# Torch7
# Scientific computing for Lua(JIT)

www.torch.ch

Ronan Collobert | Koray Kavukcuoglu | Clément Farabet | …

- **1: Getting started**

    - ➡ Torch's main site and resources: www.torch.ch
    On Github: https://github.com/torch

    - ➡ Torch cheat sheet
    https://github.com/torch/torch7/wiki/Cheatsheet

    - ➡ Tutorials for Torch: http://torch.madbits.com
    On Github: https://github.com/clementfarabet/torch-tutorials

    - ➡ Lua: http://www.lua.org
    LuaJIT: http://luajit.org/luajit.html

# Torch7



- **Torch has been around since 2000**
  - ➡ Ronan Collobert has been the main dev for all
  - ➡ 4 versions (odd numbers)
  - ➡ Various languages (C, C++, now Lua+C)
  - ➡ A liberal BSD license
  - ➡ Includes lots of packages for neural networks, optimization, graphical models, image processing
  - ➡ More than 50,000 downloads, universities and major industrial labs (Google, Facebook, Twitter)
- **Torch always aimed large-scale learning**
  - ➡ Speech, image and video applications
  - ➡ Large-scale machine-learning applications

# Torch7



- **Why a mixed language approach?**
  - ➡ Complex applications => proper scripting language (LuaJIT)
  - ➡ Fast and demanding applications => compiled and optimized backend (C,C++,CUDA,OpenMP)
- **LuaJIT is a great scripting environment**
  - ➡ Fastest scripting language, with a transparent JIT compiler
  - ➡ Simple, readable (like Python), with clean/consistent constructs
  - ➡ The cleanest interface to C (even cleaner/simpler with FFI)
  - ➡ Embeddable into any environment (iPhone apps, Video games, web backends ...)

# Torch7



- **Why build Torch around LuaJIT and not simply use Python?**
  - ➡ We are obsessed with speed: LuaJIT is very lightweight, and rarely gets in your way (manipulate raw C pointers straight from LuaJIT)
  - ➡ We wanted to build applications: the complete Torch framework (Lua included) is self-contained, so you can transform your scripts into easily distributable programs
  - ➡ We wanted to easily port our code to any platform: the complete Torch framework runs on iPhone, with no modification to our scripts
  - ➡ We wanted easy extensibility: LuaJIT's FFI interface is one of the simplest to learn, it's easy to integrate any library into Torch

- **Lua provides a unique, universal data structure: the table**
  - ➡ The Lua table can be used as an array, dictionary (hash table), class, object, struct, list, …

```
my_table = { 1, 2, 3 }
my_table = { my_var = 'hello', my_other_var = 'bye' }
my_table = { 1, 2, 99, my_var = 'hello' }
my_function = function() print('hello world') end
my_table[my_function] = 'this prints hello world'
my_function()
print(my_table[my_function])
```

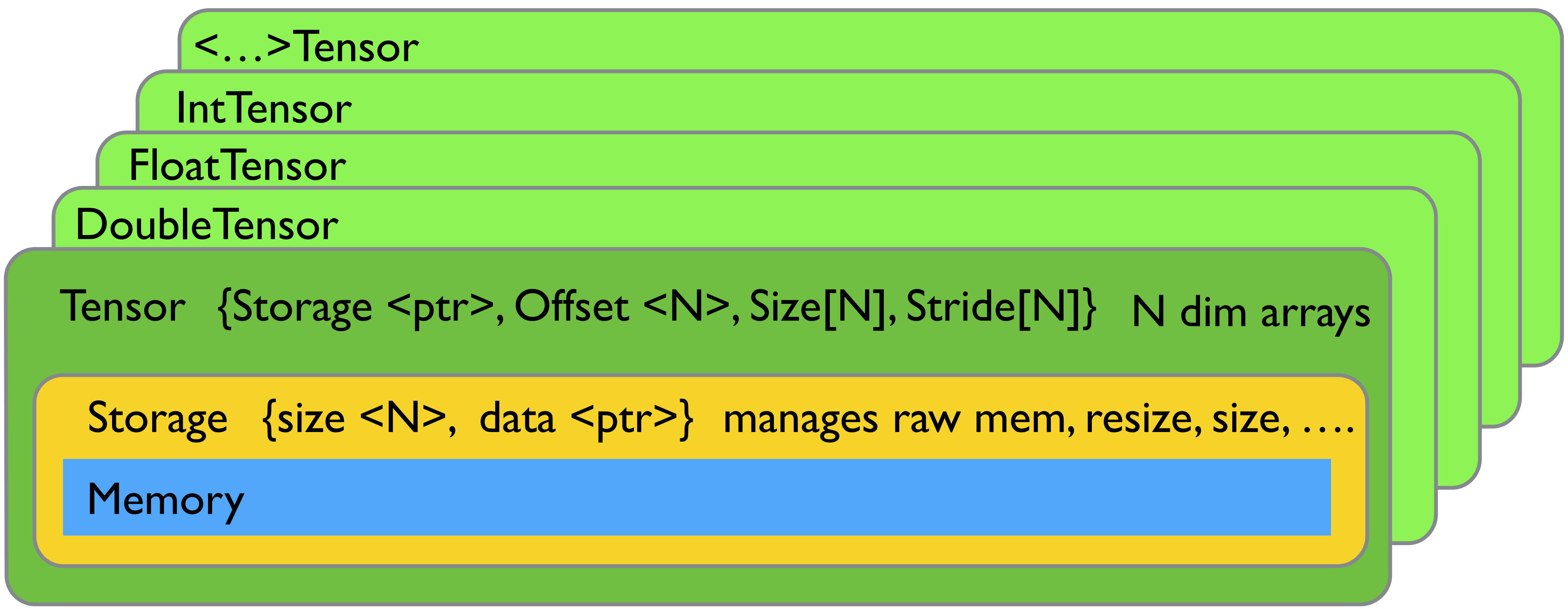
```
Torch 7.0  Copyright (C) 2001-2011 Idiap, NEC Labs, NYU
hello world
this prints hello world
```

- **Lua supports closures**
  - ➡ Closures allow very flexible programmatic constructs: on-the-fly object creation, flexible data structure creation, ...

# Torch7



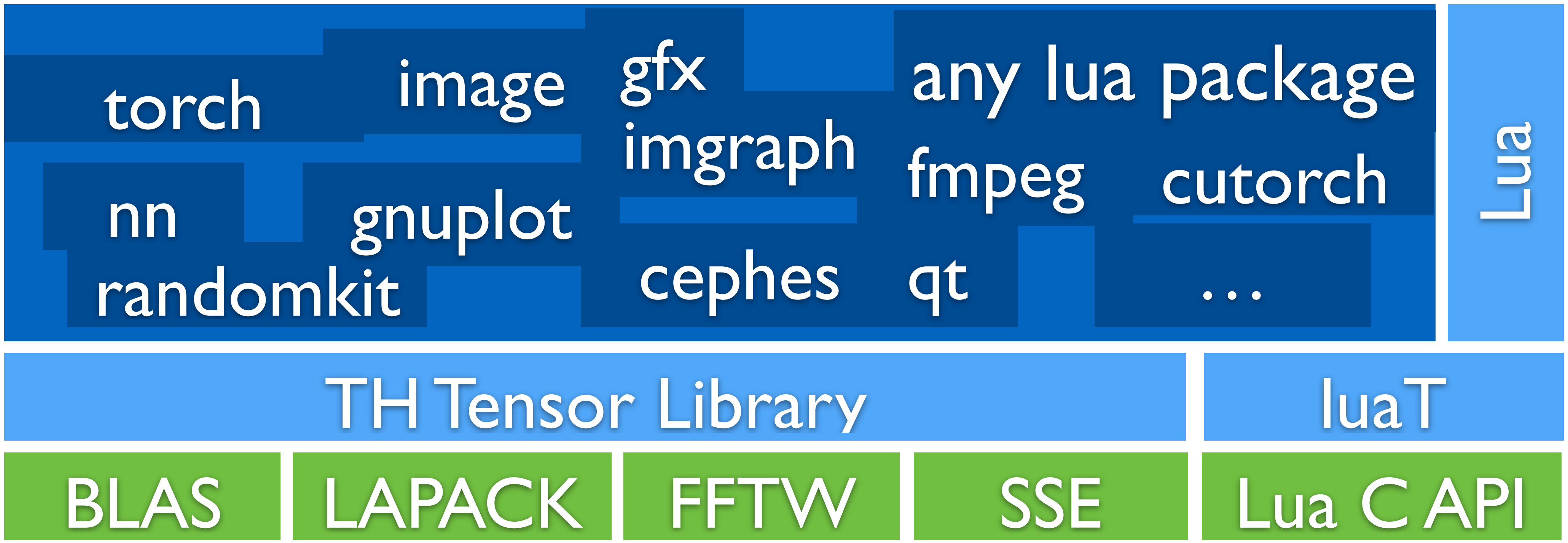
- **Torch7 extends Lua’s table with a Tensor object:**
  - ➡ An N-Dimensional array type, which supports views
  - ➡ A Tensor is a view of a chunk of memory
  - ➡ A chunk of memory might have **several views** (Tensors) pointing to it, with different geometries



# Torch7



- **Torch7 provides a rich set of packages**
  - ➡ Based on Matlab's common routines (zeros,ones,eye, ...)
  - ➡ Linear algebra stuff
  - ➡ Convolutions, Fourier transform, …
  - ➡ plotting
  - ➡ statistics
  - ➡ …



- **Package Manager**
  - ➡ Many more packages are available via Lua's package manger: `luarocks`
  - ➡ Check out what's available here: github.com/torch/rocks

- **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`

# Torch7



- **Optimized backends**

  ➡ **CPU, using OpenMP + SSE**
  ➡ **GPU, using CUDA**
  ➡ **cutorch : TH/torch for CUDA**
  ➡ **cunn : nn for CUDA**
  ➡ **wrappers for cuda-convnet**

- **For up-to-date benchmarking comparing caffe/theano/torch/cuda-convet/…**

  https://github.com/soumith/convnet-benchmarks

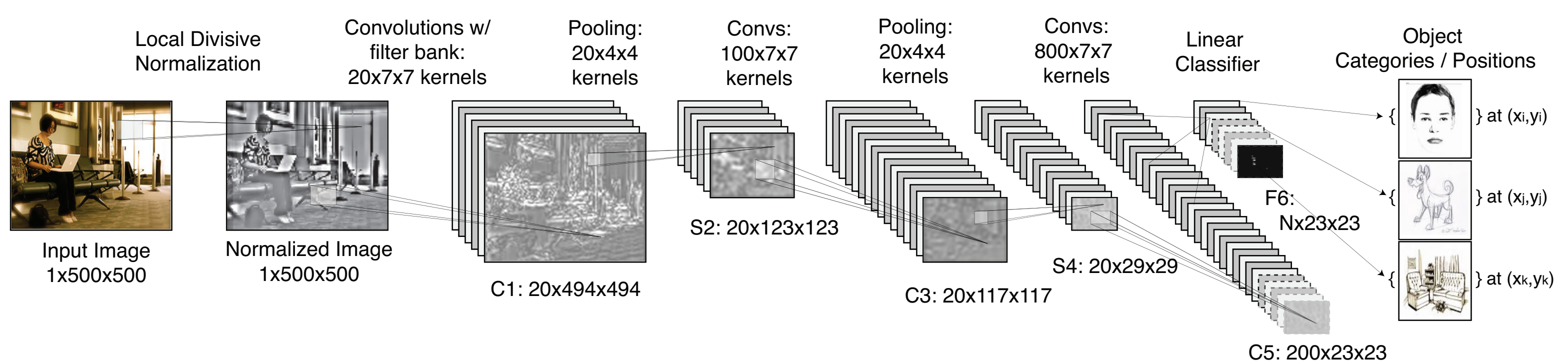
- **Going Further:**
  - ➡ **Torch7**:
    http://www.torch.ch/
    https://github.com/torch
  - ➡ **Basic Demos**: a bunch of demos/tutorials to get started
    https://github.com/clementfarabet/torch7-demos
  - ➡ **Deep-Learning Tutorials**: supervised and unsupervised learning
    http://code.madbits.com
  - ➡ **luarocks**: Lua's package manager, to get new packages:
    $ luarocks search --all # list all packages
    $ luarocks install optim # install optim package
  - ➡ **Torch Group**: get help!
    https://groups.google.com/forum/?fromgroups#!forum/torch7

- **2: Supervised Learning**

  ➡ pre-process the (train and test) data, to facilitate learning
  ➡ describe a model to solve a classification task
  ➡ choose a loss function to minimize
  ➡ define a sampling procedure (stochastic, mini-batches), and apply one of several optimization techniques to train the model's parameters
  ➡ estimate the model's performance on unseen (test) data
  ➡ do all the exercises!

- **Example: convolutional network, for natural images**
  - ➡ **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() )
```



- **Example: logistic regression**

  ➡ **step 4/5: define a closure that estimates `f(x)` and `df/dx` stochastically**

```
-- 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
```

- **Example: logistic regression**

➡ **step 5/5: estimate parameters (train the model), stochastically**

```
-- 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
```

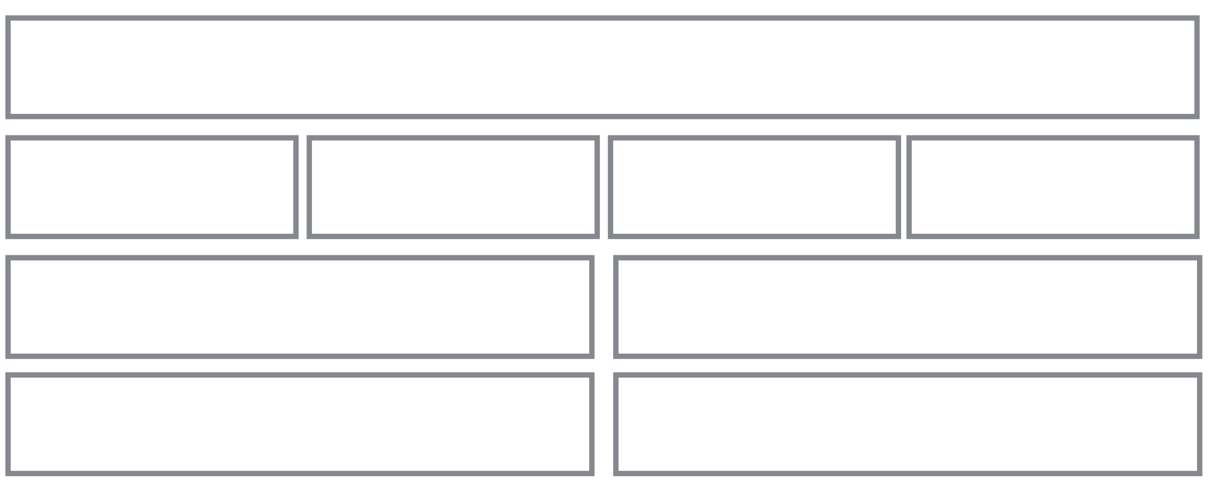
- **Example: optimize differently**

➡ **step 5/5: estimate parameters (train the model), using LBFGS**

```
-- LBFGS parameters
lbfgs_params = {lineSearch = optim.lswolfe}

-- 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.lbfgs(feval,x,lbfgs_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
```

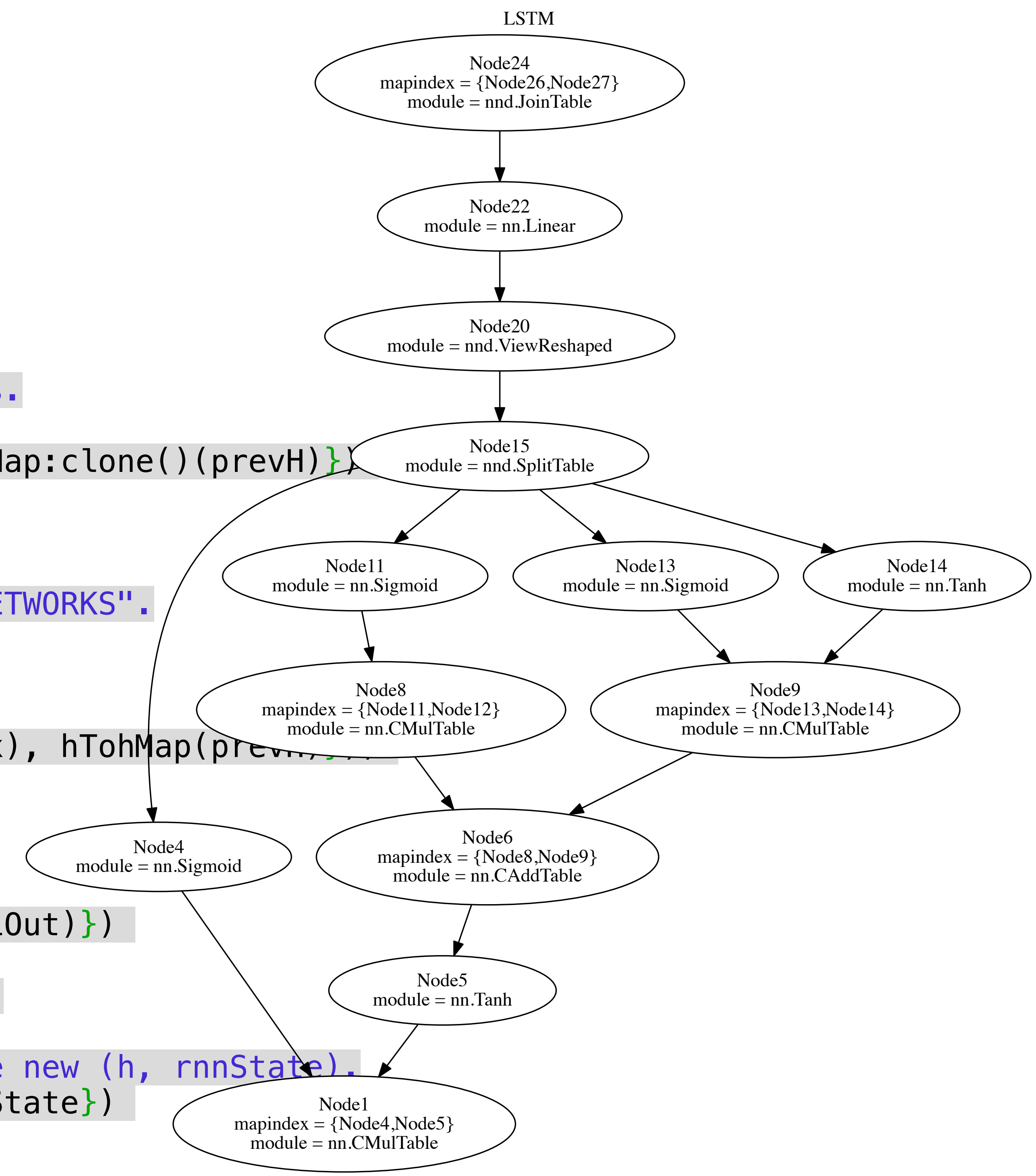
- **Arbitrary models can be constructed using lego-like containers:**

```
nn.Sequential()          -- sequential modules
nn.ParallelTable()       -- parallel modules
nn.ConcatTable()         -- shared modules
nn.SplitTable()          -- (N)dim Tensor -> table of (N-1)dim Tensors
nn.JoinTable()           -- table of (N-1)dim Tensors -> (N)dim Tensor
```



- **Or using graph container directly**

```lua
function nnd.Lstm(xTohMap, hTohMap)

    local x = nn.Identity()()
    local prevRnnState = nn.Identity()()
    local prevH, prevCell = prevRnnState:split(2)

    -- The input sum produces (Wx + Wh + b).
    -- Each input sum will use different weight matrices.
    local function newInputSum()
        return nn.CAddTable()({xTohMap:clone()(x), hTohMap:clone()(prevH)})
    end

    -- The following are equations (3) to (7) from
    -- "SPEECH RECOGNITION WITH DEEP RECURRENT NEURAL NETWORKS".
    -- The peep-hole connections are not used.
    local inGate = nn.Sigmoid()(newInputSum())
    local forgetGate = nn.Sigmoid()(newInputSum())
    local cellGate = nn.Tanh()(nn.CAddTable()({xTohMap(x), hTohMap(prevH)}))
    local cellOut = nn.CAddTable()({
            nn.CMulTable()({forgetGate, prevCell}),
            nn.CMulTable()({inGate, cellGate})})
    local outGate = nn.Sigmoid()(newInputSum())
    local hOut = nn.CMulTable()({outGate, nn.Tanh()(cellOut)})

    local nextRnnState = nn.Identity()({hOut, cellOut})

    -- The LSTM takes (x, prevRnnState) and computes the new (h, rnnState).
    return nn.gModule({x, prevRnnState}, {hOut, nextRnnState})
end
```



## Changing the backend: CUDA

➡ **cunn**: that package re-defines lots of **nn** modules with CUDA
➡ to use CUDA, Tensors simply need to be cast as CudaTensors

```lua
-- 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)
```

# Torch7



- **Torch7 @ Google Deepmind**

  - ➡ Used exclusively for research and prototyping
  - ➡ Unsupervised learning
  - ➡ Supervised learning
  - ➡ Reinforcement Learning
  - ➡ Sequence Prediction
  - ➡ Many internal and external open sourced packages
    - ➡ logging
    - ➡ functional programming
    - ➡ datasets
    - ➡ random number generators (randomkit)
    - ➡ statistical distributions
    - ➡ mathematical functions (cephes)
    - ➡ many patches to torch ecosystem

# Torch7

- **x: Torch at Facebook**

# Torch7



- **We use Torch and LuaJIT at Facebook**
    - ➡ First open contributions released
    - ➡ Improving parallelism for multi-GPUs (model, data, DAG model)
    - ➡ Improving host-device communications (overlapping)
    - ➡ Computation kernels speed (e.g. convolutions in time/freq. domains)
- **See** https://github.com/facebook/fblualib

# Torch7



- **Torch packages released**

  - ➡ **fb.thrift**: fast serialization library
  - ➡ **fb.debugger**: source-level Lua debugger
  - ➡ **fb.python**: bridge between Lua and Python
  - ➡ C++ LuaUtils: collection of C++ utilities for writing Lua extensions
  - ➡ fb.util: collection of low-level Lua utilities
  - ➡ fb.editline: command line editing library based on libedit
  - ➡ fb.trepl: configurable Read-Eval-Print loop with line editing and autocompletion
  - ➡ fb.ffivector: vector of POD types does not count toward the Lua heap limit
  - ➡ fb.mattorch: library for r/w Matlab .mat files from Torch (without Matlab installed)

# Torch7



- **fb.thrift**
  - ➡ Thrift serialization for arbitrary Lua objects
  - ➡ Thrift is the multi-platform, multi-language serialization used in production at FB
  - ➡ Built-in optional compression

- **Serialization / Deserialization of Lua objects**
  - ➡ Supported types: scalars, tables, function with upvalues, torch.Tensor
  - ➡ Arbitrary cyclic object graphs
  - ➡ 3-8x faster speeds than default Torch serialization

- **fb.thrift**
  - ➡ Example

```lua
local thrift = require('fb.thrift')

local obj = { foo = 2 }  -- arbitrary Lua object

-- Serialization
-- to Lua string
local str = thrift.to_string(obj)

-- to open io.file object
local f = io.open('/tmp/foo', 'wb')
thrift.to_file(obj, f)

-- Deserialization
-- from Lua string
local obj = thrift.from_string(str)

-- from open io.file object
local f = io.open('/tmp/foo')
local obj = thrift.from_file(obj)
```

# Torch7



- **fb.debugger**
  - ➡ full-featured source-level Lua debugger
  - ➡ does not require Torch

- **2 modes of operation**
  - ➡ directly within the code

```
local debugger = require('fb.debugger')
...
-- At the point of interest, enter the debugger
debugger.enter()
...
```

  - ➡ on uncaught errors: with fb.trepl, set the environment variable

`LUA_DEBUG_ON_ERROR=1`

- **Debugger inspired by gdb, used similarly**
  - ➡ traditional commands `backtrace | continue | print ...`
  - ➡ list all commands `help`

- **fb.python**
  - ➡ bridge between Lua and Python
  - ➡ enables seamless integration between languages
  - ➡ use SciPy with Lua tensors almost as efficiently as with native numpy arrays
  - ➡ on the fly data conversion, use numpy/scipy/matplotlib with Torch tensors
  - ➡ py.exec(code, locals) executes a given Python code string (no return)
  - ➡ py.eval(code, locals) evaluate a given Python code string (and returns a value)

- **Data model**
  - ➡ Lua and Python do not match exactly, need conversion
  - ➡ data transferred between Lua and Python by **value**
  - ➡ tables are copied **deeply**
  - ➡ tensors **share** data but not metadata
  - ➡ opaque references allow user to

# Torch7



- **fb.python**
  - ➡ Example
  - ➡ '[===[' multiline string syntax (python is sensitive to identation)
  - ➡ values converted automatically between Python and Lua
  - ➡ py.eval creates a local Python environment
  - ➡ with value 'a' of type '*Python float*'
  - ➡ return value of type '*Python float*' is converted to '*Lua int*'
  - ➡ Python to Lua and Lua to Python have specific conversion rules
  - ➡ When existing conversion rules are insufficient, opaque references can be used

```
py.exec([=[
import numpy as np
def foo(x):
  return x + 1
]=])

print(py.eval('foo(a) + 10'), {a = 42})  -- prints 53
```

# Torch7

*Torch at Facebook*

- **fb.python**
  - ➡ opaque references encapsulate any Python object
  - ➡ used in place of Lua values to pass arguments to Python
  - ➡ opaque references support function calls, lookup, arithmetic operations
  - ➡ operations on opaque references always return opaque references
  - ➡ so chaining is possible transparently
  - ➡ need py.eval at the end of an operation chain to convert back to Lua

```lua
-- np is opaque reference to Python numpy module
local np = py.import('numpy')

-- t1 is opaque reference to numpy.ndarray
local t1 = np.tri(10).transpose()

-- t2 is t1 converted to torch Tensor
local t2 = py.eval(t1)

local nltk = py.import('nltk')
local tokenized = py.eval(nltk.word_tokenize('Hello world, cats are cool'))
```