MACHINE LEARNING WITH TORCH + AUTOGRAD

ALEX WILTSCHKO
RESEARCH ENGINEER
TWITTER

@AWILTSCH

MATERIAL DEVELOPED WITH

SOUMITH CHINTALA

HUGO LAROCHELLE

RYAN ADAMS

LUKE ALONSO

CLEMENT FARABET

FIRST HALF:

TORCH BASICS & OVERVIEW OF TRAINING NEURAL NETS

SECOND HALF:

AUTOMATIC DIFFERENTIATION AND TORCH-AUTOGRAD

torch

An array programming library for Lua, looks a lot like NumPy and Matlab

WHAT IS torch ?

- Interactive scientific computing framework in Lua

Strings, numbers, tables - a tiny introduction

In [ ]:
```
a = 'hello'
```

In [ ]:
```
print(a)
```

In [ ]:
```
b = {}
```

In [ ]:
```
b[1] = a
```

In [ ]:
```
print(b)
```

In [ ]:
```
b[2] = 30
```

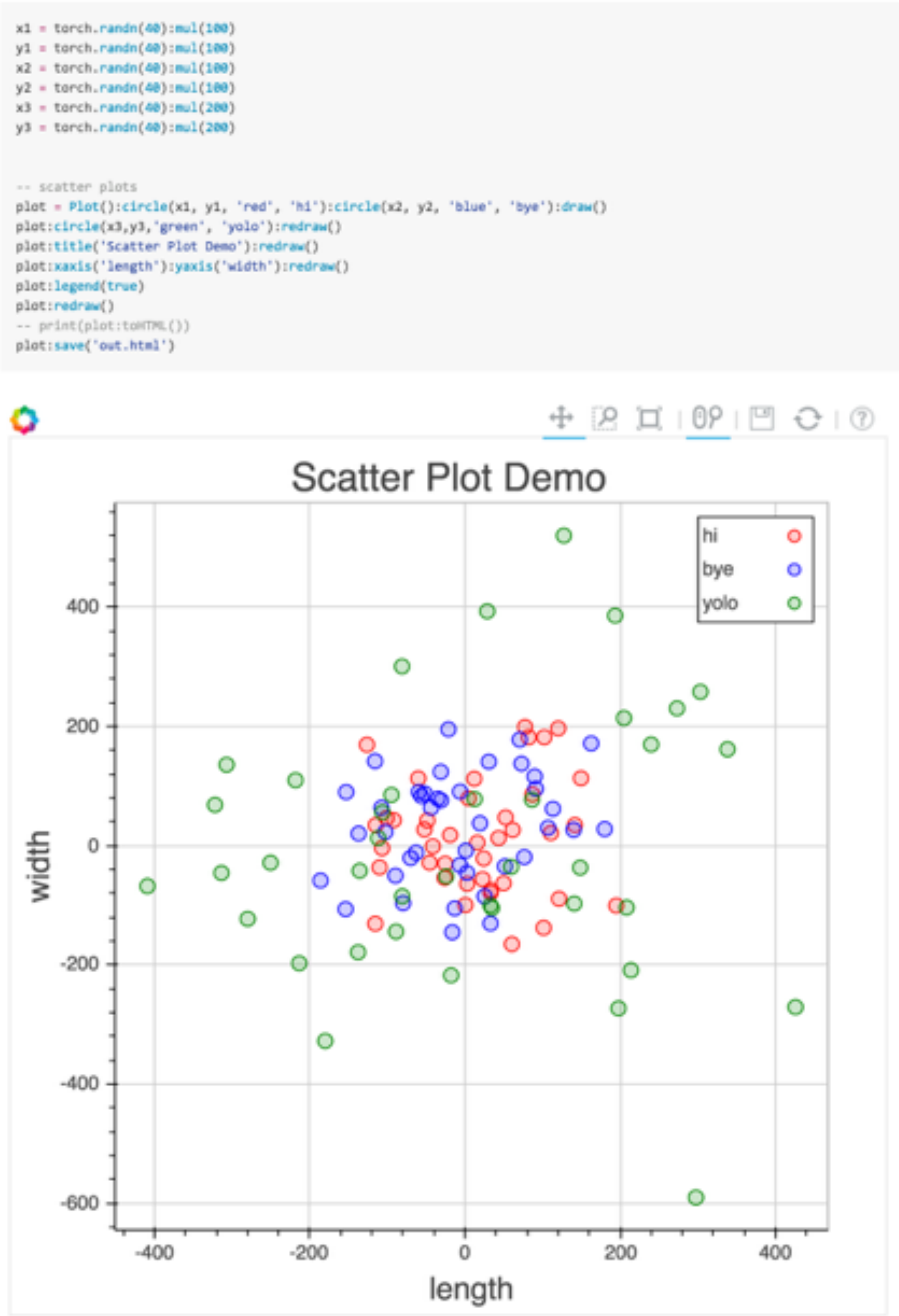
In [ ]:
```
for i=1,#b do -- the # operator is the length operator in Lua
    print(b[i])
end
```

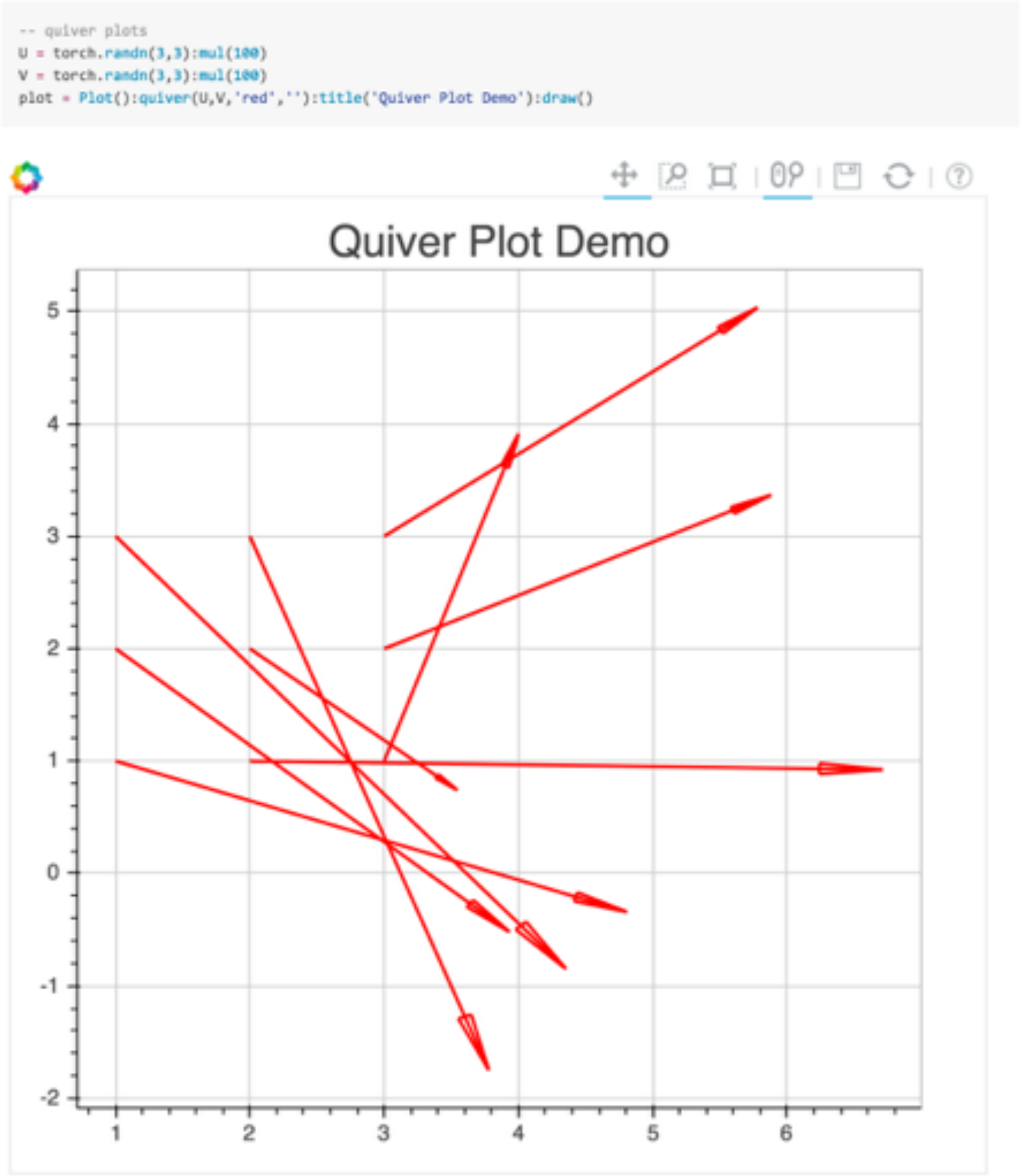
WHAT IS torch ?

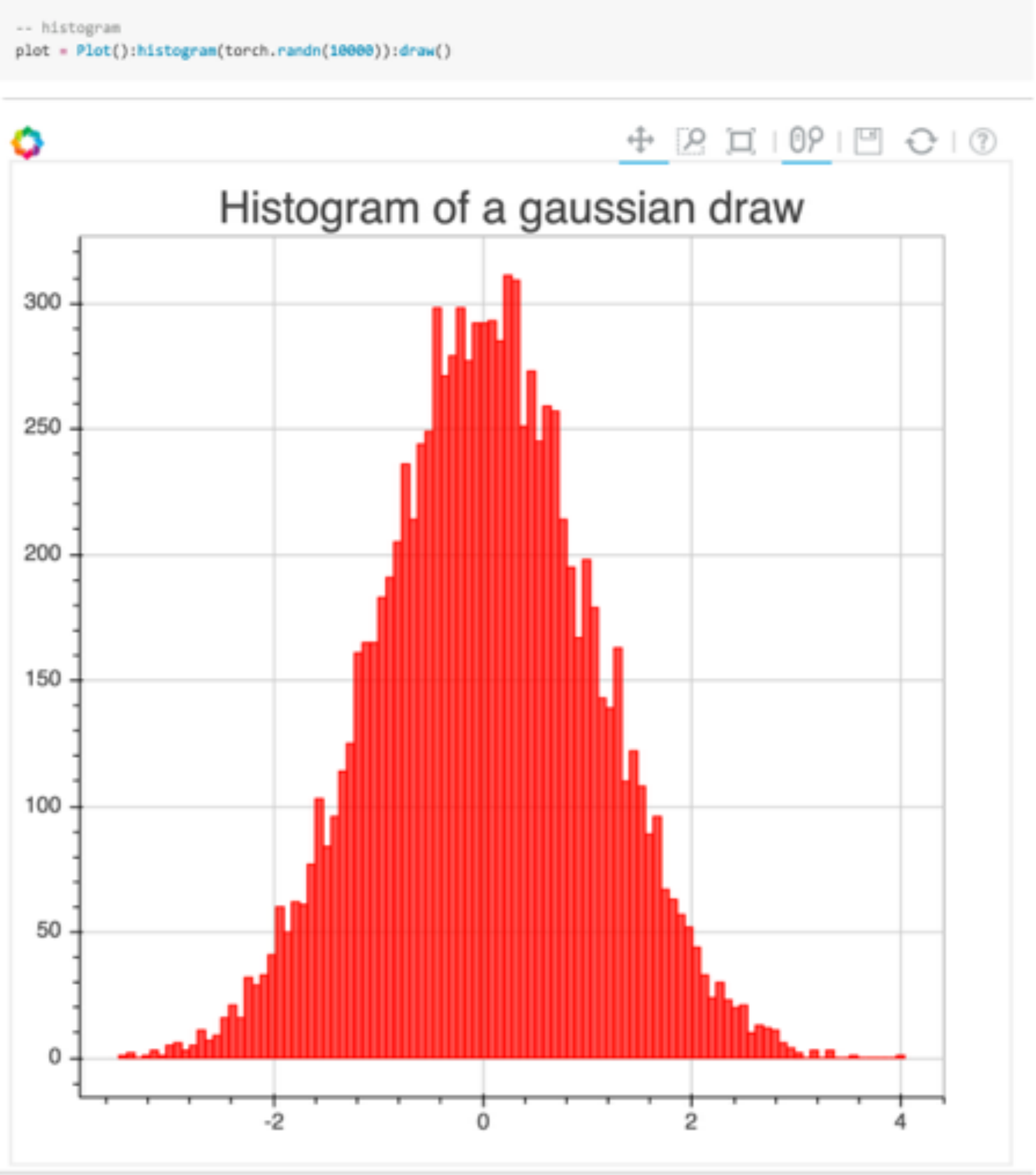
- 150+ Tensor functions
  - Linear algebra
  - Convolutions
  - Tensor manipulation
    - Narrow, index, mask, etc.
  - Logical operators
- Fully documented: https://github.com/torch/torch7/tree/master/doc

WHAT IS torch ?

- Similar to Matlab / Python+Numpy







WHAT IS torch ?

Lots of functions that can operate on tensors, all the basics, slicing, BLAS, LAPACK, cephes, rand

```
-- Scalar & tensor arithmetic
A = torch.eye(3)
b = 4
c = 2
print(A*b - c)
```

```
 2 -2 -2
-2  2 -2
-2 -2  2
[torch.DoubleTensor of size 3x3]
```

```
-- Max
print(torch.max(torch.FloatTensor{1,3,5}))
```

```
5
```

```
-- Clamp
torch.clamp(torch.range(0,4),0,2)
```

```
 0
 1
 2
 2
 2
[torch.DoubleTensor of size 5]
```

WHAT IS torch ?

Lots of functions that can operate on tensors, all the basics, slicing, BLAS, LAPACK, cephes, rand

```
-- Boolean fns
A = torch.range(1,5)
print(torch.le(A,3))
```

```
 1
 1
 1
 0
 0
[torch.ByteTensor of size 5]
```

WHAT IS torch ?

Lots of functions that can operate on tensors, all the basics, slicing, BLAS, LAPACK, cephes, rand

Special functions

```lua
-- Special functions
require 'cephes'
print(cephes.gamma(0.5))
```

```
1.7724538509055
```

```lua
print(cephes.atan2(3,1))
```

```
1.2490457723983
```

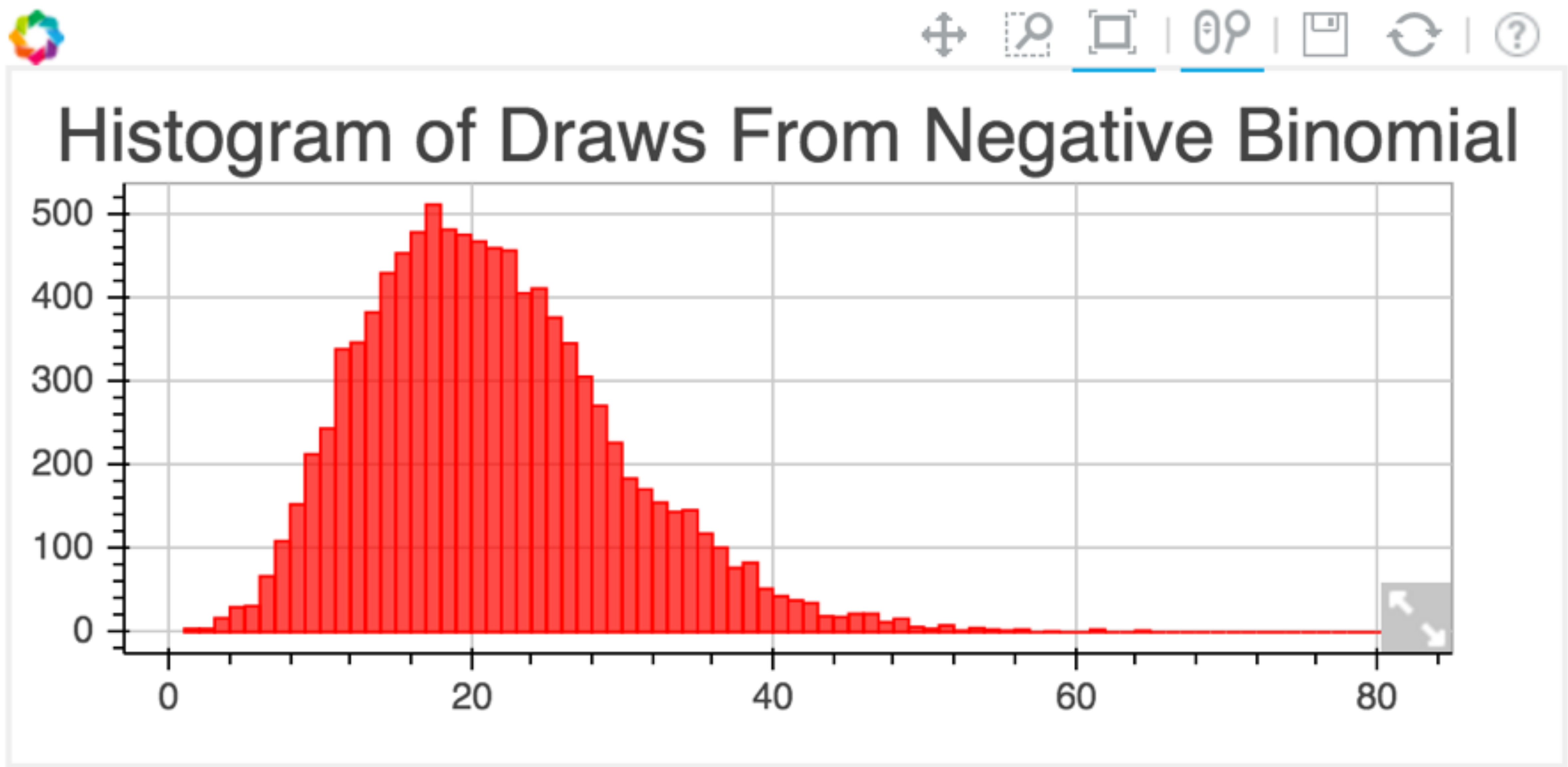
http://deepmind.github.io/torch-cephes/

WHAT IS torch ?

Lots of functions that can operate on tensors, all the basics, slicing, BLAS, LAPACK, cephes, rand

```
-- Sampling from a distribution
require 'randomkit'
a = torch.zeros(10000)
randomkit.negative_binomial(a,9,0.3)
```

```
Plot = require 'itorch.Plot'
local p = Plot()
    :histogram(a,80,1,80)
    :title("Histogram of Draws From Negative Binomial")
    :draw();
```



WHAT IS torch ?

- Inline help

```
In [10]: ?torch.cmul

Out[10]: ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
         [res] torch.cmul([res,] tensor1, tensor2)


         Element-wise multiplication of tensor1 by tensor2 .
         The number of elements must match, but sizes do not matter.
          > x = torch.Tensor(2, 2):fill(2)
         > y = torch.Tensor(4):fill(3)
         > x:cmul(y)
         > = x
          6   6
          6   6
         [torch.DoubleTensor of size 2x2]

          z = torch.cmul(x, y) returns a new Tensor .

          torch.cmul(z, x, y) puts the result in z .

          y:cmul(x) multiplies all elements of y with corresponding elements
         of x .

          z:cmul(x, y) puts the result in z .
         ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Good docs online

http://torch.ch/docs/

WHAT IS torch ?

- Little language overhead compared to Python / Matlab
- JIT compilation via LuaJIT
    - Fearlessly write for-loops

      Code snippet from a core package

```
function NarrowTable:updateOutput(input)
   for k,v in ipairs(self.output) do self.output[k] = nil end
   for i=1,self.length do
      self.output[i] = input[self.offset+i-1]
   end
   return self.output
end
```

- Plain Lua is ~10kLOC of C, small language

LUA IS DESIGNED TO INTEROPERATE WITH C

FFI allows easy integration with C

- The "FFI" allows easy integration with C code
- Been copied by many languages (e.g. cffi in Python)
- No Cython/SWIG required to integrate C code
- Lua originally designed to be embedded!
  - World of Warcraft
  - Adobe Lightroom
  - Redis
  - nginx
- Lua originally chosen for *embedded* machine learning

WHAT IS torch ?

- Easy integration into and from C
- Example: using CuDNN functions

```
for g = 0, self.groups - 1 do
    errcheck('cudnnConvolutionForward', cudnn.getHandle(),
             one:data(),
             self.iDesc[0], input:data() + g*self.input_offset,
             self.weightDesc[0], self.weight:data() + g*self.weight_offset,
             self.convDesc[0], self.fwdAlgType[0],
             self.extraBuffer:data(), self.extraBufferSizeInBytes,
             zero:data(),
             self.oDesc[0], self.output:data() + g*self.output_offset);
end
```

WHAT IS torch ?

- Strong GPU support

CUDA Tensors

Tensors can be moved onto GPU using the :cuda function

In [ ]:
```
require 'cutorch';
a = a:cuda()
b = b:cuda()
c = c:cuda()
c:mm(a,b) -- done on GPU
```

torch COMMUNITY

NVIDIA

Facebook AI Research

UNIVERSITY OF OXFORD

École des Ponts ParisTech

NYU

canary

inria

Stanford University

PURDUE UNIVERSITY

IIIT-H

idiap RESEARCH INSTITUTE

VisionLabs visual recognition company

Etsy

Yandex

element

TERADEEP

AMD

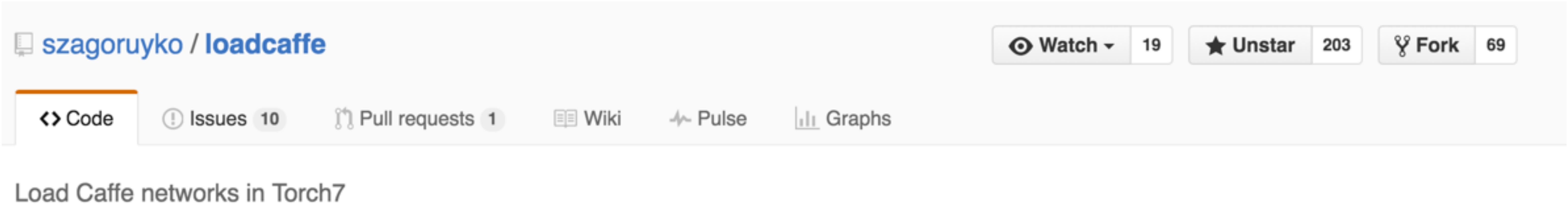
IBM

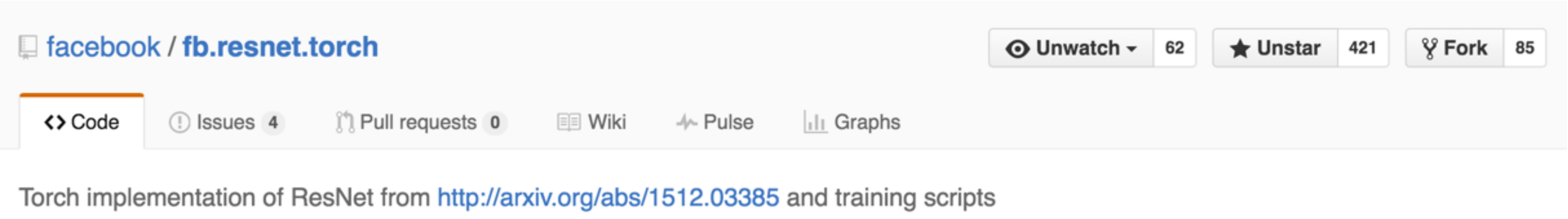
MULTICOREWARE

Moodstocks

torch COMMUNITY

Code for cutting edge models shows up for Torch very quickly





torch COMMUNITY

karpathy / neuraltalk2

Watch 171 | Unstar 2,153 | Fork 320

Code | Issues 45 | Pull requests 4 | Wiki | Pulse | Graphs

Efficient Image Captioning code in Torch, runs on GPU

NeuralTalk2

Recurrent Neural Network captions your images. Now much faster and better than the original NeuralTalk. Compared to the original NeuralTalk this implementation is **batched, uses Torch, runs on a GPU, and supports CNN finetuning**. All of these together result in quite a large increase in training speed for the Language Model (~100x), but overall not as much because we also have to forward a VGGNet. However, overall very good models can be trained in 2-3 days, and they show a much better performance.

This is an early code release that works great but is slightly hastily released and probably requires some code reading of inline comments (which I tried to be quite good with in general). I will be improving it over time but wanted to push the code out there because I promised it to too many people.

This current code (and the pretrained model) gets ~0.9 CIDEr, which would place it around spot #8 on the codalab leaderboard. I will submit the actual result soon.


a man is playing tennis on a tennis court


a train is traveling down the tracks at a train station


a cake with a slice cut out of it


a bench sitting on a patch of grass next to a sidewalk

You can find a few more example results on the demo page. These results will improve a bit more once the last few bells and whistles are in place (e.g. beam search, ensembling, reranking).

There's also a fun video by @kcimc, where he runs a neuraltalk2 pretrained model in real time on his laptop during a walk in Amsterdam.

torch COMMUNITY

jcjohnson / **neural-style**

Watch 389 | Unstar 7,152 | Fork 901

Code | Issues 98 | Pull requests 12 | Wiki | Pulse | Graphs

Torch implementation of neural style algorithm

neural-style

This is a torch implementation of the paper A Neural Algorithm of Artistic Style by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge.

The paper presents an algorithm for combining the content of one image with the style of another image using convolutional neural networks. Here's an example that maps the artistic style of The Starry Night onto a night-time photograph of the Stanford campus:

Neural Conversational Model in Torch

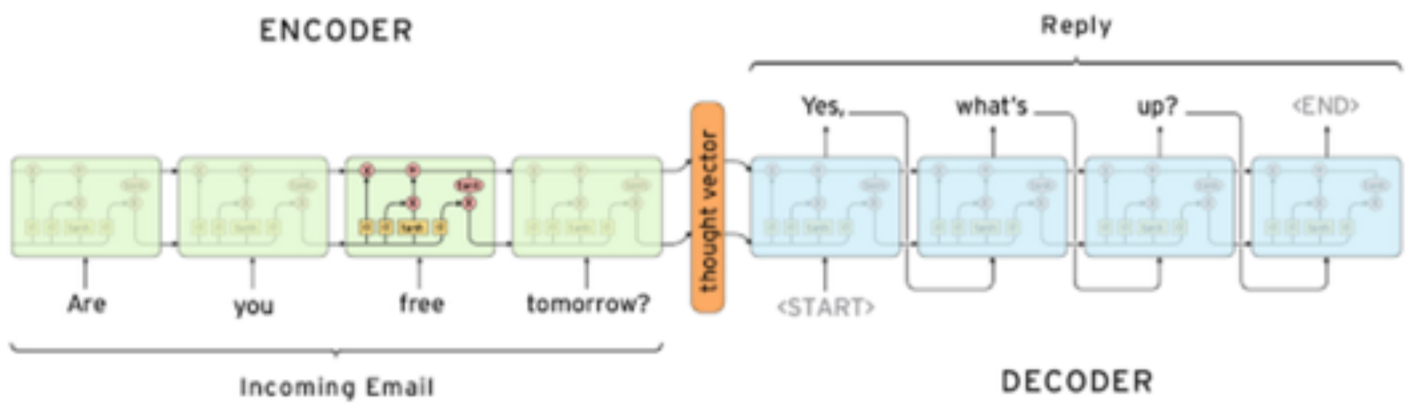
This is an attempt at implementing Sequence to Sequence Learning with Neural Networks (seq2seq) and reproducing the results in A Neural Conversational Model (aka the Google chatbot).

The Google chatbot paper became famous after cleverly answering a few philosophical questions, such as:

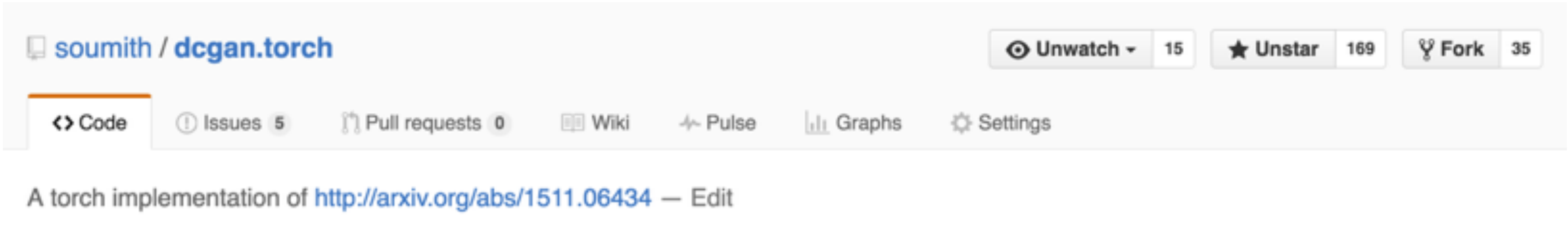
> **Human:** What is the purpose of living?
> **Machine:** To live forever.

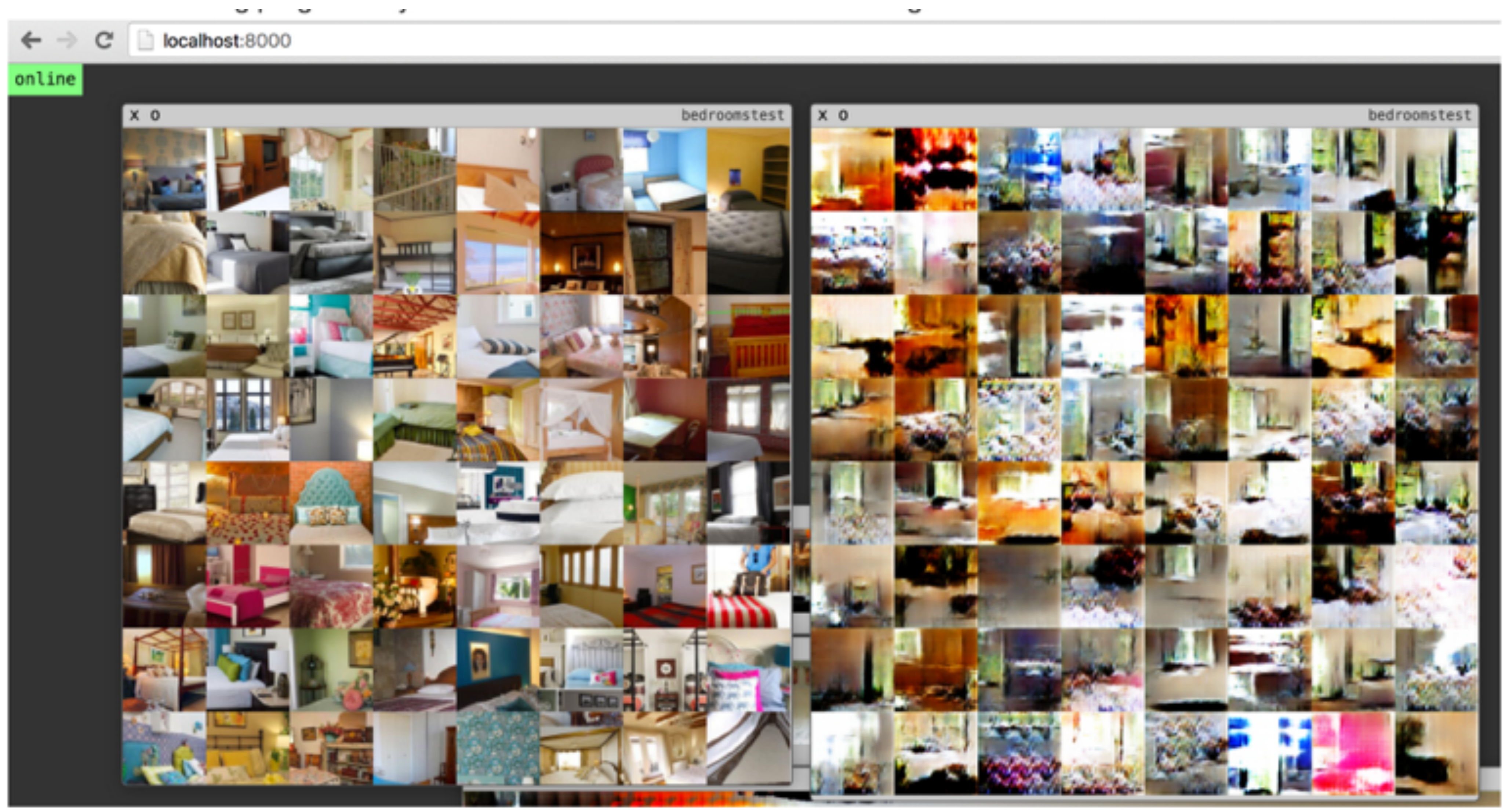
How it works

The model is based on two LSTM layers. One for encoding the input sentence into a "thought vector", and another for decoding that vector into a response. This model is called Sequence-to-sequence or seq2seq.



Source: *http://googleresearch.blogspot.ca/2015/11/computer-respond-to-this-email.html* torch COMMUNITY





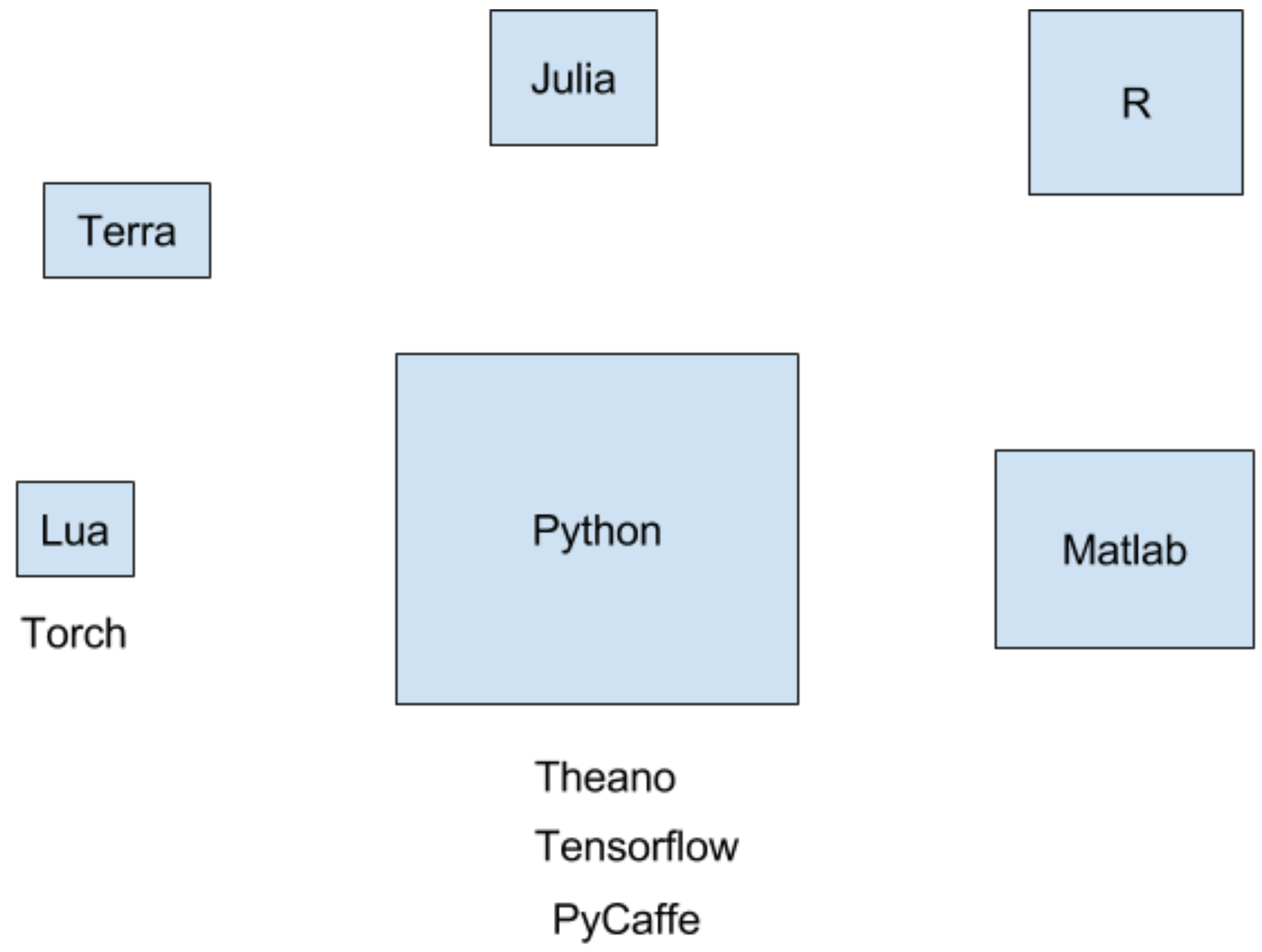
TORCH - WHERE DOES IT FIT?

How big is its ecosystem?

Smaller than Python for general data science

Strong for deep learning

Switching from Python to Lua can be smooth



TORCH - WHERE DOES IT FIT?

Is it for research or production? It can be for both

But mostly used for research.

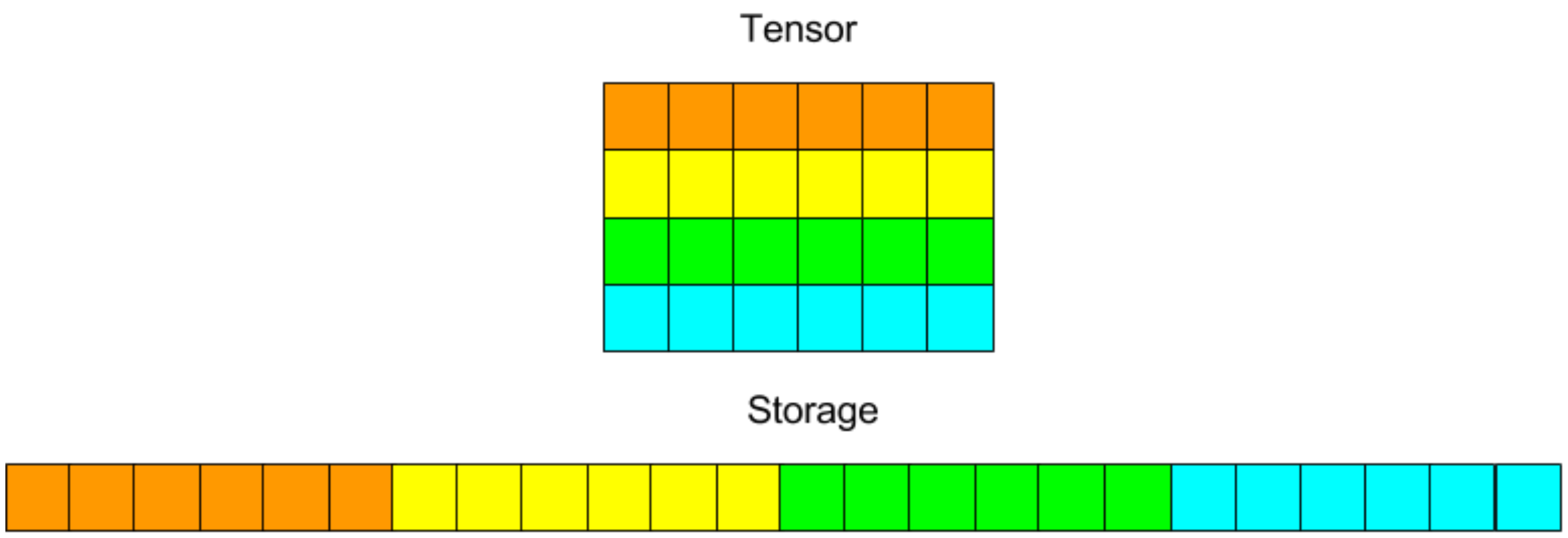
There is no silver bullet

CORE PHILOSOPHY

- Interactive computing
  - No compilation time
- Imperative programming
  - Write code like you always did, not computation graphs in a "mini-language" or DSL
- Minimal abstraction
  - Thinking linearly
- Maximal Flexibility
  - No constraints on interfaces or classes

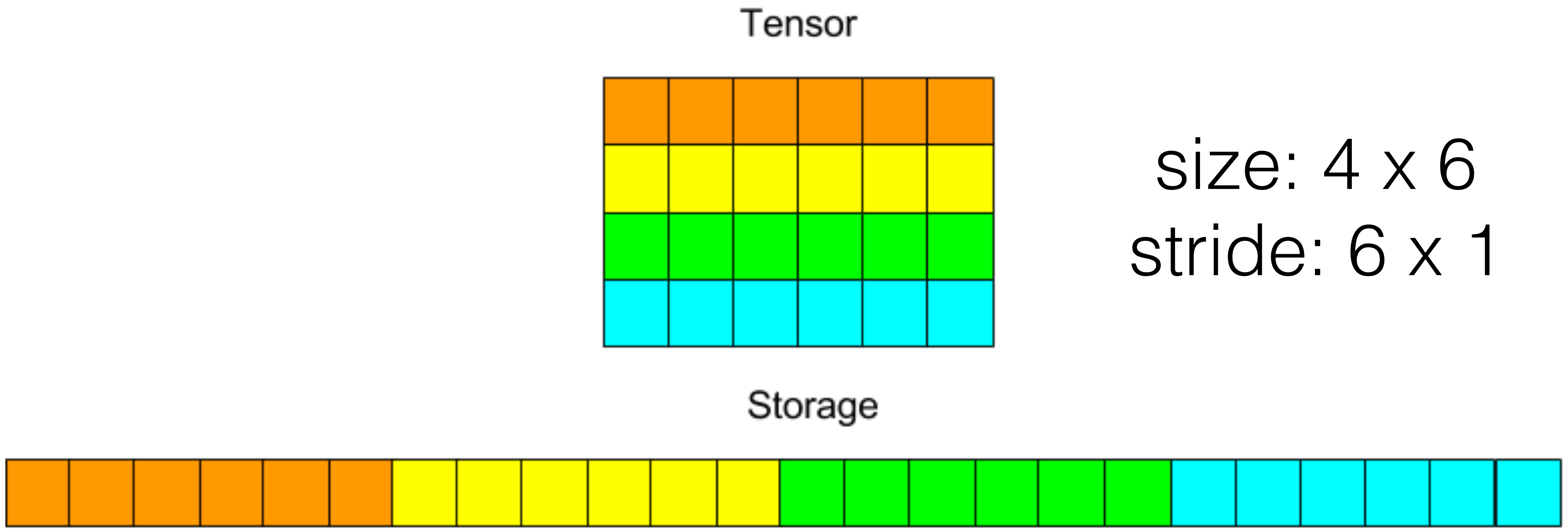
TENSORS AND STORAGES

- Tensor = n-dimensional array
- Row-major in memory



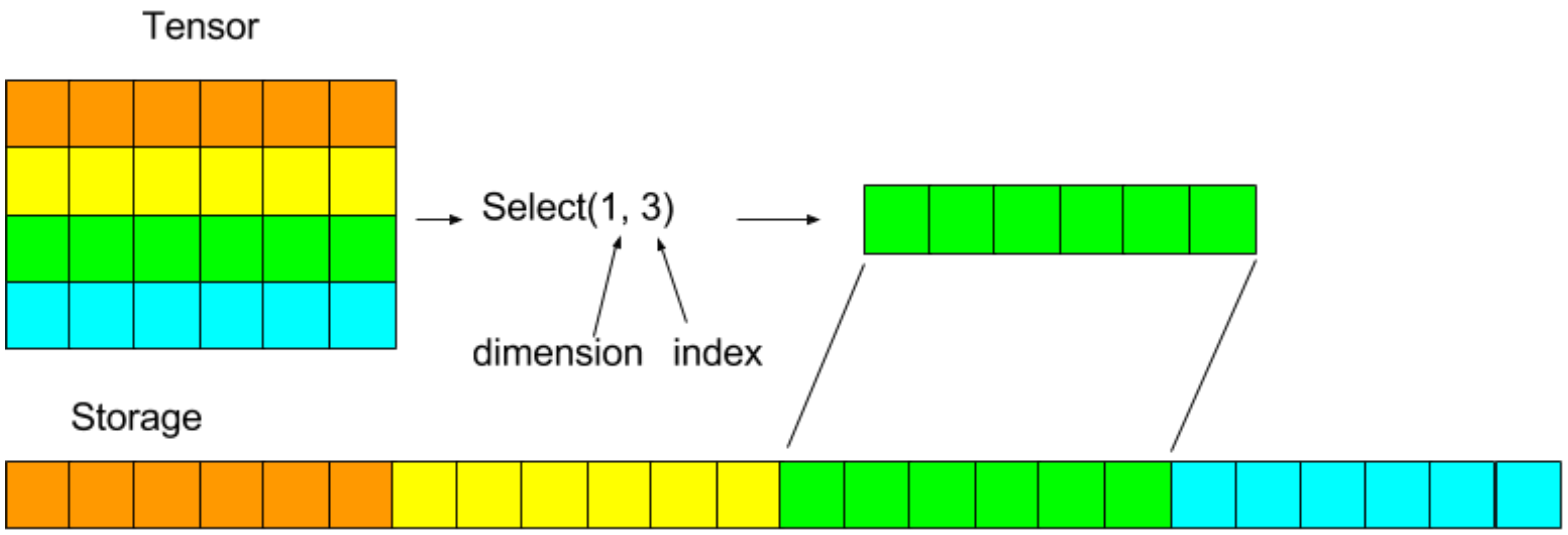
TENSORS AND STORAGES

- Tensor = n-dimensional array
- Row-major in memory



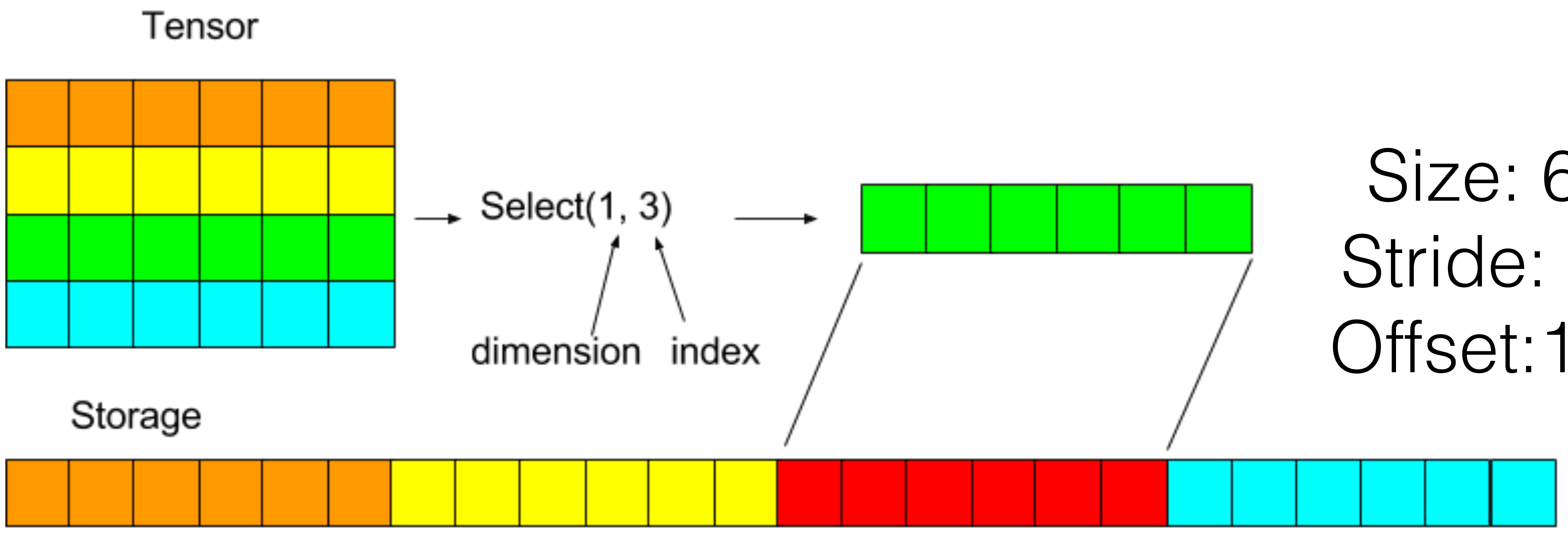
TENSORS AND STORAGES

- Tensor = n-dimensional array
- 1-indexed



TENSORS AND STORAGES

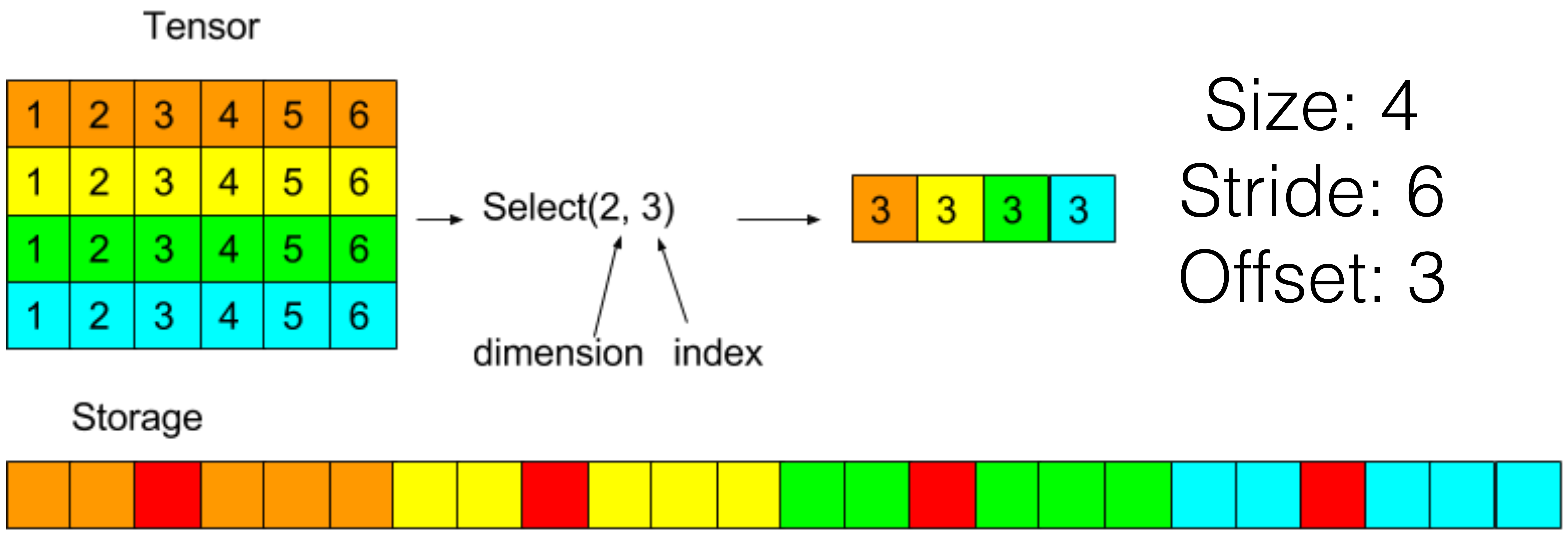
- Tensor = n-dimensional array
- Tensor: size, stride, storage, storageOffset



Size: 6
Stride: 1
Offset:13

TENSORS AND STORAGES

- Tensor = n-dimensional array
- Tensor: size, stride, storage, storageOffset



Size: 4
Stride: 6
Offset: 3

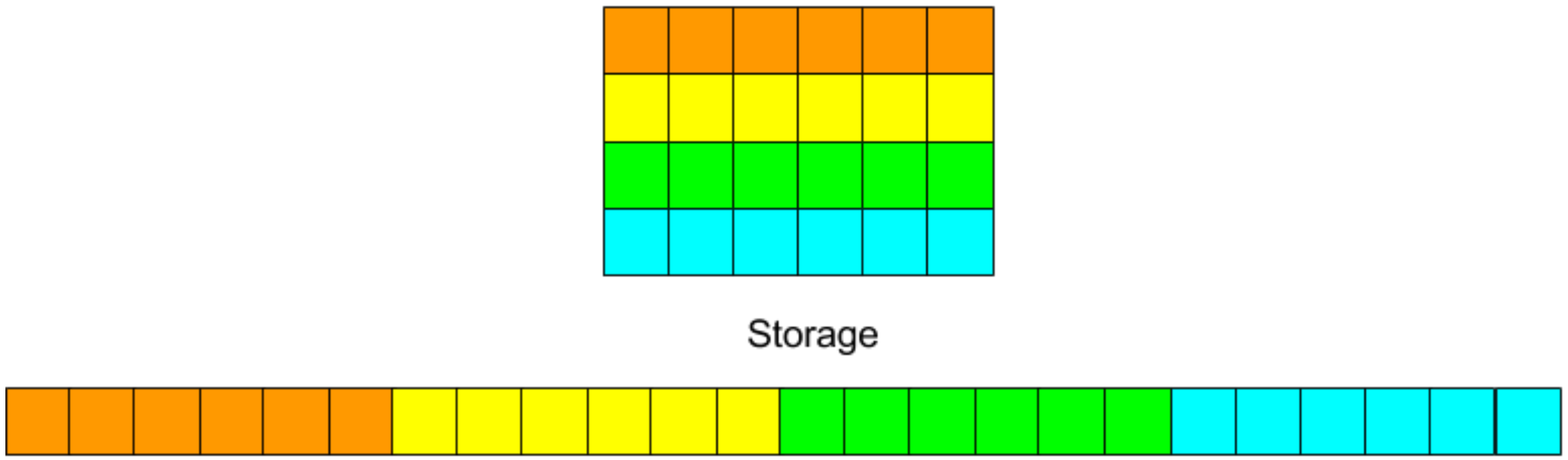
TENSORS AND STORAGES

In [1]:
```
require 'torch';
```

In [2]:
```
a = torch.DoubleTensor(4, 6) -- DoubleTensor, uninitialized memory
a:uniform() -- fills a with uniform noise with mean = 0, stdv = 1
```

In [3]:
```
print(a)
```

Out[3]:
```
 0.4332  0.5716  0.5750  0.8167  0.1997  0.6187
 0.7775  0.3575  0.0749  0.4028  0.0532  0.4481
 0.5088  0.1795  0.6948  0.5700  0.7679  0.6176
 0.9225  0.7270  0.2223  0.1087  0.2717  0.8853
[torch.DoubleTensor of size 4x6]
```



TENSORS AND STORAGES

```
In [1]: require 'torch';
```

```
In [2]: a = torch.DoubleTensor(4, 6) -- DoubleTensor, uninitialized memory
        a:uniform() -- fills a with uniform noise with mean = 0, stdv = 1
```

```
In [3]: print(a)

Out[3]:  0.4332  0.5716  0.5750  0.8167  0.1997  0.6187
         0.7775  0.3575  0.0749  0.4028  0.0532  0.4481
         0.5088  0.1795  0.6948  0.5700  0.7679  0.6176
         0.9225  0.7270  0.2223  0.1087  0.2717  0.8853
        [torch.DoubleTensor of size 4x6]
```

```
In [4]: b = a:select(1, 3)
```

```
In [5]: print(b)

Out[5]:  0.5088
         0.1795
         0.6948
         0.5700
         0.7679
         0.6176
        [torch.DoubleTensor of size 6]
```

TENSORS AND STORAGES

Underlying storage is shared

```
In [6]: b:fill(3);

In [7]: print(b)

Out[7]:  3
         3
         3
         3
         3
         3
        [torch.DoubleTensor of size 6]


In [8]: print(a)

Out[8]:  0.4332  0.5716  0.5750  0.8167  0.1997  0.6187
         0.7775  0.3575  0.0749  0.4028  0.0532  0.4481
         3.0000  3.0000  3.0000  3.0000  3.0000  3.0000
         0.9225  0.7270  0.2223  0.1087  0.2717  0.8853
        [torch.DoubleTensor of size 4x6]
```

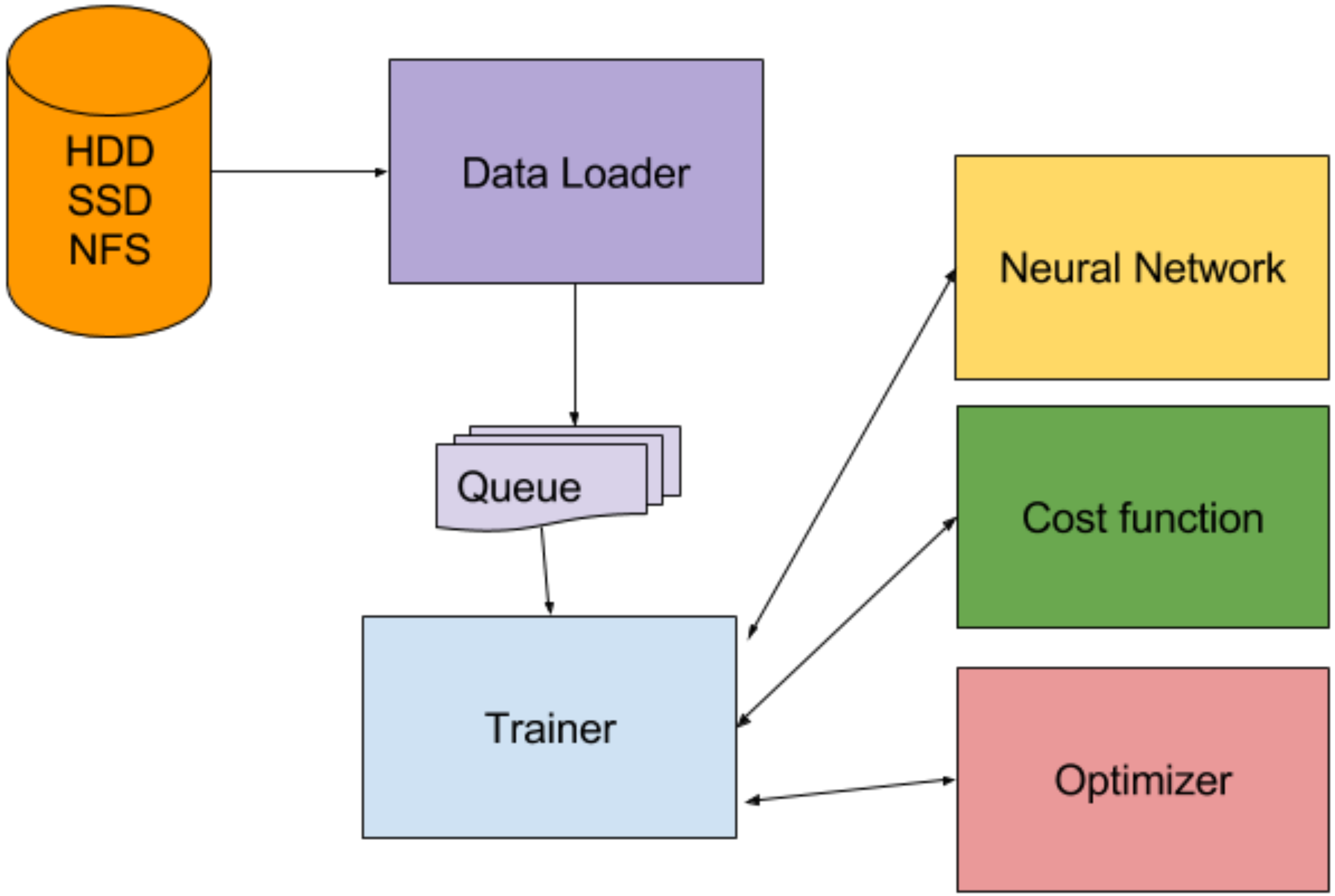
TENSORS AND STORAGES

- GPU support for all operations:
  - require ‘cutorch’
  - torch.CudaTensor = torch.FloatTensor on GPU
- Fully multi-GPU compatible

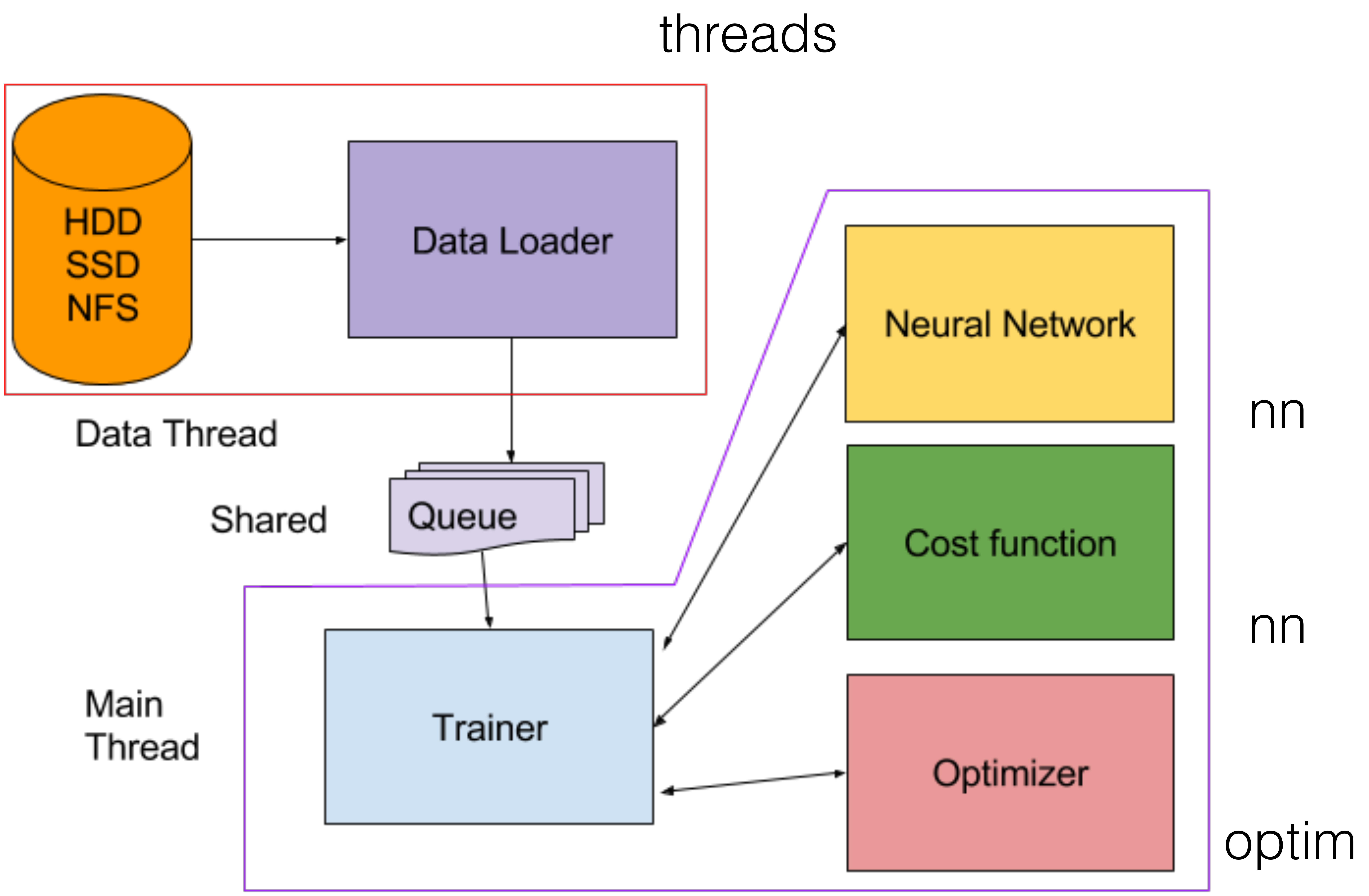
```
In [ ]: require 'cutorch'
        a = torch.CudaTensor(4, 6):uniform()
        b = a:select(1, 3)
        b:fill(3)
```

TRAINING CYCLE

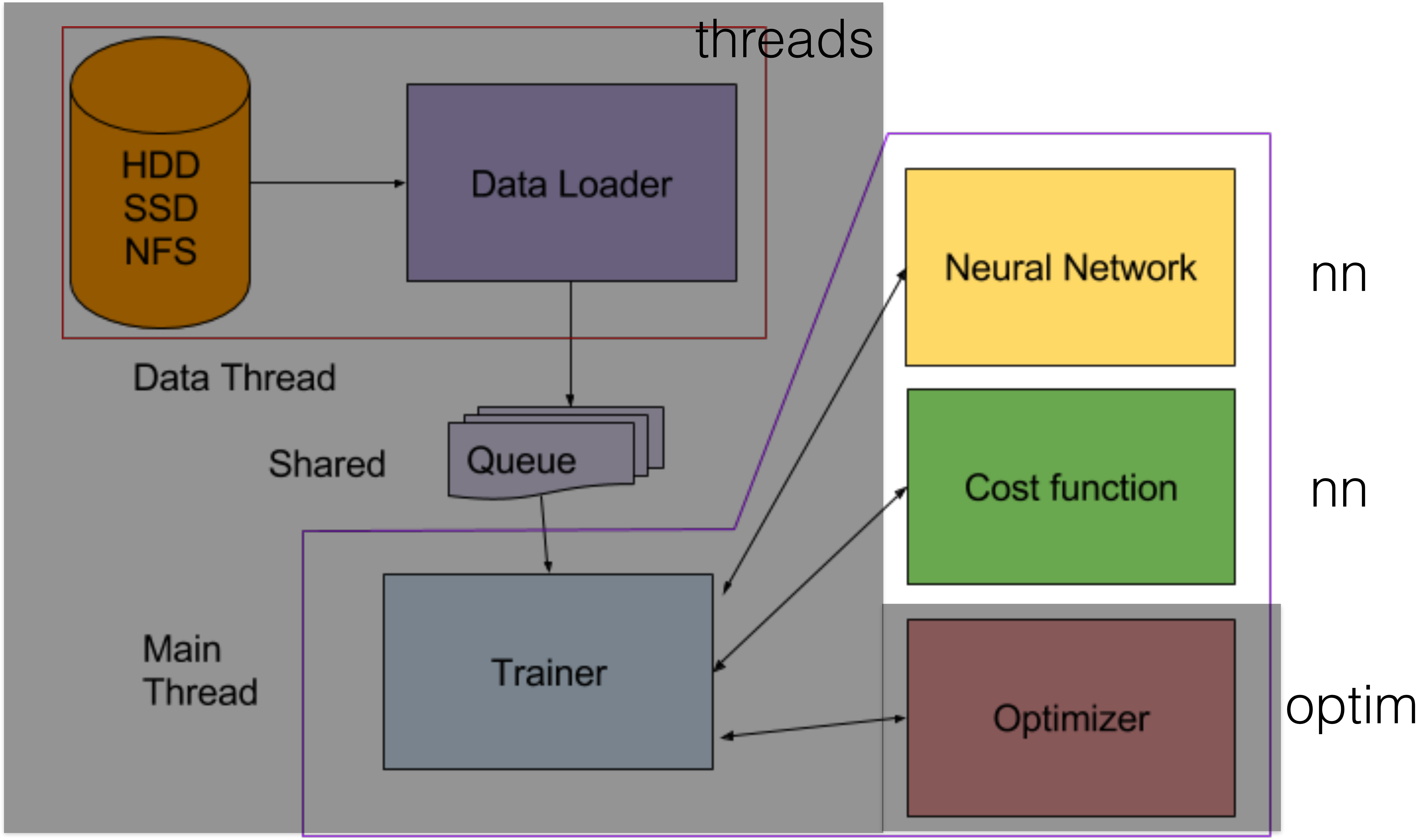
Moving parts



TRAINING CYCLE



THE **NN** PACKAGE

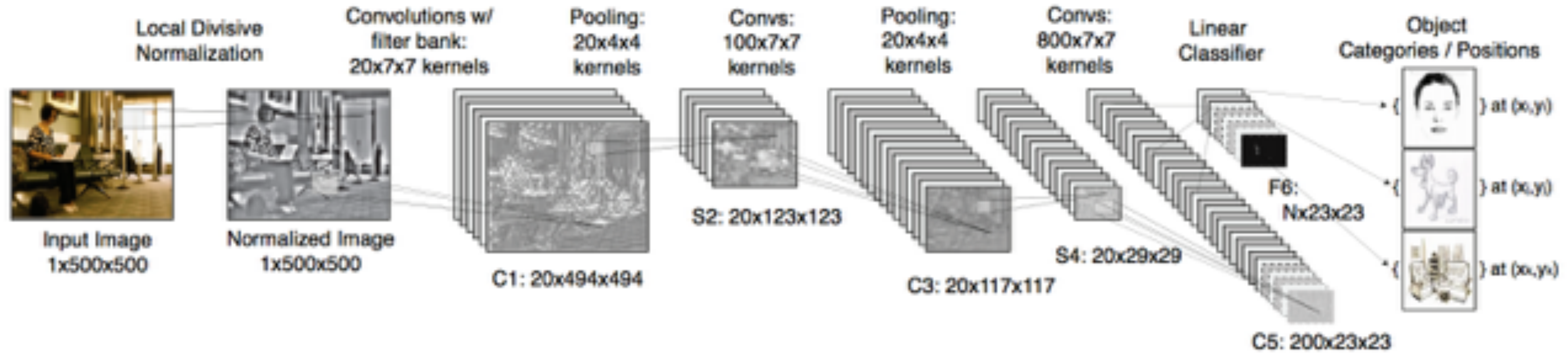


THE **NN** PACKAGE

- nn: neural networks made easy
- building blocks of differentiable modules

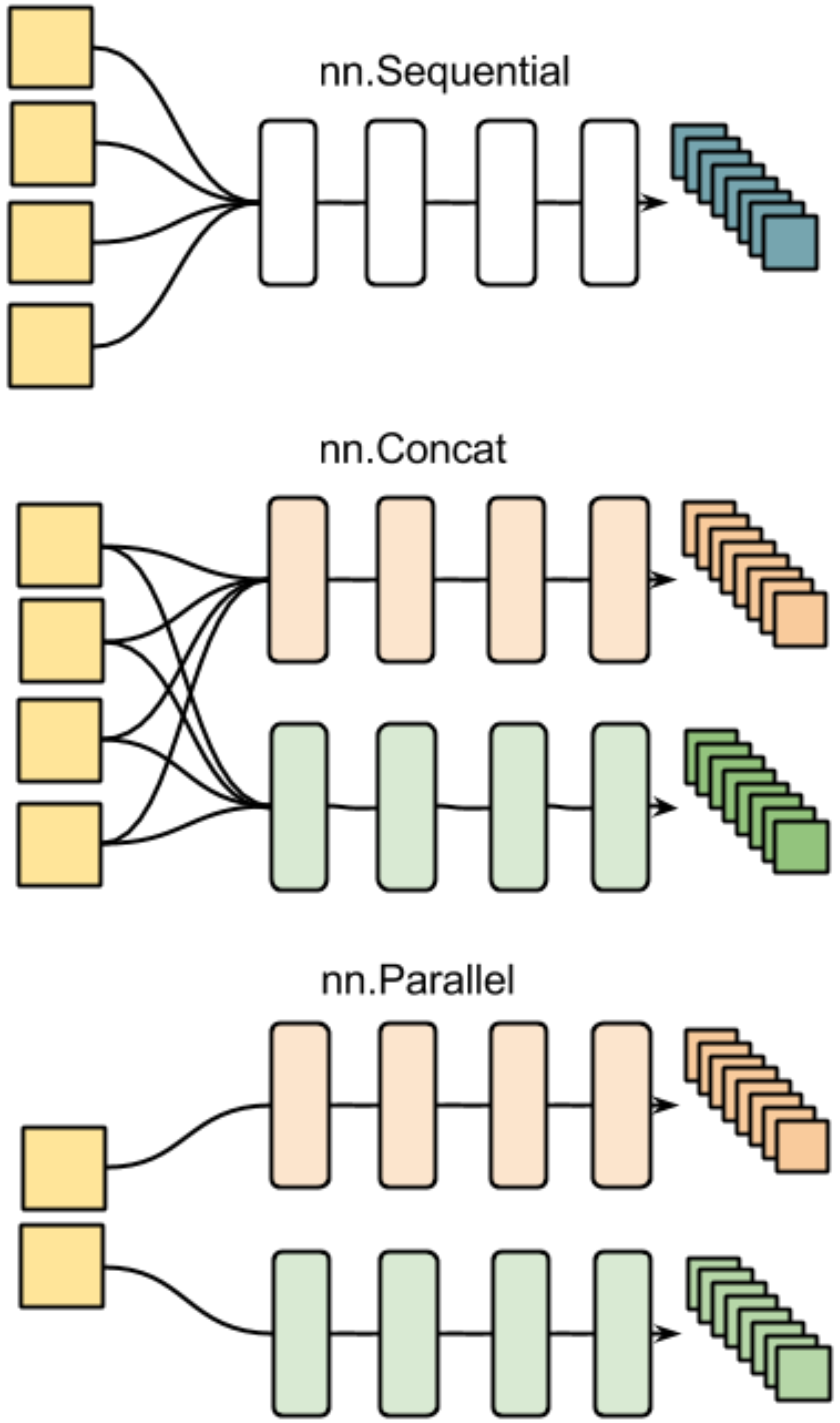
➡ **define a model with pre-normalization, to work on raw RGB images:**

```
model = nn.Sequential()

model:add( nn.SpatialConvolution(3,16,5,5) )
model:add( nn.Tanh() )
model:add( nn.SpatialMaxPooling(2,2,2,2) )
model:add( nn.SpatialContrastiveNormalization(16, image.gaussian(3)) )

model:add( nn.SpatialConvolution(16,64,5,5) )
model:add( nn.Tanh() )
model:add( nn.SpatialMaxPooling(2,2,2,2) )
model:add( nn.SpatialContrastiveNormalization(64, image.gaussian(3)) )

model:add( nn.SpatialConvolution(64,256,5,5) )
model:add( nn.Tanh() )
model:add( nn.Reshape(256) )
model:add( nn.Linear(256,10) )
model:add( nn.LogSoftMax() )
```



THE **NN** PACKAGE

Compose networks
like Lego blocks



THE **NN** PACKAGE

- When training neural nets, autoencoders, linear regression, convolutional networks, and any of these models, we're interested in gradients, and loss functions

- The **nn** package provides a large set of transfer functions, which all come with three methods:
  - `upgradeOutput()      -- compute the output given the input`
  - `upgradeGradInput()   -- compute the derivative of the loss wrt input`
  - `accGradParameters()  -- compute the  derivative of the loss wrt weights`

- The **nn** package provides a set of common loss functions, which all come with two methods:
  - `upgradeOutput()      -- compute the output given the input`
  - `upgradeGradInput()   -- compute the derivative of the loss wrt input`

THE **NN** PACKAGE

CUDA Backend via the cunn package

require 'cunn'

```
-- define model
model = nn.Sequential()
model:add( nn.Linear(100,1000) )
model:add( nn.Tanh() )
model:add( nn.Linear(1000,10) )
model:add( nn.LogSoftMax() )

-- re-cast model as a CUDA model
model:cuda()

-- define input as a CUDA Tensor
input = torch.CudaTensor(100)
-- compute model's output (is a CudaTensor as well)
output = model:forward(input)

-- alternative: convert an existing DoubleTensor to a CudaTensor:
input = torch.randn(100):cuda()
output = model:forward(input)
```

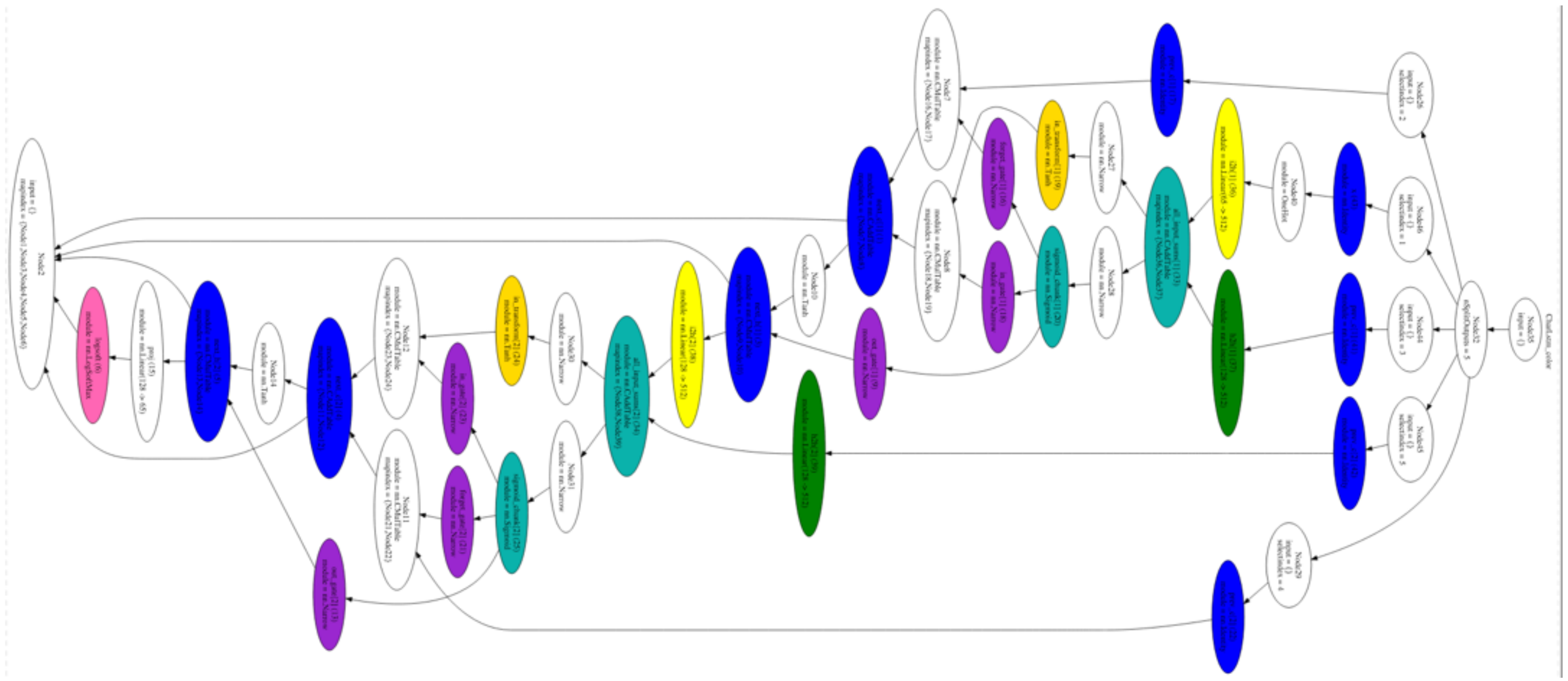
THE **NNGRAPH** PACKAGE

Graph composition using chaining

In [ ]:
```
-- it is common style to mark inputs with identity nodes for clarity.
input = nn.Identity()()

-- each hidden layer is achieved by connecting the previous one
-- here we define a single hidden layer network
h1 = nn.Tanh()(nn.Linear(20, 10)(input))
output = nn.Linear(10, 1)(h1)
mlp = nn.gModule({input}, {output})

x = torch.rand(20)
dx = torch.rand(1)
mlp:updateOutput(x)
mlp:updateGradInput(x, dx)
mlp:accGradParameters(x, dx)

-- draw graph (the forward graph, '.fg')
-- this will produce an SVG in the runtime directory
graph.dot(mlp.fg, 'MLP', 'MLP')
itorch.image('MLP.svg')
```

ADVANCED NEURAL NETWORKS

- nngraph
  - easy construction of complicated neural networks

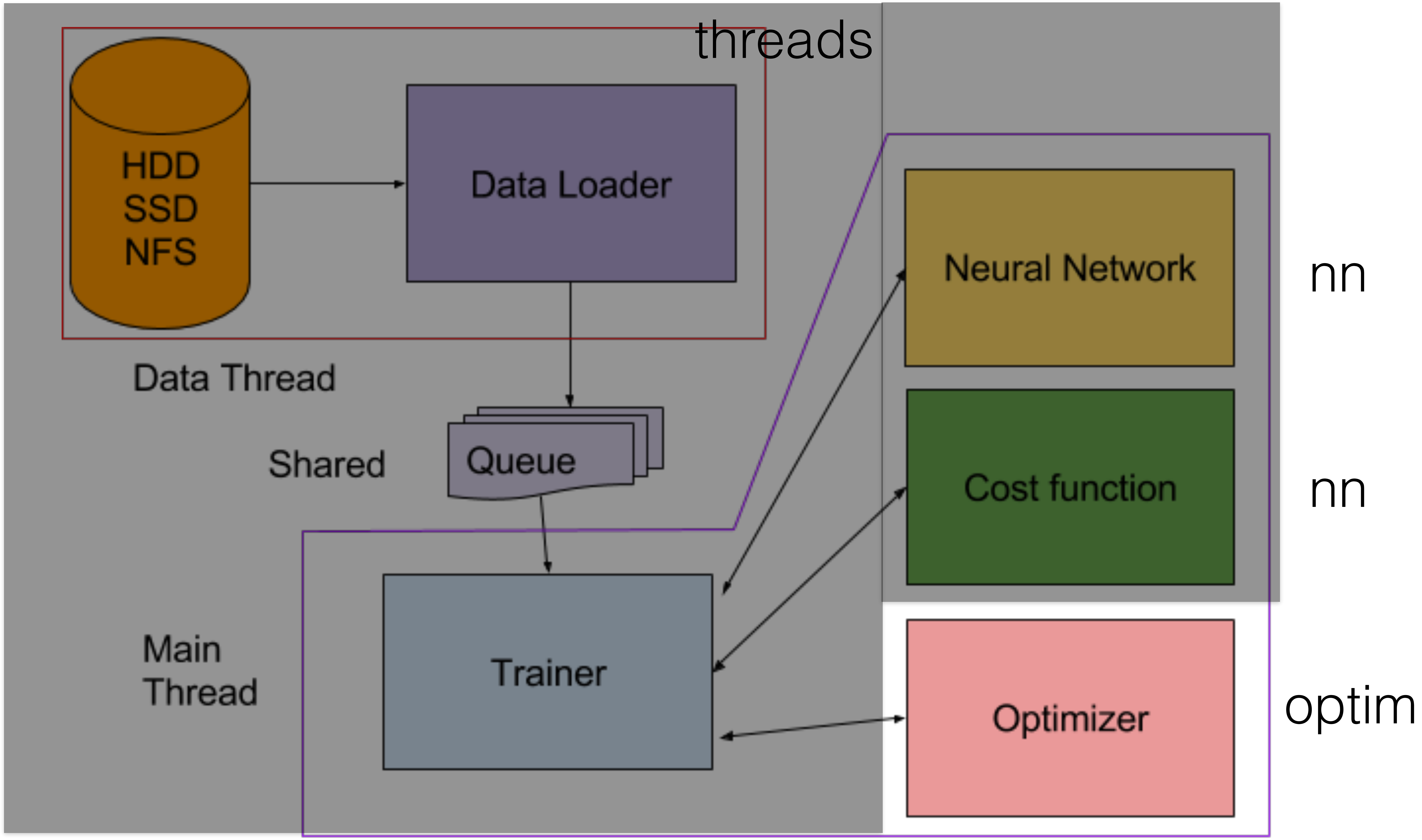


TORCH-AUTOGRAD BY

- Write imperative programs
- Backprop defined for every operation in the language

```
neuralNet = function(params, x, y)
   local h1 = t.tanh(x * params.W[1] + params.b[1])
   local h2 = t.tanh(h1 * params.W[2] + params.b[2])
   local yHat = h2 - t.log(t.sum(t.exp(h2)))
   local loss = - t.sum(t.cmul(yHat, y))
   return loss
end

-- gradients:
dneuralNet = grad(neuralNet)

-- some data:
x = t.randn(1,100)
y = t.Tensor(1,10):zero() y[1][3] = 1

-- compute loss and gradients wrt all parameters in params:
dparams, loss = dneuralNet(params, x, y)
```

THE **OPTIM** PACKAGE



THE **OPTIM** PACKAGE

- *Stochastic Gradient Descent*
- *Averaged Stochastic Gradient Descent*
- *L-BFGS*
- *Congugate Gradients*
- *AdaDelta*
- *AdaGrad*
- *Adam*
- *AdaMax*
- *FISTA with backtracking line search*
- *Nesterov's Accelerated Gradient method*
- *RMSprop*
- *Rprop*
- *CMAES*

THE **OPTIM** PACKAGE

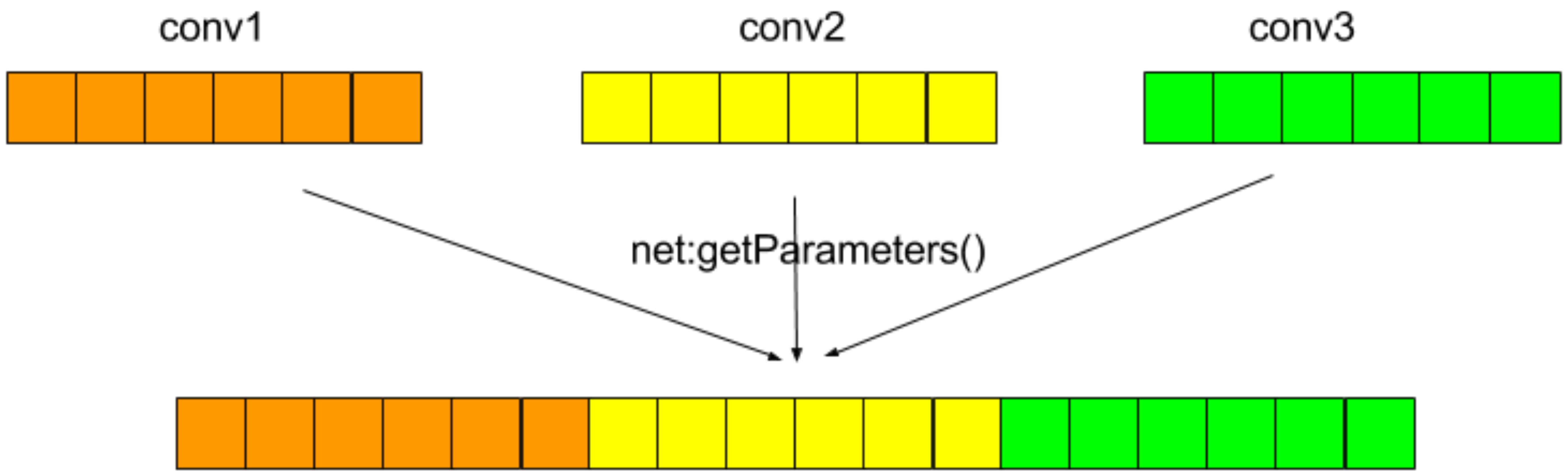
A purely functional view of the world

```lua
config = {
   learningRate = 1e-3,
   momentum = 0.5
}

for i, sample in ipairs(training_samples) do
    local func = function(x)
       -- define eval function
       return f, df_dx
    end
    optim.sgd(func, x, config)
end
```

THE **OPTIM** PACKAGE

Collecting the parameters of your neural net

- Substitute each module weights and biases by one large tensor, making weights and biases point to parts of this tensor



TORCH AUTOGRAD

Industrial-strength, extremely flexible implementation of **automatic differentiation,**
for all your crazy ideas

TORCH AUTOGRAD

Industrial-strength, extremely flexible implementation of **automatic differentiation,**
for all your crazy ideas

Inspired by the original Python autograd from Ryan Adams' HIPS group: github.com/hips/autograd

Props to:

— Dougal Maclaurin

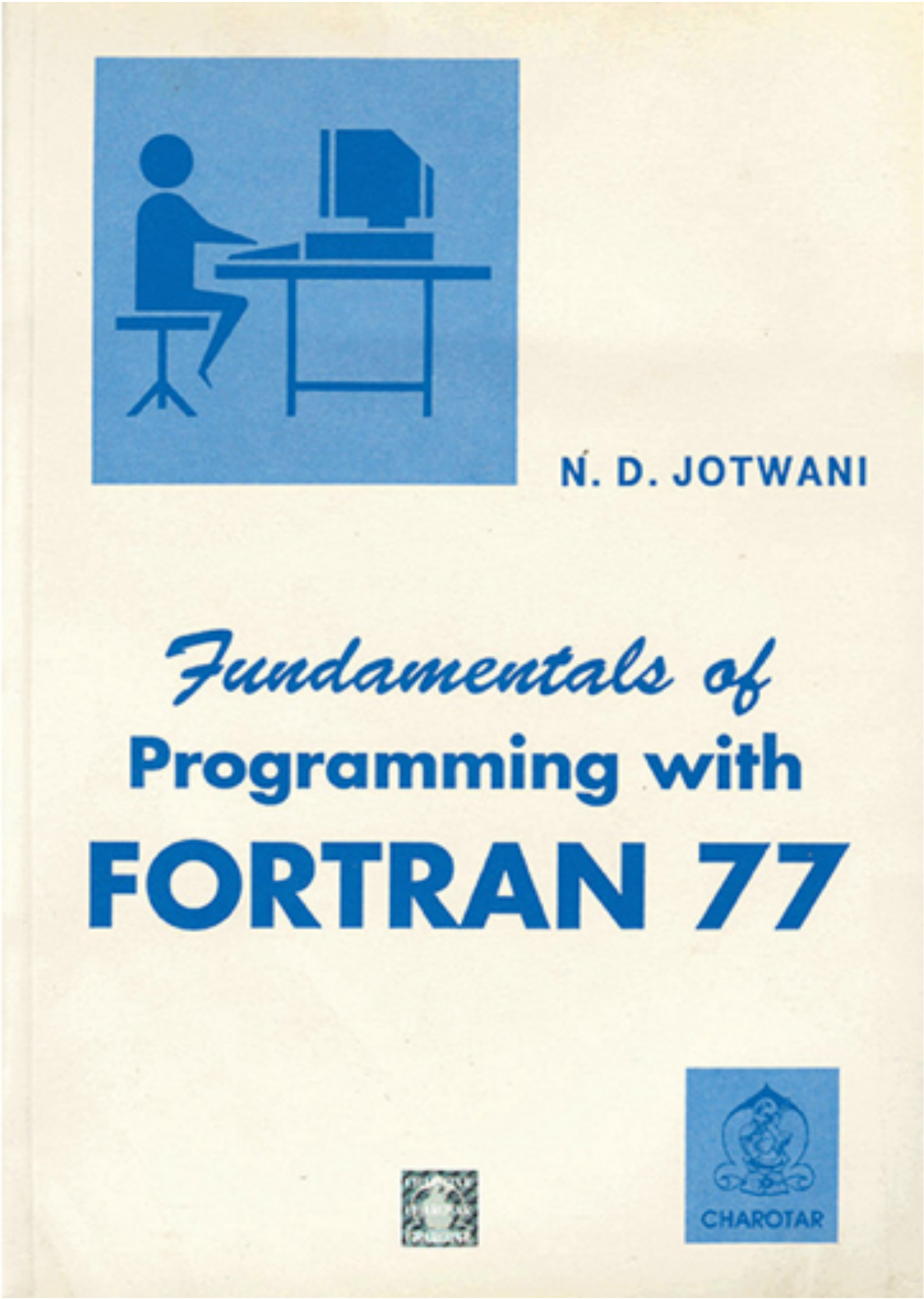
— David Duvenaud

— Matt Johnson

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)

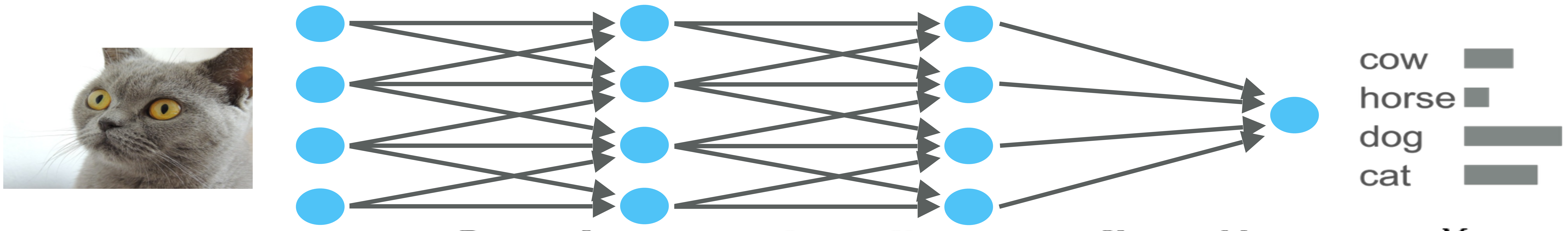
- Rediscovered several times (Widrow and Lehr, 1990)
- Described and implemented for FORTRAN by Speelpenning in 1980 (although forward-mode variant that is less useful for ML described in 1964 by Wengert).
- Popularized in connectionist ML as "backpropagation" (Rumelhart et al, 1986)
- In use in nuclear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

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

- Two main modes:
  - Forward mode
  - Reverse mode (backprop)

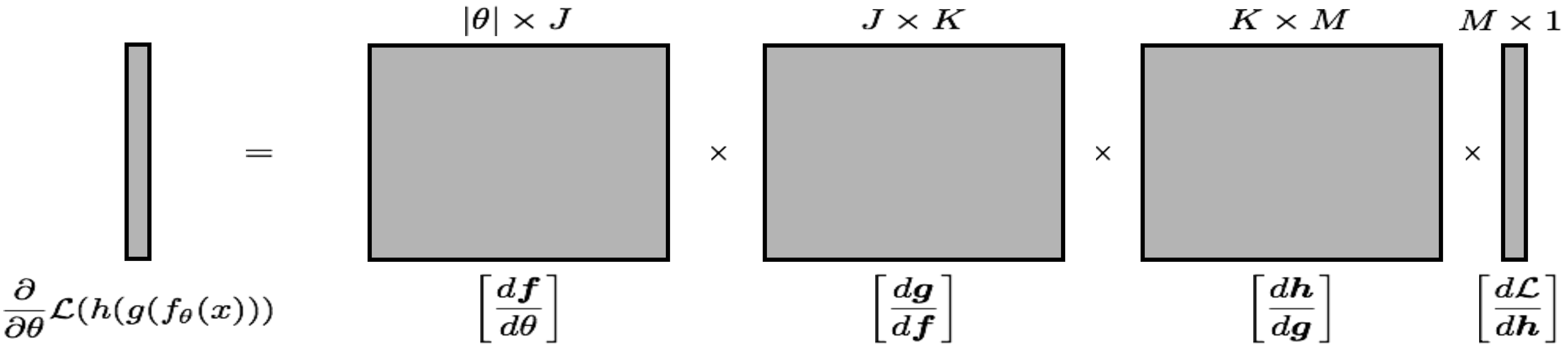
Different applications of the chain rule

FORWARD MODE (SYMBOLIC VIEW)

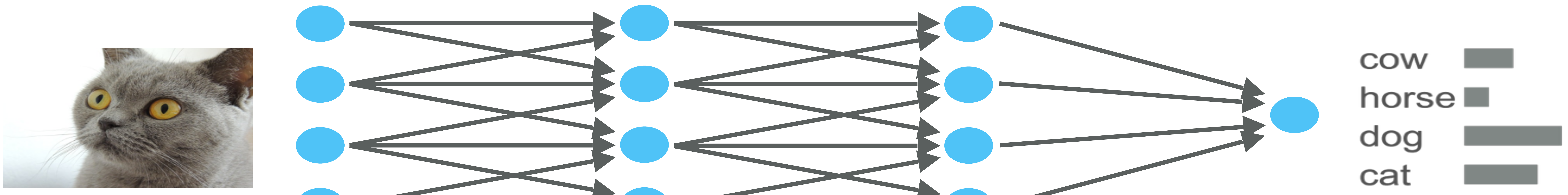


$f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^J$ $\quad g : \mathbb{R}^J \rightarrow \mathbb{R}^K$ $\quad h : \mathbb{R}^K \rightarrow \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \rightarrow \mathbb{R}$

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

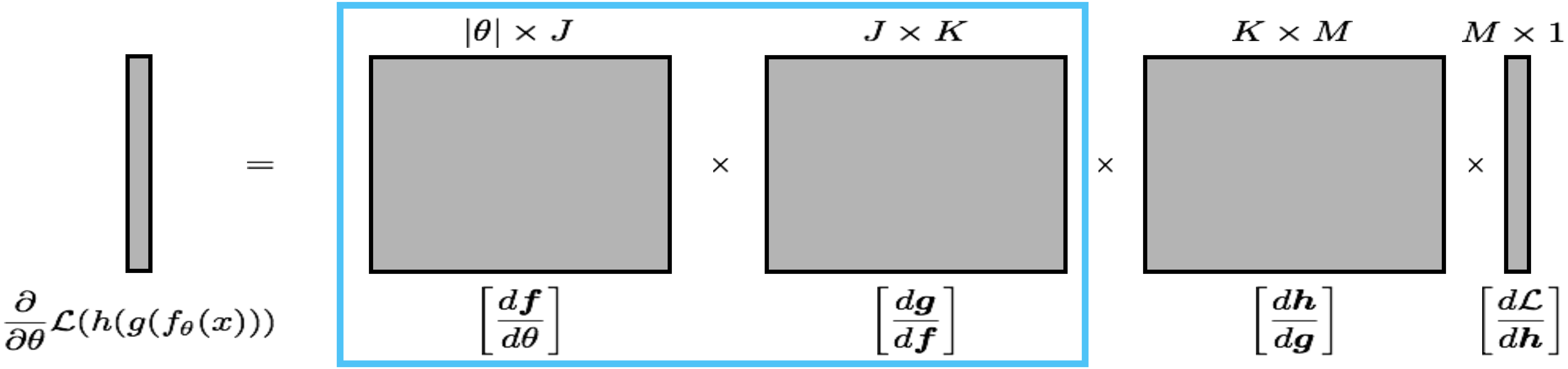


FORWARD MODE (SYMBOLIC VIEW)

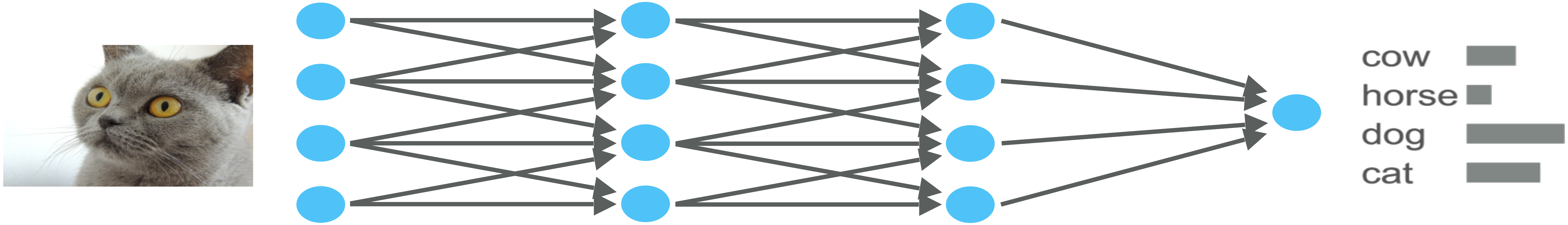


$f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^J$ $\quad g : \mathbb{R}^J \rightarrow \mathbb{R}^K$ $\quad h : \mathbb{R}^K \rightarrow \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \rightarrow \mathbb{R}$

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

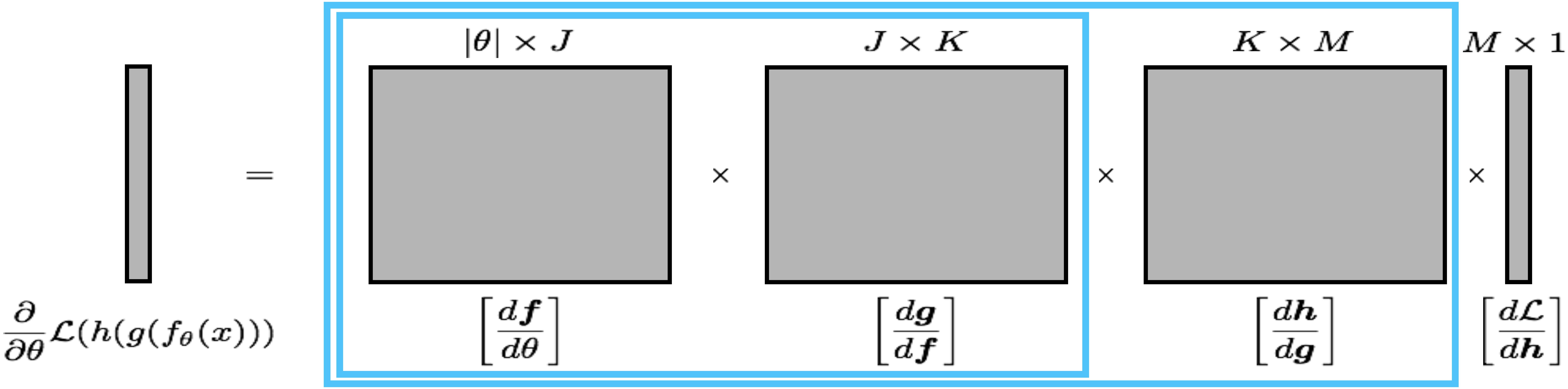


FORWARD MODE (SYMBOLIC VIEW)

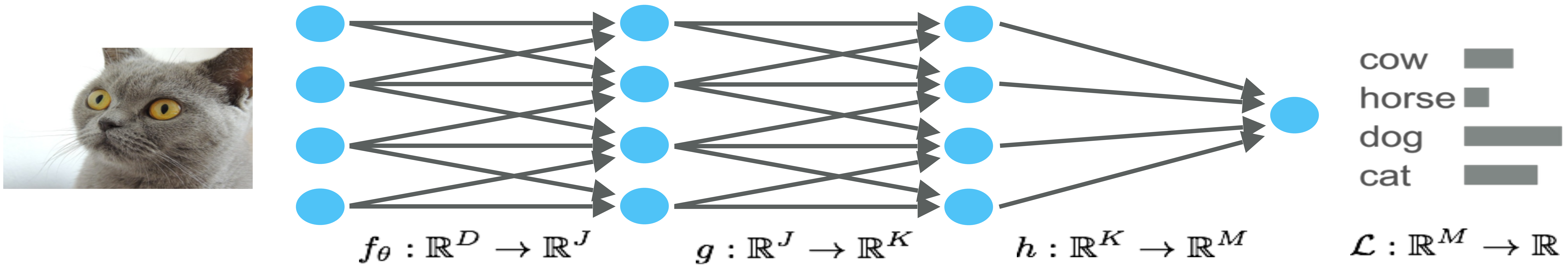


$f_\theta : \mathbb{R}^D \to \mathbb{R}^J$ $\quad g : \mathbb{R}^J \to \mathbb{R}^K$ $\quad h : \mathbb{R}^K \to \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

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

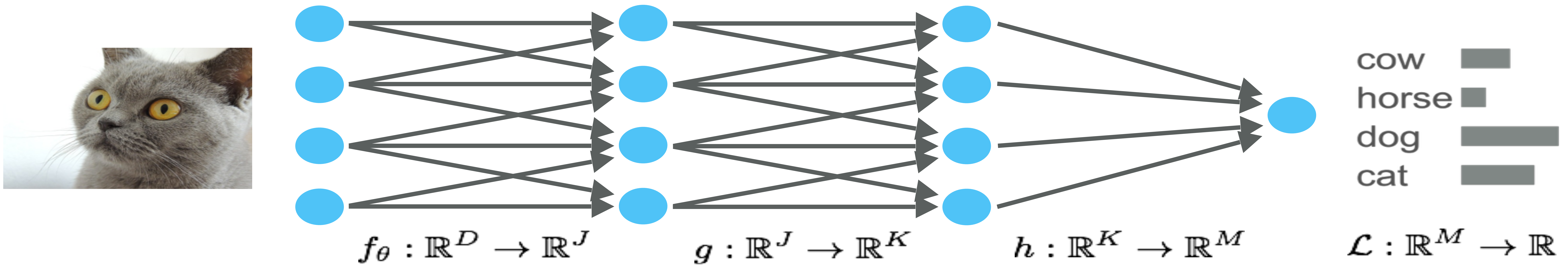


FORWARD MODE (SYMBOLIC VIEW)



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

FORWARD MODE (SYMBOLIC VIEW)



Left to right:

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

$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x)))) = \underset{|\theta| \times J}{\left[\frac{d\boldsymbol{f}}{d\theta}\right]} \times \underset{J \times K}{\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]} \times \underset{K \times M}{\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]} \times \underset{M \times 1}{\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]}$$

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

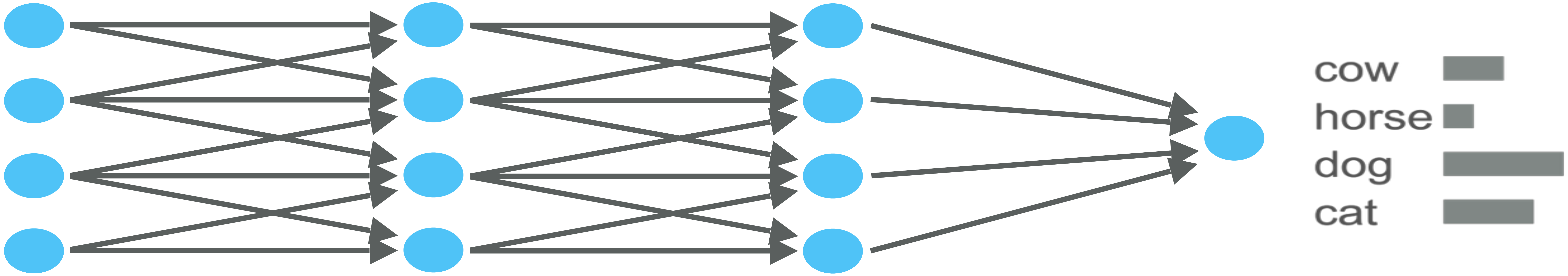
d = a * math.sin(b) = 2.728

ddda = math.sin(b) = 0.909

return 0.909

From Baydin 2016

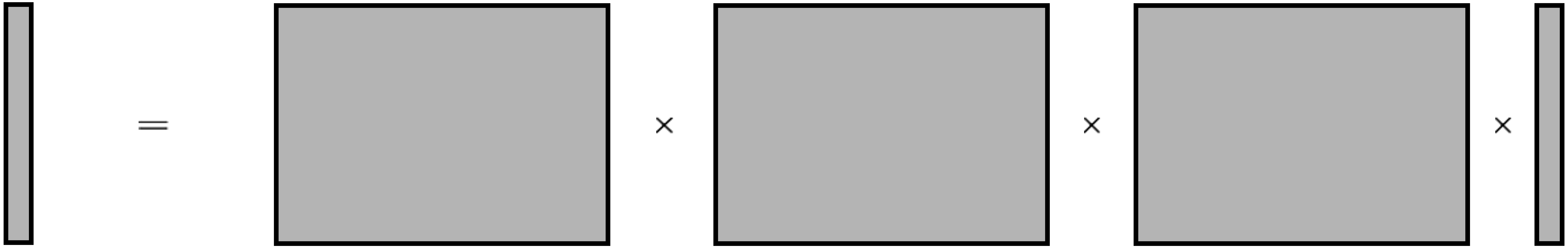
REVERSE MODE (SYMBOLIC VIEW)



$f_\theta : \mathbb{R}^D \to \mathbb{R}^J$ $\quad g : \mathbb{R}^J \to \mathbb{R}^K$ $\quad h : \mathbb{R}^K \to \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

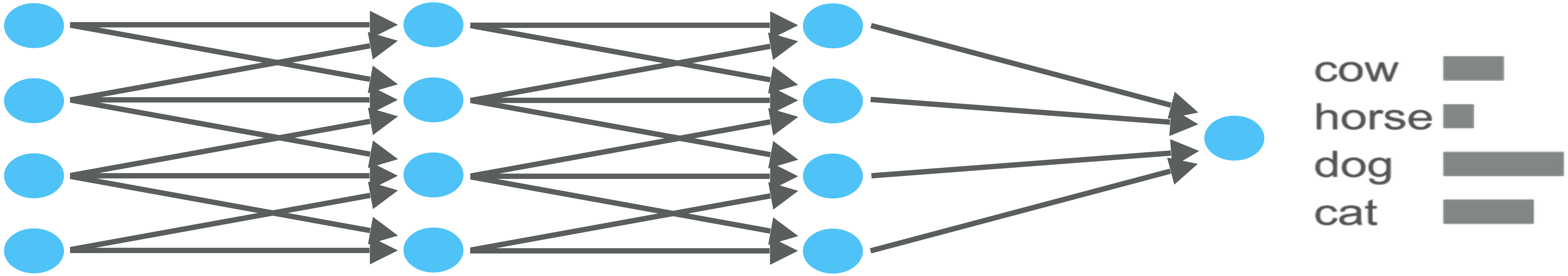
$$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x)))) = \left[\frac{d\boldsymbol{f}}{d\theta}\right]\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$$

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



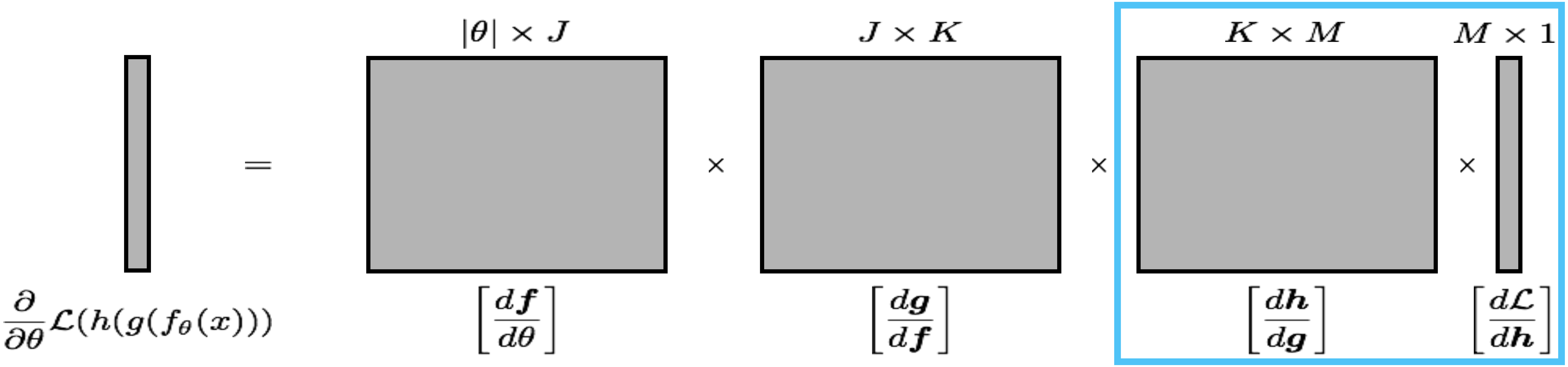
$\frac{\partial}{\partial \theta}\mathcal{L}(h(g(f_\theta(x))))$ $\quad = \quad$ $\left[\frac{d\boldsymbol{f}}{d\theta}\right]$ $\times$ $\left[\frac{d\boldsymbol{g}}{d\boldsymbol{f}}\right]$ $\times$ $\left[\frac{d\boldsymbol{h}}{d\boldsymbol{g}}\right]$ $\times$ $\left[\frac{d\mathcal{L}}{d\boldsymbol{h}}\right]$

REVERSE MODE (SYMBOLIC VIEW)

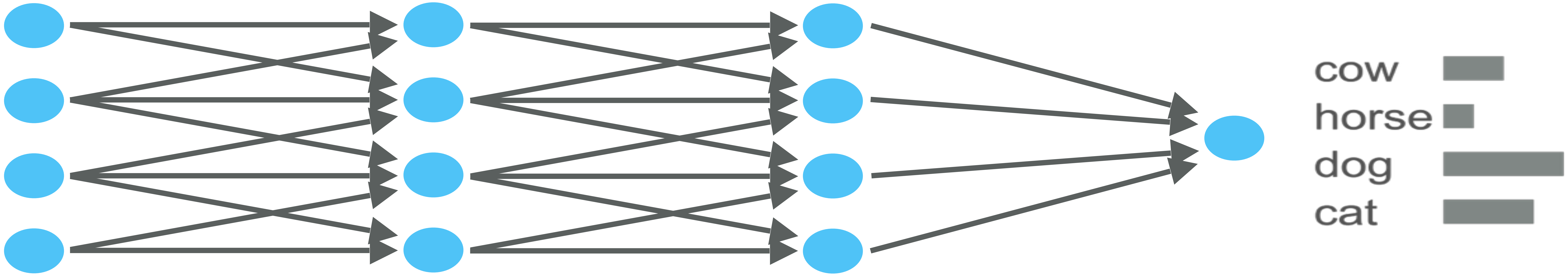


$f_\theta : \mathbb{R}^D \to \mathbb{R}^J$ $\quad g : \mathbb{R}^J \to \mathbb{R}^K$ $\quad h : \mathbb{R}^K \to \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

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



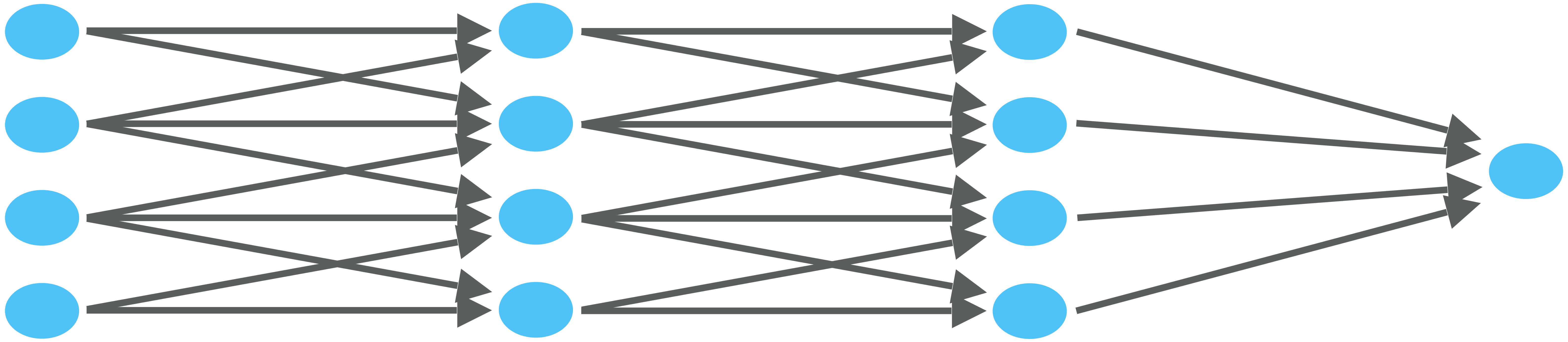
REVERSE MODE (SYMBOLIC VIEW)



$f_\theta : \mathbb{R}^D \to \mathbb{R}^J$ $\quad g : \mathbb{R}^J \to \mathbb{R}^K$ $\quad h : \mathbb{R}^K \to \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

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

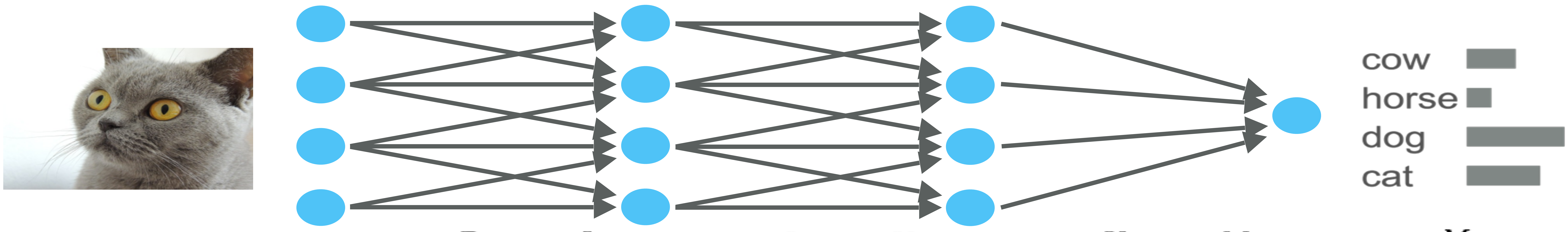
REVERSE MODE (SYMBOLIC VIEW)



$f_\theta : \mathbb{R}^D \to \mathbb{R}^J$ $\quad g : \mathbb{R}^J \to \mathbb{R}^K$ $\quad h : \mathbb{R}^K \to \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

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

REVERSE MODE (SYMBOLIC VIEW)



$f_\theta : \mathbb{R}^D \to \mathbb{R}^J$ $\quad g : \mathbb{R}^J \to \mathbb{R}^K$ $\quad h : \mathbb{R}^K \to \mathbb{R}^M$ $\quad \mathcal{L} : \mathbb{R}^M \to \mathbb{R}$

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

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 ->

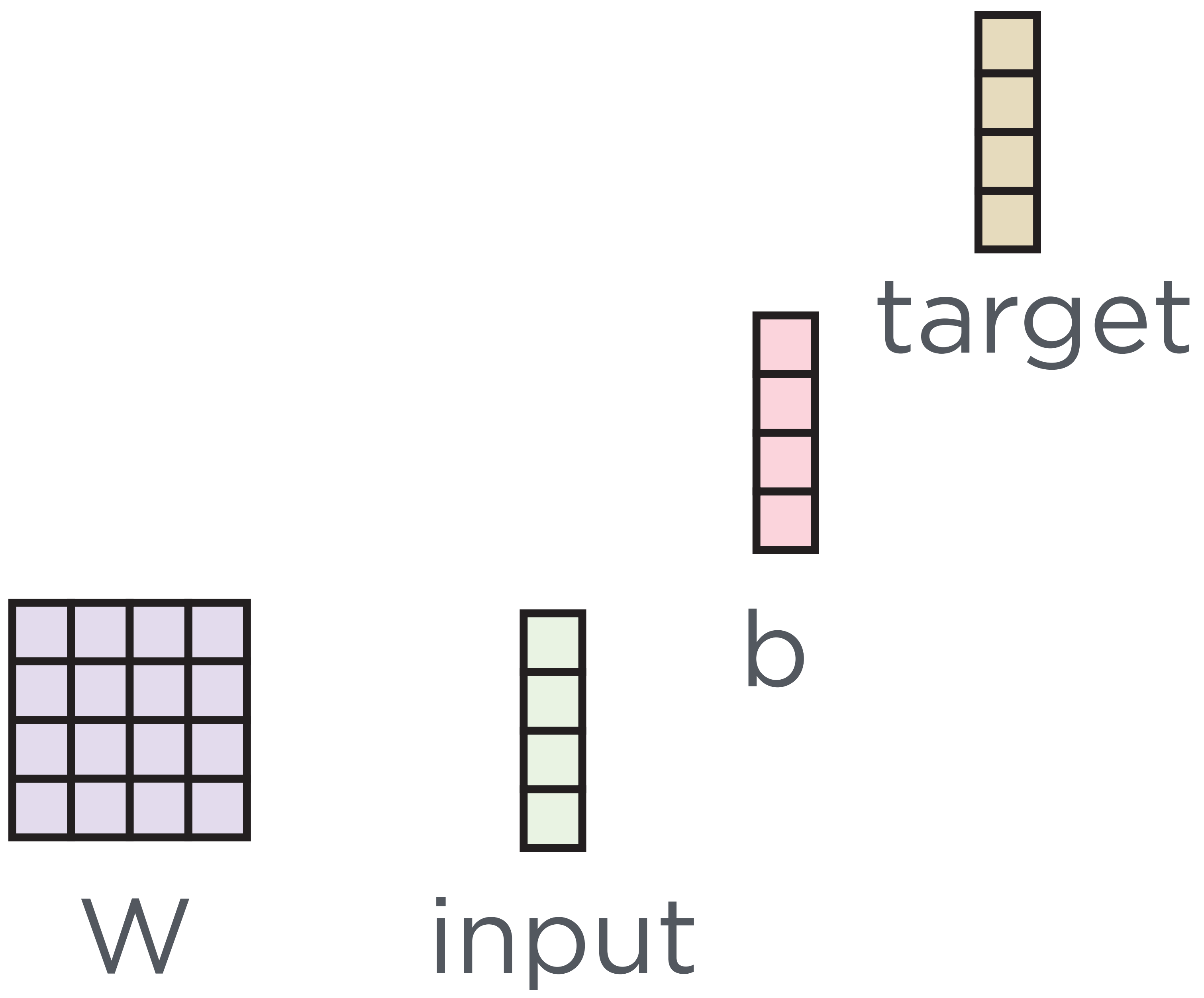
This is a how the parameters are updated

```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

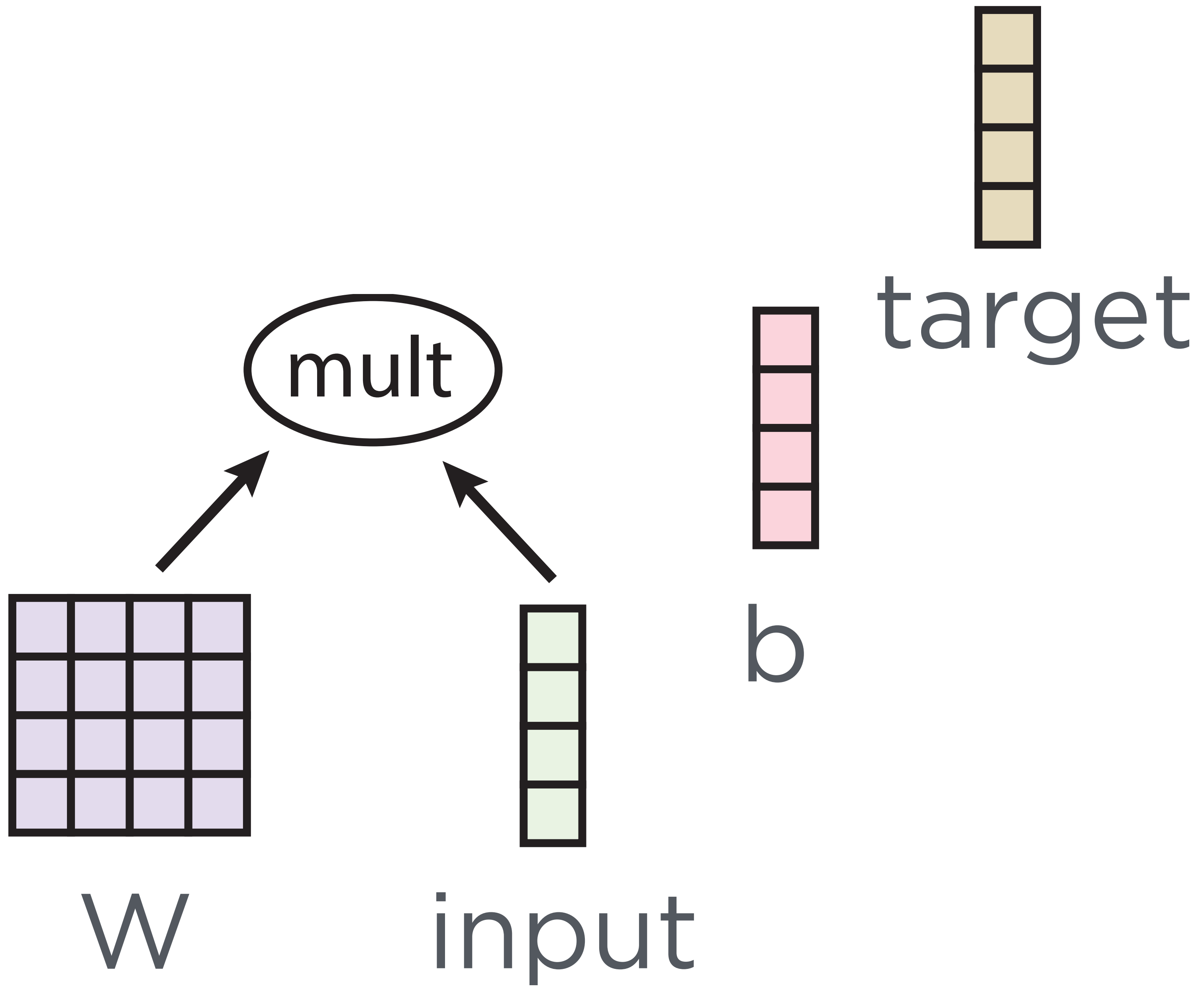
```
params = {W=torch.randn(4,4),b=torch.randn(4)}
input = torch.randn(4)
target = torch.randn(4)
function simpleFn(params, input, target)
   local h1 = params.W*input
   local h2 = h1 + params.b
   local h3 = h2 - target
   local h4 = torch.pow(h3,2)
   local h5 = torch.sum(h4)
   return h5
end
```



WHAT'S ACTUALLY HAPPENING?

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

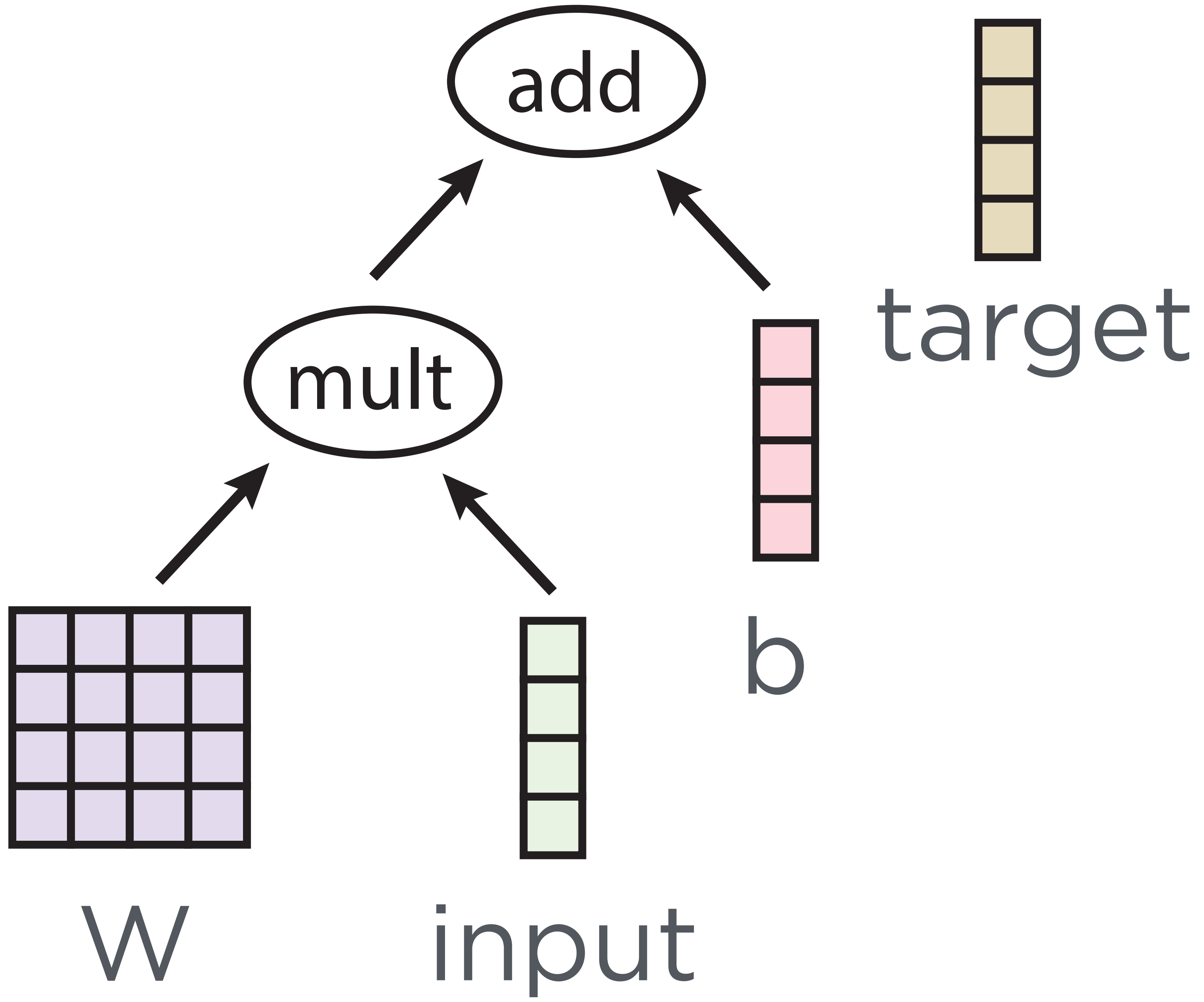
```
params = {W=torch.randn(4,4),b=torch.randn(4)}
input = torch.randn(4)
target = torch.randn(4)
function simpleFn(params, input, target)
    local h1 = params.W*input
    local h2 = h1 + params.b
    local h3 = h2 - target
    local h4 = torch.pow(h3,2)
    local h5 = torch.sum(h4)
    return h5
end
```



WHAT'S ACTUALLY HAPPENING?

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

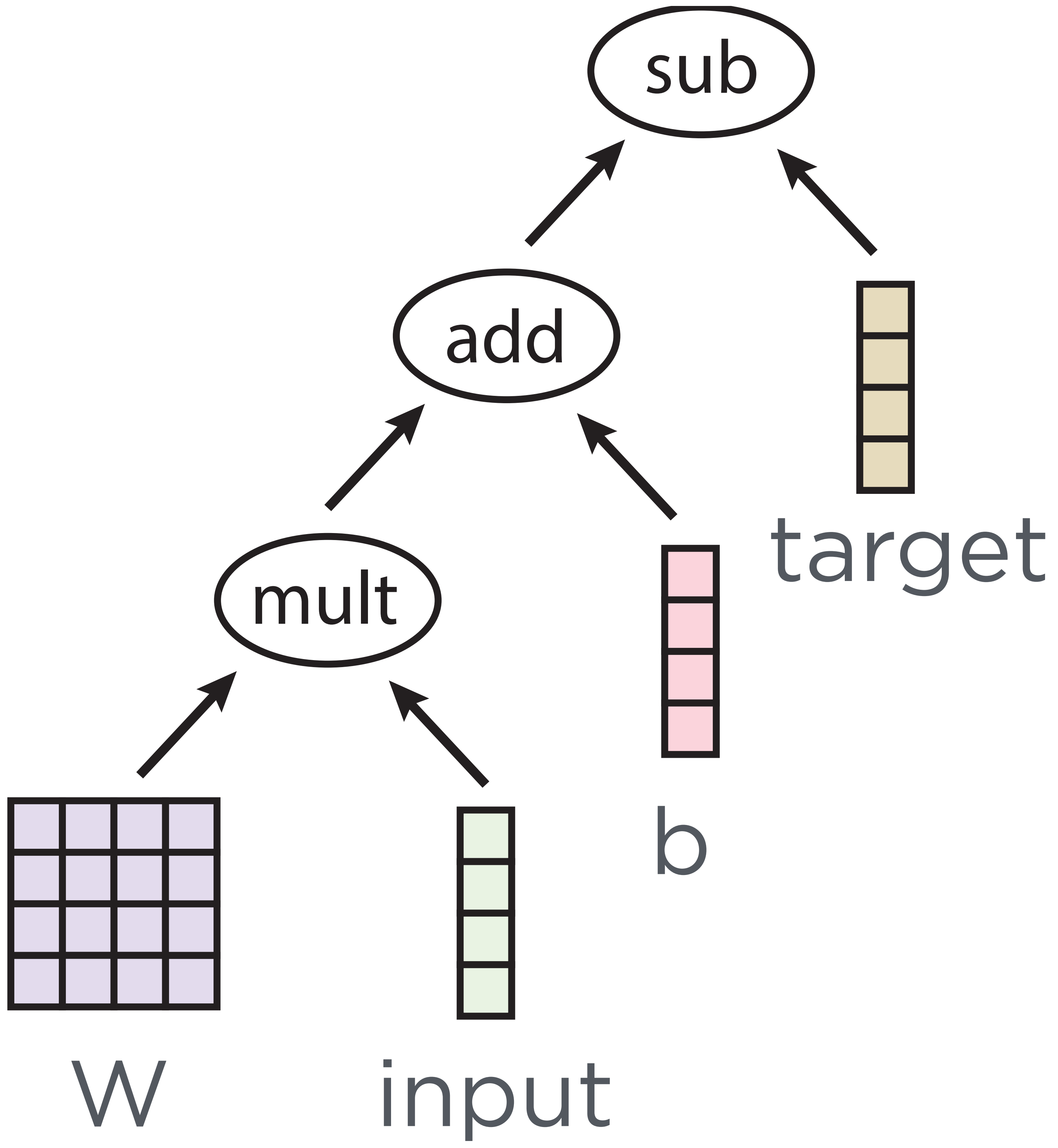
```
params = {W=torch.randn(4,4),b=torch.randn(4)}
input = torch.randn(4)
target = torch.randn(4)
function simpleFn(params, input, target)
    local h1 = params.W*input
    local h2 = h1 + params.b
    local h3 = h2 - target
    local h4 = torch.pow(h3,2)
    local h5 = torch.sum(h4)
    return h5
end
```



WHAT'S ACTUALLY HAPPENING?

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

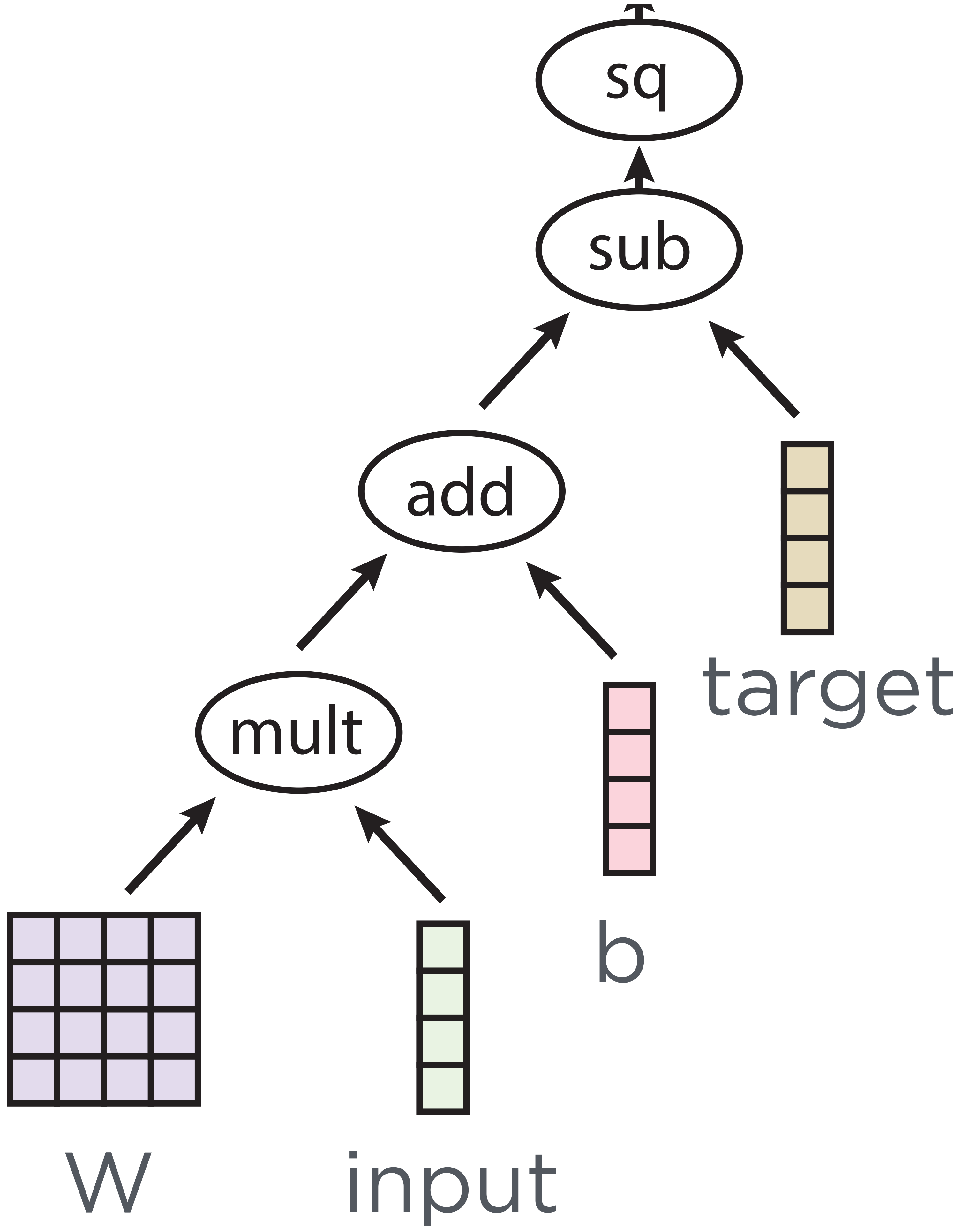
```
params = {W=torch.randn(4,4),b=torch.randn(4)}
input = torch.randn(4)
target = torch.randn(4)
function simpleFn(params, input, target)
    local h1 = params.W*input
    local h2 = h1 + params.b
    local h3 = h2 - target
    local h4 = torch.pow(h3,2)
    local h5 = torch.sum(h4)
    return h5
end
```



WHAT'S ACTUALLY HAPPENING?

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

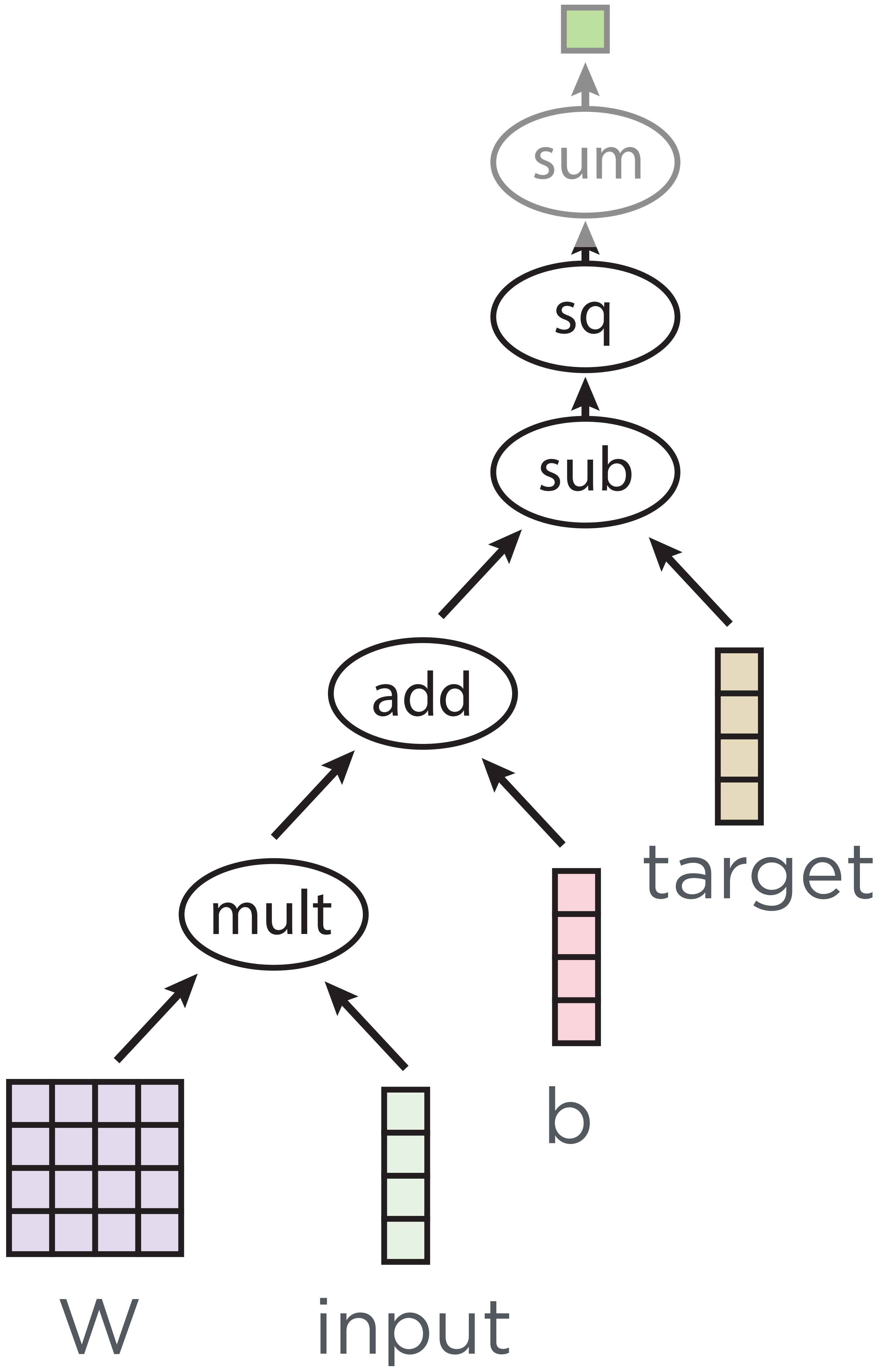
```lua
params = {W=torch.randn(4,4),b=torch.randn(4)}
input = torch.randn(4)
target = torch.randn(4)
function simpleFn(params, input, target)
    local h1 = params.W*input
    local h2 = h1 + params.b
    local h3 = h2 - target
    local h4 = torch.pow(h3,2)
    local h5 = torch.sum(h4)
    return h5
end
```



WHAT'S ACTUALLY HAPPENING?

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

```lua
params = {W=torch.randn(4,4),b=torch.randn(4)}
input = torch.randn(4)
target = torch.randn(4)
function simpleFn(params, input, target)
    local h1 = params.W*input
    local h2 = h1 + params.b
    local h3 = h2 - target
    local h4 = torch.pow(h3,2)
    local h5 = torch.sum(h4)
    return h5
end
```



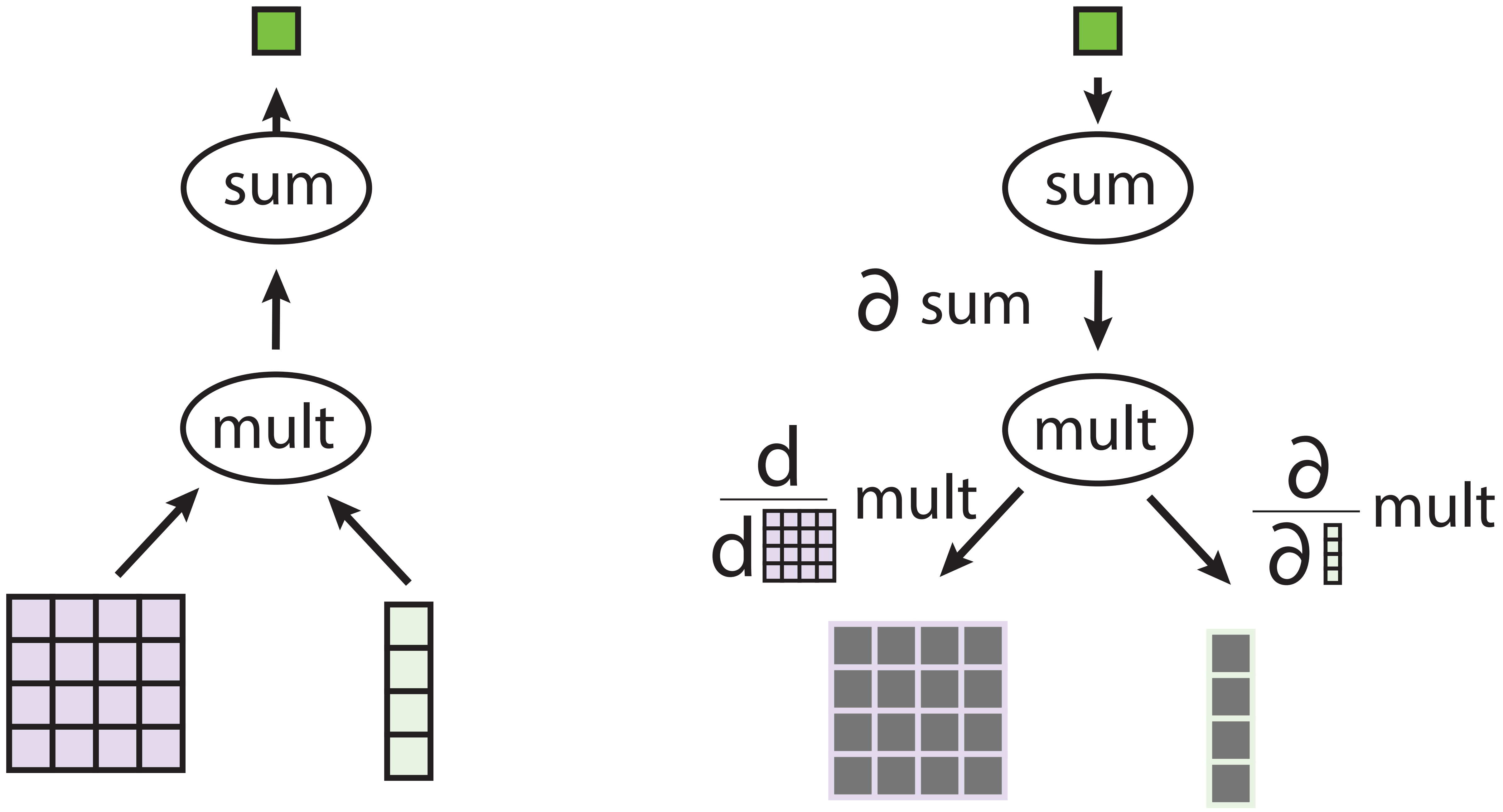
WE TRACK COMPUTATION VIA OPERATOR OVERLOADING

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, ...
```

CALCULATING THE GRADIENT

When it comes time to evaluate partial derivatives, we just have to look up the partial derivatives from a table in reverse order on the tape

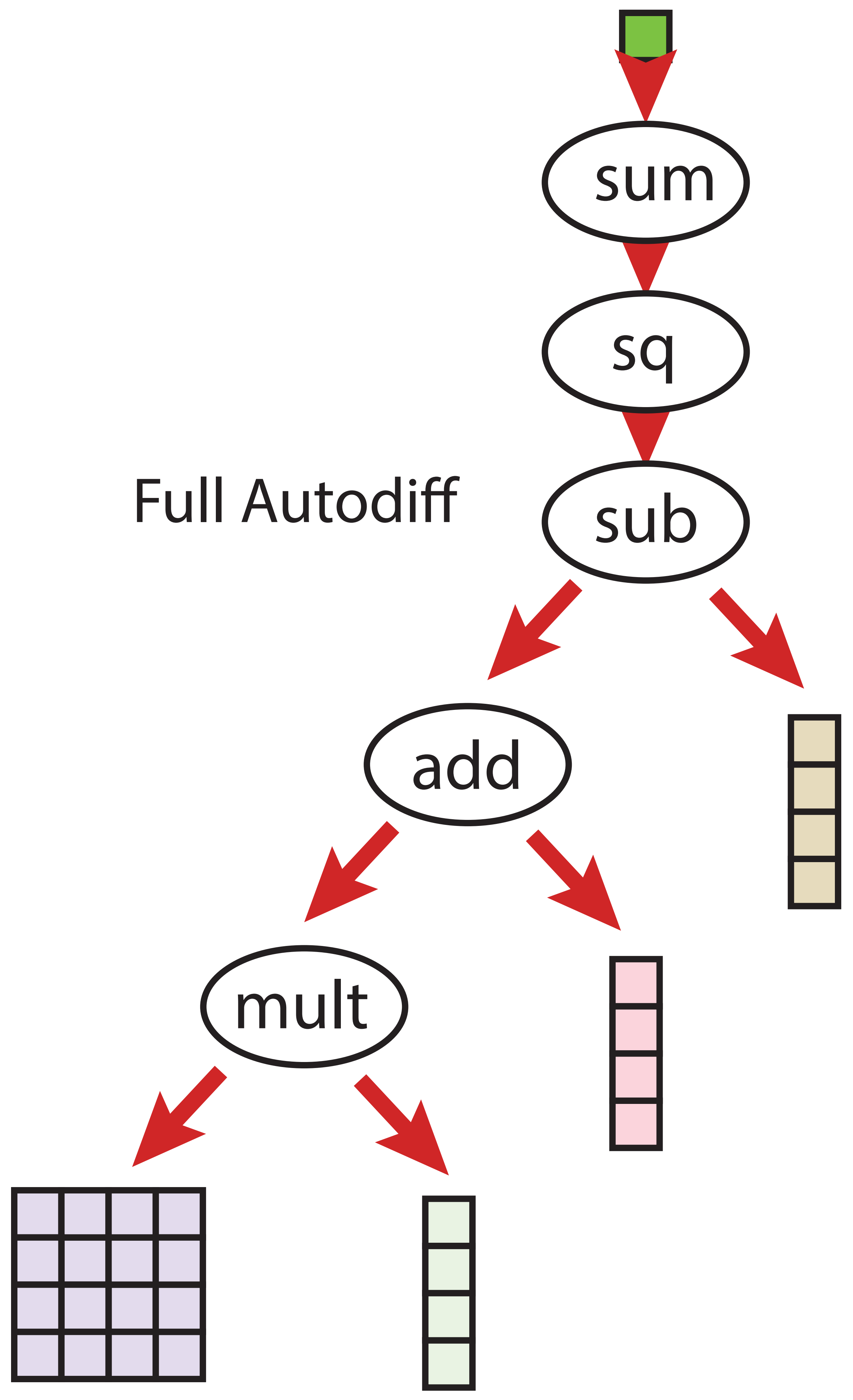


WHAT'S ACTUALLY HAPPENING?

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
})
```

Full Autodiff



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

```lua
-- 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

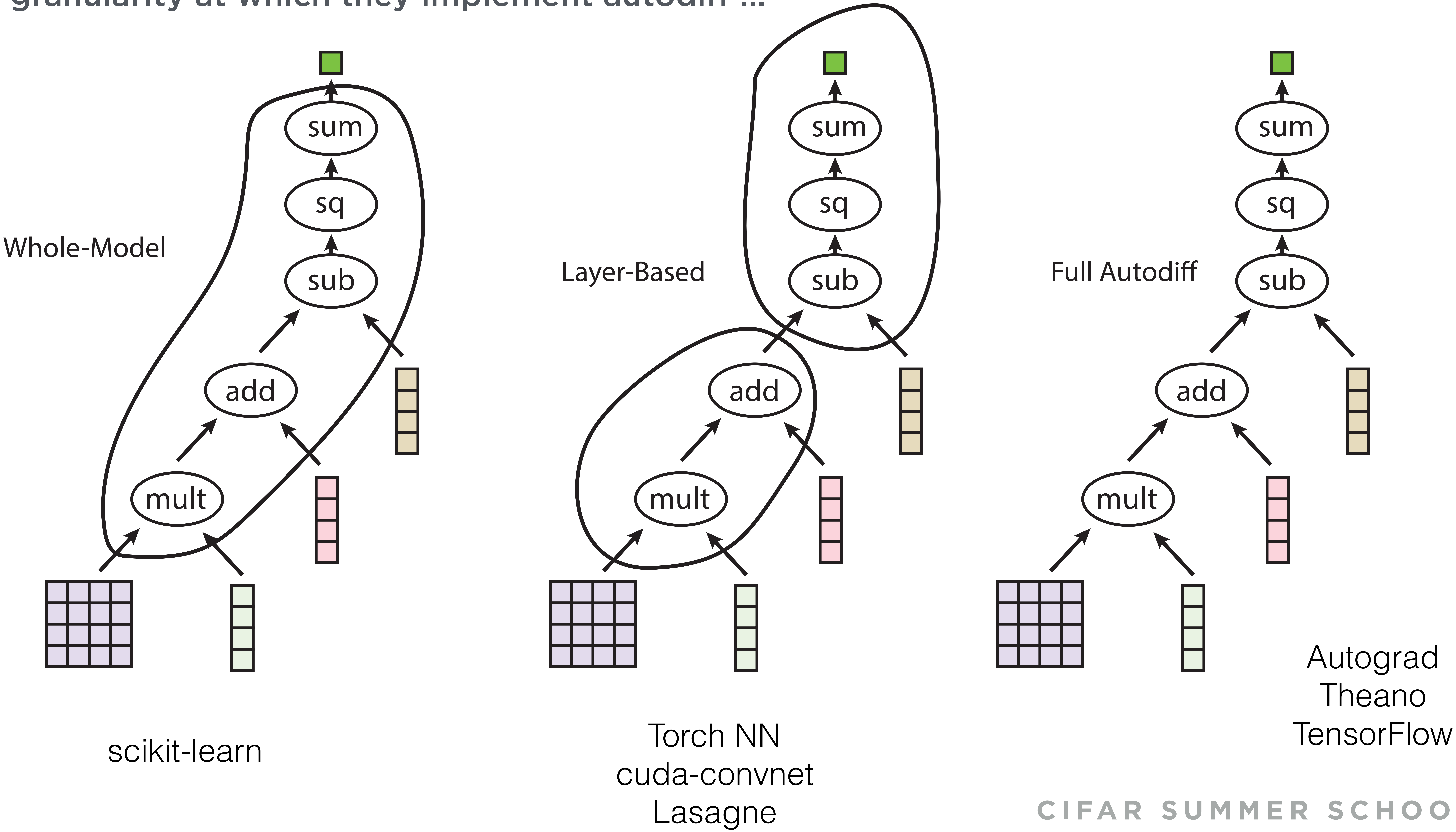
Need new or tweaked partial derivatives? Not a problem.

```
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]
```

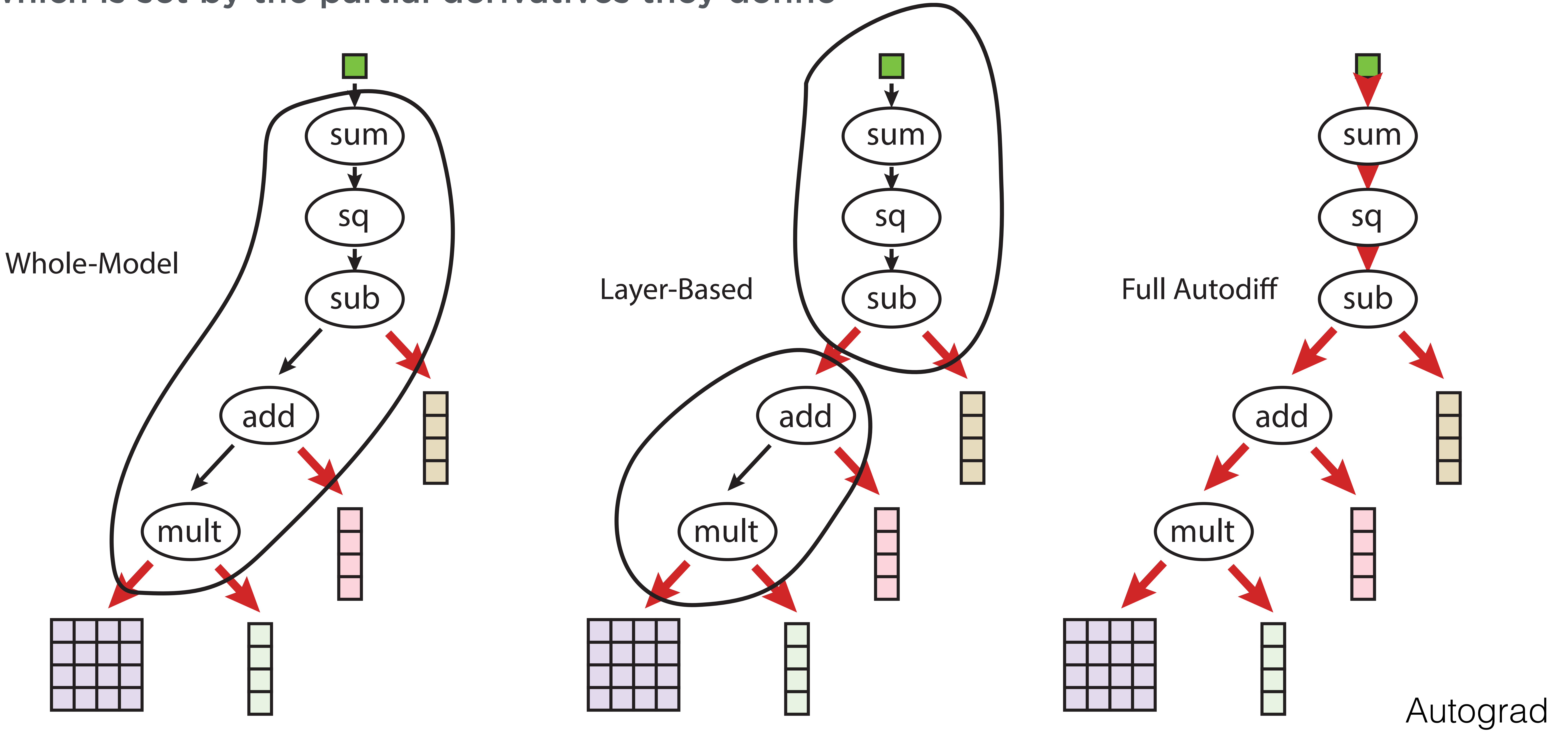
SO WHAT DIFFERENTIATES N.NET LIBRARIES?

The granularity at which they implement autodiff …



SO WHAT DIFFERENTIATES N.NET LIBRARIES?

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

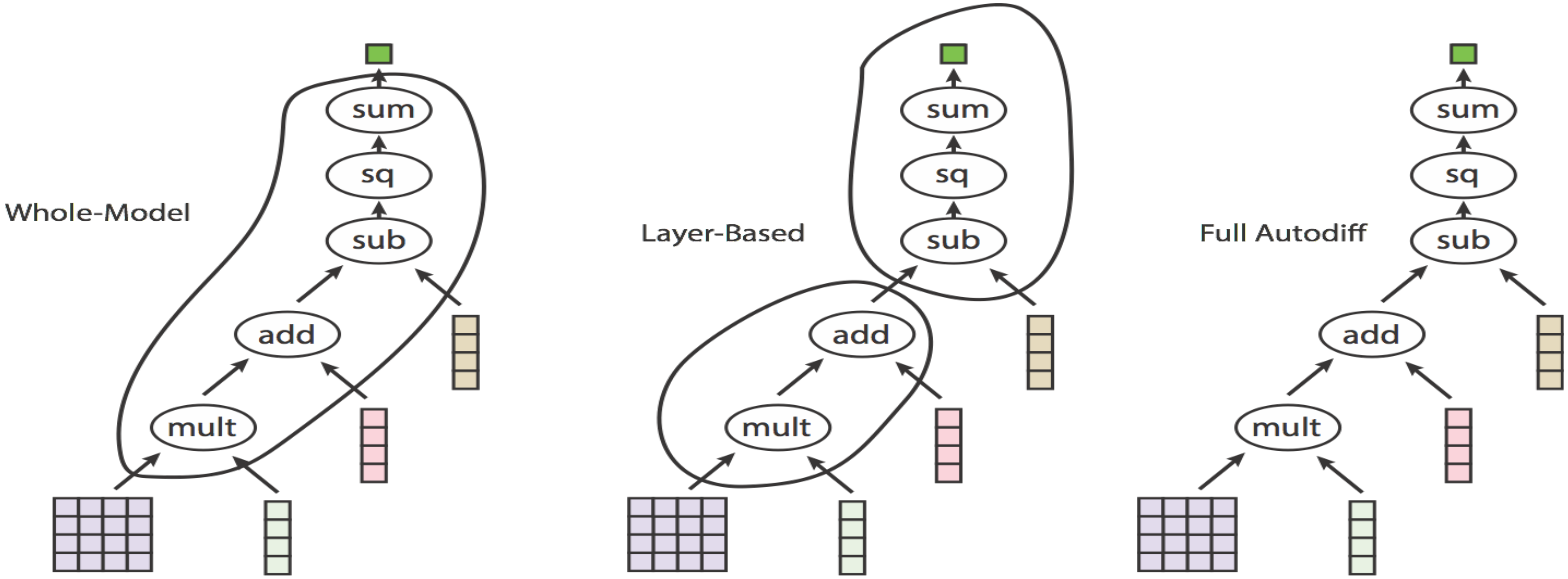


scikit-learn

Torch NN
cuda-convnet
Lasagne

Autograd
Theano
TensorFlow

We want no limits on the models we can write



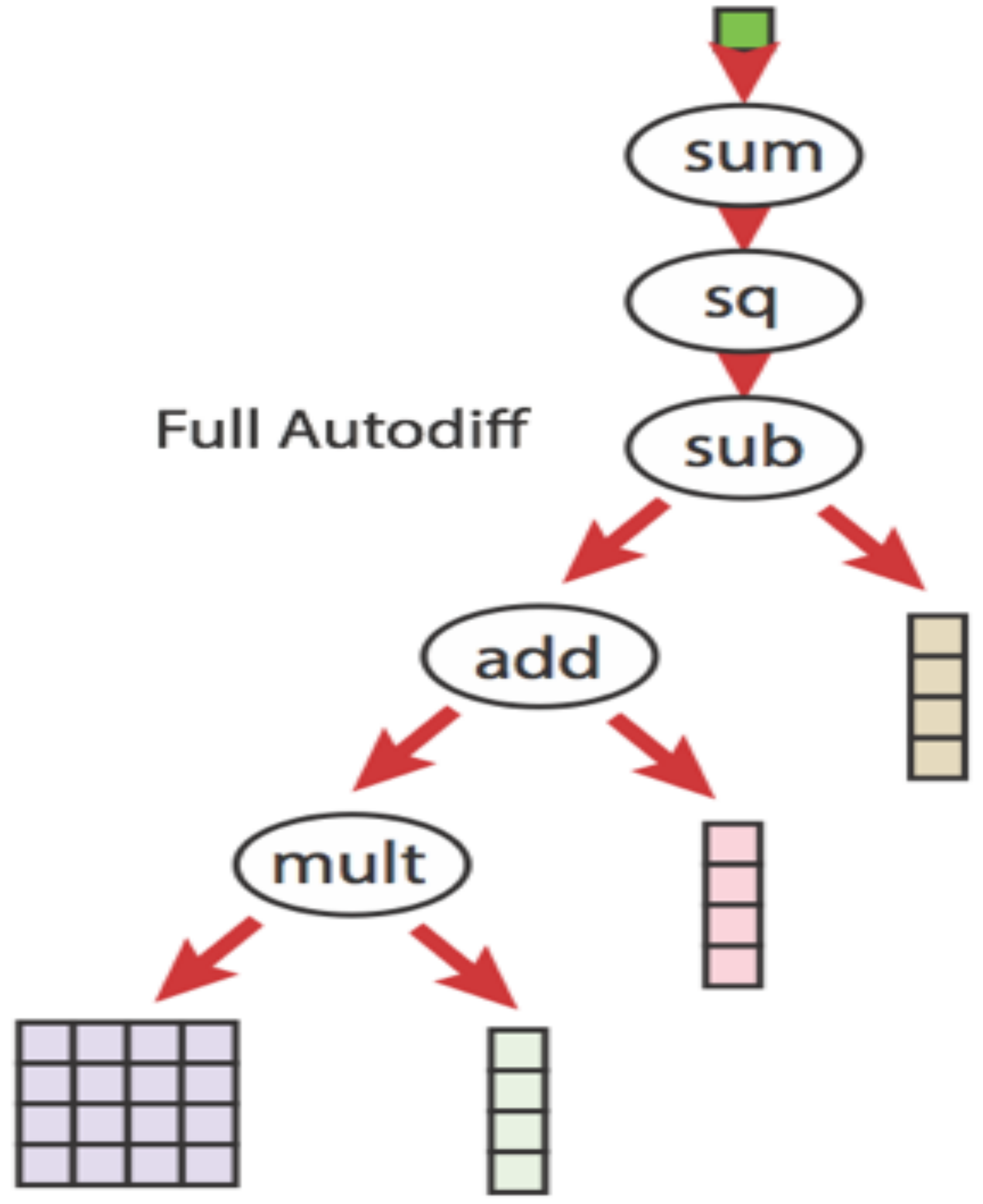
Why can't we mix these styles?

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
```

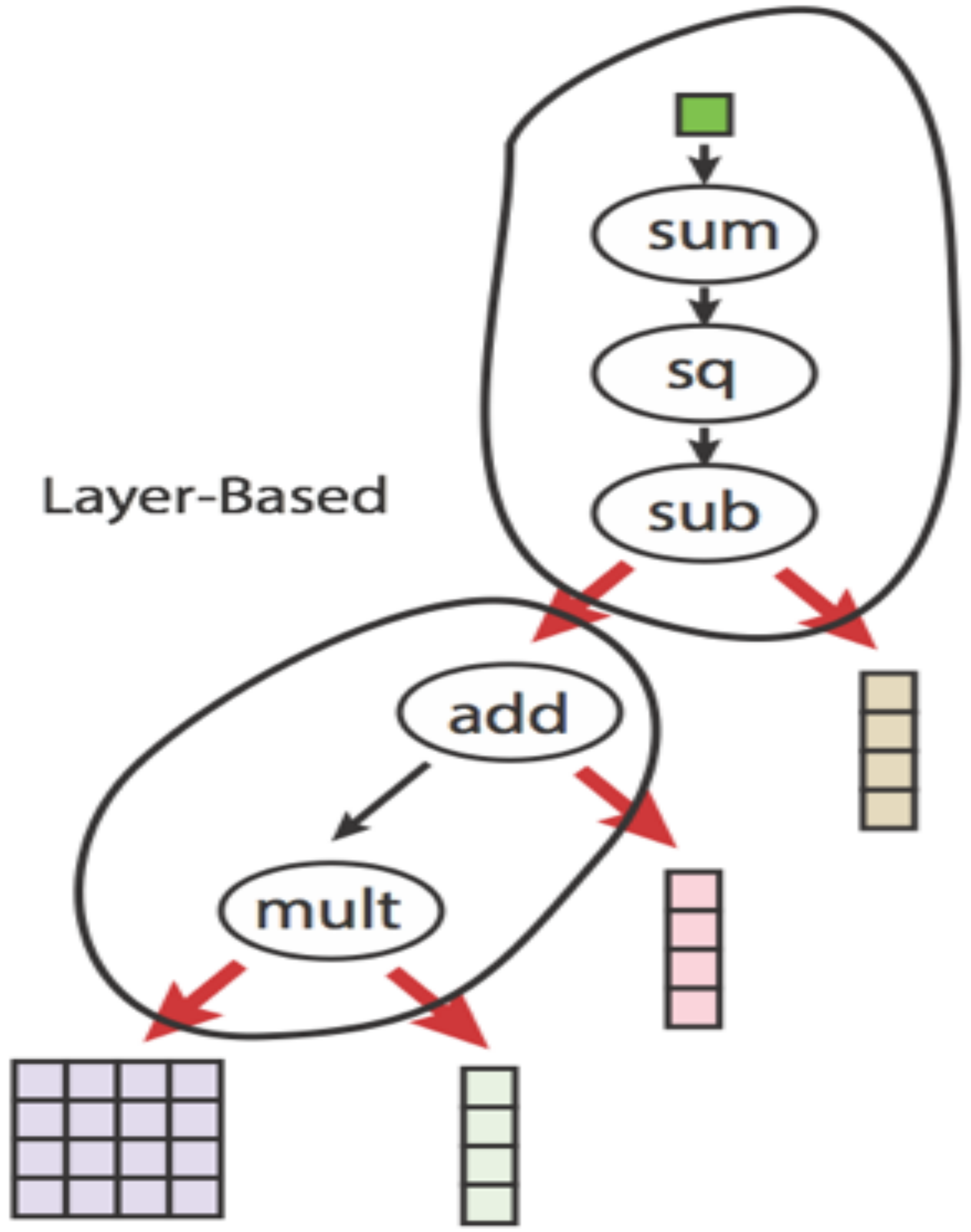
Full Autodiff



NEURAL NET THREE WAYS

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

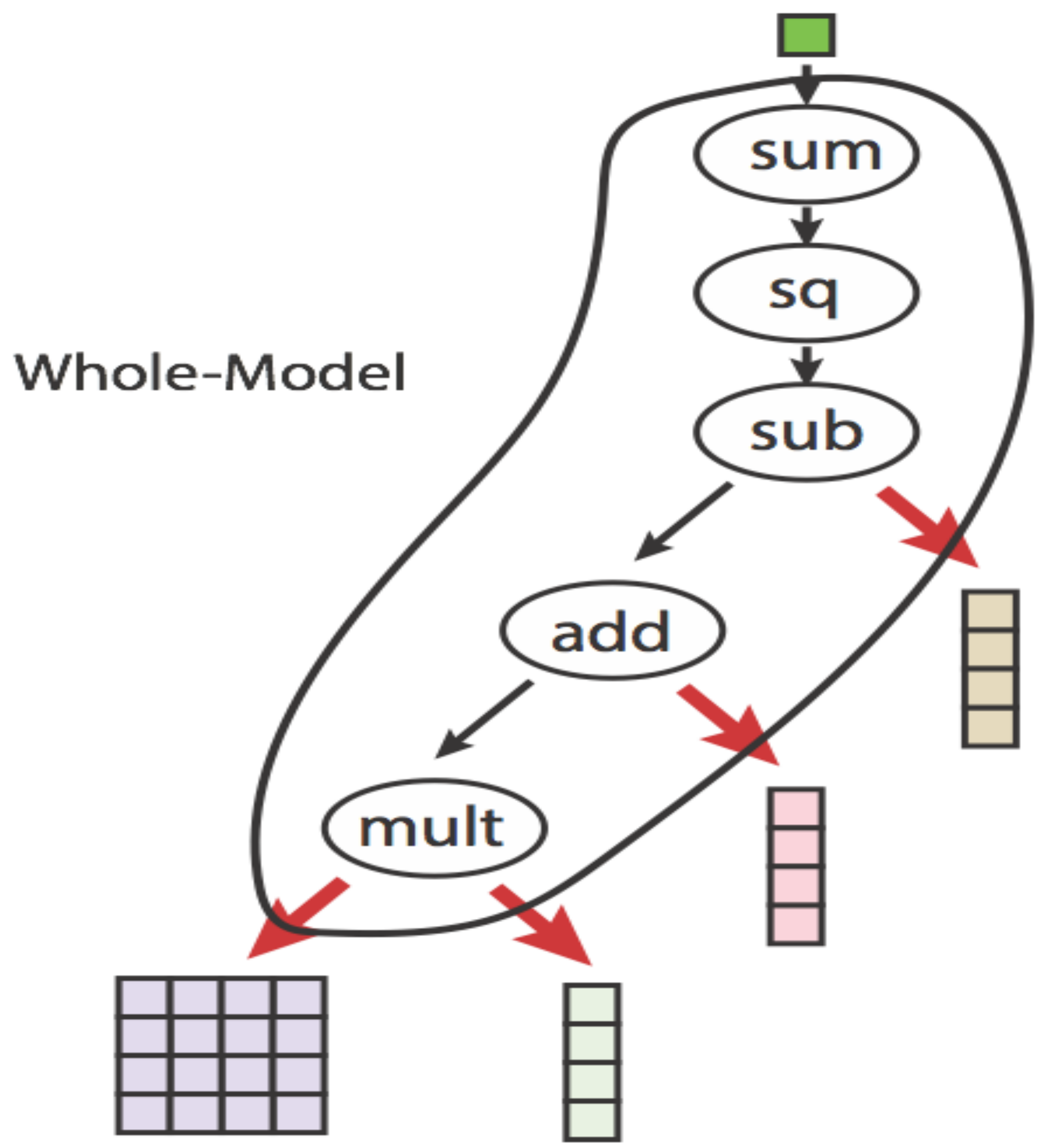
```
-- Define our layers and their parameters
local params = {}
local linear1, linear2, linear3, acts1, acts2, lsm, lossf
linear1, params.linear1 = autograd.nn.Linear(784, 50)
acts1 = autograd.nn.Tanh()
linear2,params.linear2 = autograd.nn.Linear(50, 50)
acts2 = autograd.nn.Tanh()
linear3, params.linear3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



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

Making their way from atmospheric science (and others) to machine learning

- **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* first to implement this generally for neural nets.
- **Mixing forward and reverse mode** — called "cross-country elimination". No need to evaluate partial derivatives in one direction! For diamond or hour-glass shaped compute graphs, this will be more efficient than one method alone.
- **Stencils** — image processing (convolutions) and element-wise ufuncs can be phrased as stencil operations. More efficient, general-purpose implementations of differentiable stencils needed (computer graphics do this, Guenter 2007, extending with DeVito et al., 2016).
- **Source-to-source** — All neural net autodiff packages use either ahead-of-time compute graph construction, or operator-overloading. The original method for autodiff (in FORTRAN, in the 80s) was source transformation. I believe still gold-standard for performance. Challenge (besides wrestling with host language) is control flow.
- **Higher-order gradients** — hessian = grad(grad(f)). Not many efficient implementations. 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
```

PRACTICAL SESSION

We'll work through (all in an iTorch notebook)

— Torch basics

— Running code on the GPU

— Training a CNN on CIFAR-10

— Using autograd to train neural networks

We have an autograd Slack team: http://autograd.herokuapp.com/

Join #summerschool channel

QUESTIONS?

Happy to help at the practical session

Find me at:

@awiltsch

awiltschko@twitter.com

github.com/alexbw