AUTOMATIC DIFFERENTIATION

AN ABSTRACTION FOR MACHINE LEARNING

ALEX WILTSCHKO

RESEARCH ENGINEER

TWITTER

ADVANCED TECHNOLOGY GROUP

@AWILTSCH

EVOLUTION OF MODELING IN BIOLOGY

Through the eyes of a formerly practicing neuroscientist

1. **Partial differential equations (PDEs)** — Physical simulation, or simulation by physical analogy

2. **Probabilistic Graphical Models (PGMs)** — Instantiating causal or correlative relationships directly in a computer program

3. **Neural networks** — Enormously data hungry adaptive basis regression, in an era of enormous data

NEURAL NETWORKS IN BIOLOGY

That have nothing to do with networks of neurons

Predicting:

- DNA Binding (e.g. Kelley et al @ MIA)
- Predicting molecular properties (Duvenuad et al @ MIA)
- Behavioral modeling (Johnson et al)
- DNA expression
- DNA methylation state
- Protein folding
- Image correction

NEURAL NETWORKS IN BIOLOGY

Biology now produces enough data to keep the deep learning furnace burning

No Data

Some Data

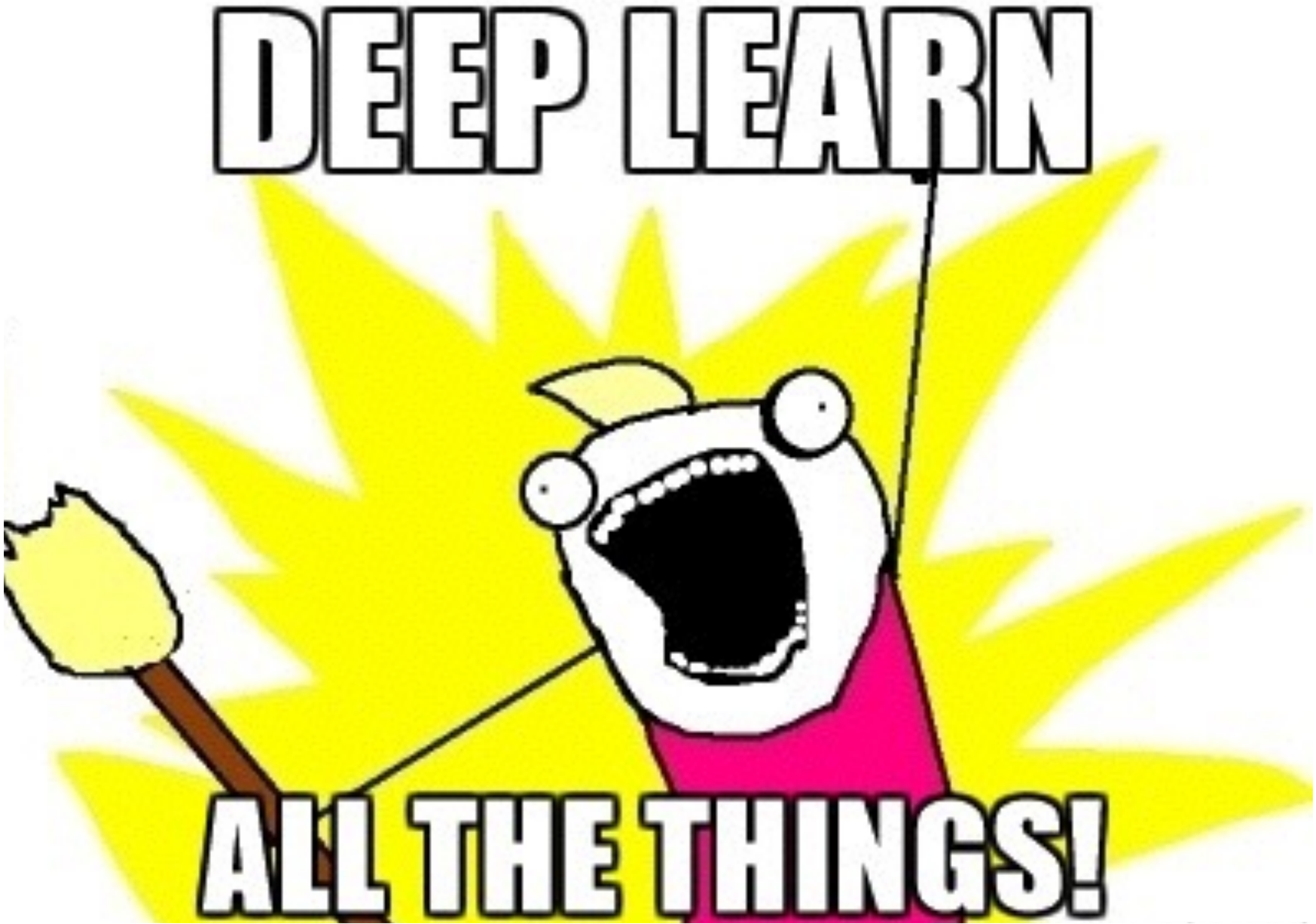
Broad-sized Data

1. **Partial differential equations (PDEs)** — Phy
simulation by physical analogy

2. **Probabilistic Graphical Models (PGMs)** — I
or correlative relationships directly in a comp

3. **Neural networks** — Enormously data hungr
regression, in an era of enormous data

NEURAL NETWORKS IN BIOLOGY



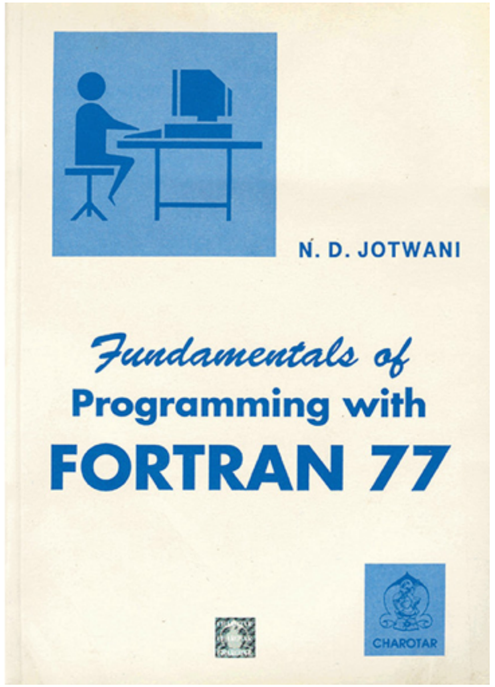
NEURAL NETWORKS IN BIOLOGY

But how?

WE WORK ON TOP OF STABLE ABSTRACTIONS

We should take these for granted, to stay sane!

Arrays



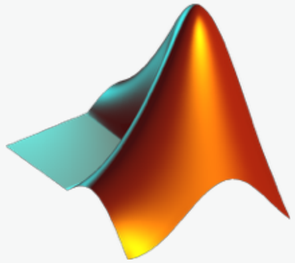
Est: 1957

Linear Algebra

BLAS
LINPACK
LAPACK

Est: 1979
(now on GitHub!)

Common Subroutines



NumPy

Est: 1984

MACHINE LEARNING HAS OTHER ABSTRACTIONS

These assume all the other lower-level abstractions in scientific computing

All gradient-based optimization (that includes neural nets) relies on **Automatic Differentiation (AD)**

"Mechanically calculates derivatives as functions expressed as computer programs, at machine precision, and with complexity guarantees." (Barak Pearlmutter).

Not finite differences -- generally bad numeric stability. We still use it as "gradcheck" though.
Not symbolic differentiation -- no complexity guarantee. Symbolic derivatives of heavily nested functions (e.g. all neural nets) can quickly blow up in expression size.

AUTOMATIC DIFFERENTIATION IS *THE* ABSTRACTION FOR GRADIENT-BASED ML

All gradient-based optimization (that includes neural nets) relies on **Automatic Differentiation** (AD)

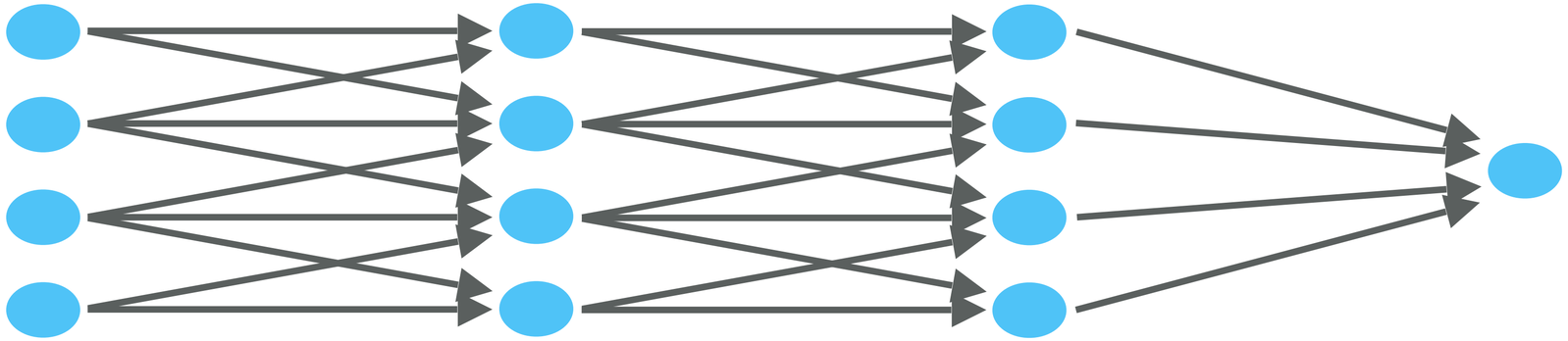
- Rediscovered several times (Widrow and Lehr, 1990)
- Described and implemented for FORTRAN by Speelpenning in 1980 (although forward-mode variant that is less useful for ML described in 1964 by Wengert).
- Popularized in connectionist ML as "backpropagation" (Rumelhart et al, 1986)
- In use in nuclear science, computational fluid dynamics and atmospheric sciences (in fact, their AD tools are more sophisticated than ours!)

AUTOMATIC DIFFERENTIATION IS *THE* ABSTRACTION FOR GRADIENT-BASED ML

All gradient-based optimization (that includes neural nets) relies on **Reverse-Mode Automatic Differentiation** (AD)

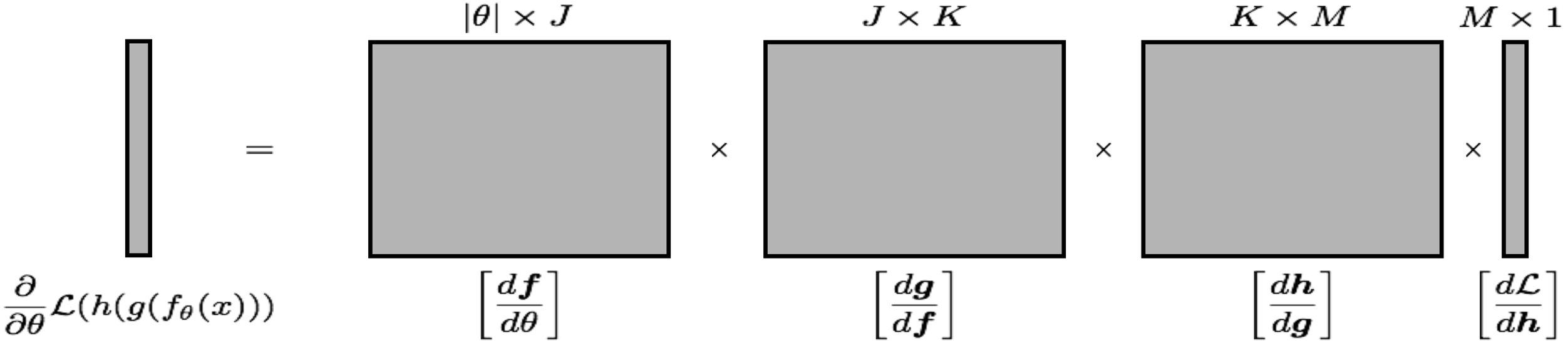
- Rediscovered several times (Widrow and Lehr, 1990)
- Described and implemented for FORTRAN by Speelpenning in 1980 (although forward-mode variant that is less useful for ML described in 1964 by Wengert).
- Popularized in connectionist ML as "backpropagation" (Rumelhart et al, 1986)
- In use in nuclear science, computational fluid dynamics and atmospheric sciences (in fact, their AD tools are more sophisticated than ours!)

FORWARD MODE (SYMBOLIC VIEW)

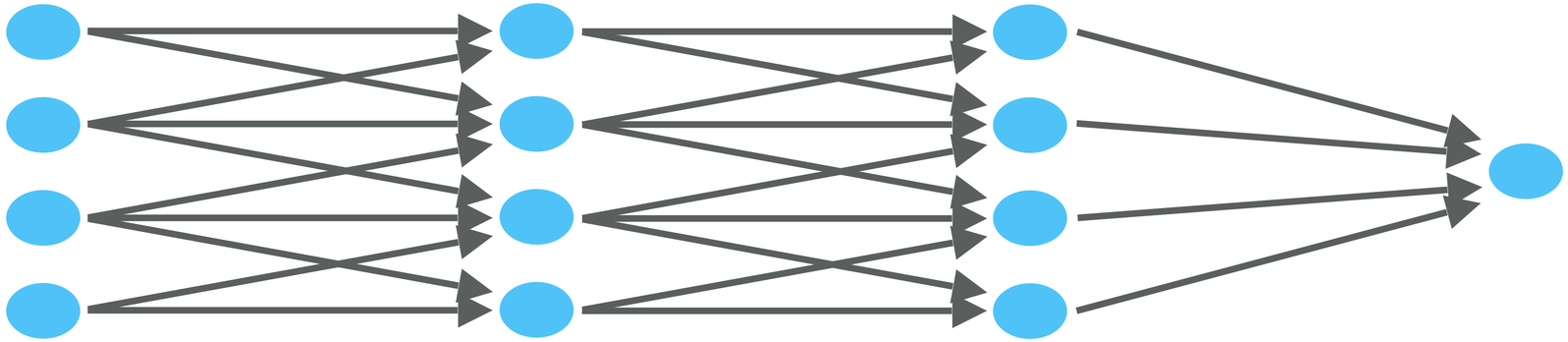


$\mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

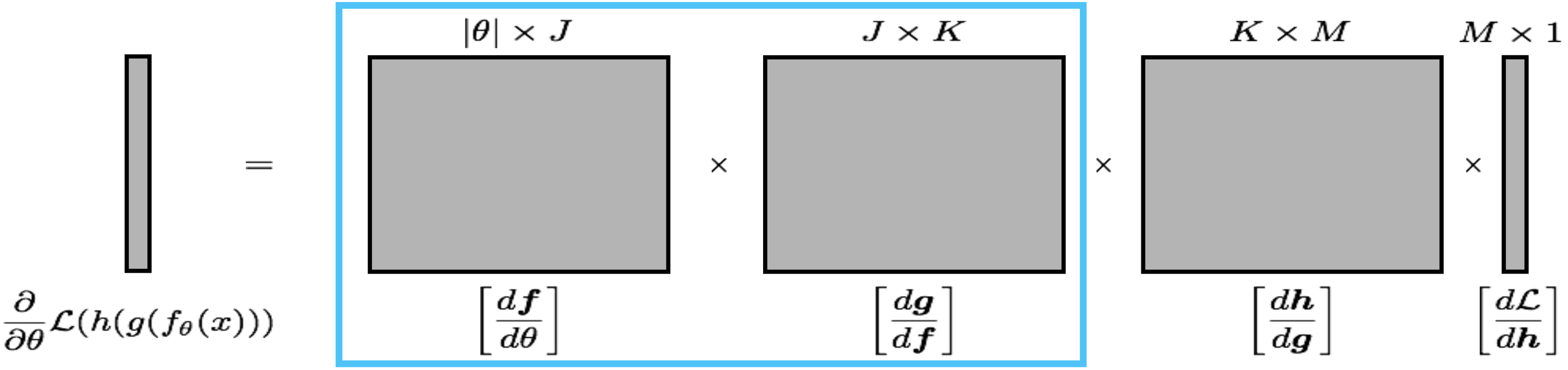
$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$



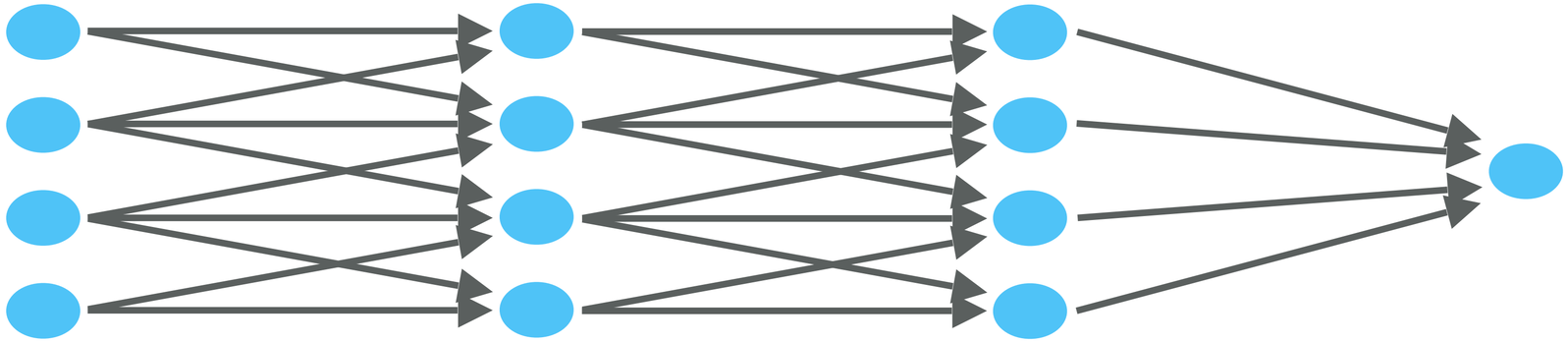
FORWARD MODE (SYMBOLIC VIEW)



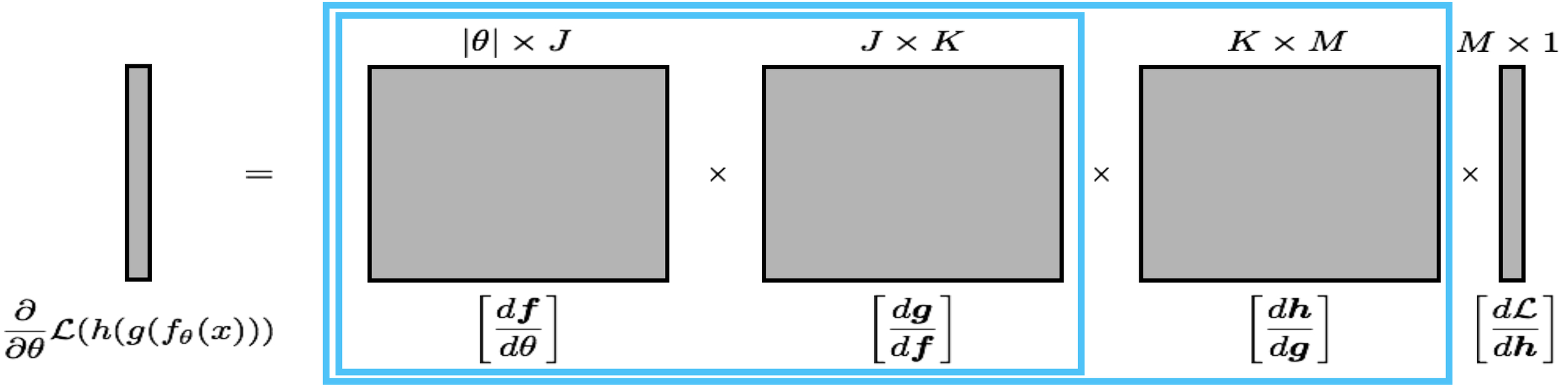
$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$



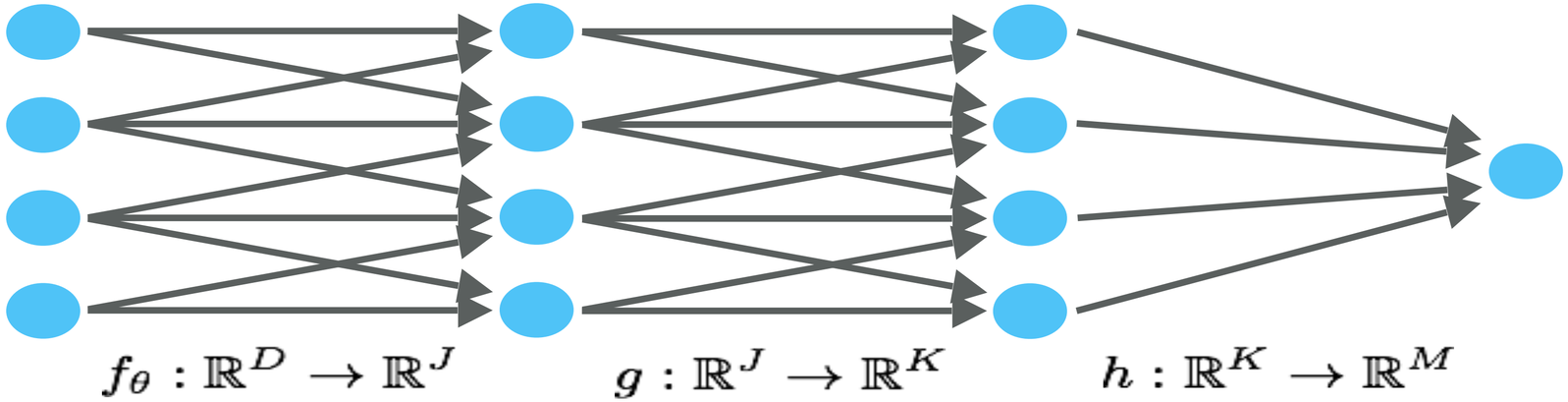
FORWARD MODE (SYMBOLIC VIEW)



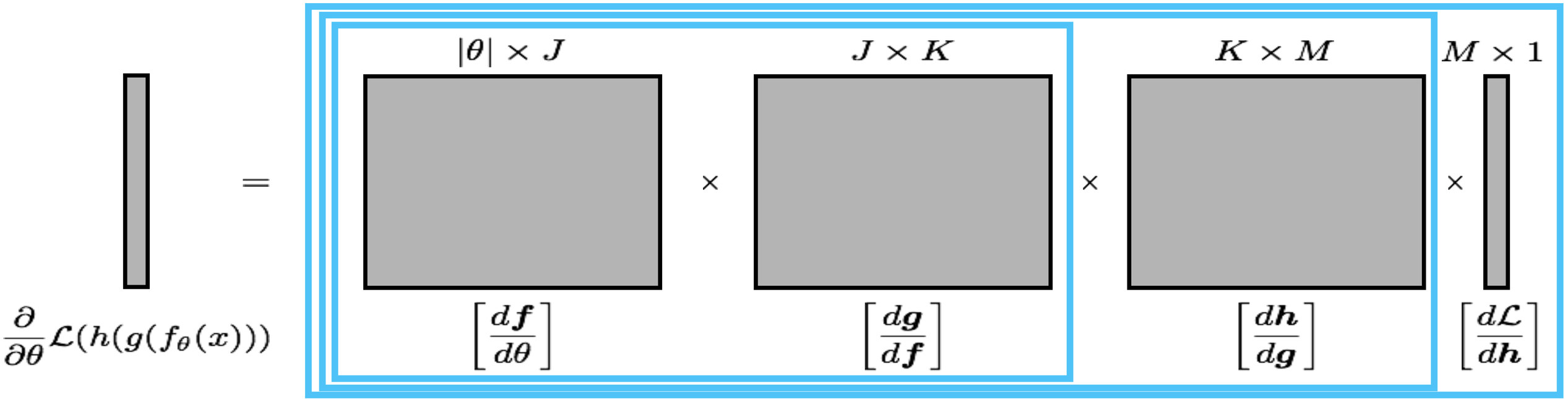
$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$



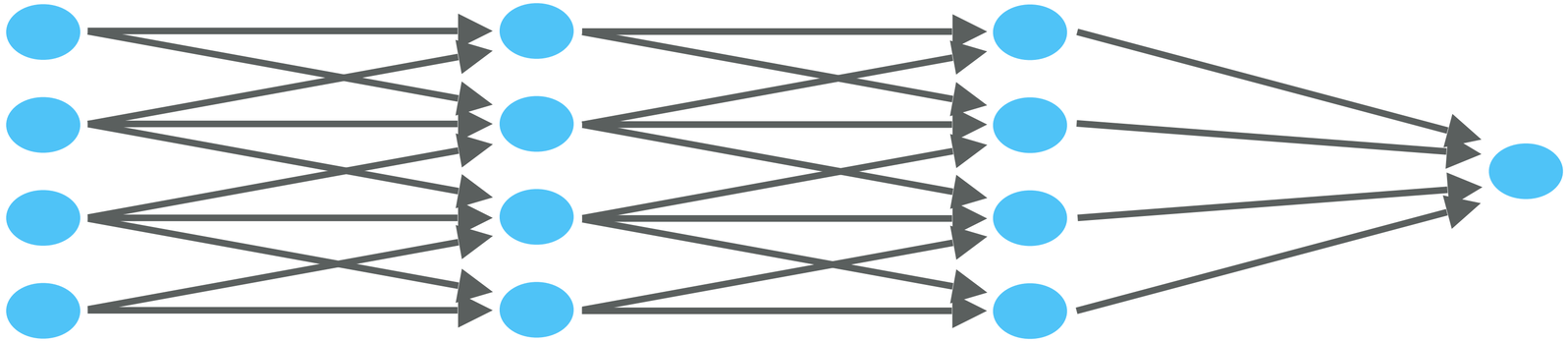
FORWARD MODE (SYMBOLIC VIEW)



$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$



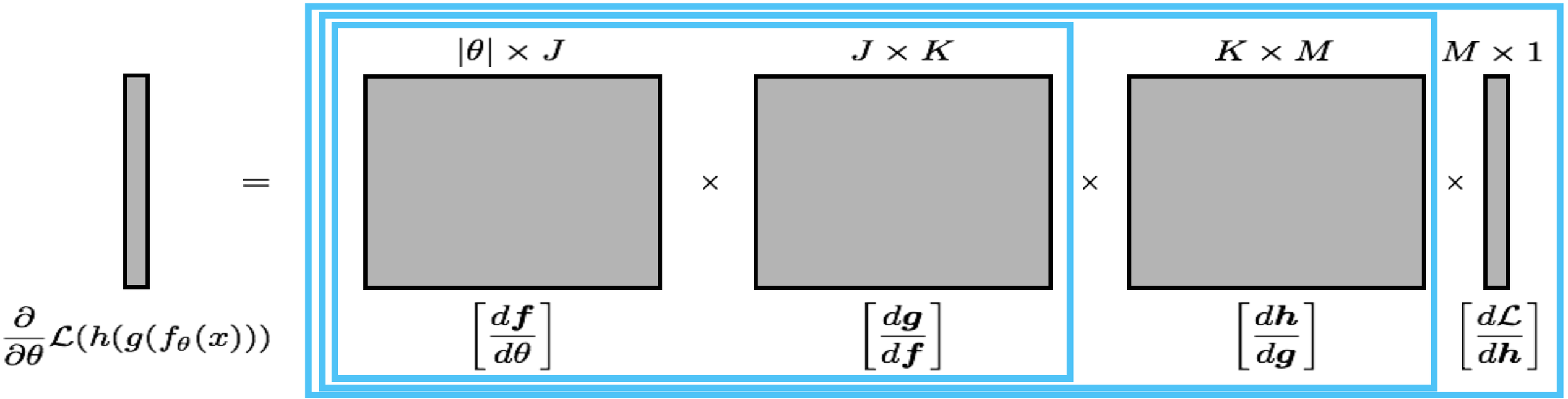
FORWARD MODE (SYMBOLIC VIEW)



$\mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

Left to right:

$$O(|\theta|JK + |\theta|KM + |\theta|M)$$



FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

dada = 1

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

dada = 1

b = 2

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

dada = 1

b = 2

dbda = 0

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

dada = 1

b = 2

dbda = 0

c = 1

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

dada = 1

b = 2

dbda = 0

c = 1

dcda = 0

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3	a = 3
	dada = 1
b = 2	b = 2
	dbda = 0
c = 1	c = 1
	dcda = 0
d = a * math.sin(b) = 2.728	d = a * math.sin(b) = 2.728
return 2.728	

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

dada = 1

b = 2

dbda = 0

c = 1

dcda = 0

d = a * math.sin(b) = 2.728

ddda = math.sin(b) = 0.909

From Baydin 2016

FORWARD MODE (PROGRAM VIEW)

Left-to-right evaluation of partial derivatives (not so great for optimization)

We can write the evaluation of a program in a sequence of operations, called a "trace", or a "Wengert list"

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

dada = 1

b = 2

dbda = 0

c = 1

dcda = 0

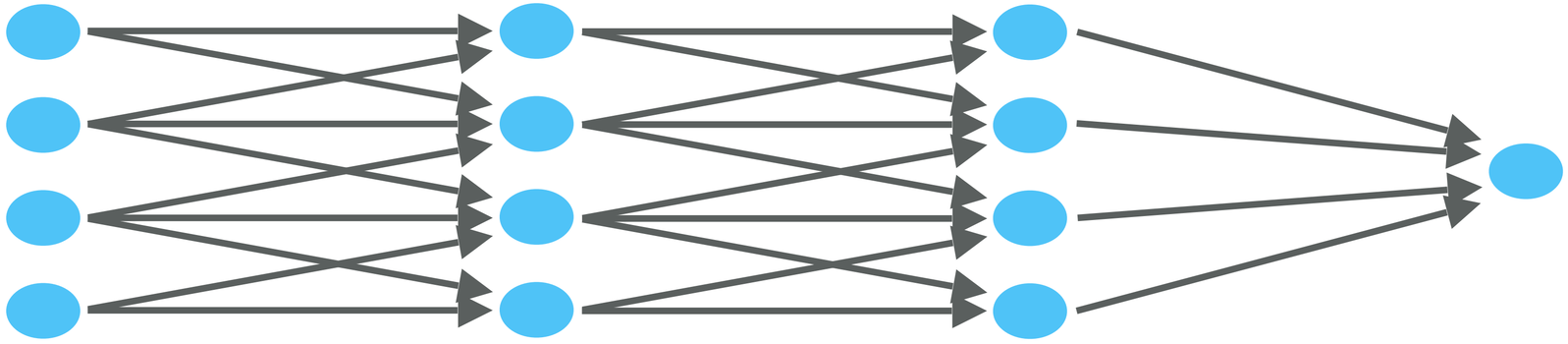
d = a * math.sin(b) = 2.728

ddda = math.sin(b) = 0.909

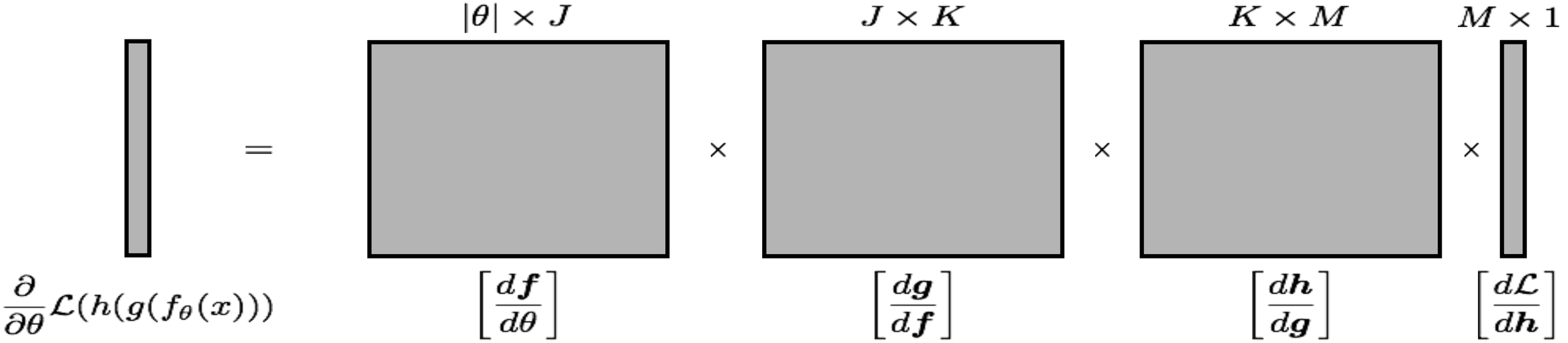
return 0.909

From Baydin 2016

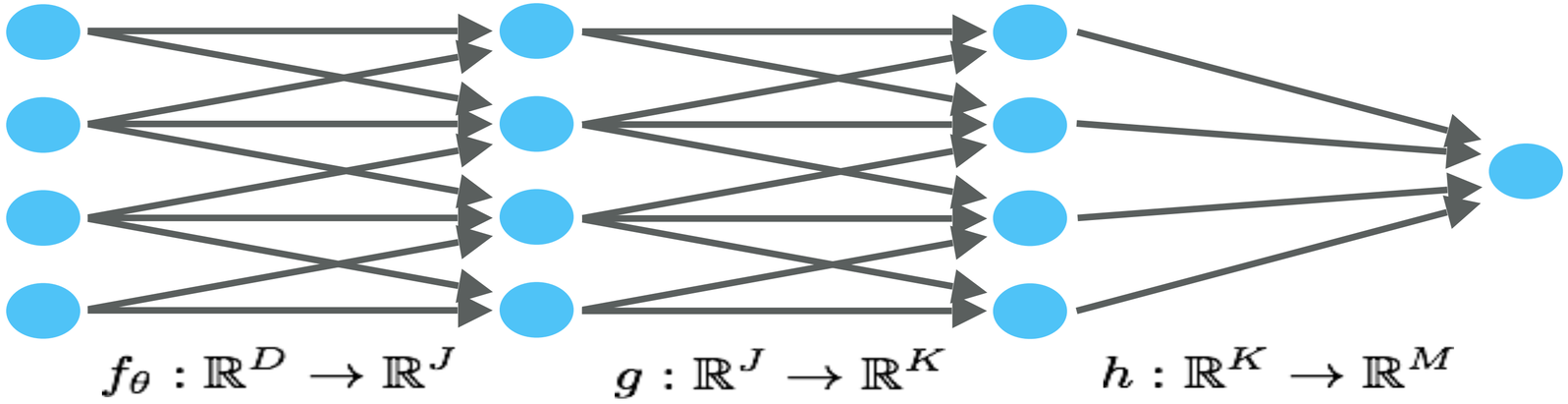
REVERSE MODE (SYMBOLIC VIEW)



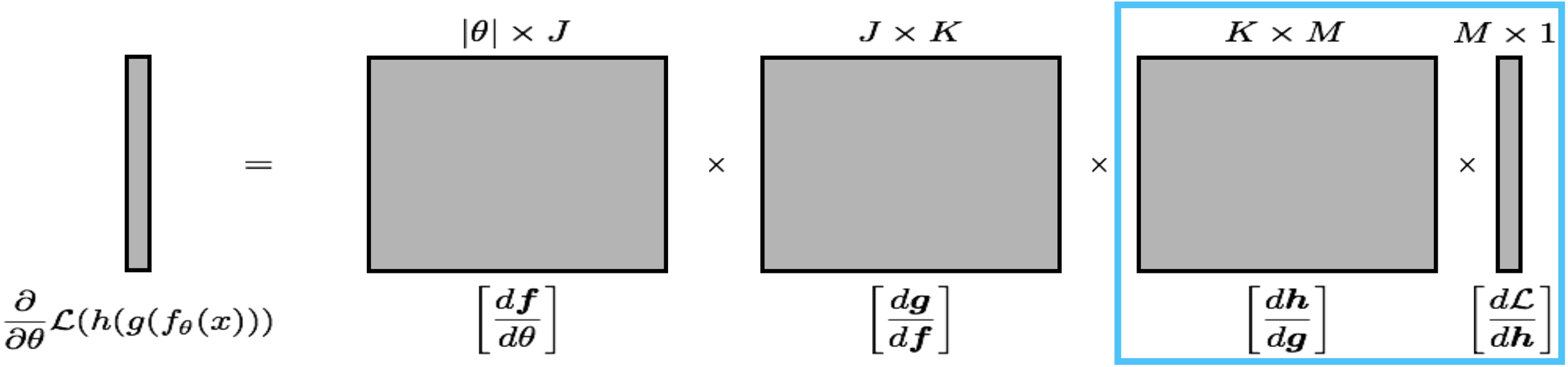
$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$



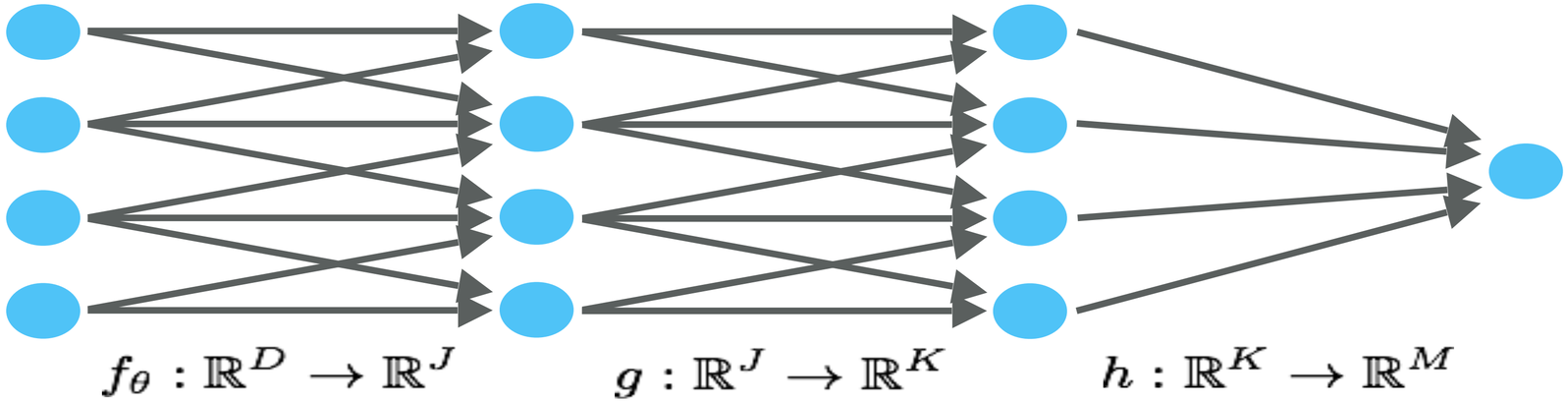
REVERSE MODE (SYMBOLIC VIEW)



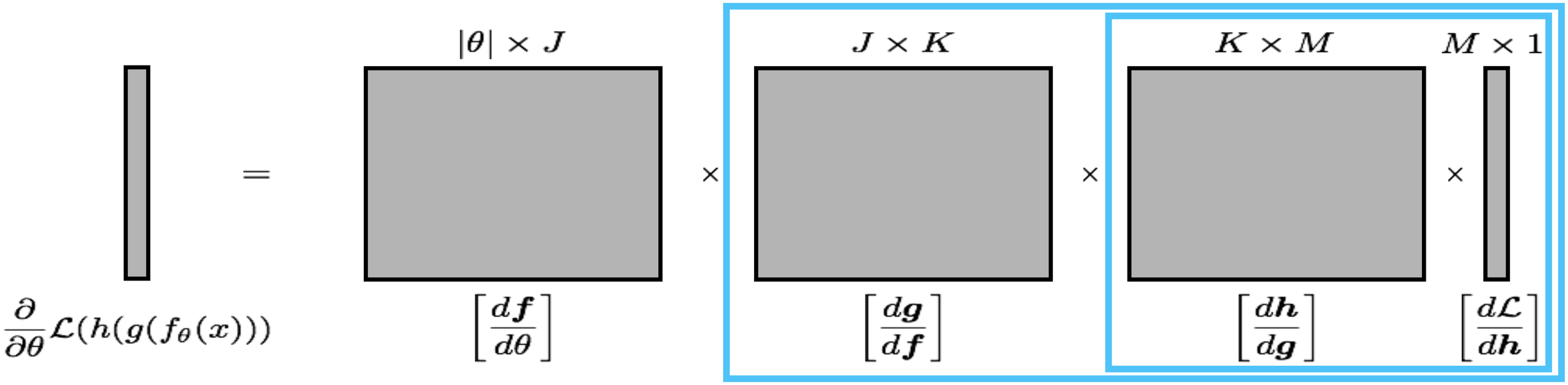
$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$



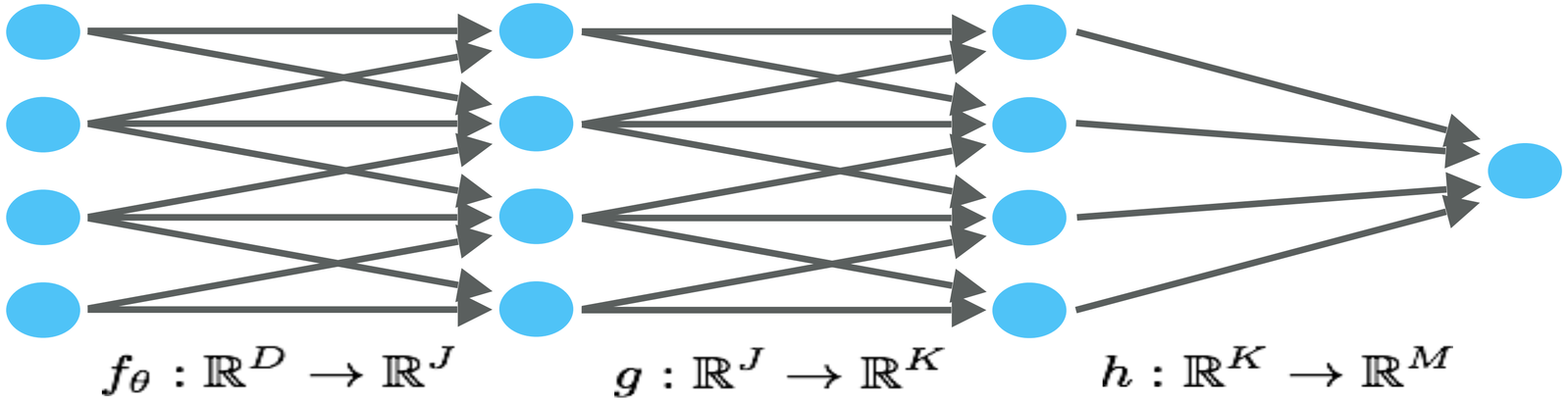
REVERSE MODE (SYMBOLIC VIEW)



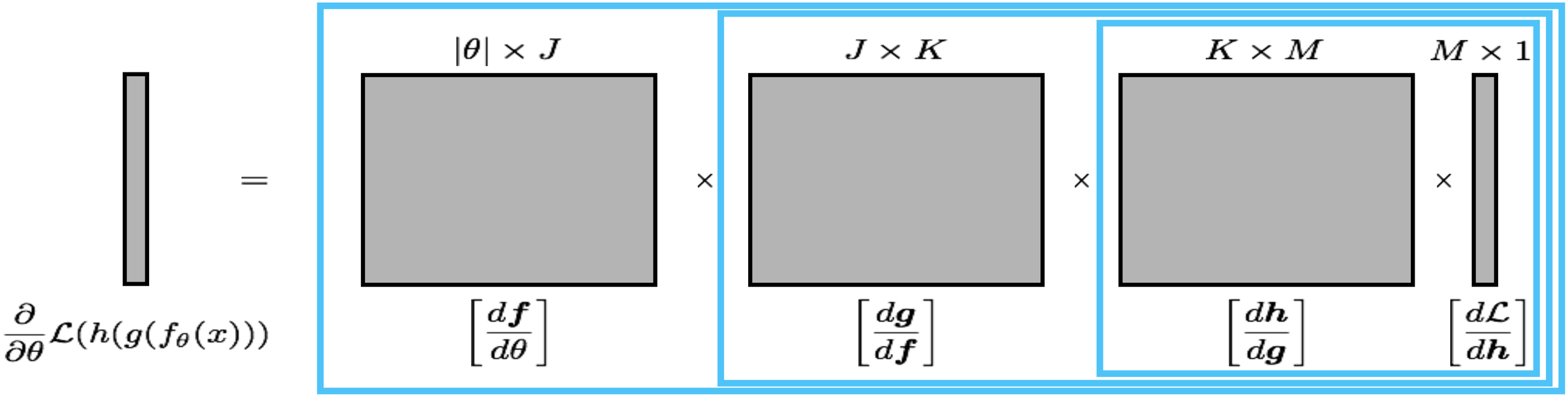
$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$



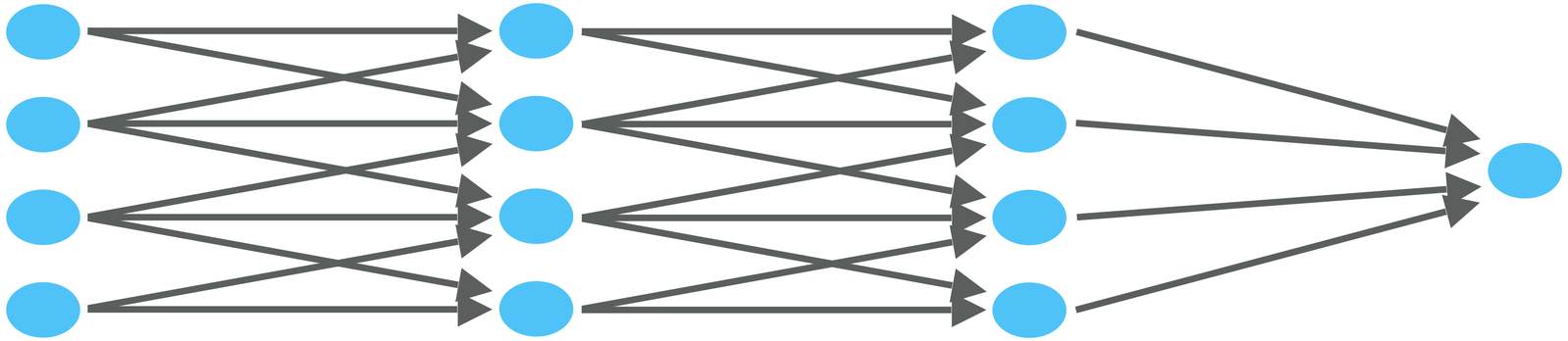
REVERSE MODE (SYMBOLIC VIEW)



$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$

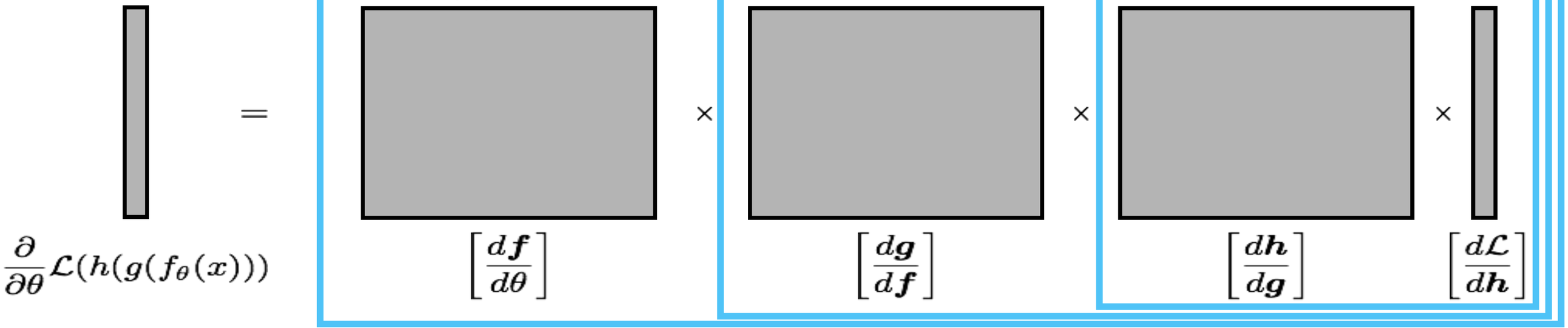


REVERSE MODE (SYMBOLIC VIEW)



Right to left:

$$O(KM + JK + \theta J)$$



REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

From Baydin 2016

MIA

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

b = 2

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

b = 2

c = 1

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

dddd = 1

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

dddd = 1

ddda = dd * math.sin(b) = 0.909

From Baydin 2016

REVERSE MODE (PROGRAM VIEW)

Right-to-left evaluation of partial derivatives (the right thing to do for optimization)

```
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
print(f(3,2,1))
```

```
2.727892280477
```

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

return 2.728

a = 3

b = 2

c = 1

d = a * math.sin(b) = 2.728

dddd = 1

ddda = dd * math.sin(b) = 0.909

return 0.909, 2.728

From Baydin 2016

A trainable neural network in torch-autograd

Any numeric function can go here

These two fn's are split only for clarity

This is the API >

This is a how the parameters are updated

```lua
torch = require 'torch'
params = {
   W = {torch.randn(64*64,50),torch.randn(50,4)},
   b = {torch.randn(64*64), torch.randn(4)}
}

function neuralNetwork(params, image)
   local h1 = torch.tanh(image*params.W[1] + params.b[1])
   local h2 = torch.tanh(h1*params.W[2] + params.b[2])
   return torch.log(torch.sum(torch.exp(h2)))
end

function loss(params, image, trueLabel)
   local prediction = neuralNetwork(params, image)
   return torch.sum(torch.pow(prediction-trueLabel,2))
end

grad = require 'autograd'
dloss = grad(loss)

for _,datapoint in dataset() do
   -- Calculate our gradients
   local gradients = dloss(params, datapoint.image, datapoint.label)
   -- Update parameters
   for i=1,#params.W do
      params.W[i] = params.W[i] - 0.01*gradients.W[i]
      params.b[i] = params.b[i] - 0.01*gradients.b[i]
   end
end
```

A trainable neural network in torch-autograd

Any numeric function can go here

These two fn's are split only for clarity

This is the API >

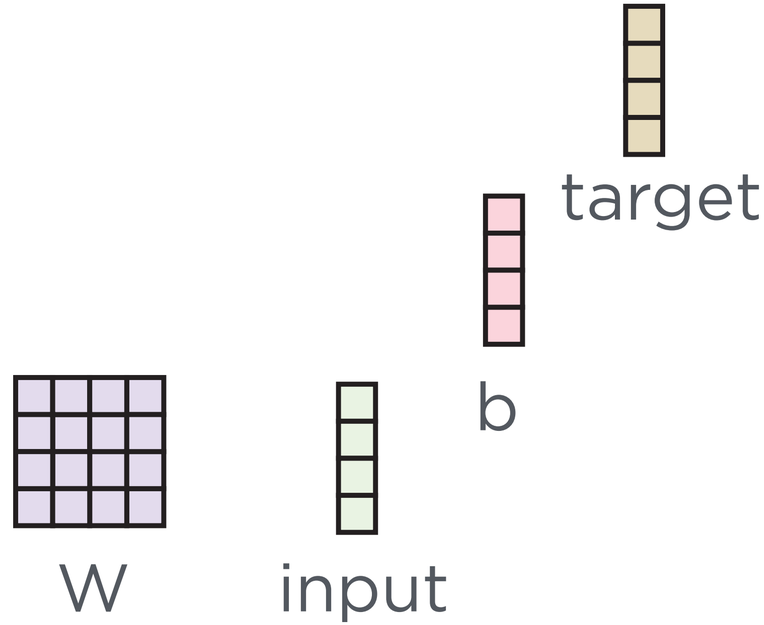
This is a how the parameters are updated

```lua
torch = require 'torch'
params = {
   W = {torch.randn(64*64,50),torch.randn(50,4)},
   b = {torch.randn(64*64), torch.randn(4)}
}

function neuralNetwork(params, image)
   local h1 = torch.tanh(image*params.W[1] + params.b[1])
   local h2 = torch.tanh(h1*params.W[2] + params.b[2])
   return torch.log(torch.sum(torch.exp(h2)))
end

function loss(params, image, trueLabel)
   local prediction = neuralNetwork(params, image)
   return torch.sum(torch.pow(prediction-trueLabel,2))
end

grad = require 'autograd'
dloss = grad(loss)

for _,datapoint in dataset() do
   -- Calculate our gradients
   local gradients = dloss(params, datapoint.image, datapoint.label)
   -- Update parameters
   for i=1,#params.W do
      params.W[i] = params.W[i] - 0.01*gradients.W[i]
      params.b[i] = params.b[i] - 0.01*gradients.b[i]
   end
end
```

WHAT'S ACTUALLY HAPPENING?

As torch code is run, we build up a compute graph

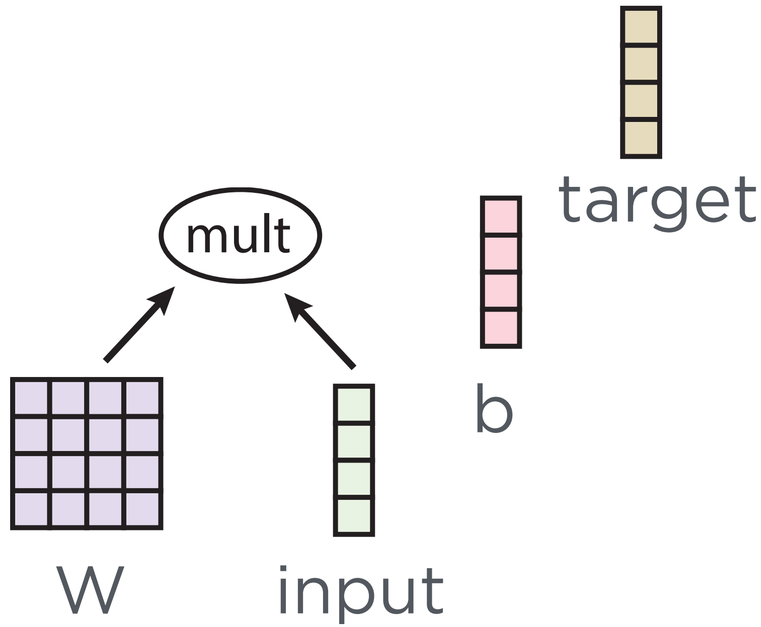
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



WHAT'S ACTUALLY HAPPENING?

As torch code is run, we build up a compute graph

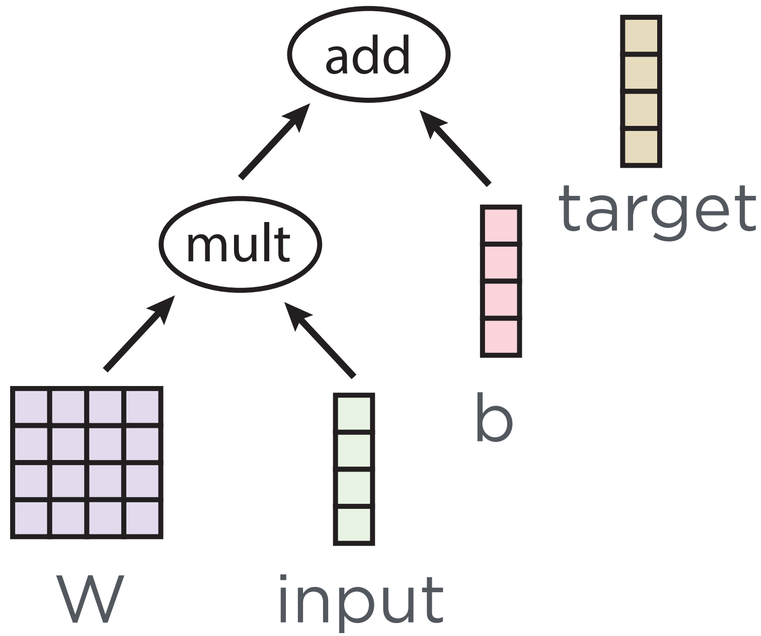
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



WHAT'S ACTUALLY HAPPENING?

As torch code is run, we build up a compute graph

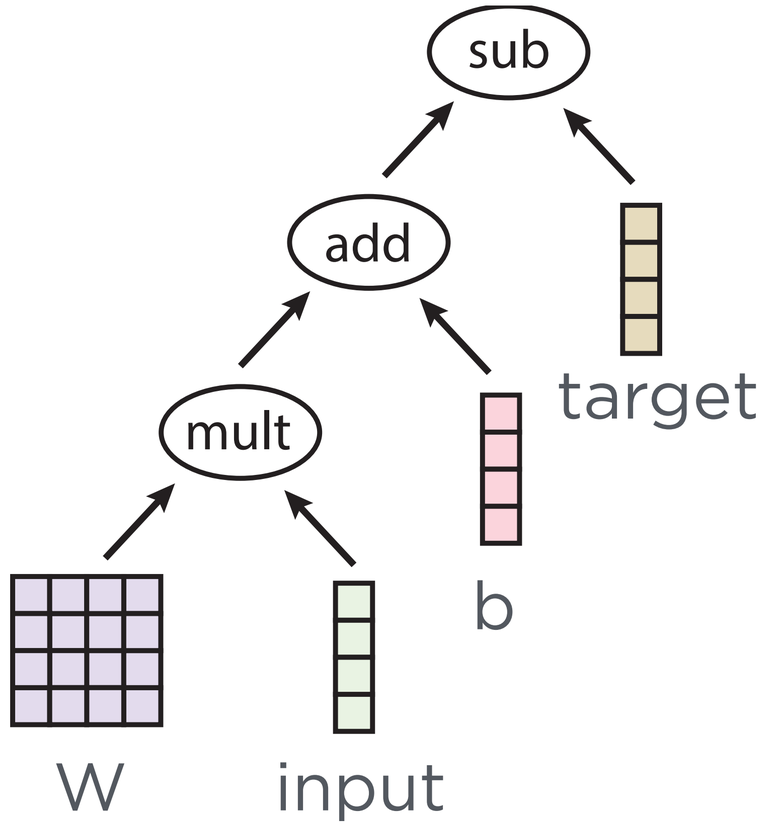
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



WHAT'S ACTUALLY HAPPENING?

As torch code is run, we build up a compute graph

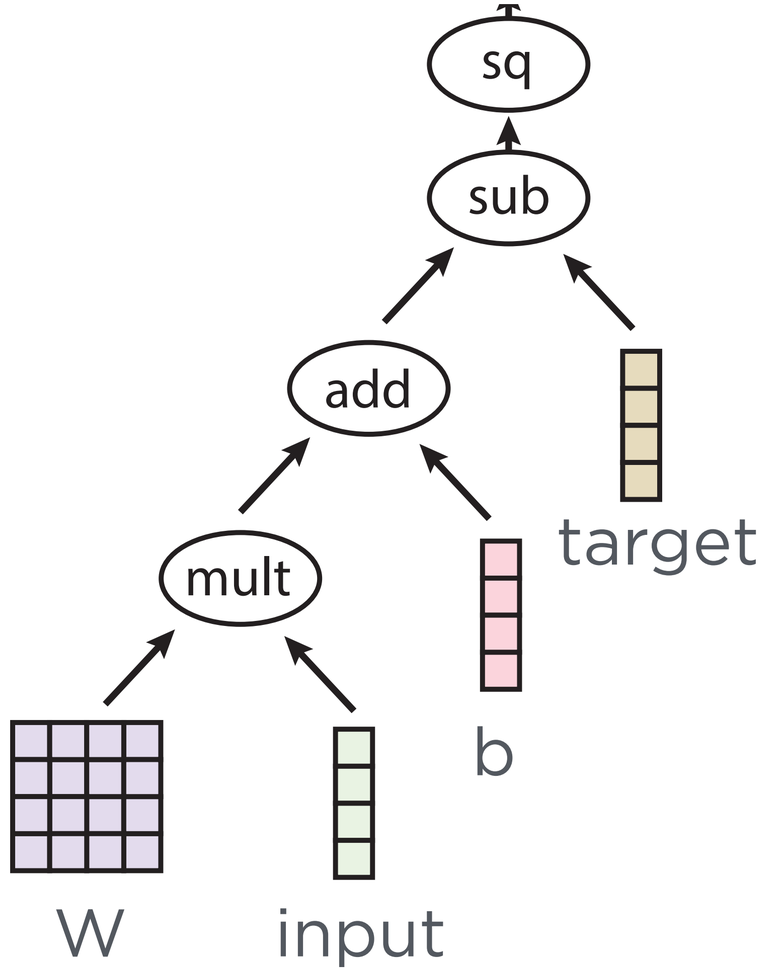
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



WHAT'S ACTUALLY HAPPENING?

As torch code is run, we build up a compute graph

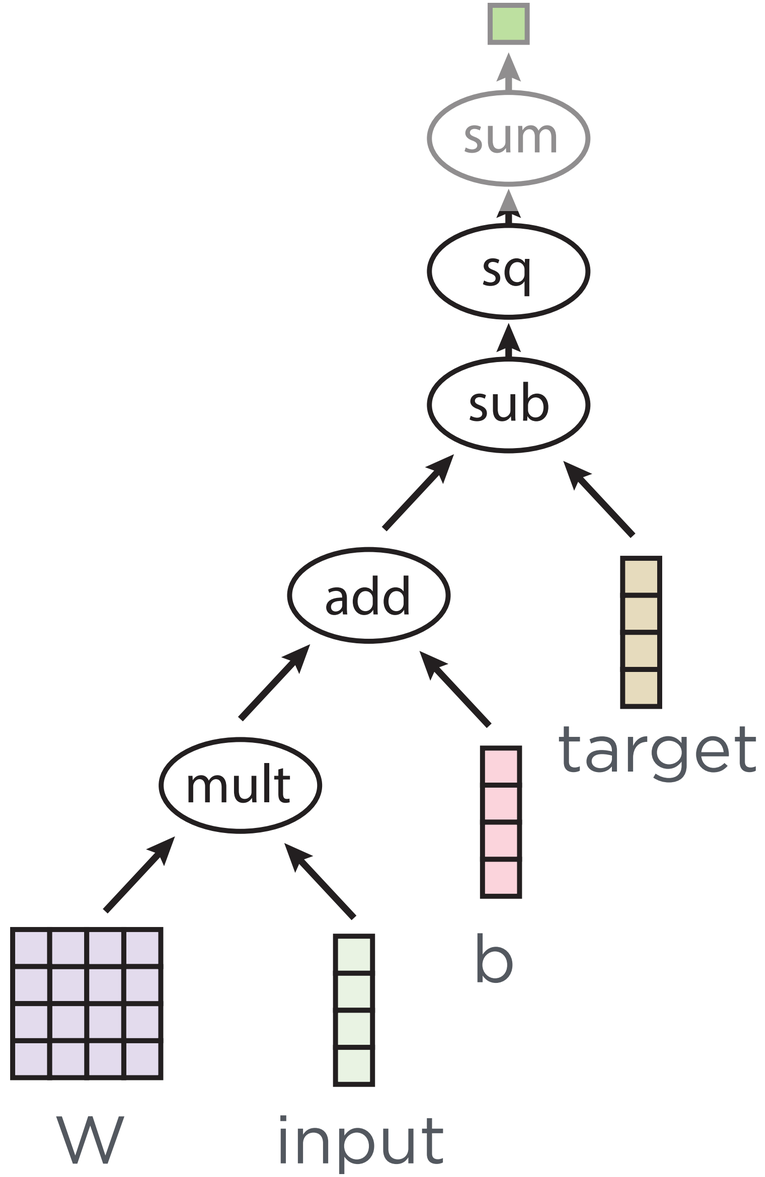
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



WHAT'S ACTUALLY HAPPENING?

As torch code is run, we build up a compute graph

```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



WE TRACK COMPUTATION VIA OPERATOR OVERLOADING

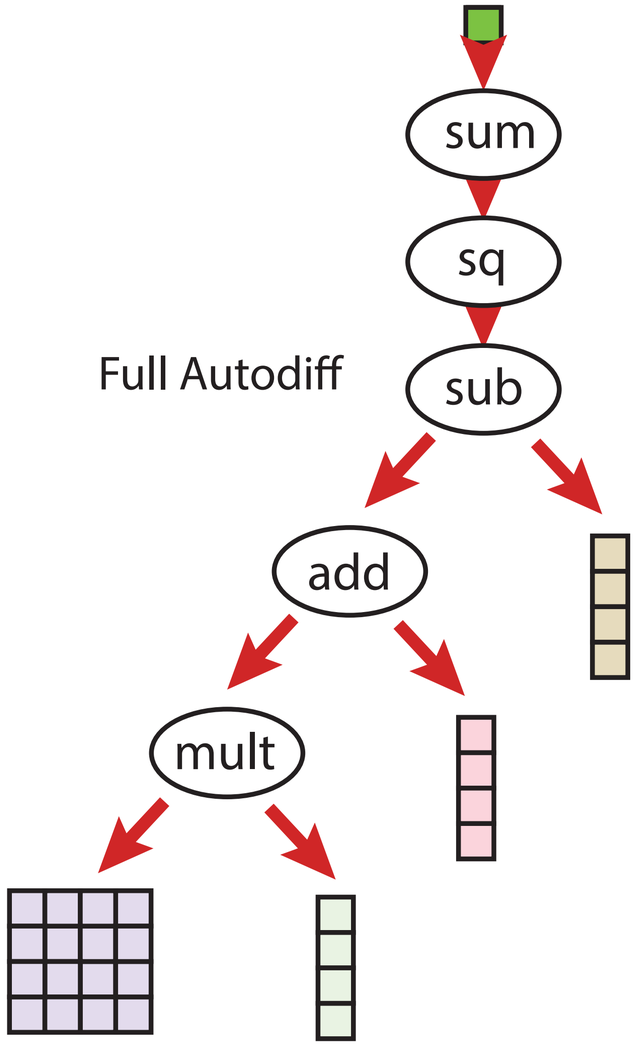
Linked list of computation forms a "tape" of computation

```lua
local origSum = torch.sum
torch.sum = function(arg)

   -- Check if the argument has been used before in an overloaded function
   if not isNodeType(arg) then
      return origSum(arg)
   else
      -- Run the function
      local outputVal = origSum(unpackNode(arg))

      -- Build a data structure that will track computaiton via linked list
      local outputNode = {fn=origSum,parent=arg,val=outputVal}
   end
end

-- Now overload all other numeric functions...
-- sin,cos,tan,sinh,cosh,tanh,add,sub,mul,div,pow
-- select,narrow,size,new,zeros, ...
```

WHAT'S ACTUALLY HAPPENING?

When it comes time to evaluate partial derivatives, we just have to look up the partial derivatives from a table

```
gradients[torch.sqrt] = {
   function(g, ans, x) return torch.cmul(torch.cmul(g,0.5), torch.pow(x,-0.5)) end
})
gradients[torch.sin] = {
   function(g, ans, x) return torch.cmul(g, torch.cos(x)) end
})
gradients[torch.cos] = {
   function(g, ans, x) return torch.cmul(g, -torch.sin(x)) end
})
gradients[torch.tan] = {
   function(g, ans, x) return torch.cdiv(g, torch.pow(torch.cos(x), 2.0)) end
})
gradients[torch.log] = {
   function(g, ans, x) return torch.cdiv(g,x) end
})
```



We can then calculate the derivative of the loss w.r.t. inputs via the chain rule!

AUTOGRAD EXAMPLES

Autograd gives you derivatives of numeric code, without a special mini-language

```
-- Arithmetic is no problem
grad = require 'autograd'
function f(a,b,c)
    return a + b * c
end
df = grad(f)
da, val = df(3.5, 2.1, 1.1)
print("Value: "..val)
print("Gradient: "..da)
```

```
Value: 5.81
Gradient: 1
```

AUTOGRAD EXAMPLES

Control flow, like if-statements, are handled seamlessly

```
-- If statements are no problem
grad = require 'autograd'
function f(a,b,c)
    if b > c then
        return a * math.sin(b)
    else
        return a + b * c
    end
end
g = grad(f)
da, val = g(3.5, 2.1, 1.1)
print("Value: "..val)
print("Gradient: "..da)
```

```
Value: 3.0212327832711
Gradient: 0.86320936664887
```

AUTOGRAD EXAMPLES

Scalars are good for demonstration, but autograd is most often used with tensor types

```
-- Of course, works with tensors
grad = require 'autograd'
function f(a,b,c)
    if torch.sum(b) > torch.sum(c) then
        return torch.sum(torch.cmul(a,torch.sin(b)))
    else
        return torch.sum(a + torch.cmul(b,c))
    end
end
g = grad(f)
a = torch.randn(3,3)
b = torch.eye(3,3)
c = torch.randn(3,3)
da, val = g(a,b,c)
print("Value: "..val)
print("Gradient: ")
print(da)
```

```
Value: 0.40072414956087
Gradient:
 0.8415  0.0000  0.0000
 0.0000  0.8415  0.0000
 0.0000  0.0000  0.8415
[torch.DoubleTensor of size 3x3]
```

AUTOGRAD EXAMPLES

Autograd shines if you have dynamic compute graphs

```
-- Autograd for loop
function f(a,b)
    for i=1,b do
        a = a*a
    end
    return a
end
g = grad(f)
da, val = g(3,2)
print("Value: "..val)
print("Gradient: "..da)
```

```
Value: 81
Gradient: 108
```

AUTOGRAD EXAMPLES

Recursion is no problem.
Write numeric code as you ordinarily would, autograd handles the gradients

```
-- Autograd recursive function
function f(a,b)
    if b == 0 then
        return a
    else
        return f(a*a,b-1)
    end
end
g = grad(f)
da, val = g(3,2)
print("Value: "..val)
print("Gradient: "..da)
```

```
Value: 81
Gradient: 108
```

AUTOGRAD EXAMPLES

Need new or tweaked partial derivatives? Not a problem.

```
-- New ops aren't a problem
function f(a)
    return torch.sum(torch.floor(torch.pow(a,3)))
end
g = grad(f)
da, val = g(torch.eye(3))
print("Value: "..val)
print("Gradient:")
print(da)
```

```
Value: 3
Gradient:
 0  0  0
 0  0  0
 0  0  0
[torch.DoubleTensor of size 3x3]
```

AUTOGRAD EXAMPLES

Need new or tweaked partial derivatives? Not a problem.

```
-- New ops aren't a problem
grad = require 'autograd'
special = {}
special.floor = function(x) return torch.floor(x) end
-- Overload our new mini-module, called "special"
grad.overload.module("special",special,function(module)
    -- Define a gradient for the member function "floor"
    module.gradient("floor", {
                -- Here's our new partial derivative
                -- (if we had two arguments,
                -- we'd define two functions)
                function(g,ans,x)
                    return g
                end
            })
    end)
```

AUTOGRAD EXAMPLES

Need new or tweaked partial derivatives? Not a problem.

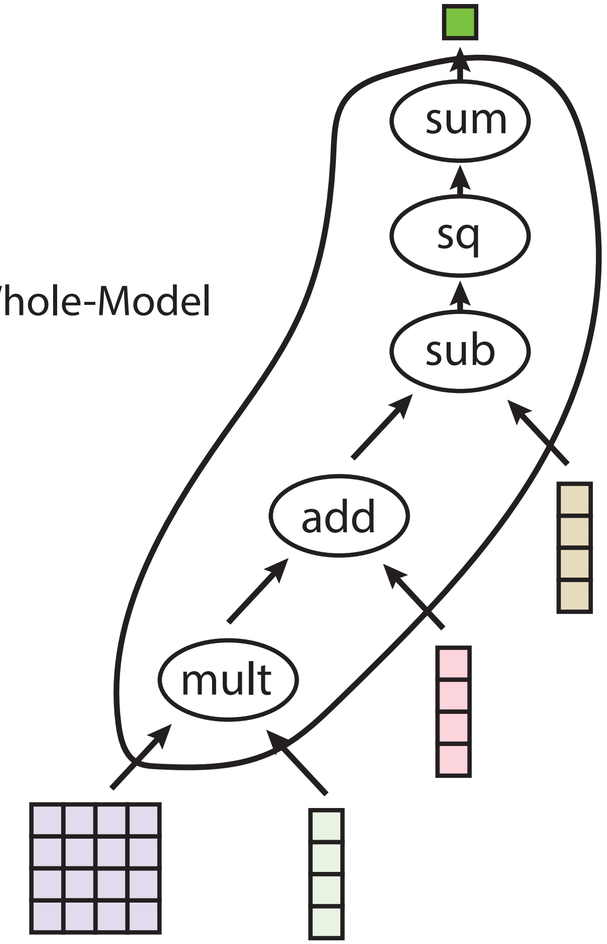
```
function f(a)
    return torch.sum(special.floor(torch.pow(a,3)))
end
g = grad(f)
da, val = g(torch.eye(3))
print("Value: "..val)
print("Gradient:")
print(da)
```

```
Value: 3
Gradient:
 3  0  0
 0  3  0
 0  0  3
[torch.DoubleTensor of size 3x3]
```

SO WHAT DIFFERENTIATES N.NET LIBRARIES?

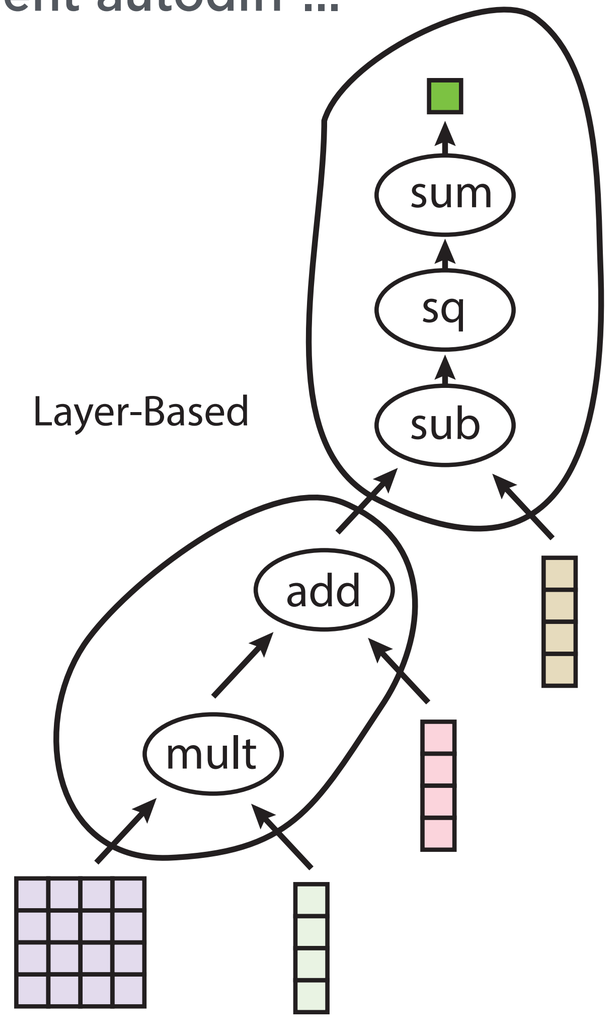
The granularity at which they implement autodiff ...

Whole-Model



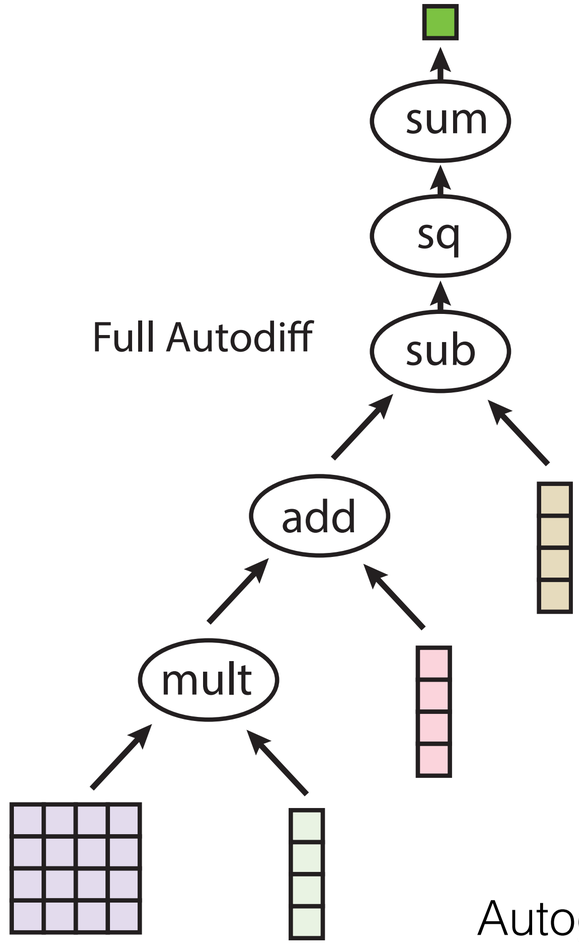
scikit-learn

Layer-Based



Torch NN
Keras
Lasagne

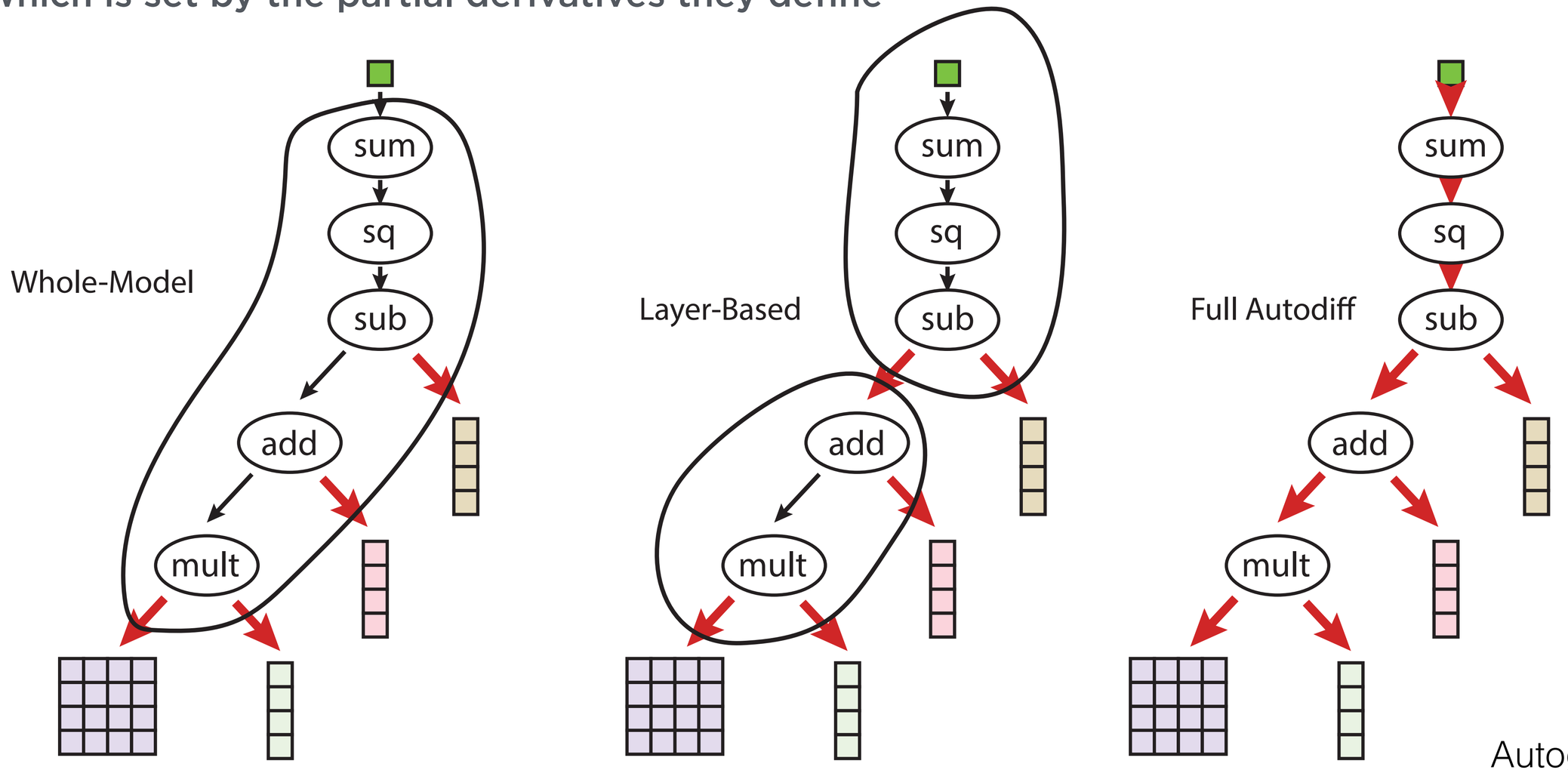
Full Autodiff



Autograd
Theano
TensorFlow

SO WHAT DIFFERENTIATES N.NET LIBRARIES?

... which is set by the partial derivatives they define



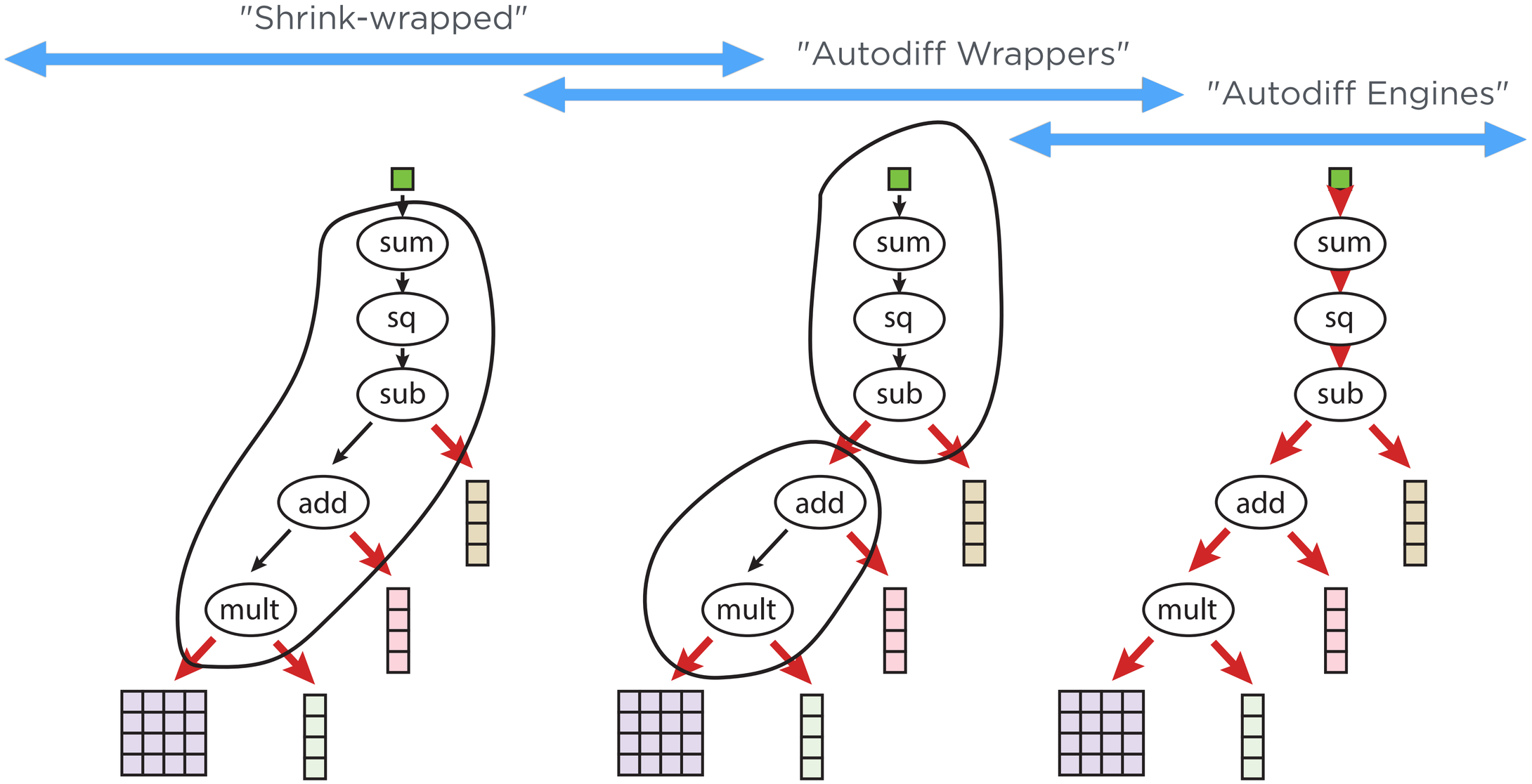
scikit-learn

Torch NN
Keras
Lasagne

Autograd
Theano
TensorFlow

SO WHAT DIFFERENTIATES N.NET LIBRARIES?

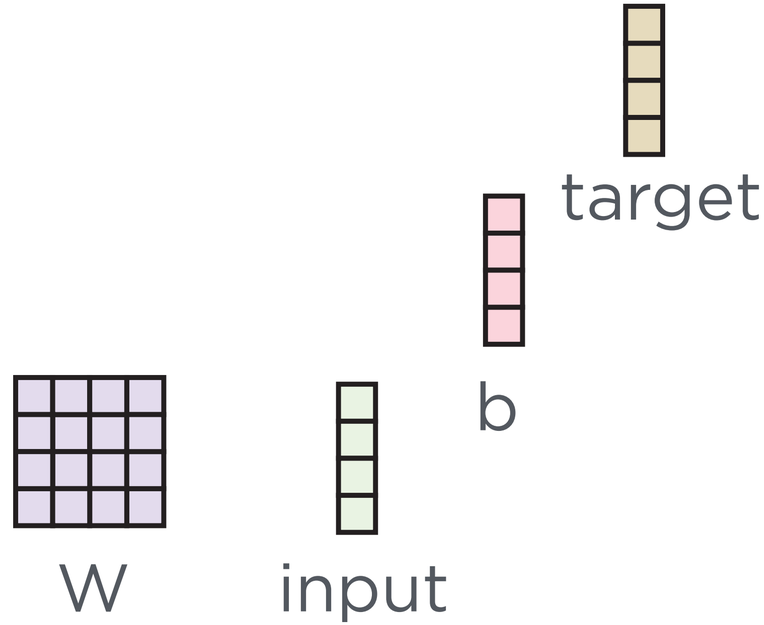
Do they actually implement autodiff, or do they wrap a separate autodiff library?



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

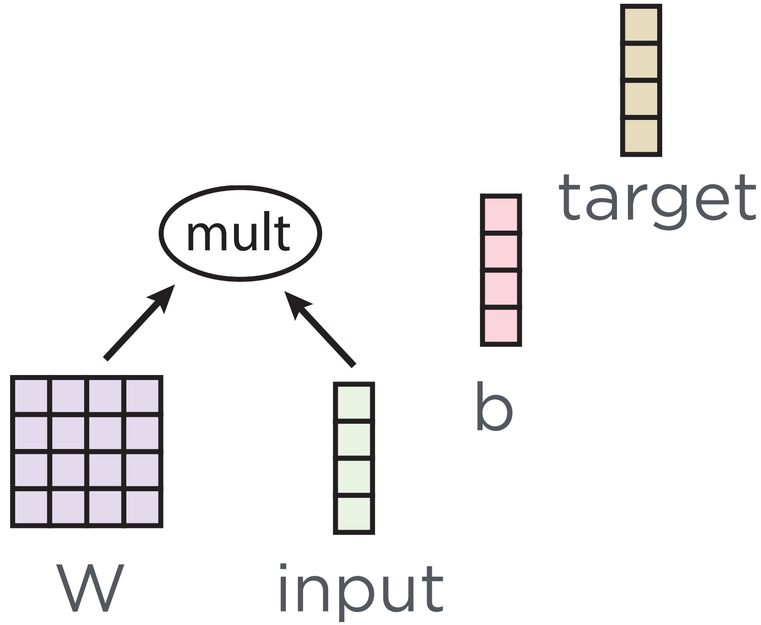
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

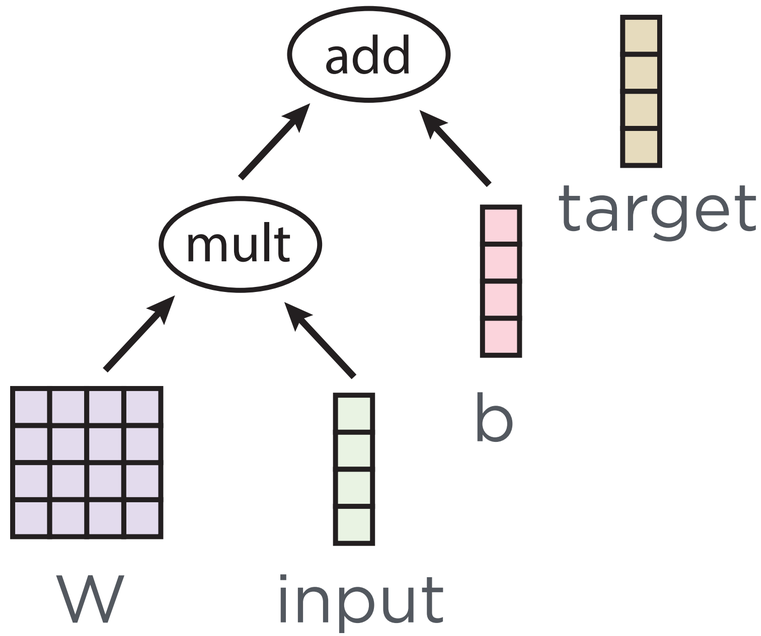
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

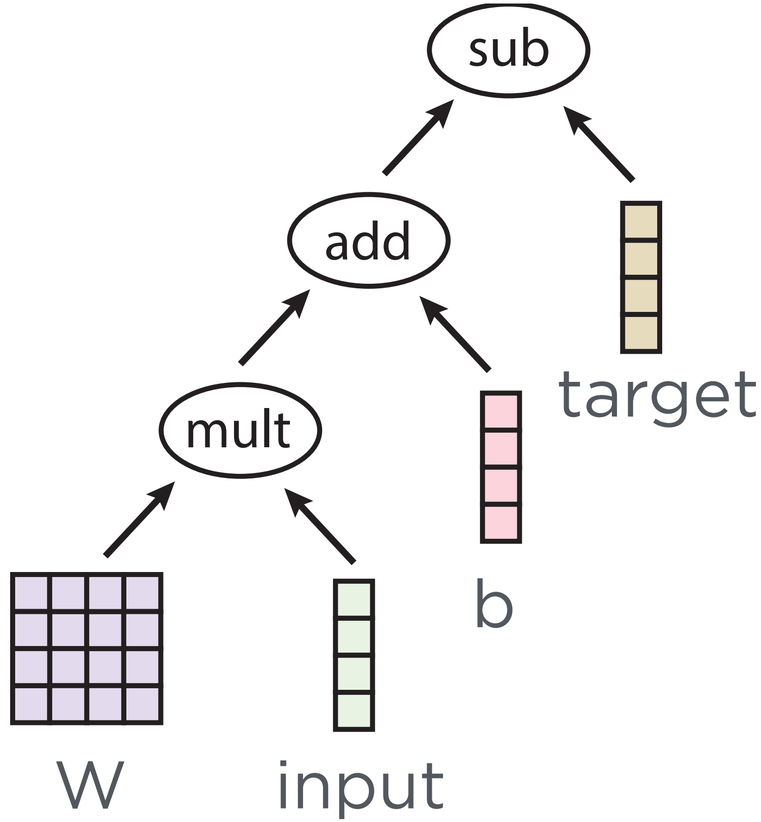
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

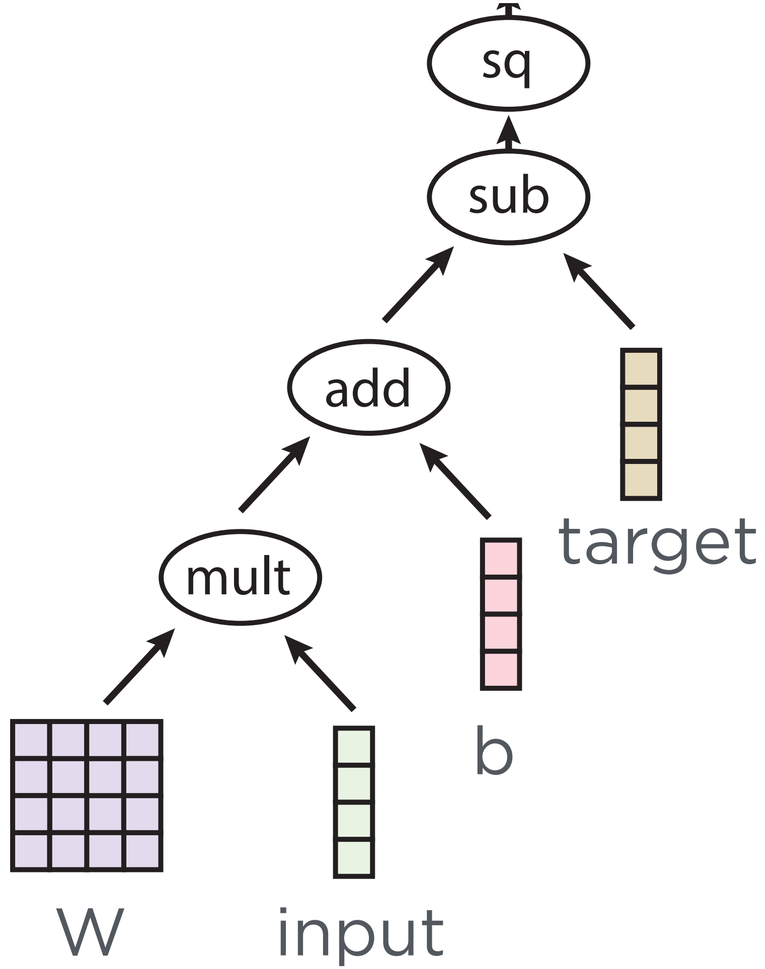
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

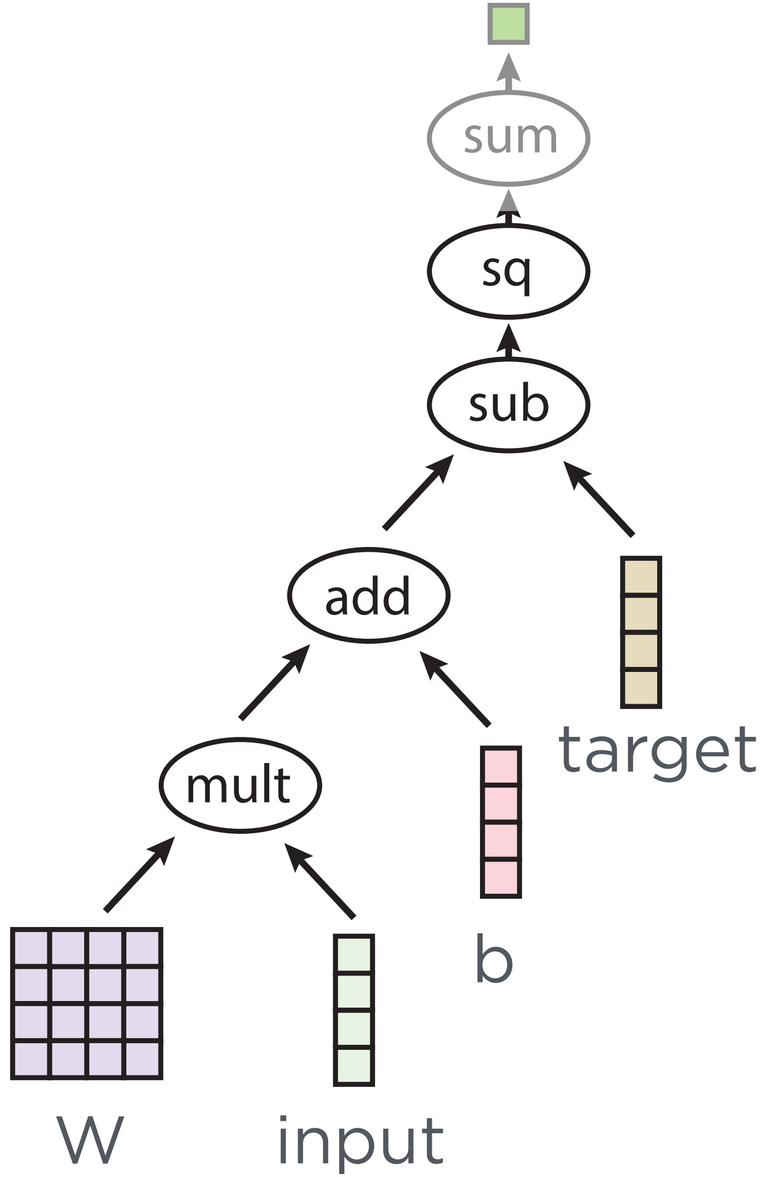
```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

```
1   params = {W=torch.randn(4,4),b=torch.randn(4)}
2   input = torch.randn(4)
3   target = torch.randn(4)
4   function simpleFn(params, input, target)
5       local h1 = params.W*input
6       local h2 = h1 + params.b
7       local h3 = h2 - target
8       local h4 = torch.pow(h3,2)
9       local h5 = torch.sum(h4)
10      return h5
11  end
```



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

Explicit	Ahead-of-Time	Just-in-Time
NN	TensorFlow	Autograd
Caffe	Theano	Chainer
No compiler optimizations, no dynamic graphs	Dynamic graphs can be awkward to work with	No compiler optimizations

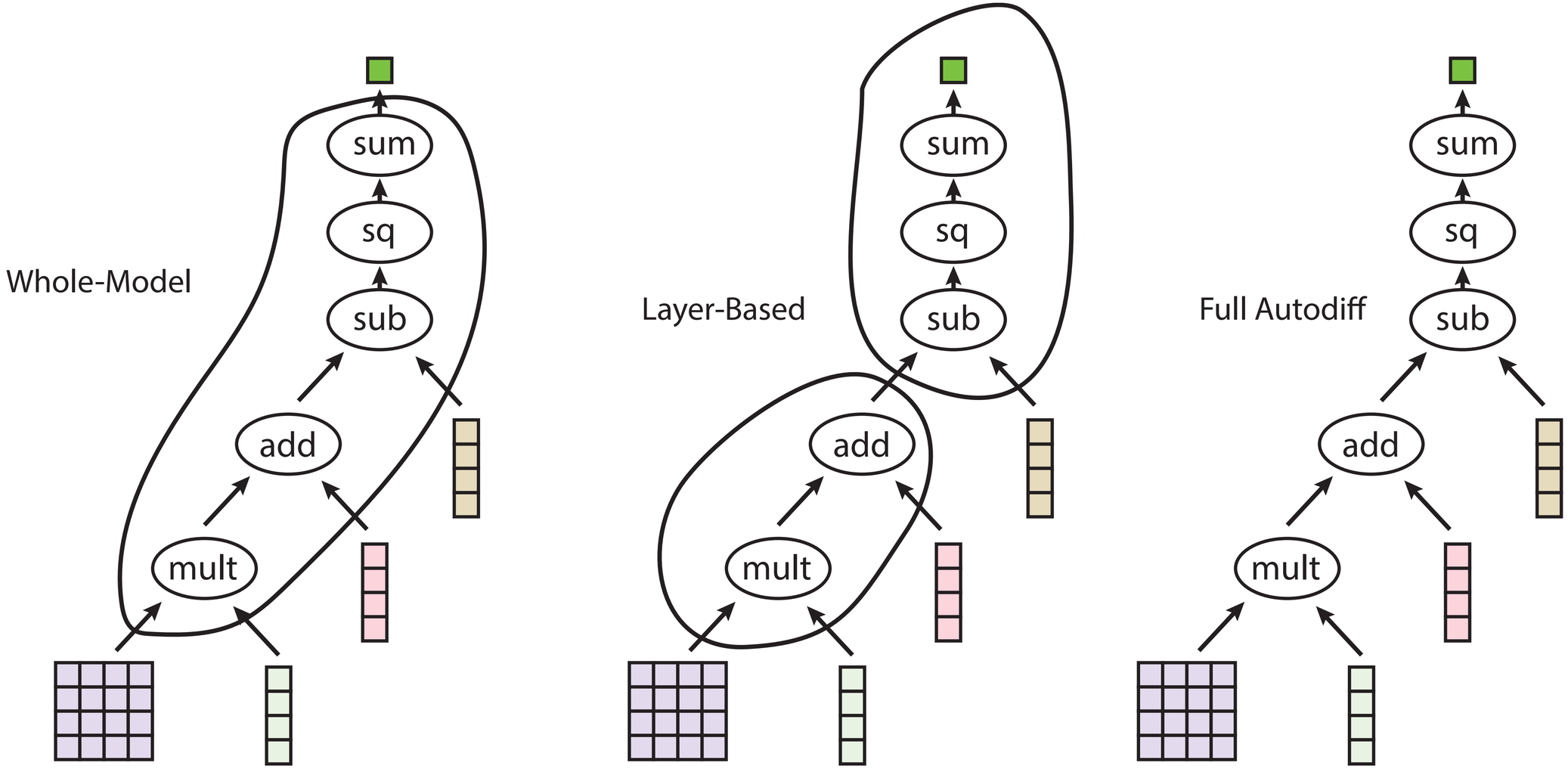
SO WHAT DIFFERENTIATES N.NET LIBRARIES?

What is the graph?

Static Dataflow	Hybrid	JIT Dataflow	Sea of Nodes (Click & Paleczny, 1995)
NN	TensorFlow	Autograd	
			?
Caffe	Theano	Chainer	
Ops are nodes, edges are data, graph can't change	Ops are nodes, edges data, but the runtime has to work hard for control flow	Ops are nodes, edges are data, graph can change freely	Control flow and dataflow merged in this AST representation

WE WANT NO LIMITS ON THE MODELS WE WRITE

Why can't we mix different levels of granularity?

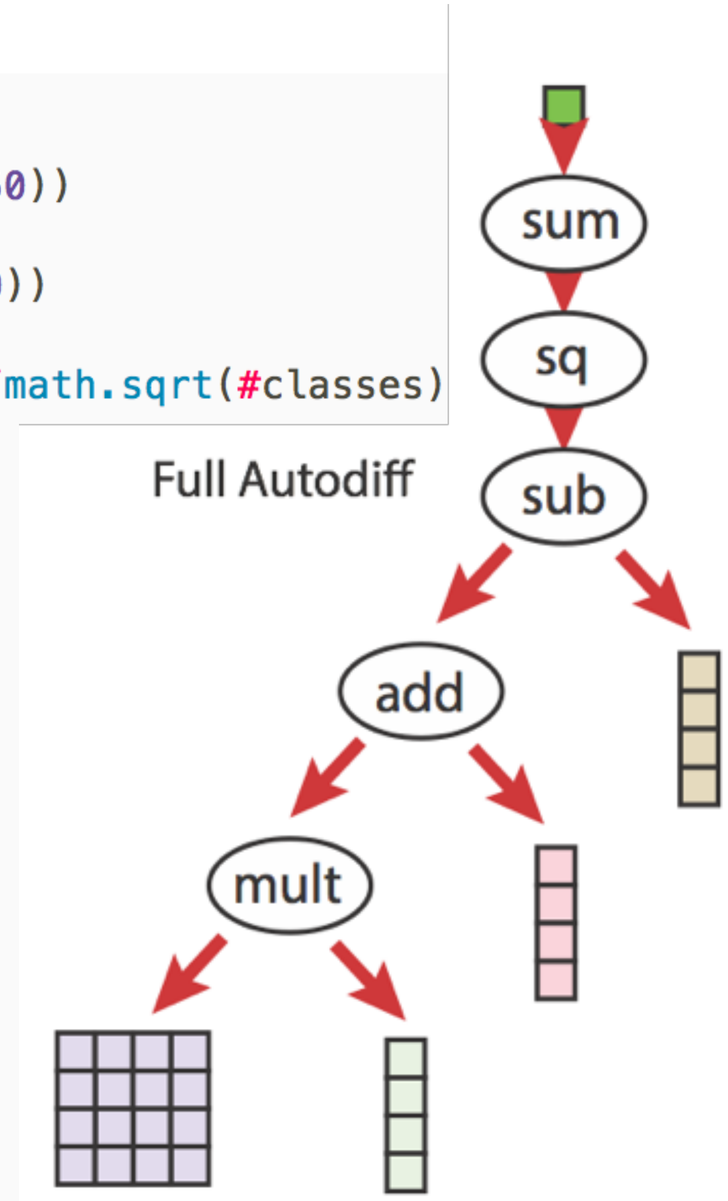


These divisions are usually the function of wrapper libraries

NEURAL NET THREE WAYS

The most granular — using individual Torch functions

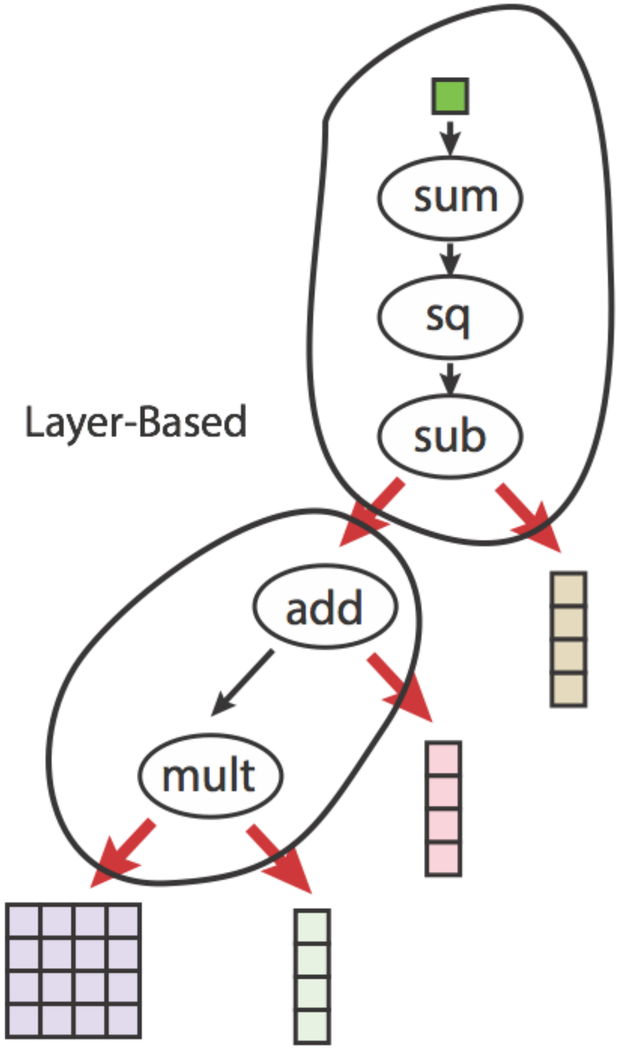
```
-- Define our parameters
local W1 = torch.FloatTensor(784,50):uniform(-1/math.sqrt(50),1/math.sqrt(50))
local B1 = torch.FloatTensor(50):fill(0)
local W2 = torch.FloatTensor(50,50):uniform(-1/math.sqrt(50),1/math.sqrt(50))
local B2 = torch.FloatTensor(50):fill(0)
local W3 = torch.FloatTensor(50,#classes):uniform(-1/math.sqrt(#classes),1/math.sqrt(#classes))
local B3 = torch.FloatTensor(#classes):fill(0)
local params = {
   W = {W1, W2, W3},
   B = {B1, B2, B3},
}

-- Define our neural net
local function mlp(params, input, target)
   local h1 = torch.tanh(input * params.W[1] + params.B[1])
   local h2 = torch.tanh(h1 * params.W[2] + params.B[2])
   local h3 = h2 * params.W[3] + params.B[3]
   local prediction = autograd.util.logSoftMax(h3)
   local loss = autograd.loss.logMultinomialLoss(prediction, target)
   return loss, prediction
end
```



NEURAL NET THREE WAYS

Composing pre-existing NN layers. If we need layers that have been highly optimized, this is good

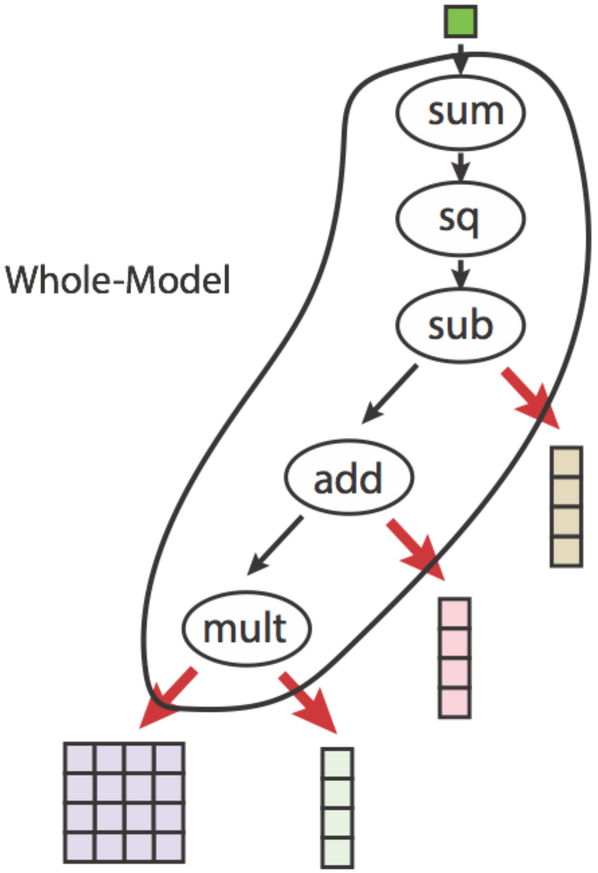
```
-- Define our layers and their parameters
local params = {}
local linear1, linear2, linear3, acts1, acts2, lsm, lossf
linear1, params.linear1 = autograd.nn.Linear(784, 50)
acts1 = autograd.nn.Tanh()
linear2,params.linear2 = autograd.nn.Linear(50, 50)
acts2 = autograd.nn.Tanh()
linear3, params.linear3 = autograd.nn.Linear(50,#classes)
lsm = autograd.nn.LogSoftMax()
lossf = autograd.nn.ClassNLLCriterion()

-- Tie it all together
local function mlp(params)
   local h1 = acts1(linear1(params.linear1, params.x))
   local h2 = acts2(linear2(params.linear2, h1))
   local h3 = linear3(params.linear3, h2)
   local prediction = lsm(h3)
   local loss = lossf(prediction, target)
   return loss, prediction
end
```



NEURAL NET THREE WAYS

We can also compose entire networks together (e.g. image captioning, GANs)

```
-- Grab the neural network all at once
local f,params = autograd.model.NeuralNetwork({
   inputFeatures = 784,
   hiddenFeatures = {50,#classes},
   classifier = true,
})
lsm = autograd.nn.LogSoftMax()
lossf = autograd.nn.ClassNLLCriterion()

-- Link the model and the loss
local loss = function(params, input, target)
   local prediction = lsm(f(params, input))
   local loss = lossf(prediction,target)
   return loss,prediction
end
```



NEURAL NETWORKS IN BIOLOGY

That have nothing to do with networks of neurons

Predicting:

- DNA Binding (e.g. Kelley et al @ MIA)
- Predicting molecular properties (Duvenuad et al @ MIA)
- Behavioral modeling (Johnson et al)
- DNA expression
- DNA methylation state
- Protein folding
- Image correction

IMPACT AT TWITTER

Prototyping without fear

- We try crazier, potentially high-payoff ideas more often, because autograd makes it essentially free to do so (can write "regular" numeric code, and automagically pass gradients through it)
- We use weird losses in production: large classification model uses a loss computed over a tree of class taxonomies
- Models trained with autograd running on large amounts of media at Twitter
- Often "fast enough”, no penalty at test time
- "Optimized mode" is nearly a compiler, but still a work in progress

OTHER AUTODIFF IDEAS

That haven't fully landed in machine learning yet

- **Checkpointing** — don't save all of the intermediate values. Recompute them when you need them (memory savings, potentially speedup if compute is faster than load/store, possibly good with pointwise functions like ReLU). MXNet I *think* is the first to implement this generally for neural nets.
- **Mixing forward and reverse mode** — called "cross-country elimination". No need to evaluate partial derivatives only in one direction! For diamond or hour-glass shaped compute graphs, perhaps dynamic programming can find the right order of partial derivative folding.
- **Stencils** — image processing (convolutions) and element-wise ufuncs can be phrased as stencil operations. More efficient general-purpose implementations of differentiable stencils needed (compute graphics does this, Guenter 2007, extending with DeVito et al 2016).
- **Source-to-source** — All neural net autodiff packages are either AOT or JIT graph construction, with operator overloading. The original autodiff (in FORTRAN, in the 80s) was source transformation. Considered gold-standard for performance in autodiff field. Challenge is control flow.
- **Higher-order gradients** — hessian = grad(grad(f)). Not many efficient implementations, need to take advantage of sparsity. Fully closed versions in e.g. autograd, DiffSharp, Hype.

YOU SHOULD BE USING IT

It's easy to try

```
# Install anaconda if you don't have it (instructions here for OS X)
wget http://repo.continuum.io/miniconda/Miniconda-latest-MacOSX-x86_64.sh
sh Miniconda-latest-MacOSX-x86_64.sh -b -p $HOME/anaconda

# Add anaconda to your $PATH
export PATH=$HOME/anaconda/bin:$PATH

# Install Lua & Torch
conda install lua=5.2 lua-science -c alexbw

# Available versions of Lua: 2.0, 5.1, 5.2, 5.3
# 2.0 is LuaJIT
```

YOU SHOULD BE USING IT

It's easy to try

- **Anaconda** is the de-facto distribution for scientific Python.
- **Works with Lua & Luarocks** now.
- https://github.com/alexbw/conda-lua-recipes

```
# Install anaconda if you don't have it (instructions here for OS X)
wget http://repo.continuum.io/miniconda/Miniconda-latest-MacOSX-x86_64.sh
sh Miniconda-latest-MacOSX-x86_64.sh -b -p $HOME/anaconda

# Add anaconda to your $PATH
export PATH=$HOME/anaconda/bin:$PATH

# Install Lua & Torch
conda install lua=5.2 lua-science -c alexbw

# Available versions of Lua: 2.0, 5.1, 5.2, 5.3
# 2.0 is LuaJIT
```

YOU SHOULD BE USING IT

It's easy to try

- **Anaconda** is the de-facto distribution for scientific Python.
- **Works with Lua & Luarocks** now.
- https://github.com/alexbw/conda-lua-recipes

```
# Install anaconda if you don't have it (instructions here for OS X)
wget http://repo.continuum.io/miniconda/Miniconda-latest-MacOSX-x86_64.sh
sh Miniconda-latest-MacOSX-x86_64.sh -b -p $HOME/anaconda

# Add anaconda to your $PATH
export PATH=$HOME/anaconda/bin:$PATH

# Install Lua & Torch
conda install lua=5.2 lua-science -c alexbw

# Available versions of Lua: 2.0, 5.1, 5.2, 5.3
# 2.0 is LuaJIT
```

QUESTIONS?

Happy to help. Find me at:

@awiltsch

awiltschko@twitter.com

github.com/alexbw