Deep Learning with Torch7

By Abhishek Aggarwal

Co-founder, CTO

True AI

Table of Contents

Introduction to Torch

Neural networks in Torch

Writing custom neural network module

Torch7

- A machine learning programming framework built on Lua(JIT), C++, and CUDA
- Open-source, BSD license
- Used by AI labs in industries and academic labs
  - Facebook AI research
  - Google DeepMind (discontinued after release of TensorFlow recently)
  - Twitter Cortex
- Installation
  - **$ git clone https://github.com/torch/distro.git ~/torch --recursive**
  - **$ cd ~/torch; bash install-deps; ./install.sh**
- Run Torch7
  - **$ th main.lua**

Lua Basics

- Lua is interpreted language like Python and MATLAB
- Learn lua in 15 mins - http://tylerneylon.com/a/learn-lua/
- Variables

```lua
num = 42  -- All numbers are doubles.
s = 'walternate'  -- Immutable strings like Python.
t = "double-quotes are also fine"
```

- Loops

```lua
while num < 50 do
  num = num + 1
end
fredSum = 0
for j = 100, 1, -1 do fredSum = fredSum + j end
```

Lua Basics (continued)

- Conditional statements

```
if num > 40 then
  print('over 40')
elseif s ~= 'walternate' then  -- ~= is not equals.
  print('Winter is coming, ' .. line)
end
aBoolValue = false
ans = aBoolValue and 'yes' or 'no'  --> 'no'
```

- Functions

```
function fib(n)
  if n < 2 then return 1 end
  return fib(n - 2) + fib(n - 1)
end
```

Lua Basics (continued)

- scope
  - Unless specified with the keyword ***local***, every variable or function is global by default
  - `local g; g  = function (x) return math.sin(x) end`
- Tables
  - Tables are the only compound data structure, tables are associative arrays
  - Dictionary

```
t = {key1 = 'value1', key2 = false}
print(t.key1)  -- Prints 'value1'.
t.newKey = {}  -- Adds a new key/value pair.
t.key2 = nil   -- Removes key2 from the table.
```

  - Iteration of dictionary

```
for key, val in pairs(u) do
  print(key, val)
end
```

Lua Basics (continued)

- Tables (continued)

```
v = {'value1', 'value2', 1.21, 'gigawatts'}
for i = 1, #v do  -- #v is the size of v for lists.
  print(v[i])  -- Indices start at 1 !! SO CRAZY!
end
```

- Table as Class

```
Dog = {}
function Dog:new()
  newObj = {sound = 'woof'}
  self.__index = self
  return setmetatable(newObj, self)
end
function Dog:makeSound()
  print('I say ' .. self.sound)
end
```

Torch Core Package

- **torch**: tensors, class factory, serialization, BLAS
- **nn**: neural networks, Modules and Criterions
- **optim**: SGD, LBFGS and other optimization functions
- **gnuplot**: ploting and data visualization
- **paths**: functions related paths, and file system
- **image**: save, load, crop, etc.
- **trepl**: the torch LuaJIT interpreter (like Ipython)
- **cwrap**: wrapping C/CUDA functions in lua
- Much more at https://github.com/torch
- Cheatsheet: https://github.com/torch/torch7/wiki/Cheatsheet

Installation of packages

- Luarocks is the package manager supported by torch

```
$ luarocks install image     # an image library for Torch7
$ luarocks install nnx       # lots of extra neural-net modules
$ luarocks install camera    # a camera interface for Linux/MacOS
$ luarocks install ffmpeg    # a video decoder for most formats
```

Tensor class

- Tensor is the most important class for math operations.
- Tensor object is a serializable, multidimensional matrix
- Built on top of Storage class, Tensor defines particular way of viewing a storage.
- ```
  x = torch.Tensor(5,6)
  x:fill(0) -- fill with zeros
  x:uniform(-1,1) -- fill with random values between -1 and 1
  x:nDimension() -- return 2
  x:nElements() -- returns 30
  x:size() -- returns torch.LongTensor({5,6})
  x:size(2) -- returns 6
  y = x[2][4]
  ```
- Documention: https://github.com/torch/torch7/blob/master/doc/tensor.md
- Documention: https://github.com/torch/torch7/blob/master/doc/maths.md
  ```

Introduction to Torch

Neural networks in Torch

Writing custom neural network module

nn package

- Implements most neural network related layers and modules.
- *cunn* is the cuda counterpart of *nn*.
- Documentation: https://github.com/torch/nn
- Implemented ~140 kinds of network layer/criterion (April 2016)

```
mlp = nn.Sequential()
mlp:add( nn.Linear(10, 25) ) -- 10 input, 25 hidden units
mlp:add( nn.Tanh() ) -- some hyperbolic tangent transfer function
mlp:add( nn.Linear(25, 1) ) -- 1 output

print(mlp:forward(torch.randn(10)))
```

Layers and Criterions

- Layers
  - Convolution: nn.SpatialConvolution, nn.SpatialFullConvolution, etc.
  - Activations: nn.Sigmoid, nn.Tanh, nn.Sigmoid, etc.
  - Normalization: nn.BatchNormalization, nn.SpatialBatchNormalization, etc.
- Criterions (aka loss functions)
  - Classification: nn.ClassNLLCriterion, nn.BCECriterion
  - Regression: nn.MSECriterion, nn.WeightedMSECriterion
  - Multi-task: nn.MultiCriterion

Example MLP

- Build MLP

```
require "nn"
mlp = nn.Sequential();  -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20; -- parameters
mlp:add(nn.Linear(inputs, HUs))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(HUs, outputs))
criterion = nn.MSECriterion()
```

- Forward and Backward pass

```
-- feed it to the neural network and the criterion
criterion:forward(mlp:forward(input), output)
-- train over this example in 3 steps
-- (1) zero the accumulation of the gradients
mlp:zeroGradParameters()
-- (2) accumulate gradients
mlp:backward(input, criterion:backward(mlp.output, output))
-- (3) update parameters with a 0.01 learning rate
mlp:updateParameters(0.01)
```

Using optim package

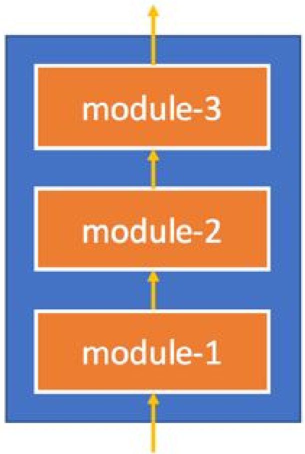
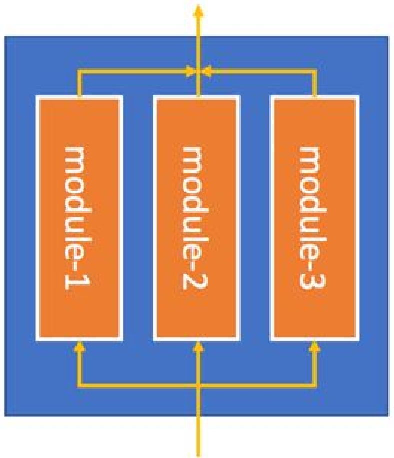
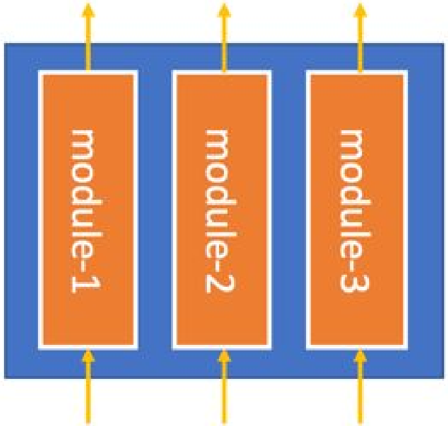
```lua
local optimState = {learningRate=0.01}
require 'optim'
local params, gradParams = model:getParameters()
require 'optim'

for epoch=1,50 do
local function feval(params)
  gradParams:zero()

  local outputs = model:forward(batchInputs)
  local loss = criterion:forward(outputs, batchLabels)
  local dloss_doutput = criterion:backward(outputs, batchLabels)
  model:backward(batchInputs, dloss_doutput)

  return loss,gradParams
end
optim.sgd(feval, params, optimState)
end
```

nn containers

nn.Sequential	nn.Concat/nn.ConcatTable	nn.Parallel/nn.ParallelTable
		

- Difficult to make arbitrary graphs with containers

http://mclab.eic.hust.edu.cn/valse2016/dl/%E7%9F%B3%E8%91%86%E5%85%89.pdf

nngraph

- *nn* provides great set of layers and modules for use in DL.
- However, building arbitrary complex networks using containers is difficult.
- *nngraph* is a wrapper around *nn*.
- Documentation: https://github.com/torch/nngraph
- Every module can be made node (called *gnode*) in the graph
    - `input = nn.Identity()()`
    - `L1 = nn.Tanh()(nn.Linear(10, 20)(input))`
- Works with all nn modules and criterions.
- Allows even printing of forward and backward graph.
- Allows annotation of *gnodes* which helps in debugging.

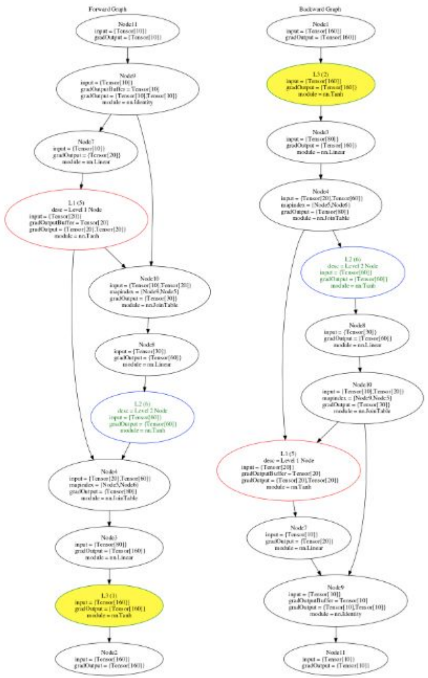
nngraph example

```
input = nn.Identity()()
L1 = nn.Tanh()(nn.Linear(10, 20)(input)):annotate{
   name = 'L1', description = 'Level 1 Node',
   graphAttributes = {color = 'red'}
}
L2 = nn.Tanh()(nn.Linear(30, 60)(nn.JoinTable(1)({input, L1}))):annotate{
   name = 'L2', description = 'Level 2 Node',
   graphAttributes = {color = 'blue', fontcolor = 'green'}
}
L3 = nn.Tanh()(nn.Linear(80, 160)(nn.JoinTable(1)({L1, L2}))):annotate{
   name = 'L3', descrption = 'Level 3 Node',
   graphAttributes = {color = 'green',
   style = 'filled', fillcolor = 'yellow'}
}

g = nn.gModule({input},{L3})

indata = torch.rand(10)
gdata = torch.rand(160)
g:forward(indata)
g:backward(indata, gdata)

graph.dot(g.fg, 'Forward Graph', '/tmp/fg')
graph.dot(g.bg, 'Backward Graph', '/tmp/bg')
```



Introduction to Torch

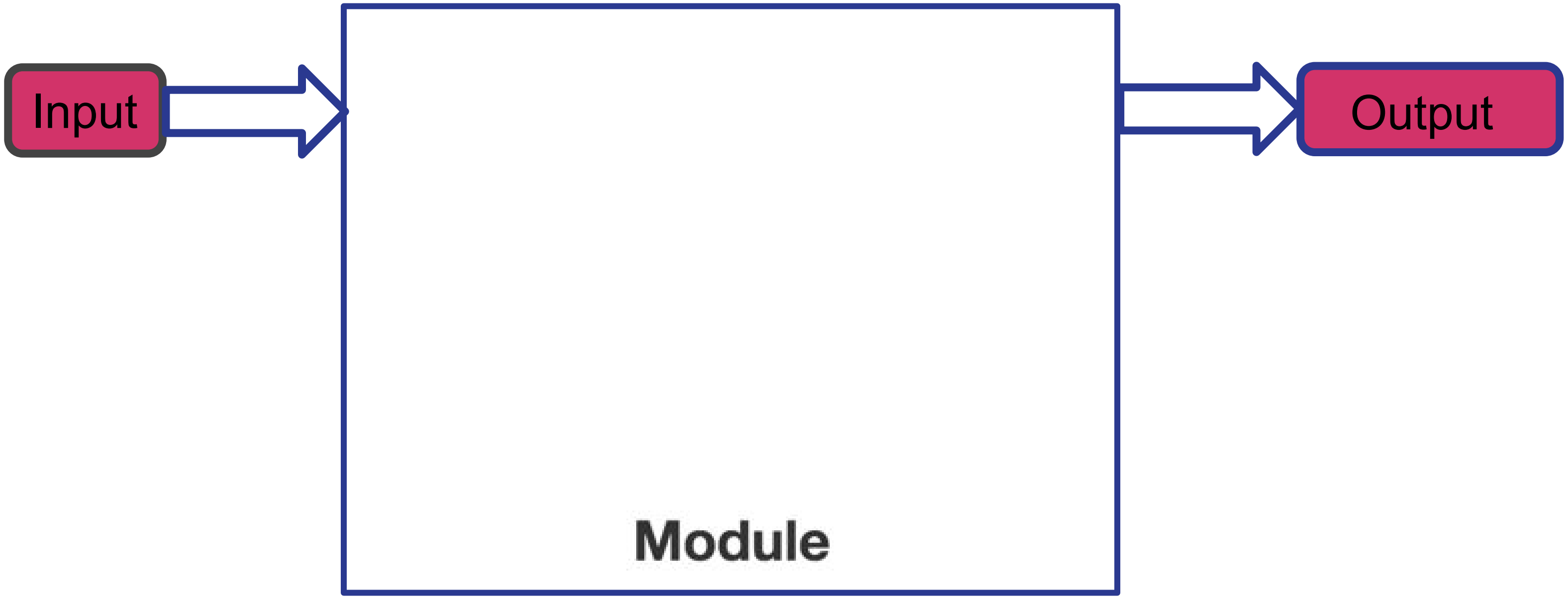
Neural networks in Torch

Writing custom neural network module

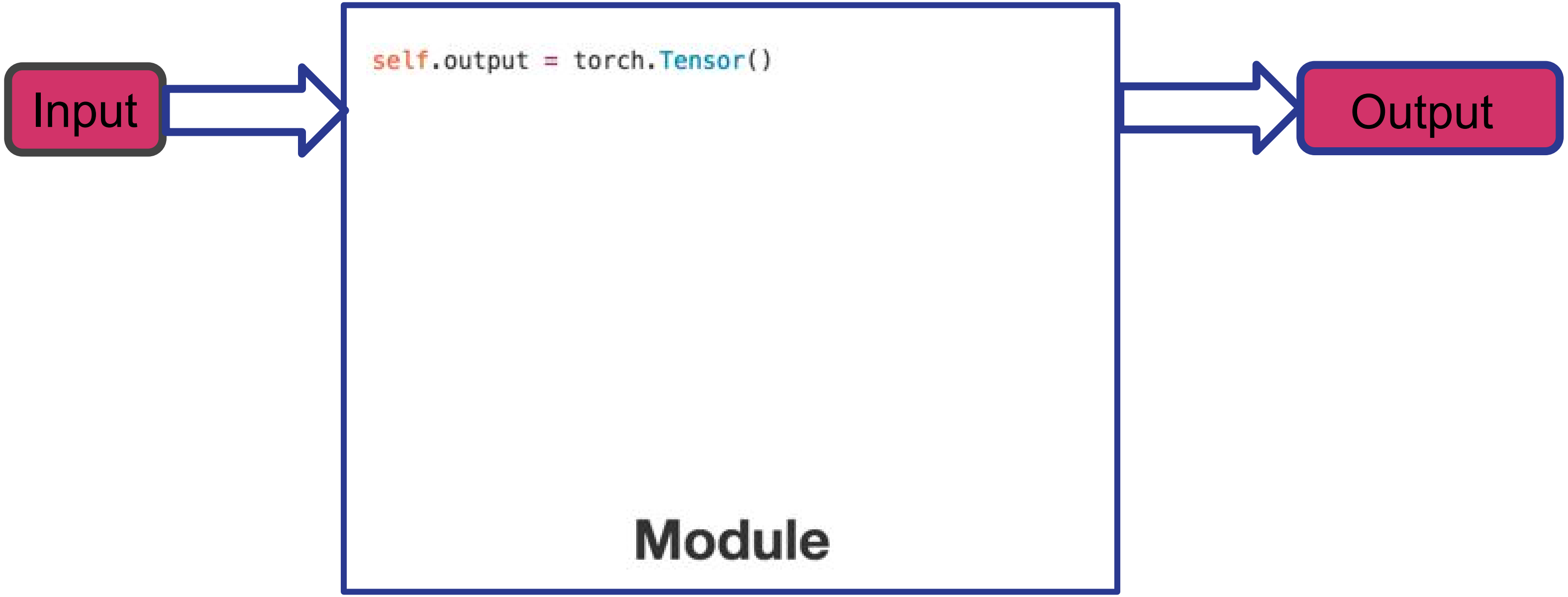
nn.Module API

- *Module* is an abstract class which defines fundamental methods necessary for a training a neural network.
- Everything in nn is inherited from *nn.Module*
- This includes layers, criterions and containers.
- *nn.Module* defines an API required for doing forward and backward pass.
- Thus containers (such *nn.Sequential*) are *nn.Modules* that contain other *nn.Modules* (such as layers and criterions.)
- At high lever, all nn.Module must implement
  - **forward** - takes input and returns output.
  - **backward** - takes input and updates gradients with respect to input (and updates gradients of weights, if any)
- Documentation: https://github.com/torch/nn/blob/master/Module.lua

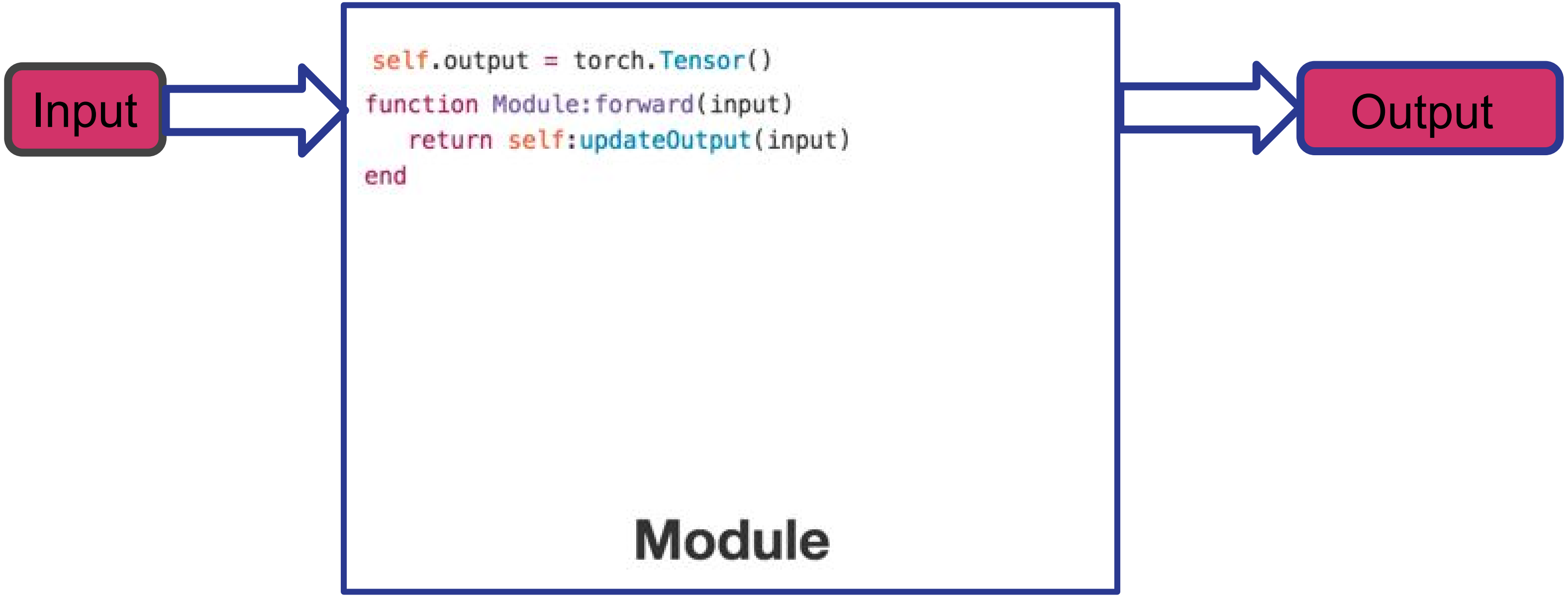
nn.Module



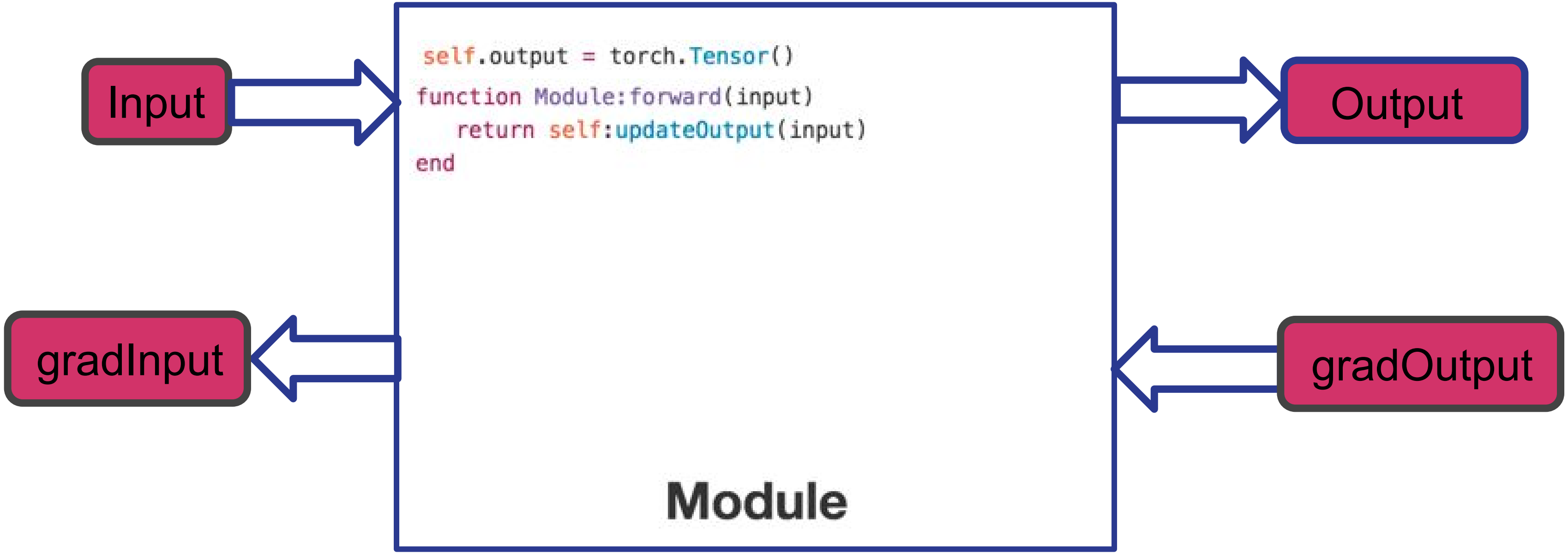
nn.Module



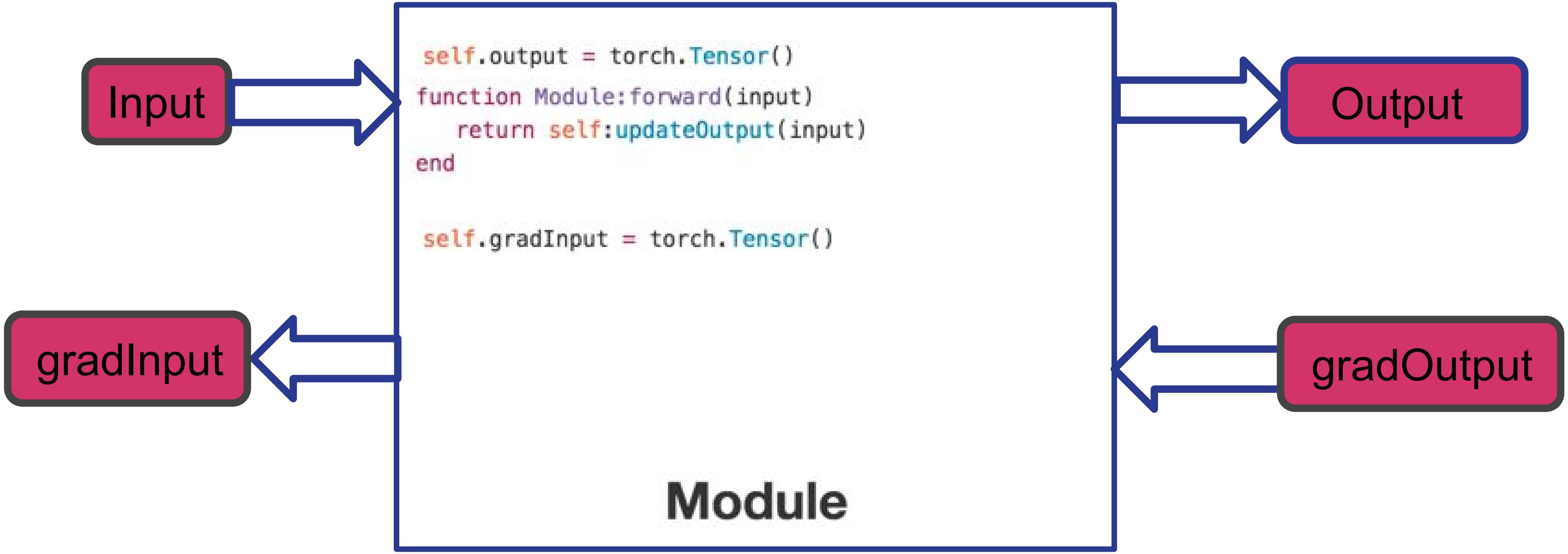
nn.Module


Input
self.output = torch.Tensor()
function Module:forward(input)
return self:updateOutput(input)
end
Module
Output

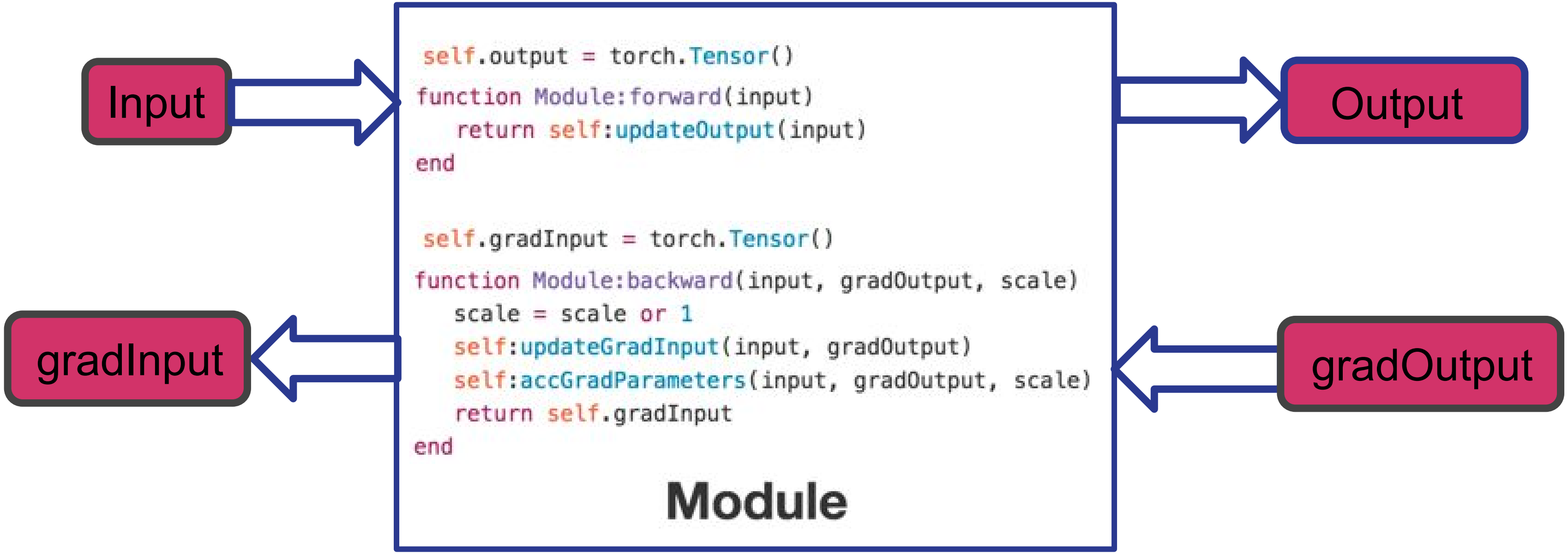
nn.Module


Input
self.output = torch.Tensor()
function Module:forward(input)
return self:updateOutput(input)
end
Output
gradInput
gradOutput
Module

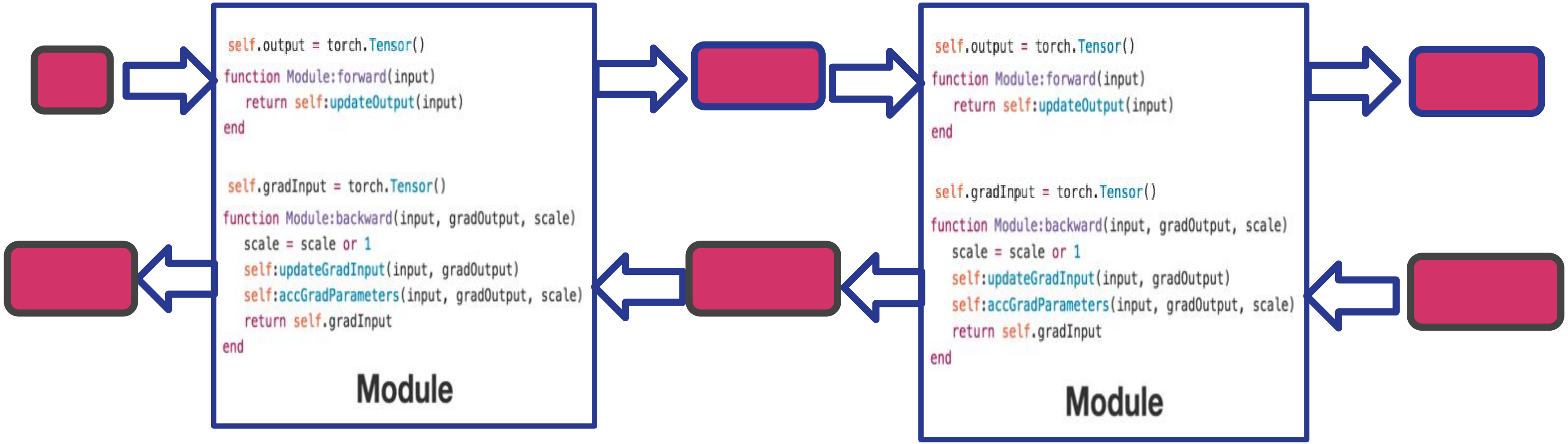
nn.Module


Input
self.output = torch.Tensor()
function Module:forward(input)
return self:updateOutput(input)
end
Output
self.gradInput = torch.Tensor()
gradInput
gradOutput
Module

nn.Module


Input
Output
self.output = torch.Tensor()
function Module:forward(input)
return self:updateOutput(input)
end
self.gradInput = torch.Tensor()
function Module:backward(input, gradOutput, scale)
scale = scale or 1
self:updateGradInput(input, gradOutput)
self:accGradParameters(input, gradOutput, scale)
return self.gradInput
end
Module
gradInput
gradOutput

Cascade of nn.Modules



ReQLu

- We are gc $z = \begin{cases} x^2 + x & \text{if} \quad x > 0 \\ 0 & \text{otherwise} \end{cases}$ er function, called Rectified Quadratic Linear Unit, or ReQLu.

- Note that this Module has no parameters.
- Implementation:
  - Inherit ReQLu from nn.Module
  - Override updateOutput
  - Override updateGradInput

ReQLu: Forward Pass

```lua
local ReQLu, parent = torch.class('ReQLu', 'nn.Module')
-- transfer function f(x) = x^2 + x if x > 0 else 0

function ReQLu:__init()
  parent.__init(self)
  -- the two states module needs to maintain are outputs in forward and backward pass
  self.output = torch.Tensor()
  self.gradInput = torch.Tensor()
end

-- define input to output mapping (forward pass)
function ReQLu:updateOutput(input)
  -- make sure the input is two dimensional ( batch_size x input_dimension)
  assert(input:nDimension() == 2)

  -- calculate output without mask
  self.output:resizeAs(input):copy(input)
  self.output:cmul(input):add(input)

  -- apply mask
  local mask = input:gt(0):typeAs(input)
  self.output:cmul(mask)
  return self.output
end
```

$$z = \begin{cases} x^2 + x & \text{if} \quad x > 0 \\ 0 & \text{otherwise} \end{cases}$$

ReQLu: Backward Pass

- Define backward pass

$$\frac{\partial z}{\partial x} = \left\{ \begin{array}{ll} 2x+1 & \text{if} \quad x>0 \\ 0 & \text{otherwise} \end{array} \right\}$$

$$gradInput = \frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial z} . \frac{\partial z}{\partial x} = gradOutput . \frac{\partial z}{\partial x}$$

```
-- define gradOutput to gradInput mapping (backward pass)
function ReQLu:updateGradInput(input, gradOutput)
  self.gradInput:resizeAs(input)

  -- calculate dz/dx (without masking)
  self.gradInput:copy(2*input):add(torch.ones(input:size()))

  -- apply mask
  local mask = input:gt(0):typeAs(input)  -- convert from ByteTensor to Tensor
  self.gradInput:cmul(mask)

  -- calculate gradInput by multiplying it with gradOutput
  self.gradInput:cmul(gradOutput)
  return self.gradInput
end
```

ReQLuScaled

- We are going to create a simple transfer function, called Rectified Quadratic Linear Unit Scaled, or ReQLu Scaled.

$$z = \begin{cases} ax^2 + bx & if \quad x > 0 \\ 0 & otherwise \end{cases}$$

- Two parameters: *a* and *b*
- Implementation:
  - Inherit ReQLu from nn.Module
  - Override *updateOutput*
  - Override *updateGradInput*
  - Override *accGradParameters*

ReQLuScaled: Implementation

```lua
local ReQLuScaled, parent = torch.class('ReQLuScaled', 'nn.Module')
-- transfer function f(x) = a*x^2 + b*x if x > 0 else 0

function ReQLuScaled:__init()
  parent.__init(self)
  -- the two states module needs to maintain are outputs in forward and backward pass
  self.output = torch.Tensor()
  self.gradInput = torch.Tensor()

  -- declare weights
  self.a = torch.Tensor(1)
  self.b = torch.Tensor(1)

  -- declare grad of weights
  self.grad_a = torch.Tensor(1)
  self.grad_b = torch.Tensor(1)
end
```

$$z = \begin{cases} ax^2 + bx & \text{if} \quad x > 0 \\ 0 & \text{otherwise} \end{cases}$$

ReQLuScaled: Forward Pass

```
-- define input to output mapping (forward pass)
function ReQLuScaled:updateOutput(input)
  -- make sure the input is two dimensional ( batch_size x input_dimension)
  assert(input:nDimension() == 2)

  -- calculate output without mask
  self.output:resizeAs(input):copy(input)
  self.output:cmul(self.a[1] * input)
  self.output:add(self.b[1] * input)

  -- apply mask
  local mask = input:gt(0):typeAs(input)
  self.output:cmul(mask)
  return self.output
end
```

$$z = \begin{cases} ax^2 + bx & \textit{if} \quad x > 0 \\ 0 & \textit{otherwise} \end{cases}$$

ReQLuScaled: Backward Pass I

$$\frac{\partial z}{\partial x} = \begin{cases} 2ax + b & \text{if} \quad x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$gradInput = \frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial z} . \frac{\partial z}{\partial x} = gradOutput . \frac{\partial z}{\partial x}$$

```
-- define gradOutput to gradInput mapping (backward pass)
function ReQLuScaled:updateGradInput(input, gradOutput)
  self.gradInput:resizeAs(input)

  -- calculate dz/dx (without masking)
  self.gradInput:copy(2*self.a[1] * input):add(self.b[1] * torch.ones(input:size()))

  -- apply mask
  local mask = input:gt(0):typeAs(input)  -- convert from ByteTensor to Tensor
  self.gradInput:cmul(mask)

  -- calculate gradInput by multiplying it with gradOutput
  self.gradInput:cmul(gradOutput)
  return self.gradInput
end
```

ReQLuScaled: Backward Pass II

```lua
function ReQLuScaled:accGradParameters(input, gradOutput)
  -- calculate gradient wrt output
  local grad_a = torch.cmul(input, input)
  local grad_b = input

  -- apply mask
  local mask = input:gt(0):typeAs(input)  -- convert from ByteTensor to Tensor
  grad_a:cmul(mask)
  grad_b:cmul(mask)

  -- multiply by gradOutput
  grad_a:cmul(gradOutput)
  grad_b:cmul(gradOutput)

  -- update gradients
  self.grad_a = torch.sum(grad_a)
  self.grad_b = torch.sum(grad_b)
end

-- override the parameters function
function ReQLuScaled:parameters()
  local weights = {self.a, self.b}
  local gradWeights =  {self.grad_a, self.grad_b}
  return weights, gradWeights
end
```

$$\frac{\partial z}{\partial a} = \left\{ \begin{array}{lll} x^2 & if & x > 0 \\ 0 & otherwise & \end{array} \right\}$$

$$\frac{\partial loss}{\partial a} = \frac{\partial loss}{\partial z} . \frac{\partial z}{\partial a} = gradOutput. \frac{\partial z}{\partial a}$$

$$\frac{\partial z}{\partial b} = \left\{ \begin{array}{lll} x & if & x > 0 \\ 0 & otherwise & \end{array} \right\}$$

$$\frac{\partial loss}{\partial b} = \frac{\partial loss}{\partial z} . \frac{\partial z}{\partial b} = gradOutput. \frac{\partial z}{\partial b}$$

Usage and Code

- The modules that we created can be used just like other modules

```
model = nn.Sequential()
model:add(nn.Reshape(28*28))
model:add(nn.Linear(28*28, 225))
model:add(ReQLuScaled())
model:add(nn.Linear(225, 144))
model:add(nn.Tanh())
model:add(nn.Linear(144, 10))
model:add(nn.LogSoftMax())
```

- Code that uses these modules to classify MNIST digits
  - https://github.com/abhitopia/TorchTalkDLSummerCampLondon

True AI IS HIRING

SENIOR SOFTWARE ENGINEER

As a Senior Software Engineer, you will be working closely with our researchers and leading the development of technical infrastructure at True AI from ground up, building highly scalable and real time system architecture that blends well with underlying deep learning algorithms

DEEP LEARNING RESEARCH ENGINEER

We need a Deep Learning Research Engineer who is passionate about taking AI to the next level, and who is interested in building the company alongside the founders.

FULL STACK DEVELOPER

We need a Full Stack Developer who is passionate about taking AI to the next level, and who is interested in building the company alongside the founders. You will be playing a key role in the development of our main web application and browser based plugins, as well as integration on backend.

+ INTERNSHIPS