Differentiable Programming

Sean Gasiorowski (SLAC)
sgaz@slac.stanford.edu

August 14th, 2024

Machine Learning for Fundamental Physics School 2024
Lawrence Berkeley National Laboratory

SLAC NATIONAL ACCELERATOR LABORATORY

Stanford University | U.S. DEPARTMENT OF ENERGY

Outline:

Talk:

- What is differentiable programming/why do we care?
- Basics of automatic differentiation
- Tips and tricks

Tutorial:

- Fitting parameters with differentiable programming
- How to deal with hard edges
- Differentiable pipelines: simulators and neural networks

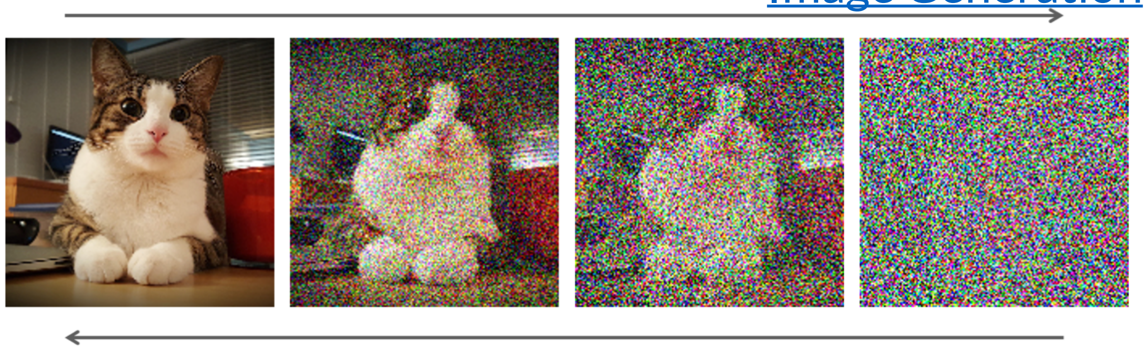
What is Differentiable Programming?

Machine Learning

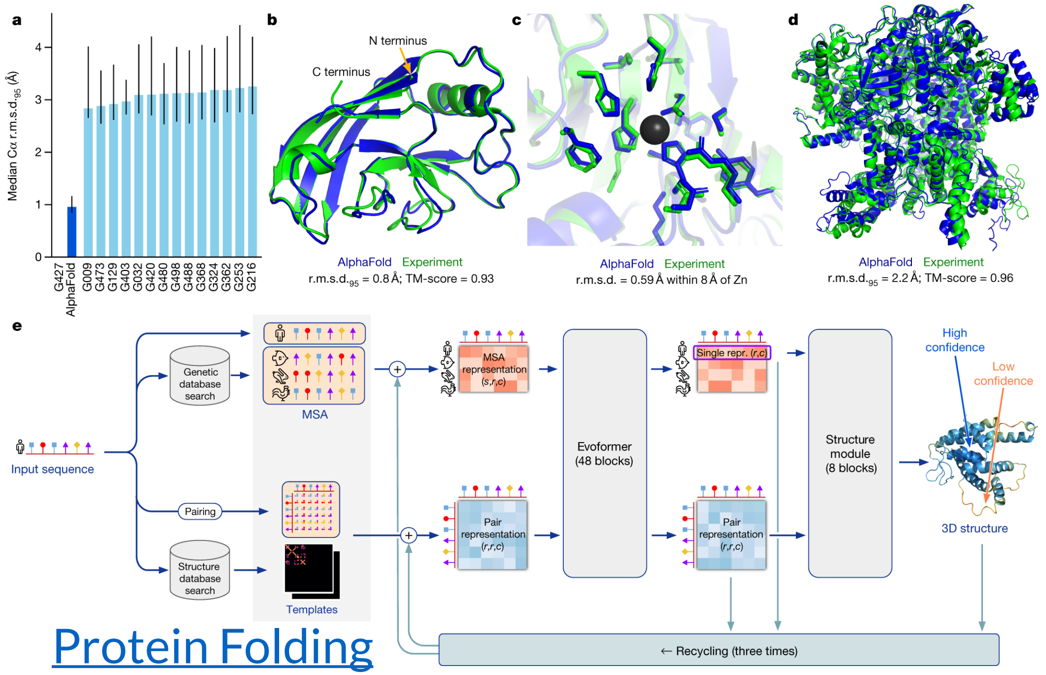
Neural networks are the backbone of modern machine learning

Large Language Models

Image Generation



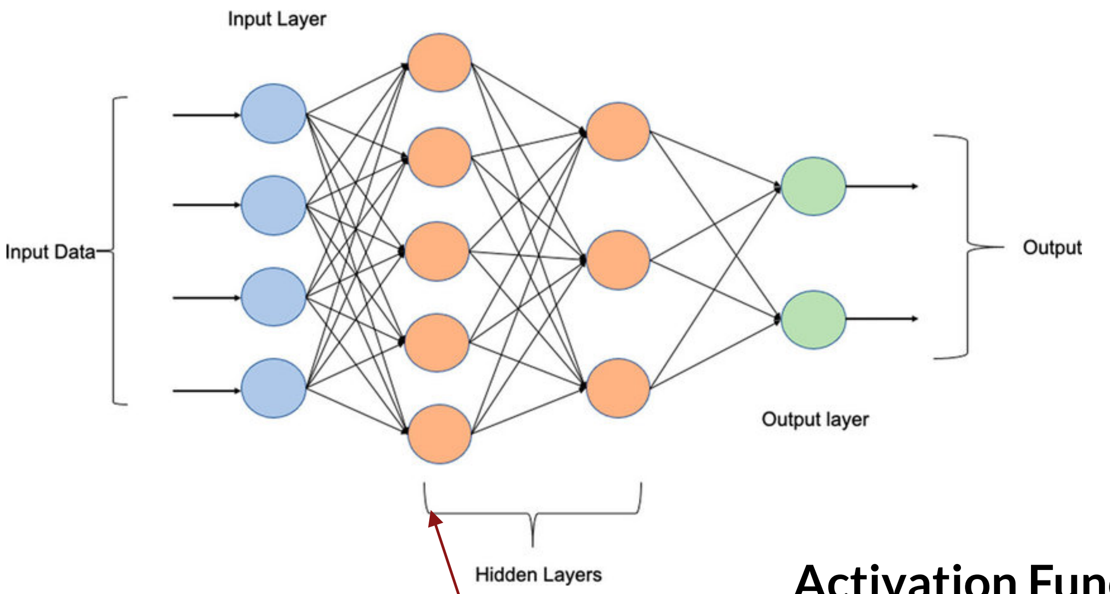
Semantic Segmentation



Protein Folding

How do machines learn?

When we train a neural network, what's happening?



Activation Function

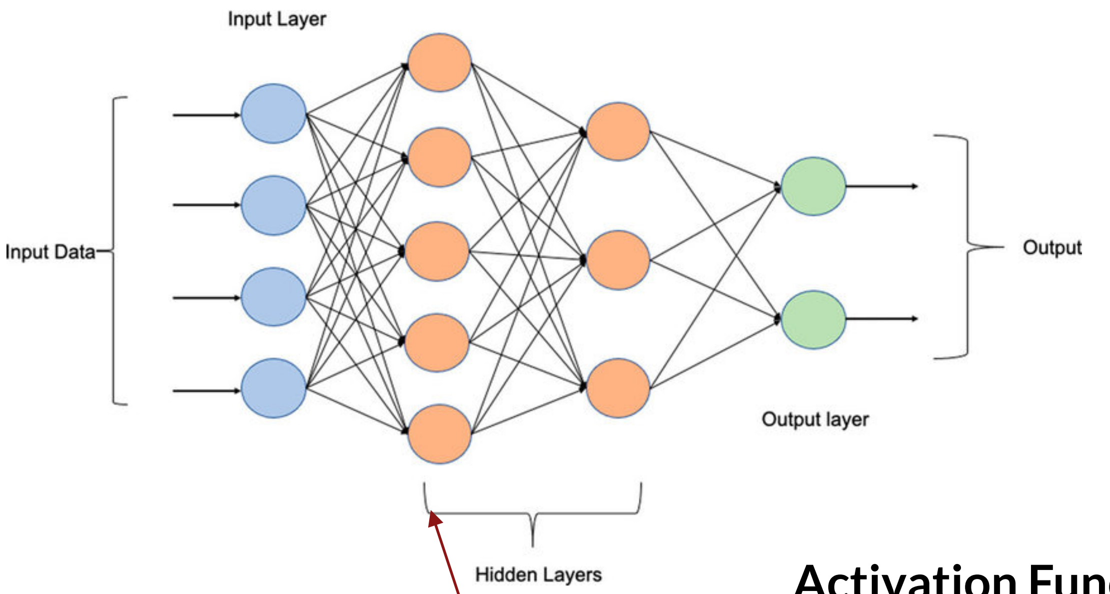
$$h^{(i)}(\mathbf{x}) = \phi^{(i)}(\mathbf{w}^T\mathbf{x} + b)$$

Weights

Biases

How do machines learn?

When we train a neural network, what's happening?



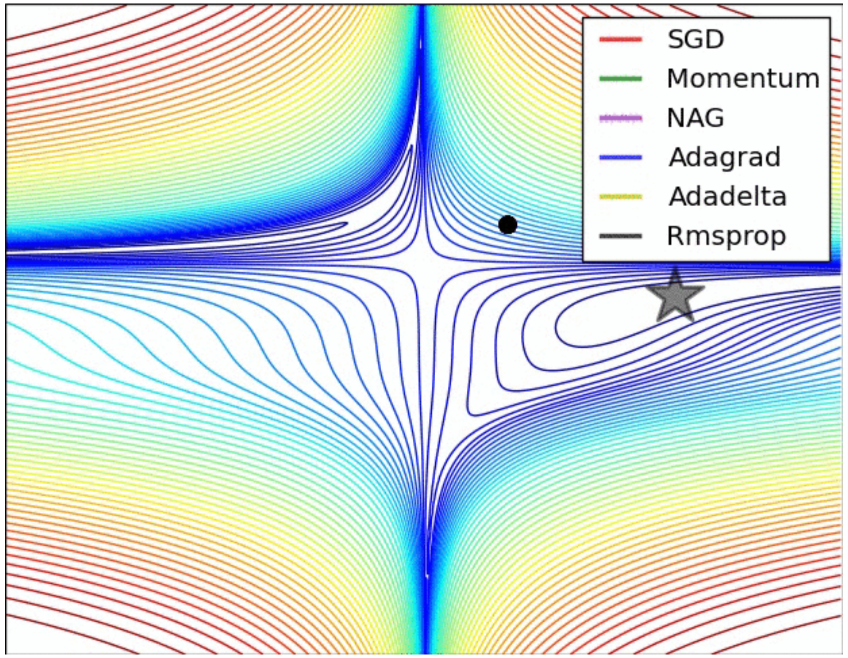
Activation Function

$$h^{(i)}(\mathbf{x}) = \phi^{(i)}(\mathbf{w}^T\mathbf{x} + b)$$

Weights

Biases

Ruder, link



NN weights and biases are adjusted to **minimize a loss function** using an **optimizer**

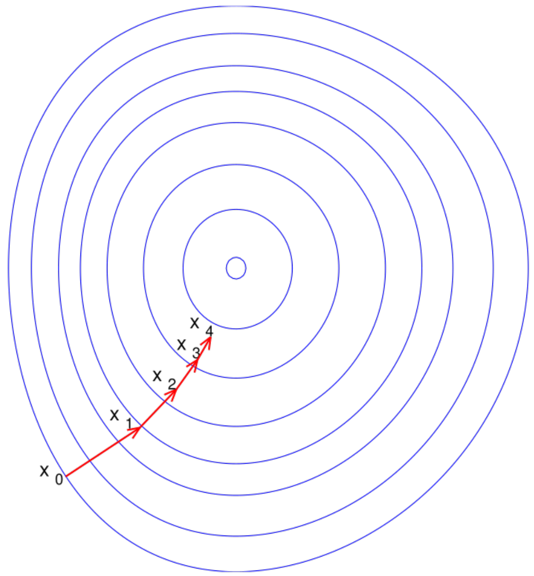
Breaking down an optimizer

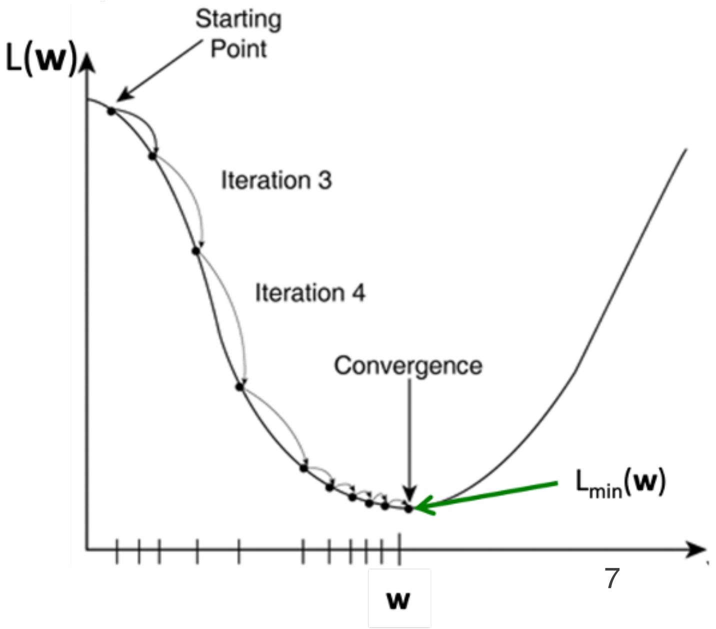
E.g. supervised learning:

- **Data** with labels: $\{(x_i, y_i)\}_{i=1}^N$
- **Model:** $h(x_i; \mathbf{w})$ (parameters $\mathbf{w}$)
- Element-wise **loss** (e.g. squared error, cross-entropy):
  $\mathscr{L}_i(\mathbf{w}) \equiv \mathscr{L}(y_i, h(x_i; \mathbf{w}))$

Gradient descent: Minimize total loss $\mathscr{L}(\mathbf{w}) = \dfrac{1}{N}\sum_{i=1}^{N} \mathscr{L}_i(\mathbf{w})$. At iteration t:

- Compute gradient $\nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}^{(t)})$
- Update model weights as: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \cdot \nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}^{(t)})$, where η is a learning rate controlling the size of the gradient step.
- Negative gradient gives (local) direction of steepest descent





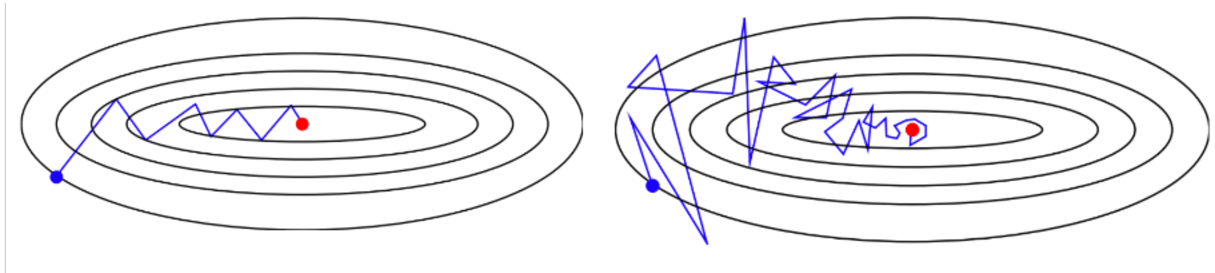
Breaking down an optimizer

Gradient descent is the foundation of most common optimizers

- **In practice:** stochastic/mini-batch gradient descent is used
  - Cost of full gradient descent scales with the number of samples:

$$\nabla_{\mathbf{w}} \mathscr{L}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\mathbf{w}} \mathscr{L}_i(\mathbf{w})$$

  - Instead, compute each update over a randomly sampled data point/batch of points
    - Unbiased estimator of full gradient: on average moves in the right direction
- **Benefits:** less costly to compute/faster, randomness may help break out of local minima
- Common extensions: momentum, Adam, RMSProp, …

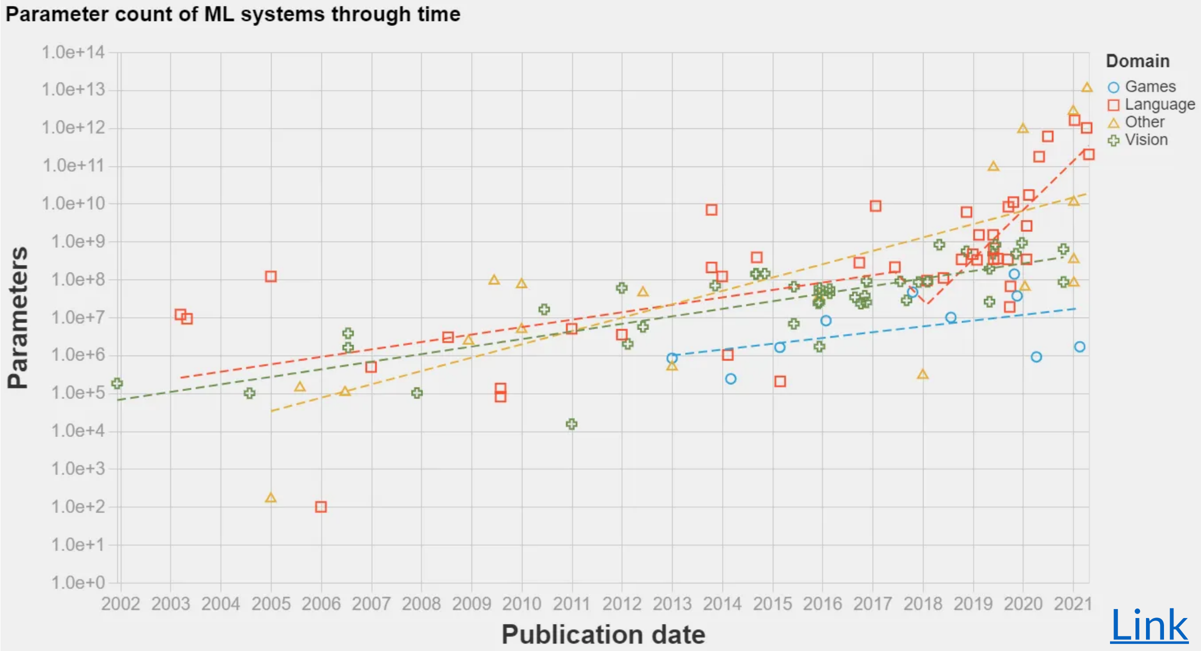


Batch gradient descent *Stochastic gradient descent*

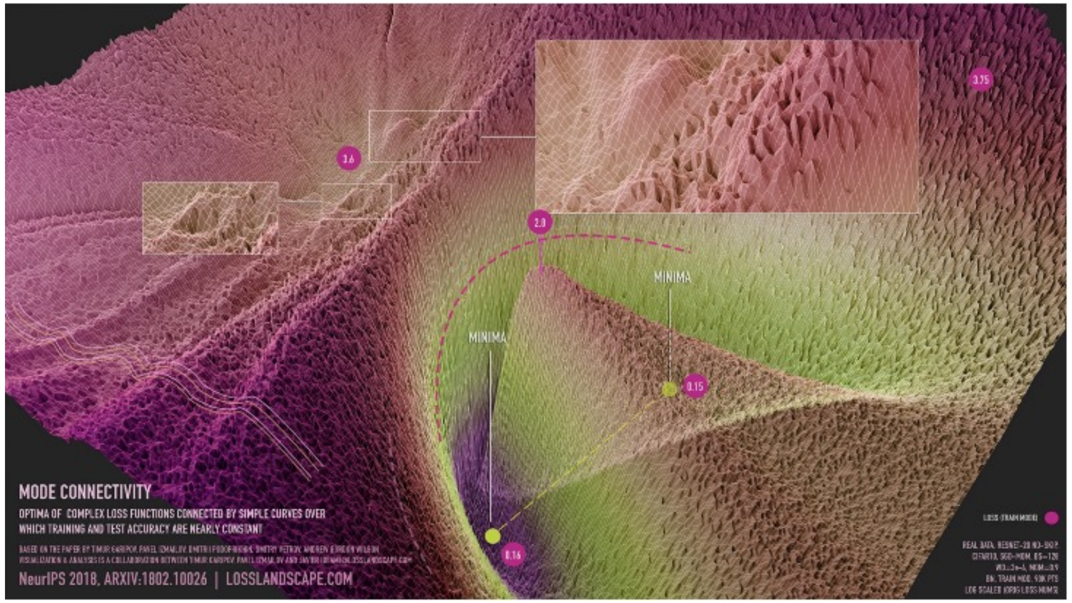
Why gradients?

Gradient-based optimizers have been used to train models with (at least) O(10^{11}) parameters

- => works well for high dimensional optimization
- Batch methods/SGD => scalable with dataset size
- Gradients are **easy to compute**



Link



https://arxiv.org/abs/1802.10026

B Details of Model Training

To train all versions of GPT-3, we use Adam with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$

https://arxiv.org/abs/2005.14165

SLAC

How to Compute Gradients

Popularity of gradient-based methods => good toolkits for computing gradients!

- Fundamental component of common ML libraries
- All use a common technique: **automatic differentiation**
  - a.k.a. **backpropagation** (for neural networks), autodiff, autograd, AD

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts

Nature **323**, 533-536 (1986)

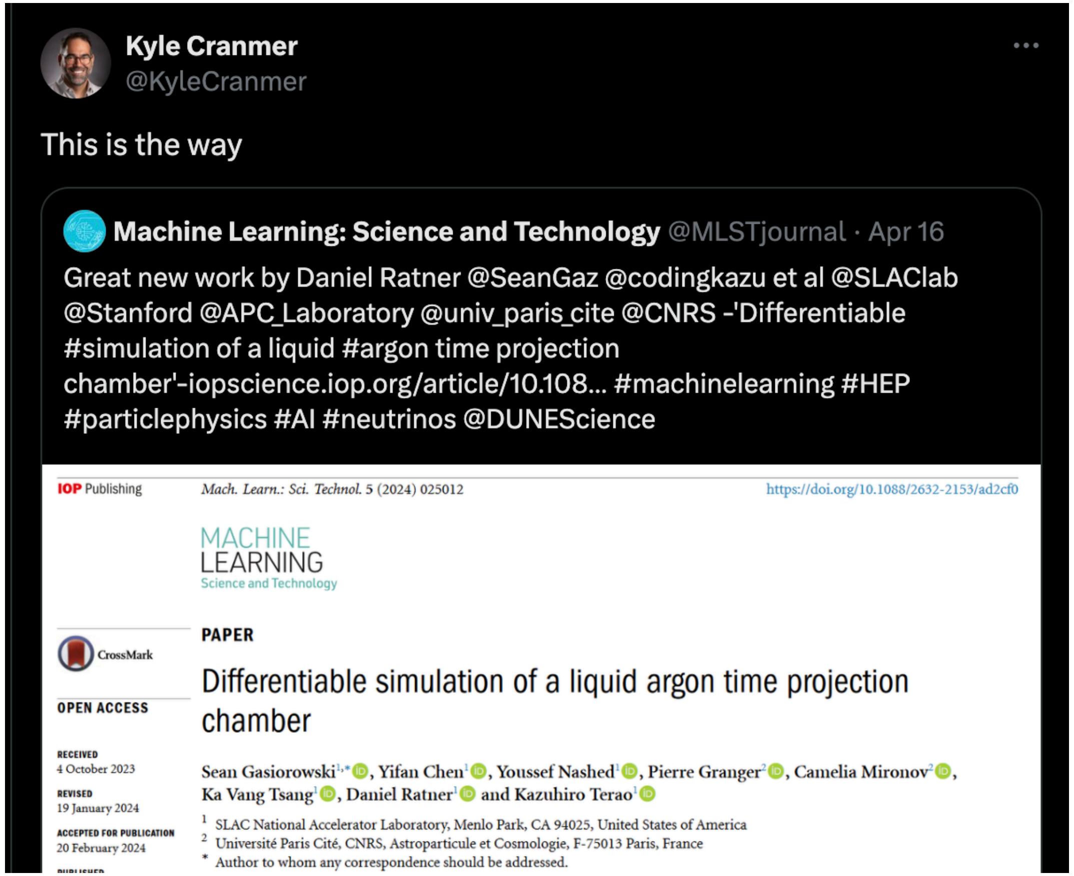
PyTorch

TensorFlow

Neural networks are just code

Machine learning libraries are able to efficiently calculate gradients with respect to neural network parameters

- Neural networks are just differentiable functions
- Why stop at neural networks?
- **Differentiable programming:** use ML libraries to write code (neural networks, but also e.g. exact physics simulators)
  - The **same techniques** that enable neural network training can be used to calculate gradients with respect to code parameters



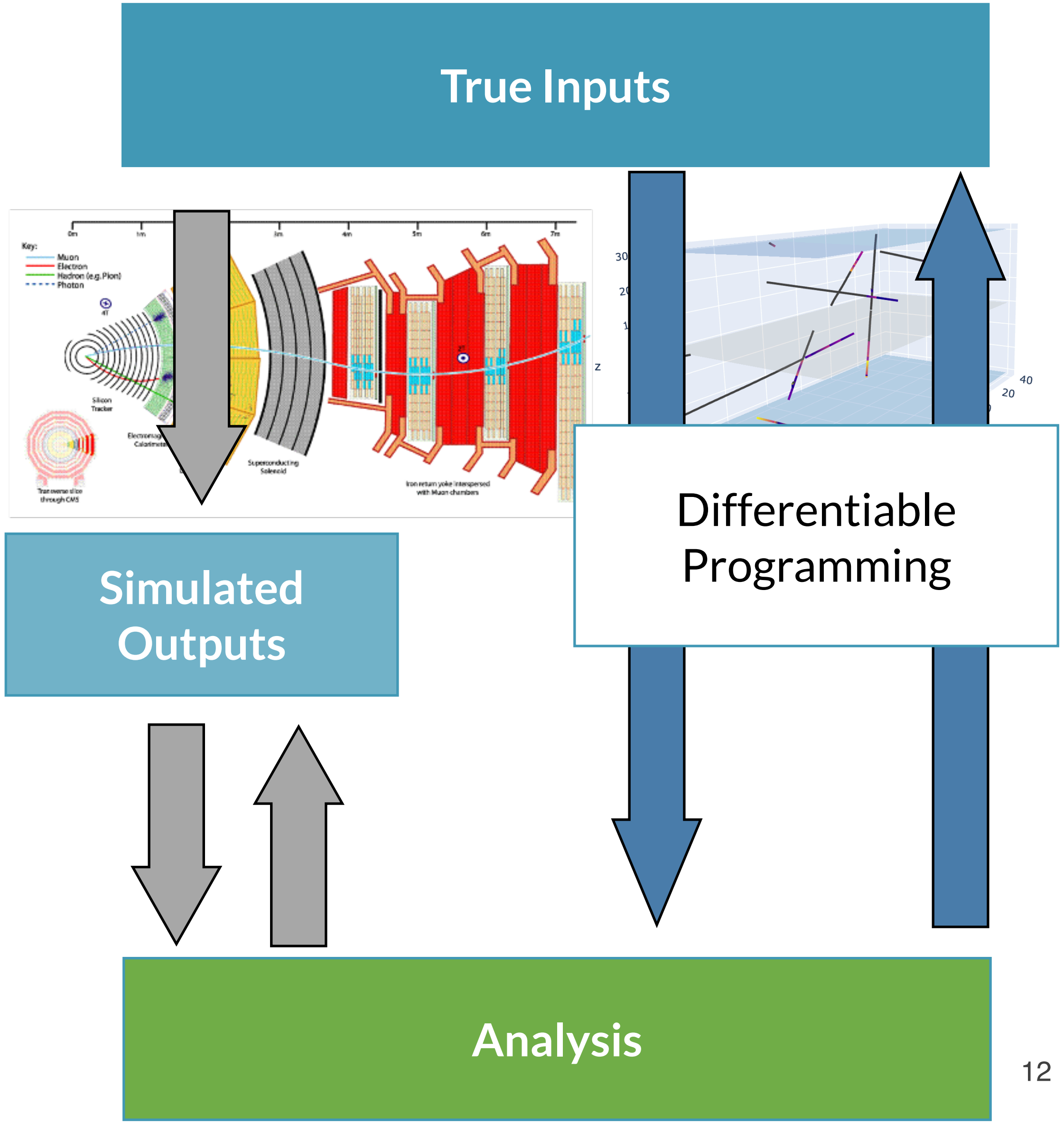
Why do we care?

Simulators are very important to HEP, but we often only use inputs and outputs

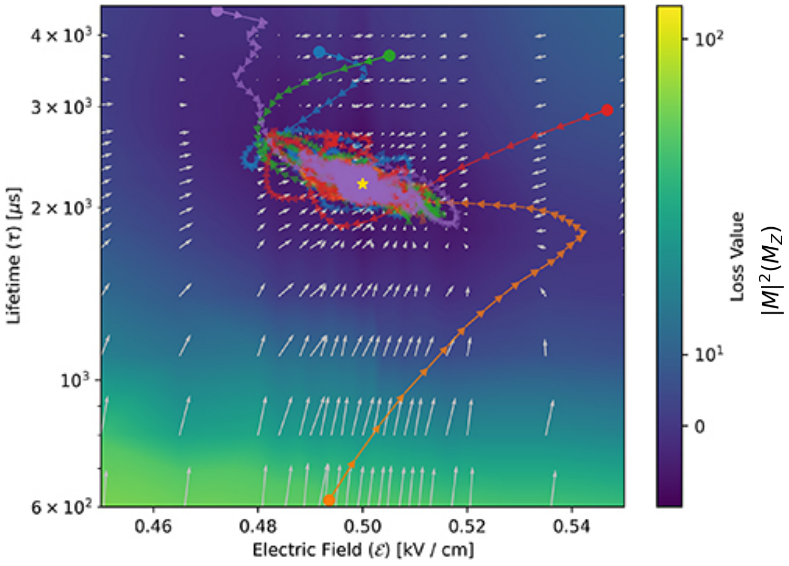
- Differentiable simulators can be directly used in ML pipelines — **explicitly use physics**, rather than relying on examples!
- Gradient information can be used to augment simulator output
- Fits of simulation to data can be used to understand and adjust underlying processes (e.g. **detector conditions/calibration**)

Analysis workflows feature many parameters (cuts, binning) that are often painstakingly tuned

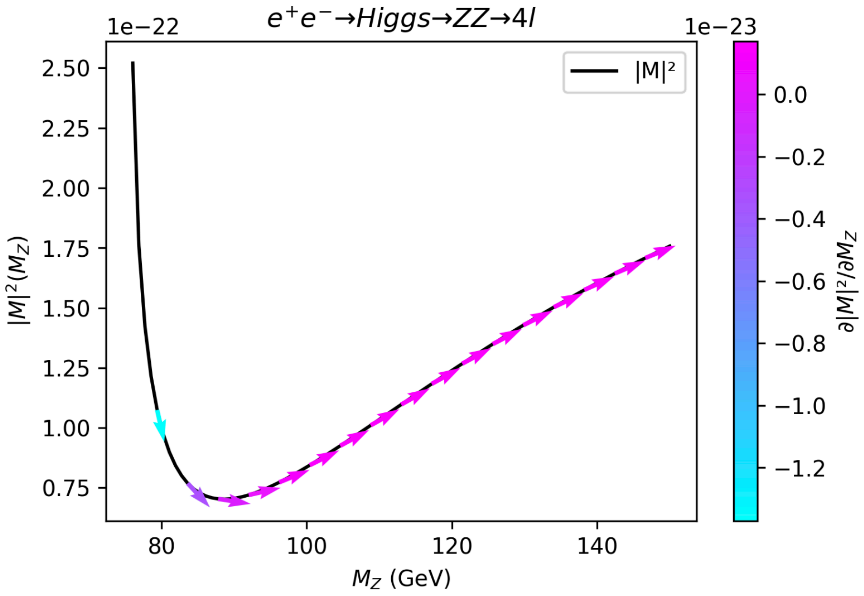
- Differentiable programming can make **optimizing** these **many parameters** possible



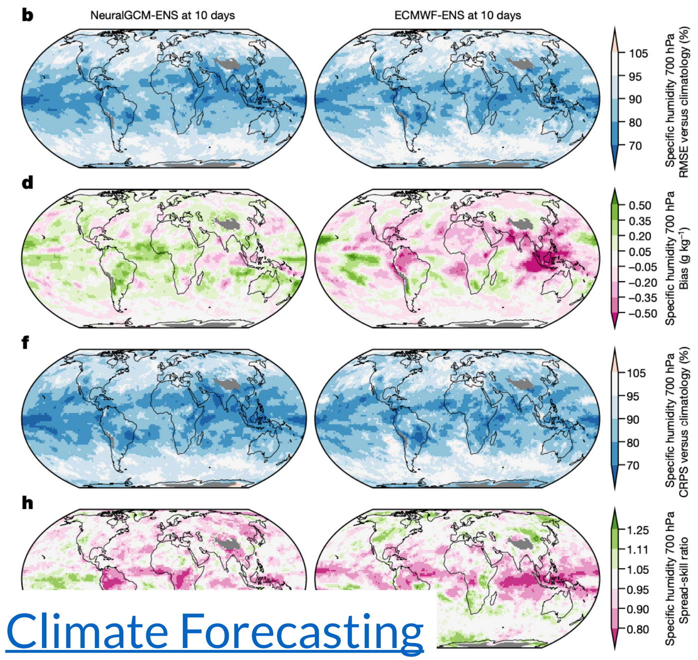
Differentiable Programming: Applications



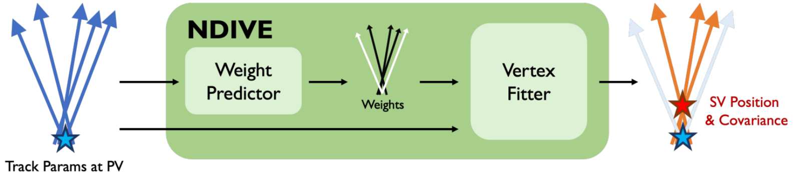
Neutrino Simulation



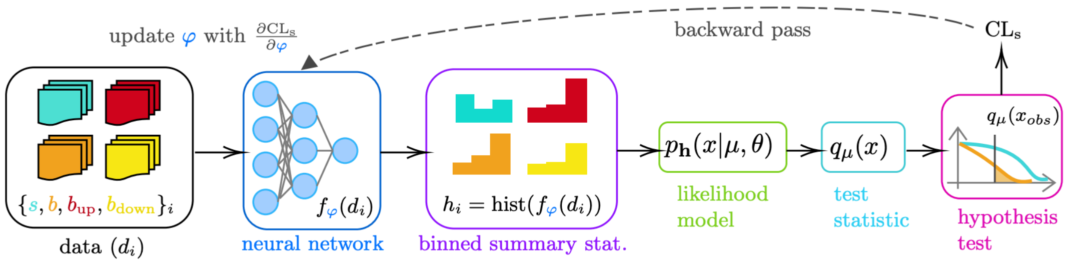
MadJAX



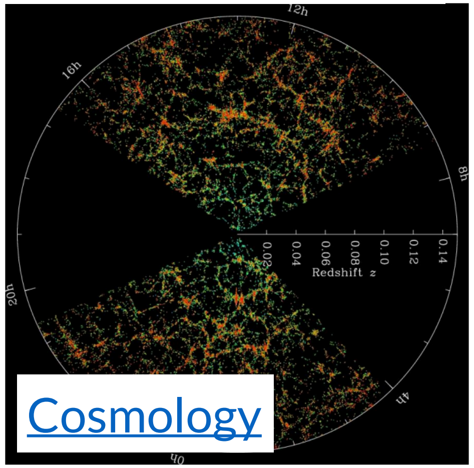
Climate Forecasting



Flavor Tagging

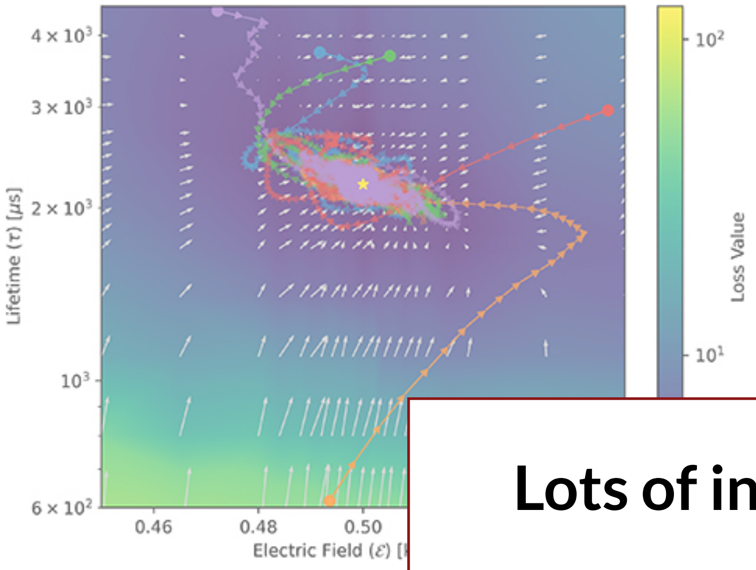


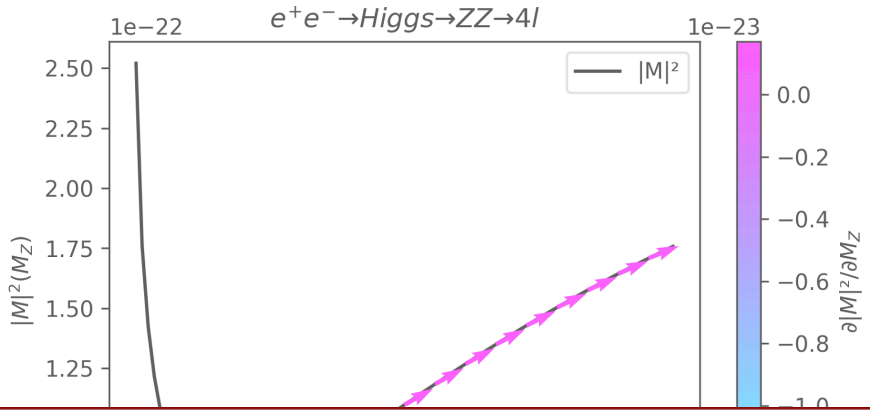
HEP Analysis

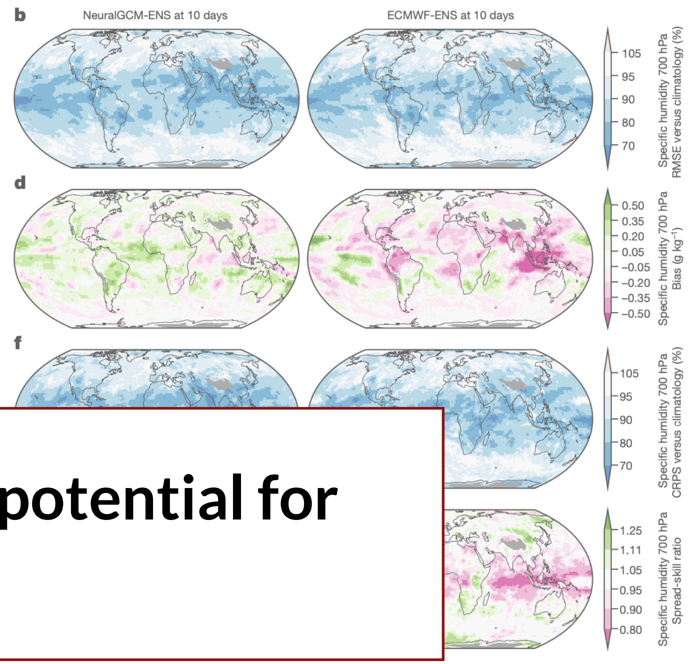


Cosmology

Differentiable Programming: Applications

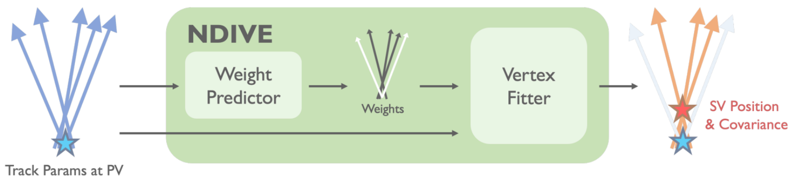




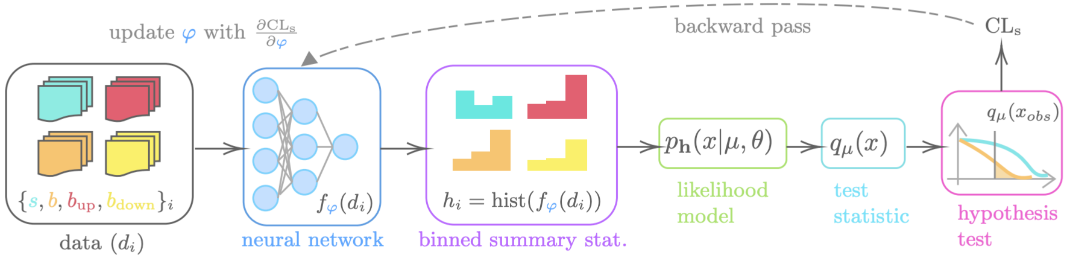


[Neutrino Simula](#)

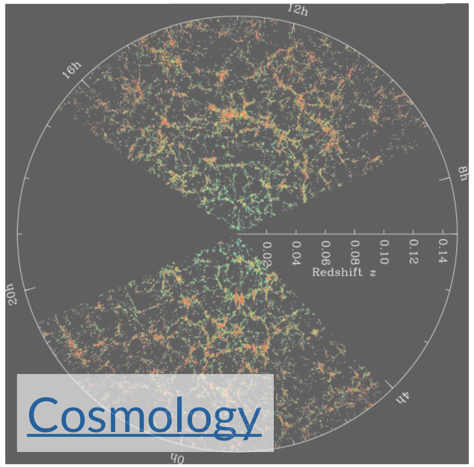
Lots of interest in the community + lots of potential for applications!



[Flavor Tagging](#)



[HEP Analysis](#)

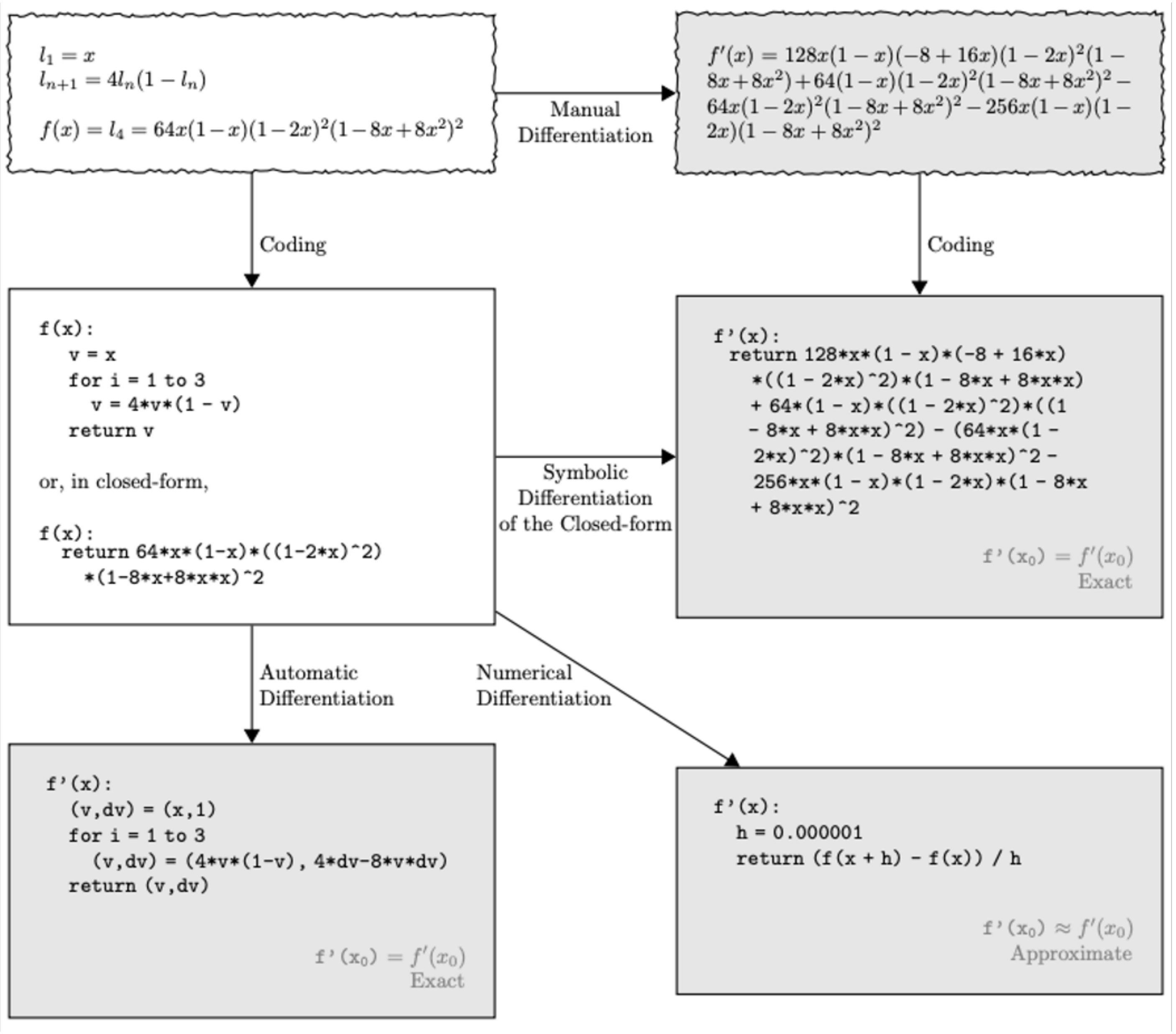


[Cosmology](#)

How does it work: Automatic Differentiation

Ways to Compute Derivatives of Code

Section modified from M. Kagan



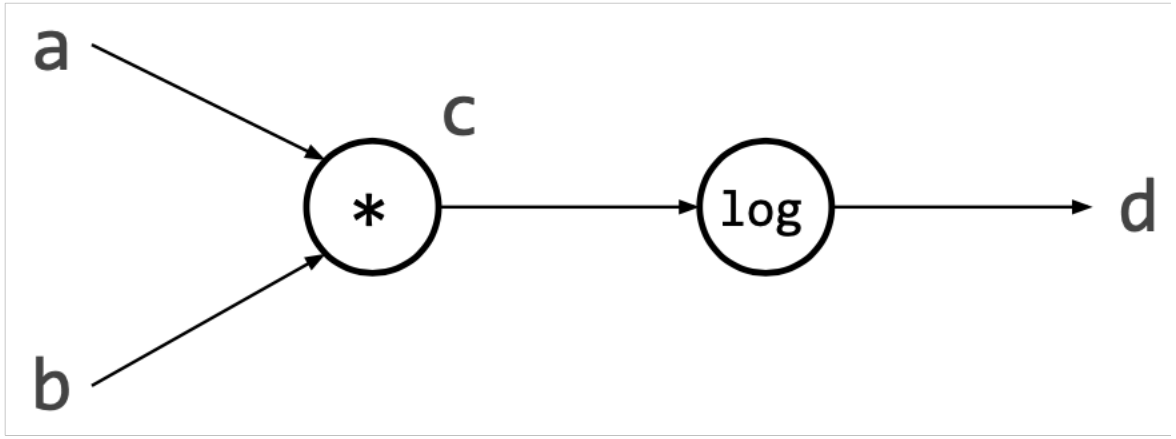
Baydin, Pearlmutter, Radul, Siskind. 2018. "Automatic Differentiation in Machine Learning: a Survey." Journal of Machine Learning Research (**JMLR**)

Ways to Compute Derivatives of Code

Automatic differentiation:

- Principle: break down arbitrary computer program into a graph of fundamental operations with known derivatives
- **Exact** gradient calculation, broadly applicable
- Scales well! Gradient cost ~ original code cost
  - e.g. neural networks ($f : \mathbb{R}^n \rightarrow \mathbb{R}$), forward + backward pass (gradients) ~2x cost of just forward (no gradients)

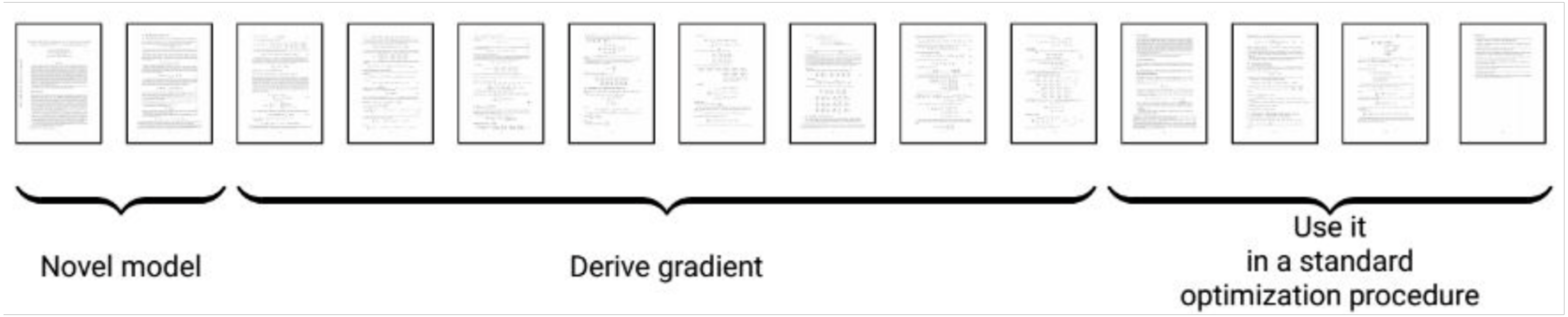
```
f(a, b):
    c = a * b
    d = log(c)
    return d
```



Ways to Compute Derivatives of Code

Manual differentiation:

- Derive expression by hand, then code it up
- Can be useful, but also labor intensive, case-by-case



Ways to Compute Derivatives of Code

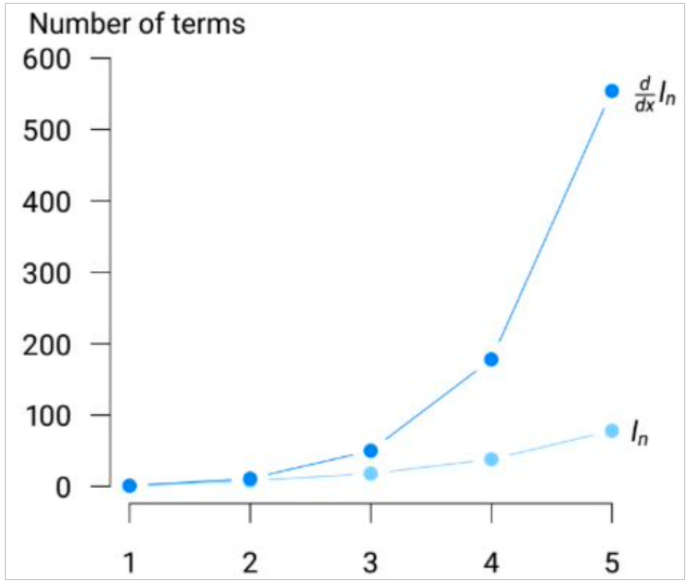
Symbolic differentiation:

- e.g. Mathematica, SymPy
- Gets messy/costly with number of terms
- Only applicable to closed form expressions (no control flow)

```
D[x^2, x]

2 x
```

Logistic map $l_{n+1} = 4l_n(1 - l_n), l_1 = x$

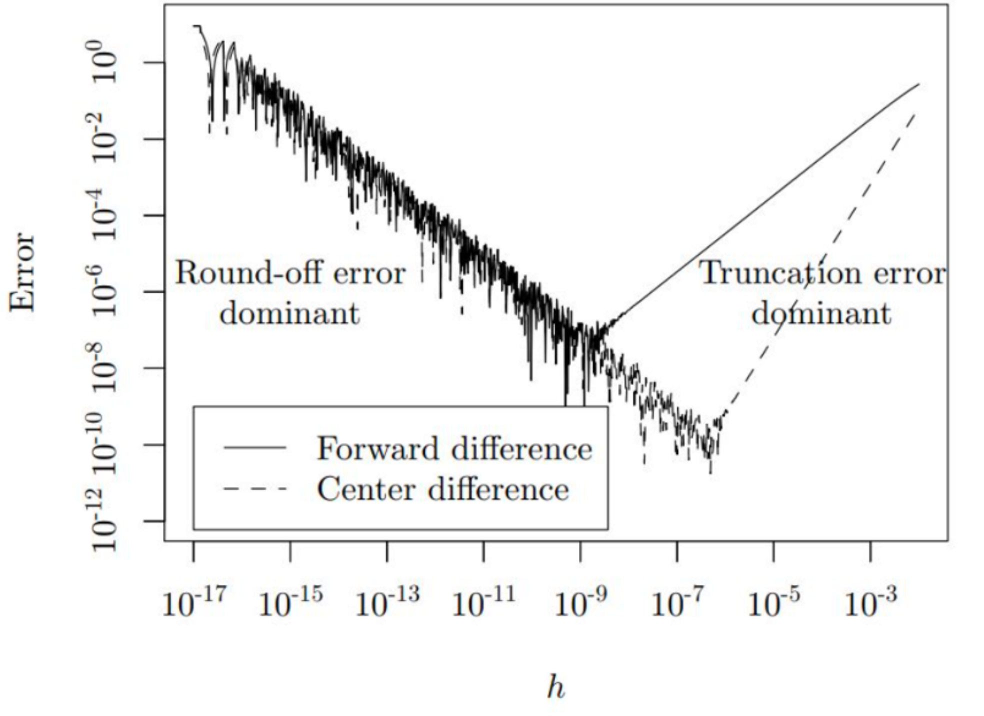
n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2 (1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$



Ways to Compute Derivatives of Code

Numerical differentiation (finite differences):

- $\dfrac{\partial f(\mathbf{x})}{\partial x_i} \approx \dfrac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, 0 < h \ll 1$
- Blows up with input dimensionality (one function eval per basis vector $\mathbf{e}_i$)
- Approximation errors from choices of h



SLAC

Automatic Differentiation: The Chain Rule in Disguise

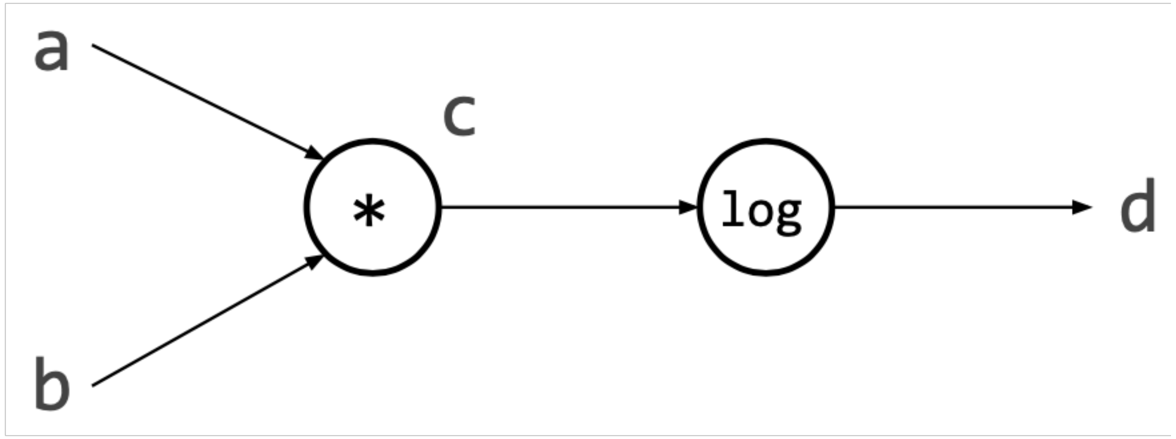
$$f(a, b) = \log(a \cdot b)$$

$$\nabla f(a, b) = \left(\frac{1}{a}, \frac{1}{b}\right)$$

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

Example: $\log(a \cdot b)$

- Represent as a **computational graph** showing all operations, dependencies

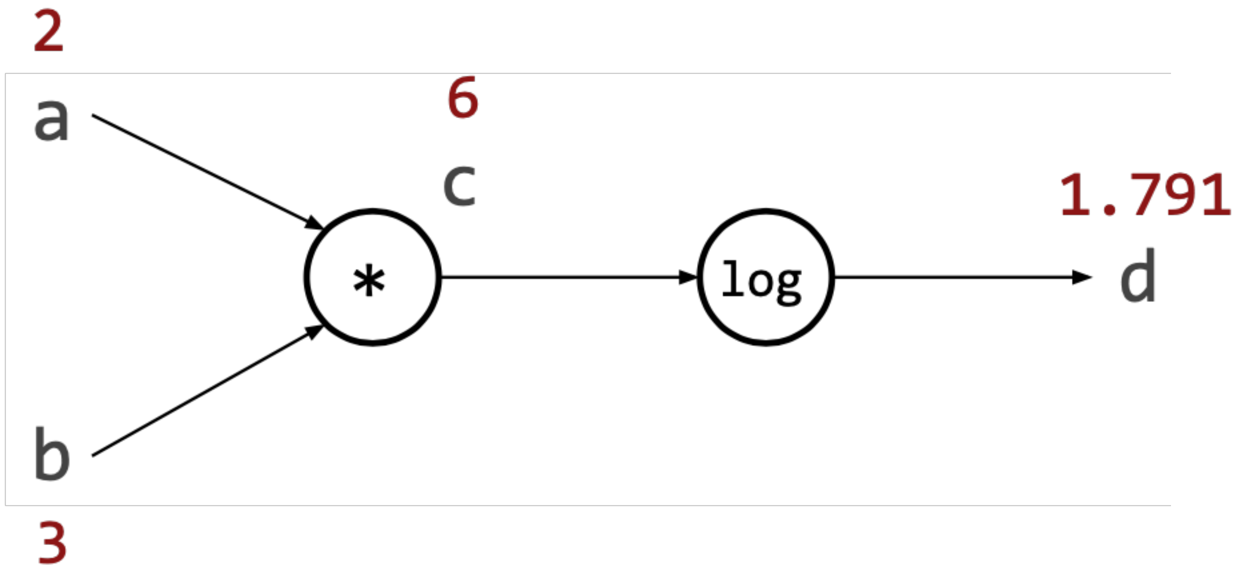


Automatic Differentiation: The Chain Rule in Disguise

Normal (forward) evaluation of the code for values of a, b results in a set of intermediate values (primals) at each stage of the computation

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

```
f(2, 3) = 1.791
```



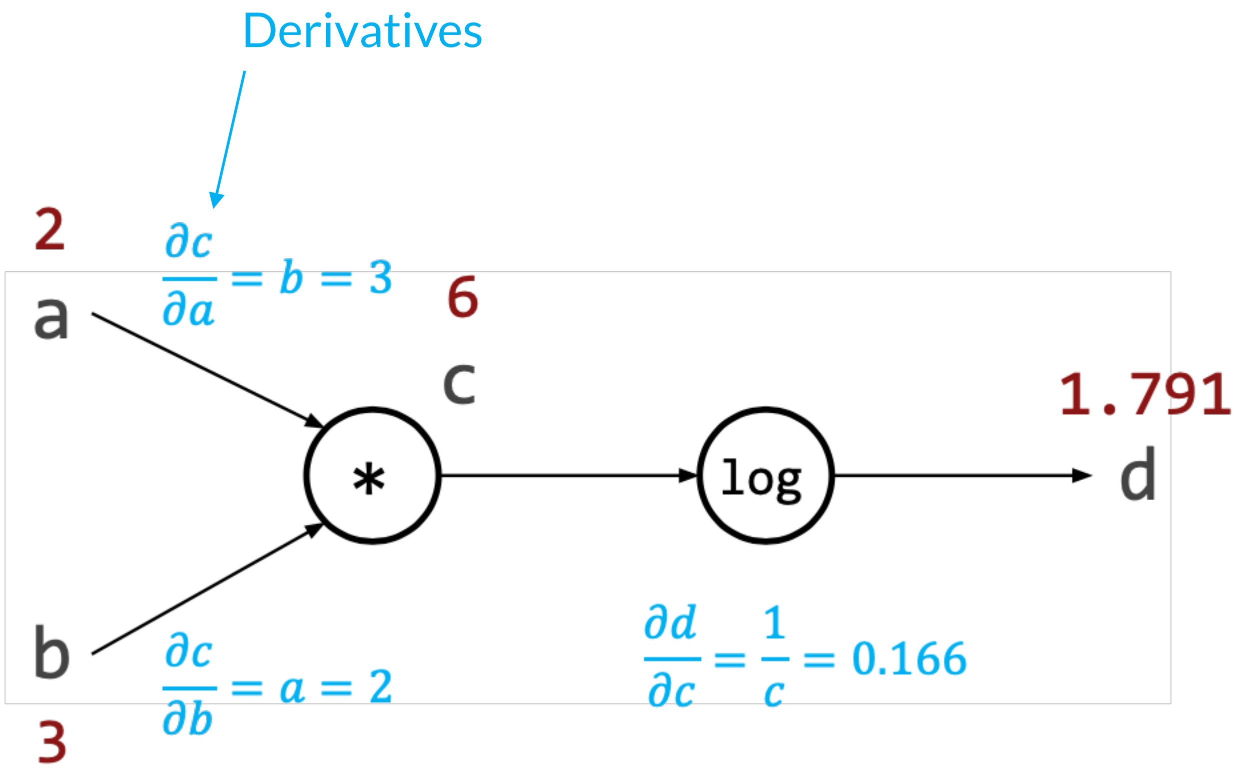
"Primals": intermediate function values

Automatic Differentiation: The Chain Rule in Disguise

The final result is a composition of the primal operations. The derivative of the final result is a product of the derivatives of each operation (via the chain rule).

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

```
f(2, 3) = 1.791
df(2,3) = [0.5, 0.333]
```



Chain Rule: $\frac{\partial d}{\partial a} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial a} = 0.166 * 3 = 0.5$

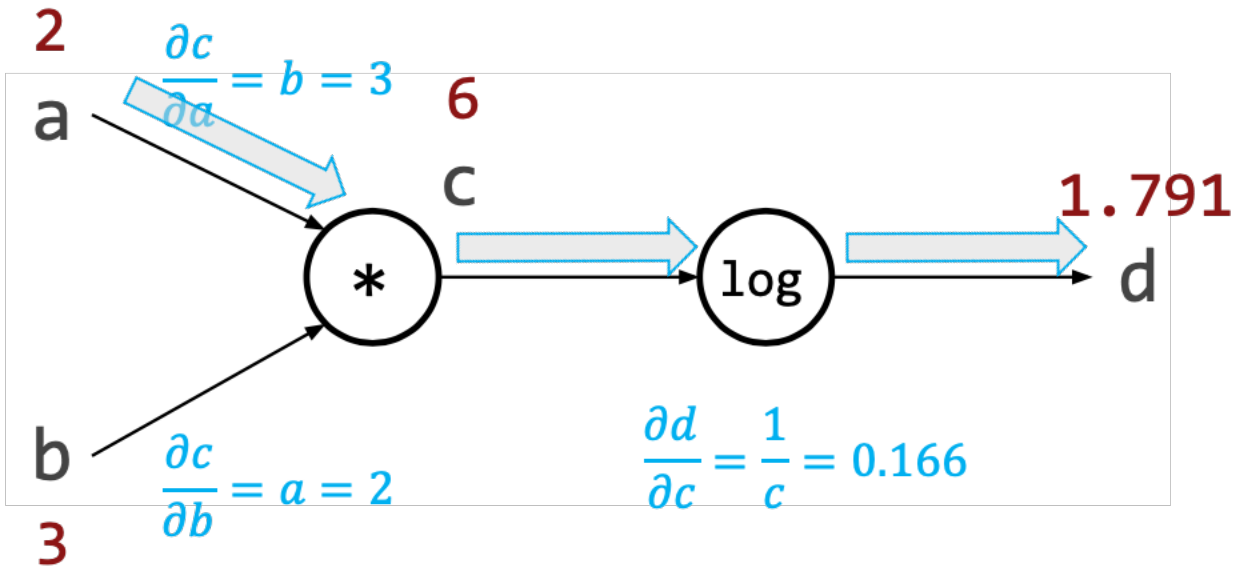
Automatic Differentiation: The Chain Rule in Disguise

Different modes of automatic differentiation <=> different order of evaluation of terms in the chain rule

- **Forward mode AD:** Inner (inputs) to outer (end result)

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

```
f(2, 3) = 1.791
df(2,3) = [0.5, 0.333]
```



Chain Rule: $\frac{\partial d}{\partial a} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial a} = 0.166 * 3 = 0.5$

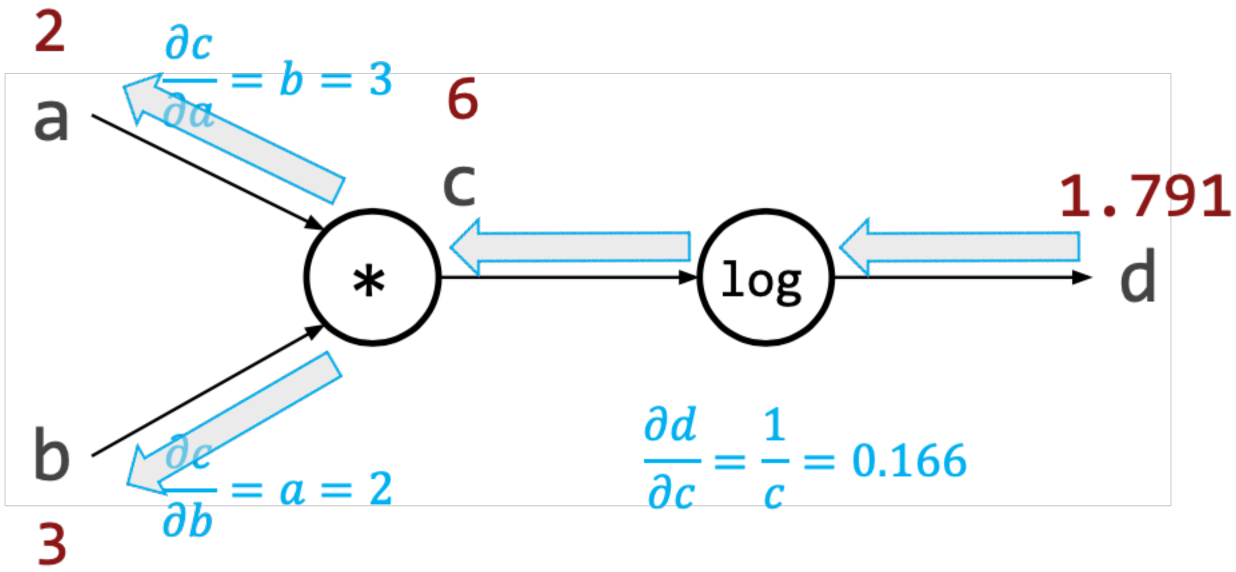
Outer ⇐ Inner

Automatic Differentiation: The Chain Rule in Disguise

Different modes of automatic differentiation <=> different order of evaluation of terms in the chain rule

- **Reverse mode AD (cf. backprop):** Outer (end result) to inner (inputs)

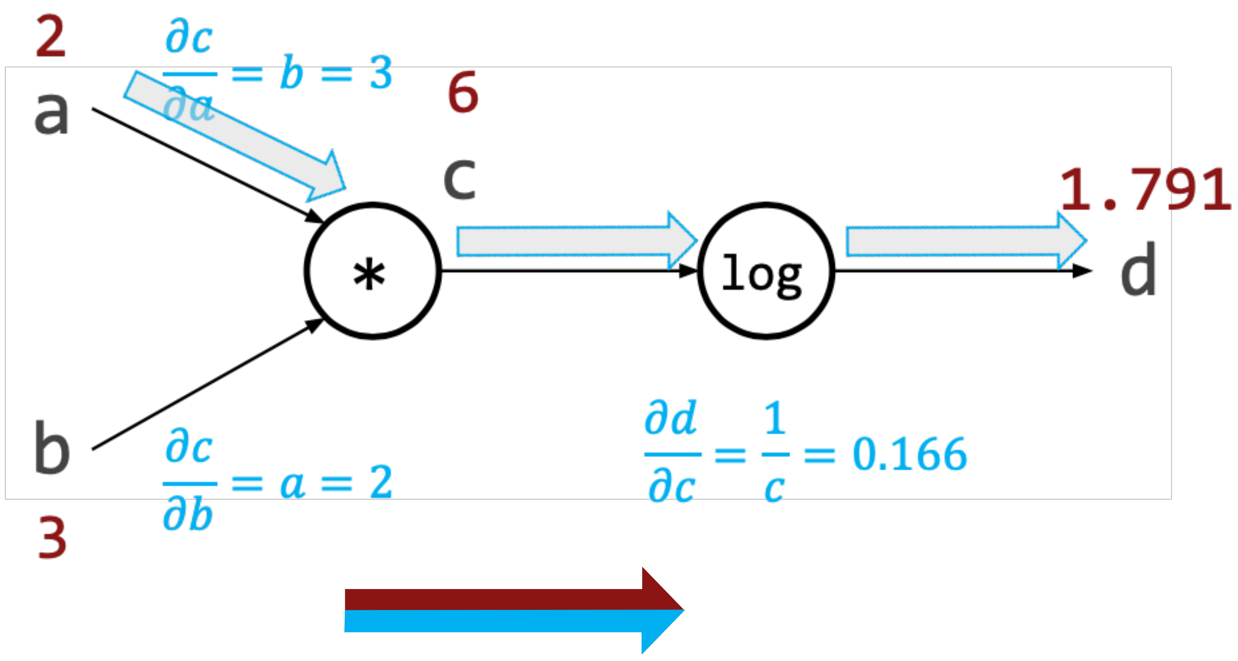
```
f(a, b):
    c = a * b
    d = log(c)
    return d

f(2, 3) = 1.791
df(2,3) = [0.5, 0.333]
```

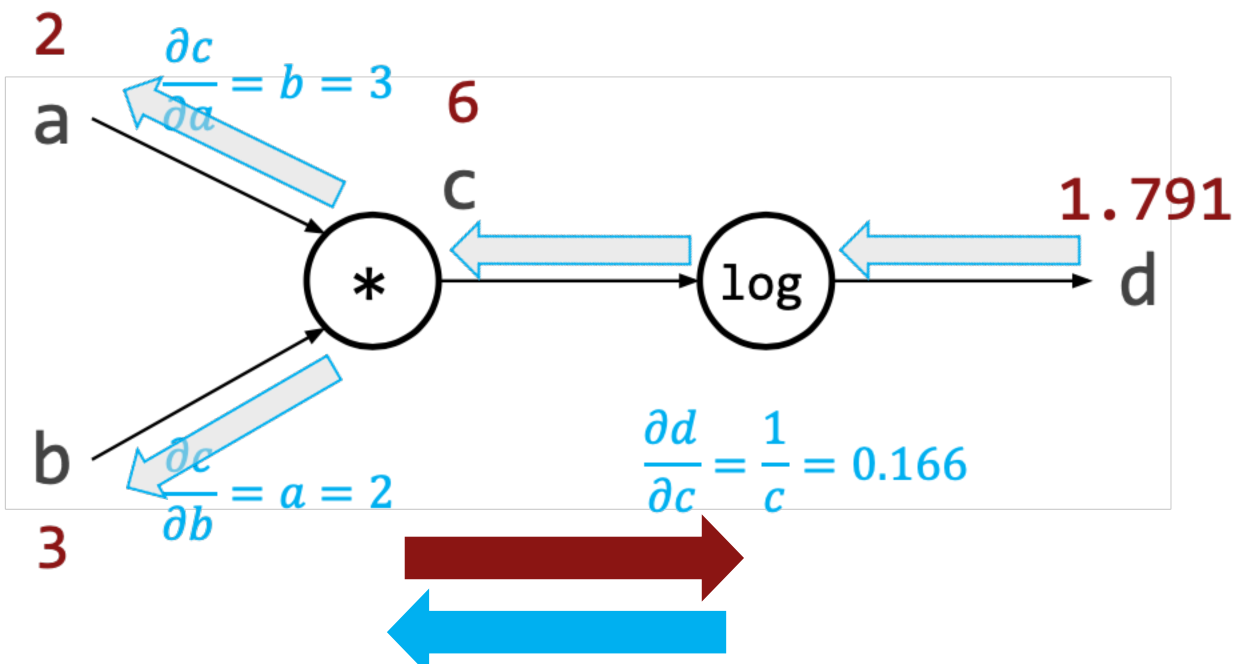


Chain Rule: $\frac{\partial d}{\partial a} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial a} = 0.166 * 3 = 0.5$

Outer ⇨ Inner

Automatic Differentiation: Forward vs Reverse Mode





Forward mode:
Compute primals and derivatives on single forward pass: follow the evaluation flow.

Additional sweep needed for each independent variable (e.g. b vs a)

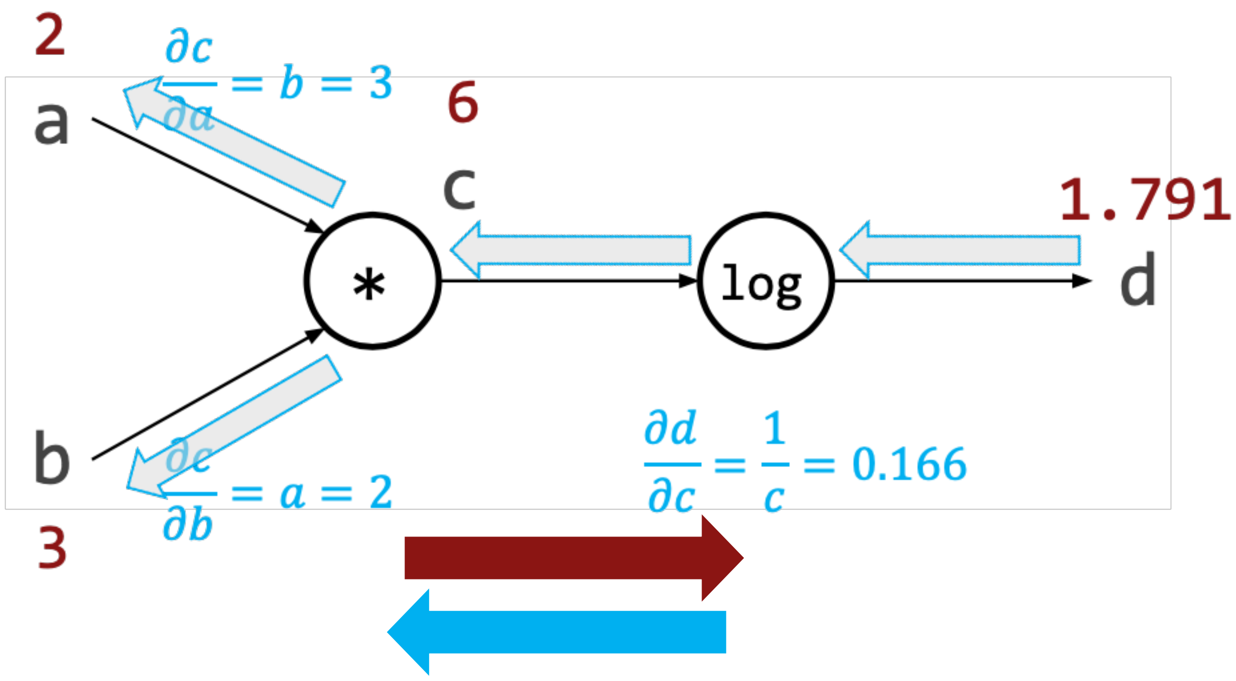
Reverse mode:
Compute and store primals on forward pass, compute and accumulate derivatives on backward pass

Additional sweep for needed for each dependent variable (e.g. multiple outputs)

Automatic Differentiation: Forward vs Reverse Mode

Neural networks usually have large number of inputs, small number of outputs (e.g. scalar loss function)

- => backpropagation <=> reverse mode AD more efficient



Reverse mode:
Compute and store primals on forward pass, compute and accumulate derivatives on backward pass

Additional sweep for needed for each dependent variable (e.g. multiple outputs)

How to compute efficiently?

$$\mathbf{f}(\mathbf{x}) : \mathbb{R}^N \to \mathbb{R}^M$$

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \boxed{\begin{matrix} \frac{\partial f_1}{\partial x_1} \\ \vdots \\ \frac{\partial f_M}{\partial x_1} \end{matrix}} & \begin{matrix} \cdots \\ \ddots \\ \cdots \end{matrix} & \begin{matrix} \frac{\partial f_1}{\partial x_N} \\ \vdots \\ \frac{\partial f_M}{\partial x_N} \end{matrix} \end{pmatrix}$$

Forward mode (single evaluation):
Derivatives of all M outputs w.r.t. one input => column of Jacobian matrix

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \boxed{\begin{matrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \end{matrix}} \\ \begin{matrix} \vdots & \ddots & \vdots \end{matrix} \\ \begin{matrix} \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{matrix} \end{pmatrix}$$

Reverse mode (single evaluation):
Derivatives of one output w.r.t. N inputs => row of Jacobian matrix

How to compute efficiently?

$$\mathbf{f}(\mathbf{x}) : \mathbb{R}^N \to \mathbb{R}^M$$

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

Forward mode (single evaluation):

Derivatives of all M outputs w.r.t. one input => column of Jacobian matrix

Relevant column can be extracted by multiplying by an appropriate basis vector:

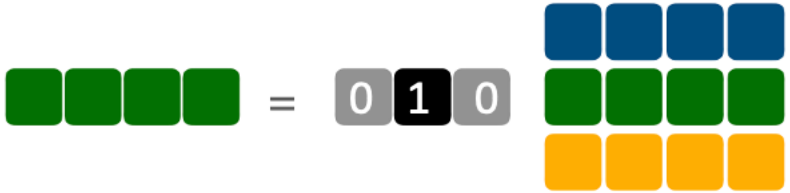
Forward mode AD <=> **Jacobian-vector product (JVP)**

How to compute efficiently?

$$\mathbf{f}(\mathbf{x}) : \mathbb{R}^N \rightarrow \mathbb{R}^M$$

$$\frac{d\mathbf{f}(\mathbf{x})}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix}$$

Reverse mode (single evaluation):
Derivatives of one output w.r.t. N inputs => row of Jacobian matrix



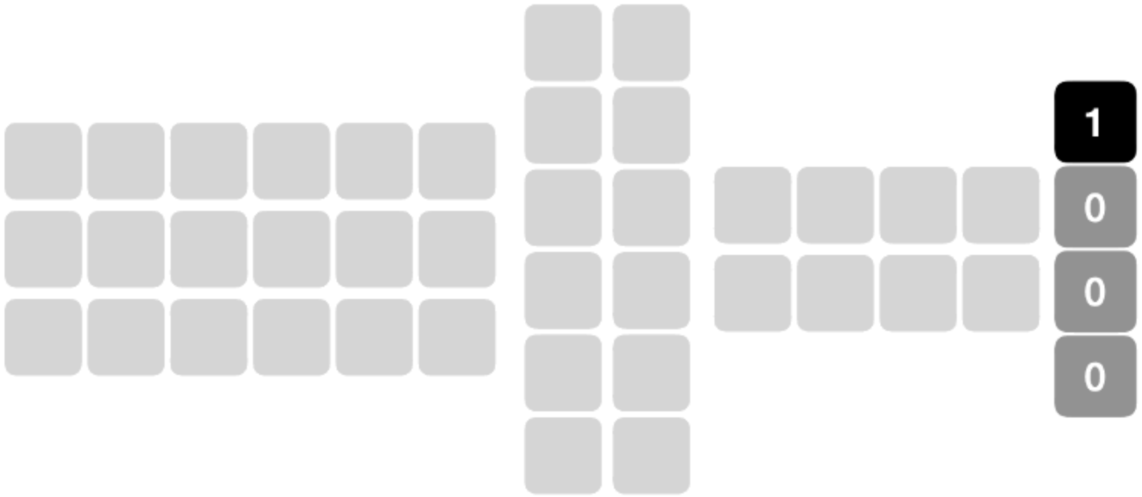
Relevant row can be extracted by multiplying by an appropriate basis vector:

Reverse mode AD <=> **vector-Jacobian product (VJP)**

How to compute efficiently?

Chain Rule: Jacobian matrix of function composition is product of Jacobian matrices of constituent functions

- e.g.: $J_{\mathbf{f}\circ\mathbf{g}(\mathbf{x})} = J_{\mathbf{f}}(\mathbf{g}(\mathbf{x})) \cdot J_{\mathbf{g}}(\mathbf{x})$
- Vector-Jacobian/Jacobian-vector product for **elementary operations** + composition => gradient computation
- See e.g. https://theoryandpractice.org/stats-ds-book/autodiff-tutorial.html for explicit examples



$$c_i = Me_i = M_3 M_2 M_1 e_i$$

Video: L. Heinrich

Tips and tricks

Things to know: Frameworks and Advantages

Much of the modern ML ecosystem is in Python

- **Advantages:** Quick start/ease of use, compatibility with other pieces of ML code
- **Disadvantages:**
  - Designed for neural networks/interpreted => loops can be slow
  - Mixed support for e.g. compilation, forward mode AD, etc

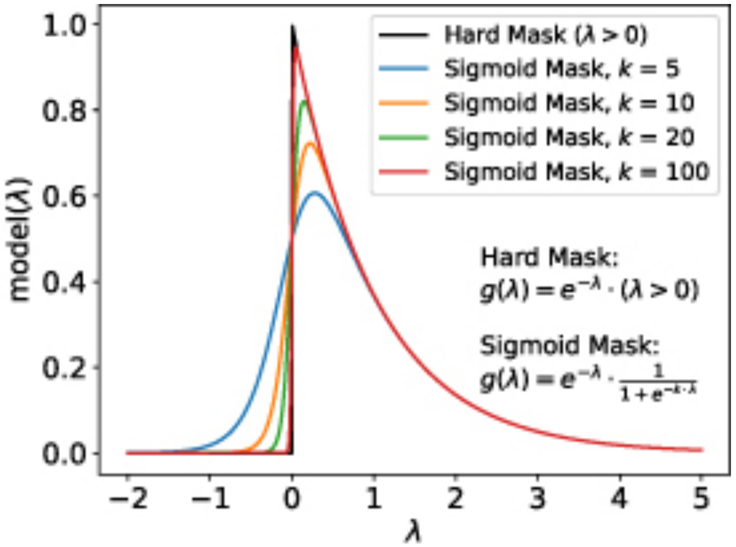
Rising interest in Julia:

- Community of AD support (e.g. Enzyme), potential performance advantages

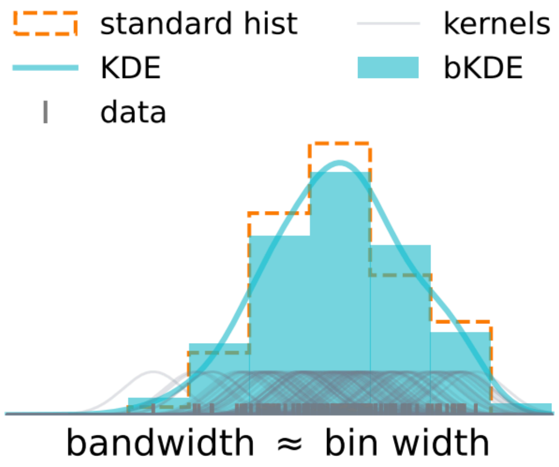
Things to know: Differentiability

Not everything is (trivially/usefully) differentiable!

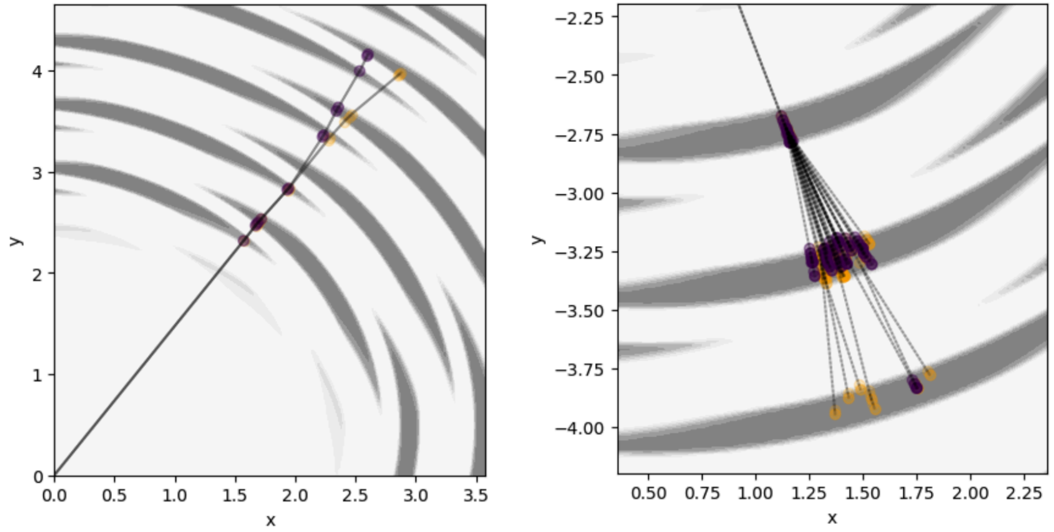
- But some workarounds/ways to get useful derivative information



[Hard cuts](#)



[Histograms](#)



[Branched processes](#)

Some Common Issues

Gradients can do a lot! But there's still some engineering involved in getting a good optimization:

My convergence is slow:

- Play with batching, explore GPU (multi-GPU) acceleration
- Experiment with different learning rates and optimizers

My optimization gets stuck at local minima:

- Check for model degeneracies/decouple parameters
- Start with a good guess

My convergence is unstable (e.g. sensitive to learning rate choice):

- Apply constraints: parameter/gradient clipping, loss modification/regularization
- Second order optimization

Everything breaks on real data:

- Calibrate the simulation (make it more like real data)
- Learn effects not in the simulation using, e.g., neural networks

Modified from Y. Nashed

Conclusions + Comments on Tutorial

Differentiable programming represents a broad class of tools including (but not limited to!) neural networks

- Can use common ML tools to write exact physics code that is **optimizable** both on its own and in conjunction with neural networks
- Automatic differentiation is just a clever use of the chain rule

Tutorial:

- Please pull/start with a fresh clone: https://github.com/ml4fp/2024-lbnl/tree/main
  - “diffprog” folder
  - Use pytorch-2.0.1