# Differentiable Programming

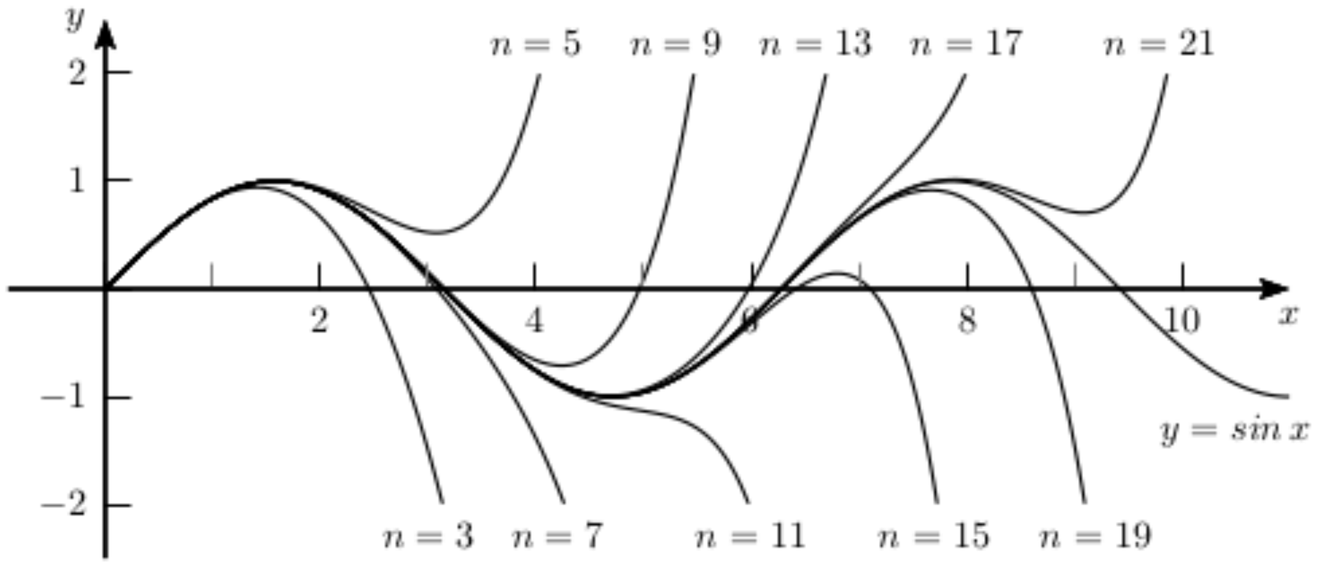
## ÖAW AI Winter School

Lukas Heinrich

# Why Derivatives are important

Derivatives at a point encode non-local information about functions

- valuable if we do not have global knowledge but can only evalulate the function (and now maybe its derivatives) locally
- From Taylor Expansion: higher order derivatives ↔ longer reach



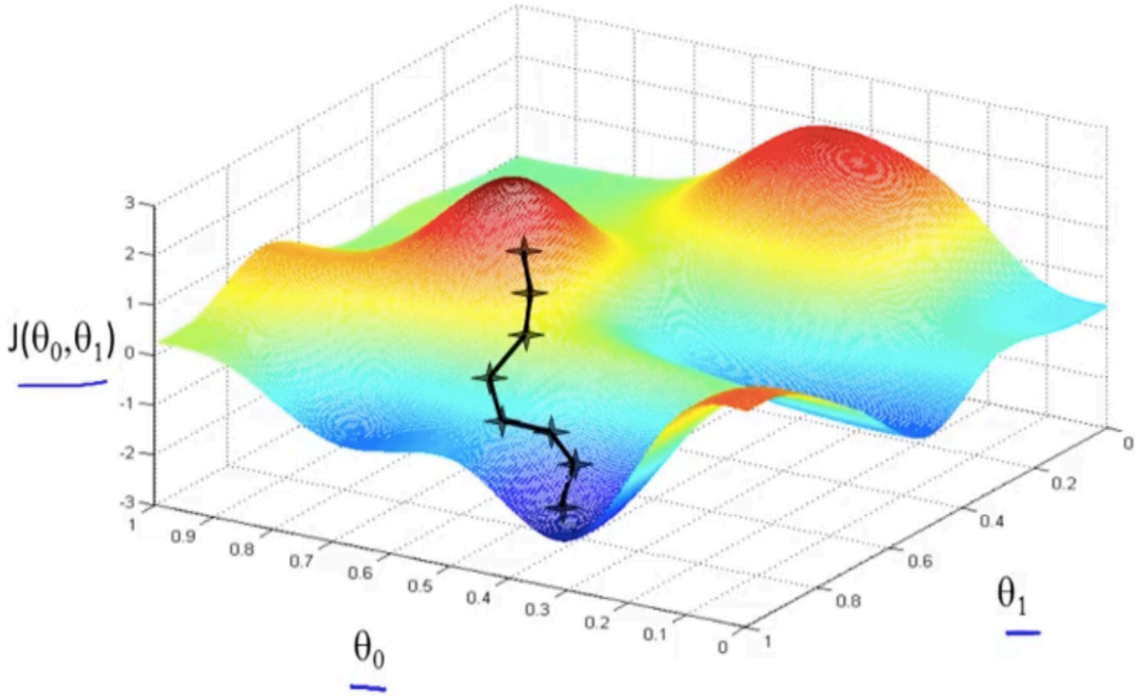
# Derivatives in ML & Physics

In optimization tasks (training Neural Nets, finding best-fit parmeters) having a cheap way to compute gadients is crucial

Gives you a sense of direction
in high-dimensional space

→ walk towards a minimum
by just following the gradient

→ crucial ingredient ot make
e.g. Deep Learning work

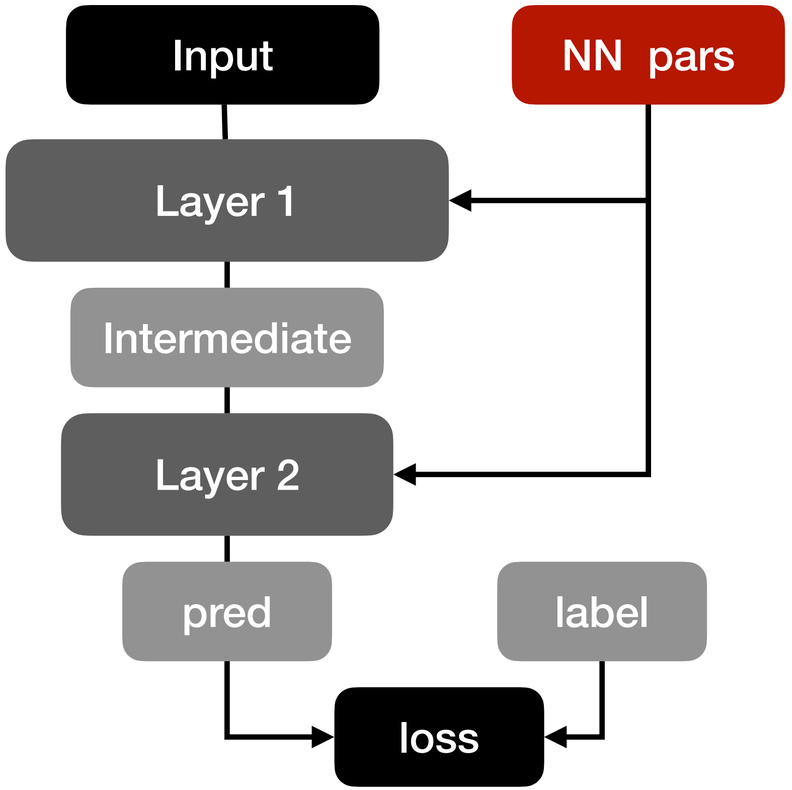


# Example: Neural Networks

Training neural networks:
Gradient of loss function w.r.t.
neural network parameters

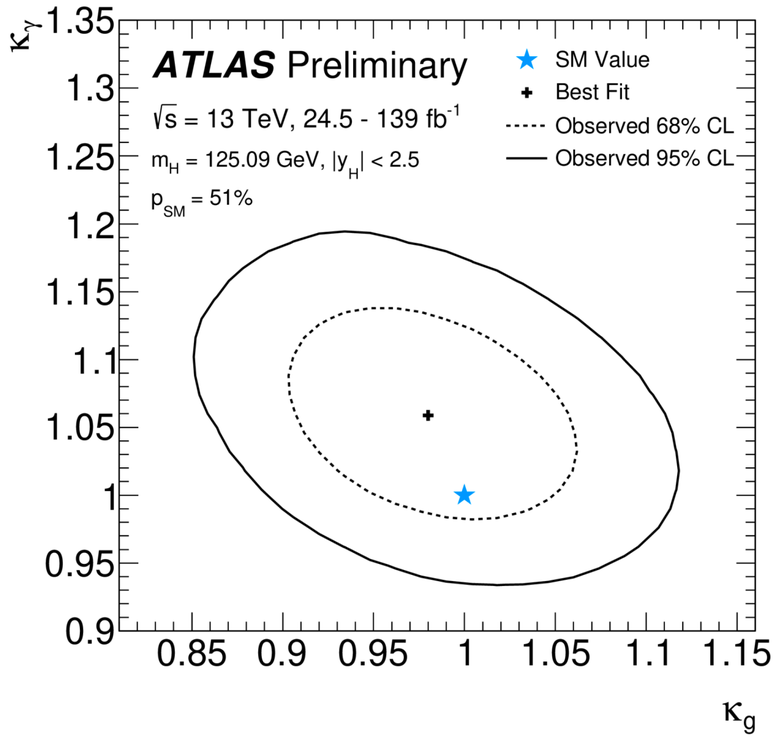
$$y = \mathrm{Loss}(x; \phi)$$

$$\phi \leftarrow \phi - \nabla_{\phi} \mathrm{Loss}$$



# Example: Statistical Analysis

Maximum likelihood fit:
Gradient of likelihood function w.r.t. model parameters useful to find best-fit point (MIGRAD)




$$p(x \mid \alpha)$$

$$\hat{\alpha} = \text{argmax}_{\alpha} p(x \mid \alpha)$$

# Standard Ways to get Derivatives

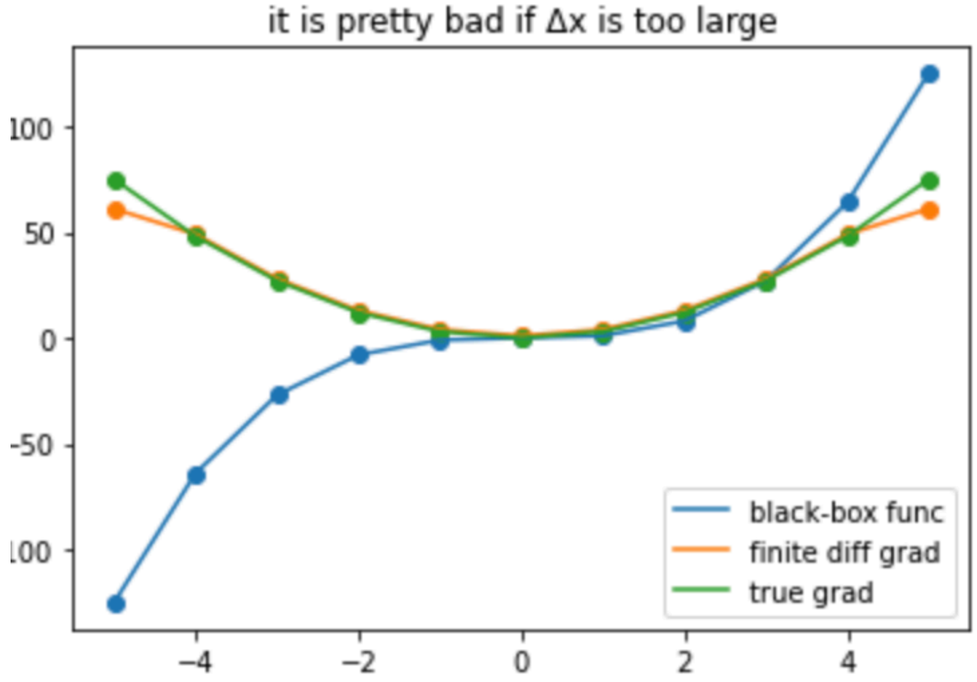
For an arbitrary function, the easiest way to get a derivative is through "numeric differentiation" (also called "finite differences")

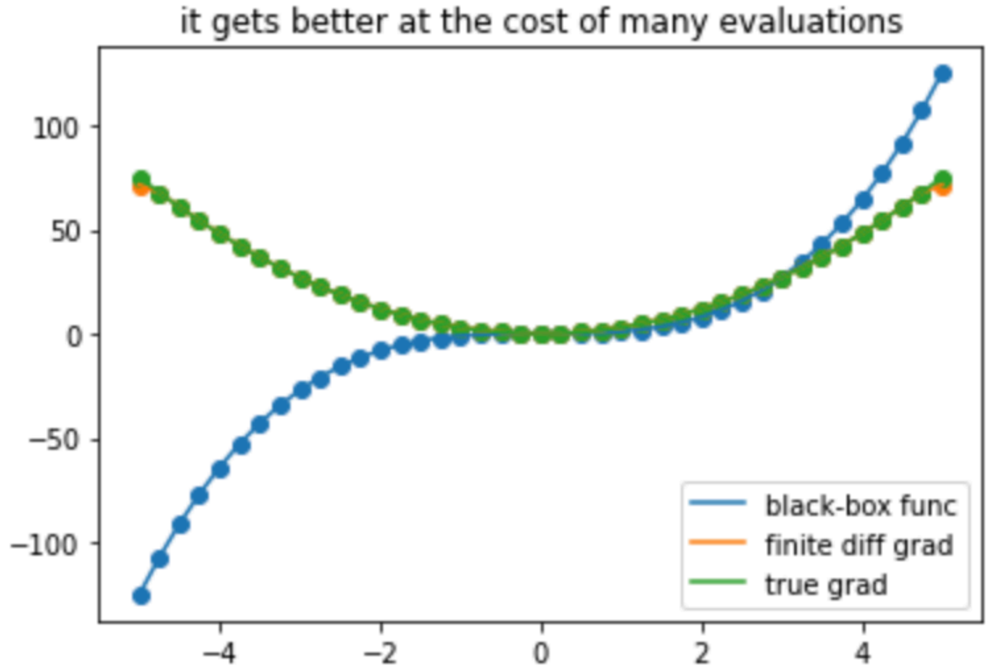
$$\frac{\partial f}{\partial x}\Big|_{x=x_0} \approx \frac{\Delta y}{\Delta x} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

# Standard Ways to get Derivatives

Pro: very easy to code up, works in any programming language.

Con: to be precise you need a small $\Delta x$ - does not work in high-D
(completely infeasible for neural nets w/ millions of params)
will always stay an approximation, never exact



# Standard Ways to get Derivatives

Computer Algebra Systems allow you to get exact gradients!
(Mathematica, SymPy) through "symbolic diffferentiation"

```
import sympy

symbolic_x = sympy.symbols('x')
symbolic_func = symbolic_x**3
symbolic_func
```

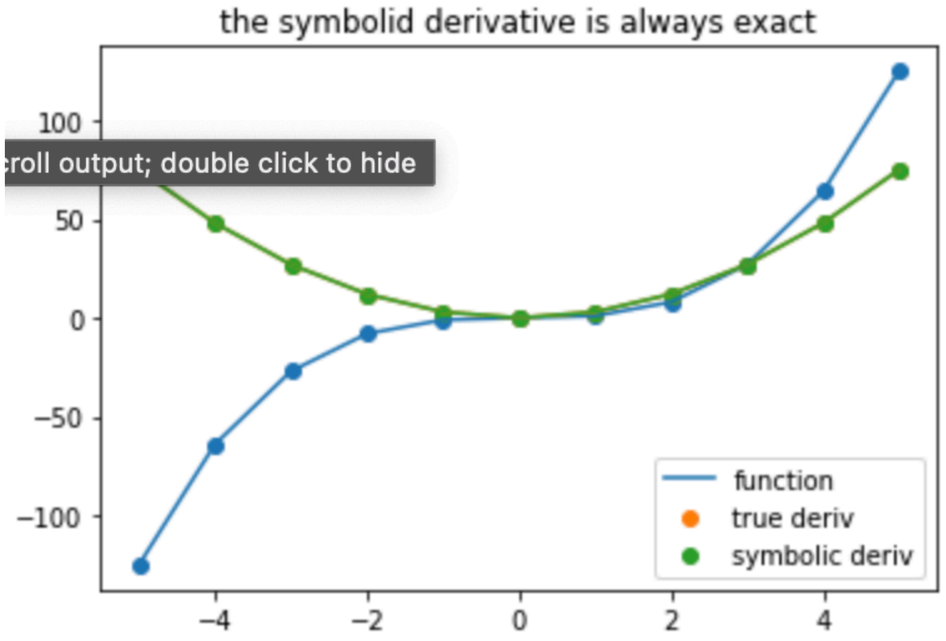
$$x^3$$

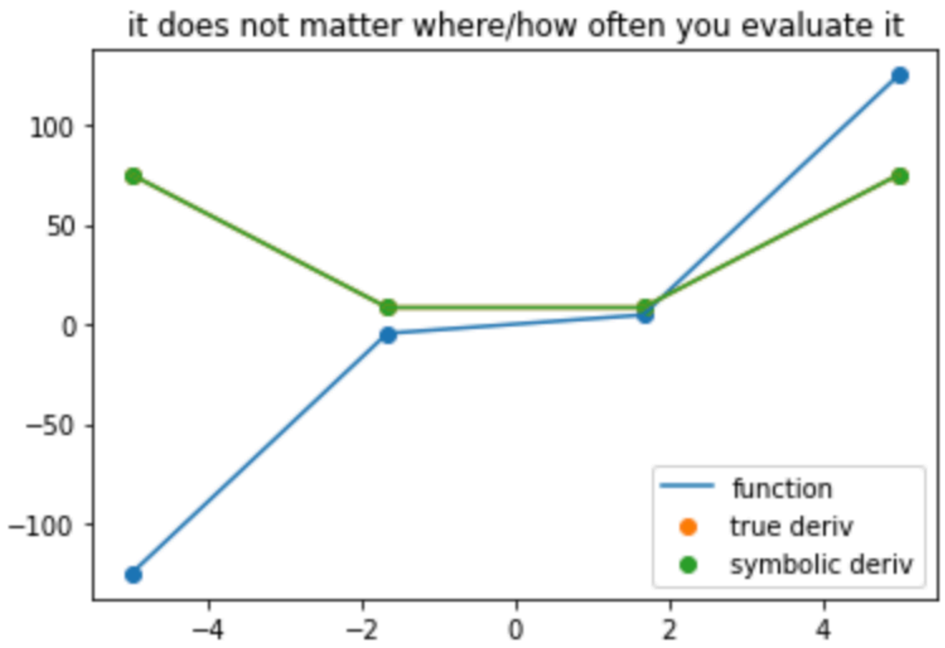
```
symbolic_deriv = symbolic_func.diff(symbolic_x)
symbolic_deriv
```

$$3x^2$$

# Standard Ways to get Derivatives

## Pro: Gradients are exact independent of where you evaluate



# Standard Ways to get Derivatives

Pro: Gradients are exact independent of where you evaluate

Con: Symbolic frameworks can be inefficient/memory-intensive (repeated subexprs, etc...) & hard to integrate into larger systems

```
quad_6_times.diff(symbolic_x)
```

$$486x + 81\,(4x+6)\left(x^2+3x+4\right) + 27\left(12x + 2\,(4x+6)\left(x^2+3x+4\right) + 18\right)\left(3x^2+9x+\left(x^2+3x+4\right)^2+16\right)$$
$$+ 9\left(36x + 6\,(4x+6)\left(x^2+3x+4\right) + 2\left(12x + 2\,(4x+6)\left(x^2+3x+4\right) + 18\right)\left(3x^2+9x+\left(x^2+3x+4\right)^2+16\right) + 54\right)\left(9x^2+27x\right.$$
$$\left. + 3\left(x^2+3x+4\right)^2 + \left(3x^2+9x+\left(x^2+3x+4\right)^2+16\right)^2 + 52\right)$$
$$+ 3\left(108x + 18\,(4x+6)\left(x^2+3x+4\right) + 6\left(12x + 2\,(4x+6)\left(x^2+3x+4\right) + 18\right)\left(3x^2+9x+\left(x^2+3x+4\right)^2+16\right)\right.$$
$$\left. + 2\left(36x + 6\,(4x+6)\left(x^2+3x+4\right) + 2\left(12x + 2\,(4x+6)\left(x^2+3x+4\right) + 18\right)\left(3x^2+9x+\left(x^2+3x+4\right)^2+16\right) + 54\right)\left(9x^2+27x+3\left(x^2+\right.\right.\right.$$

# Standard Ways to get Derivatives

Automatic Differentiation is a third method that

- produces exact gradients like symbolic differentiation
- is more efficient than symbolic differentiation
- more easily integratable into standard programming languages

Numeric
Differrentiation

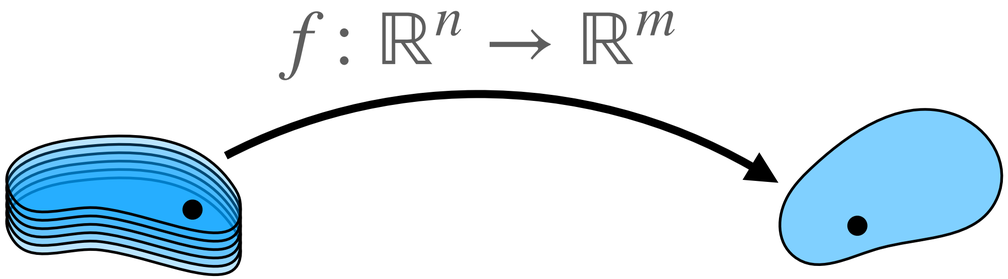
Symbolic
Differrentiation

**Automatic**
**Differrentiation**

# Smooth Functions

In general we're interested in derivatives functions that map between spaces with different dimensionality

- how do gradients look like in this case?

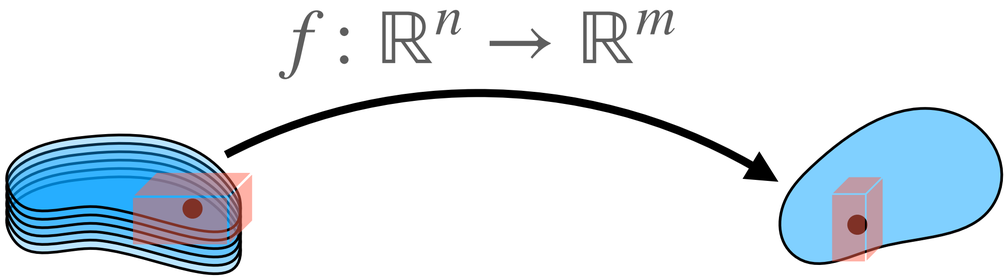



$$y = f(x)$$

# Smooth Functions

In general we're interested in derivatives functions that map between spaces with different dimensionality

- how do gradients look like in this case?
- Jacobian Matrix captures full first-order derivatives




$$y = f(x)$$

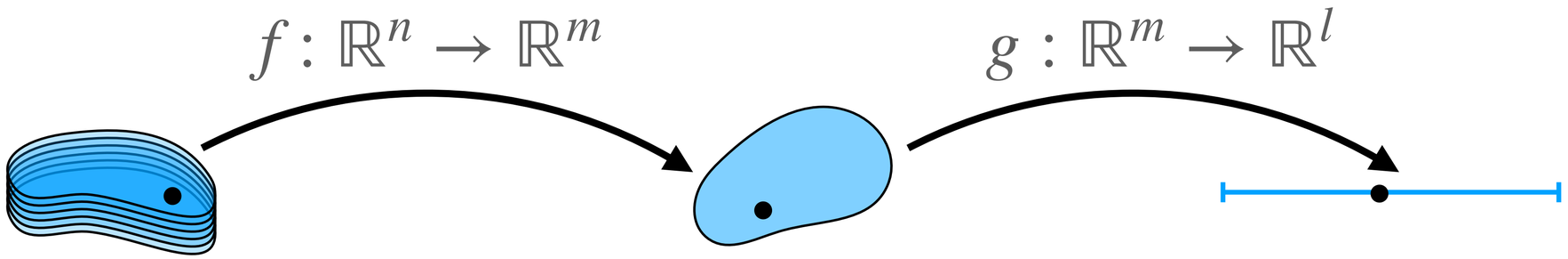
$$dy = J_f \, dx$$

$$J_f = \frac{\partial(y_1, \ldots, y_m)}{\partial(x_1, \ldots, x_n}$$

# Composition

We also often chain functions together

- how are gradients of ingredients related to gradients of total?

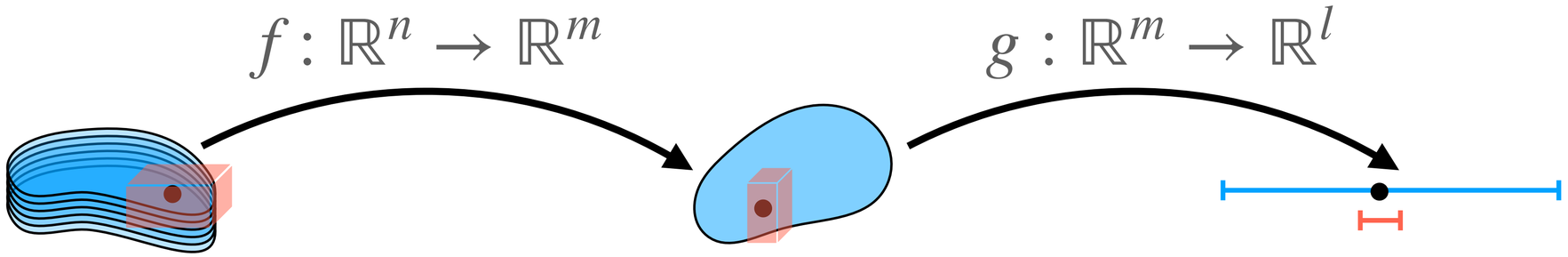



$$z = (g \circ f)(x) = g(f(x)) = g(y)$$

# Composition

We also often chain functions together

- how are gradients of ingredients related to gradients of total?
- just the matrix product of individual Jacobians



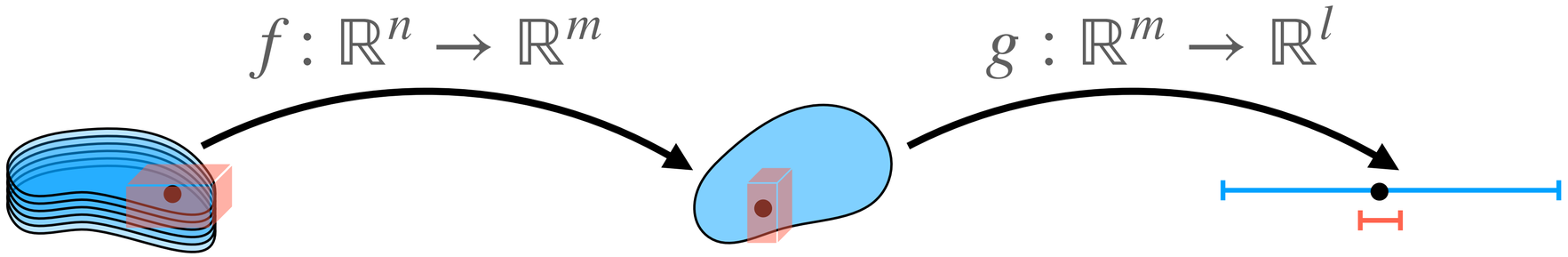

$$z = (g \circ f)(x) = g(f(x)) = g(y)$$

$$dz = J_{g \circ f} = J_g J_f \, dx = J_g dy$$

# Upshot: Jacobians are all we need

Jacobian Matrices fully capture the gradient information

- we'll look at efffective ways to calculate them

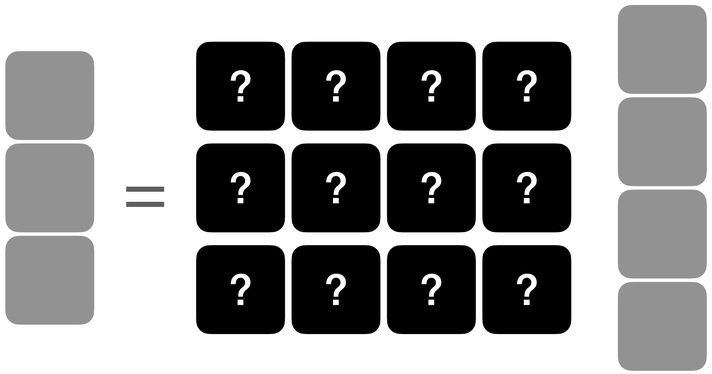



$$dz = J_{g \circ f} = J_g J_f \, dx = J_g dy$$

# Inspecting Linear Maps via Appplication

Linear Maps (i.e. Matrices) can be fully characterized by how they act on vectors

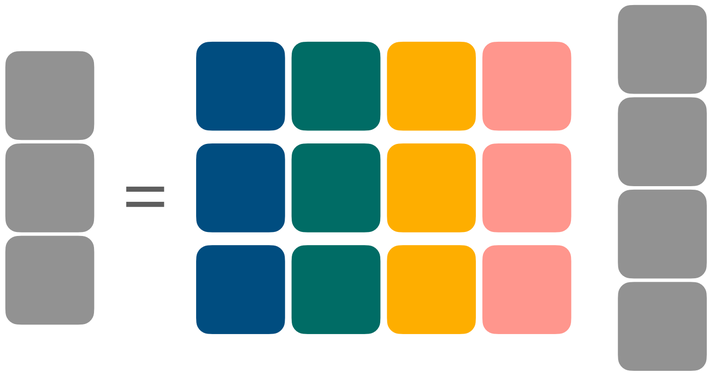
**Q: can we extract values of this matrix by a good choice of vector ?**



# Inspecting Linear Maps via Appplication

Linear Maps (i.e. Matrices) can be fully characterized by how they act on vectors

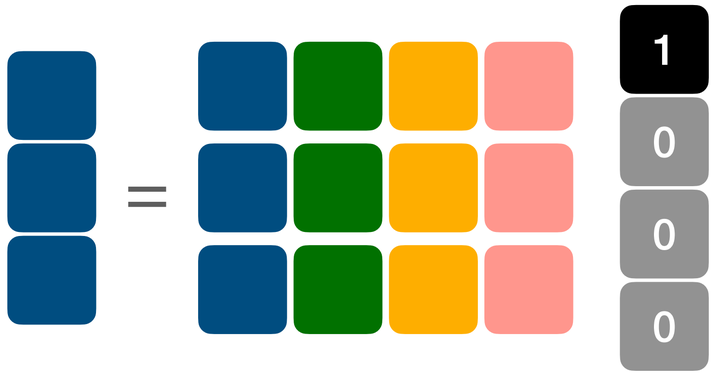
**Q: can we extract values of this matrix by a good choice of vector ?**



# Inspecting Linear Maps via Appplication

Linear Maps (i.e. Matrices) can be fully characterized by how they act on vectors

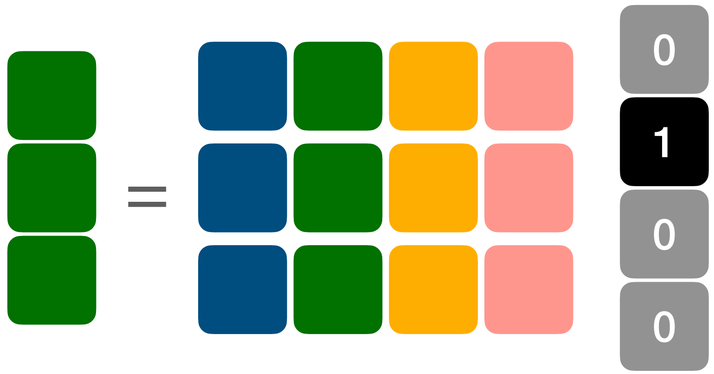
**Q: can we extract values of this matrix by a good choice of vector ?**



# Inspecting Linear Maps via Appplication

Linear Maps (i.e. Matrices) can be fully characterized by how they act on vectors

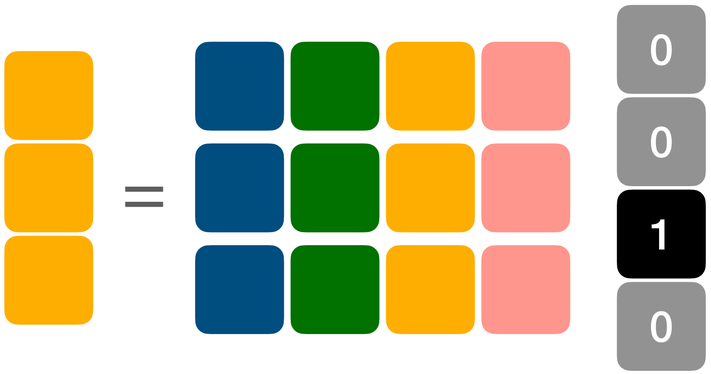
**Q: can we extract values of this matrix by a good choice of vector ?**



# Inspecting Linear Maps via Appplication

Linear Maps (i.e. Matrices) can be fully characterized by how they act on vectors

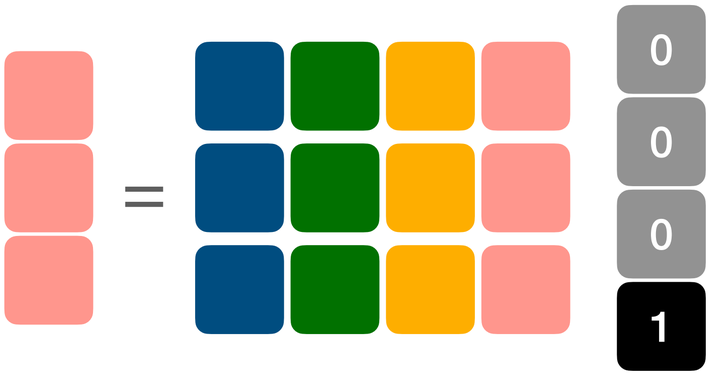
**Q: can we extract values of this matrix by a good choice of vector ?**



# Inspecting Linear Maps via Appplication

Linear Maps (i.e. Matrices) can be fully characterized by how they act on vectors

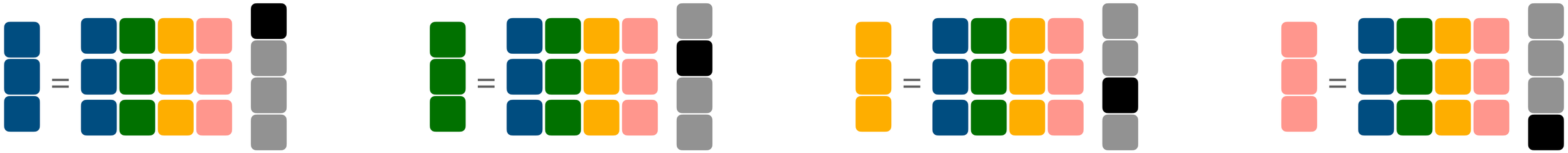
**Q: can we extract values of this matrix by a good choice of vector ?**



# Inspecting Linear Maps via Appplication

**Take-Away:** by computing Matrix-Vector Products (MVP) with basis vectors we can extact columns unknown linear map / matrix

- do not need the explicit matrix, just ability to compute MVPs
- to get full Jacobian we need to compute N matrix-vector products



# Inspecting Linear Maps via Appplication

**Gives us a new way to "store"/express matrics via computer programs instead of arrays of numbers in memory**

- **useful if matrix is sparse or regular (coding logic << enumeration)**
- **recover the array-picture by running program multiple times on all basis vectors**

$$\begin{bmatrix} 2 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

```python
def mvp(inp):
    x,y,z  = inp
    return np.array([
        2 * x + 3 * y,
        5*z
    ])
```

```
mvp([2.,3.,1.])
```

```
array([13.,  5.])
```

```python
def explicit(inp):
    matrix = np.array([
        [2,3,0],
        [0,0,5]
    ])
    return np.matmul(matrix,inp)
```

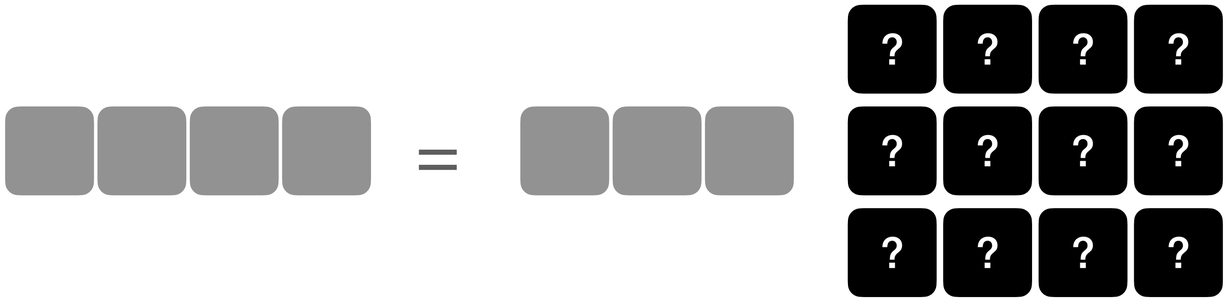
```
explicit([2.,3.,1.])
```

```
array([13.,  5.])
```

# Inspecting Linear Maps via Appplication

**If we change the order we can extract rows!**
**Vector-Matrix Products instead of Matrix-Vector Products**

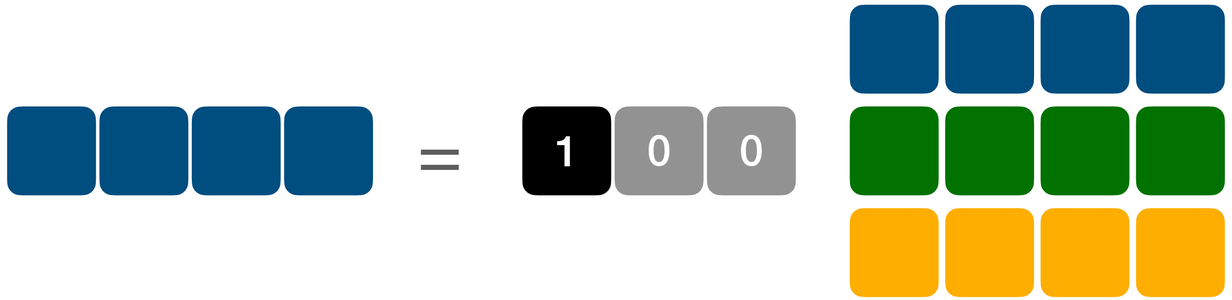
- do not need the explicit matrix, just ability to compute VMPs



# Inspecting Linear Maps via Appplication

**If we change the order to extract rows!**
**Vector-Matrix Products instead of Matrix-Vector Products**

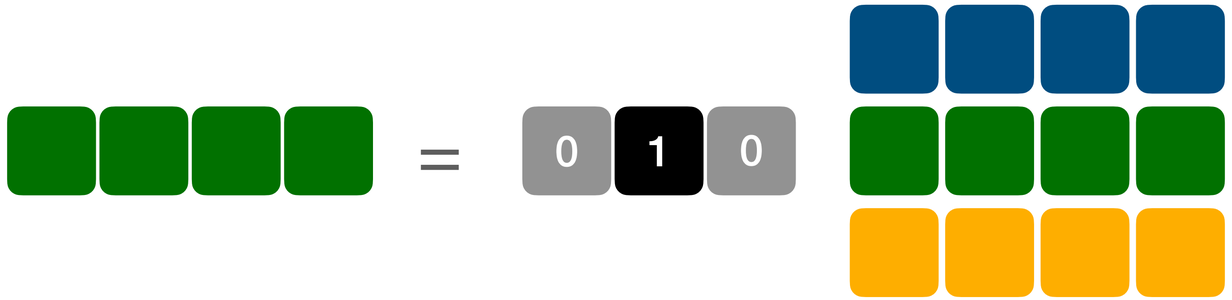
- do not need the explicit matrix, just ability to compute VMPs



# Inspecting Linear Maps via Appplication

**If we change the order to extract rows!**
**Vector-Matrix Products instead of Matrix-Vector Products**

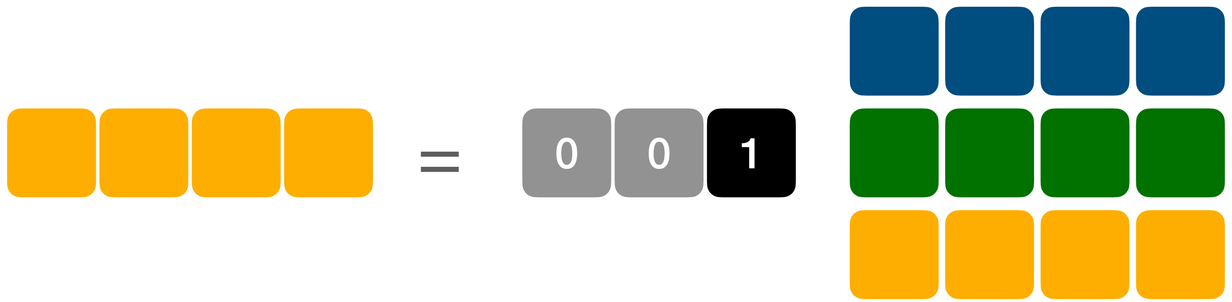
- do not need the explicit matrix, just ability to compute VMPs



# Inspecting Linear Maps via Appplication

**If we change the order to extract rows!**
**Vector-Matrix Products instead of Matrix-Vector Products**

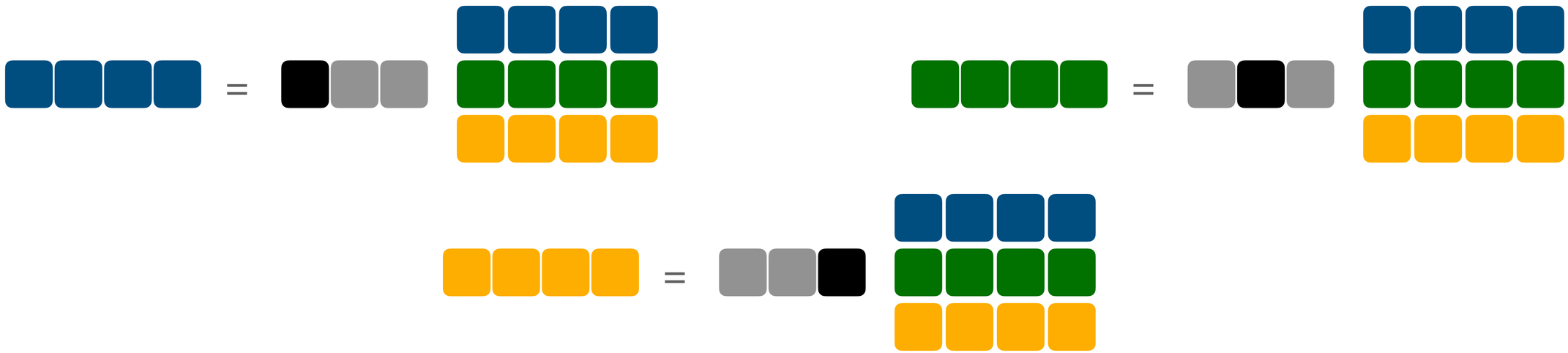
- do not need the explicit matrix, just ability to compute VMPs



# Inspecting Linear Maps via Appplication

**If we change the order to extract rows!**
**Vector-Matrix Products instead of Matrix-Vector Products**

- do not need the explicit matrix, just ability to compute VMPs



# Again as Programs

Again, savings if elements are easier/compactly expressed by logic than enumeration

$$\begin{bmatrix} 2 & 3 & 0 \\ 0 & 5 & 4 \end{bmatrix}$$

```
def vmp(out):
    a,b  = out
    return np.array([
        2*a,
        3*a + 5*b,
        4*b
    ])
```

```
vmp([2,3])
```

```
array([ 4, 21, 12])
```

```
def explicit(out):
    matrix = np.array([
       [2,3,0],
       [0,5,4]
    ])
    return np.matmul(np.array(out).T,matrix)
```

```
explicit([2,3])
```

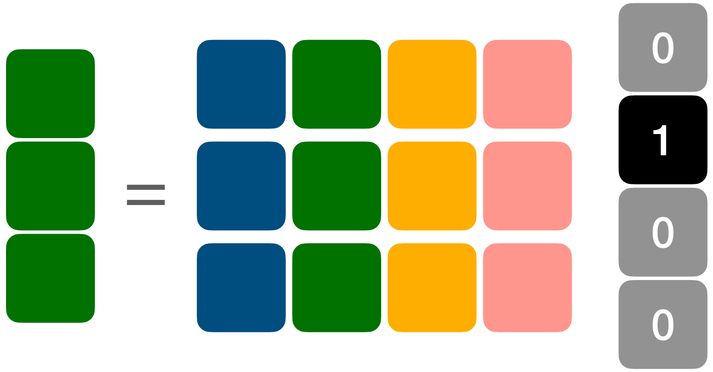
```
array([ 4, 21, 12])
```

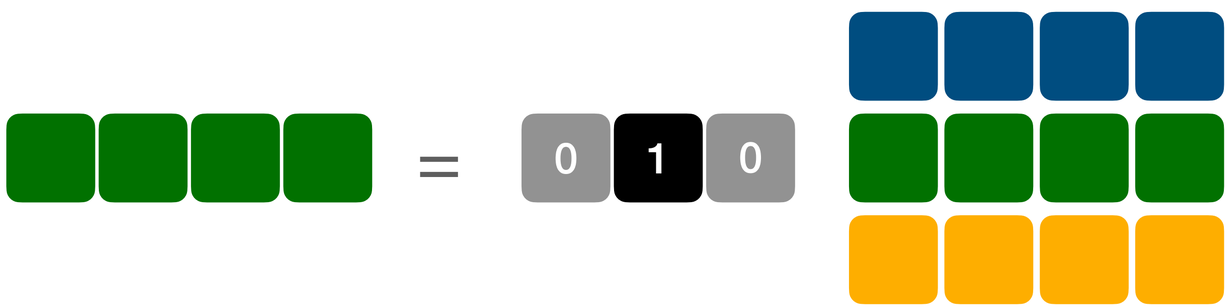
# Upshot: Row- or Columnwise Extraction

**We can fully characterize a Matrix through its products with vectors**

- Matrix-vector products extract columns (N times for full Matrix)
- vector-Matrix products extract rows (M times for full Matrix)

Gives us a new way to "store" a matrix: as a computer program (e.g. source code)  mapping vectors to vectors vs as array of numbers.

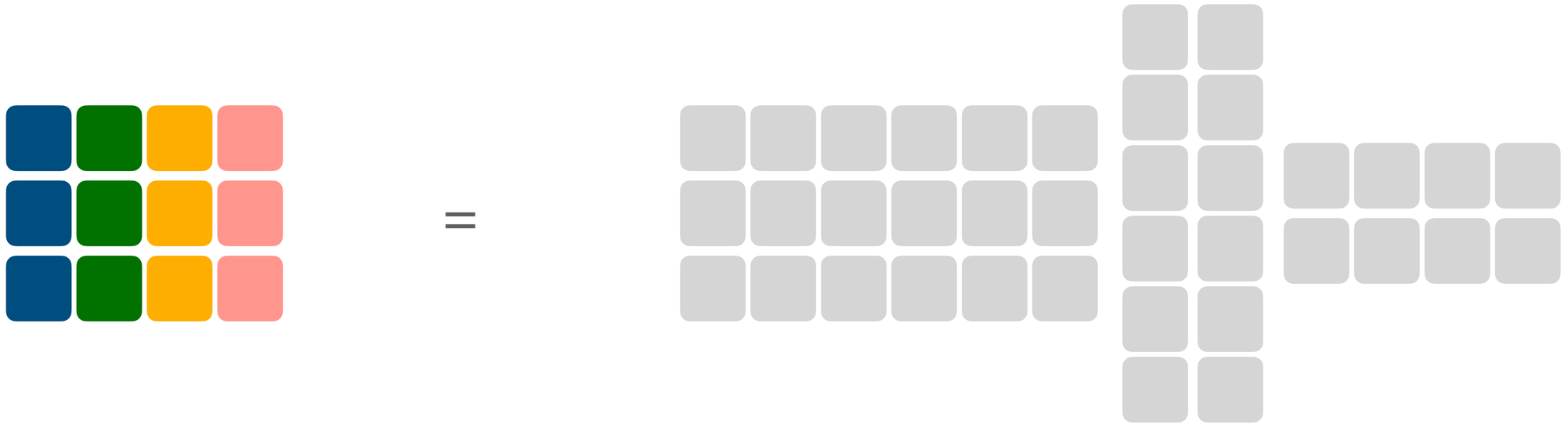


# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**

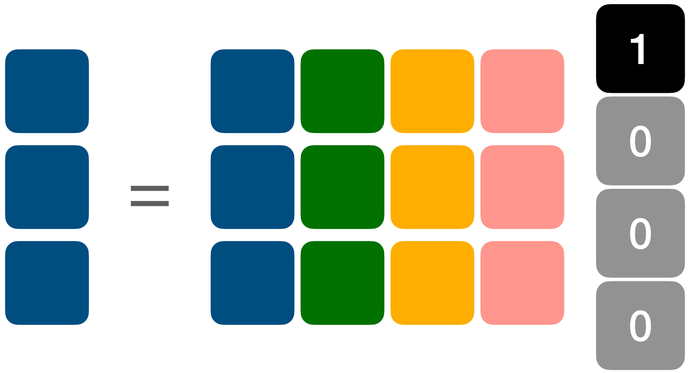


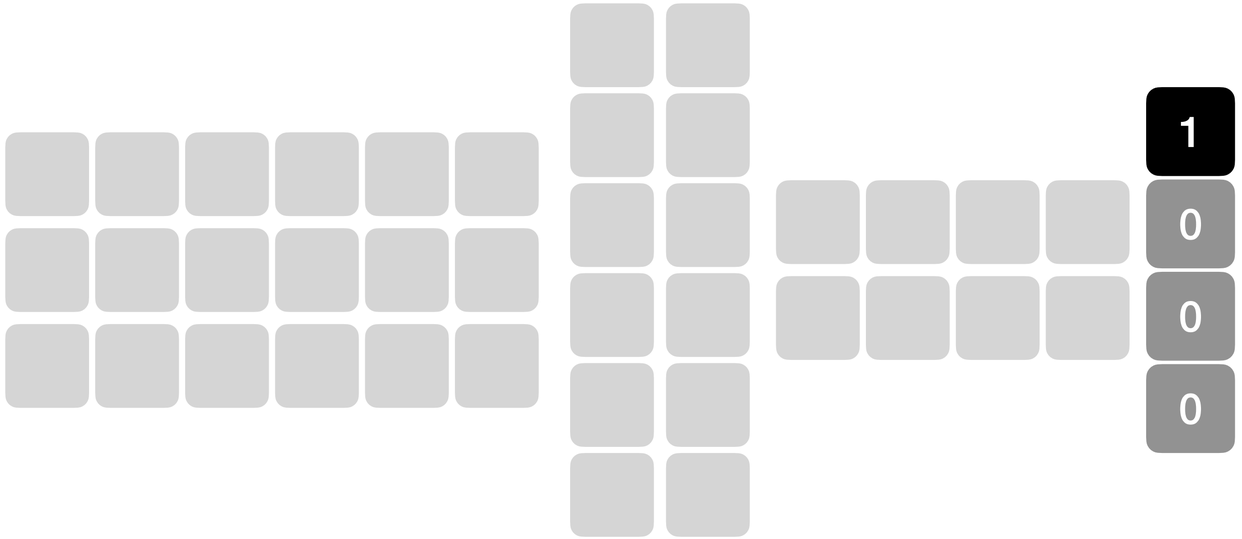
$M$ $M_3M_2M_1$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



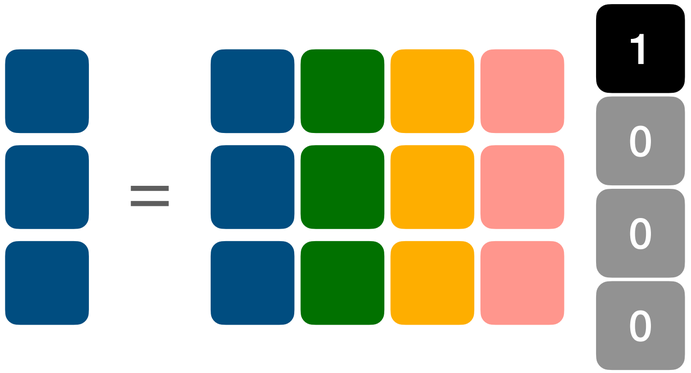

$$c_i = Me_i$$

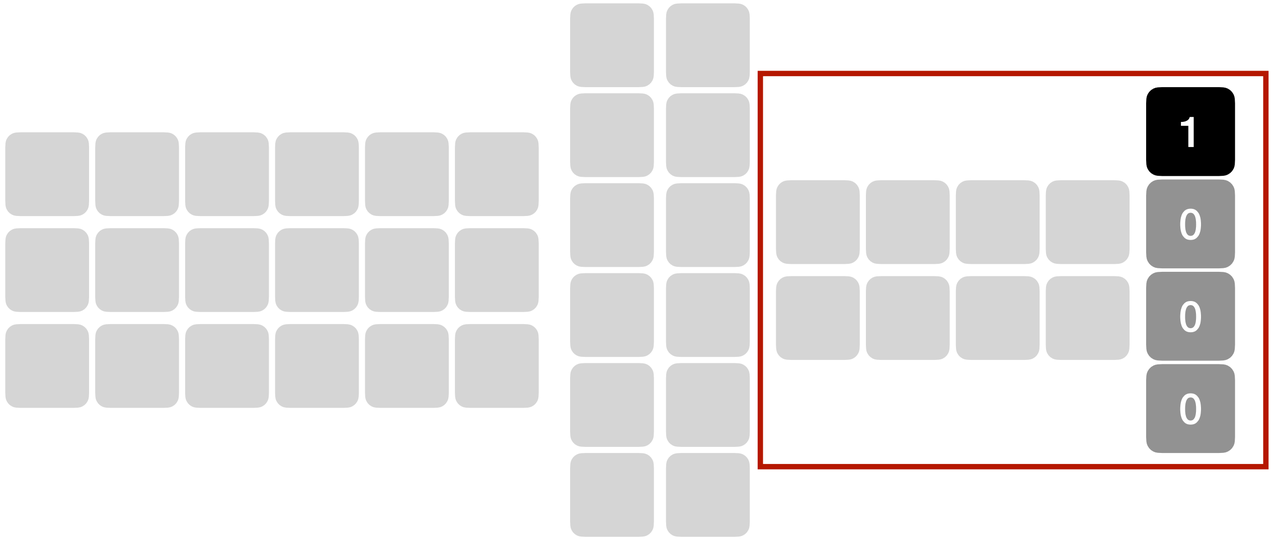
$$c_i = Me_i = M_3 M_2 M_1 e_i$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



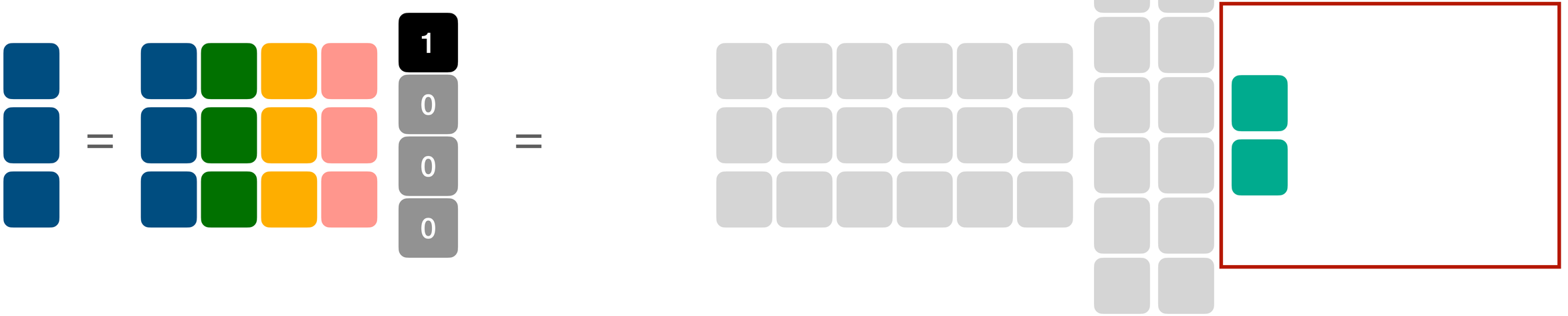

$$c_i = Me_i$$

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

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



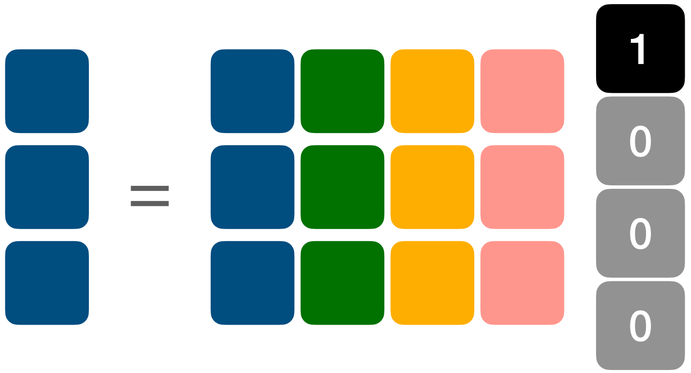

$$c_i = Me_i$$

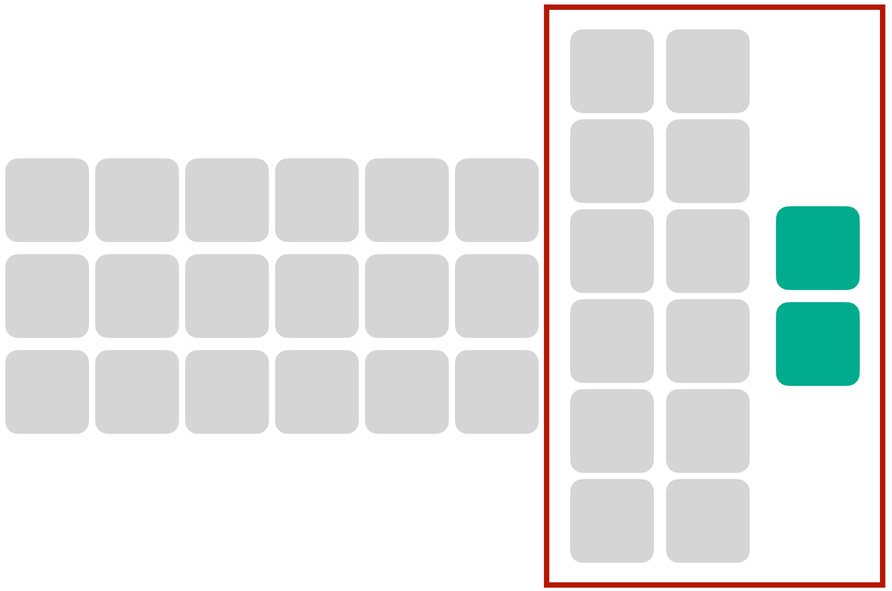
$$c_i = Me_i = M_3 M_2 v_1$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**




$$c_i = Me_i$$

$$c_i = Me_i = M_3 M_2 v_1$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**

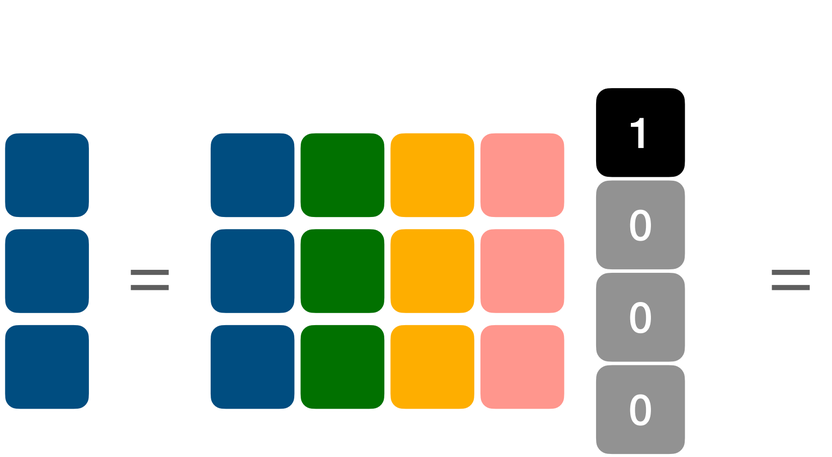

$$c_i = Me_i$$

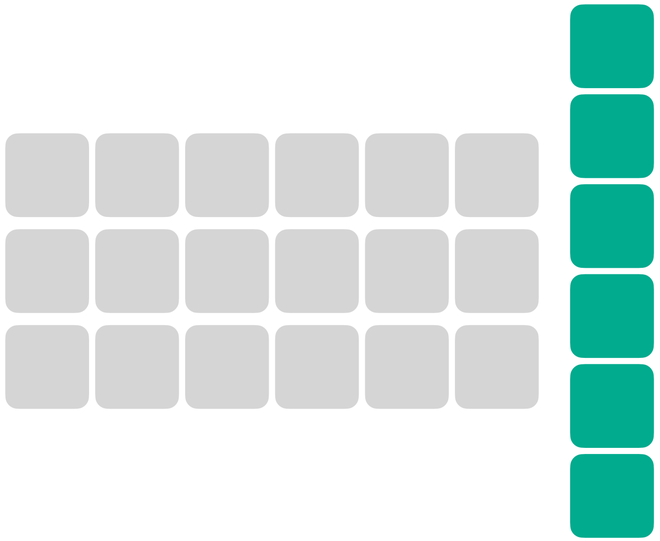
$$c_i = Me_i = M_3 v_2$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



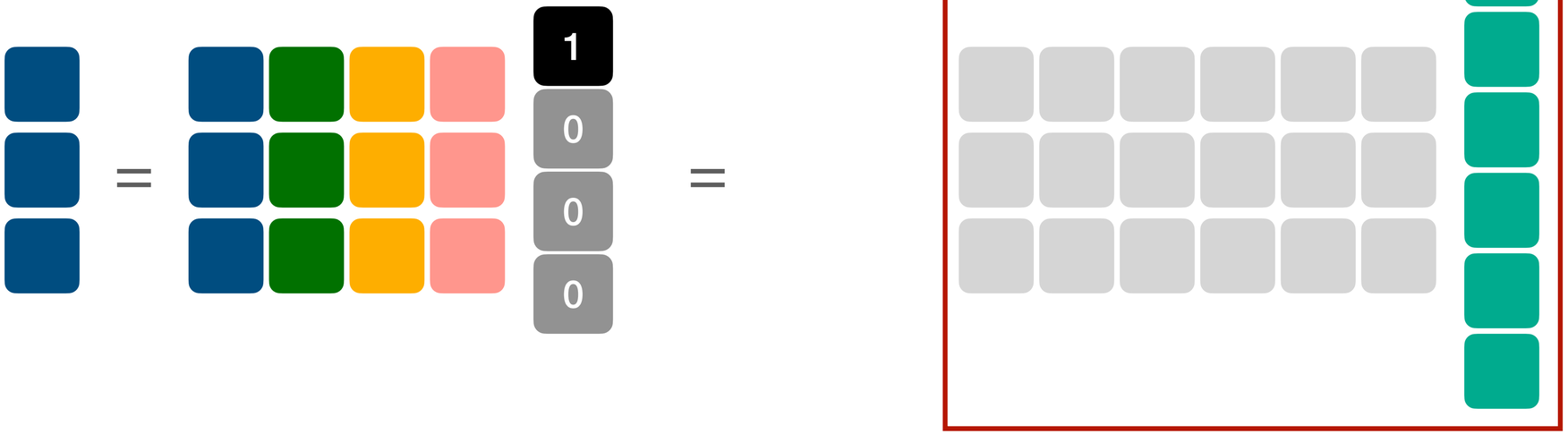

$$c_i = Me_i$$

$$c_i = Me_i = M_3 v_2$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



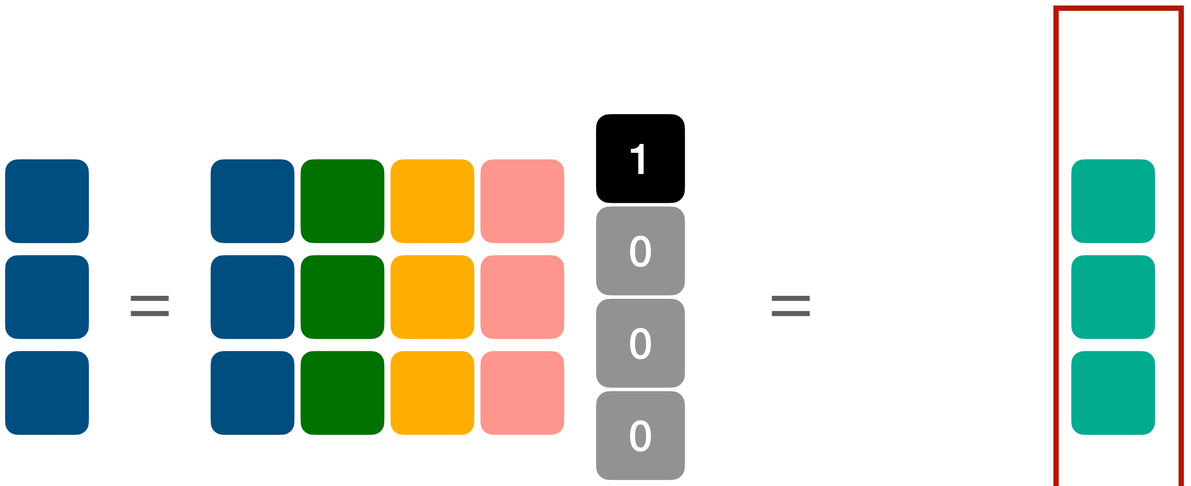

$$c_i = Me_i$$

$$c_i = Me_i = M_3 v_2$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



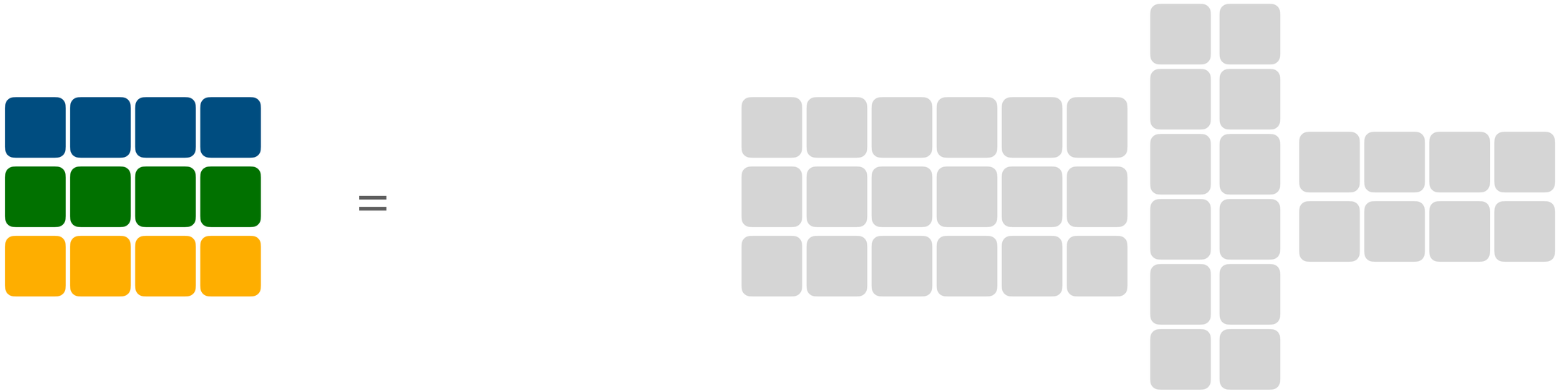

$$c_i = Me_i$$

$$c_i = Me_i = v_3$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



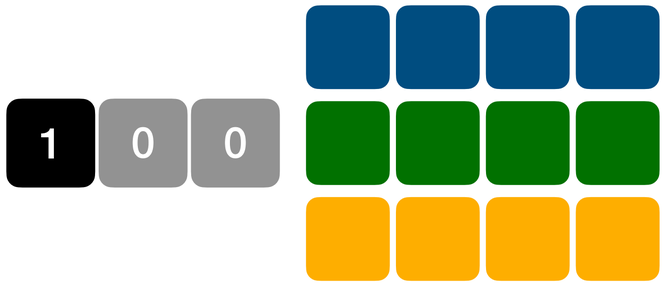
$M$

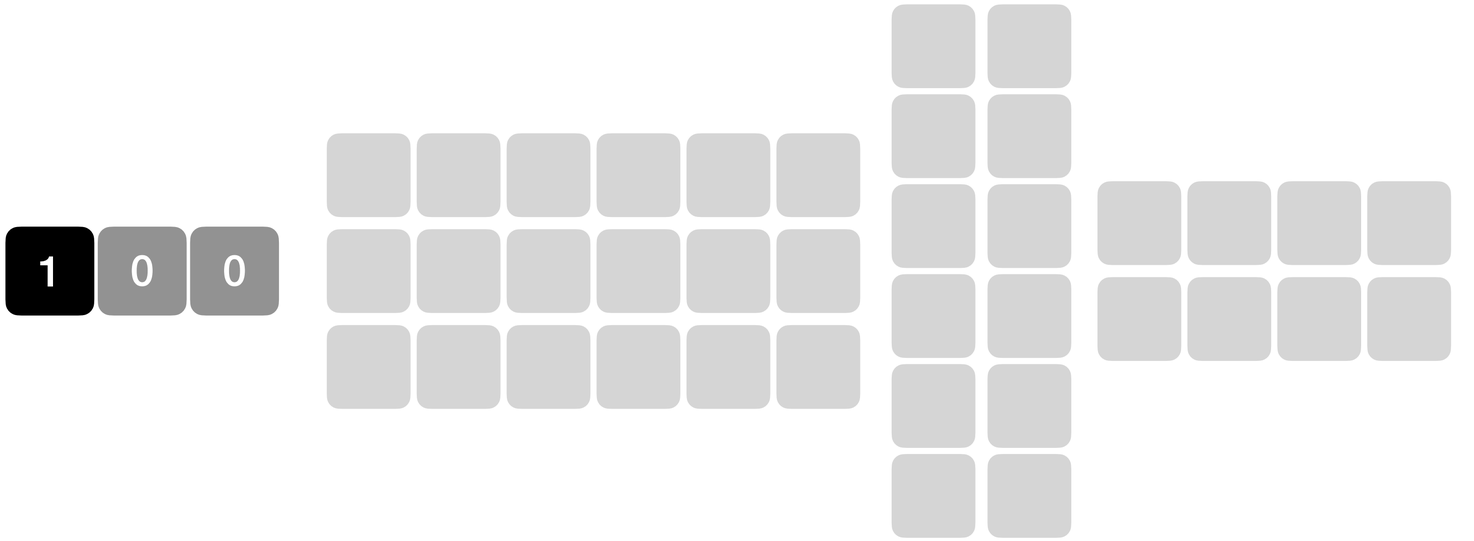
$M = M_1 M_2 M_3$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



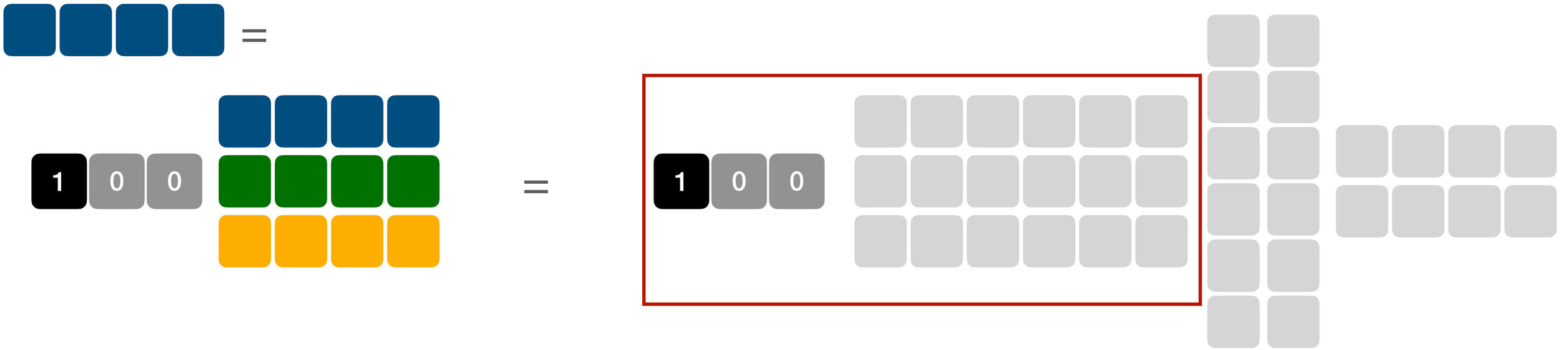

$$r_i = e_i^T M$$

$$r_i = e_i^T M = e_i^T M_3 M_2 M_1$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



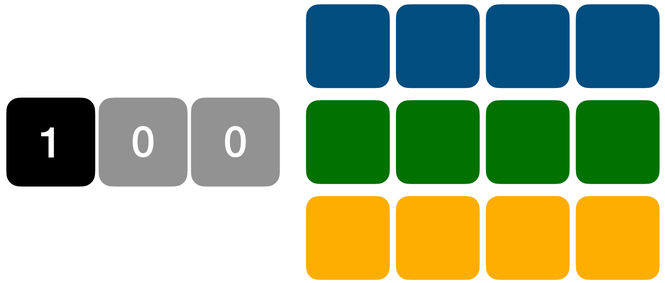

$$r_i = e_i^T M$$

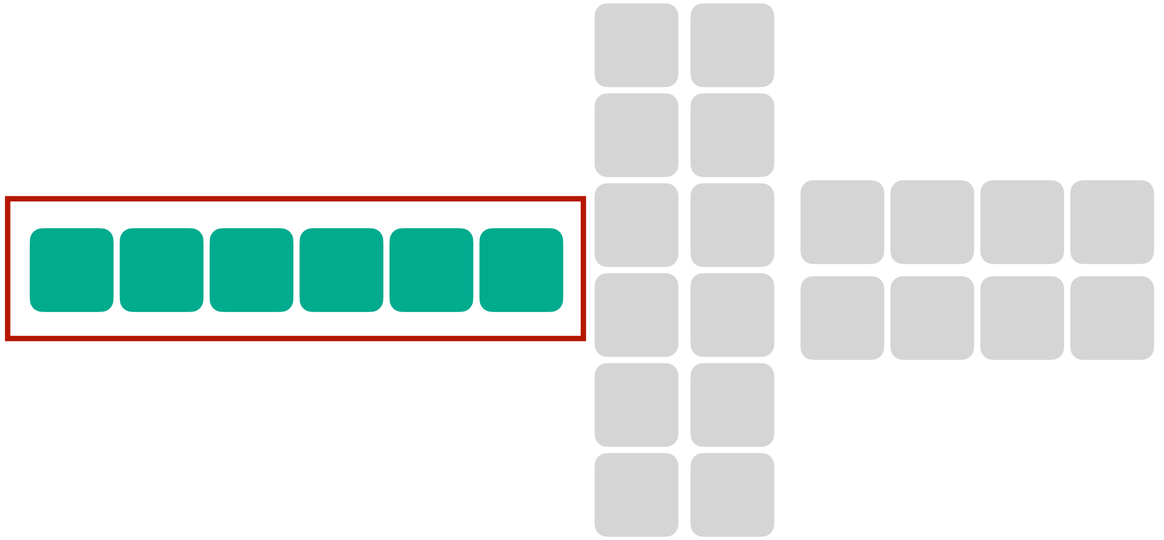
$$r_i = e_i^T M = e_i^T M_3 M_2 M_1$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



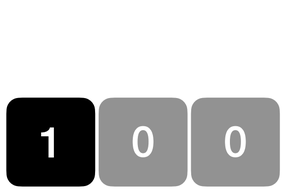
$$r_i = e_i^T M$$

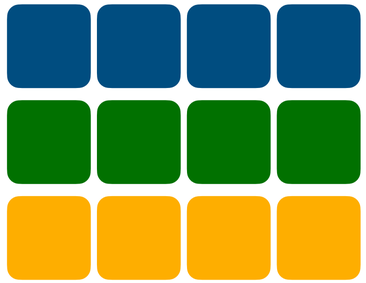
$$r_i = e_i^T M = \bar{v}_1 M_2 M_1$$

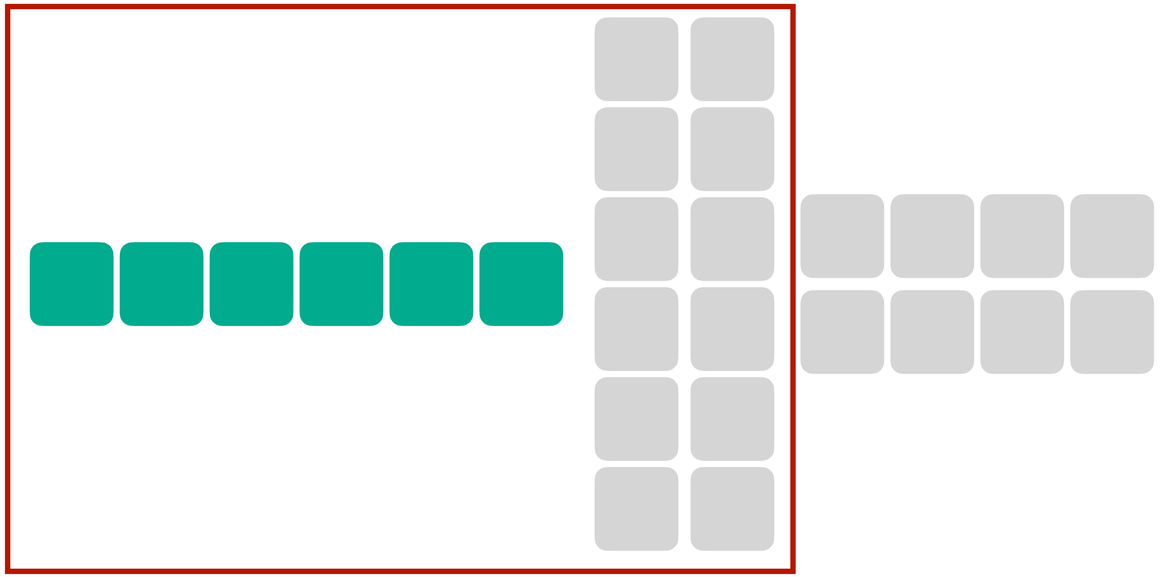
# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**





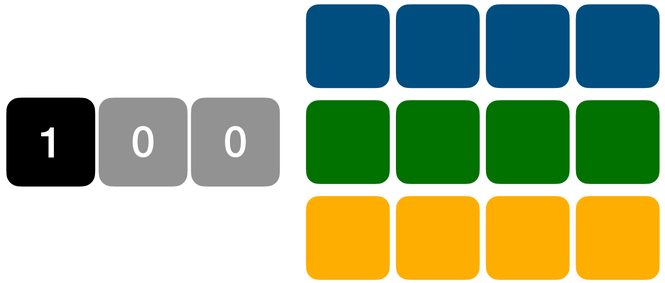
$$r_i = e_i^T M$$

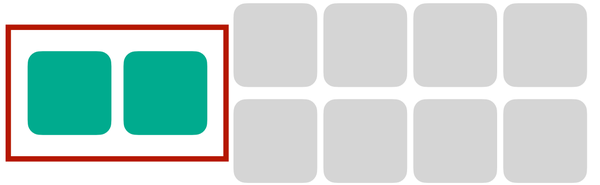
$$r_i = e_i^T M = \bar{v}_1 M_2 M_1$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



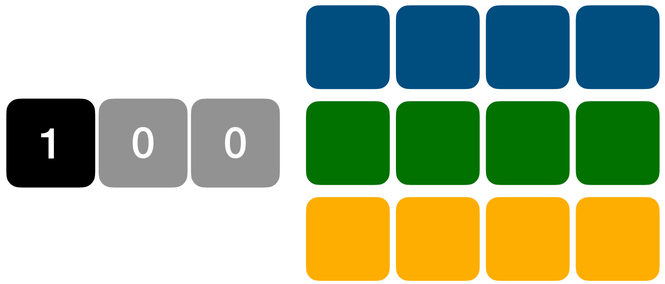
$$r_i = e_i^T M$$

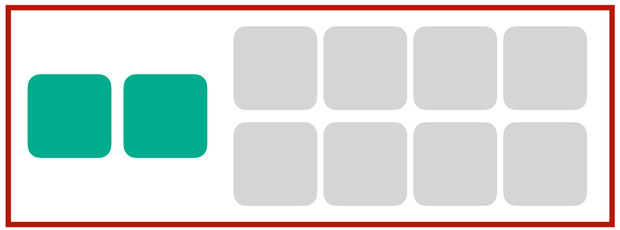
$$r_i = e_i^T M = \bar{v}_2 M_1$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**



$$r_i = e_i^T M$$

$$r_i = e_i^T M = \bar{v}_2 M_1$$

# Compositions

**Matrix-Vector/Vector Matrix Products allow us to characterize Matrix Composition without expensive Matrix multiplication**

- **just successive MVP/VMP until exhausted**


$$r_i = e_i^T M$$

$$r_i = e_i^T M = \bar{v}_3$$

# Upshot: Forward and Backward

MVPs/VMPs can characterize a Products of Marices efficiently
Depending on the type of product either go forwards or backwards

- to get a row/column we never need explicit representations of $M_i$.
  Ability to compute MVP/VMPs is all we need ("matrix-free approach)

$$c_i = Me_i = M_3M_2M_1e_i$$

$$c_i = Me_i = M_3M_2v_1$$

$$c_i = Me_i = M_3v_2$$

$$c_i = Me_i = v_3$$

forward

$$r_i = e_i^TM = e_i^TM_3M_2M_1$$

$$r_i = e_i^TM = \bar{v}_1M_2M_1$$

$$r_i = e_i^TM = \bar{v}_2M_1$$

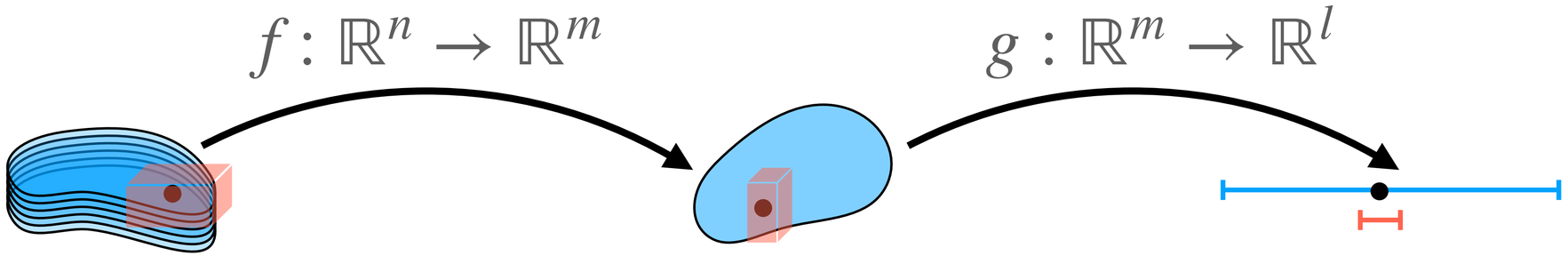
$$r_i = e_i^TM = \bar{v}_3$$

backward (or reverse)

# Back to Derivatives

From our discussion we now have a tool to efficiently compute Jacobian matrices of deep compositions of functions

- need only ability to compute Jacobian-vector products (JVP) or vector-Jacobian products (VJP)
- as in the Matrix-case: we can represent Jacobians as computer programs that map vectors to vectors




$$J_{g \circ f} = J_g J_f$$

# Forward and Backward Propagation

As in the Matrix-case, we can compute Jacobians in

- forward-mode (with Jacobian-Vector Products)
- reverse-mode (with Vector-Jacobian Products)

$$c_i = J_{k \circ h \circ g \circ f}\, e_i = J_k J_h J_g J_f\, e_i$$

$$r_i = e_i^T J_{k \circ h \circ g \circ f} \;\; = e_i^T J_k J_h J_g J_f$$

# Forward and Backward Propagation

As in the Matrix-case, we can compute Jacobians in

- forward-mode (with Jacobian-Vector Products)

$$c_i = J_{k \circ h \circ g \circ f}\ e_i = J_k J_h J_g J_f\ e_i \qquad (J_f v)_i = \sum_k J_{f\,ik} v_k = \sum_k \frac{\partial y_i}{\partial x_k} v_k$$

- reverse-mode (with Vector-Jacobian Products)
  - also known as "Backpropagation" in ML

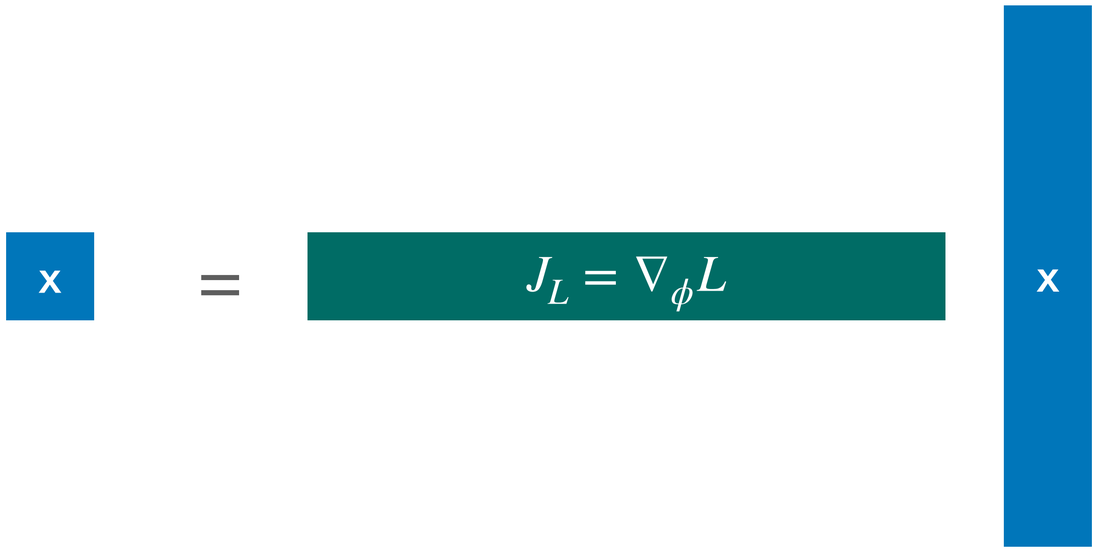
$$r_i = e_i^T J_{k \circ h \circ g \circ f} \ = e_i^T J_k J_h J_g J_f \qquad (\bar{v} J_f)_j = \sum_k \bar{v}_k J_{f\,kj} = \sum_k \bar{v}_k \frac{\partial y_k}{\partial x_j}$$

# Why Backpropagation for ML?

Neural Net Loss functions map network parameters to losses

$$L : \mathbb{R}^N \to \mathbb{R}$$

Shape of the Jacobian: a single row! (i.e. the gradient $\nabla_\phi L$)



# Example

$$f : \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} xy \\ y^3 \end{bmatrix} \qquad J_f = \begin{bmatrix} \partial_x(xy) & \partial_y(xy) \\ \partial_x(y^3) & \partial_y(y^3) \end{bmatrix}_{x=x_0, y=y_0} = \begin{bmatrix} y & x \\ 0 & 3y^2 \end{bmatrix}_{x=x_0, y=y_0}$$

```python
import numpy as np
def func(inp):
    x,y = inp
    return np.array([
        x*y,
        y**3
    ])
```

```python
def explicit(v, at_point):
    x,y = at_point
    jacobian = np.array([
        [y, x],
        [0, 3*y**2]
    ])
    return np.matmul(jacobian,v)
```

```python
def jvp(v, at_point):
    v1,v2 = v
    x,y = at_point

    return np.array([
        y*v1 + x*v2,
        3*y**2 * v2
    ])
```

Note: JVP program depends on the point where the derivative is taken

```python
explicit([1.2,3.4], at_point = [2,3])
```

```
array([10.4, 91.8])
```

```python
jvp([1.2, 3.4],at_point = [2,3])
```

```
array([10.4, 91.8])
```

# Composition

The JVP/VJP programs must be generated as you step through the composition (b/c of position dependence of Jacobian at each step)

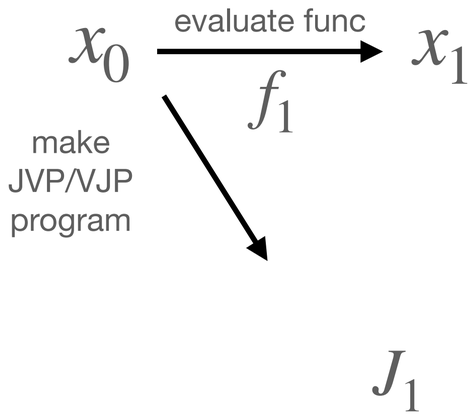
$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$

$x_0$

# Composition

**The JVP/VJP programs must be generated as you step through the composition (b/c of position dependence of Jacobian at each step)**

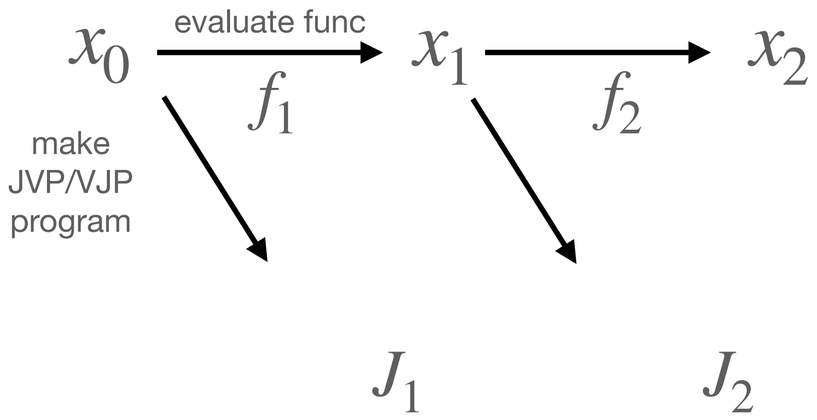
$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



# Composition

**The JVP/VJP programs must be generated as you step through the composition (b/c of position dependence of Jacobian at each step)**

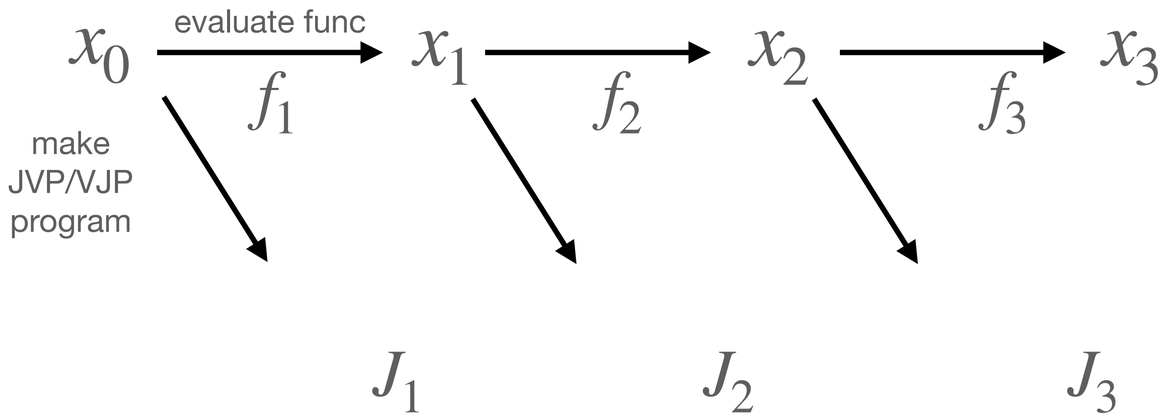
$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



# Composition

**The JVP/VJP programs must be generated as you step through the composition (b/c of position dependence of Jacobian at each step)**

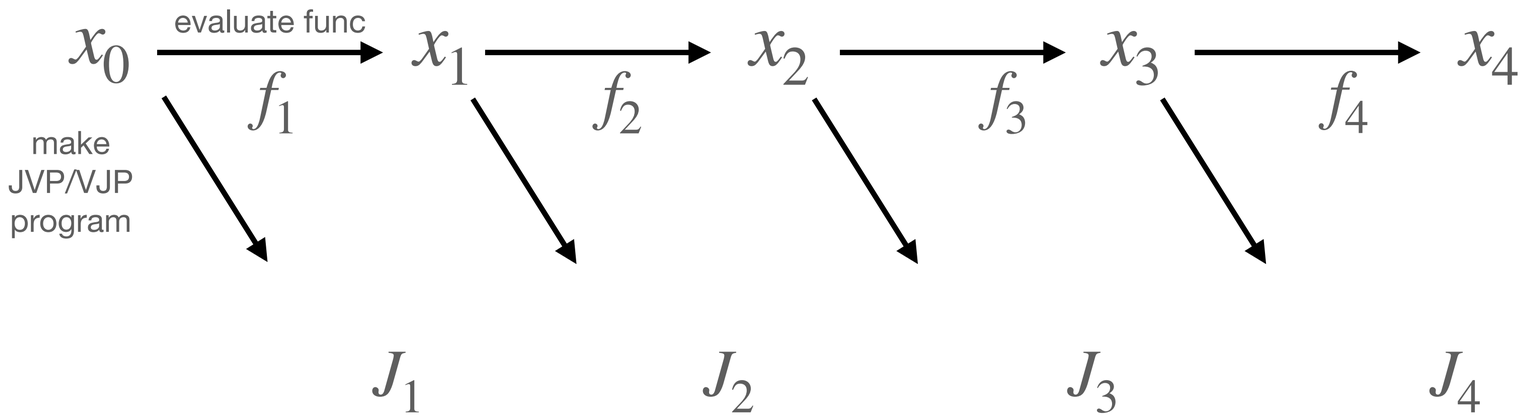
$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



# Composition

**The JVP/VJP programs must be generated as you step through the composition (b/c of position dependence of Jacobian at each step)**

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$

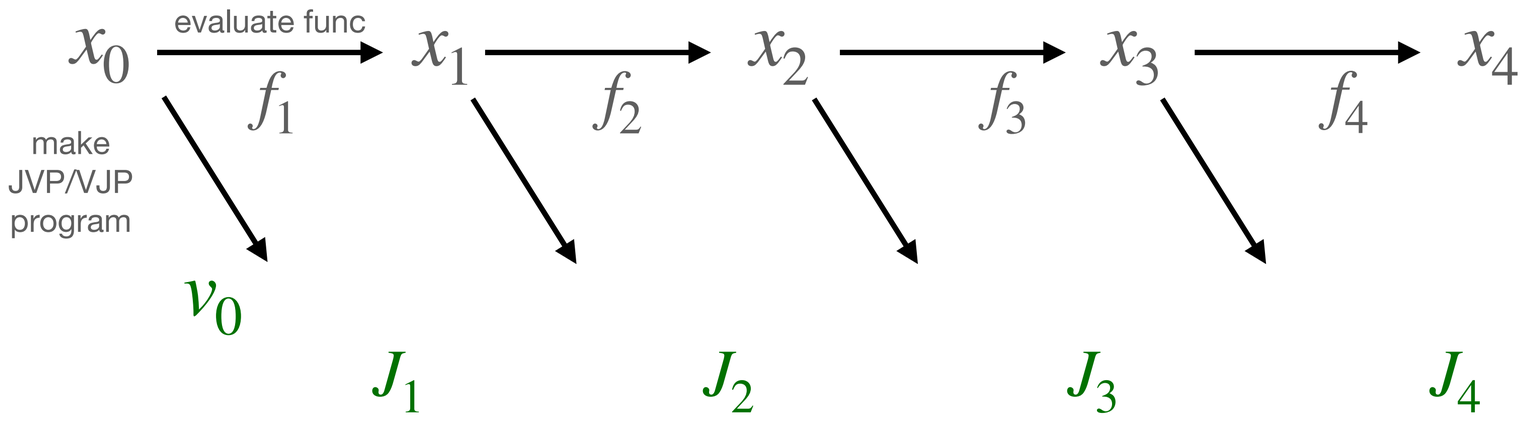


# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward is in the same order as original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



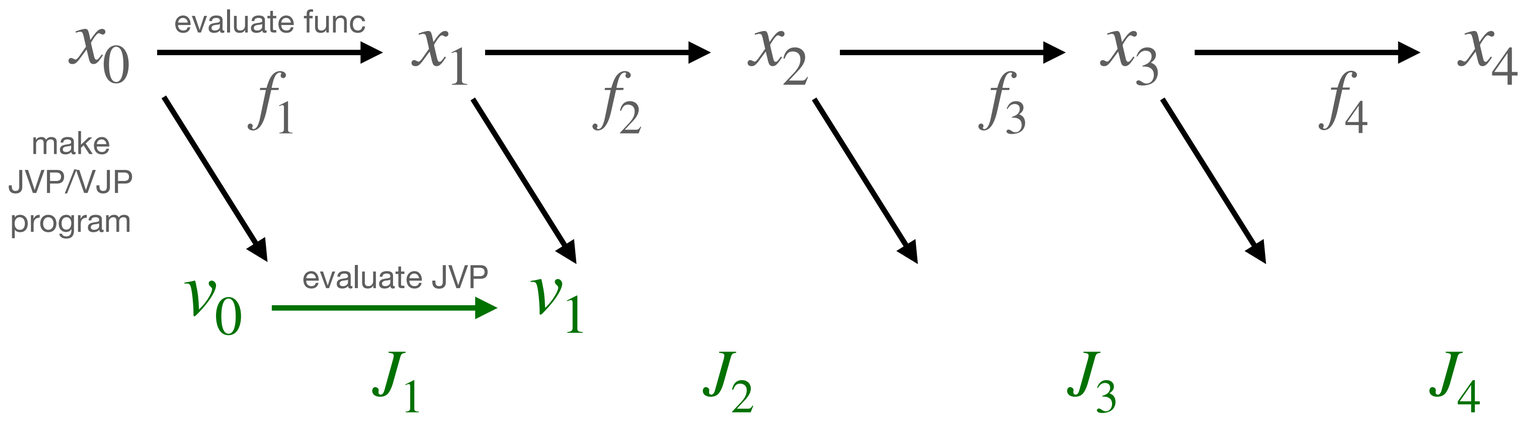

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward is in the same order as original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



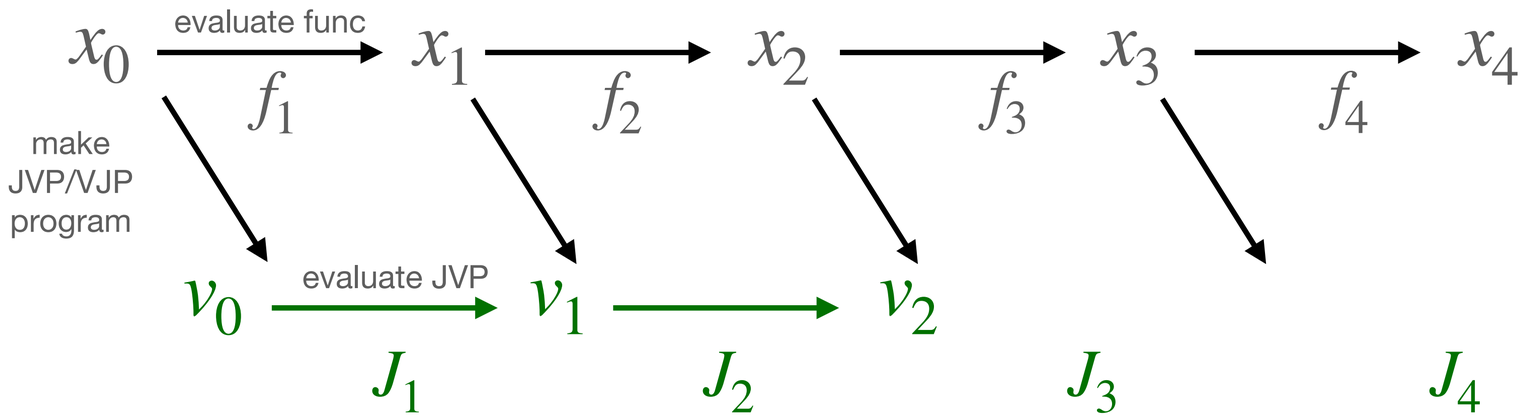

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward is in the same order as original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



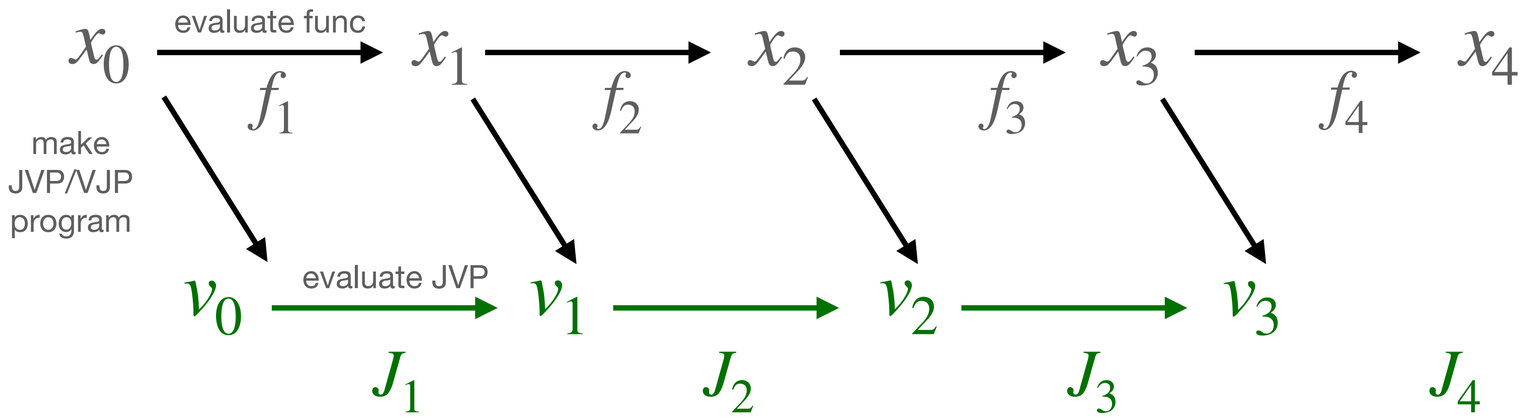

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward is in the same order as original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



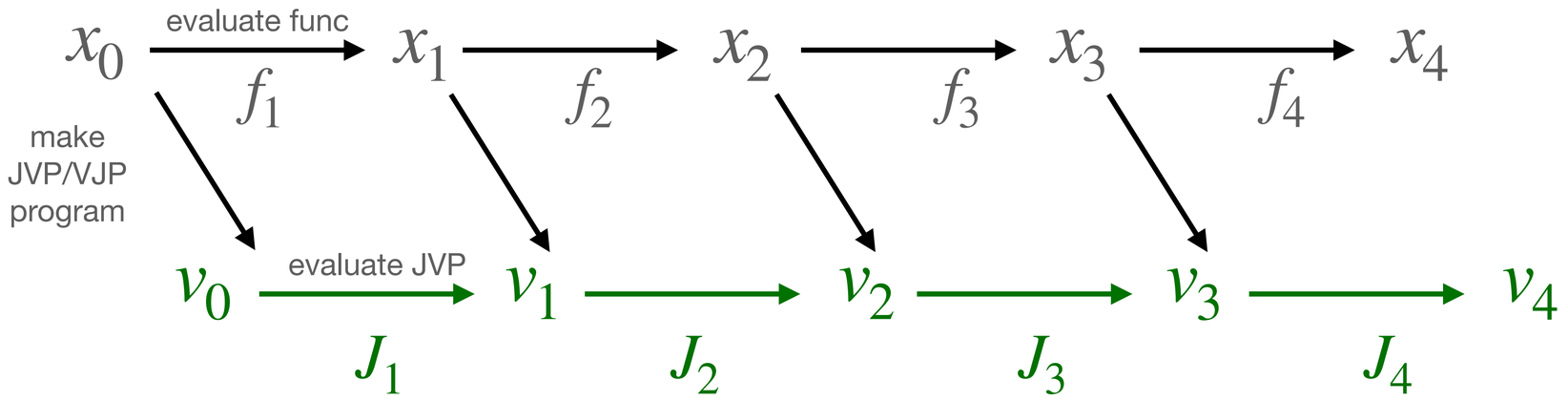

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward is in the same order as original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



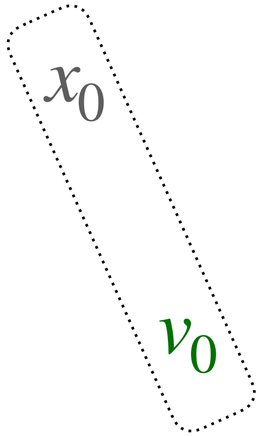

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward can be done "on-the-fly". The $J_i$ become available as-you-go

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



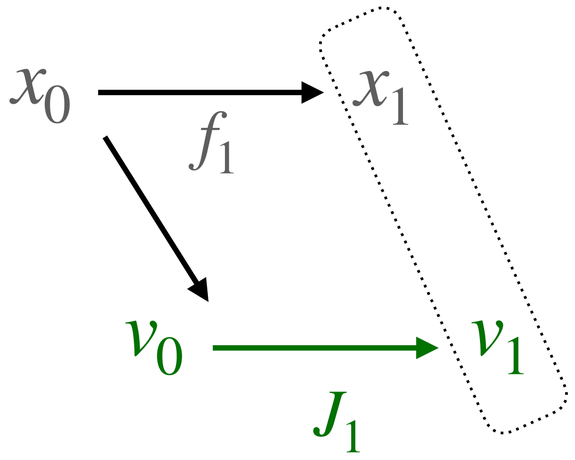

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward can be done "on-the-fly". The $J_i$ become available as-you-go

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



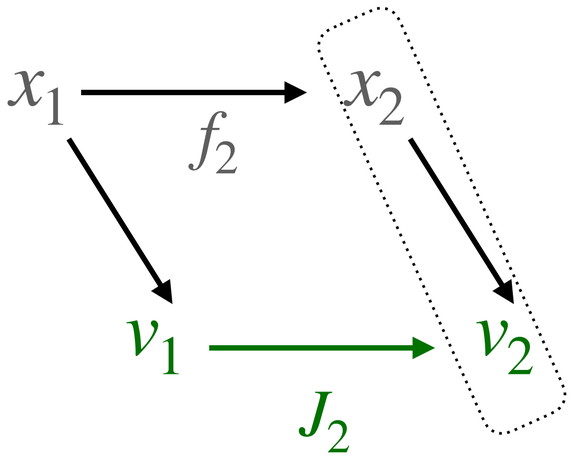

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward can be done "on-the-fly". The $J_i$ become available as-you-go

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



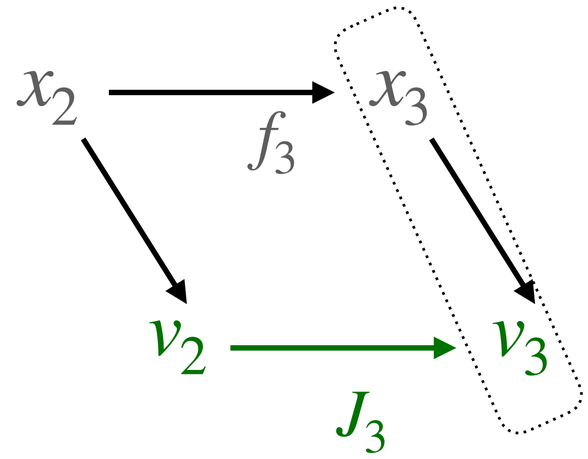

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward can be done "on-the-fly". The $J_i$ become available as-you-go

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



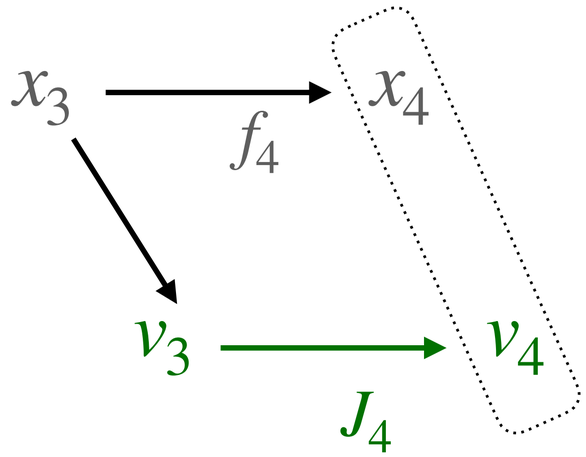

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Forward can be done "on-the-fly". The $J_i$ become available as-you-go

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



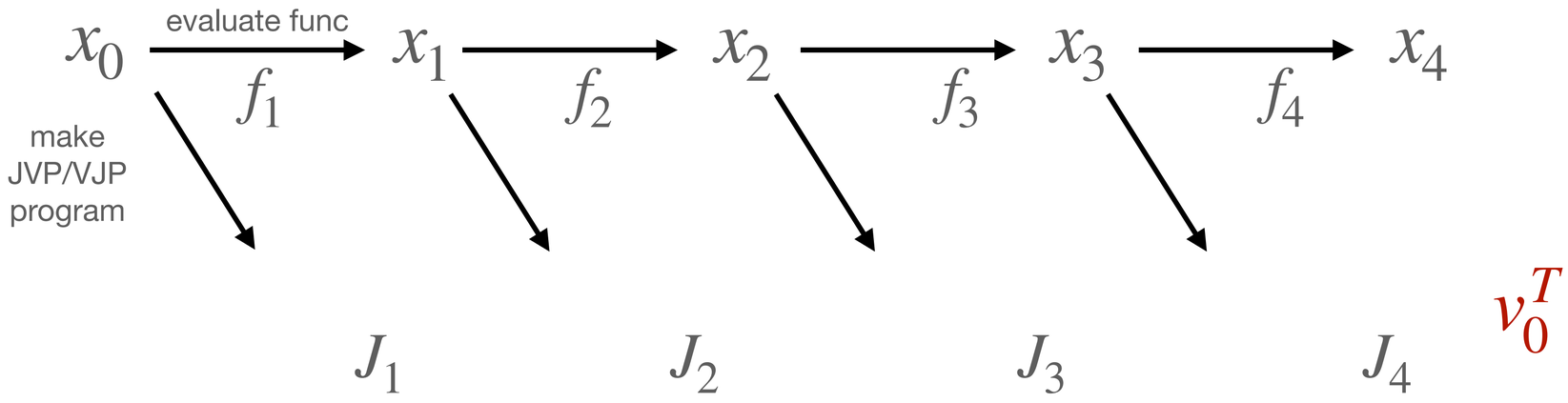

$$c = Jv_0 = J_4 J_3 J_2 J_1 v_0$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Backward is in the rerverse order from original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



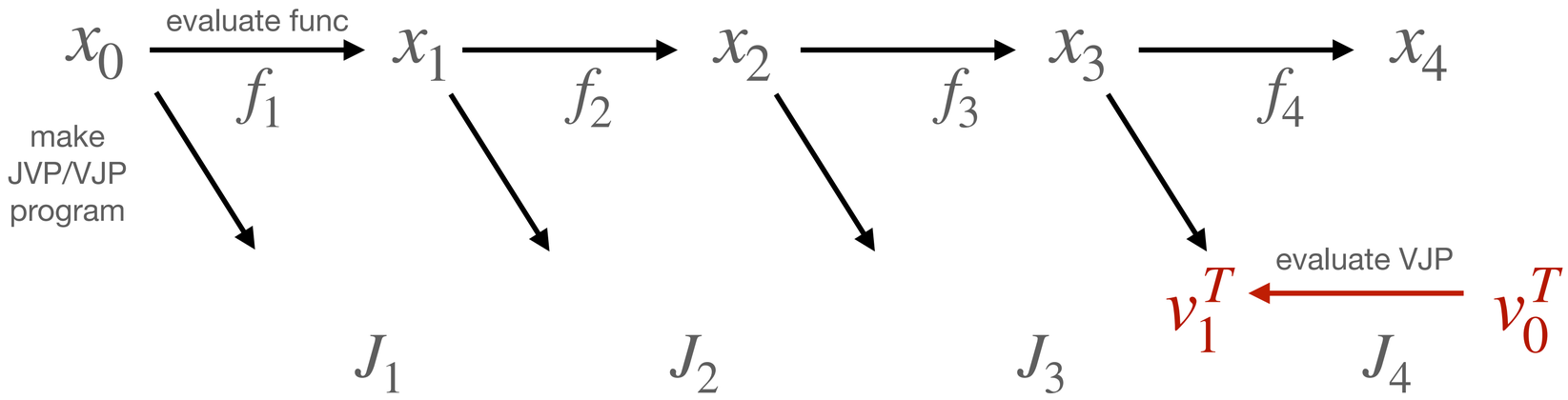

$$c = v_0^T J = v_0^T J_4 J_3 J_2 J_1$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Backward is in the rerverse order from original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



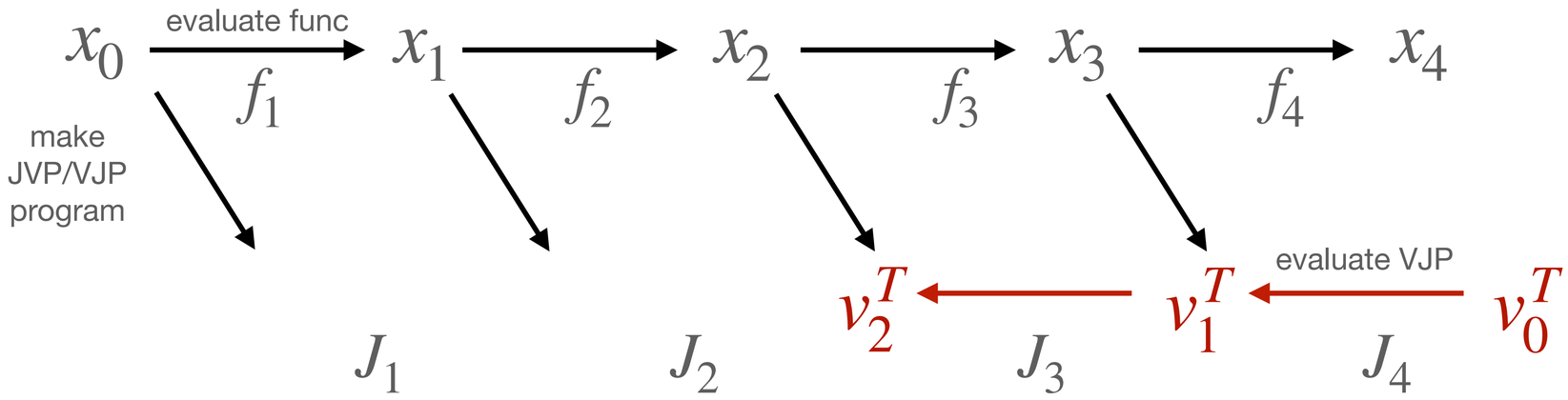

$$c = v_0^T J = v_0^T J_4 J_3 J_2 J_1$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Backward is in the rerverse order from original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



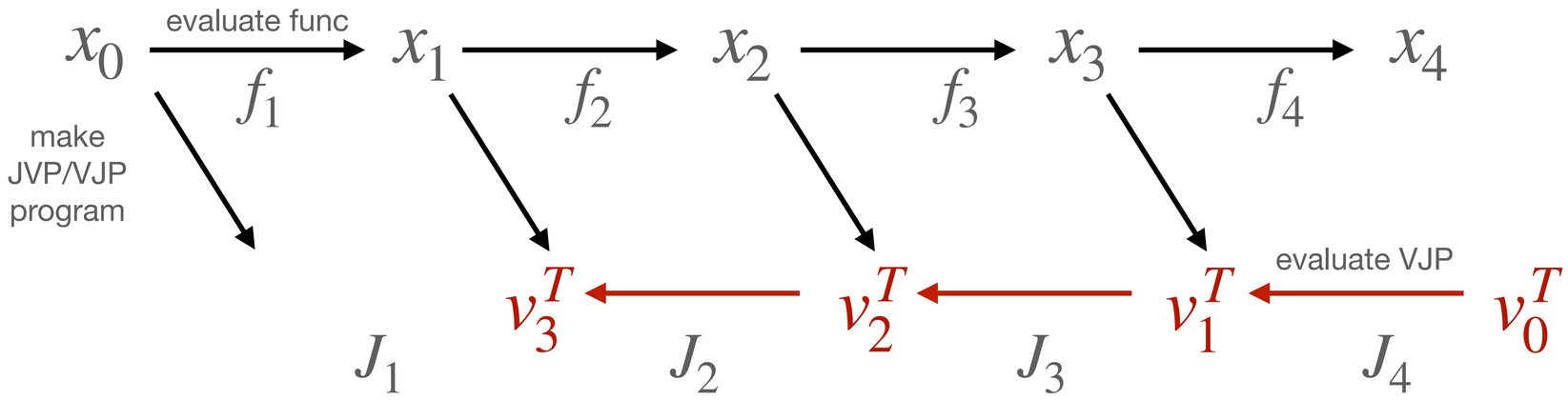

$$c = v_0^T J = v_0^T J_4 J_3 J_2 J_1$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Backward is in the rerverse order from original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



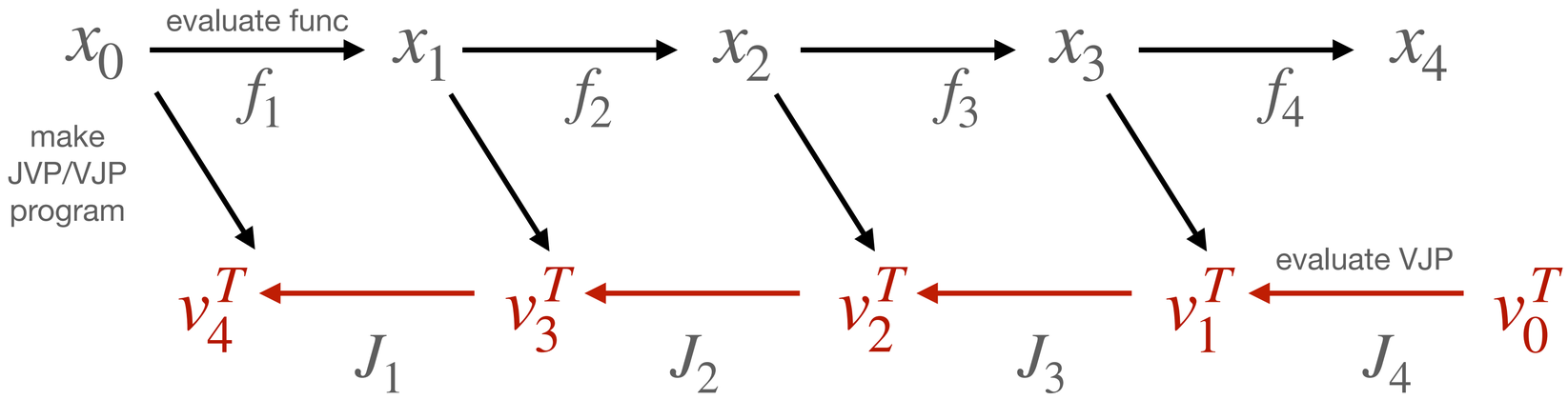

$$c = v_0^T J = v_0^T J_4 J_3 J_2 J_1$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Backward is in the rerverse order from original composition

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



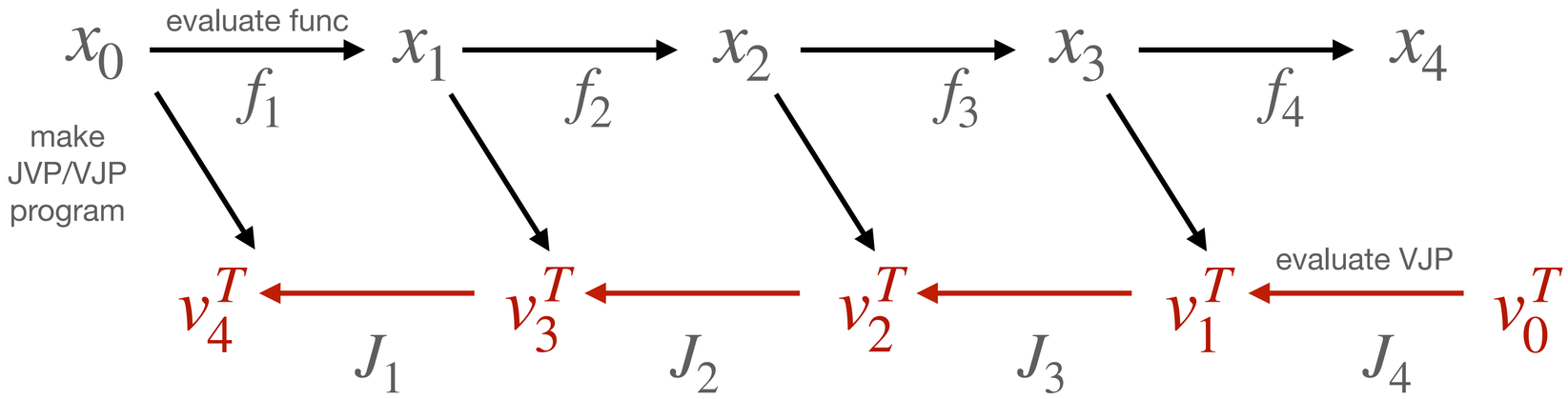

$$c = v_0^T J = v_0^T J_4 J_3 J_2 J_1$$

# Composition

Once you have the JVP programs you can evaluate the JVP/VJPs

Backward cannot be done "on-the-fly", needs all $J_i$ before starting

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$



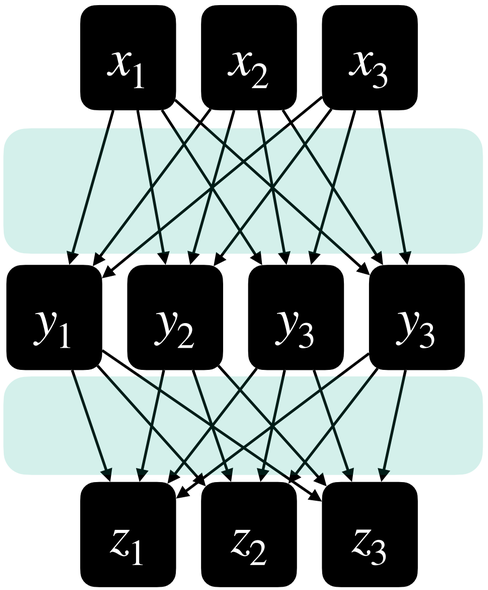

$$c = v_0^T J = v_0^T J_4 J_3 J_2 J_1$$

# The Graph Picture

Computation are naturally expressed as graphs.

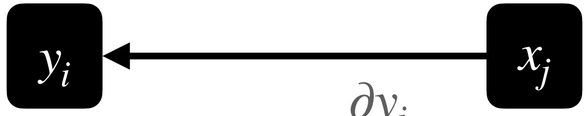
$y = f(x)$

$z = g(y)$




$(g \circ f)(x)$

- edges represent a data dependence
- correspond to Jacobian matrix element




$$J_{ij} = \frac{\partial y_i}{\partial x_j}$$

- Matrix Multiplication: summation over edges.

$$(J_f v)_i = \sum_k J_{f\,ik} v_k \qquad (\bar{v} J_f)_j = \sum_k \bar{v}_k J_{f\,kj}$$

(generalizes beyond "feed-forward" graphs)

# Upshot: Jacobians as Programs

- Since Jacobians are Matrices we can use our tools to express rows & columns of them as programs (JVP, VJP)

- Jacobians of deep compositions are easy to compute without ever explicitly calculating all matrix elements once we have these Jacobian Programs for the individual functions being composed

  - Corollary: based on a small set  building blocks (where we manually code JVP, VJP) we can compute Jacobians (i.e. derivatives) automatically for an almost unlimited set of functions (all the ways the building blocks can be built

# Upshot: Jacobians as Programs

- Since Jacobians are Matrices we can use our tools to express rows & columns of them as programs (JVP, VJP)

- Jacobia[illegible] without ever ex[illegible] have these Jacobian Programs for the individual functions being composed

  - Corollary: based on a small set  building blocks (where we manually code JVP, VJP) we can compute Jacobians (i.e. derivatives) automatically for an almost unlimited set of functions (all the ways the building blocks can be built

Automatic Differentiation

# Automatic Differentiation Systems

Systems that allow you to write numerical programs: $\mathbb{R}^n \to \mathbb{R}^m$
(i.e. complex compositios of basic building blocks),
that are efficietly differentiable

They do it by:

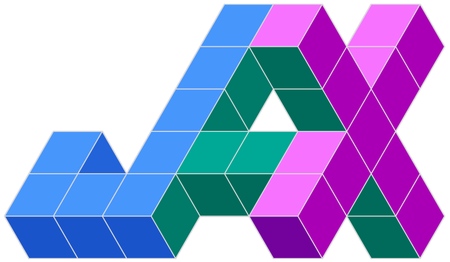
- implementing `(+,-,*,/,√,^,exp,log,sin,cos,tan,...)`
- JVP/VJP for basic operations
- automating for you the composition for running either forward or backward propagation

# Automatic Differentiation Systems

Most Deep Learning Framework are at their core Autodiff systems

- I'll focus on Jax, since it's more elegeant from a AD perspective

TensorFlow



PYTORCH

# Beyond Deep Learning

But there is a long list of non-DL focused AD frameworks as well

- idea exist in many language (C++, Julia, Fortran, ...)


autodiff (C++)

Enzyme.jl (Julia)

# Example: Autodiff with Jax

```
import numpy as np
def func(inp):
    x,y = inp
    return np.array([
        x*y,
        y**3
    ])
```

```
def jvp(inp, at_point):
    v1,v2 = inp
    x,y = at_point
    return np.array([
        y*v1 + x*v2,
        3*y**2 * v2
    ])
```

```
func(np.array([2.,3.]))
```

```
array([ 6., 27.])
```

```
jvp(np.array([0.,1.]),np.array([2.,3.]))
```

```
array([ 2., 27.])
```

Manual

```
import jax.numpy as jnp
def func(inp):
    x,y = inp
    return jnp.array([
        x*y,
        y**3
    ])
```

```
import jax
jax.jvp(func,
        (jnp.array([2.,3.]),),
        (jnp.array([0.,1.]),),
)
```

```
(DeviceArray([ 6., 27.], dtype=float32),
 DeviceArray([ 2., 27.], dtype=float32))
```

Automatic

# Example: Autodiff with Jax

```python
import numpy as np
def func(inp):
    x,y = inp
    return np.array([
        x*y,
        y**3
    ])
```

```python
def vjp(out, at_point):
    v1,v2 = out
    x,y = at_point
    return np.array([
        v1*y,
        v1*x + v2*3*y**2,
    ])
```

```python
func([2.,3.])
```

```
array([ 6., 27.])
```

```python
vjp([4.,5.],[2.,3.])
```

```
array([ 12., 143.])
```

Manual

```python
import jax.numpy as jnp
def func(inp):
    x,y = inp
    return jnp.array([
        x*y,
        y**3
    ])
```

```python
import jax
value, backward = jax.vjp(func,jnp.array([2.,3.]))
value
```

```
DeviceArray([ 6., 27.], dtype=float32)
```

```python
backward(jnp.array([4.,5.]))
```

```
(DeviceArray([ 12., 143.], dtype=float32),)
```

Automatic

# Higher-level APIs

As a standard user you care about the derivatives/Jacobians.

Autodiff frameworks give you nice wrappers.

Thinking in terms of JVP/VJP is not necessary for day-to-day use (but useful to understand once)

```
import numpy as np
import matplotlib.pyplot as plt

import jax
import jax.numpy as jnp
```

Getting the derivative for an arbitrary python function with a single line

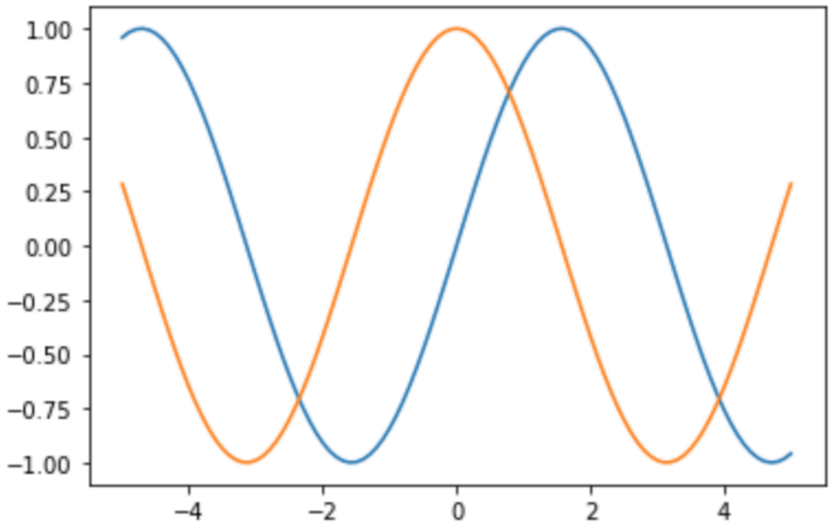
```
def func(x):
    return jnp.sin(x)

derivative = jax.grad(func)
```

```
xspan = np.linspace(-5,5,101)
y = jnp.array([func(x) for x in xspan])
d = jnp.array([derivative(x) for x in xspan])
```

```
plt.plot(xspan,y)
plt.plot(xspan,d)
```

```
[<matplotlib.lines.Line2D at 0x7f7874acab10>]
```



# Higher-level APIs

## With autodiff you can not only get first-order derivatives

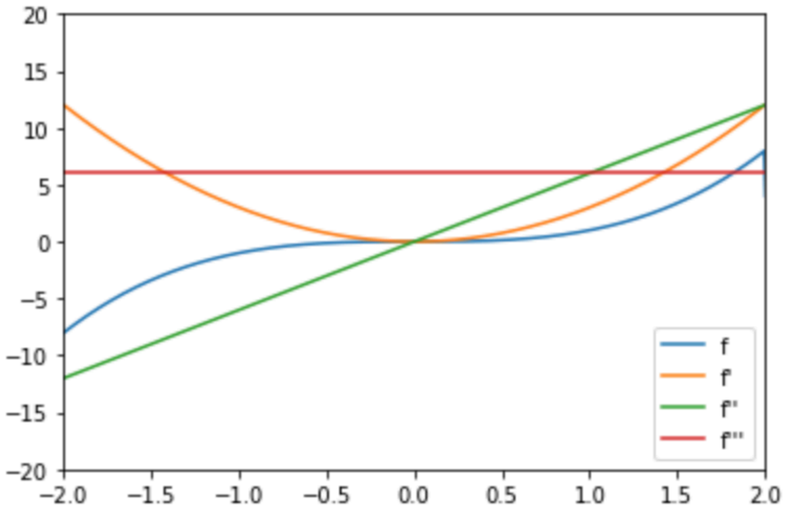
```
def f(x):
    return x**3
```

```
print(f(4.0))
print(jax.grad(f)(4.0)) #boom!
print(jax.grad(jax.grad(f))(4.0)) #boom!
print(jax.grad(jax.grad(jax.grad(f)))(4.0)) #boom!
print(jax.grad(jax.grad(jax.grad(jax.grad(f))))(4.0)) #boom!
```

```
64.0
48.0
24.0
6.0
0.0
```

```
g1i = jax.vmap(jax.grad(f))(xi)
g2i = jax.vmap(jax.grad(jax.grad(f)))(xi)
g3i = jax.vmap(jax.grad(jax.grad(jax.grad(f))))(xi)
plt.plot(xi,yi,  label = "f")
plt.plot(xi,g1i, label = "f'")
plt.plot(xi,g2i, label = "f''")
plt.plot(xi,g3i, label = "f'''")
plt.xlim(-2,2)
plt.ylim(-20,20)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f0aecabd910>
```



# Higher-level APIs

## With autodiff you can not only get first-order derivatives

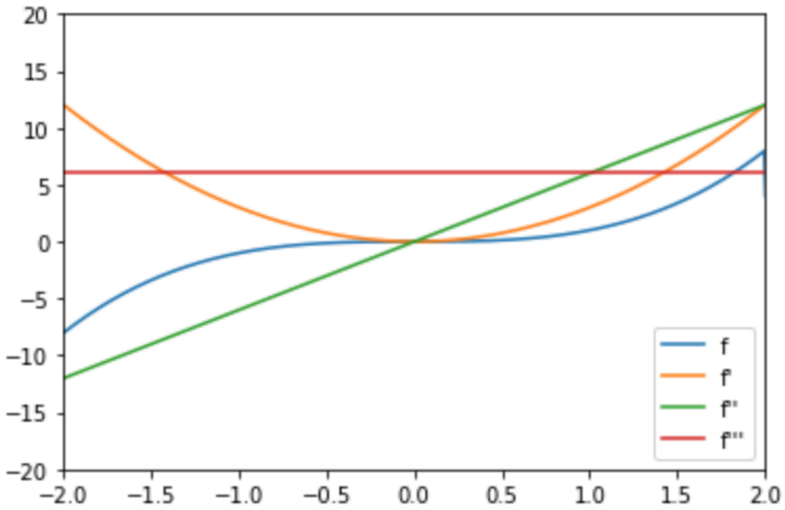
```
def f(x):
    return x**3
```

```
print(f(4.0))
print(jax.grad(f)(4.0)) #boom!
print(jax.grad(jax.grad(f))(4.0)) #boom!
print(jax.grad(jax.grad(jax.grad(f)))(4.0)) #boom!
print(jax.grad(jax.grad(jax.grad(jax.grad(f))))(4.0)) #boom!
```

```
64.0
48.0
24.0
6.0
0.0
```

```
g1i = jax.vmap(jax.grad(f))(xi)
g2i = jax.vmap(jax.grad(jax.grad(f)))(xi)
g3i = jax.vmap(jax.grad(jax.grad(jax.grad(f))))(xi)
plt.plot(xi,yi,  label = "f")
plt.plot(xi,g1i, label = "f'")
plt.plot(xi,g2i, label = "f''")
plt.plot(xi,g3i, label = "f'''")
plt.xlim(-2,2)
plt.ylim(-20,20)
plt.legend()
```

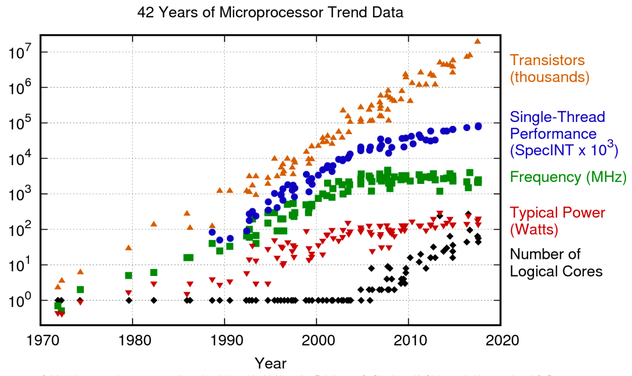
```
<matplotlib.legend.Legend at 0x7f0aecabd910>
```



# Applications

# ML Opportunities in Fundamental Physics

## Acceleration of Computation
(e.g. sometimes by searching for a good approximation)





turing lecture

DOI:10.1145/3282307

**Innovations like domain-specific hardware, enhanced security, open instruction sets, and agile chip development will lead the way.**

BY JOHN L. HENNESSY AND DAVID A. PATTERSON

**A New Golden Age for Computer Architecture**

WE BEGAN OUR Turing Lecture June 4, 2018[11] with a review of computer architecture since the 1960s. In addition to that review, here, we highlight current challenges and identify future opportunities, projecting another golden age for the field of computer architecture in the next decade, much like the 1980s when we did the research that led to our award, delivering gains in cost, energy, and security, as well as performance.

*who cannot remember the past are condemned it."* —George Santayana, 1905

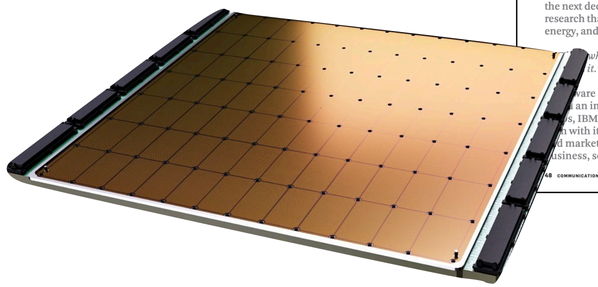
engineers, including ACM A.M. Turing Award laureate Fred Brooks, Jr., thought they could create a single ISA that would efficiently unify all four of these ISA bases.

They needed a technical solution for how computers as inexpensive as

» key insights

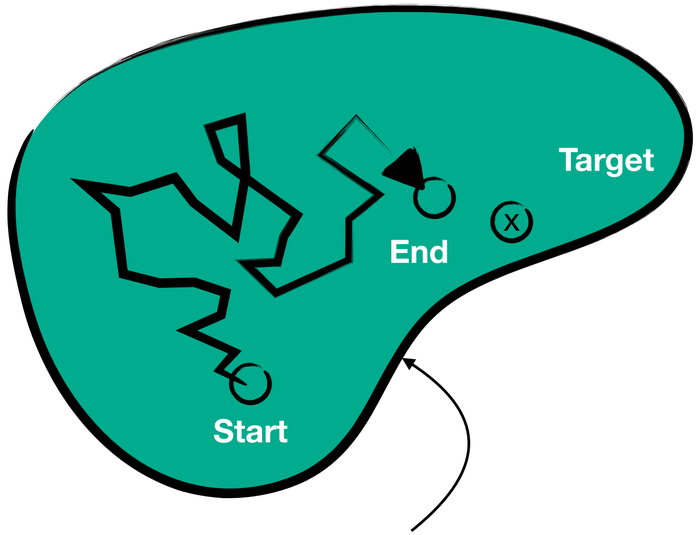
- Software advances can inspire architecture innovation.
- Elevating the hardware/software interface creates opportunities for architecture innovation.
- The marketplace ultimately settles architecture debates.

COMMUNICATIONS OF THE ACM | FEBRUARY 2018 | VOL. 62 | NO. 2



***simulation side: the physics is fixed:***
***nothing to search for →speed up simulation***

## Search for new (better) Algorithms
(e.g. targeted search based on samples)



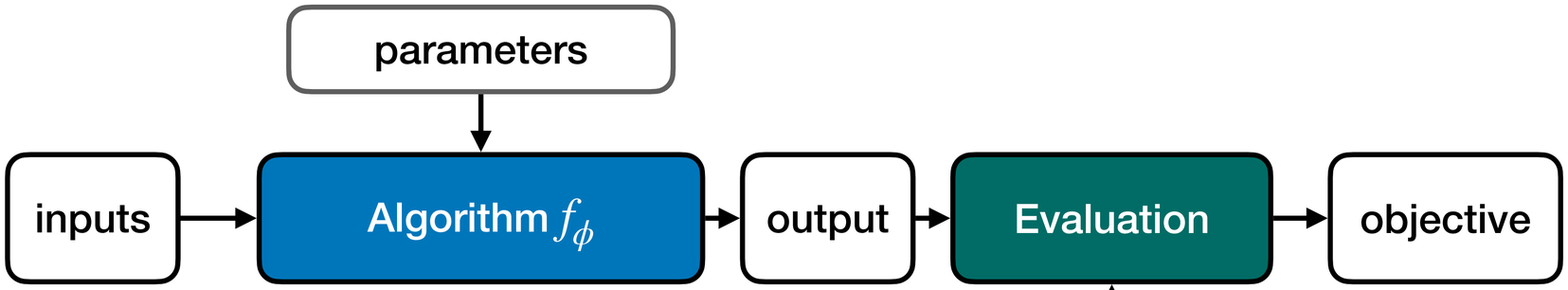

space of possible algorithms

***up to us to find best observables***
***→search for best reconstruction***

# Lightning Summary of ML

**Learning: data-driven search for a function with optimal performance in a huge**

**Space of Algorithms**




**Performance:**

*training data*

*algorithm*

$$\mathbb{E}_{(x,y)}\mathcal{L}(f_\phi(x), y)$$

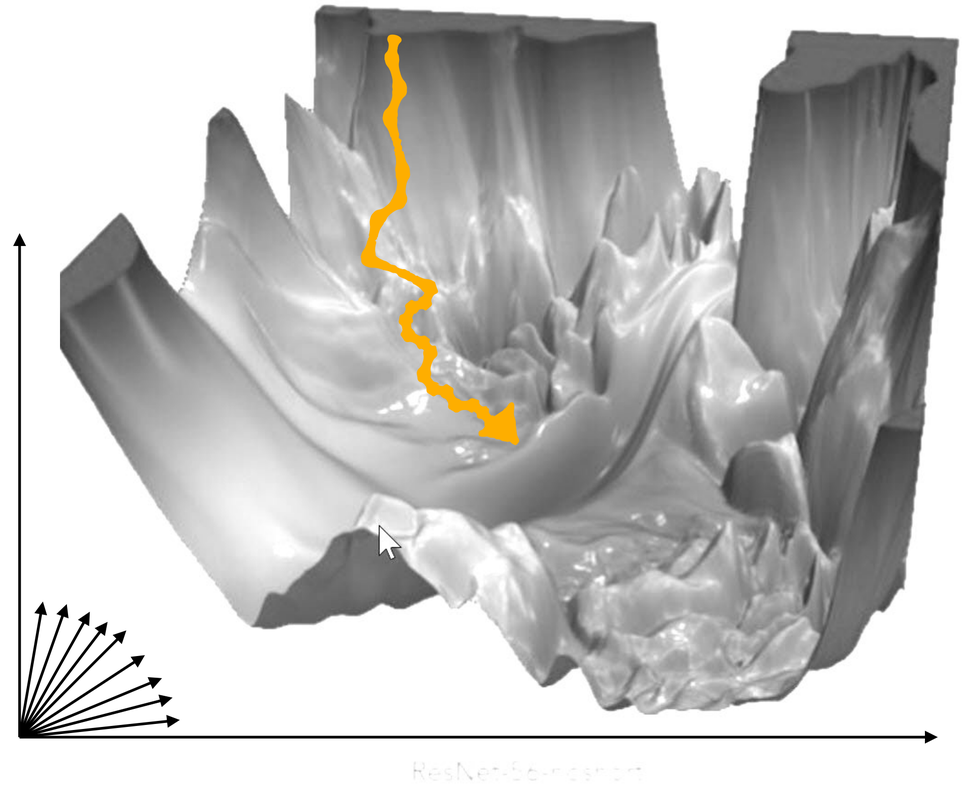
*performance evaluation*

**How do we learn practically?**

# Lightning Summary of ML

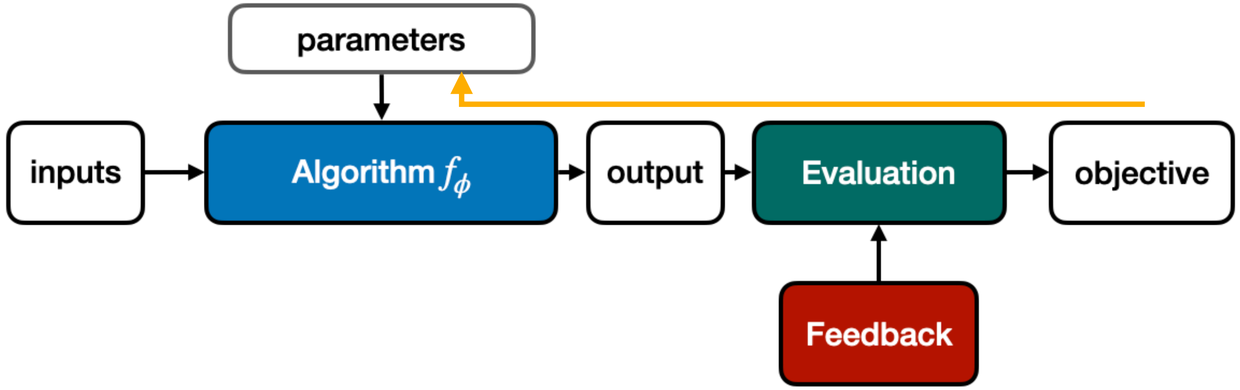
search space should be large enough → trillions of parameters! How could this work?

→ gradient-based optimization (“good sense of direction”)



*To deal with hyper-planes in a 14-dimensional space, visualize a 3D space and say 'fourteen' to yourself very loudly. -Hinton (DL pioneer)*

$$\frac{\partial L}{\partial \phi} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial \phi}$$

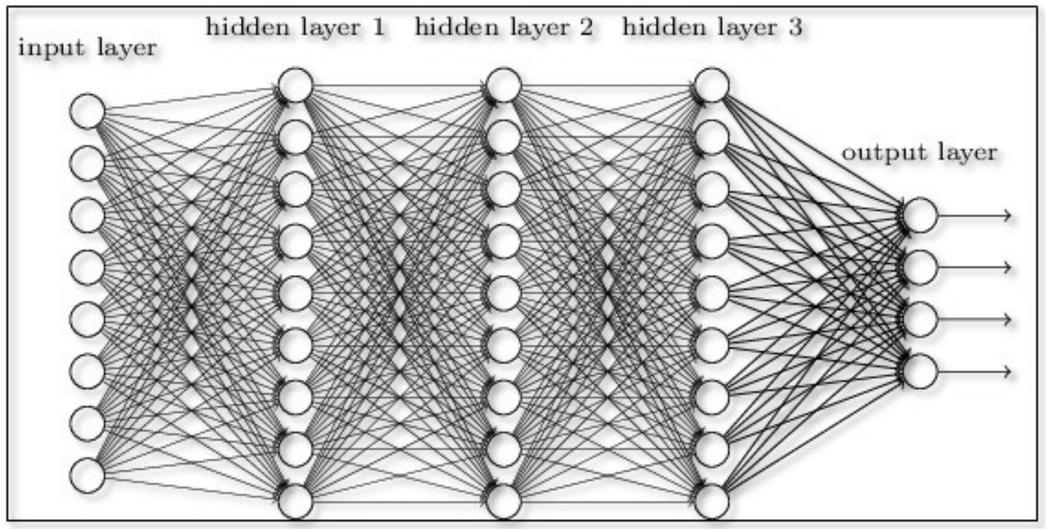



→ requires algorithms and evaluation to be differentiable

# Finding the right Search Space

**At first**

**fixed but generic, large and easily differentiable function class:**




***manual derivation of efficient gradient computation***

**Increasingly**

**domain-specific, arbitrary computation encoding e.g. symmetries, dynamics, …**

$$R_g y = f(R_g x) \qquad \dot{x} = f(x)$$

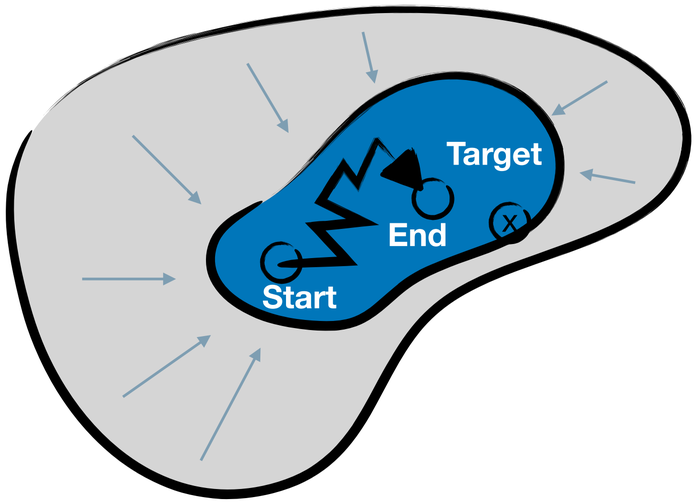

[M. Bronstein]

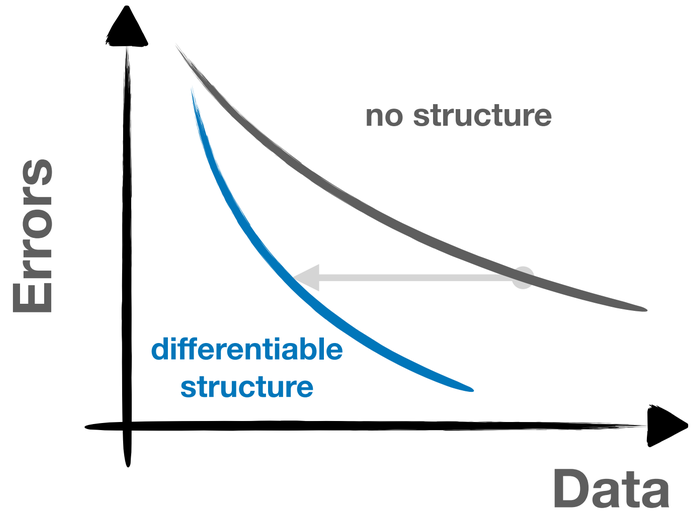
***?***

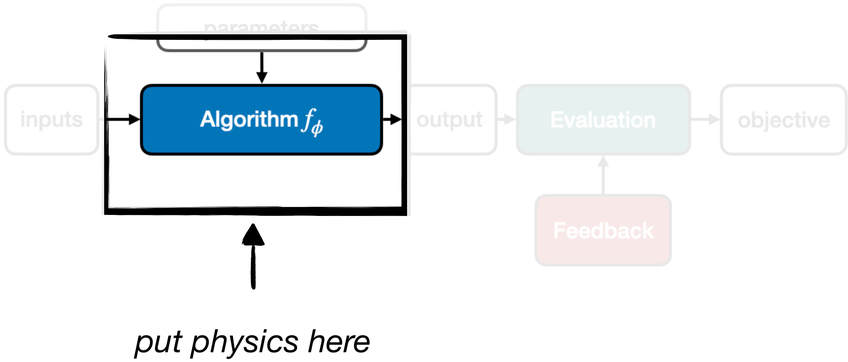
# Differentiable Programming in ML

**Immediate Gains from DiffProg:** allows us to add physics into ML models

- **bias towards good solutions by constraining solution space**
- hard-coded knowledge does not need to be learned from data (efficiency)

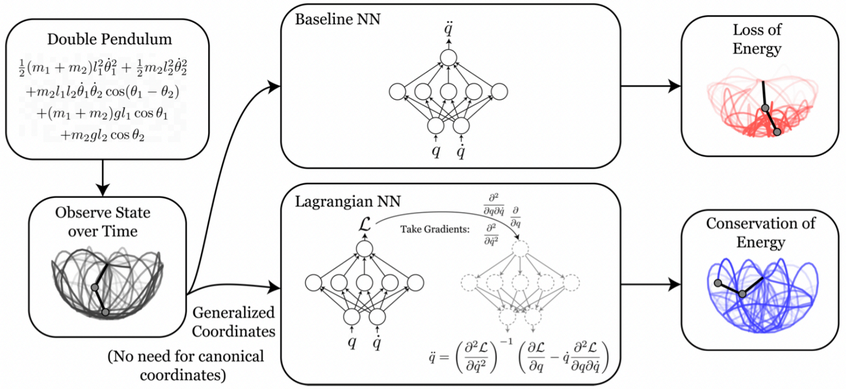


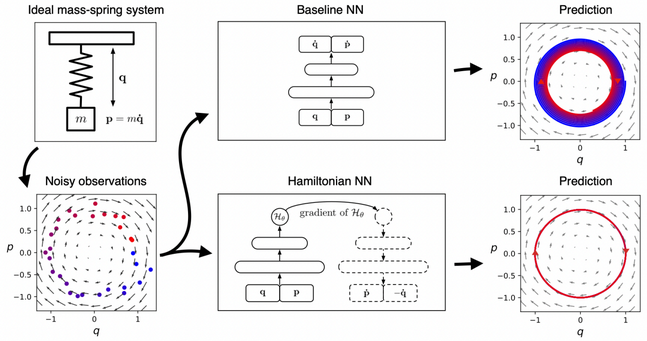

*put physics here*

# Differentiable Programming in ML



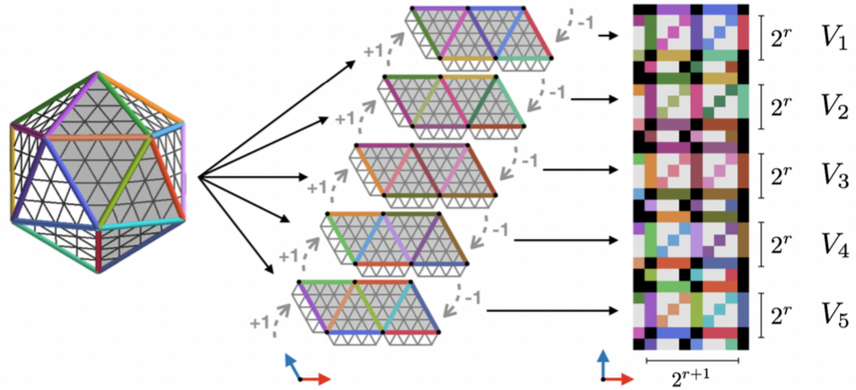

**Lagrangian Neural Nets**

arXiv: 2003.04630

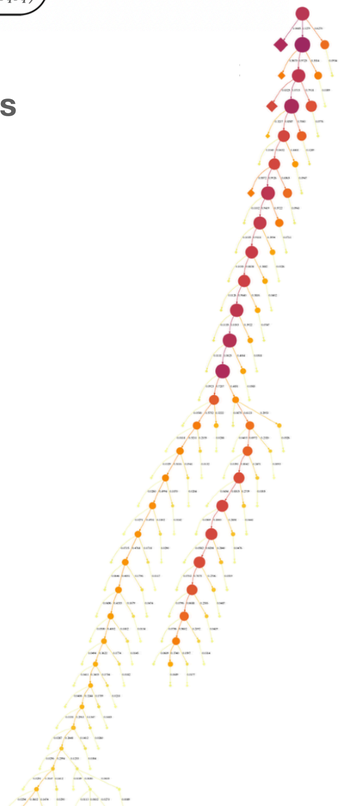



**Hamiltonian Neural Nets**

arXiv:1906.01563

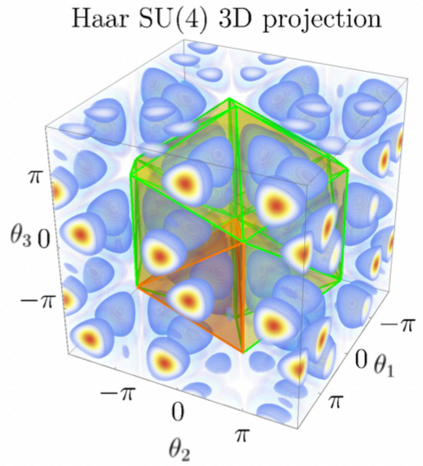


**Gauge-Equivariant Convolutional Neural Networks**

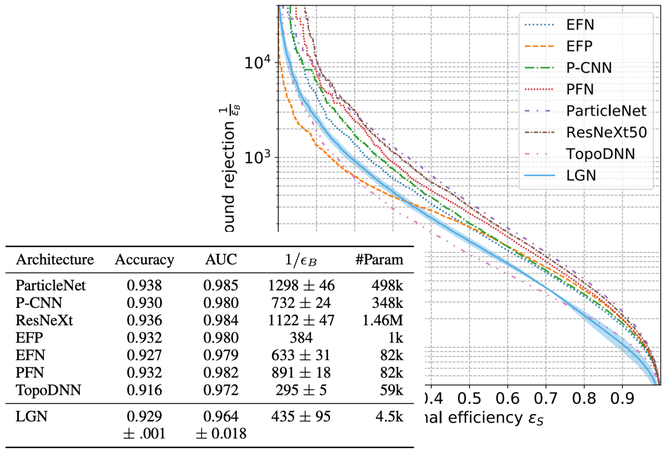


**Neural Nets with QCD-like Structure**

arXiv:1702.00748




**SU(N)-Equivariant Normalizing Flows**



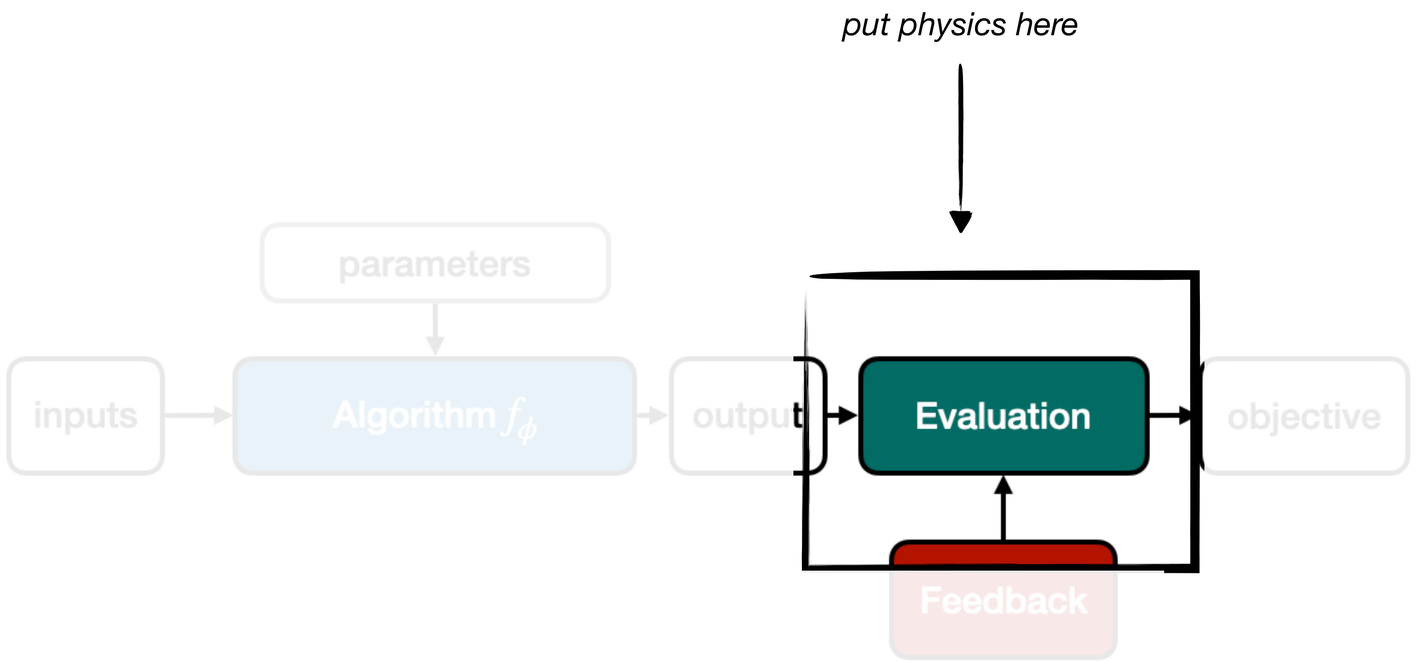

| Architecture | Accuracy | AUC | $1/\epsilon_B$ | #Param |
|---|---|---|---|---|
| ParticleNet | 0.938 | 0.985 | $1298 \pm 46$ | 498k |
| P-CNN | 0.930 | 0.980 | $732 \pm 24$ | 348k |
| ResNeXt | 0.936 | 0.984 | $1122 \pm 47$ | 1.46M |
| EFP | 0.932 | 0.980 | 384 | 1k |
| EFN | 0.927 | 0.979 | $633 \pm 31$ | 82k |
| PFN | 0.932 | 0.982 | $891 \pm 18$ | 82k |
| TopoDNN | 0.916 | 0.972 | $295 \pm 5$ | 59k |
| LGN | 0.929 $\pm$ .001 | 0.964 $\pm$ 0.018 | $435 \pm 95$ | 4.5k |

**Lorentz-Invariance**

arXiv:2006.04780

# Differentiable Programming in ML

**Complementary Approach:** add physics-driven *evaluation*

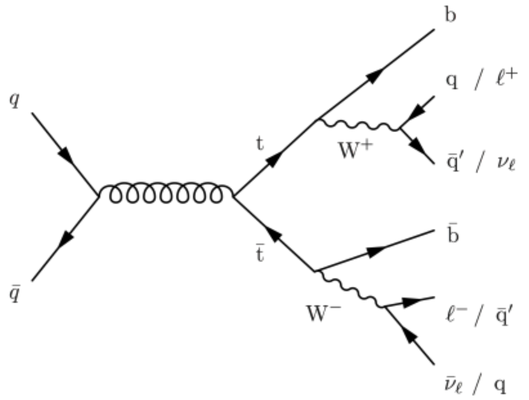


# Differentiable Programming in ML

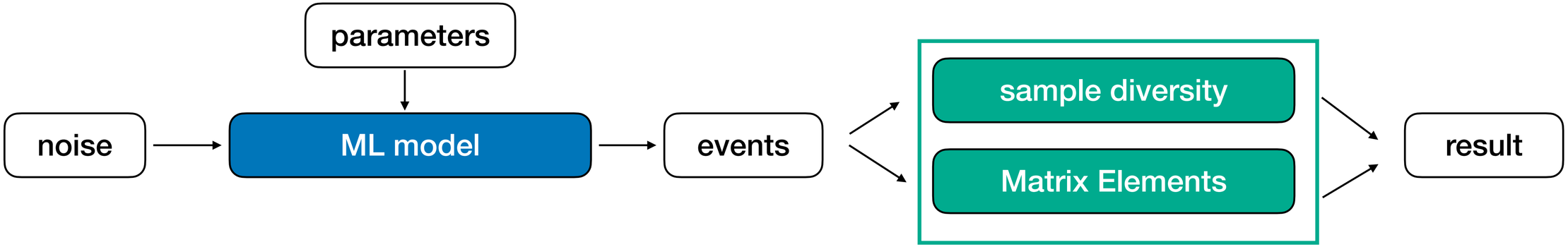
**Training Fast Simulators:** *produce events at correct relative proportions*

At parton level, events should follow Matrix Element proportions

$$\sigma(x,\theta) = \sum_i |\mathscr{M}_i(x,\theta)|^2$$

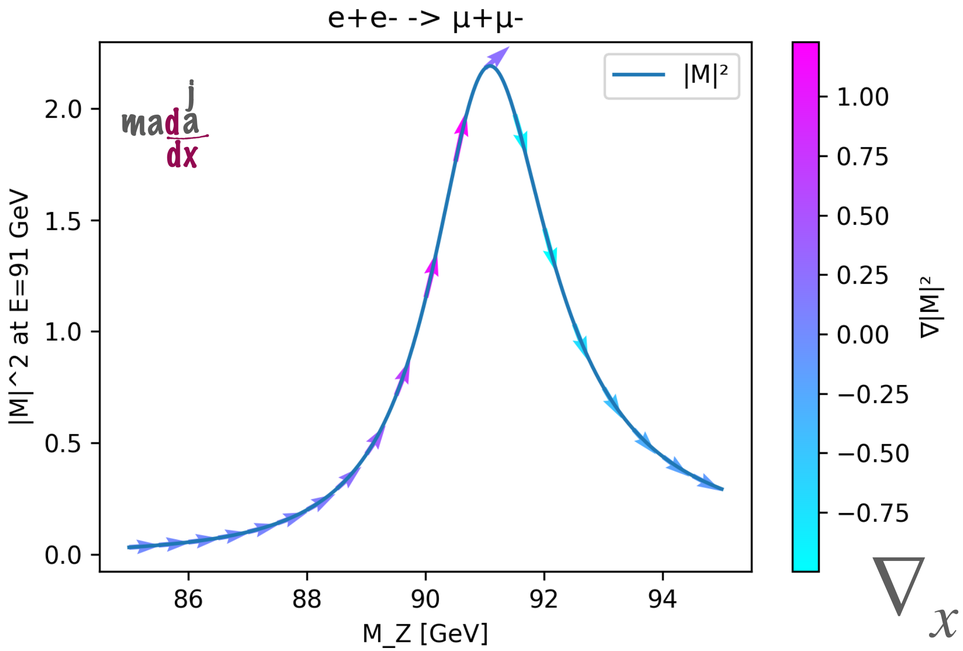


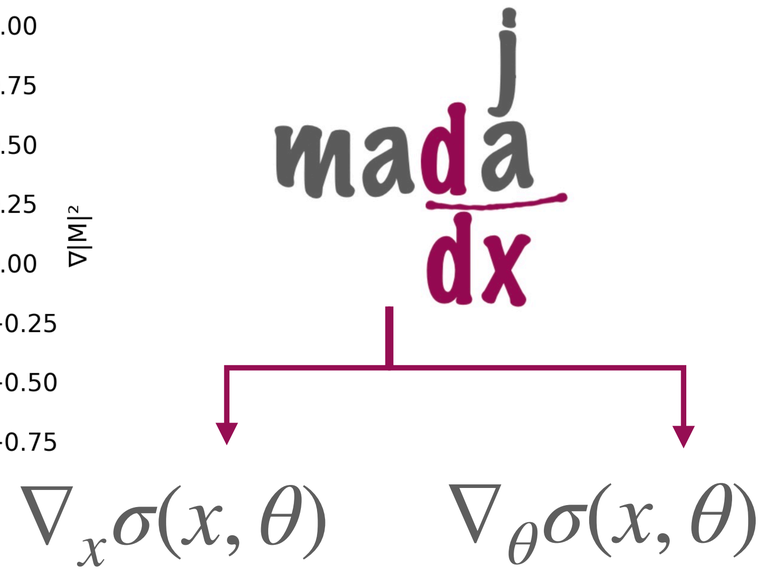

If we have ***differentiable Matrix Elements*** $|\mathscr{M}|^2(\{\overrightarrow{p_i}\},\theta)$ we can check directly



# Differentiable Programming in ML

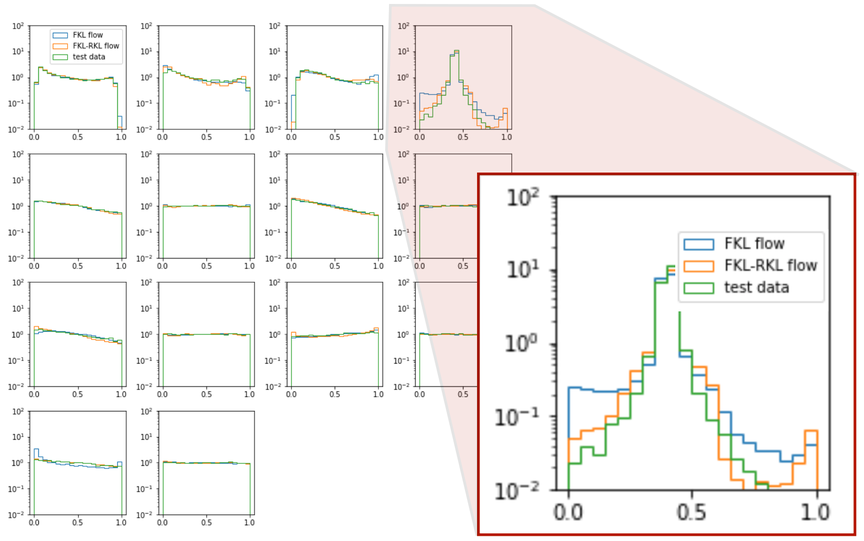
**MadJax:** MadGraph calculations (originally FORTRAN) transpiled into differentiable programming language (JAX) → **usable as evaluation function during training**




**phase-space derivatives**

**theory Parameter derivatives**




*better description of density than pure ML training*

```
mg5_aMC  –mode=madjax_me_gen -f ee_to_mumu.mg5
```

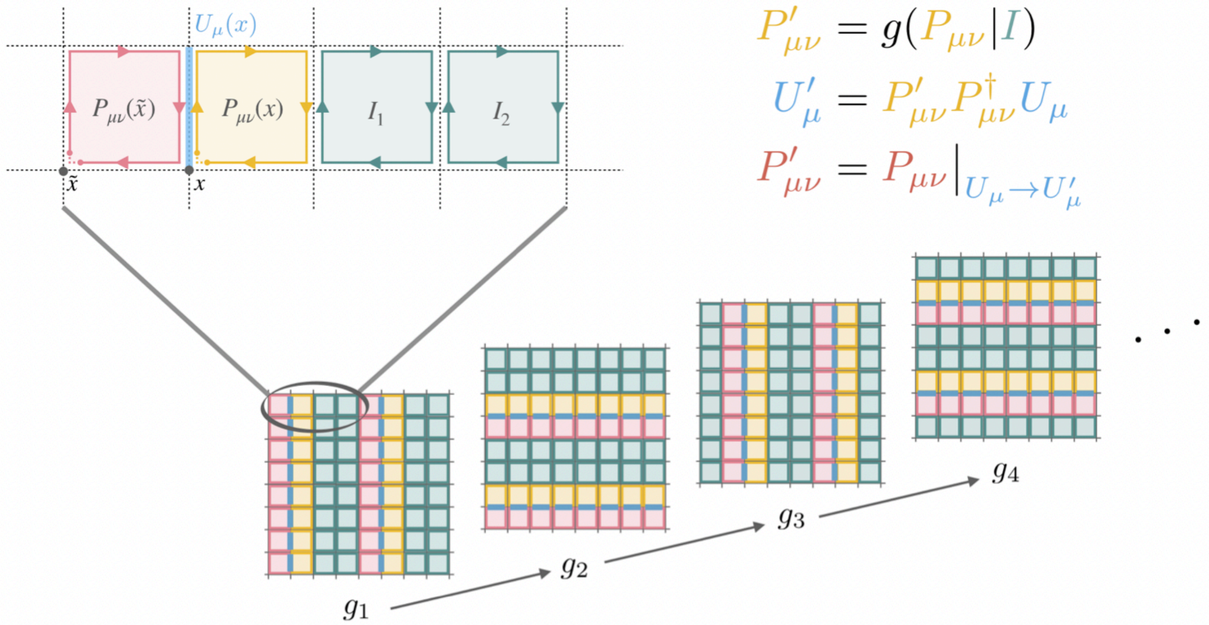
[LH, M. Kagan]
arxiv:2203.00057

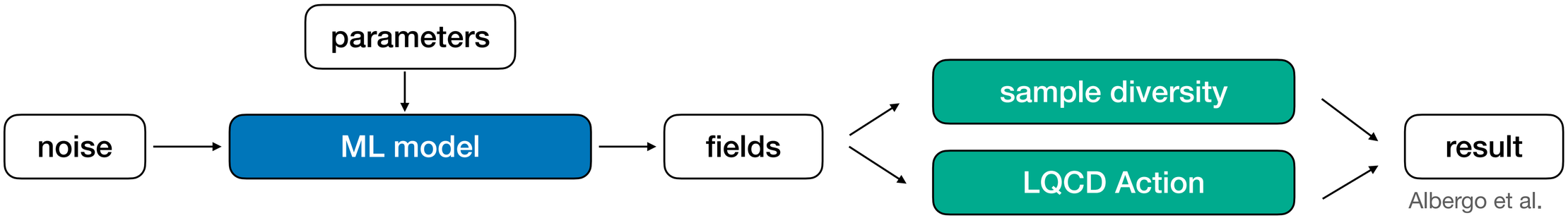
# Differentiable Programming in ML

**Same approach in Lattice QCD:**

Learn **proposal distribution** for sampling of fields on a lattice (for MCMC / IS)

- encode symmetries in ML sampler
- evaluate on LQCD action in DiffProg language (pytorch)




Albergo et al.

arxiv: 1904.12072

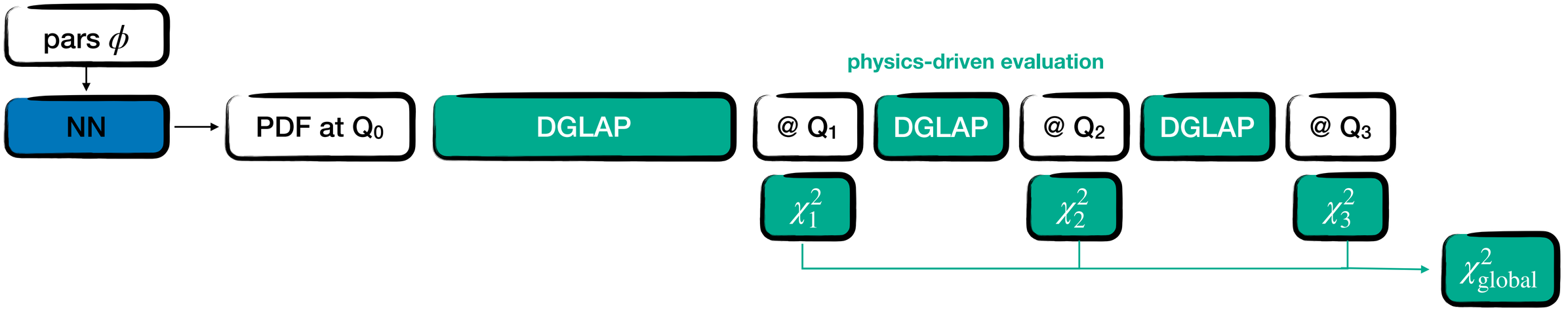
arxiv:2101.08176

physics-driven evaluation

# Differentiable Programming in ML

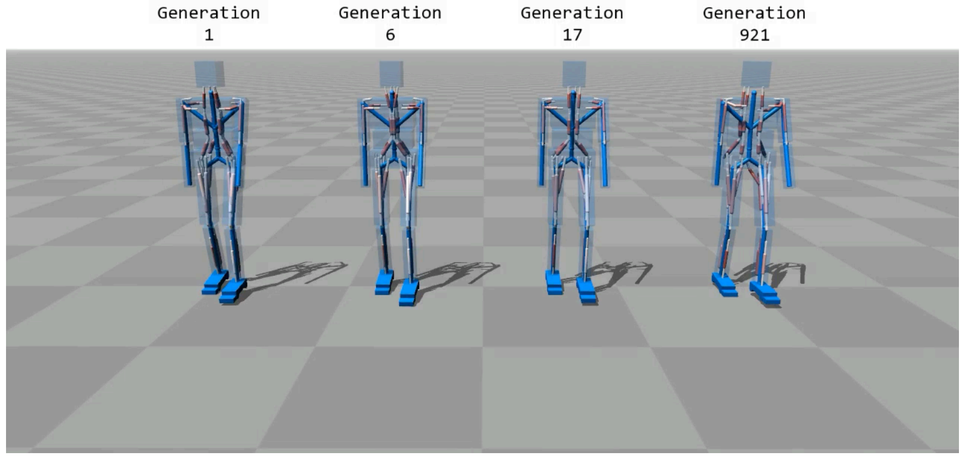
**Parton Density Functions:** DP can train NNPDF as it was meant to be trained

One of the early use-cases of NNs in HEP: PDF parametrizations




Curiosity:
traditionally **not(!) trained via gradient-descent**
→ too difficult to get gradients

→ use genetic algorithms (mutation + select)
→ works but is slow




genetic algorithms [Source]

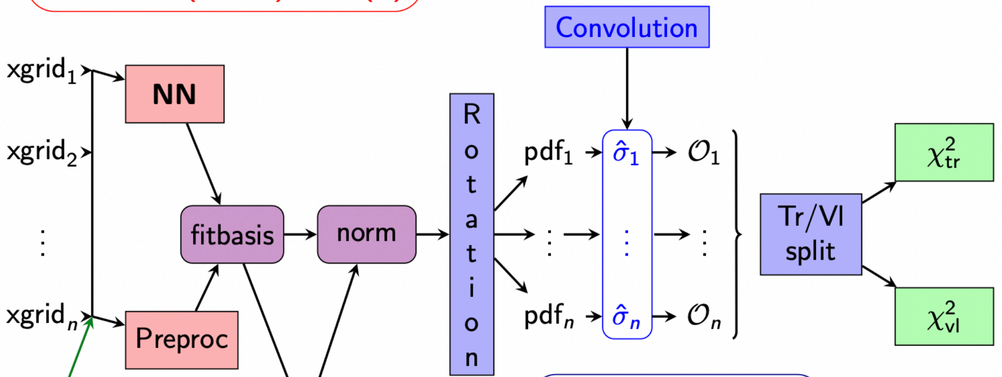
# Differentiable Programming in ML

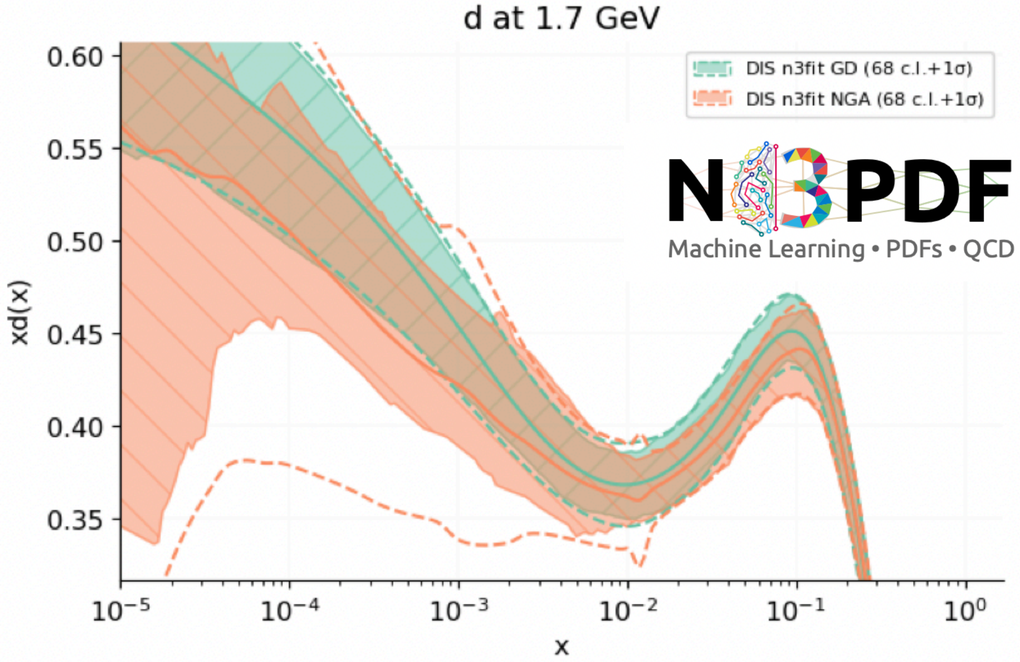
**More recently:** PDF evolution kernels implemented in DiffProg (Tensorflow)

- allows finally for a gradient-based training of NN

For all fits shown in this paper we utilize gradient descent (GD) methods to substitute the previously used genetic algorithm. This change can be shown to greatly reduce the computing cost of a fit while maintaining a very similar (and in occasions improved) $\chi^2$-goodness. The less stochastic nature of GD methods also produces more stable fits than its GA counterparts. The main reason why the GD methods had not been tested before were due to the difficulty of computing the gradient of the loss function (mainly due to the convolution with the fastkernel tables) in a efficient way. This is one example on how the usage of new technologies can facilitate new studies thanks to differentiable programming and distributed computing.

$$f_i = A_i x^{\alpha_i}(1-x)^{\beta_i} NN(x)$$



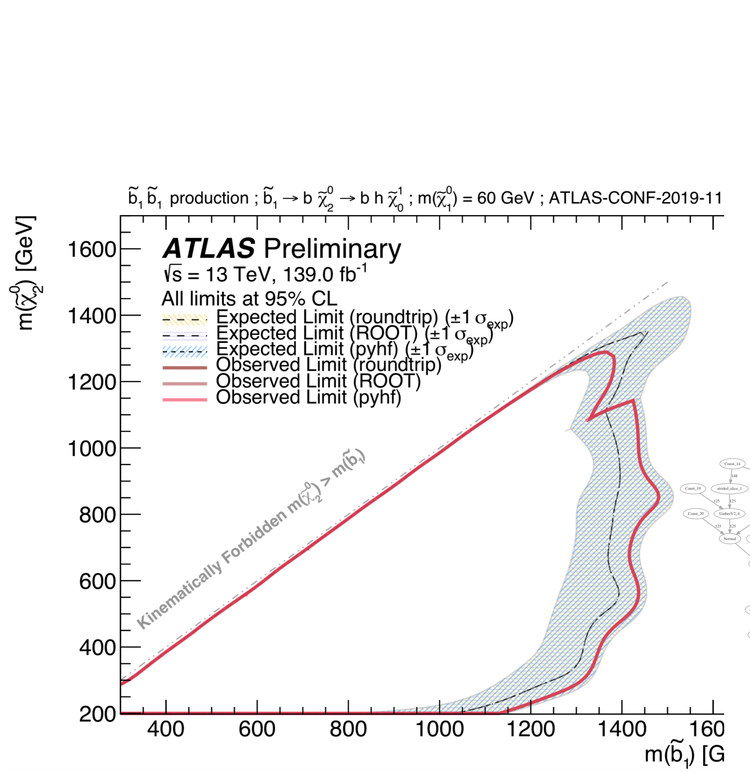



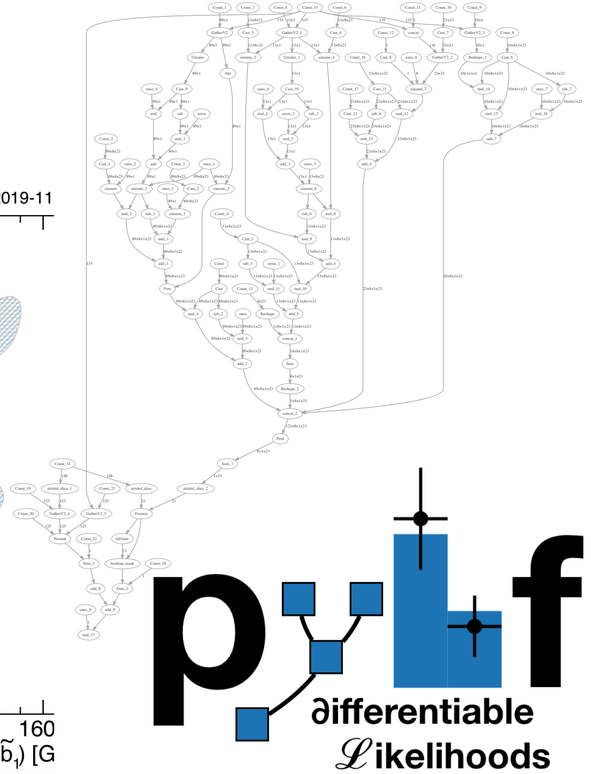

arxiv: 1907.05075
[Carrazza et al]

# Differentiable Programming Beyond ML

**Gradients useful far beyond ML:** e.g. complex fits via differentiable programming

## Binned Likelihoods (LHC, EIC, Belle-II, …)

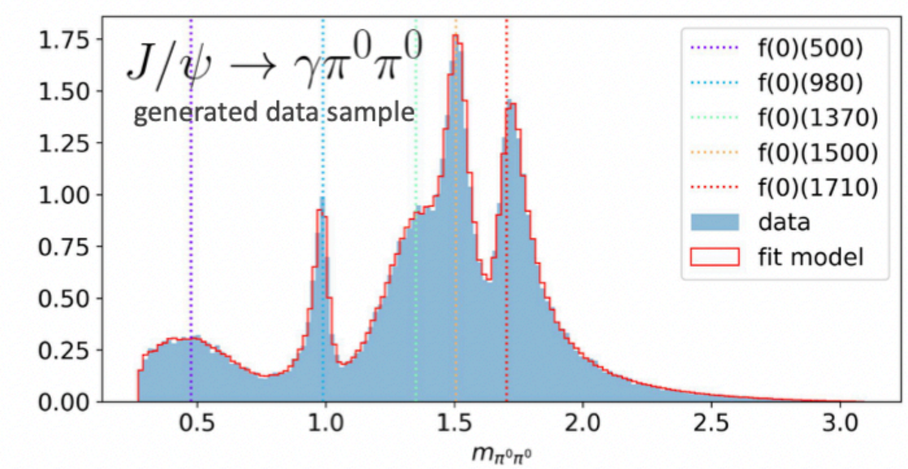


pyhf
differentiable
Likelihoods

pyhf [LH, G. Start, M. Feickert]

## Partial Wave Analysis




```
some_amplitude = model.components[
    R"A_{D^{0}_{0} \to K^{0}_{0} a_{0}(980)^{0}_{0}; a_{0}(980)^{0}_{0} \to K^{+}_{0} K^{-}_{0}}"
]
```

$$\frac{C_{D^0\to K_0^0 a_0(980)_0^0; a_0(980)^0\to K_0^+K_0^-}\,\Gamma_{a_0(980)^0} m_{a_0(980)^0}\sqrt{B_0^2\left(\left(d_{a_0(980)^0}\right)^2 q_{122}^2\left(m_{12}^2\right)\right)}\,D^0_{0,0}\left(-\phi_{1+2},\theta_{1+2},0\right)D^0_{0,0}\left(-\phi_{1,1+2},\theta_{1,1+2},0\right)}{-m_{12}^2+\left(m_{a_0(980)^0}\right)^2-i m_{a_0(980)^0}\Gamma\left(m_{12}^2\right)}$$

$$= \frac{C_{D^0\to K_0^0 a_0(980)_0^0; a_0(980)^0\to K_0^+K_0^-}\,\Gamma_{a_0(980)^0} m_{a_0(980)^0}}{-\frac{i\Gamma_{a_0(980)^0}\left(m_{a_0(980)^0}\right)^2\sqrt{\frac{\left(m_{12}^2-(m_1-m_2)^2\right)\left(m_{12}^2-(m_1+m_2)^2\right)}{m_{12}^2}}}{m_{12}\sqrt{\frac{\left(\left(m_{a_0(980)^0}\right)^2-(m_1-m_2)^2\right)\left(\left(m_{a_0(980)^0}\right)^2-(m_1+m_2)^2\right)}{\left(m_{a_0(980)^0}\right)^2}}} - m_{12}^2+\left(m_{a_0(980)^0}\right)^2}$$

$$= \frac{0.074}{-m_{12}^2+0.96-\frac{0.599\sqrt{m_{12}^2-0.975}}{m_{12}}}$$

ComPWA [R. deBoer, M. Mikhasenko]