Differentiable Programming in Kotlin

Irene Dea

Differentiable Programming Team


Steffi Stumpos


Melissa Greuter


Samantha Andow


Irene Dea


Johann George


Shannon Yang


Alanna Tempest


Emilio Arroyo-Fang


Christy Warden


Neal Gafter


Xipeng Shen


Olin Shivers

What is Differentiable Programming?

Neural Networks are the beginning of a fundamental shift in how we write software.

- Andrej Karpathy, [Software 2.0]

Differentiable Programming is a shift away from increasingly heavily parameterised [machine learning] models, to ***simpler ones* that take more advantage of problem structure****.*

- Mike Innes, [What Is Differentiable Programming]

Define networks procedurally in a data-dependent way (with loops and conditionals), allowing them to change dynamically as a function of the input data fed to them.

It's really very much like a regular program, except it's parameterized, automatically differentiated, and trainable/optimizable.

- Yann Lecun, Deep Learning est mort. Vive Differentiable Programming!

Motivations

- Popular frameworks are oriented towards traditional ML use cases
- There are many other use cases:
  - Computer graphics
  - Physics simulations
  - Probabilistic programming
  - And many more!

What We Need

- Fast Language
- Automatic Differentiation (AD)
- Memory Safety
- Strong Types
- Static Compilation

Performance
Usability
Flexibility

Our Approach

A compiler-aware framework for differentiation in Kotlin

- Customizable, extensible API
- Compile-time optimizations
- Compile-time shape checking

Why Kotlin?

- Usability
- Speed!
- Compiler Plugins
- Kotlin is *way* more than just Android
- Can target JVM, Native (LLVM), or Javascript

API: derivatives

Our Kotlin API supports scalar and tensor math, including derivatives. Given a Kotlin function, for example *f(x) = sin(x)*

```
fun f(x: DValue) = sin(x)
```

To compute the first derivative *f′(x) = d/dx f(x)* you write

```
fun fp(x: DValue) = derivative(x, ::f)
```

Our derivatives are computed at machine (float) precision, so fp(x) == cos(x).

```
fp(DFloat.PI) shouldBeExactly -1F
```

API: derivatives

We support both forward and reverse differentiation and, by nesting, higher-order derivatives.

```
fun fp1(x: DValue) = forwardDerivative(x, ::f)

fun fp2(x: DValue) = reverseDerivative(x, ::f)

fun fpp(x: DValue) = forwardDerivative(x,
                        { x -> reverseDerivative(x, ::f) })
```

API: more

We support many other components practically needed for AI applications:

- Tensors of arbitrary rank
- Sampling from random distributions
- Slicing, indexing, and concatenating tensors
- Computational layers commonly used for ML applications
    - Dense layer, convolution, norms, loss functions, ...
- Optimizers, Learning loops, …

API: extensibility

Additionally, our API is designed to be customizable and extensible!

- User-defined differentiable types
- Trainable layers and components
- Optimizable by compiler plugins

API: performance

We are performant on a number of different use-cases.

API: performance

We are performant on a number of different use-cases.

- Kotlin
  - Fast custom logic, mixed workloads

API: performance

We are performant on a number of different use-cases.

- Kotlin
  - Fast custom logic, mixed workloads
- MKL-DNN
  - Traditional ML Models

API: performance

We are performant on a number of different use-cases.

- Kotlin
  - Fast custom logic, mixed workloads
- MKL-DNN
  - Traditional ML Models
- Sparse tensors
  - Graph problems, natural language processing, one hot or categorical data
  - Using Eigen library and a TACO-inspired data representation

API: performance

We are performant on a number of different use-cases.

- Kotlin
  - Fast custom logic, mixed workloads
- MKL-DNN
  - Traditional ML Models
- Sparse tensors
  - Graph problems, natural language processing, one hot or categorical data
  - Using Eigen library and a TACO-inspired data representation
- Compile-time optimizations
  - API is designed to be optimizable by a compiler plugin

AD Optimize Plugin

Problem

- AD compute tree is built and stored at runtime
- Scalar operations are boxed

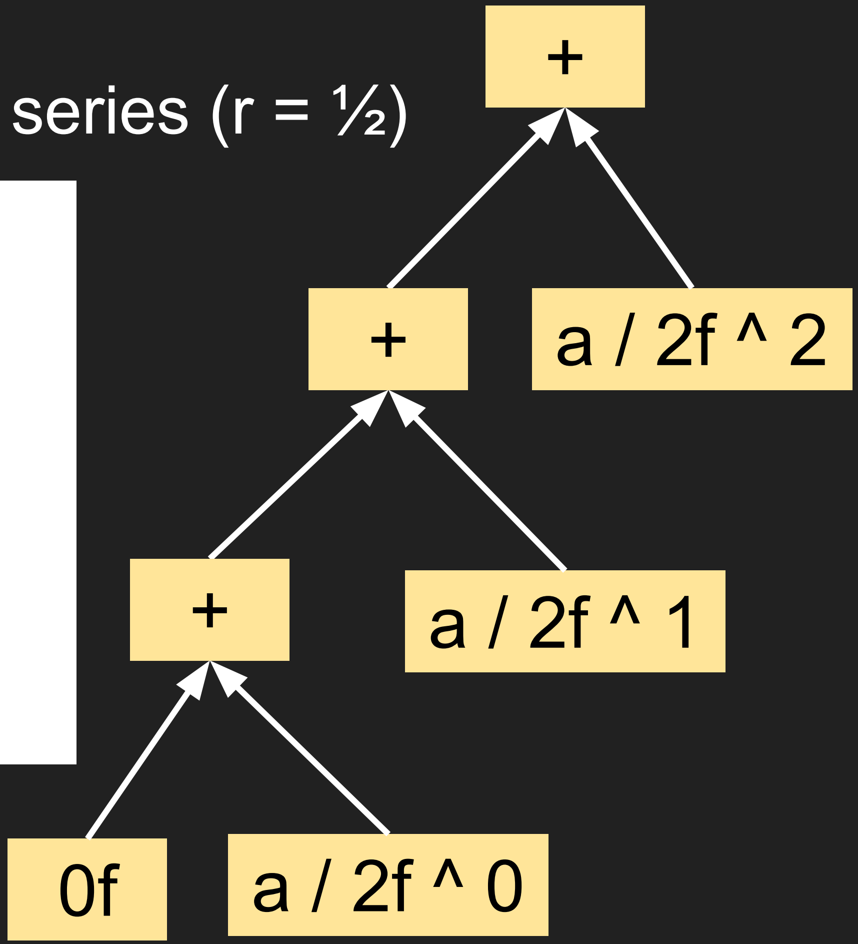
Solution

- Inline differentiable computations
- Unbox scalars

AD Optimize Plugin: geometric series (r = ½)

```
fun foo(a:DScalar): DScalar{
   var y = DScalar(0f)
   for (i in 0 until 1000) {
     y += a / 2f.pow(i)
   }
   return y
}
val x:DScalar = DScalar(5f)
val derivative =
     reverseDerivative(x, ::foo)
```

1) Unbox Scalars
2) Inline derivative computation



AD Optimize Plugin: geometric series (r = ½)

```
@ADOptimize
fun foo(a:DScalar): DScalar{
   var y = DScalar(0f)
   for (i in 0 until 1000) {
     y += a / 2f.pow(i)
   }
   return y
}
val x:DScalar = DScalar(5f)
val derivative =
     reverseDerivative(x, ::foo)
```

a / 2f ^ 0 + a / 2f ^ 1 + ...

1) Unbox Scalars
2) Inline derivative computation

AD Optimize Plugin: geometric series (r = ½)

```
@ADOptimize
fun foo(a:DScalar): DScalar{
   var y = DScalar(0f)
   for (i in 0 until 1000) {
     y += a / 2f.pow(i)
   }
   return y
}
val x:DScalar = DScalar(5f)
val derivative =
     reverseDerivative(x, ::foo)
```

a / 2f ^ 0 + a / 2f ^ 1 + ...

1) Unbox Scalars
2) Inline derivative computation

But we can do more!

Coarsening Optimization: Concept

AD with Coarsening

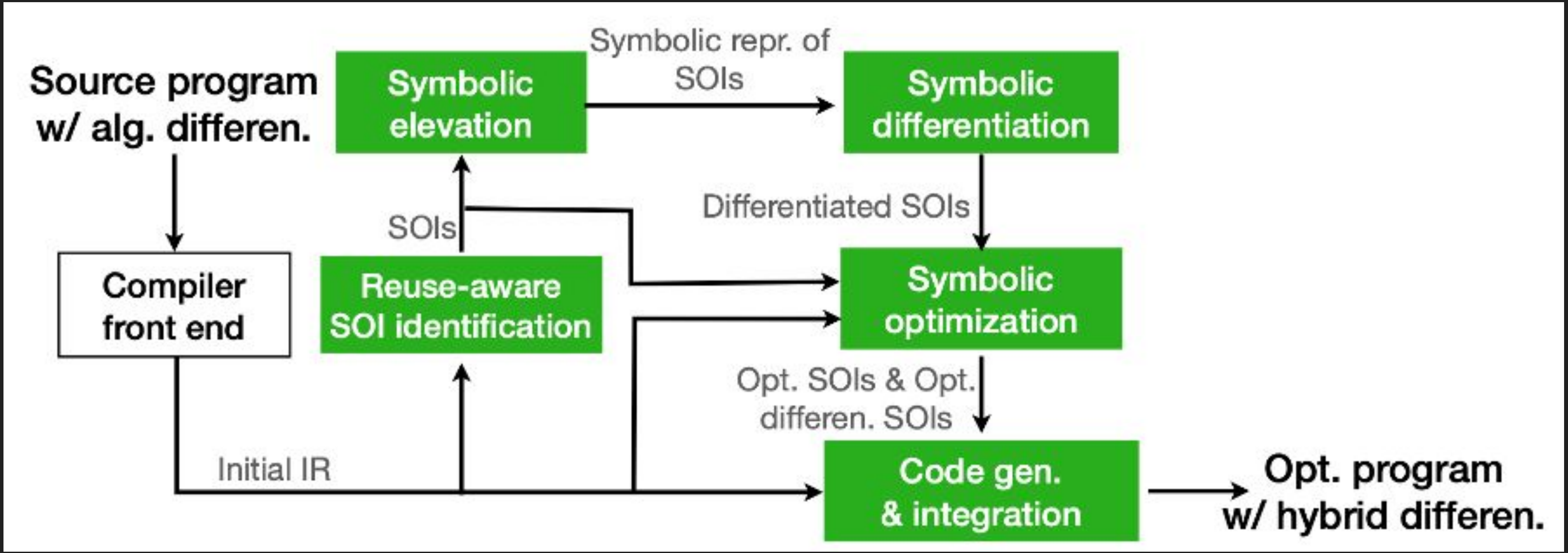
Algorithmic Differentiation ⟷ **Symbolic Differentiation**

Best of both worlds

Finest granularity: Each operation

Largest granularity: Entire calculation

Coarsening Optimization: Workflow



Coarsening: geometric series (r = ½)

```
fun foo(a:DScalar): DScalar{
   var y = DScalar(0f)
   for (i in 0 until 1000) {
     y += a / 2f.pow(i)
   }
   return y
}
val x:DScalar = DScalar(5f)
val derivative =
     reverseDerivative(x, ::foo)
```

a * (1 - 0.5 ^ 1001) / (1 - 0.5)

For $r \neq 1$, the sum of the first n+1 terms of a geometric series, up to and including the r^n term, is

$$a + ar + ar^2 + ar^3 + \cdots + ar^n = \sum_{k=0}^{n} ar^k = a\left(\frac{1-r^{n+1}}{1-r}\right),$$

Coarsening: geometric series (r = ½)

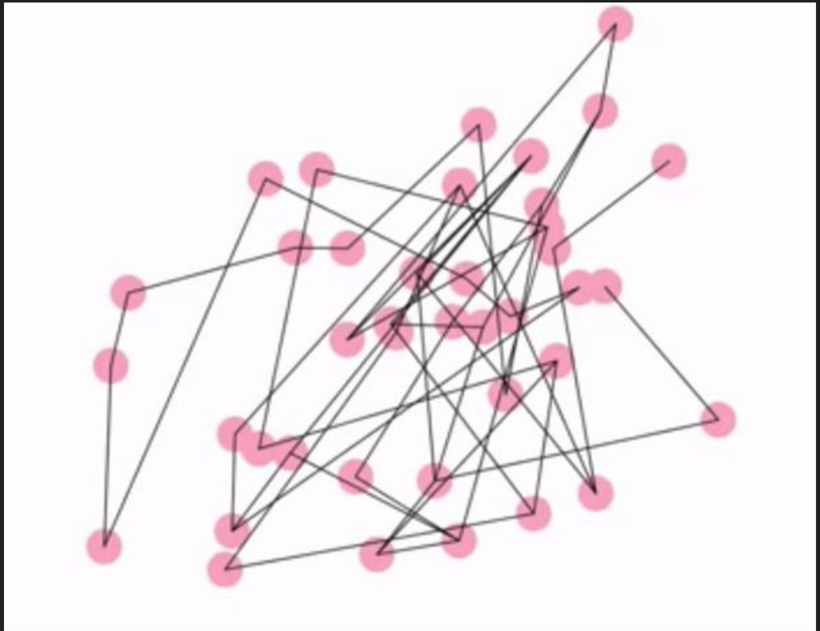
```
fun foo(a:DScalar): DScalar{
   return a * DScalar((1f - 0.5f.pow(1001) / (1 - 0.5f))
}

fun fooGrad(a:DScalar): DScalar{
   return DScalar((1f - 0.5f.pow(1001) / (1 - 0.5f))
}
val x:DScalar = DScalar(5f)
val derivative = fooGrad(x)
```

For $r \neq 1$, the sum of the first n+1 terms of a geometric series, up to and including the r^n term, is

$$a + ar + ar^2 + ar^3 + \cdots + ar^n = \sum_{k=0}^{n} ar^k = a\left(\frac{1 - r^{n+1}}{1 - r}\right),$$

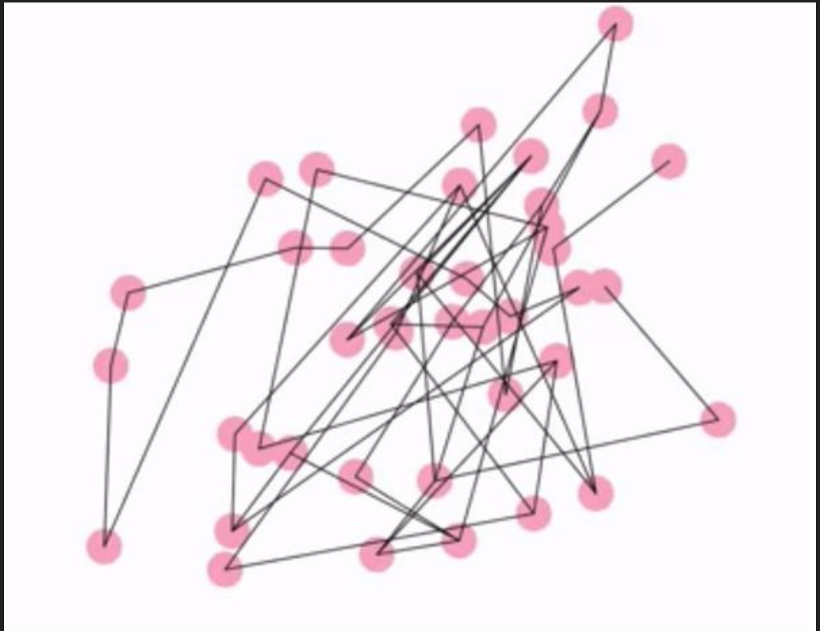
Time Reduction on Hookean Spring



	Primal (ms)	Gradients (ms)	Total (ms)
10 vertices	67=>0	93=>14	160=>15 **(11X)**
20 vertices	74=>0	106=>26	180=>27 **(6.7X)**
40 vertices	159=>0	221=>49	380=>51 **(7.5X)**

Time Reduction on Hookean Spring

Speedups of
1-2 orders of magnitude



	Primal (ms)	Gradients (ms)	Total (ms)
10 vertices	67=>0	93=>14	160=>15 **(11X)**
20 vertices	74=>0	106=>26	180=>27 **(6.7X)**
40 vertices	159=>0	221=>49	380=>51 **(7.5X)**

Static Shape Checking

- Compile-time tensor shape inference
- Compile-time tensor shape checking
- Real time feedback in IntelliJ
- Integration with our API
- User-defined shape functions

Static Shape Checking: example

```
@DeclareParams("A: _", "B: _", "C: _")
fun matmul(
    x: @ShapeOf("[A, B]") Tensor,
    y: @ShapeOf("[B, C]") Tensor
) : @ShapeOf("[A, C]") Tensor {
    …
}
```

Static Shape Checking: example

```
@DeclareParams("A: _", "B: _", "C: _")
fun matmul(
    x: @ShapeOf("[A, B]") Tensor,
    y: @ShapeOf("[B, C]") Tensor
) : @ShapeOf("[A, C]") Tensor {
    …
}

val a = Tensor(Shape(1, 2), ...)  // [1,2]
val b = Tensor(Shape(2, 3), ...)  // [2,3]
val res = matmul(a,b)             // [1,3]
```

Static Shape Checking: example

```
@DeclareParams("A: _", "B: _", "C: _")
fun matmul(
    x: @ShapeOf("[A, B]") Tensor,
    y: @ShapeOf("[B, C]") Tensor
) : @ShapeOf("[A, C]") Tensor {
    …
}

val a = Tensor(Shape(1, 2), ...)  // [1,2]
val b = Tensor(Shape(2, 3), ...)  // [2,3]
val res = matmul(a,b)             // [1,3]

val badRes = matmul(b,a)      // ERROR: 2 != 3
```

Static Shape Checking: IntelliJ

```
val a = Tensor(Shape( …dims: 1,2), floatArrayOf(1f, 1f, 1f))
val b = Tensor(Shape( …dims: 2,3), floatArrayOf(1f, 1f, 1f, 1f , 1f, 1f))

matmul(a,b)
```

Tensor Shape([1, 3])

```
val a = Tensor(Shape( …dims: 1,2), floatArrayOf(1f, 1f, 1f))
val b = Tensor(Shape( …dims: 2,3), floatArrayOf(1f, 1f, 1f, 1f , 1f, 1f))

matmul(b,a)
```

[SHAPE_FUNCTION_ERROR] Shape Dimension Mismatch: 2 != 3

More Complex Shape Checking

```
@ShapeFunction
fun broadcast(a: Shape, b: Shape) : Shape? { … }

------------------------------------------------------------------------

@DeclareParams("A: [___]", "B: [___]")
fun add(a: @ShapeOf("A") Tensor, b: @ShapeOf("B") Tensor) :
    @ShapeOf("broadcast(A, B)") Tensor { … }
```

Use Case: Probabilistic Programming

- Collaboration with Facebook's PPL, Bean Machine
- Probabilistic Programming can benefit from
  - Higher order differentiation
  - Performant scalar support
  - Fast execution of native language
  - Sparse tensors
  - AD Optimize
  - Coarsening

Summary

- Performance
  - sparse tensors, MKL-DNN, Kotlin, optimization plugins
- Usability
  - functional API, static shape checking
- Flexibility
  - extensible API and plugins, probabilistic programming

Future Work

- Performance
  - Develop new optimizations enabled by compiler plugins
- Usability
  - Continue our work on static analyses
- Flexibility
  - Collaborate with users

Thank you!