# Lab Setup

SSH or Remote Desktop clients:

- SSH Access Software (recommended): PuTTy for Windows can be downloaded from www.putty.org
  - Alternatively you may use a provided browser-based SSH option
- Remote Desktop Software: Download NoMachine now for best performance from www.nomachine.com/download
  - Alternatively you may use a VNC client or the provided browser-based VNC option

Connection Instructions :

1. Navigate to nvlabs.qwiklab.com;
2. Login or create a new account;
3. Select the *Instructor-Led Hands-on Labs* Class;
4. Find the lab called *Applied Deep Learning for Vision and Natural Language with Torch7*;
   - select it, click Select, and finally click Start
5. After a short wait, lab instance Connection information will be shown

Please ask Lab Assistants for help!

---

class: center, middle

# Torch 7: Applied Deep Learning for Vision and Natural Language

Nicholas Leonard

https://github.com/nicholas-leonard

Element Inc.

April 6, 2016

---

# Agenda

1. Introduction
2. Packages
3. Tensors
4. Logistic Regression (Exercise 1)

5. Deep Learning
6. Multi-Layer Perceptron (Exercise 2)
7. Convolutional Neural Network (Exercise 3)
8. Recurrent Neural Network (Exercise 4)

---

# Introduction

My background:

- 2003-2008 : Bac. Degree in Comp. Sci. at Royal Military College of Canada ;
- 2008-2012 : Army Signals Officer :
  - management (office politics, emails), no code, no science ;
  - hobby : neural networks using Python and numpy ;
- 2012-2014 : Master's Degree in Deep Learning at University of Montreal ;
  - LISA/MILA lab ;
  - Yoshua Bengio and Aaron Courville as co-directors ;
  - 2012-2013 : Python, Theano and Pylearn2 ;
  - 2013-today : Lua, Torch7 ;
- 2014-today : research engineer at Element Inc. :
  - biometrics startup ;
  - deep learning on smart phones (Android, iOS) ;
  - open source contributions (Torch7).

---

## Introduction - Lua

Why take the time to learn Lua :

- easy interface between low-level C/CUDA/C++ and high-level Lua ;
- light-weight : used for embedded systems ;
- tables :
  - can be used as lists, dictionaries, packages, classes and objects ;
  - make it easy to extend existing classes (at any level) ;
- fast for-loops (LuaJIT) ;
- closures ;

Example :

```
a = {1,2,a=3, print=function(self) print(self) end}
a:print() -- i.e. a.print(a)
```

Output :

```
{
  1 : 1
  2 : 2
  print : function: 0x417f11e0
  a : 3
}
```

---

## Introduction - Torch 7

What's up with Torch 7?

- a powerful N-dimensional array ;
- lots of routines for indexing, slicing, transposing, ... ;
- amazing interface to C, via LuaJIT ;
- linear algebra routines ;
- easy modular neural networks ;
- numeric optimization routines ;
- fast and efficient GPU support ;
- ports to iOS, Android and FPGA backends ;
- under development since October 2002 ;
- used by Facebook, Google [DeepMind], Twitter, NYU, ... ;
- documentation, tutorials, demos, examples ;

---

## Introduction - Useful Links

Main : http://torch.ch/

Cheatsheet: https://github.com/torch/torch7/wiki/Cheatsheet

Github: https://github.com/torch/torch7

Google Group for new users and installation queries:
https://groups.google.com/forum/embed/?place=forum%2Ftorch7#!forum/torch7

Advanced only: https://gitter.im/torch/torch7

---

# Packages

The Torch 7 distribution is made up of different packages, each its own github repository :

- **torch7/cutorch** : tensors, BLAS, file I/O (serialization), OOP, unit testing and cmd-line argument parsing ;
- **nn/cunn** : easy and modular way to build and train neural networks using `modules` and `criterions` ;
- **nngraph** : nn with support for more complicated graphs ;
- **optim** : optimization package for nn. Provides training algorithms like SGD, LBFGS, etc. Uses closures ;
- **trepl** : torch read–eval–print loop, Lua interpreter, `th>` ;
- **paths** : file system manipulation package ;
- **image** : for saving, loading, constructing, transforming and displaying images ;

Refer to the torch.ch website for a more complete list of official packages.

---

## Packages - Unofficial

Many more unofficial packages out there :

- **dpnn** : extensions to the nn library. ;
- **rnn** : recurrent neural network library. Implements RNN, GRU, LSTM, BRNN, and RAM ;
- **nnx/cunnx** : experimental neural network modules and criterions : `SpatialReSampling`, `SoftMaxTree`, etc. ;
- **dataload** : library for loading and iterating through datasets ;
- **moses** : utility-belt library for functional programming in Lua, mostly for tables ;
- **threads/parallel** : libraries for multi-threading or multi-processing ;

---

# Tensors

Tensors are the main class of objects used in Torch 7 :

- An N-dimensional array that views an underlying `Storage`;
- Different Tensors can share the same `Storage`;
- Different types : `FloatTensor`, `DoubleTensor`, `IntTensor`, `CudaTensor`, ...;
- Implements many Basic Linear Algebra Sub-routines (BLAS) :
    - `torch.addmm` : matrix-matrix multiplication ;
    - `torch.addmv` : matrix-vector multiplication ;
    - `torch.addr` : outer-product between vectors ;
    - etc.
- Supports random initialization, indexing, transposition, sub-tensor extractions, and more ;
- Most operations for Float/Double are also implemented for Cuda Tensors (via **cutorch**) ;

## Tensors - Initialization

A `2x3` Tensor :

```
th> a = torch.FloatTensor(2,3)
-- initialized with garbage content (whatever was already there)
th> a
 8.6342e+19  4.5694e-41  8.6342e+19
 4.5694e-41  0.0000e+00  0.0000e+00
[torch.FloatTensor of size 2x3]
```

Fill with ones :

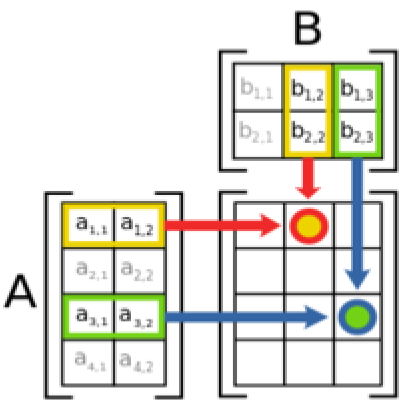
```
th> a:fill(1)
 1  1  1
 1  1  1
[torch.FloatTensor of size 2x3]
```

Random uniform initialization :

```
th> a:uniform(0,1) -- random uniform between 0 and 1
 0.6323  0.9232  0.2930
 0.8412  0.5131  0.9101
[torch.FloatTensor of size 2x3]
```

---

## Tensors - BLAS

.center[

]

Tensors are all about basic linear algebra. Let's multiply an `input` and a `weight` matrix into an `output` matrix :

```
batchSize, inputSize, outputSize = 4, 2, 3
```

```
input = torch.FloatTensor(batchSize, inputSize):uniform(0,1)
weight = torch.FloatTensor(outputSize, inputSize):uniform(0,1)
output = torch.FloatTensor(batchSize, outputSize):zero()
-- matrix matrix multiply :
output:addmm(0, output, 1, input, weight:t())
```

This is a common operation used by the popular `nn.Linear` module.

---

## Tensors - CUDA

Previous matrix-matrix multiply using CUDA :

```
require 'cutorch'
input = input:cuda() or torch.CudaTensor(batchSize, inputSize):uniform(0,1)
weight = weight:cuda() or torch.CudaTensor(outputSize, inputSize):uniform(0,1)
output = output:cuda() or torch.CudaTensor(batchSize, outputSize):zero()
-- matrix matrix multiply :
output:addmm(0, output, 1, input, weight:t())
```

So basically, no difference except for use of `torch.CudaTensor`.

---

# Neural Network library

The **nn** package :

- implements feed-forward neural networks ;
- neural networks form a computational flow-graph of transformations ;
- backpropagation is gradient descent using the chain rule ;

.center[$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$]

Two abstract classes :

- `nn.Module` : differentiable transformations of input to output ;
- `nn.Criterion` : cost function to minimize. Outputs a scalar loss;

Let's use it to build a simple logistic regressor...

---

# Logistic Regression - Module

A binary logisitic regressor `Module` with 2 input units and 1 output.

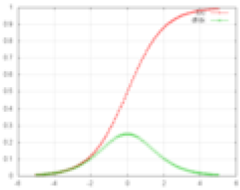
```
require 'nn'
module = nn.Sequential()
module:add(nn.Linear(2, 1))
module:add(nn.Sigmoid())
```

The above implements :

.center[$y = \sigma(Wx + b)$]

where the sigmoid (logistic function) is defined as :

.center[$$\sigma(z) = \frac{1}{1 - e^{-z}}$$]

.center[]

---

# Logistic Regression - Criterion and Data

A binary cross-entropy `Criterion` (which expects 0 or 1 valued targets) :

```
criterion = nn.BCECriterion()
```

The BCE loss is defined as :

.center[
$$-\sum_i [t_i \log(y_i) + (1 - t_i) \log(1 - y_i)]$$
]

Some random dummy dataset with 10 samples:

```
inputs = torch.Tensor(10,2):uniform(-1,1)
```

```
targets = torch.Tensor(10):random(0,1)
```

---

## Logistic Regression - Training

Function for one epoch of stochastic gradient descent (SGD)

```
require 'dpnn'
function trainEpoch(module, criterion, inputs, targets)
   for i=1,inputs:size(1) do
      local idx = math.random(1,inputs:size(1))
      local input, target = inputs[idx], targets:narrow(1,idx,1)
      -- forward
      local output = module:forward(input)
      local loss = criterion:forward(output, target)
      -- backward
      local gradOutput = criterion:backward(output, target)
      module:zeroGradParameters()
      local gradInput = module:backward(input, gradOutput)
      -- update
      module:updateGradParameters(0.9) -- momentum (dpnn)
      module:updateParameters(0.1) -- W = W - 0.1*dL/dW
   end
end
```

---

## Exercise 1 : Logistic Regression

Modify the `logistic-regression.lua` script to do the following :

1. Train for 100 epochs;
2. Each call to `trainEpoch` prints the mean error of that epoch;
3. Bonus : Use `Softmax` + `ClassNLLCriterion` instead of `Sigmoid` + `BCECriterion`.

Should output something like :

```
$ th logistic-regression.lua
...
Epoch 99 : mean loss = 0.038046
Epoch 100 : mean loss = 0.036116
```

Time : 10 min.

Solution found in `solution/logistic-regression.lua`.

---

## Exercise 1 : Take-away points

Like Python, Lua is relatively easy to use;

Stochastic Gradient Descent is simple: * forward, backward, update; * error is minimized after multiple iterations through data;

Can use BCE or NLL to minimize error of binary classification problems;

Logistic regression models are very simple models.

---

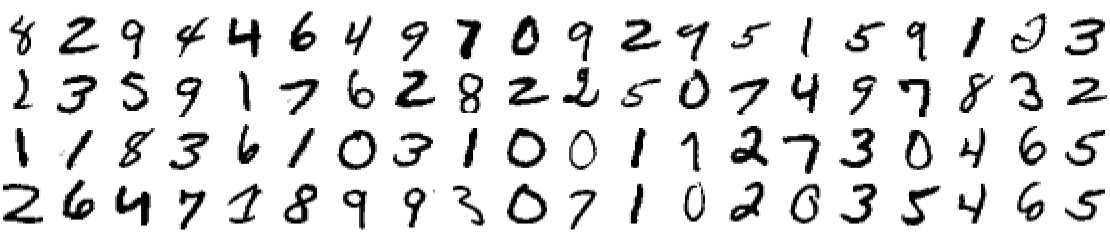
## Deep Learning

What is deep learning?

- **collection of techniques** to improve the optimization and **generalization** of neural networks :
  - rectified linear units ;
  - dropout ;
  - batch normalization ;
  - weight decay regularization ;
  - momentum learning ;
- **stacking layers** of transformations to create successively more abstract **levels of representations** ;
  - depth over breadth ;
  - deep multi-layer perceptrons ;
- **shared parameters** :
  - convolutional neural networks ;
  - recurrent neural networks ;
- **technological improvements** :
  - massively parallel processing : GPUs, CUDA ;
  - fast libraries : torch, cudnn, cuda-convnet, theano, tensorflow;

---

## Deep Learning - MNIST dataset

.center[]

**dataload** package makes it easy to obtain the MNIST dataset :

```
local dl = require 'dataload'
local trainset, validset, testset = dl.loadMNIST()
```

Each *batch* is a 4D `inputs` tensor and a `targets` 1D tensor:

```
inputs, targets = trainset:sample(32)
print(inputs:size(), targets:size())
 32 1 28 28
[torch.LongStorage of size 4]

 32
[torch.LongStorage of size 1]
```

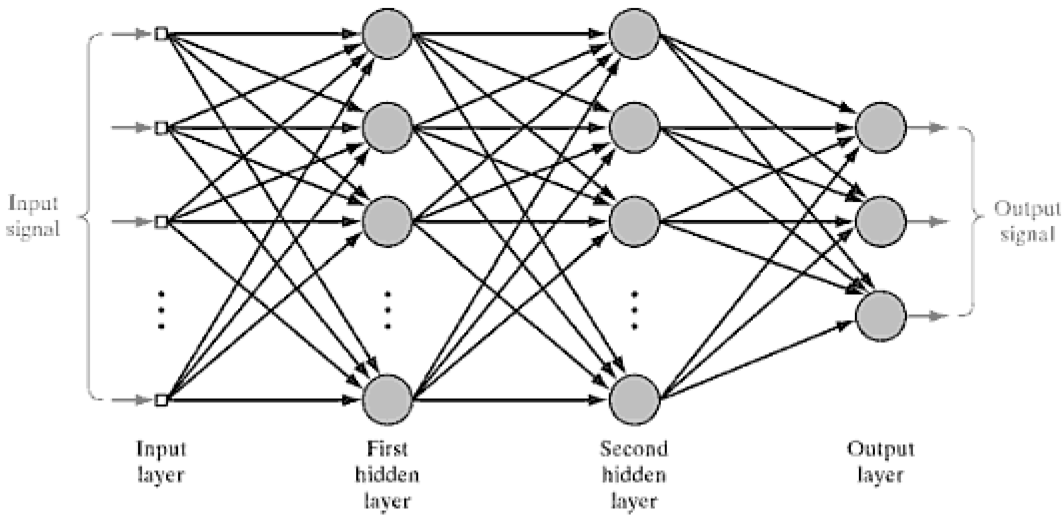
---

background-image: url(https://raw.githubusercontent.com/nicholas-leonard/slides/master/we-need-to-go-deeper.jpg)

# Deep Learning

---

# Multi-Layer Perceptron

.center[


]

An MLP is a stack of non-linear layers :

- each layer is an affine transform (`Linear`) followed by a transfer function (`Tanh`, `ReLU`, `SoftMax`) ;
- parameters (`weight`, `bias`) are found in the `Linear` module ;
- parameters are varied to fit the data ;
- transfer functions help to model complex relationships between input and output (non-linear);

## Multi-Layer Perceptron - Module and Criterion

An MLP with 2 layers of hidden units :

```
module = nn.Sequential()
module:add(nn.Convert()) -- casts input to model type (float -> double)
module:add(nn.Collapse(3)) -- collapse 3D to 1D
module:add(nn.Linear(1*28*28, 200))
module:add(nn.Tanh())
module:add(nn.Linear(200, 200))
module:add(nn.Tanh())
module:add(nn.Linear(200, 10))
module:add(nn.LogSoftMax()) -- for classification problems
```

Negative Log-Likelihood (NLL) Criterion :

```
criterion = nn.ClassNLLCriterion()
```

## Multi-Layer Perceptron - Cross-validation

A function to evaluate performance on the validation set :

```
require 'optim'
cm = optim.ConfusionMatrix(10)
function classEval(module, inputs, targets)
   cm:zero()
   for idx=1,inputs:size(1) do
      local input, target = inputs[idx], targets:narrow(1,idx,1)
      local output = module:forward(input)
      cm:add(output, target)
   end
   cm:updateValids()
   return cm.totalValid
end
```

Measure model's ability to *generalize* to new data.

## Multi-Layer Perceptron - Early-Stopping

Early-stopping on the validation set :

```
bestAccuracy, bestEpoch = 0, 0
wait = 0
for epoch=1,300 do
   trainEpoch(module, criterion, trainInputs, trainTargets)
   local validAccuracy = classEval(module, validInputs, validTargets)
   if validAccuracy > bestAccuracy then
      bestAccuracy, bestEpoch = validAccuracy, epoch
      torch.save("/path/to/saved/model.t7", module)
      print(string.format("New maxima : %f @ %f", bestAccuracy, bestEpoch))
      wait = 0
   else
      wait = wait + 1
      if wait > 30 then break end
   end
end
```

Early-stops when no new maxima has been found for 30 consecutive epochs.

---

# Exercise 2 : Multi-Layer Perceptron

Modify the `multi-layer-perceptron.lua` script to do the following :

1. take options from the command-line;
2. add `Dropout` between hidden layers;
3. Bonus : write script to evaluate saved model on test set;
4. Bonus : make the number of hidden layers a hyper-parameter.

Time : 10 min.

Evaluation script should output something like :

```
th evaluate-mlp.lua
Test accuracy=0.977000
```

Solution found in `solution/[train|evaluate]-mlp.lua`.

---

# Exercise 2 : Take-away points

Easier to try different hyper-parameters (options) from the cmd-line;

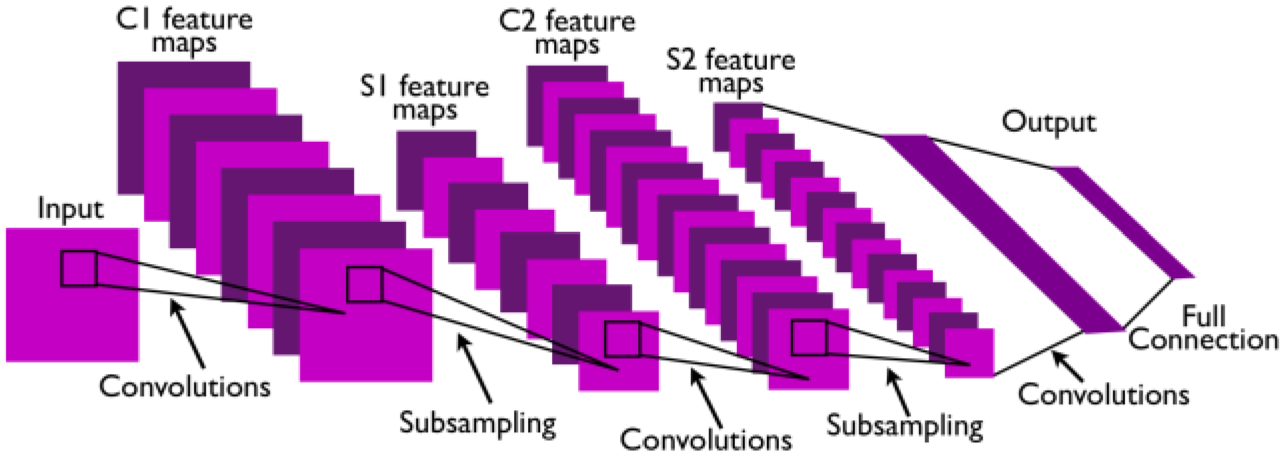
Dropout can help with generalization, but increases convergence time;

Experimentation usually requires two scripts and 3 datasets : * training : optimize model on *training set*, early-stop on *validation set*; * evaluation : measure performance on *test set*;

Deeper models (more layers) can help with generalization.

---

# Convolutional Neural Network

.center[


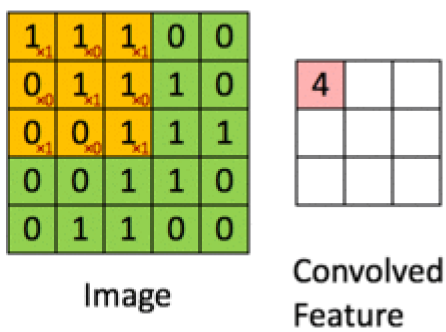
]

CNNs are often stacks of meta-layers each made from 3 layers :

1. **convolution** : convolve a kernel over the image along height and width axes ;
2. **transfer function** : a non-linearity like `Tanh` or `ReLU` ;
3. **sub-sampling** : reduce the size (height x width) of feature maps by pooling them spatially;

---

## Convolutional Neural Network - Convolution

.center[


]

Convolution modules typically have the following arguments :

- `padSize` : how much zero-padding to add around the input image ;
- `inputSize` : number of input channels (e.g. 3 for RGB image) ;
- `outputSize` : number of output channels (number of filters) ;
- `kernelSize` : height and width of the kernel ;
- `kernelStride` : step-size of the kernel (typically 1) ;

Parameters of the convolution (i.e. the kernel) :

- `weight` : 4D Tensor `outputSize x inputSize x kernelSize x kernelSize` ;
- `bias` : 1D Tensor `outputSize` ;

---

## Convolutional Neural Network - SpatialConvolution

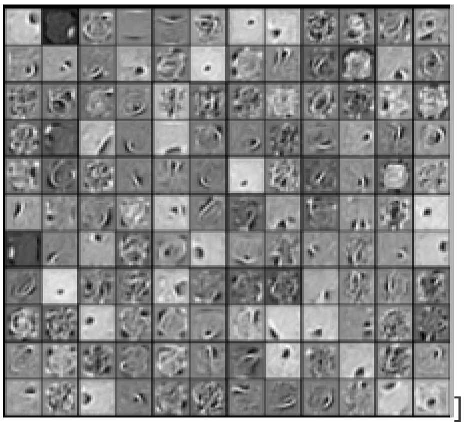
`SpatialConvolution` with 3 input and 4 output channels using a `5x5` kernel on a `12x12` image :

```
input = torch.rand(3,12,12)
conv = nn.SpatialConvolution(3,4,5,5)
output = conv:forward(input) -- size is 4 x 8 x 8
```

Now with 2 pixels of padding on each side:

```
conv = nn.SpatialConvolution(3,4,5,5,1,1,2,2)
output = conv:forward(input) -- size is 4 x 12 x 12
```

Learns filters like :

.center[]

---

## Convolutional Neural Network - Sub-sampling

Sub-sampling modules :

- typically max-pooling is used : `SpatialMaxPooling` ;
- makes the model more invariant to translation ;
- reduces the size of the spatial dimensions ;

`SpatialMaxPooling` to pool inputs in a `2x2` area with a stride of 2:

```
input = torch.range(1,16):double():resize(1,4,4)
```

```
pool = nn.SpatialMaxPooling(2,2,2,2)
output = pool:forward(input)
print(input, output)
(1,.,.) =
   1   2   3   4
   5   6   7   8
   9  10  11  12
  13  14  15  16
[torch.DoubleTensor of size 1x4x4]

(1,.,.) =
   6   8
  14  16
[torch.DoubleTensor of size 1x2x2]
```

---

## Convolutional Neural Network - MNIST

Convolutional Neural Network for the MNIST dataset :

```
cnn = nn.Sequential()
-- 2 conv layers :
cnn:add(nn.Convert())
cnn:add(nn.SpatialConvolution(1, 16, 5, 5, 1, 1, 2, 2))
cnn:add(nn.ReLU())
cnn:add(nn.SpatialMaxPooling(2, 2, 2, 2))
cnn:add(nn.SpatialConvolution(16, 32, 5, 5, 1, 1, 2, 2))
cnn:add(nn.ReLU())
cnn:add(nn.SpatialMaxPooling(2, 2, 2, 2))
-- 1 dense hidden layer :
outsize = cnn:outside{1,1,28, 28} -- output size of convolutions
cnn:add(nn.Collapse(3))
cnn:add(nn.Linear(outsize[2]*outsize[3]*outsize[4], 200))
cnn:add(nn.ReLU())
-- output layer
cnn:add(nn.Linear(200, 10))
cnn:add(nn.LogSoftMax())
```

---

## Convolutional Neural Network - Print Module

The `cnn` looks like this :

```
print(cnn)
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) -> (10) ->
(11) -> output]
  (1): nn.Convert
  (2): nn.SpatialConvolution(1 -> 16, 5x5, 1,1, 2,2)
  (3): nn.ReLU
```

```
  (4): nn.SpatialMaxPooling(2,2,2,2)
  (5): nn.SpatialConvolution(16 -> 32, 5x5, 1,1, 2,2)
  (6): nn.ReLU
  (7): nn.SpatialMaxPooling(2,2,2,2)
  (8): nn.Linear(1568 -> 200)
  (9): nn.ReLU
  (10): nn.Linear(200 -> 10)
  (11): nn.LogSoftMax
}
```

---

## Exercise 3 : Convolutional Neural Network

Use the `convolution-neural-network.lua` script.

Modify the script to do the following :

1. add `SpatialMaxPooling` to convolution layers;
2. add `[Spatial]BatchNormalization` to convolution and hidden layers ;
3. Bonus : efficiently save model to disk using `Serial`.

Try the `-cuda` flag.

Evaluate with `th solution/evaluate-mlp.lua -modelpath '/home/ubuntu/save/cnn.t7'`.

Time : 10 min.

Solution found in `solution/convolution-neural-network.lua`.

---

## Exercise 3 : Take-away points

CNNs have better performance than MLPs for images;

We can get a `4-10x` speedup using NVIDIA GPUs;

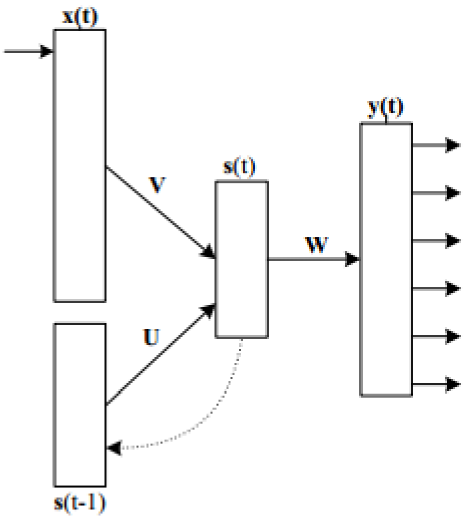
`Serial` doesn't save `output` and `gradInput` to disk : 1.5 vs 9 MB;

Batch normalization can help with convergence and generalization;

Pooling reduces the width/height of representations.

---

# Recurrent Neural Network

.center[

]

Simple RNN :

- for modeling sequential data like text, speech, videos ;
- 3 layers : input (`V`), recurrent (`U`) and output (`W`) layer ;
- feed the previous state as input to next state ;
- long sequences suffer from exploding and vanishing gradient ;

---

# Recurrent Neural Network - Language Model

Maximize likelihood of next word given previous words (input -> target) :

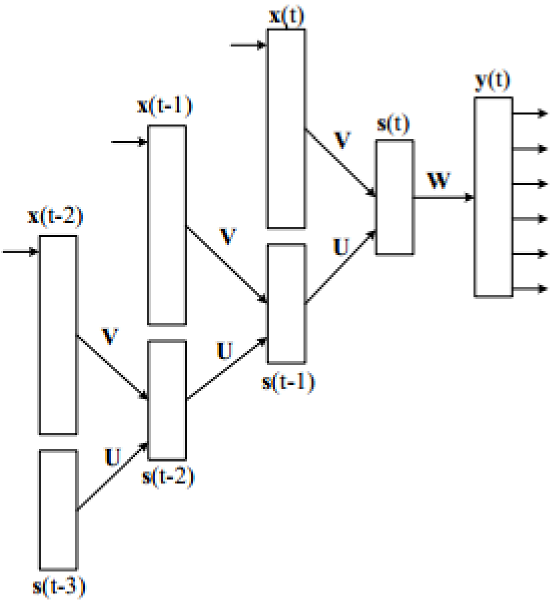
1. `we` -> `need`
2. `we, need` -> `to`
3. `we, need, to` -> `go`
4. `we, need, to, go` -> `deeper`

Neural network language model (NNLM) :

- learn an embedding space of words ;
- each word is a vector of parameters ;
- embedding space is implemented using `LookupTable` ;
- embedding space is a `weight` matrix of size `vocabSize x embedSize` ;

---

# Recurrent Neural Network - BPTT

.center[]



Back-propagation through time :

- forward-propagate for `T` time-steps ;
- unfold network for `T` time-steps ;
- back-propagate through unfolded network ;
- accumulate parameter gradients (sum over time-steps) ;

---

## Recurrent Neural Network - Penn Tree Bank

Penn Tree Bank dataset :

- common benchmark for language models ;
- 10000 word vocabulary ;
- approx. 1 million words of text ;

Use **dataload** to get Penn Tree Bank dataset :

```
trainset, validset, testset = dl.loadPTB{batchsize,1,1}
```

Batch of 3 sample sequences of length 5:

```
print(inputs:t(), targets:t())
  238   808   951  1326  1477
 8311  7671  4749    49  2308
 3660  9800  4765  5375  6018
[torch.IntTensor of size 3x5]

  808   951  1326  1477  1692
 7671  4749    49  2308  9120
```

```
 9800  4765  5375  6018  7523
[torch.IntTensor of size 3x5]
```

---

## Recurrent Neural Network - rnn

Use the **rnn** package to build an RNNLM.

A module that implements recurrence `{x[t], h[t-1]} -> h[t]` :

```
rm = nn.Sequential() -- input is {x[t], h[t-1]}
   :add(nn.ParallelTable()
      :add(nn.LookupTable(10000, 200)) -- input layer (V)
      :add(nn.Linear(200, 200))) -- recurrent layer (U)
   :add(nn.CAddTable())
   :add(nn.Sigmoid()) -- output is h[t]
```

Wrap into a `Recurrence` module and add an output layer:

```
rnn = nn.Sequential()
   :add(nn.Recurrence(rm, 200, 0))
   :add(nn.Linear(200, 10000)) -- output layer (W)
   :add(nn.LogSoftMax())
```

Wrap into a `Sequencer` to handle one sequence per `forward` call:

```
rnn = nn.Sequencer(rnn)
```

---

## Exercise 4 : Recurrent Language Model

Modify the `recurrent-language-model.lua` script to do the following :

1. use `LSTM` , `FastLSTM` or `GRU` instead of `Recurrence`;
2. train to reach `150` perplexity (PPL) on validation set;
3. use `evaluate-rnnlm.lua` to sample text from saved models.

```
nn.Serial @ nn.Sequential {
  [input -> (1) -> (2) -> (3) -> output]
  (1): nn.LookupTable
  (2): nn.SplitTable
  (3): nn.Sequencer @ nn.Recursor @ nn.Sequential {
    [input -> (1) -> (2) -> (3) -> (4) -> output]
    (1): nn.FastLSTM(200 -> 200)
    (2): nn.FastLSTM(200 -> 200)
    (3): nn.Linear(200 -> 10000)
```

```
    (4): nn.LogSoftMax
  }
}
```

Time : 15 min;

Solutions in `solution/train-rnnlm.lua`.

---

## Exercise 4 : Take-away points

LSTM and GRU can learn longer sequences (`seqlen`) than RNNs;

Stacking LSTM/GRU/RNNs can give even better results;

Optimizing hyper-parameters is a process of trial and error that takes time;

Using a learning rate schedule can help :

```
th recurrent-language-model.lua -progress -cuda -lstm -seqlen 20 -hiddensize
'{200,200}'
-batchsize 20 -startlr 1 -cutoff 5 -maxepoch 13
-schedule
'{[5]=0.5,[6]=0.25,[7]=0.125,[8]=0.0625,[9]=0.03125,[10]=0.015625,[11]=0.0078125,[1
2]=0.00390625}'
```

Evaluation loops through one continous sequence;

---

## Exercise 4 : Generating text

Generating text using a model with 116 test PPL :

```
th evaluate-rnnlm.lua -xplogpath /home/ubuntu/save/rnnlm/rnnlm200x200.t7 -nsample
200
```

will result in someting like this :

```
mr. jones contends that in his first popularity the organizations are
almost <unk> by their side <eos> it is particularly almost he adds <eos>
but it 's also a <unk> unsuccessfully <eos> i felt somebody wanted
westridge and financial commercial steelmakers <eos> an oct.
N bankruptcy-law article is comment <eos> <unk> <unk> <eos> there 's no
signs of internal mergers and companies involved in the life of light
insider business <eos> michael <unk> several years old and the mr.
buffett as a former abortions in the senate to abortion-rights its mind
```

```
is mr. straszheim 's remarks with a career replacement works <eos> <unk>
education was named a director of this plastics media <eos>
```

Not bad for 1h of training!

---

## Recurrent Neural Network - Character LM

References : * https://github.com/karpathy/char-rnn * https://github.com/hughperkins/char-lstm

Text generated using char-level LM trained on 1M reddit comments:

```
<post>
Diablo
<comment score=1>
I liked this game so much!! Hope telling that numbers' benefits and
features never found out at that level is a total breeze
because it's not even a developer/voice opening and rusher runs
the game against so many people having noticeable purchases of selling
the developers built or trying to run the patch to Jagex.
</comment>
```

Looks good! Wait a second...

---

# Questions ?

.center[