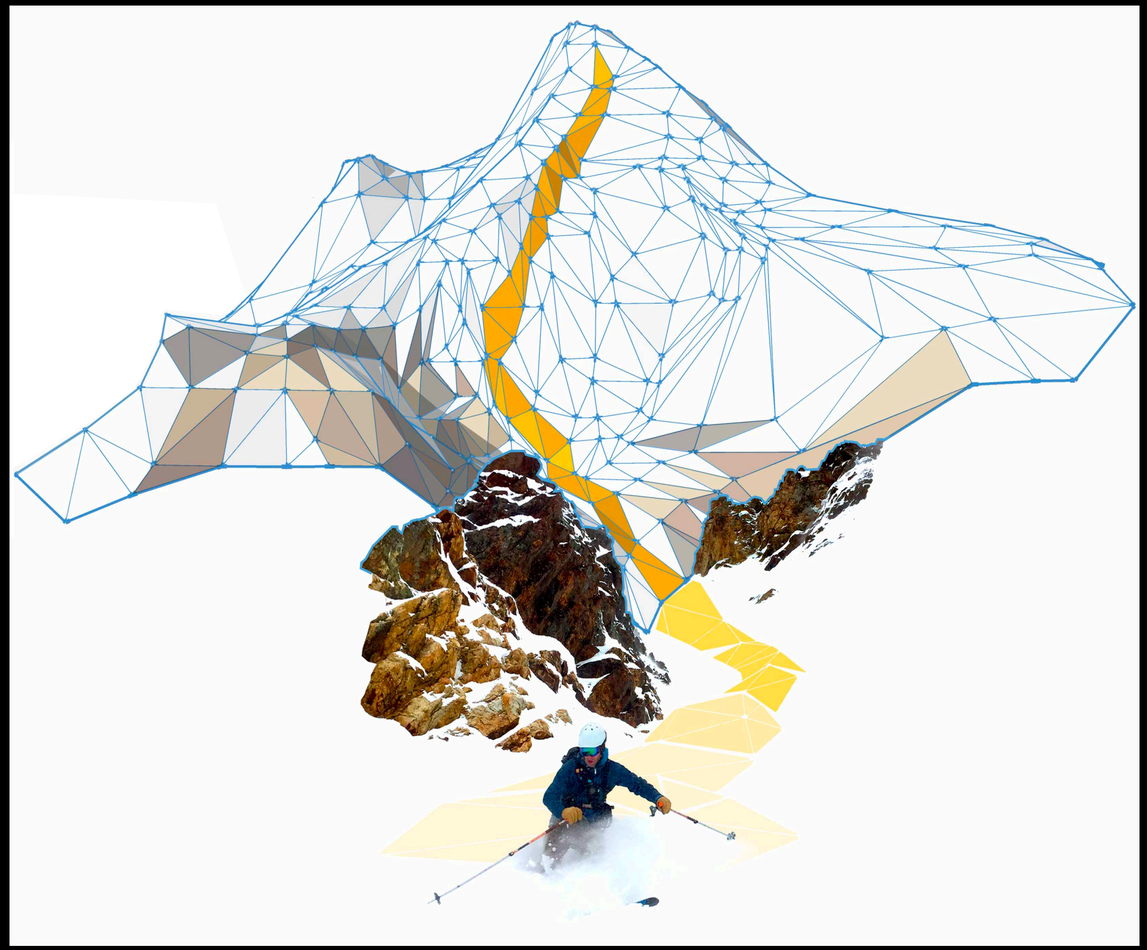
BACK TO SCHOOL: TEACHING COMPILER TO DIFFERENTIATE... PROGRAMS!

ANTON KOROBEYNIKOV
ACCESS SOFTEK TOOLCHAINS / COMPILER TOOLCHAINS OY / LLVM FOUNDATION

GRADIENT DESCENT: THE MAGIC BEHIND DEEP LEARNING

- How do you optimize systems with tons of parameters?
- Start from a loss function
  - How far off-target are you?
- Find how to make that go “downhill”
- From that, work backwards to find contributions from each parameter
- Incredibly fast process, but requires **differentiation**



Paradigm shift from *deep learning* to *differential computing*

DIFFERENTIAL PROGRAMMING BASICS

- Parametrized function approximator:

$$f : A \to B \qquad \hat{f}_w : A \to B \qquad w \in P$$

- Some loss function, e.g.:

$$\mathcal{L}(w) = \left\| f(a) - \hat{f}_w(a) \right\|^2$$

- Gradient descent:
  - Given a training sample $(a, f(a))$ and some initialization of w at w^i
  - Update w proportional to learning rate in the direction of $\nabla\mathcal{L}$:

$$w^{i+1} = w^i - r\nabla\mathcal{L}(w^i)$$

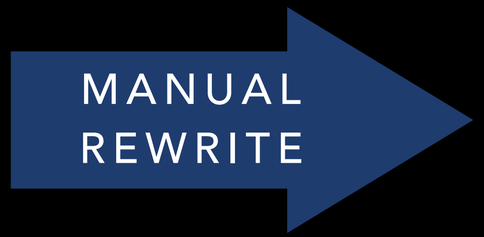
What if f and $\mathcal{L}$ are defined by a program?

EXISTING DIFFPROG APPROACHES (1/3)

Differentiable DSL (TensorFlow, PyTorch, DiffTaichi):

- Provide a new language designed to be differentiated
- Requires rewriting everything in the DSL and the DSL must support all operations in original code
- Fast if DSL matches original code well

```
double relu3(double val) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```




```
import tensorflow as tf

x = tf.Variable(3.14)

with tf.GradientTape() as tape:
  out = tf.cond(x > 0,
            lambda: tf.math.pow(x,3),
            lambda: 0
        )
print(tape.gradient(out, x).numpy())
```

EXISTING DIFFPROG APPROACHES (2/3)

Operator overloading (Adept, JAX, Matlogica AADC):

- Differentiable versions of existing language constructs (double => adouble / idouble, np.sum => jax.sum)
- May require rewriting to use non-standard language utilities
- Often dynamic: storing instructions/values to later be interpreted

```
// Rewrite to accept either
//    double or adouble
template<typename T>
T relu3(T val) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```

```
adept::Stack stack;
adept::adouble inp = 3.14;

// Store all instructions into stack
adept::adouble out(relu3(inp));
out.set_gradient(1.00);

// Interpret all stack instructions
double res = inp.get_gradient(3.14);
```

EXISTING DIFFPROG APPROACHES (3/3)

Source rewriting (Tapenade, ADIC, Zygote):

- Statically analyze program to produce a new gradient function in the source language
- Re-implement parsing and semantics of given language
- Requires all code to be available ahead of time => hard to use with external libraries

```
// myfile.c
double relu3(double x) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```


```
// grad_myfile.c
double grad_relu3(double x) {
  if (x > 0)
    return 3 * pow(x,2)
  else
    return 0;
}
```

ONE LANGUAGE, ALL THE WAY DOWN

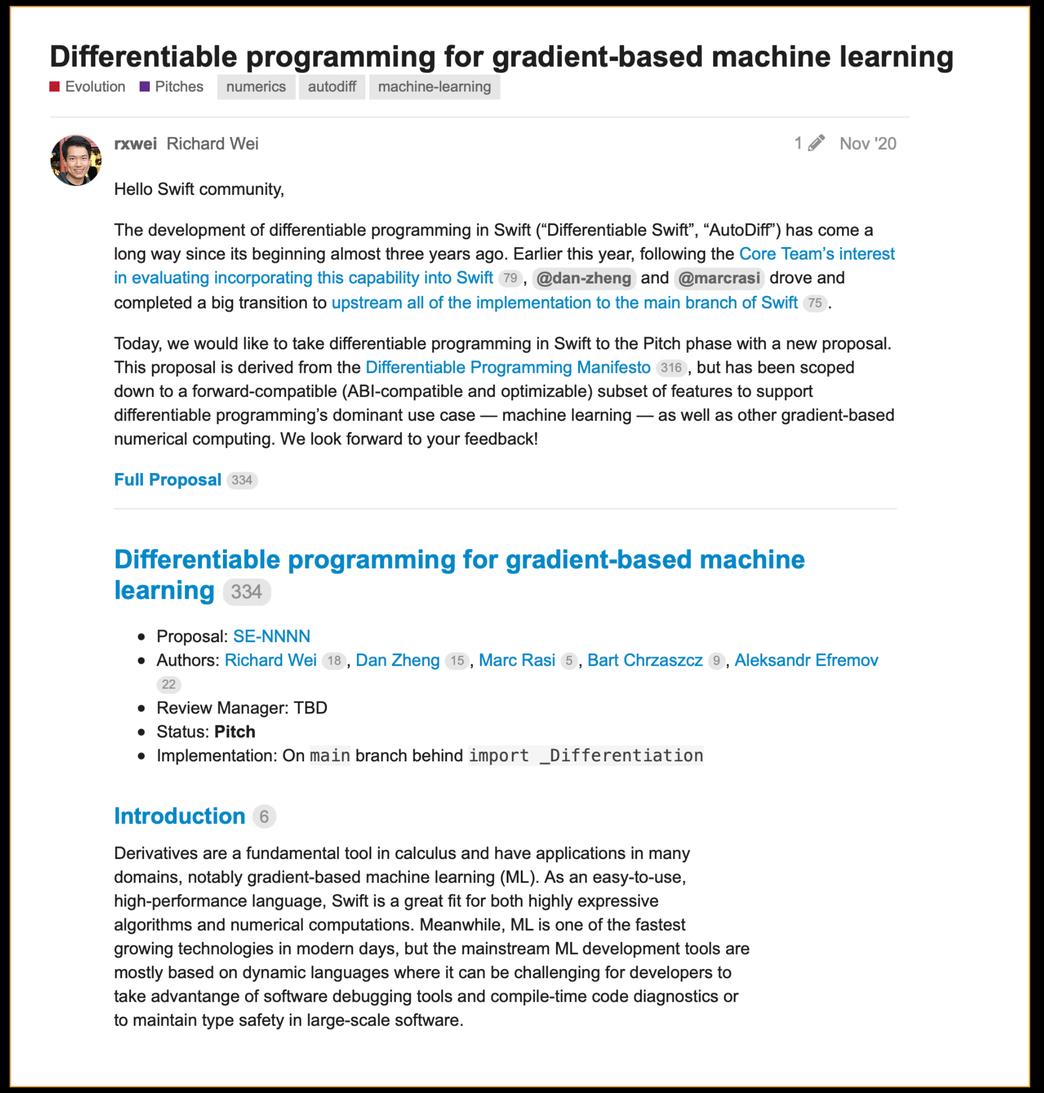
- Python / C++ dichotomy
  - Easy-to-use high-level language
  - Fast, compiled language under the hood

What if one language could provide both?

DIFFERENTIABLE SWIFT

DIFFERENTIABLE SWIFT

- Experimental feature
  - Available today in swift.org toolchains
  - `import _Differentiation`
- Can be used with stock Xcode toolchains, with some setup
- Supported on all Swift platforms (MacOS, Linux, Windows)



Differentiable programming for gradient-based machine learning

Evolution · Pitches · numerics · autodiff · machine-learning

rxwei Richard Wei — 1 edit — Nov '20

Hello Swift community,

The development of differentiable programming in Swift ("Differentiable Swift", "AutoDiff") has come a long way since its beginning almost three years ago. Earlier this year, following the Core Team's interest in evaluating incorporating this capability into Swift 79, **@dan-zheng** and **@marcrasi** drove and completed a big transition to upstream all of the implementation to the main branch of Swift 75.

Today, we would like to take differentiable programming in Swift to the Pitch phase with a new proposal. This proposal is derived from the Differentiable Programming Manifesto 316, but has been scoped down to a forward-compatible (ABI-compatible and optimizable) subset of features to support differentiable programming's dominant use case — machine learning — as well as other gradient-based numerical computing. We look forward to your feedback!

Full Proposal 334

Differentiable programming for gradient-based machine learning 334

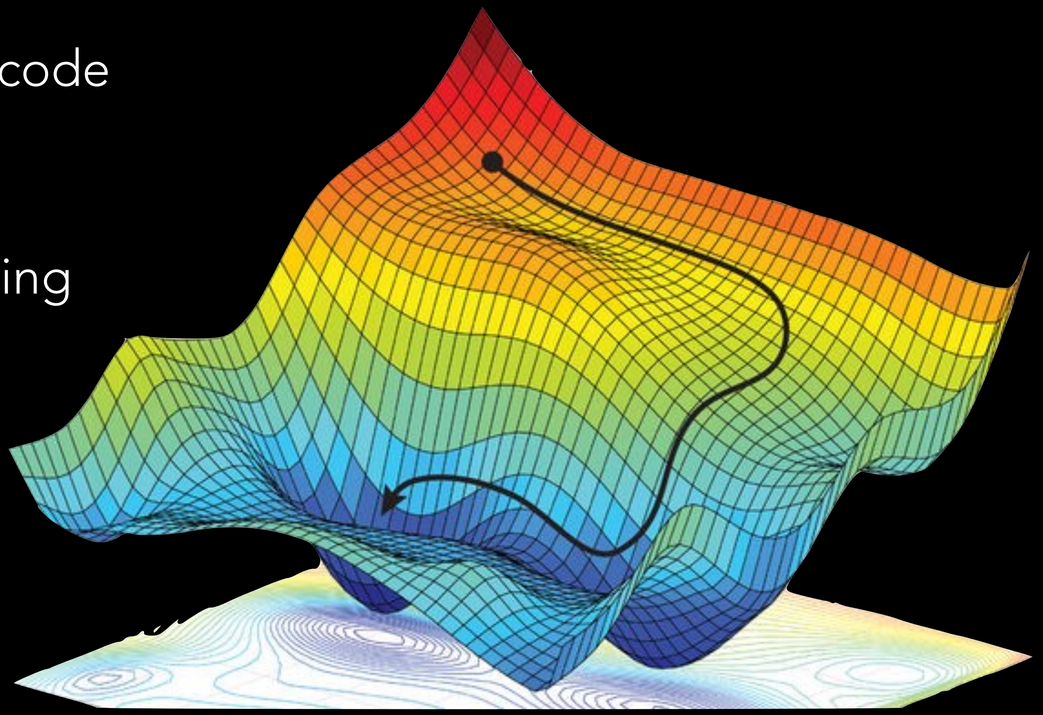
- Proposal: SE-NNNN
- Authors: Richard Wei 18, Dan Zheng 15, Marc Rasi 5, Bart Chrzaszcz 9, Aleksandr Efremov 22
- Review Manager: TBD
- Status: **Pitch**
- Implementation: On `main` branch behind `import _Differentiation`

Introduction 6

Derivatives are a fundamental tool in calculus and have applications in many domains, notably gradient-based machine learning (ML). As an easy-to-use, high-performance language, Swift is a great fit for both highly expressive algorithms and numerical computations. Meanwhile, ML is one of the fastest growing technologies in modern days, but the mainstream ML development tools are mostly based on dynamic languages where it can be challenging for developers to take advantange of software debugging tools and compile-time code diagnostics or to maintain type safety in large-scale software.

DIFFERENTIABLE SWIFT

- First-class automatic differentiation
  - Automatically generates derivatives for arbitrary Swift code
  - Builds derivatives at the SIL level
  - Broad support for language features and types, including control flow
- Typechecking for differentiability
  - Minimizes cryptic error messages
  - Boosts developer productivity



Unique among ahead-of-time compiled systems languages

FIRST-CLASS?

Differentiation of computable functions is not computable[1,2],
so it can't be implemented as a library.

[1] Marian Boykan Pour-El and Ian Richards. Differentiability properties of computable functions — a summary. 1978.
[2] Marian Boykan Pour-El and Ian Richards. Computability and noncomputability in classical analysis. 1983.

IMPLEMENTATION DETAIL

COMPILER TRANSFORMATION?

FIRST-CLASS LANGUAGE SUPPORT

- Differentiable data structures
- Differentiable functions in the type system

DIFFERENTIABLE DATA STRUCTURES

DIFFERENTIABLE DATA STRUCTURES

```
struct Float // ✅
struct Double // ✅
struct Tensor<Scalar> // 🤷 depends on Scalar
struct SIMD4<Scalar> // 🤷 depends on Scalar
struct Int // ❌
```

DIFFERENTIABLE DATA STRUCTURES

```
struct Float // ✅
struct Double // ✅
struct Tensor<Scalar> // 🤷 depends on Scalar
struct SIMD4<Scalar> // 🤷 depends on Scalar
```

DIFFERENTIABLE DATA STRUCTURES

```
struct Float: VectorArithmetic
struct Double: VectorArithmetic
struct Tensor<Scalar: Numeric>: VectorArithmetic
struct SIMD4<Scalar: Numeric>: VectorArithmetic
```

DIFFERENTIABLE DATA STRUCTURES

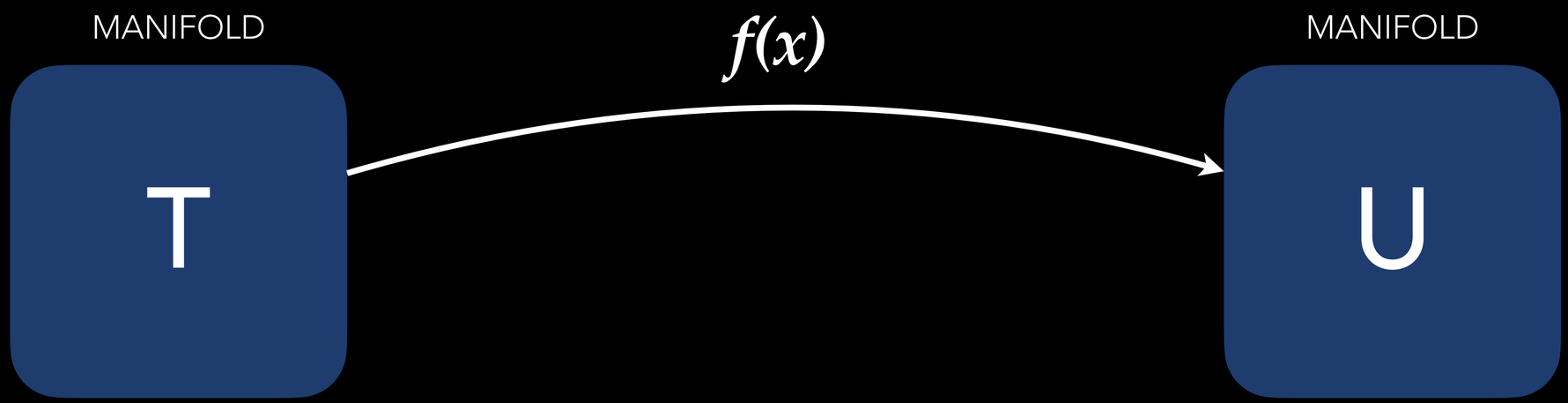
```
struct CustomLayer {
    var weight, bias: Tensor<Float>
    var useBias: Bool
}
```

DIFFERENTIABLE DATA STRUCTURES

```
struct CustomLayer: VectorArithmetic {
    var weight, bias: Tensor<Float>
    var useBias: Bool ✖
}
```

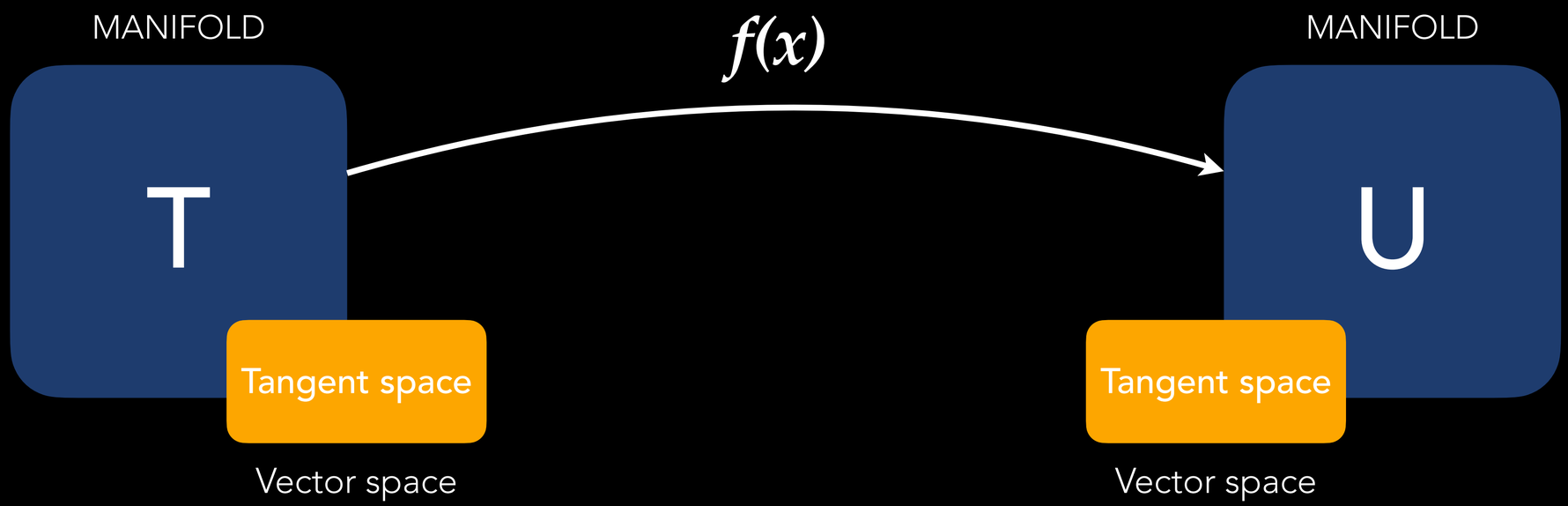
DIFFERENTIAL GEOMETRY

DIFFERENTIAL GEOMETRY

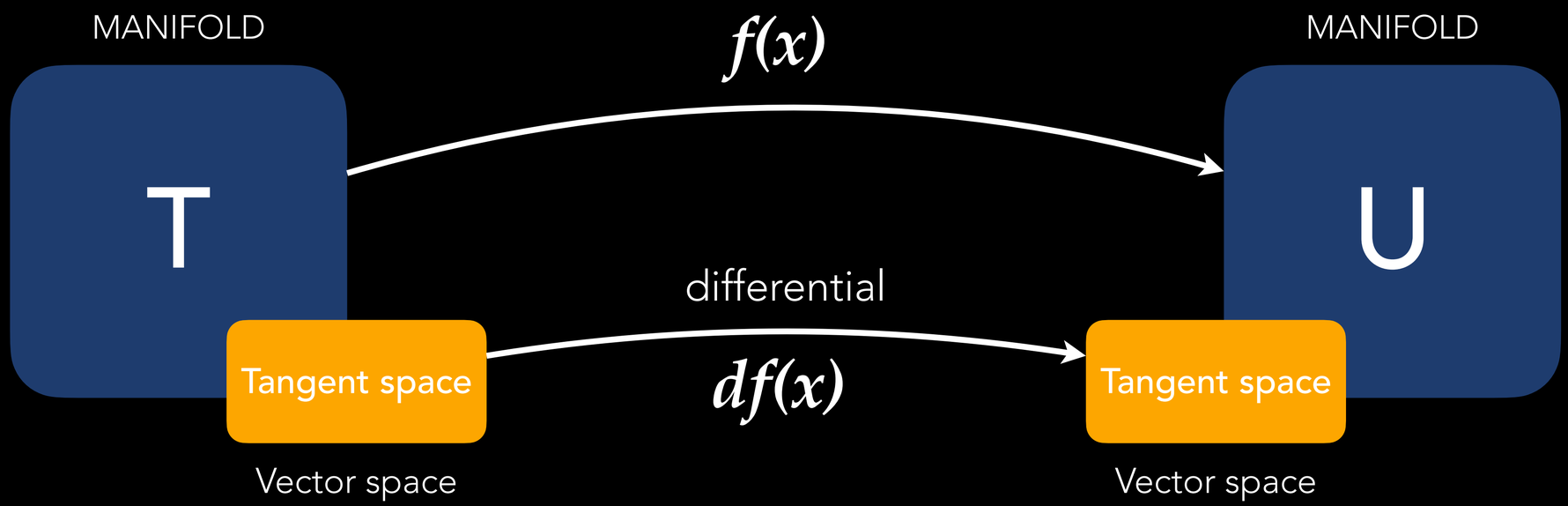



```
f: (T) -> U
```

DIFFERENTIAL GEOMETRY

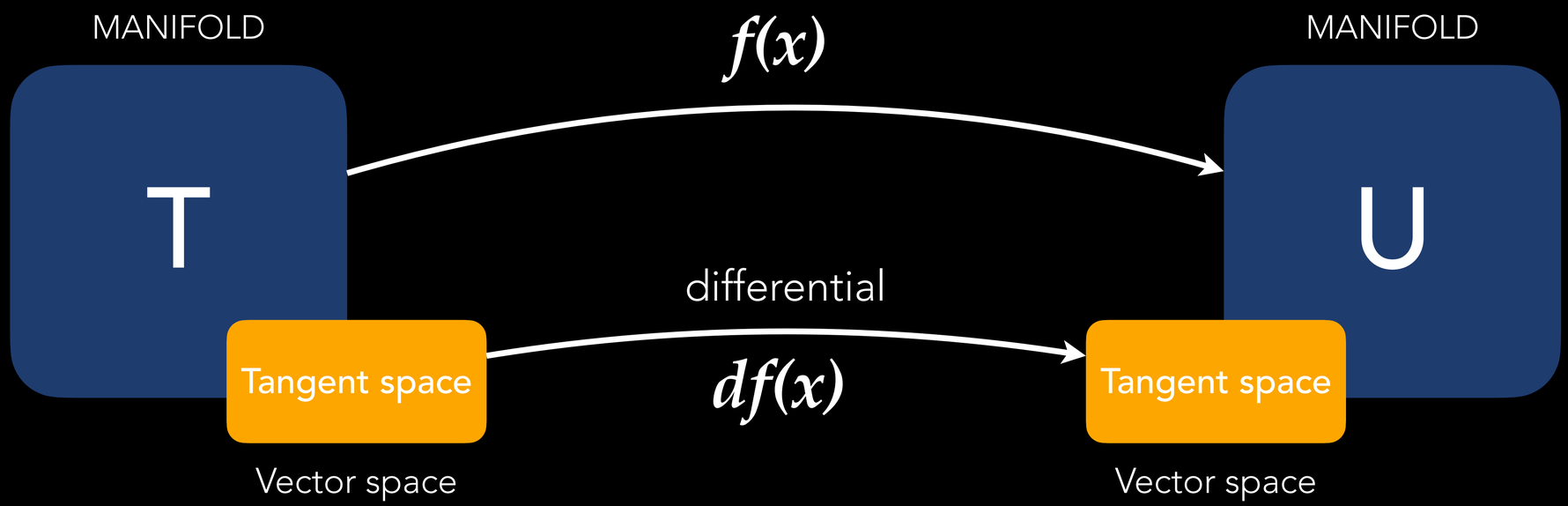


DIFFERENTIAL GEOMETRY




```
df: (T.TangentVector) -> U.TangentVector
```

DIFFERENTIAL GEOMETRY




```
exponentialMap: (T, T.TangentVector) -> T
```

```
protocol Differentiable
```

PROTOCOL DIFFERENTIABLE

```
protocol Differentiable {
  /// A type representing the differentiable value's derivatives.
  /// Mathematically, this is equivalent to the tangent bundle of
  /// the differentiable manifold represented by the differentiable
  /// type.
  associatedtype TangentVector: Differentiable & AdditiveArithmetic
  /// Moves 'self' along the given direction. In Riemannian
  /// geometry, this is equivalent to exponential map, which moves
  /// 'self' on the geodesic surface along the given tangent
  /// vector.
  mutating func move(along direction: TangentVector)
}
```

```
protocol Differentiable {
  /// A type representing the differentiable value's derivatives.
  /// Mathematically, this is equivalent to the tangent bundle of
  /// the differentiable manifold represented by the differentiable
  /// type.
  associatedtype TangentVector: Differentiable & AdditiveArithmetic
  /// Moves 'self' along the given direction. In Riemannian
  /// geometry, this is equivalent to exponential map, which moves
  /// 'self' on the geodesic surface along the given tangent
  /// vector.
  mutating func move(along direction: TangentVector)
}

extension Differentiable where Self == TangentVector {
  mutating func move(along direction: TangentVector) {
    self += direction
  }
}
```

STANDARD LIBRARY TYPES & EXTENSIONS

```
extension Float: Differentiable {
  typealias TangentVector = Self
}

extension Double: Differentiable {
  typealias TangentVector = Self
}

extension SIMD4: Differentiable where Scalar: Differentiable {
  typealias TangentVector = Self
}
```

OTHER LIBRARY TYPES & EXTENSIONS

```
// struct Tensor<Scalar>
extension Tensor: Differentiable where Scalar: Differentiable {
  typealias TangentVector = Self
}
```

OTHER LIBRARY TYPES & EXTENSIONS

```
// struct Array<Element>
extension Array: Differentiable where Element: Differentiable {
  struct TangentVector { var elements: [Element.TangentVector] }
  …
}

// struct Dictionary<Key: Hashable, Value>
extension Dictionary: Differentiable where Value: Differentiable {
  typealias TangentVector = Dictionary<Key, Value.TangentVector>
...
}

// enum Optional<Wrapped>
extension Optional: Differentiable where Wrapped: Differentiable {
  typealias TangentVector = Optional<Element.TangentVector>
...
}
```

CUSTOM TYPES & EXTENSIONS

```
struct CustomLayer: Differentiable {
  var weight, bias: Tensor<Float>
  var useBias: Bool
}
```

CUSTOM TYPES & EXTENSIONS

```
struct CustomLayer: Differentiable {
  var weight, bias: Tensor<Float>
  var useBias: Bool

  // The compiler synthesizes this for you!
  struct TangentVector: Differentiable & AdditiveArithmetic {
    var weight, bias: Tensor<Float>
  }

  // And this!
  mutating func move(along direction: TangentVector) {
    weight.move(along: direction.weight)
    bias.move(along: direction.bias)
  }
}
```

CUSTOM TYPES & EXTENSIONS

```
struct Point<T: Real>: Differentiable & AdditiveArithmetic {
  var x: T
  var y: T
}
```

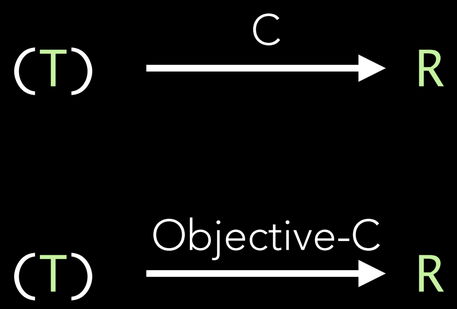
DIFFERENTIABLE FUNCTIONS

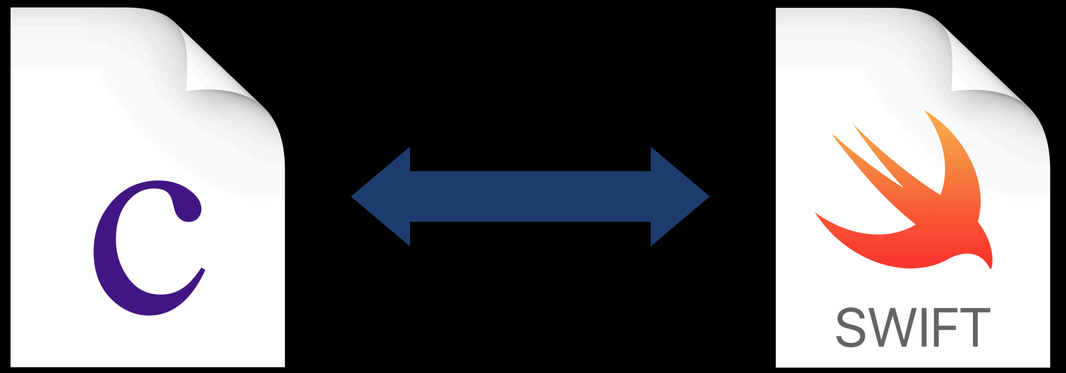
DIFFERENTIABLE FUNCTIONS

```
(T) -> R
```

```
extension Array {
  func map<T>(_ f: (Element) -> T) -> [T]
}

[1, 2].map { x in x + 1 }
```

CALLING CONVENTIONS





Analogy: Calling conventions in language interoperability

CALLING CONVENTIONS


```c
int addOne(int x) { return x + 1; }
int (*addOneFunctionPointer)(int) = addOne; // (Int) -> Int
```


```swift
// Imported as:
let addOneFunctionPointer: @convention(c) (Int) -> Int
addOneFunctionPointer as (Int) -> Int

[1, 2, 3].map(addOneFunctionPointer) // [2, 3, 4]
```

CALLING CONVENTIONS

```
@convention(c) (Int) -> Int
```

FUNCTION POINTER

subtype

```
(Int) -> Int
```

FUNCTION POINTER

CONTEXT POINTER

DIFFERENTIABLE FUNCTION

```
(T) -> U
```

subtype

```
@differentiable (T) -> U
```

UPCASTING & DOWNCASTING

Upcasting to non-`@differentiable`:

```
let f0: @differentiable (Float) -> (Float) = ...
let f2: (Float) -> (Float) = f0
```

Conversion to `@differentiable` requires *differentiation*:

```
func addOne(_ x: Float) -> Float {
  x + 1
}

let _: @differentiable (Float) -> (Float) = addOne // Okay!
```

DIFFERENTIABILITY REQUIREMENTS

1. Explicit `@differentiable` attribute:

```
@differentiable(reverse)
func sinOfSin(_ x: Double) -> Double
```

2. Composition of `@differentiable`:

```
func foo(_ x: Float, _ y: Float) -> Float {
  cos(x) + y
}
```

DIFFERENTIATION: MOTIVATING EXAMPLE

$$(x, t)$$

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

$$\frac{\partial \mathcal{L}_{\text{reg}}}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{2}\left(\sigma(wx + b) - t\right)^2 + \frac{\lambda}{2}w^2\right]$$

APPROACHES TO AUTOMATIC DIFFERENTIATION

- Program representation:

$$y = F(x) = F_q \circ F_{q-1} \circ F_1$$

- Forward mode:

$$\frac{\partial F}{\partial x} = \frac{\partial F_q}{\partial F_1}\frac{\partial F_{q-1}}{\partial x} = \frac{\partial F_q}{\partial F_{q-1}}\left(\frac{\partial F_{q-1}}{\partial F_{q-2}}\frac{\partial F_{q-2}}{\partial x}\right) = \dots$$

- Reverse mode:

$$\frac{\partial F}{\partial x} = \frac{\partial F_q}{\partial F_1}\frac{\partial F_1}{\partial x} = \left(\frac{\partial F_q}{\partial F_2}\frac{\partial F_2}{\partial F_1}\right)\frac{\partial F_1}{\partial x} = \dots$$

A PRIMER ON REVERSE-MODE AD

$$(x, t)$$

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$\mathcal{R} = \frac{1}{2}w^2$$

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

$$\bar{v} := \frac{\partial \mathcal{L}_{\text{reg}}}{\partial v}$$

$$\bar{\mathcal{R}} = \lambda; \quad \bar{\mathcal{L}} = 1$$

$$\bar{y} = \bar{\mathcal{L}}\frac{d\mathcal{L}}{dy} + \bar{\mathcal{R}}\frac{d\mathcal{R}}{dy} = \bar{\mathcal{L}}(y - t)$$

$$\bar{z} = \bar{y}\frac{dy}{dz} = \bar{y}\sigma'(z)$$

$$\bar{w} = \bar{z}\frac{dz}{dw} + \bar{\mathcal{R}}\frac{d\mathcal{R}}{dw} = \bar{z}x + \bar{\mathcal{R}}w$$

$$\bar{b} = \bar{z}\frac{dz}{db} = \bar{z}$$

DEFINING CUSTOM DERIVATIVES

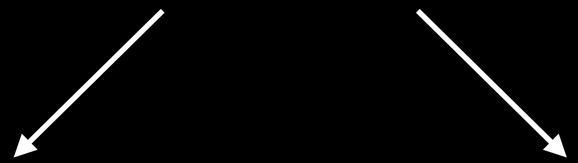
```
import Foundation

public func sqrt(_ __x: Double) -> Double



@derivative(of: sqrt)
func sqrtVJP(_ x: Double) -> (value: Double, pullback: (Double) -> Double) {
    let output = sqrt(x)
    func pullback(_ dv: Double) -> Double {
        return dv / (2 * output)
    }
    return (value: output, pullback: pullback)
}
```

DIFFERENTIATING API / DIFFERENTIAL OPERATORS

```
func valueWithPullback<T, R>(
    at x: T, of f: @differentiable(reverse) (T) -> R
) -> (value: R,
      pullback: @differentiable(reverse) (R.TangentVector) -> T.TangentVector)

func valueWithGradient<T, R>(
    at x: T, of f: @differentiable (T) -> R
) -> (value: R, gradient: T.TangentVector) {
  let (value, pb) = valueWithPullback(at: x, of: f)
  return (value, pb(T.TangentVector(1))
}

func gradient<T, R>(
    at x: T, of f: @differentiable(reverse) (T) -> R
) -> T.TangentVector {
  return pullback(at: x, of: f)(R(1))
}                                    ...
```

DIFFERENTIAL OPERATORS

```
import _Differentiation

@differentiable(reverse)
func square(_ x: Float) -> Float {
    x * x
}

let (value, gradient) = valueWithGradient(at: 3.0, of: square)
```

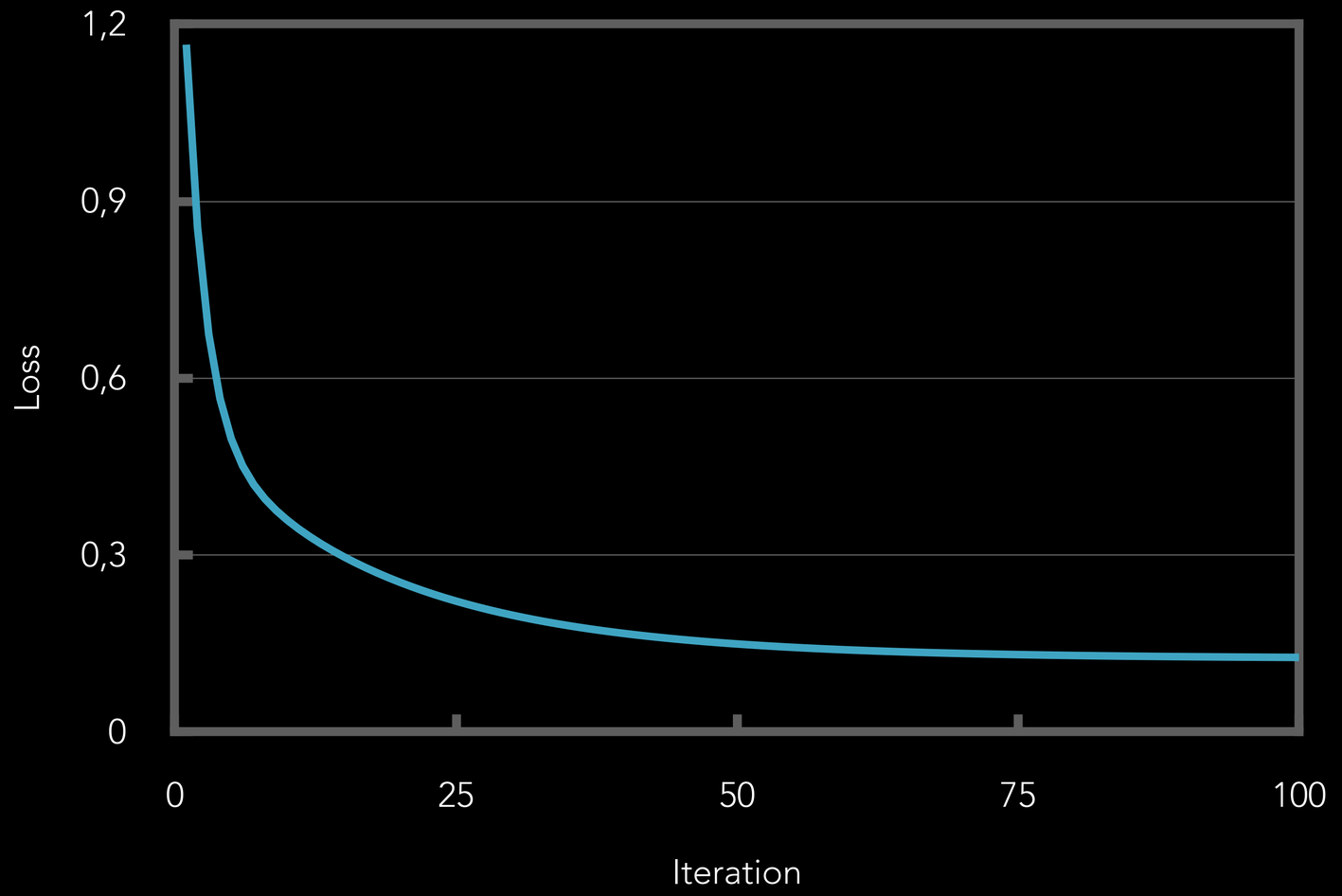


9.0 6.0

GRADIENT DESCENT

```
var model = Perceptron()
for _ in 0..<100 {
    let (loss, pullback) = valueWithPullback(at: model, of: loss)
    print(loss)
    let gradient = pullback(-0.04)
    model.move(by: gradient)
}
```

```
struct Perceptron: Differentiable {
    var weight: SIMD2<Float> = .random(in: -1..<1)
    var bias: Float = 0

    @differentiable(reverse)
    func callAsFunction(_ input: SIMD2<Float>) -> Float {
        (weight * input).temporarySum() + bias
    }
}

let andGateData: [(x: SIMD2<Float>, y: Float)] = [
    (x: [0, 0], y: 0),
    (x: [0, 1], y: 0),
    (x: [1, 0], y: 0),
    (x: [1, 1], y: 1),
]

@differentiable(reverse)
func loss(model: Perceptron) -> Float {
    var loss: Float = 0
    for (x, y) in andGateData {
        let prediction = model(x)
        let error = y - prediction
        loss = loss + error * error / 2
    }
    return loss
}
```

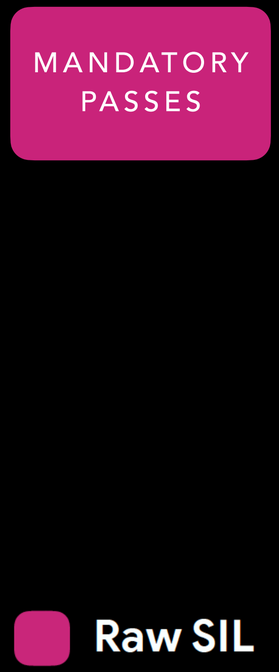


COMPILER

SWIFT COMPILER PIPELINE

Source code AST Raw SIL Canonical SIL LLVM IR

MANDATORY OPTIMIZATIONS



- Closure capture promotion
- Definitive initialization
- Automatic differentiation
- Mandatory inlining
- Memory access exclusivity diagnostics
- Data flow diagnostics
- Infinite recursion diagnostics

SWIFT INTERMEDIATE LANGUAGE (SIL)

```
func foo(_ x: Float) -> Float {
    return sin(x) * cos(x)
}
```

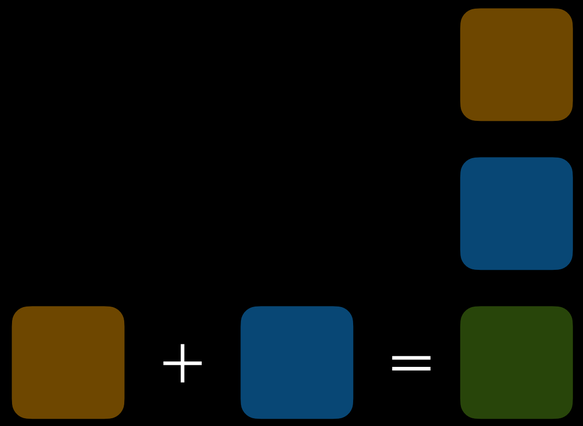
```
// Simplified SIL:
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(%x)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

DIFFERENTIATION STEPS

1. Analysis: What needs to be differentiated?
2. Diagnose: Can everything be differentiated?
3. Transform: Emit derivatives.

ACTIVITY ANALYSIS

VARIED-USEFUL-ACTIVE VALUES



Varied: Depends on the input

Useful: Contributes to output

Active: Needs a derivative!

VARIED VALUES

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(3)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

Varied: Depends on the input

USEFUL VALUES

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(3)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

Useful: Contributes to output

ACTIVE VALUES

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(3)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(3)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(3)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

Active: Needs a derivative!

ACTIVE VALUES

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(3)
  %y2 = sin(%x)
  %y3 = *(%y1,%y2)
  return %y3
}
```

COMPILER TRANSFORM

COMPILER TRANSFORM

1. VJP generation

2. Pullback generation

VJP GENERATION

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(%x)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

```
sil @vjp_foo : $(Float) -> Float {
bb0(%x):
  (%y1, %pb_cos) = vjp_cos(%x)
  (%y2, %pb_sin) = vjp_sin(%x)
  (%y3, %pb_mul) = vjp_mul(%y1, %y2)
  %pb_tuple = (%pb_mul, %pb_sin, %pb_cos)
  %pb_foo = partial_apply @pb_foo(%pb_tuple)
  return %pb_foo
}
```

PULLBACK GENERATION

```
sil @foo : $(Float) -> Float {
bb0(%x):
  %y1 = cos(%x)
  %y2 = sin(%x)
  %y3 = *(%y1, %y2)
  return %y3
}
```

```
sil @pb_foo : $(Float, PBTuple) -> Float {
bb0(%dy3, %pb_tuple):
  (%pb_mul, %pb_sin, %pb_cos) = %pb_tuple
  (%dy1, %dy2) = %pb_mul(%dy3)
  %dx2 = %pb_sin(%dy2)
  %dx1 = %pb_cos(%dy1)
  %dx = +(%dx2, %dx1)
  return %dx
}
```

CONTROL FLOW DIFFERENTIATION

```
func cond(_ x: Float) -> Float {
    if x < 0 {
        return x + x
    }
    return cos(x)
}
```

```
sil @cond: $(Float) -> Float

bb0(%x : $Float):
 %bool = %x < Float.init(0)
 cond_br %bool, bb1, bb2

bb1: // Preds: bb0
 %sum = %x + %x
 br bb3(%sum)

bb2: // Preds: bb0
 %cosx = cos(%x)
 br bb3(%cosx)

bb3(%result : $Float): // Preds: bb1, bb2
 return %result
```

$$f(x) = \begin{cases} g(x), & x > c, \\ h(x), & otherwise \end{cases} \quad \Rightarrow \quad f'(x) = \begin{cases} g'(x), & x > c, \\ h'(x), & otherwise \end{cases}$$

- Conditions: need extra structures to trace control flow
- Loops: need to have dynamically-allocated storage for values produced

STATIC DIAGNOSTICS

CROSS-MODULE OPACITY

```
import Glibc
let y = derivative(at: 1.0) { x in
    sinf(x)
}
test.swift:4:5: error: expression is not differentiable
sinf(x)
^
test.swift:4:5: note: cannot differentiate functions that have not been
marked '@differentiable' and that are defined in other modules
sinf(x)
^
```

NON-DIFFERENTIABLE TYPE CONVERSION

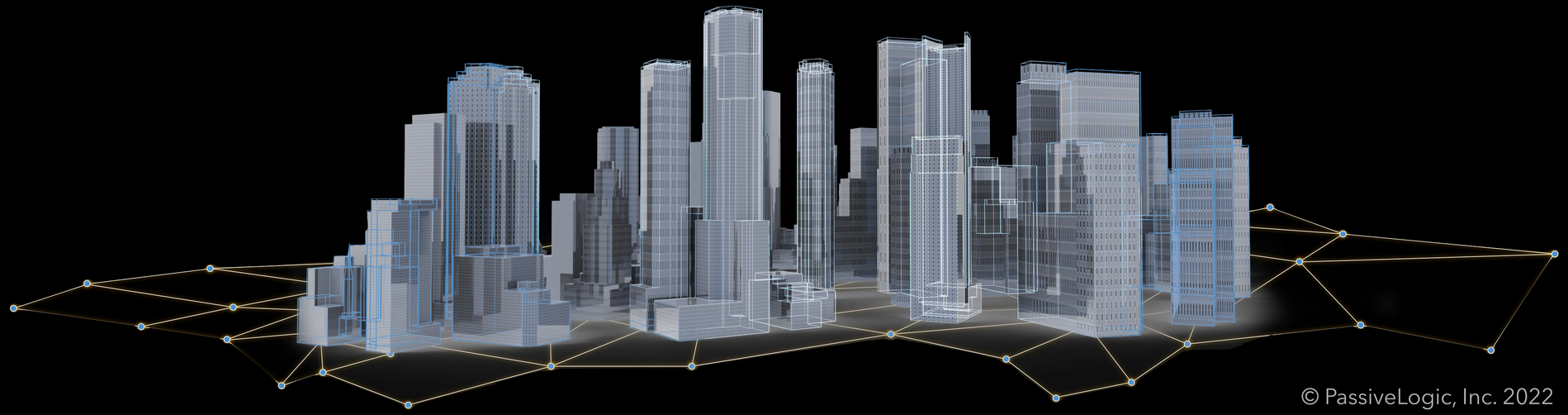
```
let grad = gradient(at: 1.0) { x in
    Double(Int(x)) + 2
}
```

```
test.swift:1:27: error: function is not differentiable
let grad = gradient(at: 1.0) { x in
                             ^~~~~~
test.swift:2:12: note: cannot differentiate through a non-differentiable
result; do you want to add '.withoutDerivative()'?
    Double(Int(x)) + 2
           ^
```

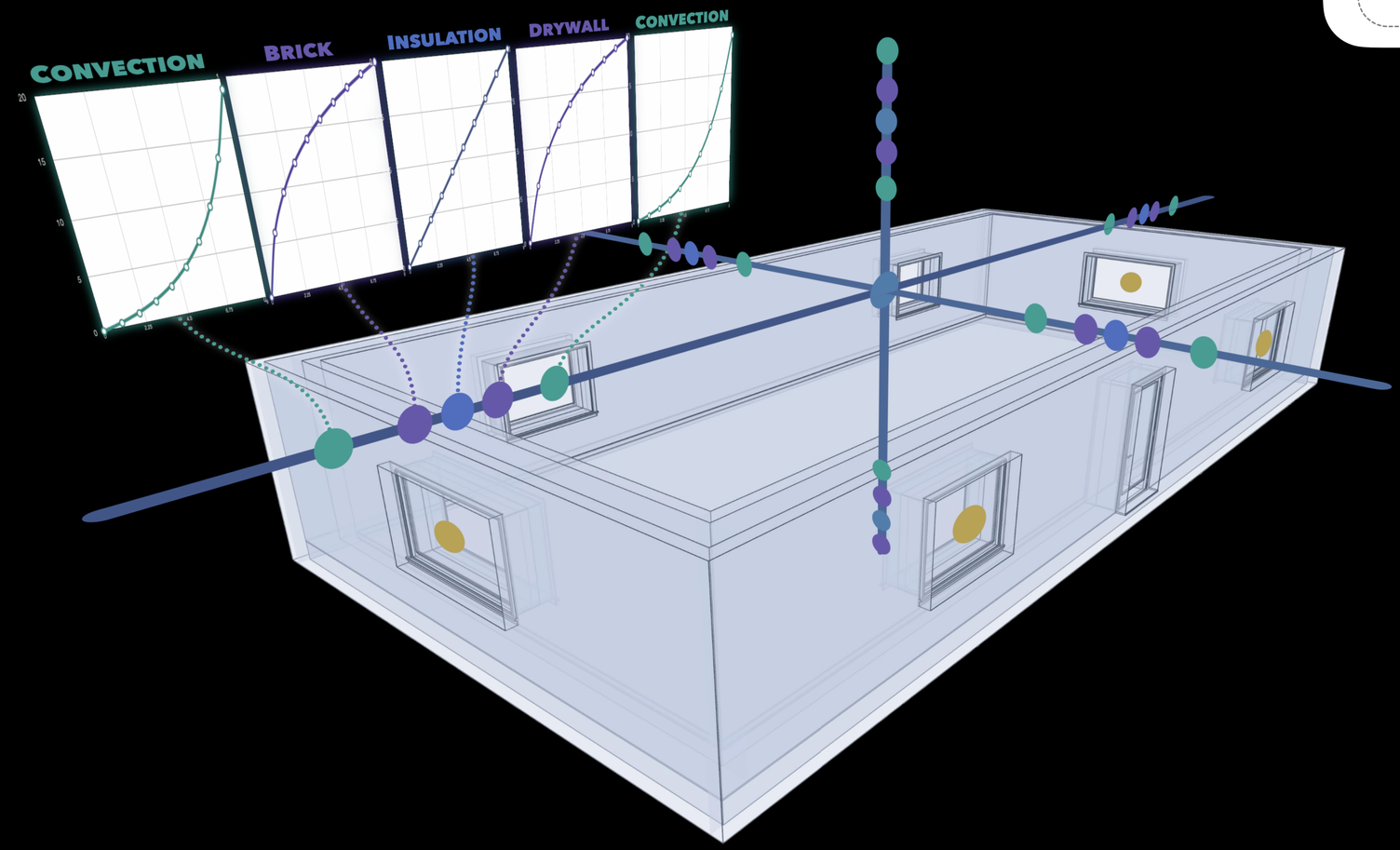
SOME COOL EXAMPLES

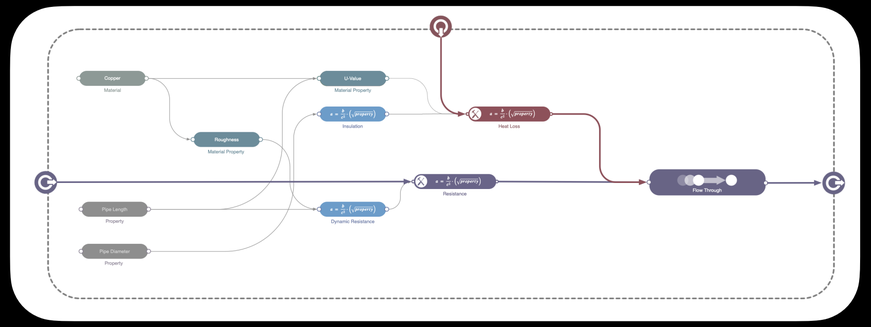
DIGITAL TWINS MODELS FOR BUILDINGS

- Buildings are ~40% of global energy consumption, <3% are automated
- Differentiable digital building twins in Swift make automation possible



DIGITAL TWINS MODELS FOR BUILDINGS



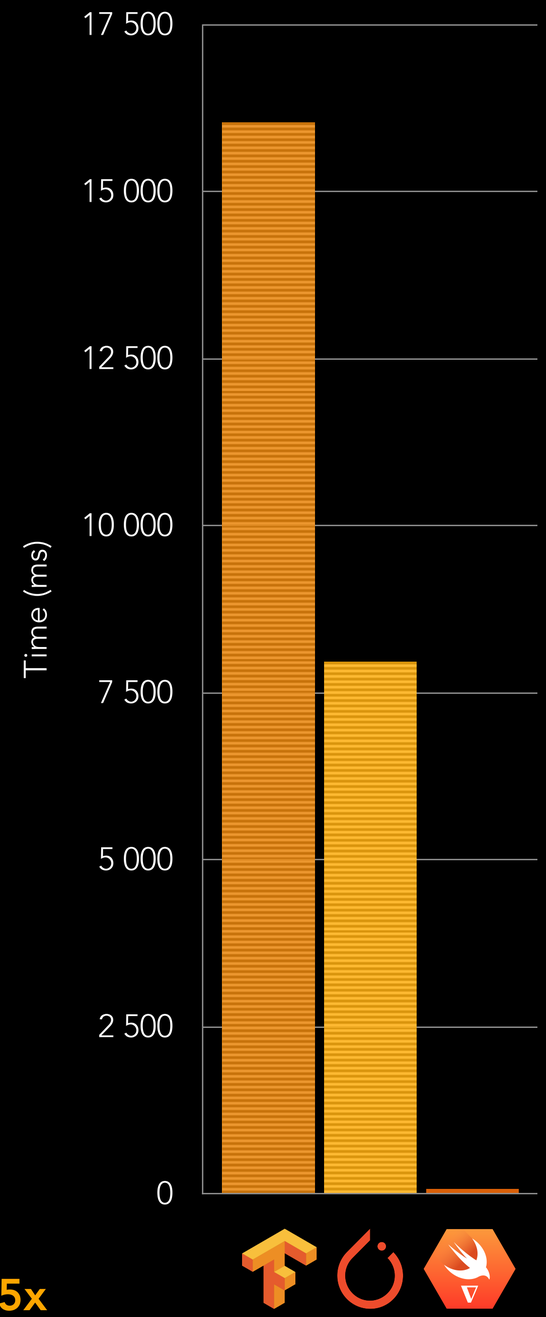


Why not traditional deep learning?

- Every building is unique
- No one model fits all
- Slow training data generation (annual cycle)
  - 1000 datasets => 1000 years!

```swift
@differentiable(reverse)
func computeLoadPower(floor: SlabType, tube: TubeType, quanta: QuantaType) -> QuantaAndPower
{
    let resistance_abs = computeResistance(floor: floor, tube: tube, quanta: quanta)

    let conductance: Float = 1/resistance_abs
    let dTemp = floor.temp - quanta.temp
    let power = dTemp * conductance

    var updatedQuanta = quanta
    updatedQuanta.power = power
    let loadPower = -power

    return QuantaAndPower(quanta: updatedQuanta, power: loadPower)
}
```

```python
@tf.function
def computeLoadPower(floor, tube, quanta):
    resistance_abs = computeResistance(floor, tube, quanta)

    conductance = 1/resistance_abs
    dTemp = floor[SlabTypeIndices.itemp] - quanta[QuantaIndices.itemp]
    power = dTemp * conductance

    loadPower = -power

    resultQuanta = quanta * tf.constant([0.0, 1, 1, 1, 1]) + power * tf.constant([1.0, 0, 0, 0, 0])

    return (resultQuanta, loadPower)
```



- Differentiable Swift is **189X faster** than Python **TensorFlow**
- Differentiable Swift is **117X faster** than **PyTorch**
- ...and that's without latest Swift optimizations we did recently that improved timings **10x-15x**

ACKNOWLEDGEMENTS

- Richard Wei
- Dan Zheng
- Brad Larson
- Andrew Savonichev
- Everyone involved & contributed to Differential Swift

Q & A