Models as Code
Differentiable Programming with Julia

Dr. Viral B. Shah and Dr. Elliot Saba
Software 2.0: Who will own the platform?

Yann LeCun
Director of AI Research, Facebook

OK, Deep Learning has outlived its usefulness as a buzz-phrase.
Deep Learning est mort. Vive Differentiable Programming!

Andrej Karpathy
Director of AI, Tesla

Software 2.0: Write a rough skeleton of the code, and use the computational resources at our disposal to search this space for a program that works.

Chris Lattner
Senior Director, TensorFlow, TPU (Google)

Julia is another great language with an open and active community. They are currently investing in machine learning techniques. The Julia community shares many common values as with our project.
3 Posters at NeurIPS 2018 (MLSys Workshop)

- Julia on TPUs
- Differentiable Programming
- Deep Learning
function foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c = tanh.(b)
    r = a + c
    return r
end

function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, Jtanh = ∇tanh.(b)
    a + c, function (∆r)
    ∆c = ∆r, ∆a = ∆r
    (Δtanh, Δb) = Jtanh(Δc)
    (ΔY, Δx) = (Δb * x', Y' * Δb)
    (ΔZ = Δa * x', Δx += Z' * Δa)
    (ΔW = ΔZ * Y', ΔY = W' * ΔZ')
    (nothing, ∆W, ∆Y, ∆x)
end
end

M. Innes. Don't Unroll Adjoint:
Differentiating SSA-Form Programs
(arXiv:1810.07951)
Cataloging the Visible Universe through Bayesian Inference at Petascale

Jeffrey Regier*, Kiran Pamnnay*, Keno Fischer†, Andreas Noack†, Maximilian Lam*, Jarrett Revels§, Steve Howard§, Ryan Giordano§, David Schlegel$, Jon McAuliffe§, Rollin Thomas§, Prabhat||

*Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
†Parallel Computing Lab, Intel Corporation
‡Julia Computing
§Computer Science and AI Laboratories, Massachusetts Institute of Technology
$Department of Statistics, University of California, Berkeley
||Lawrence Berkeley National Laboratory

Most light sources are near the detection limit.
Celeste: Custom sparsity patterns and storage

Matrix structure

Storage
CUDAnative.jl: Native code generation on GPUs

```julia
function vadd(gpu, a, b, c)
    i = threadIdx().x + blockDim().x * ((blockIdx().x-1) + (gpu-1) * gridDim().x)
    @inbounds c[i] = a[i] + b[i]
    return
end

a, b, c = (CuArray(...) for _ in 1:3)
@cuda threads=length(a) vadd(1, a, b, c)
```

```
@device_code_ptx @cuda vadd(1, a, a, a)
//@
//@ Generated by LLVM NVPTX Back-End
//@
.
.visible .entry ptxcall_vadd_23(
    .param .u64 ptxcall_vadd_23_param_0,
    .param .align 8 .b8 ptxcall_vadd_23_param_1[16],
    .param .align 8 .b8 ptxcall_vadd_23_param_2[16],
    .param .align 8 .b8 ptxcall_vadd_23_param_3[16]
) {
    mov.u32 %r1, %tid.x;
    mov.u32 %r2, %ntid.x;
    mov.u32 %r3, %ctaid.x;
    ...
}
```

Provides:
- CUDA intrinsics
- SPMD Programming model
- GPU memory management

CUDAnative.jl: As fast as CUDA C
Julia runs on Google TPUs

Performance on par with TensorFlow on TPUs

Scales to pods (512 TPU cores - 4.3 PF_{16}/s on ResNet50)

Fischer et al. Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs
(archive:1810.09868)
Google.ai Lead Jeff Dean on Julia

Julia + TPUs = fast and easily expressible ML computations!

Keno Fischer @KenoFischer
Our new paper today: arxiv.org/abs/1810.09868. Compile your julialang code straight to @Google's CloudTPU. Must go faster! We'll have an (alpha quality) repo up soon for people to start playing with this.

6:23 AM - 24 Oct 2018

240 Retweets 617 Likes
Best in class packages in many domains

- Differential Equations
- Graph Processing
- Data Science
- Image Processing
- Deep Learning
- Operations Research
- Signal Processing
- Computational Biology
Composability: DifferentialEquations.jl + Flux.jl (Neural ODEs)

Composability: DifferentialEquations.jl + Measurements.jl

\[ g = 9.79 \pm 0.02 \] # Gravitational constants
\[ L = 1.00 \pm 0.01 \] # Length of the pendulum

# Initial speed & angle, time span
\[ u_0 = [\theta \pm \theta, \pi/60 \pm 0.01] \]
\[ tspan = (0.0, 6.3) \]

# Define the problem

```julia
function pendulum(du, u, p, t)
    \[ \theta = u[1] \]
    \[ d\theta = u[2] \]
    \[ du[1] = d\theta \]
    \[ du[2] = -(g/L)*\theta \]
end
```

# Pass to solvers

```julia
prop = ODEProblem(pendulum, u_0, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)
```

# Analytic solution

\[ u = u_0[2] \times \cos(\sqrt{g/L} \times sol.t) \]

---


Giordano. Uncertainty propagation with functionally correlated quantities (arXiv:1610.08716)
Personalized Medicine in Partnership with UMB

Joga Gobburu
Professor, School of Pharmacy (UMB)
ex-Director, Division of Pharmacometrics, US FDA

Vijay Ivaturi
Professor, School of Pharmacy (UMB)

- Accurate personalized drug dosage calculation
- Clinical trial at Johns Hopkins with Vancomycin
- Demonstrate savings. Average cost of stay is $25,000. 20% savings expected.
- FDA approval and roll-out
Neural SDEs in Finance
The future for optimizing portfolios of hundreds of thousands of options

Chris Rackauckas et al.

- Semilinear Parabolic Form
  \[
  \frac{\partial u}{\partial t}(t, x) + \frac{1}{2} \text{Tr} \left( \sigma \sigma^T(t, x) (\text{Hess}_u(u(t, x))) \right) + \nabla u(t, x) \cdot \mu(t, x) + f(t, x, u(t, x), \sigma^T(t, x) \nabla u(t, x)) = 0
  \]

Then the solution of Eq. 1 satisfies the following BSDE (cf., e.g., refs. 8 and 9):

\[
\begin{align*}
    u(t, X_t) &= u(0, X_0) \\
    &= -\int_t^T \left[ f(s, X_s, u(s, X_s), \sigma^T(s, X_s) \nabla u(s, X_s)) \right] ds + \int_t^T \left[ \nabla u(s, X_s) \right]^T \sigma(s, X_s) dW_s. \\
    &= u \text{ via:}
\end{align*}
\]

- Financial Quants optimize portfolios through PDEs: Hamilton-Jacobi-Bellman, Nonlinear Black-Scholes.
- High dimensional PDEs are unsolvable by traditional mechanisms.
- New (2018) idea: transform it into a Backwards stochastic differential equation with a neural network inside of it.
- New computational challenge: get neural networks working inside of fancy mathematical (adaptive high order etc.) stochastic differential equation libraries.

We solve this by differentiating an existing library!

Solving high-dimensional partial differential equations using deep learning, 2018, PNAS, Han, Jentzen, E
Neural SDEs in action
Neural SDEs in action: non-linearly extrapolate time series with error bounds

See DiffEqFlux.jl Blog Post For Code Examples
https://github.com/FluxML/model-zoo

Julia is the only language with fast SDE solvers
And
Julia is the only language which can differentiate all of its differential equation solvers.
The next generation of tools for quants Is in Julia!
Fixing Boston’s school buses with route optimization

U.S. NEWS

THE NUMBERS | By Jo Craven McGinty

How Do You Fix School Bus Routes? Call MIT

A trio of MIT researchers recently tackled a tricky vehicle-routing problem when they set out to improve the efficiency of the Boston Public Schools bus system.

Last year, more than 30,000 students rode 650 buses to 230 schools at a cost of $120 million.

In hopes of spending less this year, the school system offered $15,000 in prize money in a contest that challenged competitors to reduce the number of buses.

The winners—Dimitris Bertsimas, co-director of MIT’s Operations Research Center and doctoral students Arthur Delaurier and Sebastien Martin—designed a system that drops as many as 75 bus routes.

A school system says the plan, which will eliminate

Road Test

Route planners often grapple with some form of the Traveling Salesman Problem where the solution is the shortest route that passes through each city once before returning home. Traveling to the nearest neighbor seems logical, but usually isn’t optimal.

The task of plotting school-bus routes resembles the classic math exercise known as the Traveling Salesman Problem, where the goal is to find the shortest path through a series of cities, visiting each only once, before returning home.

The nearest-neighbor solution usually produces the shortest route.

In one 42-city example that starts in Phoenix, following the nearest-neighbor approach usually produces the shortest route.

5

Optimal tour

9

Nearest neighbor tour

65 miles

54 miles

9 a.m.

9 a.m.

They first paired clusters overall system, Dr. Bertsimas says. Individual students may be on a bus for more or less time than last year, but in keeping with school-system rules, no bus trip should last more than one hour, Mr. Delaurier said.

Whether the plan will work as predicted remains to be seen. A previous effort to automate the system failed in 2013 when buses following routes created with software ran perpetually late. To avoid similar problems this year, the school system’s transportation staff vetted the MIT routes, making tweaks as needed.

“We wanted to make sure we were not picking up students on small streets with single-lane access, and staggered school days that start at 7:30 a.m., 8:30 a.m. or 9:00 a.m. They first paired clusters overall system,” Dr. Bertsimas said.
New climate model to be built from the ground up

Scientists and engineers will collaborate in a new Climate Modeling Alliance to advance climate modeling and prediction.

Study: Adding power choices reduces cost and risk of carbon-free electricity

To curb greenhouse gas emissions, nations, states, and cities should aim for a mix of fuel-saving, flexible, and highly reliable sources.
How to implement all that?

Easy, generic, fast: *pick two*

Common approaches

- Heroic C++ template code plus Python/R wrapper
- Problem-specific compilers
Language for describing what to specialize on.

- Design descriptive types for the domain at hand
- Write methods for whatever cases you can handle
- Compiler generates specializations
- These need to be *de-coupled*
Sliding scale of specialization

Array
Array{Int}
Array{Int,2}
Array{:<Any,2}
Array{:<Real,2}
SArray{(2,3),Float64,2}

Some kind of array
Element type known to be Int
... and 2-dimensional
... or unknown element type
... or unknown, but Real, element type
2d Float64 with dimensions 2x3
Other special matrix types

- Diagonal
- UniformScaling
- Symmetric, Hermitian
- LowerTriangular, UpperTriangular
- Bidiagonal, Tridiagonal, SymTridiagonal
- Adjoint, Transpose
Example: one-hot vector type

```haskell
struct OneHotVector <: AbstractVector{Bool}
    index::Int
    len::Int
end

size(xs::OneHotVector) = (xs.len,)

getindex(xs::OneHotVector, i::Integer) = i == xs.index

A::AbstractMatrix * b::OneHotVector = A[:, b.index]
```
Other special matrix types

- Diagonal
- UniformScaling
- Symmetric, Hermitian
- LowerTriangular, UpperTriangular
- Bidiagonal, Tridiagonal, SymTridiagonal
- Adjoint, Transpose
Recent work: type system formalization

Julia Subtyping: a Rational Reconstruction

F. Zappa Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, J. Vitek

OOPSLA 2018

Paper: https://www.di.ens.fr/~zappa/projects/lambdajulia/
[TOP] \[ \begin{array}{c}
E \vdash t : Any \vdash E \\
E \vdash a : a \vdash E
\end{array} \]
[REFL] \[ \begin{array}{c}
E \vdash a : a \vdash E
\end{array} \]
[TUPLE] \[ \begin{array}{c}
E \vdash a_1 : a'_1 \vdash E_1 \ldots E_{n-1} \vdash a_n : a'_n \vdash E_n \\
\text{consistent}(E_n)
\end{array} \]
\[ \begin{array}{c}
E \vdash \text{Tuple}\{a_1, \ldots, a_n\} < : \text{Tuple}\{a'_1, \ldots, a'_n\} \vdash E_n
\end{array} \]
[TUPLE\_LIFT\_UNION] \[ \begin{array}{c}
t' = \text{liftunion}(\text{Tuple}\{a_1, \ldots, a_n\})
\end{array} \]
\[ \begin{array}{c}
E \vdash t' : t \vdash E'
\end{array} \]
\[ \begin{array}{c}
E \vdash \text{Tuple}\{a_1, \ldots, a_n\} < : t \vdash E'
\end{array} \]
[TUPLE\_UNLIFT\_UNION] \[ \begin{array}{c}
t' = \text{unliftunion}(\text{Union}\{t_1, \ldots, t_n\})
\end{array} \]
\[ \begin{array}{c}
E \vdash t : t' \vdash E'
\end{array} \]
\[ \begin{array}{c}
E \vdash t < : \text{Union}\{t_1, \ldots, t_n\} \vdash E'
\end{array} \]
[UNION\_LEFT] \[ \begin{array}{c}
E \vdash t_1 < : t \vdash E_1 \ldots \text{resetocc}_E(E_{n-1}) \vdash t_n < : t \vdash E_n
\end{array} \]
\[ \begin{array}{c}
E \vdash \text{Union}\{t_1, \ldots, t_n\} < : t \vdash \text{maxocc}_E, \text{max}_{E_1, \ldots, E_n}(E_n)
\end{array} \]
[UNION\_RIGHT] \[ \begin{array}{c}
\exists j. E \vdash t < : t_j \vdash E'
\end{array} \]
\[ \begin{array}{c}
E \vdash t < : \text{Union}\{t_1, \ldots, t_n\} \vdash E'
\end{array} \]
[APP\_INV] \[ \begin{array}{c}
n \leq m \quad E_0 = \text{add}(\text{Barrier}, E)
\forall 0 < i \leq n. \quad E_{i-1} \vdash a_i : a'_i \vdash E_i \land E'_i \vdash a'_i < : a_i \vdash E_i
\end{array} \]
\[ \begin{array}{c}
E \vdash \text{name}\{a_1, \ldots, a_m\} < : \text{name}\{a'_1, \ldots, a'_n\} \vdash \text{del}(\text{Barrier}, E_n)
\end{array} \]
[APP\_SUPER] \[ \begin{array}{c}
\text{name}\{T_1, \ldots, T_m, \ldots\} < : t'' \in \text{tds}
\end{array} \]
\[ \begin{array}{c}
E \vdash t''[a_1/T_1 \ldots a_m/T_m] < : t' \vdash E'
\end{array} \]
\[ \begin{array}{c}
E \vdash \text{name}\{a_1, \ldots, a_m\} < : t' \vdash E'
\end{array} \]
[L\_INTRO] \[ \begin{array}{c}
\text{add}(^{L}_{i_1}T_{i_2}, E) \vdash t < : t' \vdash E'
\end{array} \]
\[ \begin{array}{c}
E \vdash t \text{ where } t_1 < : T < : t_2 < : t' < : \text{del}(T, E')
\end{array} \]
[R\_INTRO] \[ \begin{array}{c}
\text{add}(^{R}_{i_1}T_{i_2}, E) \vdash t < : t' \vdash E'
\end{array} \]
\[ \begin{array}{c}
\text{consistent}(T, E')
\end{array} \]
\[ \begin{array}{c}
E \vdash t < : t' \text{ where } t_1 < : T < : t_2 < : \text{del}(T, E')
\end{array} \]
Recent work: type system formalization

Julia Subtyping: a Rational Reconstruction
F. Zappa Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, J. Vitek

OOPSLA 2018

Paper: https://www.di.ens.fr/~zappa/projects/lambdajulia/
Exploring novel data types: BFloat16

- New numeric type used for machine learning on TPUs
- 8 mantissa bits, 8 exponent bits
- Efficient Julia implementation is <100 LOC
- Harmonic sum in floating point (Source: Nick Higham's blog)

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Computed Sum</th>
<th>Number of terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfloat16</td>
<td>5.0625</td>
<td>65</td>
</tr>
<tr>
<td>fp16</td>
<td>7.0859</td>
<td>513</td>
</tr>
<tr>
<td>fp32</td>
<td>15.404</td>
<td>2097152</td>
</tr>
<tr>
<td>fp64</td>
<td>34.122</td>
<td>2.81 \cdots \times 10^{14}</td>
</tr>
</tbody>
</table>
A Global Community
Over 3 Million Downloads. 2,500 Packages.
A Growing Community

Julia GitHub Stars

*Julia Language Only - Does Not Include Julia Packages*

- 2012: 1,230
- 2013: 2,550
- 2014: 3,840
- 2015: 5,130
- 2016: 6,450
- 2017: 7,740
- 2018: 9,030
- 2019: 19,472
If $d$ is a valid descent direction, then there must exist a sufficiently small step size that satisfies the sufficient decrease condition. We can thus start with a large step size and decrease it by a constant reduction factor until the sufficient decrease condition is satisfied. This algorithm is known as backtracking line search because of how it backtracks along the descent direction. Backtracking line search is shown in figure 4.2 and implemented in algorithm 4.2. We walk through the procedure in example 4.2.

![Graph showing the sufficient decrease condition.](image)

Algorithm 4.2. The backtracking line search algorithm, which takes objective function $f$, its gradient $\nabla f$, the current design point $x$, a descent direction $d$, and the maximum step size $\alpha$. We can optionally specify the reduction factor $p$ and the first Wolfe condition parameter $\beta$.

```plaintext
function backtracking_line_search(f, \nabla f, x, d, \alpha; p=0.5, \beta=1e-4)
    y, g = f(x), \nabla f(x)
    while \( f(x + \alpha \cdot d) > y + \beta \cdot \alpha \cdot (g \cdot d) \)
        \( \alpha *= p \)
    end
    \( \alpha \)
end
```
Algorithm 7.8. DIRECT, which takes the multidimensional objective function $f$, vector of lower bounds $a$, vector of upper bounds $b$, tolerance parameter $\epsilon$, and number of iterations $k_{max}$. It returns the best coordinate.
Figure 7.20. The DIRECT method after 16 iterations on the Branin function, appendix B.3. Each cell is bordered by white lines. The cells are much denser around the minima of the Branin function, as the DIRECT method procedurally increases its resolution in those regions.
Some of the Universities Teaching Julia
For the development of Julia, an innovative environment for the creation of high-performance tools that enable the analysis and solution of computational science problems.

Julia allows researchers to write high-level code in an intuitive syntax and produce code with the speed of production programming languages. Julia has been widely adopted by the scientific computing community for application areas that include astronomy, economics, deep learning, energy optimization, and medicine.

In particular, the Federal Aviation Administration has chosen Julia as the language for the next generation airborne collision avoidance system.
JuliaCon 2019 in Baltimore

Sponsorship opportunities open
A main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows.

Guy Steele, “Growing a language”, 1998
An Introduction to Zygote: Linear Regression

In this notebook, we will define Linear Regression in Zygote from scratch, showing how easy it is to take derivatives of custom code.

In [1]:
```py
# Initialize environment in current directory, to load
import Pkg; Pkg.activate(@__DIR__); Pkg.instantiate();

using Zygote, LinearAlgebra
```

This example will showcase how we do a simple linear fit with Zygote, making use of complex datastructures, a home-grown stochastic gradient descent optimizer, and some good old-fashioned math. We start with the problem statement: We wish to learn the mapping $f(X) \rightarrow Y$, where $X$ is a matrix of vector observations, $f()$ is a linear mapping function and $Y$ is a vector of scalar observations.

Because we like complex objects, we will define our linear regression as the following object:

In [2]:
```py
mutable struct LinearRegression
    # These values will be implicitly learned
    weights::Matrix
    bias::Float64
    # These values will not be learned
    name::String
end
```
This example will showcase how we do a simple linear fit with Zygote, making use of complex data structures, a home-grown stochastic gradient descent optimizer, and some good old-fashioned math. We start with the problem statement: We wish to learn the mapping \( f(X) \to Y \), where \( X \) is a matrix of vector observations, \( f() \) is a linear mapping function and \( Y \) is a vector of scalar observations.

Because we like complex objects, we will define our linear regression as the following object:

```julia
mutable struct LinearRegression
    # These values will be implicitly learned
    weights::Matrix
    bias::Float64
    # These values will not be learned
    name::String

    LinearRegression(nparams, name) = LinearRegression(randn(1, nparams), 0.0, name)
end
```

We will define two verbs to act upon a `LinearRegression` object; `predict()`, to perform the linear regression, and `loss()` to measure the \( L_2 \) norm between a target and our current prediction.

```julia
function predict(model::LinearRegression, X)
    return model.weights * X .+ model.bias
end
```
# Now we begin our "training loop", where we take examples from `X`,
calculate loss with respect to the corresponding entry in `Y`, find the
# gradient upon our model, update the model, and continue. Before we jump
# in, let's look at what `Zygote.gradient` gives us:
```
model = LinearRegression(size(X, 1), "Example")
```
# Calculate gradient upon `model` for the first example in our training set
gs = Zygote.gradient(model) do m
        return loss(m, X[:, 1], Y[3])
end
```

The `gs` object is a Tuple containing one element per argument to `gradient()`, so we take the first one to get the gradient upon `model`:
```
In [16]: gs = gs[1]
Out[16]: (weights = [0.597277958228296 1.41050189509322 0.5269798784518928 0.24937720677402483], bias = -1.0, name = nothing)
```

Because our LinearRegression object is mutable, the gradient holds a reference to it, which we peel via `gs[]`:
```
In [17]: gs = gs[]
Out[17]: (weights = [0.597277958228296 1.41050189509322 0.5269798784518928 0.24937720677402483], bias = -1.0, name = nothing)
```
```python
bias_gt = 0.4

# Generate a dataset of many observations
X = randn(length(weights_gt), 10000)
Y = weights_gt * X .+ bias_gt

# Add a little bit of noise to `X` so that we do not have an exact solution,
# but must instead do a least-squares fit:
X .+= 0.001 .* randn(size(X))

Out[6]: 4×10000 Array{Float64,2}:
       0.19203  1.91616  0.843656  -1.57857  0.246021  -0.597997
         0.48234  1.32798  -0.899407   0.043473  1.448148  -0.273722
         1.55865  0.0716222  1.89751   0.396072  0.638415  -0.273722
         0.558112  1.620704  0.954263   0.442166  -0.187531  -0.922375

In [8]:
# Now we begin our "training loop", where we take examples from `X`,
# calculate loss with respect to the corresponding entry in `Y`, find the
# gradient upon our model, update the model, and continue. Before we jump
# in, let's look at what `Zygote.gradient()` gives us:
model = LinearRegression(size(X, 1), "Example")

g = Zygote.gradient(model, X[:, 1], Y[1])

# Calculate gradient upon `model` for the first example in our training set
```
```
We will define two verbs to act upon a `LinearRegression` object: `predict()`, to perform the linear regression, and `loss()` to measure the $\ell_2$ norm between a target and our current prediction.

```julia
# Our linear regression looks very familiar; w^T X + b
function predict(model::LinearRegression, X)
    return model.weights' * X .+ model.bias
end

# Our "loss" that must be minimized is the $l_2$ norm between our current prediction and our ground-truth $Y$
function norm(model::LinearRegression, X, Y)
    return norm(predict(model, X) - Y, 2)
end
```

Our ground truth values (that we will learn, to prove that this works)

```julia
weights_gt = [1.0, 2.7, 0.3, 1.2]'
bias_gt = 0.4

# Generate a dataset of many observations
X = randn(length(weights_gt), 10000)
Y = weights_gt * X .+ bias_gt
```
# Now we begin our "training loop", where we take examples from `X`,
# calculate loss with respect to the corresponding entry in `Y`, find the
# gradient upon our model, update the model, and continue. Before we jump
# in, let's look at what `Zygote.gradient()` gives us:

```julia
model = LinearRegression(size(X, 1), "Example")
```

# Calculate gradient upon `model` for the first example in our training set
```julia
grads = Zygote.gradient(model) do m
    return loss(m, X[1,1], Y[1])
end
```

```
(Base.RefValue{Any}((weights = [-1.192026114794742, -0.482340240894484, 1.5586551748882012, -0.5581118308574825], bias = 1.0, name = nothing)),)
```

The `grads` object is a Tuple containing one element per argument to `gradient()`, so we take the first one to get the gradient upon `model`:

```
In [16]:
grads = grads[1]
```

```
(Base.RefValue{Any}((weights = [0.597277958228296, -1.4105189509322, 0.5269798784518928, 0.24937720677402483], bias = -1.0, name = nothing)),)
```

Because our `LinearRegression` object is mutable, the gradient holds a reference to it, which we peel via `grads[]`:

```
In [17]:
grads = grads[]
```
Zygote.jl: General Purpose Automatic Differentiation

```
function foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c = tanh.(b)
    r = a + c
    return r
end

function gradfoo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, Jtanh = grad(tanh.(b))
    a + c, function (∆r)
        ∆c = ∆r, ∆a = ∆r
        (Jtanh, ∆b) = Jtanh(∆c)
        (∆Y, ∆x) = (∆b * x', Y' * ∆b)
        (∆Z = ∆a * x', ∆x += Z' * ∆a)
        (∆W = ∆Z * Y', (∆Y = W' * ∆Z)'
    end
end
```

M. Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs (arXiv:1810.07951)
Celeste.jl: Julia at Peta-scale
Cori: 650,000 cores. 1.3M threads. 60 TB of data.

Cataloging the Visible Universe through Bayesian Inference at Petascale

Jeffrey Regier*, Kiran Pamnani†, Keno Fischer†, Andreas Noack†, Maximillian Lam*, Jarrett Revels†, Steve Howard†, Ryan Giordano†, David Schlegel†, Jon McAuliffe†, Rollin Thomas†, Prabhat†

*Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
†Parallel Computing Lab, Intel Corporation
†Julia Computing
†Computer Science and AI Laboratories, Massachusetts Institute of Technology
*Department of Statistics, University of California, Berkeley
†Lawrence Berkeley National Laboratory

Most light sources are near the detection limit.
Zygote.jl: General Purpose Automatic Differentiation

function foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c = tanh.(b)
    r = a + c
    return r
end

function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, ∇tanh = ∇tanh.(b)
    a + c, function (Δr)
        Δc = Δr, Δa = Δr
        (Δtanh, Δb) = ∇tanh(Δc)
        (ΔY, Δx) = (Δb * x', Y' * Δb)
        (ΔZ = Δa * x', Δx += Z * Δa)
        (ΔW = ΔZ * Y', ΔY = W' * ΔZ')
        (nothing, ΔW, ΔY, Δx)
    end
end

M. Innes. Don't Unroll Adjoint:
Differentiating SSA-Form Programs
(arXiv:1810.07951)
The `grads` object is a Tuple containing one element per argument to `gradient()`, so we take the first one to get the gradient upon `model`:

```
In [16]: 1 grads = grads[1]
```

```
Out[16]: Base.RefValue(Any)(weights = [0.59727795828296 -1.41050189509322 0.5269798784518928 0.24937720677402483], bias = -1.0, name = nothing))
```

Because our `LinearRegression` object is mutable, the gradient holds a reference to it, which we peel via `grads[]`:

```
In [17]: 1 grads = grads[]
```

```
Out[17]: (weights = [0.59727795828296 -1.41050189509322 0.5269798784518928 0.24937720677402483], bias = -1.0, name = nothing)
```

We now get a `NamedTuple` so we can now do things like `grads.weights`. Note that while `weights` and `bias` have gradients, `name` just naturally has a gradient of `nothing`, because it was not involved in the calculation of the output loss.
# Calculate gradient upon `model` for the first example in our training set

```julia
grads = Zygote.gradient(model) do m
    return loss(m, X[1], Y[1])
end
```

```
Out[8]: (Base.RefValue{Any}((weights = [-1.192026114794742 -0.482340240894484 1.5586551748882012 -0.558118308574825], bias = 1.0, name = nothing)),)
```

The `grads` object is a Tuple containing one element per argument to `gradient()`, so we take the first one to get the gradient upon `model`:

```
In [9]: grads = grads[1]
```

```
Out[9]: Base.RefValue{Any}((weights = [-1.192026114794742 -0.482340240894484 1.5586551748882012 -0.558118308574825], bias = 1.0, name = nothing))
```

Because our `LinearRegression` object is mutable, the gradient holds a reference to it, which we peel via `grads[]`:

```
In [17]: grads = grads[]
```

```
Out[17]: (weights = [0.5972779582828296 -1.41050189509322 0.5269798784518928 0.24937720677402483], bias = -1.0, name = nothing)
```

We now get a `NamedTuple` so we can now do things like `grads.weights`. Note that while `weights` and `bias` have gradients, `name` just naturally has a gradient of `nothing`, because it was not involved in the calculation of the output loss.
Because our LinearRegression object is mutable, the gradient holds a reference to it, which we peel via `grads[]`:

```
In [10]: grads = grads[]
```

We now get a `NamedTuple` so we can now do things like `grads.weights`. Note that while `weights` and `bias` have gradients, `name` just naturally has a gradient of `nothing`, because it was not involved in the calculation of the output loss.

```
In [18]: grads.weights
```

```
Out[18]: 1x4 Array{Float64,2}
    0.597278  -1.4105  0.52698  0.249377
```

Next, we will define an update rule that will allow us to modify the weights of our model according to the gradients, using the simplest gradient descent update rule. We'll then run a training loop to update our weights with the loss from the training set, as we would expect:

```
# Let's define

function sgd_update!(model::LinearRegression, grads, \eta = 0.001)
    model.weights .= \eta * grads.weights
end
```
We now get a NamedTuple so we can now do things like `grads.weights`. Note that while `weights` and `bias` have gradients, `name` just naturally has a gradient of `nothing`, because it was not involved in the calculation of the output loss.

```julia
In [11]: gradients.weights
Out[11]: 1×4 Array{Float64,2}:
    -1.19203  -0.48234   1.55866  -0.558112
```

Next, we will define an update rule that will allow us to modify the weights of our model according to the gradients, using the simplest gradient descent update rule. We'll then run a training loop to update our weights with the loss from the training set, as we would expect:

```julia
In [21]:
function sgd_update!(model::LinearRegression, grads, η = 0.001)
    model.weights .-= η .* grads.weights
    model.bias -= η .* grads.bias
end
```

```julia
Out[21]: sgd_update! (generic function with 2 methods)
```

```julia
In [22]: # Now let's do that for each example in our training set:
# info("Running train loop for $(size(X,2)) iterations")
for idx in 1:size(X, 2)
    grads = Zygote.gradient(m -> loss(m, X[:, idx], Y[idx]), model)[1])
```

```julia
```
Next, we will define an update rule that will allow us to modify the weights of our model according to the gradients, using the simplest gradient descent update rule. We'll then run a training loop to update our weights with the loss from the training set, as we would expect:

```julia
In [21]:

function sgd_update!(model::LinearRegression, grads, η = 0.001)
    model.weights .= η * grads.weights
    model.bias  -= η * grads.bias
end

Out[21]:
```

Now let's do that for each example in our training set:
```julia
In [22]:

@info("Running train loop for \(\{size(X, 2)\}\) iterations")
for idx in 1:size(X, 2)
    grads = Zygote.gradient(m -> loss(m, X[:, idx], Y[idx]), model)[1][1]
    sgd_update!(model, grads)
end
```

```
In [23]:
weights_gt
```
```julia
function sgd_update!(model::LinearRegression, grads, η = 0.001)
    model.weights .= η * grads.weights
    model.bias -= η * grads.bias
end
```

```
# Let's define

function sgd_update!(model::LinearRegression, grads, η = 0.001)
    model.weights .= η * grads.weights
    model.bias -= η * grads.bias
end

Out[12]: sgd_update! (generic function with 2 methods)
```

```
# Now let's do that for each example in our training set:

@info("Running train loop for $(size(X,2)) iterations")
for idx in 1:size(X, 2)
    grads = Zygote.gradient(m -> loss(m, X[:, idx], Y[idx]), model)[1][]
    sgd_update!(model, grads)
end

Info: Running train loop for 10000 iterations
```

```
In [22]: weights_gt
Out[23]: 1x4 Adjacent{Float64,Array{Float64,1}}:
1.0 2.7 0.3 1.2
```

```
In [24]: model.weights
Out[24]: 1x4 Array{Float64,2}:
0.999031 2.69703 0.301551 1.20023
```

```
In [25]: bias_gt
```
```julia
# Let's define
function sgd_update!(model::LinearRegression, grads, η = 0.001)
    model.weights .-= η * grads.weights
    model.bias -= η * grads.bias
end
```

```
# Now let's do that for each example in our training set:
@info("Running train loop for \$(\text{size}(X, 2))\ iterations")
for idx in 1:size(X, 2)
    grads = Zygote.gradient(m -> loss(m, X[:, idx], Y[idx]), model)[1][]
    sgd_update!(model, grads)
end
```

```
weights_gt
1×4 Adjacent{Float64,Array{Float64,1}}:
  1.0  2.7  0.3  1.2

model.weights
1×4 Array{Float64,2}:
  0.999031  2.69703  0.301551  1.20023
```

```
bias_gt
0.4
```

```julia
function sgd_update!(model::LinearRegression, grads, η = 0.001)
    model.weights .= η * grads.weights
    model.bias .= η * grads.bias
end
```

```
# Let's define

Out[12]:

sgd_update! (generic function with 2 methods)
```

```
# Now let's do that for each example in our training set:

In[13]:

@info("Running train loop for $(size(X, 2)) iterations")
for idx in 1:size(X, 2)
    grads = Zygote.gradient(m -> loss(m, X[:, idx], Y[idx]), model[1])
    sgd_update!(model, grads)
end

Info: Running train loop for 10000 iterations
In Main In[13]:2
```

```
In[23]:

weights_gt

Out[23]:

1x4 Adjoint(Float64, Array{Float64,1}):
1.0  2.7  0.3  1.2
```

```
In[24]:

model.weights

Out[24]:

1x4 Array{Float64,2}:
0.999031  2.69703  0.301551  1.20023
```

```
In[25]:

bias_gt
```
model.bias -= η * grads.bias

### Out[12]:
sgd_update! (generic function with 2 methods)
# Now let's do that for each example in our training set:

```julia
@info("Running train loop for \{size(X,2)\} iterations")
for idx in 1:size(X, 2)
    grads = Zygote.gradient(m -> loss(m, X[:, idx], X[idx], model)[1][1])
    sgd_update!(model, grads)
end
```

```
INFO: Running train loop for 10000 iterations
```

```
weights_gt
```

```
1x4 AdjArray{Float64,1}:
    1.0  2.7  0.3  1.2
```

```
model.weights
```

```
1x4 Array{Float64,2}:
    0.999031  2.69703  0.301551  1.20023
```

```
bias_gt
```

```
0.4
```

```
model.bias
```

```
0.39980000000000035
```
Info: Running train loop for 10000 iterations

```
In [14]: weights_gt
Out[14]: 1x4 Adjacent{Float64,Array{Float64,1}}:
    1.0  2.7  0.3  1.2

In [15]: model.weights
Out[15]: 1x4 Array{Float64,2}:
    1.00142  2.70157  0.300252  1.20033

In [16]: bias_gt
Out[16]: 0.4

In [17]: model.bias
Out[17]: 0.3980000000000003
```
Zygote Continued: A Differentiable Raytracer

We demonstrate in this notebook differentiating through a raytracer

```
# Initialize environment differentiating in current directory, to load
import Pkg; Pkg.activate(\$DIR); Pkg.instantiate()

using RayTracer, Zygote, Flux, Images, Statistics, Interact
```

```
@ Pkg.API /Users/sabae/tmp/julia-build/julia-release-1.2/usr/share/julia/stdlib/v1.2/Pkg/src/API.jl:564

Updating registry at `~/.julia/registries/julia-release-1.2/`
Updating git-repo `https://github.com/JuliaRegistries/General`

r Warning: Some registries failed to update:
  /Users/sabae/julia/registries/General - failed to fetch from repo
@ Pkg.Types /Users/sabae/tmp/julia-build/julia-release-1.2/usr/share/julia/stdlib/v1.2/Pkg/src/Types.jl:1171
```

```
In [2]:
width = 200
height = 200

# Static camera configuration
cam = Camera()
# Center
```
```julia
width = 200
height = 200

# Static camera configuration
cam = Camera(
    # Center
    Vec3(0.0f0, 0.0f0, -5.0f0),
    # Target
    Vec3(0.0f0, 0.0f0, 0.0f0),
    # Up
    Vec3(0.0f0, 1.0f0, 0.0f0),
    # Field of View
    45.0f0,
    # Focus
    1.0f0,
    # Resolution
    width, height,
)

origin, direction = get_primary_rays(cam)

function render(scene, light)
    packed_image = raytrace(origin, direction, scene, light, origin, 0)
    array_image = reshape(hcat(packed_image.x, packed_image.y, packed_image.z), (width, height, 3, 1))

    return array_image
end

function showing(img)
    return colorview(RGB, permutedims(img[:,:,1], (3,2,1)))
end
```

# Initialize environment in current directory, to load
import Pkg; Pkg.activate(@__DIR__); Pkg.instantiate()

using RayTrace, Zygote, Flux, Images, Statistics, Interact

```
\[
\text{Updating registry at `~/julia/registries/General`}
\text{Updating git-repo `https://github.com/JuliaRegistries/General.git`}
\]
```

```
\[
\text{Info: Recompiling stale cache file `/Users/sabae/.julia/compiled/vl.2/RayTrace/sUryZ.ji` for RayTrace [60dabc86-48f7-11e9-0f01-03ab8794bbcc9]}
\]
```

```
In [2]:
width = 200
height = 200

# Static camera configuration
cam = Camera(
# Center
Vec3(0.0f0, 0.0f0, -5.0f0),
# Target
Vec3(0.0f0, 0.0f0, 0.0f0),
# Up
Vec3(0.0f0, 1.0f0, 0.0f0),
# Field of View
45.0f0,
# Focus
I
```

In [2]:

1. width = 200
2. height = 200
3. # Static camera configuration
4. cam = Camera(
5.   # Center
6.   Vec3(0.0f0, 0.0f0, -5.0f0),
7.   # Target
8.   Vec3(0.0f0, 0.0f0, 0.0f0),
9.   # Up
10.  Vec3(0.0f0, 1.0f0, 0.0f0),
11.  # Field of View
12.  45.0f0,
13.  # Focus
14.  1.0f0,
15.  # Resolution
16.  width, height,
17. )
18. origin, direction = get_primary_rays(cam)
19. 
20. function render(scene, light)
21.     packed_image = raytrace(origin, direction, scene, light, origin, 0)
22.     array_image = reshape(hcat(packed_image.x, packed_image.y, packed_image.z), (width, height, 3, 1))
23.     return array_image
24. end
25. 
26. function showing(img)
27.     return colorview(RGB, permutedims(img[:, :, 1], (3, 2, 1)))
28. end
```python
loss = mean((zeroonenorm(image_rendered) - zeroonenorm(image_gt))^2)

# Show the current loss
@show loss

# Return this loss as what is to be minimized
return loss

# Update our light position and triangle color based upon those gradients
# update!(opt, scene[j].material.color.color, grads[1][j].material.color.color)
update!(opt, light.position, grads[2].position)

if i % 10 == 1
    @info "$i$ iterations completed"
end
display(showing(render(scene, light)))
```

```
In [29]: light_gt
Out[29]: PointLight(Float32)(Vec3{Array(Float32, 1)}(Float32[1.0], Float32[1.0], Float32[0.0]), 20000.0f0, Vec3{Array(Float32, 1)}(Float32[3.6], Float32[3.0], Float32[-10.0]))

In [30]: light
Out[30]: PointLight(Float32)(Vec3{Array(Float32, 1)}(Float32[1.0], Float32[1.0], Float32[0.0]), 20000.0f0, Vec3{Array(Float32, 1)}(Float32[4.18166], Float32[2.4189723], Float32[-9.362359]))
```
Show the current loss
return loss

Return this loss as what is to be minimized

Update our light position and triangle color based upon those gradients

if i % 10 == 1
    @info "${i} iterations completed"
end
display(showing(render(scene, light)))
```julia
opt = Adam(0.1)
image_gt = render(scene_gt, light_gt)
showimg(image_gt)

for i in 1:51
    # Take gradient of the following function
    grads = gradient(scene, light) do S, L
        # First, render according to our current light and scene
        image_rendered = render(S, L)

        # Normalize
        loss = mean((zeroonenorm(image_rendered) .- zeroonenorm(image_gt)).^2)
        # Show the current loss
       @show loss
        # Return this loss as what is to be minimized
        return loss
    end
    # Update our light position and triangle color based upon those gradients
    update!(opt, scene[1].material.color.color, grads[1][1].material.color.color)
    update!(opt, light.position, grads[2].position)
    if i % 10 == 1
        @info "$i iterations completed"
        display(showimg(render(scene, light)))
    end
end
```
Zygote.jl: General Purpose Automatic Differentiation

function foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c = tanh(b)
    r = a + c
    return r
end

function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, Jtanh = ∇tanh(b)
    a + c, function (Δr)
        Δc = Δr, Δa = Δr
        (Δtanh, Δb) = Jtanh(Δc)
        (ΔY, Δx) = (Δb * x', Y' * Δb)
        (ΔZ = Δa * x', Δx += Z' * Δa)
        (ΔW = ΔZ * Y', ΔY = W' * ΔZ')
    end
end

M. Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs (arXiv:1810.07951)
Target and environment variables

- wind = -10 m/s
- target = 50 m

Control Parameters

- angle = 25°
- weight = 200 kg

Loss

\[(target\text{\_distance} - actual\text{\_distance})^2\]

Neural Network

ODE Solver

Backpropagation
Backpropagation

Loading your Trebuchet

Today we practice the ancient medieval art of throwing stuff. First up, we load our trebuchet simulator, Trebuchet.jl.

```
In [1]:
# Initialize environment in current directory, to load
1 import Pkg; Pkg.activate(_DIR_); Pkg.instantiate()
2 using Trebuchet

   Info: activating environment at `~/.src/msr_talk/Project.toml`
   @ Pkg.API /Users/sabae/tmp/julia-build/julia-release-1.2/usr/share/julia/stdlib/v1.2/Pkg/src/API.jl:564

   Updating registry at `~/.julia/registries/General`
   Updating git-repo `https://github.com/JuliaRegistries/General.git`

   Info: Precompiling Trebuchet [98b73d46-197d-11e9-11eb-69a6ff759d3a]
   @ Base loading.jl:1242
```

We can see what the trebuchet looks like, by explicitly creating a trebuchet state, running a simulation, and visualising the trajectory.
We can see what the trebuchet looks like, by explicitly creating a trebuchet state, running a simulation, and visualising the trajectory.

```python
In [*]:
1: t = TrebuchetState()
2: simulate(t)
3: visualise(t)
```

For training and optimisation, we don't need the whole visualisation, just a simple function that accepts and produces numbers. The `shoot` function just takes a wind speed, angle of release and counterweight mass, and tells us how far the projectile got.

```python
In [3]:
1: function shoot(wind, angle, weight)
2:     Trebuchet.shoot((wind, Trebuchet.deg2rad(angle), weight))
3: end

Out[3]: shoot (generic function with 1 method)
```

```python
In [4]:
1: shoot(0, 30, 400)

Out[4]: 98.60072421662711
```

It's worth playing with these parameters to see the impact they have. How far can you throw the projectile, tweaking only the angle of release?

There's actually a much better way of aiming the trebuchet. Let's load up a machine learning library, Flux, and see what we can do.

```python
In [5]:
1: pathof(Trebuchet)

Out[5]: "/Users/sabae/.julia/packages/Trebuchet/dUL6T/src/Trebuchet.jl"
```
We can see what the trebuchet looks like, by explicitly creating a trebuchet state, running a simulation, and visualising the trajectory.

```
In [2]: t = TrebuchetState()
simulate(t)
visualise(t)
```

Out[2]:

1 m/s
In [22]:
0. t = TrebuchetState(release_angle = deg2rad(19), wind_speed = -10)
1. simulate(t)
2. visualise(t)
In [24]:
1. t = TrebuchetState()
2. simulate(t)
3. visualise(t, 50)

Out[24]:

1 m/s

Distance 0 m