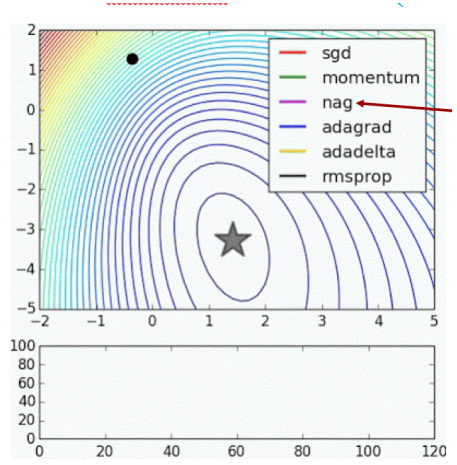
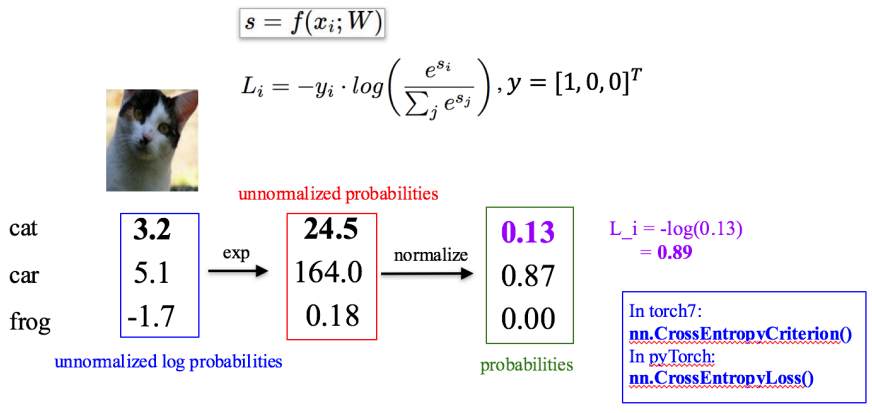
Deep Learning (for Computer Vision)

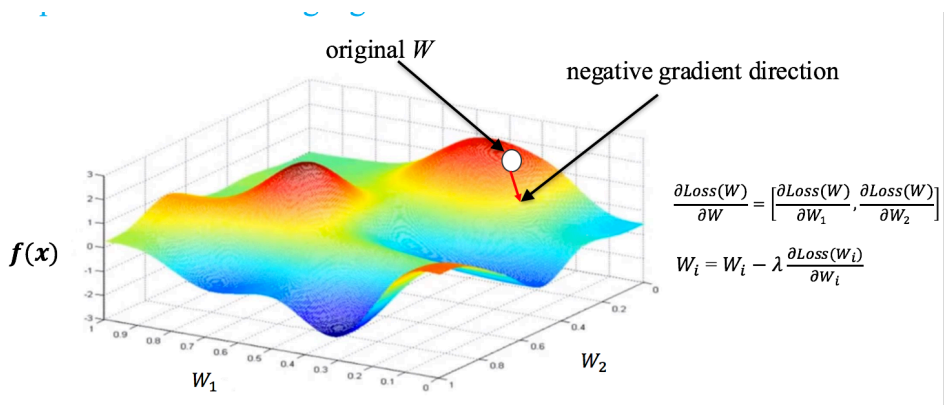
Arjun Jain | 25 March 2017

Recap

- Softmax classifier, cross-entropy loss function, regularization
- Optimization: vanilla gradient descent, stochastic gradient descent
- Vanilla momentum, Nesterov momentum, AdaGrad, RMSProp, ADAM
- Second order optimization methods, it's issues with deep learning
- Good learning rate, learning rate decay







Agenda this Week

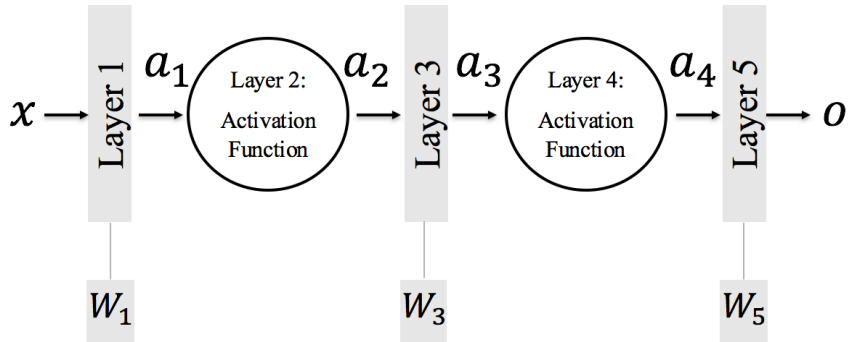
- Feed forwards networks, training using back propagation, chain rule
- DNN Building Blocks
  - Activation Layers (ReLU, Sigmoid, etc.)
  - Fully Connected Layer
  - Convolution Layer
  - Max Pooling Layer

Sources

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

- **http://cs231n.stanford.edu/**

Multiple Layers –Feed Forward - Composition of Functions



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

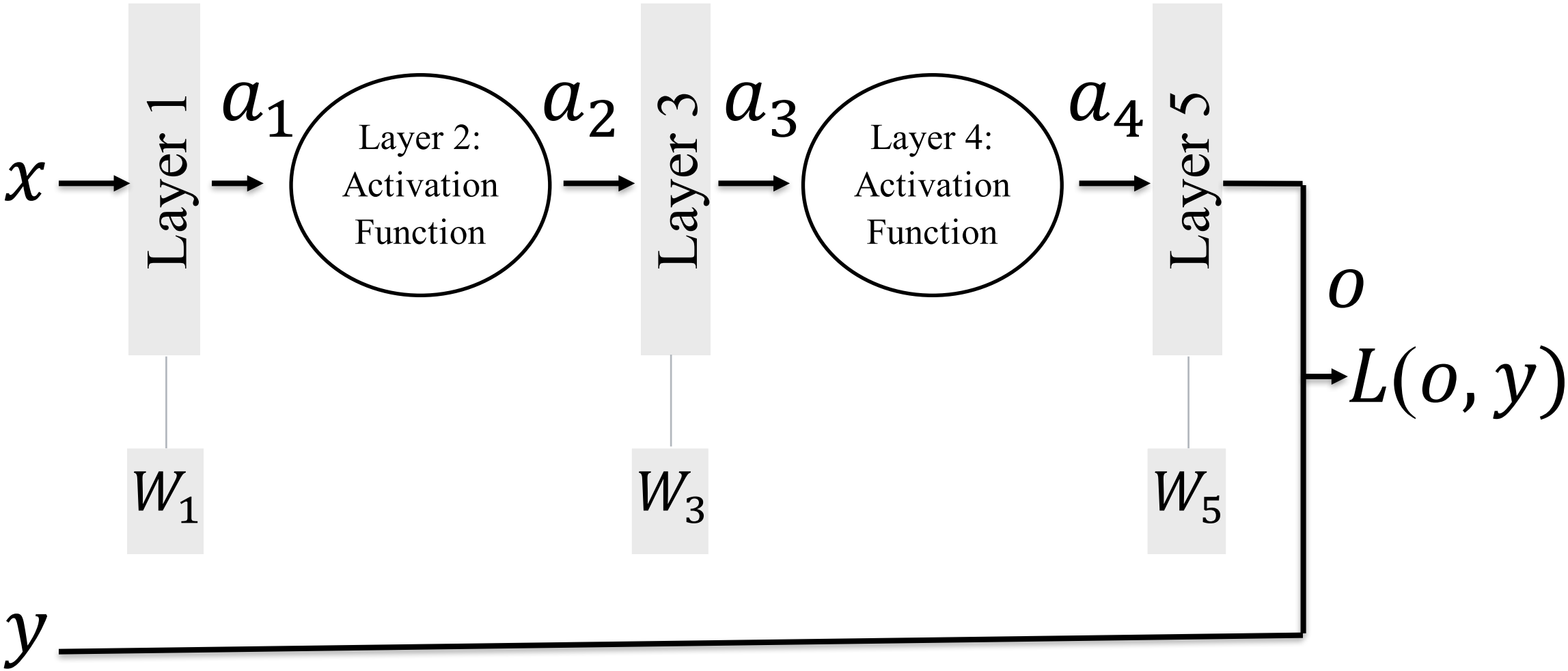
$$a_2 = G(a_1)$$

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

$$a_4 = J(a_3)$$

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

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.

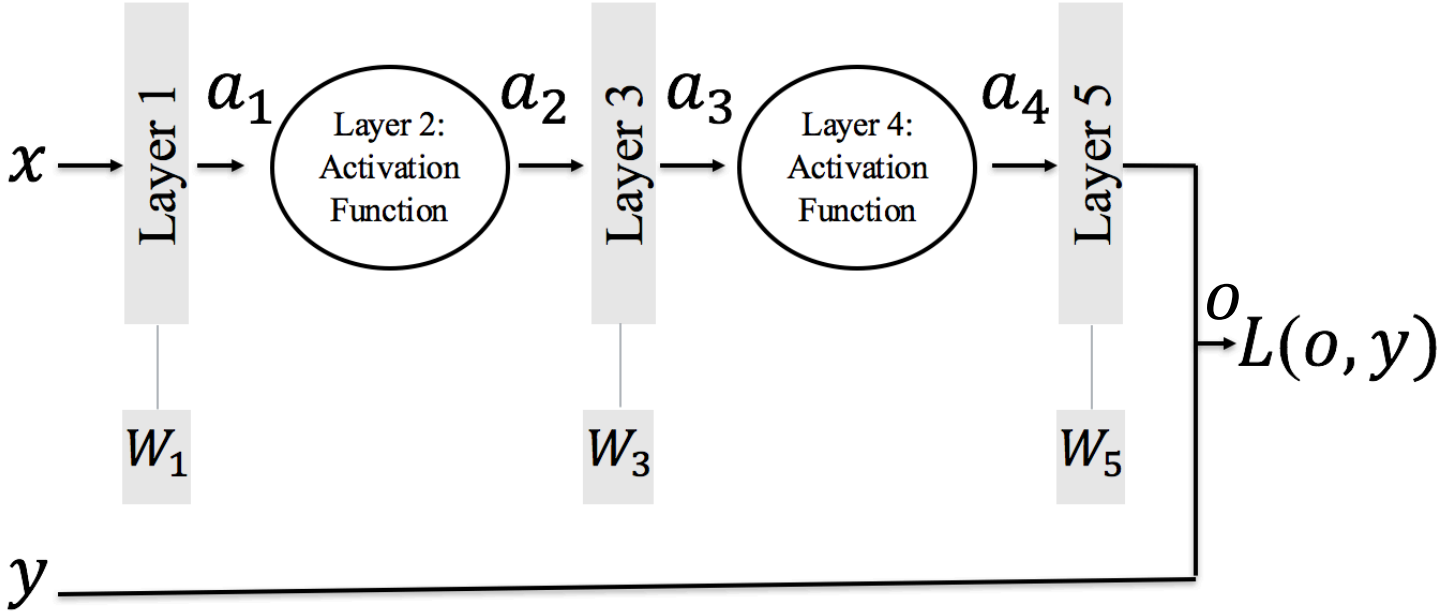
Let $x \in R^n$ (a column vector) and let $f : R^n \rightarrow R^m$. The derivative of f with respect to x is the $m \times n$ matrix:

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

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

http://www.cs.huji.ac.il/~csip/tirgul3_derivatives.pdf

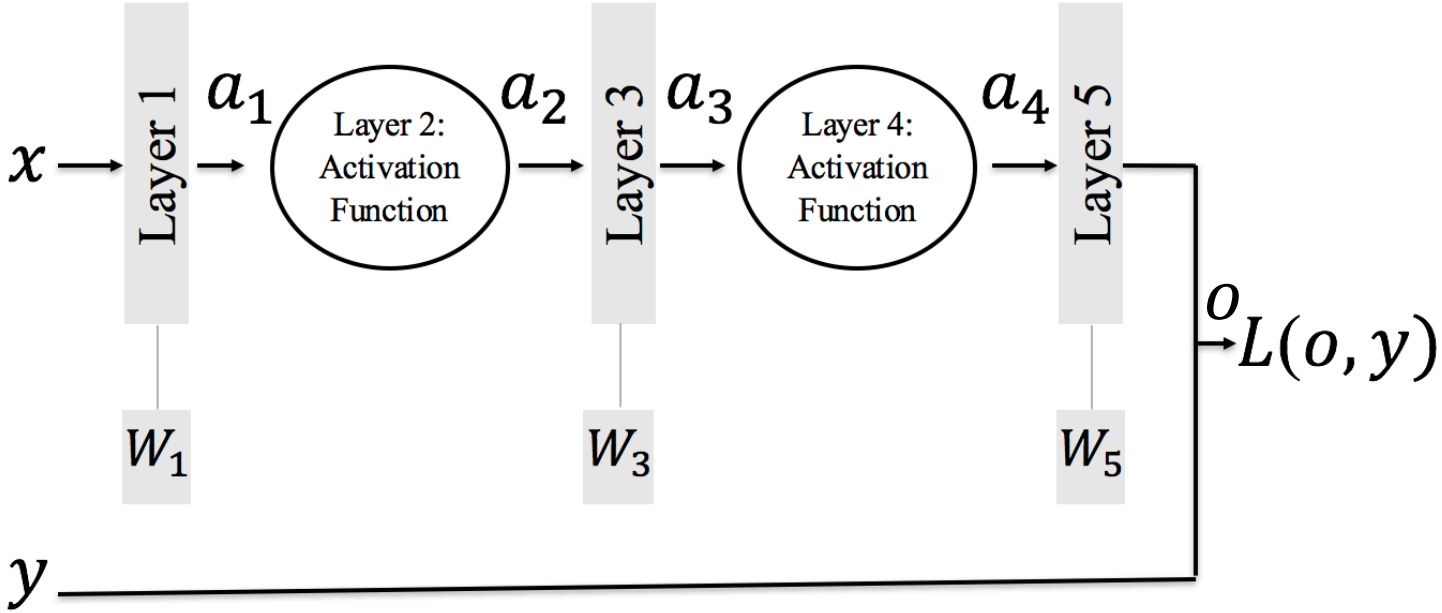
Multiple Layers – Back Prop: Chain Rule



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



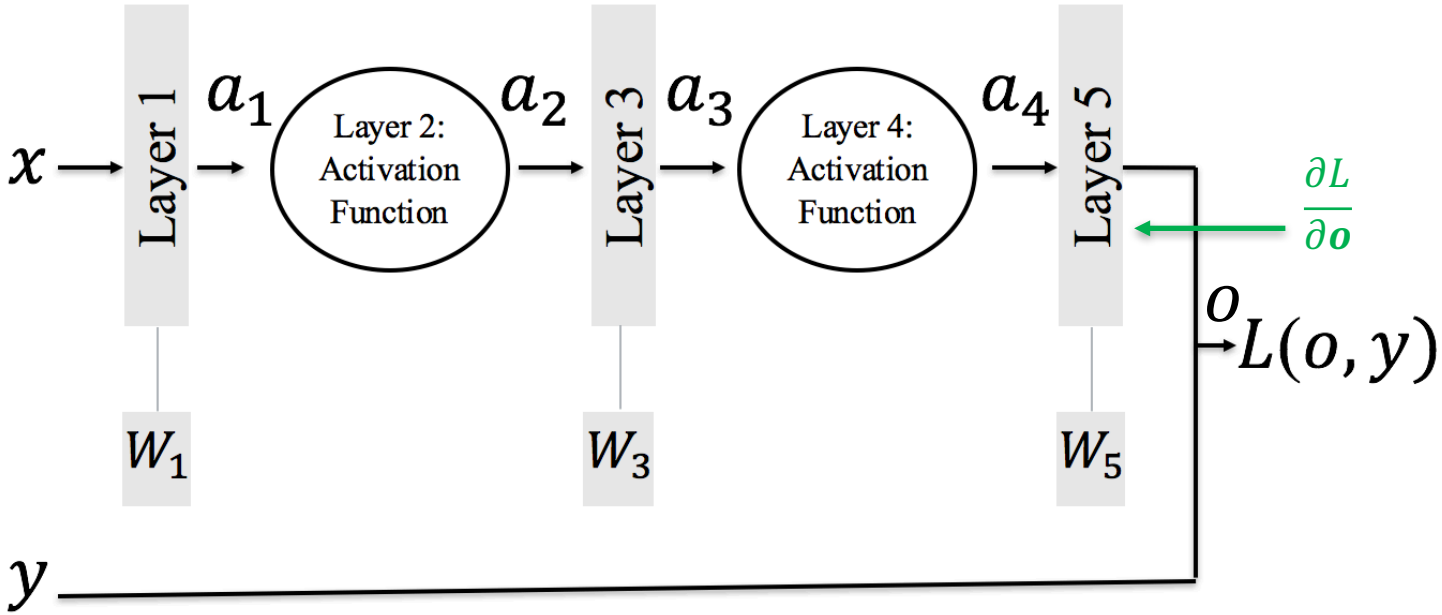
We want:

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

Compute:

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

Multiple Layers – Back Prop: Chain Rule



We want:

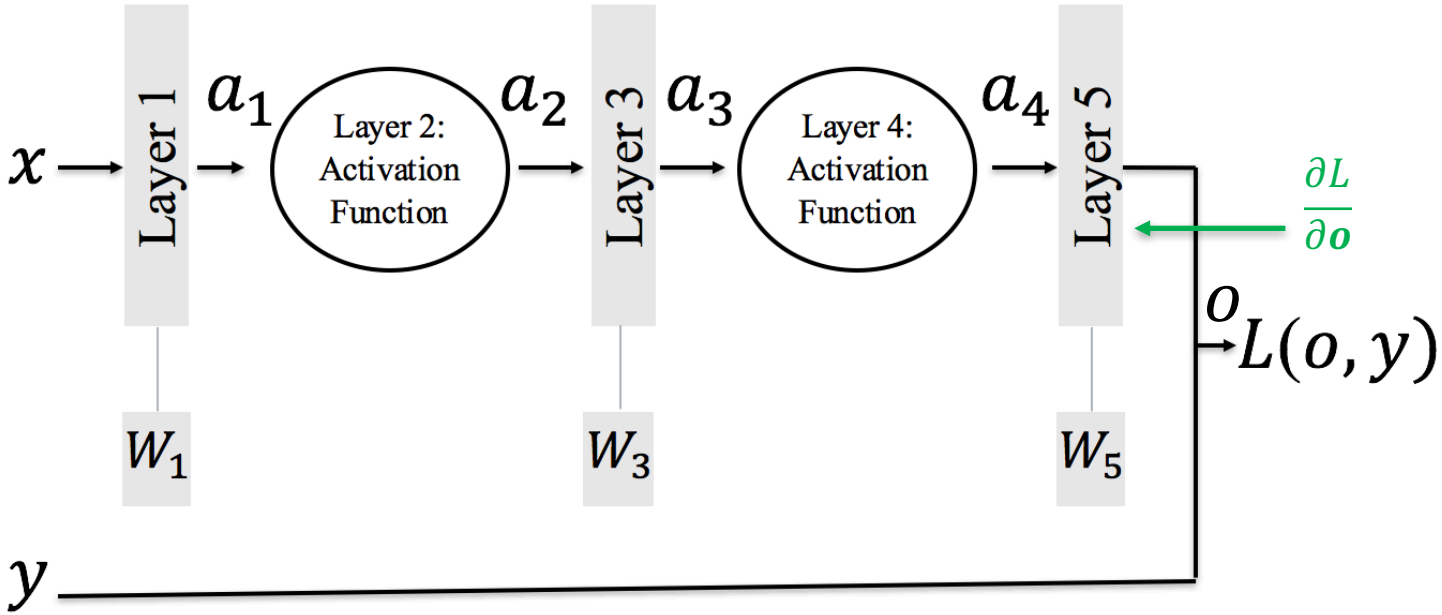
$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

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:

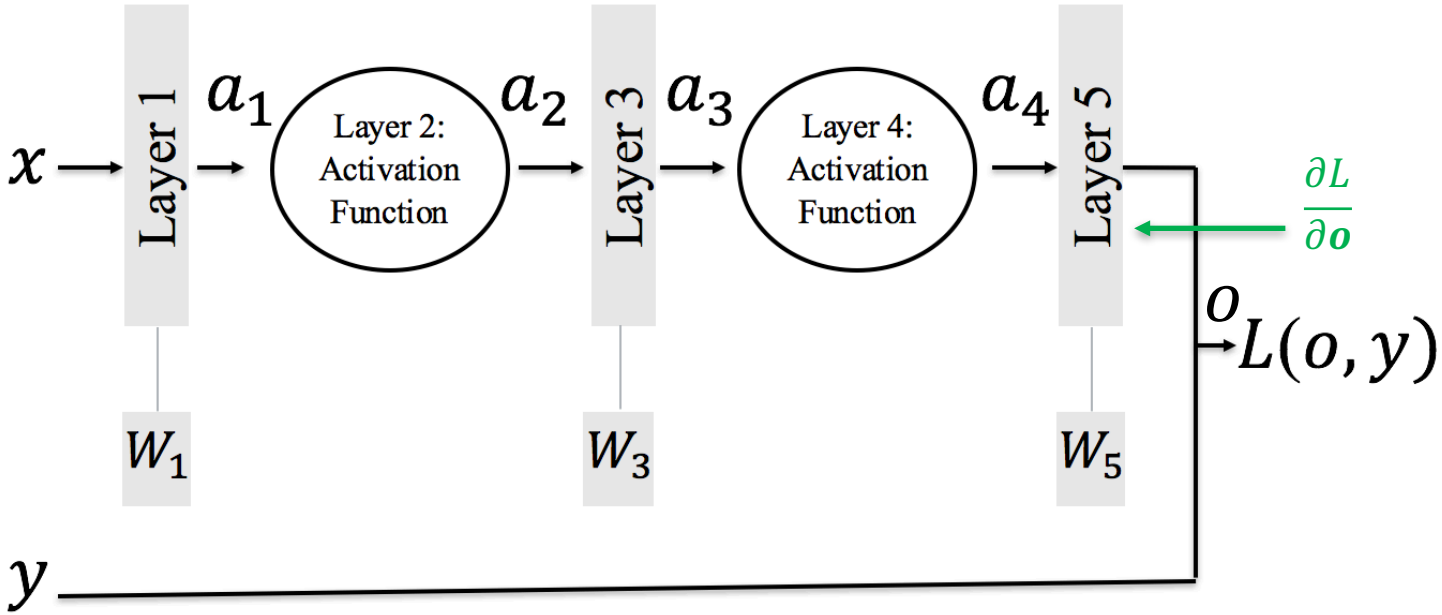
$$\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$$

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:

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

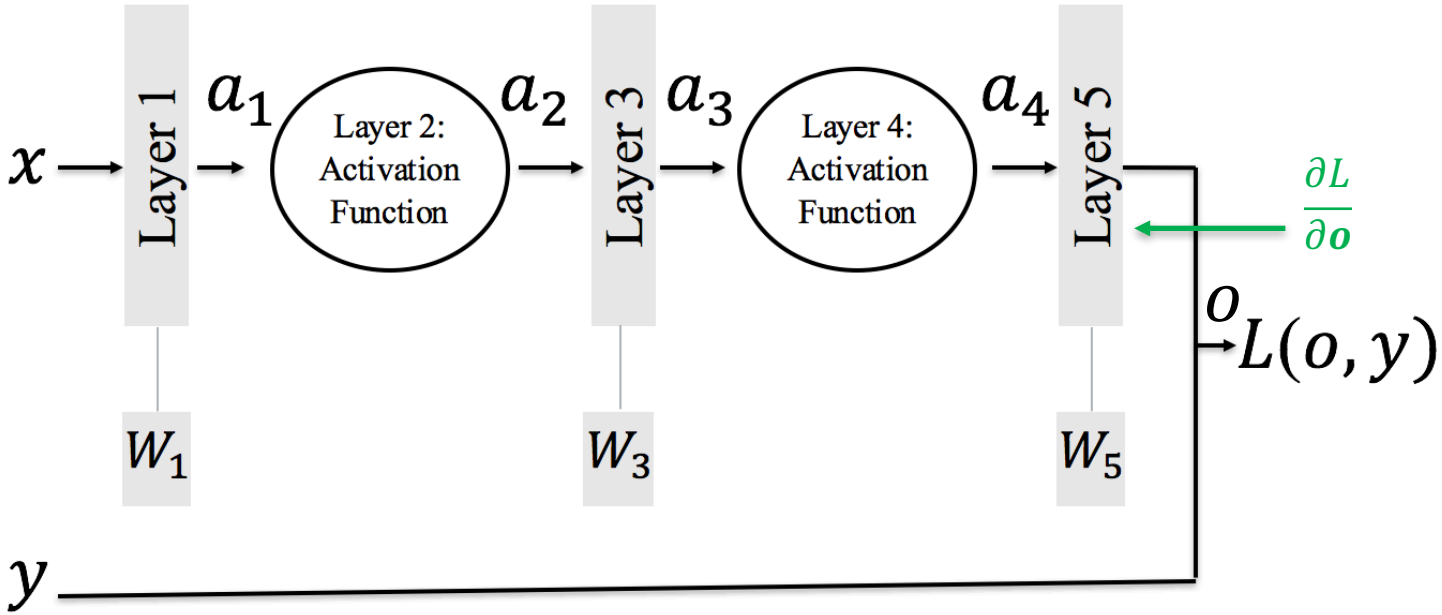
Compute:

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

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

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



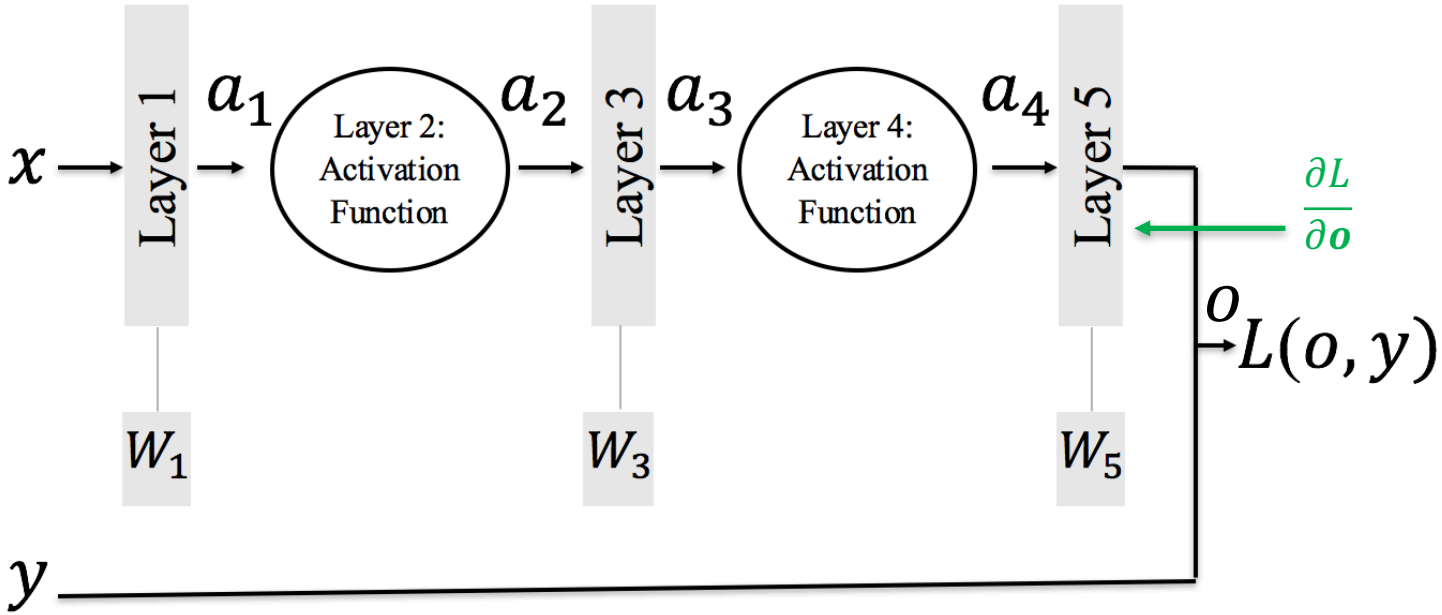
We want:

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

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, so some are also calling this differentiable programming

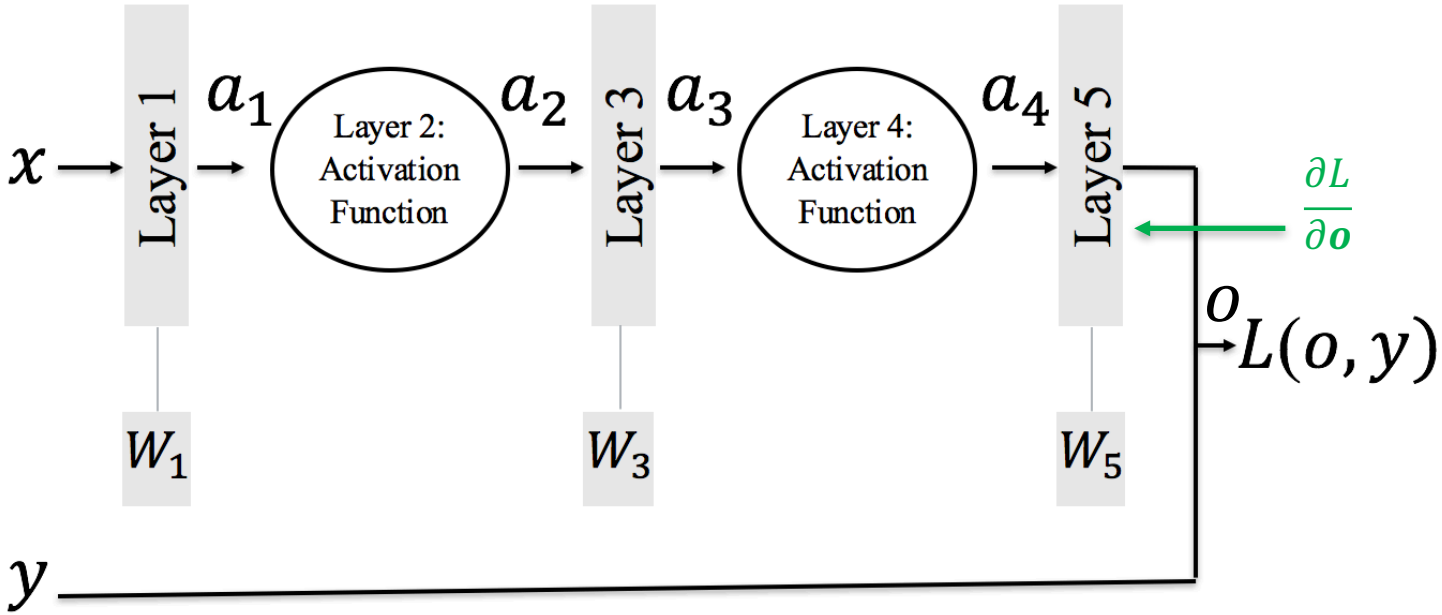
We want: $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$

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 $\mathbb{R}^{m \times \dim(W_5)}$

Multiple Layers – Back Prop: Chain Rule



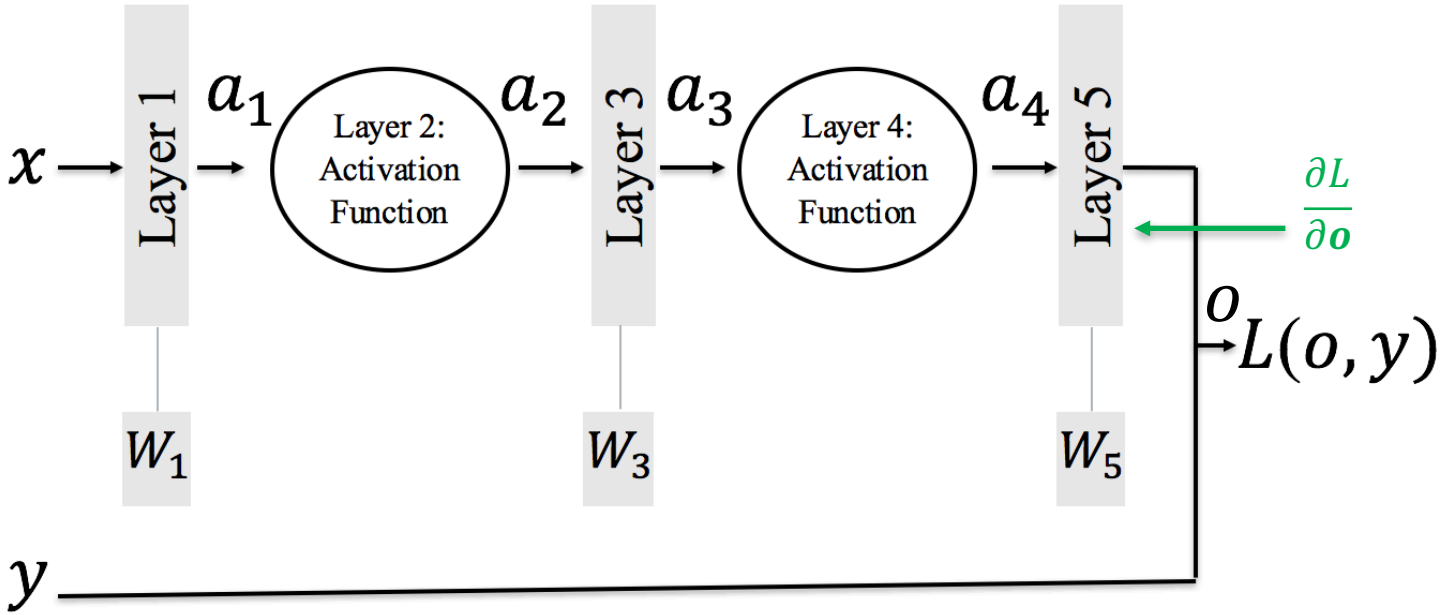
We want: $\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$

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: $\left[\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}\right]_{kl} = \frac{\partial [K(\boldsymbol{a_4}, \boldsymbol{W_5})]_k}{\partial [\boldsymbol{W}_5]_l}$

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

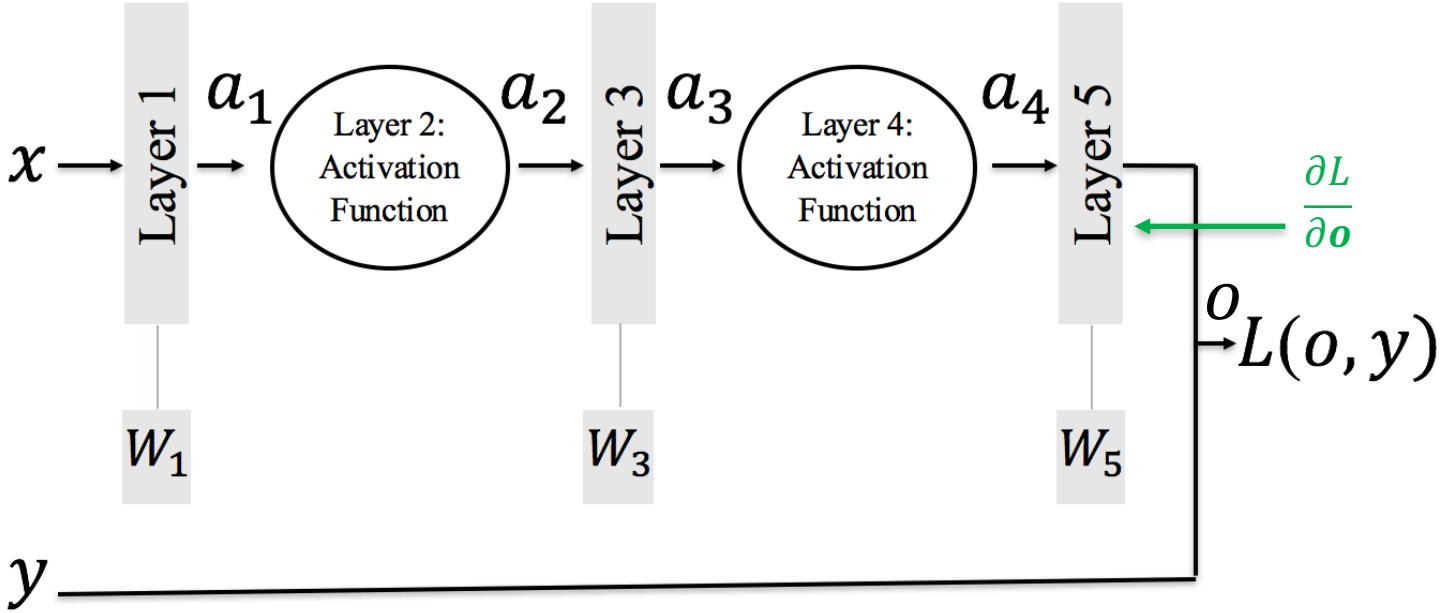
We want: $\frac{\partial L}{\partial \boldsymbol{W_1}}, \frac{\partial L}{\partial \boldsymbol{W_3}}, \frac{\partial L}{\partial \boldsymbol{W_5}}$

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:

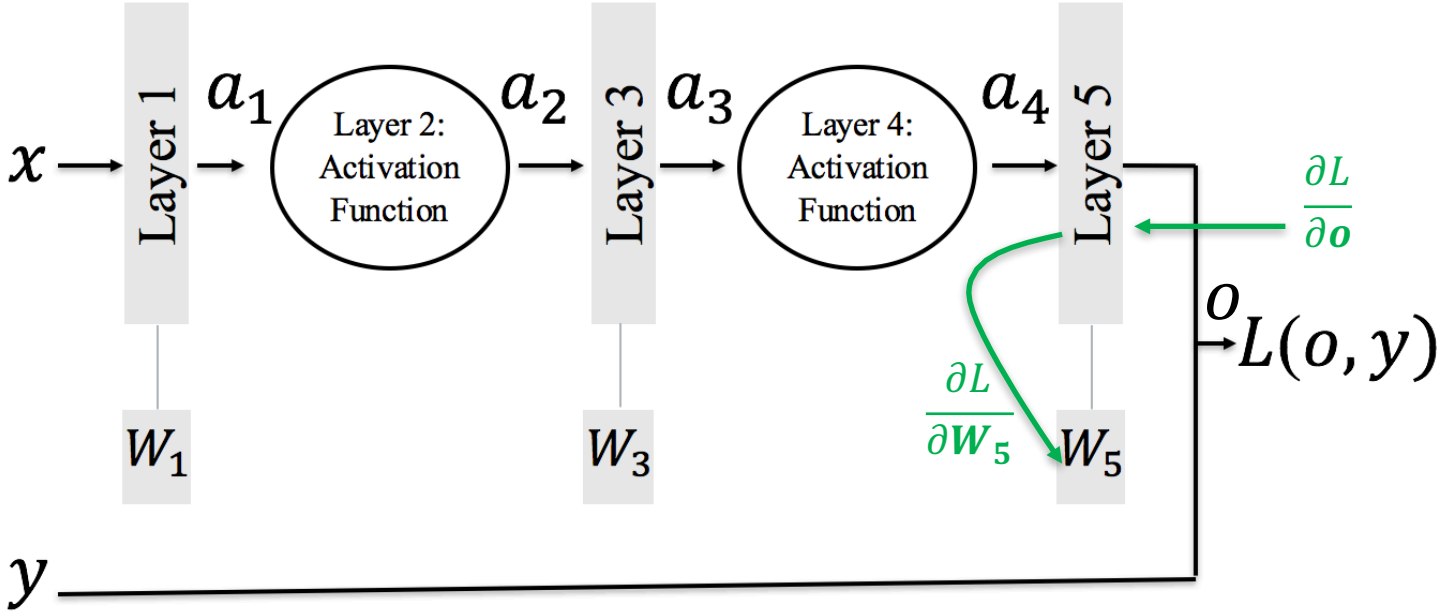
$$\frac{\partial L}{\partial \boldsymbol{W_5}} = \frac{\partial L}{\partial \boldsymbol{o}} \times \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W_5}}$$

Remember:

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

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

Multiple Layers – Back Prop: Chain Rule



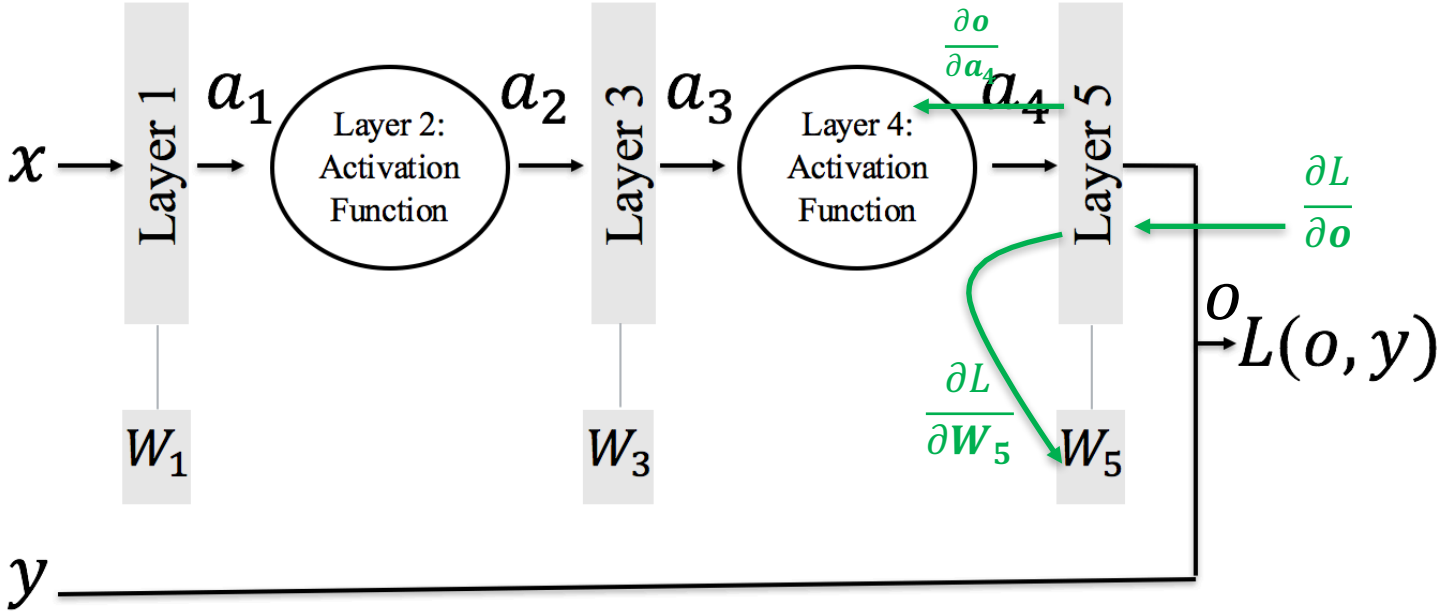
We want:

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

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



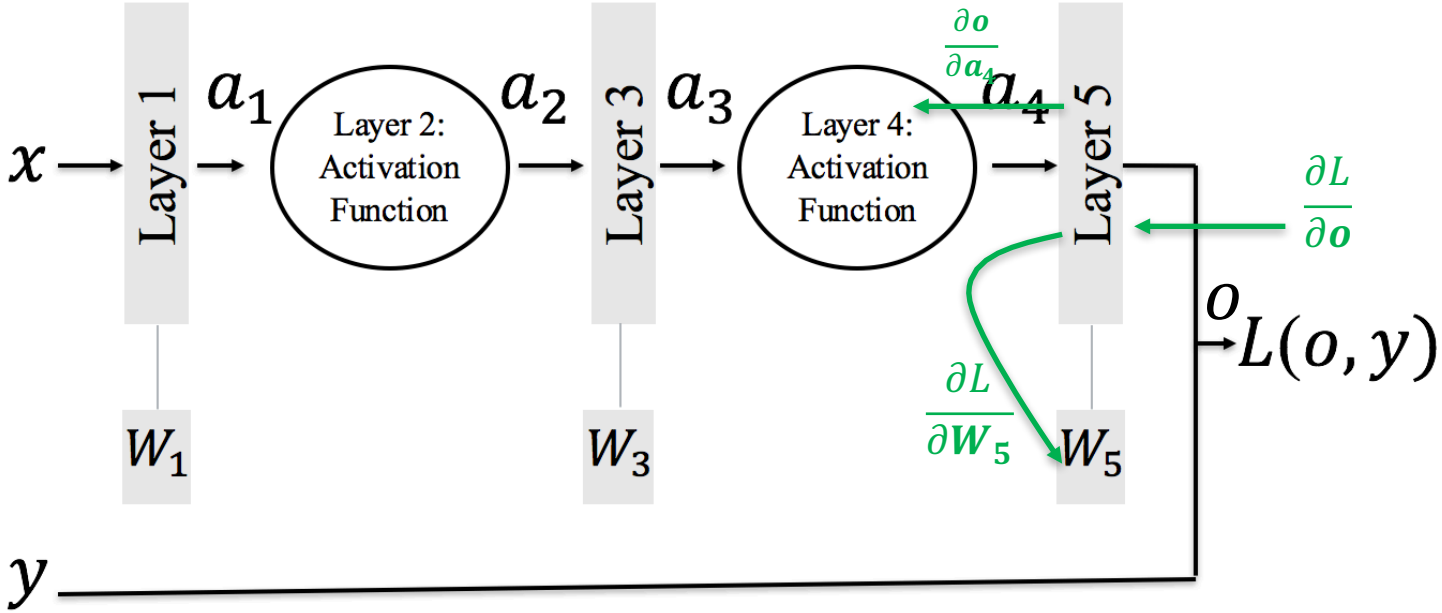
We want:

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

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

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

Multiple Layers – Back Prop: Chain Rule



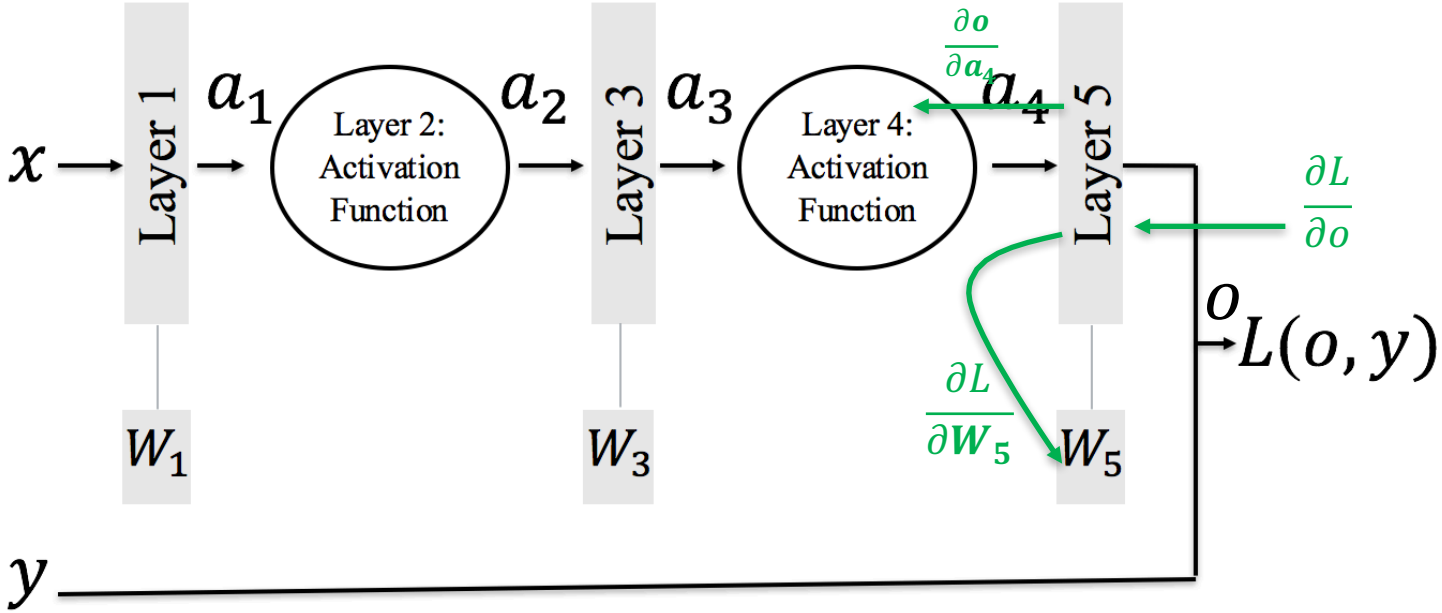
We want:

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

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)

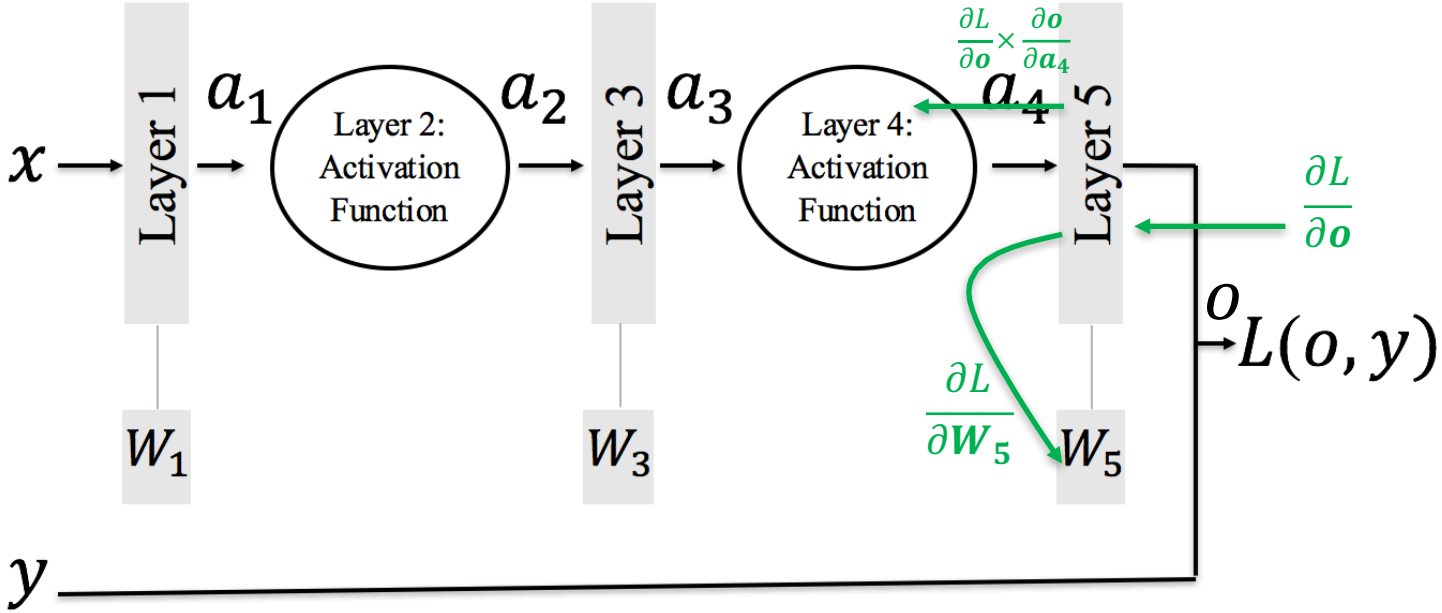
We want:

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

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:

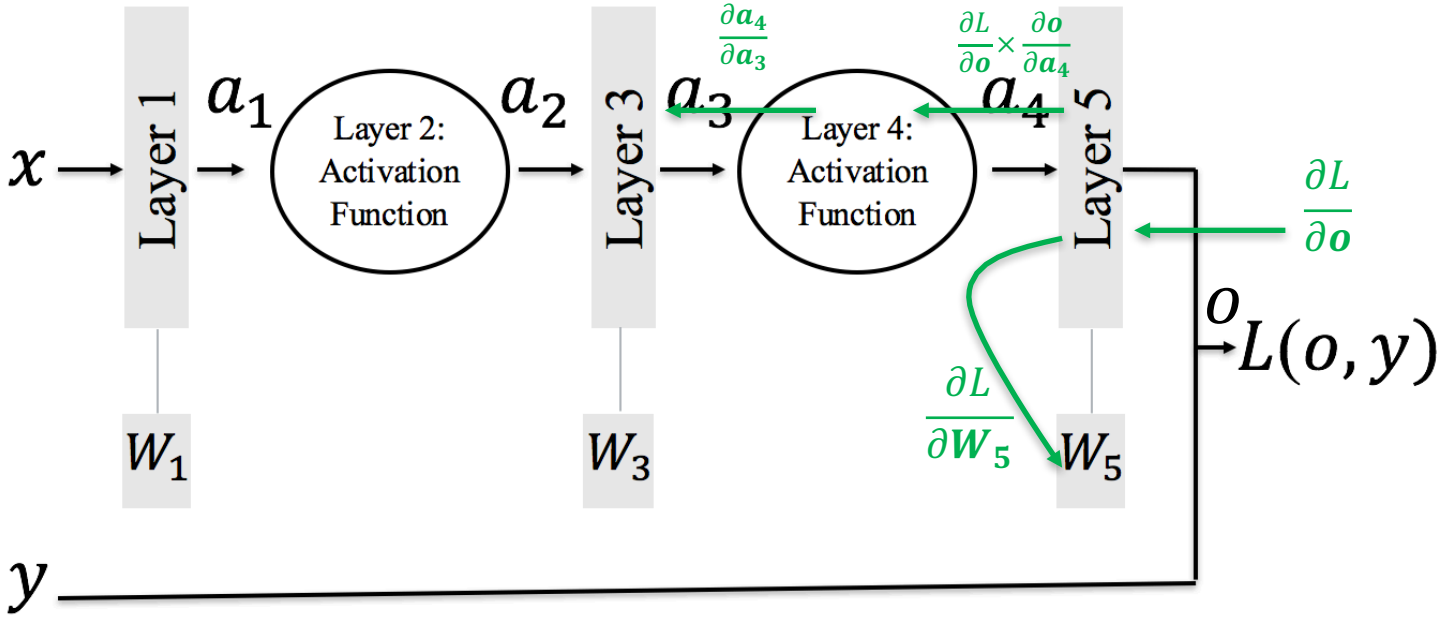
$$\frac{\partial L}{\partial \boldsymbol{o}} \times \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{a_4}} \in \mathbb{R}^{1 \times \dim(a_4)}$$

Remember:

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

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

Multiple Layers – Back Prop: Chain Rule



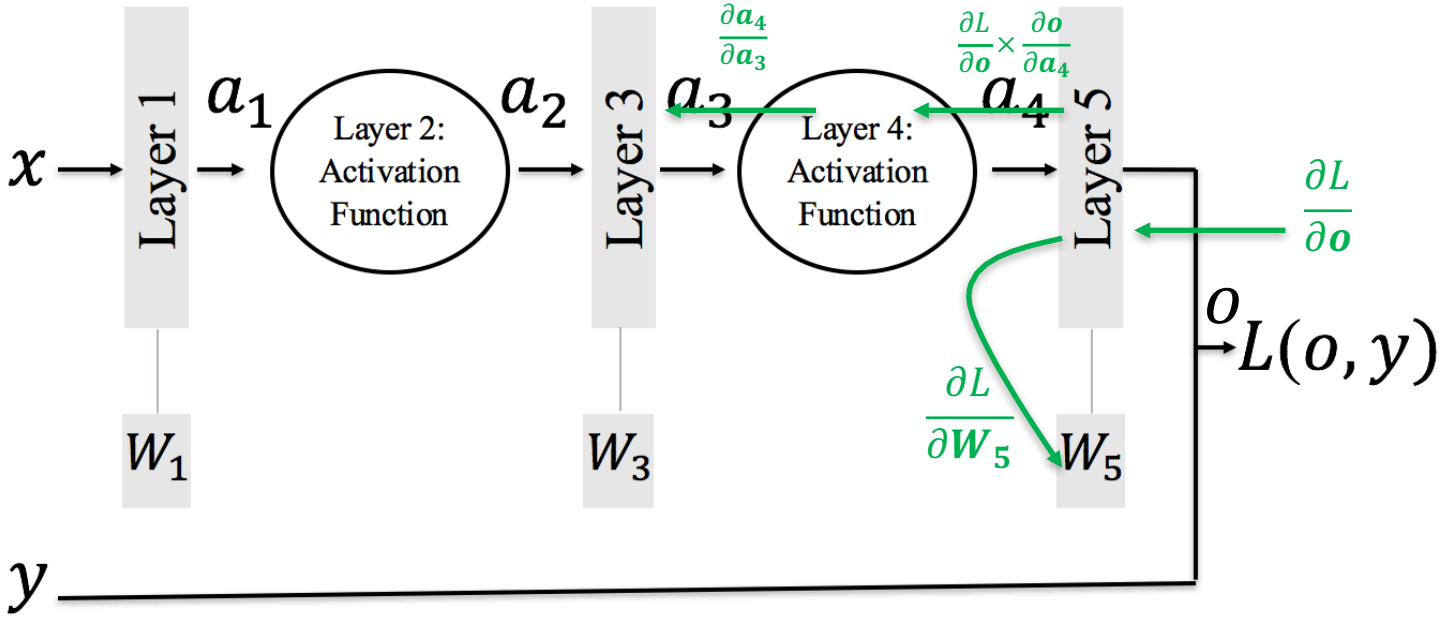
We want:

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

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:

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

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

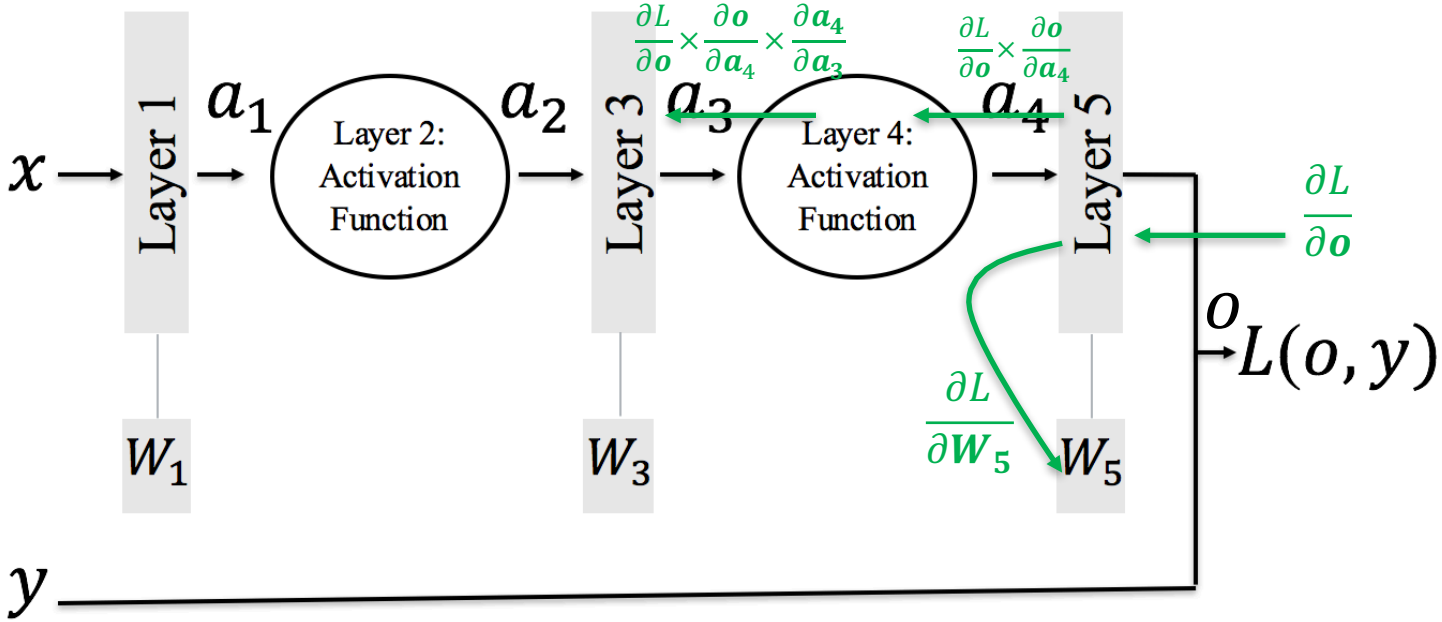
We want:

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

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:

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

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

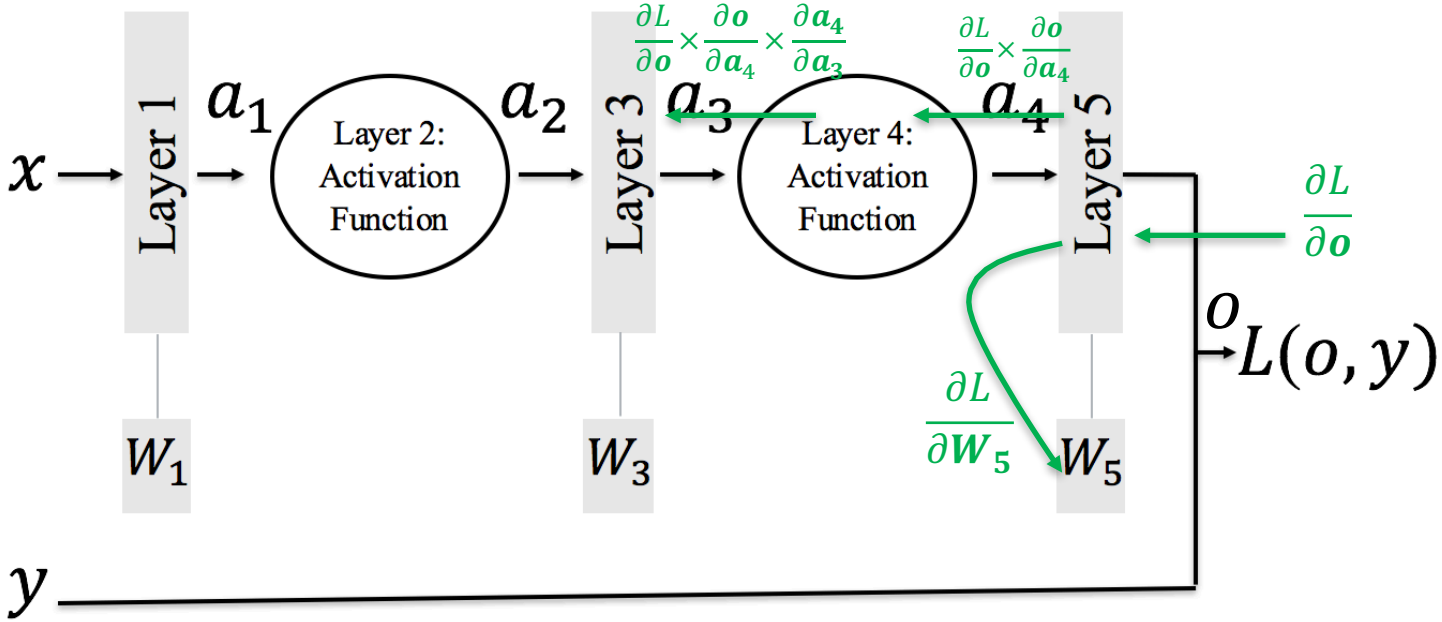
We want:

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

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



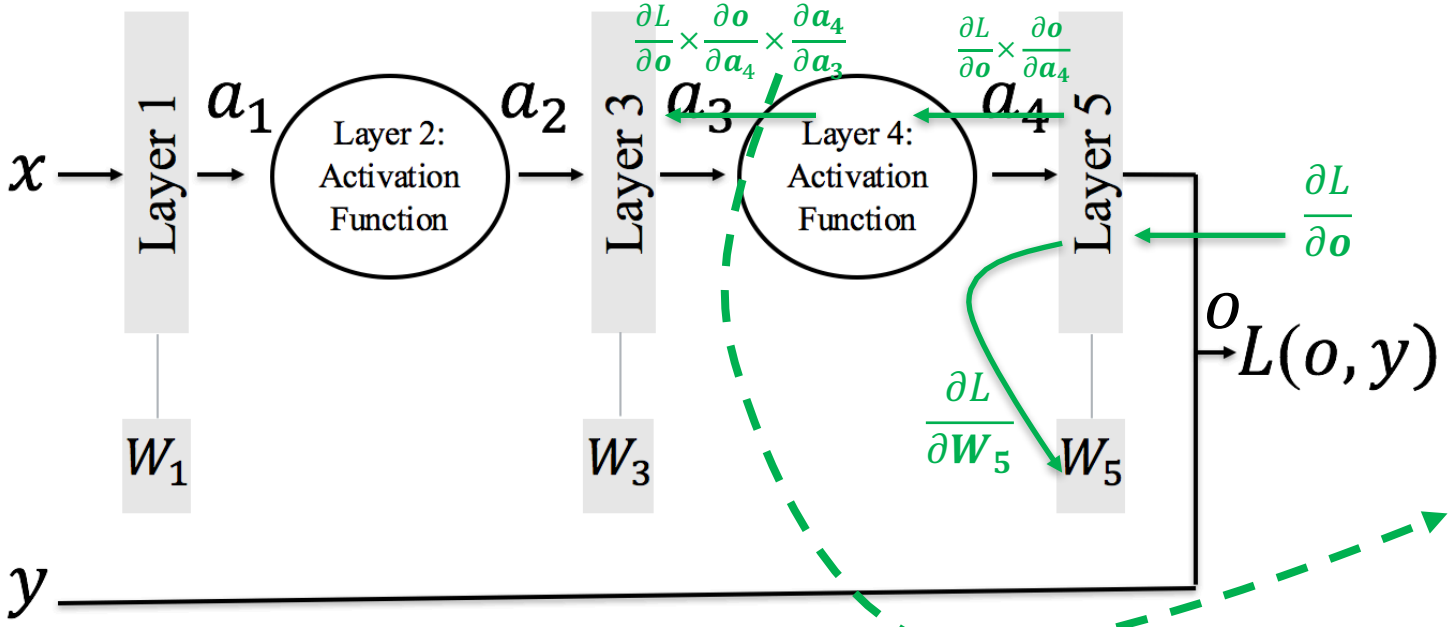
We want:

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

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:

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

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

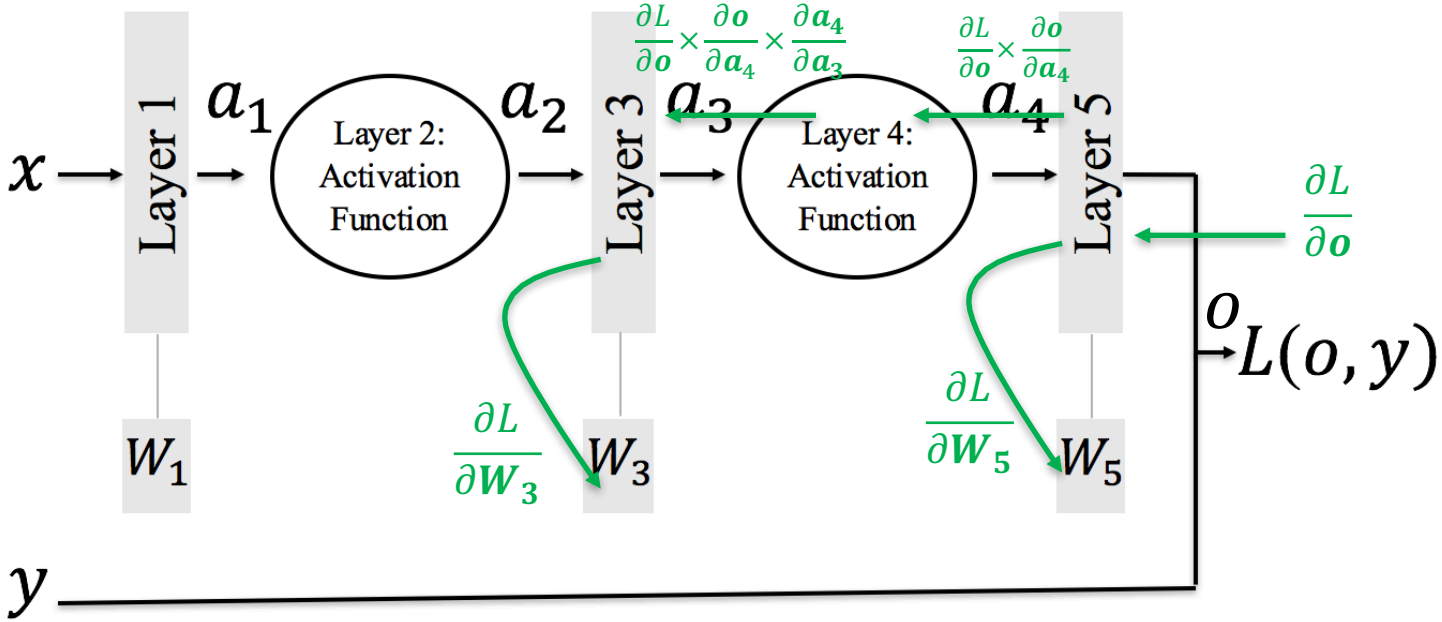
We want:

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

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:

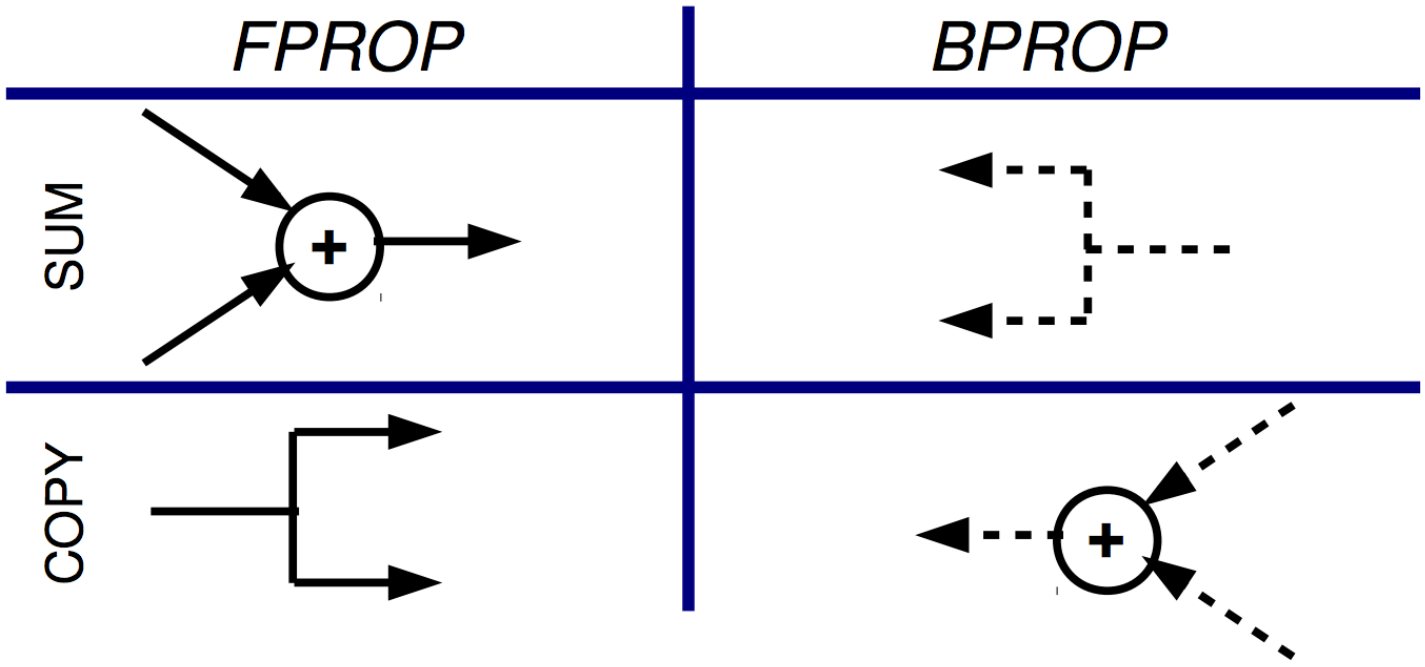
$$\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_3}, \frac{\partial L}{\partial W_5}$$

Now we can compute:

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

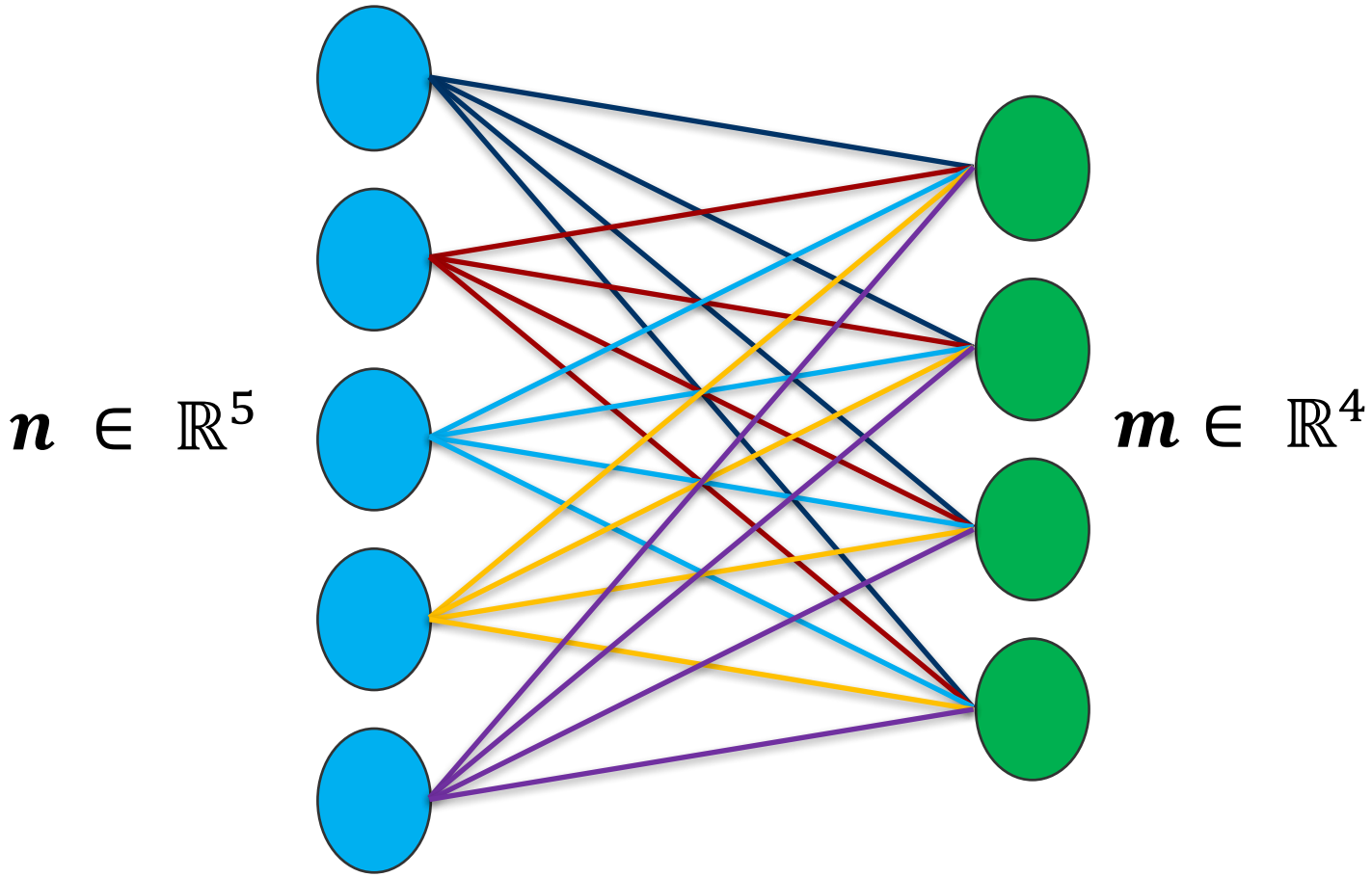
Multiple Layers – Back Prop: Chain Rule

FPROP and BPROP are duals of each other:

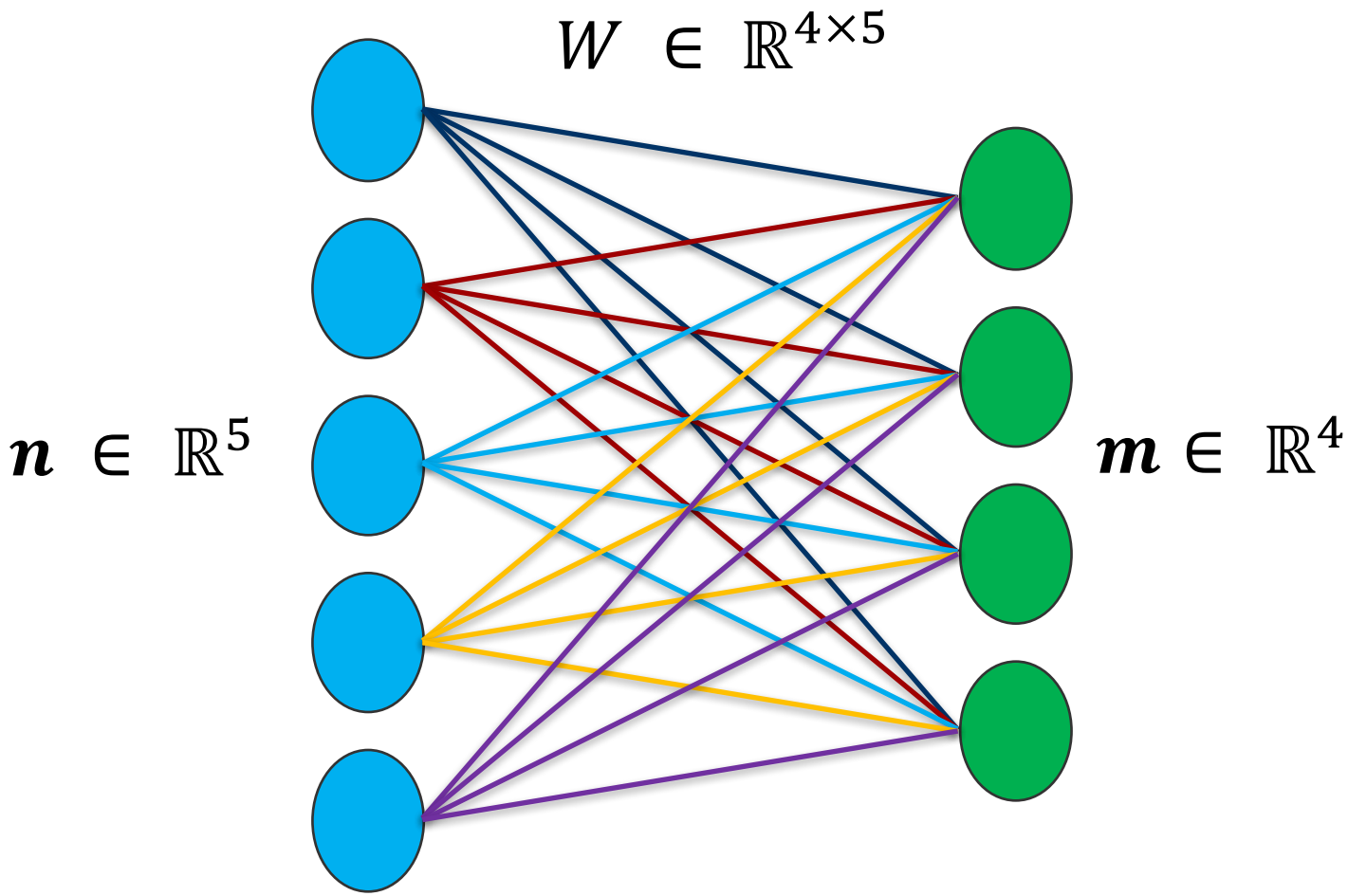


Building Blocks:
Fully Connected

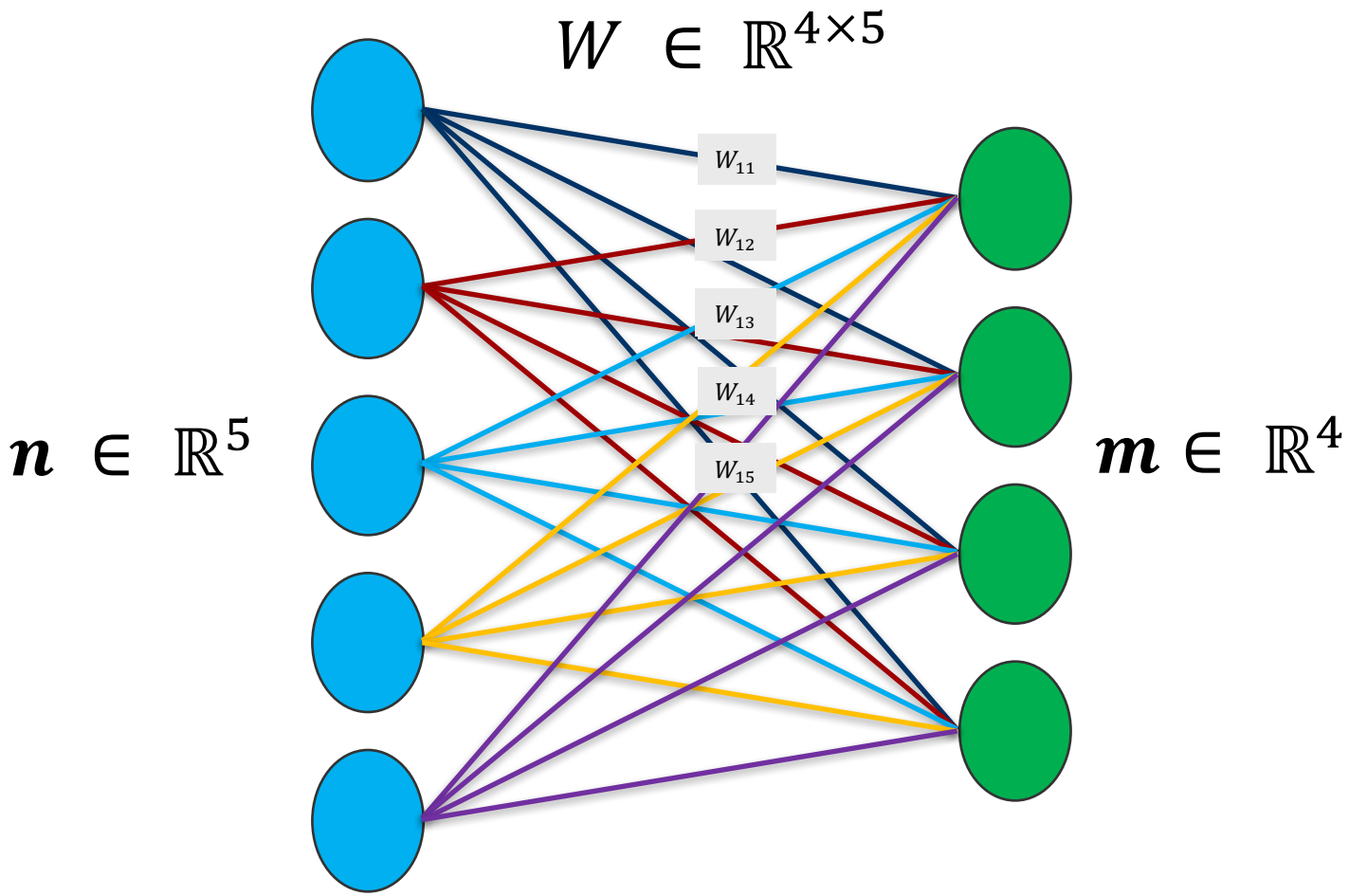
Building Blocks – Fully Connected



Building Blocks – Fully Connected

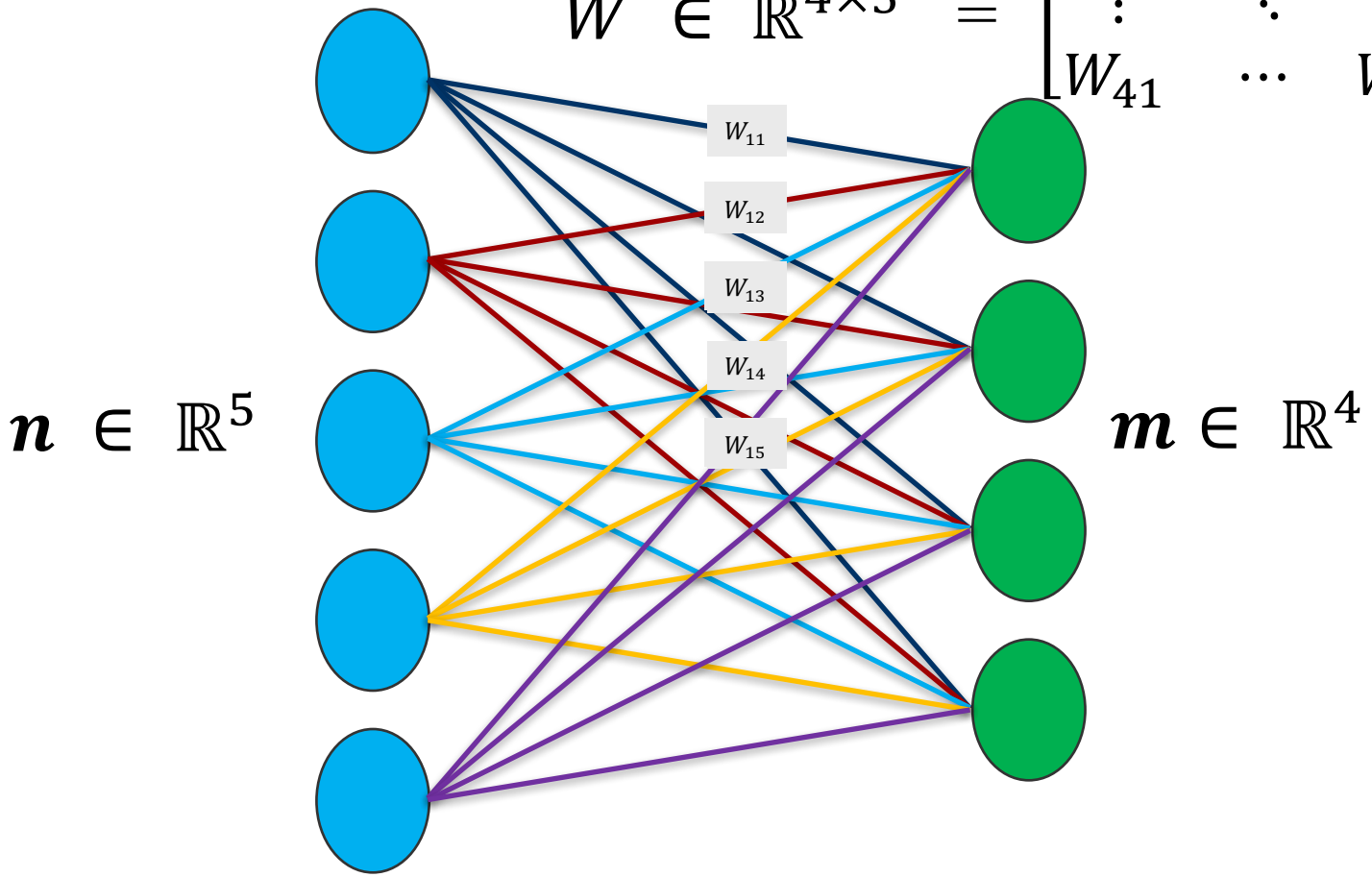


Building Blocks – Fully Connected



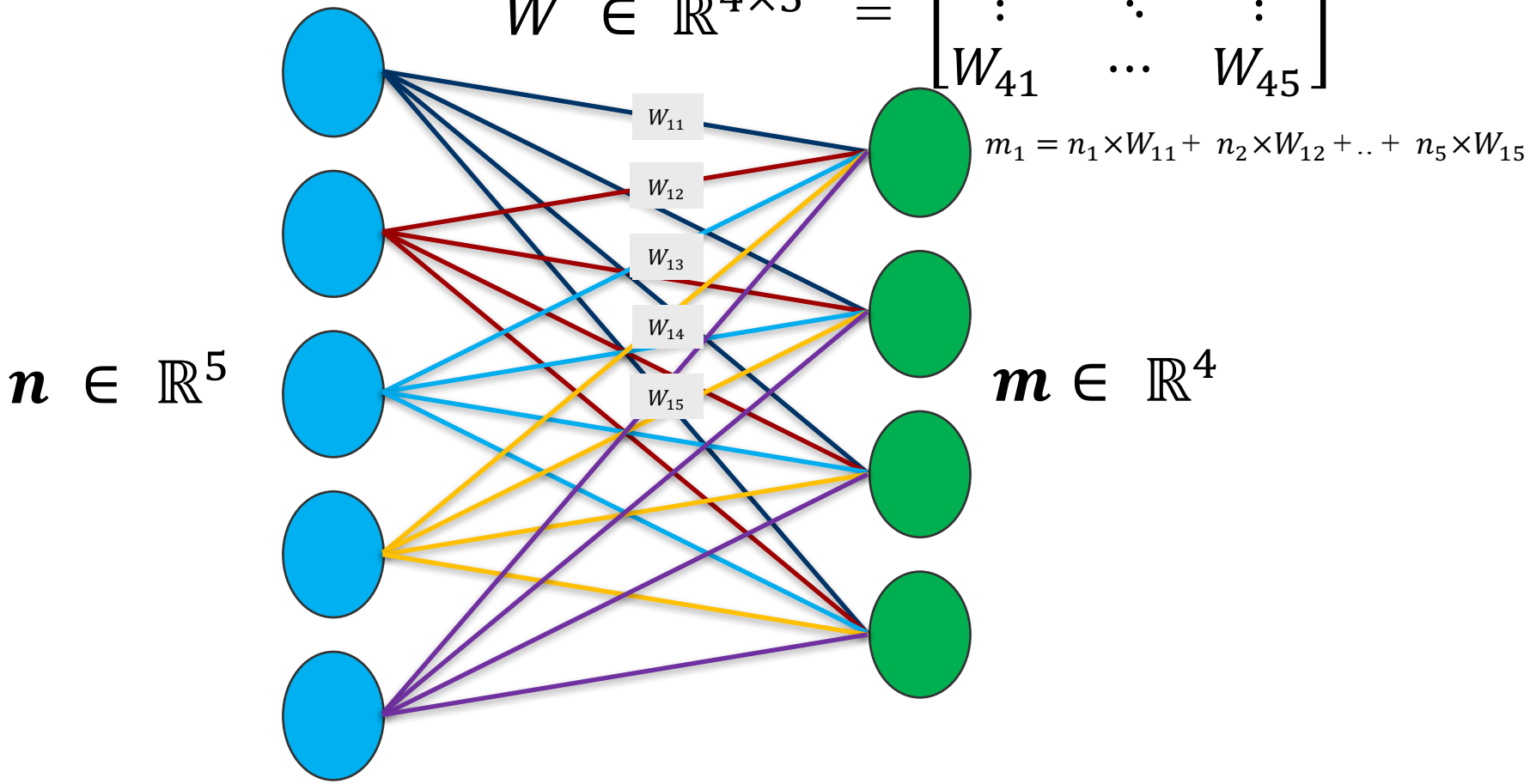
Building Blocks – Fully Connected

$$W \in \mathbb{R}^{4\times5} = \begin{bmatrix} W_{11} & \cdots & W_{15} \\ \vdots & \ddots & \vdots \\ W_{41} & \cdots & W_{45} \end{bmatrix}$$



Building Blocks – Fully Connected

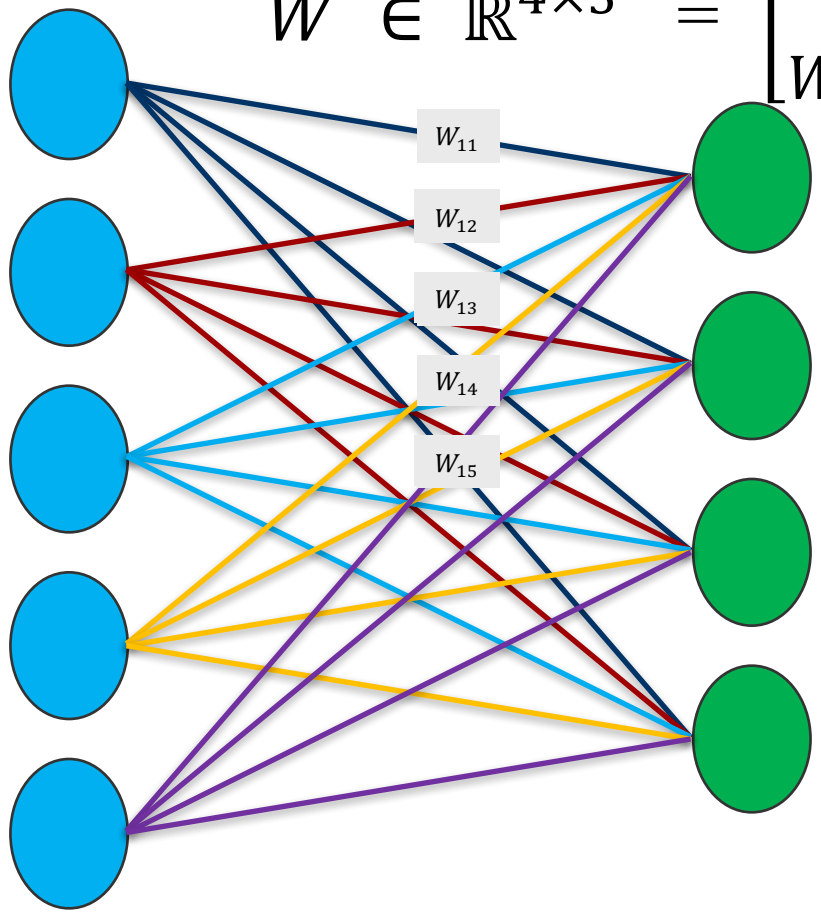
$$W \in \mathbb{R}^{4\times5} = \begin{bmatrix} W_{11} & \cdots & W_{15} \\ \vdots & \ddots & \vdots \\ W_{41} & \cdots & W_{45} \end{bmatrix}$$



$$m_1 = n_1 \times W_{11} + n_2 \times W_{12} + .. + n_5 \times W_{15}$$

Building Blocks – Fully Connected

$$W \in \mathbb{R}^{4\times5} = \begin{bmatrix} W_{11} & \cdots & W_{15} \\ \vdots & \ddots & \vdots \\ W_{41} & \cdots & W_{45} \end{bmatrix}$$



$\boldsymbol{n} \in \mathbb{R}^5$

$m_1 = n_1 \times W_{11} + n_2 \times W_{12} + .. + n_5 \times W_{15}$

$m_2 = n_1 \times W_{21} + n_2 \times W_{22} + .. + n_5 \times W_{25}$

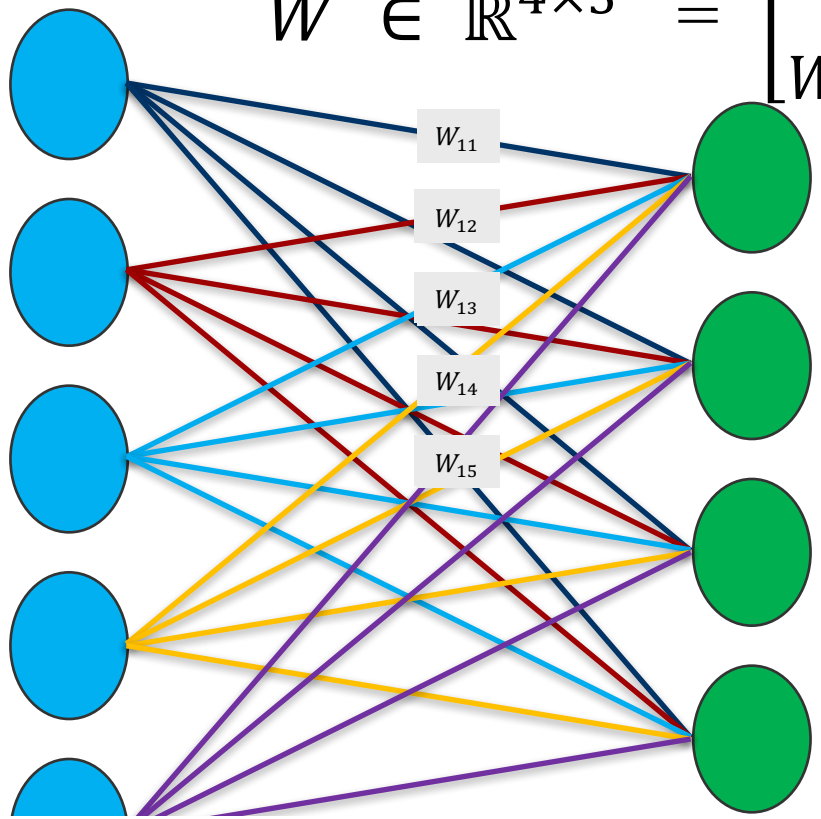
$\boldsymbol{m} \in \mathbb{R}^4$

$m_3 = n_1 \times W_{31} + n_2 \times W_{32} + .. + n_5 \times W_{35}$

$m_4 = n_1 \times W_{41} + n_2 \times W_{24} + .. + n_5 \times W_{45}$

Building Blocks – Fully Connected

$$W \in \mathbb{R}^{4\times5} = \begin{bmatrix} W_{11} & \cdots & W_{15} \\ \vdots & \ddots & \vdots \\ W_{41} & \cdots & W_{45} \end{bmatrix}$$



$\boldsymbol{n} \in \mathbb{R}^5$

$m_1 = n_1 \times W_{11} + n_2 \times W_{12} + .. + n_5 \times W_{15}$

$m_2 = n_1 \times W_{21} + n_2 \times W_{22} + .. + n_5 \times W_{25}$

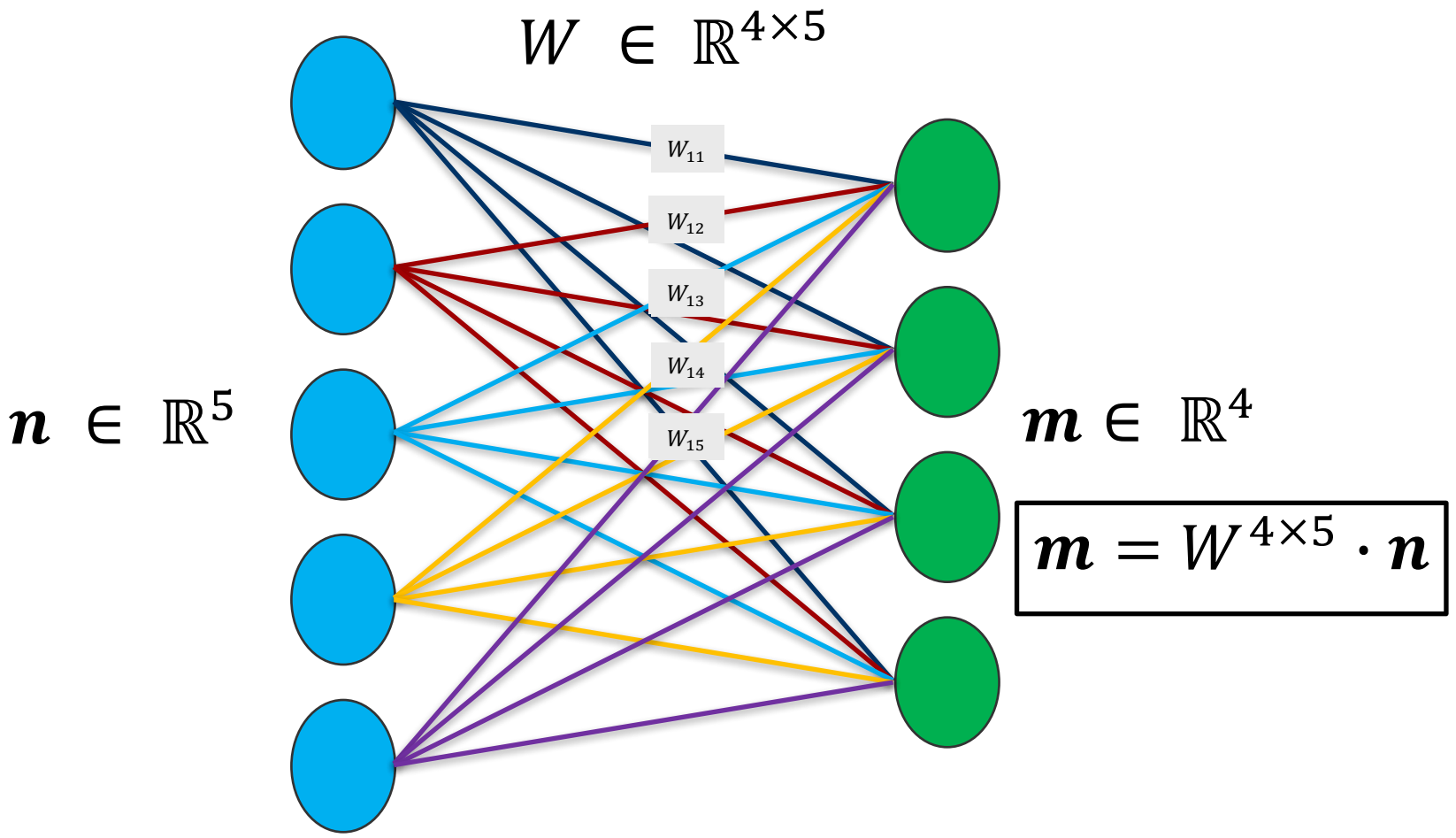
$\boldsymbol{m} \in \mathbb{R}^4$

$m_3 = n_1 \times W_{31} + n_2 \times W_{32} + .. + n_5 \times W_{35}$

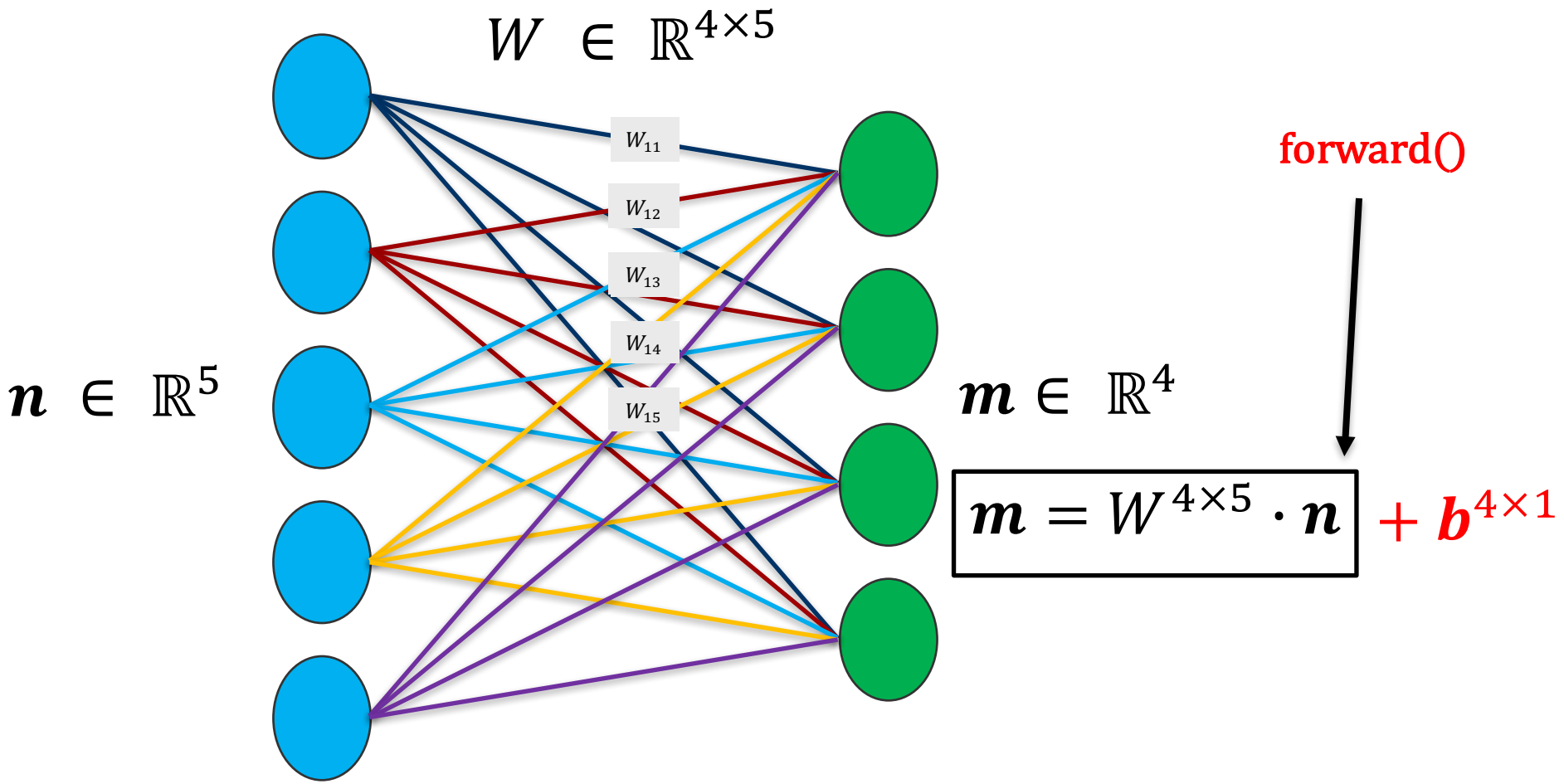
$m_4 = n_1 \times W_{41} + n_2 \times W_{24} + .. + n_5 \times W_{45}$

Why is it called fully connected?

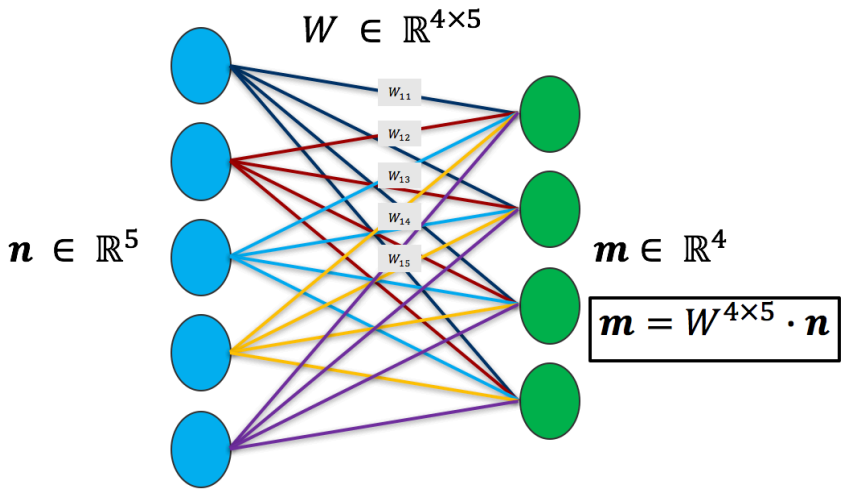
Building Blocks – Fully Connected

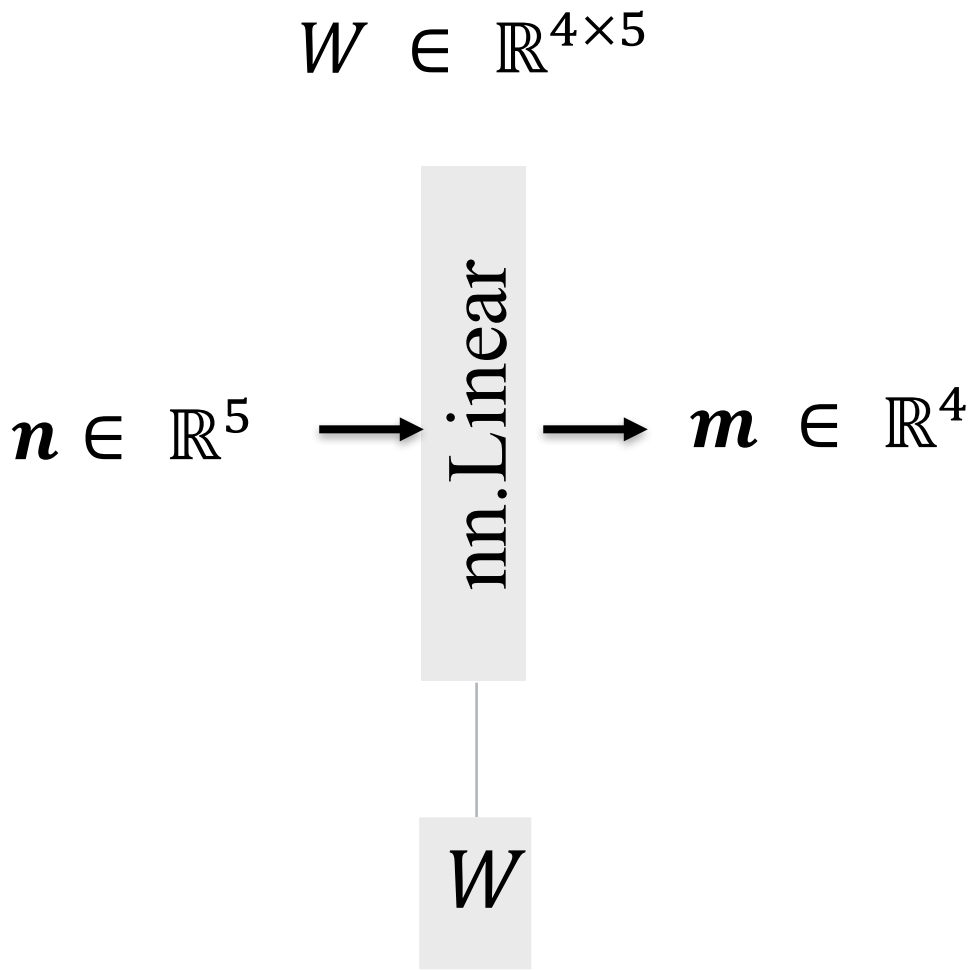


Building Blocks – Fully Connected

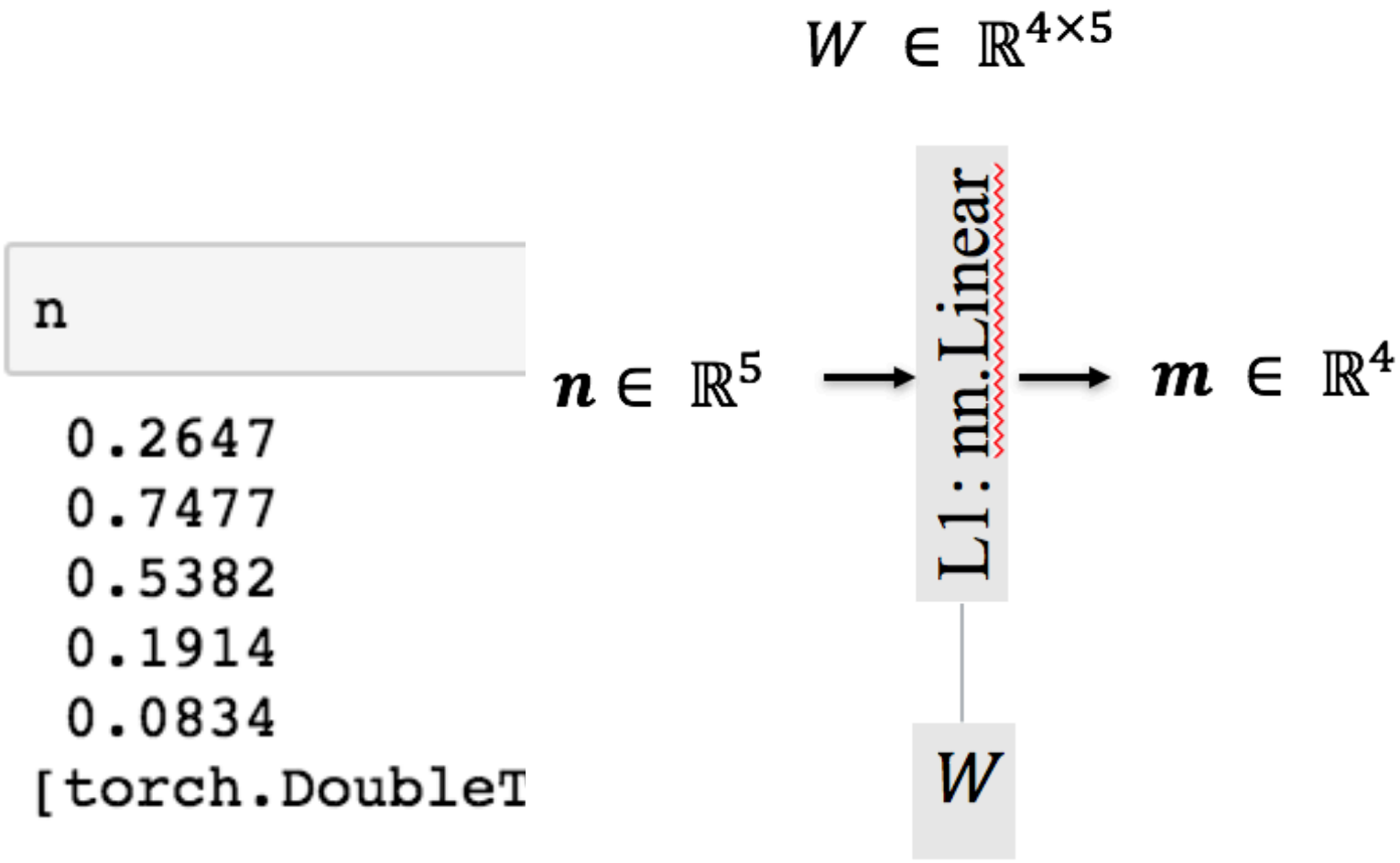


Building Blocks – Fully Connected





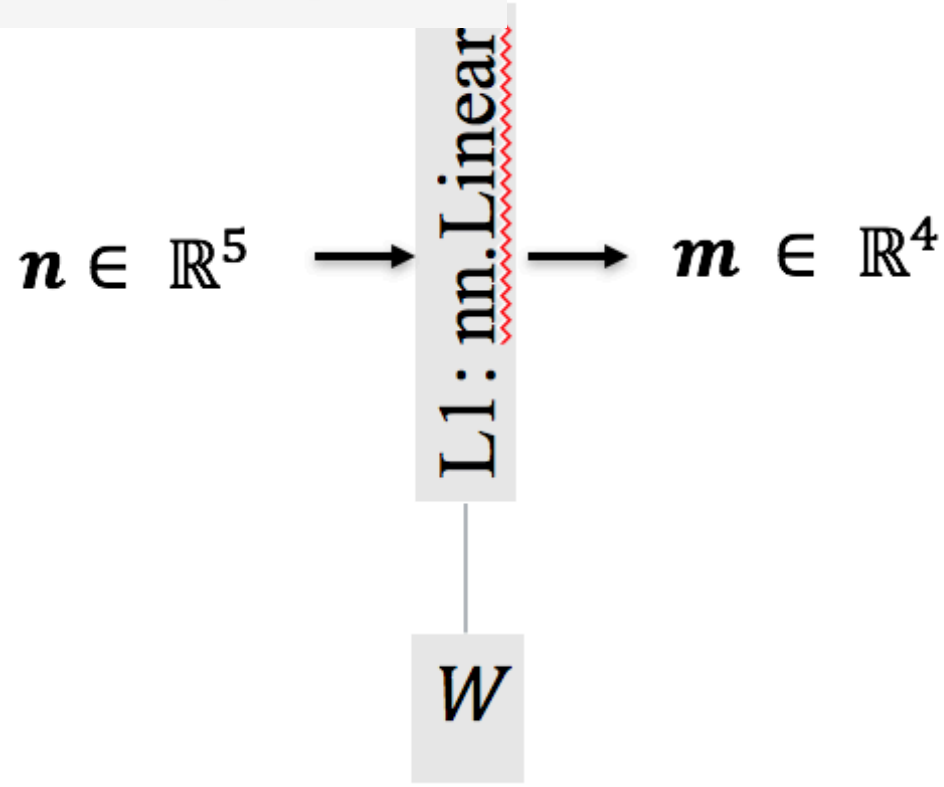
Building Blocks – Fully Connected – Forward: In Torch7



```
require 'nn';
n = torch.rand(5)
lin = nn.Linear(5,4)
m = lin:forward(n)
```

$\mathbb{R}^{4\times5}$

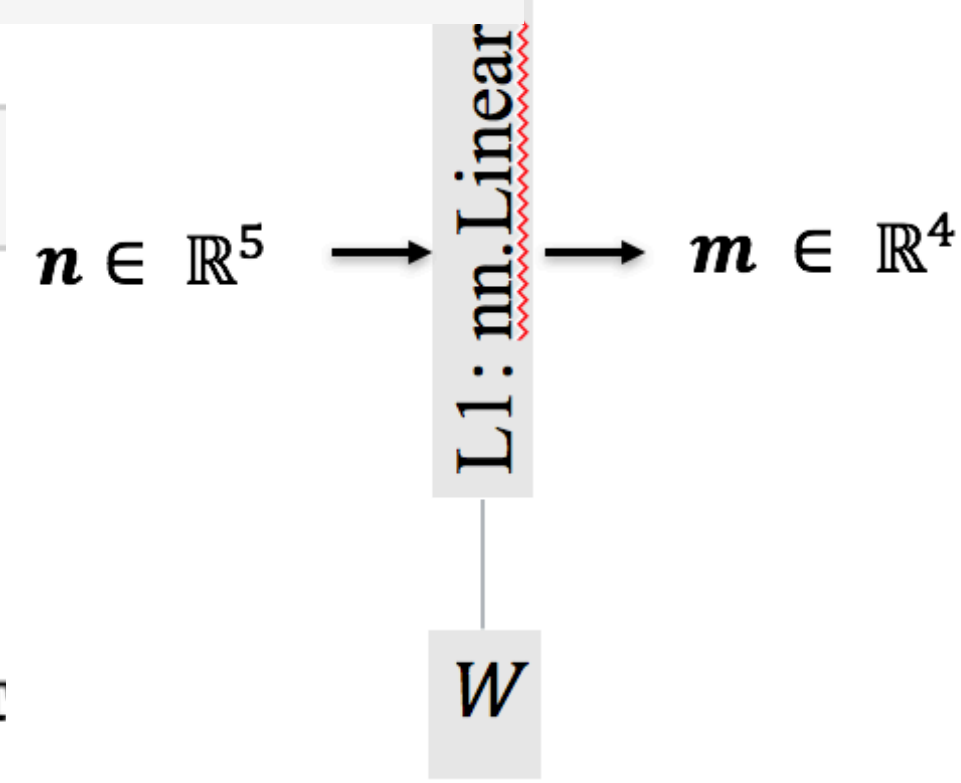
```
n

 0.2647
 0.7477
 0.5382
 0.1914
 0.0834
[torch.DoubleT
```



```
require 'nn';
n = torch.rand(5)
lin = nn.Linear(5,4)
m = lin:forward(n)
```

$\mathbb{R}^{4\times5}$

```
n

 0.2647
 0.7477
 0.5382
 0.1914
 0.0834
[torch.DoubleT
```




```
m

 0.1242
 0.2382
-0.6489
-0.2896
[torch.Doubl
```

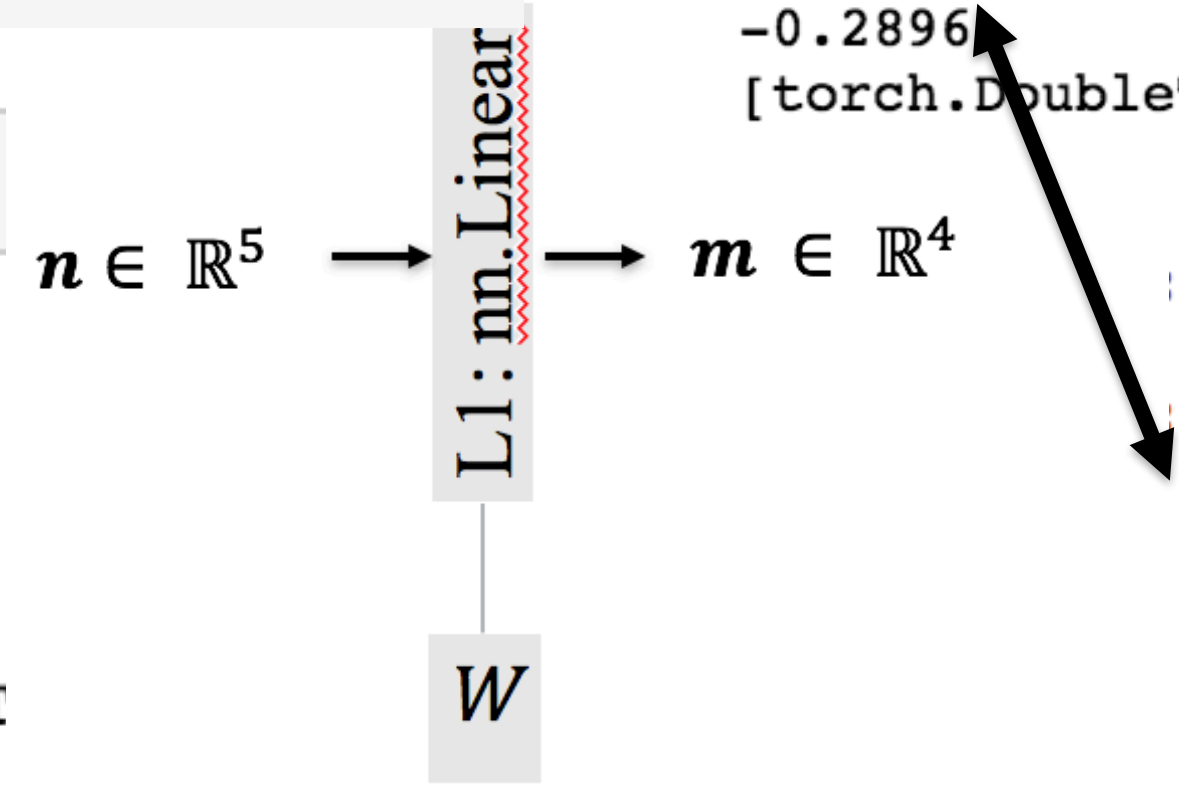
```
require 'nn';
n = torch.rand(5)
lin = nn.Linear(5,4)
m = lin:forward(n)
```

```
m_ = lin.weight*n + lin.bias
print(m_)
```

```
 0.1242
 0.2382
-0.6489
-0.2896
[torch.DoubleTensor of size 4
```

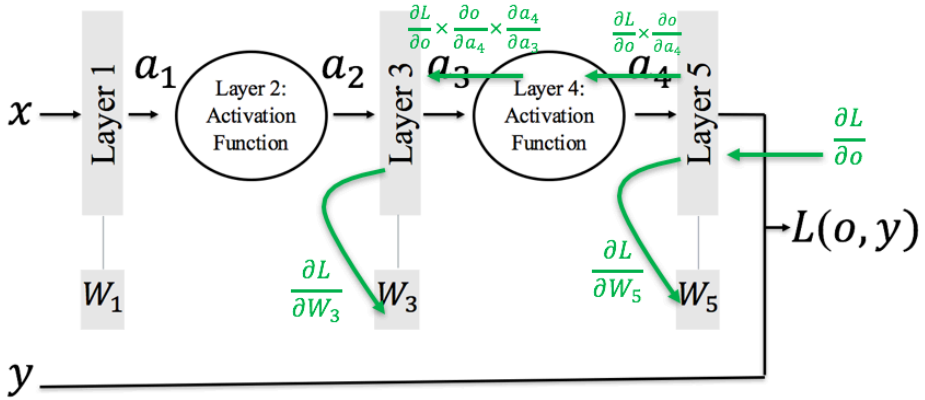
$\mathbb{R}^{4\times5}$

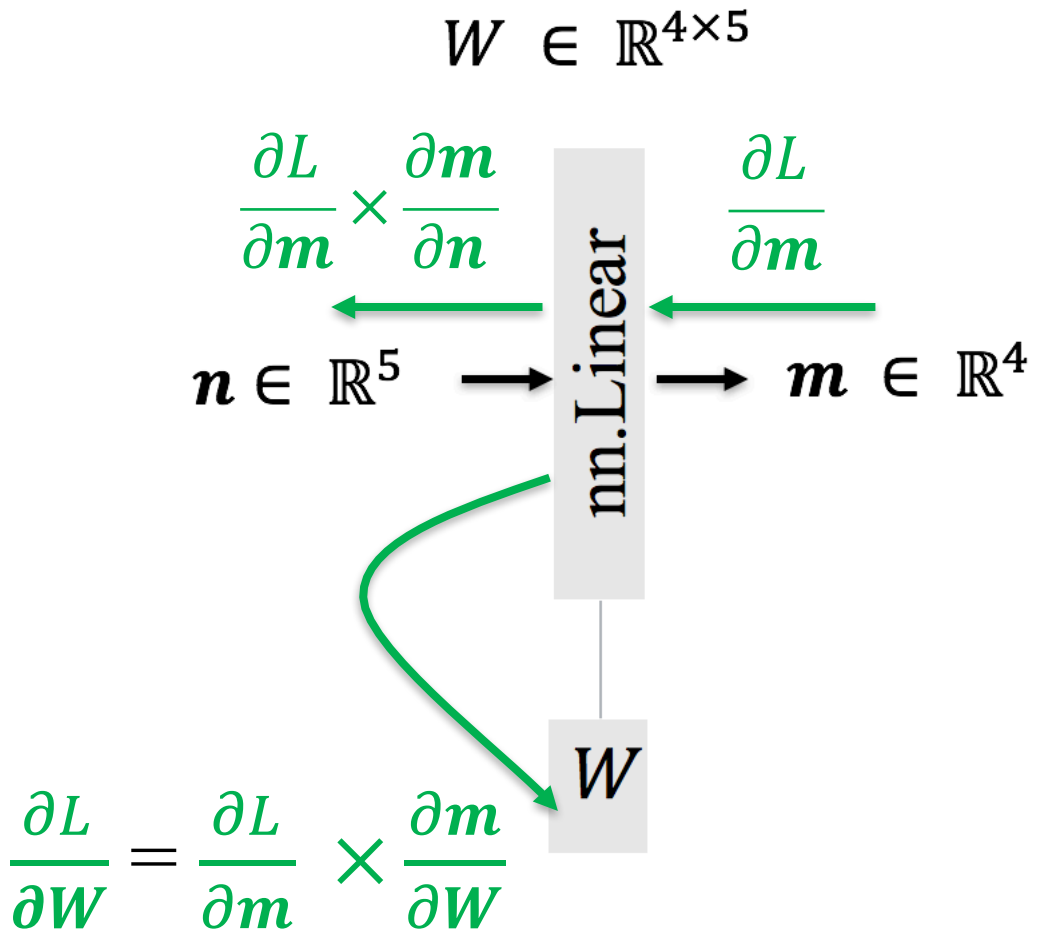
```
n

 0.2647
 0.7477
 0.5382
 0.1914
 0.0834
[torch.DoubleT
```



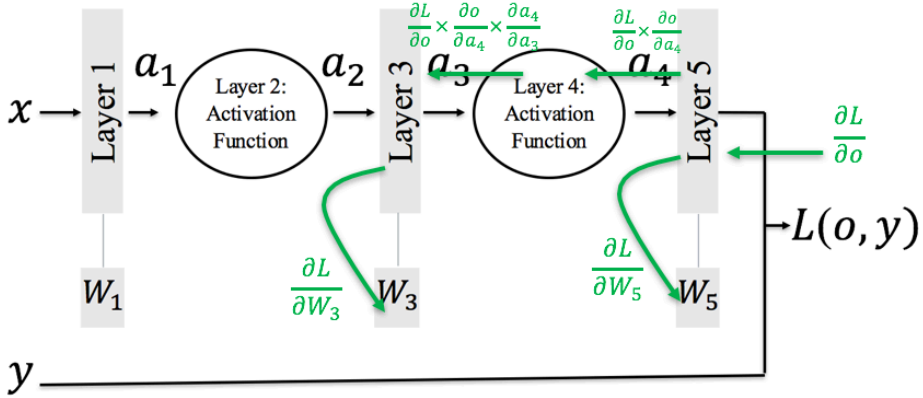

```
m

 0.1242
 0.2382
-0.6489
-0.2896
[torch.Doubl
```

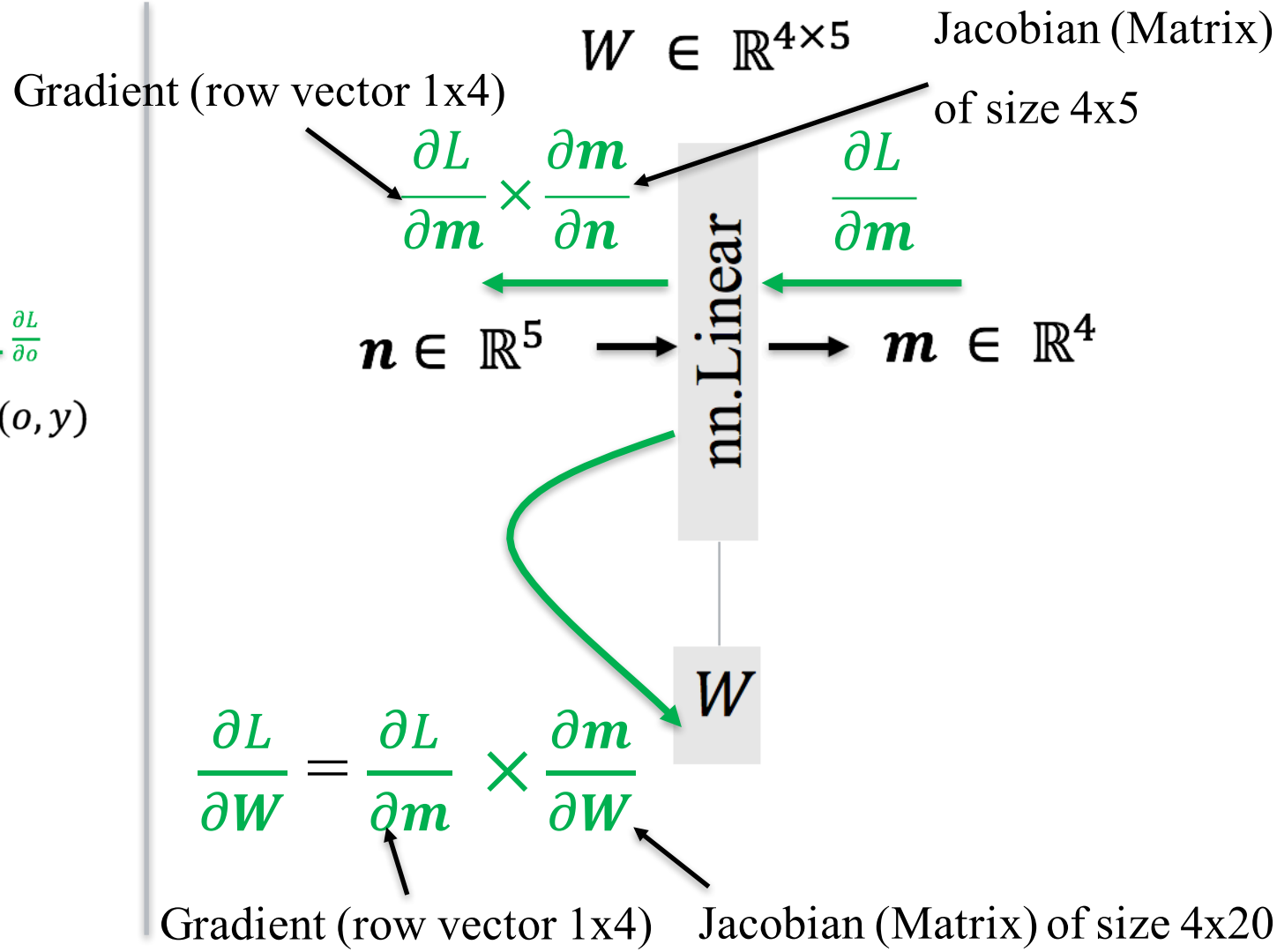
Building Blocks – Fully Connected - Backward



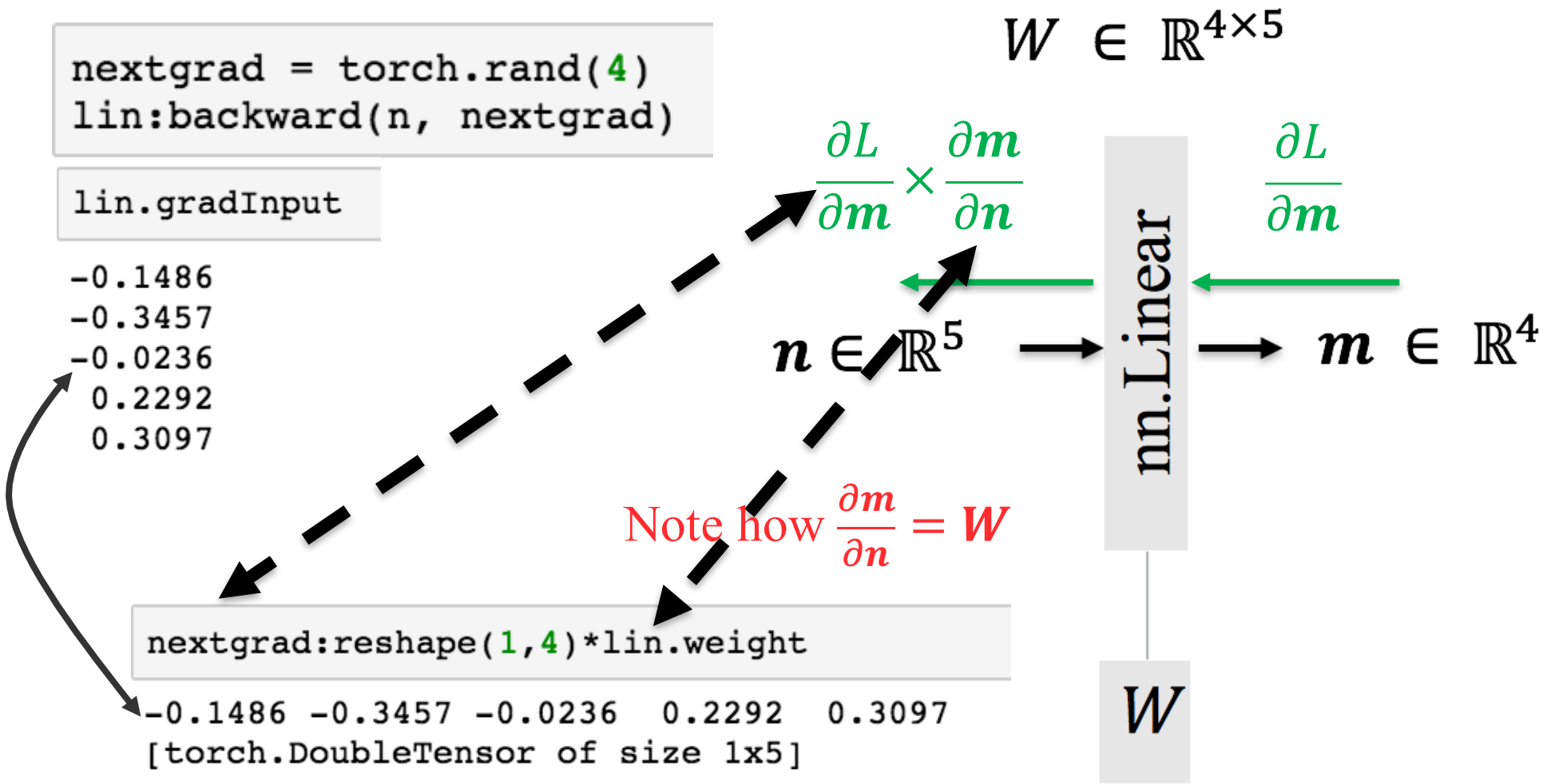


Building Blocks – Fully Connected - Backward





Fully Connected - Backward



Fully Connected - Backward

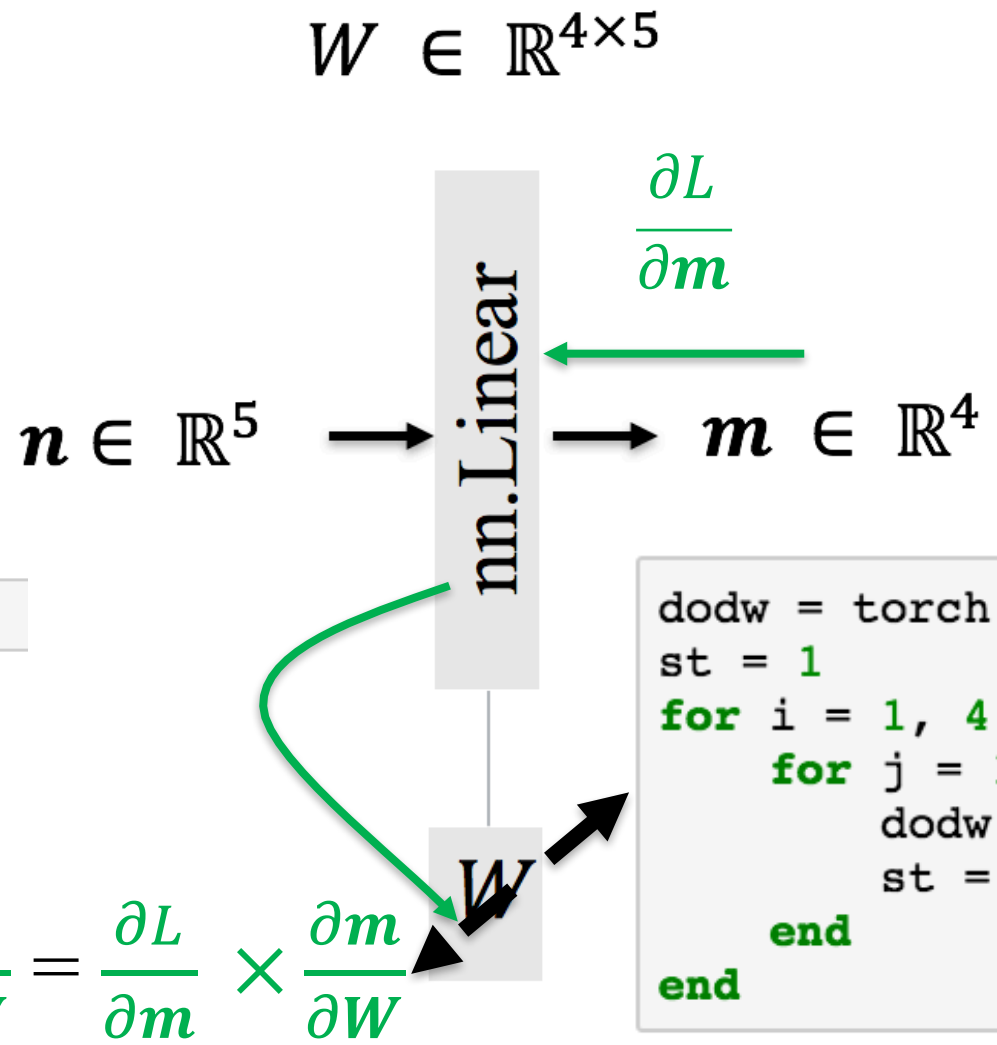
```
nextgrad = torch.rand(4)
lin:backward(n, nextgrad)
```

```
lin.gradWeight
```

```
 0.2135  0.6033  0.4343  0.1544  0.0673
 0.0556  0.1572  0.1132  0.0402  0.0175
 0.0850  0.2402  0.1729  0.0615  0.0268
 0.1717  0.4850  0.3491  0.1242  0.0541
[torch.DoubleTensor of size 4x5]
```

```
(nextgrad:reshape(1,4) * dodw):reshape(4,5)
```

```
 0.2135  0.6033  0.4343  0.1544  0.0673
 0.0556  0.1572  0.1132  0.0402  0.0175
 0.0850  0.2402  0.1729  0.0615  0.0268
 0.1717  0.4850  0.3491  0.1242  0.0541
[torch.DoubleTensor of size 4x5]
```

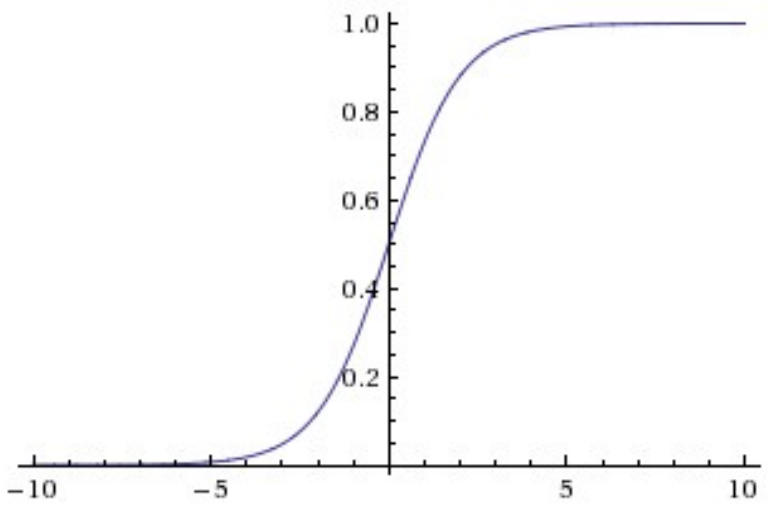



```
dodw = torch.Tensor(4,20)
st = 1
for i = 1, 4 do
    for j = 1, 5 do
        dodw[i][st]=n[j]
        st = st + 1
    end
end
```

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

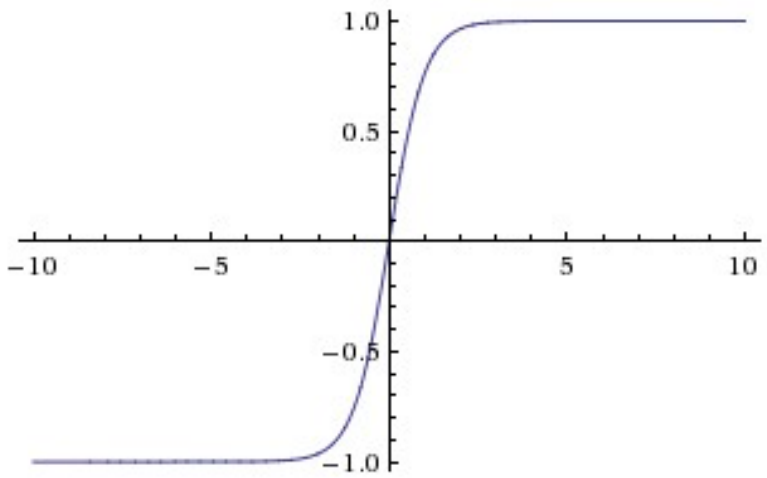
3 problems:

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 :(

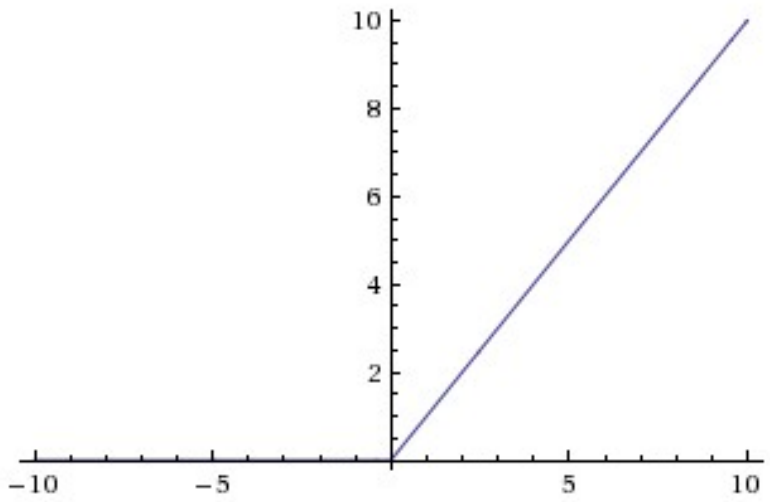


Tanh

[LeCun et al., 1991]

Activation Functions: ReLU

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



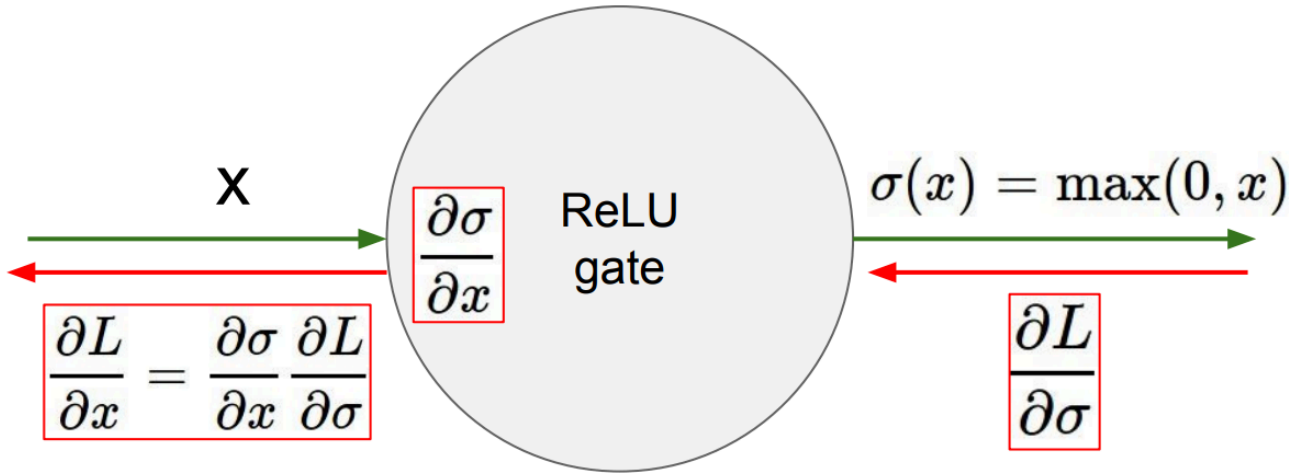
ReLU (Rectified Linear Unit)

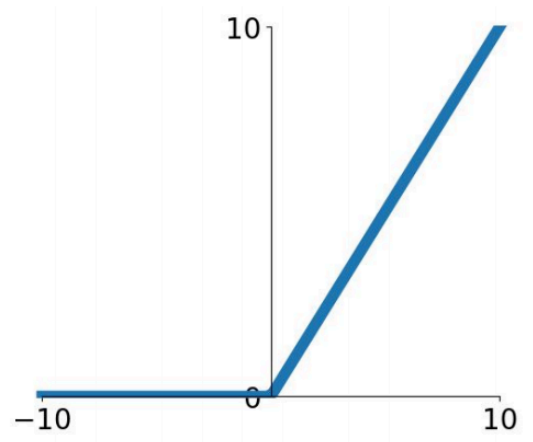
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

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

[Nair and Hinton et al., 2010]

Activation Functions: ReLU - Problem

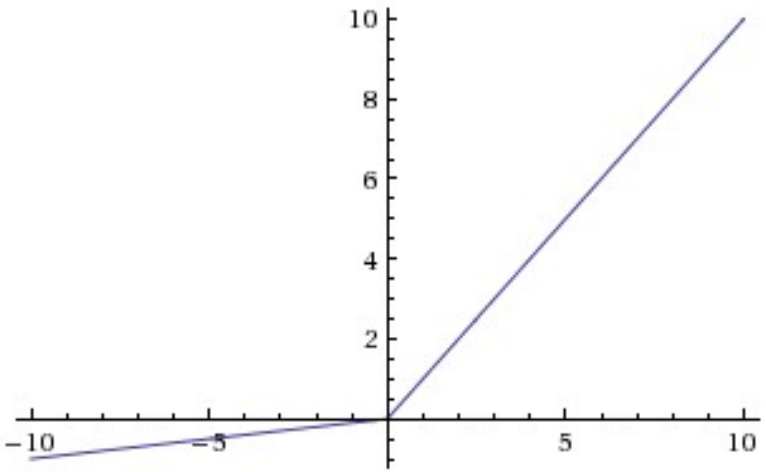




- What happens when x = -10?
- What happens when x = 0?
- What happens when x = 10?

Activation Functions: Leaky ReLU

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



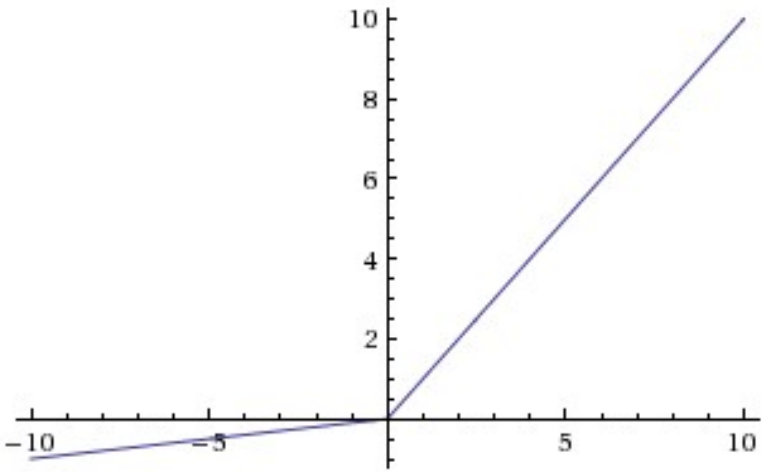
Leaky ReLU

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

[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)

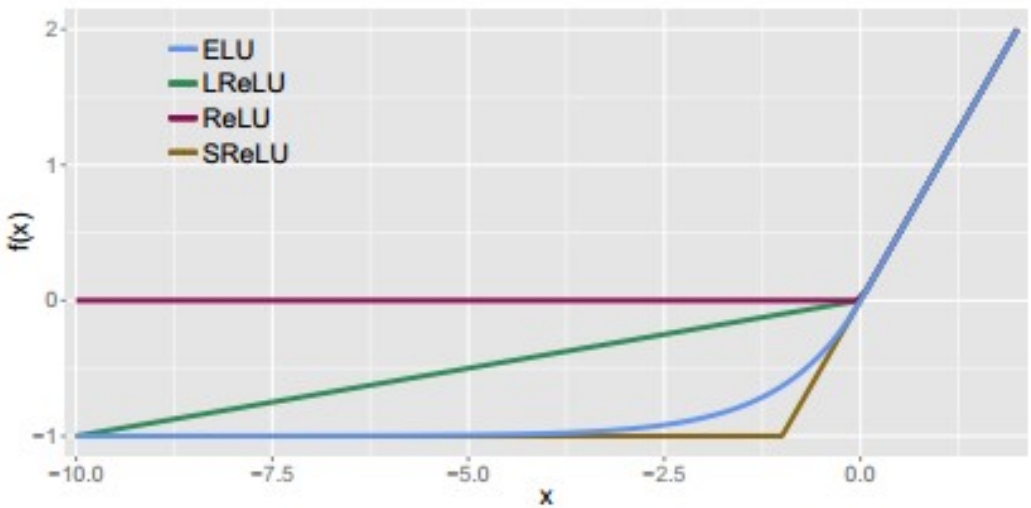
$$f(x) = \max(\alpha x, x)$$

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

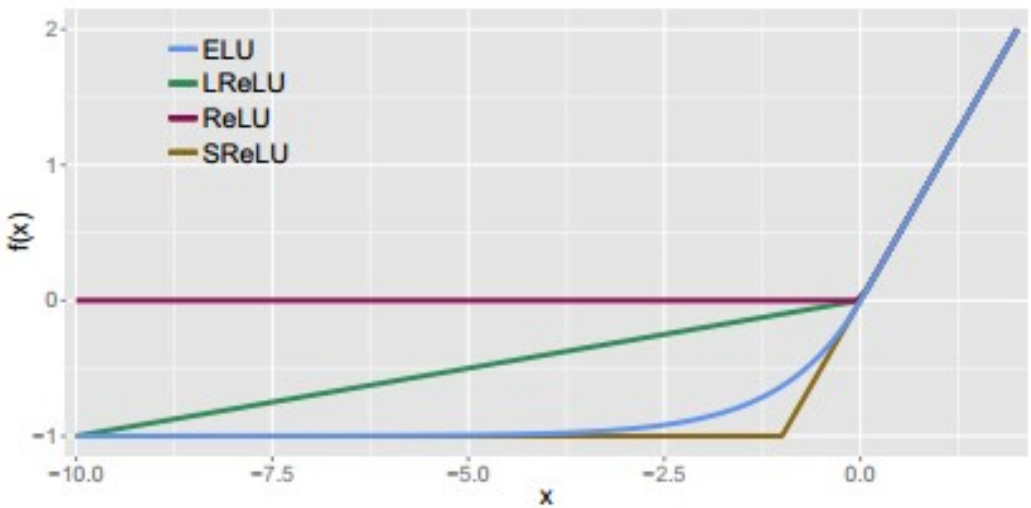
- Computation requires exp()

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

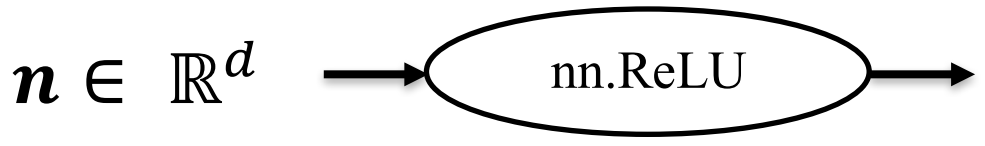
- Computation requires exp()

[Clevert et al., 2015]

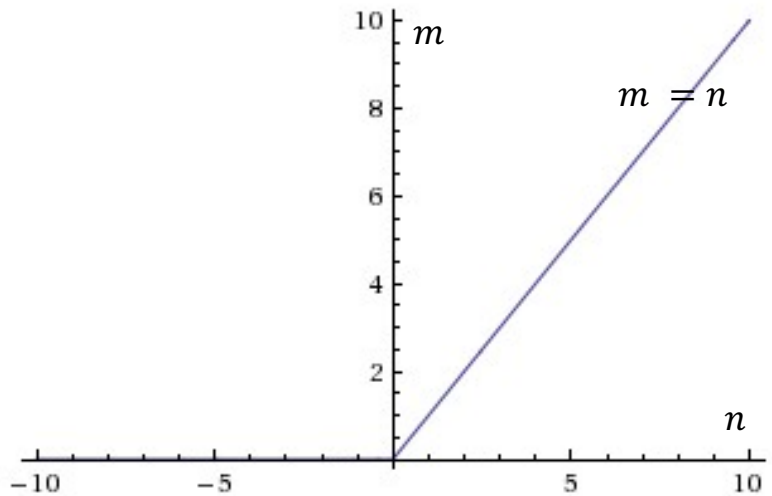
TLDR: In practice:

- 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)



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



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

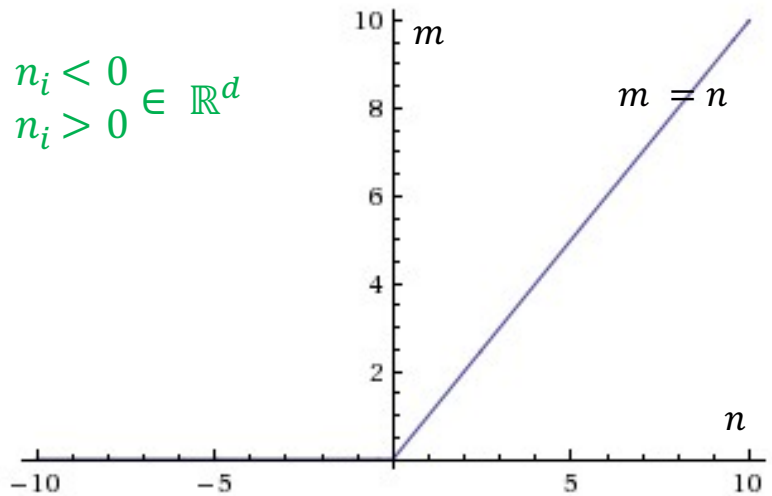
Activation Function: ReLU (details)

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

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

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

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



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

Activation Function: ReLU (forward)

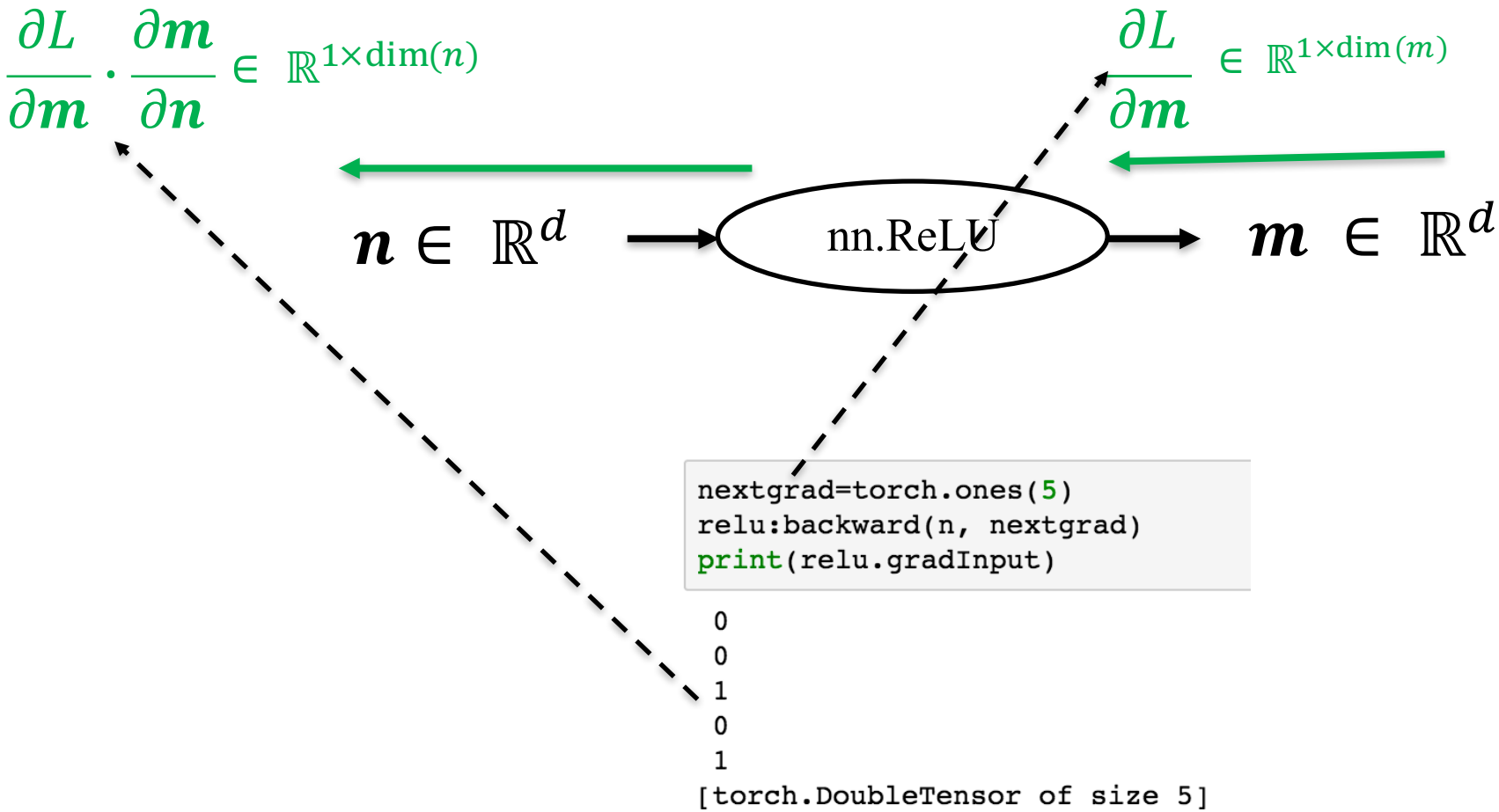

```
require 'nn';
n = torch.rand(5) - 0.5
print(n)
```

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

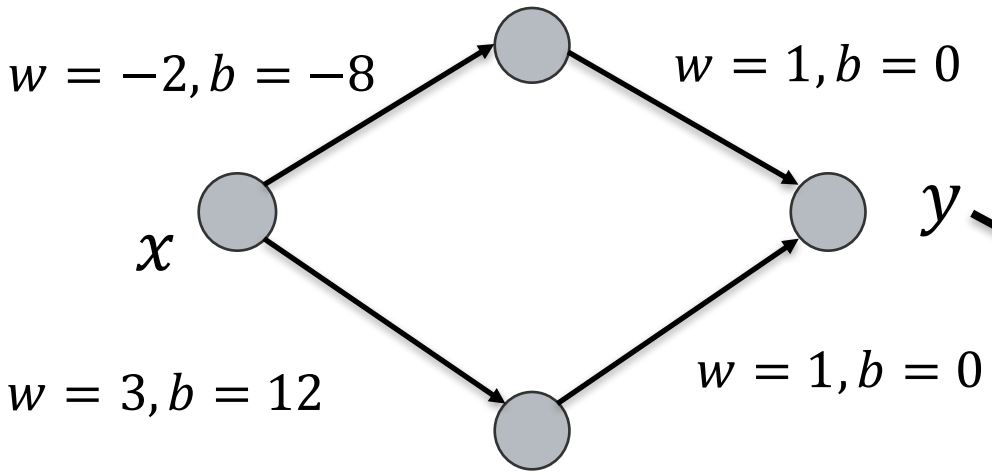
```
relu = nn.ReLU()
m = relu:forward(n)
print(m)
```

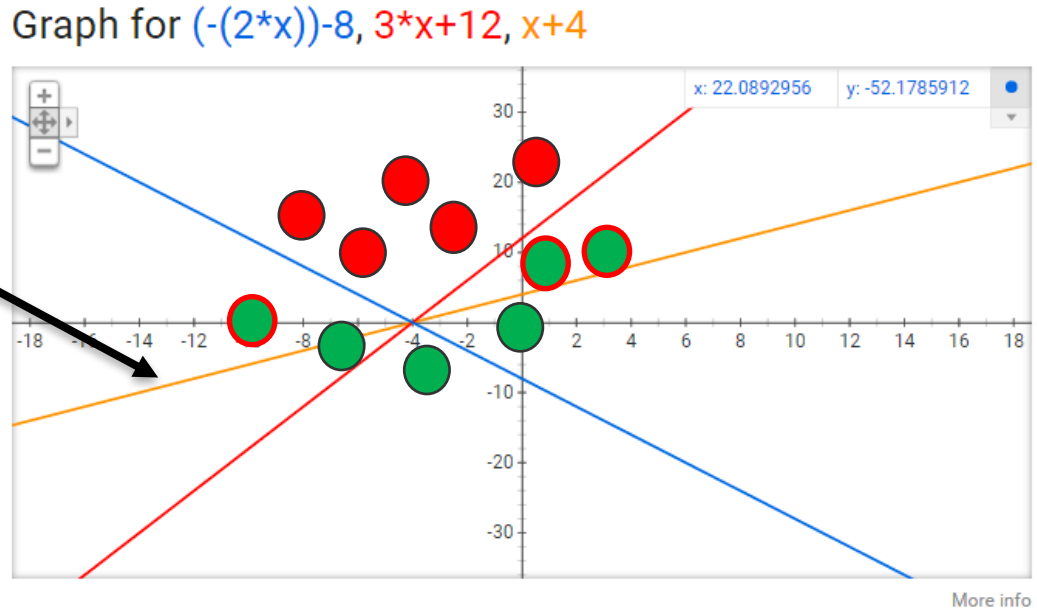
```
 0.0000
 0.0000
 0.4794
 0.0000
 0.4201
[torch.DoubleTensor of size 5]
```

Activation Function: ReLU (backward)

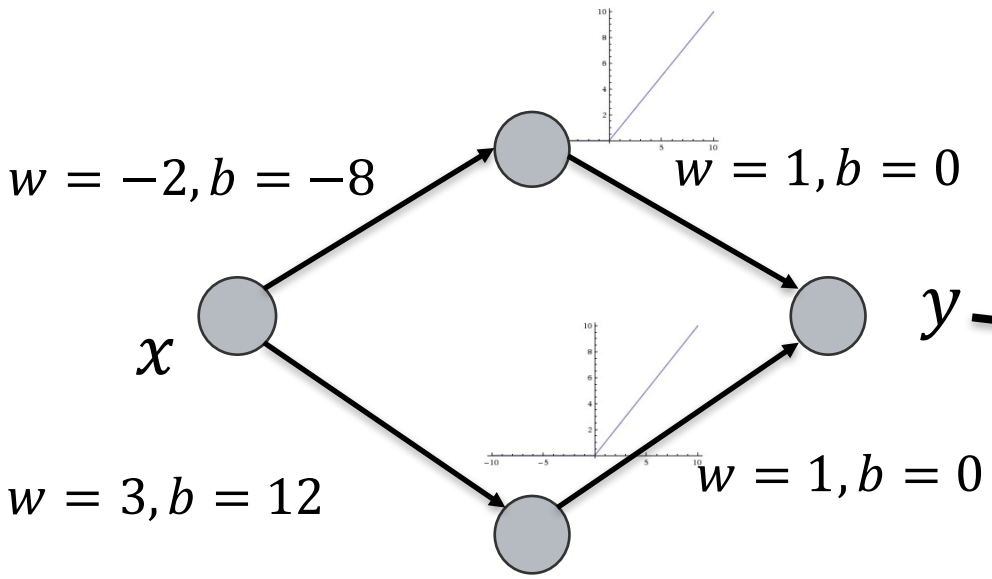


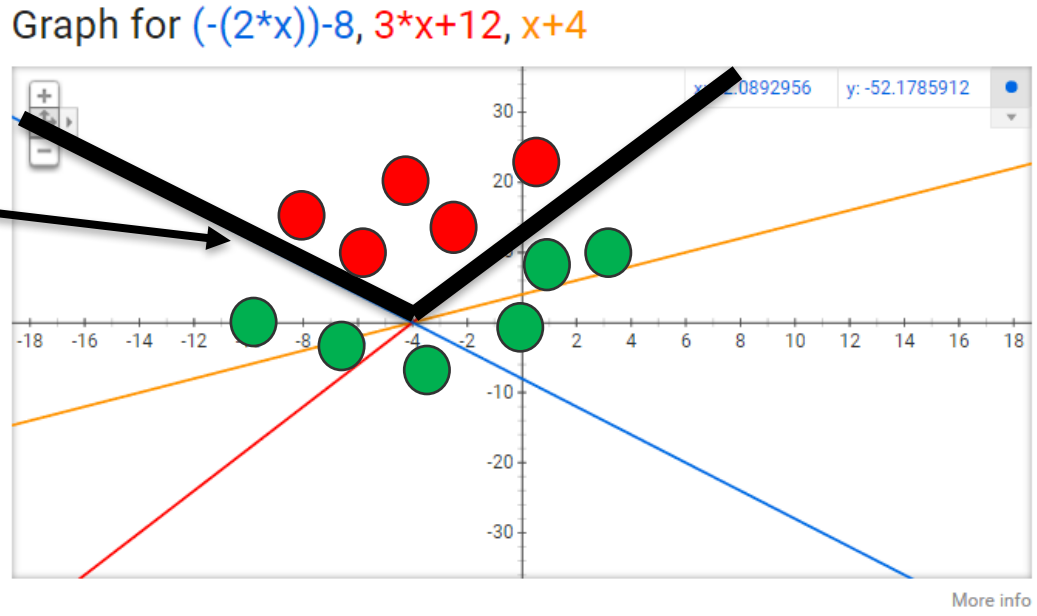
Building Blocks – ReLU



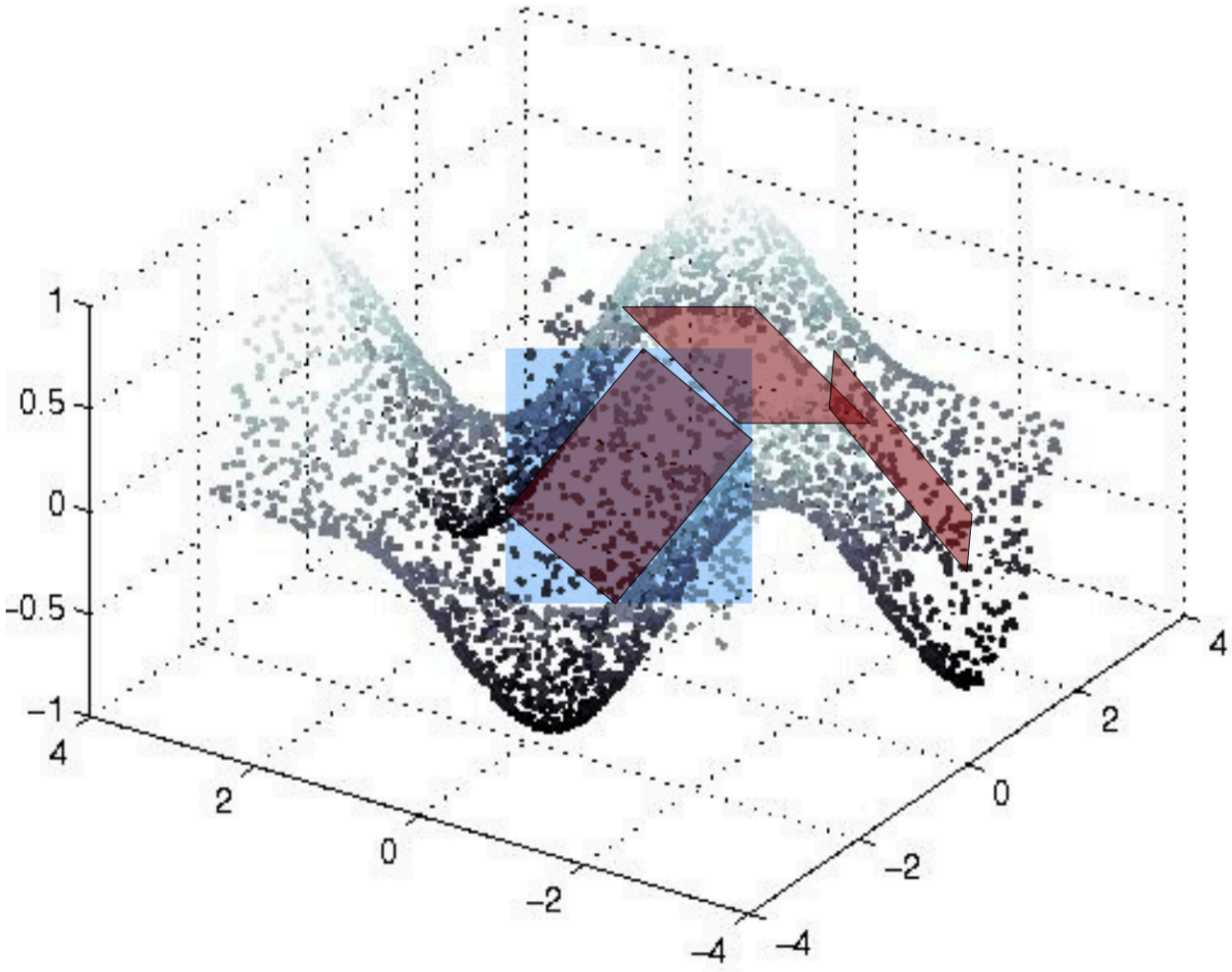


Building Blocks – ReLU

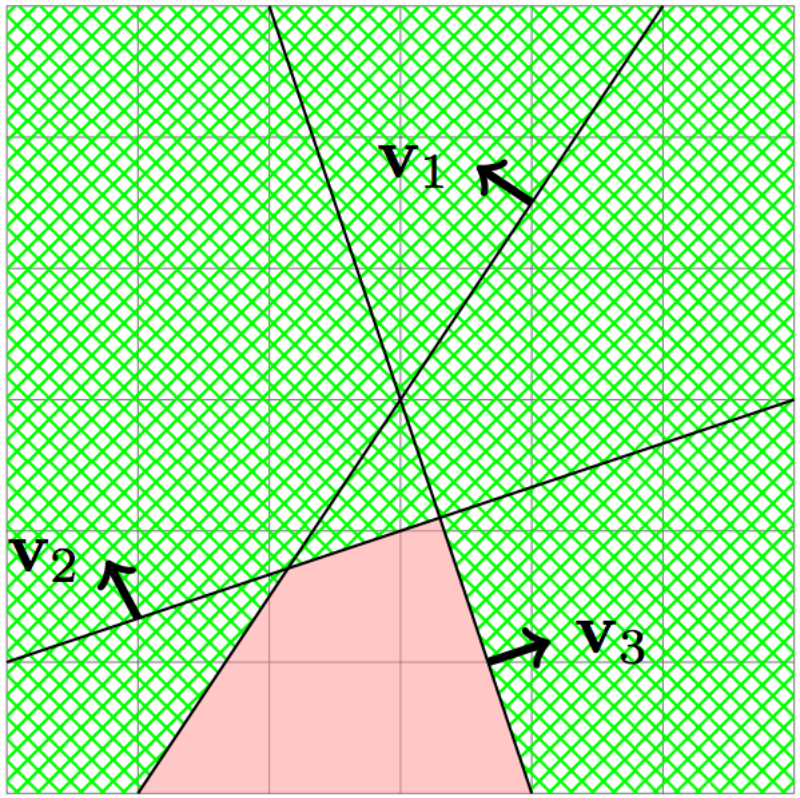




Building Blocks – ReLU



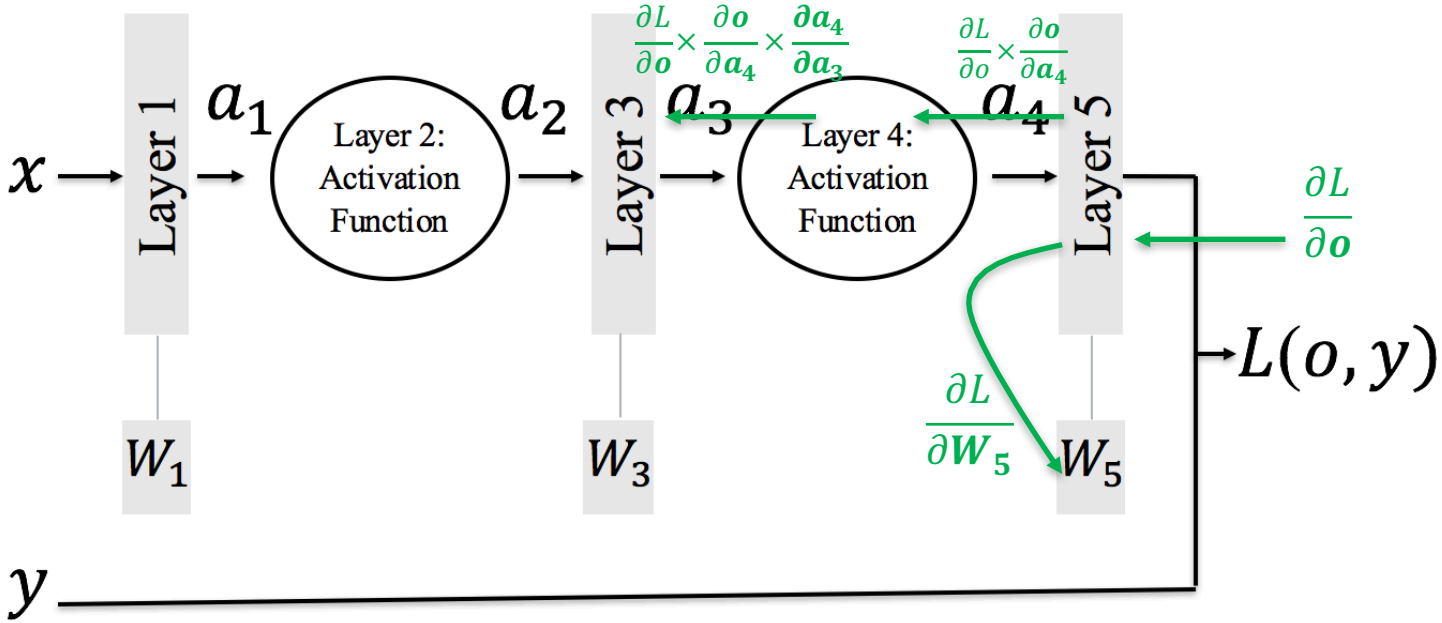
Building Blocks – ReLU



- 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/Exploding Gradients



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

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.

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`

Torch7 vs PyTorch

Torch

(-) Lua

(-) No autograd

(+) More stable

(+) Lots of existing code

(0) Fast

PyTorch

(+) Python

(+) Autograd

(-) Newer, still changing

(-) Less existing code

(0) Fast

Conclusion: Probably use PyTorch for new projects

Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static**)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic**)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

New graph each iteration

Static vs Dynamic Graphs

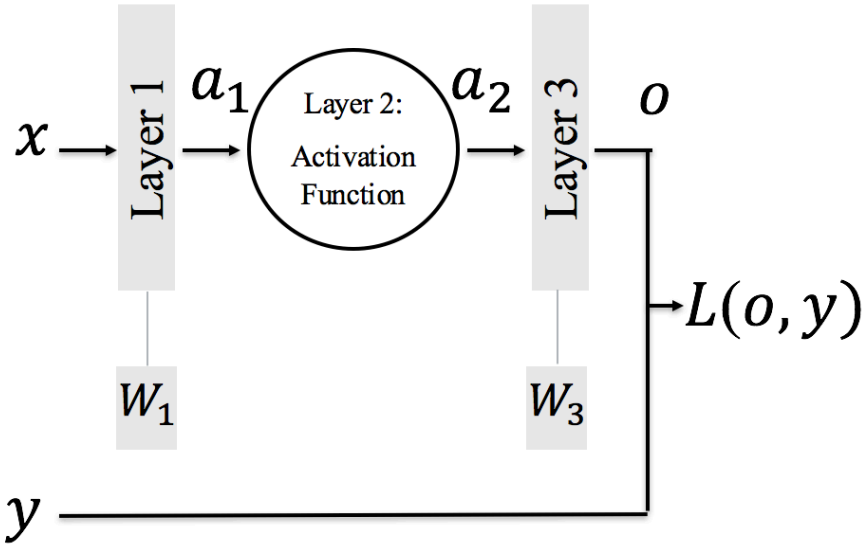
- With static graphs, framework can optimize the graph for you before it runs (Theano, TensorFlow e.g. Fused ReLU, so no extra output variable in memory, backprop also combined)
- Static Graphs: Once graph is built, can serialize it and run it without the code that built the graph
- Dynamic Graphs: Graph building and execution are intertwined, so always need to keep code around
- Dynamic Graphs: Backprop is exactly based on the forward code path, more intuitive, do not need to think in a functional programming way

PyTorch


Andrej Karpathy
@karpathy
I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.
12:26 AM - May 27, 2017
1,409
393 people are talking about this
Andrej Karpathy @karpathy 27 May
I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.
Sean Robertson
@sprobertson
Talk to your doctor to find out if PyTorch is right for you.
12:33 AM - May 27, 2017
130
See Sean Robertson's other Tweets


Andrej Karpathy @karpathy 27 May
I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.
Adam Will
@adam_will_do_it
PyTorch gave me so much life that my skin got cleared, my grades are up, my bills are paid and my crops are watered.
12:29 AM - May 27, 2017
23
See Adam Will 's other Tweets
Mariya
@thinkmariya
So have I! But my hair is also shiner and I've lost weight. @PyTorch for the win. twitter.com/karpathy/statu…
12:38 AM - May 27, 2017
23
See Mariya's other Tweets

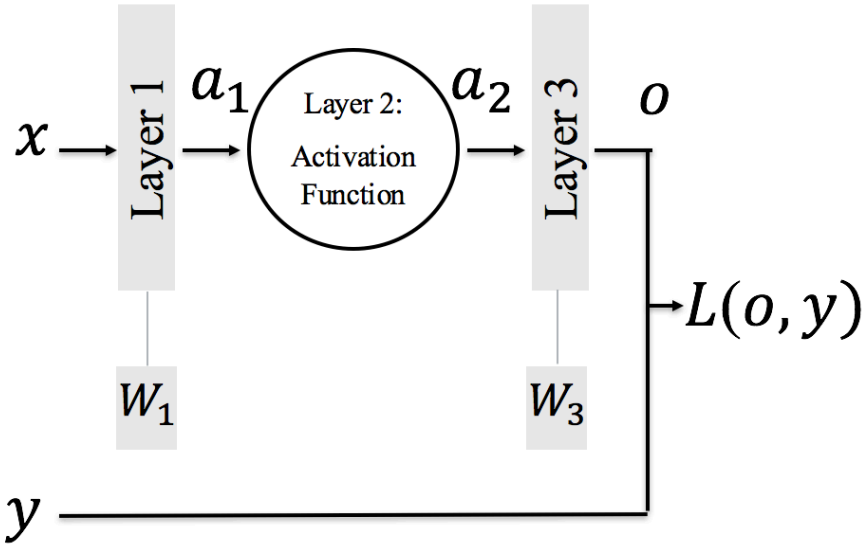
Multiple Layers – Feed Forward – In Torch7



- 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)`

Advice: Understand the engineering behind at least ONE framework in detail

Theano: A Python framework for fast computation of mathematical expressions

(The Theano Development Team)*

Orhan Firat,[1,23] Mathieu Germain,[1] Xavier Glorot,[1,18] Ian Goodfellow,[1,24] Matt Graham,[25] Caglar Gulcehre,[1]
Philippe Hamel,[1] Iban Harlouchet,[1] Jean-Philippe Heng,[1,26] Balázs Hidasi,[27] Sina Honari,[1] Arjun Jain,[28]
Sébastien Jean,[1,11] Kai Jia,[29] Mikhail Korobov,[30] Vivek Kulkarni,[6] Alex Lamb,[1] Pascal Lamblin,[1] Eric Larsen,[1,31]
César Laurent,[1] Sean Lee,[17] Simon Lefrancois,[1] Simon Lemieux,[1] Nicholas Léonard,[1] Zhouhan Lin,[1]
Jesse A. Livezey,[32] Cory Lorenz,[33] Jeremiah Lowin, Qianli Ma,[34] Pierre-Antoine Manzagol,[1] Olivier Mastropietro,[1]

[26]*Meiji University, Tokyo, Japan*
[27]*Gravity R&D*
[28]*Indian Institute of Technology, Bombay, India*
[29]*Megvii Technology Inc.*
[30]*ScrapingHub Inc.*
[31]*CIRRELT and Département d'informatique et recherche opérationnelle, Université de Montréal, QC, Canada*

https://github.com/torch/cunn/blob/master/lib/THCUNN/VolumetricConvolution.cu

```
6   // Kernel for fast unfold+copy
7   // Borrowed from Theano
8   // Authors: Arjun Jain, Frédéric Bastien, Jan Schlüter, Nicolas Ballas
9   template <typename Dtype>
10  __global__ void im3d2col_kernel(const int n, const Dtype* data_im,
11                                  const int height, const int width, const int depth,
```

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

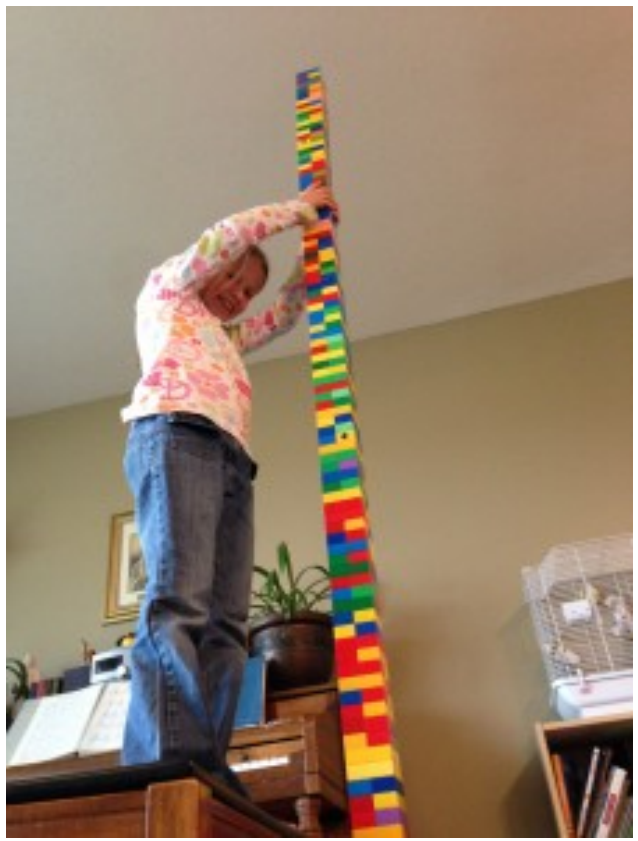
```
local Module = torch.class('nn.Module')

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
```

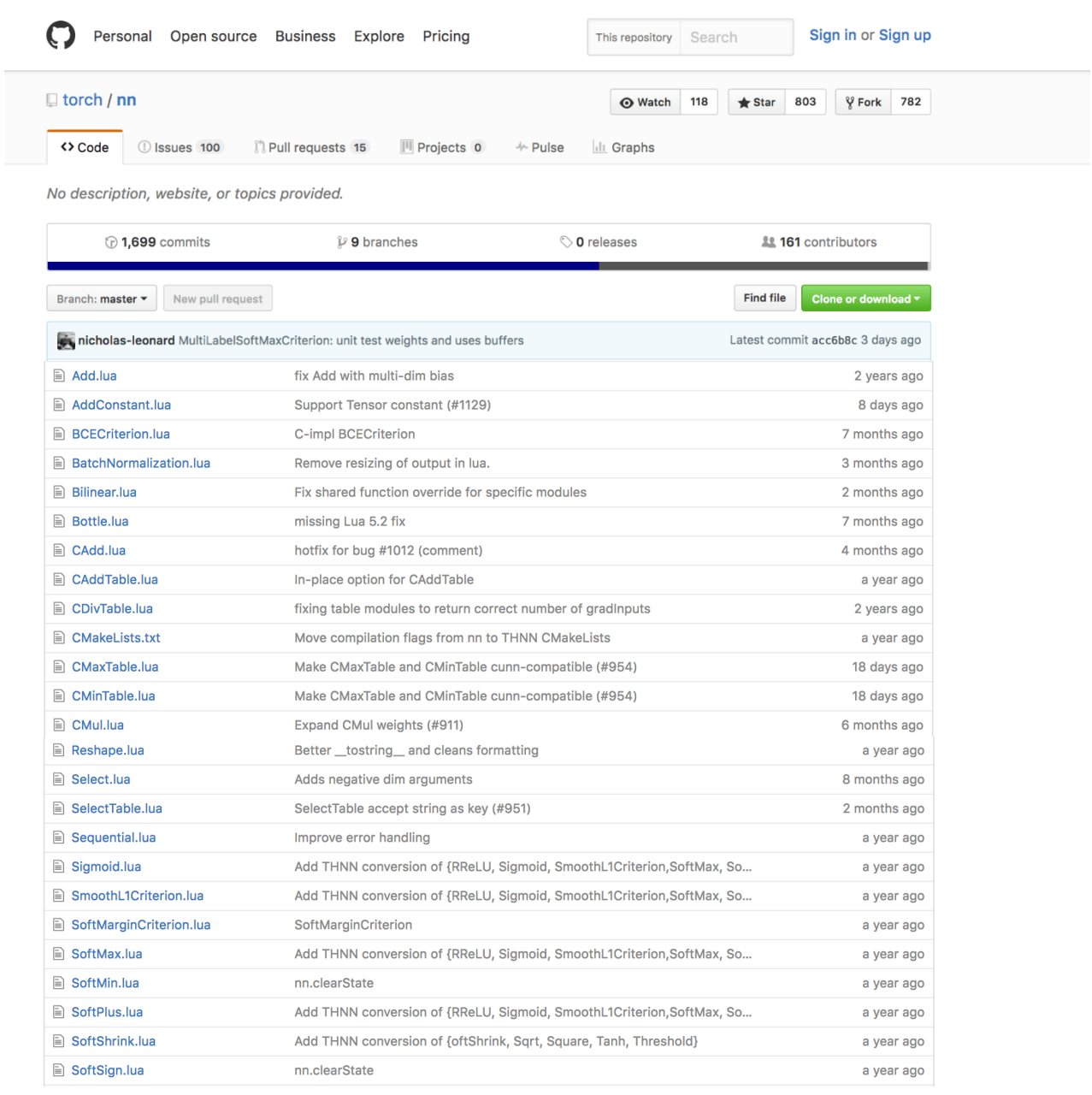
- If the module has parameters, this will update these parameters, according to the accumulation of the gradients with respect to these parameters, accumulated through backward() calls.
- The update is basically:

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

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

Thank you!