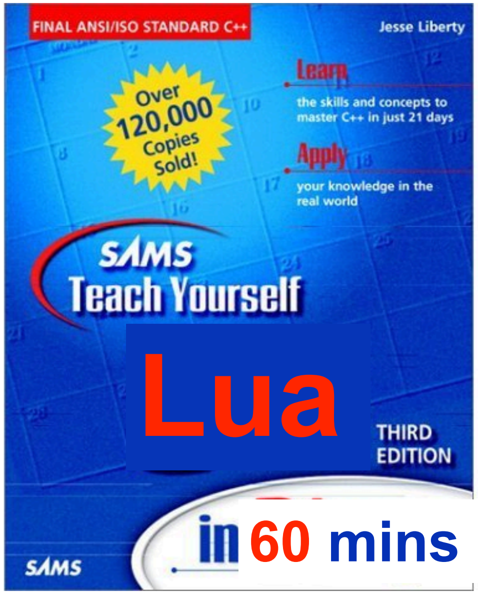
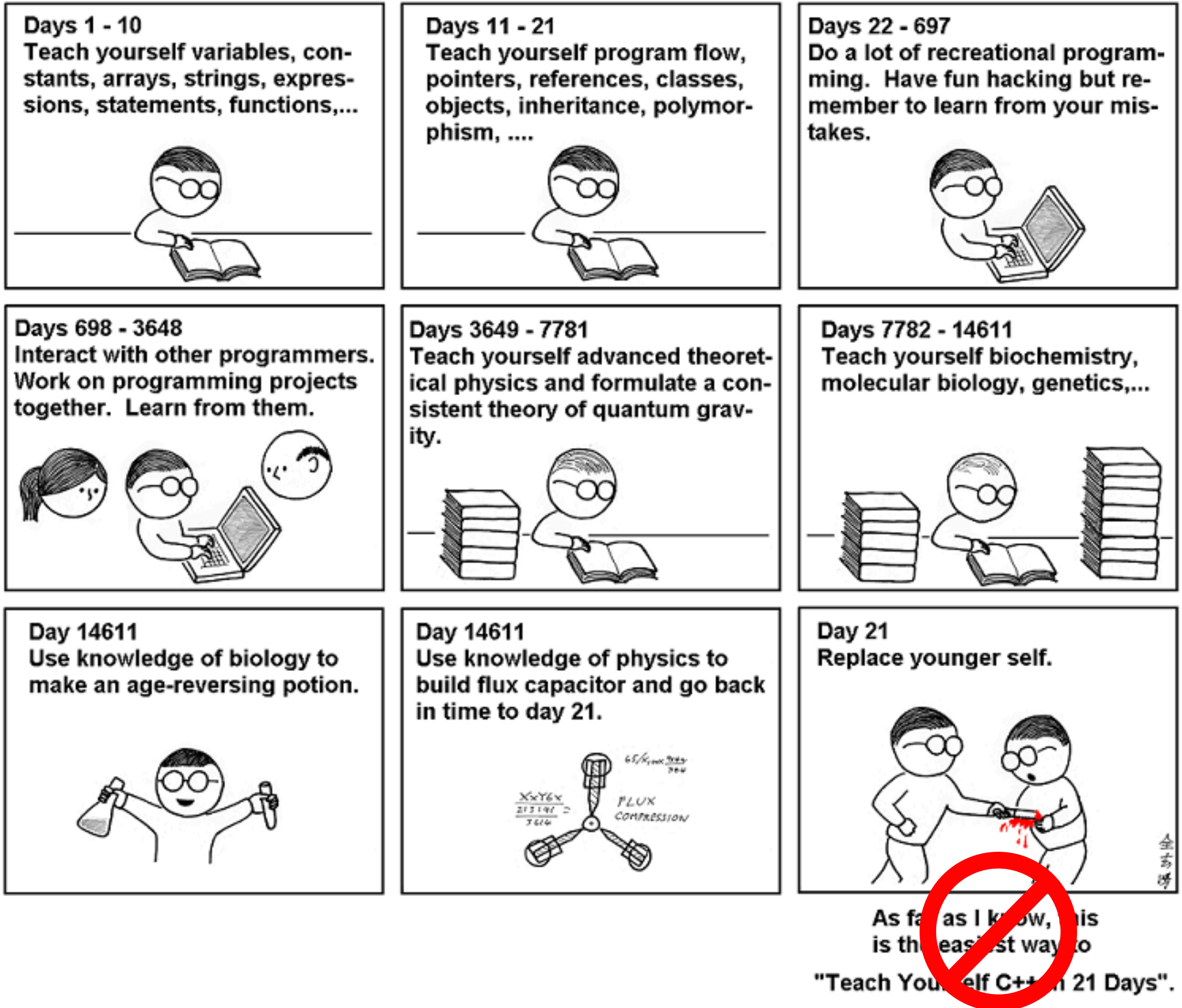
Learn Torch
in 60 mins

Kai Zhao   James Cross   Liang Huang

Dept. EECS, Oregon State University




Days 1 - 10
Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...
Days 11 - 21
Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism, ...
Days 22 - 697
Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.
Days 698 - 3648
Interact with other programmers. Work on programming projects together. Learn from them.
Days 3649 - 7781
Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.
Days 7782 - 14611
Teach yourself biochemistry, molecular biology, genetics,...
Day 14611
Use knowledge of biology to make an age-reversing potion.
Day 14611
Use knowledge of physics to build flux capacitor and go back in time to day 21.
FLUX
COMPRESSION
Day 21
Replace younger self.
As fa as I k ow, is
is th eas st way o
"Teach You elf C++ n 21 Days".

Toolkits in the Thriving Deep Learning Community

Schedule of the Tutorial

Torch (Today)

Provides high-level abstractions as well as low-level access

DyNet (Dec. 2nd)

Specialized for dynamically changing networks

TensorFlow (Dec. 9th)

Industrial level toolkit

Support massive GPU clusters

In Today's Tutorial

- Basics of Lua Language
- Basic Tensor Operations in Torch
- Building a Simple Network
- Training the Simple Network
- Building a Recurrent Neural Network

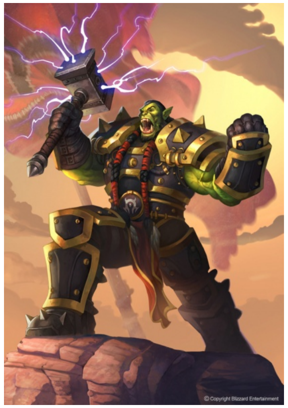
In Today's Tutorial

- Basics of Lua Language
- Basic Tensor Operations in Torch
- Building a Simple Network
- Training the Simple Network
- Building a Recurrent Neural Network

Lua: Simple & Fast

- Designed as an embedded language. (e.g., in *World of Warcraft*)
  - Extremely simple grammar:
    - Atomic data types: float, string, boolean
    - Everything else is a hash table
    - First-class functions
  - Surprisingly fast as a script language:
    - LuaJIT is very well developed
    - Speed comparable to Java
  - Easy interaction with C/C++
    - Simple interface
    - Little overhead

Learn Lua for the Horde!



Lua: Data Types & Flow Control

- Variables, Flow Control, and Functions
  - Same as most imperative languages
  - Every undefined variable is by default *nil*
- Only one compound data type: *table*

```
-- Dict literals have string keys by default:
t = {key1 = 'value1', key2 = false}

-- String keys can use js-like dot notation:
print(t.key1)  -- Prints 'value1'.
t.newKey = {}  -- Adds a new key/value pair.
t.key2 = nil   -- Removes key2 from the table.

-- Literal notation for any (non-nil) value as key:
u = {['@!#'] = 'qbert', [{}] = 1729, [6.28] = 'tau'}
print(u[6.28])  -- prints "tau"

-- Key matching is basically by value for numbers and strings,
but by identity
-- for tables.
a = u['@!#']  -- Now a = 'qbert'.
b = u[{}]     -- We might expect 1729, but it's nil
```

https://learnxinyminutes.com/docs/lua/

Lua: Data Types & Flow Control

- Variables, Flow Control, and Functions
  - Same as most imperative languages
  - Every undefined variable is by default *nil*
- Only one compound data type: *table*
  - List/Array: table w/ consecutive int. keys (index from 1)

```
-- List literals implicitly set up int keys:
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
-- A 'list' is not a real type. v is just a table with
-- consecutive integer keys, treated as a list.
```

  - Iterate through table

```
for key, val in pairs(u) do  -- Table iteration.
  print(key, val)
end
```

https://learnxinyminutes.com/docs/lua/

Lua: OOP

- Class is just another table

```lua
Dog = {}                                          -- 1.

function Dog:new()                                -- 2.
  local newObj = {sound = 'woof'}                 -- 3.
  self.__index = self                             -- 4.
  return setmetatable(newObj, self)               -- 5.
end

function Dog:makeSound()                          -- 6.
  print('I say ' .. self.sound)
end

mrDog = Dog:new()                                 -- 7.
mrDog:makeSound()  -- 'I say woof'                -- 8.
```

  - Definition member function
    function tablename:fn(…) … **end**
    equals to
    function tablename.fn(self, …) … **end**

In Today’s Tutorial

- Basics of Lua Language
- Basic Tensor Operations in Torch
- Building a Simple Network
- Training the Simple Network
- Building a Recurrent Neural Network

Torch: Tensors

- Basic operands: Tensors
  - Types: ByteTensor, CharTensor, ShortTensor, IntTensor, LongTensor, FloatTenor, DoubleTensor

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

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

- Simple Operators
  - E.g., multiplication

```
-- 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 in c
```

Torch: Other Operations

- Constructors: torch.ones() torch.zeros()
- Element-wise Operators: abs(), pow()
- Column-wise Operators: sum(), max()
- Matrix-wise Operators: trace(), norm()

```
torch.cat(torch.ones(3), torch.zeros(2))
1
1
1
0
0
[torch.DoubleTensor of size 5]
```

```
torch.cat(torch.ones(3, 2), torch.zeros(2, 2), 1)
1  1
1  1
1  1
0  0
0  0
[torch.DoubleTensor of size 5]
```

In Today's Tutorial

- Basics of Lua Language
- Basic Tensor Operations in Torch
- Building a Simple Network
- Training the Simple Network
- Building a Recurrent Neural Network

Neural Networks

- Package ‘nn’
  - Basic neural network modules
  - Construction methods
- Linear module as an example

```
require 'nn';
lin = nn.Linear(5, 3)
```

  - Just another table

```
lin
nn.Linear(5 -> 3)
{
  gradBias : DoubleTensor - size: 3
  weight : DoubleTensor - size: 3x5
  _type : torch.DoubleTensor
  output : DoubleTensor - empty
  gradInput : DoubleTensor - empty
  bias : DoubleTensor - size: 3
  gradWeight : DoubleTensor - size: 3x5
}
```

output = weight * X + bias

Neural Networks

- Forward/Backward already defined for modules
    - Forward

```
y = lin:forward(x)
print(y)
```

    - Backward

```
lin:backward(x, grad)
```

        - Call :zeroGradParameters() before backward

    - Now we can manually do gradient descent

```
lin.weight:add(0.1*lin.gradWeight)
lin.bias:add(0.1*lin.gradBias)
```

I believe in graduate student descent.

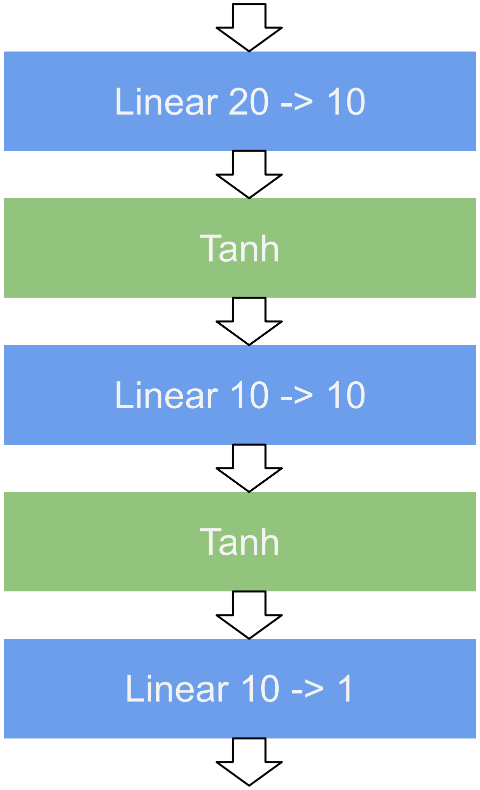
— David McAllester

Neural Networks

- Composing more complicated networks
  - Use package ‘nn’

```
net = nn.Sequential();
net:add(nn.Linear(20, 10));
net:add(nn.Tanh());
net:add(nn.Linear(10, 10));
net:add(nn.Tanh());
net:add(nn.Linear(10, 1));
```

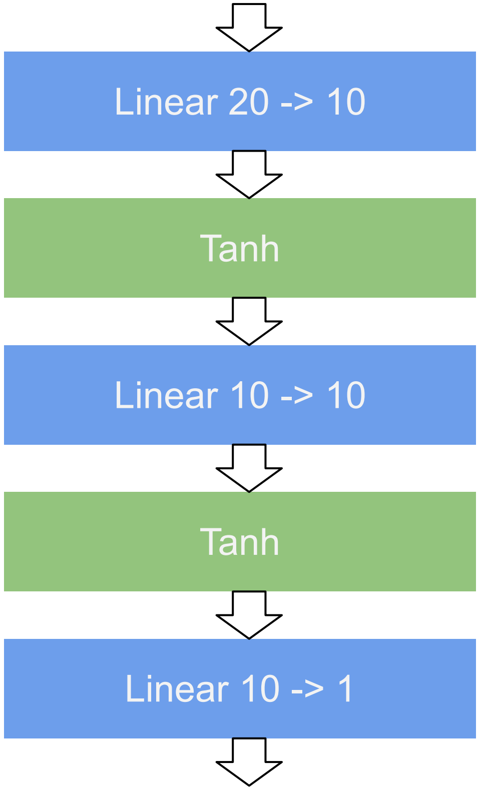
```
x = torch.rand(20)
y1 = net:forward(x)
print(y1)
-0.2648
[torch.DoubleTensor of size 1]
```



Neural Networks

- Composing more complicated networks
  - Use package ‘nngraph’

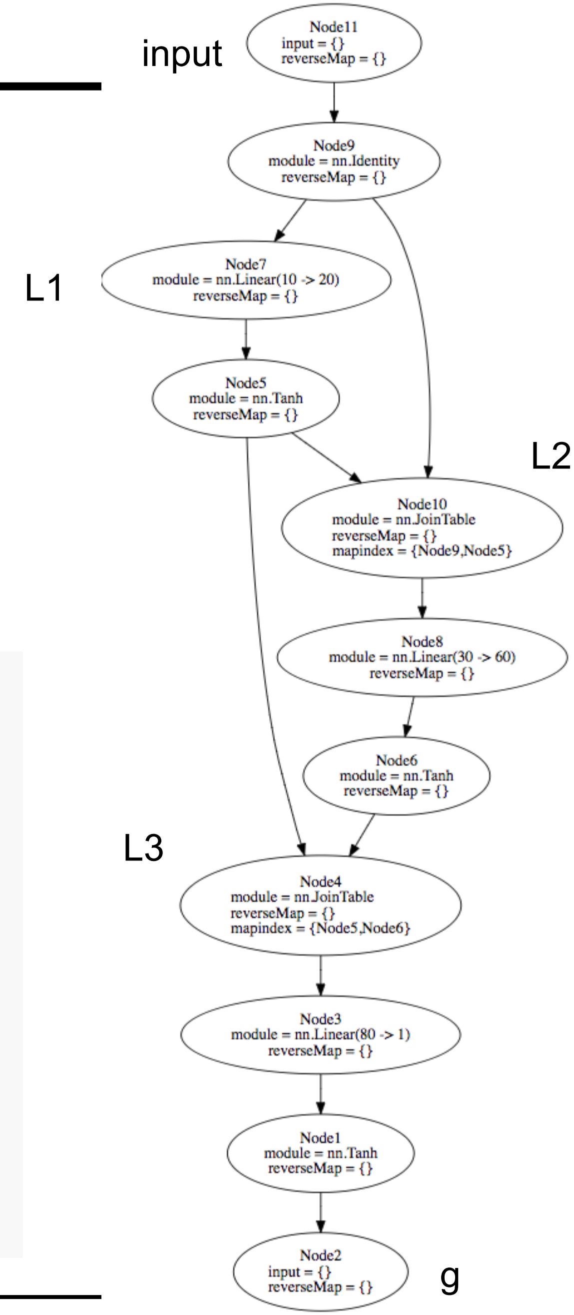
```
require 'nngraph';
g1 = - nn.Linear(20, 10)

g2 = g1
   - nn.Tanh()
   - nn.Linear(10, 10)
   - nn.Tanh()
   - nn.Linear(10, 1)
gnet = nn.gModule({g1}, {g2})
```



Neural Networks

- Composing more complicated networks
  - ‘nngraph’ is easier to use than ‘nn’

```
input = - nn.Identity()
L1 = input
   - nn.Linear(10, 20)
   - nn.Tanh()
L2 = {input, L1}
   - nn.JoinTable(1)
   - nn.Linear(30, 60)
   - nn.Tanh()
L3 = {L1, L2}
   - nn.JoinTable(1)
   - nn.Linear(80, 1)
   - nn.Tanh()
g = nn.gModule({input},{L3})
```



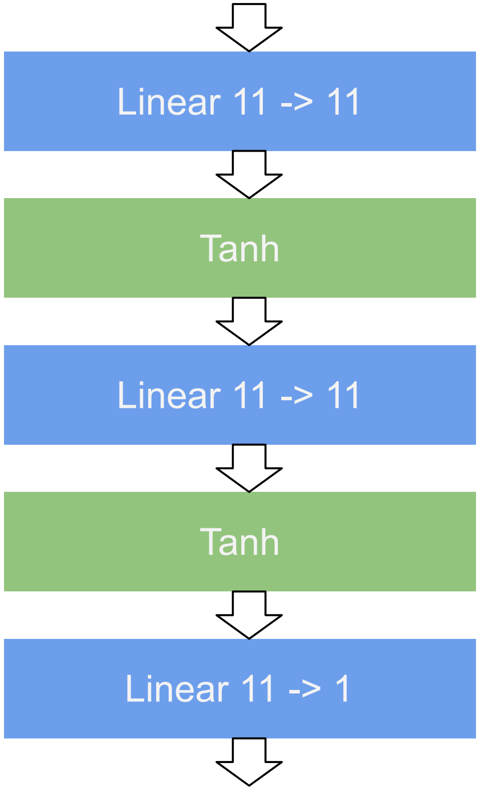
In Today’s Tutorial

- Basics of Lua Language
- Basic Tensor Operations in Torch
- Building a Simple Network
- Training the Simple Network
- Building a Recurrent Neural Network

Training: Setup

- Dataset: UCI red wine quality
  - X: 11 dims real numbers in [0,1]
  - Y: real number in [0, 1]
- Loss: Mean Square Error

```
torch.manualSeed(1234)
-- build the network
g1 = - nn.Linear(11, 11)
g2 = g1
    - nn.Tanh()
    - nn.Linear(11, 11)
    - nn.Tanh()
    - nn.Linear(11, 1)
winenet = nn.gModule({g1}, {g2})
-- mean square error
loss = nn.MSECriterion()
```



Training: General Framework

```
W, gradW = winenet:getParameters()
optimState = {}

for epoch = 1, n_epoches do
   local total_loss = 0
   for i=1, n_examples do
     x = xx[i]
     y = torch.Tensor({yy[i]})
     winenet:zeroGradParameters()
     function feval()
        local predicted = winenet:forward(x)
        local L = loss:forward(predicted, y)
        total_loss = total_loss + L
        local dL_dy = loss:backward(predicted, y)
        winenet:backward(x, dL_dy) -- computes and updates gradW
        return L, gradW
     end
     optim.sgd(feval, W, optimState)
   end
   print('at epoch', epoch, 'avg loss', total_loss/n_examples)
end
```

get the location of the weights and the grad weights

clean accumulated grad weights

forward through network

calculate loss

get grad from loss

backward through network

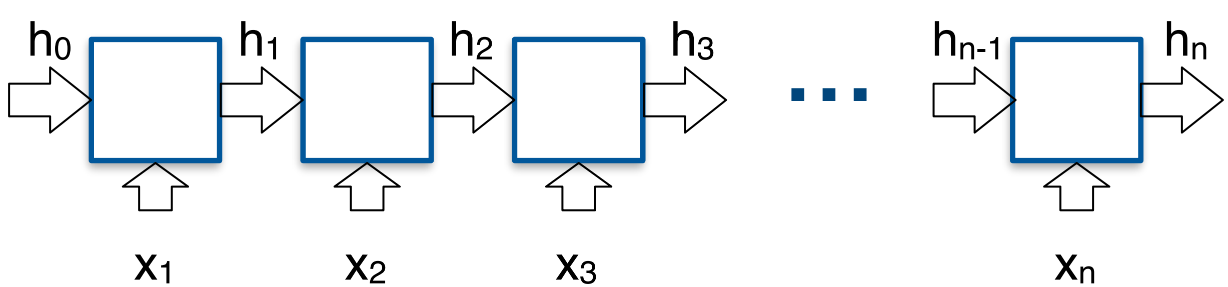
update weights using accumulated grad weights

In Today's Tutorial

- Basics of Lua Language
- Basic Tensor Operations in Torch
- Building a Simple Network
- Training the Simple Network
- Building a Recurrent Neural Network

A Simple Recurrent Network

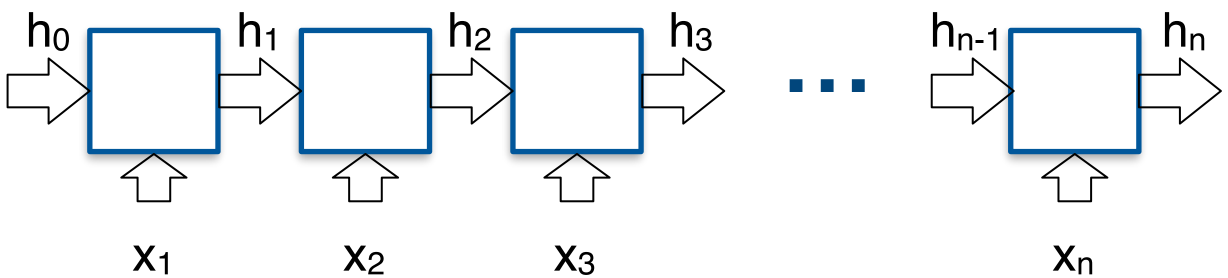
- The input of the network is a sequence of vectors:
  $x_t,\ t = 1\ldots n$
- At each step we recurrently calculate function:
  $h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$
- Parameters W_x,W_h, and b are shared in different steps
- Size of the network varies with the input length



A Simple Recurrent Network

- Building block: $h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$

```
ht1 = - nn.Identity()
xt = - nn.Identity()
ht = {ht1, xt}
    - nn.JoinTable(1)
    - nn.Linear(20, 10)
    - nn.Tanh()
stepfunction = nn.gModule({ht1, xt}, {ht})
```



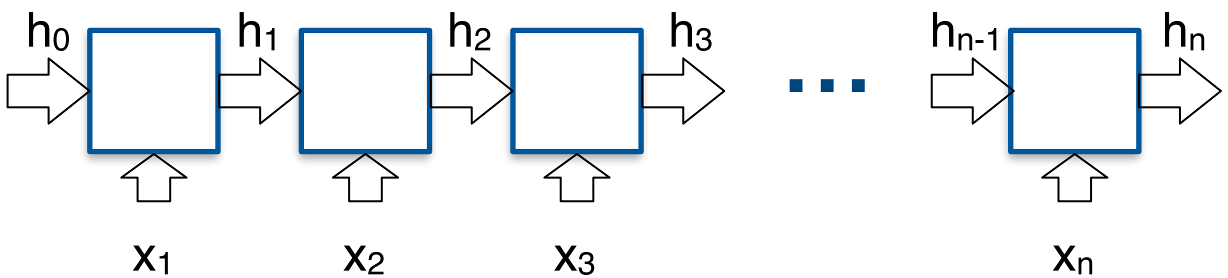
A Simple Recurrent Network

- Clone step function many times
- All clones **share** the same parameter at every time step
  - point to the **same** memory

```
W, gradW = stepfunction:getParameters()

clones = {}
N = 100
for i = 1, N do
    clones[i] = stepfunction:clone()
    share_params(clones[i], stepfunction)
end
```

getParameters() reallocate memory
do it before cloning

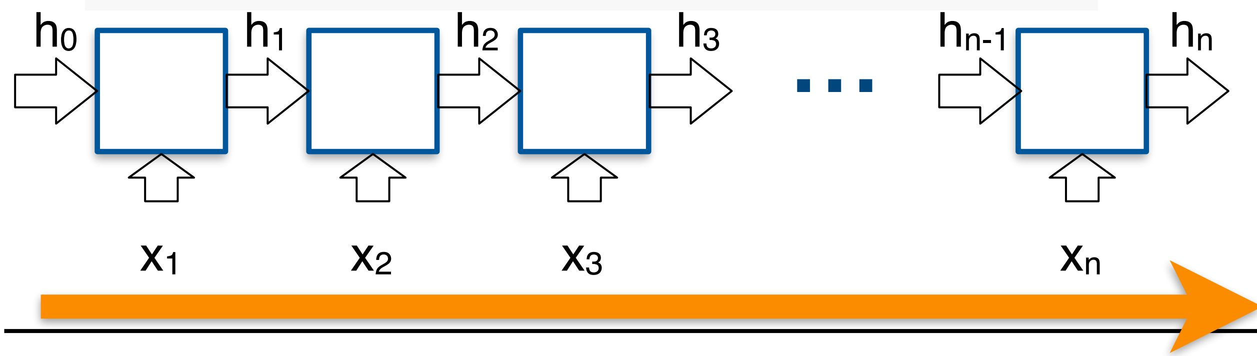
skipped for space reason, refer to ipython notebook



A Simple Recurrent Network

- Forward through time:

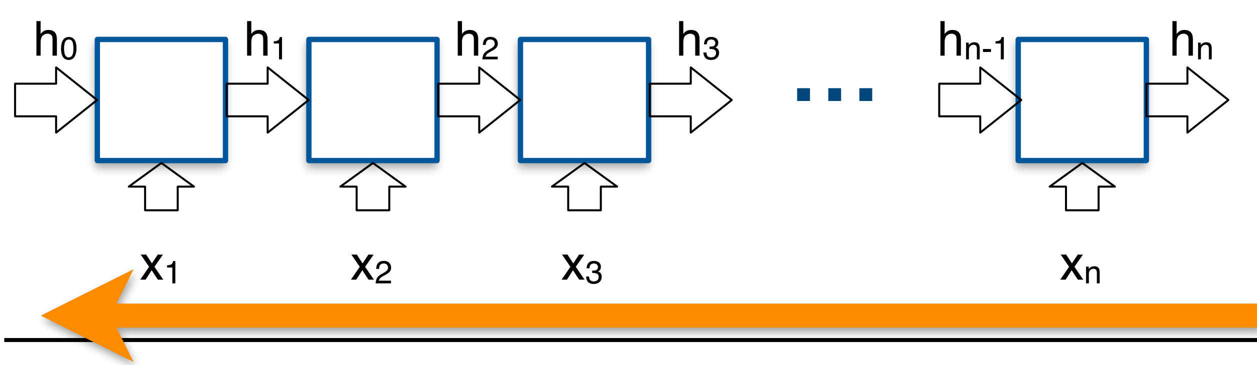
```
n = 7
h0 = torch.rand(10)
x = torch.rand(n, 10)
h = torch.zeros(n+1, 10)

W:uniform(-1, 1)

-- forward
h[1] = h0
for i = 1, n do
    h[i+1] = clones[i]:forward{h[i], x[i]}
end
```



A Simple Recurrent Network

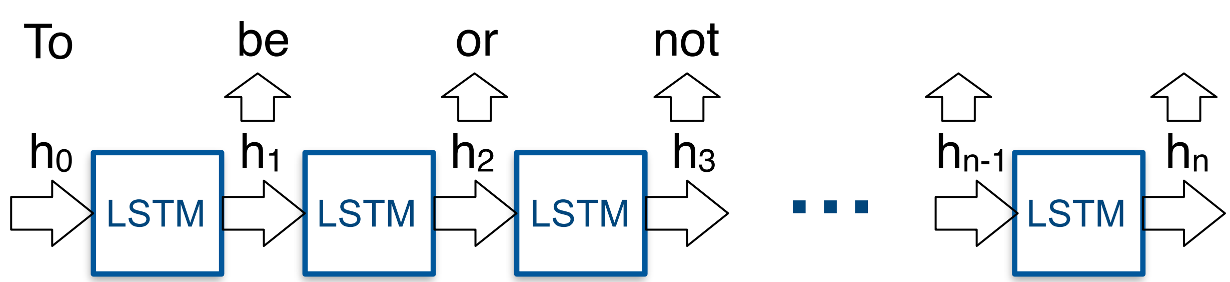
- Backward through time:

```
stepfunction:zeroGradParameters()

grad_h = torch.zeros(h:size())
grad_h[n+1] = torch.rand(10)
for i = n, 1, -1 do
    local grads = clones[i]:backward({h[i], x[i]}, grad_h[i+1])
    grad_h[i], grad_xi = unpack(grads)
end
```



A Simple Recurrent Network

- Simple RNNs sometimes are not very effective
  - Vanishing gradient problem
- LSTM/GRU become more popular
  - Several packages contain these modules
- We do not cover them in this tutorial
  - But we demo an example (from Andrej Karpathy)
  - Text Generation trained on Shakespeare's works



FIN

thanks!

Next Session: Dec 2, 2016

Deep Learning with DyNet

References

- *Learn X in Y minutes, X=Lua*
  https://learnxinyminutes.com/docs/lua/
- *Deep Learning with Torch* in CVPR 2015
  https://github.com/soumith/cvpr2015/blob/master/Deep%20Learning%20with%20Torch.ipynb
- *Torch Video Tutorials*
  https://github.com/Atcold/torch-Video-Tutorials
- *The Unreasonable Effectiveness of Recurrent Neural Networks*
  http://karpathy.github.io/2015/05/21/rnn-effectiveness/