STA 221: LECTURE 13

KRISHNA BALASUBRAMANIAN

(UNIVERSITY OF CALIFORNIA, DAVIS)

Intro to PyTorch

PyTorch

PyTorch provides a framework for computing gradients efficiently (so it is easy to do gradient descent!)

Neural networks classifiers/regressors could be implemented efficiently using PyTorch.

Basics in `Lecture13.ipynb`

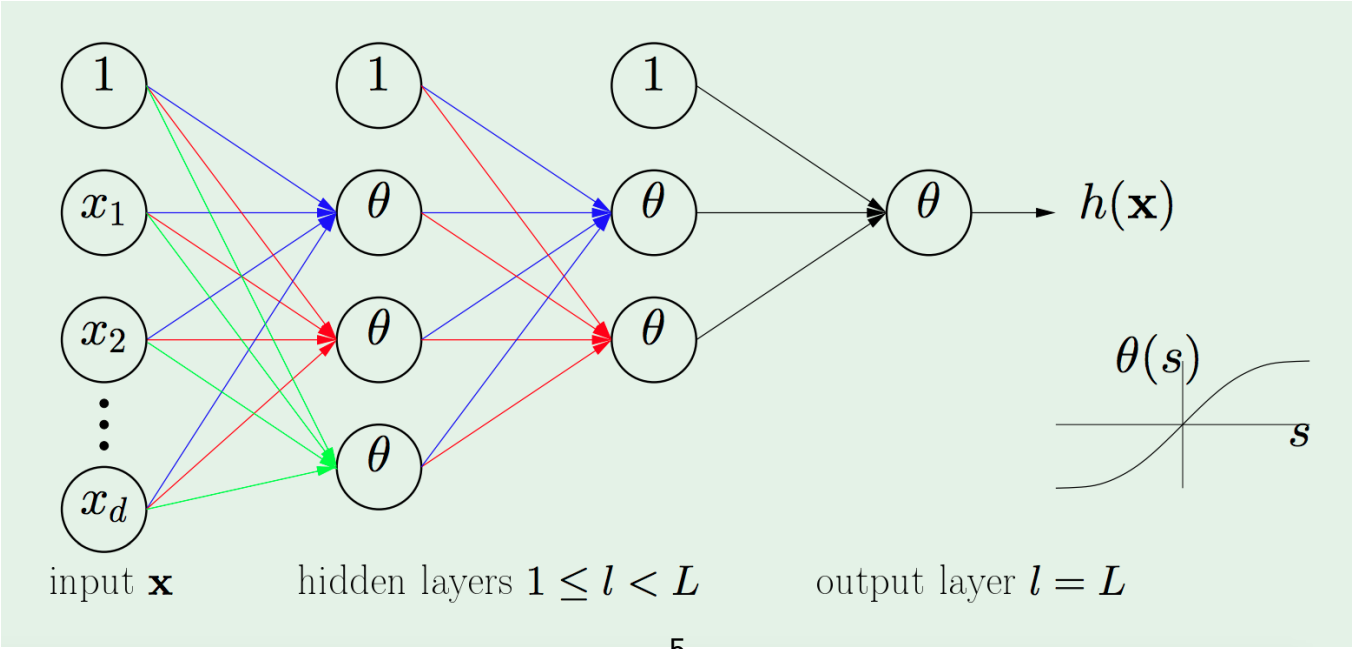
Neural Networks

Neural Network

Input layer: d neurons (input features)
Neurons from layer 1 to L: Linear combination of previous layers + activation function

$$\theta(\boldsymbol{w}^T\boldsymbol{x}), \quad \theta(x) = \text{sign}(x)$$

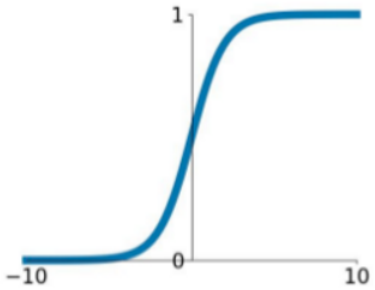
θ: activation function
Final layer: one neuron $\Rightarrow$ prediction by $\text{sign}(h(\boldsymbol{x}))$



Activation Function

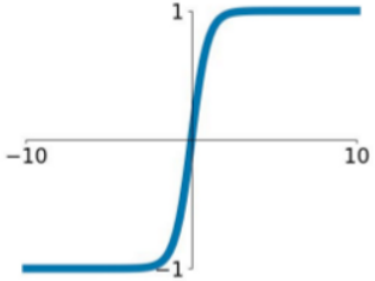
Sigmoid

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



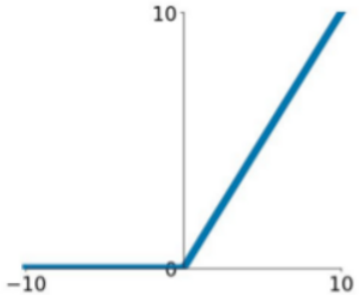
tanh

$$\tanh(x)$$



ReLU

$$\max(0, x)$$



Formal Definitions

$$w_{ij}^{(l)} \quad \begin{cases} 1 \leq l \leq L & \text{: layers} \\ 0 \leq i \leq d^{(l-1)} & \text{: inputs} \\ 1 \leq j \leq d^{(l)} & \text{: outputs} \end{cases}$$

j-th neuron in the l-th layer:

$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)})$$

Output:

$$h(\boldsymbol{x}) = x_1^{(L)}$$

Formal Definitions

$$w_{ij}^{(l)} \quad \begin{cases} 1 \leq l \leq L & \text{: layers} \\ 0 \leq i \leq d^{(l-1)} & \text{: inputs} \\ 1 \leq j \leq d^{(l)} & \text{: outputs} \end{cases}$$

j-th neuron in the l-th layer:

$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)})$$

Output:

$$h(\boldsymbol{x}) = x_1^{(L)}$$

Formal Definitions

$$w_{ij}^{(l)} \quad \begin{cases} 1 \leq l \leq L & \text{: layers} \\ 0 \leq i \leq d^{(l-1)} & \text{: inputs} \\ 1 \leq j \leq d^{(l)} & \text{: outputs} \end{cases}$$

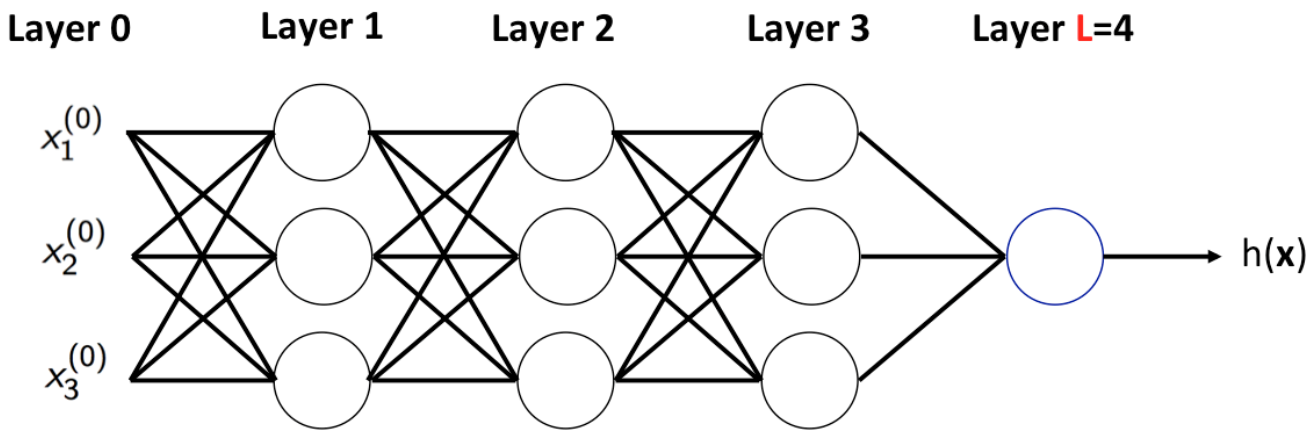
j-th neuron in the l-th layer:

$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)})$$

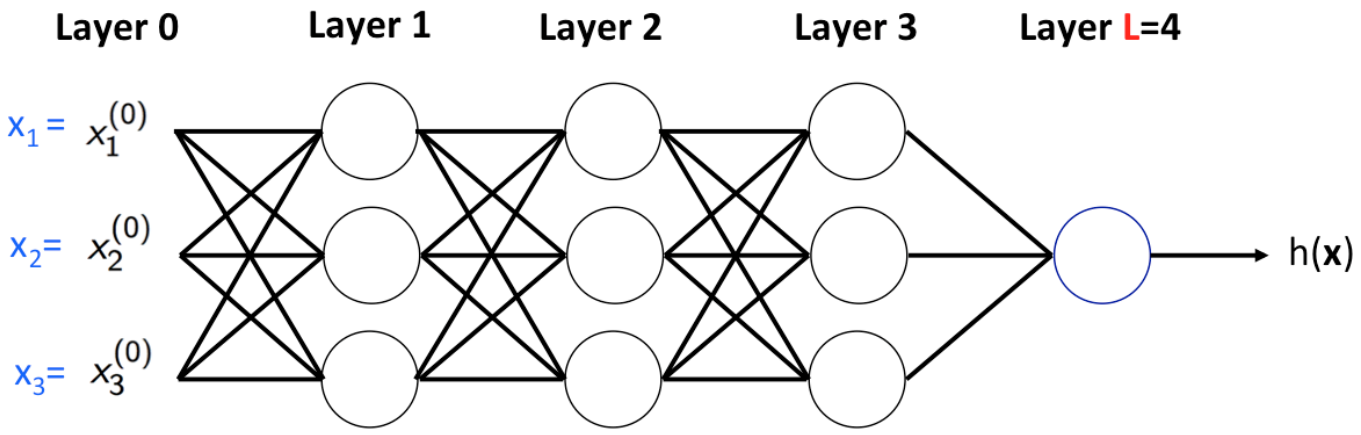
Output:

$$h(\boldsymbol{x}) = x_1^{(L)}$$

Forward propagation



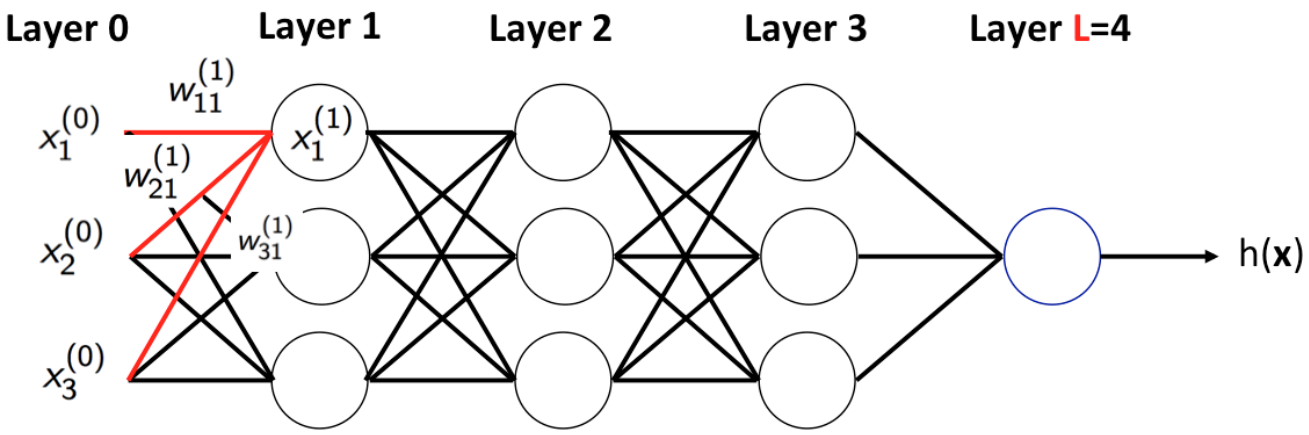
Forward propagation



features for one data point

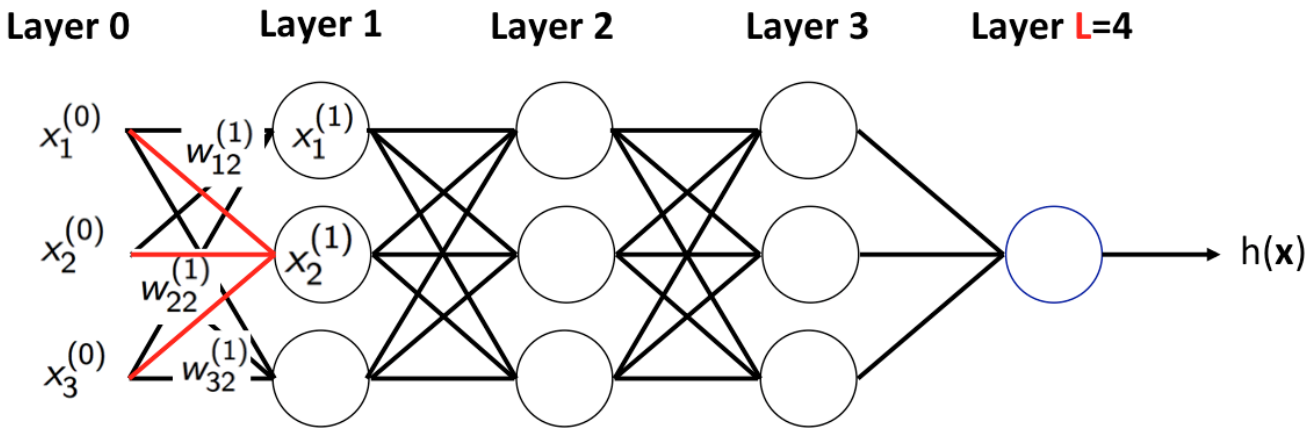
$\mathbf{x} = [x_1, x_2, x_3]$

Forward propagation



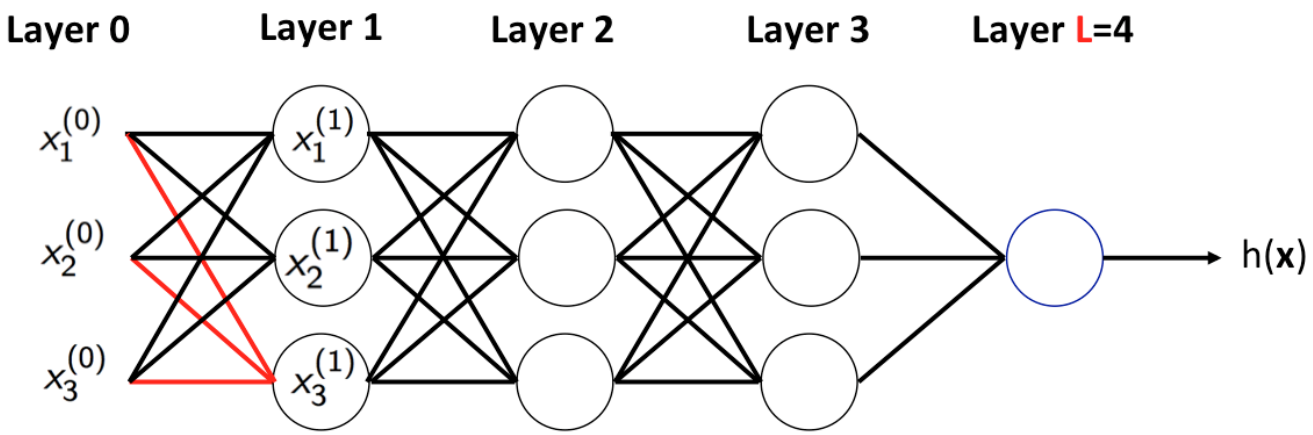
$$x_1^{(1)} = \theta\left(\sum_{i=1}^{3} w_{i1}^{(1)} x_i^{(0)}\right)$$

Forward propagation

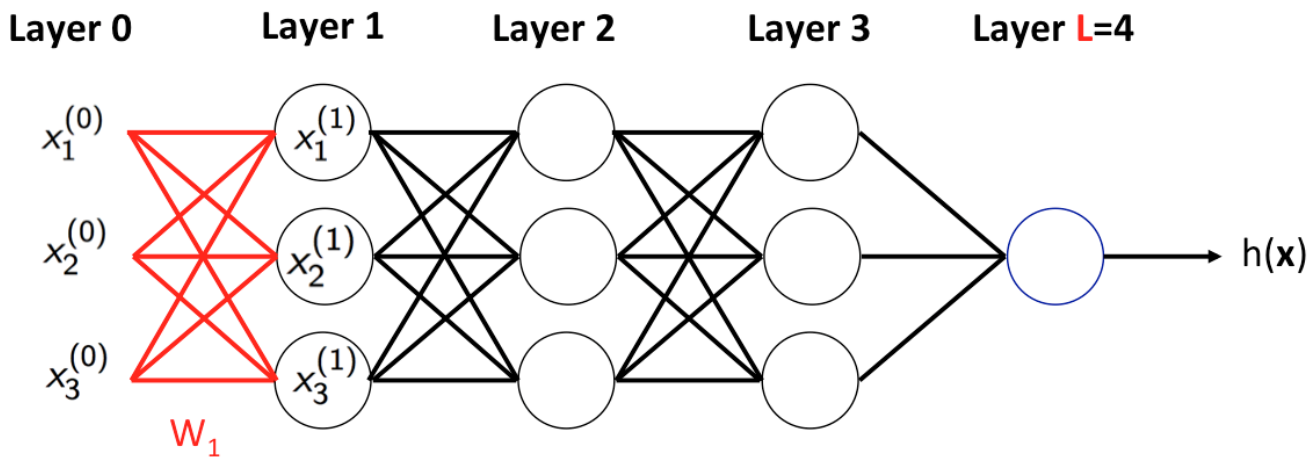


$$x_2^{(1)} = \theta\left(\sum_{i=1}^{3} w_{i2}^{(1)} x_i^{(0)}\right)$$

Forward propagation

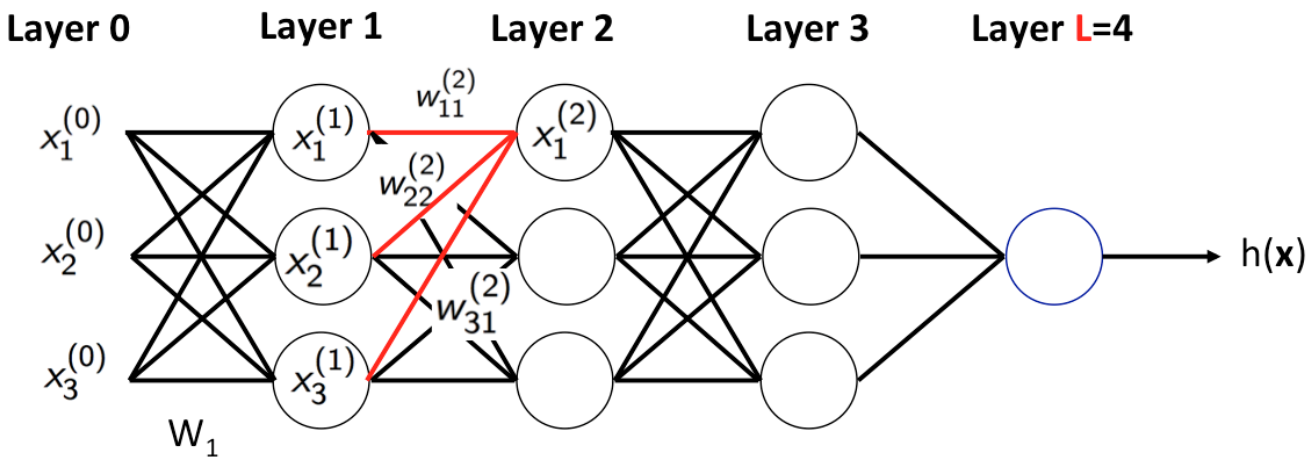


Forward propagation



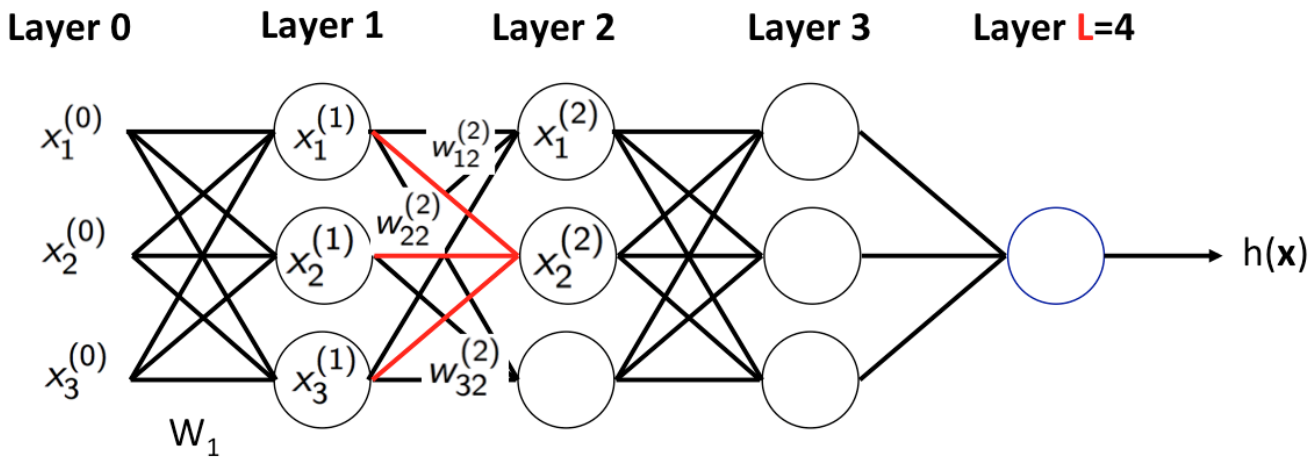
$$\boldsymbol{x}^{(1)} = \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} = \theta\left(\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} & w_{33}^{(1)} \end{bmatrix} \times \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \\ x_3^{(0)} \end{bmatrix} \right) = \theta(W_1 \boldsymbol{x}^{(0)})$$

Forward propagation



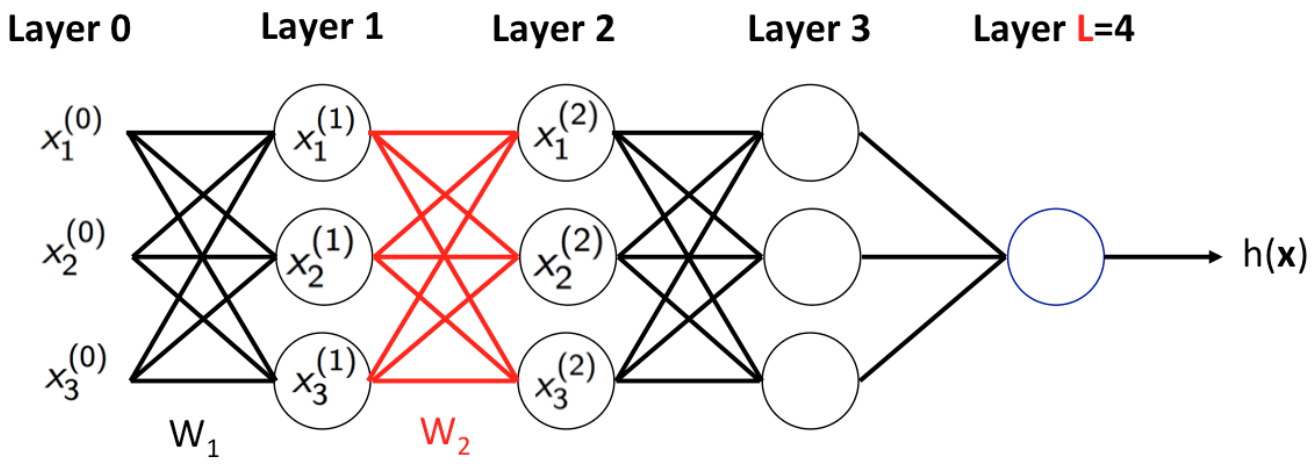
$$x_1^{(2)} = \theta\left(\sum_{i=1}^{3} w_{i1}^{(2)} x_i^{(1)}\right)$$

Forward propagation



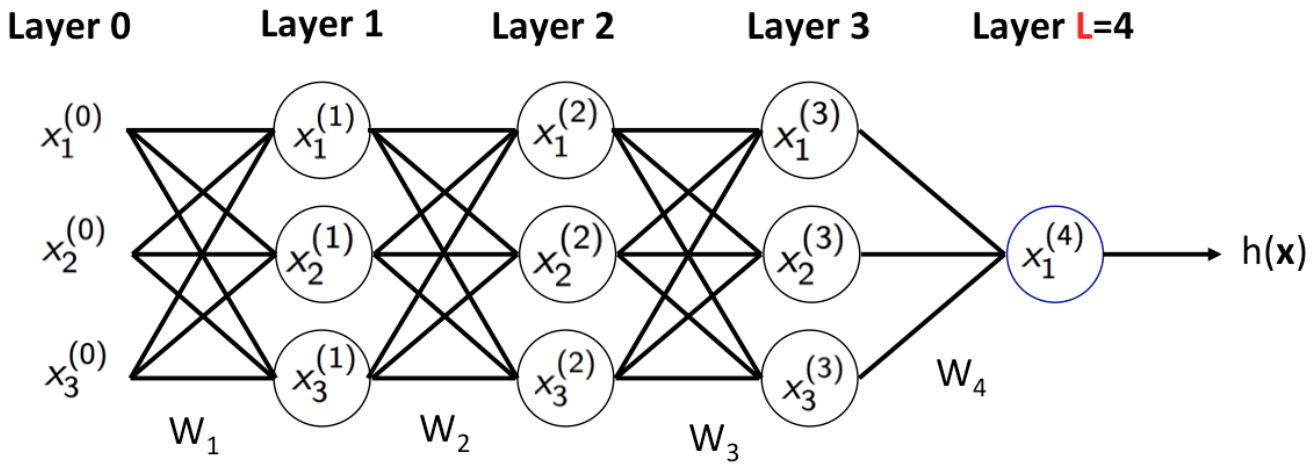
$$x_2^{(2)} = \theta\left(\sum_{i=1}^{3} w_{i2}^{(2)} x_i^{(1)}\right)$$

Forward propagation



$$\boldsymbol{x}^{(2)} = \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{bmatrix} = \theta\left(\begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} & w_{31}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} & w_{32}^{(2)} \\ w_{13}^{(2)} & w_{23}^{(2)} & w_{33}^{(2)} \end{bmatrix} \times \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} \right) = \theta(W_2 \boldsymbol{x}^{(1)})$$

Forward propagation



$$h(\boldsymbol{x}) = x_1^{(4)} = \theta(W_4 \boldsymbol{x}^{(3)}) = \theta(W_4 \theta(W_3 \boldsymbol{x}^{(2)}))$$
$$= \cdots = \theta(W_4 \theta(W_3 \theta(W_2 \theta(W_1 \boldsymbol{x}))))$$

Stochastic Gradient Descent

All the weights $W = \{W_1, \cdots, W_L\}$ determine $h(\boldsymbol{x})$

Neural network training: Solving

$$\min_W \frac{1}{N} \sum_{n=1}^{N} e(h(\boldsymbol{x}_n), y_n)$$

by Stochastic Gradient Descent (SGD)

Each iteration of SGD:

Pick a sample $(\boldsymbol{x}_n, y_n)$ randomly
Update $W \leftarrow W - \eta \times \nabla e(h(\boldsymbol{x}_n), y_n)$

Other algorithms: Adam, Adagrad, Momentum SGD

(main computation is still $\nabla e(h(\boldsymbol{x}_n), y_n)$)

Stochastic Gradient Descent

All the weights $W = \{W_1, \cdots, W_L\}$ determine $h(\boldsymbol{x})$

Neural network training: Solving

$$\min_W \frac{1}{N} \sum_{n=1}^{N} e(h(\boldsymbol{x}_n), y_n)$$

by Stochastic Gradient Descent (SGD)

Each iteration of SGD:

Pick a sample $(\boldsymbol{x}_n, y_n)$ randomly

Update $W \leftarrow W - \eta \times \nabla e(h(\boldsymbol{x}_n), y_n)$

Other algorithms: Adam, Adagrad, Momentum SGD

(main computation is still $\nabla e(h(\boldsymbol{x}_n), y_n)$)

Stochastic Gradient Descent

All the weights $W = \{W_1, \cdots, W_L\}$ determine $h(\boldsymbol{x})$

Neural network training: Solving

$$\min_W \frac{1}{N} \sum_{n=1}^{N} e(h(\boldsymbol{x}_n), y_n)$$

by Stochastic Gradient Descent (SGD)

Each iteration of SGD:

Pick a sample $(\boldsymbol{x}_n, y_n)$ randomly
Update $W \leftarrow W - \eta \times \nabla e(h(\boldsymbol{x}_n), y_n)$

Other algorithms: Adam, Adagrad, Momentum SGD

(main computation is still $\nabla e(h(\boldsymbol{x}_n), y_n)$)

Stochastic Gradient Descent

All the weights $W = \{W_1, \cdots, W_L\}$ determine $h(\boldsymbol{x})$

Neural network training: Solving

$$\min_W \frac{1}{N} \sum_{n=1}^{N} e(h(\boldsymbol{x}_n), y_n)$$

by Stochastic Gradient Descent (SGD)

Each iteration of SGD:

Pick a sample $(\boldsymbol{x}_n, y_n)$ randomly
Update $W \leftarrow W - \eta \times \nabla e(h(\boldsymbol{x}_n), y_n)$

Other algorithms: Adam, Adagrad, Momentum SGD

(main computation is still $\nabla e(h(\boldsymbol{x}_n), y_n)$)

How to compute gradient?

Error on example $(\boldsymbol{x}_n, y_n)$ is

$$e(h(\boldsymbol{x}_n), y_n) = e(W)$$

To implement SGD, we need the gradient

$$\nabla e(W): \quad \{\frac{\partial e(W)}{\partial w_{ij}^{(l)}}\} \text{ for all } i, j, l$$

How to compute gradient?

Error on example $(\boldsymbol{x}_n, y_n)$ is

$$e(h(\boldsymbol{x}_n), y_n) = e(W)$$

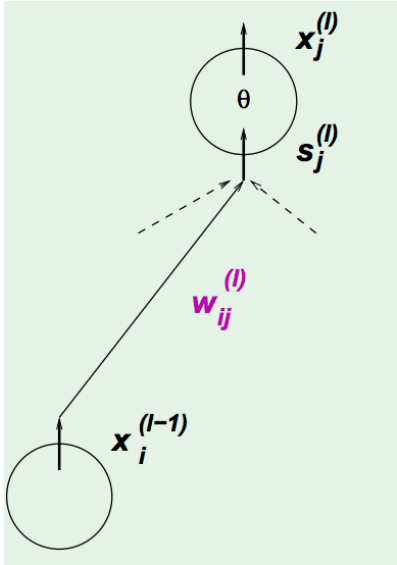
To implement SGD, we need the gradient

$$\nabla e(W): \quad \{\frac{\partial e(W)}{\partial w_{ij}^{(l)}}\} \text{ for all } i, j, l$$

Computing Gradient $\frac{\partial e(W)}{\partial w_{ij}^{(l)}}$

Use chain rule:

$$\frac{\partial e(W)}{\partial w_{ij}^{(l)}} = \frac{\partial e(W)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$



$s_j^{(l)} = \sum_{i=1}^{d} x_i^{(l-1)} w_{ij}^{(l)}$

We have $\frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$

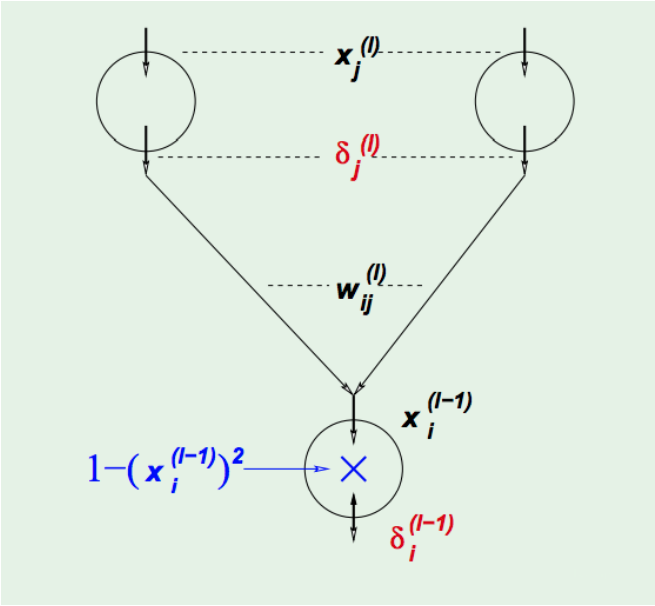
Computing Gradient $\frac{\partial e(W)}{\partial w_{ij}^{(l)}}$

Define $\delta_j^{(l)} := \frac{\partial e(W)}{\partial s_j^{(l)}}$

Compute by layer-by-layer:

$$\begin{aligned}
\delta_i^{(l-1)} &= \frac{\partial e(W)}{\partial s_i^{(l-1)}} \\
&= \sum_{j=1}^{d} \frac{\partial e(W)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial s_i^{l-1}} \\
&= \sum_{j=1}^{d} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(s_i^{(l-1)}),
\end{aligned}$$

where $\theta'(s) = 1 - \theta^2(s)$ for tanh and $\delta_i^{(l-1)} = (1 - (x_i^{(l-1)})^2) \sum_{j=1}^{d} w_{ij}^{(l)} \delta_j^{(l)}$



Final layer

(Assume square loss)

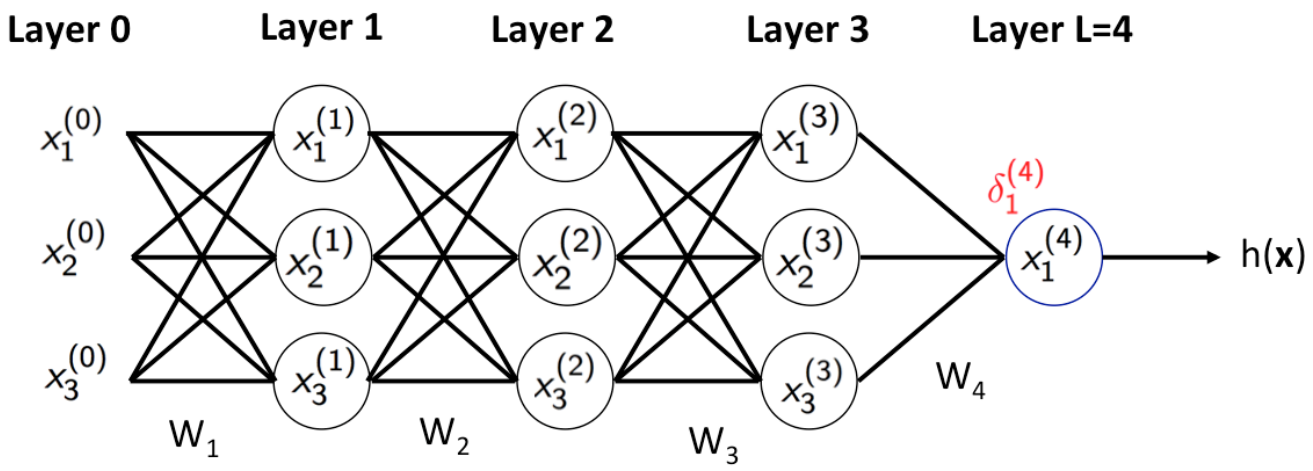
$$e(W) = (x_1^{(L)} - y_n)^2$$

$$x_1^{(L)} = \theta(s_1^{(L)})$$

So,

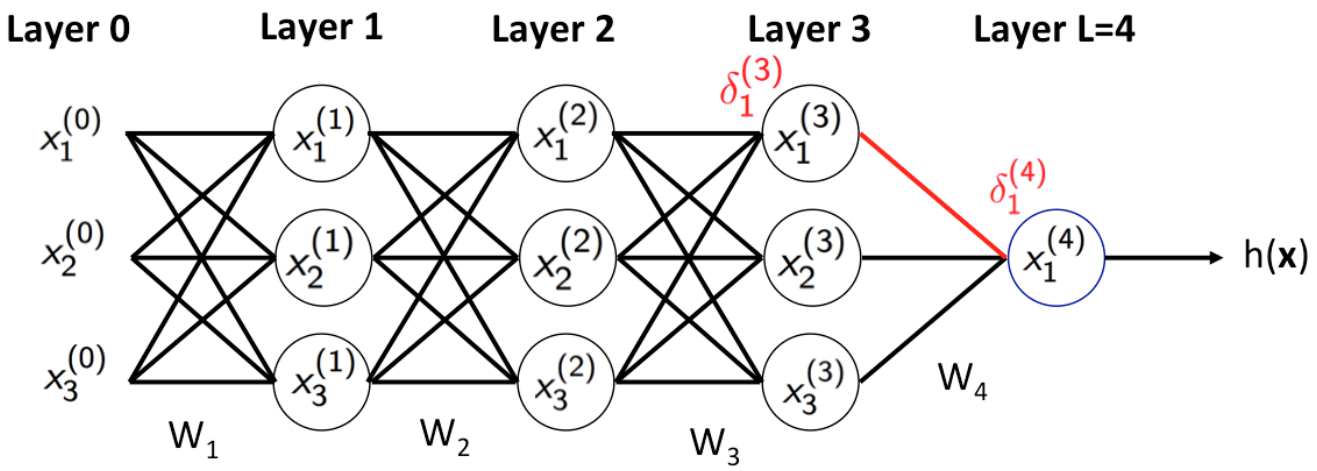
$$\begin{aligned}
\delta_1^{(L)} &= \frac{\partial e(W)}{\partial s_1^{(L)}} \\
&= \frac{\partial e(W)}{\partial x_1^{(L)}} \times \frac{\partial x_1^{(L)}}{s_1^{(L)}} \\
&= 2(x_1^{(L)} - y_n) \times \theta'(s_1^{(L)})
\end{aligned}$$

Backward propagation



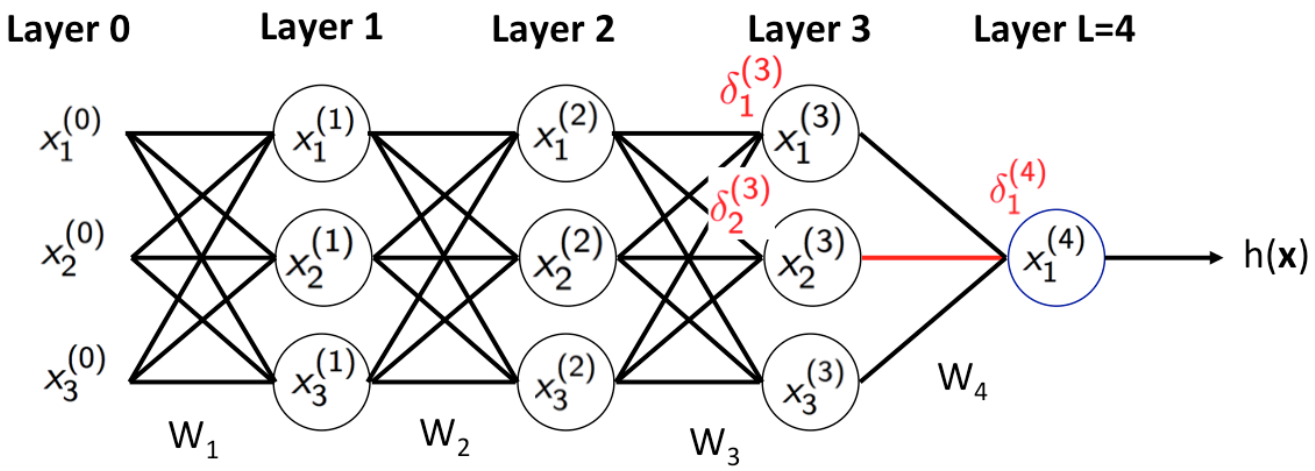
$$\delta_1^{(4)} = 2(x_1^{(4)} - y_n) \times (1 - (x_1^{(4)})^2)$$

Backward propagation



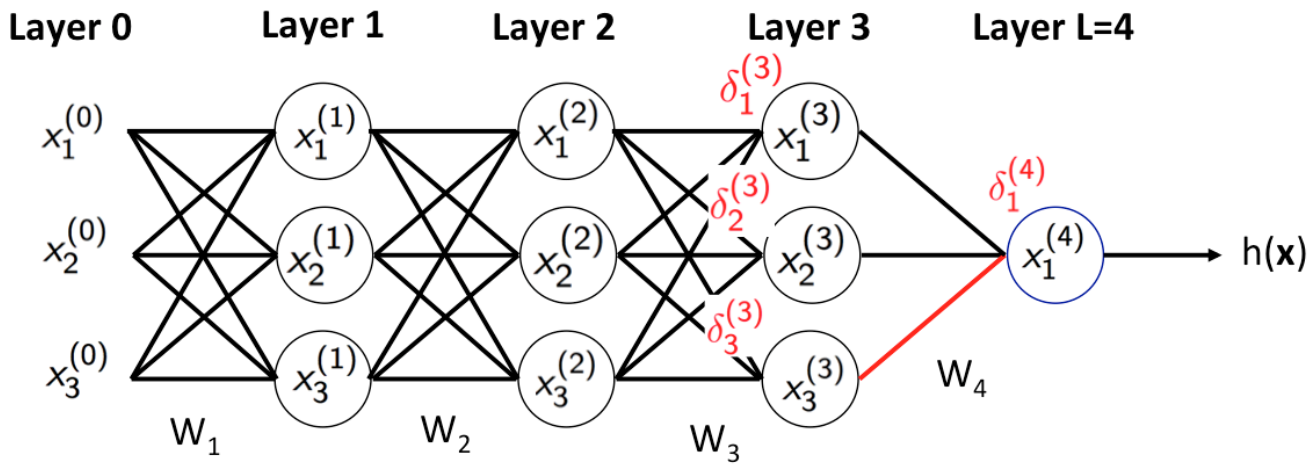
$$\delta_1^{(3)} = (1 - (x_1^{(3)})^2) \times \delta_1^{(4)} \times w_{11}^{(4)}$$

Backward propagation



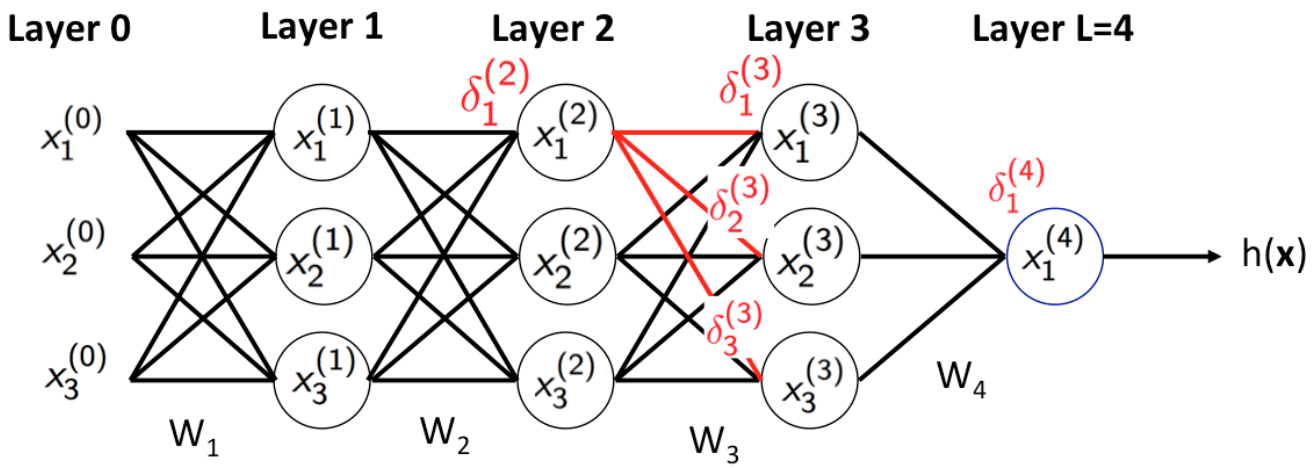
$$\delta_2^{(3)} = (1 - (x_2^{(3)})^2) \times \delta_1^{(4)} \times w_{21}^{(4)}$$

Backward propagation



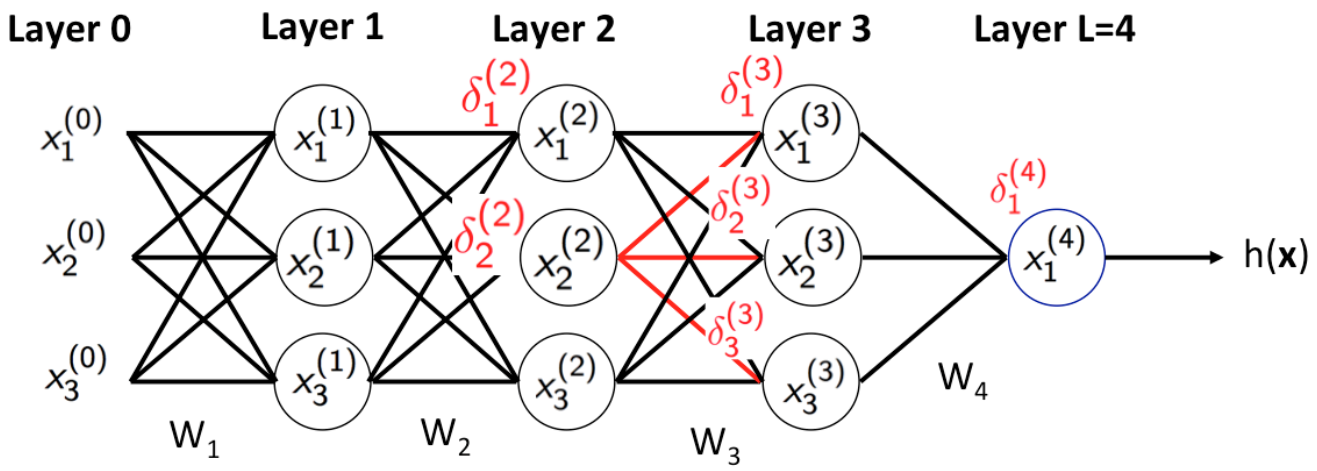
$$\delta_3^{(3)} = (1 - (x_3^{(3)})^2) \times \delta_1^{(4)} \times w_{31}^{(4)}$$

Backward propagation



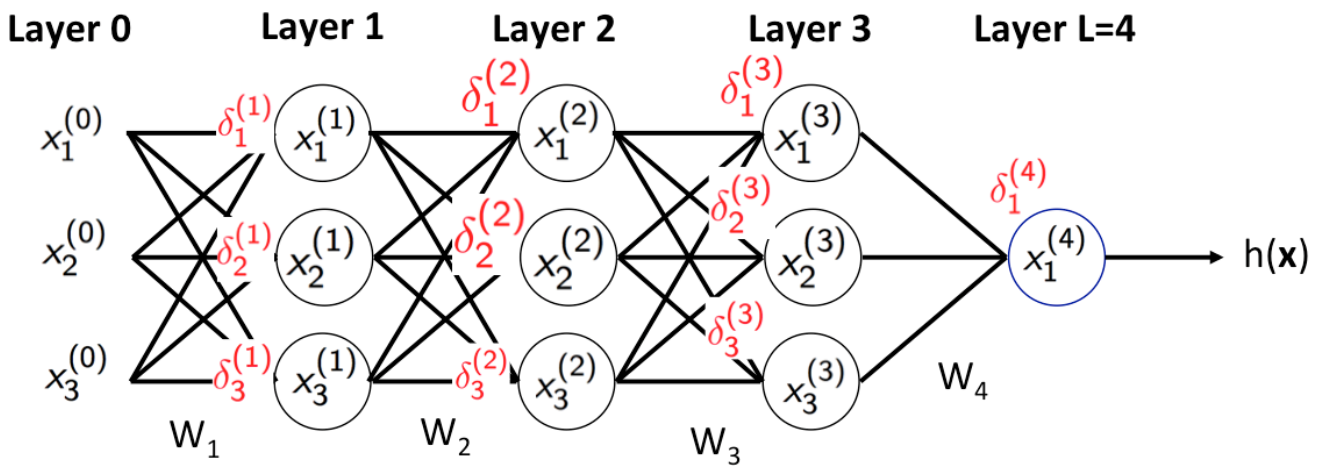
$$\delta_1^{(2)} = (1 - (x_1^{(2)})^2) \sum_{j=1}^{3} \delta_j^{(3)} w_{1j}^{(3)}$$

Backward propagation



$$\delta_2^{(2)} = (1 - (x_2^{(2)})^2) \sum_{j=1}^{3} \delta_j^{(3)} w_{2j}^{(3)}$$

Backward propagation



SGD for neural networks

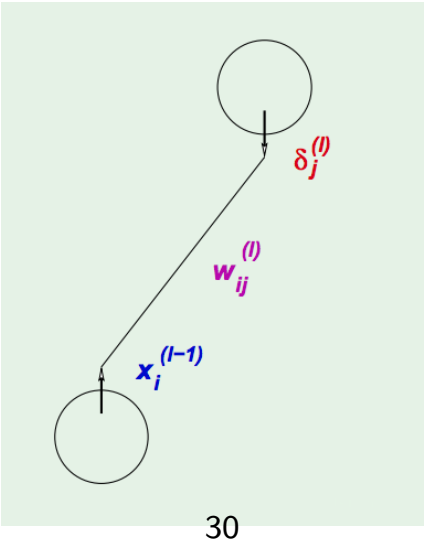
Initialize all weights $w_{ij}^{(l)}$ **at random**

For iter $= 0, 1, 2, \cdots$

Forward: Compute all $x_j^{(l)}$ from input to output

Backward: Compute all $\delta_j^{(l)}$ from output to input

Update all the weights $w_{ij}^{l} \leftarrow w_{ij}^{(l)} - \eta x_i^{(l-1)} \delta_j^{(l)}$



Backpropagation

Just an automatic way to apply chain rule to compute gradient

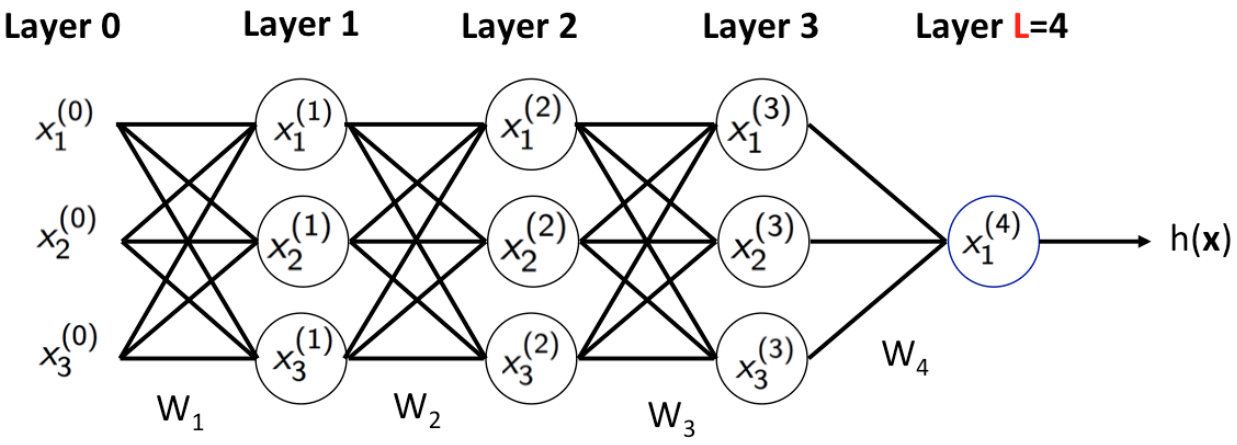
Auto-differentiation (AD) — as long as we define derivative for each basic function, we can use AD to compute any of their compositions

Implemented in most deep learning packages

(e.g., pytorch, tensorflow)

Convolutional Neural Network

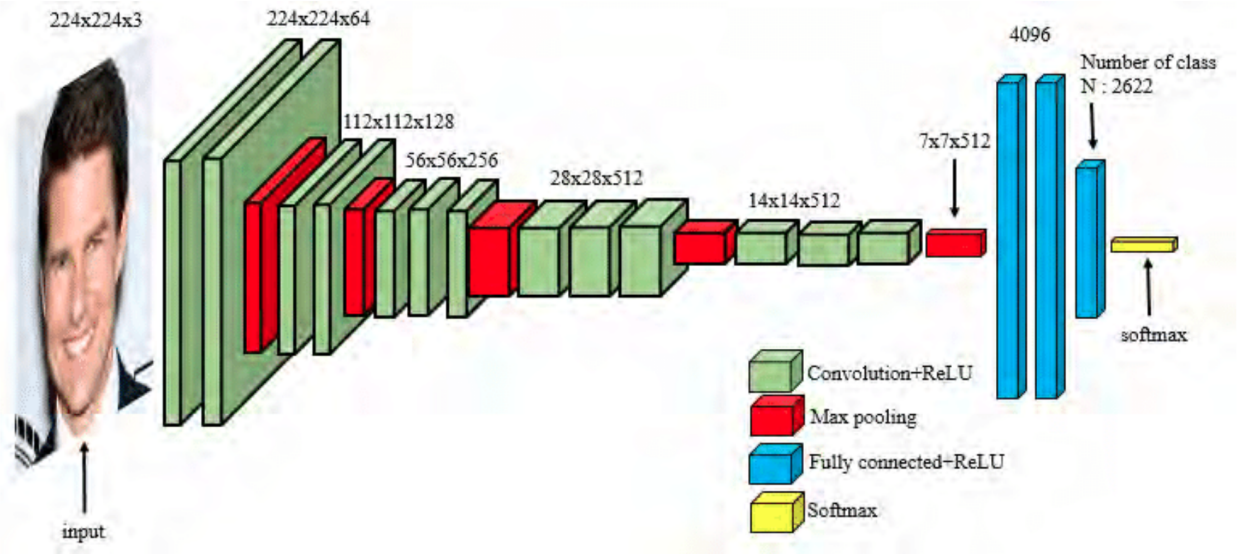
Neural Networks



$$h(\boldsymbol{x}) = x_1^{(4)} = \theta(W_4 \boldsymbol{x}^{(3)}) = \theta(W_4 \theta(W_3 \boldsymbol{x}^{(2)}))$$
$$= \cdots = \theta(W_4 \theta(W_3 \theta(W_2 \theta(W_1 \boldsymbol{x}))))$$

Fully connected networks $\Rightarrow$ doesn't work well for computer vision applications

Structure of VGG

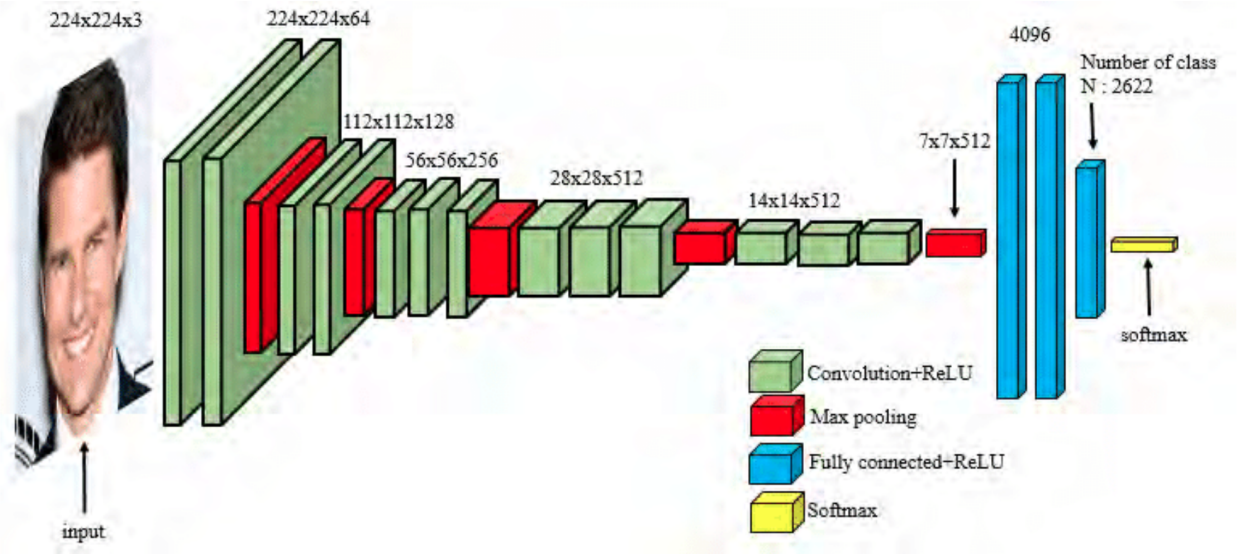


Two important layers:

- Convolution
- Pooling

The structure of CNN

Structure of VGG



Two important layers:

- Convolution
- Pooling

Convolutional Layer

Fully connected layers have too many parameters

$\Rightarrow$ poor performance

Example: VGG first layer

Input: $224 \times 224 \times 3$
Output: $224 \times 224 \times 64$
Number of parameters:
$(224 \times 224 \times 3) \times (224 \times 224 \times 64) =$ 483 billion

Convolutional layer:

Local connectivity
Parameter sharing

Convolutional Layer

Fully connected layers have too many parameters

$\Rightarrow$ poor performance

Example: VGG first layer

Input: $224 \times 224 \times 3$
Output: $224 \times 224 \times 64$
Number of parameters:
$(224 \times 224 \times 3) \times (224 \times 224 \times 64) =$ 483 billion

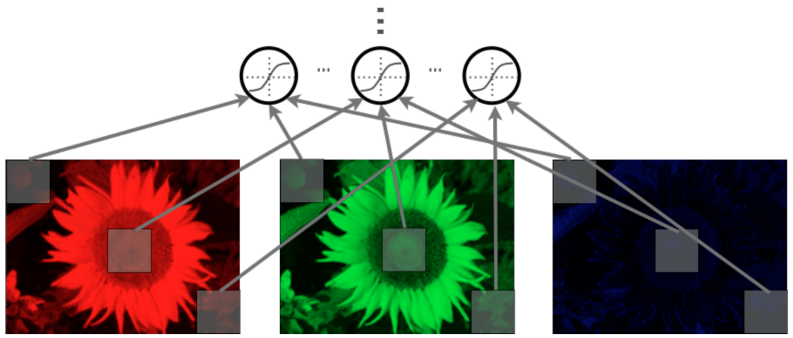
Convolutional layer:

Local connectivity
Parameter sharing

Local connectivity

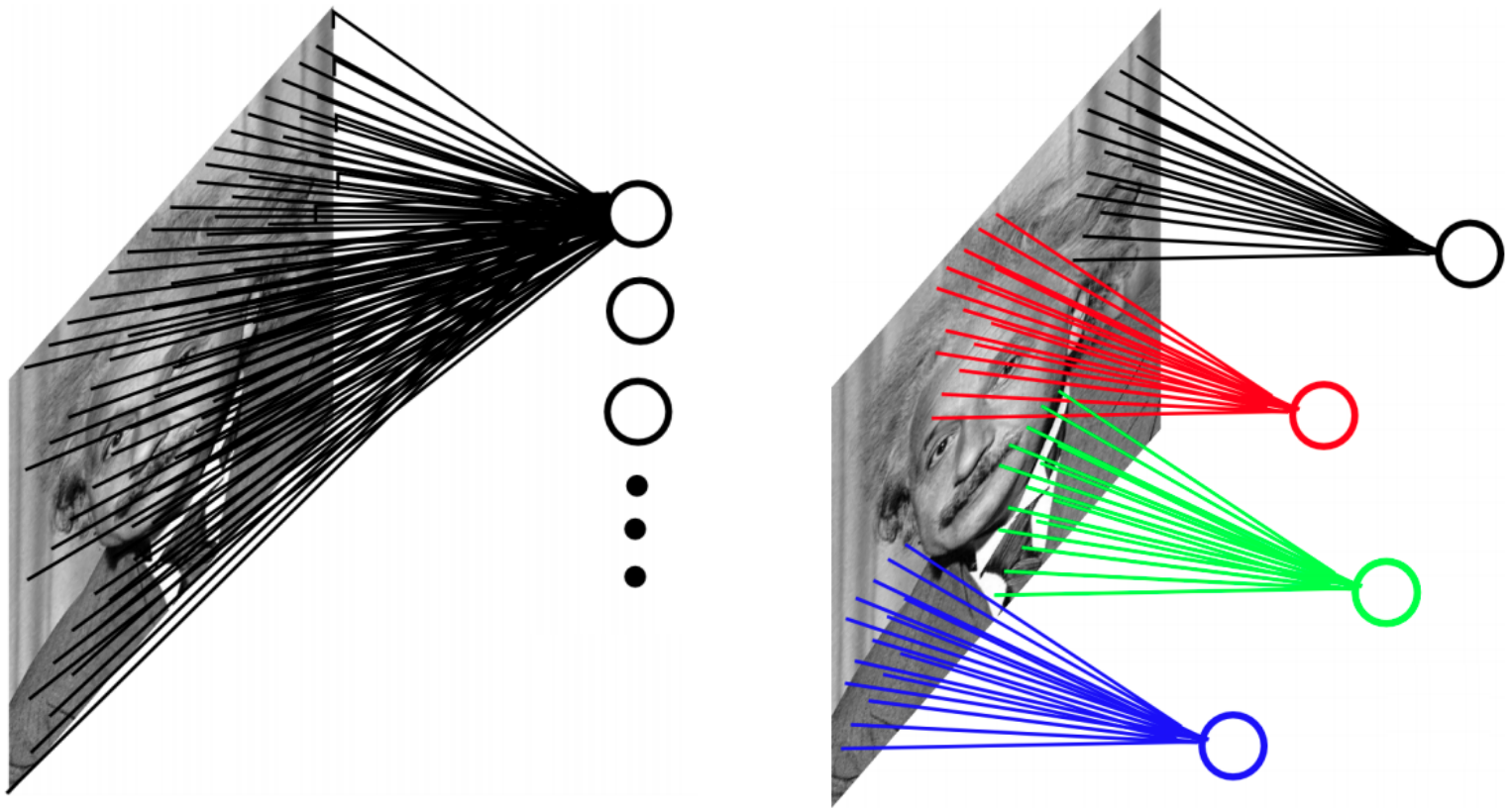
Each hidden unit is connected only to a sub-region of input

It is connected to all channels (R, G, B)



(Figure from Salakhutdinov 2017)

Local connectivity



(Figure from Salakhutdinov 2017)

Parameter Sharing

Making one reasonable assumption:

If one feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position (x_2, y_2)

Using the convolution operator

Convolution

The convolution of an image x with a kernel k is computed as

$$(x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{p,q}$$

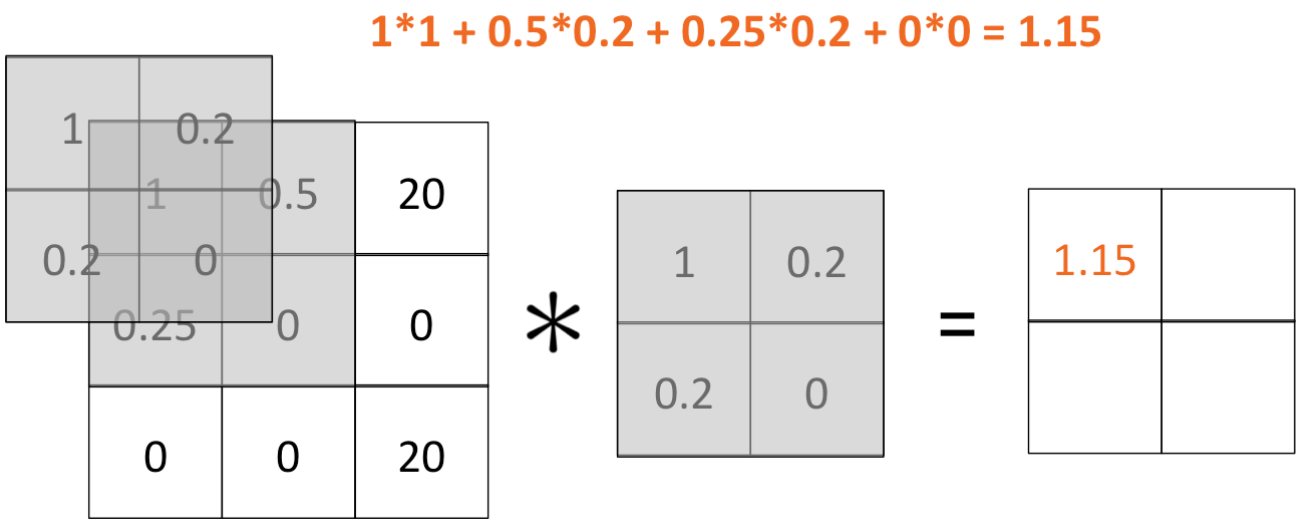
1	0.5	20
0.25	0	0
0	0	20

$*$

1	0.5
0.25	0

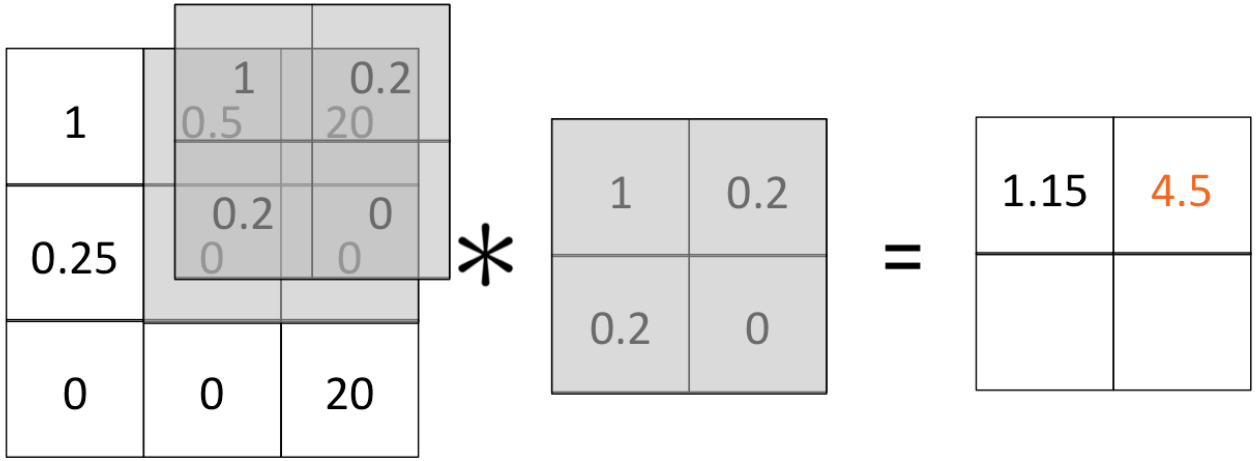
$=$

Convolution



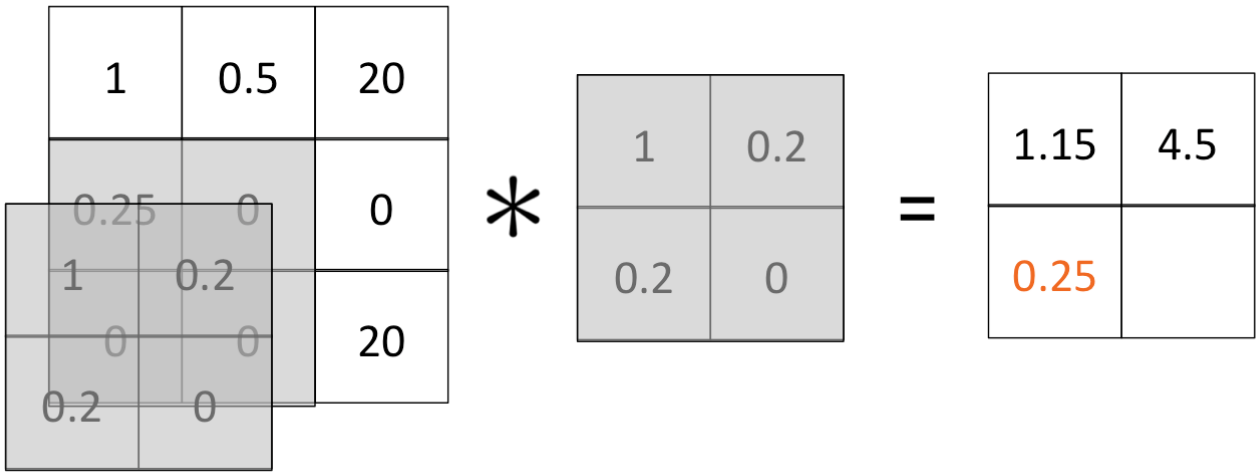
Convolution

0.5*1 + 20*0.2 + 0*0.2 + 0*0 = 4.5



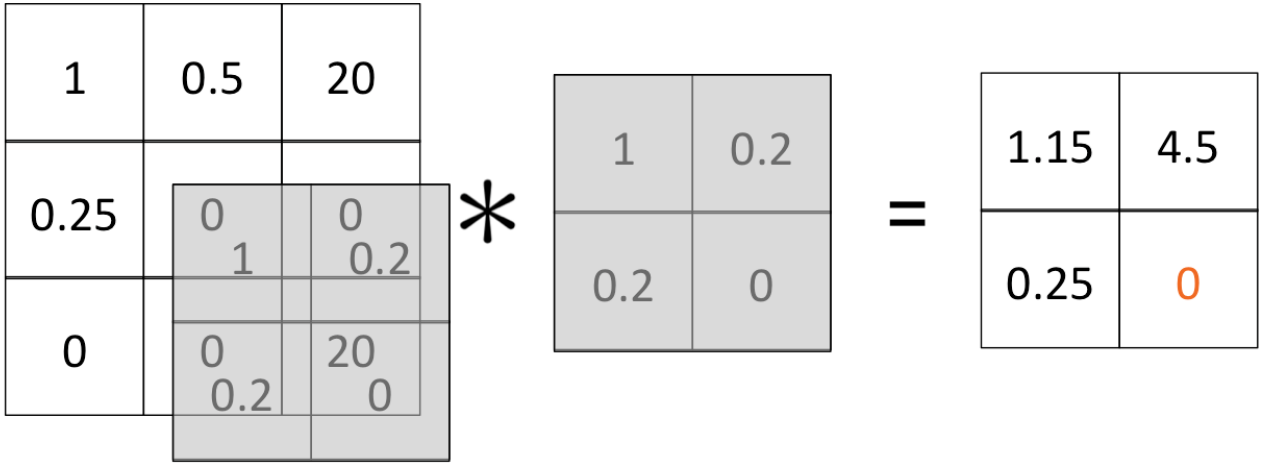
Convolution

0.25*1 + 0*0.2 + 0*0.2 + 0*0 = 0.25



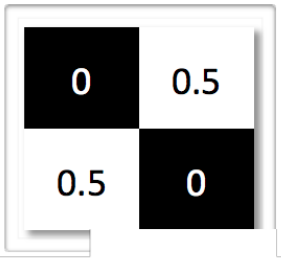
Convolution

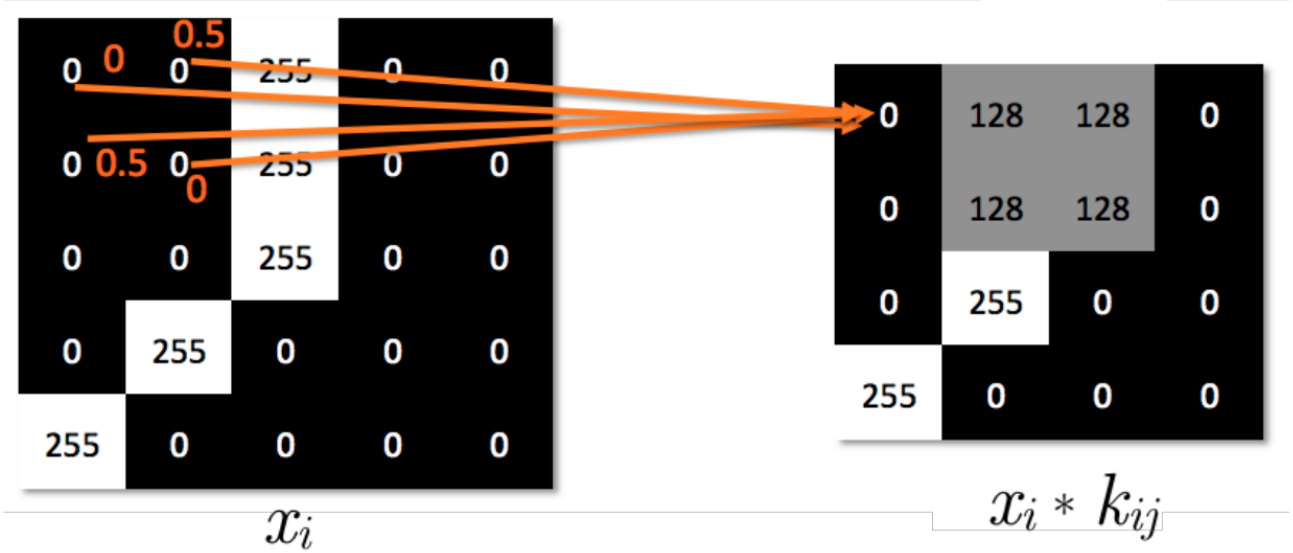
0*1 + 0*0.2 + 0*0.2 + 20*0 = 0



Illustration

$$x * k_{ij}, \quad \text{where } W_{ij} = \tilde{W}_{ij}$$





Element-wise activation function after convolution

$\Rightarrow$ detector of a feature at any position in the image

$$x * k_{ij}, \quad \text{where } W_{ij} = \tilde{W}_{ij}$$

0	0.5
0.5	0

0	0	255	0	0
0	0	255	0	0
0	0	255	0	0
0	255	0	0	0
255	0	0	0	0

x_i

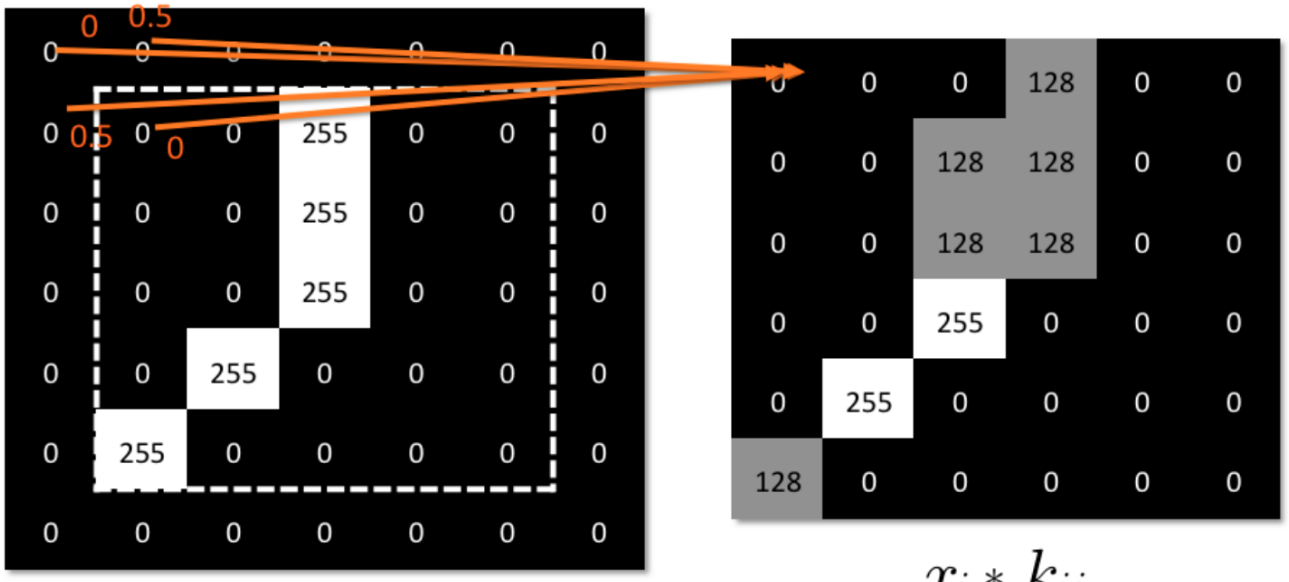
0.02	0.19	0.19	0.02
0.02	0.19	0.19	0.02
0.02	0.75	0.02	0.02
0.75	0.02	0.02	0.02

$\text{sigm}(0.02\ x_i * k_{ij}\ \text{-}4)$

Padding

Use zero padding to allow going over the boundary

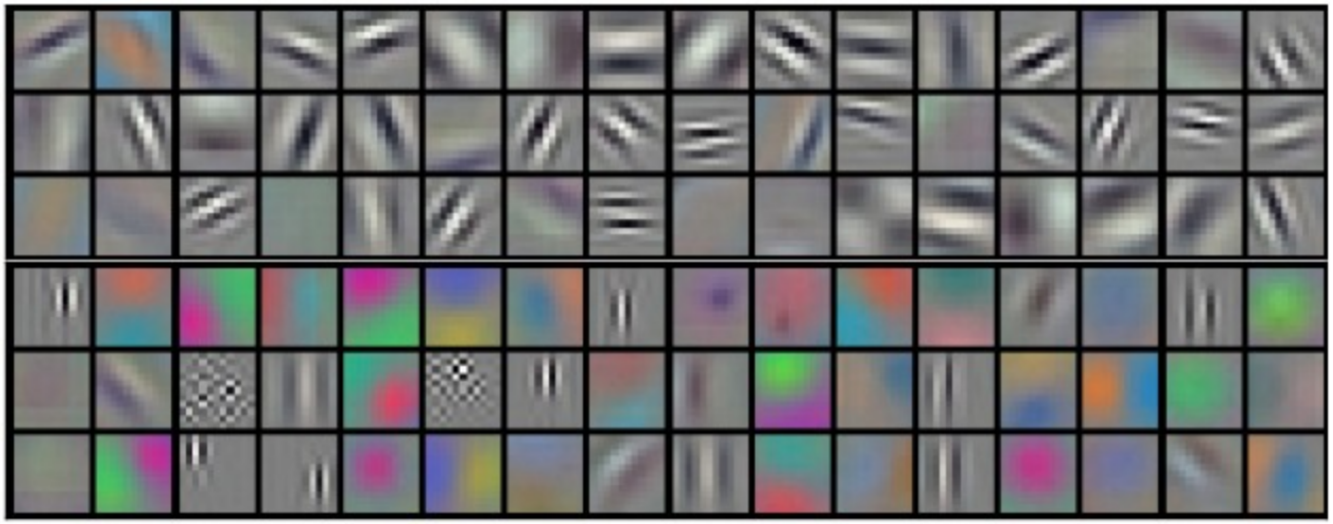
Easier to control the size of output layer



x_i $\qquad$ $x_i \underline{*} k_{ij}$

Learned Kernels

Example kernels learned by AlexNet



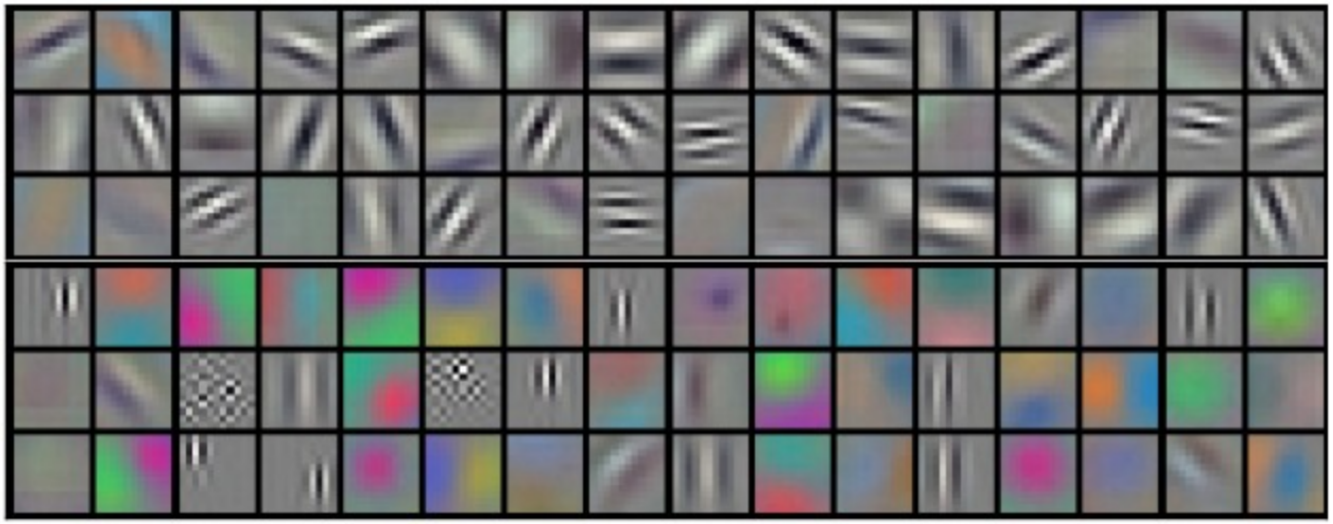
Number of parameters:

Example: 200×200 image, 100 kernels, kernel size 10×10

$\Rightarrow 10 \times 10 \times 100 = 10\text{K}$ parameters

Learned Kernels

Example kernels learned by AlexNet



Number of parameters:

Example: 200×200 image, 100 kernels, kernel size 10×10

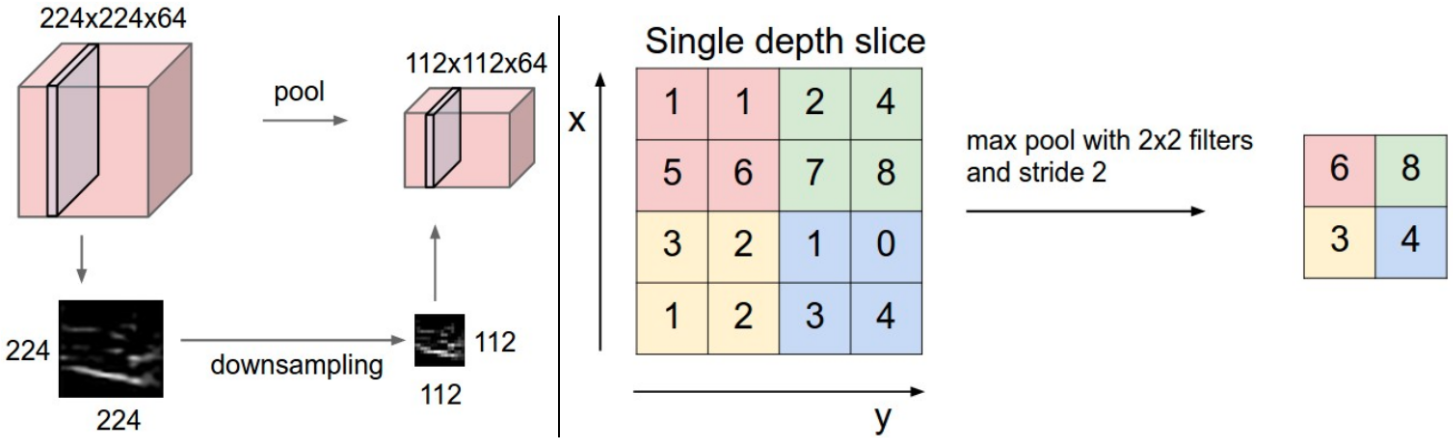
$\Rightarrow 10 \times 10 \times 100 = 10\text{K}$ parameters

Pooling

Common to insert a pooling layer in-between successive convolutional layers

Reduce the size of representation (avoid overfitting?)

Example: Max Pooling

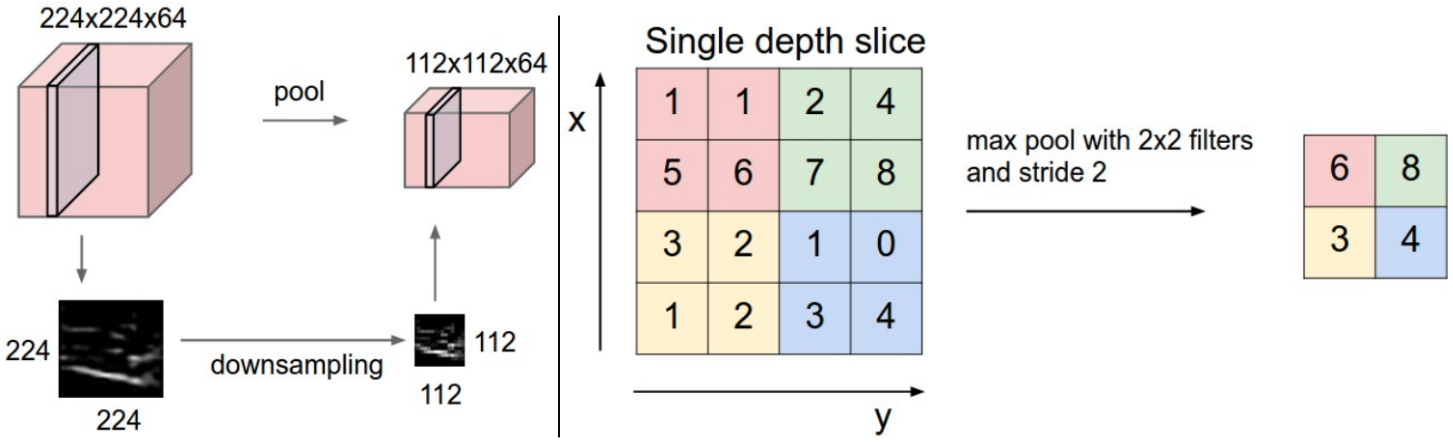


Pooling

Common to insert a pooling layer in-between successive convolutional layers

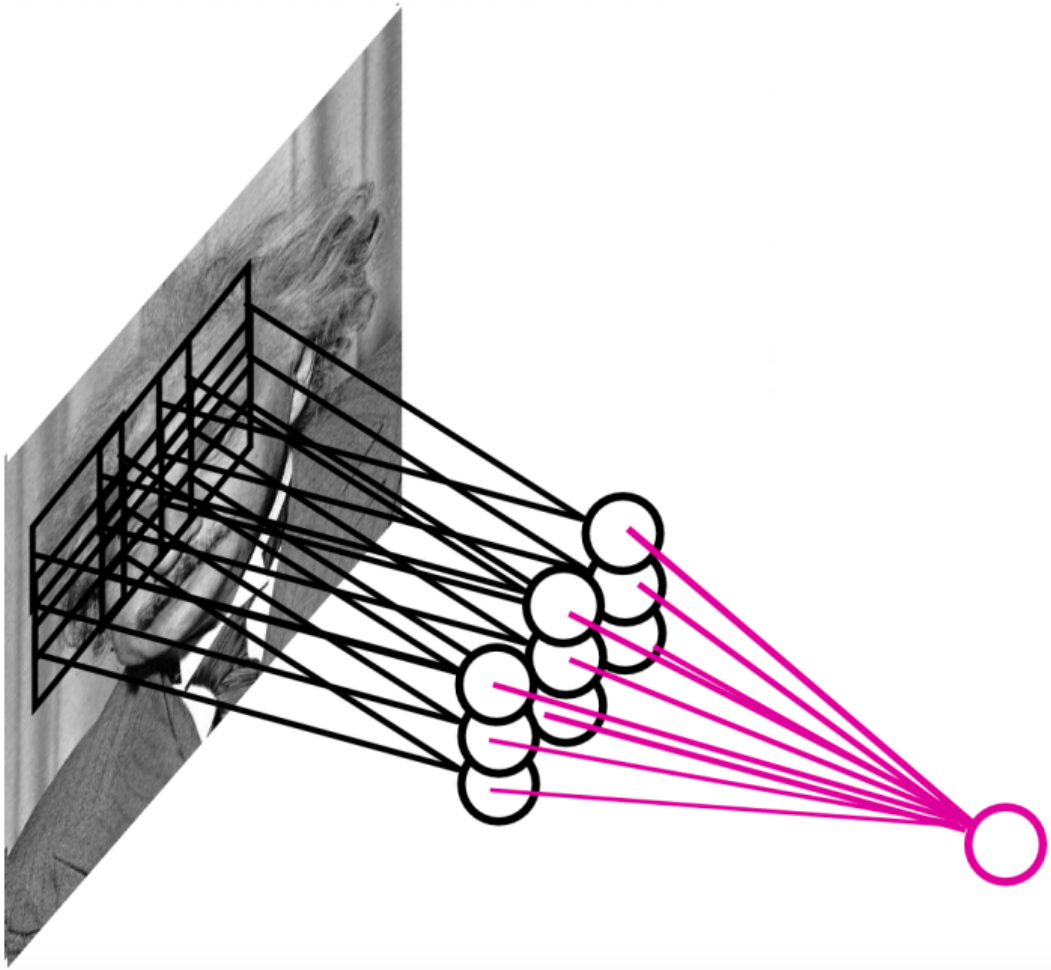
Reduce the size of representation (avoid overfitting?)

Example: Max Pooling

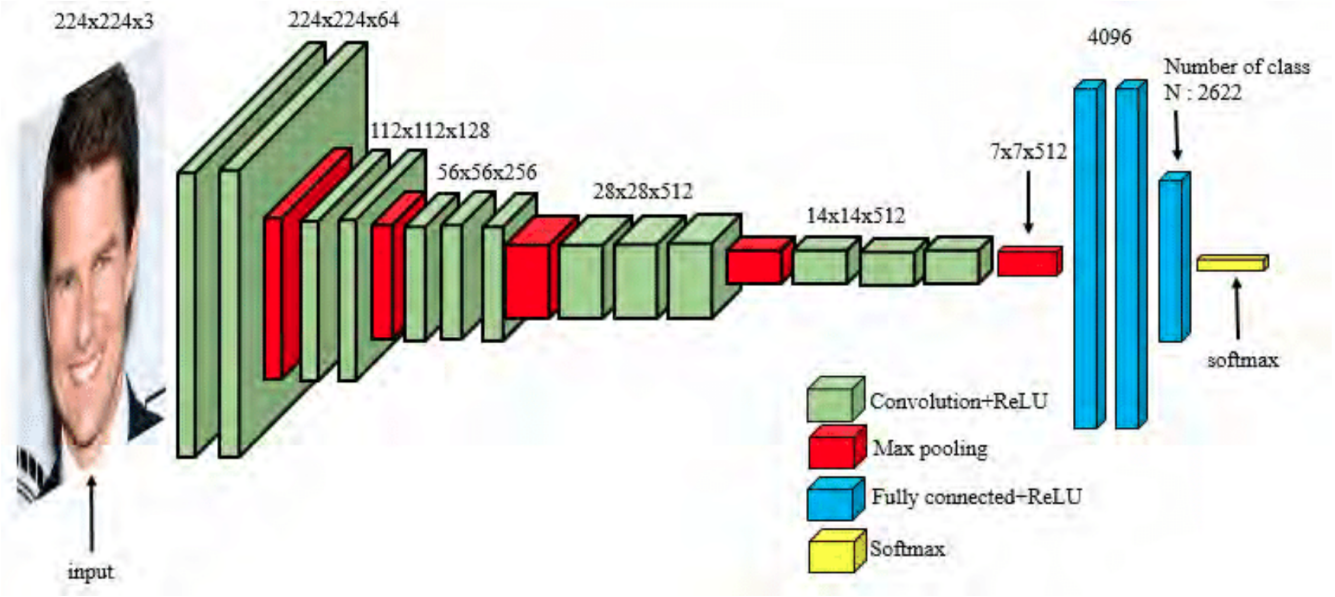


Pooling

By pooling, we gain robustness to the exact spatial location of features



Example: VGG Network



Example: VGG Network

```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000
```

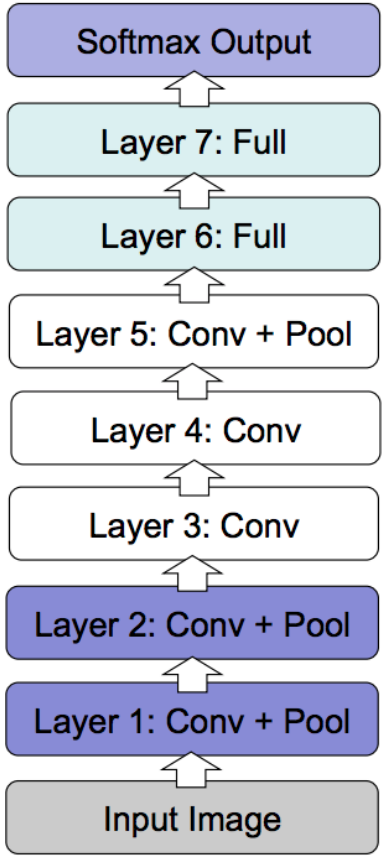
Output provides an estimate of the conditional probability of each class

Training:

Apply SGD to minimize in-sample training error
Backpropagation can be extended to convolutional layer and pooling layer to compute gradient!

AlexNet

8 layers total

Trained on ImageNet dataset

18.2% top-5 error