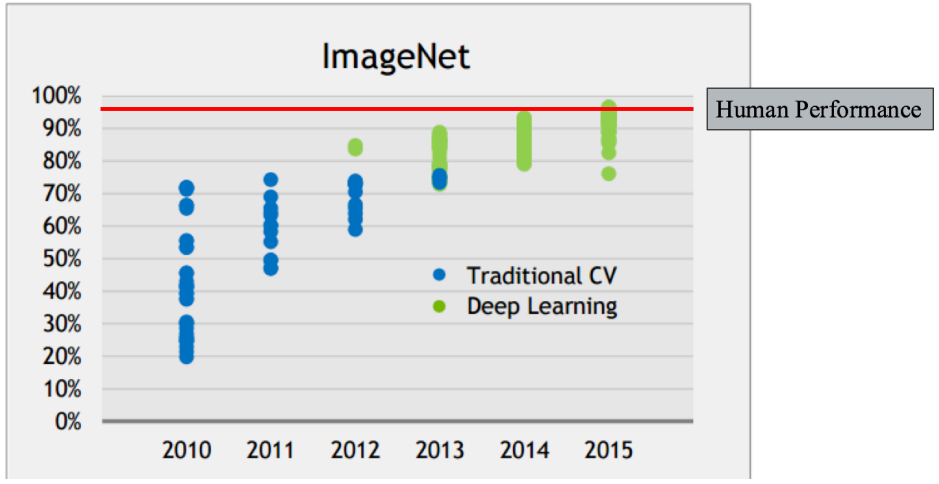
Deep Learning (for Computer Vision)

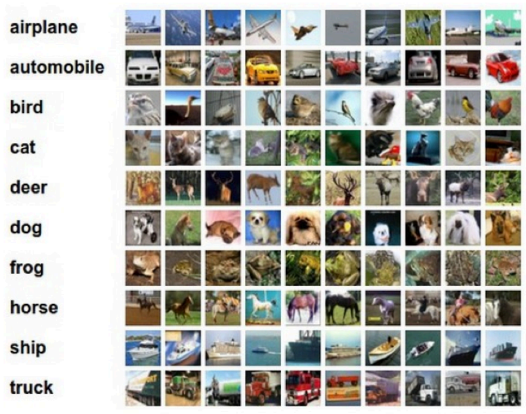
Arjun Jain | 25 March 2017

Recap

- Brief history
- Applications
- Data-driven paradigm, CIFAR-10 dataset
- NN classifier
- Linear Classifier:
  - Loss function
  - Optimization
  - Regularization
- Feed forward networks



10 labels
50,000 training images
10,000 test images.



Agenda this Week

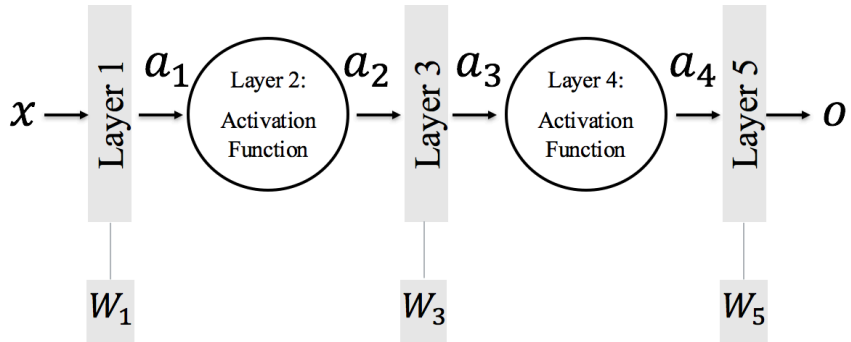
- Introduction, applications and achievements of Deep Learning
- The data driven paradigm: image classification using k-nearest neighbor and linear classification
- Optimization using gradient descent
- Feed forwards networks, training using back propagation, chain rule
- DNN Building Blocks
  - Activation Layers (ReLU, Sigmoid, etc.)
  - Fully Connected Layer
  - Convolution Layer
  - Max Pooling Layer

Sources

A lot of the material has been shamelessly and gratefully collected from:

- **http://cs231n.stanford.edu/**
- http://www.slideshare.net/kuwajima/cnnbp
- http://wiseodd.github.io/techblog/2016/07/18/convnet-maxpool-layer/
- https://github.com/e-lab/torch-toolbox/blob/master/Weight-init/weight-init.lua
- http://www.cs.huji.ac.il/~csip/tirgul3_derivatives.pdf
- http://cilvr.cs.nyu.edu/doku.php?id=deeplearning:slides:start#on-line_material_from_other_sources
- https://github.com/torch/nn/blob/31d7d2bc86a914e2a9e6b3874c497c60517dc853/doc/module.md
- https://web.stanford.edu/group/pdplab/pdphandbook/handbookch6.html
- http://neuralnetworksanddeeplearning.com/chap2.html

Multiple Layers –Feed Forward - Composition of Functions



$$a_1 = F(x, W_1),\ x \in \mathbb{R}^m$$

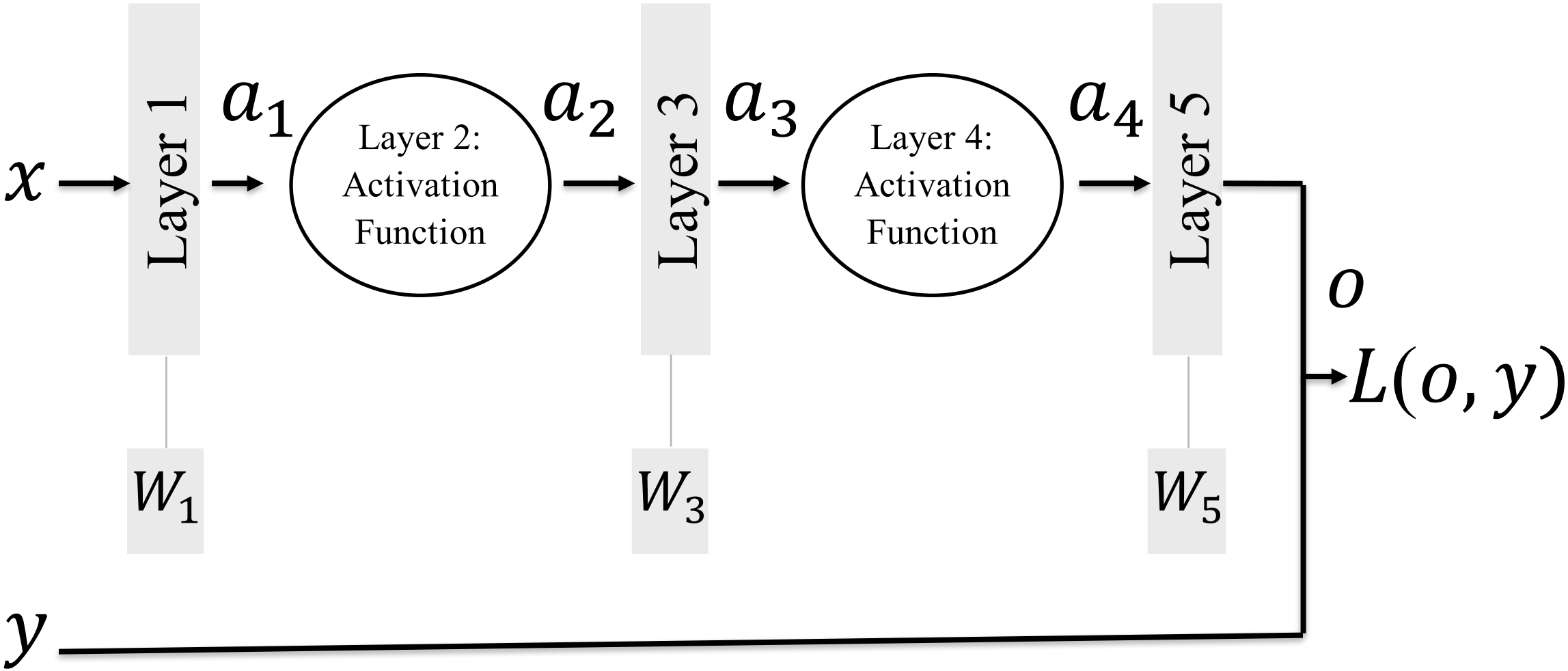
$$a_2 = G(a_1)$$

$$a_3 = H(a_2, W_3),$$

$$a_4 = J(a_3)$$

$$o = K(a_4, W_5) = K(J(H(G(F(x, W_1)), W_3)), W_5) \in \mathbb{R}^n$$

Multiple Layers – Feed Forward - Loss



30 sec. Vector Calculus Refresher

Let $x \in R^n$ (a column vector) and let $f : R^n \rightarrow R$. The derivative of f with respect to x is the row vector:

$$\frac{\partial f}{\partial x} = (\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n})$$

$\frac{\partial f}{\partial x}$ is called the gradient of f.

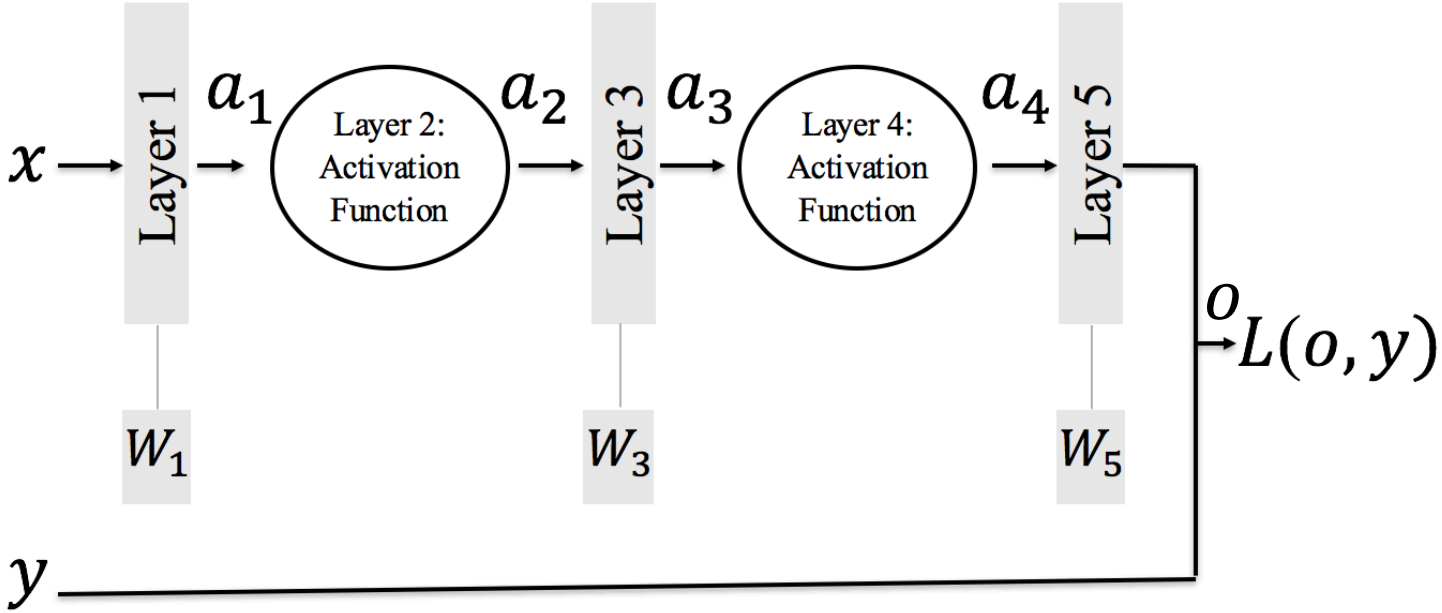
Let $x \in R^n$ (a column vector) and let $f : R^n \rightarrow R^m$. The derivative of f with respect to x is the $m \times n$ matrix:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f(x)_1}{\partial x_1} & \cdots & \frac{\partial f(x)_1}{\partial x_n} \\ \vdots & & \\ \frac{\partial f(x)_m}{\partial x_1} & \cdots & \frac{\partial f(x)_m}{\partial x_n} \end{bmatrix} \quad (2)$$

$\frac{\partial f}{\partial x}$ is called the Jacobian matrix of f.

sip/tirgul3_derivatives.pdf

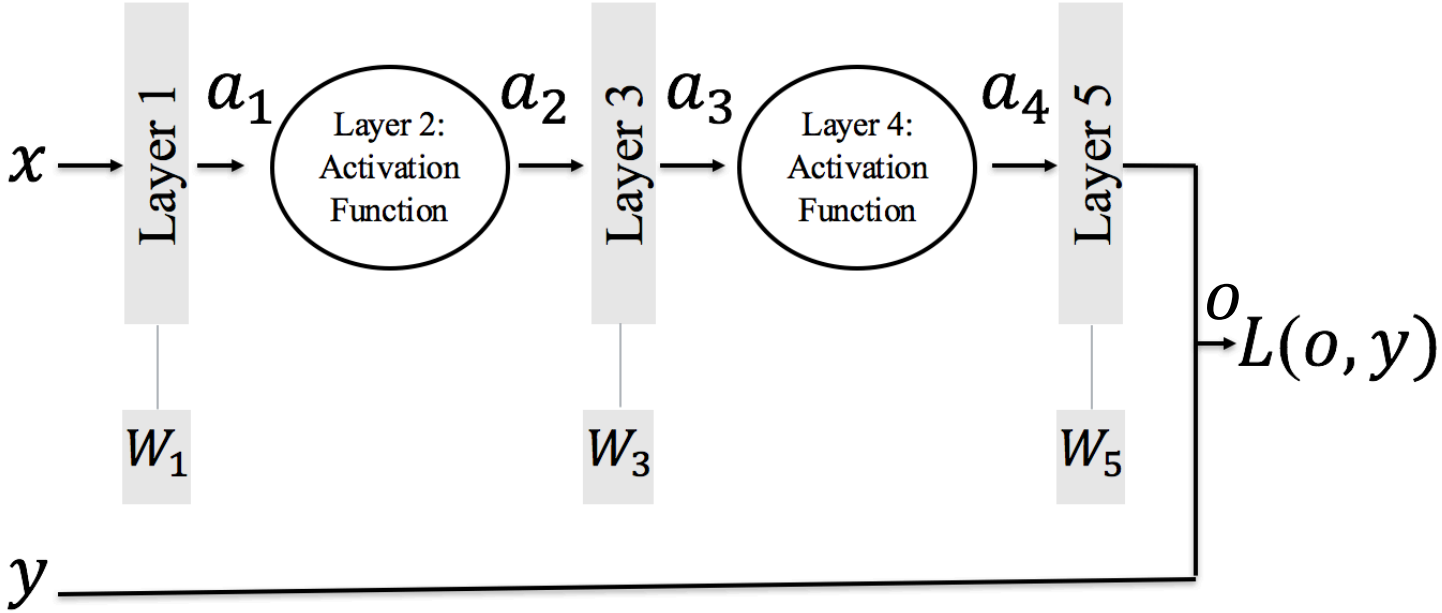
Multiple Layers – Back Prop: Chain Rule



We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

Multiple Layers – Back Prop: Chain Rule



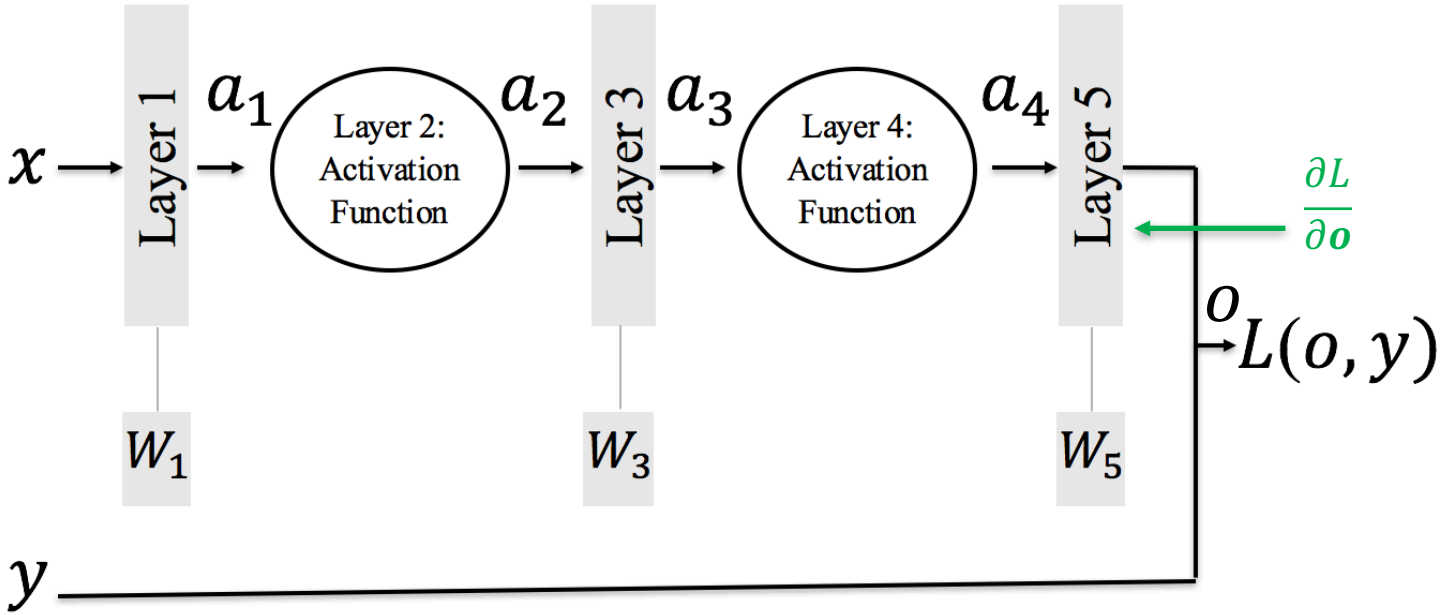
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

Compute:

$$\frac{\partial L}{\partial \boldsymbol{o}}$$

Multiple Layers – Back Prop: Chain Rule



We want:

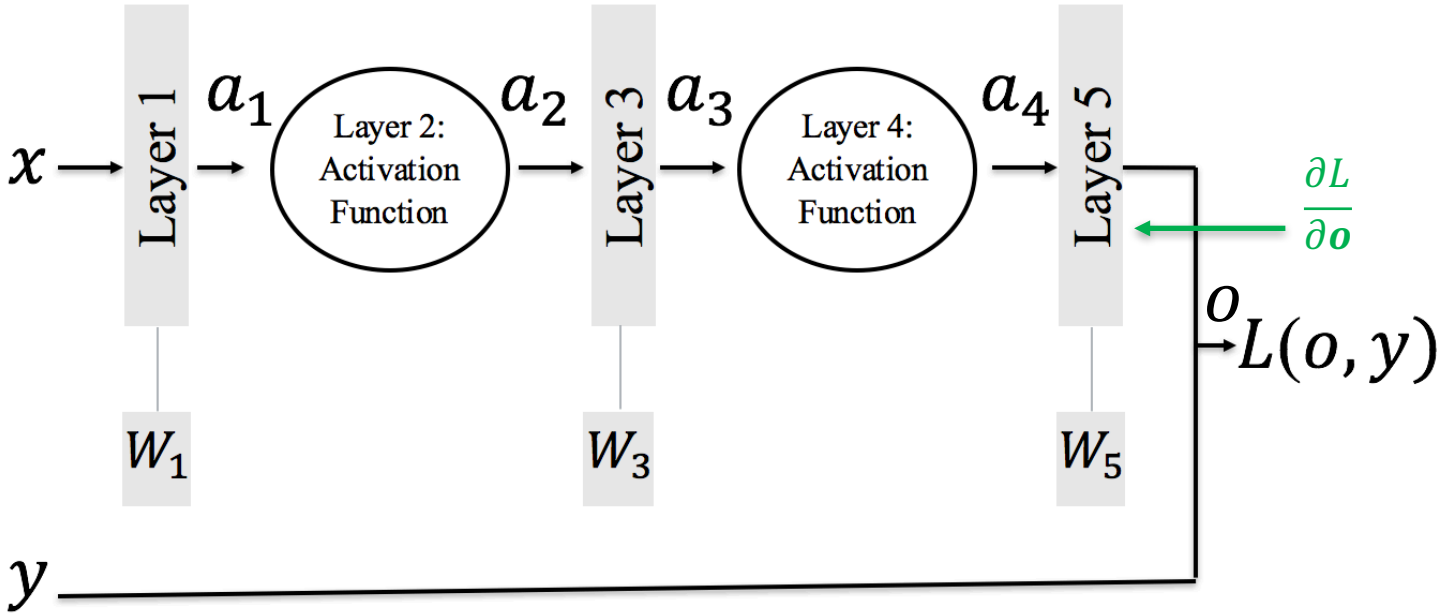
$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

Compute:

$$\frac{\partial L}{\partial \boldsymbol{o}}$$

E.g: $L(\boldsymbol{o}, \boldsymbol{y}) = {}^1/_2 \parallel \boldsymbol{o} - \boldsymbol{y} \parallel^2$ then: $\frac{\partial L}{\partial \boldsymbol{o}}$

Multiple Layers – Back Prop: Chain Rule



We want:

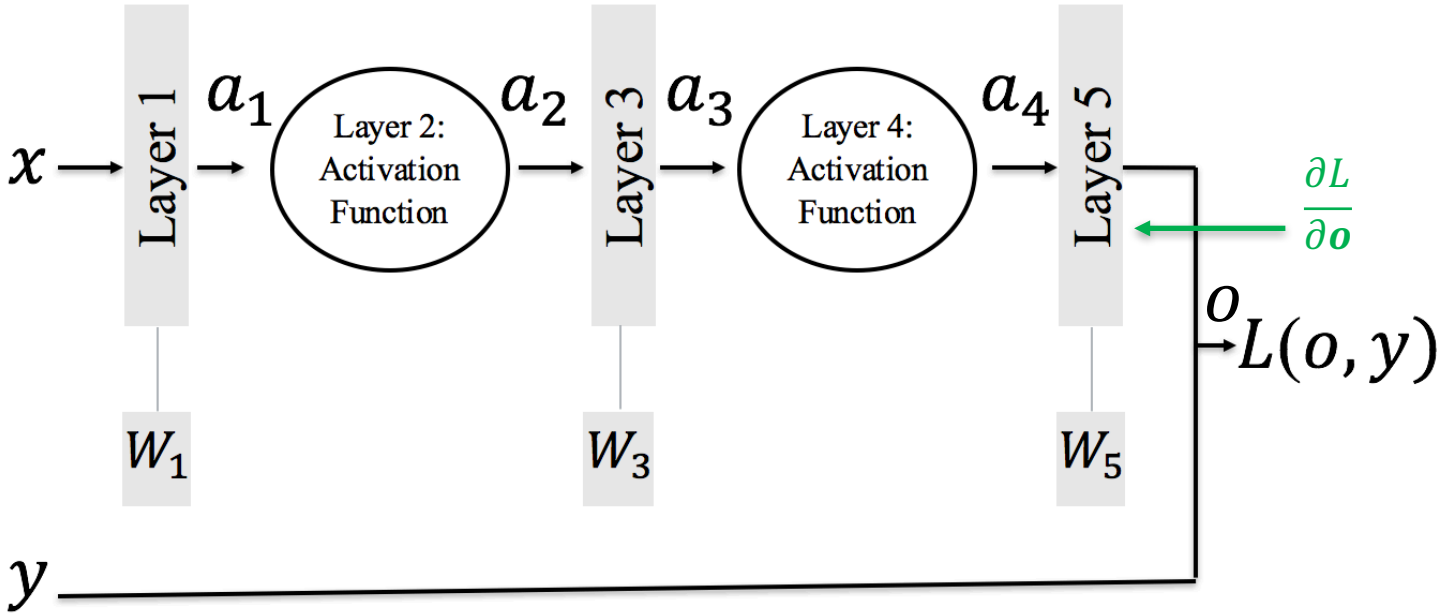
$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

Compute:

$$\frac{\partial L}{\partial \boldsymbol{o}}$$

E.g: $L(\boldsymbol{o}, \boldsymbol{y}) = {}^1/_2 \parallel \boldsymbol{o} - \boldsymbol{y} \parallel^2$ then: $\frac{\partial L}{\partial \boldsymbol{o}} = (\boldsymbol{o} - \boldsymbol{y})$

Multiple Layers – Back Prop: Chain Rule



We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

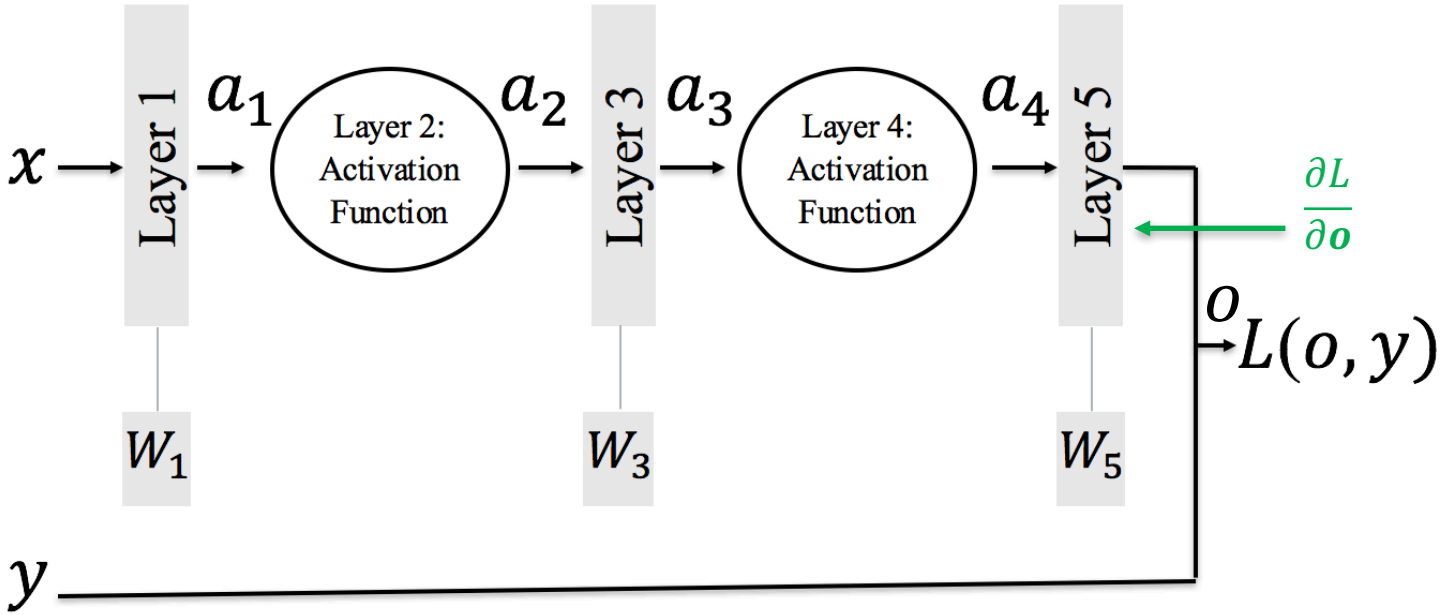
Compute:

$$\frac{\partial L}{\partial \boldsymbol{o}}$$

$$\frac{\partial L}{\partial \boldsymbol{o}} \in \mathbb{R}^{1 \times n}$$

E.g: $L(\boldsymbol{o}, \boldsymbol{y}) = {}^1/_2 \parallel \boldsymbol{o} - \boldsymbol{y} \parallel^2$ then: $\frac{\partial L}{\partial \boldsymbol{o}} = (\boldsymbol{o} - \boldsymbol{y})$

Multiple Layers – Back Prop: Chain Rule



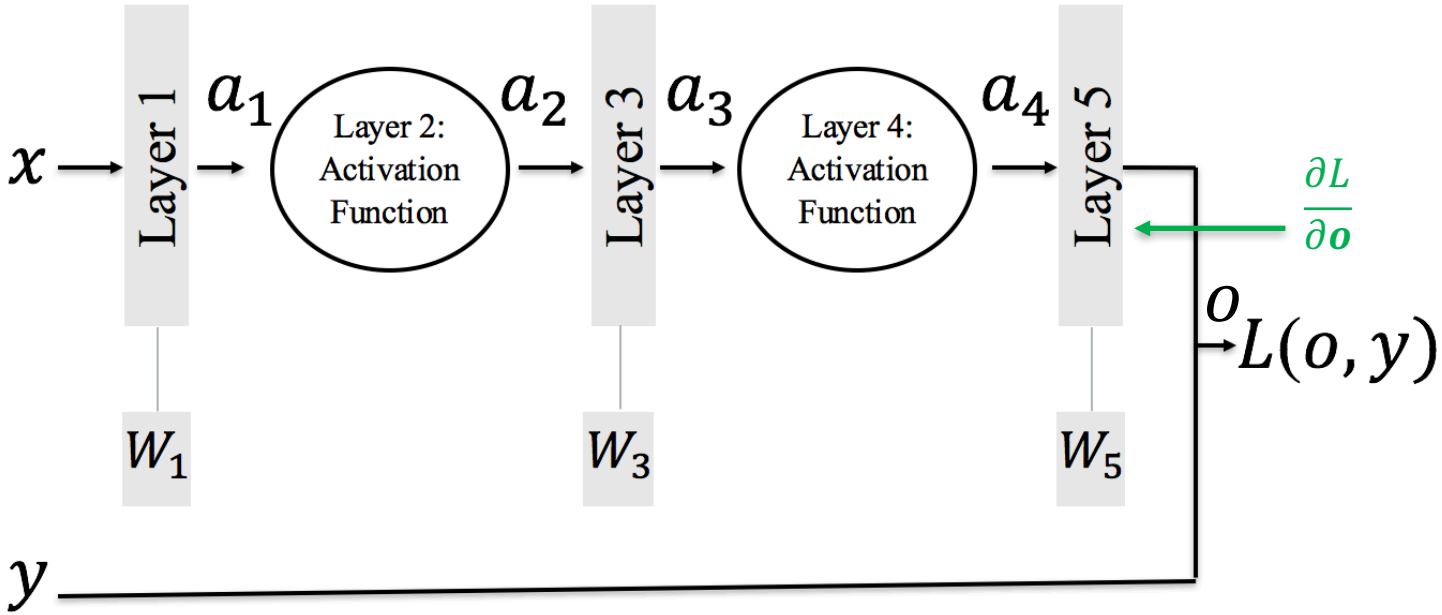
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

We can also compute:

$$\frac{\partial L}{\partial \boldsymbol{o}}, \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}$$

Multiple Layers – Back Prop: Chain Rule



Some differentiable function with respect to a_4 and W_5

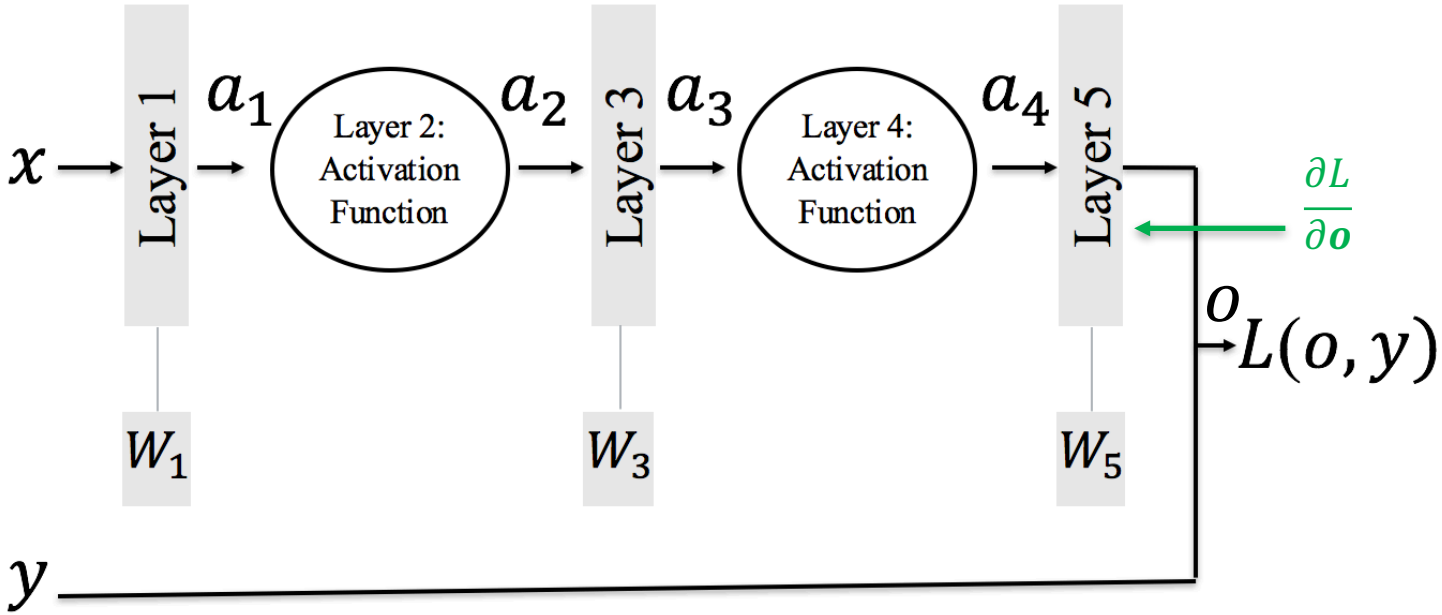
We want: $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$

We can also compute: $\frac{\partial L}{\partial o}, \frac{\partial o}{\partial W_5}$

since: $o = K(a_4, W_5)$

then: $\frac{\partial o}{\partial W_5}$ is a Jacobian of size $\mathbb{R}^{n \times \dim(W_5)}$

Multiple Layers – Back Prop: Chain Rule



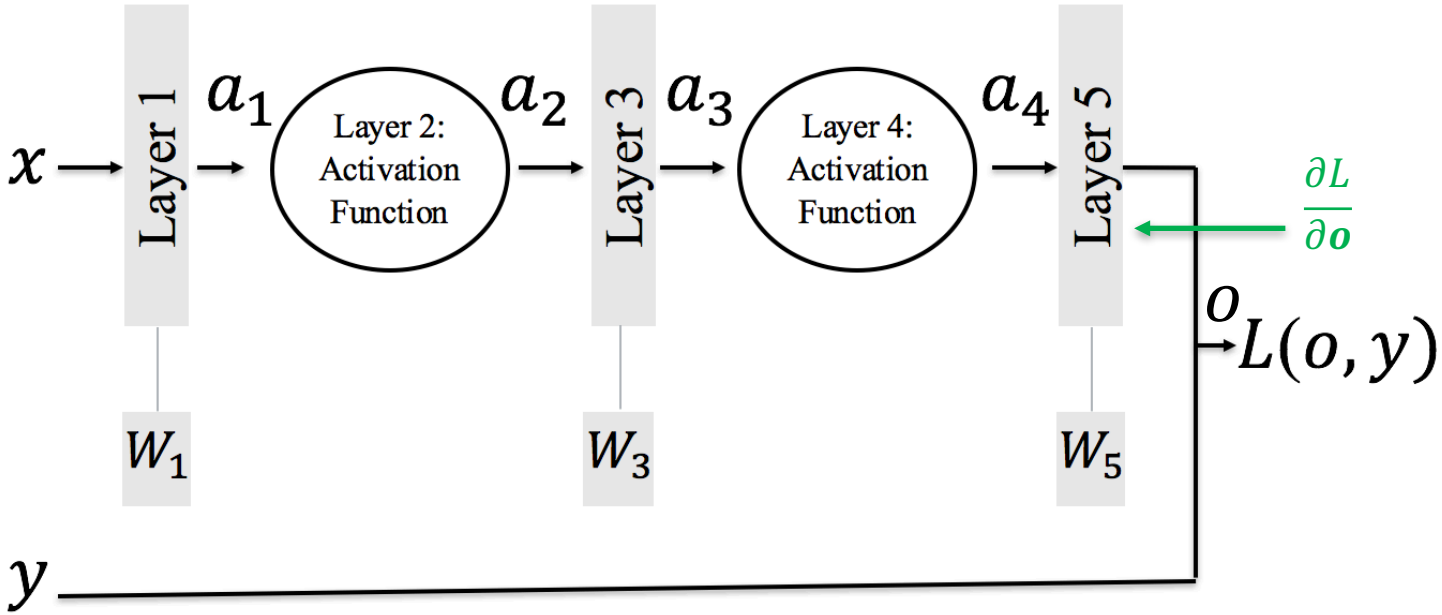
We want: $\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$

We can also compute: $\frac{\partial L}{\partial \boldsymbol{o}}, \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}$

since: $o = K(a_4, W_5)$

then: $\left[\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}\right]_{kl} = \frac{\partial [K(a_4, \boldsymbol{W_5})]_k}{\partial [\boldsymbol{W_5}]_l}$

Multiple Layers – Back Prop: Chain Rule



Element (k, l) of the jacobian indicates how much the k-th output wiggles when we wiggle the l-th weight

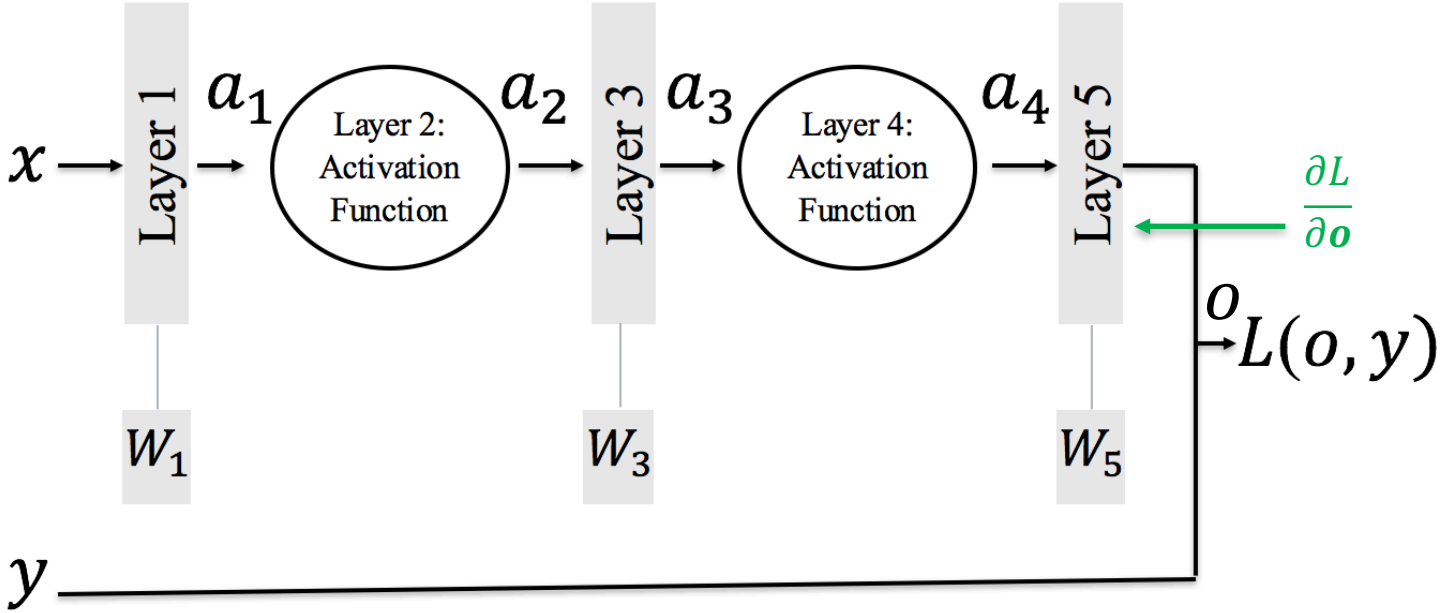
We want: $\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$

We can also compute: $\frac{\partial L}{\partial \boldsymbol{o}}, \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}$

since: $o = K(a_4, W_5)$

then: $\left[\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}\right]_{kl} = \frac{\partial [K(\boldsymbol{a_4}, \boldsymbol{W_5})]_k}{\partial [\boldsymbol{W_5}]_l}$

Multiple Layers – Back Prop: Chain Rule



We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

We can also compute:

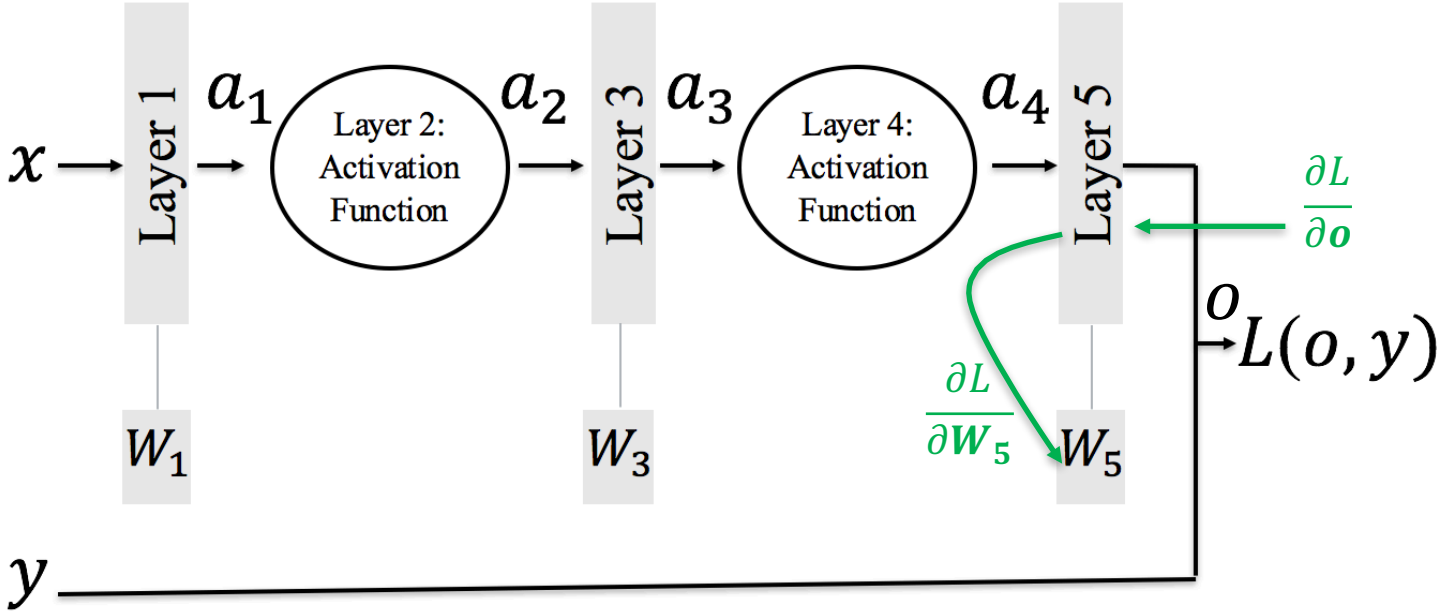
$$\frac{\partial L}{\partial \boldsymbol{W_5}} = \frac{\partial L}{\partial \boldsymbol{o}} \times \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}$$

Remember:

$$\frac{\partial L}{\partial \boldsymbol{o}} \in \mathbb{R}^{1 \times n}$$

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}} \in \mathbb{R}^{n \times \dim(W_5)}$$

Multiple Layers – Back Prop: Chain Rule



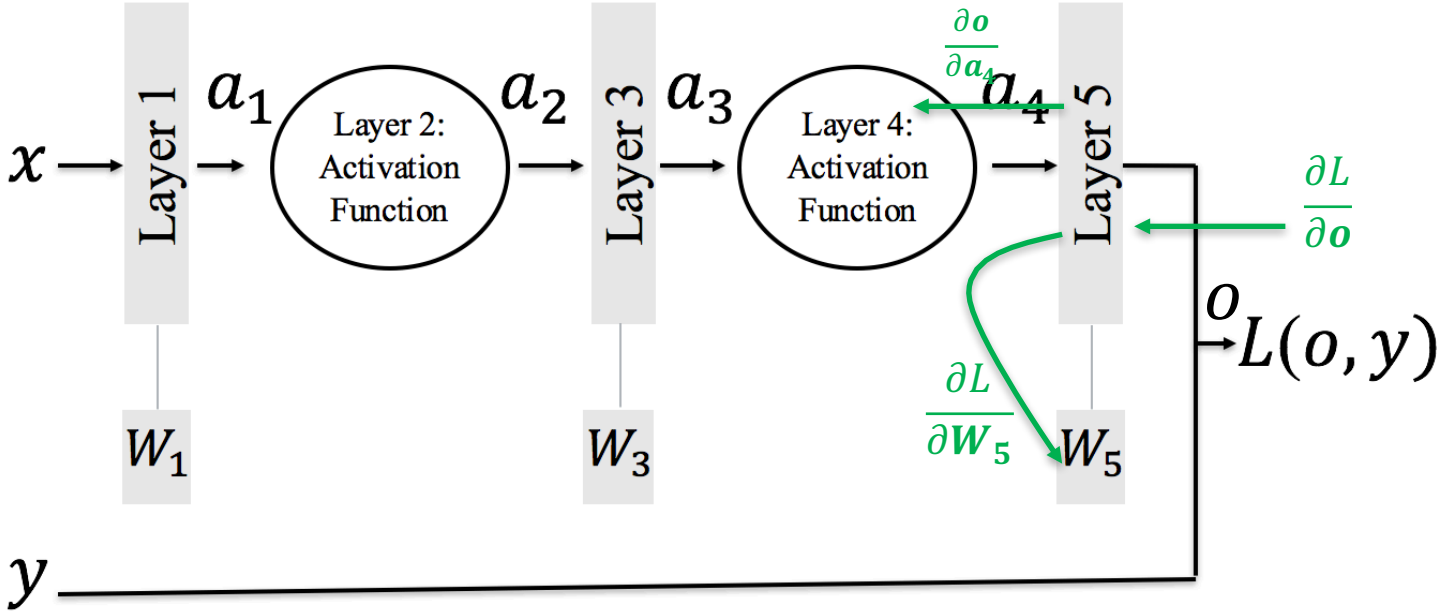
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

We know:

$$\frac{\partial L}{\partial \boldsymbol{W_5}} = \frac{\partial L}{\partial \boldsymbol{o}} \times \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}} \epsilon \, \mathbb{R}^{1 \times \dim(W_5)}$$

Multiple Layers – Back Prop: Chain Rule



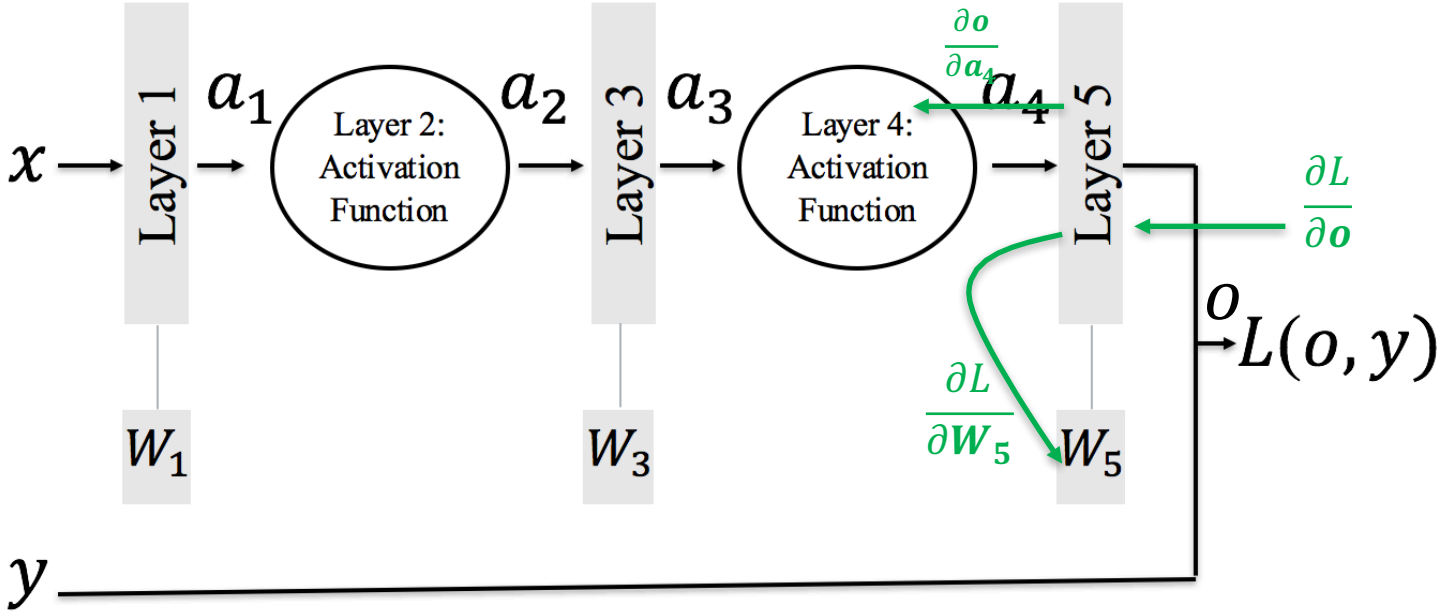
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

since: $\boldsymbol{o = K(a_4, W_5)}$

then: $\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}}$ is a Jacobian of size $\mathbb{R}^{n \times \dim(a_4)}$

Multiple Layers – Back Prop: Chain Rule



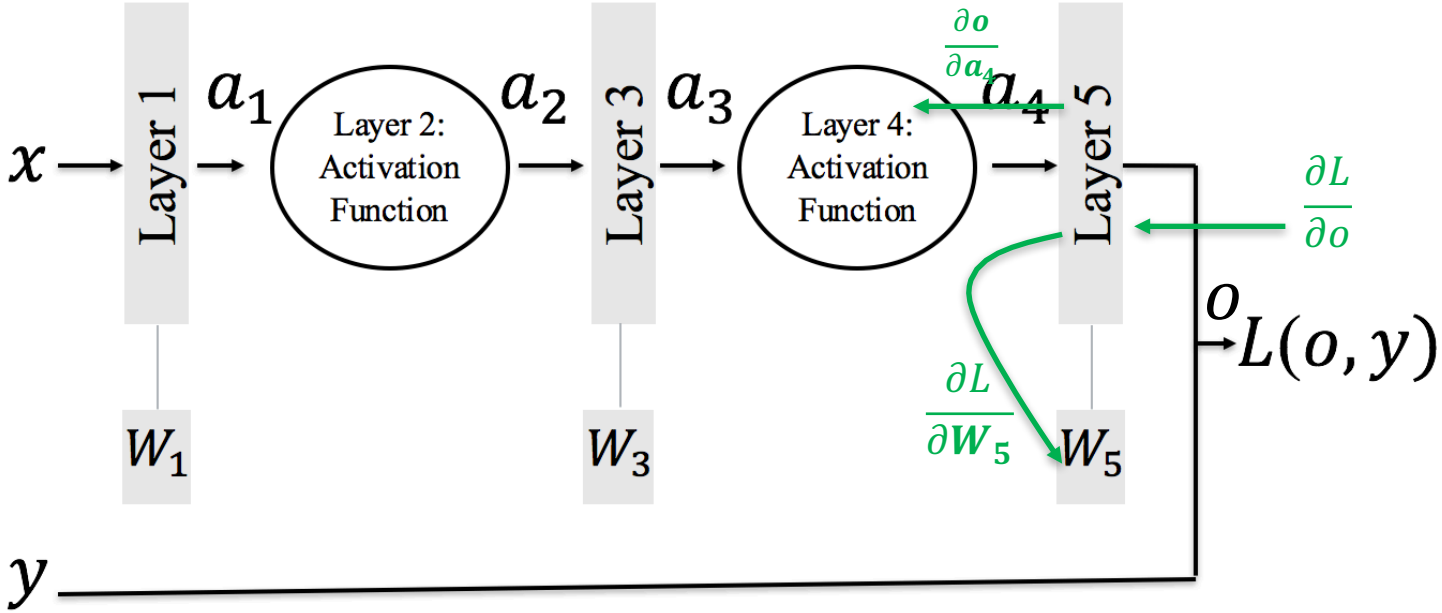
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

since: $\boldsymbol{o = K(a_4, W_5)}$

then: $$\left[\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}}\right]_{kl} = \frac{\partial [K(\boldsymbol{a_4}, \boldsymbol{W_5})]_k}{\partial [\boldsymbol{a_4}]_l}$$

Multiple Layers – Back Prop: Chain Rule



Element (k, l) of the jacobian indicates how much the k-th output wiggles when we wiggle the l-th output of the previous layer (or input to this layer)

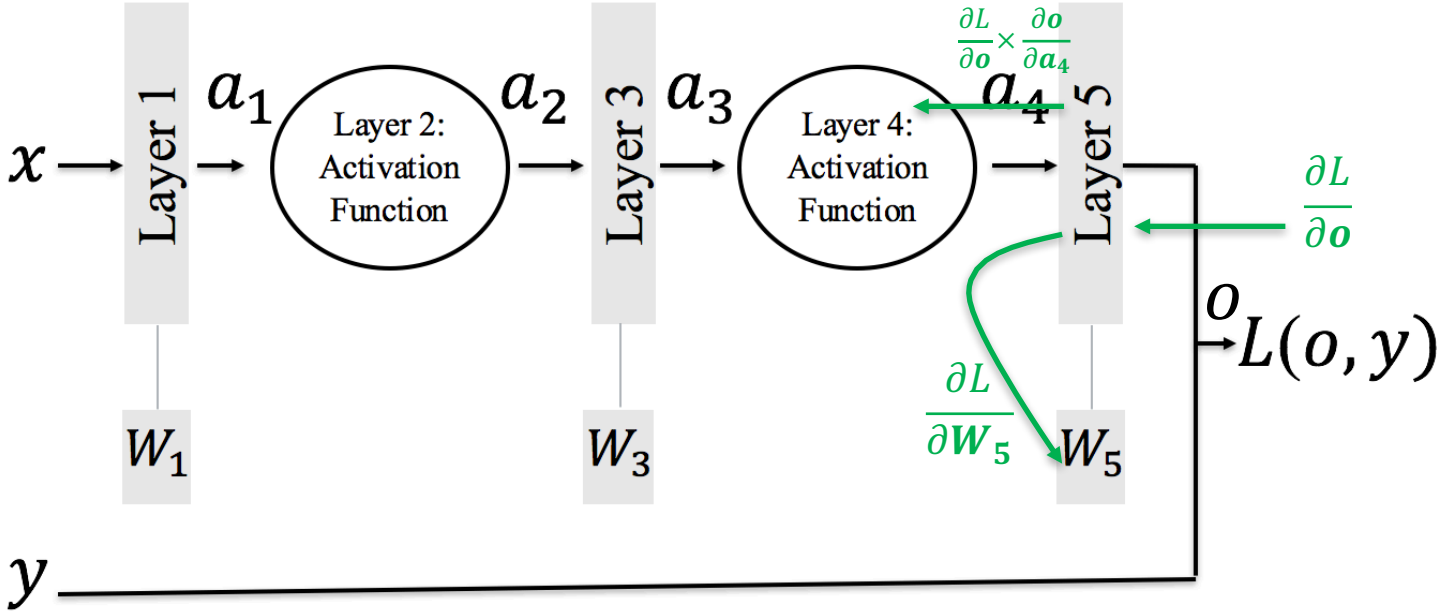
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

since: $\boldsymbol{o} = K(\boldsymbol{a_4}, \boldsymbol{W_5})$

then: $$\left[\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}}\right]_{kl} = \frac{\partial [K(\boldsymbol{a_4}, \boldsymbol{W_5})]_k}{\partial [\boldsymbol{a_4}]_l}$$

Multiple Layers – Back Prop: Chain Rule



We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

Backpropagate:

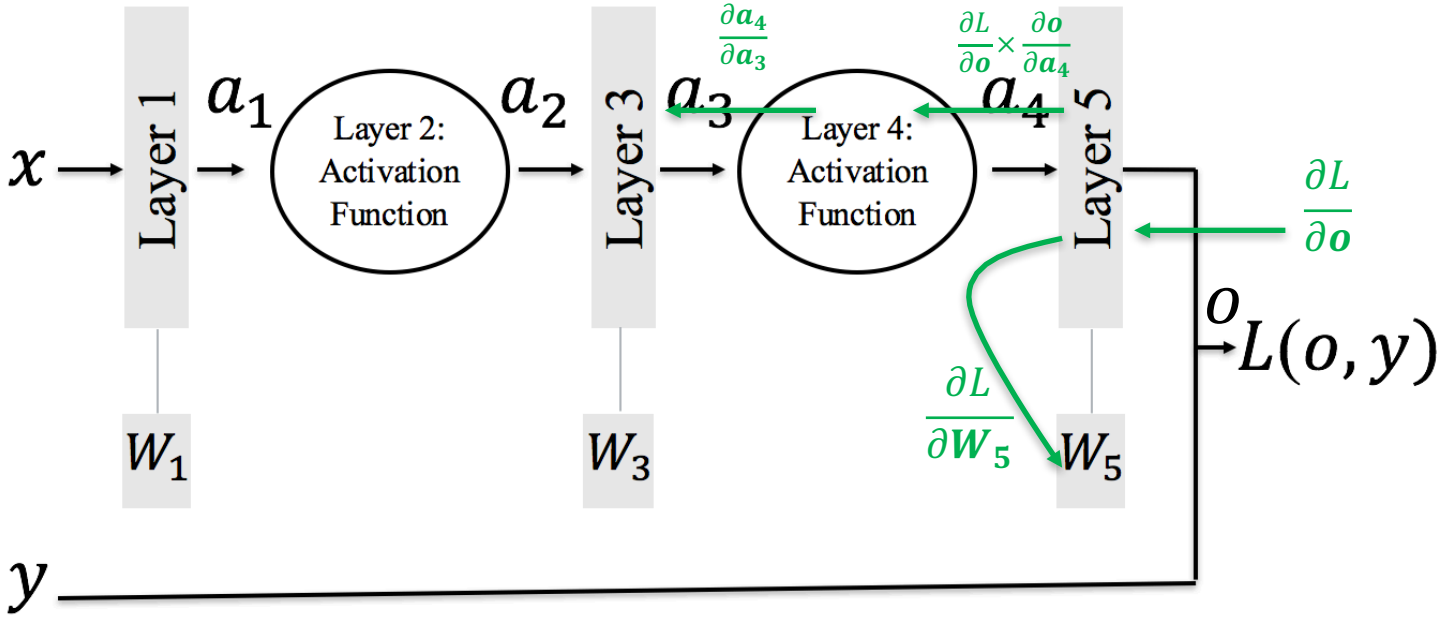
$$\frac{\partial L}{\partial \boldsymbol{o}} \times \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}} \in \mathbb{R}^{1 \times \dim(a_4)}$$

Remember:

$$\frac{\partial L}{\partial \boldsymbol{o}} \in \mathbb{R}^{1 \times n}$$

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}} \in \mathbb{R}^{n \times \dim(a_4)}$$

Multiple Layers – Back Prop: Chain Rule



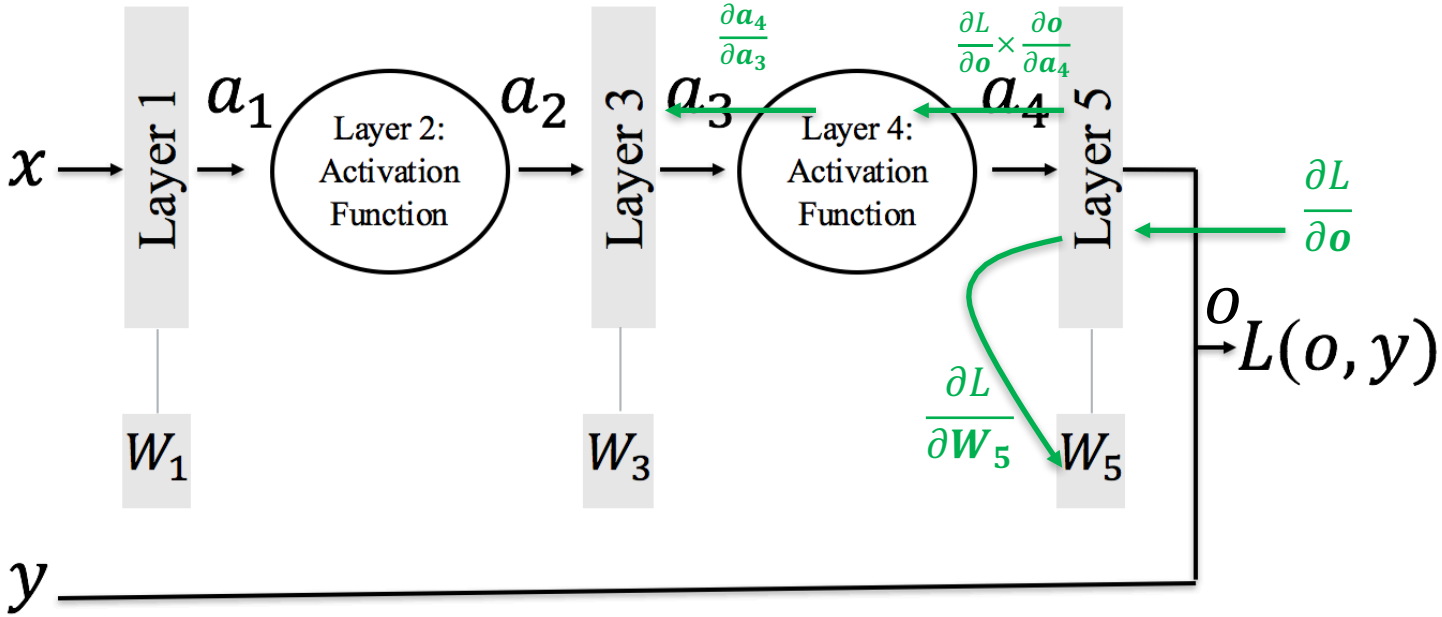
We want:

$$\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$$

since: $a_4 = J(a_3)$

then: $\frac{\partial a_4}{\partial a_3}$ is a Jacobian of size $\mathbb{R}^{\dim(a_4) \times \dim(a_3)}$

Multiple Layers – Back Prop: Chain Rule



Remember:

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}} \times \frac{\partial L}{\partial \boldsymbol{o}} \in \mathbb{R}^{1 \times \dim(a_4)}$$

$$\frac{\partial \boldsymbol{a_4}}{\partial \boldsymbol{a_3}} \in \mathbb{R}^{\dim(a_4) \times \dim(a_3)}$$

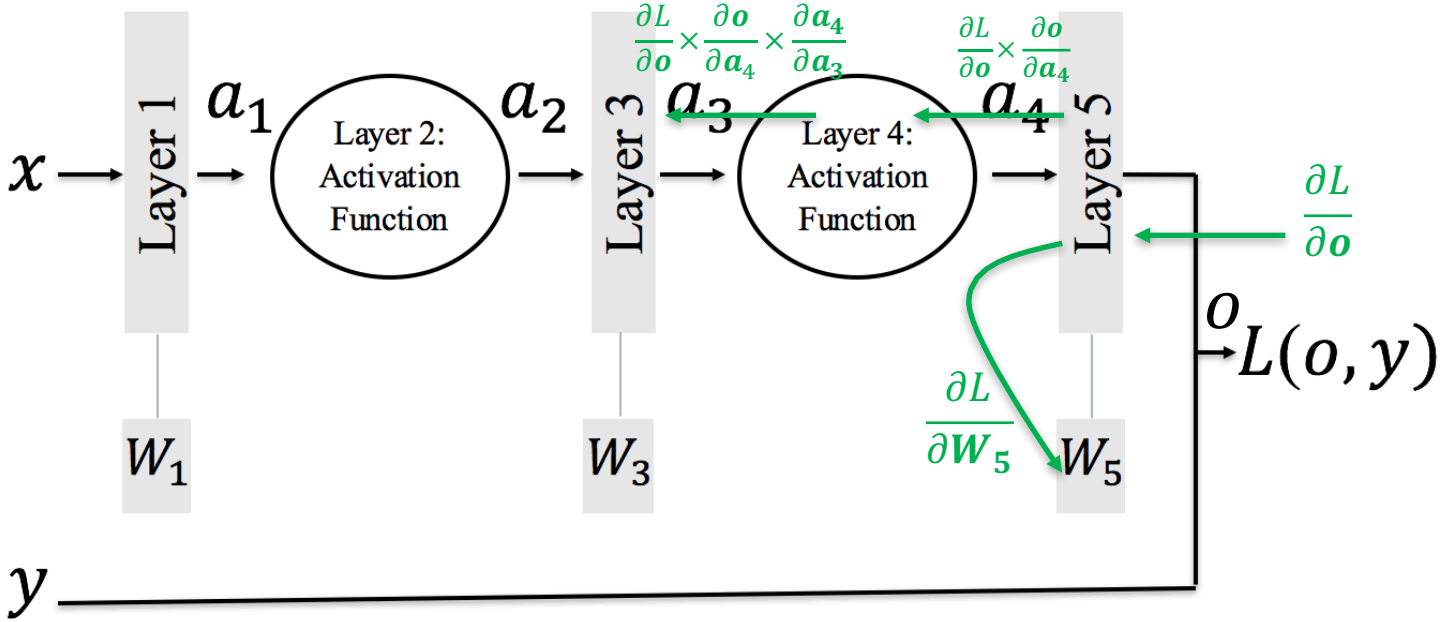
We want:

$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

since: $\boldsymbol{a_4 = J(a_3)}$

then: $\frac{\partial \boldsymbol{a_4}}{\partial \boldsymbol{a_3}}$ is a Jacobian of size $\mathbb{R}^{\dim(a_4) \times \dim(a_3)}$

Multiple Layers – Back Prop: Chain Rule



Remember:

$$\frac{\partial o}{\partial a_4} \times \frac{\partial L}{\partial o} \in \mathbb{R}^{1 \times \dim(a_4)}$$

$$\frac{\partial a_4}{\partial a_3} \in \mathbb{R}^{\dim(a_4) \times \dim(a_3)}$$

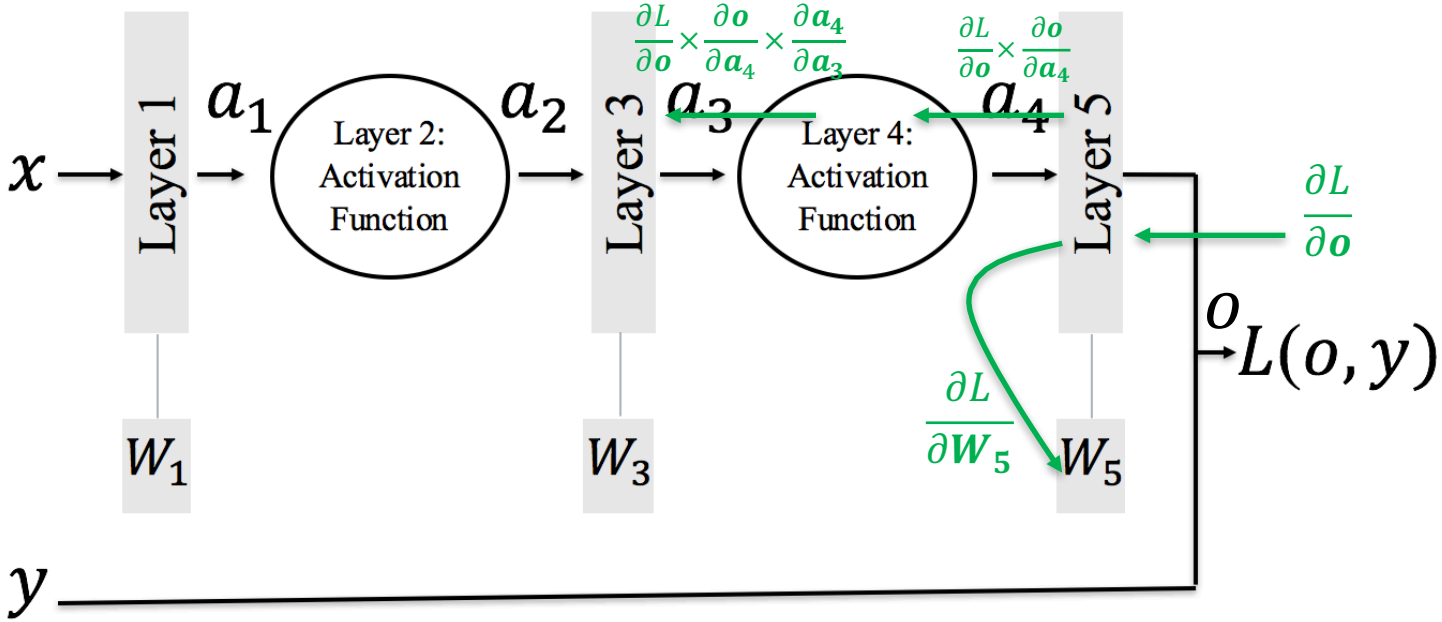
We want:

$$\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$$

Backpropagate:

$$\frac{\partial L}{\partial o} \times \frac{\partial o}{\partial a_4} \times \frac{\partial a_4}{\partial a_3} \in \mathbb{R}^{1 \times \dim(a_3)}$$

Multiple Layers – Back Prop: Chain Rule



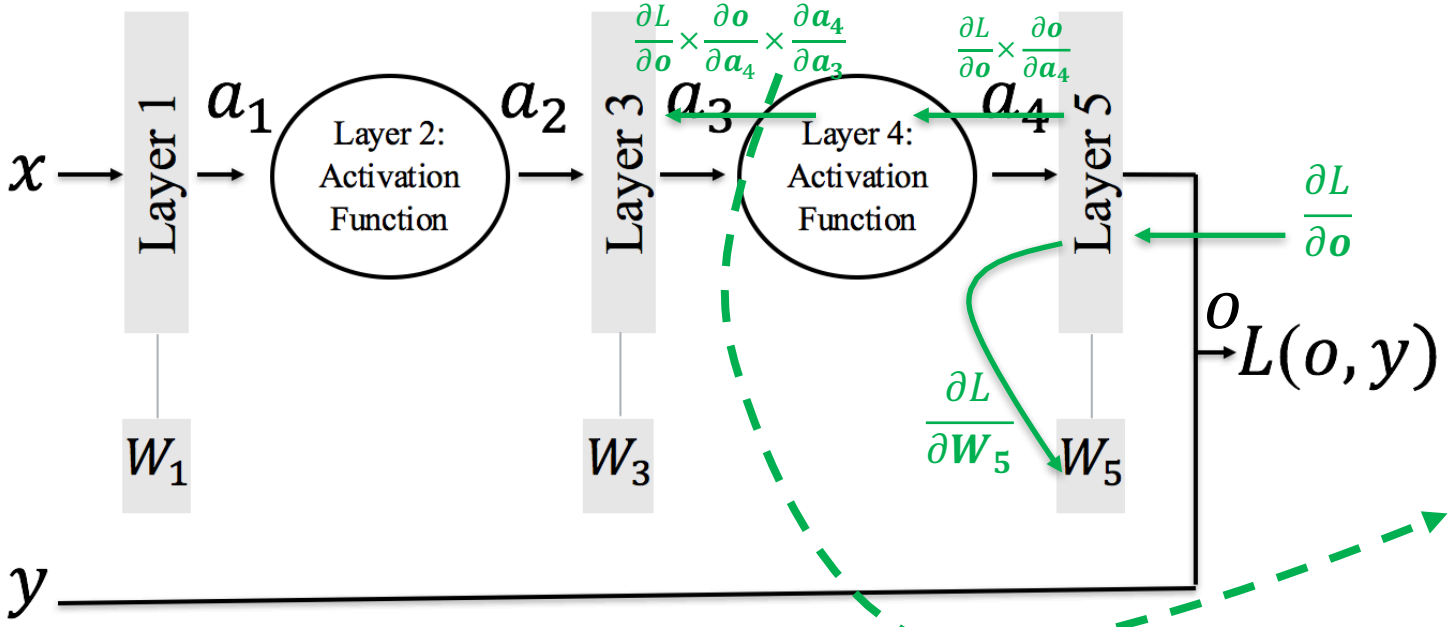
We want:

$$\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$$

since: $a_3 = K(a_2, W_3)$

then: $\frac{\partial a_3}{\partial W_3}$ is a Jacobian of size $\mathbb{R}^{\dim(a_3) \times \dim(W_3)}$

Multiple Layers – Back Prop: Chain Rule



Remember:

$$\frac{\partial L}{\partial o} \times \frac{\partial o}{\partial a_4} \times \frac{\partial a_4}{\partial a_3} \in \mathbb{R}^{1 \times \dim(a_3)}$$

$$\frac{\partial a_3}{\partial W_3} \in \mathbb{R}^{\dim(a_3) \times \dim(W_3)}$$

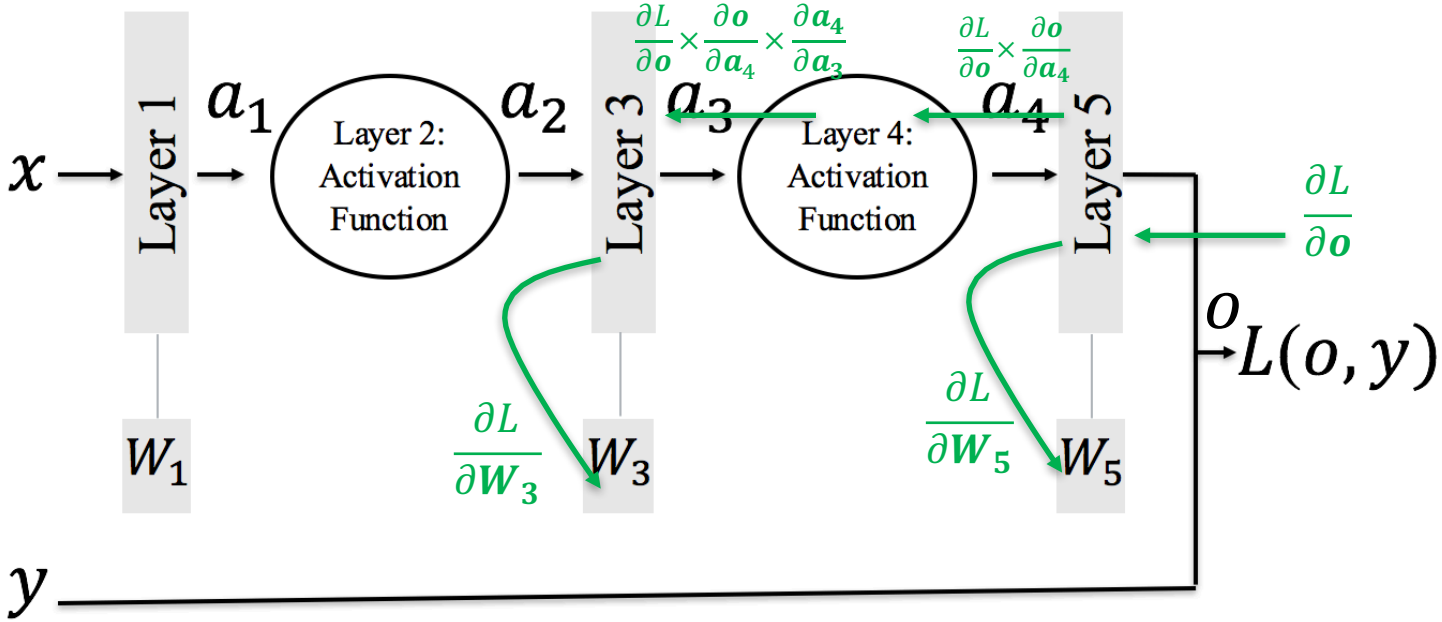
We want:

$$\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$$

Now we can compute:

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o} \times \frac{\partial o}{\partial a_4} \times \frac{\partial a_4}{\partial a_3} \times \frac{\partial a_3}{\partial W_3}$$

Multiple Layers – Back Prop: Chain Rule



Remember:

$$\frac{\partial L}{\partial o} \times \frac{\partial o}{\partial a_4} \times \frac{\partial a_4}{\partial a_3} \in \mathbb{R}^{1 \times \dim(a_3)}$$

$$\frac{\partial a_3}{\partial W_3} \in \mathbb{R}^{\dim(a_3) \times \dim(W_3)}$$

We want:

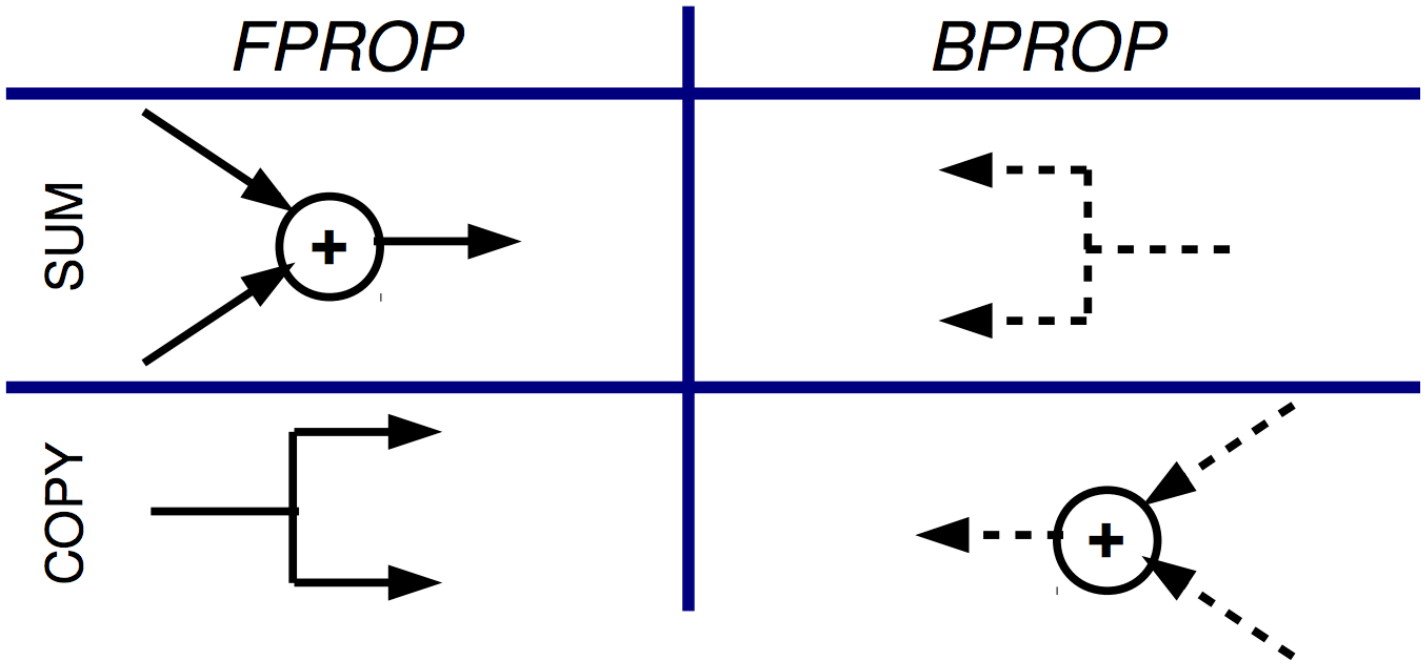
$$\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$$

Now we can compute:

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o} \times \frac{\partial o}{\partial a_4} \times \frac{\partial a_4}{\partial a_3} \times \frac{\partial a_3}{\partial W_3}$$

Multiple Layers – Back Prop: Chain Rule

FPROP and BPROP are duals of each other:



Why Torch7

Torch7 is based on the Lua language

- Simple and lightweight scripting language, dominant in the game industry
- Has a native just-in-time compiler (write double loops!)
- Has a simple foreign function interface to call C/C++ functions from Lua

Torch7 is an extension of Lua with

- A multidimensional array engine with CUDA and OpenMP backends
- A machine learning library that implements multilayer nets, convolutional nets, unsupervised pre-training, etc
- Various libraries for data/image manipulation and computer vision
- A quickly growing community of users

Simple installation on Ubuntu and Mac OSX:

- `git clone https://github.com/torch/distro.git ~/torch --recursive cd ~/torch; bash install-deps; ./install.sh`

Why Torch7

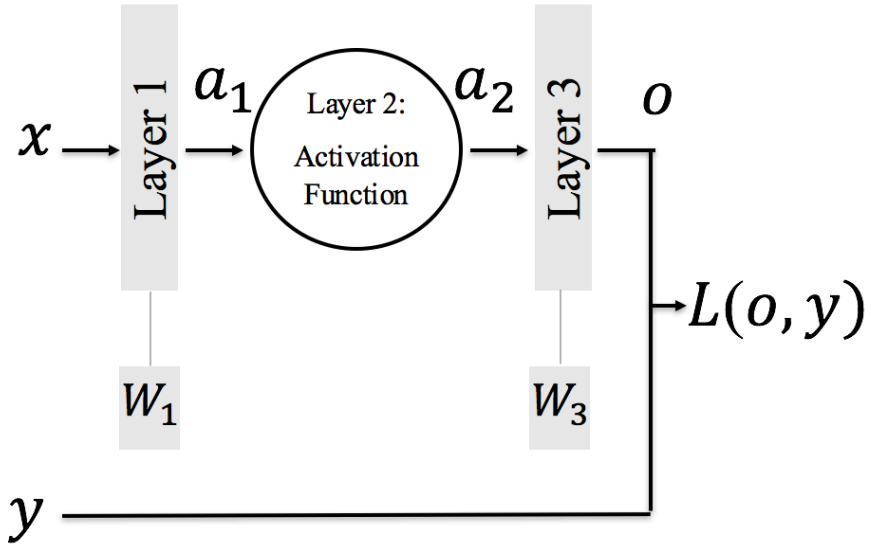
Andrej Karpathy, Machine Learning PhD student at Stanford

Written Sep 9, 2016 · Upvoted by Apurv Verma, ML@GATech and Manohar Kuse, PhD Candidate researching computer vision and machine learning in robotics.

I then switched to Torch which I liked a lot and still like. Torch is simple: there is this Tensor object that you can do various operations on transparently either on CPU or GPU and then there is this thin deep-learning-specific wrapper around that. You can understand almost everything under you, inspect it, change it, it makes sense. It's a lot of power in a lightweight package.

I now use TensorFlow, as does everyone else at OpenAI. If I have to be honest, I feel like my code complexity has increased and I spend more time debugging than what I'm used to with Torch. This could also be temporary, while I'm still learning.

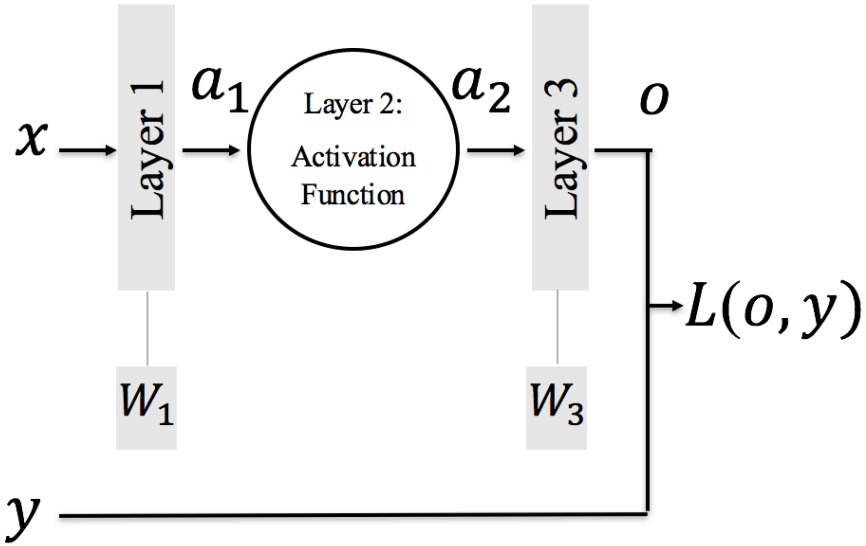
Multiple Layers – Feed Forward – In Torch7



- Example: 3 modules layer1, layer2, layer3
- By hand:
  - `a1 = layer1:forward(x)`
  - `a2 = layer2:forward(a1)`
  - `o = layer3:forward(a2)`
- Using `nn.Sequential`:
  - `model = nn.Sequential()`
  - `model:add(layer1)`
  - `model:add(layer2)`
  - `model:add(layer3)`
  - `o = model:forward(x)`

  (output is returned, but also stored internally)

Multiple Layers – Feed Forward – In Torch7



- `criterion = nn.SomeCriterion()`
- `loss = criterion:forward(o, y)`
- `dl_do = criterion:backward(o, y)`
- Gradient with respect to input is returned
- Arguments are input and gradient with respect to outpu
- By hand:
  - `l3_grad = layer3:backward(a2, dl_do)`
  - `l2_grad = layer2:backward(a1, l3_grad)`
  - `l1_grad = layer1:backward (x, l2_grad)`
- Using nn.Sequential:
  - `l1_grad = model:backward(x, dl_do)`

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```lua
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
   self._type = self.output:type()
end

function Module:updateOutput(input)
   return self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end

function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

```lua
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end

function Module:zeroGradParameters()
   local _,gradParams = self:parameters()
   if gradParams then
      for i=1,#gradParams do
         gradParams[i]:zero()
      end
   end
end
```

- Module is an abstract class which defines fundamental methods necessary for a training a neural network
- Modules contain two states variables: **output** and **gradInput**

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```lua
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
   self._type = self.output:type()
end

function Module:updateOutput(input)
   return self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end

function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

```lua
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end

function Module:zeroGradParameters()
   local _,gradParams = self:parameters()
   if gradParams then
      for i=1,#gradParams do
         gradParams[i]:zero()
      end
   end
end
```

- Takes an input and computes the corresponding output of the module. In general input and output are **Tensors**
- After a forward(), the output state variable should have been updated to the new value

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```lua
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
   self._type = self.output:type()
end

function Module:updateOutput(input)
   return self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end

function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

```lua
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end

function Module:zeroGradParameters()
   local _,gradParams = self:parameters()
   if gradParams then
      for i=1,#gradParams do
         gradParams[i]:zero()
      end
   end
end
```

- It is not advised to override this function. Instead, one should implement updateOutput(input) function. The forward module in the abstract parent class Module will call updateOutput(input)

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```lua
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
   self._type = self.output:type()
end

function Module:updateOutput(input)
   return self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end

function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

```lua
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end

function Module:zeroGradParameters()
   local _,gradParams = self:parameters()
   if gradParams then
      for i=1,#gradParams do
         gradParams[i]:zero()
      end
   end
end
```

- **backward()** Performs a backpropagation step through the module, with respect to the given input.
- A backpropagation step consist in computing two kind of gradients: $\frac{\partial a_l}{\partial a_{l-1}}$ & $\frac{\partial a_l}{\partial W_l}$ **Remember?**

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```lua
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
   self._type = self.output:type()
end

function Module:updateOutput(input)
   return self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end

function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

```lua
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end

function Module:zeroGradParameters()
   local _,gradParams = self:parameters()
   if gradParams then
      for i=1,#gradParams do
         gradParams[i]:zero()
      end
   end
end
```

- A function call to updateGradInput(input, gradOutput)
- A function call to accGradParameters(input, gradOutput, scale)

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```lua
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
   self._type = self.output:type()
end

function Module:updateOutput(input)
   return self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end

function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

```lua
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end

function Module:zeroGradParameters()
   local _,gradParams = self:parameters()
   if gradParams then
      for i=1,#gradParams do
         gradParams[i]:zero()
      end
   end
end
```

- What if module has no parameters (such as nn.ReLU)? It does nothing

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```lua
local Module = torch.class('nn.Module')

function Module:__init()
   self.gradInput = torch.Tensor()
   self.output = torch.Tensor()
   self._type = self.output:type()
end

function Module:updateOutput(input)
   return self.output
end

function Module:forward(input)
   return self:updateOutput(input)
end

function Module:backward(input, gradOutput, scale)
   scale = scale or 1
   self:updateGradInput(input, gradOutput)
   self:accGradParameters(input, gradOutput, scale)
   return self.gradInput
end
```

```lua
function Module:updateGradInput(input, gradOutput)
   return self.gradInput
end

function Module:accGradParameters(input, gradOutput, scale)
end

function Module:zeroGradParameters()
   local _,gradParams = self:parameters()
   if gradParams then
      for i=1,#gradParams do
         gradParams[i]:zero()
      end
   end
end
```

- After every update, clear the accumulated gradients

In Torch7:

https://github.com/torch/nn/blob/master/Module.lua

```
local Module = torch.class('nn.Module')

function Module:updateParameters(learningRate)
   local params, gradParams = self:parameters()
   if params then
      for i=1,#params do
         params[i]:add(-learningRate, gradParams[i])
      end
   end
end
```

- If the module has parameters, this will update these parameters, according to the accumulation of the gradients with respect to these parameters, accumulated through backward() calls.
- The update is basically:

  parameters = parameters - learningRate * gradients_wrt_parameters
- If the module does not have parameters, it does nothing

Homework

Assignment

We will use Kaggle to host the leader board for the assignment. You will be able to submit your model predictions on Kaggle using your @iitb.ac.in email ids and check where you stand as compared to others in real time. Complete details about the assignment can be found here. Sample data can be downloaded from here. Due Date: 14 April 2017, 23:55.

Assignment

This competition is private-entry. You've been invited to participate.

Get higest test set classification accuracy.

As a part of the IIT Bombay CS763 (Spring 2017) Deep Learning Module assignment, Kaggle is helping us host the CIFAR-10 leaderboard so that we can compare the accuracy of our trained models.

CIFAR-10 is an established computer-vision dataset used for object recognition. It consists of totally 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. Of this, 50,000 images are used for training and 10,000 for testing.

You can see how your approach compares to the state-of-the-art methods on this page. You can also find a VGG net based network in torch here which gets >90% test set performance.

Assignment

Data Files

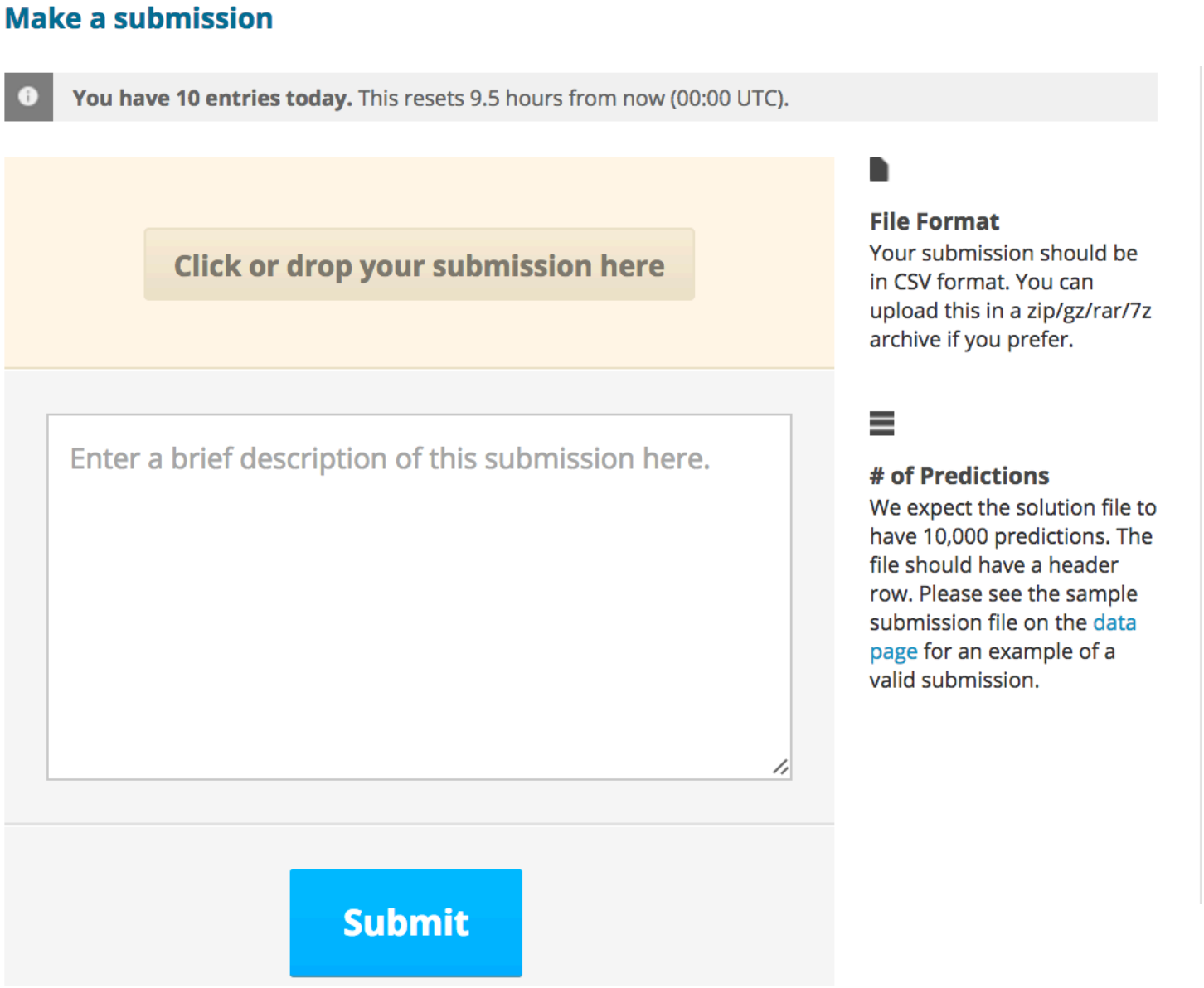
File Name	Available Formats
tr_labels	.bin (48.93 kb)
te_data	.bin (29.30 mb)
tr_data	.bin (146.48 mb)

The CIFAR-10 data consists of 60,000 32x32 color images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images in the official data. We have preserved the train/test split from the original dataset. The provided files are:

```
tr_data.bin - training images as torch binary
te_data.bin - test images as torch binary
tr_labels.bin - the training labels
```

You can refer to the iTorch notebook Single Fully Connected Layer Model which shows how to load this data and the model gives around ~40% accuracy. You will have to cross-validate using the provided training data.

Assignment


Make a submission
You have 10 entries today. This resets 9.5 hours from now (00:00 UTC).
Click or drop your submission here
Enter a brief description of this submission here.
Submit
File Format
Your submission should be in CSV format. You can upload this in a zip/gz/rar/7z archive if you prefer.
# of Predictions
We expect the solution file to have 10,000 predictions. The file should have a header row. Please see the sample submission file on the data page for an example of a valid submission.

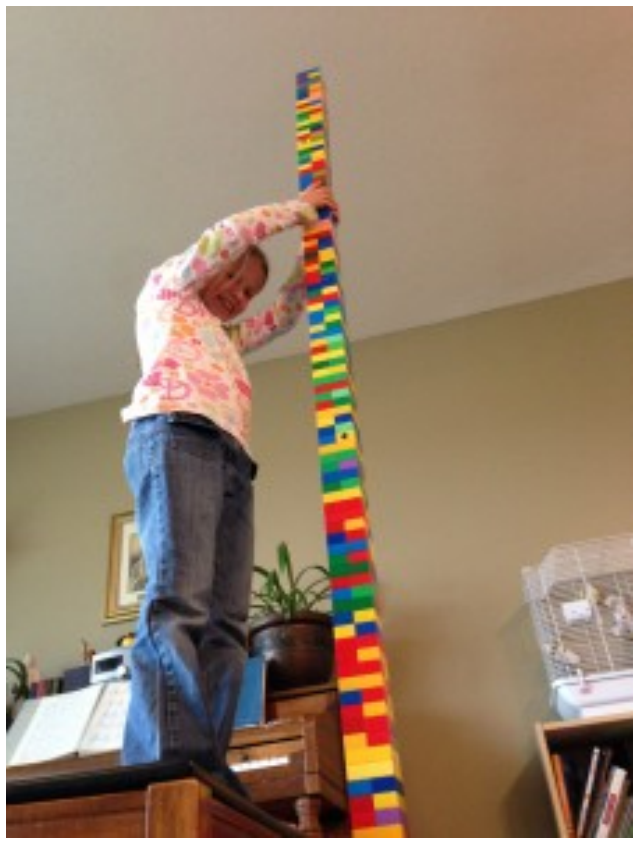
Assignment

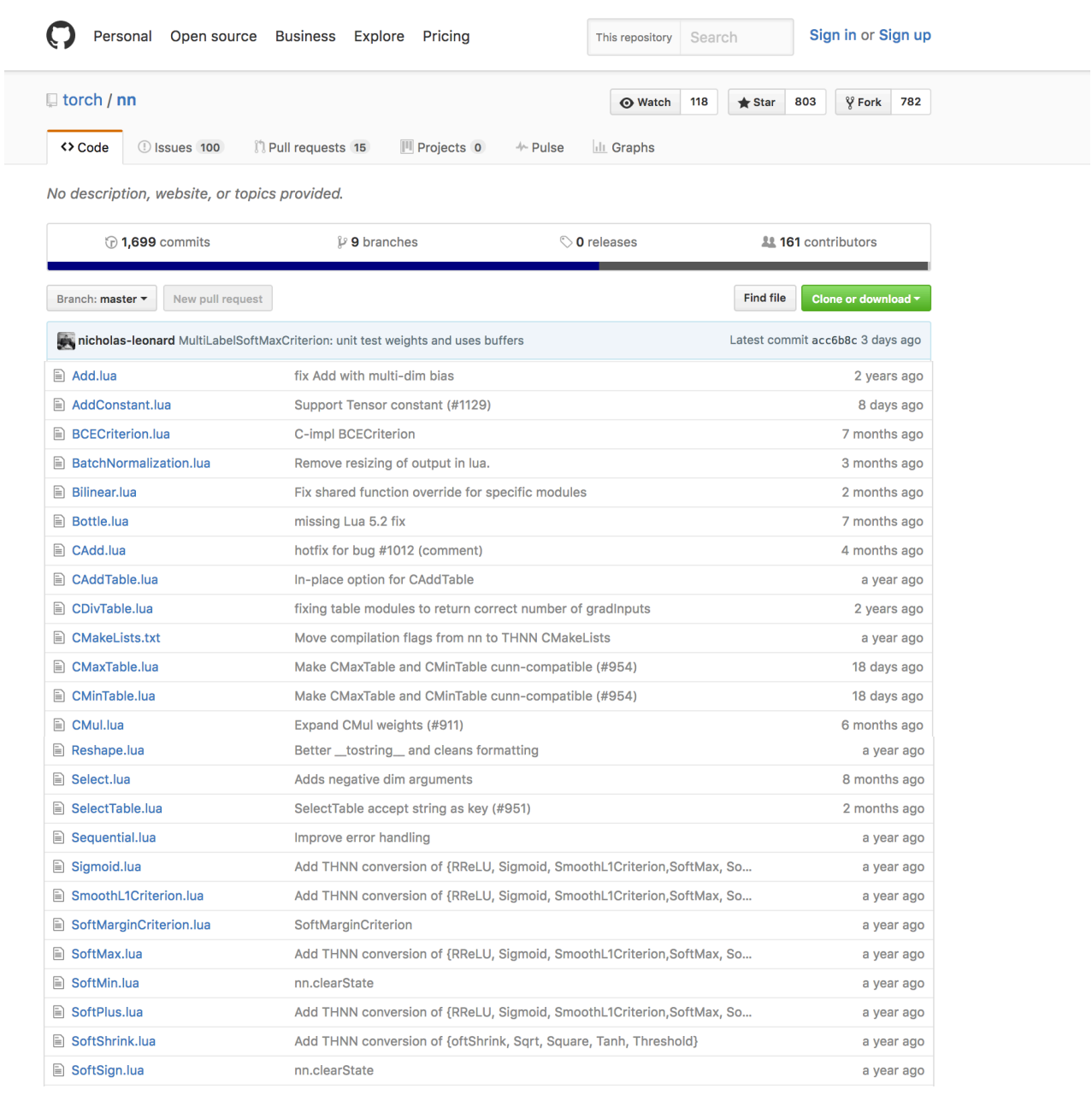
I know that the test set is publically available so:

1. The test-set may have been shuffled
2. Random noise may have been added to some test set images (so hash lookups wont work)
3. You will need to submit your training code and we will train your model on our servers (so do not train using the test set!). If we find that the training code is unable to output the same best model, no marks will be awarded for the complete assignment.

DNN Building Blocks

- Activation Layers (ReLU, Sigmoid, etc.)
- Fully Connected Layer
- Convolution Layer
- Max Pooling Layer
- …

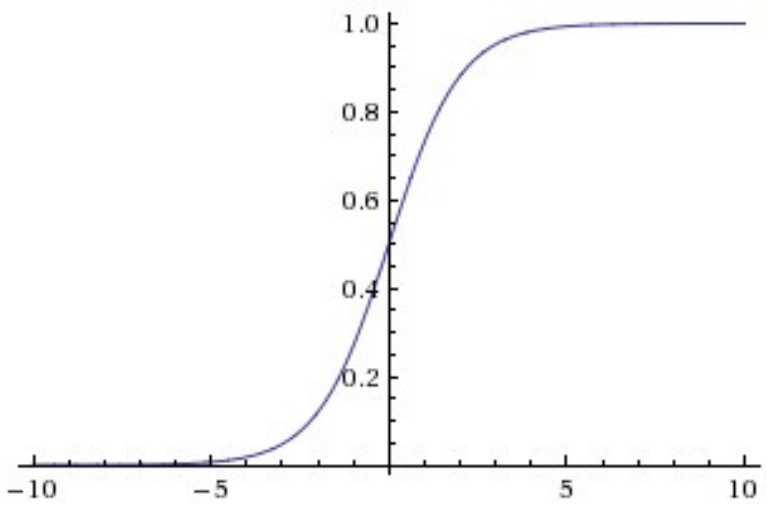



Personal Open source Business Explore Pricing
This repository Search
Sign in or Sign up
torch / nn
Watch 118 Star 803 Fork 782
Code Issues 100 Pull requests 15 Projects 0 Pulse Graphs
No description, website, or topics provided.
1,699 commits 9 branches 0 releases 161 contributors
Branch: master New pull request
Find file Clone or download
nicholas-leonard MultiLabelSoftMaxCriterion: unit test weights and uses buffers
Latest commit acc6b8c 3 days ago
Add.lua fix Add with multi-dim bias 2 years ago
AddConstant.lua Support Tensor constant (#1129) 8 days ago
BCECriterion.lua C-impl BCECriterion 7 months ago
BatchNormalization.lua Remove resizing of output in lua. 3 months ago
Bilinear.lua Fix shared function override for specific modules 2 months ago
Bottle.lua missing Lua 5.2 fix 7 months ago
CAdd.lua hotfix for bug #1012 (comment) 4 months ago
CAddTable.lua In-place option for CAddTable a year ago
CDivTable.lua fixing table modules to return correct number of gradInputs 2 years ago
CMakeLists.txt Move compilation flags from nn to THNN CMakeLists a year ago
CMaxTable.lua Make CMaxTable and CMinTable cunn-compatible (#954) 18 days ago
CMinTable.lua Make CMaxTable and CMinTable cunn-compatible (#954) 18 days ago
CMul.lua Expand CMul weights (#911) 6 months ago
Reshape.lua Better __tostring__ and cleans formatting a year ago
Select.lua Adds negative dim arguments 8 months ago
SelectTable.lua SelectTable accept string as key (#951) 2 months ago
Sequential.lua Improve error handling a year ago
Sigmoid.lua Add THNN conversion of {RReLU, Sigmoid, SmoothL1Criterion,SoftMax, So... a year ago
SmoothL1Criterion.lua Add THNN conversion of {RReLU, Sigmoid, SmoothL1Criterion,SoftMax, So... a year ago
SoftMarginCriterion.lua SoftMarginCriterion a year ago
SoftMax.lua Add THNN conversion of {RReLU, Sigmoid, SmoothL1Criterion,SoftMax, So... a year ago
SoftMin.lua nn.clearState a year ago
SoftPlus.lua Add THNN conversion of {RReLU, Sigmoid, SmoothL1Criterion,SoftMax, So... a year ago
SoftShrink.lua Add THNN conversion of {oftShrink, Sqrt, Square, Tanh, Threshold} a year ago
SoftSign.lua nn.clearState a year ago

Building Blocks:
Activation Functions

Activation Functions: Sigmoid

$$\sigma(x) = \frac{1}{(1 + e^{-x})}$$



Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

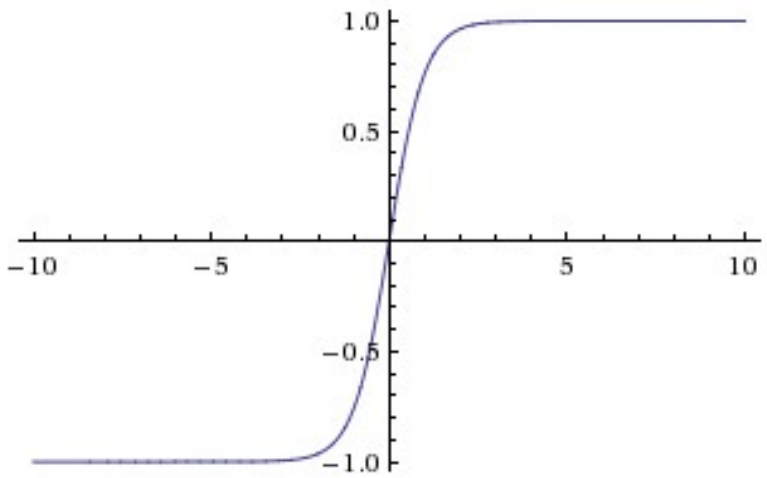
3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Functions: Tanh

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

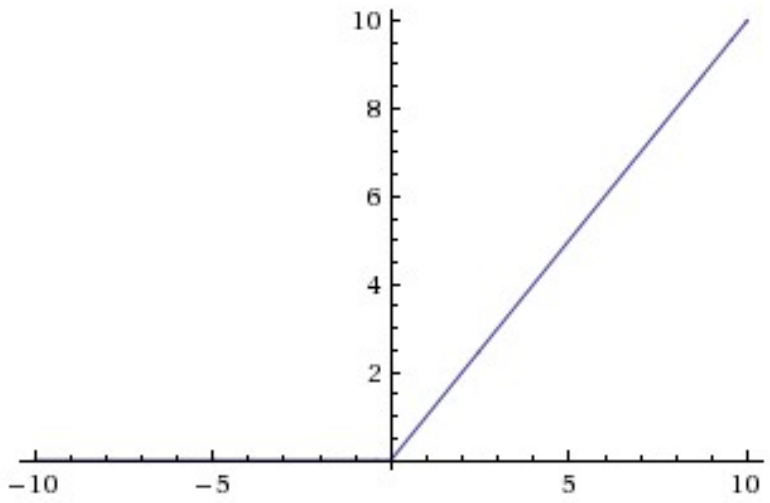


Tanh

[LeCun et al., 1991]

Activation Functions: ReLU

$$f(x) = \max(0, x)$$

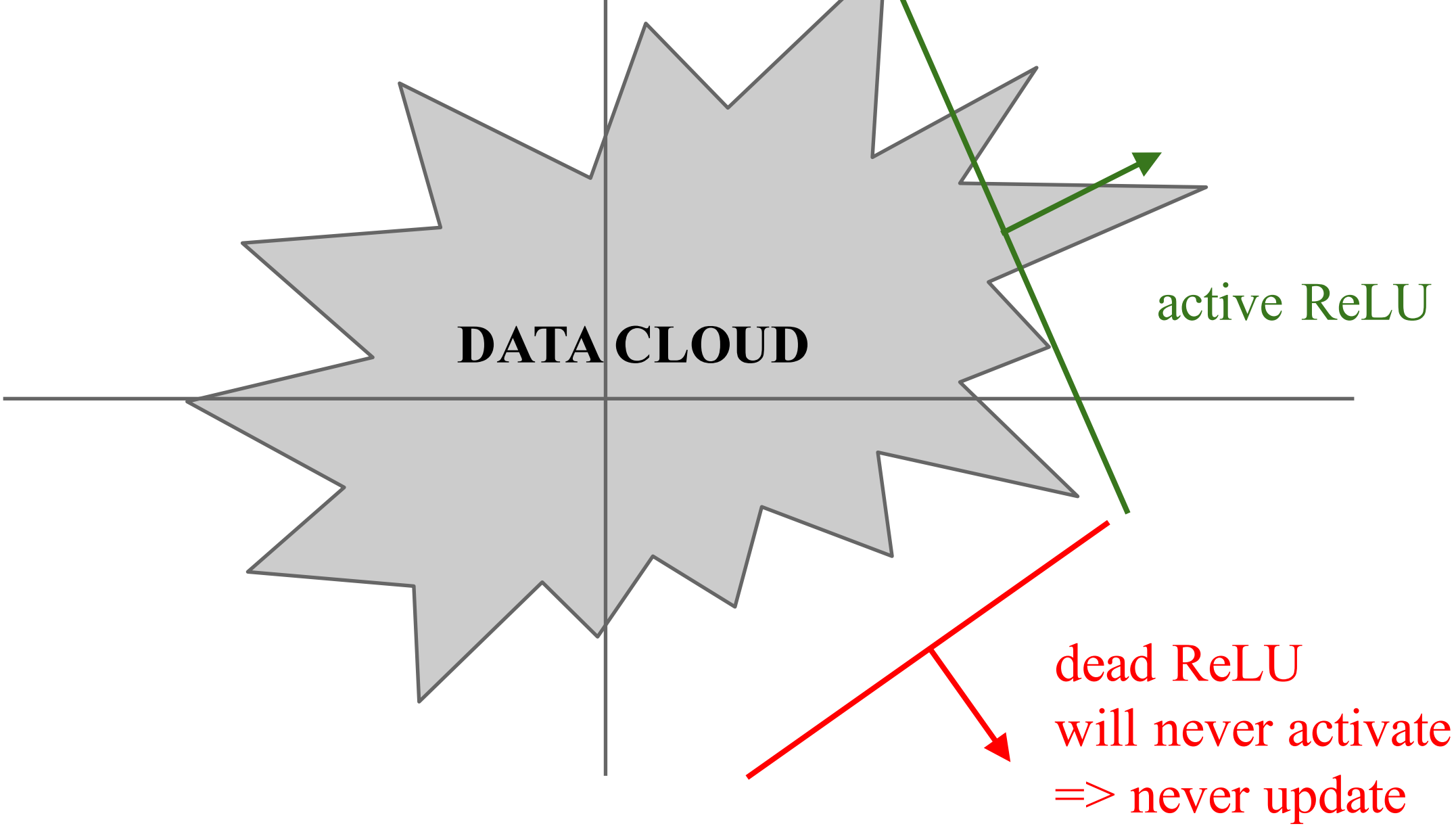


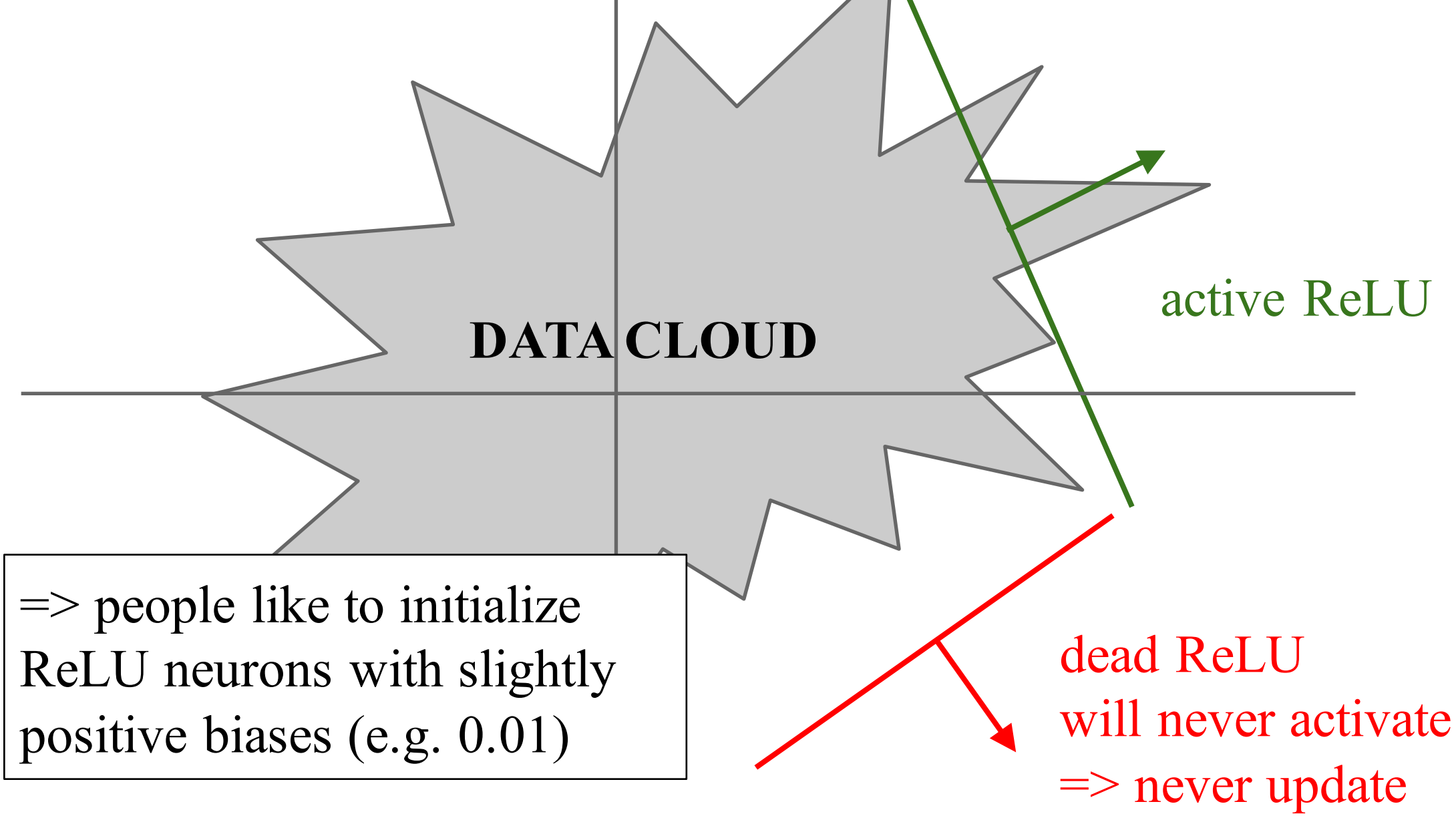
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance (hint: what is the gradient when x < 0)?

ReLU (Rectified Linear Unit)

[Nair and Hinton et al., 2010]

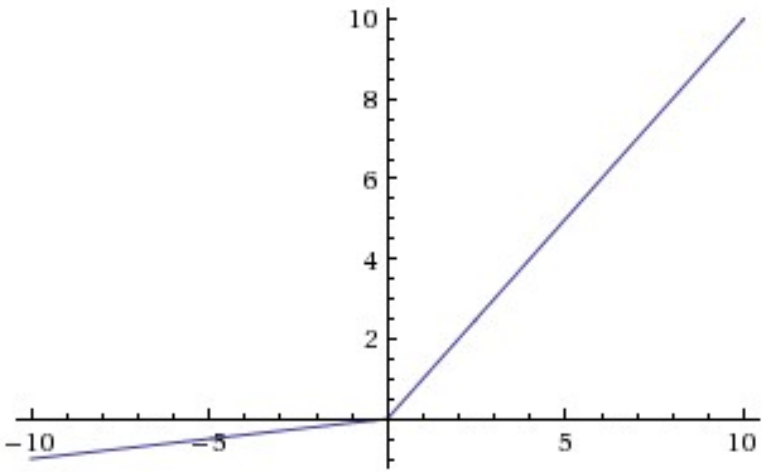

DATA CLOUD
active ReLU
dead ReLU
will never activate
=> never update



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Activation Functions: Tanh

$$f(x) = \max(0.01x, x)$$



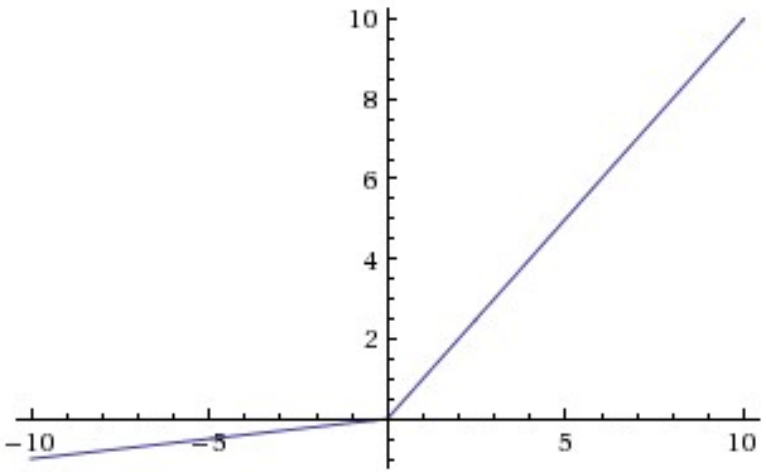
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- **will not "die".**

Leaky ReLU

[Mass et al., 2013]

Activation Functions

$$f(x) = \max(0.01x, x)$$



Leaky ReLU

[Mass et al., 2013]

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- **will not "die".**

backprop into α (learnable parameter)

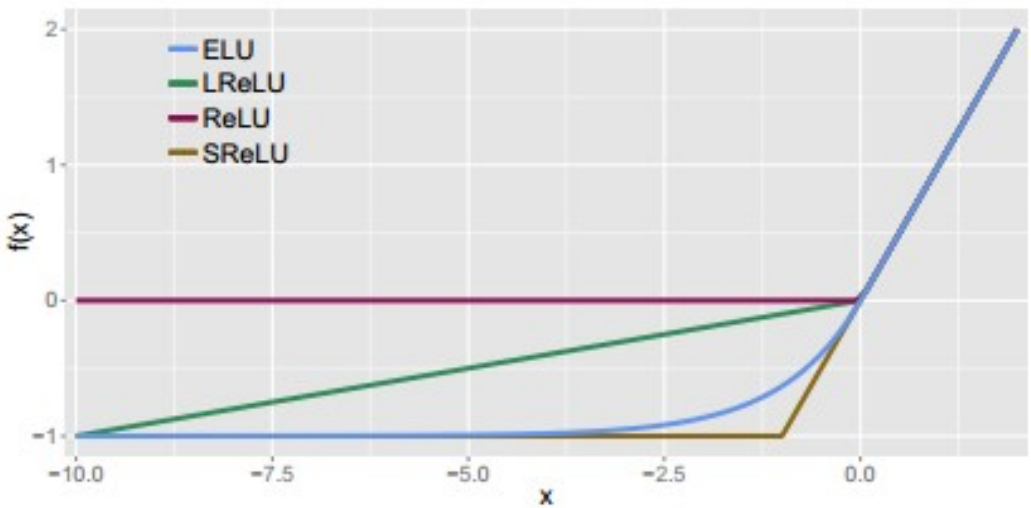
$$f(x) = \max(\alpha x, x)$$

Parametric Rectifier (PReLU)

[He et al., 2015]

Activation Functions: ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$



- All benefits of ReLU
- Does not die
- Closer to zero mean outputs

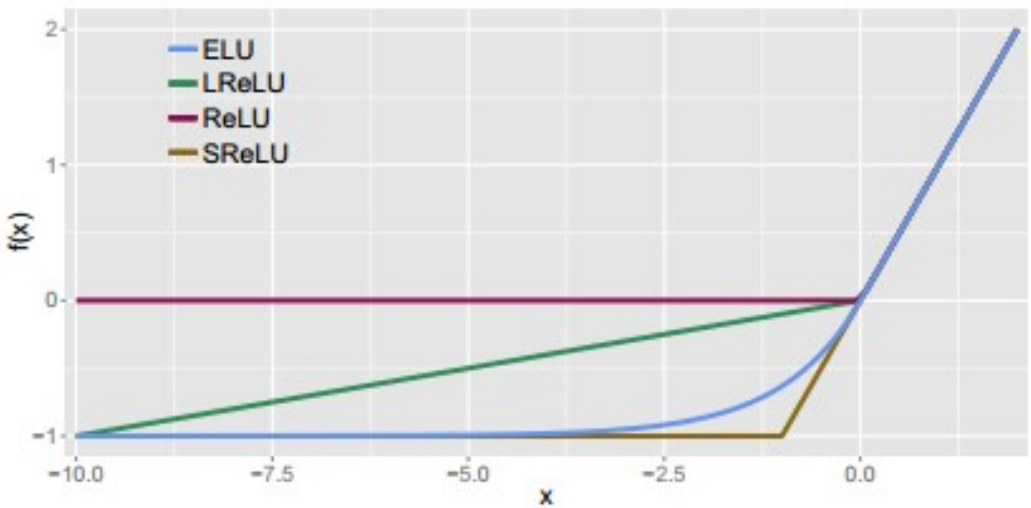
- Computation requires exp()

Exponential Linear Units (ELU)

[Clevert et al., 2015]

Activation Functions: ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$



Exponential Linear Units (ELU)

Empirical Evaluation of Rectified Activations in Convolutional Network

Bing Xu, Naiyan Wang, Tianqi Chen, Mu Li

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs

- Computation requires exp()

[Clevert et al., 2015]

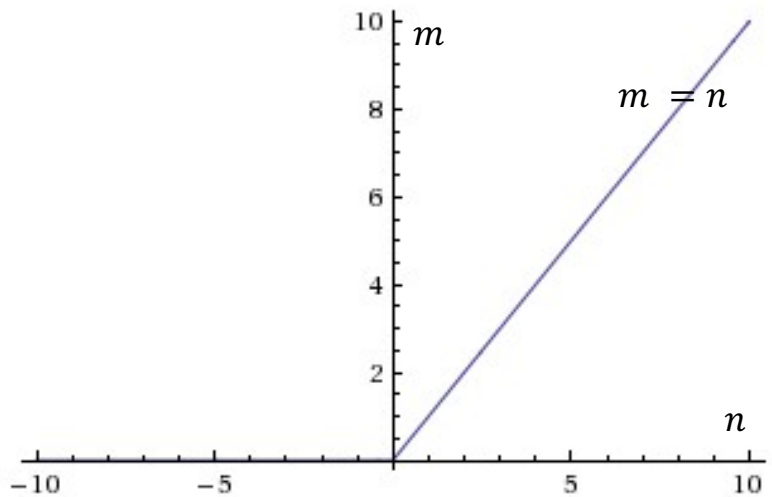
TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid

Activation Function: ReLU (details)

$\boldsymbol{n} \in \mathbb{R}^d$ → nn.ReLU → $\boldsymbol{m} \in \mathbb{R}^d$

$$m_i = \max(0, n_i)$$



$$m_i = \begin{cases} 0 & if \ \ n_i < 0 \\ n_i & if \ \ n_i > 0 \end{cases}$$

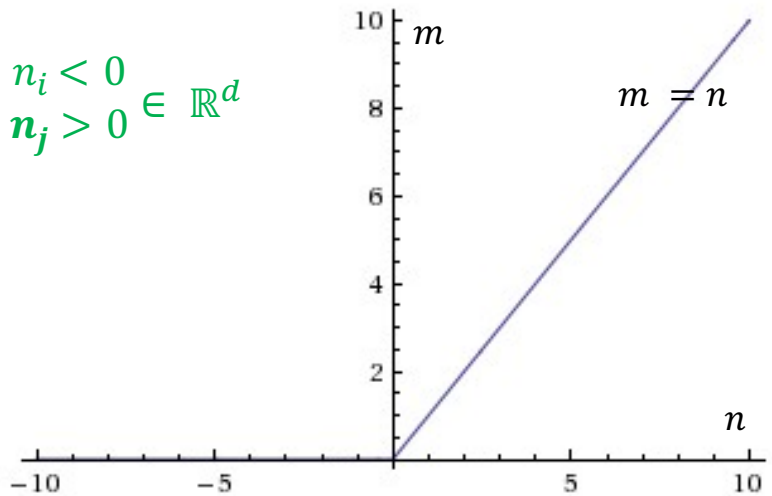
Activation Function: ReLU (details)

$$\frac{\partial L}{\partial \boldsymbol{m}} \cdot \frac{\partial \boldsymbol{m}}{\partial \boldsymbol{n}} \in \mathbb{R}^{1 \times \dim(n)} \qquad \frac{\partial L}{\partial \boldsymbol{m}} \in \mathbb{R}^{1 \times \dim(m)}$$

$\boldsymbol{n} \in \mathbb{R}^d$ → nn.ReLU → $\boldsymbol{m} \in \mathbb{R}^d$

$$m_i = \max(0, n_i)$$

$$\frac{\partial m_i}{\partial n_j} = \frac{\partial max(0, n_i)}{\partial n_j} = \begin{cases} 0 & if \ n_i < 0 \\ 1 & if \ \boldsymbol{n_j} > 0 \end{cases} \in \mathbb{R}^d$$



$$m_i = \begin{cases} 0 & if \ n_i < 0 \\ n_i & if \ n_i > 0 \end{cases}$$

Activation Function: ReLU (forward)

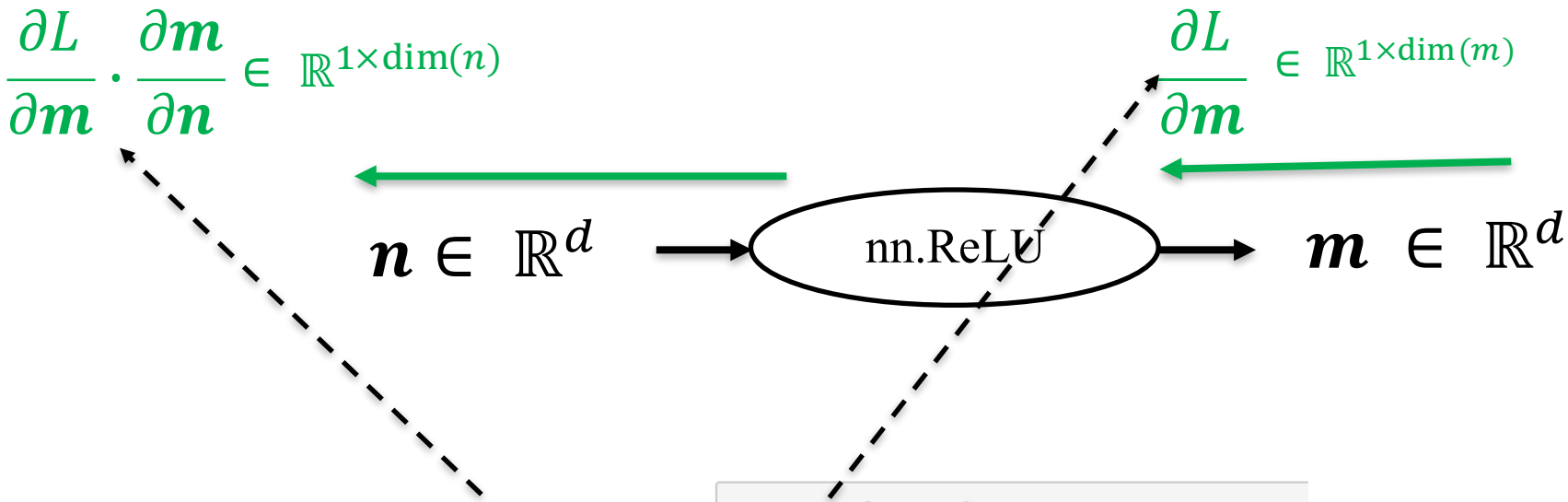

```
require 'nn';
n = torch.rand(5) - 0.5
print(n)
```

```
-0.0044
-0.1521
 0.4794
-0.1014
 0.4201
[torch.DoubleTensor of size 5]
```

```
relu = nn.ReLU()
m = relu:forward(n)
print(m)
```

```
 0.0000
 0.0000
 0.4794
 0.0000
 0.4201
[torch.DoubleTensor of size 5]
```

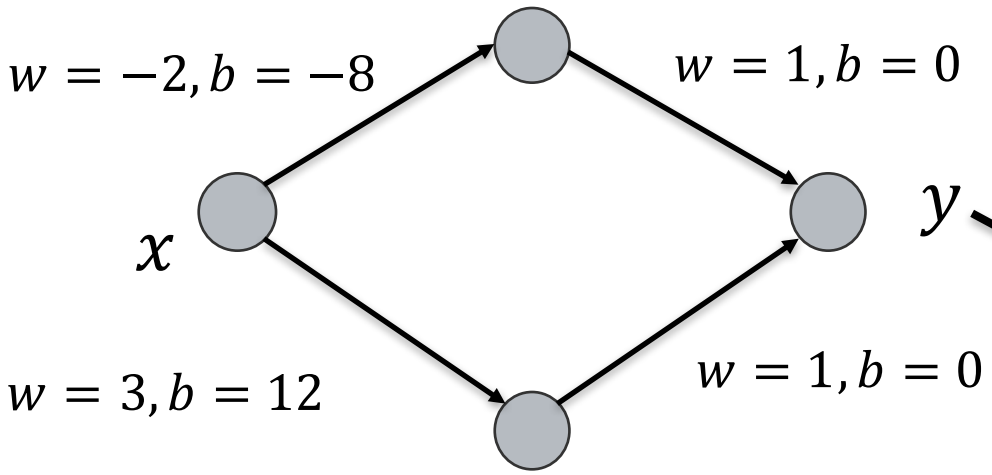
Activation Function: ReLU (backward)

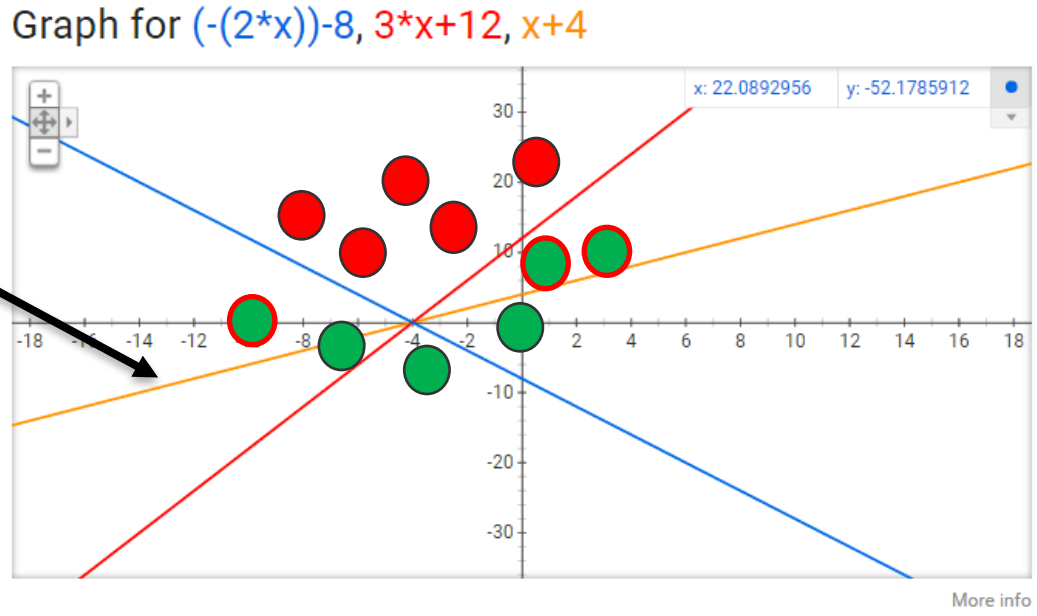



```
nextgrad=torch.ones(5)
relu:backward(n, nextgrad)
print(relu.gradInput)
```

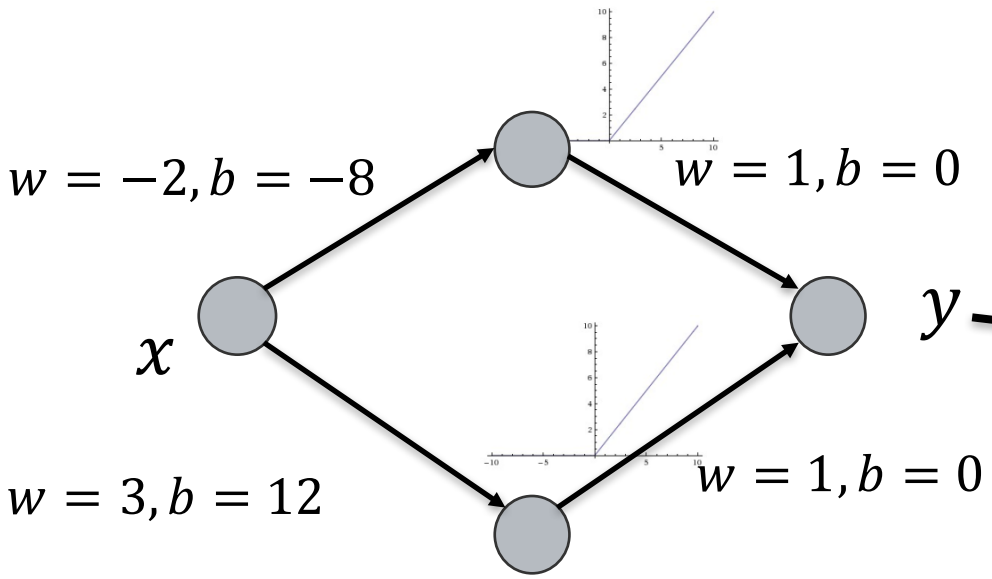
```
 0
 0
 1
 0
 1
[torch.DoubleTensor of size 5]
```

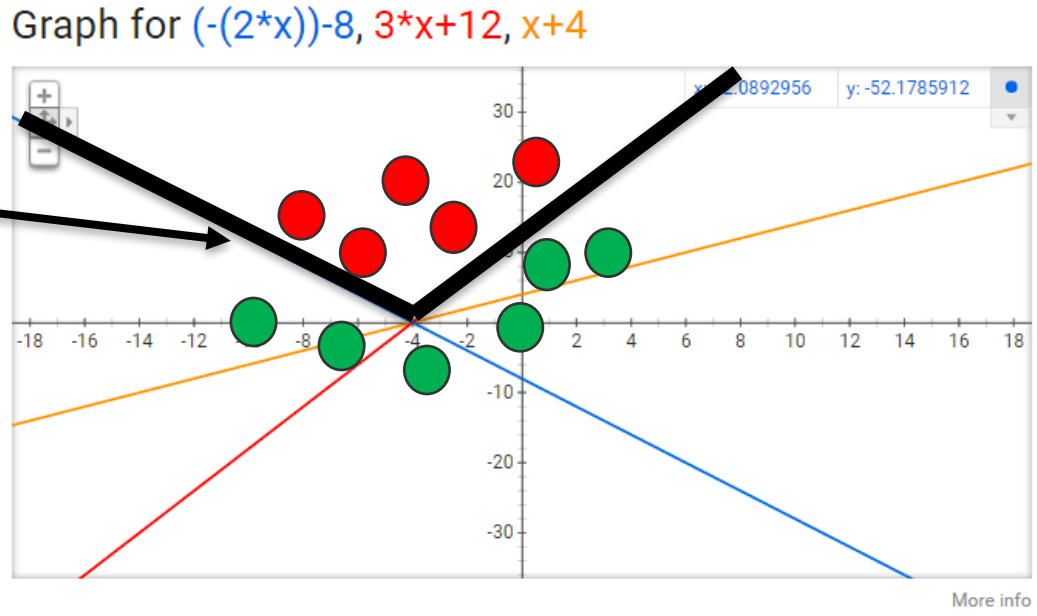
Building Blocks – ReLU



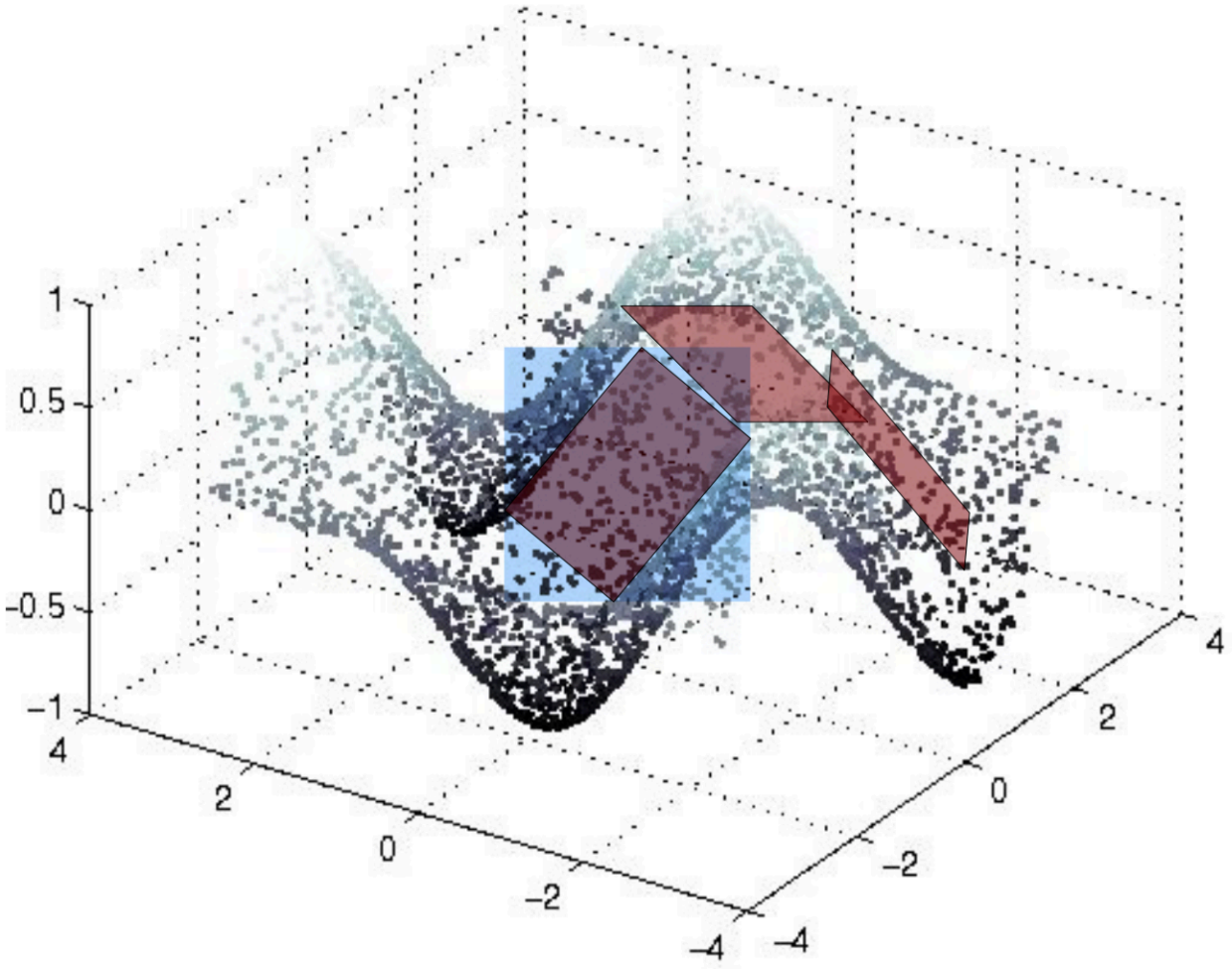


Building Blocks – ReLU

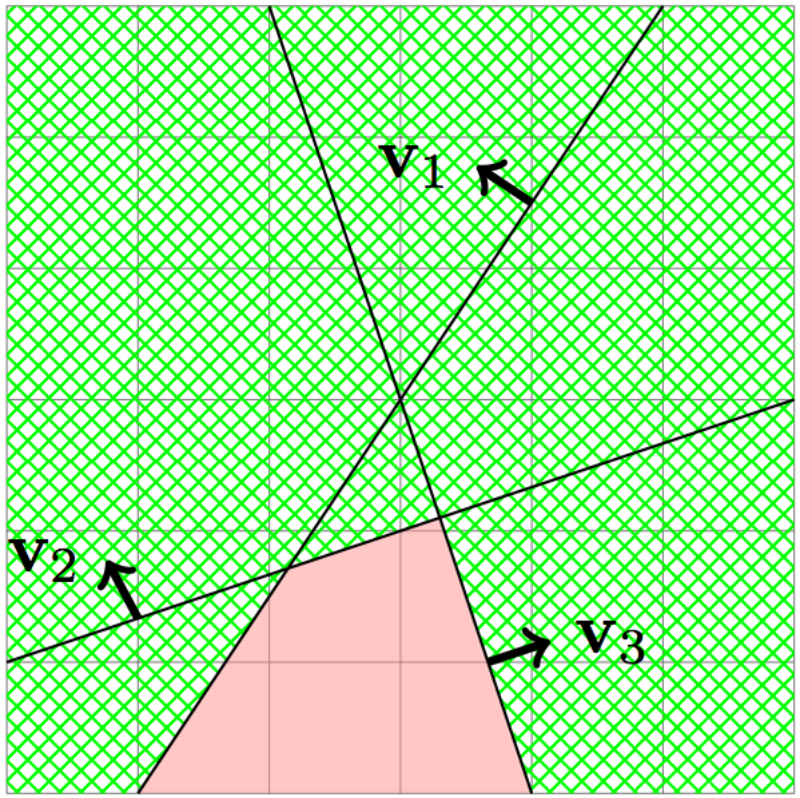




Building Blocks – ReLU



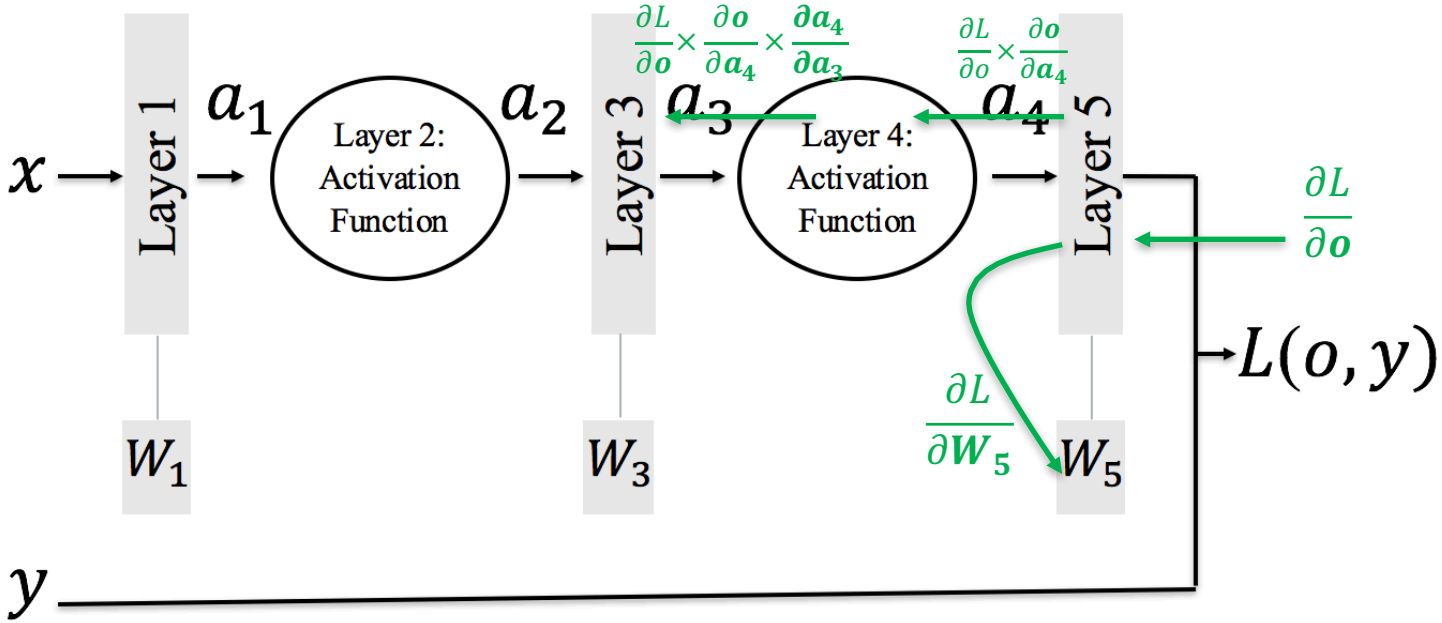
Building Blocks – ReLU



- Each hidden unit represents one hyperplane (parameterized by weight and bias) that bisects the input space into two half spaces.
- By choosing different weights in the hidden layer we can obtain arbitrary arrangement of n hyperplanes.
- The theory of hyperplane arrangement (Zaslavsky, 1975) tells us that for a general arrangement of n hyperplanes in d dimensions, the space is divided into $\sum_{s=0}^{d}\binom{n}{s}$ regions.

Expressiveness of Rectifier Networks
Xingyuan Pan, Vivek Srikumar

Vanishing Gradients



$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o} \times \frac{\partial o}{\partial a_4} \times \frac{\partial a_4}{\partial a_3} \times \frac{\partial a_3}{\partial a_2} \times \frac{\partial a_2}{\partial a_1} \times \frac{\partial a_1}{\partial W_1}$$

Thank you!