Introduction to Automatic Differentiation & Differentiable Programming

Dr Vassil Vassilev, Princeton; Dr Jonas Rembser, CERN

compiler-research.org

From Integrals to Likelihoods

Likelihoods are central for High Energy Physics

$$L(\vec{n},\vec{a}|\vec{\eta},\vec{\chi}) = \prod_{c\in unbinned\ ch}\prod_{i\in obs}\frac{f_c(\vec{x}_{ci}|\vec{\eta},\vec{\chi})}{\int f_c(\vec{x}_{ci}|\vec{\eta},\vec{\chi})\,d\vec{x}_c}\cdot \prod_{c\in binned\ ch(analytical)}\prod_{b\in obs} Pois(n_{cb}|\nu(\vec{\eta},\vec{\chi}))\cdot\prod_{\chi\in\vec{\chi}} c_\chi(a_\chi|\chi)$$

Numerical and analytic integrals

$\vec{n}: data, \vec{a}: auxilary\ data, \vec{\eta}: unconstrained\ parameters, \vec{\chi}: constrained\ parameters$

CMS Combine Paper https://arxiv.org/pdf/2404.06614

Object Oriented Math with RooFit

$$g_1(\mathrm{x}) = \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma_1}\right)^2}$$

$$g_2(\mathrm{x}) = \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma_2}\right)^2}$$

$$P_{bkg}(\mathrm{x}) = \frac{1 + a_0 * T_1(x) + a_1 * T_2(x)}{\int 1 + a_0 * T_1(x) + a_1 * T_2(x)}$$

$$S(\mathrm{x}) = f_{sig1} g_1(x) + \left(1 - f_{sig1}\right) g_2(x)$$

$$\mathrm{Model}(\mathrm{x}) = f_{bkg} P_{bkg}(x) + \left(1 - f_{bkg}\right) S(x)$$

$$a_0 = 0.5, a_1 = 0.2, f_{sig1} = 0.8, f_{bkg} = 0.5,$$

$$\mu = 5, \sigma_1 = 0.5, \sigma_1 = 1.0$$

```
RooGaussian sig1("sig1", "Signal component 1", x, mu, sigma1);
RooGaussian sig2("sig2", "Signal component 2", x, mu, sigma2);

// Build Chebychev polynomial pdf
RooChebychev bkg("bkg", "Background", x, {a0, a1});

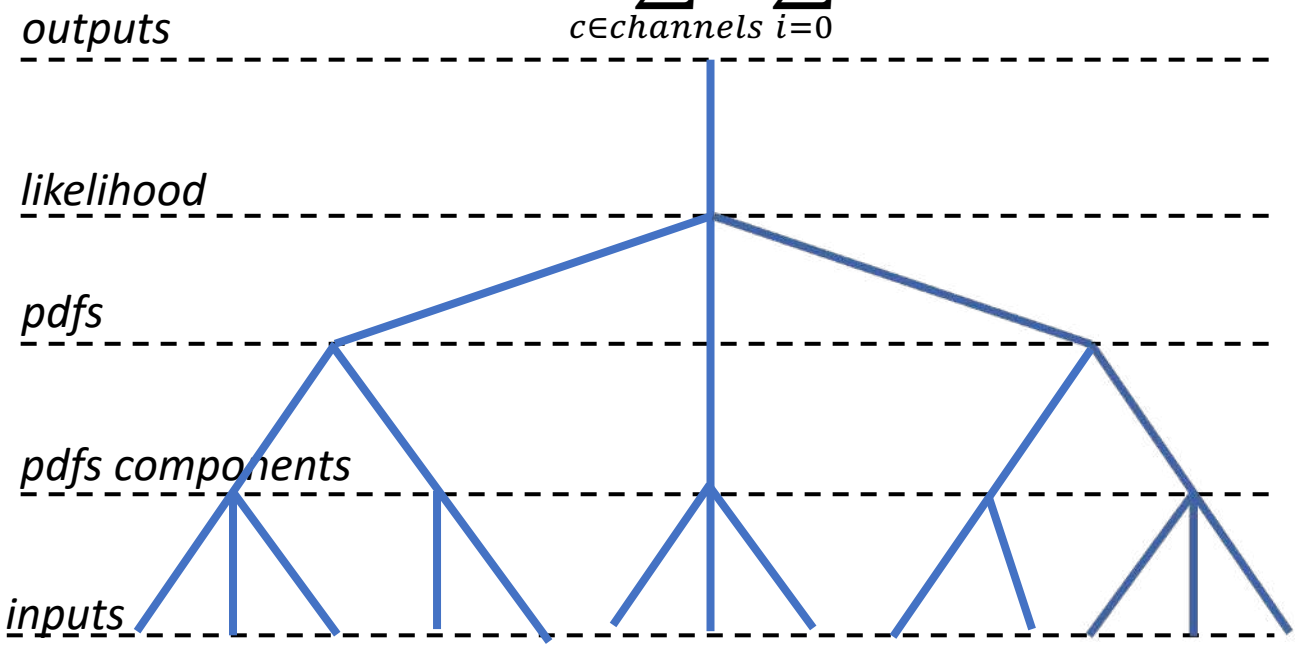
// Sum the signal components into a composite signal pdf
RooRealVar sig1frac("sig1frac", "fraction of c 1 in signal", 0.8, 0.,
1.);
RooAddPdf sig("sig", "Signal", {sig1, sig2}, sig1frac);

// Sum the composite signal and background
RooRealVar bkgfrac("bkgfrac", "fraction of background", 0.5, 0., 1.);
RooAddPdf model("model", "g1+g2+a", {bkg, sig}, bkgfrac);

// Create NLL function
std::unique_ptr<RooAbsReal> nll{model.createNLL(*data,
EvalBackend("codegen"))};
```

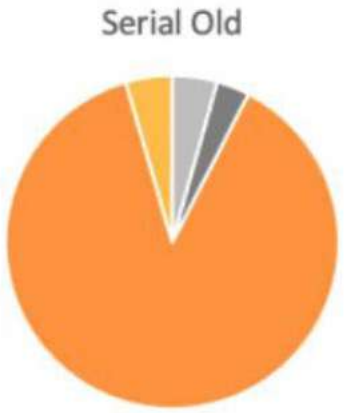
Object Oriented Math. Compute Cost

$$NLL = -\sum_{c \in channels} \sum_{i=0}^{N_c} \log\left(Model(x_i)\right)$$



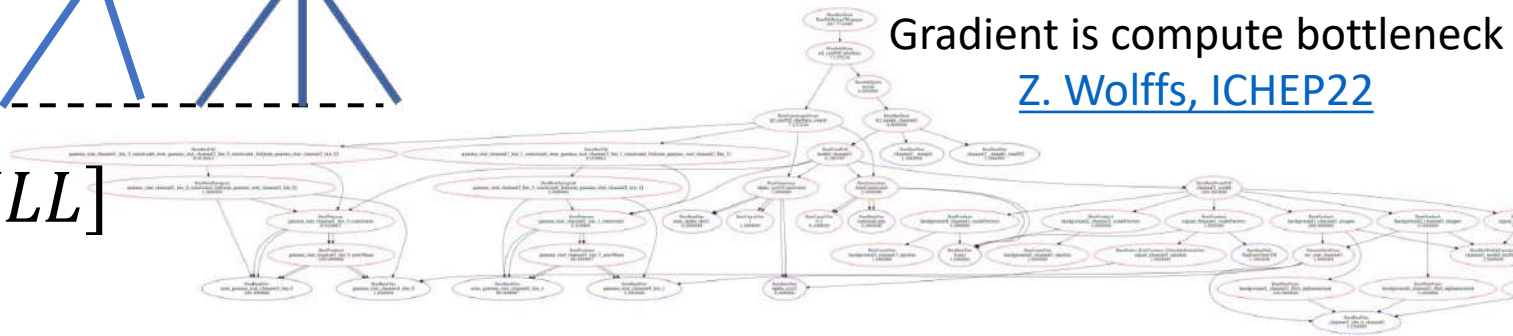
$$(\hat{\eta}, \hat{\chi}) = \arg\min_{\eta,\chi}[NLL]$$

serial old	
roofit_setup	313
migrad_seed	230
migrad_gradient	6289
migrad_descent	323



Gradient is compute bottleneck

Z. Wolffs, ICHEP22



Same Discipline. The Fix Doesn't Reach.

The profile says: the gradient. So do what we just did to smallpt: parallelize it.

And you can — the partials are independent

- finite-difference partials are independent
- RooFit could parallelize the gradient
  (Wolffs, ICHEP22 — previous slide)

But look at what parallelism actually buys

`cost(∇NLL) = O(N) · cost(NLL)`

cores → divide the CONSTANT

÷P, Amdahl-capped; descent stays sequential

N → survives. N = 672, and growing

$$f'(x) = \frac{f(x) - f(x-h)}{h}$$

And speed is not the worst of it: ∂NLL < precision(NLL).

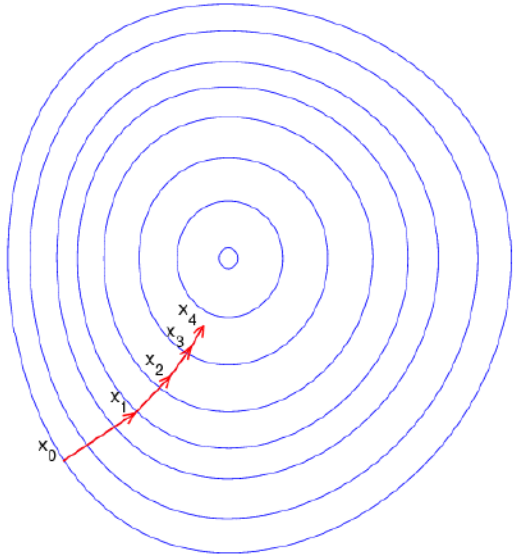
Finite differences here are not slow. They are wrong. We will see empirical evidence later on.

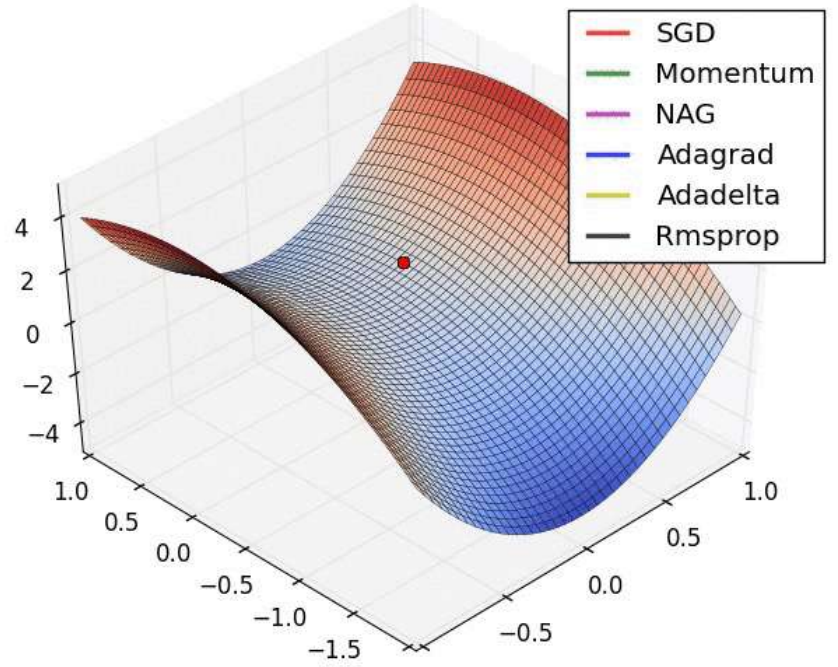
Don't compute it faster. Compute something else.

One input is easy. Real problems have millions

The gradient ∇f is the vector of partial derivatives; entry *i* is $\partial f/\partial x_i$...

$$\nabla f(x1, \ldots, x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x1, \ldots, x_n) \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial f}{\partial x_n}(x1, \ldots, x_n) \end{bmatrix}$$

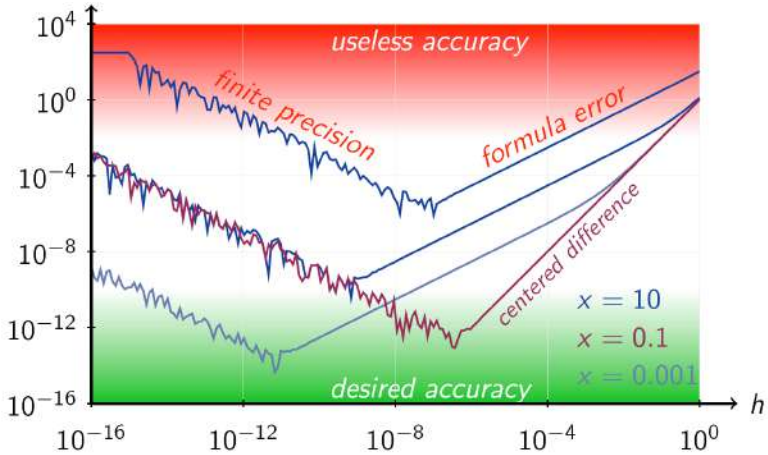




Plot credits: https://ruder.io/optimizing-gradient-descent/

Step downhill, repeat. Every parameter update in deep learning is this one line. So the real question: how do we get ∇f — for any program, fast?

Computing Derivatives



Manual

- Error prone

Numerical Differentiation (ND)

- Precision errors
- High computational complexity
- Higher order problem (formula approximated by missing higher order terms)

Symbolic Differentiation (SD)

- Closed form (no control flow), expression swell
- May require transcribing result back into code

$$\frac{d}{dx}\left(e^{e^{e^{e^{x}}}}\right) = e^{x+e^{e^{e^{x}}}+e^{e^{e^{x}}}+e^{e^{x}}+e^{x}}$$

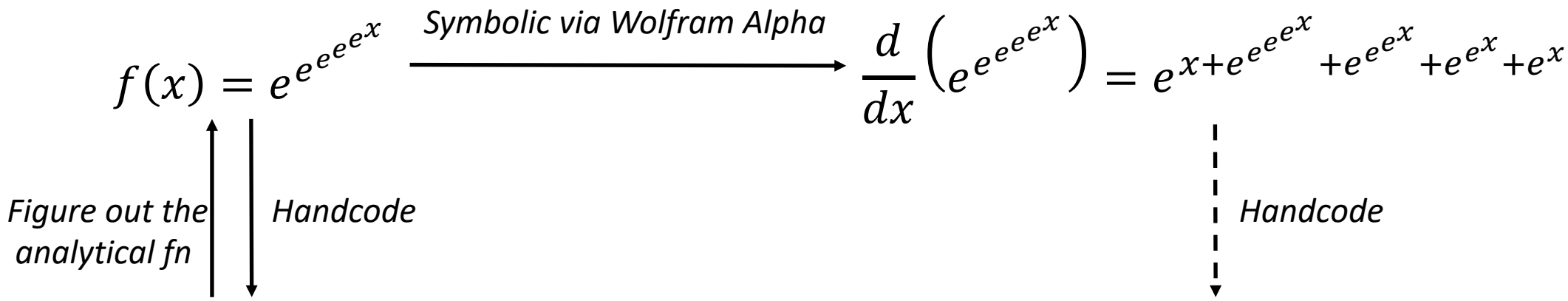
Algorithmic or Automatic Differentiation (AD)

- Automatically generate a program to compute the derivative of a given function

LLMs

- They approximate but could be useful in future…

Automatic vs Symbolic Differentiation

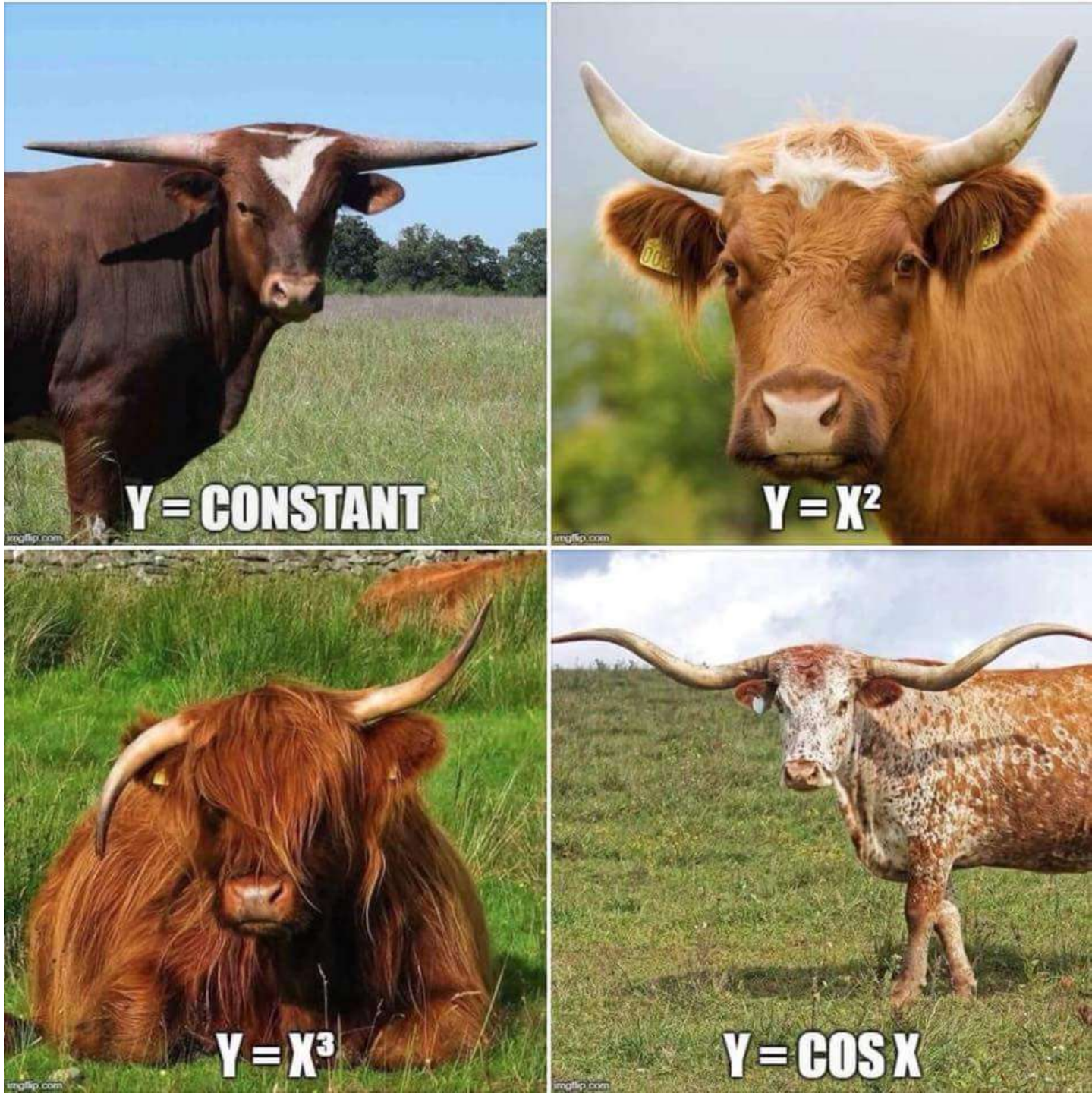



```
// f(x)=e^(e^(e^(e^(e^x))))
#include <cmath>
double f (double x, int N=5) {
  double result = x;
  for (unsigned i = 0; i < N; i++)
    result = std::exp(result);
  return result;
}
```

AD

```
double f_dx(double x, int N=5) {
  double result = x;
  double d_result = 1;
  for (unsigned i = 0; i < N; i++) {
     result = std::exp(result);
     d_result *= result;
  }
  return d_result;
}
```

AD Basics

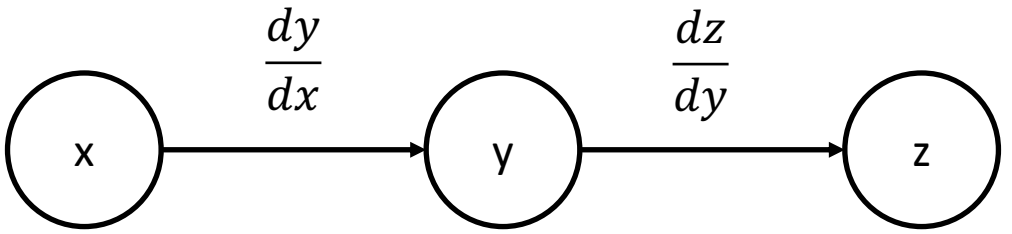


Chain Rule

```
y = f(x)
z = g(y)
```

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

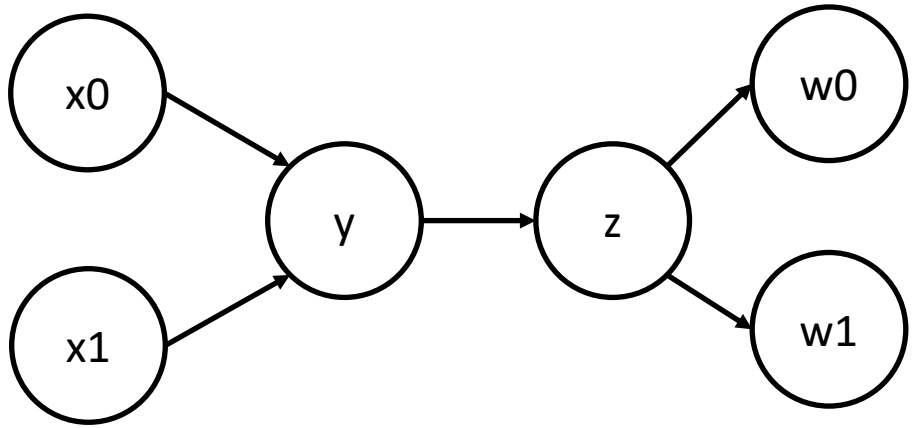
"if a car travels twice as fast as a bicycle and the bicycle is four times as fast as a walking man, then the car travels 2 × 4 = 8 times as fast as the man." G. Simmons

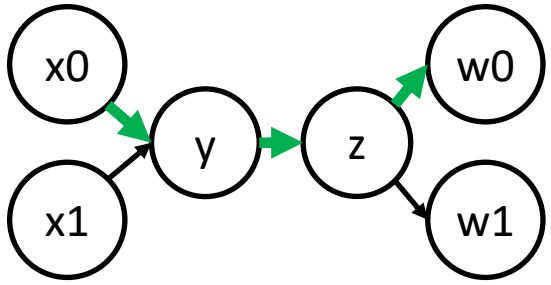


In the computational graph each node is a variable and each edge is a derivative between adjacent nodes. We recurse until every edge is an elementary rule.

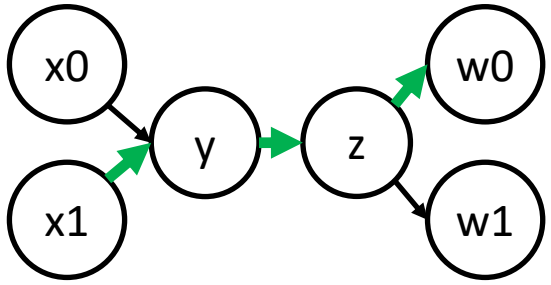
Algorithm Decomposition

```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)
```

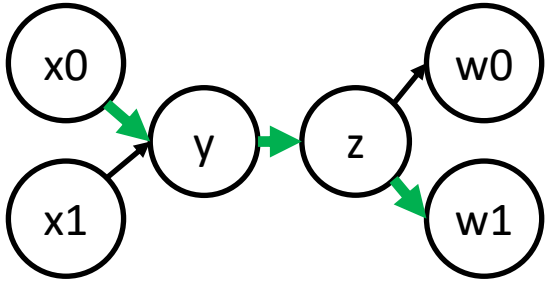




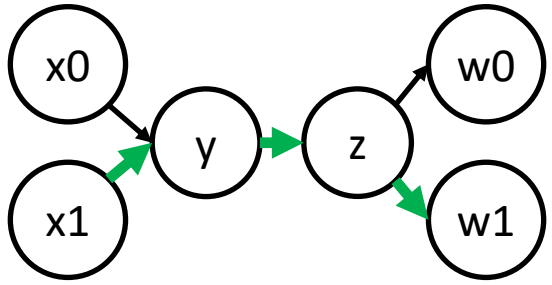
$$\frac{\partial w0}{\partial x0} = \frac{\partial w0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$



$$\frac{\partial w0}{\partial x1} = \frac{\partial w0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x1}$$



$$\frac{\partial w1}{\partial x0} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$



$$\frac{\partial w1}{\partial x1} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x1}$$

Forward Mode Step-by-Step

```
dx0dx = {1, 0}
dx1dx = {0, 1}

y = f(x0, x1)

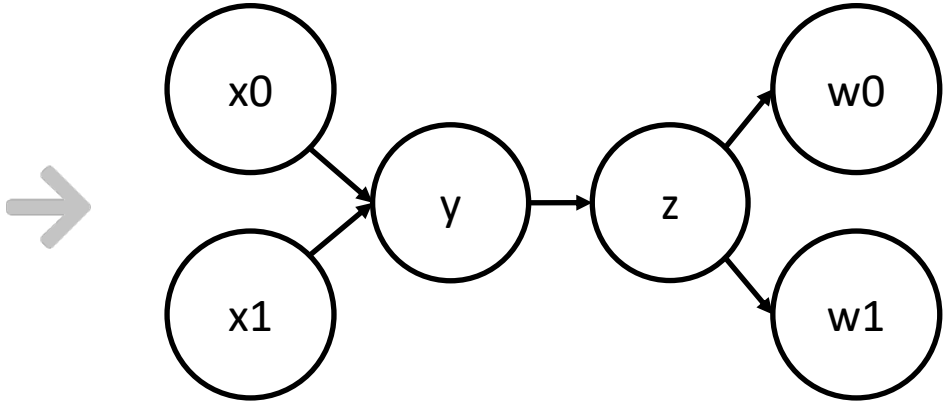
dydx = df(x0, dx0dx, x1, dx1dx)

z = g(y)

dzdx = dg(y, dydx)

w0, w1 = l(z)

dw0dx, dw1dx = dl(z, dzdx)
```



Reverse Mode Step-by-Step

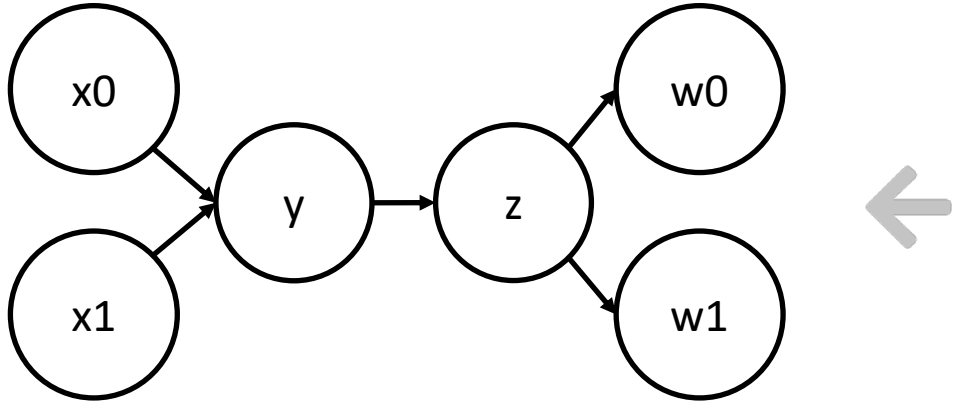
```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}

dwdz = dl(dwdw0, dwdw1)

dwdy = dg(y, dwdz)

dwdx0, dwdx1 = df(x0, x1, dwdy)
```



Vector Modes

Those {1,0} and {0,1} seeds were already a hint: widen the derivative from a scalar to a length-K bundle, and one pass carries K directional derivatives at once.

Vector forward — batched tangents

```
// seed the full identity (width = K)
dx = {{1,0,…},
      {0,1,…}, …}

// one primal, K tangents pushed through
y, dy = f(x, dx)   // dy is a bundle
```

One sweep → several columns of the Jacobian. The primal r = exp(r) is computed once; K tangents ride along it.

Vector reverse — batched adjoints

```
// seed K output adjoints at once
dw = {{1,0,…},
      {0,1,…}, …}

// one reverse sweep pulls them back
dx = bwd(…, dw)    // K gradient rows
```

One sweep → several rows of the Jacobian (a batched VJP) — the dual of vector forward.

The K directions share one primal pass and advance together on SIMD lanes

Cheap Gradient Principle

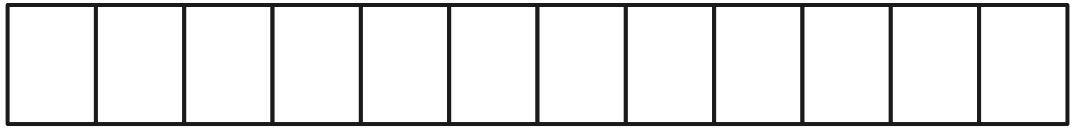
- The computational graph has **common subpaths** which can be precomputed
- If a function has a **single input**, no matter how many outputs, **forward mode** AD generates a derivative with the **same time complexity** as the original function.
- More importantly, if a function has a **single output**, no matter how many inputs, **reverse mode** AD generates the derivative with the **same time complexity** as the original function.

Forward mode → tall Jacobian ($n_{out} \gg n_{in}$); reverse mode → wide Jacobian ($n_{in} \gg n_{out}$). The shape decides.

Why ML Needs This

Training minimizes one scalar loss over millions of parameters: $L : \mathbb{R}^n \to \mathbb{R}$, $n \approx 10^6$–10^{11}.
The Jacobian is a single row — 1 × n — as wide as it gets.

$\partial L / \partial \theta$:



1 output × n inputs

Forward mode — one column per pass



… → n separate passes (O(n))

Reverse mode — the whole row in one pass



… → **1 pass (O(1) in n) — this is backprop**

The cost that makes it possible

$$\text{cost}(\nabla\theta\ L) \approx c \cdot \text{cost}(L)$$

c ≈ 2–4, independent of n

The full gradient over millions of parameters costs about one forward evaluation.

Implementation Approaches

Implementations sit on one axis: how much of the differentiation happens at compile time. Two ends — tracing and source-to-source — with most real systems in between.

- **Tracing / operator overloading / taping** — records the linear sequence of operations at runtime into a tape. Trivially handles loops, recursion, branches; but interprets a tape at runtime (can be ~100× slower). Adept, Autograd, PyTorch eager.
- **Source transformation** — a compiler reads the code and constructs a derivative program at compile time. Optimizable, no runtime tape, reusable; needs the whole program + compiler machinery. Tapenade, Clad, Enzyme.

Both are staged compilation. torch.compile slides tracing toward source-to-source; Tapenade has its own program representation to differentiate; Clad differentiates the Clang AST and rides the compiler's own optimizer — no user rewrite; Enzyme works on LLVM IR.

Becoming an AD expert in baby steps: Dual Numbers

Carry the derivative inside the number itself: adjoin ε with $\varepsilon^2 = 0$, and write every value as a + εb (b is its derivative). The algebra then does forward mode for you.

$\varepsilon^2 = 0$

$(a+\varepsilon b) + (c+\varepsilon d) = (a+c) + \varepsilon(b+d)$

$(a+\varepsilon b)(c+\varepsilon d) = ac + \varepsilon(bc+ad)$

the ε term is exactly the product rule

```
struct dfloat { double v, dv; };

dfloat operator*(dfloat x,
                 dfloat y){
  return { x.v*y.v,
           x.dv*y.v + x.v*y.dv };
}
```

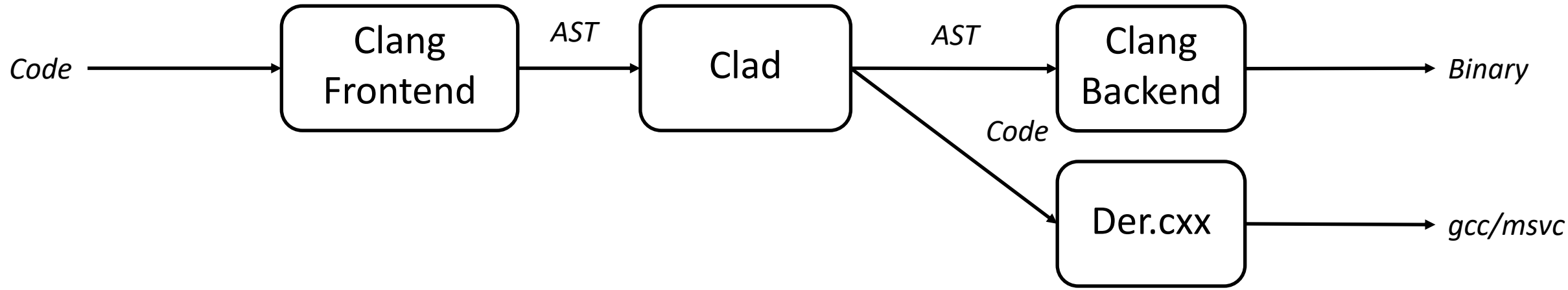
Seed an input with ε (dv = 1), run the program unchanged — the answer comes back carrying its own derivative. Operator overloading falls right out.

Compiler-Based AD with Clad

```
double fsq(double x) {
  return x * x;
}
```

```
FunctionDecl f 'double (double)'
|-ParmVarDecl x 'double'
`-CompoundStmt
  `-ReturnStmt
    `-BinaryOperator 'double' '*'
      |-ImplicitCastExpr 'double' <LValueToRValue>
      | `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
      `-ImplicitCastExpr 'double' <LValueToRValue>
        `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
```

```
FunctionDecl 0x7f7f801dbff8 <<invalid sloc>> <invalid sloc> f_darg0 'double (double)'
|-ParmVarDecl 0x7f7f801dc090 <<invalid sloc>> <invalid sloc> used x 'double'
`-CompoundStmt 0x7f7f801dc3d0 <<invalid sloc>>
  |-DeclStmt 0x7f7f801dc190 <<invalid sloc>>
  | `-VarDecl 0x7f7f801dc118 <<invalid sloc>> <invalid sloc> used _d_x 'double' cinit
  |   `-ImplicitCastExpr 0x7f7f801dc178 <<invalid sloc>> 'double' <IntegralToFloating>
  |     `-IntegerLiteral 0x7f7f801dc0f8 <<invalid sloc>> 'int' 1
  `-ReturnStmt 0x7f7f801dc398 <<invalid sloc>>
    `-BinaryOperator 0x7f7f801dc318 <<invalid sloc>> 'double' '+'
      |-BinaryOperator 0x7f7f801dc298 <<invalid sloc>, T.cpp:3:32> 'double' '*'
      | |-ImplicitCastExpr 0x7f7f801dc268 <<invalid sloc>> 'double' <LValueToRValue>
      | | `-DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
      | `-ImplicitCastExpr 0x7f7f801dc280 <col:32> 'double' <LValueToRValue>
      |   `-DeclRefExpr 0x7f7f801dc208 <col:32> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
      `-BinaryOperator 0x7f7f801dc2f0 <col:30, <invalid sloc>> 'double' '*'
        |-ImplicitCastExpr 0x7f7f801dc2c0 <col:30> 'double' <LValueToRValue>
        | `-DeclRefExpr 0x7f7f801dc1d0 <col:30> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
        `-ImplicitCastExpr 0x7f7f801dc2d8 <<invalid sloc>> 'double' <LValueToRValue>
          `-DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
```



Scales to hundreds of thousands of generated C++ fitting code in HEP: See Rembser et al. AD in RooFit, Runting et al. Latest Developments in CMS Combine

```
double f_darg0(double x) {
  double _d_x = 1;
  return _d_x * x + x * _d_x;
}
```

Clad. Usage

```
// clang -fplugin=/.../libclad.so
// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }

double pow2_darg0(double); // to be filled by clad

int main() {
 auto dfdx = clad::differentiate(pow2, 0);
 // Function execution can happen in 3 ways:
 // 1) Using CladFunction::execute method.
 double res = dfdx.execute(1);

 // 2) Using the function pointer.
 auto dfdxFnPtr = dfdx.getFunctionPtr();
 res = dfdxFnPtr(2);

 // 3) Using direct function access through fwd declaration.
 printf("res is: %f\n",pow2_darg0(3));
 printf("The derivative code is: %s\n", dfdx.getCode());
 return res;
}
```

COMPILER EXPLORER

https://godbolt.org/z/h54nxjWc6

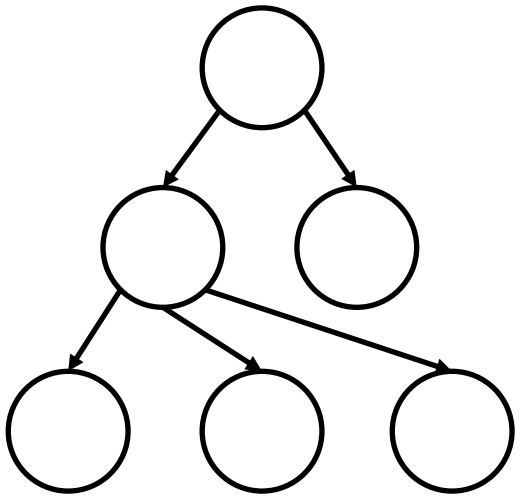
Tells the plugin to create pow2_darg0.

The programmer can use the derivative via a wrapper object, function pointer or forward declaration.

```
performance-test@vv-nuc ~/clad-build-llvm11 $ ./a.out
6.000000
The derivative code is: double pow2_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```

Clad as RooFit's AD Engine

RooFit Compute Graph



CodeGen/Flatten

Standalone Simplified Compute Graph C++

```
...
double gauss(double *x) {
   using namespace RooFit::Detail;

   return gEvaluate(x[3], (x[0] + x[1]),
(x[2] * 1.5)) /
      gIntegral(-10., 10., (x[0] +
x[1]), (x[2] * 1.5));
}
...
```

AD → Δ

Optimize

FCN

```
pdf.fitTo(data, RooFit::EvalBackend("codegen"))
pdf.createNLL(data, RooFit::EvalBackend("codegen"))
```

Most of HistFactory RooFit primitives are supported. Please reach out if you need additional primitive support

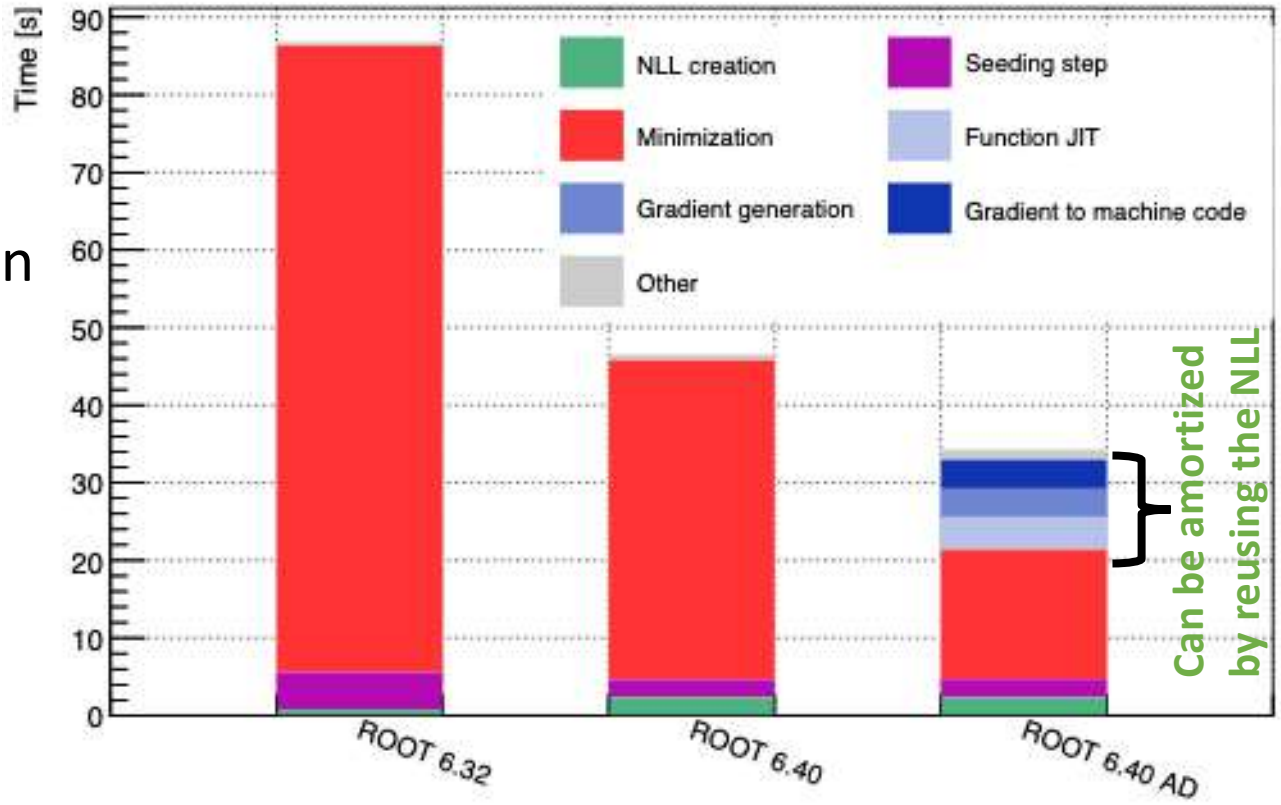
Statistical Modelling in CMS

- CMS Combine is the flagship tool for statistical modelling in CMS. It is based on RooFit but has many customizations.
- The workflows run for days once the statistical model is constructed
- Most workflows are dominated by the gradient part of the minimization step
- Clað is a compiler-based source transformation automatic differentiation tool integrated in RooFit. It is capable of generating cheap gradients whose asymptotic computational time complexity is independent on the size of the inputs

CMS Higgs Combination Benchmark

CMS published its Higgs likelihood observation model [Higgs observation likelihood]

- Very heterogeneous likelihood:
  672 parameters in **102 channels** with
  - Template histogram fits
  - Analytical shape fits, numerical integration necessary in some cases
- **Perfect example** to test the new Combine developments



CMS Higgs Observation Models. Numerical Stability

In this model we observed that the derivatives are small compared to the NLL value

- Numerical differentiation often fails because the finite differences are smaller than numerical precision on the NLL
- Essential workaround for the Higgs model is to offset the NLL by initial value with:
  `pdf.createNLL(data, RooFit::Offset(true))`

Problems with this:

- Offsetting might fail if initial value is far from the minimum
- Bookkeeping of offsets is error-prone

With AD, the offsetting is not necessary anymore!

```
36 - FCN = -9801946.549 Edm = 0.01129396511
37 - FCN = -9801946.566 Edm = 0.01497173883
38 - FCN = -9801946.574 Edm = 0.007242353199
39 - FCN = -9801946.583 Edm = 0.004954953322
40 - FCN = -9801946.589 Edm = 0.005774308843
41 - FCN = -9801946.596 Edm = 0.004695329674
42 - FCN = -9801946.602 Edm = 0.004558156748
43 - FCN = -9801946.615 Edm = 0.008141300763
44 - FCN = -9801946.625 Edm = 0.004861879849
45 - FCN = -9801946.628 Edm = 0.003472778648
46 - FCN =  -9801946.63 Edm = 0.001782083931
47 - FCN = -9801946.631 Edm = 0.0007515760698
```

Minimizer output, showing the small changes wrt. large NLL value

bine Compute Graph

No need to recompile RooFit

Visit each node

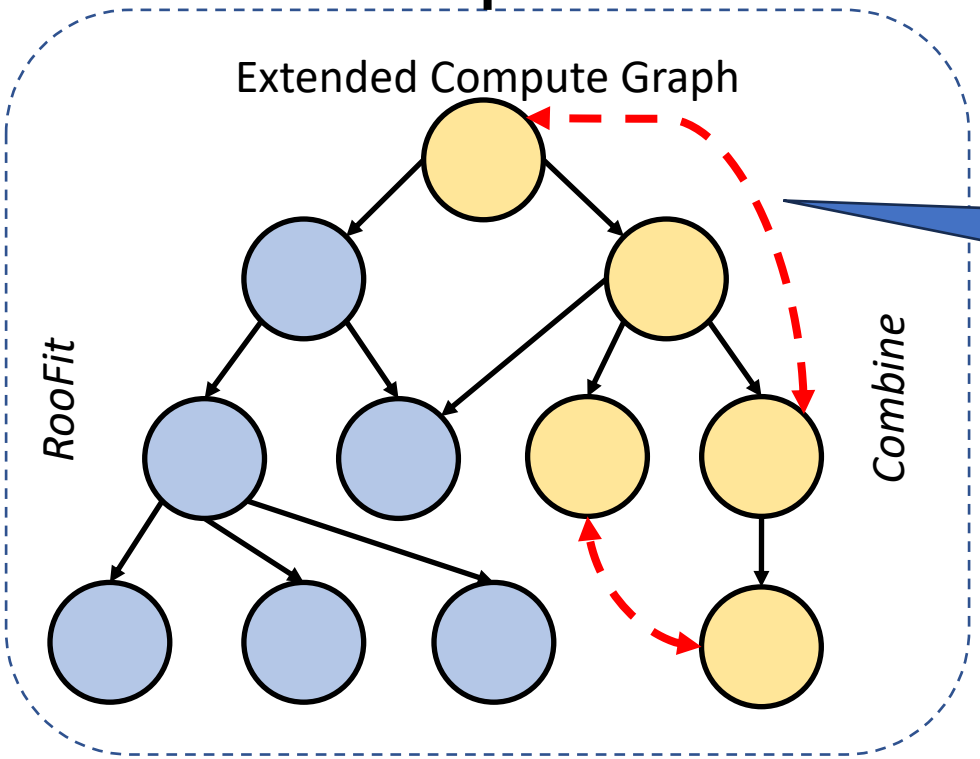
```
void codegenImpl(RooAddPdf&, CodegenContext&);
void codegenImpl(RooChebychev&, CodegenContext&);
...
void codegenImpl(ProcessNormalization&, CodegenContext&);
void codegenImpl(FastVerticalInterpHistPdf2&, CodegenContext&);
```

AD

Δ

Optimize

FCN



More cleanup is needed to avoid layering violations

Annotated Combine Compute Graph

Semantic Meaning

RooAbsArg Name

```
// ProcessNormalization::n_exp_bindijet_proc_qqH[ thetaList=(pdf_qqbar) asymmThetaList=()
otherFactorList=(r_qqH) ] = 0.95
const double t20 = RooFit::Detail::MathFuncs::processNormalization(
      0.950000, 1, 0, 1, t19, xlArr + 6, nullptr, xlArr + 6, xlArr + 6, t18);

// RooAddition::n_exp_bindijet[ n_exp_bindijet_proc_ggH + n_exp_bindijet_proc_qqH +
n_exp_bindijet_proc_bkg ] = 4.55
  const double t21 = (t17 + t20 + params[4]);


// RooNLLVar[ pdf=model_s weightVar=_weight _weight_sumW2=_weight_sumW2 ] = 0
for (int loopIdx1 = 0; loopIdx1 < 1; loopIdx1++) {
  nll_result += RooFit::Detail::MathFuncs::nll(t25, obs[3], 0, 0);
}
```

zero because of offsetting

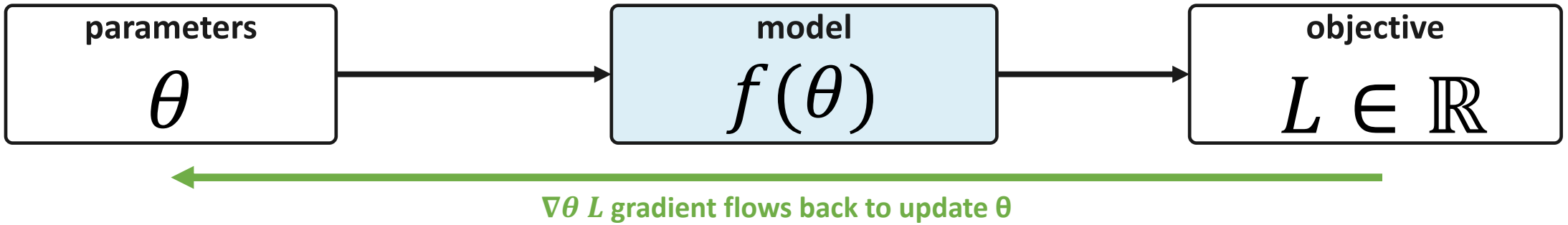
Crosscheck with RooFit evaluate

Notebook time

/image/root-demo-notebooks/blob/main/ZPeakFitCladVsRooFit.ipynb

The Model Bends to the Machinery

Every gradient-based system has the same setup: parameters → model → objective, with derivatives flowing back. The quiet cost: to keep the model differentiable through a fixed toolbox, we approximate it.



Deep learning bends the model to fit backprop

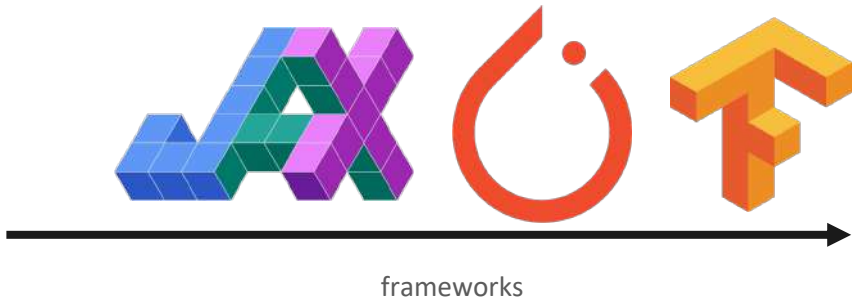
- ReLU, softmax, soft attention, smooth surrogate losses, continuous relaxations of discrete choices — concessions that keep f(θ) inside a differentiable toolbox.
- **Differentiable programming inverts the arrow**: make the machinery work on the faithful model — the real simulator, renderer, or physics — instead of approximating it down to the framework's vocabulary.

Tools Are Why DL Won

AlexNet (2012) took a researcher who could do all of this by hand. Frameworks collapsed that expertise into one call — and that, more than any single architecture, is why deep learning took off.

BEFORE — four kinds of expertise, by hand

- Optimization
- Program differentiation
- Numerical computing
- GPU hacking



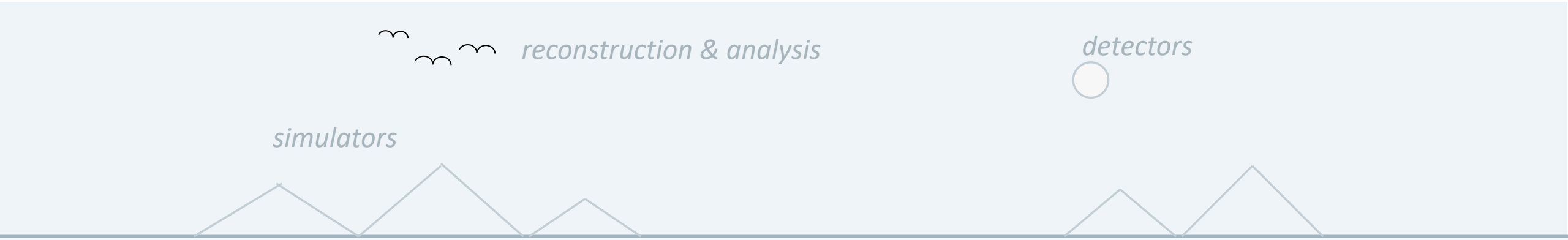
AFTER — one call

```
loss.backward()
optimizer.step()
```

The hard parts didn't vanish — the framework absorbed them. Lowering the barrier is what created the field.

DL systems assume each layer has **high arithmetic intensity** and that memory traffic between layers is negligible. **Many low-intensity ops with heavy intermediate memory** — without operation fusion the gradient is memory-bound, and the dense branching is painful to express as tensor ops.

The Framework Became the Horizon



`loss.backward()`

`matmul` `conv` `ReLU` `softmax`

one call — fast, supported, differentiable

There is no simulator.backward()

SmallPt: Now Run It Backwards

Same 99 lines. Same integrator. The other direction is called *differential rendering*.

Forward — you just did this

`scene θ ⟶ smallpt ⟶ image`

"render it fast"
profile → hot loop → vectorize, -O2, OpenMP

samples are independent — parallelism works

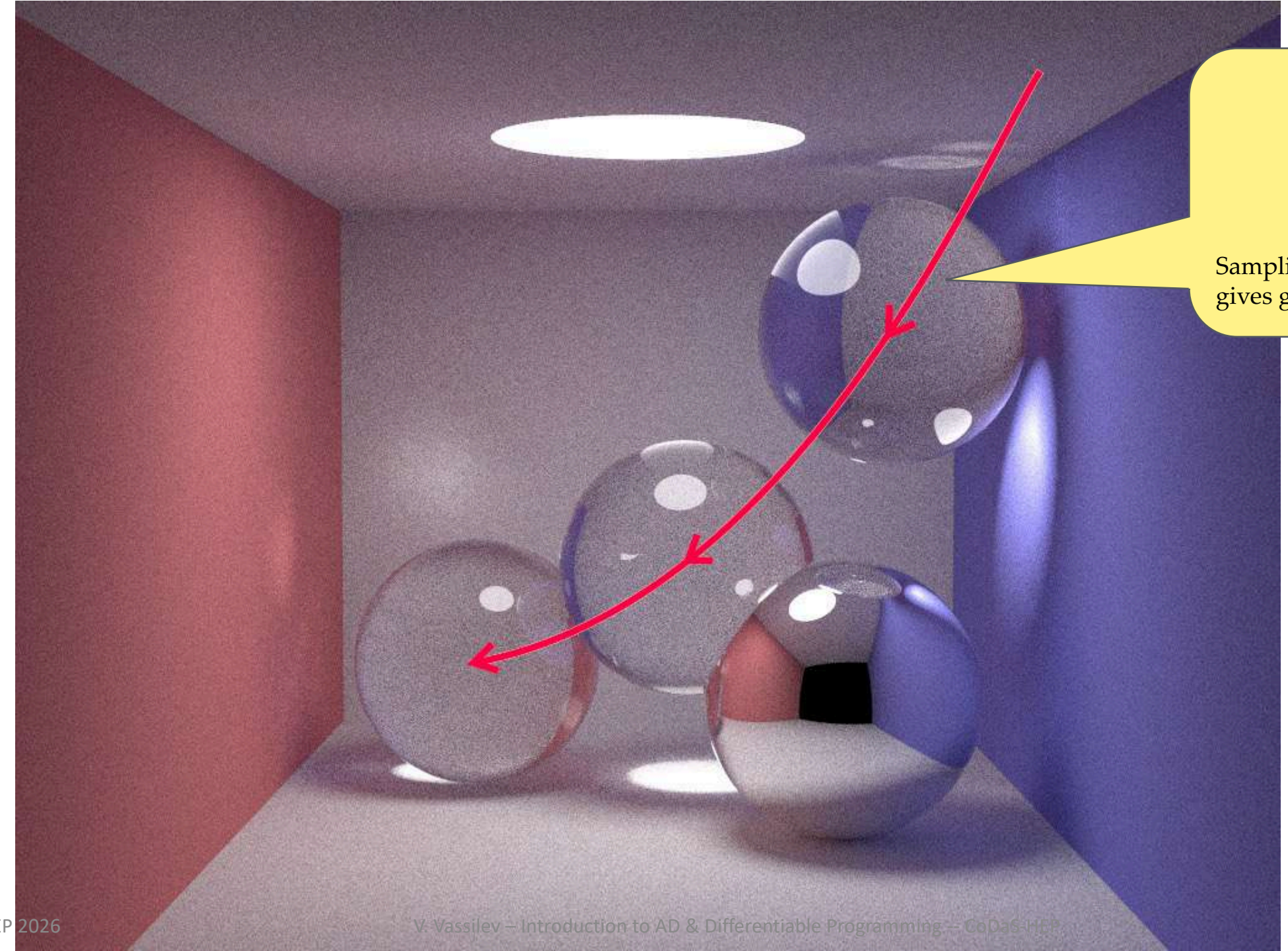
Inverse — now do this

`scene θ ⟵ ??? ⟵ image`

"given the image, recover the scene"
needs `∂image / ∂θ`

finite differences → N · cost(render). You have seen how that ends.
simulator.backward() → there isn't one.

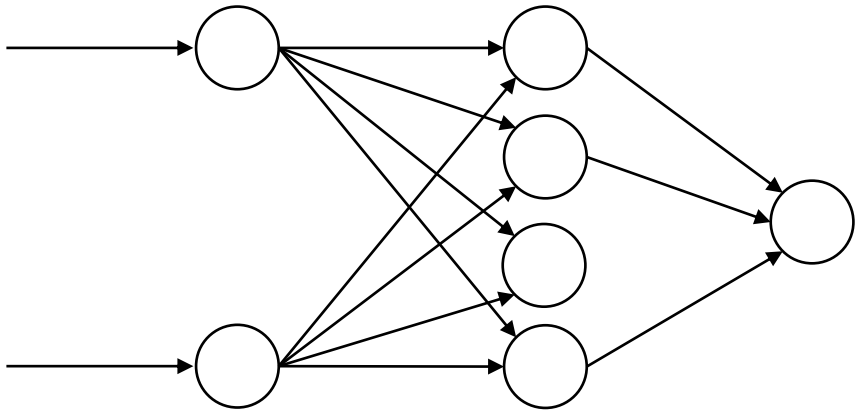
Third time, same wall: smallpt → RooFit → smallpt. The gradient was never the hard part of the physics — it was the hard part of the tooling.


Sampling at the boundary
gives gradients equal to 0

Smooth Layers vs Real Code

A neural net is a smooth stack of dense layers — differentiable by construction. Real scientific code is full of branches, loops, and discrete decisions — that is where AD actually gets hard.

Neural net — differentiable by construction



matmul · ReLU · matmul · softmax — every op has a clean derivative

In real code the devil is in the branches

```
double hit(Ray r, Mesh m){
  double t = INF;
  for (Tri f : m) {
    if (!intersect(r, f, t)) continue;
    if (t < eps || t > tmax) continue;
    t = nearer(t, hit_face(f));
  }
  return (t < INF) ? shade(t) : 0;
}
```

branches, early-outs, min/visibility — kinks and discontinuities AD must handle, not smooth away

The toolbox is easy precisely because it excludes this code. Differentiable programming is about differentiating the right panel — as written.

Faithful vs Approximated Models

Across science the real model is non-smooth, discrete, or simulation-defined. To fit backprop it gets replaced by a differentiable cartoon. Differentiable programming keeps the faithful one:

Domain	Approximated fit to DL	Faithful (DP)
Rendering	soft / blurred rasterizer	real renderer with visibility edges
Physics & contact	globally smoothed dynamics	contact-rich simulator, as written
Discrete choices	softmax / soft attention	real argmax, sorting, branching
Known physics	learned black-box surrogate	the actual conservation laws / PDE

Science Needs an Equivalent to DL Leap

There is no loss.backward() for a faithful simulator. Today the scientist must hand-derive its adjoint — or surrender to a differentiable cartoon. Compiler-based AD on the real code is the missing tooling leap.

Differentiable programming does not repeal non-smoothness: A renderer's visibility term is still discontinuous; a hard argmax still has no useful gradient. In this setup, the approximation becomes a deliberate, local choice — edge sampling, subgradients, smoothing where you choose — not a mandatory, global one.

Open problems — and where they're studied:

- Correctness at discontinuities, fixed-point loops, lookup tables, chaotic & time-averaged systems — AD can return confidently wrong derivatives. See Hückelheim et al., **Understanding Automatic Differentiation Pitfalls** (2023).
- Smoothing, relaxation, stochastic estimators, differentiable rendering & sorting — active research, surveyed in Tzu-Mao Li's **CSE 291: Differentiable Programming** (UCSD).

The goal is not to pretend these are solved but to make the faithful model the default, and the approximation an honest, local decision.

The Lever We Need to Pull Harder

The fix is not better models inside the box — it is advancing AD itself, so the gradient machinery runs on the faithful model. That frontier is exactly what the framework era taught us to ignore.

Some work happening in our own field:

- **Differentiable detectors** — MODE / TomOpt: end-to-end gradient design of real instruments (Aehle et al. 2023; Strong et al. 2024).
- **Differentiable analysis** — neos / INFERNO / relaxed: optimise summary statistics through the profile likelihood (Simpson & Heinrich 2022).
- **AD in the toolchain** — Automatic Differentiation in ROOT (Vassilev, Efremov, Shadura); Clad brings it to C++.

The win isn’t “AD for more domains” but letting the model keep its real structure and still talk to gradient descent.

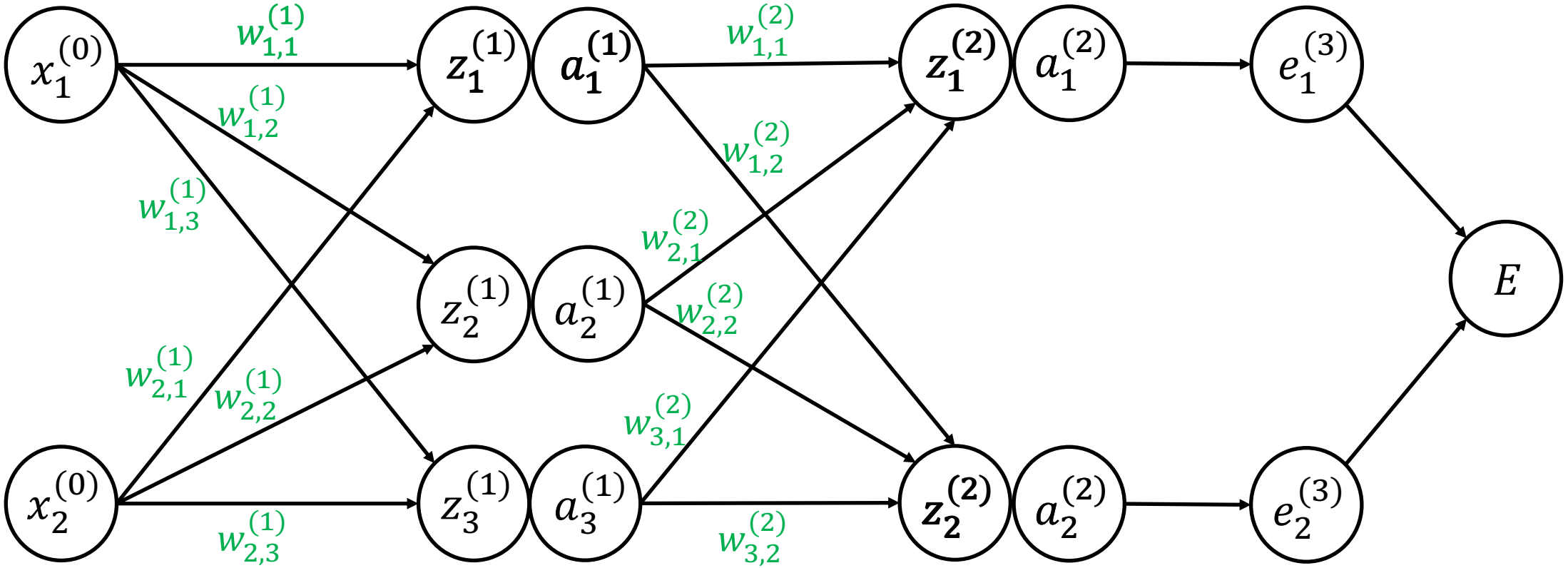
Takeaways

- **AD is exact and automatic** — not numerical approximation, not symbolic blow-up; derivatives to machine precision for arbitrary code, including loops and branches.
- **Two modes, chosen by shape** — forward for few-inputs/many-outputs, reverse for many-inputs/few-outputs. Reverse mode is backpropagation.
- **The cheap-gradient principle** — a full gradient over millions of parameters costs about one function evaluation. This is what makes modern ML — and differentiable science — feasible.
- **The tooling gap** — differentiable science needs the barrier-lowering leap DL already had. Compiler-based AD is that path for real code.

Further reading: Griewank & Walther, Evaluating Derivatives · Blondel & Roulet, The Elements of Differentiable Programming (2024)

”Attention is NOT all you need”. Thank you!

$$\frac{\partial}{\partial} = \left(\frac{\partial}{\partial}\frac{\partial}{\partial}\frac{\partial}{\partial} + \frac{\partial}{\partial}\frac{\partial}{\partial}\frac{\partial}{\partial}\right)\frac{\partial}{\partial}\frac{\partial}{\partial}$$



Backup