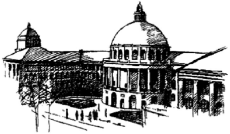


ETH Zürich

Torch - Scientific computing for Lua(JIT)

Octavian Ganea, Aurelien Lucchi

CLS Deep Learning Workshop

July 12th, 2016

Why use Torch ?

- Big community
  - Torch exists since 2000
  - FB, Twitter, IBM, Google, many universities and major labs
- Well documented
- Easy to deploy large-scale machine learning applications
  - Fast optimized backend (C, C++, Blas, CUDA)
  - Easy to run on GPU

Why use Torch ?

- Powerful
  - Easy to construct very complicated neural networks
    - lego-like base modules (network atomic ops) with automatic differentiation
  - Easy to use/modify/write any optimization method, layer, etc
  - Dynamic neural networks, unlike static ones in TF or Theano
    - best suited for research

Based on scripting language Lua

- Very easy to learn (Matlab like)
- Dense code - simple, readable, clean constructs
- Very easy to run on GPU
- Interpreted code, no time lost for compilation
- Easy portable code to other platform (e.g. iPhone)
- Interactive console

Lua basics: tables

- Unique universal data structure
- Can be used as:
  - array / list
  - record/object
  - dictionary / hash table

```
th> a = {1, 3.25, 'ML', nil, true, false, {4,5,6}}   -- (indexed) array
                                                                  [0.0000s]
th> a
{
  1 : 1
  2 : 3.25
  3 : "ML"
  5 : true
  6 : false
  7 :
    {
      1 : 4
      2 : 5
      3 : 6
    }
}
                                                                  [0.0001s]
th> a = {x=0, y=0, label="console"}    -- record/object
                                                                  [0.0000s]
th> a
{
  label : "console"
  y : 0
  x : 0
}
                                                                  [0.0001s]
th> a.label
console
                                                                  [0.0000s]
th> a['label']
console
                                                                  [0.0000s]
th> a.x = 1.0
                                                                  [0.0000s]
th>
                                                                  [0.0000s]
th> a = {["+"] = "add", ["-"] = "sub", ["*"] = "mul", ["/"] = "div"} -- dictionary / map
                                                                  [0.0000s]
th> for key,value in pairs(a) do
..>   print(key .. ' --> ' .. value)
..> end
- --> sub
/ --> div
+ --> add
* --> mul
                                                                  [0.0000s]
th>
                                                                  [0.0000s]
th> f = function() print('hello') end   -- closures
                                                                  [0.0000s]
th> f()
hello
                                                                  [0.0000s]
th> a[f] = 'my function'
                                                                  [0.0000s]
th> a
{
  function: 0x4059ed08 : "my function"
  + : "add"
  - : "sub"
  * : "mul"
  / : "div"
}
                                                                  [0.0001s]
```

Torch tensors

Torch extends Lua's table with a Tensor object

- N-dimensional array
- Different types supported: IntTensor, FloatTensor, DoubleTensor
- torch package offers Matlab's common routines:
  - zeros, ones, eye ...
  - linear algebra stuff
  - slice, view , narrow, fill

```
a = torch.Tensor(5,3) -- construct a 5x3 matrix, uninitialized
```

```
a = torch.rand(5,3)
print(a)
```

```
b=torch.rand(3,4)
```

```
-- matrix-matrix multiplication: syntax 1
a*b
```

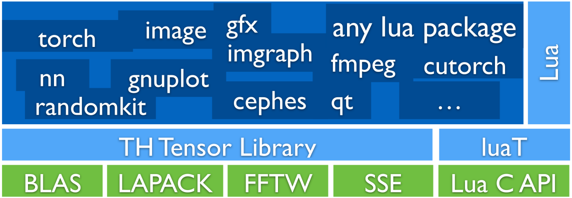
```
-- matrix-matrix multiplication: syntax 2
torch.mm(a,b)
```

```
-- matrix-matrix multiplication: syntax 3
c=torch.Tensor(5,4)
c:mm(a,b) -- store the result of a*b in c
```

- https://github.com/torch/torch7/blob/master/doc/tensor.md

Torch7 packages

- tensor handling
- neural networks
- optimization
- plotting
- statistics
- many more
- install via *luarocks*



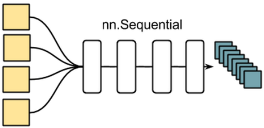
Neural networks in Torch7 - nn package

Lego like base Modules:

- automatic differentiation
- each comes with 3 methods:
  - `output = forward(input)`
  - `gradInput = backward(input, gradOutput)`
    computes the gradients of the module with respect to its own parameters, and its own inputs.
  - `zeroGradParameters()`
- internally keep two states variables: *output* and *gradInput*
- Linear, Convolution, Droput, LookupTable, non linearities, etc

Neural networks in Torch7 - nn package

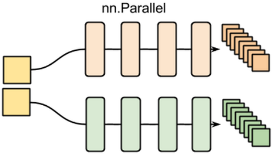
Complex networks combine modules using Containers:




```
th> model = nn.Sequential()
                                                                  [0.0001s]
th> model:add( nn.Linear(10, 25) ) -- 10 input, 25 hidden units
                                                                  [0.0001s]
th> model:add( nn.Tanh() ) -- some hyperbolic tangent transfer function
                                                                  [0.0000s]
th> model:add( nn.Linear(25, 1) ) -- 1 output
                                                                  [0.0001s]
th> print(model:forward(torch.randn(10)))
-0.3648
[torch.DoubleTensor of size 1]
```

Neural networks in Torch7 - nn package

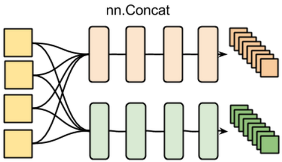
Complex networks combine modules using Containers:




```
th> model = nn.Parallel(2,1)

th> model:add(nn.Linear(10,3))
nn.Parallel {
  input
    |`-> (1): nn.Linear(10 -> 3)
     ... -> output
}

th> model:add(nn.Linear(10,2))
nn.Parallel {
  input
    |`-> (1): nn.Linear(10 -> 3)
    |`-> (2): nn.Linear(10 -> 2)
     ... -> output
}

th> print(model:forward(torch.randn(10,2)))
-0.7266
 0.2804
 1.0107
 0.8189
-0.3478
[torch.DoubleTensor of size 5]
```

Neural networks in Torch7 - nn package

Complex networks combine modules using Containers:




```
th> model=nn.Concat(1);

th> model:add(nn.Linear(5,3))
nn.Concat {
  input
    |`-> (1): nn.Linear(5 -> 3)
     ... -> output
}

th> model:add(nn.Linear(5,7))
nn.Concat {
  input
    |`-> (1): nn.Linear(5 -> 3)
    |`-> (2): nn.Linear(5 -> 7)
     ... -> output
}

th> print(model:forward(torch.randn(5)))
-0.5925
 0.3837
-1.0290
-0.2023
-0.2447
 0.8625
-0.9372
-0.0472
-0.4417
 0.7487
[torch.DoubleTensor of size 10]
```

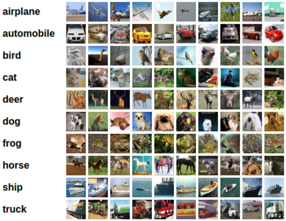
Neural networks in Torch7 - nn package

Complex networks combine modules using Containers:

- For table inputs, use nn.ParallelTable(), nn.ConcatTable(), nn.SplitTable(), nn.JoinTable()
- Arbitrary complex graphs can be made using these containers
  - Alternative: `nngraph` library

CIFAR-10 Example: Multi-layer Conv Net

- 3x32x32 input images
- 10 categories




```
net = nn.Sequential()
net:add(nn.SpatialConvolution(3, 6, 5, 5)) -- 3 input image channels, 6 output channels, 5x5 convolution kernel
net:add(nn.ReLU())                       -- non-linearity
net:add(nn.SpatialMaxPooling(2,2,2,2))     -- A max-pooling operation that looks at 2x2 windows and finds the max.
net:add(nn.SpatialConvolution(6, 16, 5, 5))
net:add(nn.ReLU())                       -- non-linearity
net:add(nn.SpatialMaxPooling(2,2,2,2))
net:add(nn.View(16*5*5))                    -- reshapes from a 3D tensor of 16x5x5 into 1D tensor of 16*5*5
net:add(nn.Linear(16*5*5, 120))             -- fully connected layer (matrix multiplication between input and weights)
net:add(nn.ReLU())                       -- non-linearity
net:add(nn.Linear(120, 84))
net:add(nn.ReLU())                       -- non-linearity
net:add(nn.Linear(84, 10))                   -- 10 is the number of outputs of the network (in this case, 10 digits)
net:add(nn.LogSoftMax())                     -- converts the output to a log-probability. Useful for classification problems
```

```
input = torch.rand(1,32,32) -- pass a random tensor as input to the network
```

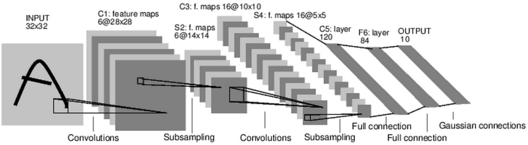
```
output = net:forward(input)
```

```
print(output)
```

```
net:zeroGradParameters() -- zero the internal gradient buffers of the network (will come to this later)
```

```
gradInput = net:backward(input, torch.rand(10))
```

```
print(#gradInput)
```



Source: https://github.com/soumith/cvpr2015/

Loss functions - Criterions

- implemented just like neural network modules
- automatic differentiation
- two functions:
  - `forward(input, target)`
  - `backward(input, target)`
- negative log likelihood, max-margin, binary cross entropy, ...
- On CIFAR example:

```
criterion = nn.ClassNLLCriterion() -- a negative log-likelihood criterion for multi-class classi
fication
criterion:forward(output, 3) -- let's say the groundtruth was class number: 3
gradients = criterion:backward(output, 3)
```

```
gradInput = net:backward(input, gradients)
```

Training

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

```lua
-- 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
```

Source: https://github.com/soumith/cvpr2015/

Training / Optimization

- Optim package - many methods, easy plug-and-play

➡ **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
```

Source: https://github.com/soumith/cvpr2015/

What about GPU ?

Easy! Only 4 operations:

- require ’cunn’
- net = net:cuda()
- criterion = criterion:cuda()
- input = input:cuda()

Everything else stays the same!

Resources and Tutorials

- Main website: `www.torch.ch`
- Torch cheat sheet:
  `https://github.com/torch/torch7/wiki/Cheatsheet`
- Tutorials:
  - Fast th basics:
    `http://rnduja.github.io/tags/torch/`
  - 60-minute blitz:
    `https://github.com/soumith/cvpr2015`
  - MNIST digit classifier (how to properly use minibatches):
    `https://github.com/torch/demos/blob/master/train-a-digit-classifier/train-on-mnist.lua`
  - Supervised, unsupervised, graphical models:
    `https://github.com/torch/tutorials`