Deep Learning with Torch

The good, the bad, the ugly since 2002

Jimmy Ba
jimmy@psi.utoronto.ca
Year 2012 Google Answer:

- Torch7 provides a Matlab-like environment for state-of-the-art machine learning algorithms. It is easy to use and provides a very efficient implementation, thanks to an easy and fast scripting language (LuaJIT) and an underlying C implementation.
What is Torch?

Year 2015 Google Answer:

- Torch is a scientific computing framework with wide support for machine learning algorithms. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation.

- A summary of core features:
  - a powerful N-dimensional array
  - lots of routines for indexing, slicing, transposing, ...
  - amazing interface to C, via LuaJIT
  - linear algebra routines
  - neural network, and energy-based models
  - numeric optimization routines
  - Fast and efficient GPU support
  - Embeddable, with ports to iOS, Android and FPGA backends
YANNL?

• Yet Another Neural Network Library?

Google says:

• The goal of Torch is to have maximum flexibility and speed in building your scientific algorithms while making the process extremely simple. Torch comes with a large ecosystem of community-driven packages in machine learning, computer vision, signal processing, parallel processing, image, video, audio and networking among others, and builds on top of the Lua community.
What is the community up to?

- MNIST, CIFAR-10/100, SVHN, ImageNet loader
- Generate Atari2600 frames and interacting with emulator
- Parallel models across GPUs (data and/or model parallelism)
- Neural Turing Machine (with LSTM cells)
- ...

https://github.com/torch/torch7/wiki/Cheatsheet
Who are maintaining Torch?

• Ronan Collobert - Research Scientist @ Facebook
• Clement Farabet - Senior Software Engineer @ Twitter
• Koray Kavukcuoglu - Research Scientist @ Google DeepMind
• Soumith Chintala - Research Engineer @ Facebook
What you need to know before getting started

• At the heart of Torch is a not so python-like programming language Lua and its JIT compiler LuaJIT

• Lua is a lightweight multi-paradigm scripting language (influenced Julia)
What you need to know before getting started

- LuaJIT is blazing fast

- In fact LuaJIT is so fast you won't have time to think about the painfulness of the Lua language itself

  - Lua is very minimum and bare bone, no power tool included
  
  - Every unspecified variable is a global variable
  
  - It only comes with one container data structure called table

  - 1-based indexing

  - collectgarbage()
What you need to know before getting started

- Learn Lua in 15 mins (http://tylerneylon.com/a/learn-lua/)

- LuaJIT limitations, gotchas and assumptions (Must read! - http://luapower.com/luajit-notes.html)
Demo

• **th**: an iPython like interpreter for Lua with Torch specific features

• train some nets
Learning from example

```
jimmy@psi:/tutorial_torch/sequential_demo$ th

```

- **th**, LuaJIT Torch interpreter, it has nice features to print the content of a data-structure, i.e., table

- can be used both interactively and standalone

```
jimmy@psi:/tutorial_torch/sequential_demo$ th main.lua
```
Learning from example

- **require**, import specific libraries to the environment

- `=`, instead of just enter the variable name
Learning from example

```
Torch7
Scientific computing for Lua.
https://github.com/torch
http://torch.ch

th> require 'torch';

th> A = torch.rand(5,5)

th> A
0.7479 0.8727 0.3354 0.2064 0.0653
0.0805 0.7630 0.2177 0.1621 0.0118
0.5408 0.3780 0.6131 0.2439 0.3217
0.0025 0.4632 0.8872 0.0191 0.5588
0.2191 0.5141 0.5195 0.2846 0.9045
[torch.DoubleTensor of dimension 5x5]
```

- ‘torch’, a numpy-like tensor library
Learning from example

- ‘cutorch’, a CUDA GPU version of the basic Torch library that introduces \texttt{:cuda()} method to all existing and future Torch objects
Define a simple model

- **input**, 1x28x28
- **Conv1**, 3x3x1x128
- **Conv2**, 3x3x128x128
- **max1**, 3x3, stride 2
- **Conv3**, 3x3x128x128
- **Conv4**, 3x3x128x256
- **max2**, 3x3, stride 2
- **fc**, 1024
- **softmax**, 10

A very sequential and computationally linear model
Define a simple model

**Input**: 1x28x28

**Conv1**: 3x3x1x128

**Conv2**: 3x3x128x128

**max1**: 3x3, stride 2

**Conv3**: 3x3x128x128

**Conv4**: 3x3x128x256

**max2**: 3x3, stride 2

**fc**: 1024

**softmax**: 10


```python
require 'nn'

function createModel(opt)
  local model = nn.Sequential()

  model:add(SpatialConvolution(1, 128, 3, 3, 1, 1))
  model:add(nn.ReLU())
  model:add(SpatialConvolution(128, 128, 3, 3, 1, 1))
  model:add(nn.ReLU())
  model:add(nn.SpatialMaxPooling(3, 3, 2, 2))
  model:add(nn.ReLU())
  model:add(SpatialConvolution(128, 256, 3, 3, 1, 1))
  model:add(nn.ReLU())
  model:add(nn.SpatialMaxPooling(3, 3, 2, 2))
  model:add(nn.View(256*3*3))

  model:add(nn.Dropout(0.5))
  model:add(nn.Linear(256*3*3, 1024))
  model:add(nn.ReLU())
  model:add(nn.Dropout(0.5))
  model:add(nn.Linear(1024, 10))
  model:add(nn.LogSoftMax())

  return model
end
```

‘nn’ is your bread and butter
NNet library
with the CUDA version ‘cunn’
A simple loss function

--- Create Model and Data ---

```python
local model = createModel(opt)

-- create negative log likelihood criterion
local loss_func = nn.ClassNLLCriterion()
```

- In `nn`, everything is treated like a layer including loss function and many other operations, e.g. activations
- every `nn` module contains `:.forward()` and `:.backward()` method
A forward-backward pass

```python
data, labels = data_train: updateOutput()
pred = model: forward(data)
cost = loss_func(pred, labels)

cost_grad = loss_func: backward(pred, labels)
model: backward(data, cost_grad)
```

- Use forward to compute model prediction and cost, the temporary activations are stored into each module/layer
- Use backward to compute gradient and the gradients are **accumulated** inside each module if not cleared

```python
-- retrieve parameters and gradients
w, dw = model: getParameters()
print('number of parameters: ' .. w:size(1))
```
Use an optimizer

-- optimizer
local optim_state = {}
optim_state.learningRate = opt.LR
optim_state.momentum = opt.momentum
optim_state.weightDecay = opt.weightDecay
local optim_func = optim.sgd

• ‘optim_state’, is a table acting like a dictionary that stores the internal state of the optimization algorithm, e.g., momentum, Hessian approximation

• ‘optim_func’, takes flatten 1D parameter vector (cost_grad_func, w, optim_state)

-- retrieve parameters and gradients
w, dw = model:getParameters()
print('number of parameters: ' .. w:size(1))
A simple learning algorithm

--- Create Model and Data ---

```python
local model = createModel(opt)
-- create negative log likelihood criterion
local loss_func = nn.ClassNLLCriterion()
-- optimizer
local optim_state = {} 
optim_state.learningRate = opt.LR
optim_state.momentum = opt.momentum 
optim_state.weightDecay = opt.weightDecay
local optim_func = optim.sgd
--- Cudify Everything ---
model:cuda()
loss_func:cuda()
data_train:cuda()
-- retrieve parameters and gradients
w, dw = model:getParameters()
print('number of parameters: ' .. w:size(1))
```

--- Start Training ---

```python
local timer = torch.Timer()
local tic, toc
 tic = timer:time().real
for epoch = 1, opt.nEpochs do
    print('Epoch '.. epoch '.. ')
    optim_func(
        function (param)
        dw:zero()
        data, labels = data_train:updateOutput()
        pred = model:forward(data)
        cost = loss_func(pred, labels)
        cost_grad = loss_func:backward(pred, labels)
        model:backward(data, cost_grad)
        return cost, dw
    end,
    w, optim_state)
end
```
Building a data loader

```lua
require 'torch'
require 'cutorch'

local dataset = torch.class('mnistLoader')

function dataset:__init(opt, dataset)
    local nRows = 28
    local nCols = 28
    local nChn = 1
    self.batchSize = opt.batchSize

    self.data = torch.Tensor(self.batchSize, nChn, nRows, nCols)
    self.labels = torch.Tensor(self.batchSize)
    local data_fname
    if dataset == 'train' then
        data_fname = 'mnist_train.t7'
    elseif dataset == 'valid' then
        data_fname = 'mnist_valid.t7'
    elseif dataset == 'test' then
        data_fname = 'mnist_test.t7'
    end
    self.dataset = torch.load(opt.dataDir .. data_fname)
    self.nData = self.dataset[1]:size(1)
    self.ind = 1
end
```

- Build a class in Lua is almost as easy as python
- `self` is used to denote object variables
Building a data loader

function dataset:updateOutput()
  if (self.ind + self.batchSize - 1) > self.nData then
    self.ind = 1
  end
  self.data:copy(self.dataset
                   [1][{{self.ind, self.ind+self.batchSize-1}, {}}]:viewAs(self.data))
  self.labels:copy(self.dataset[2][{{self.ind, self.ind+self.batchSize-1}}])
  self.ind = self.ind + self.batchSize - 1
  return self.data, self.labels
end

function dataset:cuda()
  self.data = self.data:cuda()
  self.labels = self.labels:cuda()
end

• define some class methods that useful for retrieving data and CUDify
Put everything together like a pro

```lua
require 'torch'
require 'cutorch'
require 'optim'
require 'xlua'
require 'model'
require 'mnist'

local cmd = torch.CmdLine()
  cmd:usage()
  cmd:text('MNIST Training')
  cmd:text('

---------- Training options ----------
' cmd:option('-nEpochs', 10, 'Number of total epochs to run')
               cmd:option('-batchSize', 128, 'mini-batch size (1 = pure stochastic)')

---------- Optimization options ----------

---------- Model options ----------

local opt = cmd:parse(arg)
print(opt)
```

use `torch.CmdLine()` to extract command line arguments
Building an optimizer

• What if you would like to try some new optimization algorithms on your fancy model

```python

ARGS:

- 'opfunc' : a function that takes a single input (X), the point of a evaluation, and returns f(X) and df/dX
- 'x' : the initial point
- 'config' : a table with configuration parameters for the optimizer
  - 'config.learningRate' : learning rate
  - 'config.beta1' : first moment coefficient
  - 'config.beta2' : second moment coefficient
  - 'config.epsilon' : for numerical stability
  - 'config.lambda' : first moment decay
- 'state = {t, m, v}' : a table describing the state of the optimizer; after each call the state is modified

RETURN:
- 'x' : the new x vector
- 'f(x)' : the function, evaluated before the update

]1]```
Building an optimizer

• What if you would like to try some new optimization algorithms on your fancy model

```plaintext

ARGS:

- 'opfunc' : a function that takes a single input (X), the point of a evaluation, and returns f(X) and df/dX
- 'x' : the initial point
- 'config' : a table with configuration parameters for the optimizer
  - 'config.learningRate' : learning rate
  - 'config.beta1' : first moment coefficient
  - 'config.beta2' : second moment coefficient
  - 'config.epsilon' : for numerical stability
  - 'config.lambda' : first moment decay
- 'state = {t, m, v}' : a table describing the state of the optimizer; after each call the state is modified

RETURN:
- `x` : the new x vector
- `f(x)` : the function, evaluated before the update
```
Building an optimizer

```python
function optim.adam(opfunc, x, config, state)
    -- get parameters
    local config = config or {}
    local state = state or config
    local lr = config.learningRate or 2e-6

    local beta1 = config.beta1 or 0.1
    local beta2 = config.beta2 or 0.001
    local epsilon = config.epsilon or 10e-8
    local lambda = config.lambda or 10e-8

    local fx, dfdx = opfunc(x)

    state.t = state.t or 1
    state.m = state.m or torch.Tensor():typeAs(dfdx):resizeAs(dfdx):fill(0)
    state.v = state.v or torch.Tensor():typeAs(dfdx):resizeAs(dfdx):fill(0)

    local bt1 = 1 - (1-beta1)*torch.pow(lambda, state.t-1)
    state.m = torch.add(torch.mul(dfdx, bt1), torch.mul(state.m, 1-bt1))
    state.v = torch.add(torch.mul(torch.pow(dfdx, 2), beta2), torch.mul(state.v, 1-beta2))

    local update = torch.cmul(state.m, torch.pow(
        torch.add(torch.pow(state.v, 2), epsilon),-1))
    update:mul(lr * torch.sqrt(1-torch.pow((1-beta2),2)) * torch.pow(1-torch.pow((1-beta1),2), -1))

    x:add(-update)
    state.t = state.t + 1

    -- return x*, f(x) before optimization
    return x, {fx}, update
end
```

- This 20 lines is all you need to try your beastly optimizer on the latest models.
Define a simple model

input, 1x28x28
Conv1, 3x3x1x128
Conv2, 3x3x128x128
max1, 3x3, stride 2
Conv3, 3x3x128x128
Conv4, 3x3x128x256
max2, 3x3, stride 2
fc, 1024
softmax, 10

A very sequential and computationally linear model
Define a not so simple model

Your latest idea could be really fancy

nngraph’ let you define any arbitrary directed acyclic graph
The Current State of Torch

• Used in FAIR, Google + Deepmind and other institutions

• Facebook recently keeps doing favours(?) for the Torch community
Late 2014, Facebook open sourced a collection of Lua (not just for Torch) utilities, fblualib:

- `fb.debugger` is a full-featured source-level Lua debugger. Does not require Torch.

- ...

- `fb.python` is a bridge between Lua and Python, allowing seamless integration between the two (enabling, for example, using SciPy with Lua tensors almost as efficiently as with native numpy arrays; data between Lua tensors and the corresponding Python counterpart `numpy.ndarray` objects is shared, not copied). Requires Torch.
Demo

- Convert Torch tensor to numpy.ndarray
- Plot from Torch using matplotlib
A simple learning algorithm

--- Create Model and Data ---

```python
local model = createModel(opt)
-- create negative log likelihood criterion
local loss_func = nn.ClassNLLCriterion()
-- optimizer
local optim_state = {}
optim_state.learningRate = opt.LR
optim_state.momentum = opt.momentum
optim_state.weightDecay = opt.weightDecay
local optim_func = optim.sgd
--- Cudify Everything ---
model:cuda()
loss_func:cuda()
data_train:cuda()
-- retrieve parameters and gradients
w, dw = model:getParameters()
print('number of parameters: ' .. w:size(1))
```

--- Start Training ---

```python
local timer = torch.Timer()
local tic, toc
tic = timer:time().real
for epoch = 1, opt.nEpochs do
  print('Epoch .. epoch .. ')
  optim_func()
    function (param)
      dw:zero()
      data, labels = data_train:updateOutput()
      pred = model:forward(data)
      cost = loss_func(pred, labels)
      cost_grad = loss_func:backward(pred, labels)
      model:backward(data, cost_grad)
      return cost, dw
    end,
    w, optim_state)
  end,
  toc = timer:time().real
  print('time: ' .. toc - tic)
end
```

--- important ---
A simple learning algorithm with Python integration

--- Create Model and Data ---

```python
local model = createModel(opt)
-- create negative log likelihood criterion
local loss_func = nn.ClassNLLCriterion()
-- optimizer
local optim_state = {}
optim_state.learningRate = opt.LR
optim_state.momentum = opt.momentum
optim_state.weightDecay = opt.weightDecay
local optim_func = optim.sgd
--- Cudify Everything ---
model:cuda()
loss_func:cuda()
data_train:cuda()
-- retrieve parameters and gradients
w, dw = model:getParameters()
print('number of parameters: ' .. w:size(1))
py = require 'fb.python'
py.exec('import numpy as np; import pylab as plt;')
```

--- Start Training ---

```python
local timer = torch.Timer()
local tic, toc
tic = timer:time().real
for epoch = 1, opt.nEpochs do
    print('Epoch .. epoch .. ')
    optim_func(
        function (param)
            dw:zero()
            data, labels = data_train:updateOutput()
            pred = model:forward(data)
            cost = loss_func(pred, labels)
            cost_grad = loss_func:backward(pred, labels)
            model:backward(data, cost_grad)
            return cost, dw
        end,
        w, optim_state
    py.exec('plt.plot(a);plt.show()', {a = cost})
end
```

--- important ---
Facebook Deep Learning
CUDA Library

• Early 2015, Facebook open sourced yet again a collection of deep learning utilities, fbcunn:

  • Fast spatial and temporal convolution modules that use FFT.

  • Data/Model parallelism with nn.DataParallel and nn.ModelParallel module.

  • Fast LookupTable that is used for Neural Language Models and word embeddings.

  • Hierarchical SoftMax module.

  • …
Demo

- Different convolution backend: Caffe V.S. CuFFT V.S. CuDNN
- Training on 4 GPUs
Data Parallelism

- Different convolution backend: Caffe V.S. CuFFT V.S. CuDNN
- Training on 4 GPUs
A simple learning algorithm with data parallelism
A simple learning algorithm with data parallelism

```python
--- Create Model and Data ---
local model = createModel(opt)
-- create negative log likelihood criterion
local loss_func = nn.ClassNLLCriterion()
-- optimizer
local optim_state = {}
optim_state.learningRate = opt.LR
optim_state.momentum = opt.momentum
optim_state.weightDecay = opt.weightDecay
local optim_func = optim.sgd

local ref_model = model
require 'fbcunn'
require 'fbnn'
local parallel_container = nn.DataParallel(1)
parallel_container:add(ref_model)
for i=2, opt.nGPU do
    parallel_container:add(ref_model:clone())
end
model = parallel_container
optim_state.learningRate = optim_state.learningRate * opt.n
local optim = nn.Optim(model, optim_state)
--- Cudify Everything ---
model:cuda()
loss_func:cuda()
data_train:cuda()
data_valid:cuda()
-- retrieve parameters and gradients
w, dw = ref_model:getParameters()
print('number of parameters: ' .. w:size(1))
```

```python
--- Start Training ---
local timer = torch.Timer()
local tic, toc
tic = timer:time().real
for epoch = 1, opt.nEpochs do
    print('Epoch ' .. epoch .. ' :')
    optim_func()
    function (param)
        dw:zero()
        data, labels = data_train:updateOutput()
pred = model:forward(data)
cost = loss_func(pred, labels)
cost_grad = loss_func:backward(pred, labels)
model:backward(data, cost_grad)
return cost, dw
end,
w, optim_state)
```
The Future State of Torch

- Hope the hype-train will be around for awhile
- Hope someone release a Torch implementation/wrapper of the parameter server (maybe Google will open source TensorFlow?)
“No winter lasts forever; no spring skips its turn.”

–Hal Borland