

AUTOMATIC DIFFERENTIATION

AN ABSTRACTION FOR MACHINE LEARNING



ALEX WILTSCHKO

RESEARCH ENGINEER

TWITTER

ADVANCED TECHNOLOGY GROUP

@AWILTSCH



MATERIAL DEVELOPED WITH
HUGO LAROCHELLE
RYAN ADAMS



FIRST A BIT ABOUT

NEURAL NETWORKS IN NLP

THEN MORE ABOUT

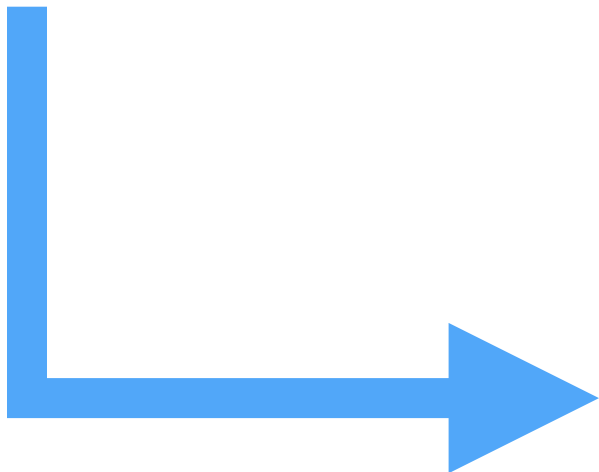
THE POWERFUL ABSTRACTION

THAT POWERS THEM



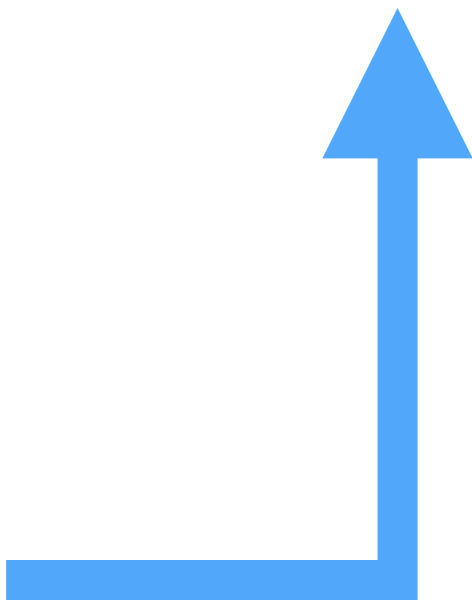
TURNING WORDS INTO NUMBERS

"This was a very good movie"

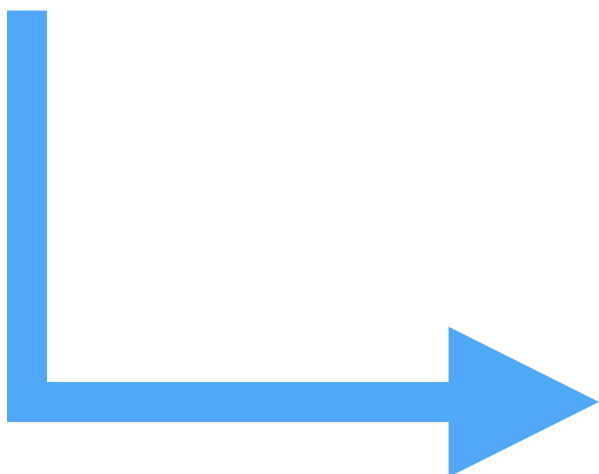


```
array([[ -0.52357694,  1.12979217,  1.26719889, ...,  0.73388794,
        -1.16502713, -1.23081258],
       [  0.90611596,  1.84870625, -0.01891658, ..., -1.2501873 ,
         0.87360637,  1.74786707],
       [  0.17103426, -0.50897518,  0.12341446, ..., -0.02443121,
         1.3935029 , -0.68816075],
       ...,
       [-0.9147852 ,  0.80959717, -1.2149677 , ..., -0.40054953,
        -1.12964909, -1.43887948],
       [-0.89976191, -1.75732653,  1.29460437, ..., -1.61950746,
         0.40906495,  0.36949282],
       [-0.37811107, -0.23104273, -1.00433316, ...,  0.60603038,
         2.04055542,  0.26861255]])
```

$p(\text{"Nice"}) = 0.93$

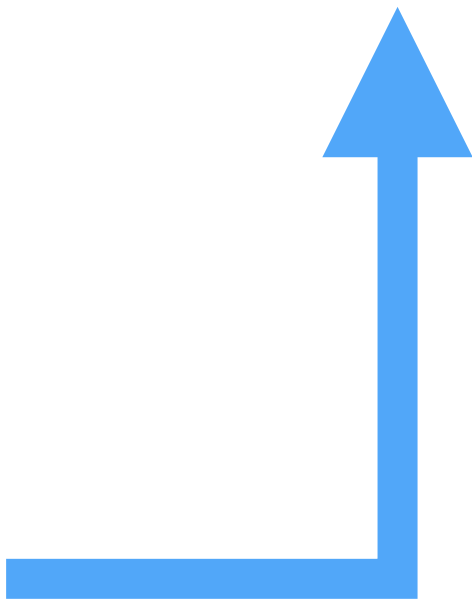


"What a disaster of a movie"



```
array([[ 1.43408359e+00, -1.14951576e+00,  1.48081682e-01, ...,
         3.15954797e-01,  2.44072829e+00,  1.24139221e+00],
       [  4.47487023e-01,  3.39139585e+00, -7.68169864e-02, ...,
        -2.33335039e+00, -1.09712552e+00,  1.30592139e+00],
       [ -2.27218218e-01,  1.15852345e+00,  4.62356615e-01, ...,
         9.38343095e-01,  7.53330730e-01,  8.17434705e-01],
       ...,
       [ -7.31496776e-01, -1.53018132e+00,  3.87623099e-01, ...,
        -2.20373032e-01,  2.95869429e-01,  1.13800609e+00],
       [  1.18416890e+00, -1.59414075e-01,  1.10052558e+00, ...,
        -3.26311286e-01, -6.44709655e-01, -2.51885786e-03],
       [ -1.87242824e+00,  1.13216335e-01, -2.48464419e-01, ...,
        -1.09979159e+00, -1.92920004e+00, -2.88726652e-01]])
```

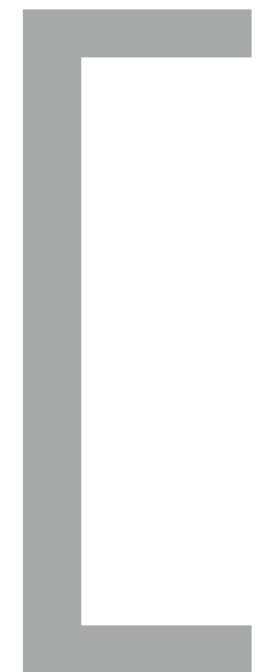
$p(\text{"Nice"}) = 0.076$



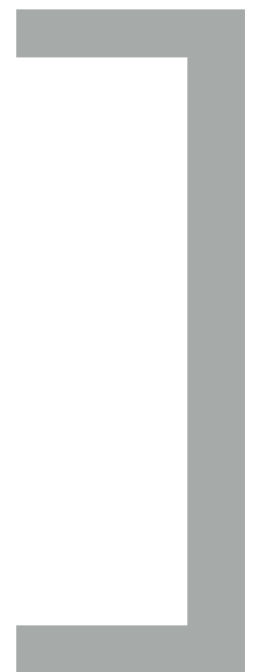
WORDS AS BIT-VECTORS

One-hot word representations are a natural first representation to try

Summer is over in Boston, I guess



summer	1	0	0	0	0	0	0	0	0
is	0	1	0	0	0	0	0	0	0
over	0	0	1	0	0	0	0	0	0
in	0	0	0	1	0	0	0	0	0
boston	0	0	0	0	1	0	0	0	0
i	0	0	0	0	0	1	0	0	0
guess	0	0	0	0	0	0	1	0	0



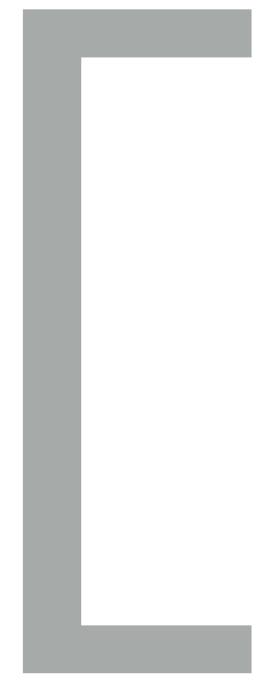
summer is over in boston i guess
bummer man



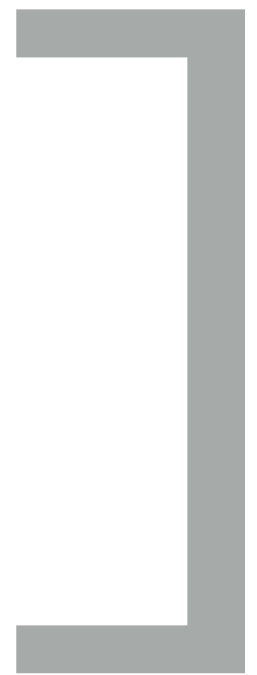
WORDS AS BIT-VECTORS

One-hot word representations are a natural first representation to try

bummer, summer is over



bummer	0	0	0	0	0	0	0	0	1	0
summer	1	0	0	0	0	0	0	0	0	0
is	0	1	0	0	0	0	0	0	0	0
over	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
summer	is	over	in	boston	i	guess	bummer	man		



WORD EMBEDDINGS

But!

bummer - summer = over - summer = 2

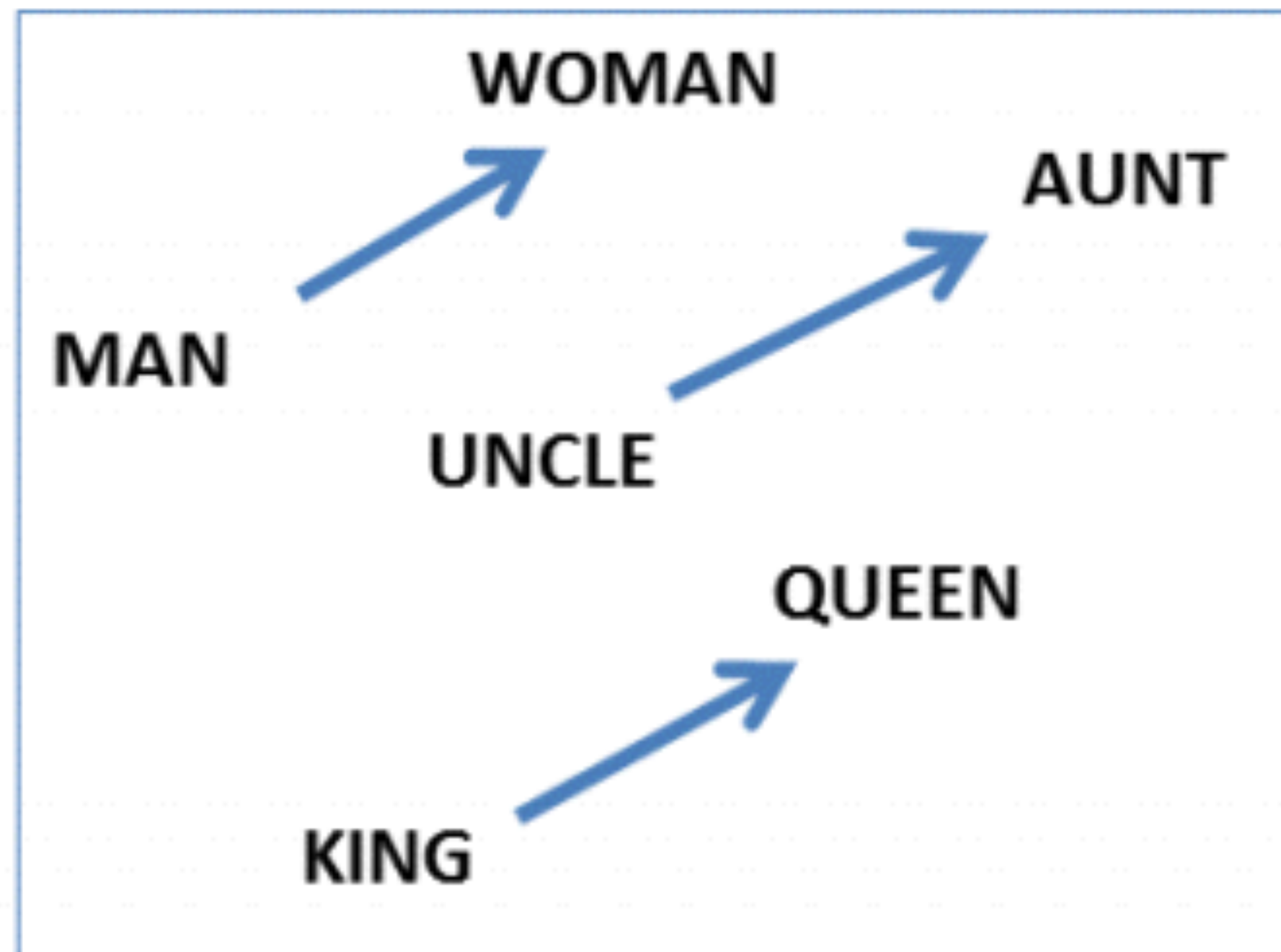
That makes zero sense!

Summer and bummer are not similar to each other.

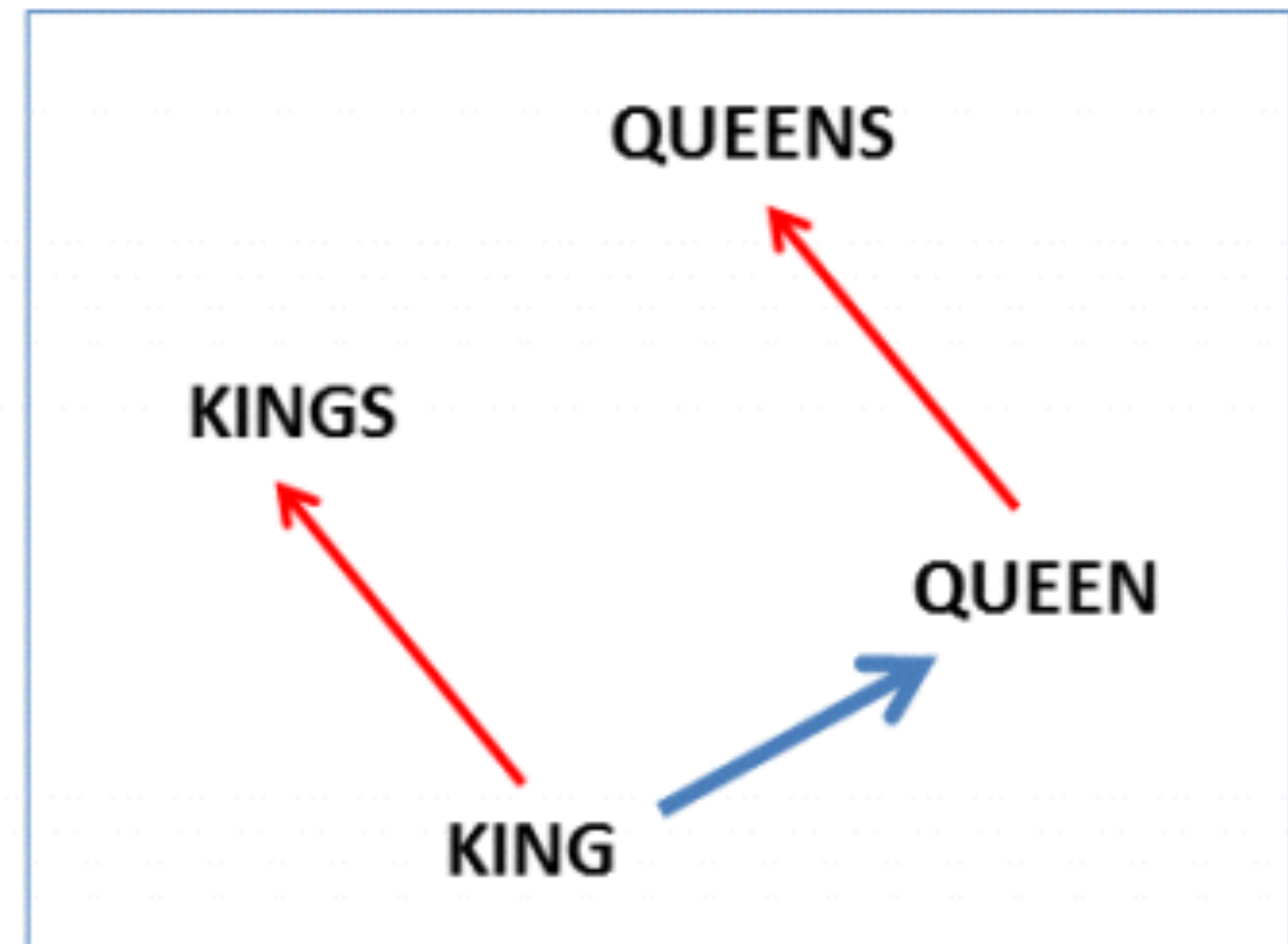


WORD EMBEDDINGS – WORD2VEC

king - queen = woman - man



kings - king + queen = queens



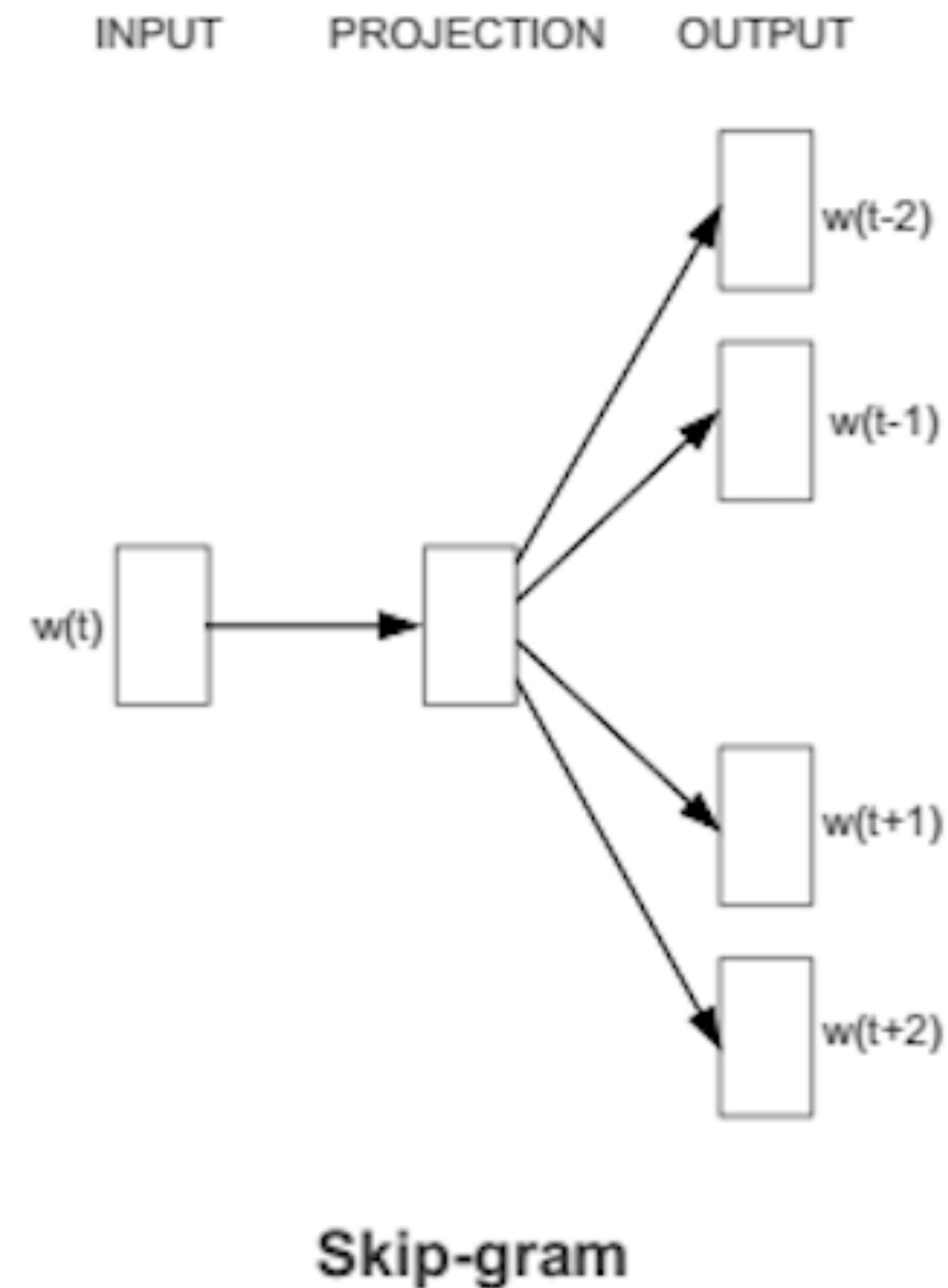
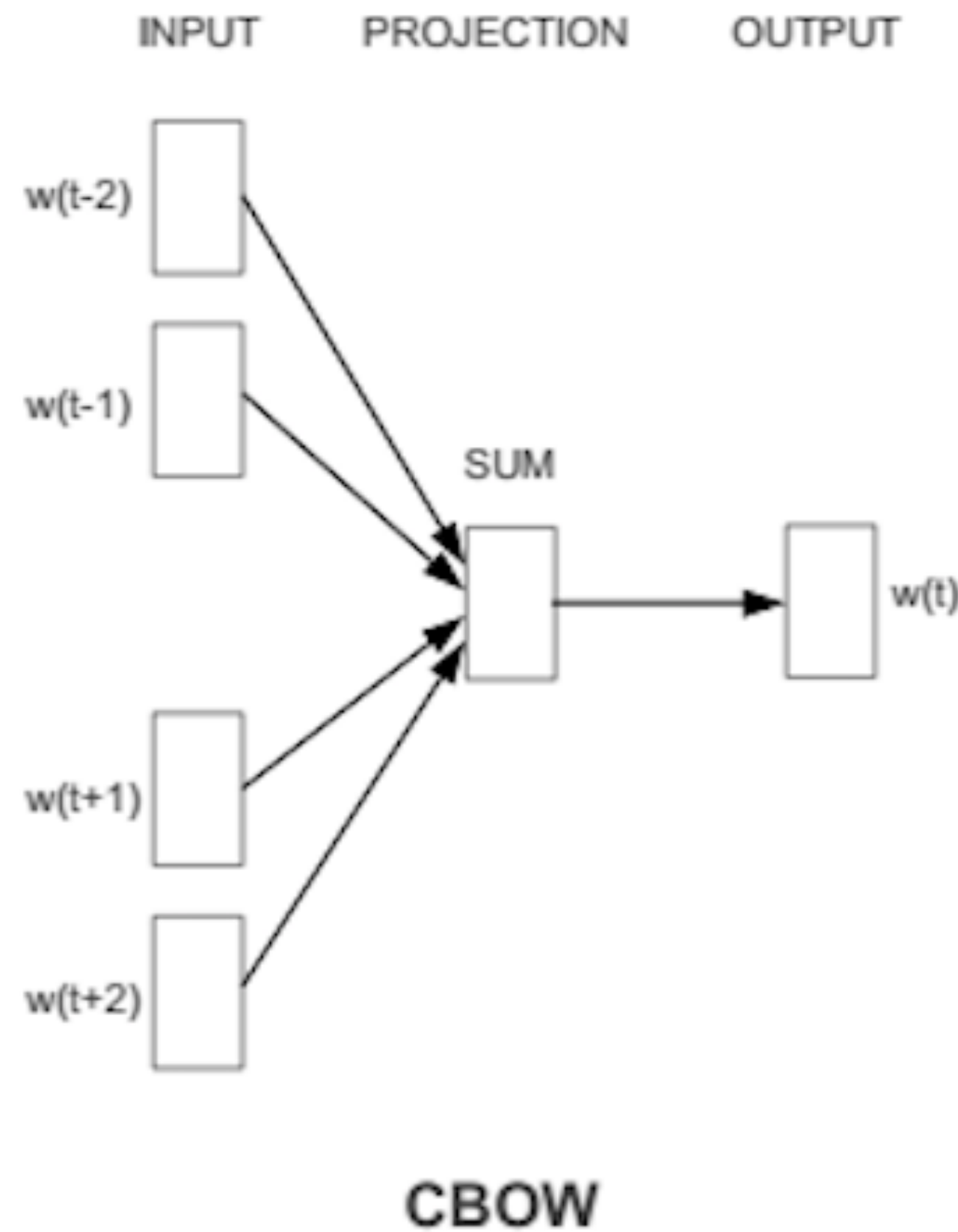
(Mikolov et al., NAACL HLT, 2013)



Image courtesy: <http://jxieeducation.com/2016-06-13/Document-Similarity-With-Word-Movers-Distance/>

LEARNING WORD EMBEDDINGS - WORD2VEC

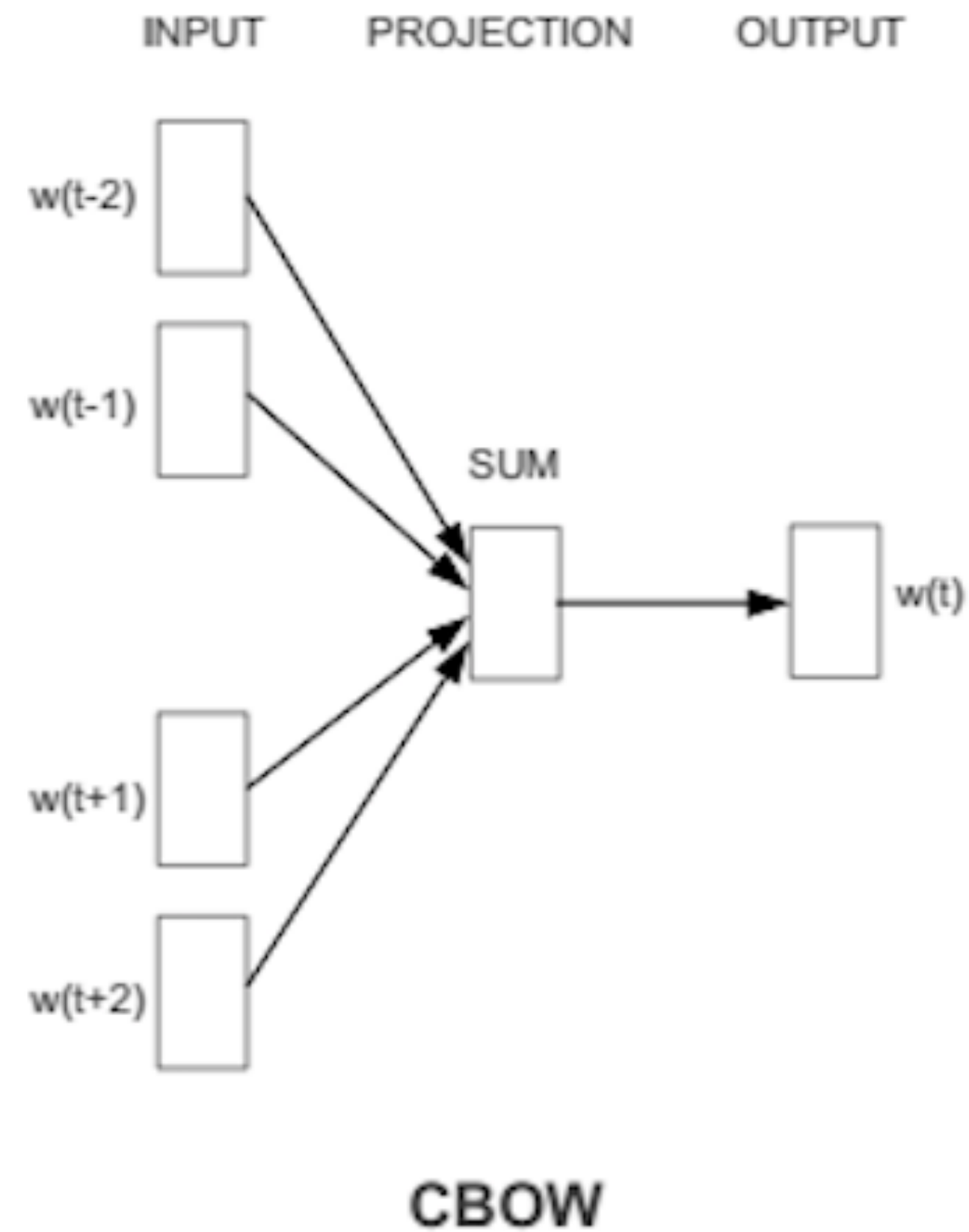
“You shall know a word by the company it keeps”
(J. R. Firth, 1957: 11)



Quote courtesy: Richard Socher

LEARNING WORD EMBEDDINGS - CBOW

“You shall know a word by the company it keeps”
(J. R. Firth, 1957: 11)

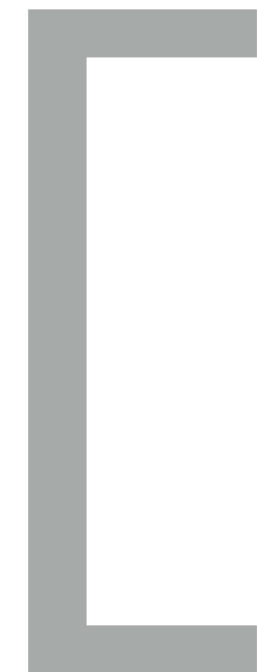


Quote courtesy: Richard Socher

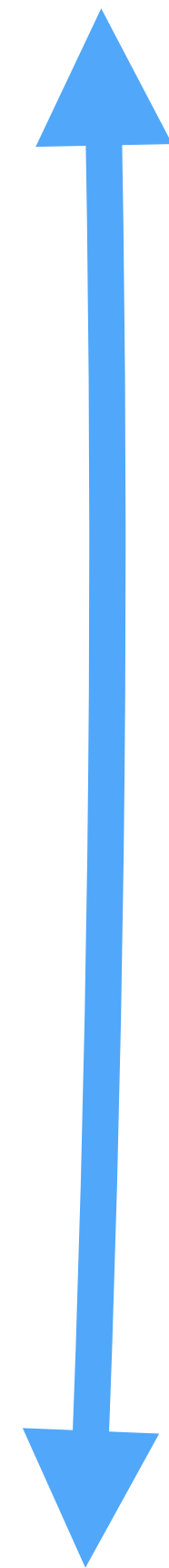
WORDS AS BIT-VECTORS

One-hot word representations are a natural first representation to try

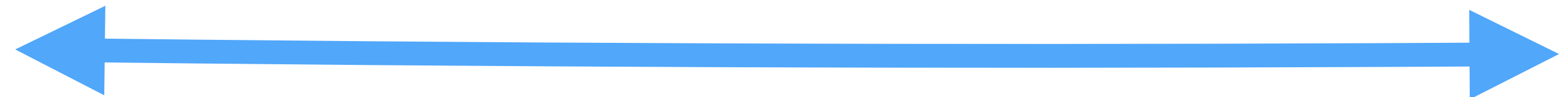
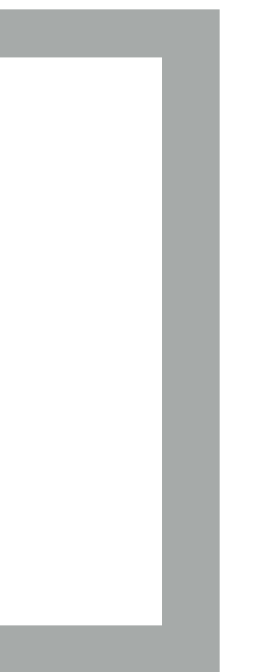
summer
is
over
in
boston
i
guess



word indices



0.3	-1.3	-0.6	-0.7	-1.6	0.4	-1.6	0.8	0.8
0.9	0.1	0.2	1.3	-0.5	0.6	-0.1	-1.3	-0.7
0.6	-0.7	0.3	-1.0	-0.9	-1.4	1.2	0.1	-0.2
-1.1	0.8	-1.5	2.6	0.9	-0.8	-0.5	0.7	-2.2
-0.0	0.8	-0.2	0.5	-0.4	-0.4	1.0	1.4	0.5
-1.1	-1.1	-1.6	-0.9	-1.4	0.6	-1.2	-0.8	3.1
-0.9	-0.5	-0.1	1.2	-2.4	0.1	0.4	0.9	0.0



word2vec dimensions

(typicall ~200. what do they mean? who knows!)



LEARNING WORD EMBEDDINGS — CBOW

We want to make this difference small (while having w's not be zero).

$$\| w[i] - (w[i-2] + w[i-1] + w[i+1] + w[i+2]) \|$$



LEARNING WORD EMBEDDINGS — CBOW

We want to make this difference small (while having w's not be zero).

$$\| w[i] - (w[i-2] + w[i-1] + w[i+1] + w[i+2]) \|$$

How? (also PS, that's not the real CBOW loss, sorry!)

CBOW is like a tiny neural network!

All neural networks are trained using something called
"Backpropagation"

We need derivatives!

Backprop is the chain rule of calculus for computing derivatives.

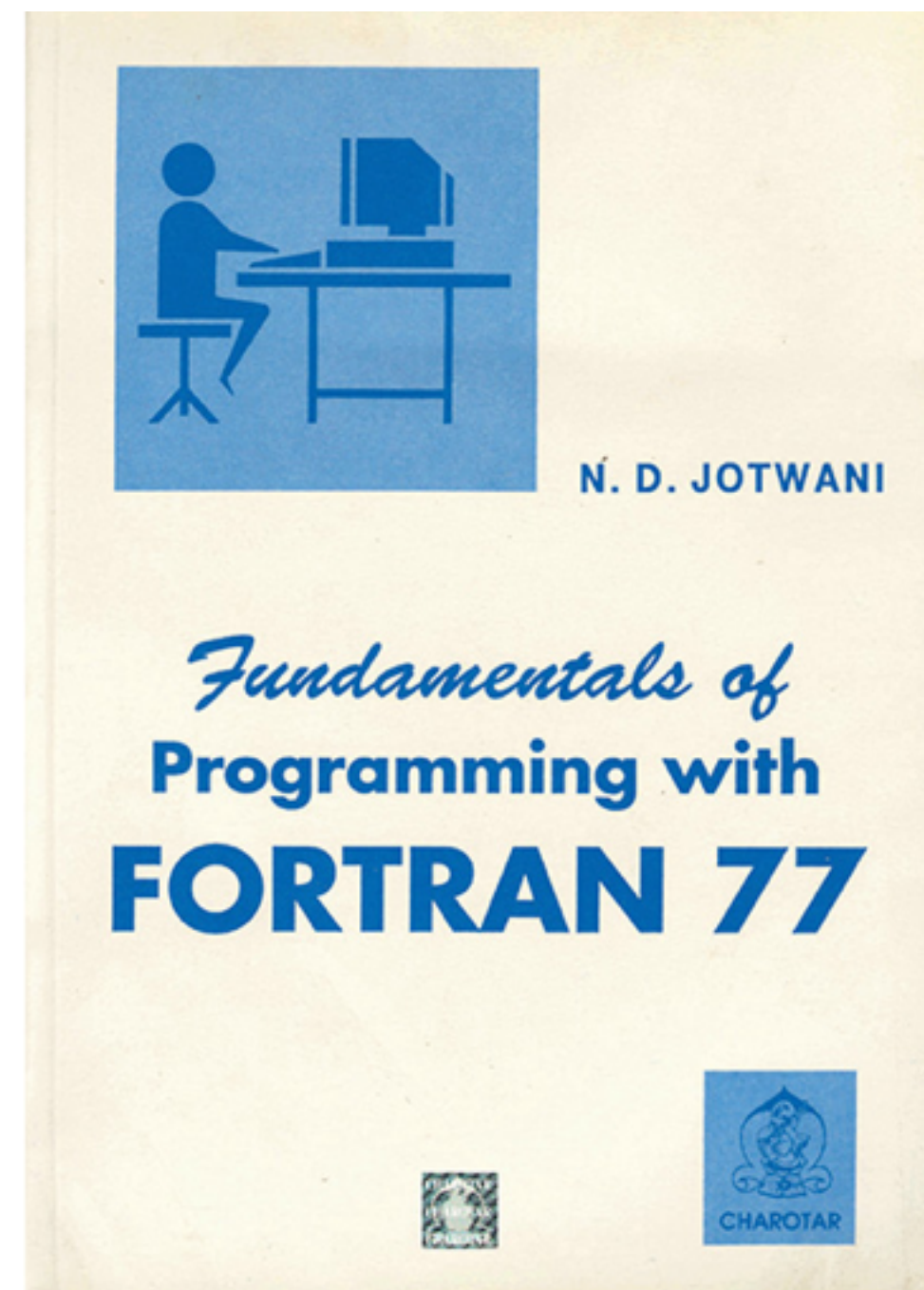
How does the error change if we change our vectors?



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



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!)



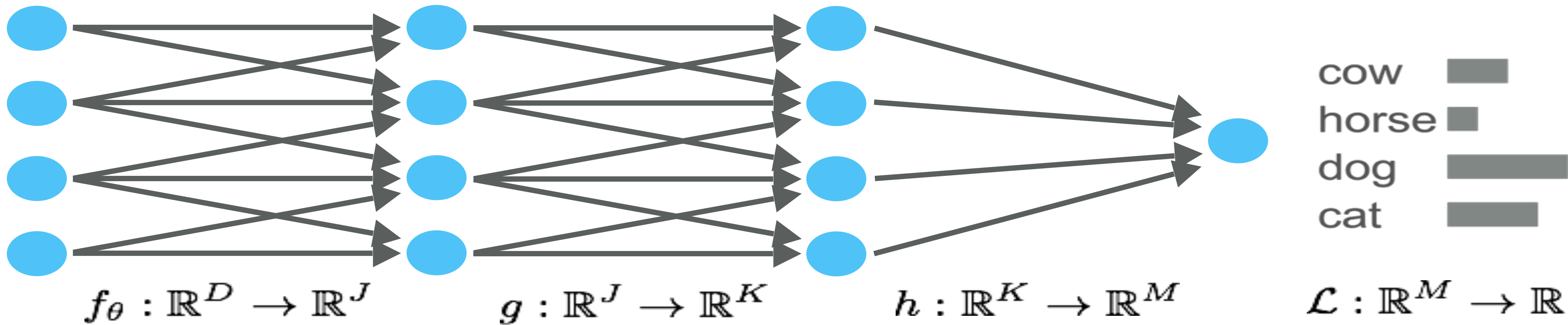
AUTOMATIC DIFFERENTIATION IS *THE* ABSTRACTION FOR GRADIENT-BASED ML

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)



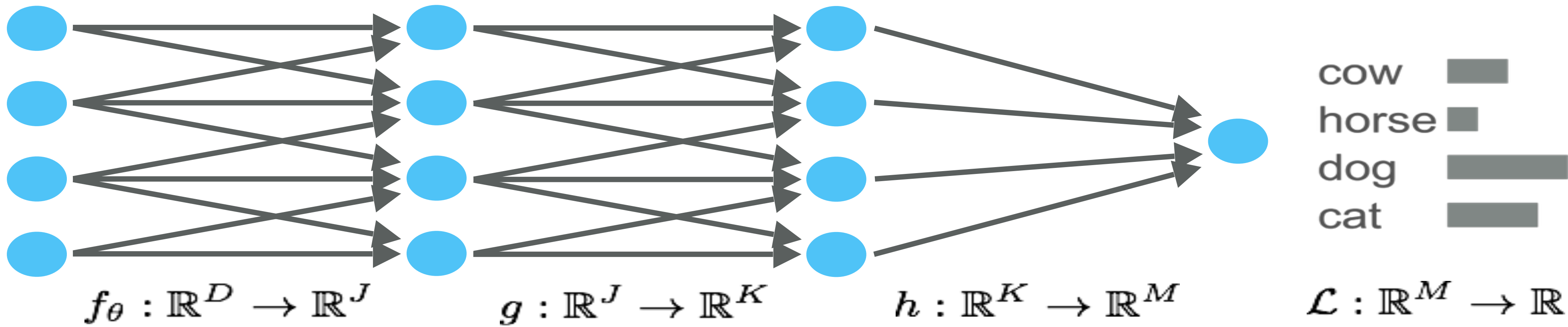
$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$$

$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) =$

$\begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \times \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \times \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \times \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$

$|\theta| \times J$ $J \times K$ $K \times M$ $M \times 1$

FORWARD MODE (SYMBOLIC VIEW)

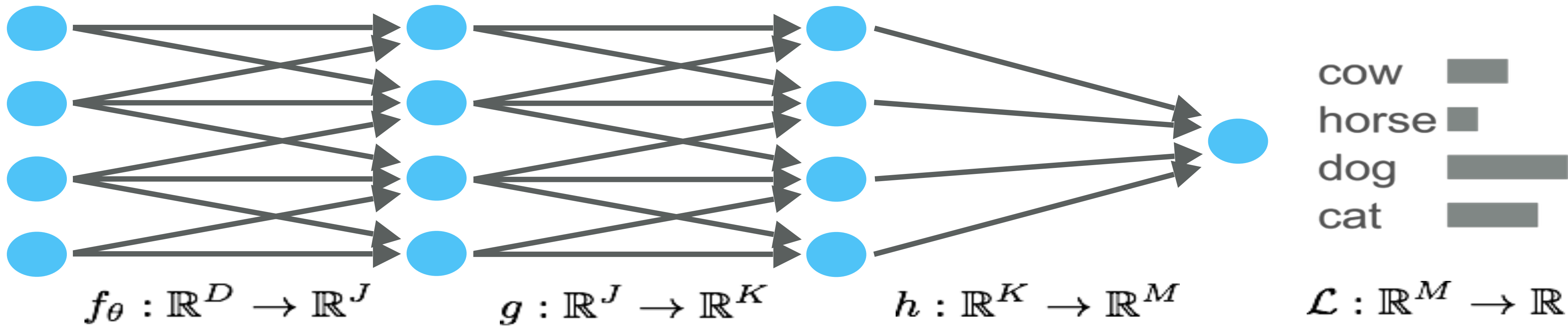


$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$$

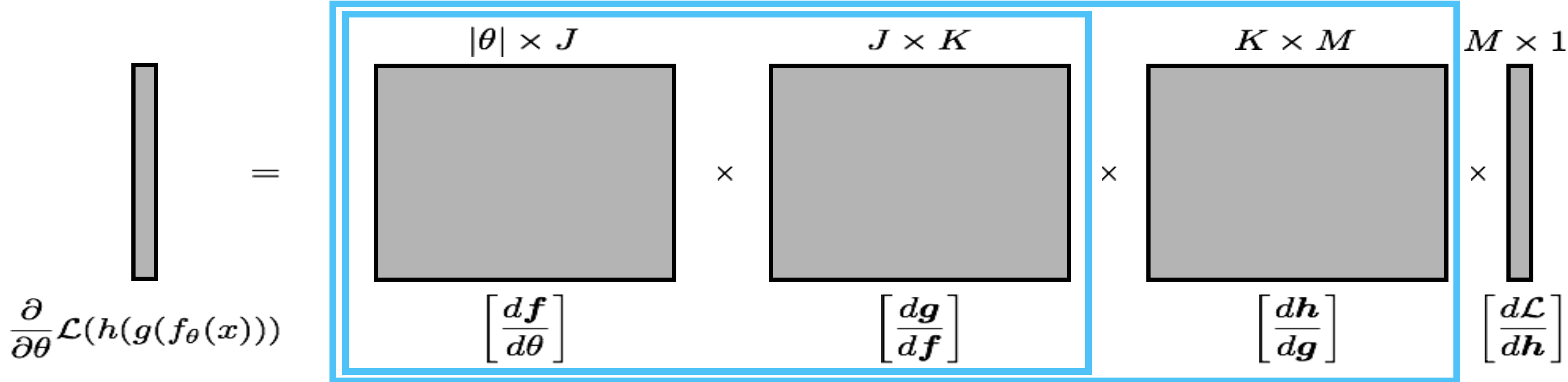
$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$

$|\theta| \times J$ $J \times K$ $K \times M$ $M \times 1$

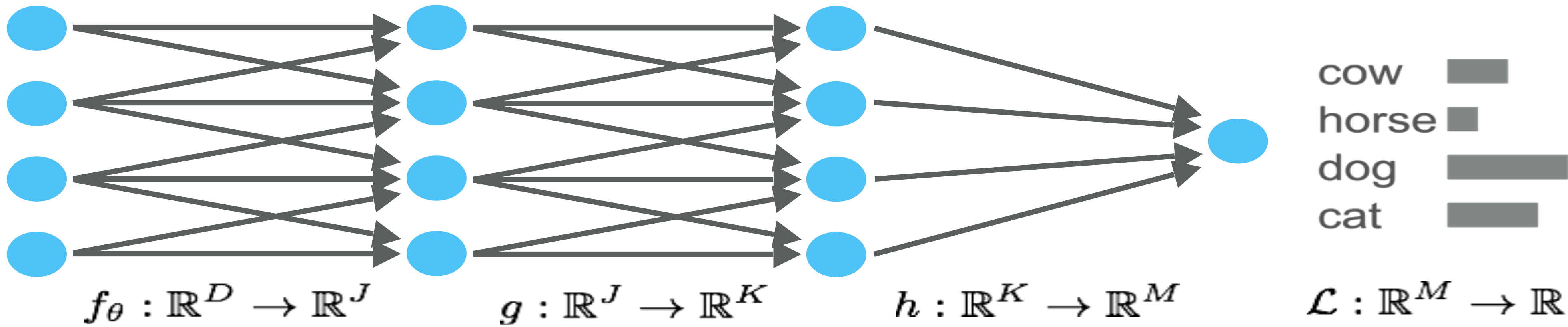
FORWARD MODE (SYMBOLIC VIEW)



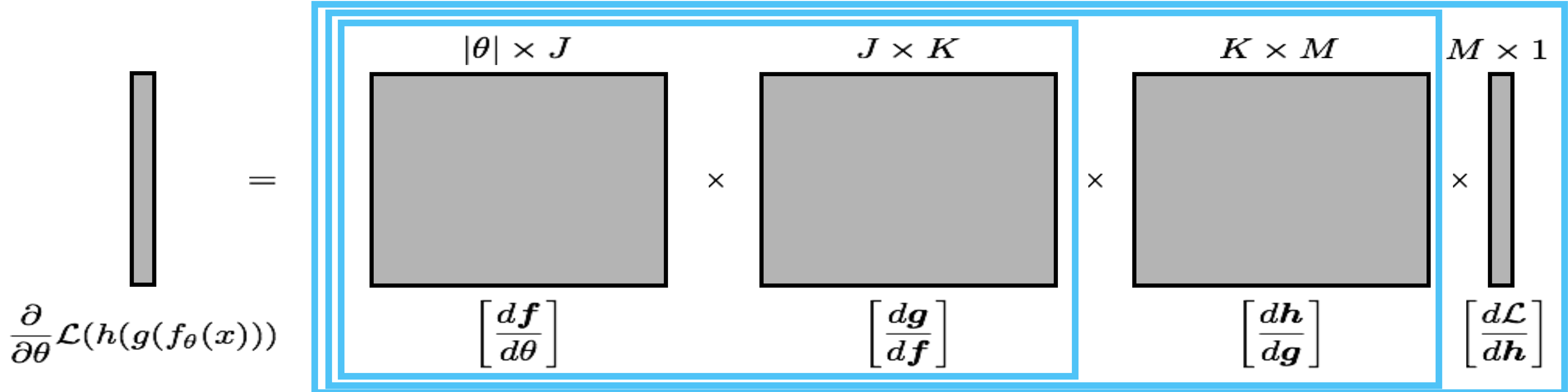
$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$$



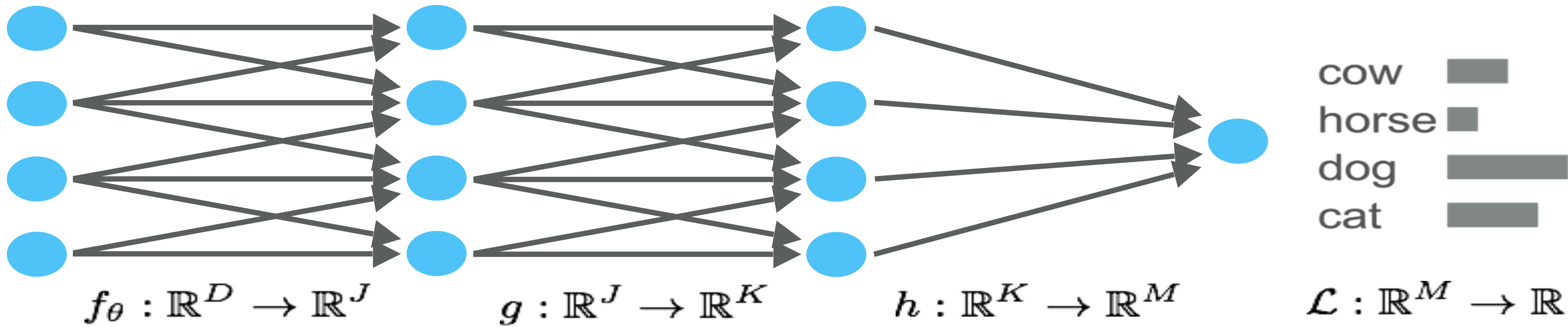
FORWARD MODE (SYMBOLIC VIEW)



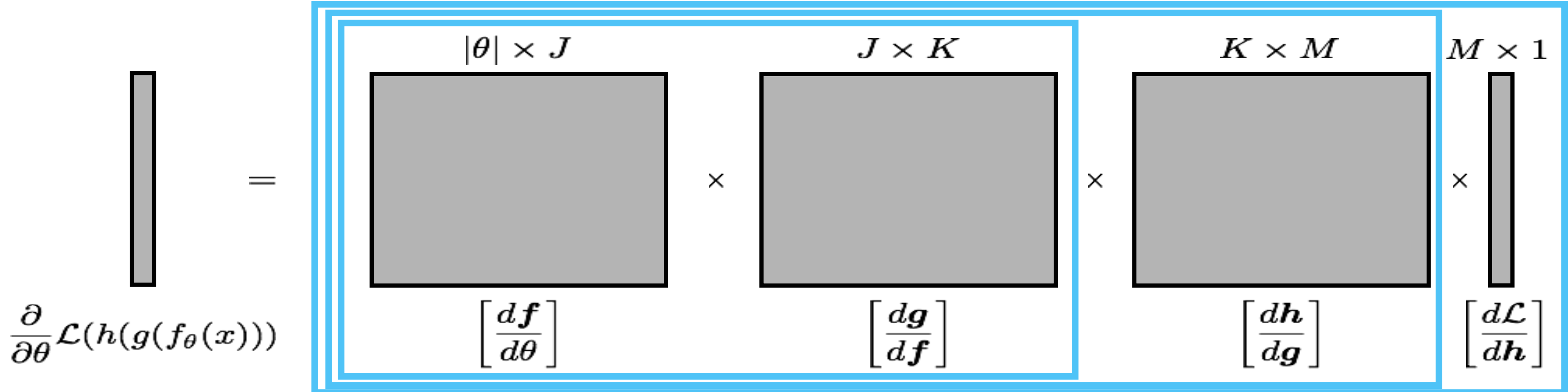
$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$$



FORWARD MODE (SYMBOLIC VIEW)



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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

b = 2

c = 1

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

return 2.728



From Baydin 2016

BOSTON NLP

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



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



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



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



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



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



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

BOSTON NLP

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

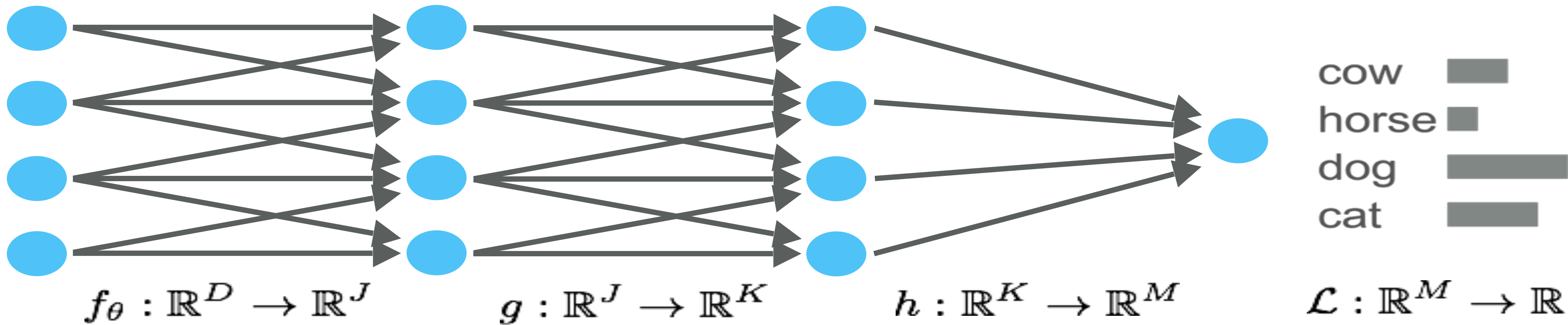
d = a * math.sin(b) = 2.728

ddda = math.sin(b) = 0.909

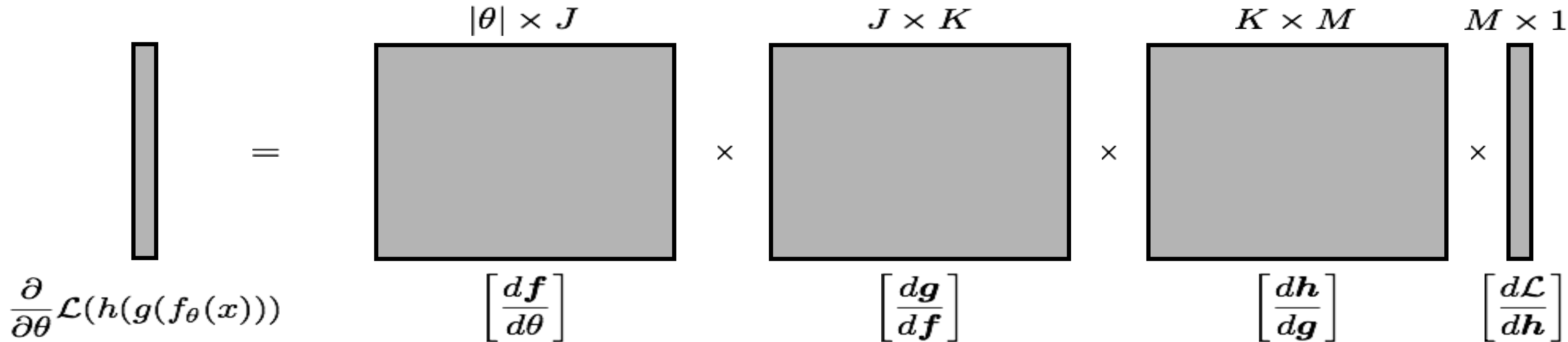
return 0.909



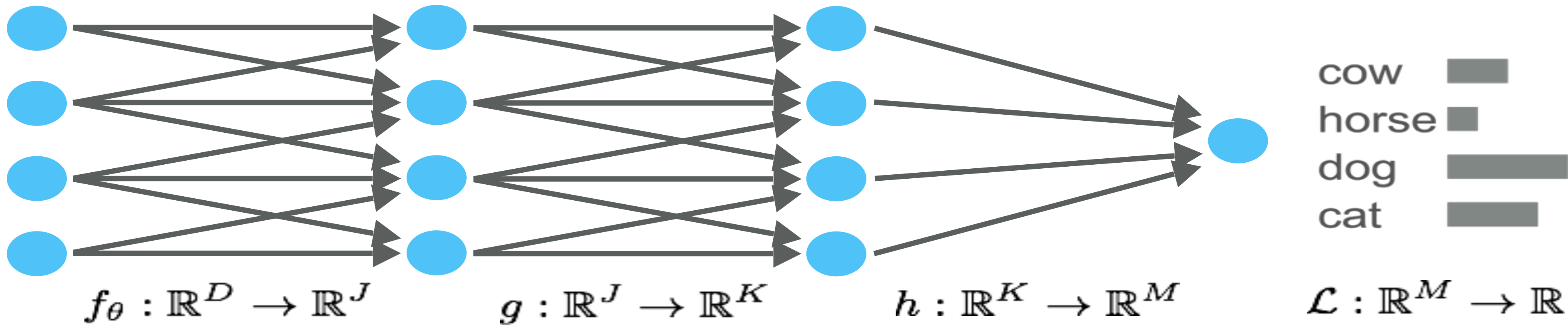
REVERSE MODE (SYMBOLIC VIEW)



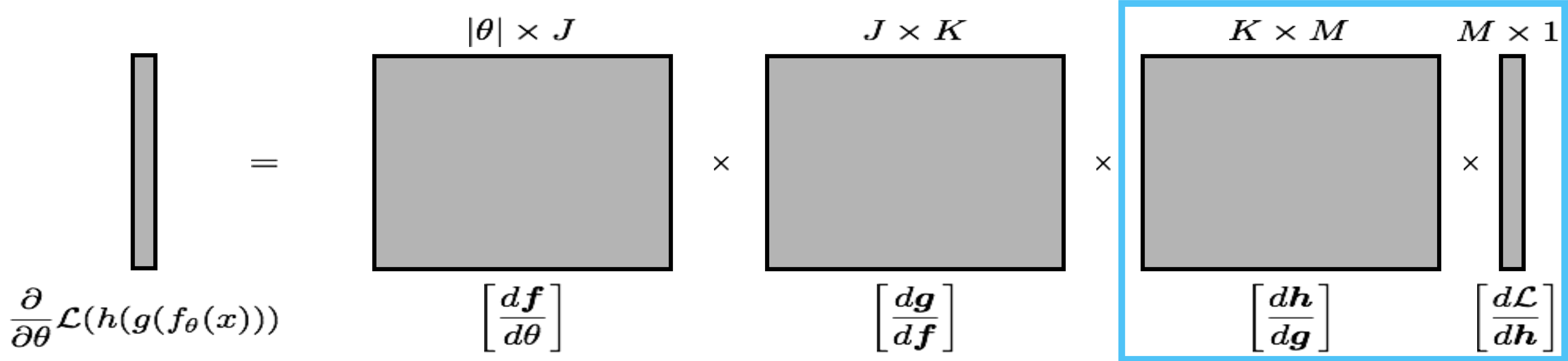
$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$$



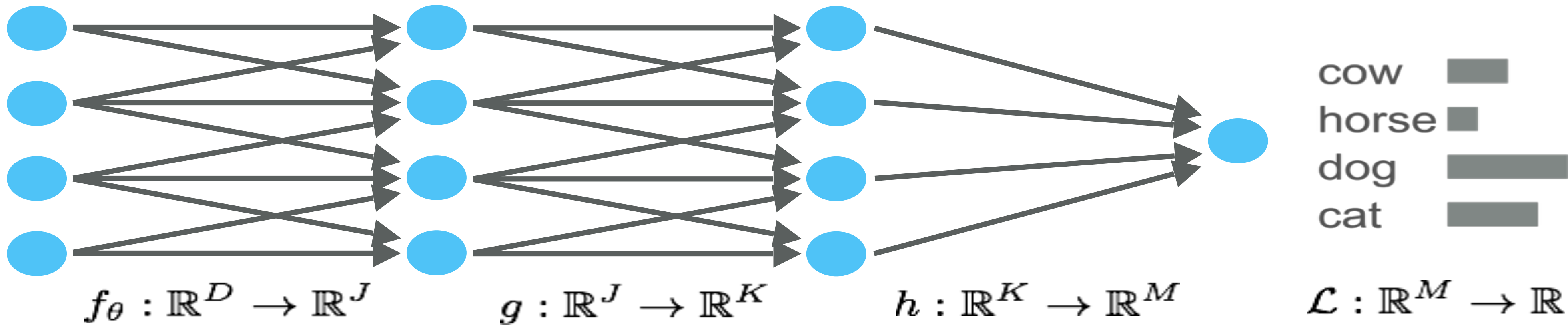
REVERSE MODE (SYMBOLIC VIEW)



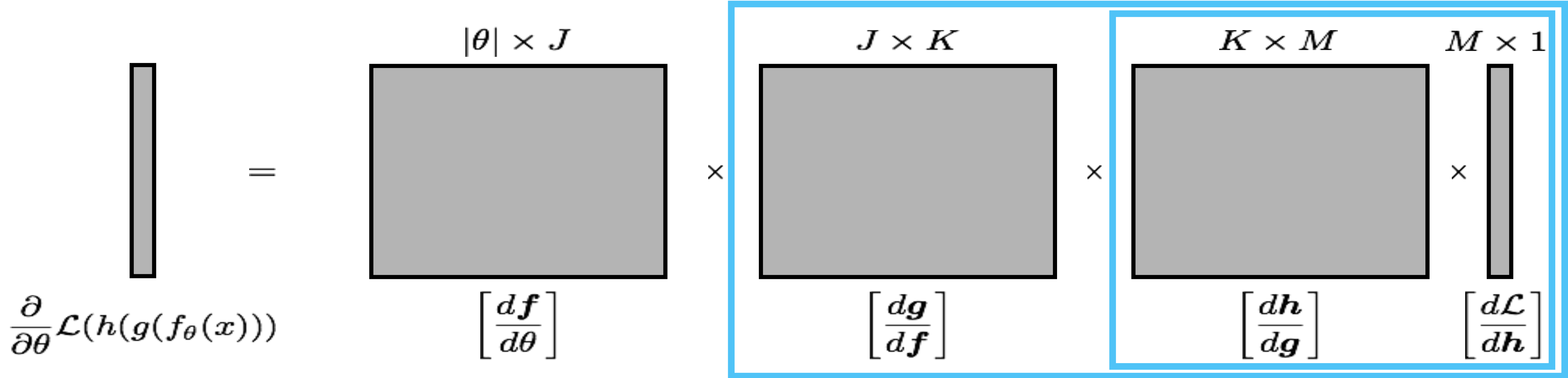
$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$$



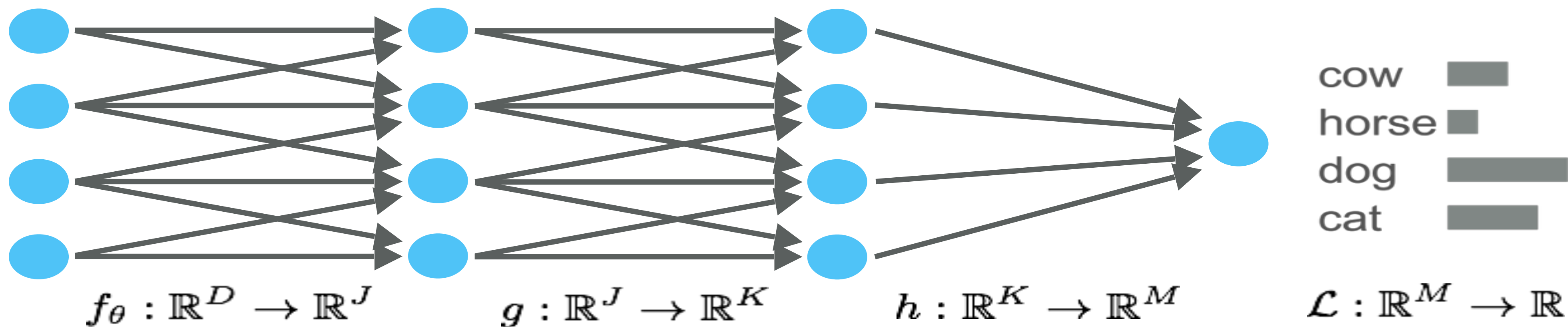
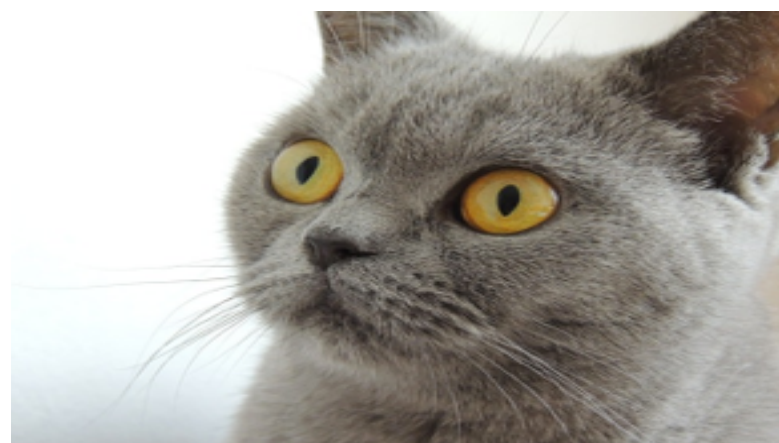
REVERSE MODE (SYMBOLIC VIEW)



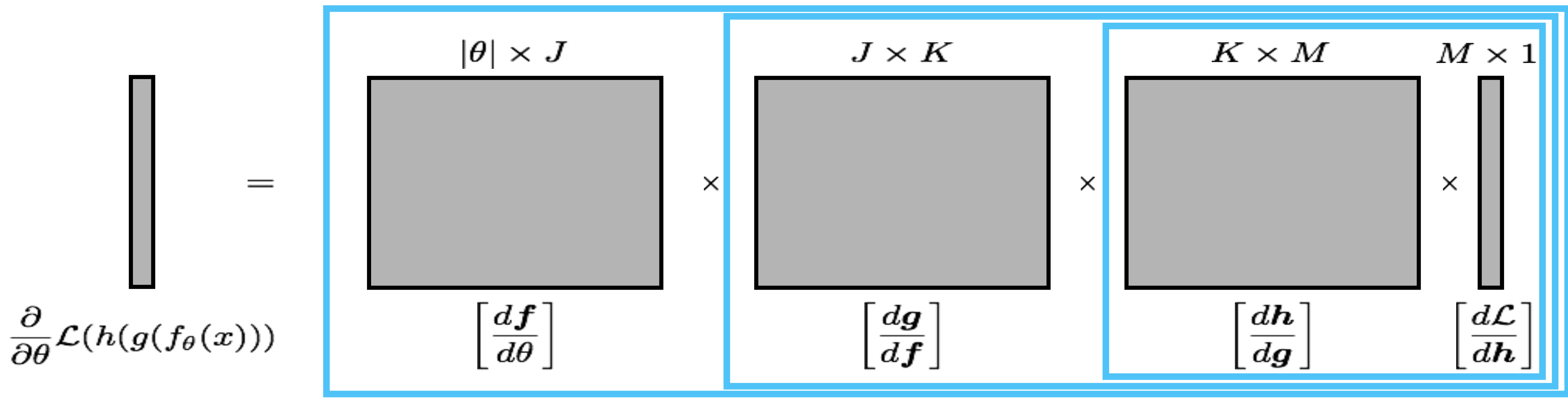
$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} \frac{d\mathbf{f}}{d\theta} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{g}}{d\mathbf{f}} \end{bmatrix} \begin{bmatrix} \frac{d\mathbf{h}}{d\mathbf{g}} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{d\mathbf{h}} \end{bmatrix}$$



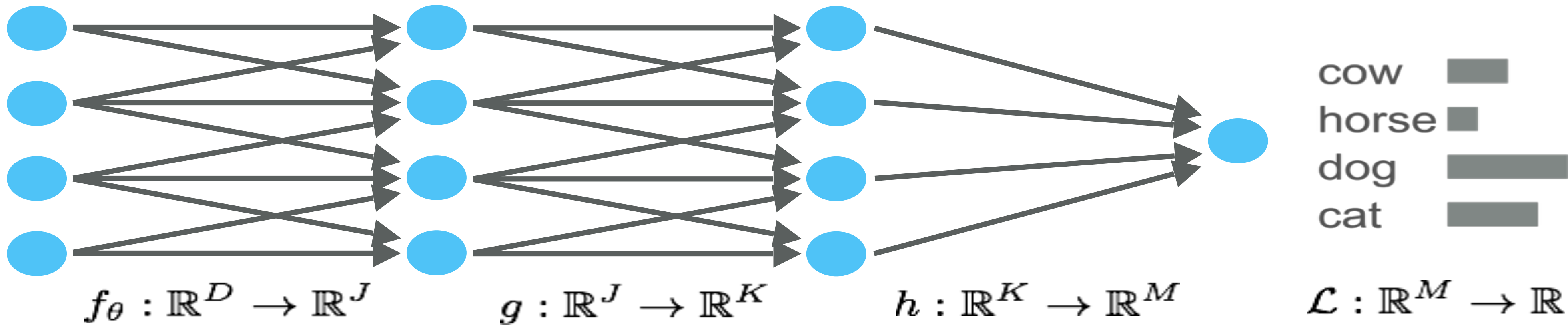
REVERSE MODE (SYMBOLIC VIEW)



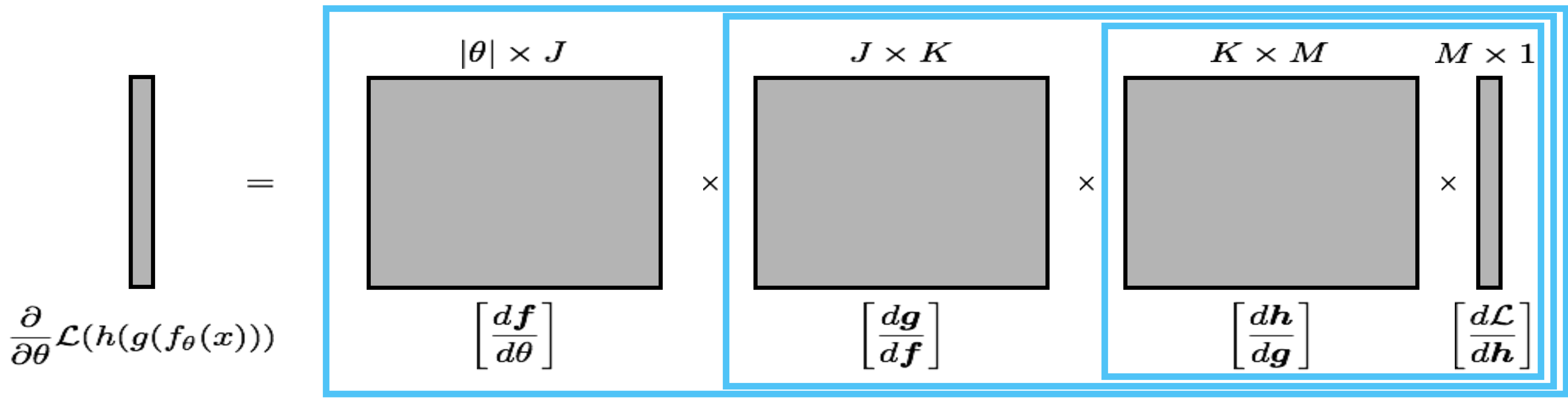
$$\frac{\partial}{\partial \theta} \mathcal{L}(h(g(f_{\theta}(x)))) = \begin{bmatrix} d\mathbf{f} \\ d\theta \end{bmatrix} \begin{bmatrix} dg \\ d\mathbf{f} \end{bmatrix} \begin{bmatrix} dh \\ dg \end{bmatrix} \begin{bmatrix} d\mathcal{L} \\ dh \end{bmatrix}$$



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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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

BOSTON NLP

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



```
2 torch = require 'torch'
3 params = {
4     W = {torch.randn(64*64,50),torch.randn(50,4)},
5     b = {torch.randn(64*64), torch.randn(4)}
6 }
7
8 function neuralNetwork(params, image)
9     local h1 = torch.tanh(image*params.W[1] + params.b[1])
10    local h2 = torch.tanh(h1*params.W[2] + params.b[2])
11    return torch.log(torch.sum(torch.exp(h2)))
12 end
13
14 function loss(params, image, trueLabel)
15    local prediction = neuralNetwork(params, image)
16    return torch.sum(torch.pow(prediction-trueLabel,2))
17 end
18
19 grad = require 'autograd'
20 dloss = grad(loss)
21
22 for _,datapoint in dataset() do
23     -- Calculate our gradients
24     local gradients = dloss(params, datapoint.image, datapoint.label)
25     -- Update parameters
26     for i=1,#params.W do
27         params.W[i] = params.W[i] - 0.01*gradients.W[i]
28         params.b[i] = params.b[i] - 0.01*gradients.b[i]
29     end
30 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

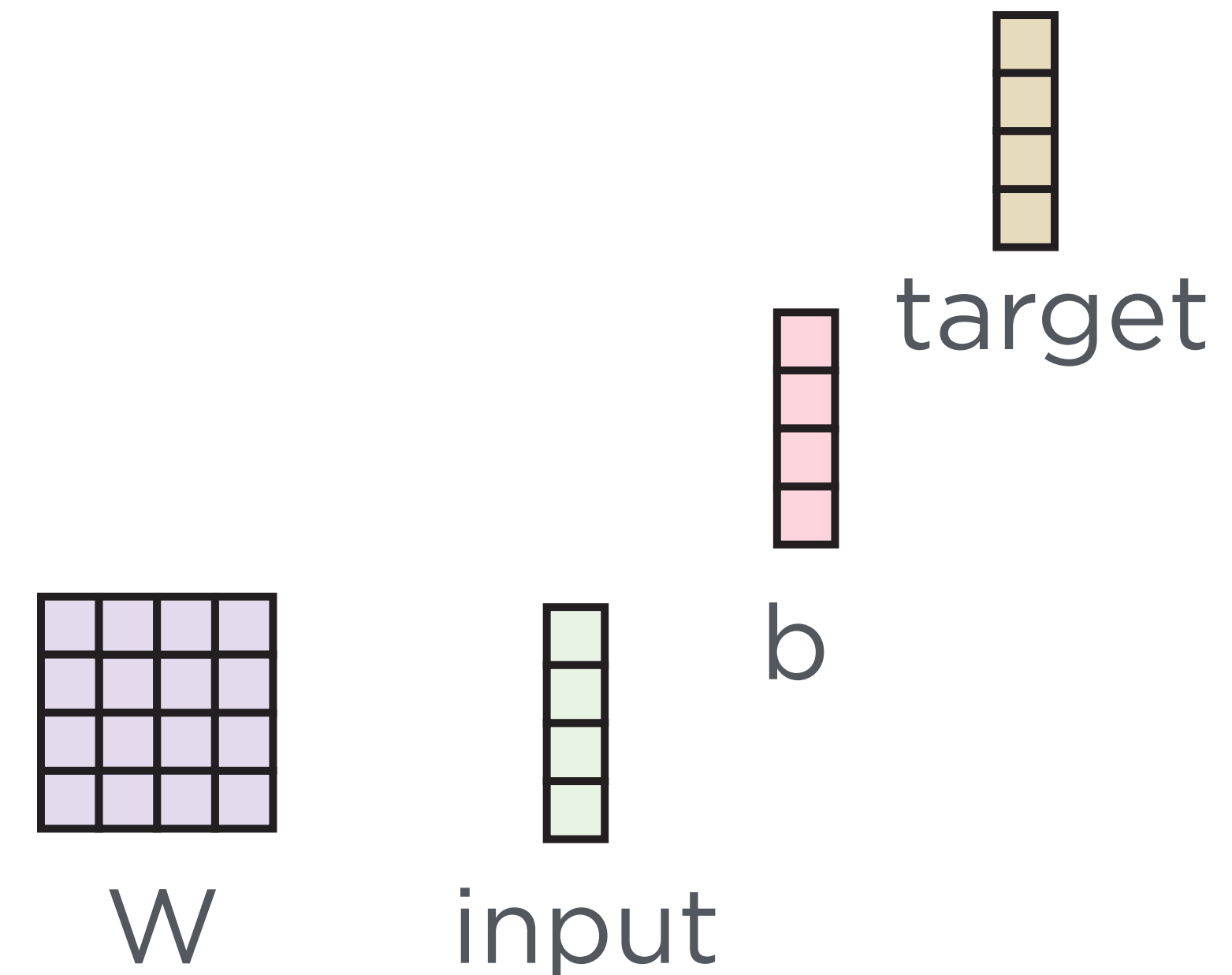
```
2 torch = require 'torch'
3 params = {
4     W = {torch.randn(64*64,50),torch.randn(50,4)},
5     b = {torch.randn(64*64), torch.randn(4)}
6 }
7
8 function neuralNetwork(params, image)
9     local h1 = torch.tanh(image*params.W[1] + params.b[1])
10    local h2 = torch.tanh(h1*params.W[2] + params.b[2])
11    return torch.log(torch.sum(torch.exp(h2)))
12 end
13
14 function loss(params, image, trueLabel)
15    local prediction = neuralNetwork(params, image)
16    return torch.sum(torch.pow(prediction-trueLabel,2))
17 end
18
19 grad = require 'autograd'
20 dloss = grad(loss)
21
22 for _,datapoint in dataset() do
23     -- Calculate our gradients
24     local gradients = dloss(params, datapoint.image, datapoint.label)
25     -- Update parameters
26     for i=1,#params.W do
27         params.W[i] = params.W[i] - 0.01*gradients.W[i]
28         params.b[i] = params.b[i] - 0.01*gradients.b[i]
29     end
30 end
```



WHAT'S ACTUALLY HAPPENING?

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

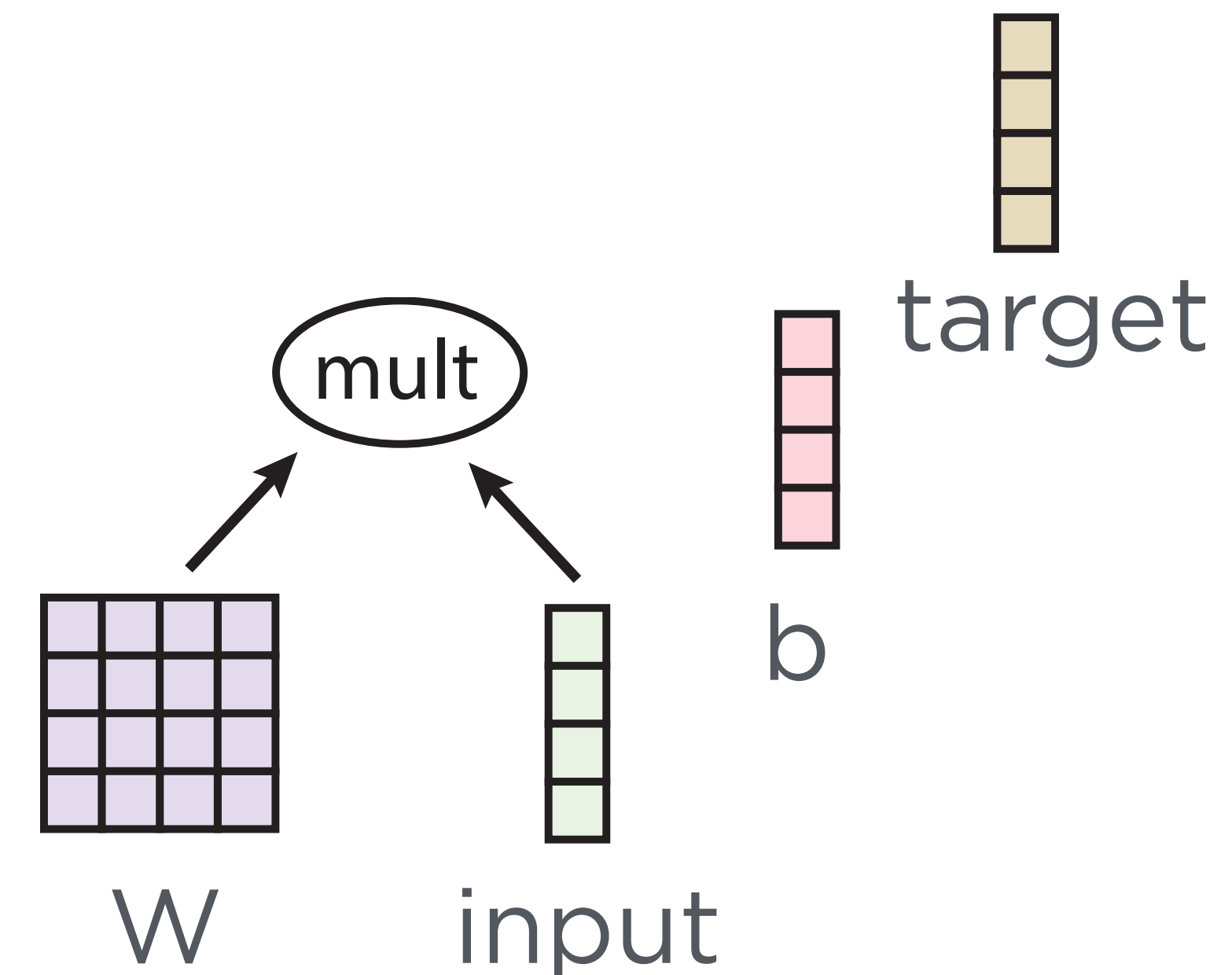
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



WHAT'S ACTUALLY HAPPENING?

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

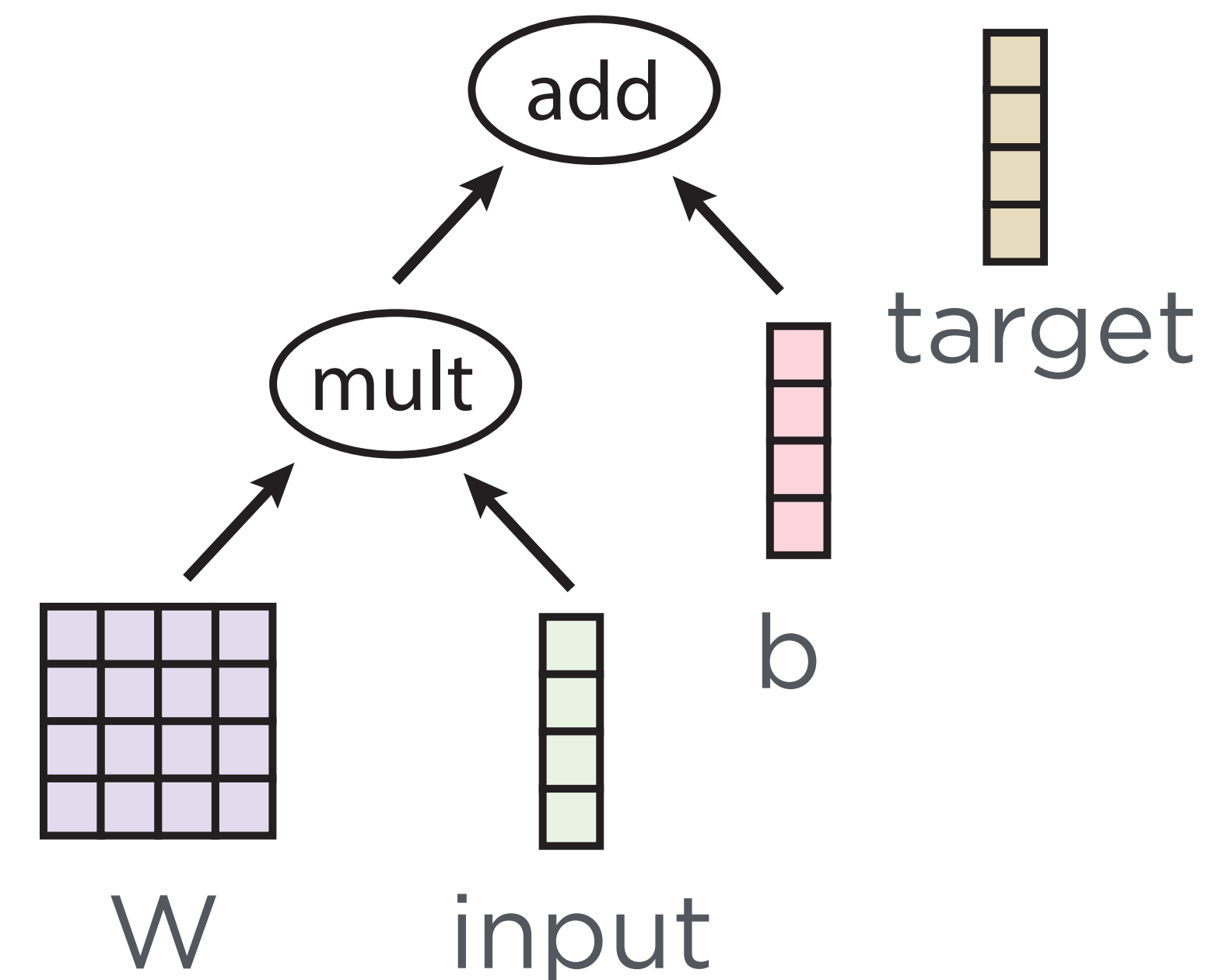
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



WHAT'S ACTUALLY HAPPENING?

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

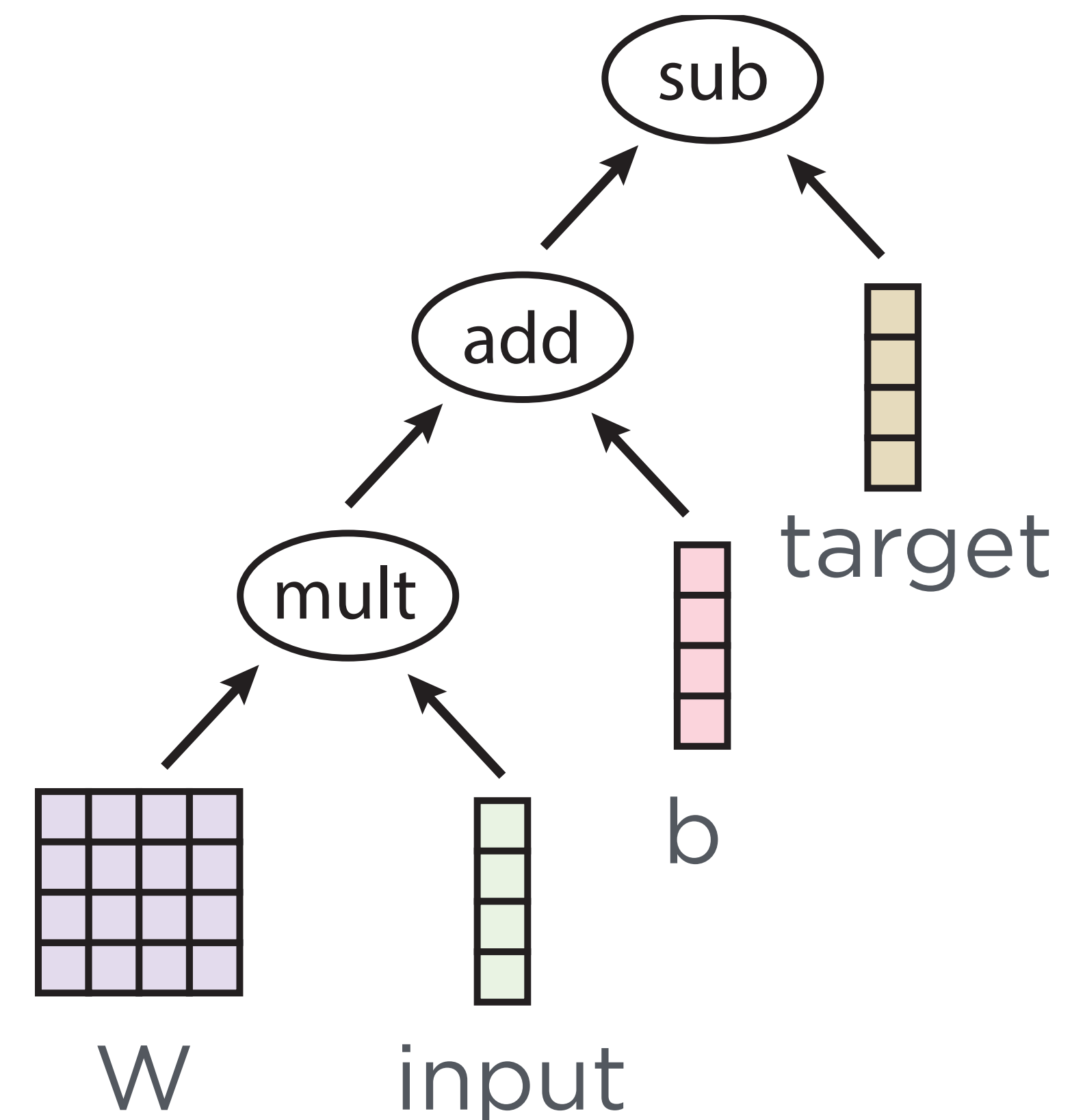
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



WHAT'S ACTUALLY HAPPENING?

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

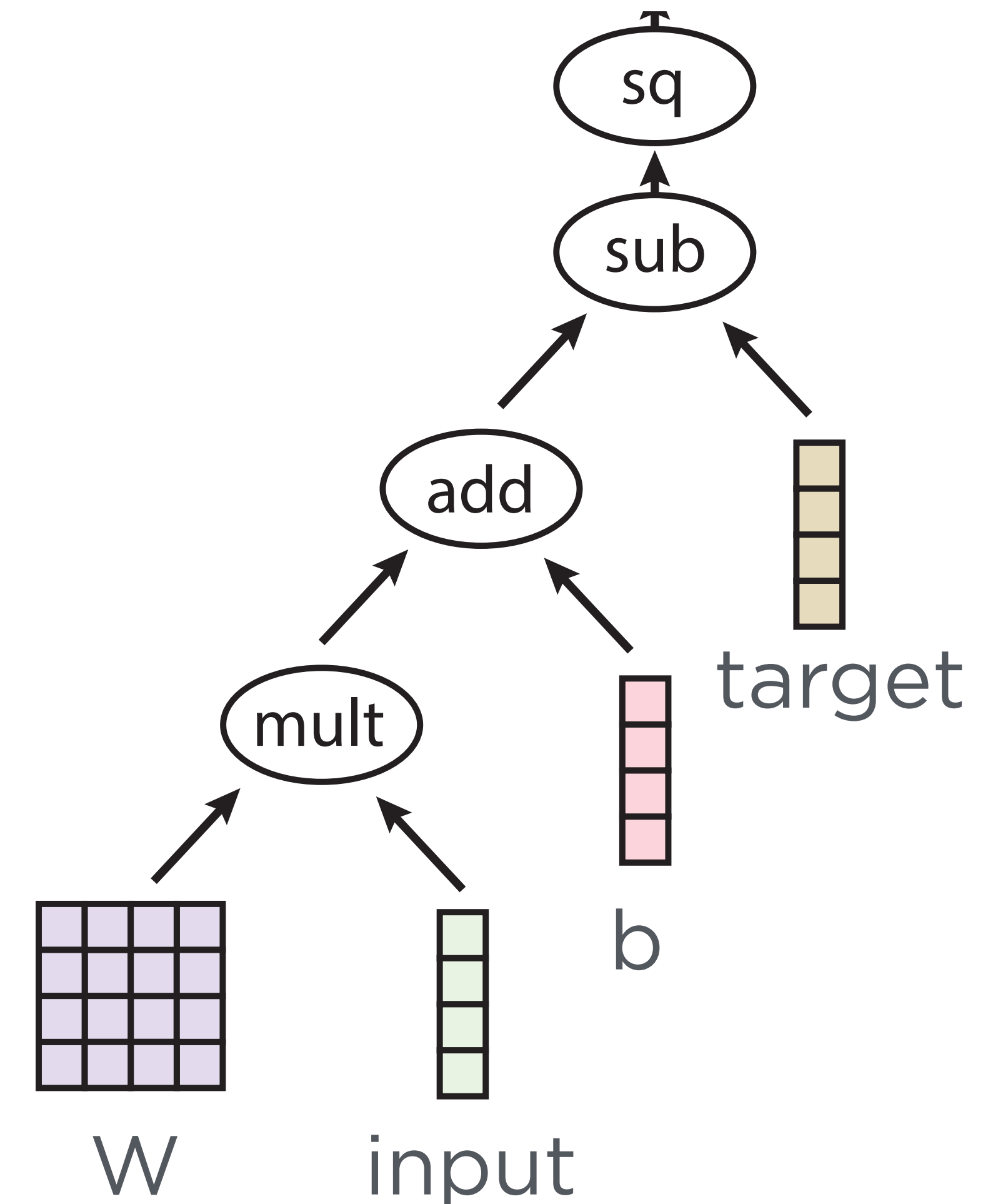
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



WHAT'S ACTUALLY HAPPENING?

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

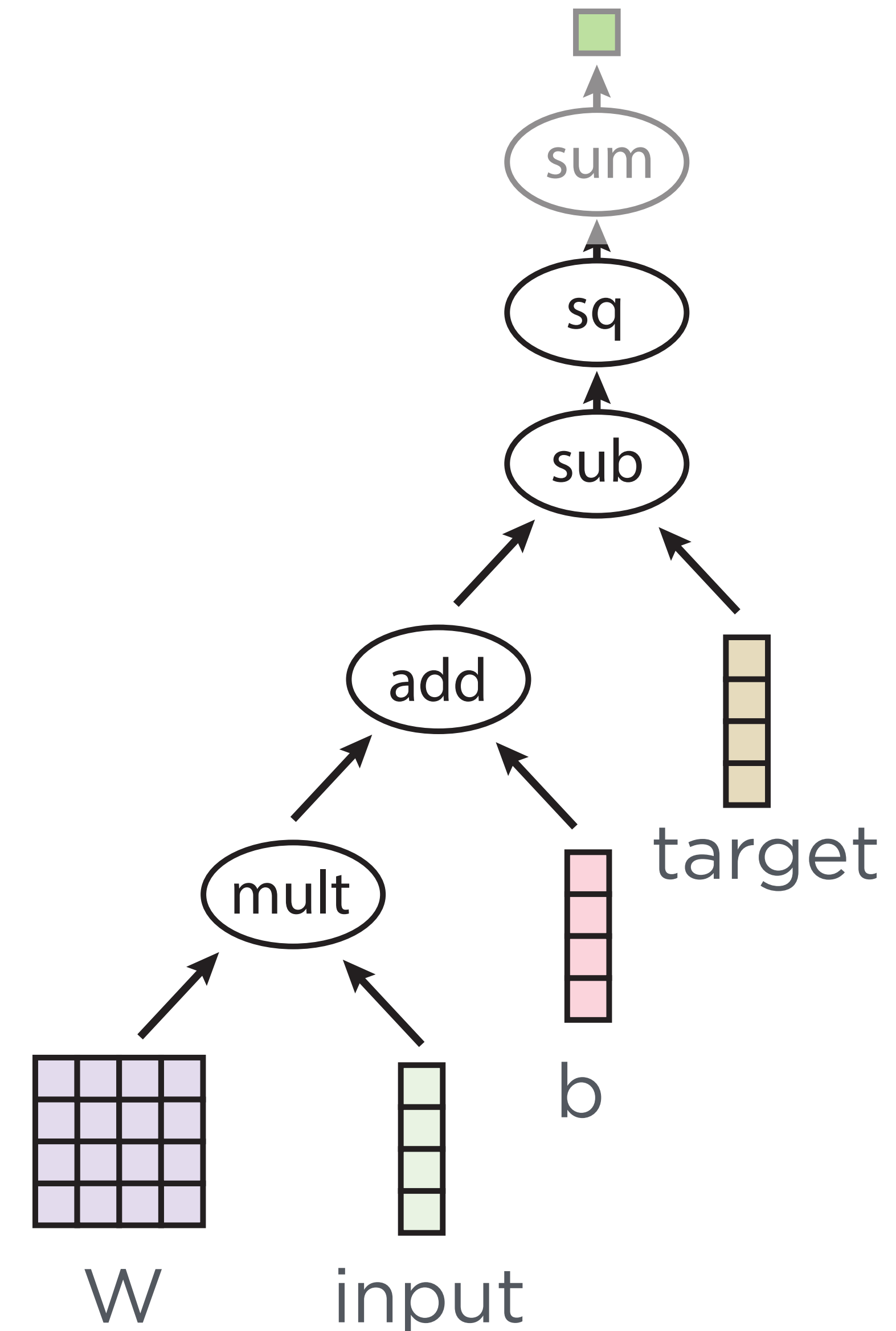
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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

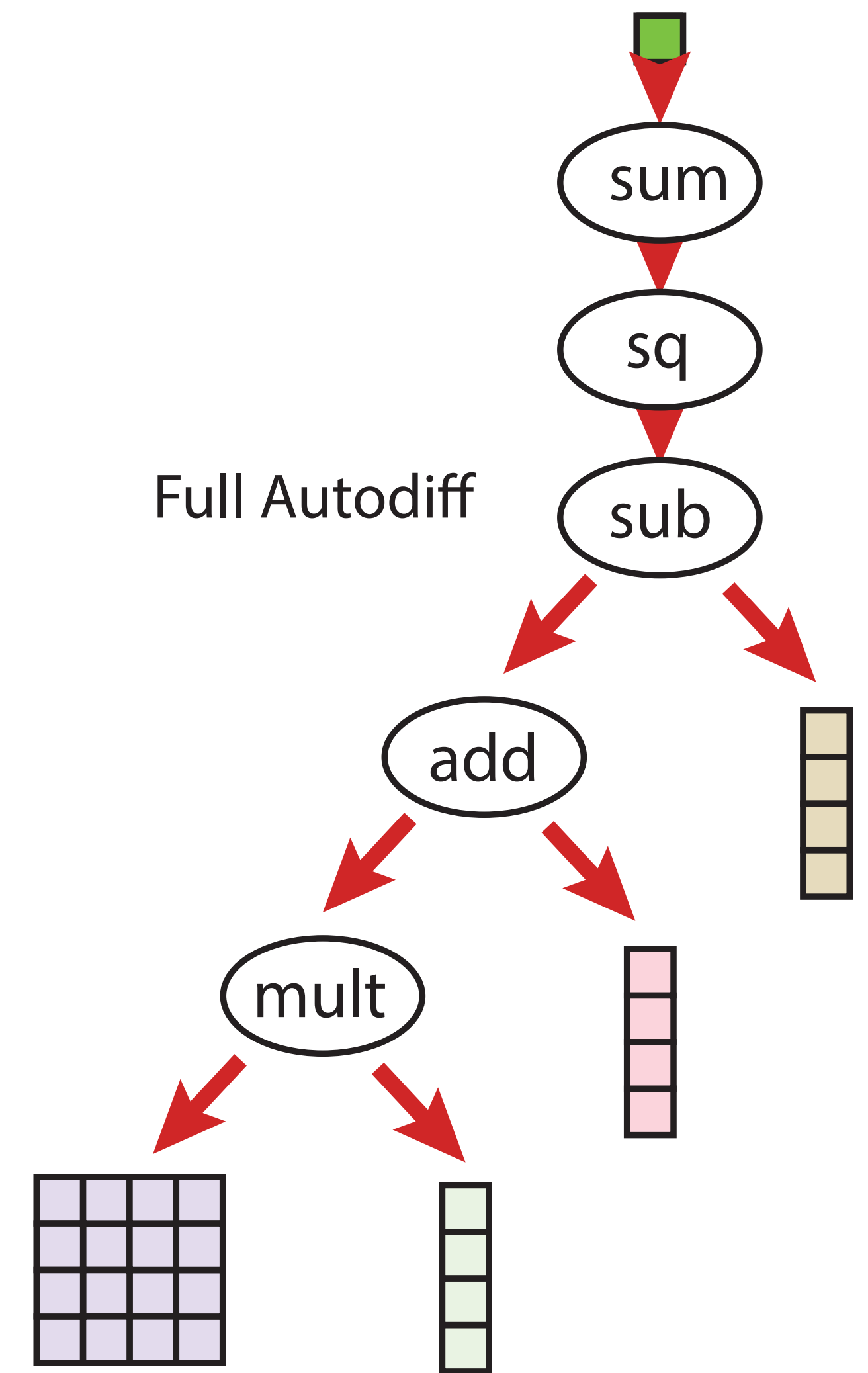
```
1  local origSum = torch.sum
2  torch.sum = function(arg)
3
4      -- Check if the argument has been used before in an overloaded function
5      if not isNodeType(arg) then
6          return origSum(arg)
7      else
8          -- Run the function
9          local outputVal = origSum(unpackNode(arg))
10
11         -- Build a data structure that will track computation via linked list
12         local outputNode = {fn=origSum,parent=arg,val=outputVal}
13     end
14 end
15
16 -- Now overload all other numeric functions...
17 -- sin,cos,tan,sinh,cosh,tanh,add,sub,mul,div,pow
18 -- 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

```
1 gradients[torch.sqrt] = {  
2   function(g, ans, x) return torch.cmul(torch.cmul(g,0.5), torch.pow(x,-0.5)) end  
3 }  
4 gradients[torch.sin] = {  
5   function(g, ans, x) return torch.cmul(g, torch.cos(x)) end  
6 }  
7 gradients[torch.cos] = {  
8   function(g, ans, x) return torch.cmul(g, -torch.sin(x)) end  
9 }  
10 gradients[torch.tan] = {  
11   function(g, ans, x) return torch.cdiv(g, torch.pow(torch.cos(x), 2.0)) end  
12 }  
13 gradients[torch.log] = {  
14   function(g, ans, x) return torch.cdiv(g,x) end  
15 }
```



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:

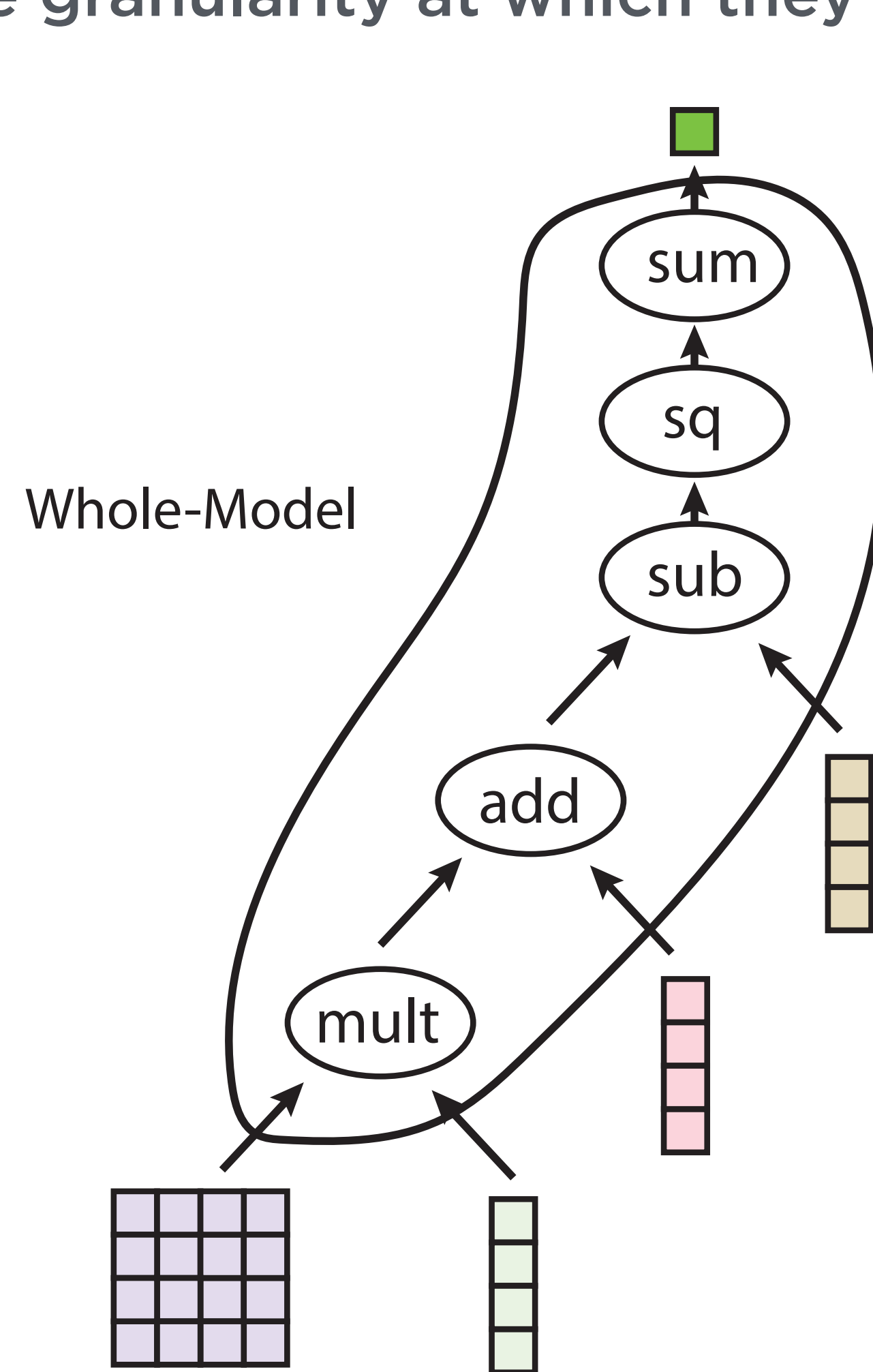
3	0	0
0	3	0
0	0	3

[torch.DoubleTensor of size 3x3]

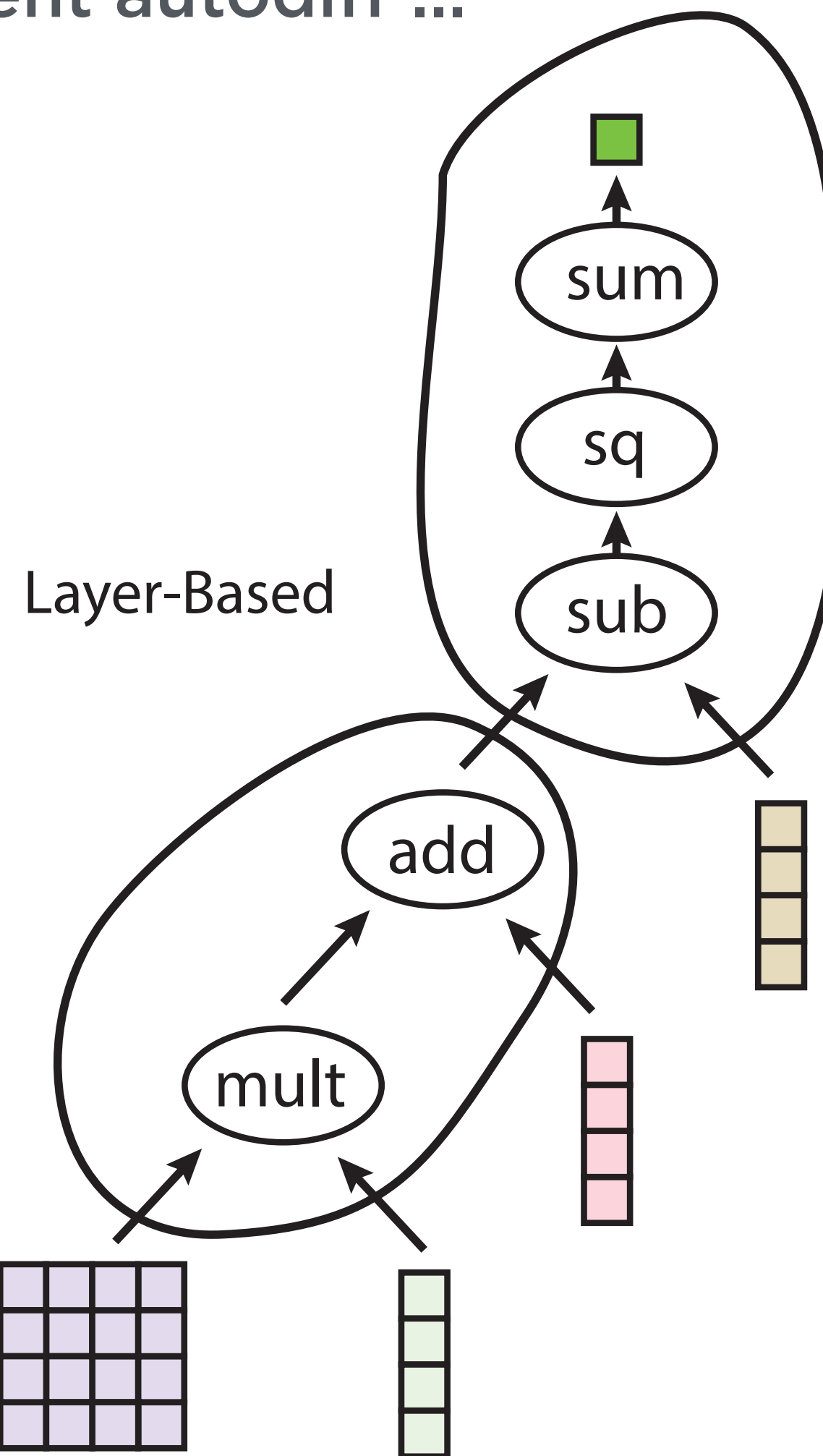


SO WHAT DIFFERENTIATES N.NET LIBRARIES?

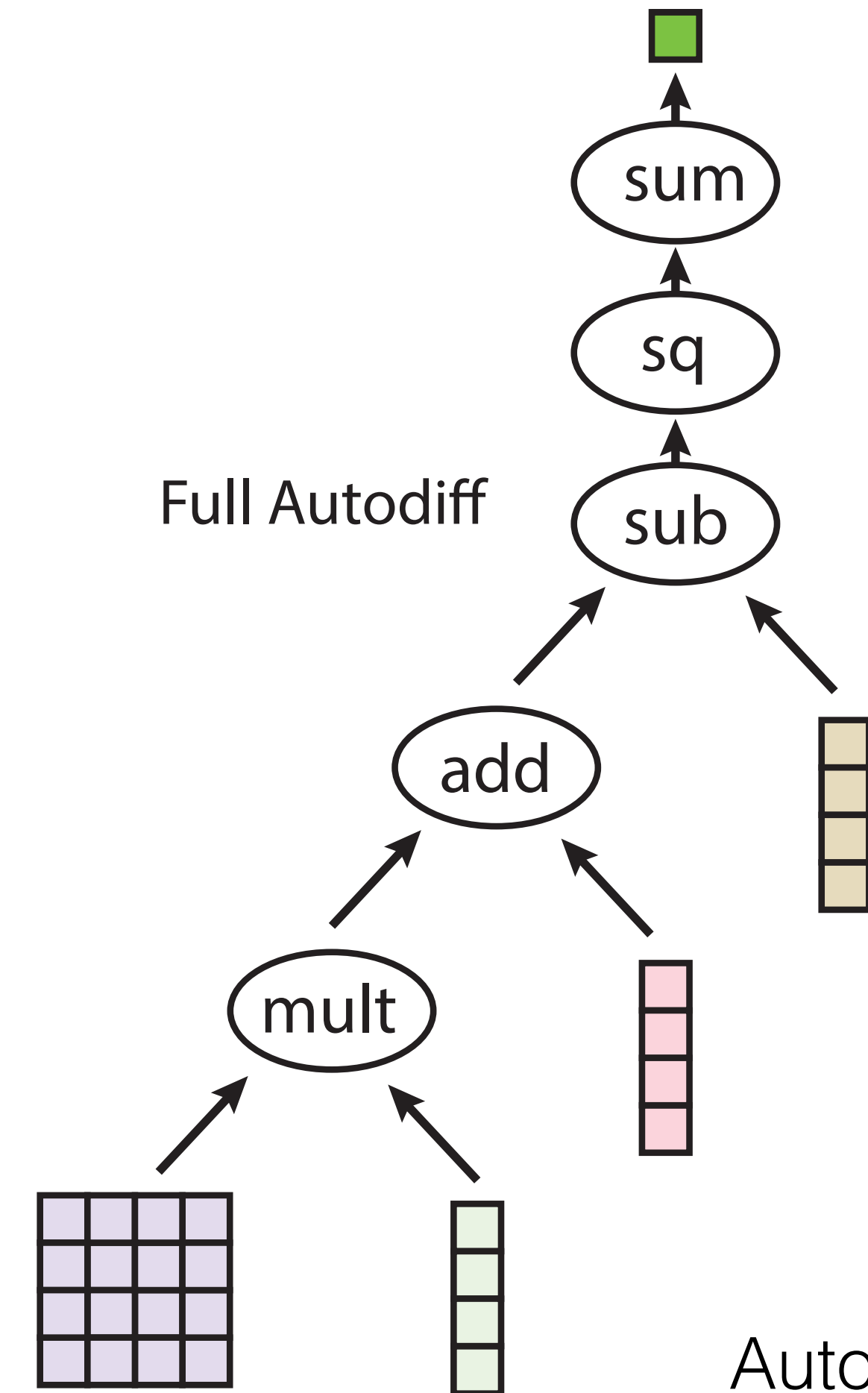
The granularity at which they implement autodiff ...



scikit-learn



Torch NN
Keras
Lasagne

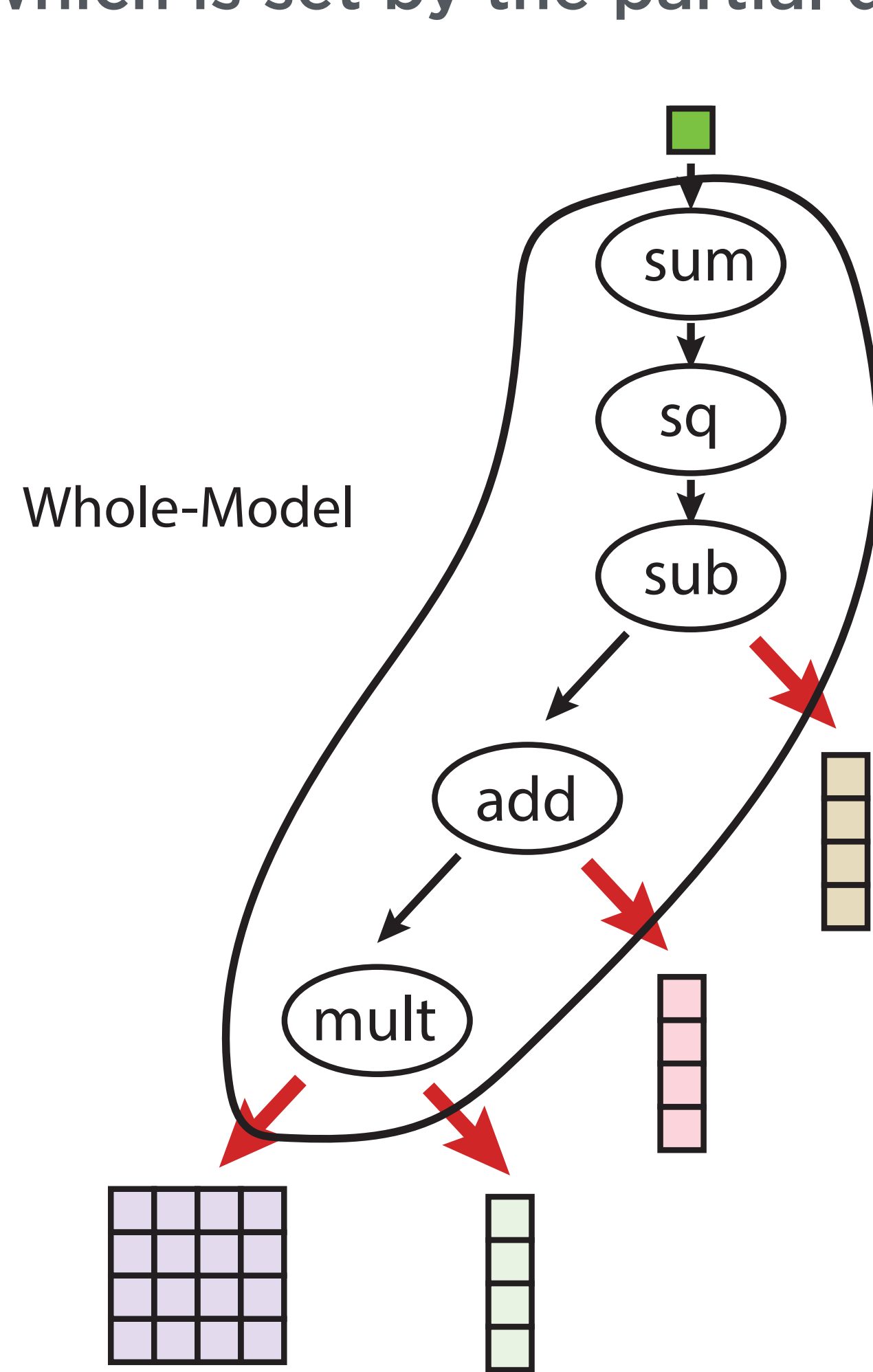


Autograd
Theano
TensorFlow

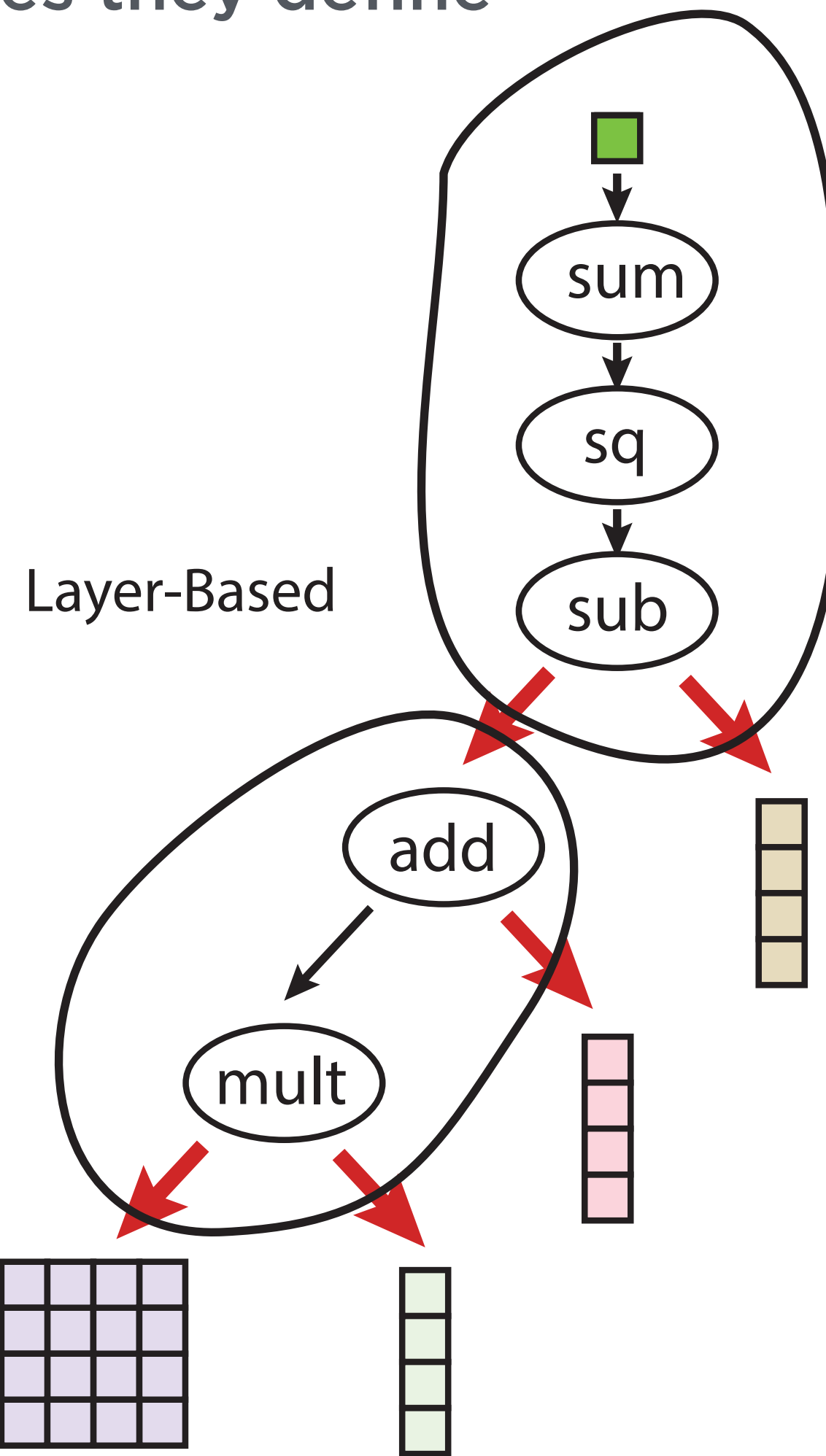


SO WHAT DIFFERENTIATES N.NET LIBRARIES?

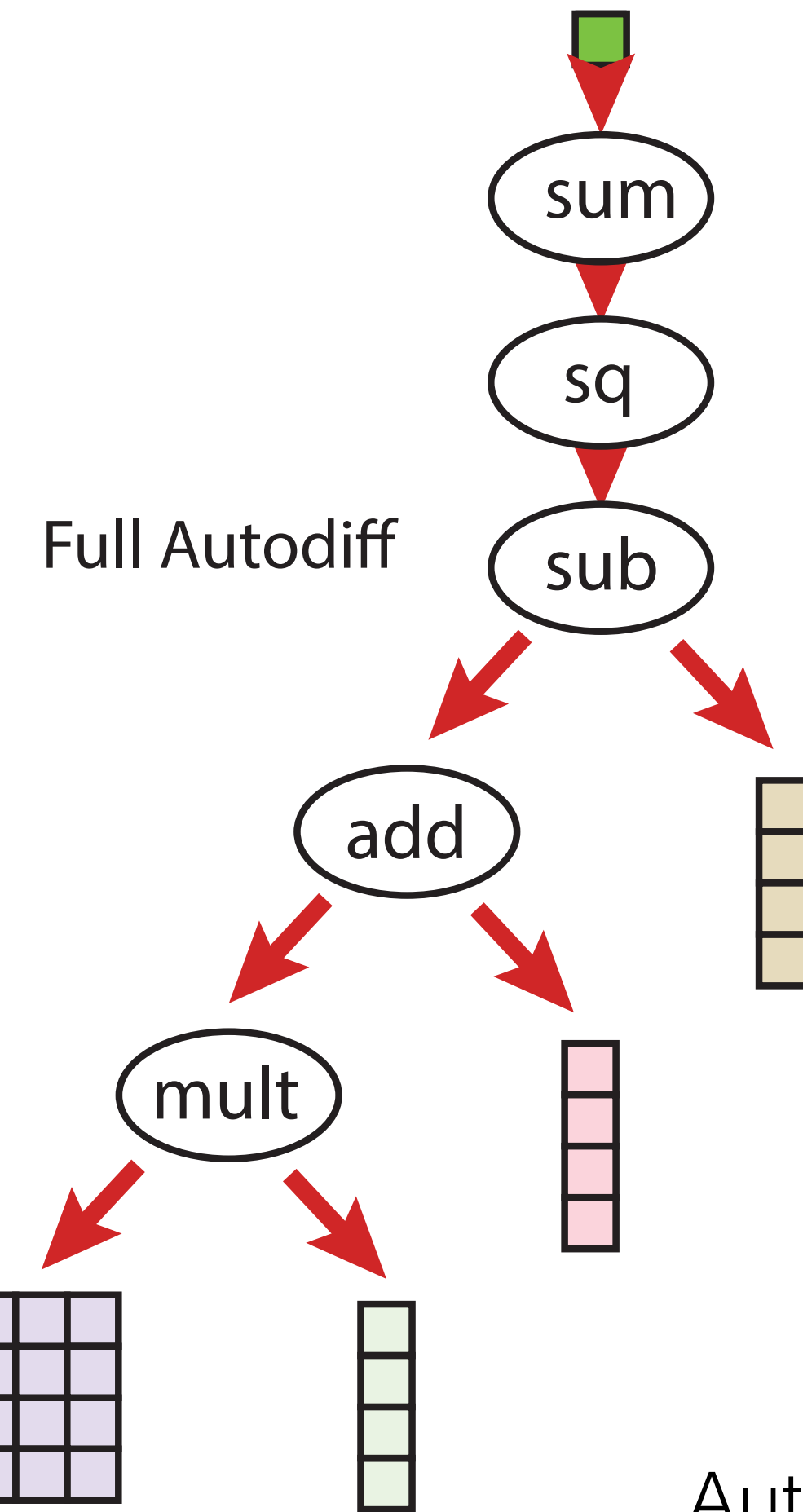
... which is set by the partial derivatives they define



scikit-learn



Torch NN
Keras
Lasagne



Autograd
Theano
TensorFlow



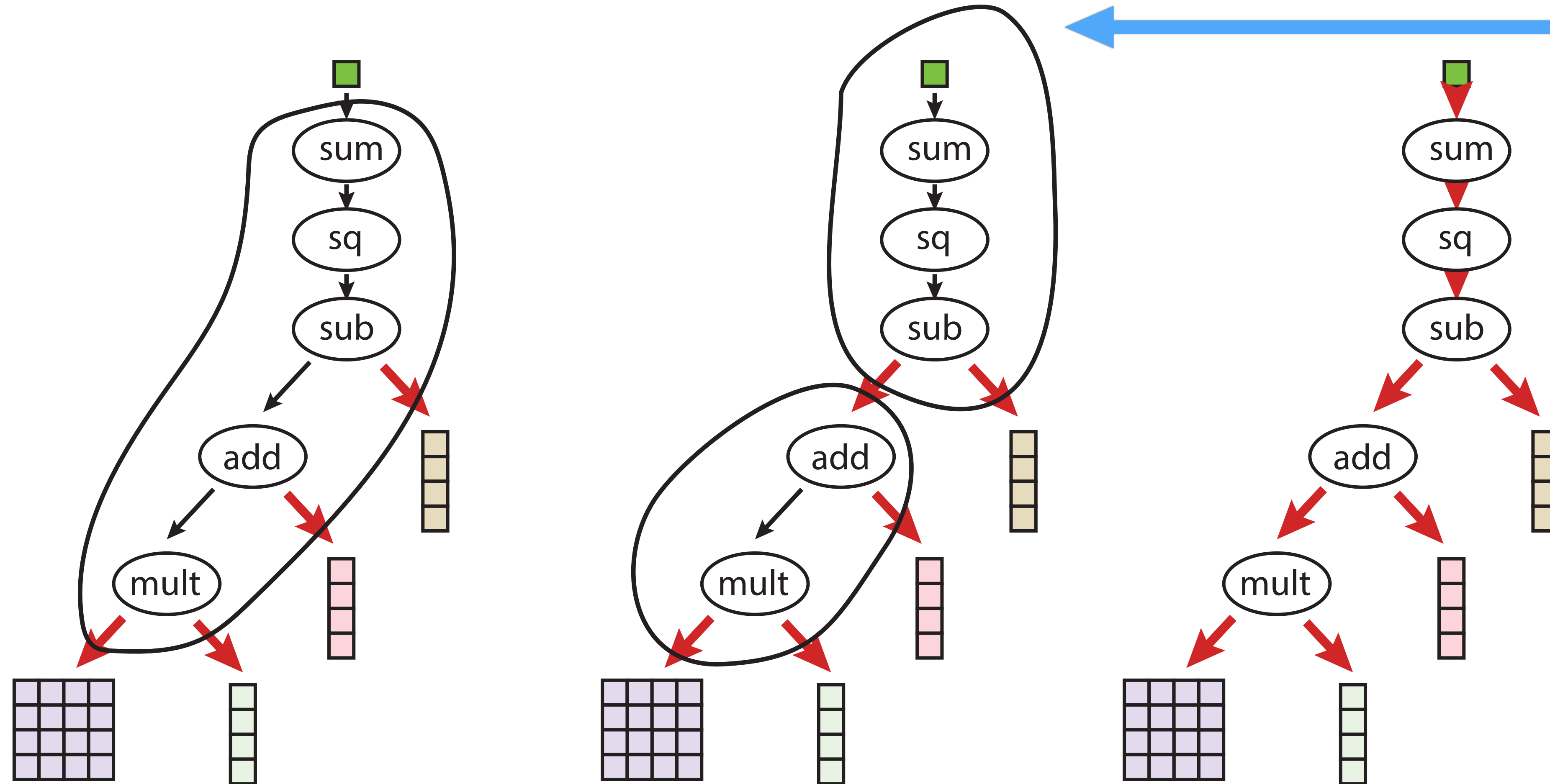
SO WHAT DIFFERENTIATES N.NET LIBRARIES?

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

"Shrink-wrapped"

"Autodiff Wrappers"

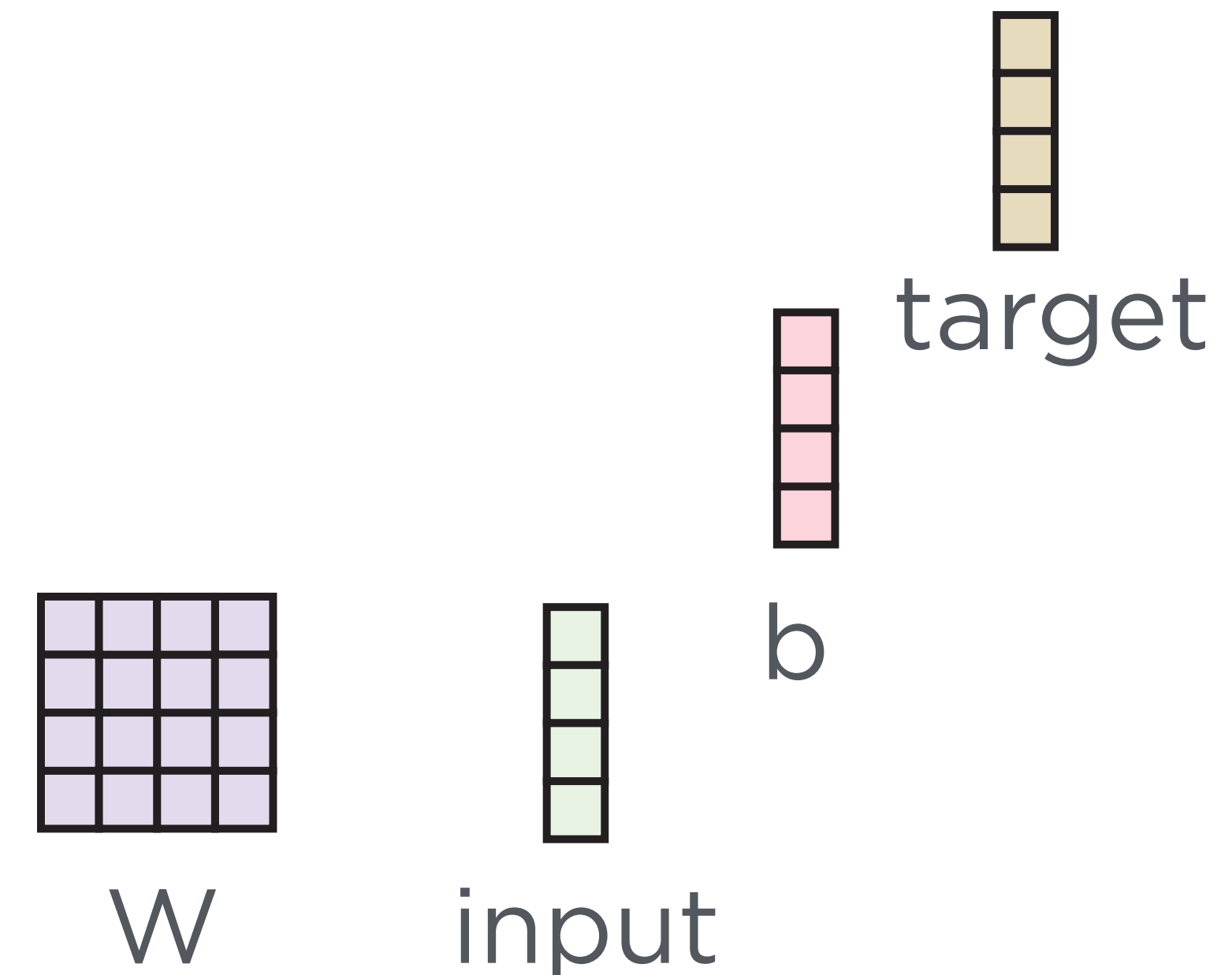
"Autodiff Engines"



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

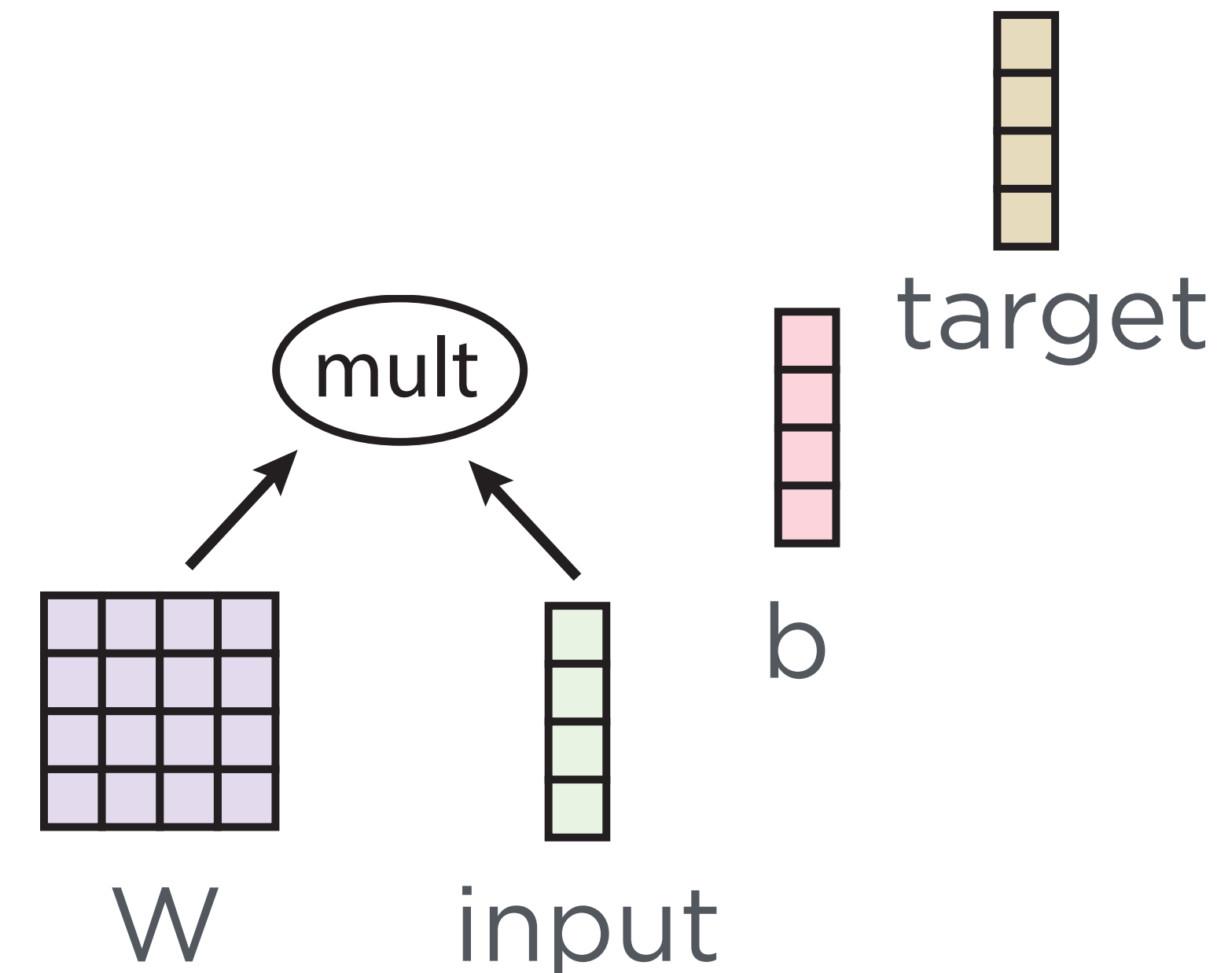
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

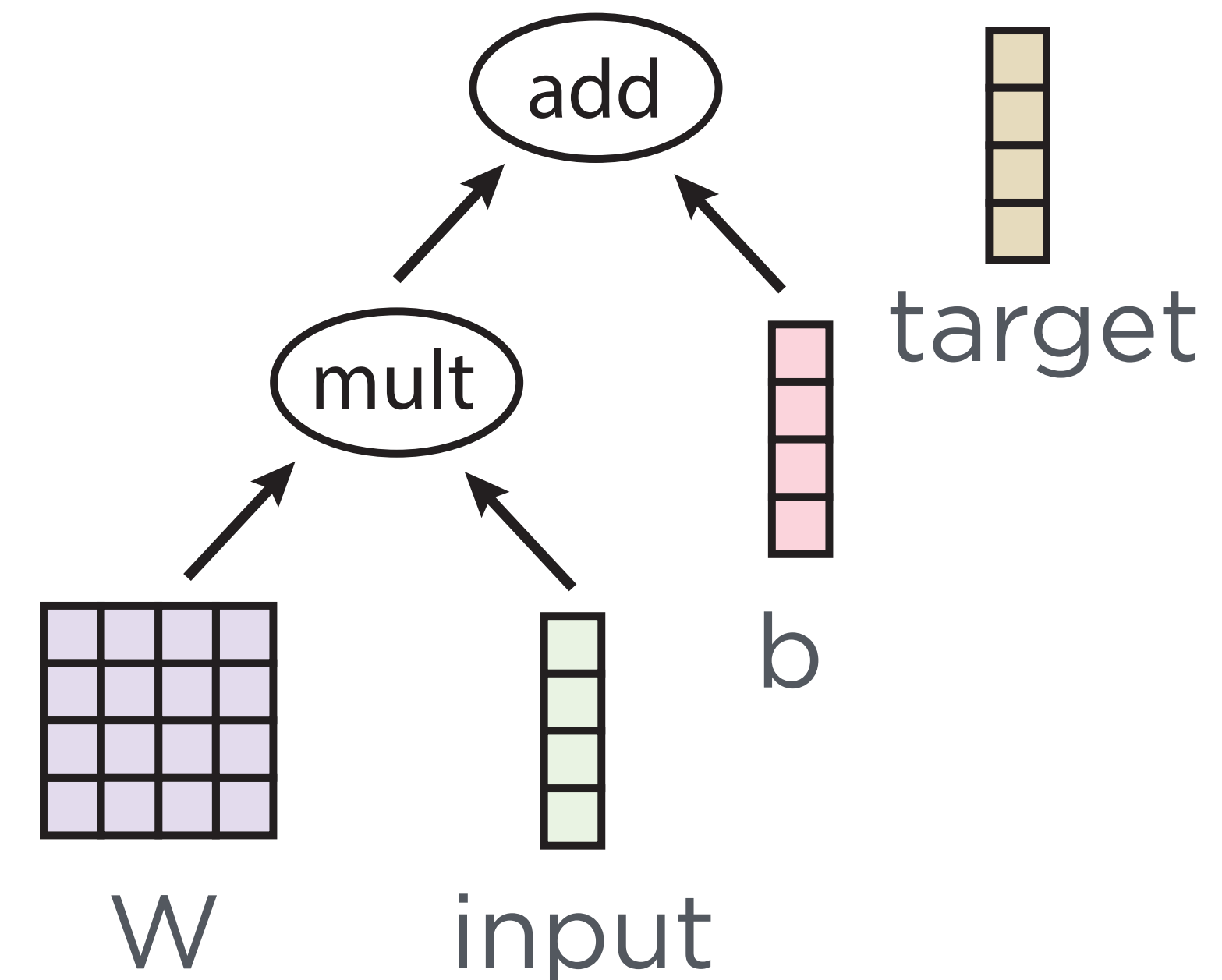
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

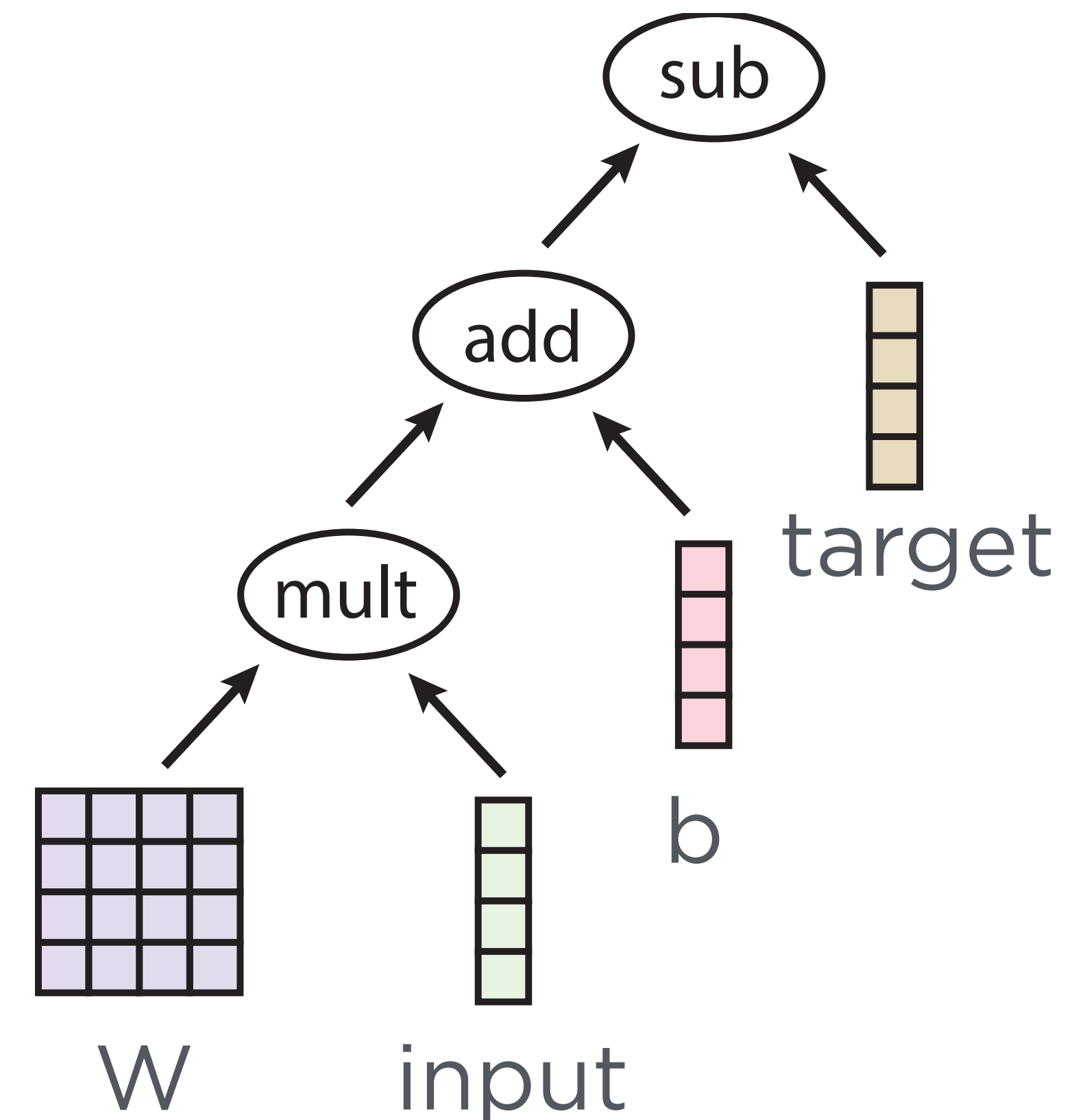
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

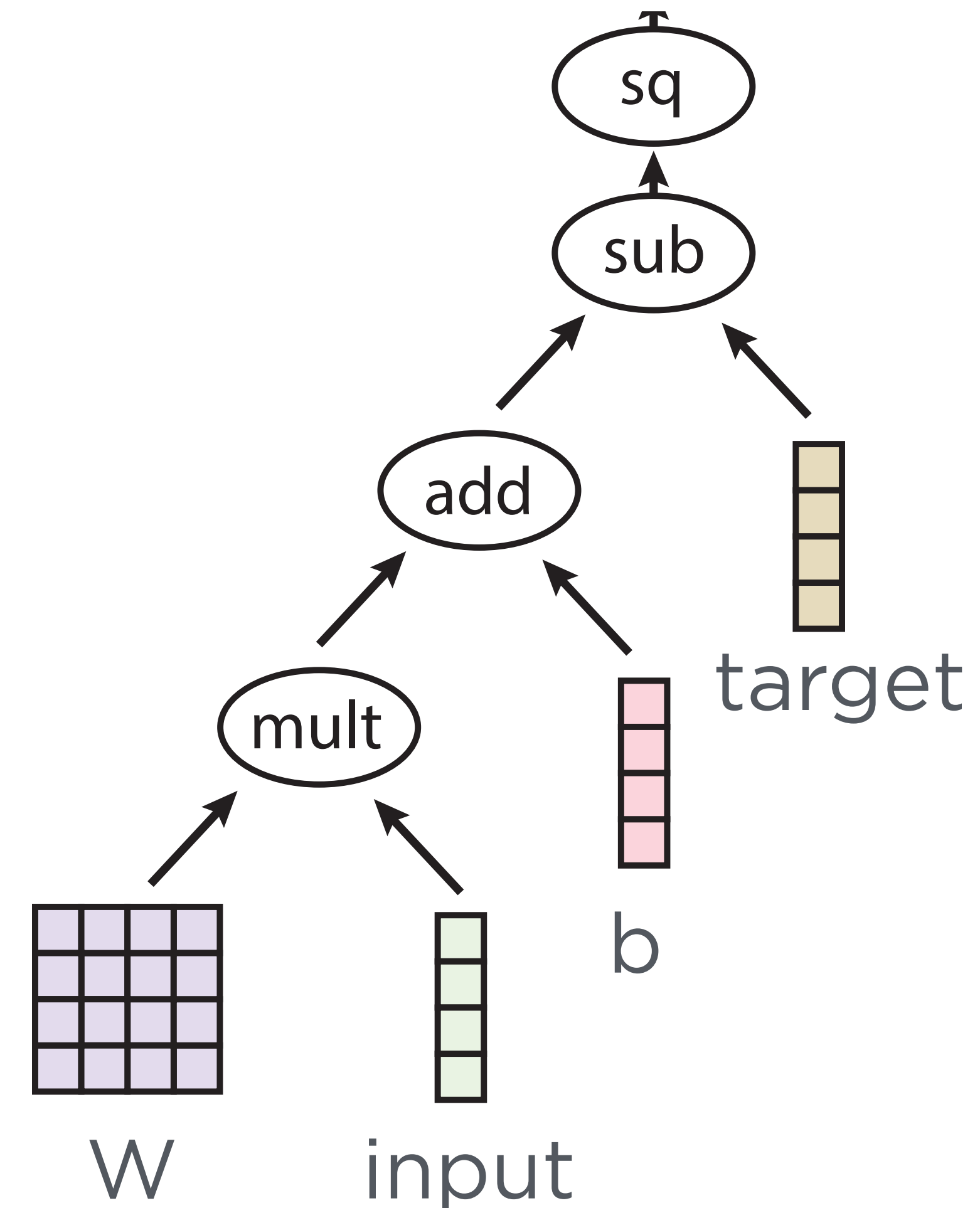
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

How is the compute graph built?

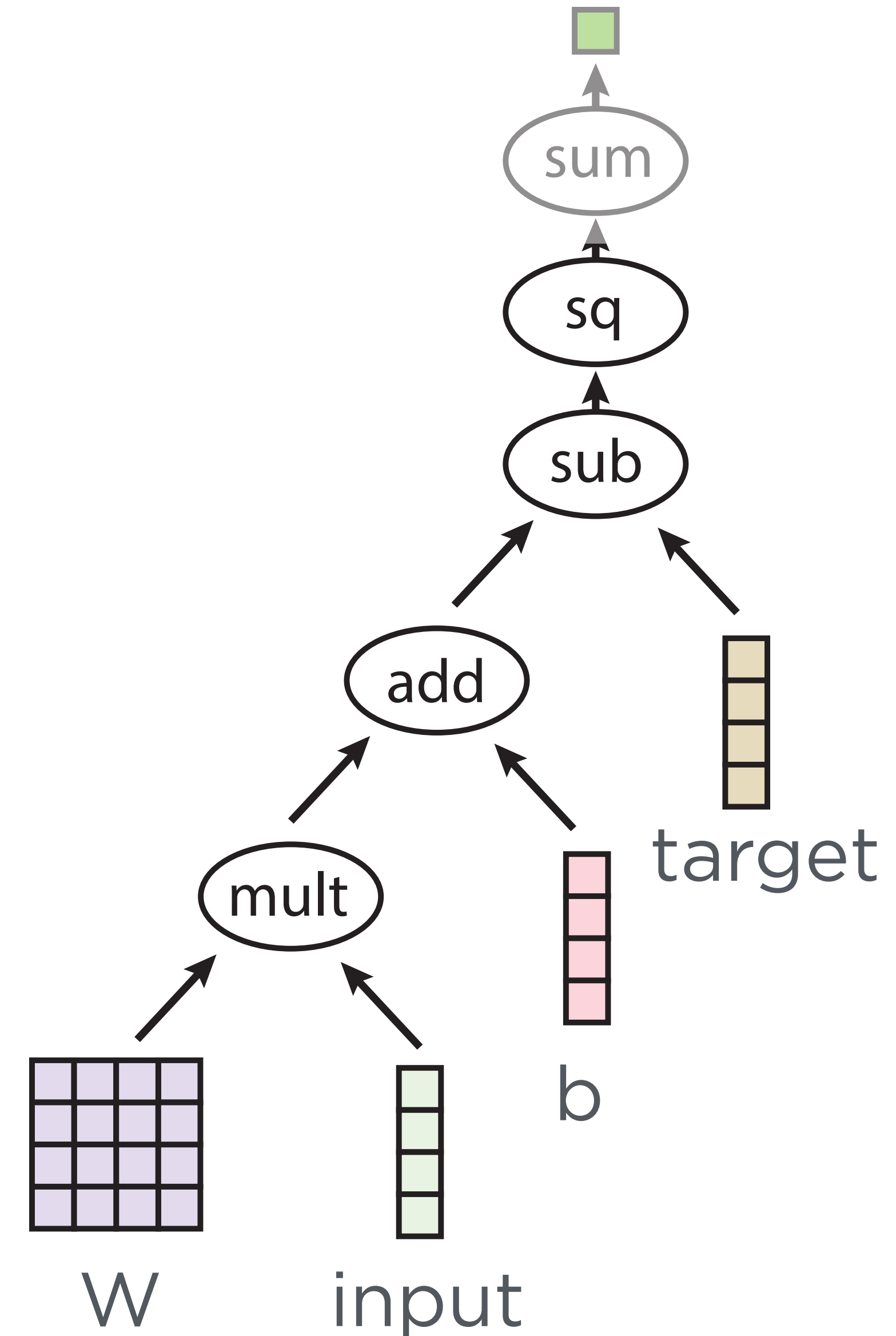
```
1  params = {W=torch.randn(4,4),b=torch.randn(4)}
2  input = torch.randn(4)
3  target = torch.randn(4)
4  ✓ function simpleFn(params, input, target)
5      local h1 = params.W*input
6      local h2 = h1 + params.b
7      local h3 = h2 - target
8      local h4 = torch.pow(h3,2)
9      local h5 = torch.sum(h4)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

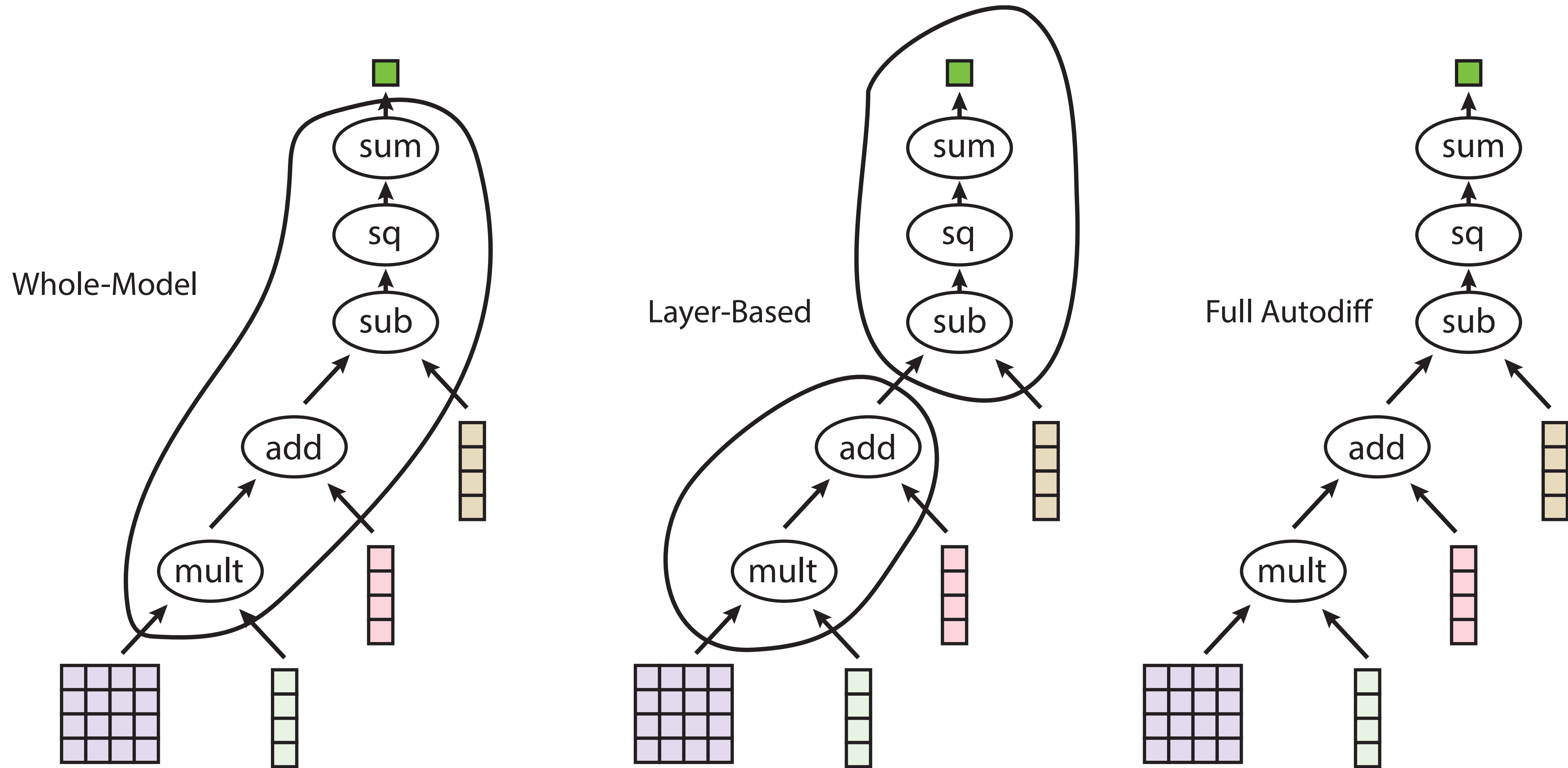
What is the graph?

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

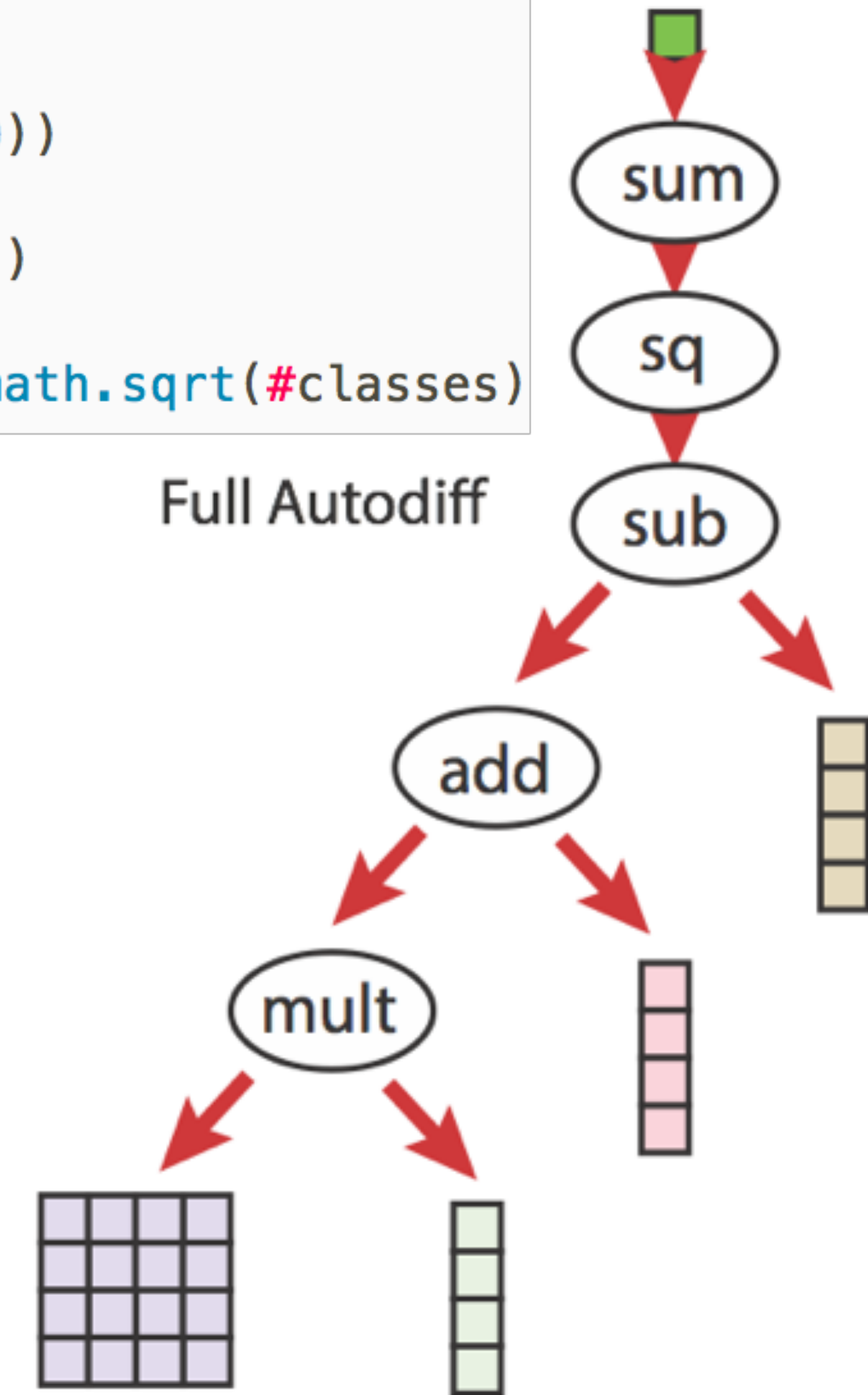


NEURAL NET THREE WAYS

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
```

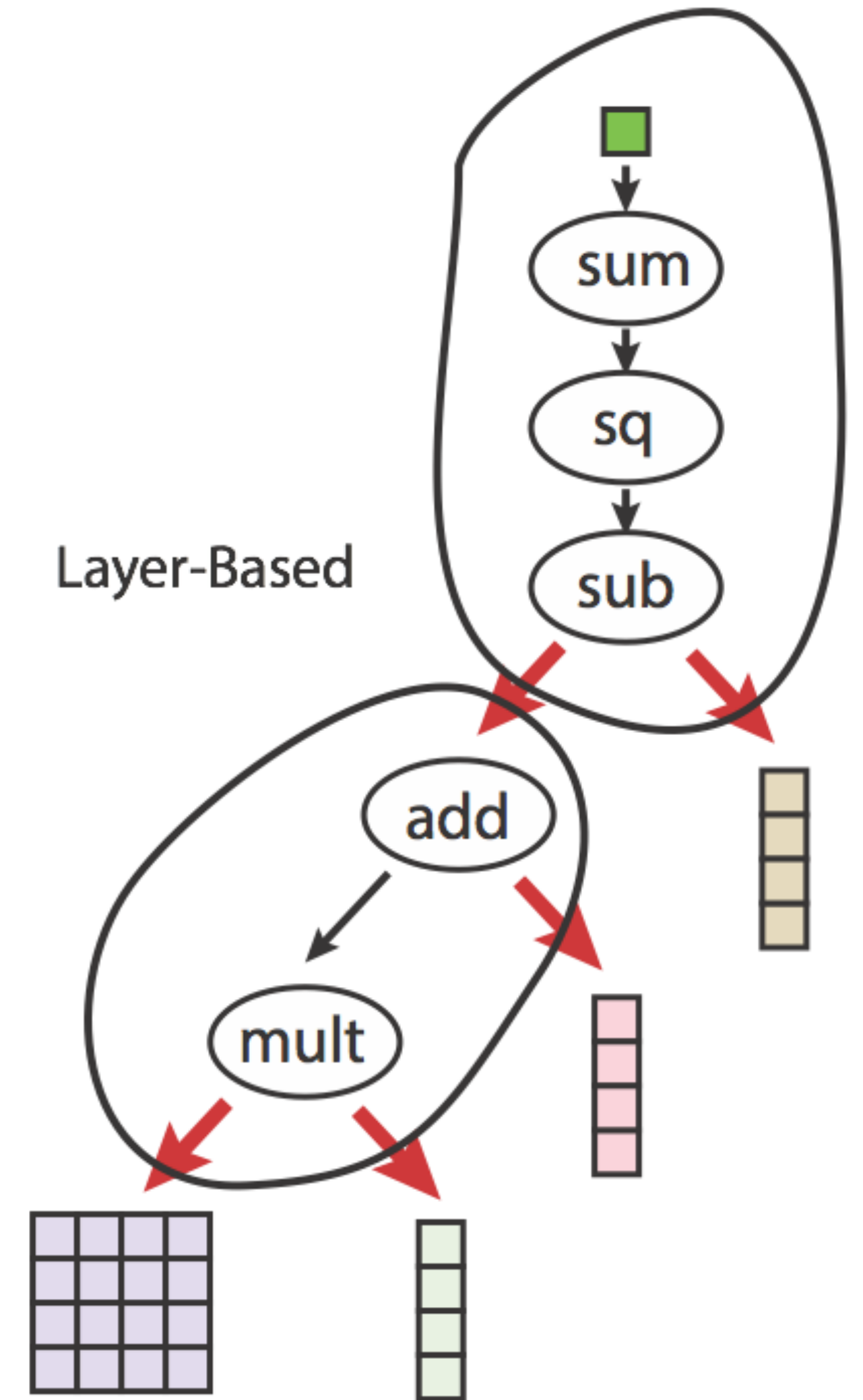


NEURAL NET THREE WAYS

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
```

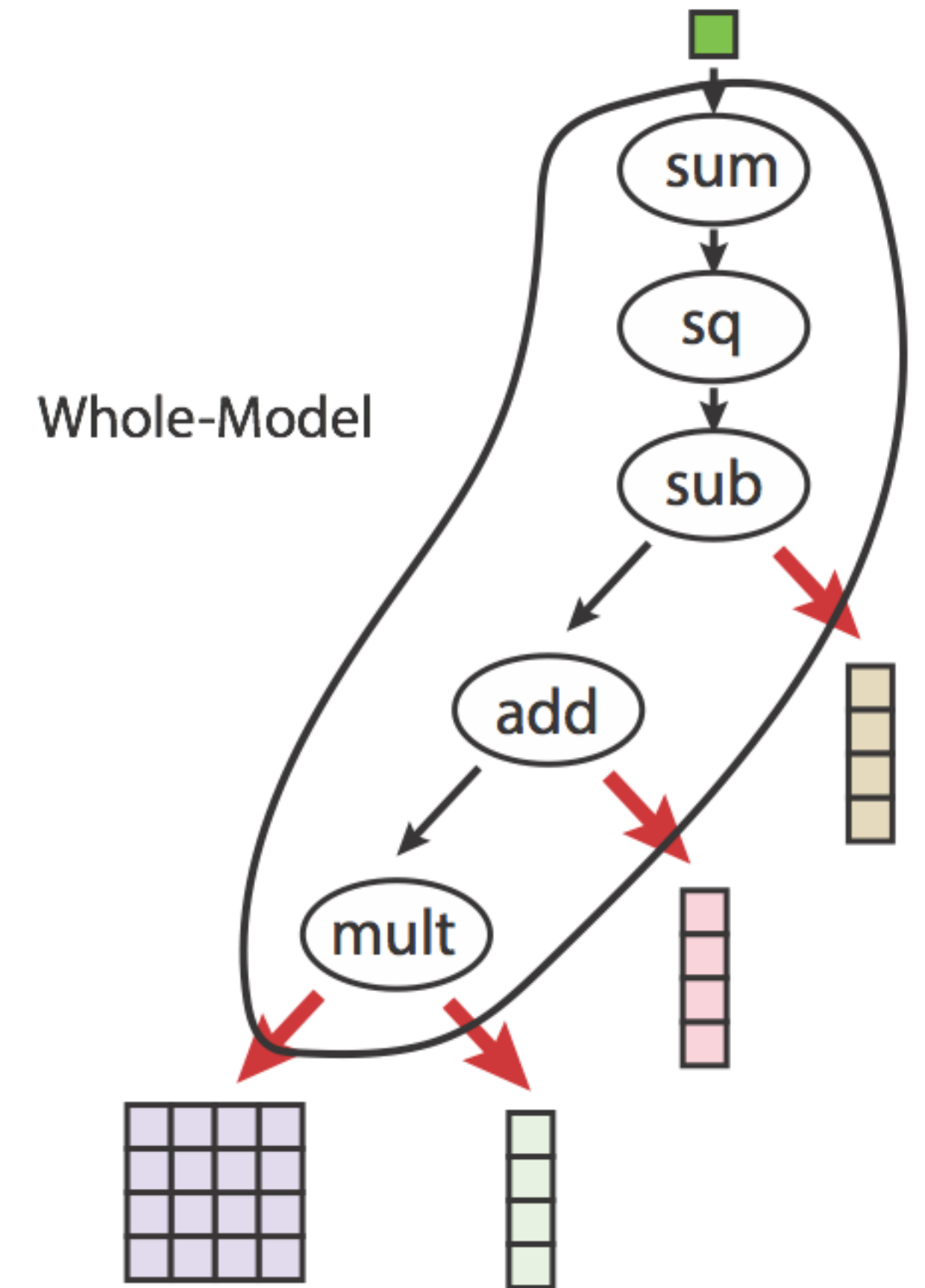


NEURAL NET THREE WAYS

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
```

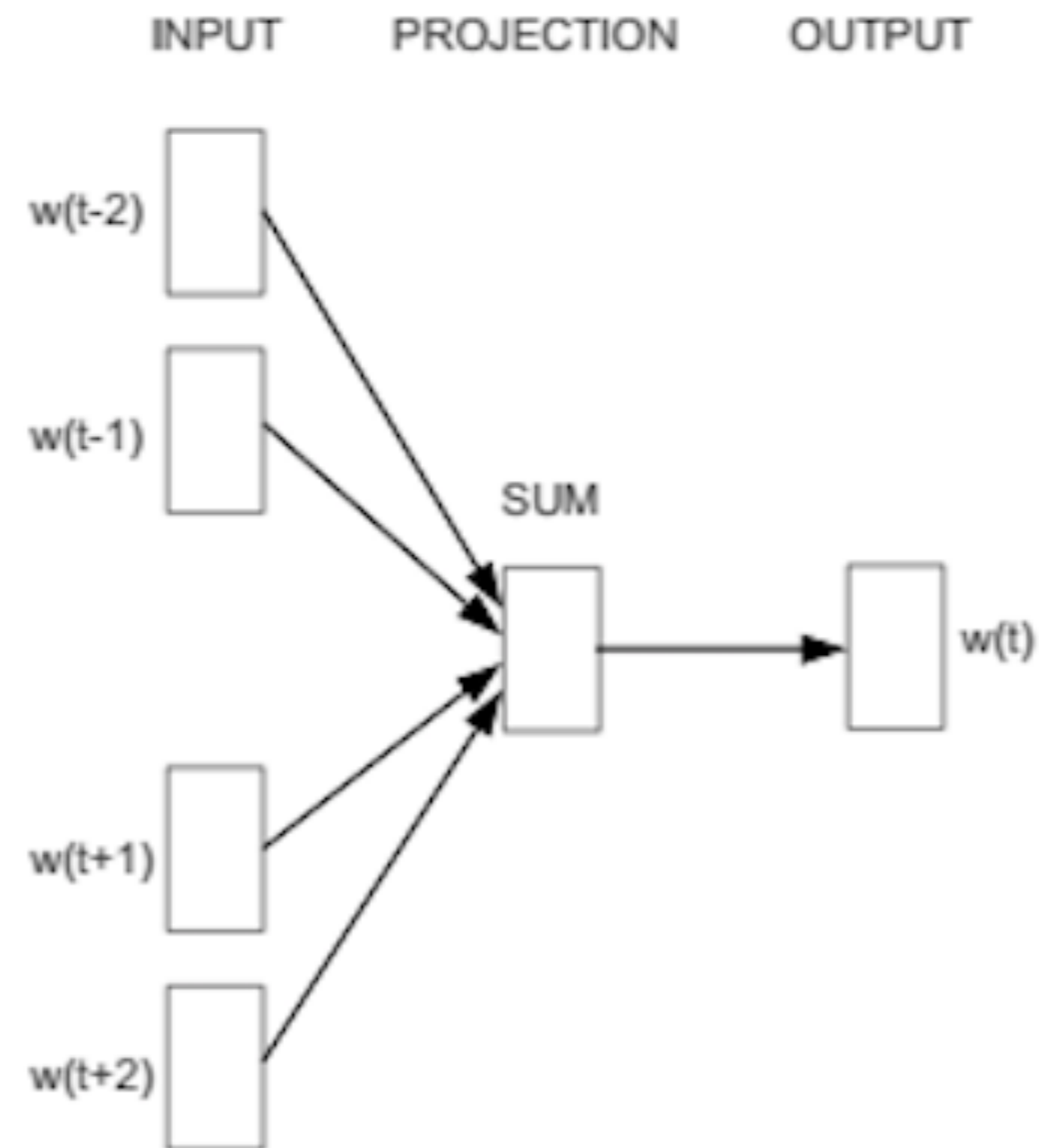


AUTODIFF & NLP

A trainable implementation of something like CBOW
(just an example! CBOW loss is more complicated)

```
def cbow(w, i):
```

```
    return abs( sum( w[i] - (w[i-2] + w[i-1] + w[i+1] + w[i+2]) ) )
```



CBOW



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 = $\text{grad}(\text{grad}(f))$. 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

```
1  # Install anaconda if you don't have it (instructions here for OS X)
2  wget http://repo.continuum.io/miniconda/Miniconda-latest-MacOSX-x86_64.sh
3  sh Miniconda-latest-MacOSX-x86_64.sh -b -p $HOME/anaconda
4
5  # Add anaconda to your $PATH
6  export PATH=$HOME/anaconda/bin:$PATH
7
8  # Install Lua & Torch
9  conda install lua=5.2 lua-science -c alexbw
10
11 # Available versions of Lua: 2.0, 5.1, 5.2, 5.3
12 # 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>

```
1  # Install anaconda if you don't have it (instructions here for OS X)
2  wget http://repo.continuum.io/miniconda/Miniconda-latest-MacOSX-x86_64.sh
3  sh Miniconda-latest-MacOSX-x86_64.sh -b -p $HOME/anaconda
4
5  # Add anaconda to your $PATH
6  export PATH=$HOME/anaconda/bin:$PATH
7
8  # Install Lua & Torch
9  conda install lua=5.2 lua-science -c alexbw
10
11 # Available versions of Lua: 2.0, 5.1, 5.2, 5.3
12 # 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>

```
1  # Install anaconda if you don't have it (instructions here for OS X)
2  wget http://repo.continuum.io/miniconda/Miniconda-latest-MacOSX-x86_64.sh
3  sh Miniconda-latest-MacOSX-x86_64.sh -b -p $HOME/anaconda
4
5  # Add anaconda to your $PATH
6  export PATH=$HOME/anaconda/bin:$PATH
7
8  # Install Lua & Torch
9  conda install lua=5.2 lua-science -c alexbw
10
11 # Available versions of Lua: 2.0, 5.1, 5.2, 5.3
12 # 2.0 is LuaJIT
```



QUESTIONS?

Happy to help. Find me at:

@awiltsch

awiltschko@twitter.com

github.com/alexbw

