torch

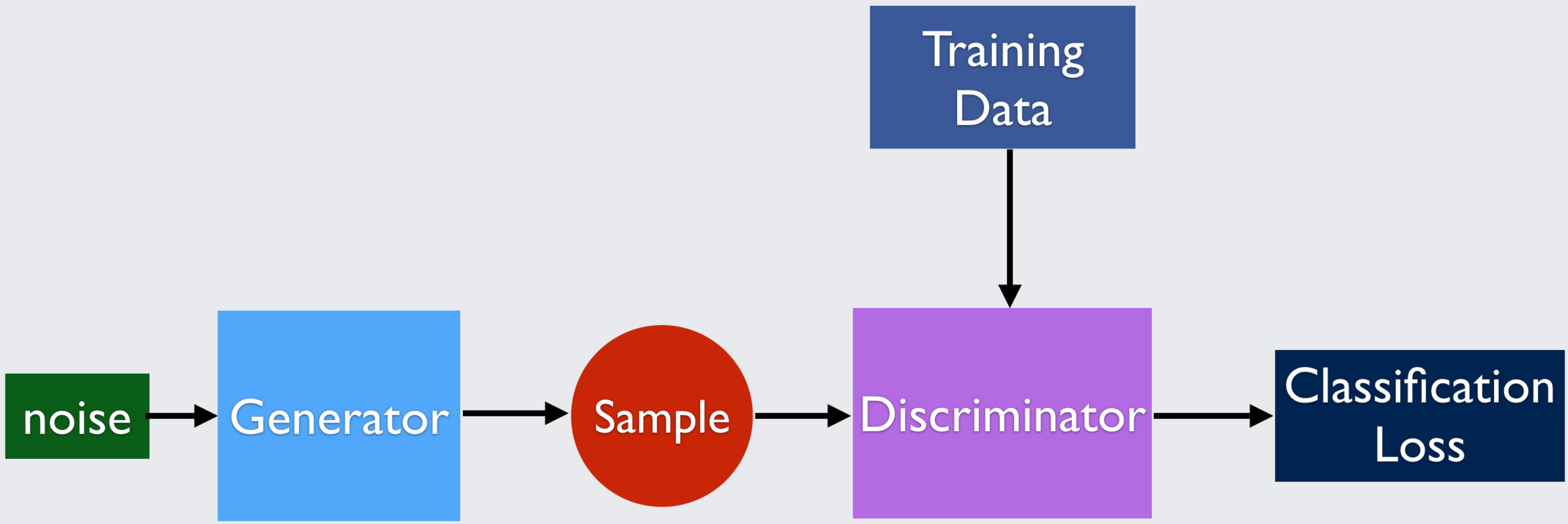
Extensions

Soumith Chintala
Facebook AI Research

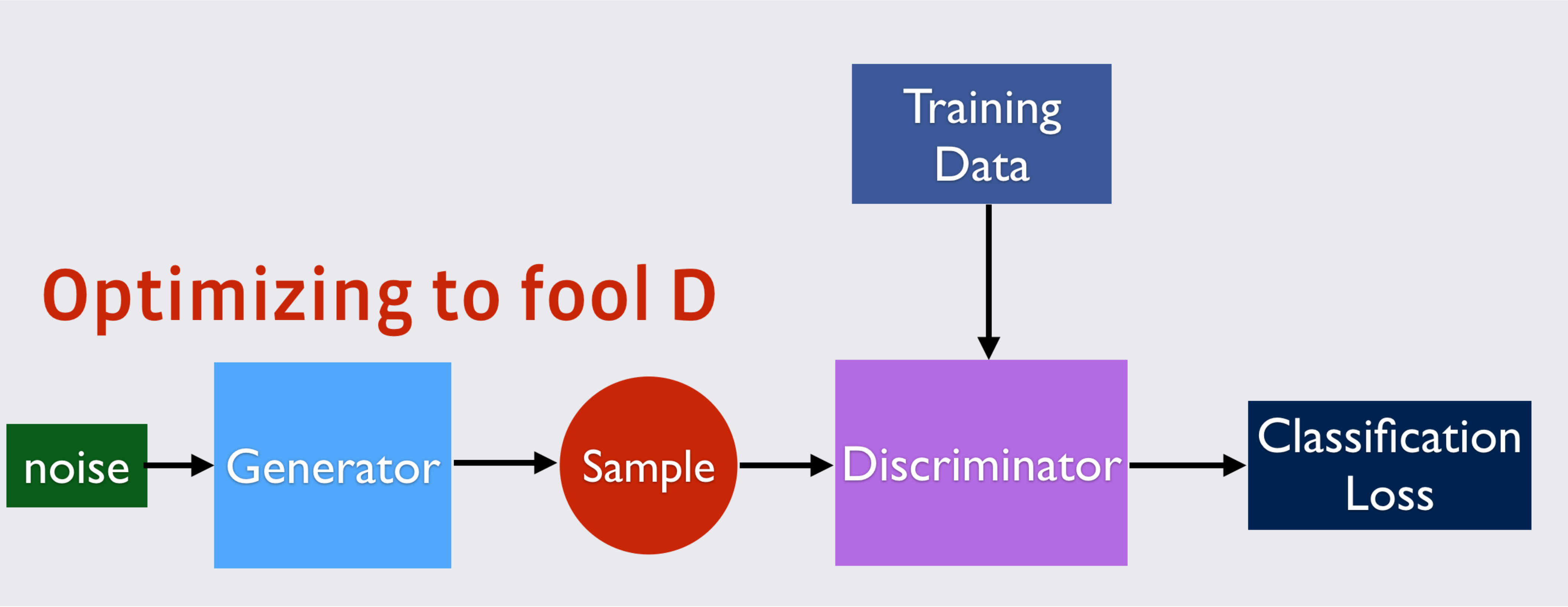
Overview

- A complete example of using nn + optim + threads for image generation
- the magic autograd package
- torchnet: common patterns for Torch by Facebook

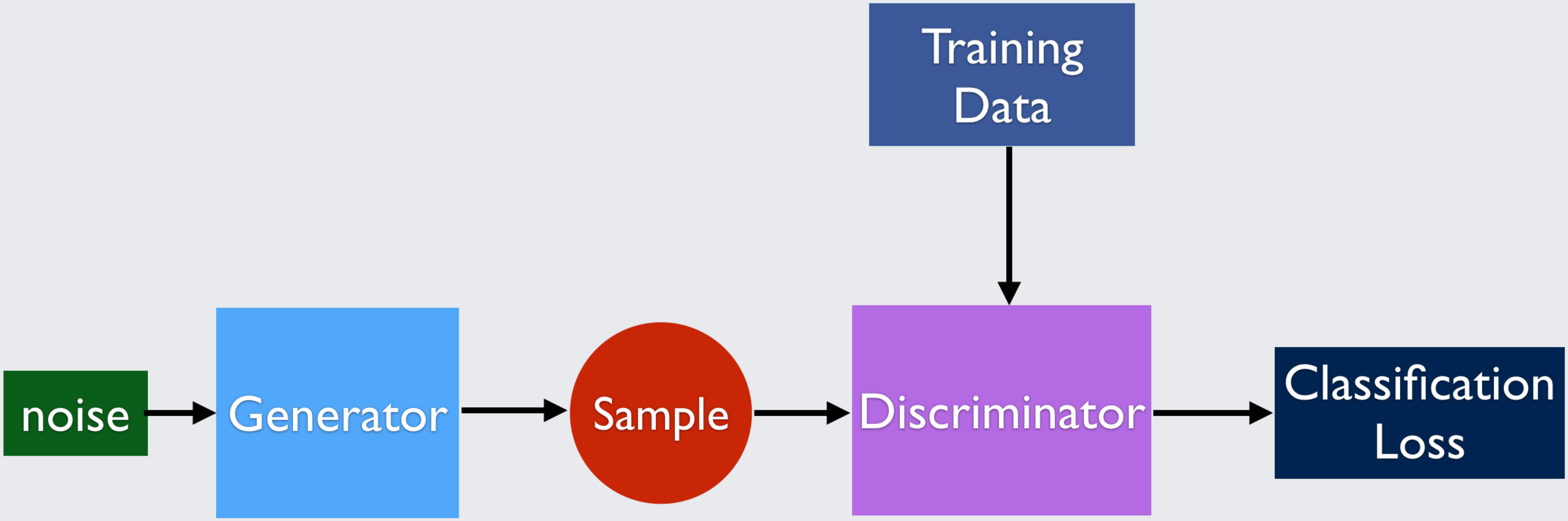
DCGAN


Training Data
noise
Generator
Sample
Discriminator
Classification Loss

DCGAN

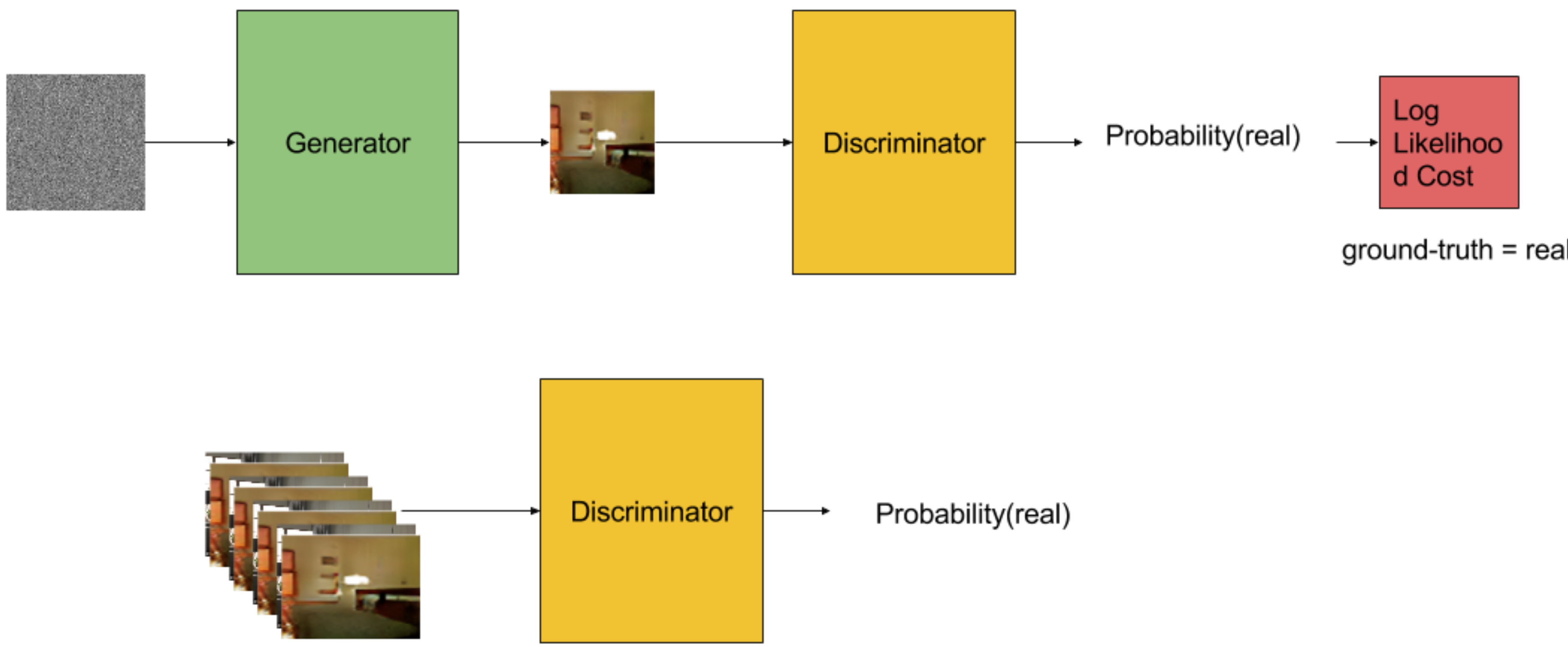

Training Data
Optimizing to fool D
noise
Generator
Sample
Discriminator
Classification Loss

DCGAN



Optimizing to not get fooled by G

DCGAN



Generator
Discriminator
Probability(real)
Log Likelihoo d Cost
ground-truth = real
Discriminator
Probability(real)

DCGAN

Boilerplate

```lua
require 'torch'
require 'nn'
require 'optim'
require 'xlua'

opt = {
   batchSize = 64,
   loadSize = 96,
   fineSize = 64,
   nz = 100,               -- #  of dim for Z
   ngf = 64,               -- #  of gen filters in first conv layer
   ndf = 64,               -- #  of discrim filters in first conv layer
   nThreads = 4,           -- #  of data loading threads to use
   niter = 25,             -- #  of iter at starting learning rate
   gpu = 1,                -- gpu = 0 is CPU mode. gpu=X is GPU mode on GPU X
   name = 'experiment1',
}

xlua.envparams(opt)

-- create data loader
local DataLoader = paths.dofile('data/data.lua')
local data = DataLoader.new(opt.nThreads, opt.dataset, opt)
print("Dataset: " .. opt.dataset, " Size: ", data:size())
```

DCGAN

Generator

```lua
local netG = nn.Sequential()
-- input is Z, going into a convolution
netG:add(nn.SpatialFullConvolution(nz, ngf * 8, 4, 4))
netG:add(nn.SpatialBatchNormalization(ngf * 8)):add(nn.ReLU(true))
-- state size: (ngf*8) x 4 x 4
netG:add(nn.SpatialFullConvolution(ngf * 8, ngf * 4, 4, 4, 2, 2, 1, 1))
netG:add(nn.SpatialBatchNormalization(ngf * 4)):add(nn.ReLU(true))
-- state size: (ngf*4) x 8 x 8
netG:add(nn.SpatialFullConvolution(ngf * 4, ngf * 2, 4, 4, 2, 2, 1, 1))
netG:add(nn.SpatialBatchNormalization(ngf * 2)):add(nn.ReLU(true))
-- state size: (ngf*2) x 16 x 16
netG:add(nn.SpatialFullConvolution(ngf * 2, ngf, 4, 4, 2, 2, 1, 1))
netG:add(nn.SpatialBatchNormalization(ngf)):add(nn.ReLU(true))
-- state size: (ngf) x 32 x 32
netG:add(nn.SpatialFullConvolution(ngf, nc, 4, 4, 2, 2, 1, 1))
netG:add(nn.Tanh())
-- state size: (nc) x 64 x 64
```

DCGAN

Discriminator

```lua
local netD = nn.Sequential()

-- input is (nc) x 64 x 64
netD:add(nn.SpatialConvolution(nc, ndf, 4, 4, 2, 2, 1, 1))
netD:add(nn.LeakyReLU(0.2, true))
-- state size: (ndf) x 32 x 32
netD:add(nn.SpatialConvolution(ndf, ndf * 2, 4, 4, 2, 2, 1, 1))
netD:add(nn.SpatialBatchNormalization(ndf * 2)):add(nn.LeakyReLU(0.2, true))
-- state size: (ndf*2) x 16 x 16
netD:add(nn.SpatialConvolution(ndf * 2, ndf * 4, 4, 4, 2, 2, 1, 1))
netD:add(nn.SpatialBatchNormalization(ndf * 4)):add(nn.LeakyReLU(0.2, true))
-- state size: (ndf*4) x 8 x 8
netD:add(nn.SpatialConvolution(ndf * 4, ndf * 8, 4, 4, 2, 2, 1, 1))
netD:add(nn.SpatialBatchNormalization(ndf * 8)):add(nn.LeakyReLU(0.2, true))
-- state size: (ndf*8) x 4 x 4
netD:add(nn.SpatialConvolution(ndf * 8, 1, 4, 4))
netD:add(nn.Sigmoid())
-- state size: 1 x 1 x 1
netD:add(nn.View(1):setNumInputDims(3))
-- state size: 1
```

torch

DCGAN

Loss

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

DCGAN

Optim configuration

```lua
optimStateG = {
   learningRate = 0.0002,
   beta1 = 0.5,
}
optimStateD = {
   learningRate = opt.lr,
   beta1 = 0.5,
}
```

DCGAN

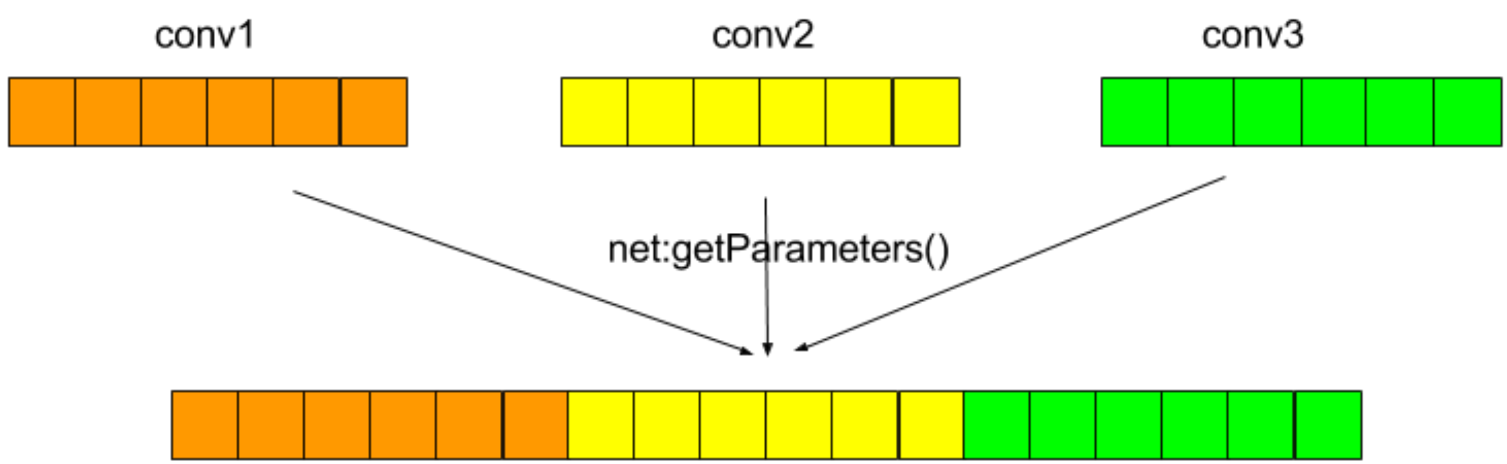
Create buffers and push them to GPU

```lua
----------------------------------------------------------------------------
local input = torch.Tensor(opt.batchSize, 3, opt.fineSize, opt.fineSize)
local noise = torch.Tensor(opt.batchSize, nz, 1, 1)
local label = torch.Tensor(opt.batchSize)
local errD, errG
----------------------------------------------------------------------------
if opt.gpu > 0 then
   require 'cunn'
   cutorch.setDevice(opt.gpu)
   input = input:cuda();  noise = noise:cuda();  label = label:cuda()
   netD:cuda();           netG:cuda();           criterion:cuda()
end
```

DCGAN

flatten the parameters

```lua
local parametersD, gradParametersD = netD:getParameters()
local parametersG, gradParametersG = netG:getParameters()
```



DCGAN

Define y = D(w_d, input)

```
-- create closure to evaluate f(X) and df/dX of discriminator
local fDx = function(x)
   gradParametersD:zero()

   -- train with real
   local real = data:getBatch()
   input:copy(real)
   label:fill(real_label)

   local output = netD:forward(input)
   local errD_real = criterion:forward(output, label)
   local df_do = criterion:backward(output, label)
   netD:backward(input, df_do)

   -- train with fake
   noise:uniform(-1, 1)
   local fake = netG:forward(noise)
   input:copy(fake)
   label:fill(fake_label)

   local output = netD:forward(input)
   local errD_fake = criterion:forward(output, label)
   local df_do = criterion:backward(output, label)
   netD:backward(input, df_do)

   errD = errD_real + errD_fake

   return errD, gradParametersD
end
```

torch

DCGAN

Define y = G(w_g, z)

```
-- create closure to evaluate f(X) and df/dX of generator
local fGx = function(x)
   gradParametersG:zero()

   noise:uniform(-1, 1) -- regenerate random noise
   local fake = netG:forward(noise)
   input:copy(fake)
   label:fill(real_label) -- fake labels are real for generator cost

   local output = netD:forward(input)
   errG = criterion:forward(output, label)
   local df_do = criterion:backward(output, label)
   local df_dg = netD:updateGradInput(input, df_do)

   netG:backward(noise, df_dg)
   return errG, gradParametersG
end
```

DCGAN

Now train!

```lua
-- train
for epoch = 1, opt.niter do
   for i = 1, data:size(), opt.batchSize do
      -- (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
      optim.adam(fDx, parametersD, optimStateD)

      -- (2) Update G network: maximize log(D(G(z)))
      optim.adam(fGx, parametersG, optimStateG)
   end
   paths.mkdir('checkpoints')
   torch.save('checkpoints/' .. opt.name .. '_' .. epoch .. '_net_G.t7', netG)
   torch.save('checkpoints/' .. opt.name .. '_' .. epoch .. '_net_D.t7', netD)
end
```

DCGAN

Now train!

[VIDEO REMOVED]

DCGAN

Generations!

```
local images = net:forward(noise)
```

Noise:

DCGAN

Generations!

```
local images = net:forward(noise)
```

Noise:

```
noise = torch.Tensor(opt.batchSize, opt.nz, opt.imsize, opt.imsize)
noise:uniform(-1, 1)
```

DCGAN

Generations!

```lua
local images = net:forward(noise)
```

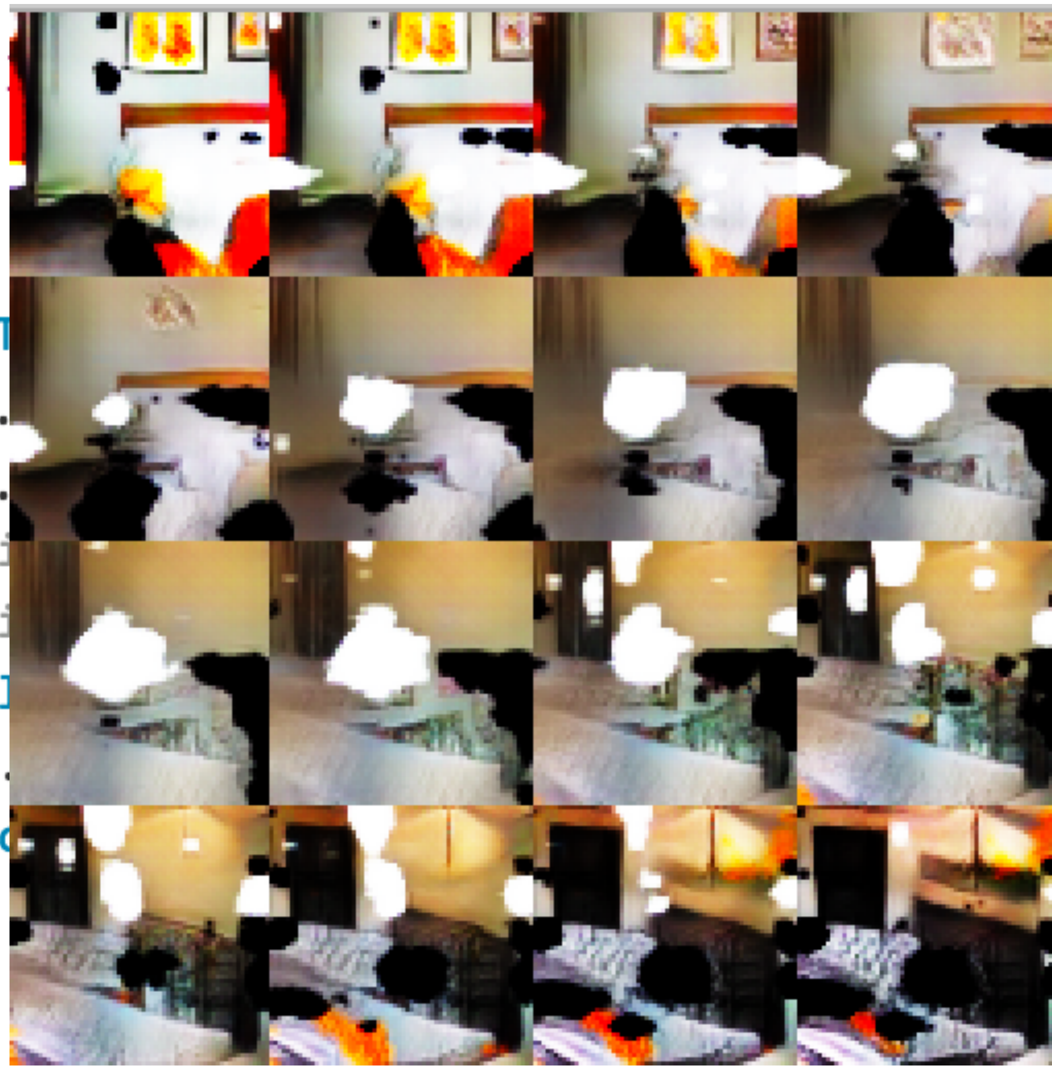
Interpolations in z space:

```lua
noise = torch.Tensor(opt.batchSize, opt.nz, opt.imsize, opt.imsize)
noiseL = torch.FloatTensor(opt.nz):uniform(-1, 1)
noiseR = torch.FloatTensor(opt.nz):uniform(-1, 1)
-- do a linear interpolation in Z space between point A and point B
-- each sample in the mini-batch is a point on the line
 line  = torch.linspace(0, 1, opt.batchSize)
 for i = 1, opt.batchSize do
     noise:select(1, i):copy(noiseL * line[i] + noiseR * (1 - line[i]))
 end
```

DCGAN

Generations!

Interpolations



```
noise = torch.T                    e, opt.imsize)
noiseL = torch.
noiseR = torch.
-- do a linear i                   and point B
-- each sample i
 line   = torch.I
 for i = 1, opt.
     noise:selec                   (1 - line[i]))
 end
```

DCGAN

Generations!

```
local images = net:forward(noise)
```

Arithmetic in z space:

https://github.com/soumith/dcgan.torch/blob/simple/arithmetic.lua

LIVE DEMO

autograd by

Basic usage

```
In [1]: autograd = require 'autograd'
```

```
In [38]: foo = function(a, b, c)
             return a + b * c
         end
```

```
In [39]: dfoo = autograd(foo)
```

```
In [40]: da, f_abc = dfoo(3.5, 2.1, 1.1)
         print('Value: ' .. f_abc, 'Gradient: ', da)
Out[40]: Value: 5.81      Gradient:        1
```

autograd by

Conditionals

```
In [41]: foo = function(a, b, c)
             if b > c then
                 return a * math.sin(b)
             else
                 return a + b * c
             end
         end

In [42]: dfoo = autograd(foo)

In [43]: da, f_abc = dfoo(3.5, 2.1, 1.1)
         print('Value: ' .. f_abc, 'Gradient: ', da)

Out[43]: Value: 3.0212327832711  Gradient:       0.86320936664887

In [44]: da, f_abc = dfoo(3.5, 0.1, 1.1)
         print('Value: ' .. f_abc, 'Gradient: ', da)

Out[44]: Value: 3.61     Gradient:       1
```

torch

autograd by

Dynamic “while” evaluations

In [45]:
```
foo = function(a, b, c)
    local val = 0
    while b > c do
        val = val + a * math.sin(b)
        b = b - 0.1
    end
    return val
end
```

In [46]:
```
dfoo = autograd(foo)
```

In [47]:
```
da, f_abc = dfoo(3.5, 2.1, 1.1)
print('Value: ' .. f_abc, 'Gradient: ', da)
```

Out[47]:
```
Value: 33.468522197289  Gradient:       9.5624349135111
```

torch

autograd by

Dependency checks

In [45]:

```
foo = function(a, b, c)
    local val = 0
    while b > c do
        val = val + a * math.sin(b)
        b = b - 0.1
    end
    return val
end
```

In [46]:

```
dfoo = autograd(foo)
```

In [47]:

```
da, f_abc = dfoo(3.5, 2.1, 1.1)
print('Value: ' .. f_abc, 'Gradient: ', da)
```

Out[47]:

```
Value: 33.468522197289  Gradient:      9.5624349135111
```

n [48]:

```
da, f_abc = dfoo(3.5, 1.0, 1.1)
print('Value: ' .. f_abc, 'Gradient: ', da)
```

```
...all/share/lua/5.1/autograd/runtime/direct/DirectTape.lua:119: A node type was not returned. This is either because
a gradient was not defined, or the input is independent of the output
```

autograd by

- Dynamic graphs
- Auto-differentiated nn modules
- Tape based autograd

Torchnet

https://github.com/torchnet/torchnet

Next steps

- http://torch.ch/
- https://github.com/szagoruyko/idiap-tutorials