torch

A Deeper Dive

Soumith Chintala
Facebook AI Research

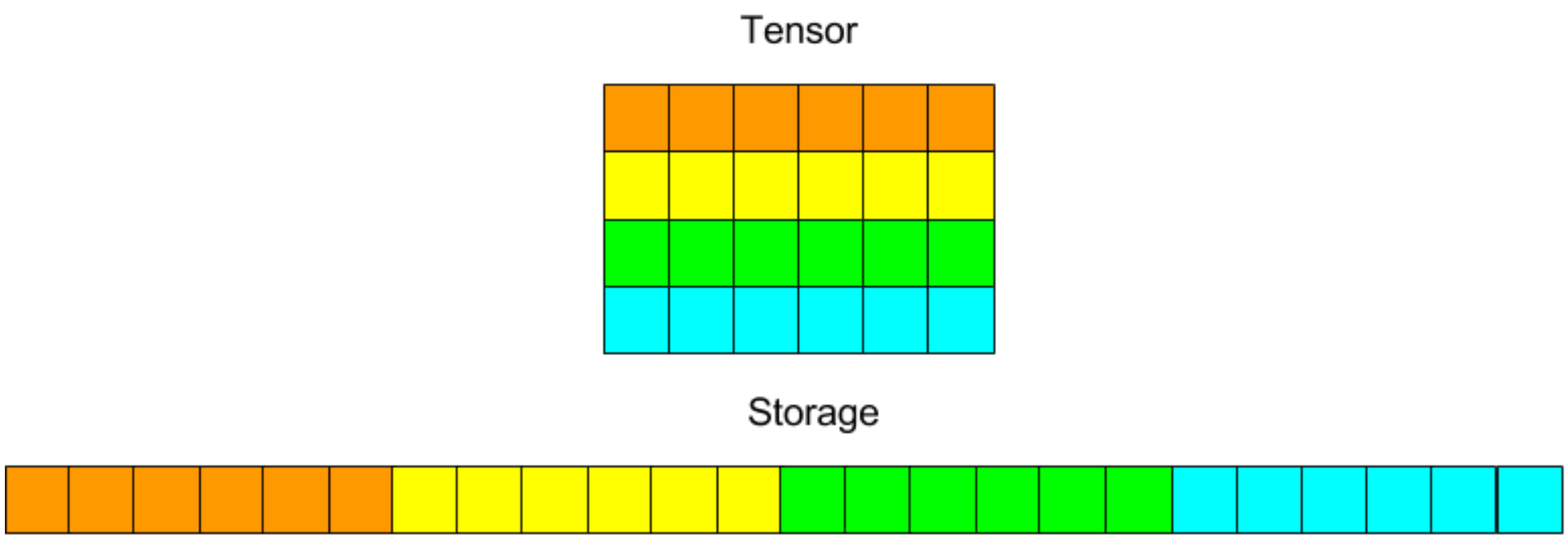
Overview

- Tensors and Storages
- Neural Networks

torch

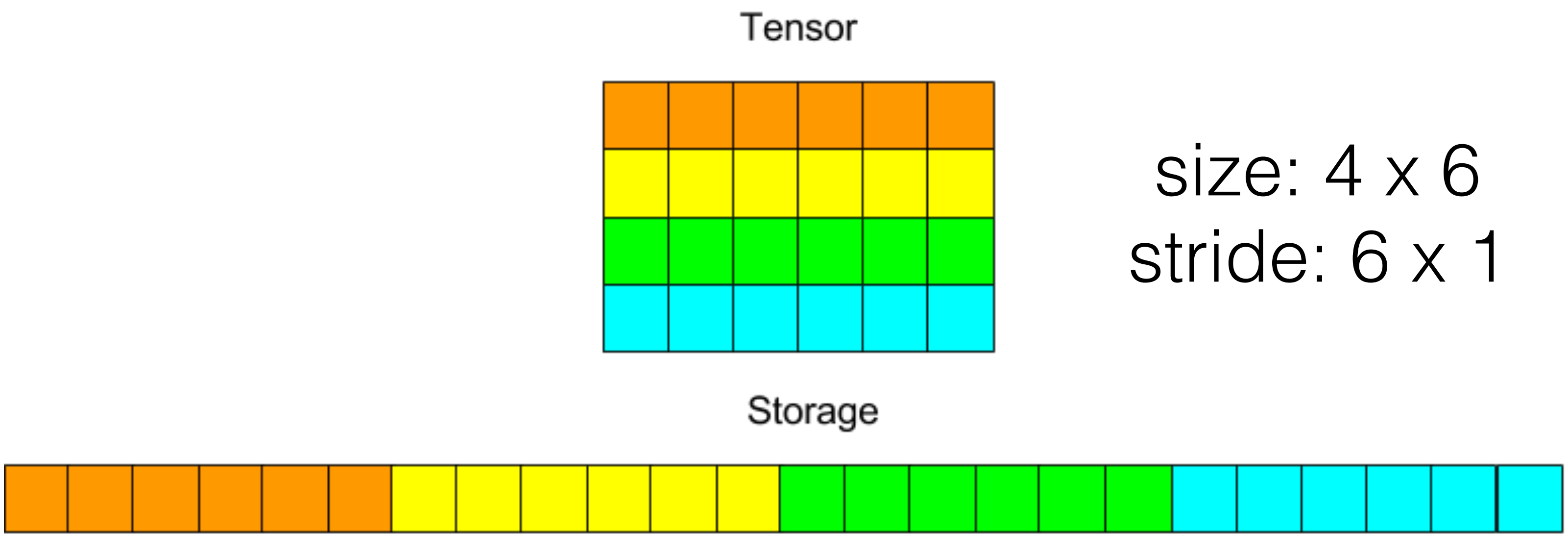
Tensors and Storages

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



Tensors and Storages

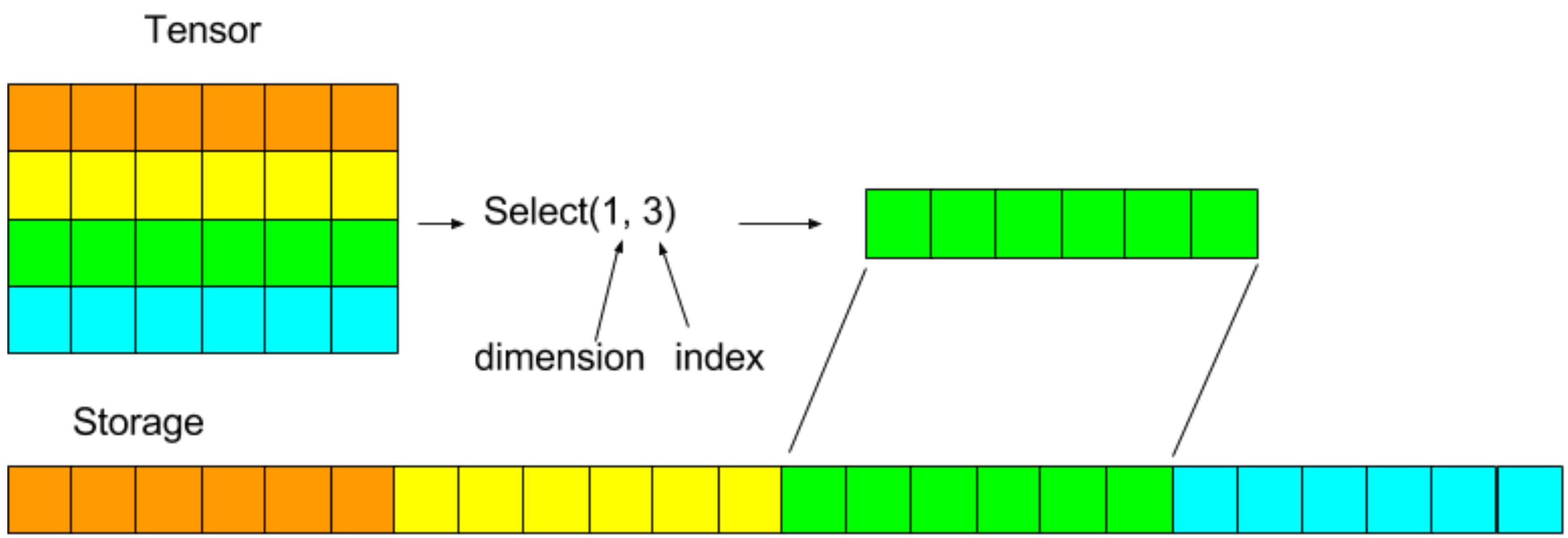
- Tensor = n-dimensional array
- Row-major in memory



size: 4 x 6
stride: 6 x 1

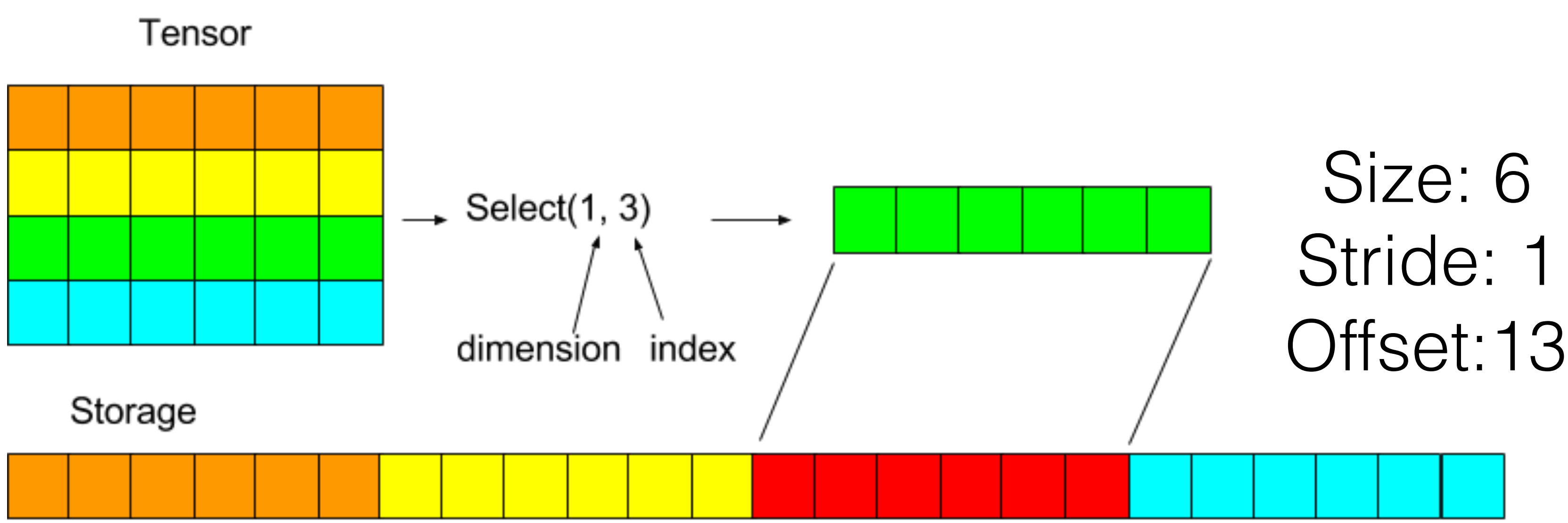
Tensors and Storages

- Tensor = n-dimensional array
- 1-indexed



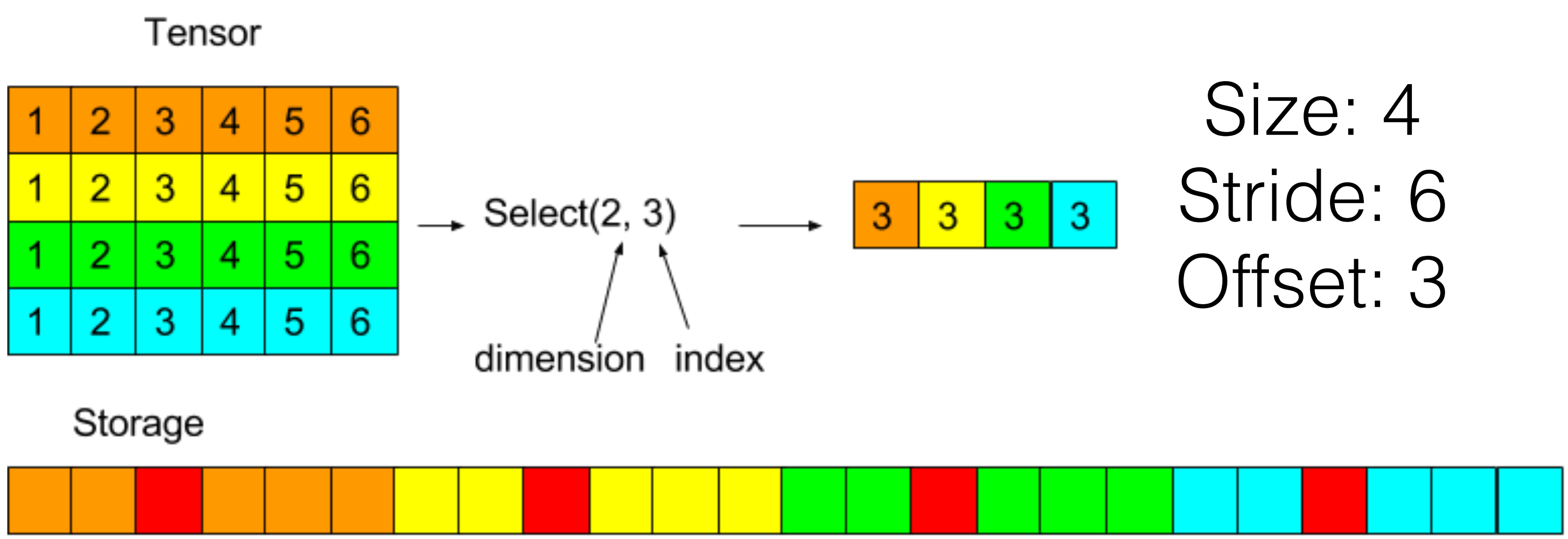
Tensors and Storages

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



Tensors and Storages

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



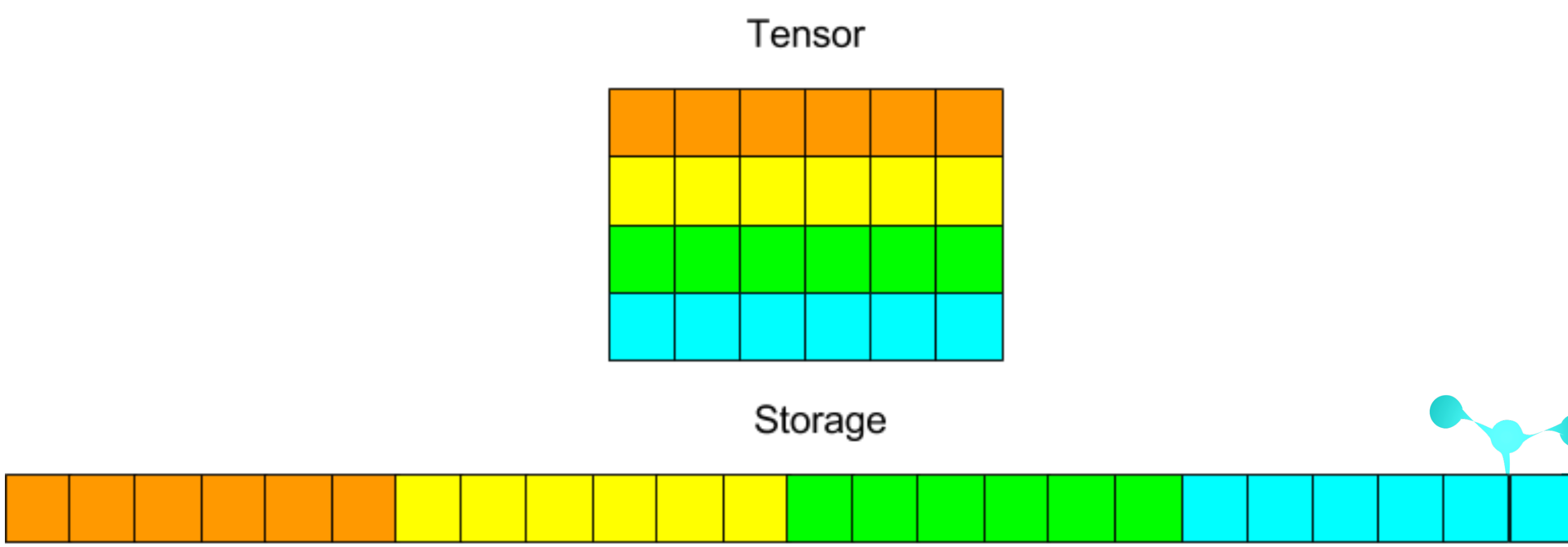
Tensors and Storages

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]
```

torch

Tensors and Storages

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

Tensors and Storages

- 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 .
         ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Tensors and Storages

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

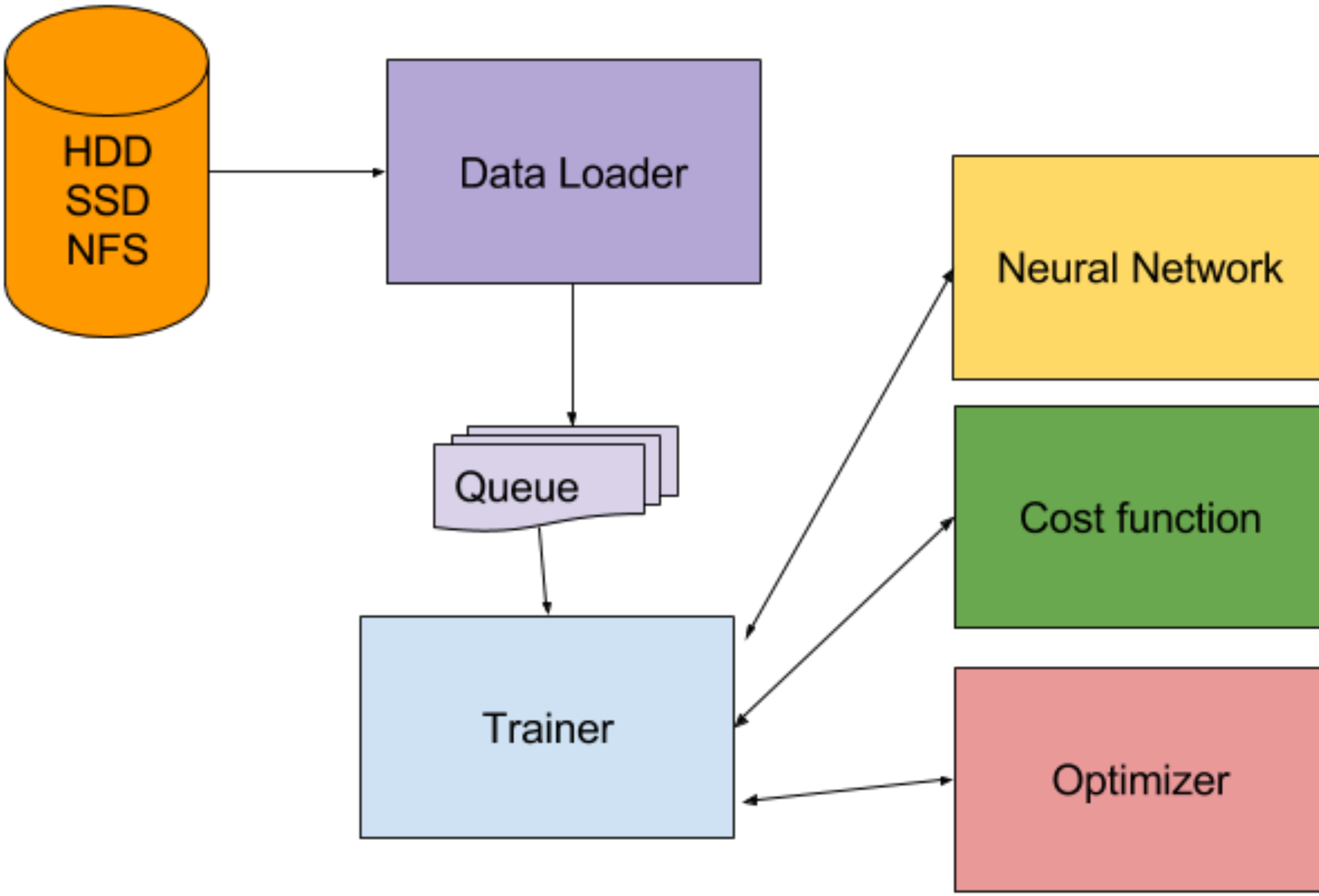
In [ ]:
```
require 'cutorch'
a = torch.CudaTensor(4, 6):uniform()
b = a:select(1, 3)
b:fill(3)
```

Training Neural Networks

torch

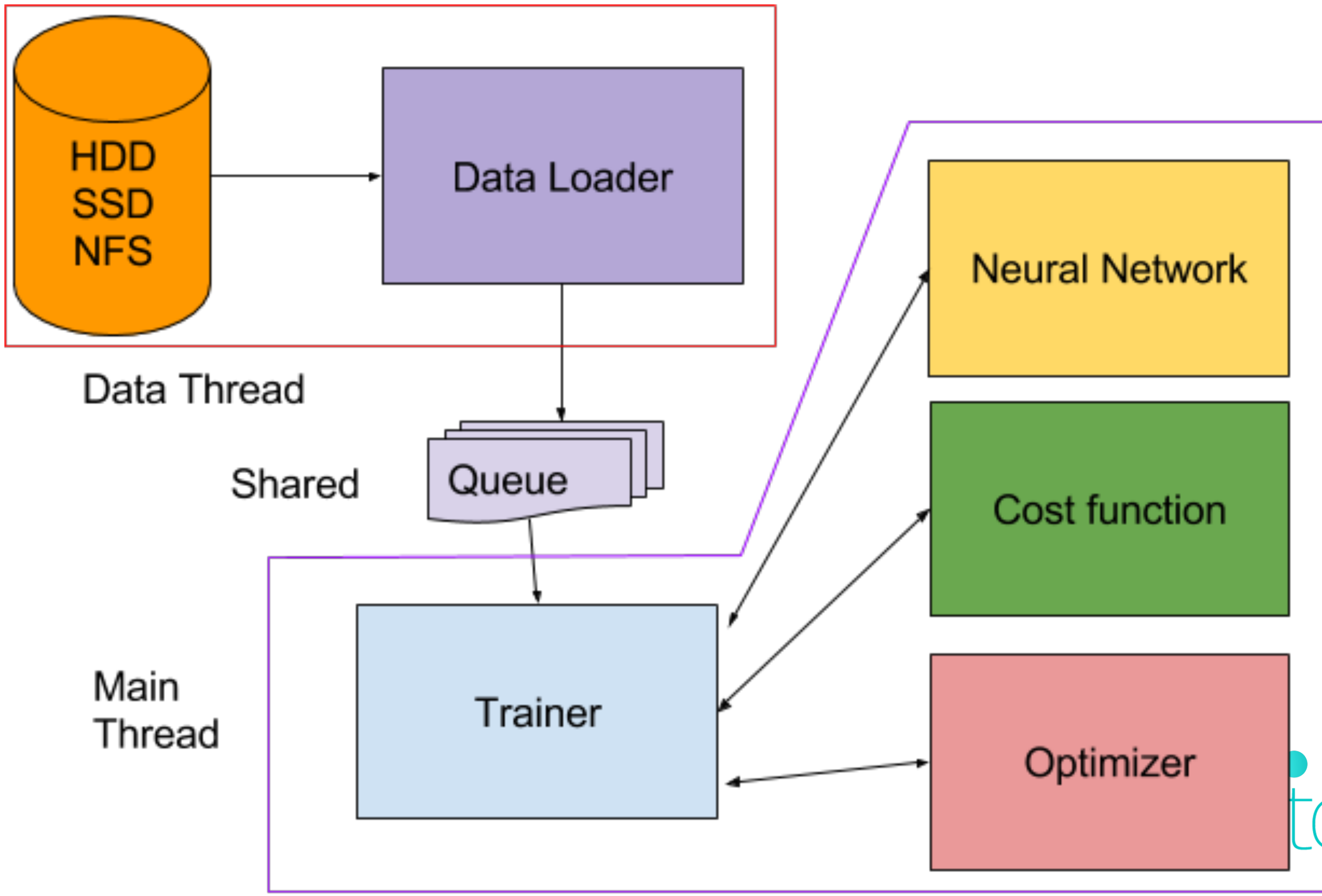
Training cycle

Moving parts

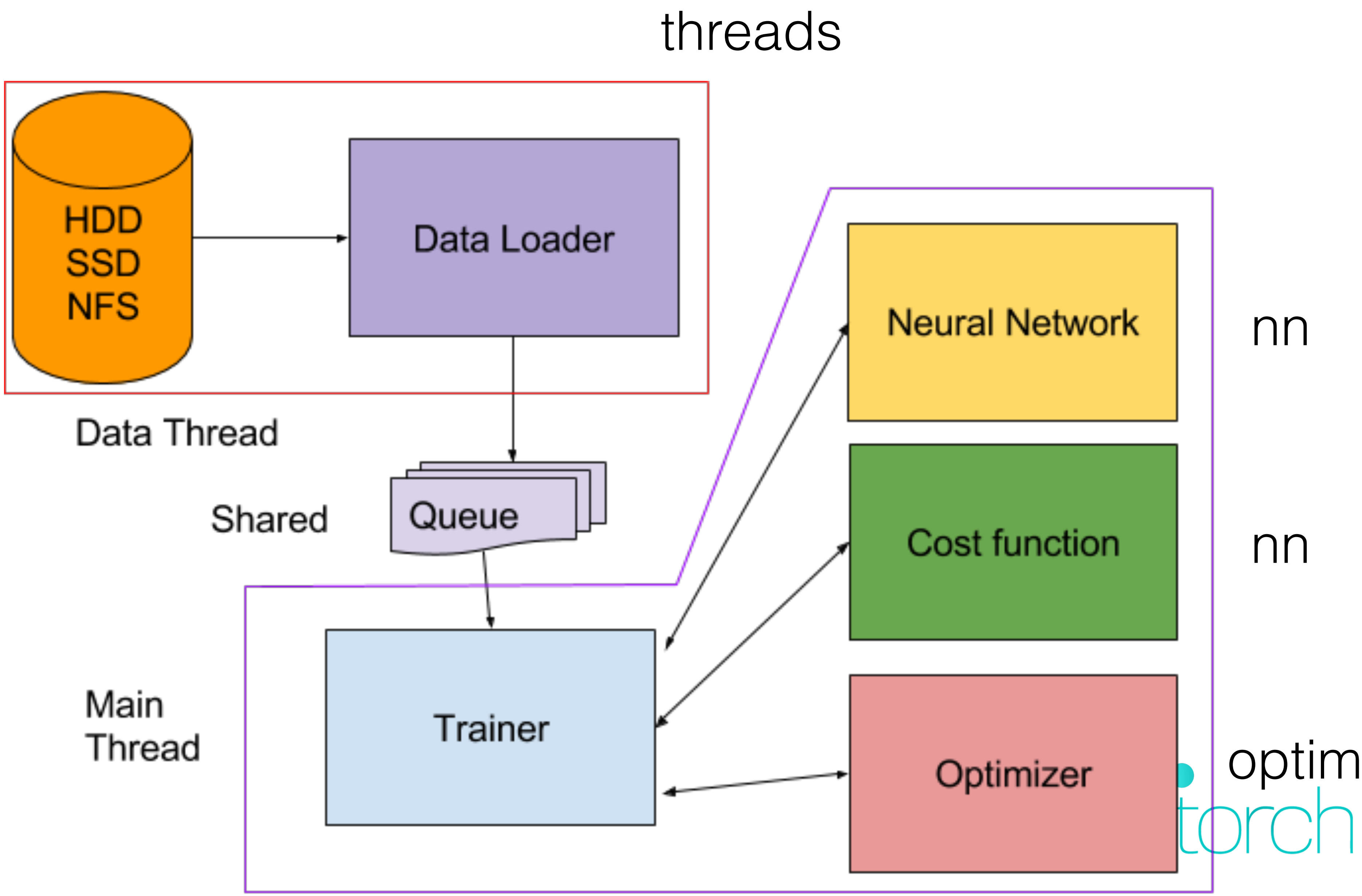


Training cycle

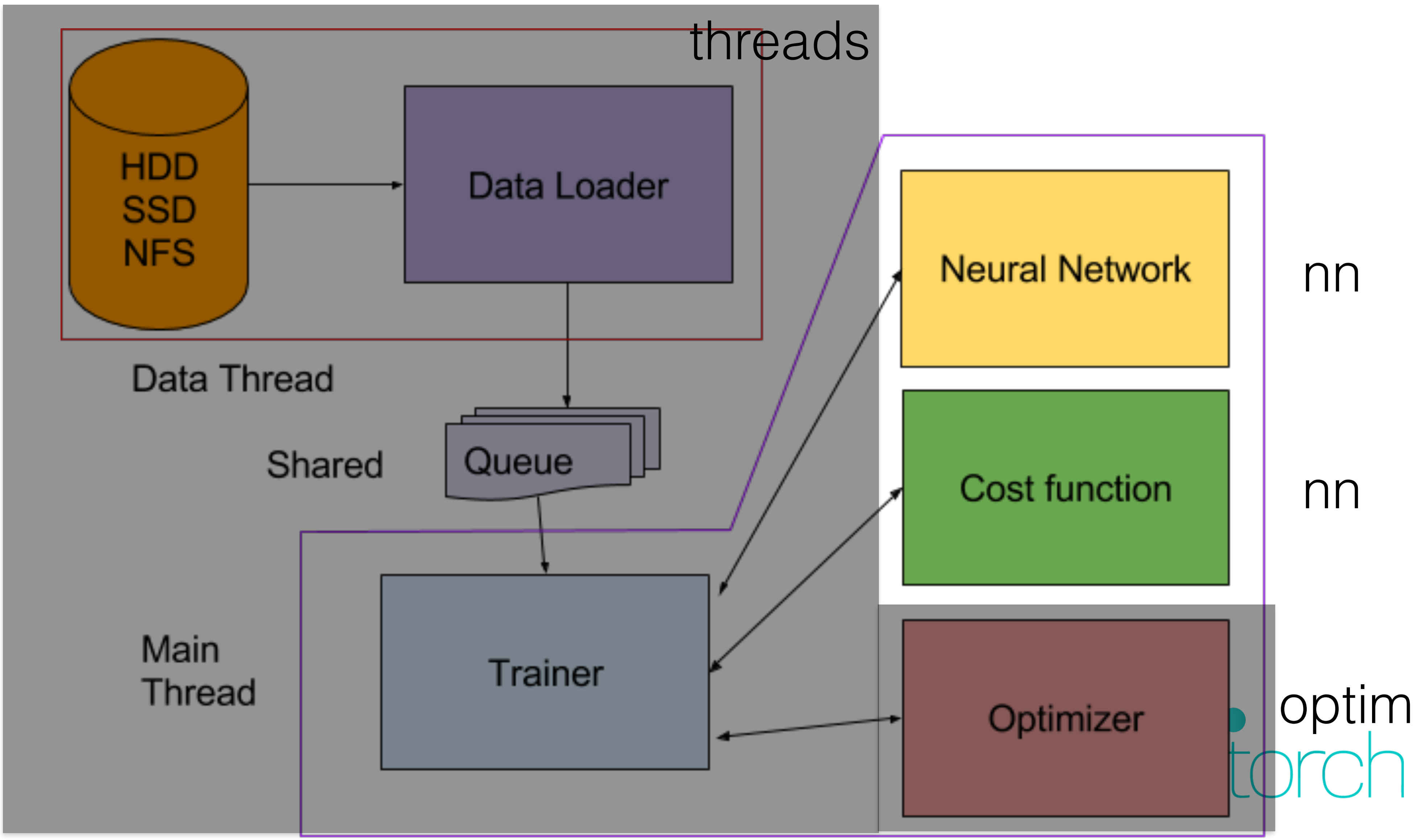
Moving parts



Training cycle



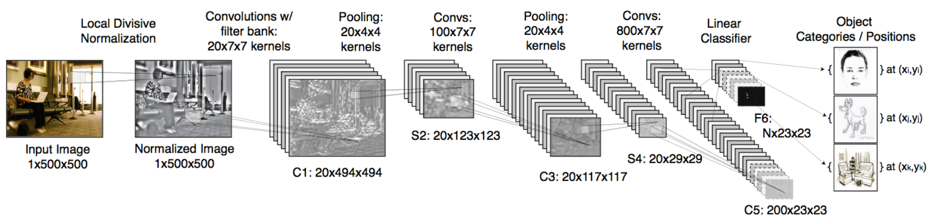
the **nn** package



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() )
```



torch

the **nn** package

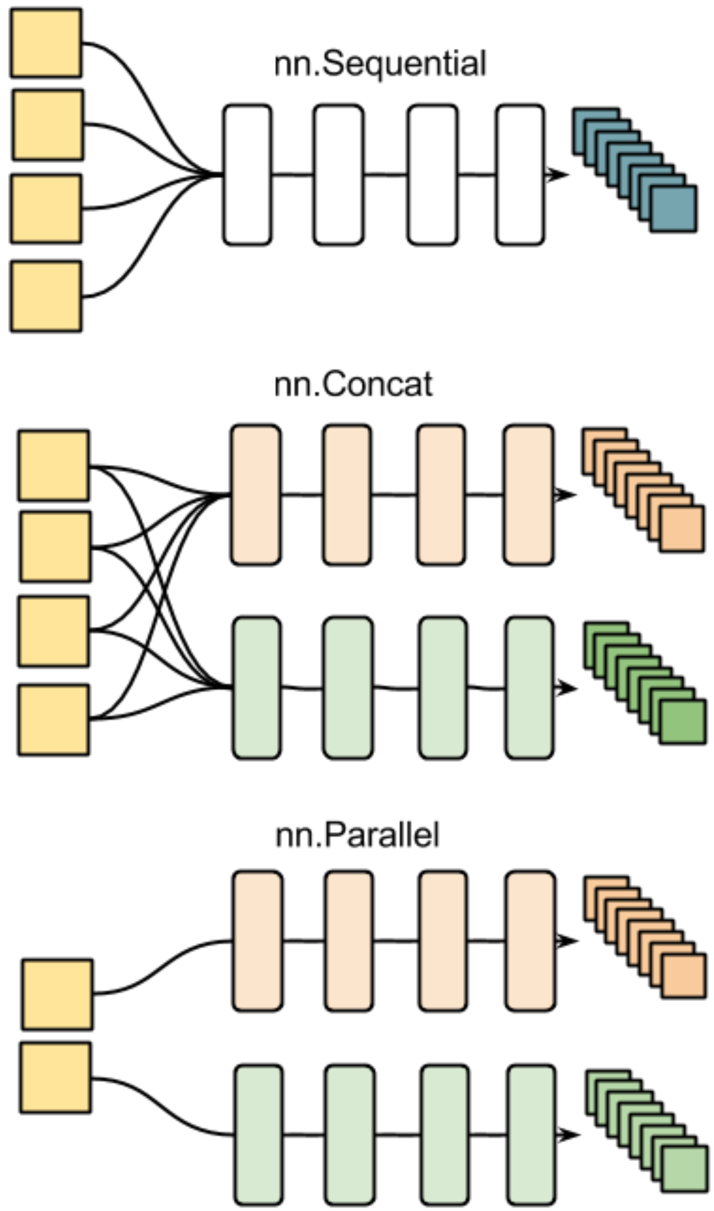
- When training neural nets, autoencoders, linear regression, convolutional networks, and any of these models, we're interested in gradients, and loss functions

- 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

Compose networks like Lego blocks



the **nn** package

CUDA Backend via the cunn package

```
-- 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)
```

torch

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')
```

torch

the **optim** package

A purely functional view of the world

```
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
```

torch

the **optim** package

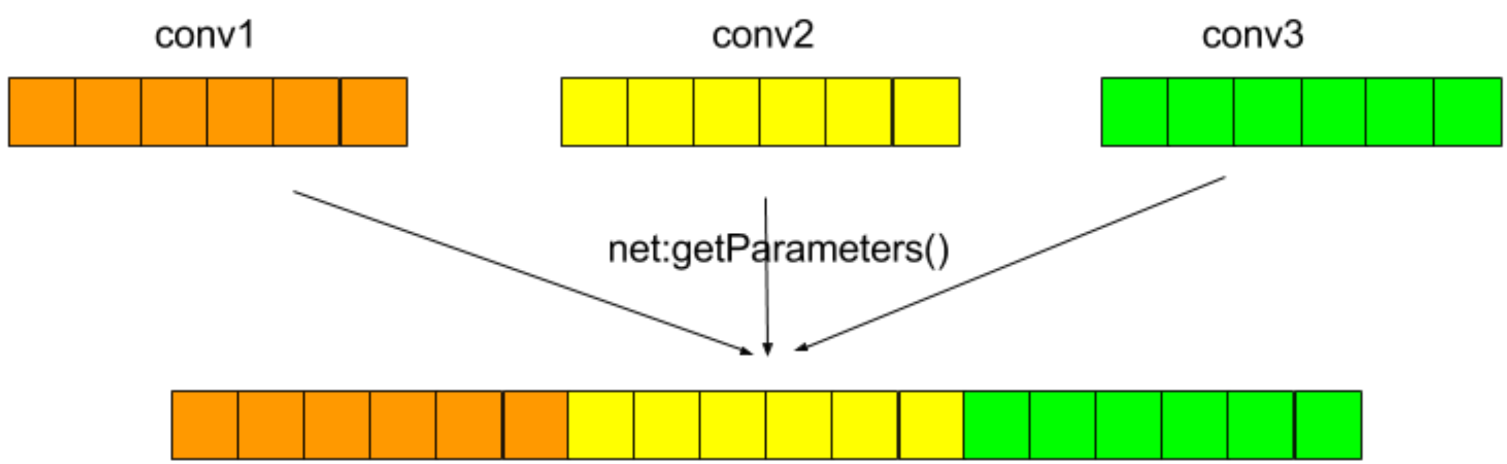
- *Stochastic Gradient Descent*
- *Averaged Stochastic Gradient Descent*
- *L-BFGS*
- *Congugate Gradients*
- *AdaDelta*
- *AdaGrad*
- *Adam*
- *AdaMax*
- *FISTA with backtracking line search*
- *Nesterov's Accelerated Gradient method*
- *RMSprop*
- *Rprop*
- *CMAES*

torch

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



the **optim** package

A purely functional view of the world

```
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

Define closure

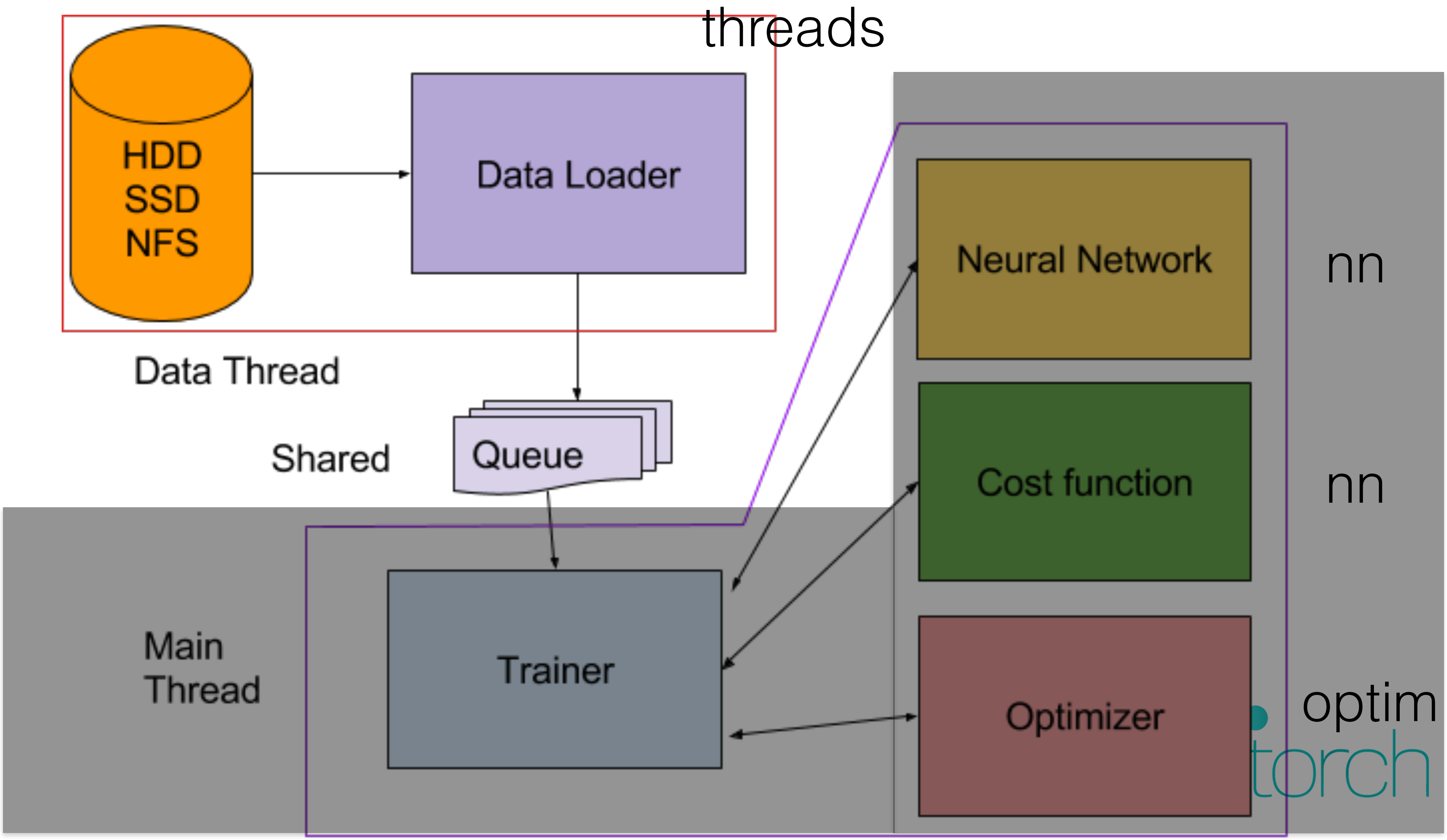
```
-- define a closure, that computes the loss, and dloss/dx
feval = function()
   -- select a new training sample
   _nidx_ = (_nidx_ or 0) + 1
   if _nidx_ > (#data)[1] then _nidx_ = 1 end

   local sample = data[_nidx_]
   local inputs = sample[1]
   local target = sample[2]

   -- reset gradients (gradients are always accumulated,
   --                  to accomodate batch methods)
   dl_dx:zero()

   -- evaluate the loss function and its derivative wrt x,
   -- for that sample
   local loss_x = criterion:forward(model:forward(inputs), target)
   model:backward(inputs, criterion:backward(model.output, target))

   -- return loss(x) and dloss/dx
   return loss_x, dl_dx
end
```

the **optim** package

Define closure

```
-- SGD parameters
sgd_params = {learningRate = 1e-3, learningRateDecay = 1e-4,
              weightDecay = 0, momentum = 0}

-- train for a number of epochs
epochs = 1e2
for i = 1,epochs do
   -- this variable is used to estimate the average loss
   current_loss = 0

   -- an epoch is a full loop over our training data
   for i = 1,(#data)[1] do

      -- one step of SGD optimization (steepest descent)
      _,fs = optim.sgd(feval,x,sgd_params)

      -- accumulate error
      current_loss = current_loss + fs[1]
   end

   -- report average error on epoch
   current_loss = current_loss / (#data)[1]
   print(' current loss = ' .. current_loss)
end
```

the **threads** package



the **threads** package

- Create data-loading threads on demand
  - (hilariously called donkeys and it stuck)
- callbacks that are executed in the main thread get data from donkeys, and call optimization functions

```lua
local ffi = require 'ffi'
local Threads = require 'threads'
Threads.serialization('threads.sharedserialize')

-- This script contains the logic to create K threads for parallel data-loading.
-- For the data-loading details, look at donkey.lua
-------------------------------------------------------------------------------
do -- start K datathreads (donkeys)
   if opt.nDonkeys > 0 then
      local options = opt -- make an upvalue to serialize over to donkey threads
      donkeys = Threads(
         opt.nDonkeys,
         function()
            require 'torch'
         end,
         function(idx)
            opt = options -- pass to all donkeys via upvalue
            tid = idx
            local seed = opt.manualSeed + idx
            torch.manualSeed(seed)
            print(string.format('Starting donkey with id: %d seed: %d', tid, seed))
            paths.dofile('donkey.lua')
         end
      );
```

the **threads** package

```lua
for i=1,opt.epochSize do
   -- queue jobs to data-workers
   donkeys:addjob(
      -- the job callback (runs in data-worker thread)
      function()
         local inputs, labels = trainLoader:sample(opt.batchSize)
         return inputs, labels
      end,
      -- the end callback (runs in the main thread)
      trainBatch
   )
end

donkeys:synchronize()
```

```lua
function trainBatch(inputsCPU, labelsCPU)
   cutorch.synchronize()
   collectgarbage()
   local dataLoadingTime = dataTimer:time().real
   timer:reset()

   -- transfer over to GPU
   inputs:resize(inputsCPU:size()):copy(inputsCPU)
   labels:resize(labelsCPU:size()):copy(labelsCPU)

   local err, outputs
   feval = function(x)
      model:zeroGradParameters()
      outputs = model:forward(inputs)
      err = criterion:forward(outputs, labels)
      local gradOutputs = criterion:backward(outputs, labels)
      model:backward(inputs, gradOutputs)
      return err, gradParameters
   end
   optim.sgd(feval, parameters, optimState)
```

Next up

- A complete example of using nn + optim + threads for image generation
- the magic autograd package
- torchnet: common patterns for Torch by Facebook