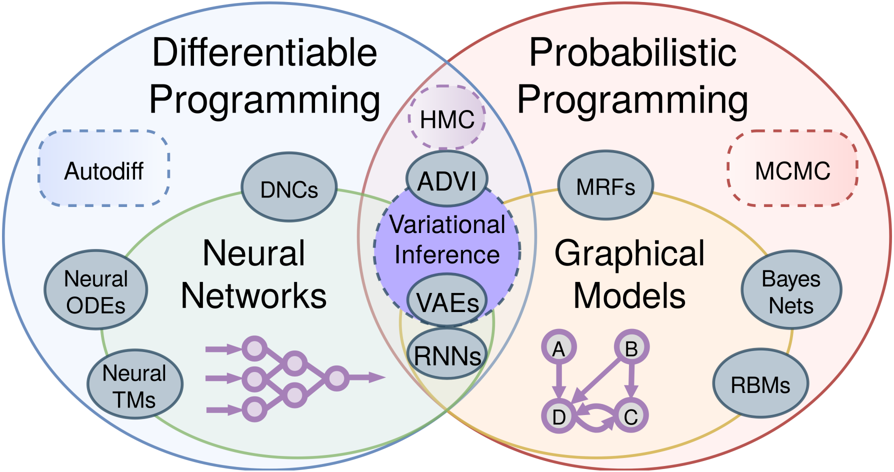
Where $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_k\}$ are free parameters and $\mathbf{x} \in \mathbb{R}^n$ is a single input. Given $\mathbf{Y} = \{\mathbf{y}^{(1)} = \mathbf{P}(\mathbf{x}^{(1)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$ from an oracle, in order to approximate $\mathbf{P}(\mathbf{x})$, repeat the following procedure until $\boldsymbol{\Theta}$ converges:

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

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

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

Why differentiable programming?



Shape checking and inference

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

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

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

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

a^b	$a^b(a' \frac{b}{a} + b' \ln a)$	`a.pow(b)` `pow(a, b)`	$(\mathrm{a} : \mathbb{R}^\tau \rightarrow \mathbb{R}, \mathrm{b} : \mathbb{R}^\lambda \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R})$

Numerical 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 $\mathbb{F}$ 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: $\forall a, b, c \in \mathbb{F}, a \times (b \times c) = (a \times b) \times c$
    - Identity: $\forall a \in \mathbb{F}, \exists 0, 1 \in F$ 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 through 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

Feature Comparison Matrix

Framework	**Language**	SD	AD	FP	TS	SS	DP	MP
Kotlin∇	Kotlin	✓	✓	✓	✓	✓	✍	✍
DiffSharp	F#	✗	✓	✓	✓	✗	✓	✗
TensorFlow.FSharp	F#	✗	✓	✓	✓	✓	✓	✗
Nexus	Scala	✗	✓	✓	✓	✓	✓	✗
Lantern	Scala	✗	✓	✓	✓	✗	✓	✗
Grenade	Haskell	✗	✓	✓	✓	✓	✗	✗
JAutoDiff	Java	✓	✓	✗	✓	✗	✗	✗
Halide	C++	✗	✓	✗	✓	✗	✓	✗
Stalin∇	Scheme	✗	✓	✗	✗	✗	✗	✗
Myia	Python	✓	✓	✓	✗	✗	✓	✍
Autograd	Python	✗	✓	✗	✗	✗	✗	✗
JAX	Python	✗	✓	✓	✗	✗	✓	✍

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

Usage

```
val z = sin(10 * (x * x + pow(y, 2))) / 10
```

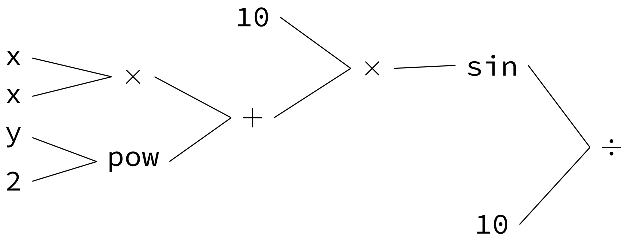


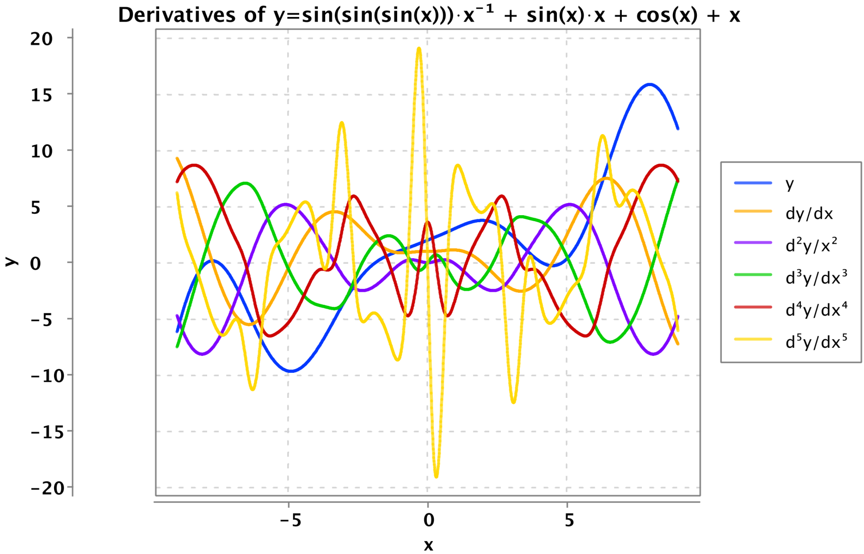
Figure: Implicit DFG constructed by the above expression, z.

Usage: Plotting higher derivatives of nested functions

```
// Use double-precision floating point numerics
with(DoublePrecision) {
  val x = Var()
  val y = sin(sin(sin(x)))/x + x*sin(x) + cos(x) + x

  // Perform lazy symbolic differentiation
  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(-9..9, dy_dx, dy2_dx, d3y_dx, d4y_dx, d5y_dx)
}
```


Derivatives of y=sin(sin(sin(x)))·x⁻¹ + sin(x)·x + cos(x) + x
20
15
10
5
0
−5
−10
−15
−20
y
−5
0
5
x
y
dy/dx
d²y/x²
d³y/dx³
d⁴y/dx⁴
d⁵y/dx⁵

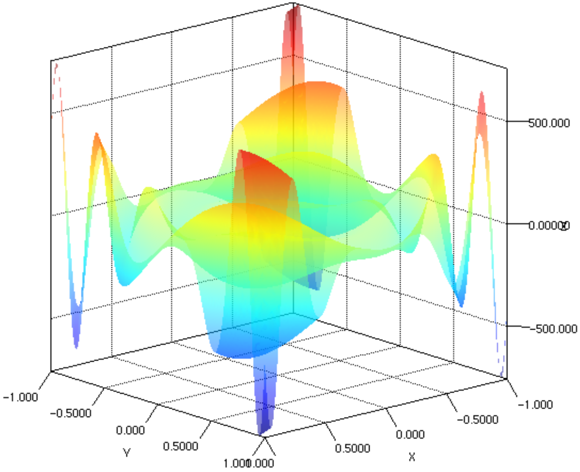
Usage: 3D plotting with mixed higher order partials

```
with(DoublePrecision) {
    val x = Var()
    val y = Var()

    val z = sin(10 * (x * x + pow(y, 2))) / 10
    val dz_dx = d(z) / d(x)
    val d2f_dxdy = d(dz_dx) / d(y)
    val d3z_d2xdy = d(d(dz_dx) / d(y)) / d(x)

    plot3d(-1, 1, d3z_d2xdy)
}
```

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



```
with(DoublePrecision) {
    val q0 = X + Y * Z + Y + 0.0
    val p0 = q(X to 1.0, Y to 2.0, Z to 3.0)
    val p1 = q(X to 1.0, Y to 1.0)(Z to 1.0)
    val p3 = q(Z to 1.0)(X to 1.0, Y to 1.0)
    val p4 = q(Z to 1.0)(X to 1.0)(Y to 1.0)
    val p5 = q(Z to 1.0)(X to 1.0) // Fn<Y>
    val q1 = X + Z + 0.0
    val p6 = q1(Y to 1.0) // Error
}
```

Vector Shape Safety

```
with(DoublePrecision) {             // Inferred type:
    val a = Vec(1.0, 2.0)           // Vec<Double, 2>
    val b = Vec(1.0, 2.0, 3.0)      // Vec<Double, 3>
    val c = b + b                   // Vec<Double, 3>
    val d = a + b                   // Compile error
    val e = b dot b                 // Vec<Double, 1>
    val f = b dot a                 // Compile error
}
```

Matrix Shape Safety

```
// Inferred type: Mat<Double, `1`, `4`>
val a = Mat(1.0, 2.0, 3.0, 4.0)
// Inferred type: Mat<Double, `4`, `1`>
val b = Mat(1.0)(2.0)(3.0)(4.0)
val c = a * b

// Does not compile, inner dimension mismatch
// a * a
// b * b
```

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

Symposium I.A. Montréal

Université de Montréal

Mila