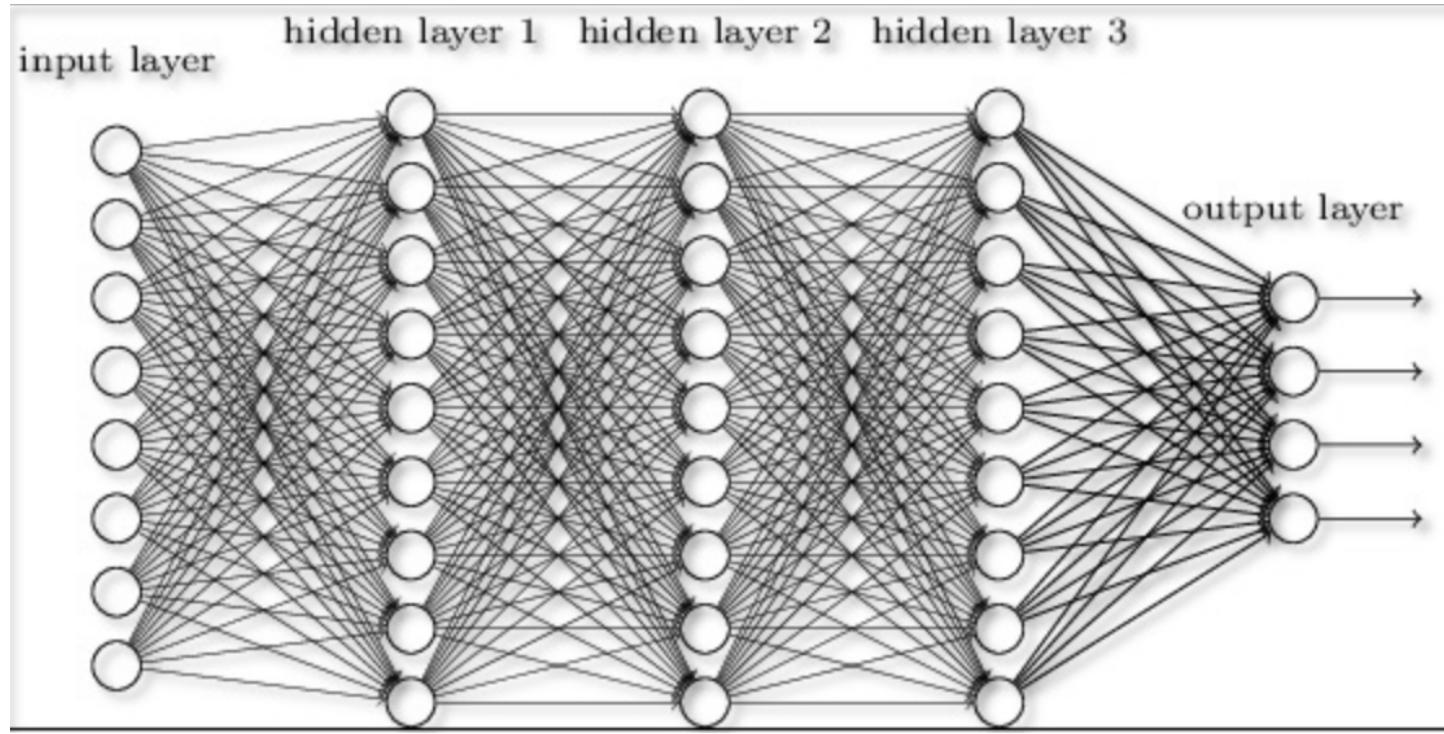


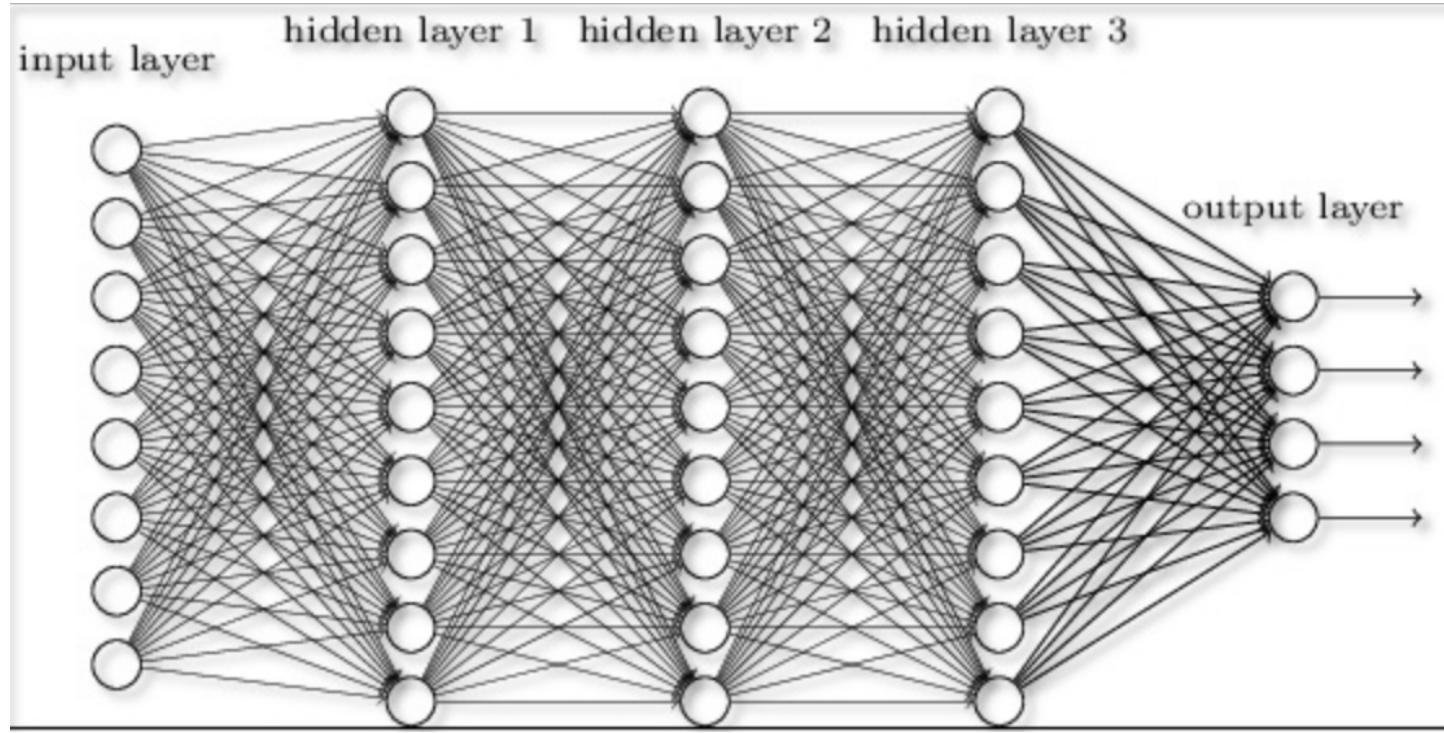
CAN WE GO DEEP NOW?

CAN WE GO DEEP NOW?

ALMOST THERE...LET'S THINK AGAIN
ABOUT WHAT WE PUT AS INPUT...

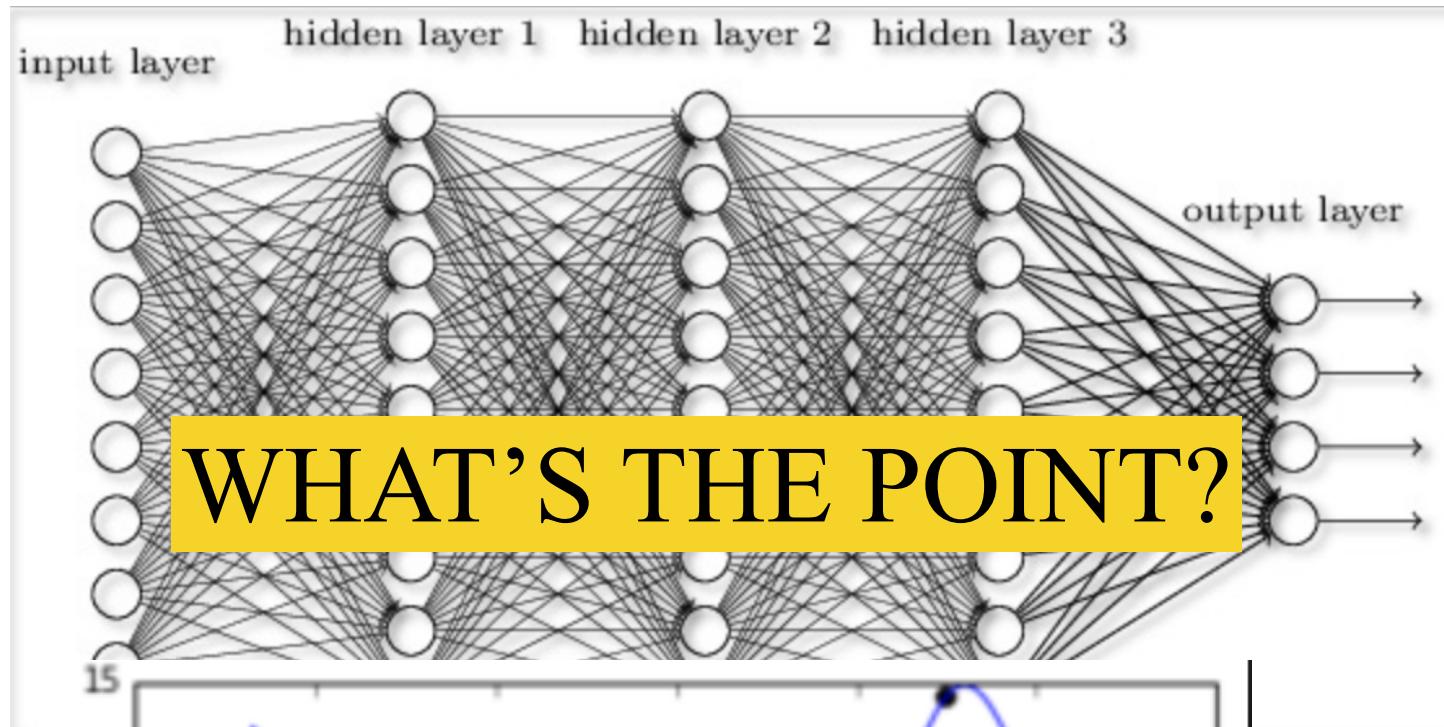


DEEPER → INCREASING NUMBER OF PARAMETERS



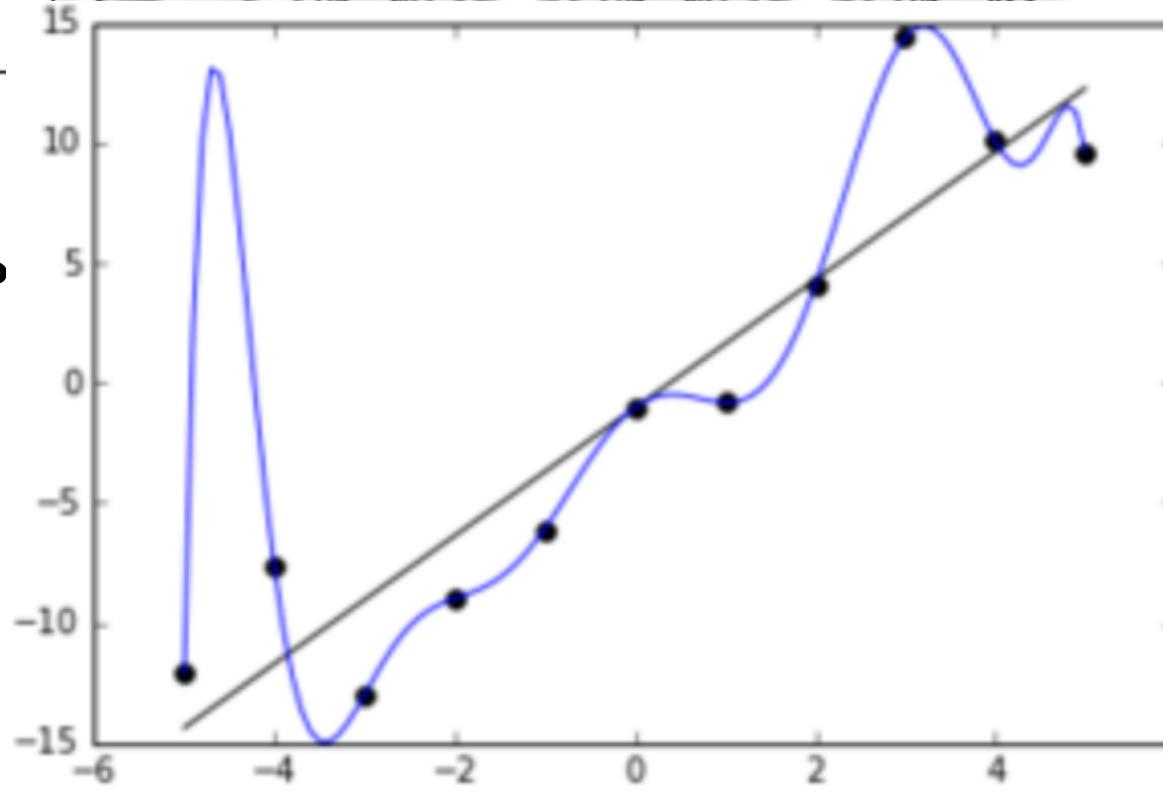
DEEPER → INCREASING NUMBER OF PARAMETERS

IF OUR INPUT CONTAINS A HANDFUL
OF PARAMETERS [colors, sizes, lines etc..]



DEEP

ETERS



What about using raw data?

ALL INFORMATION IS IN THE INPUT DATA

WHY REDUCING ?

LET THE NETWORK FIND THE INFO

What about using raw data?

ALL INFORMATION IS IN THE INPUT DATA

WHY REDUCING ?

LET THE NETWORK FIND THE INFO

LARGE DIMENSION SIGNALS SUCH AS IMAGES OR SPECTRA WOULD REQUIRE TREMENDOUSLY LARGE MODELS

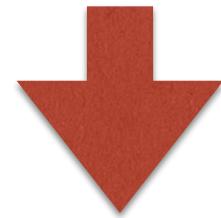
A 512x512 image as input of a fully connected layer producing output of same size:

$$(512 \times 512)^2 = 7e10$$

BUT

FEEDING INDIVIDUAL RESOLUTION ELEMENTS IS NOT
VERY EFFICIENT SINCE IT LOOSES ALL INVARIANCE TO
TRANSLATION

FEEDING INDIVIDUAL RESOLUTION ELEMENTS IS NOT
VERY EFFICIENT SINCE IT LOOSES ALL INVARIANCE TO
TRANSLATION



SO?

Discrete Convolution

1D:
[Spectra]

$$f(x) * g(x) = \sum_{k=-\infty}^{k=+\infty} f(k).g(k - x)$$

2D:
[Images]

$$f(x, y) * g(x, y) = \sum_{k=-\infty}^{k=+\infty} \sum_{l=-\infty}^{l=+\infty} f(k, l).g(x - k, y - l)$$

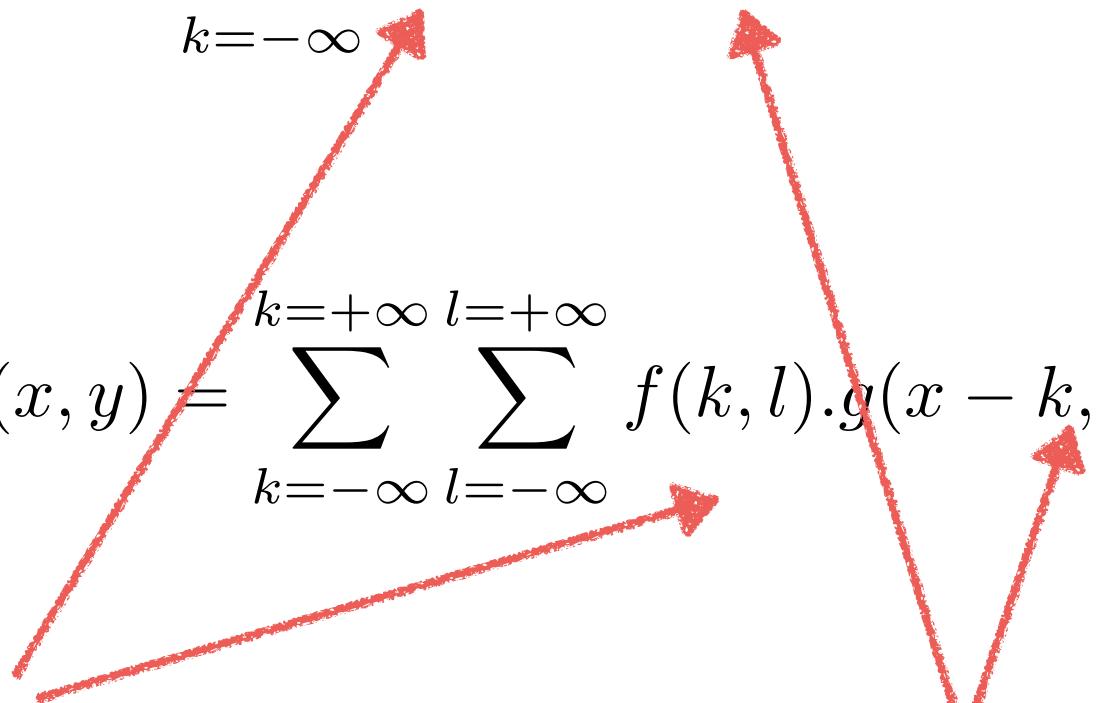
Discrete Convolution

1D:
[Spectra]

$$f(x) * g(x) = \sum_{k=-\infty}^{k=+\infty} f(k).g(k - x)$$

2D:
[Images]

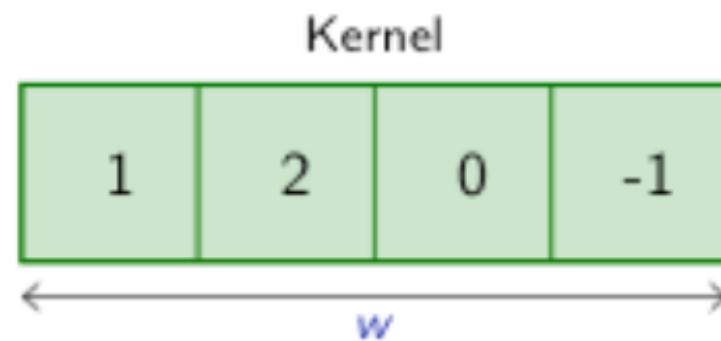
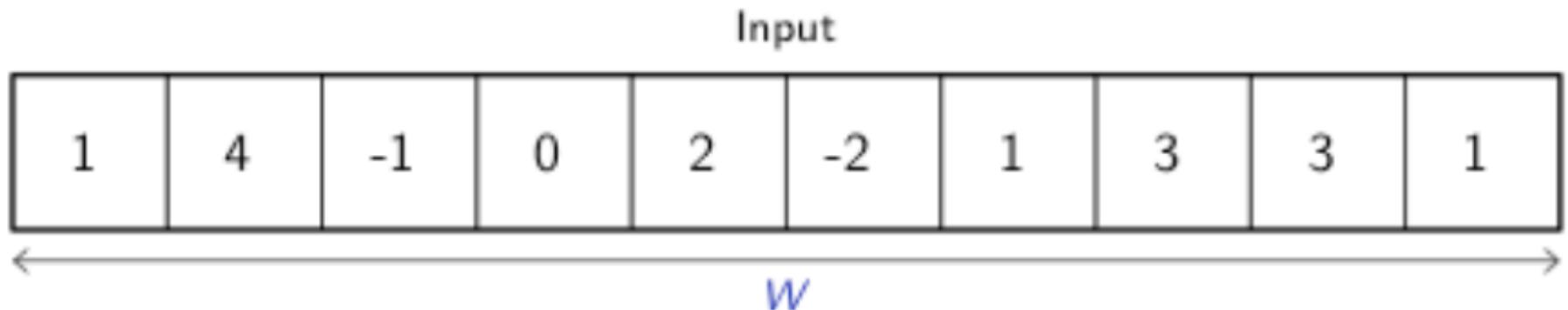
$$f(x, y) * g(x, y) = \sum_{k=-\infty}^{k=+\infty} \sum_{l=-\infty}^{l=+\infty} f(k, l).g(x - k, y - l)$$



CONVOLUTION KERNEL

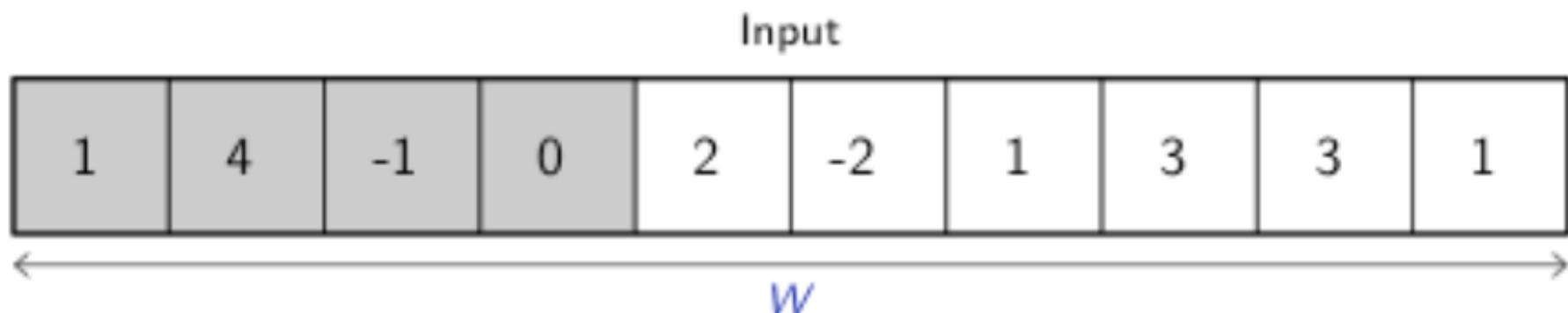
INPUT DATA

1-D CONVOLUTION



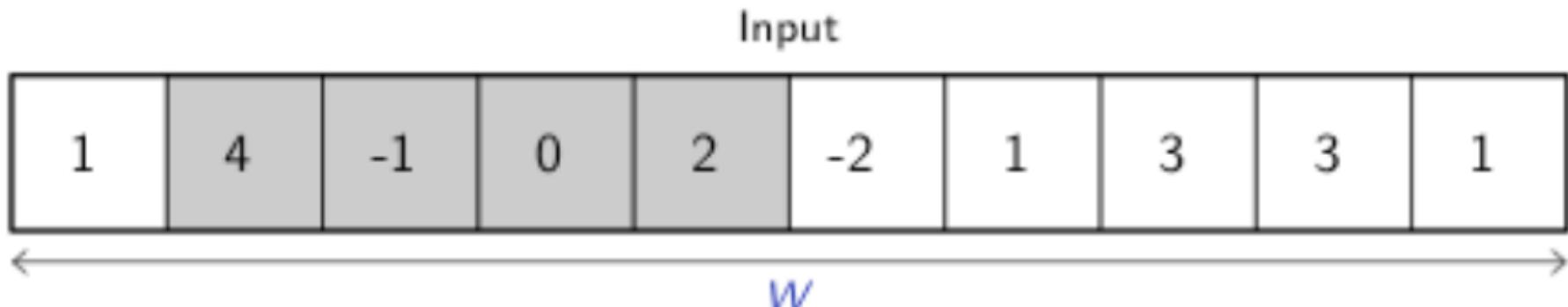
credit

1-D CONVOLUTION



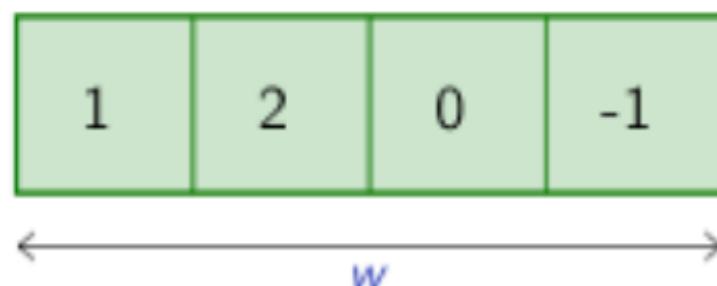
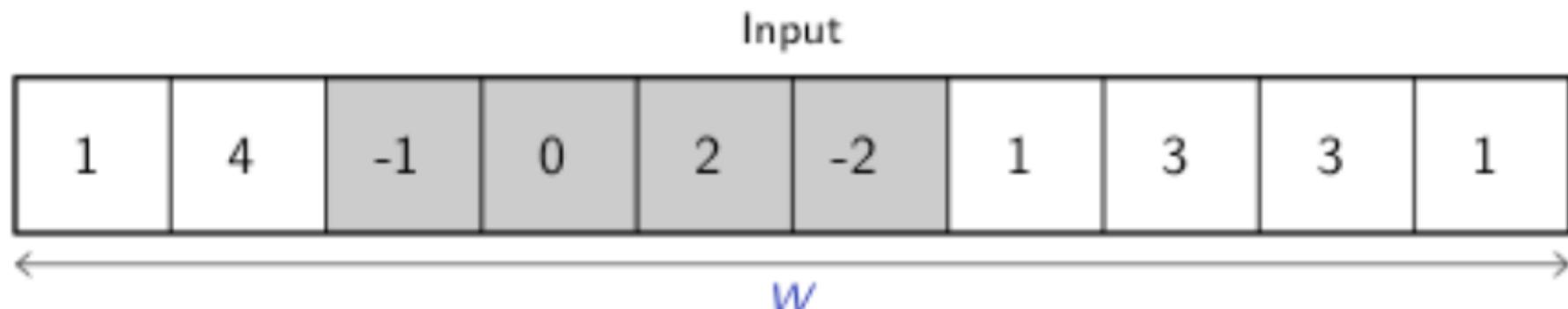
credit

1-D CONVOLUTION



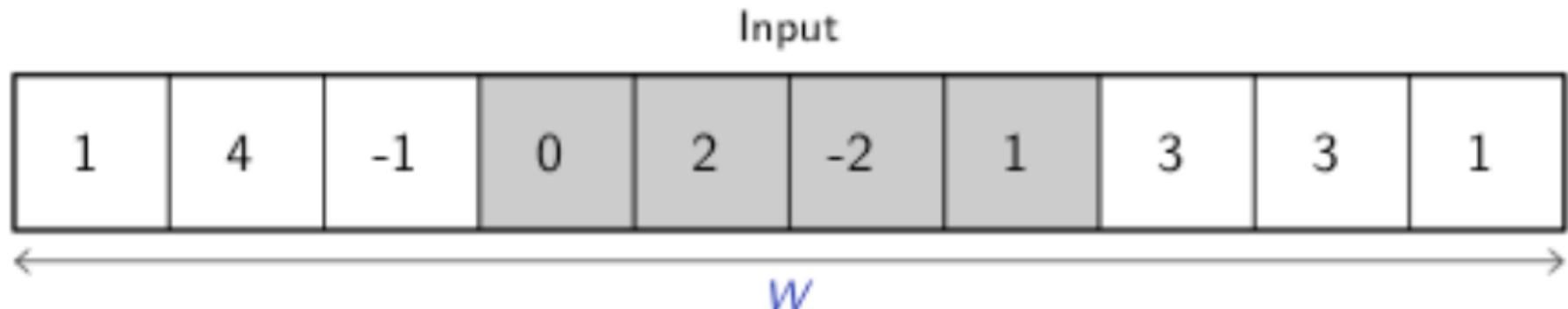
credit

1-D CONVOLUTION



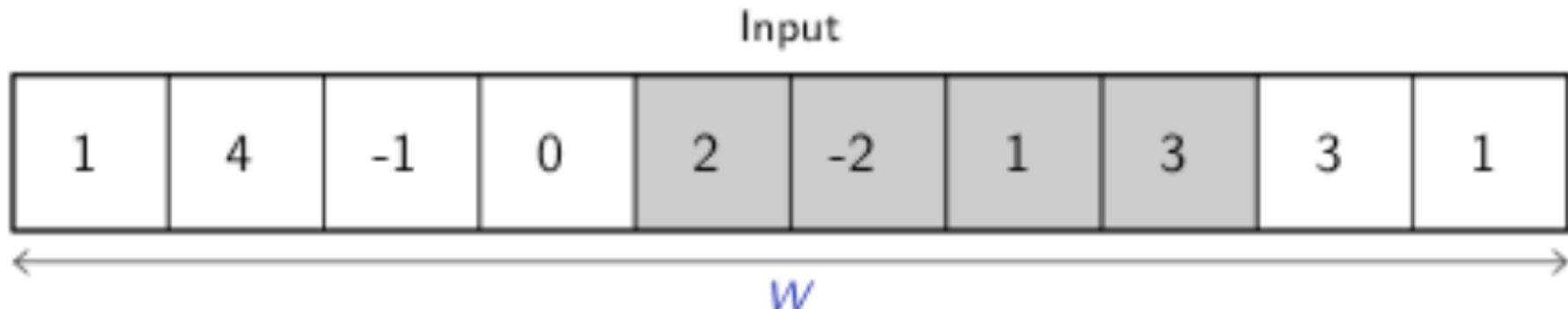
credit

1-D CONVOLUTION



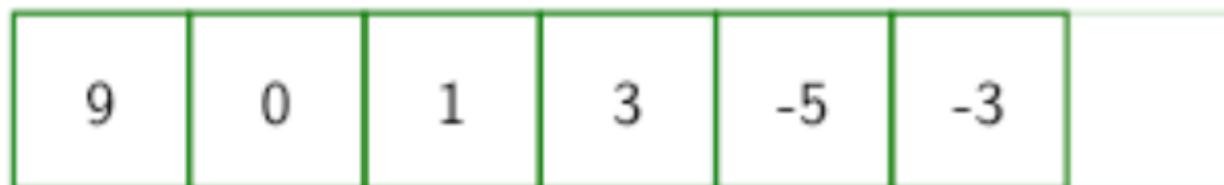
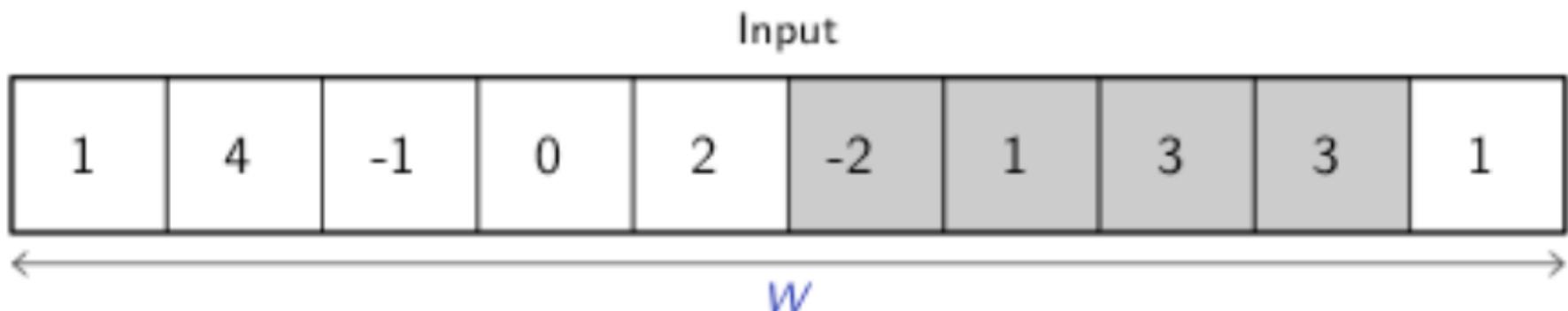
credit

1-D CONVOLUTION



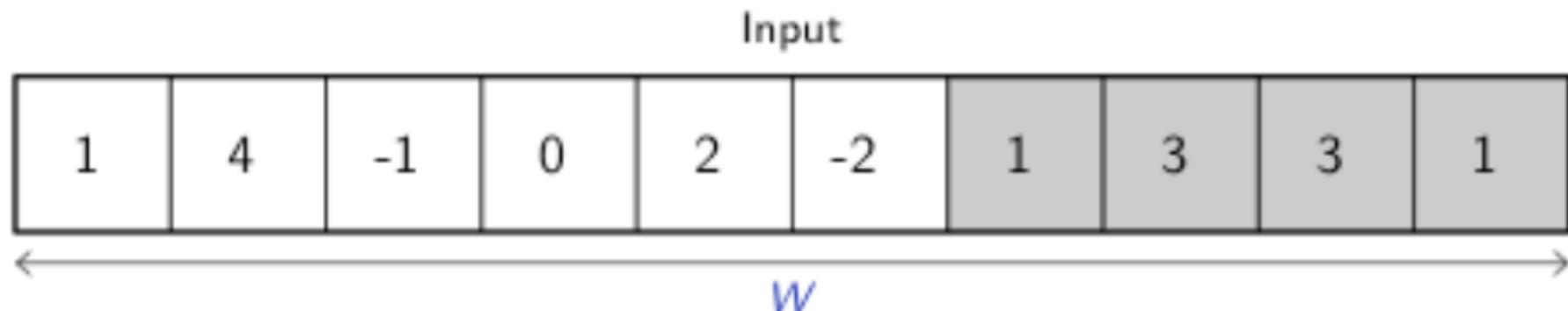
credit

1-D CONVOLUTION



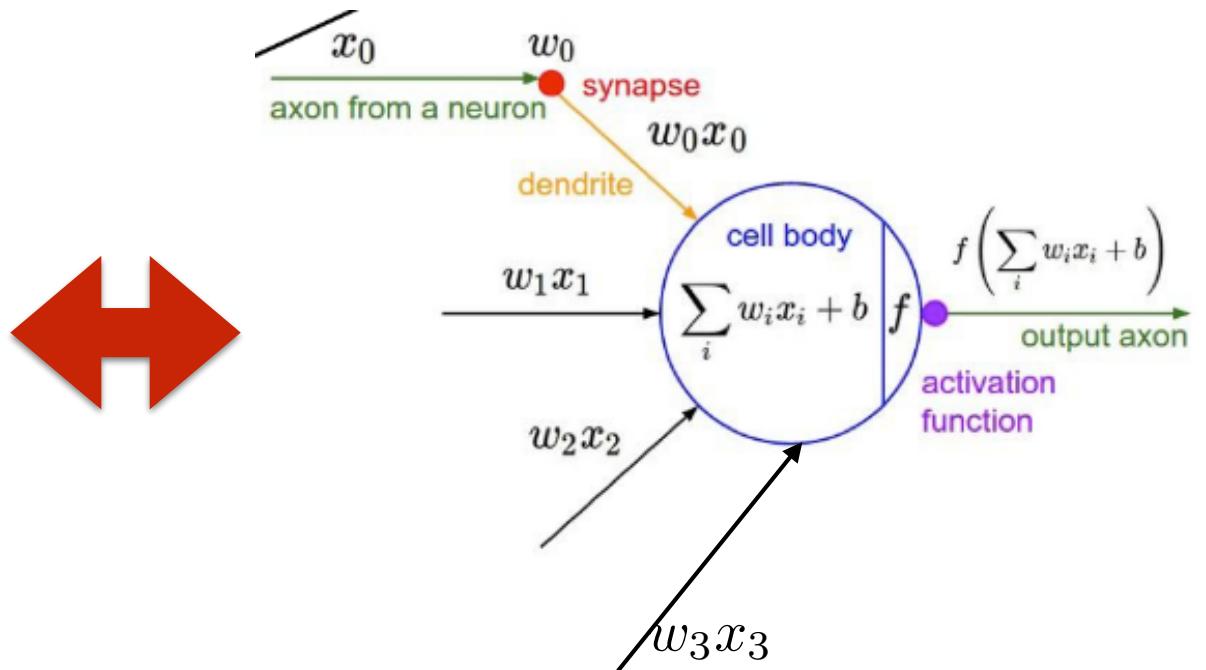
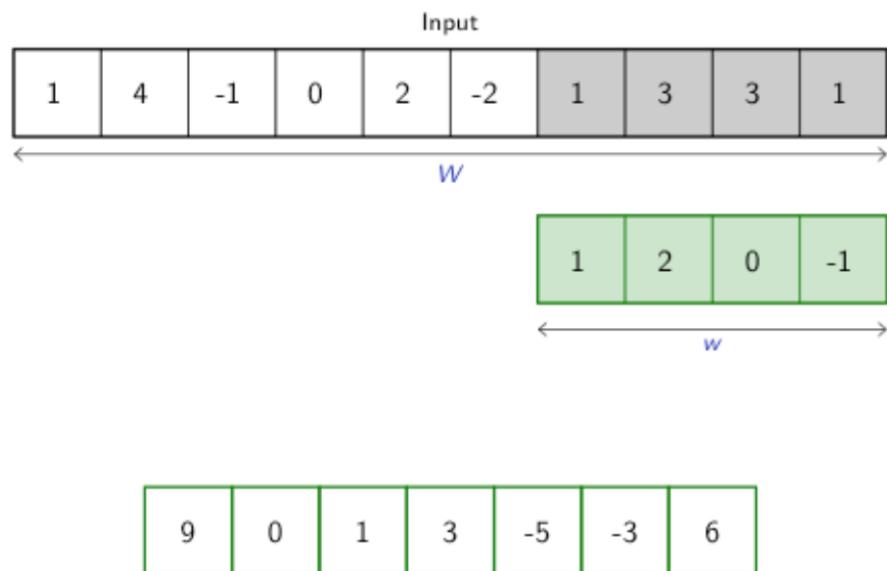
credit

1-D CONVOLUTION

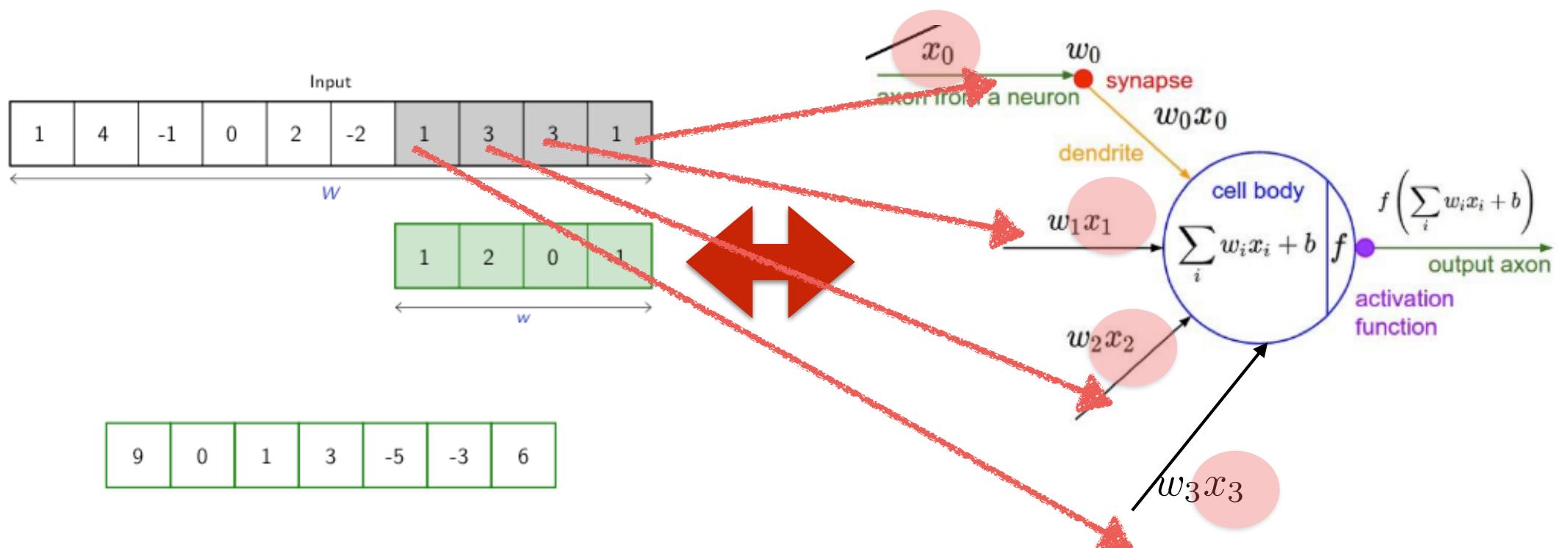


credit

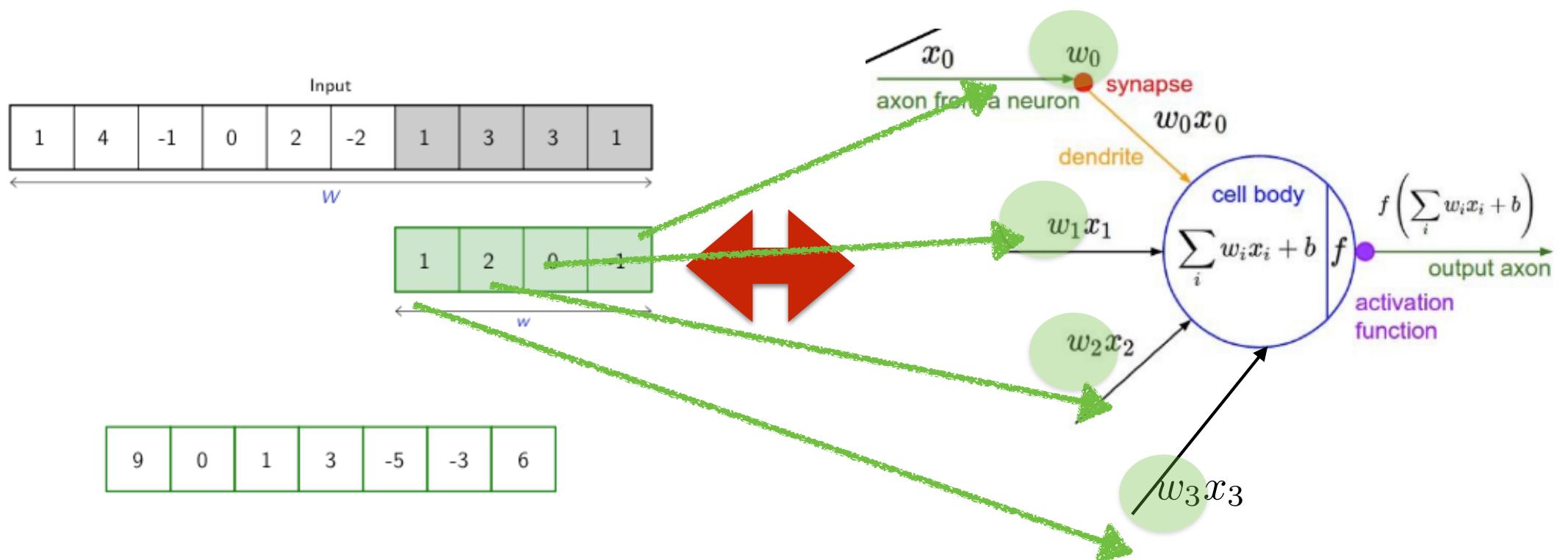
THE CONVOLUTION OPERATION IS EQUIVALENT TO A NEURON WITH AS MANY INPUTS AS KERNEL ELEMENTS AND WEIGHTS EQUAL TO THE KERNEL !



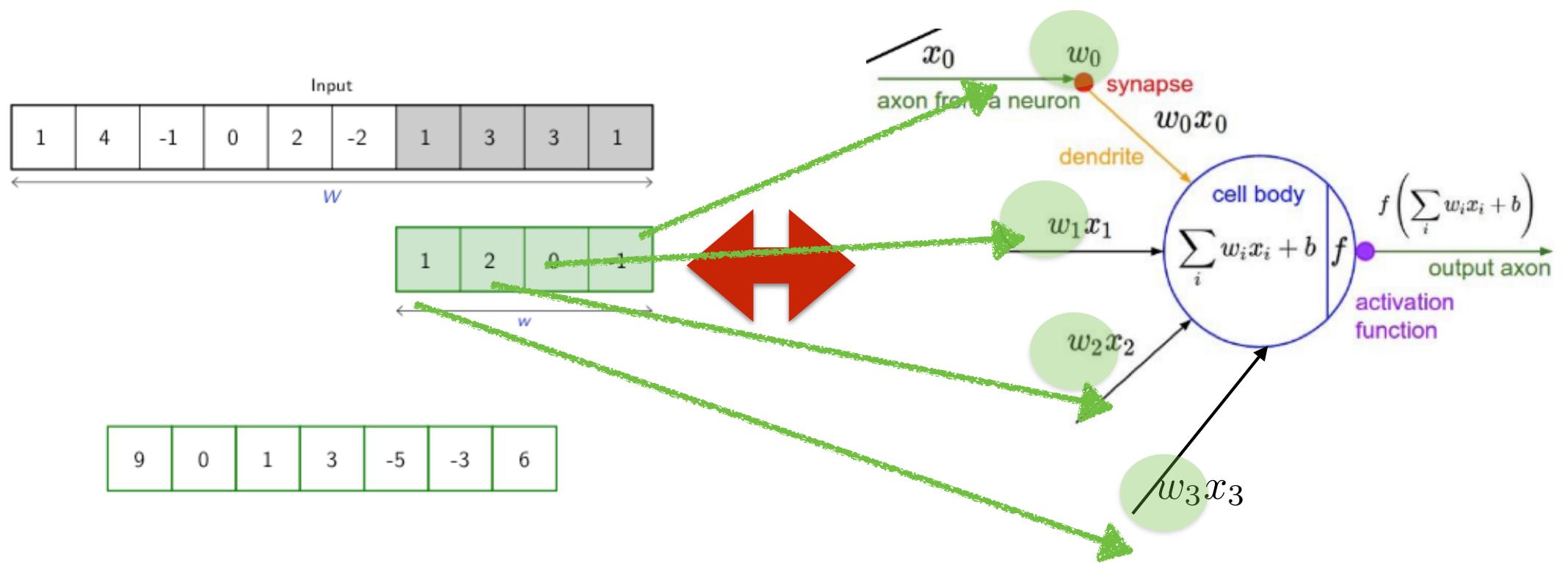
THE CONVOLUTION OPERATION IS EQUIVALENT TO A NEURON WITH AS MANY INPUTS AS KERNEL ELEMENTS AND WEIGHTS EQUAL TO THE KERNEL !



THE CONVOLUTION OPERATION IS EQUIVALENT TO A NEURON WITH AS MANY INPUTS AS KERNEL ELEMENTS AND WEIGHTS EQUAL TO THE KERNEL !



THE CONVOLUTION OPERATION IS EQUIVALENT TO A NEURON WITH AS MANY INPUTS AS KERNEL ELEMENTS AND WEIGHTS EQUAL TO THE KERNEL !



WITH THE ADVANTAGE THAT THE SAME WEIGHTS ARE APPLIED TO ALL THE SIGNAL: TRANSLATION INVARIANCE

2-D CONVOLUTION

SAME IDEA, BUT THE KERNEL IS NOW 2D

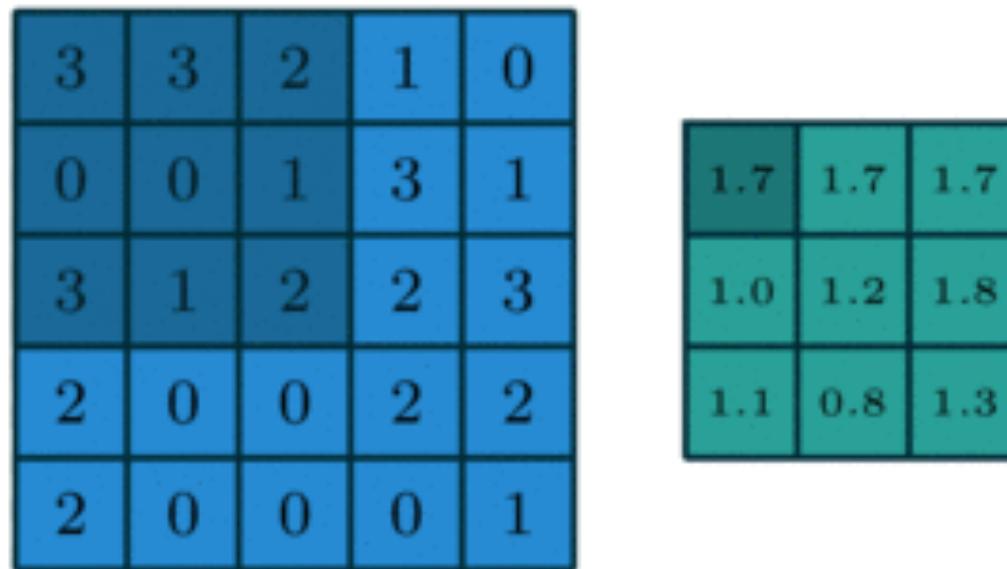
| | | | | |
|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 3 | 1 |
| 3 | 1 | 2 | 2 | 3 |
| 2 | 0 | 0 | 2 | 2 |
| 2 | 0 | 0 | 0 | 1 |

| | | |
|-----|-----|-----|
| 1.7 | 1.7 | 1.7 |
| 1.0 | 1.2 | 1.8 |
| 1.1 | 0.8 | 1.3 |

Credit: animations from https://github.com/vdumoulin/conv_arithmetic

2-D CONVOLUTION

SAME IDEA, BUT THE KERNEL IS NOW 2D



IN THE EXAMPLE: EACH 3x3 REGION GENERATES AN OUTPUT

$$Size_{output} = Size_{input} - Size_{kernel} + 1$$

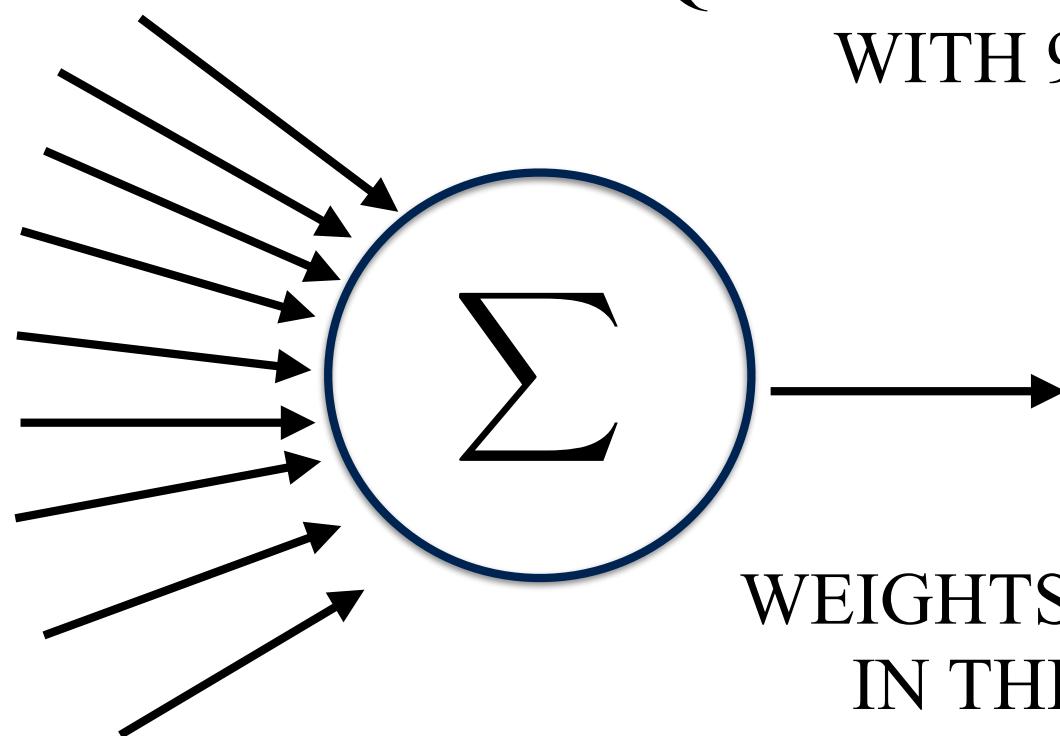
Credit: animations from https://github.com/vdumoulin/conv_arithmetic



| | | | | |
|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 3 | 1 |
| 3 | 1 | 2 | 2 | 3 |
| 2 | 0 | 0 | 2 | 2 |
| 2 | 0 | 0 | 0 | 1 |

| | | |
|-----|-----|-----|
| 1.7 | 1.7 | 1.7 |
| 1.0 | 1.2 | 1.8 |
| 1.1 | 0.8 | 1.3 |

EQUIVALENT TO A NEURON
WITH 9 INPUTS



WEIGHTS ARE CODED
IN THE KERNEL

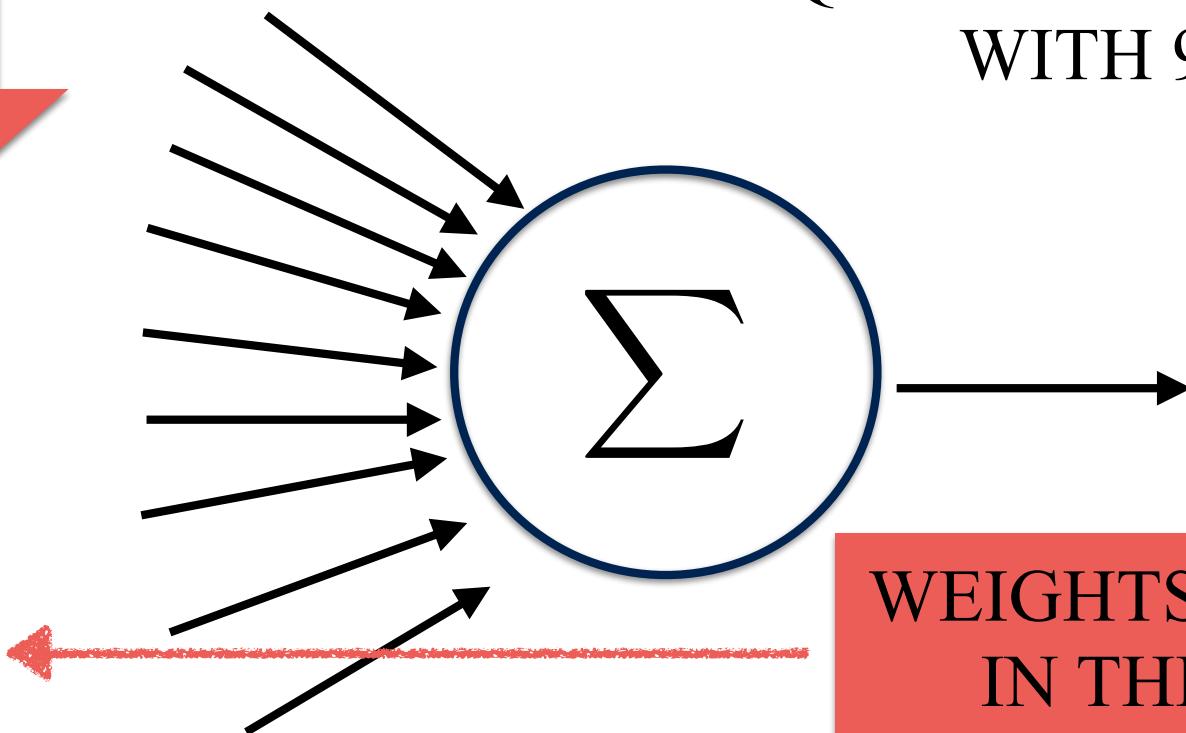


| | | | | |
|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 3 | 1 |
| 3 | 1 | 2 | 2 | 3 |
| 2 | 0 | 0 | 2 | 2 |
| 2 | 0 | 0 | 0 | 1 |

| | | |
|-----|-----|-----|
| 1.7 | 1.7 | 1.7 |
| 1.0 | 1.2 | 1.8 |
| 1.1 | 0.8 | 1.3 |

EQUIVALENT TO A NEURON
WITH 9 INPUTS

THIS IS WHAT
THE
NETWORK
LEARNS!



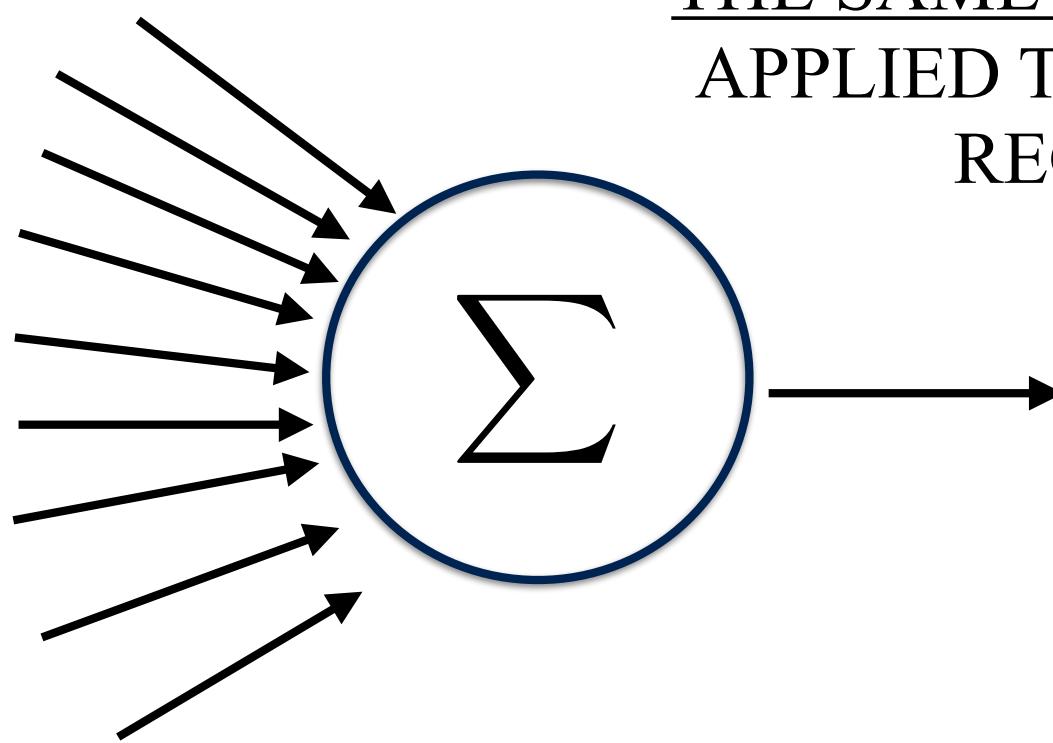
WEIGHTS ARE CODED
IN THE KERNEL



| | | | | |
|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 3 | 1 |
| 3 | 1 | 2 | 2 | 3 |
| 2 | 0 | 0 | 2 | 2 |
| 2 | 0 | 0 | 0 | 1 |

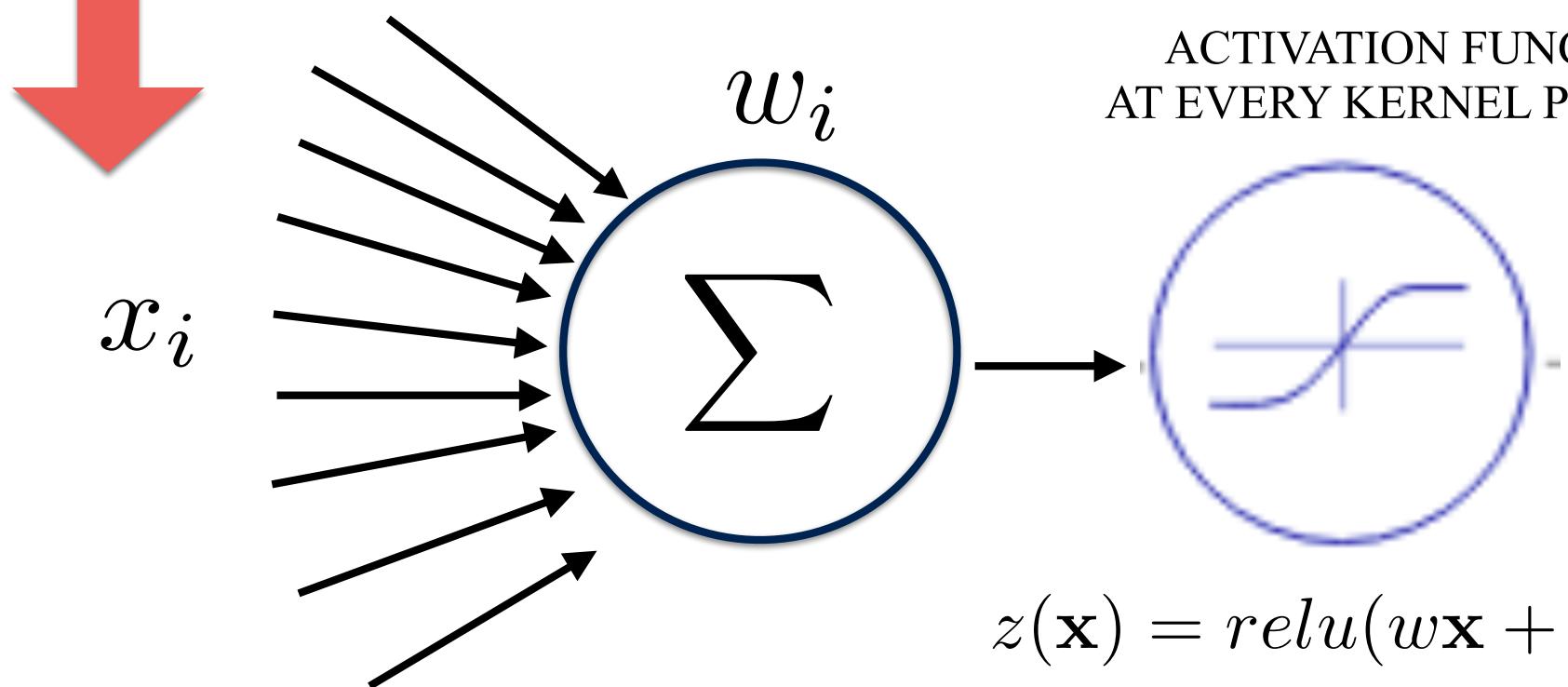
| | | |
|-----|-----|-----|
| 1.7 | 1.7 | 1.7 |
| 1.0 | 1.2 | 1.8 |
| 1.1 | 0.8 | 1.3 |

THE KEY IS AGAIN THAT
THE SAME WEIGHTS ARE
APPLIED TO ALL IMAGE
REGIONS



| | |
|-------|--|
| x_i | |
| | |

| | |
|-----------|--|
| w_i | |
| [weights] | |



x_i

| | | | | |
|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 3 | 1 |
| 3 | 1 | 2 | 2 | 3 |
| 2 | 0 | 0 | 2 | 2 |
| 2 | 0 | 0 | 0 | 1 |



$x_{i,j}$

w_i

IN DEEP NETWORKS ReLU is the most commonly used activation function - see Vanishing Gradient Problem

| | | |
|-----|-----|-----|
| 1.7 | 1.7 | 1.7 |
| 1.0 | 1.2 | 1.8 |
| 1.1 | 0.8 | 1.3 |

w_i

[weights]

ACTIVATION FUNCTION
AT EVERY KERNEL POSITION

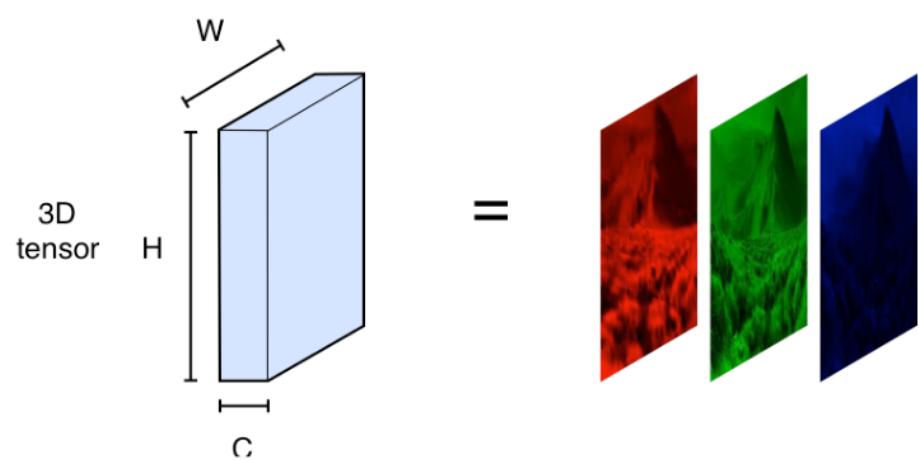
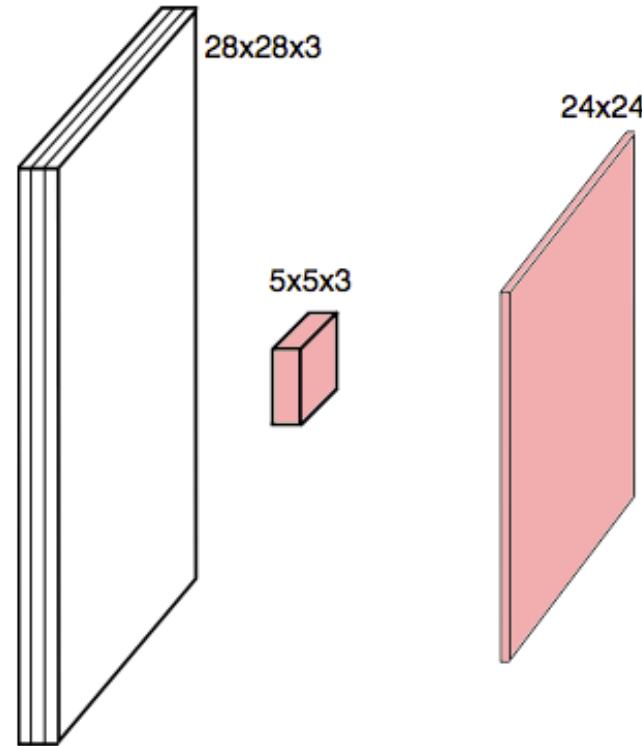


$$z(\mathbf{x}) = \text{relu}(w\mathbf{x} + b)$$



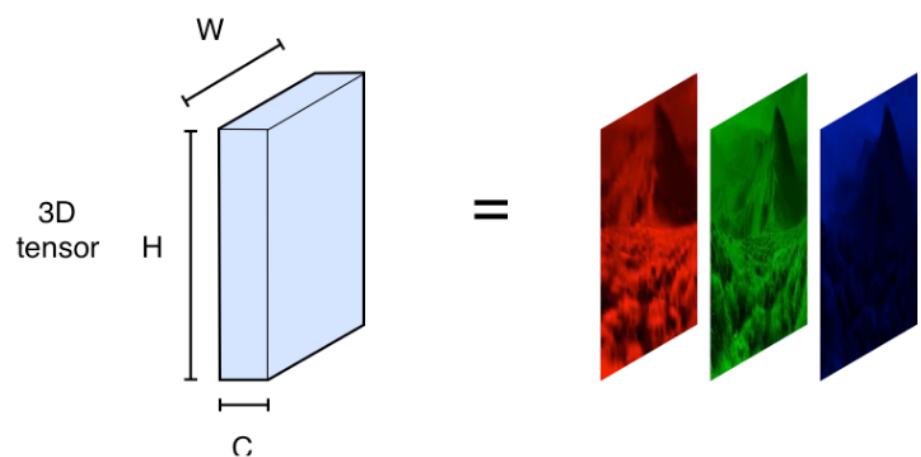
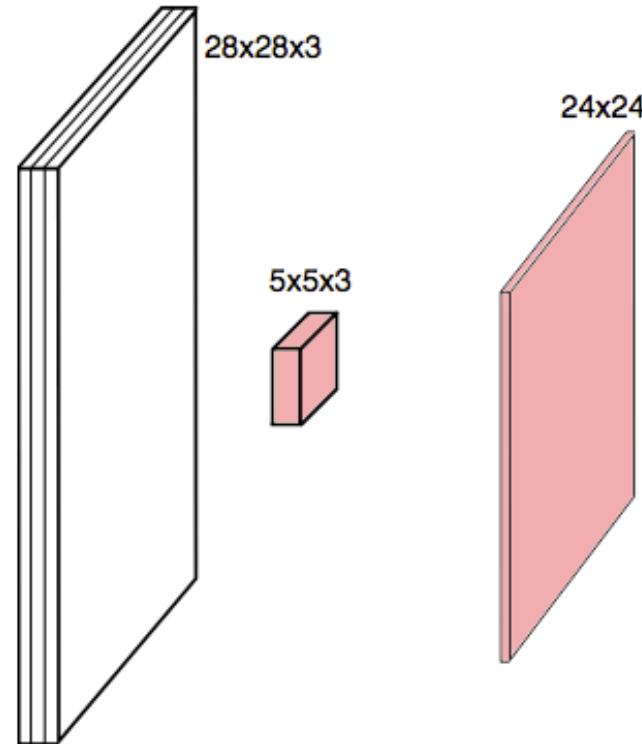
CONVOLUTIONS CAN ALSO BE COMPUTED ACROSS CHANNELS (OR COLORS)

A COLOR IMAGE IS A
TENSOR
OF SIZE height x width x
channels



CONVOLUTIONS CAN ALSO BE COMPUTED ACROSS CHANNELS (OR COLORS)

A COLOR IMAGE IS A
TENSOR
OF SIZE height x width x
channels

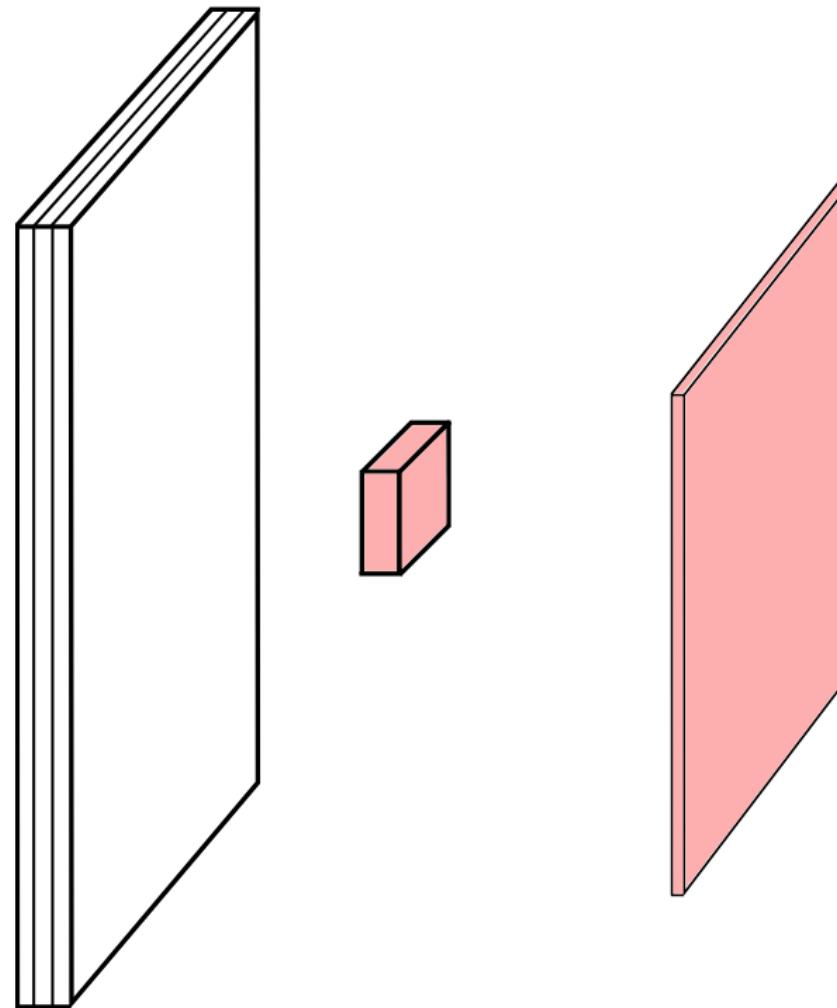


THEN THE KERNEL
HAS ALSO 3
CHANNELS

IN ASTRONOMY ...

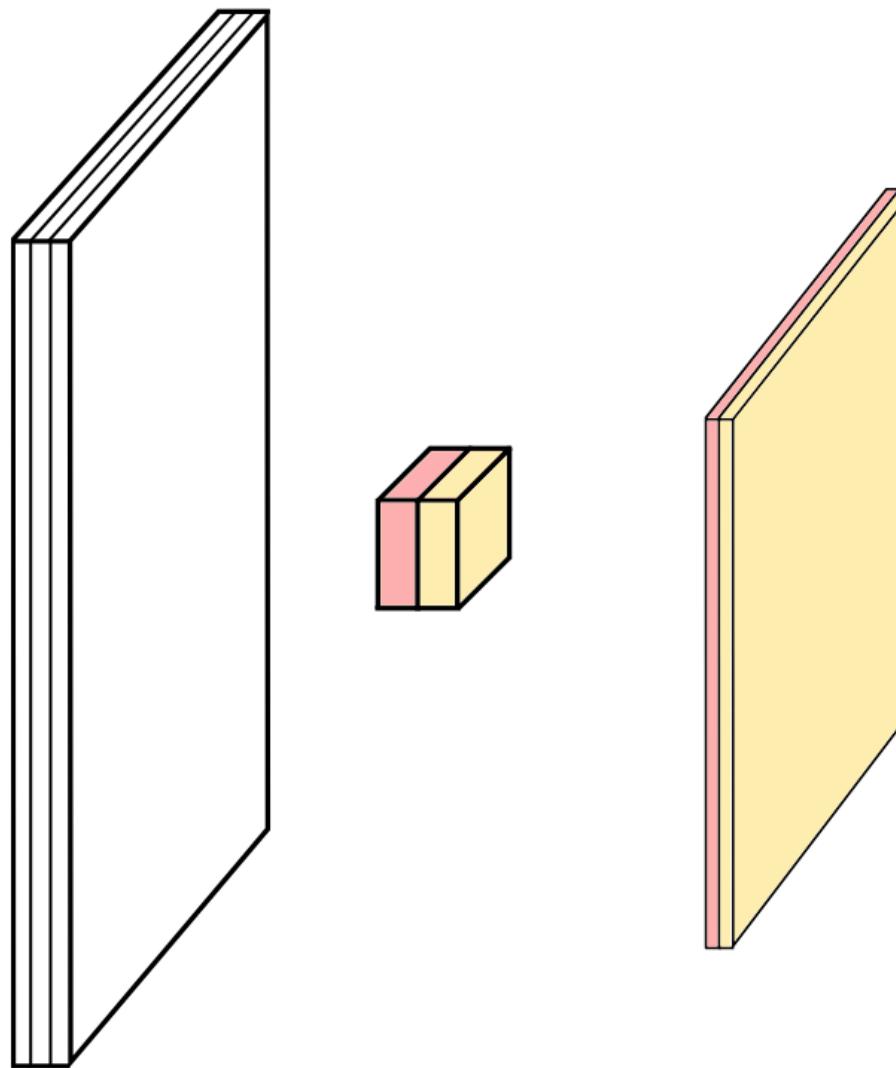
IT OPENS THE DOOR TO ANALYZE MULTIPLE
FILTERS SIMULTANEOUSLY

MULTIPLE CONVOLUTIONS WITH DIFFERENT KERNELS
CAN BE PERFORMED



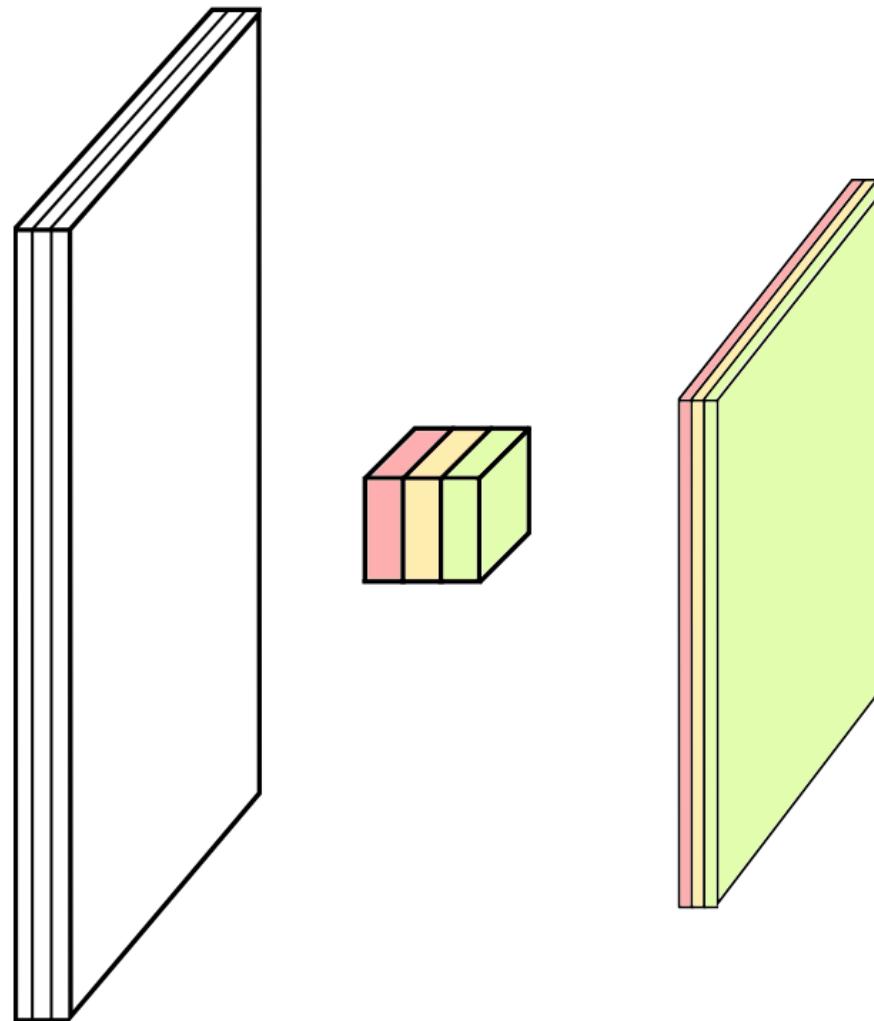
credit

MULTIPLE CONVOLUTIONS WITH DIFFERENT KERNELS CAN BE PERFORMED



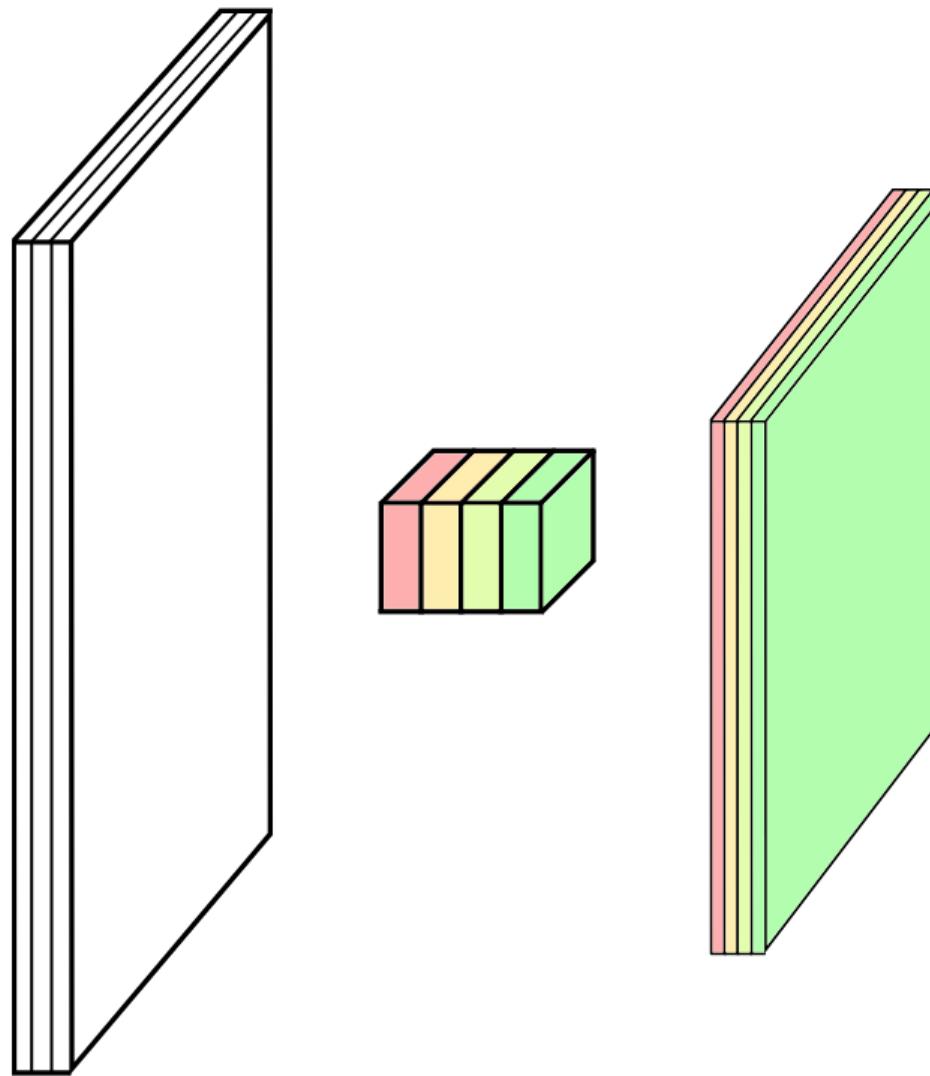
credit

MULTIPLE CONVOLUTIONS WITH DIFFERENT KERNELS CAN BE PERFORMED



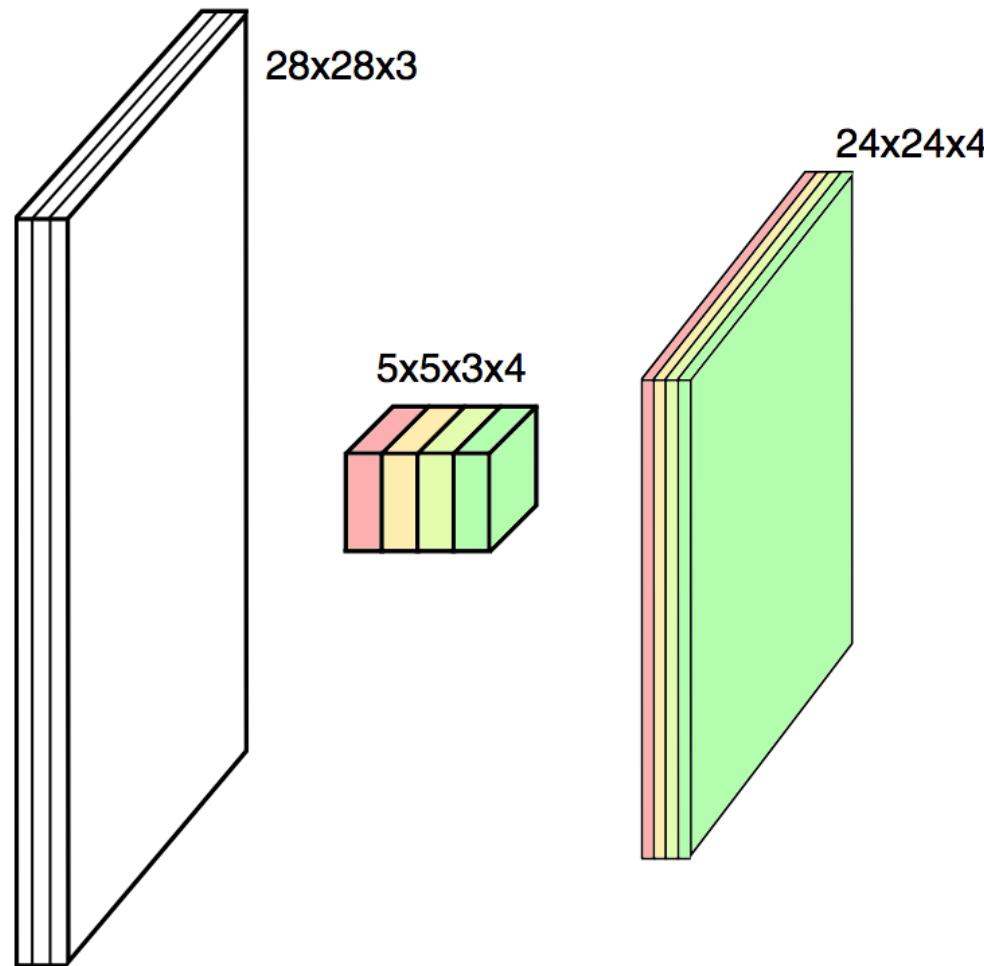
credit

MULTIPLE CONVOLUTIONS WITH DIFFERENT KERNELS CAN BE PERFORMED



credit

MULTIPLE CONVOLUTIONS WITH DIFFERENT KERNELS CAN BE PERFORMED

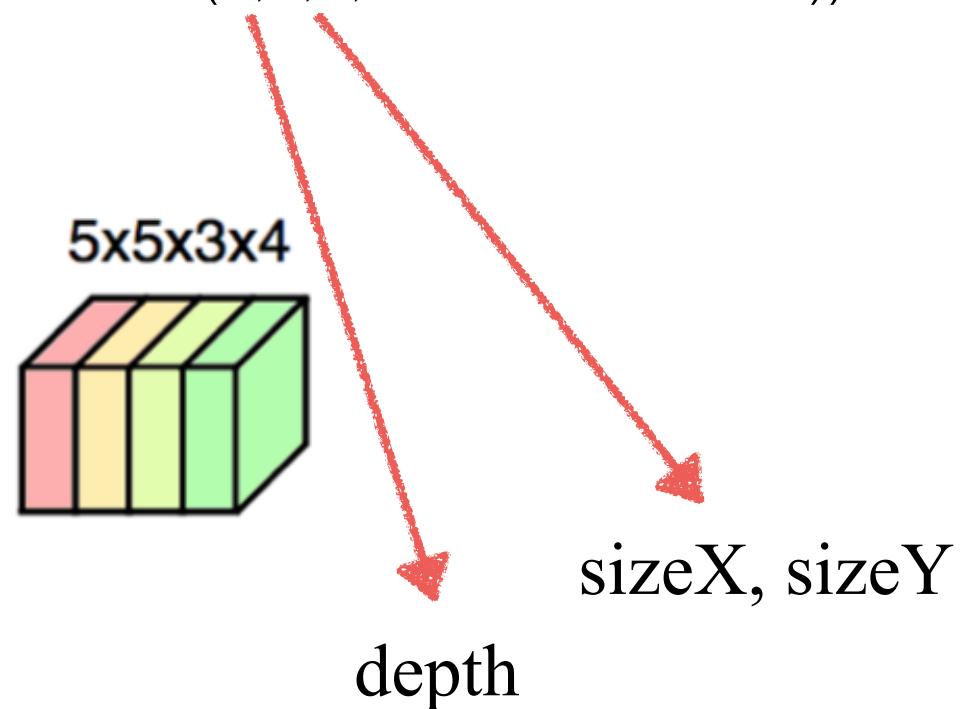


credit

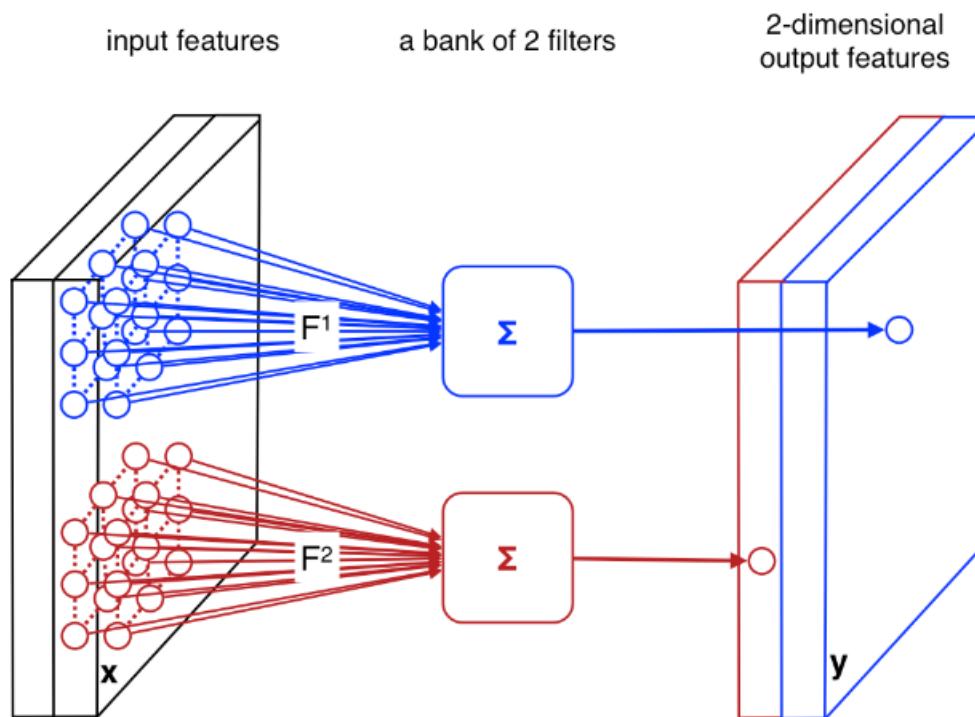
IN KERAS...

```
model = Sequential()
```

```
model.add(Convolution2D(4,5,5, activation="relu"))
```



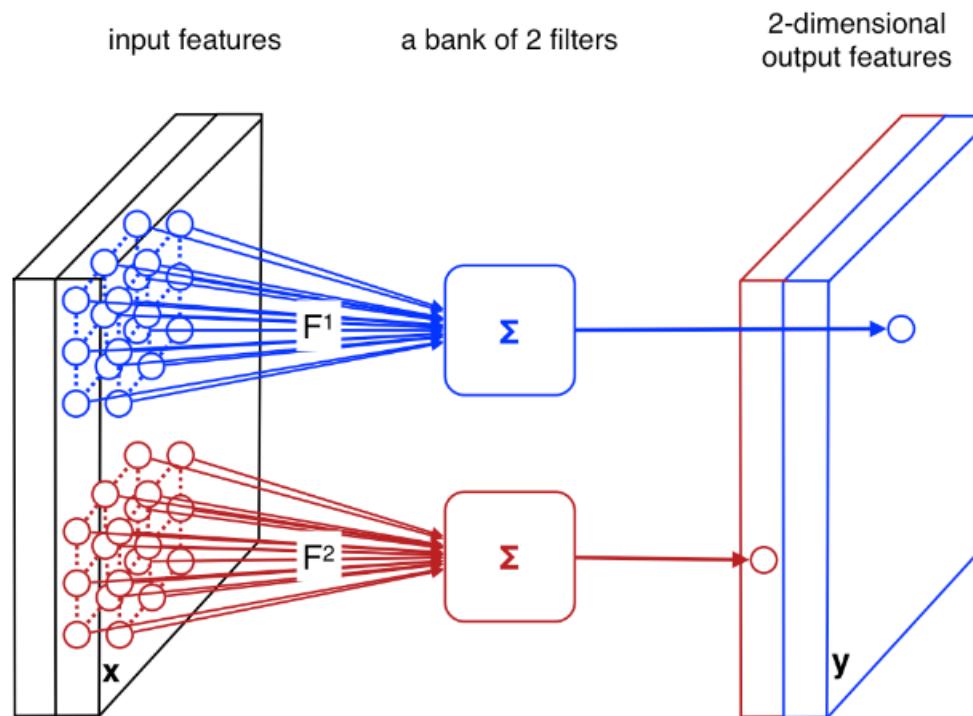
SINCE CONVOLUTIONS OUTPUT ONE SCALAR< THEY CAN BE SEEN AS AN INDIVIDUAL NEURON WITH A RECEPTIVE FIELD LIMITED TO THE KERNEL DIMENSIONS



Credit

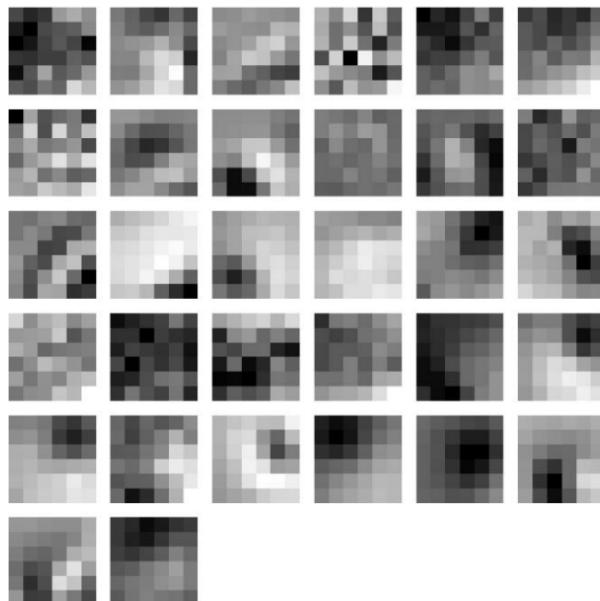
SINCE CONVOLUTIONS OUTPUT ONE SCALAR< THEY CAN BE SEEN AS AN INDIVIDUAL NEURON WITH A RECEPTIVE FIELD LIMITED TO THE KERNEL DIMENSIONS

THE SAME NEURON IS FIRED WITH DIFFERENT AREAS FROM THE INPUT

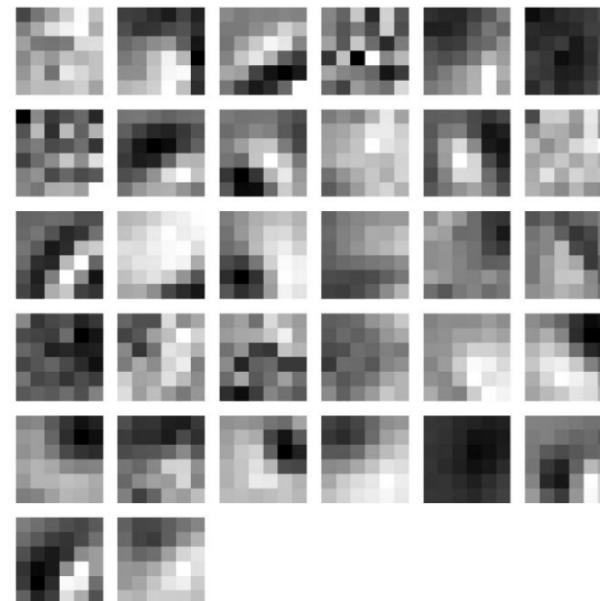


Credit

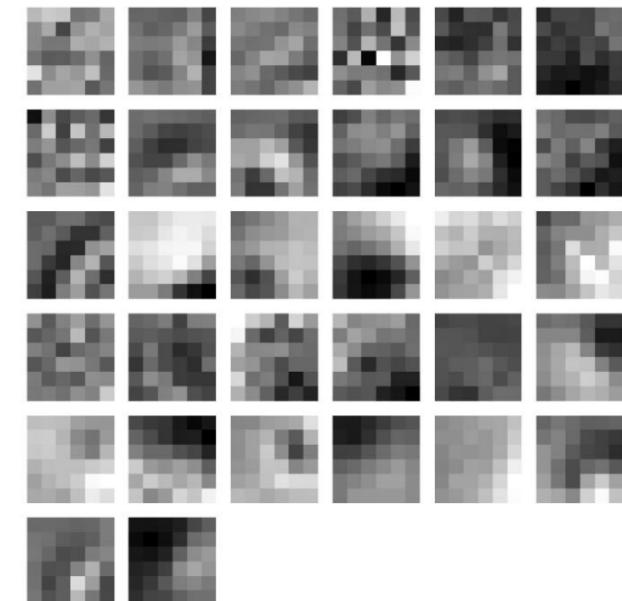
EXAMPLE OF 32 FILTERS LEARNED IN A CONVOLUTIONAL LAYER



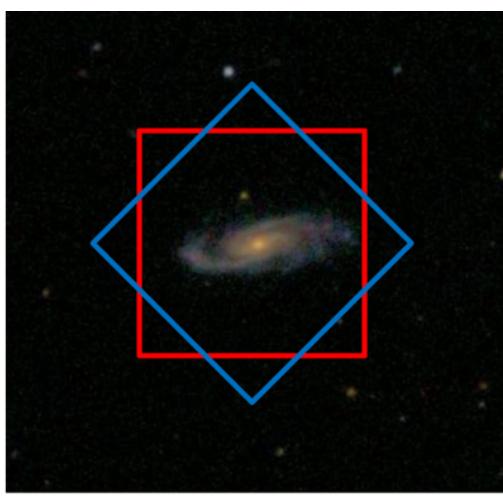
(a) red channel



(b) green channel

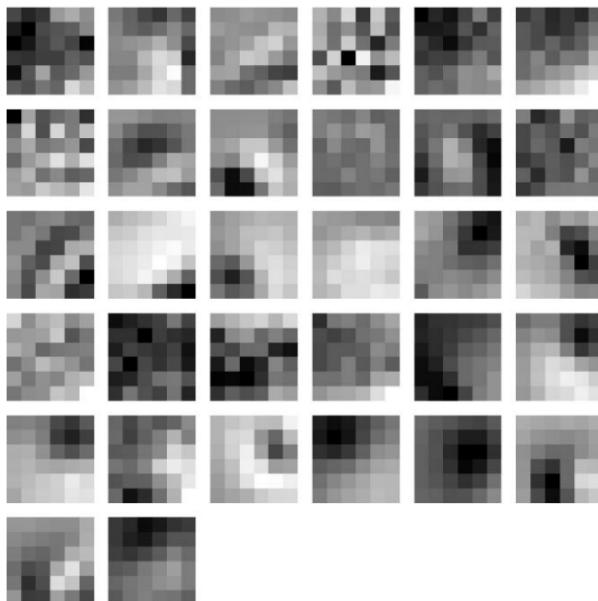


(c) blue channel

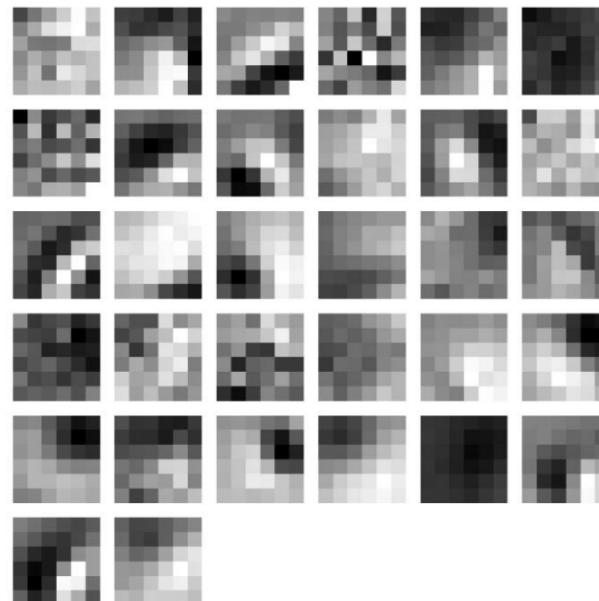


Dieleman+16

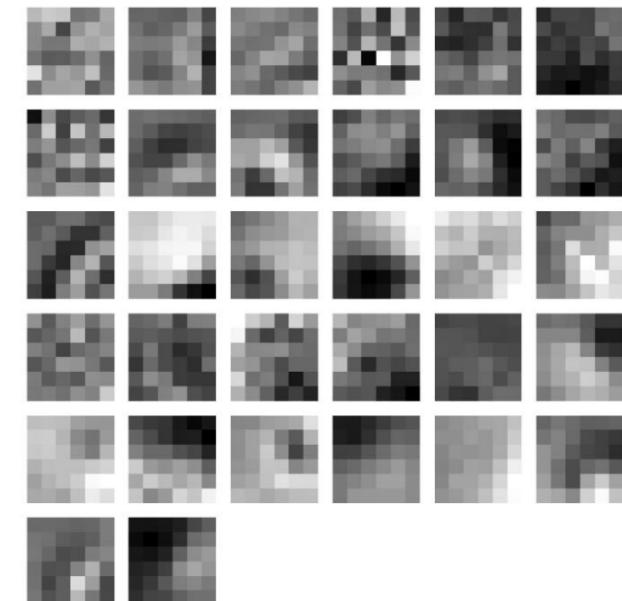
EXAMPLE OF 32 FILTERS LEARNED IN A CONVOLUTIONAL LAYER



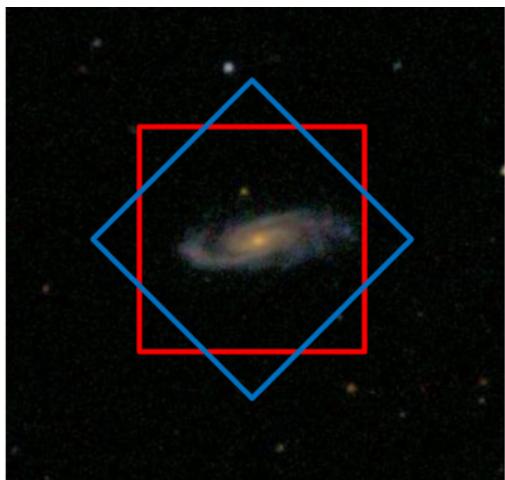
(a) red channel



(b) green channel



(c) blue channel



Dieleman+16

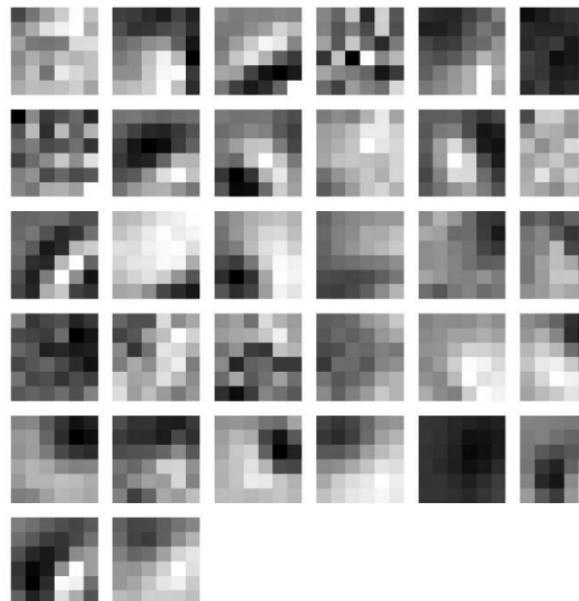
THESE ARE CALLED **FEATURE MAPS**

EXAMPLE OF 32 FILTERS APPLIED BY A CONVOLUTIONAL

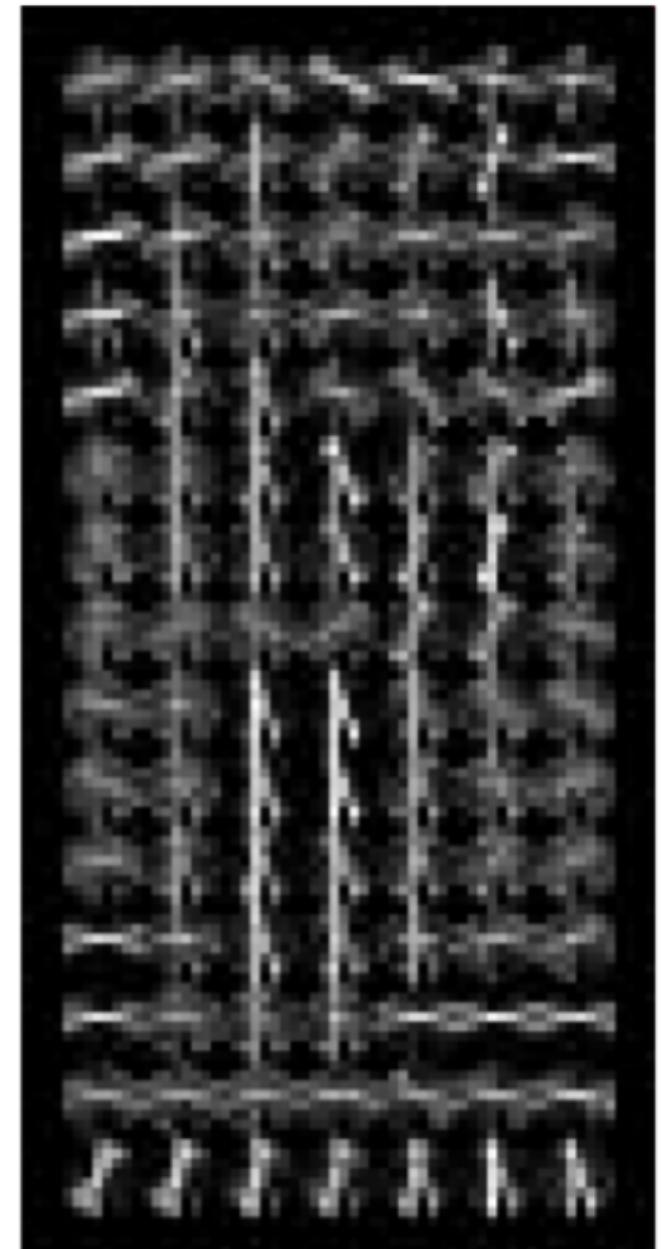
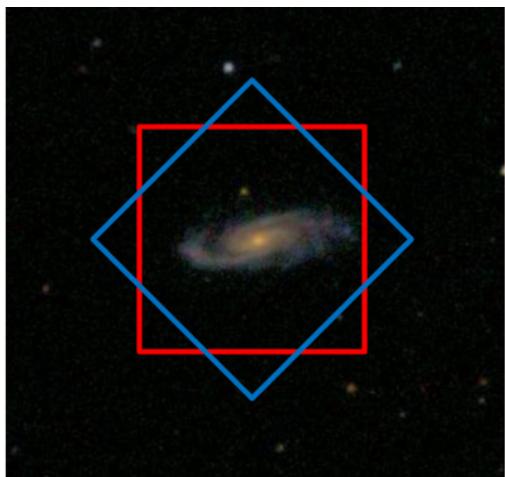


**REMEMBER
THE HoG ?**

(a) red channel



(b) green channel

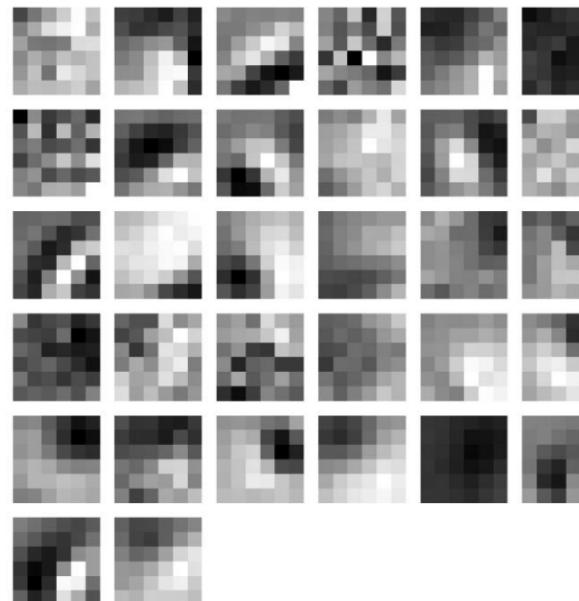


THESE ARE CALLED **FEATURE MAPS**

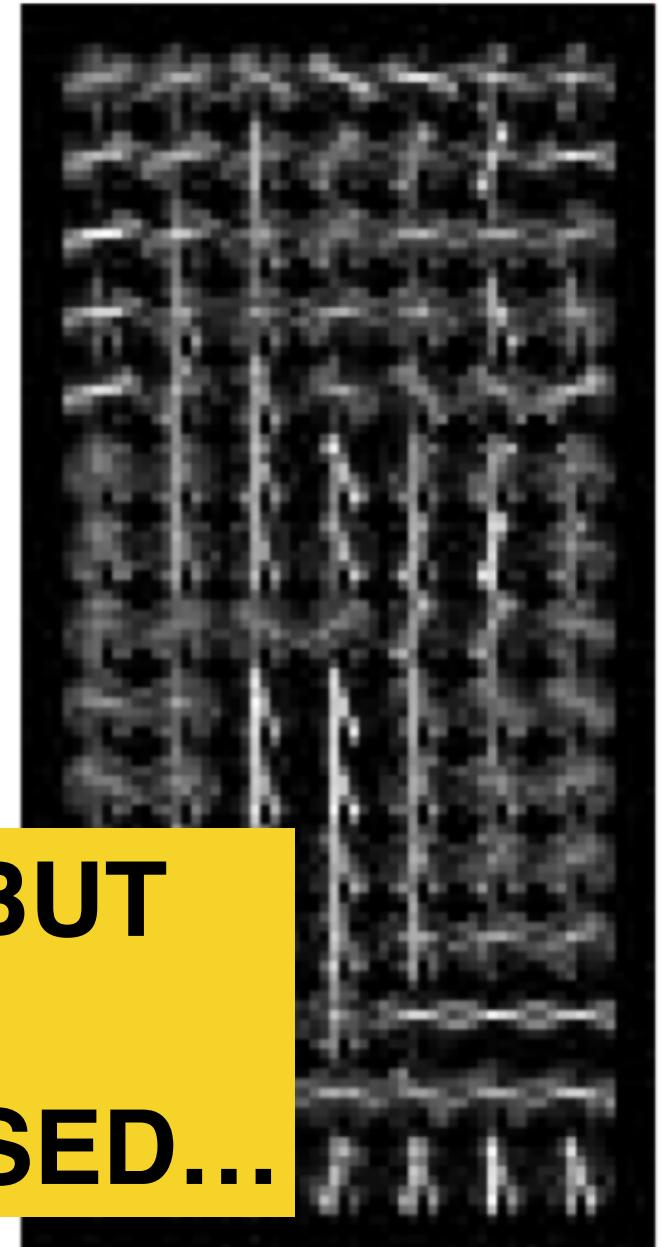
EXAMPLE OF 32 FILTERS APPLIED TO A CONVOLUTIONAL



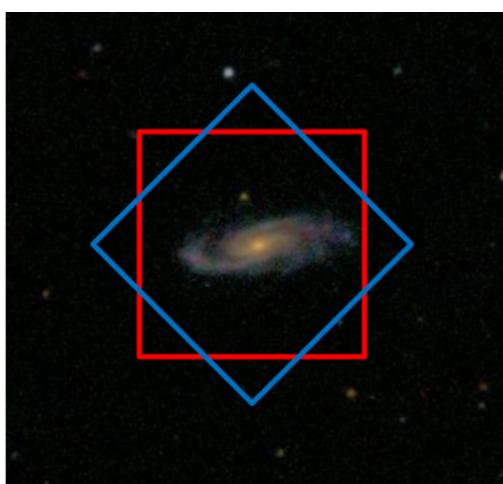
**REMEMBER
THE HoG ?**



**SIMILAR, BUT
FULLY
UNSUPERVISED...**



THESE ARE CALLED **FEATURE MAPS**



ESTIMATING SHAPES AND NUMBER OF PARAMETERS

KERNEL SHAPE:

$$(F, F, C^i, C^o)$$

PADDING:

$$P$$

STRIDES:

$$S$$

OUTPUT SIZE:

$$W_0 = (W^i - F + 2P)/S + 1$$

ESTIMATING SHAPES AND NUMBER OF PARAMETERS

KERNEL SHAPE:

$$(F, F, C^i, C^o)$$

PADDING:

$$P$$

STRIDES:

$$S$$

OUTPUT SIZE:

$$W_0 = (W^i - F + 2P)/S + 1$$

NUMBER OF PARAMETERS:

$$(F \times F \times C^i + 1) \times C^o$$

ESTIMATING SHAPES AND NUMBER OF PARAMETERS

KERNEL SHAPE:

$$(F, F, C^i, C^o)$$

PADDING:

$$P$$

STRIDES:

$$S$$

OUTPUT SIZE:

$$W_0 = (W^i - F + 2P)/S + 1$$

NUMBER OF PARAMETERS:

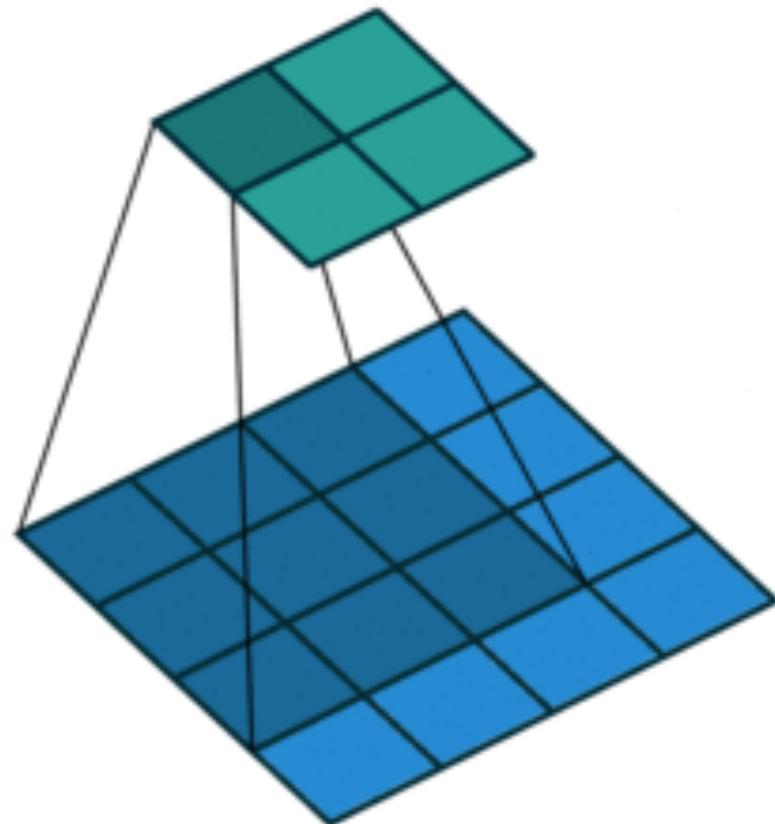
$$(F \times F \times C^i + 1) \times C^o$$



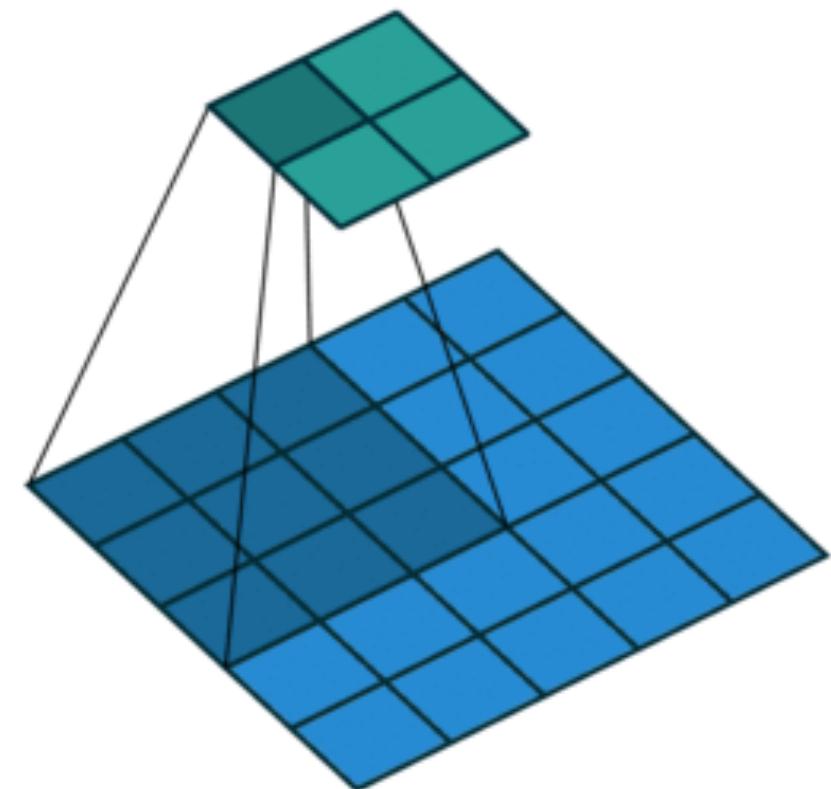
the number of parameters increases fast!

32 filters of 5*5 on a color image —> 2432 parameters to learn

OPTIONS: STRIDES

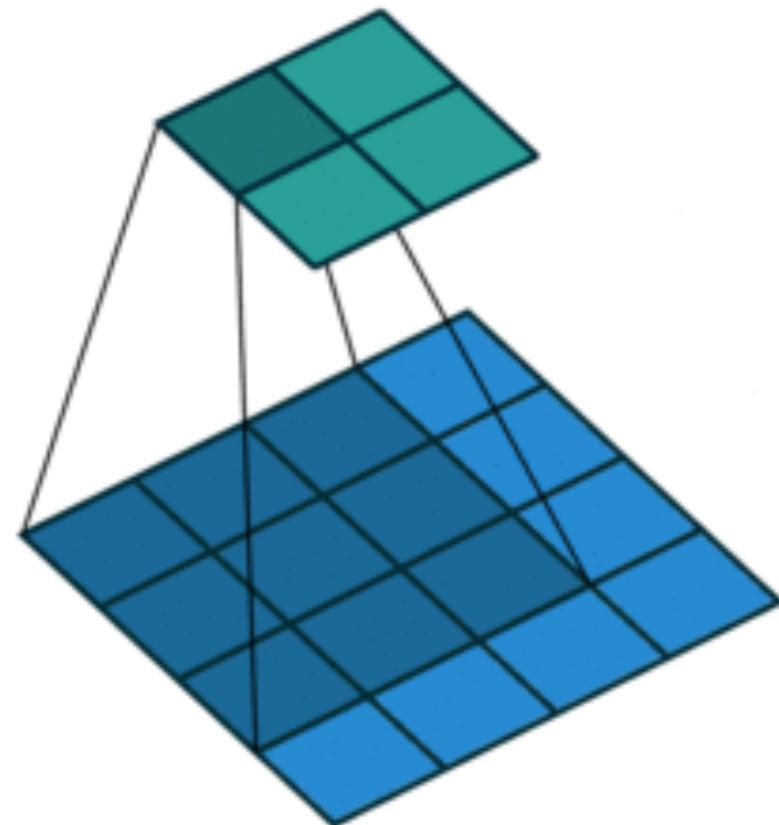


NO STRIDES

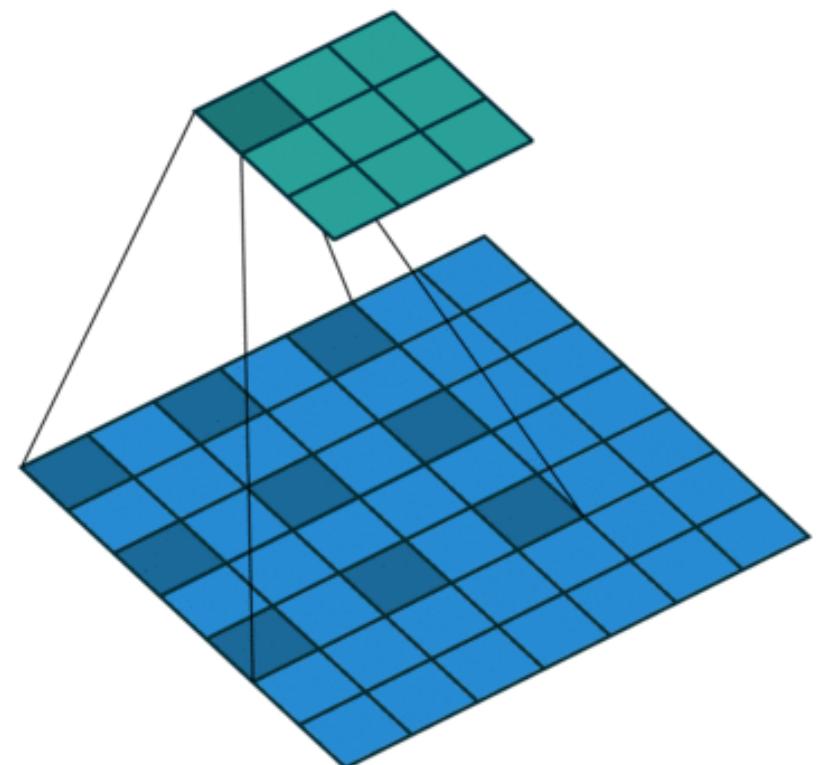


STRIDES

OPTIONS: DILATION

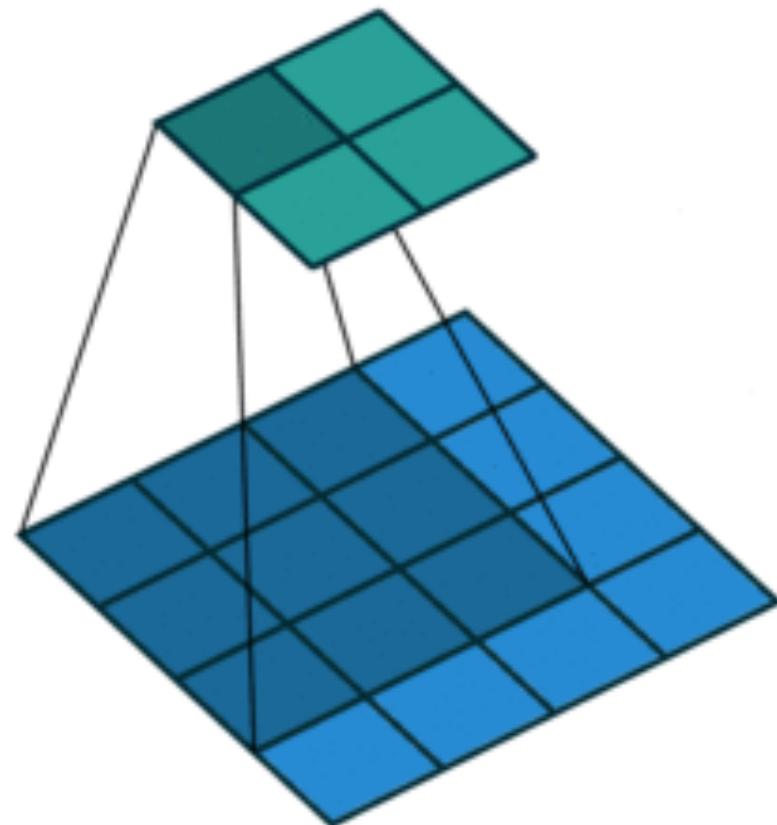


NO STRIDES

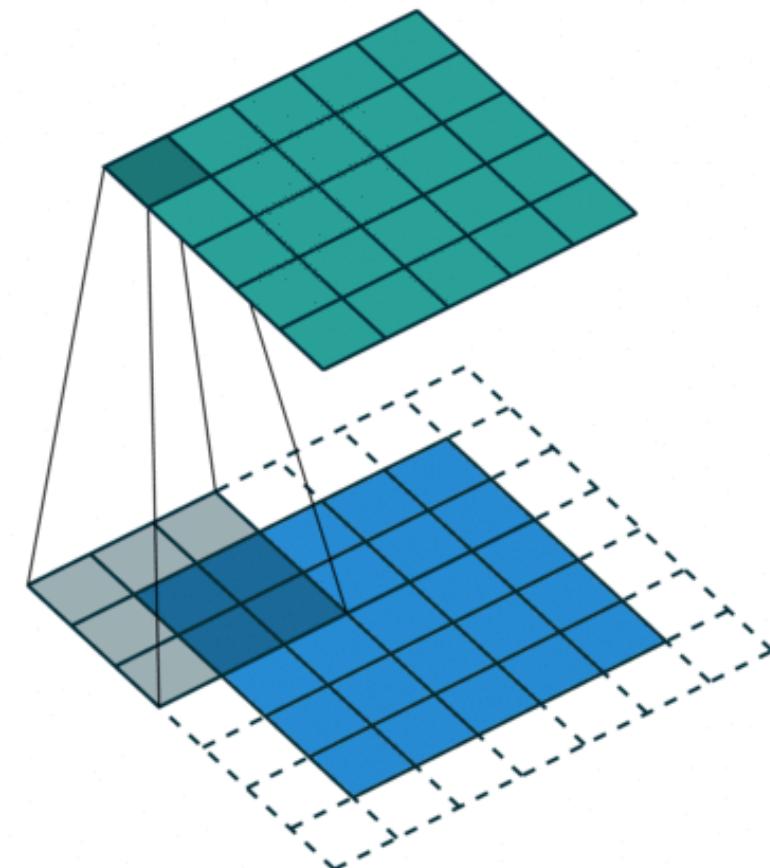


DILATION

OPTIONS: PADDING



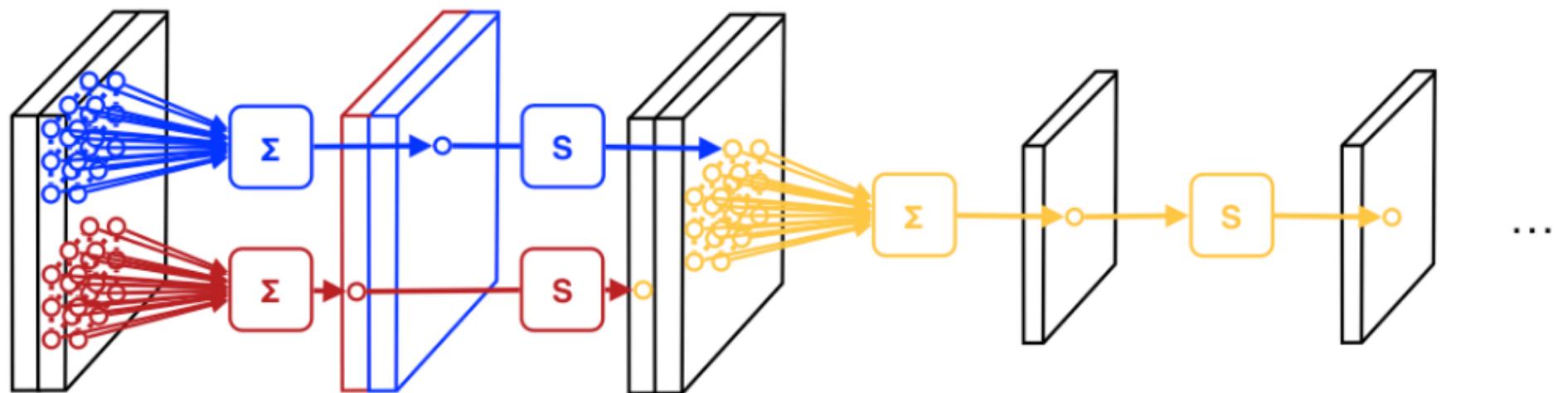
NO STRIDES



PADDING

DOWNSAMPLING

DOWNSAMPLING IS APPLIED TO REDUCE THE OVERALL SIZE OF TENSORS



POOLING

CONVOLUTIONS ARE OFTEN FOLLOWED BY AN OPERATION OF DOWNSAMPLING [POOLING]

VERY SIMPLE OPERATION - ONLY ONE OUT OF EVERY N PIXELS ARE KEPT

OFTEN MATCHED WITH AN INCREASE OF THE FEATURE CHANNELS

TYPES OF POOLING

SUM POOLING

$$y = \sum x_{uv}$$

SQUARE SUM POOLING

$$y = \sqrt{\sum x_{uv}^2}$$

MAX POOLING

$$y = \max(x_{uv})$$

TYPES OF POOLING

SUM POOLING

$$y = \sum x_{uv}$$

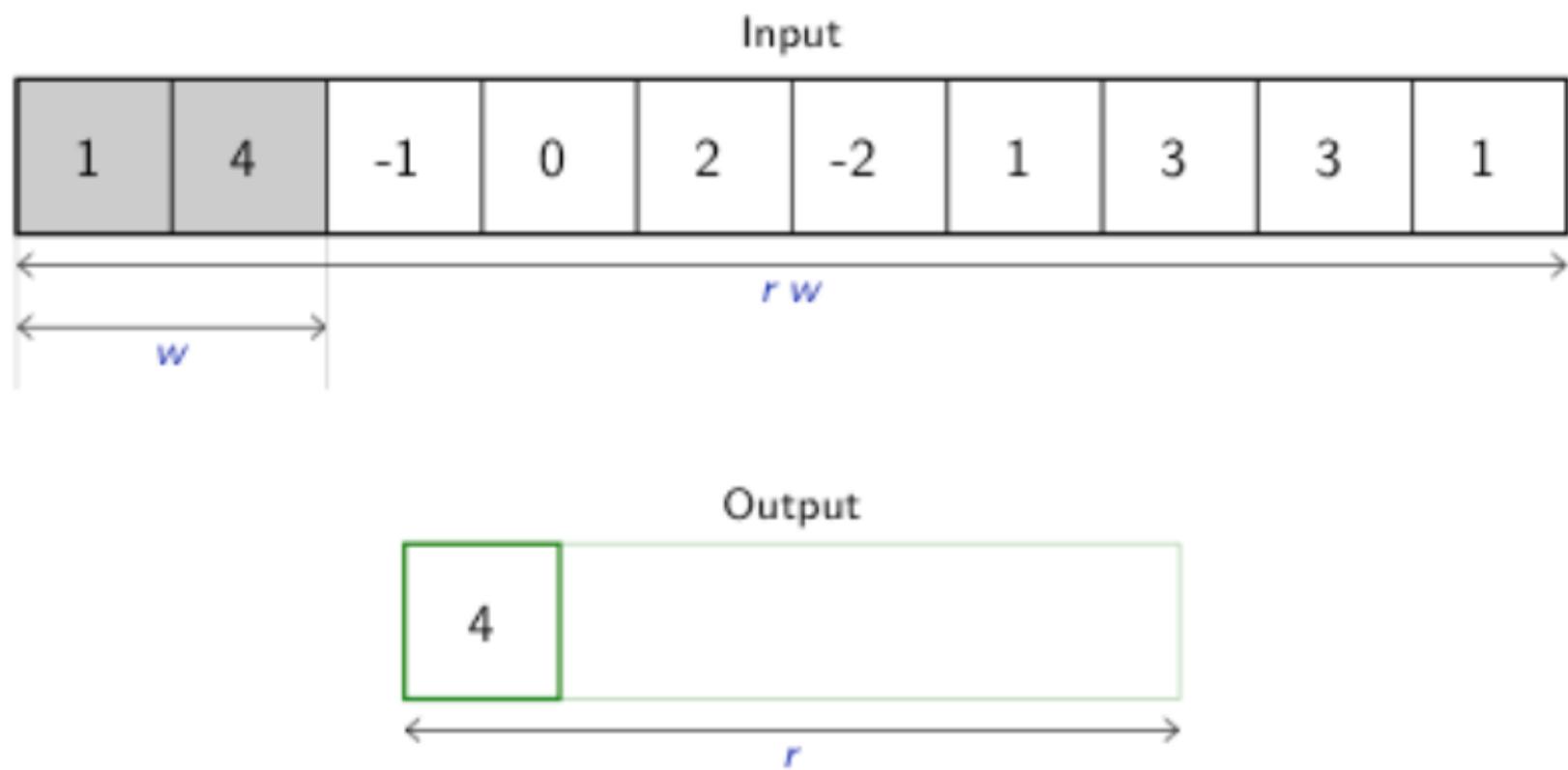
SQUARE SUM POOLING

$$y = \sqrt{\sum x_{uv}^2}$$

MAX POOLING

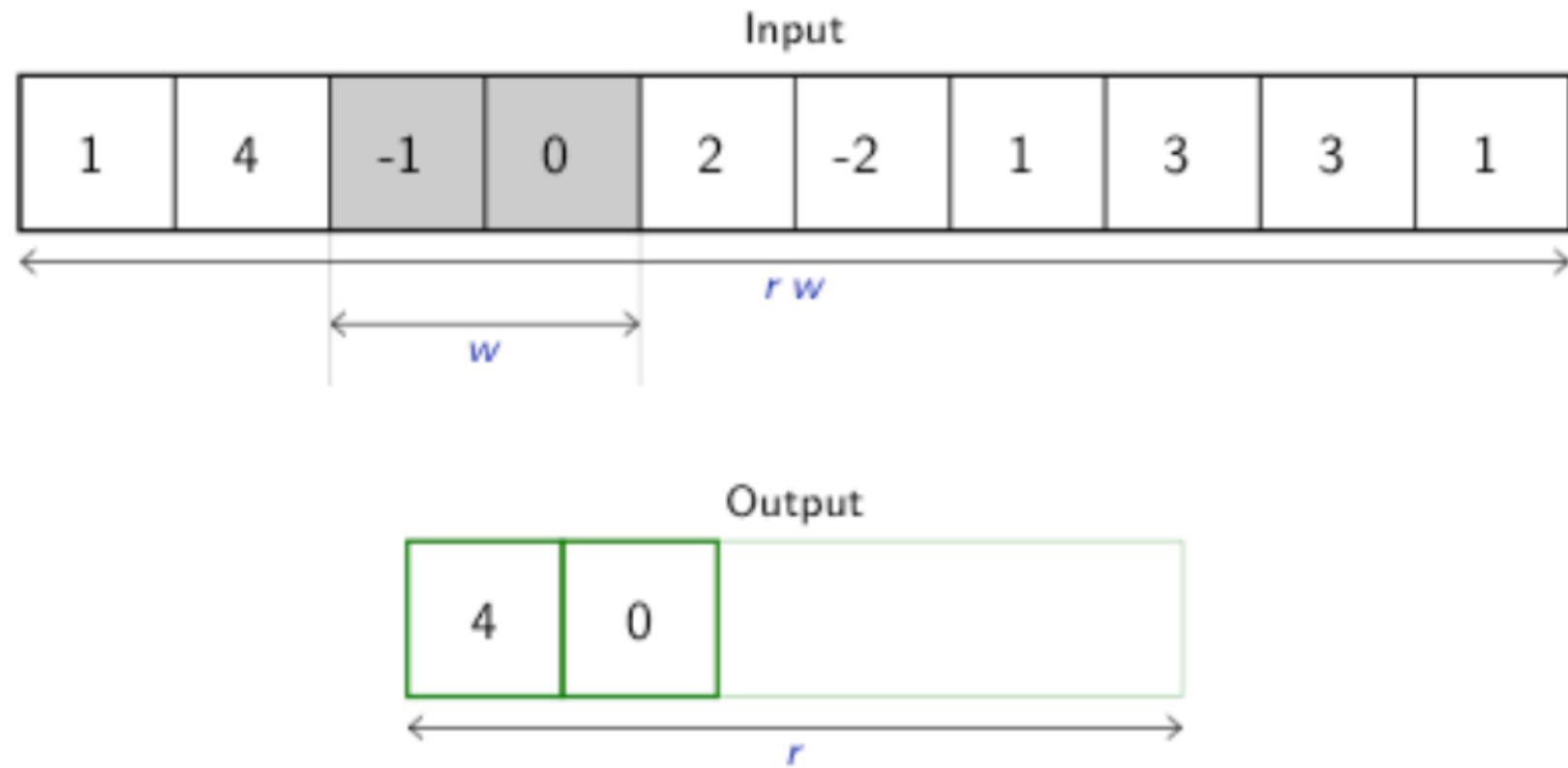
$$y = \max(x_{uv})$$

MAX POOLING 1D



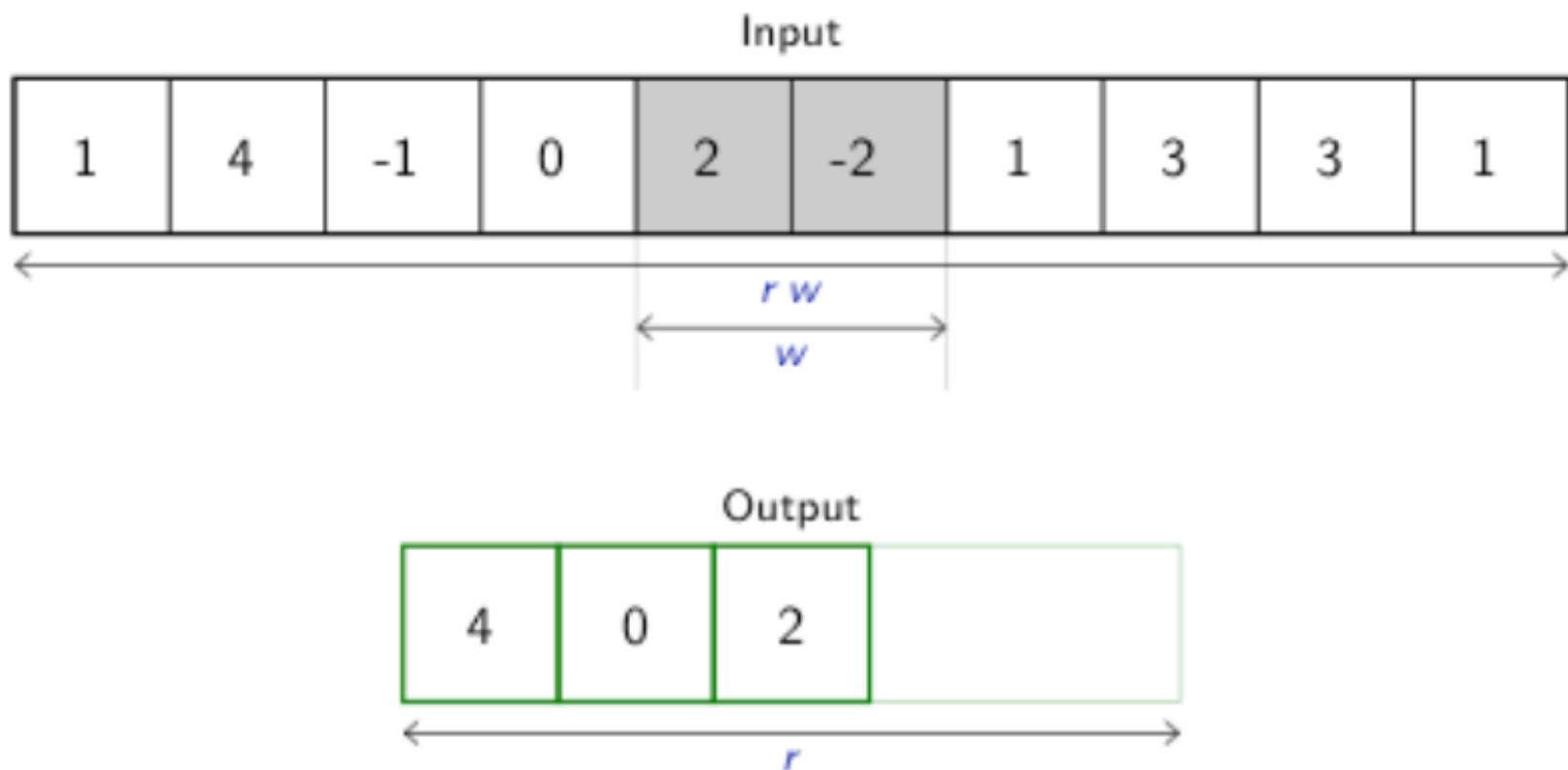
Credit: F. Fleuret

MAX POOLING 1D



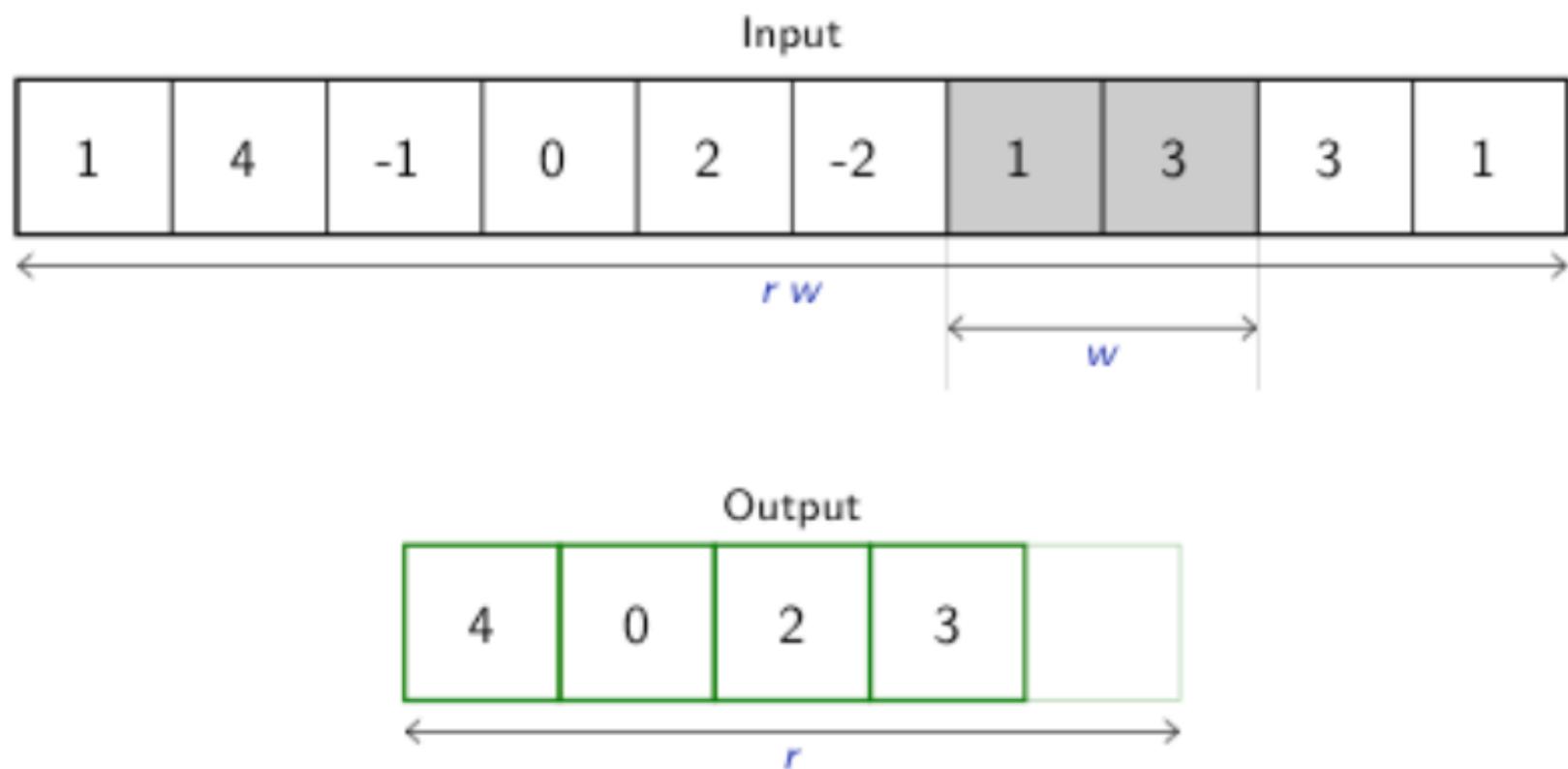
Credit: F. Fleuret

MAX POOLING 1D



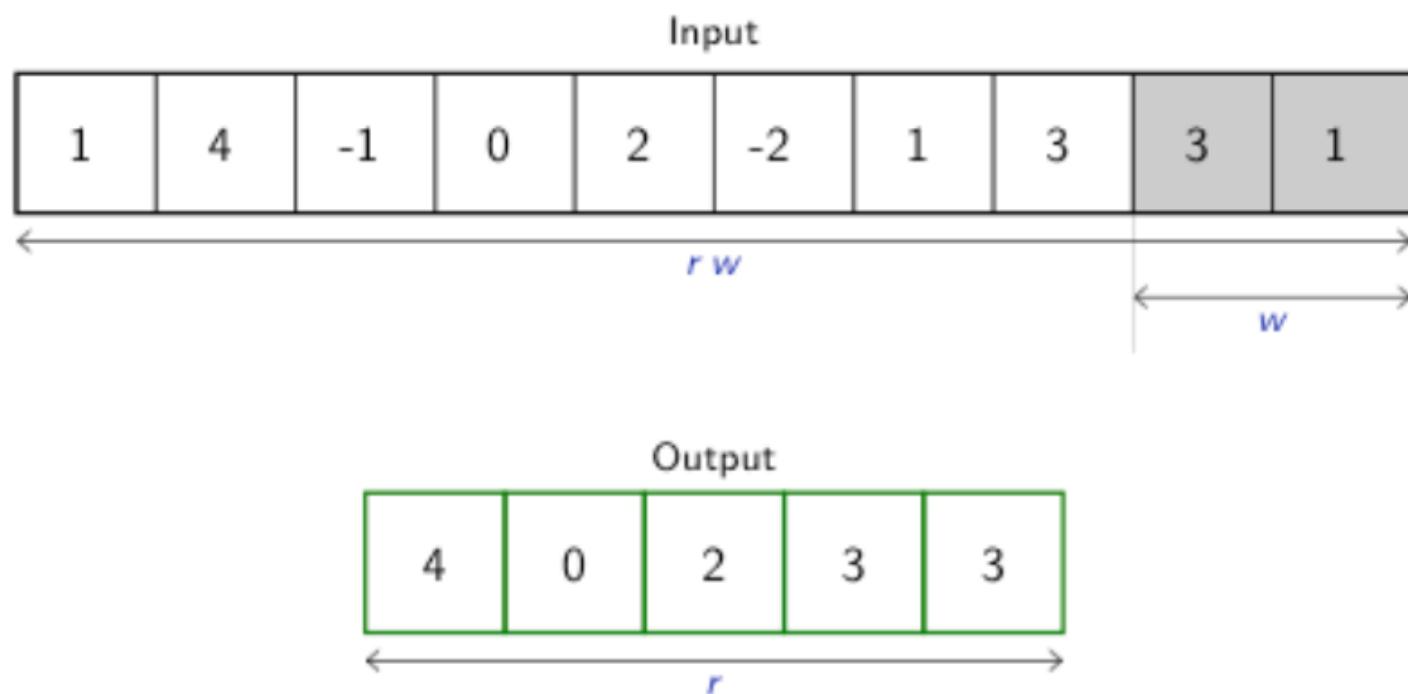
Credit: F. Fleuret

MAX POOLING 1D



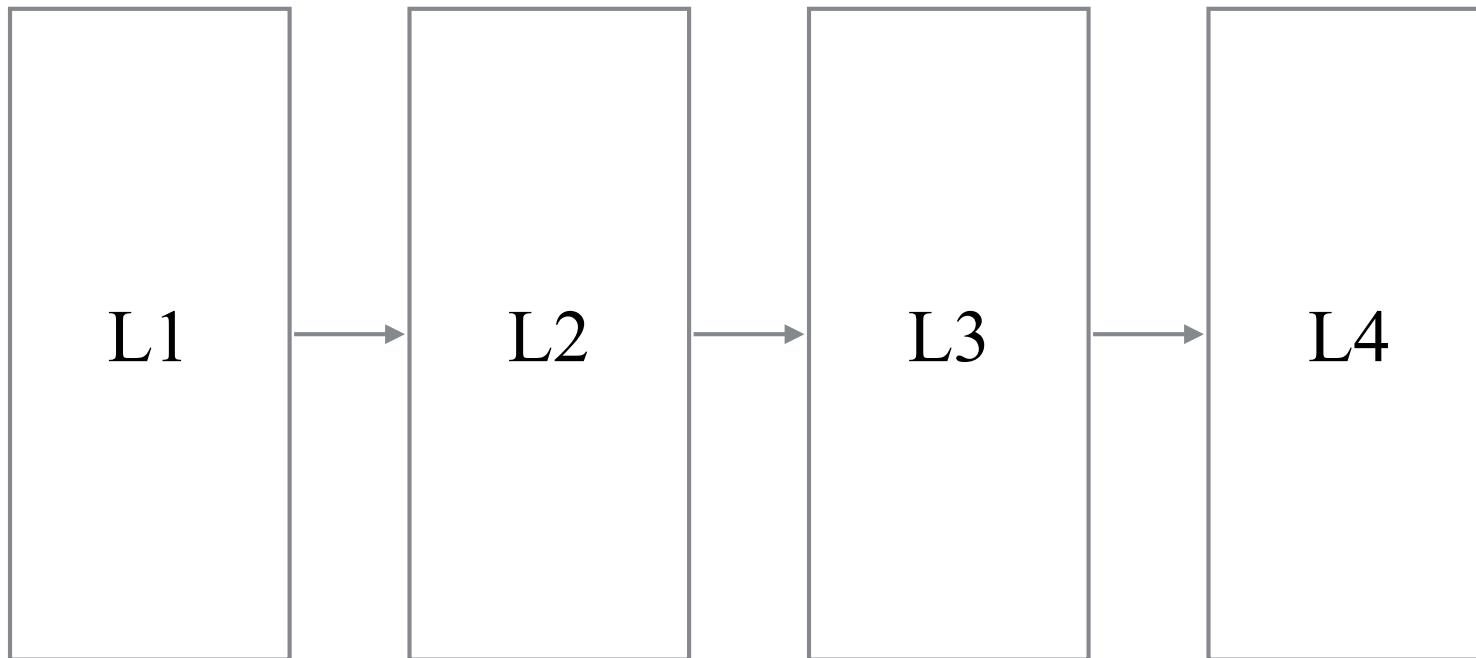
Credit: F. Fleuret

MAX POOLING 1D



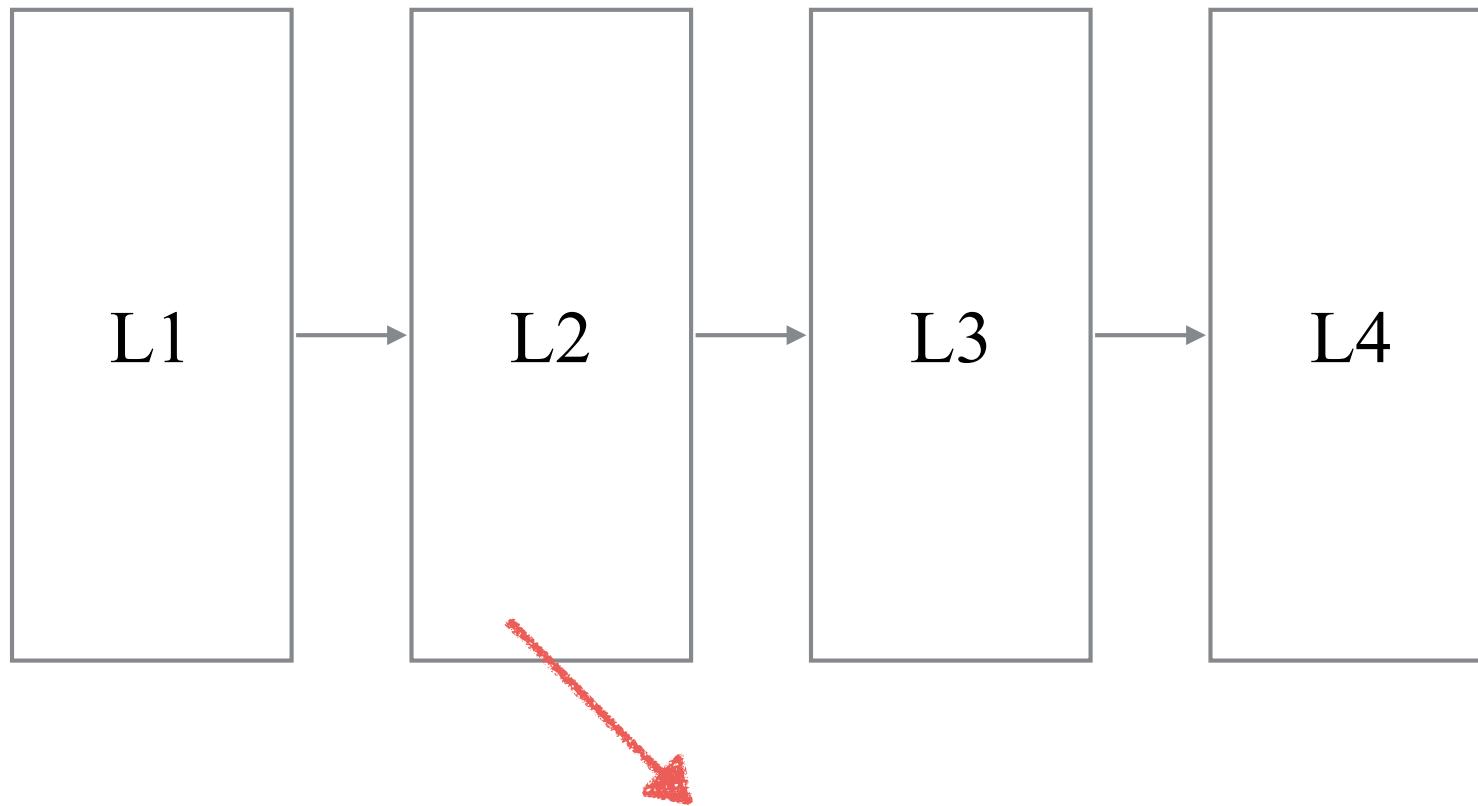
Credit: F. Fleuret

CONVNET OR CNN



A CONCATENATION OF MULTIPLE
CONVOLUTIONAL BLOCKS

CONVNET OR CNN



EACH BLOCK TYPICALLY MADE OF:

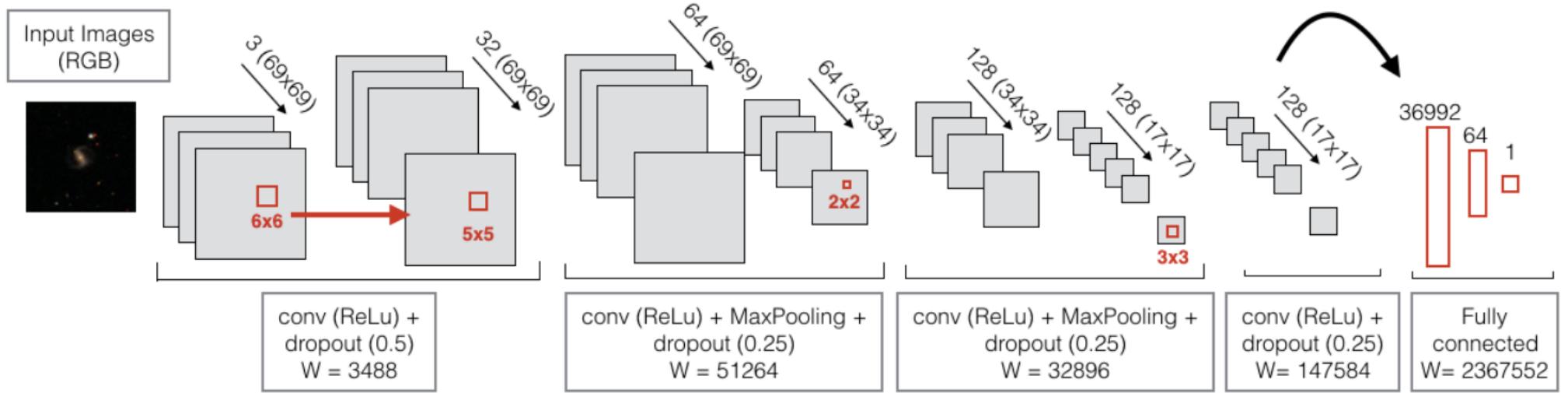
CONV

ACTIVATION

POOLING

(+dropout
for training)

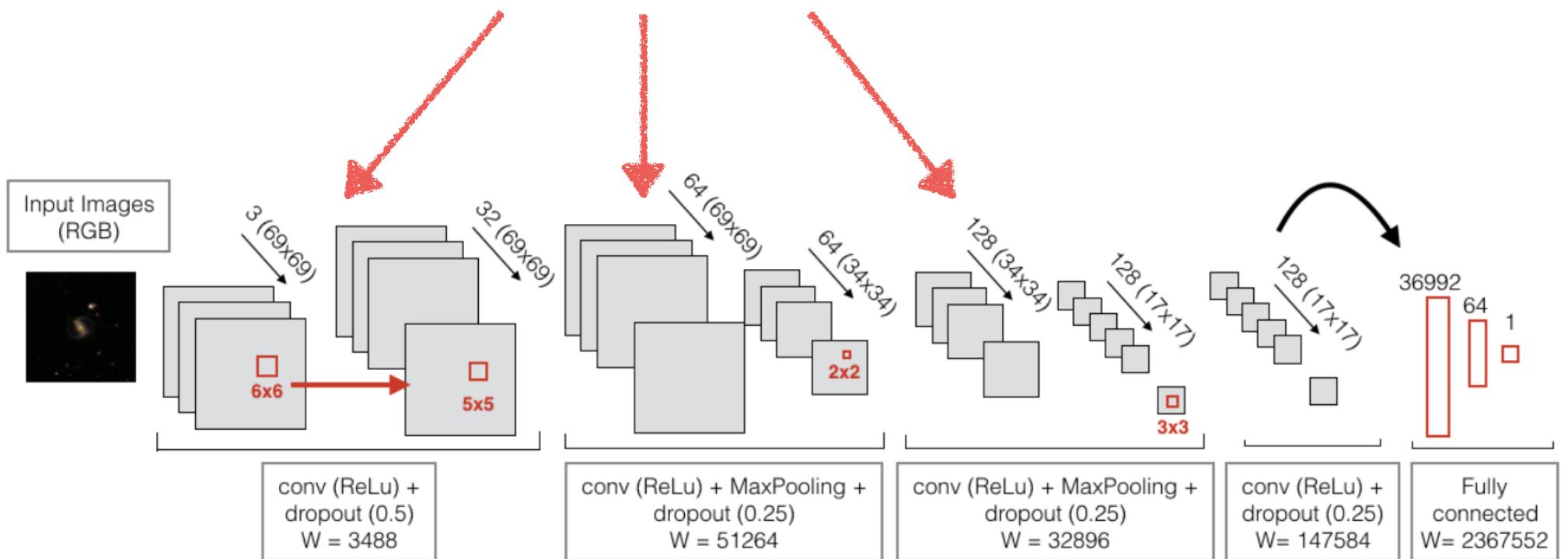
EXAMPLE OF VERY SIMPLE CNN



Dominguez-Sanchez+18

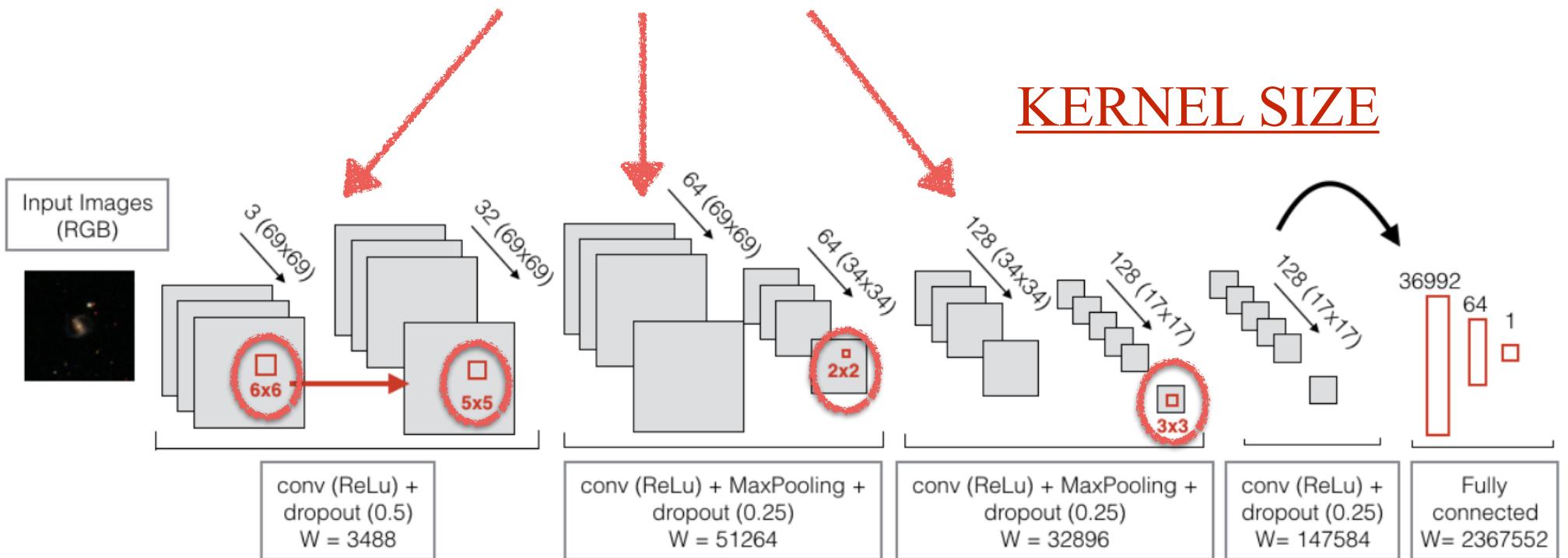
EXAMPLE OF VERY SIMPLE CNN

3 convolutional layers



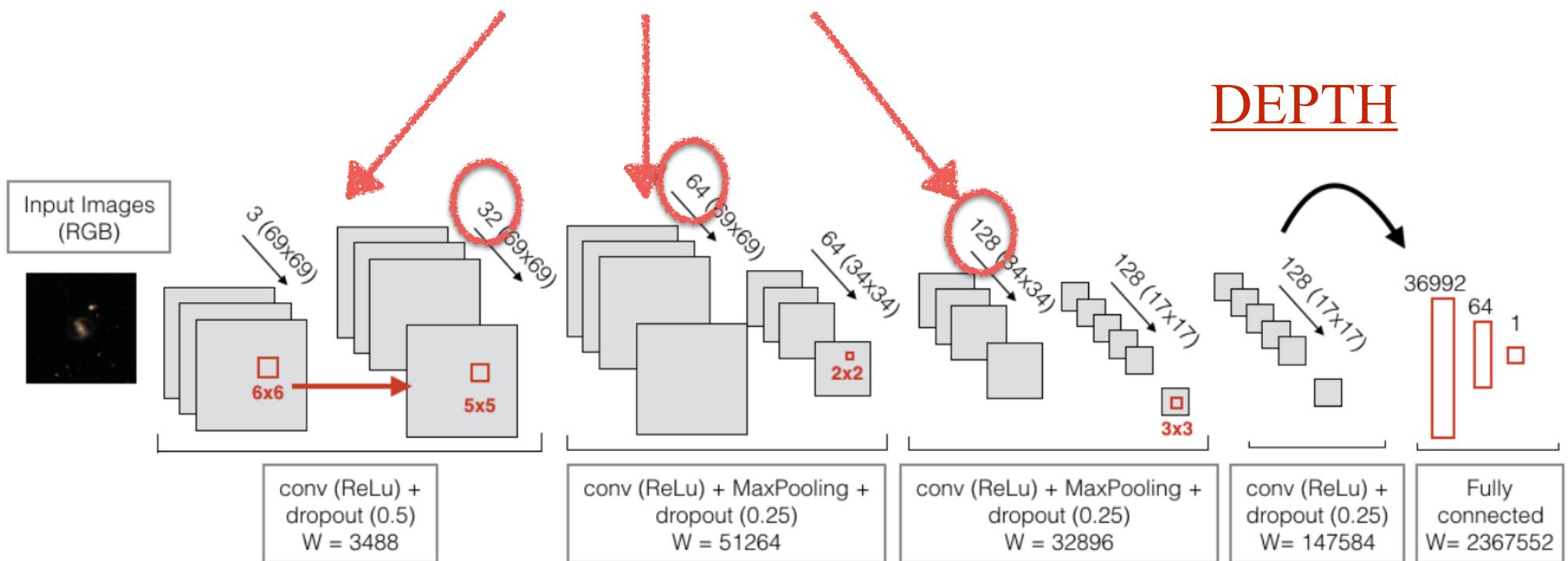
EXAMPLE OF VERY SIMPLE CNN

3 convolutional layers



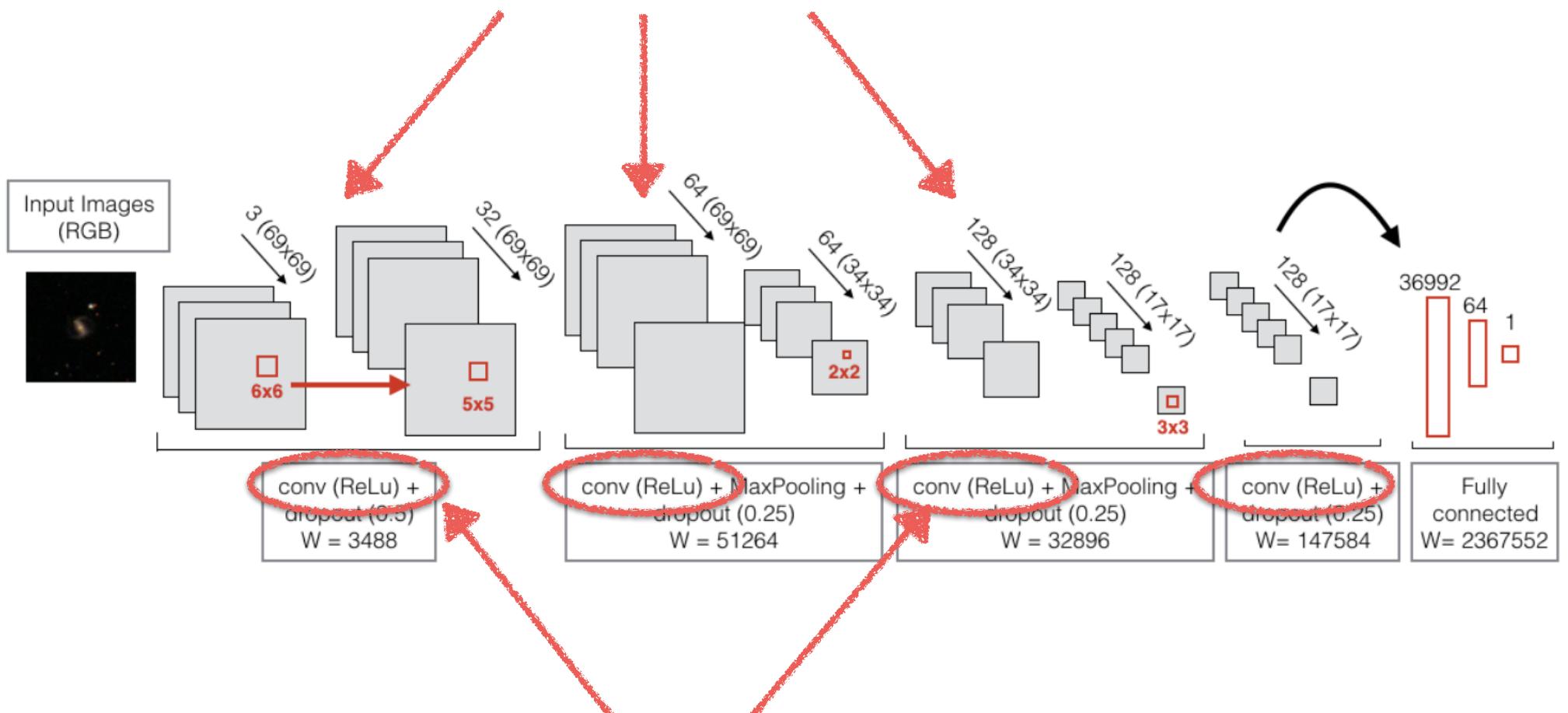
EXAMPLE OF VERY SIMPLE CNN

3 convolutional layers



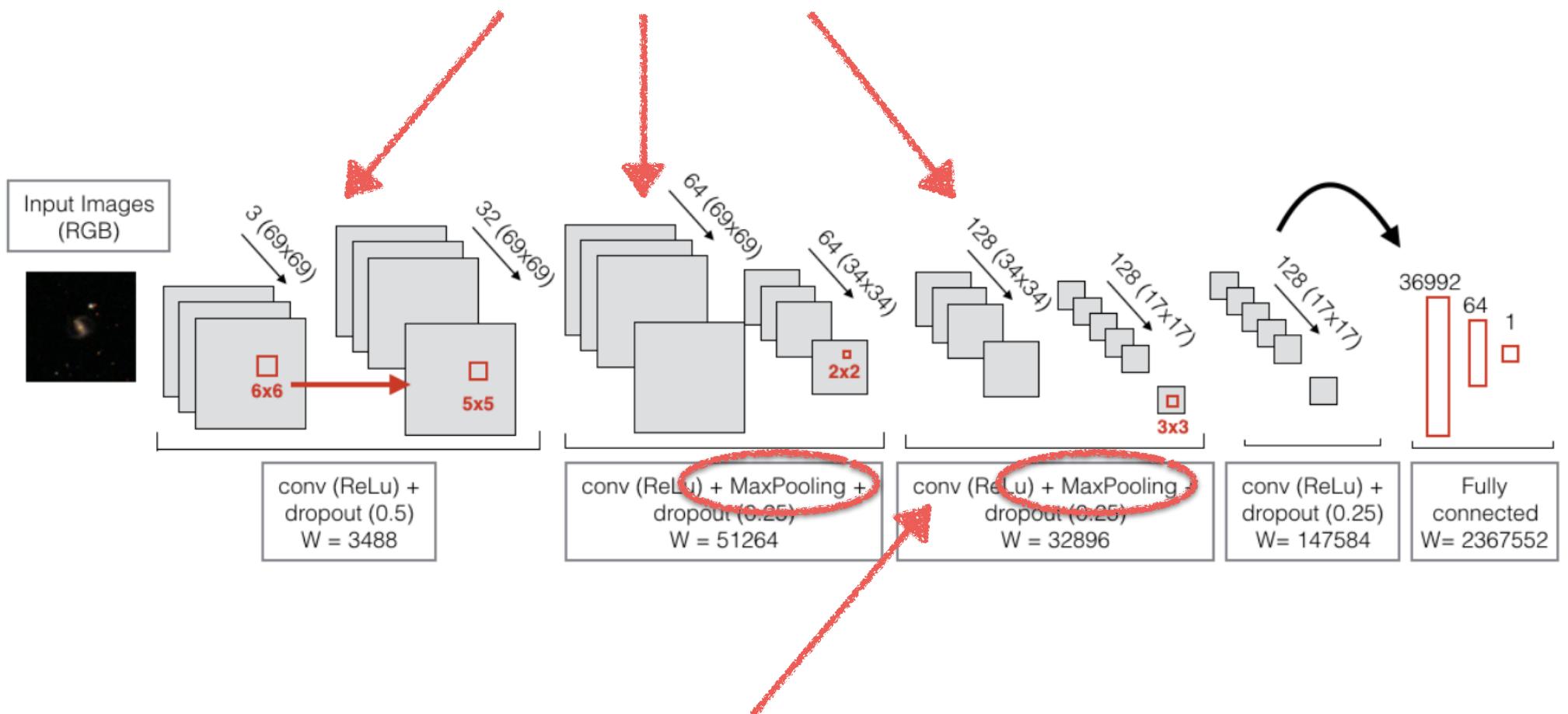
EXAMPLE OF VERY SIMPLE CNN

3 convolutional layers



EXAMPLE OF VERY SIMPLE CNN

3 convolutional layers



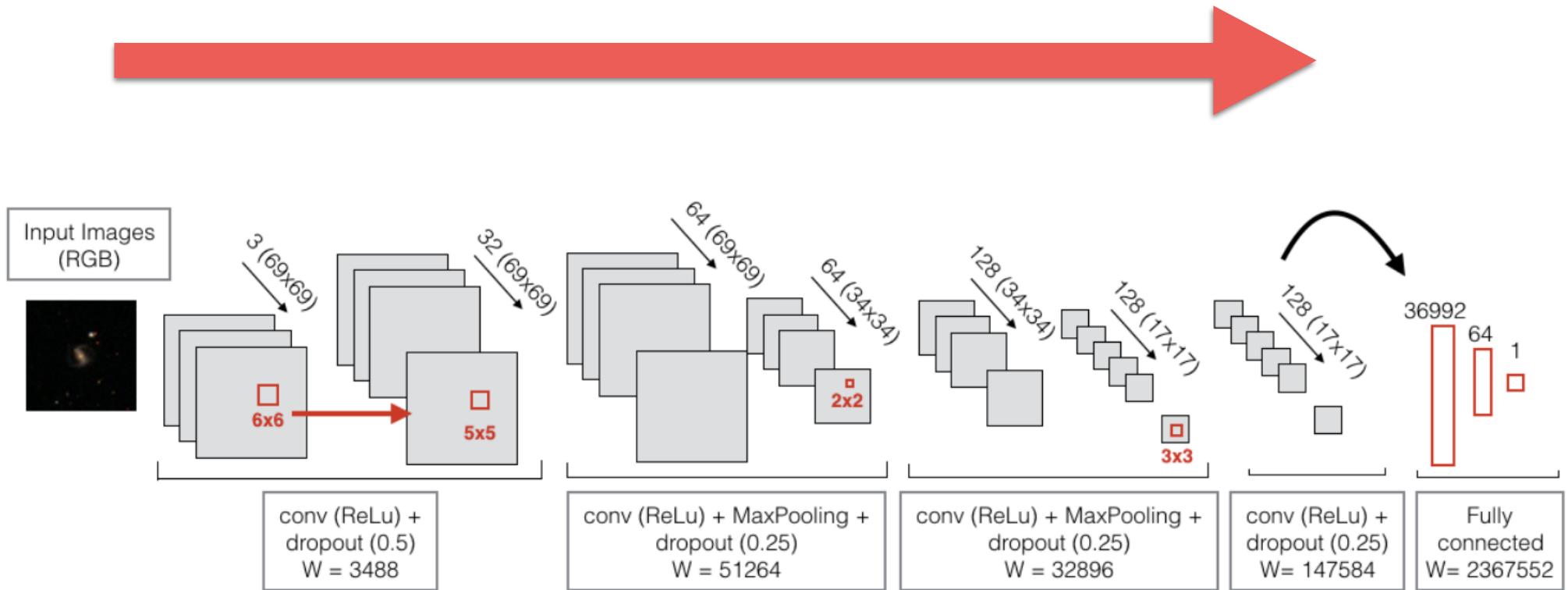
Pooling

Dominguez-Sanchez+18

EXAMPLE OF VERY SIMPLE CNN

OVERALL:

- decrease of tensor size
- increase of depth



IMPLEMENTATION IN KERAS

```
#===== Model definition=====

#Convolutional Layers

model = Sequential()
model.add(Convolution2D(32, 6,6, border_mode='same',
                       input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Convolution2D(64, 5, 5, border_mode='same'))
model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(128, 2, 2, border_mode='same'))
model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

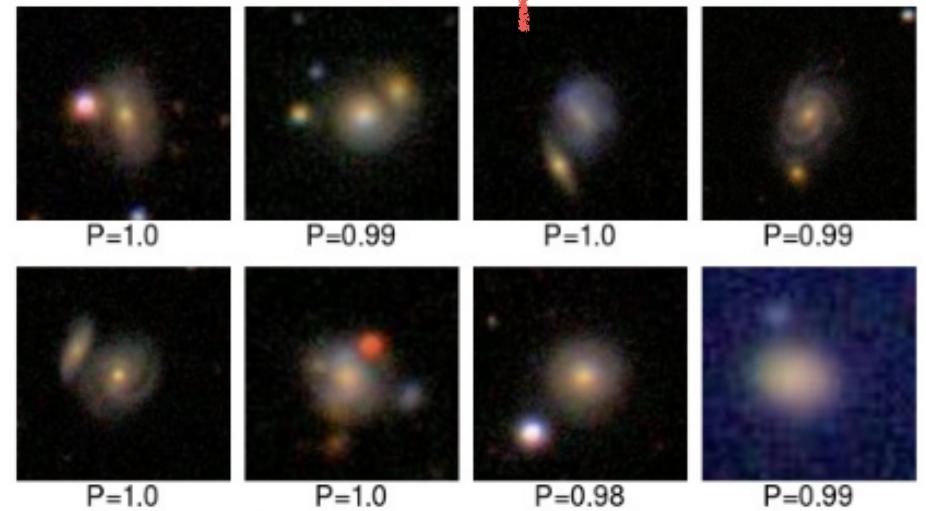
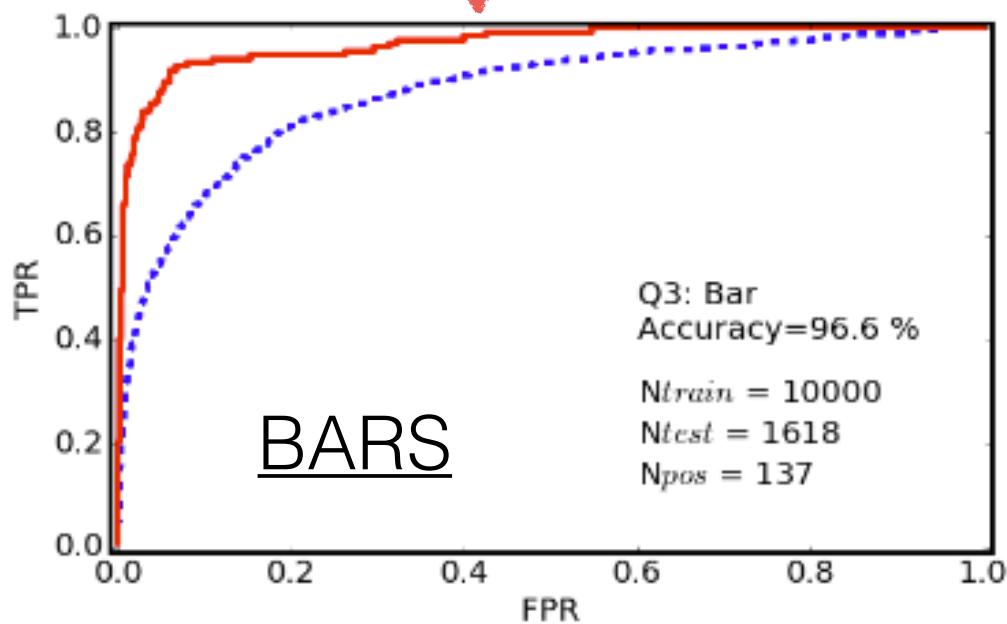
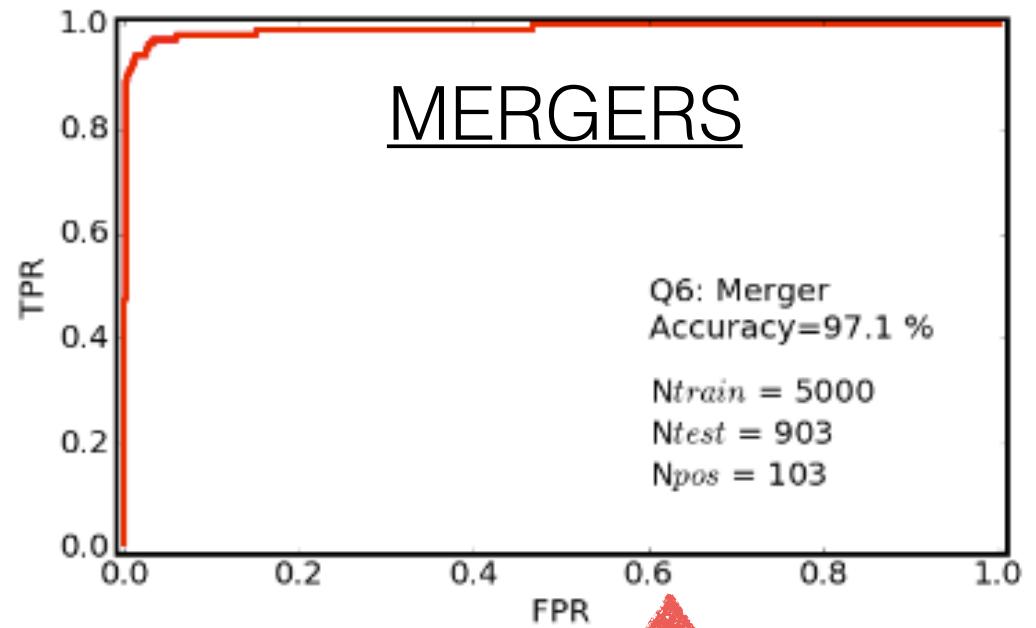
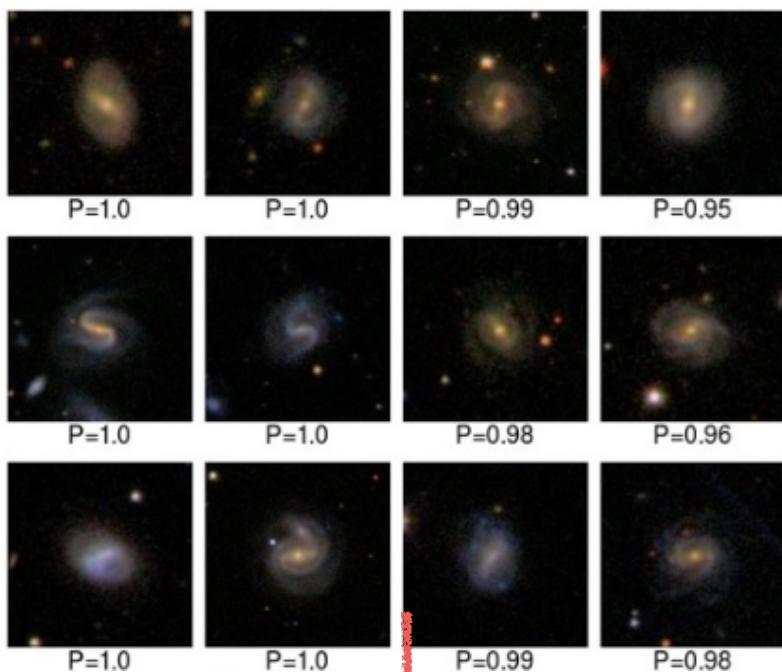
model.add(Convolution2D(128, 3, 3, border_mode='same'))
model.add(Activation('relu'))

model.add(Dropout(0.25))

#Fully Connected start here
#-----#
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(.5))
model.add(Dense(1, init='uniform', activation='sigmoid'))

print("Compilation...")

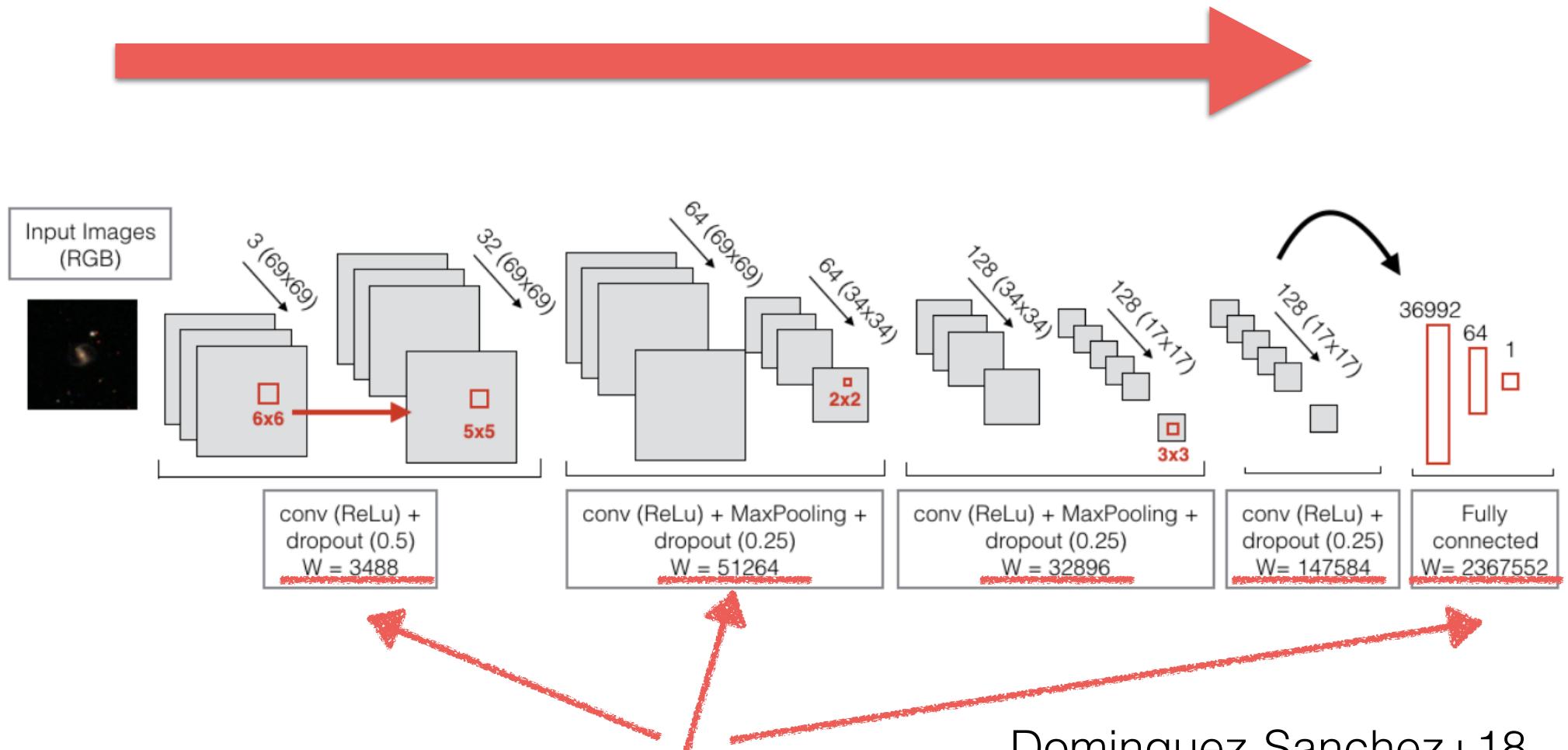
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
```



EXAMPLE OF VERY SIMPLE CNN

OVERALL:

- decrease of tensor size
- increase of depth



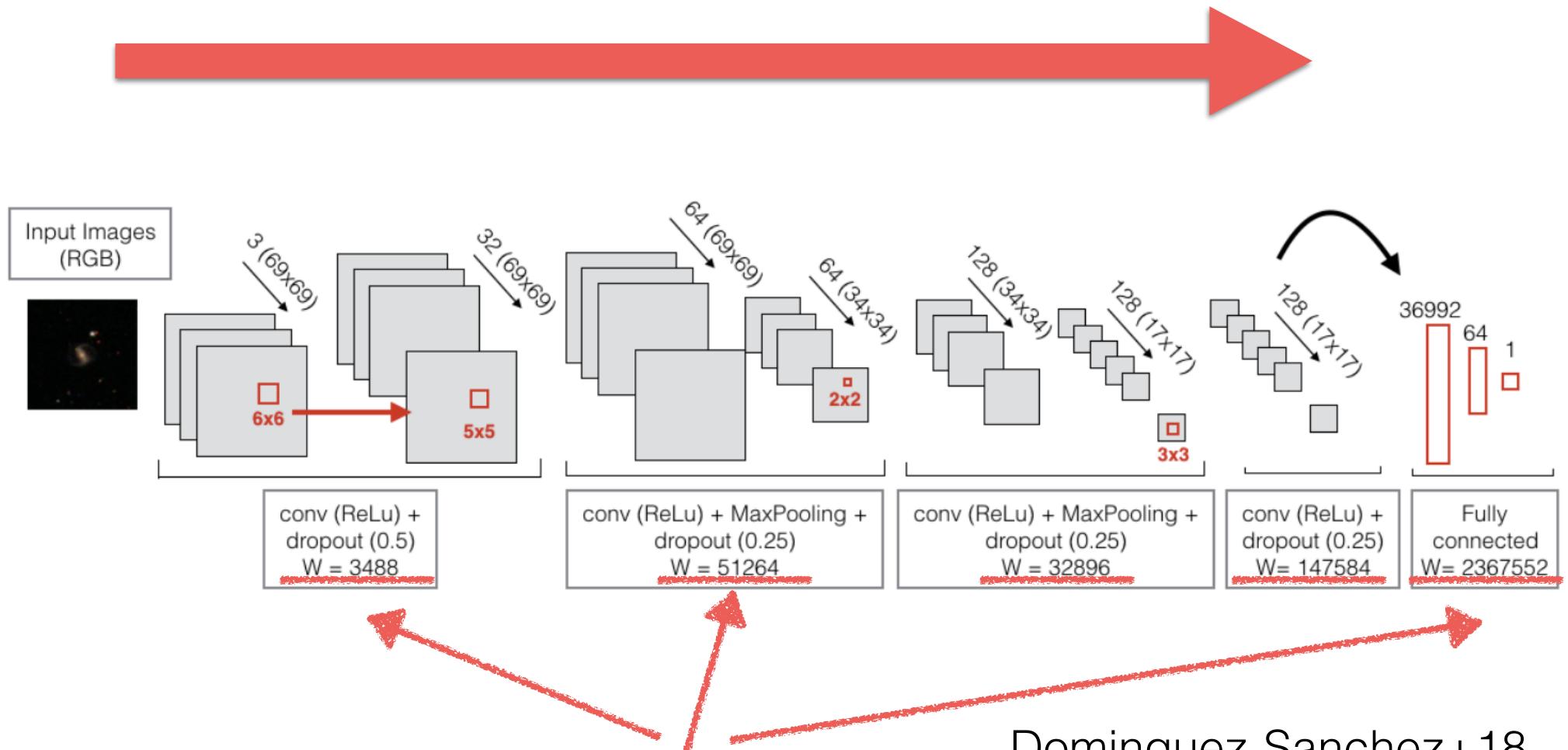
Number of parameters

Dominguez-Sanchez+18

EXAMPLE OF VERY SIMPLE CNN

OVERALL:

- decrease of tensor size
- increase of depth



2 million of parameters for this very simple network!

CHECKING THE NUMBER OF PARAMETERS / LAYERS WITH KERAS

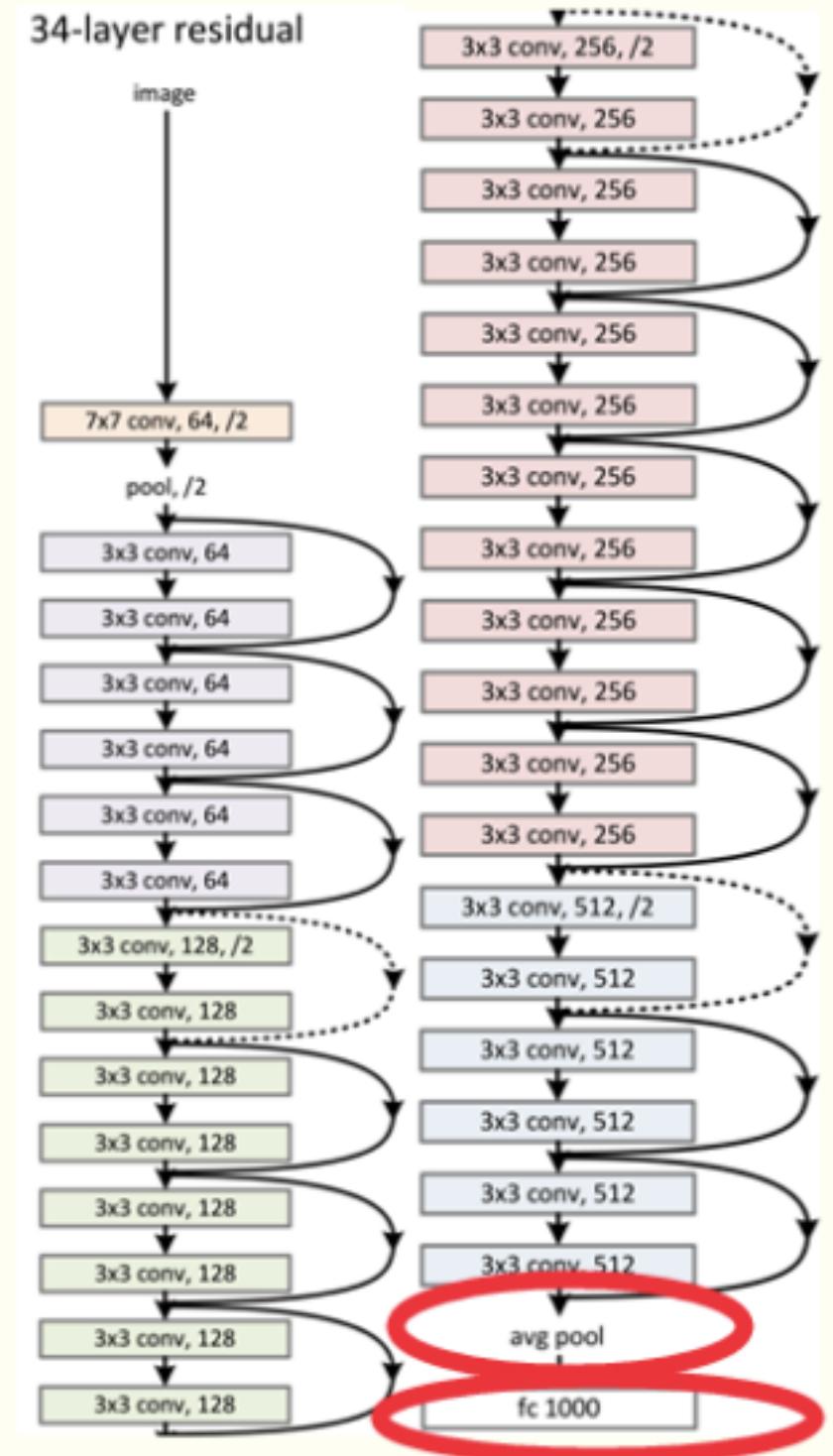
model.summary()



| Layer (type) | Output Shape | Param # |
|---|--------------------------|---------|
| <hr/> | | |
| input_1 (InputLayer) | (None, 1, 16, 112, 112) | 0 |
| conv3d_1 (Conv3D) | (None, 16, 16, 112, 112) | 448 |
| batch_normalization_1 (Batch Normalization) | (None, 16, 16, 112, 112) | 448 |
| activation_1 (Activation) | (None, 16, 16, 112, 112) | 0 |
| max_pooling3d_1 (MaxPooling3) | (None, 16, 8, 56, 56) | 0 |
| conv3d_2 (Conv3D) | (None, 32, 8, 56, 56) | 13856 |
| batch_normalization_2 (Batch Normalization) | (None, 32, 8, 56, 56) | 224 |
| activation_2 (Activation) | (None, 32, 8, 56, 56) | 0 |
| max_pooling3d_2 (MaxPooling3) | (None, 32, 4, 28, 28) | 0 |
| conv3d_3 (Conv3D) | (None, 64, 4, 28, 28) | 55360 |
| batch_normalization_3 (Batch Normalization) | (None, 64, 4, 28, 28) | 112 |
| activation_3 (Activation) | (None, 64, 4, 28, 28) | 0 |
| max_pooling3d_3 (MaxPooling3) | (None, 64, 2, 14, 14) | 0 |
| activation_12 (Activation) | (None, 64, 2, 14, 14) | 0 |
| <hr/> | | |
| Total params: 70,448 | | |
| Trainable params: 70,056 | | |
| Non-trainable params: 392 | | |

IN THE REAL LIFE...

RESNET



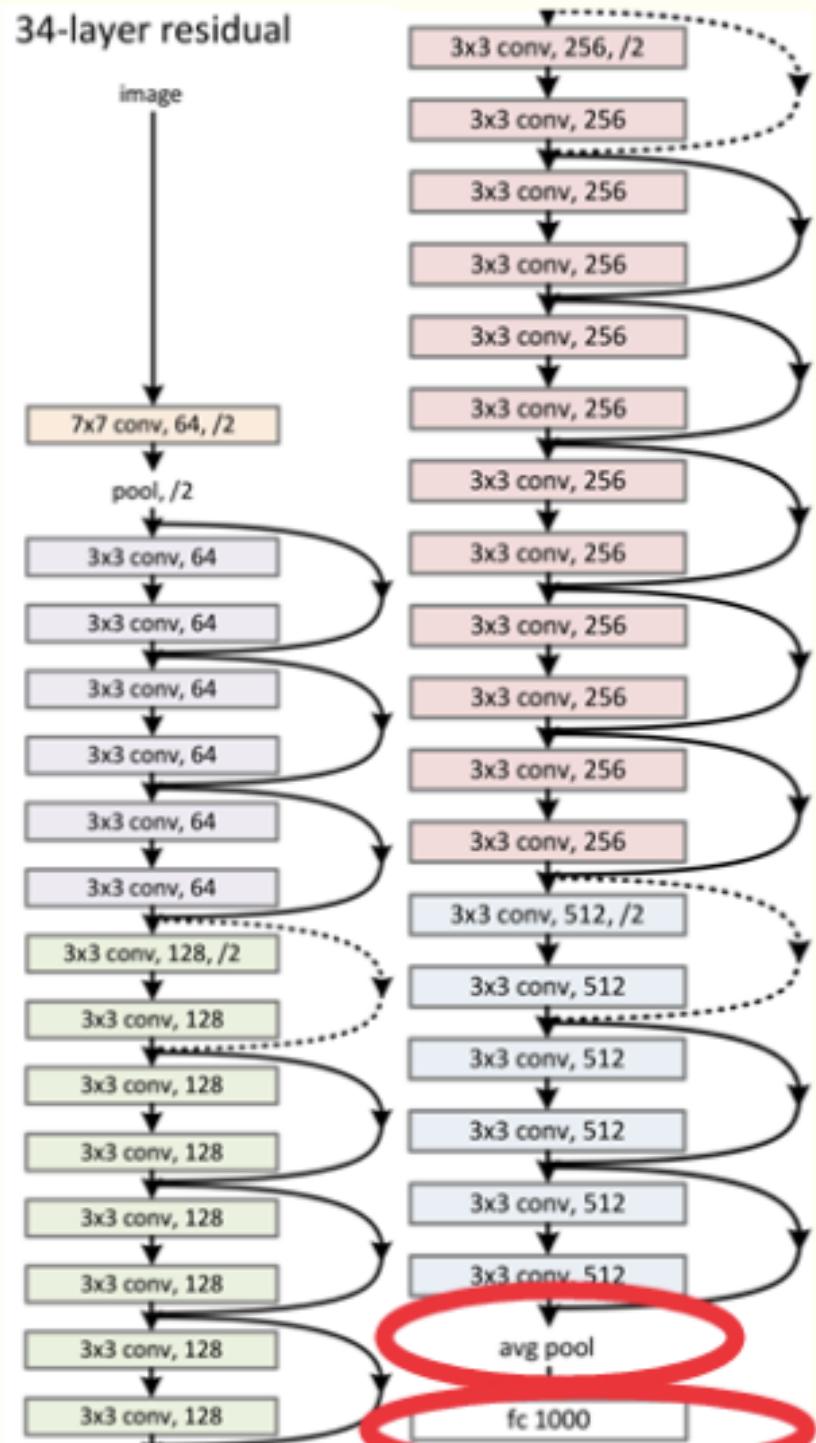
IN THE REAL LIFE...

RESNET

DO WE NEED TO GO
THIS DEEP FOR
ASTRONOMY
APPLICATIONS?

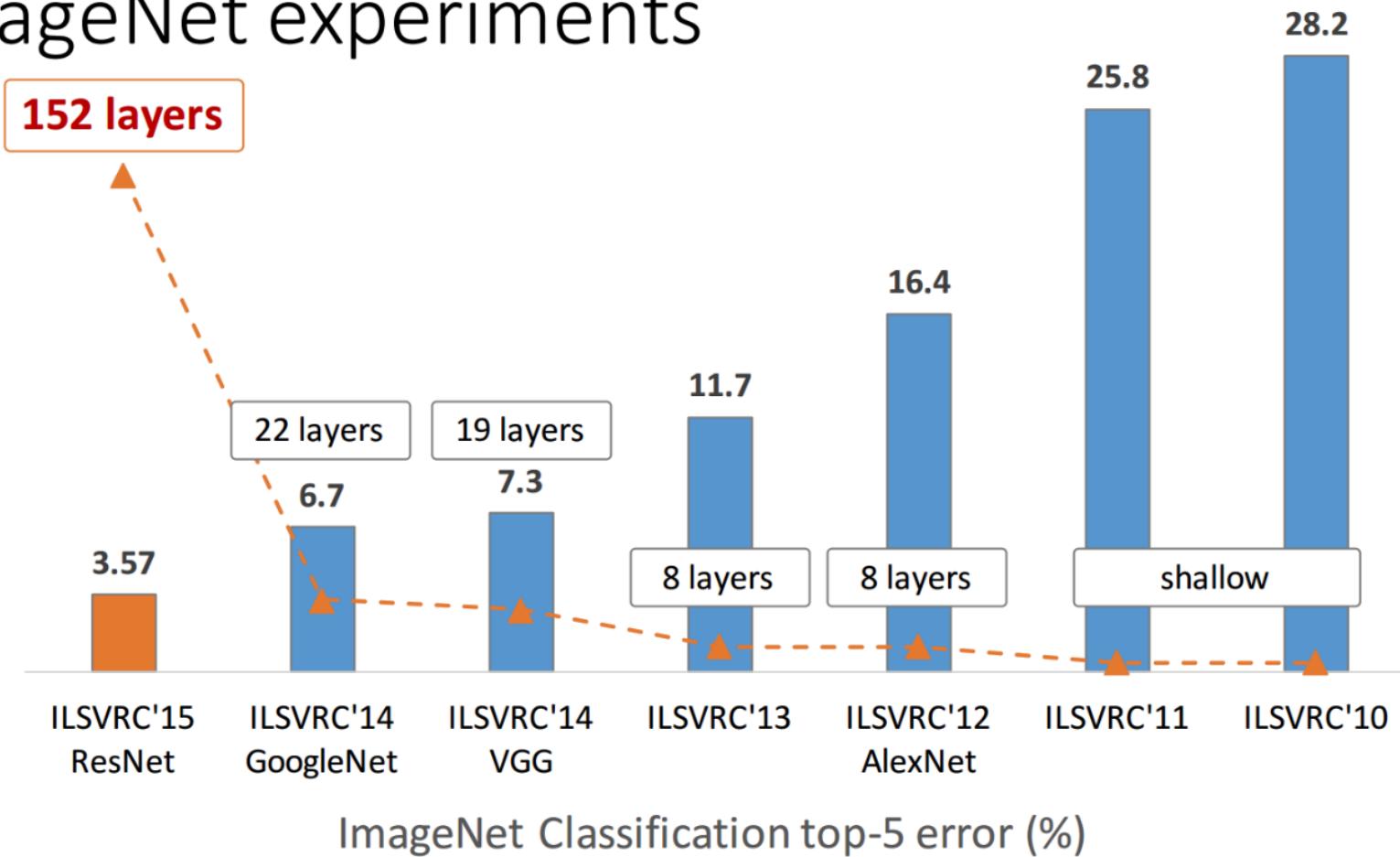
[34 layers - authors
explored up to 1202!]

34-layer residual



DEEPER TENDS TO BE BETTER...

ImageNet experiments



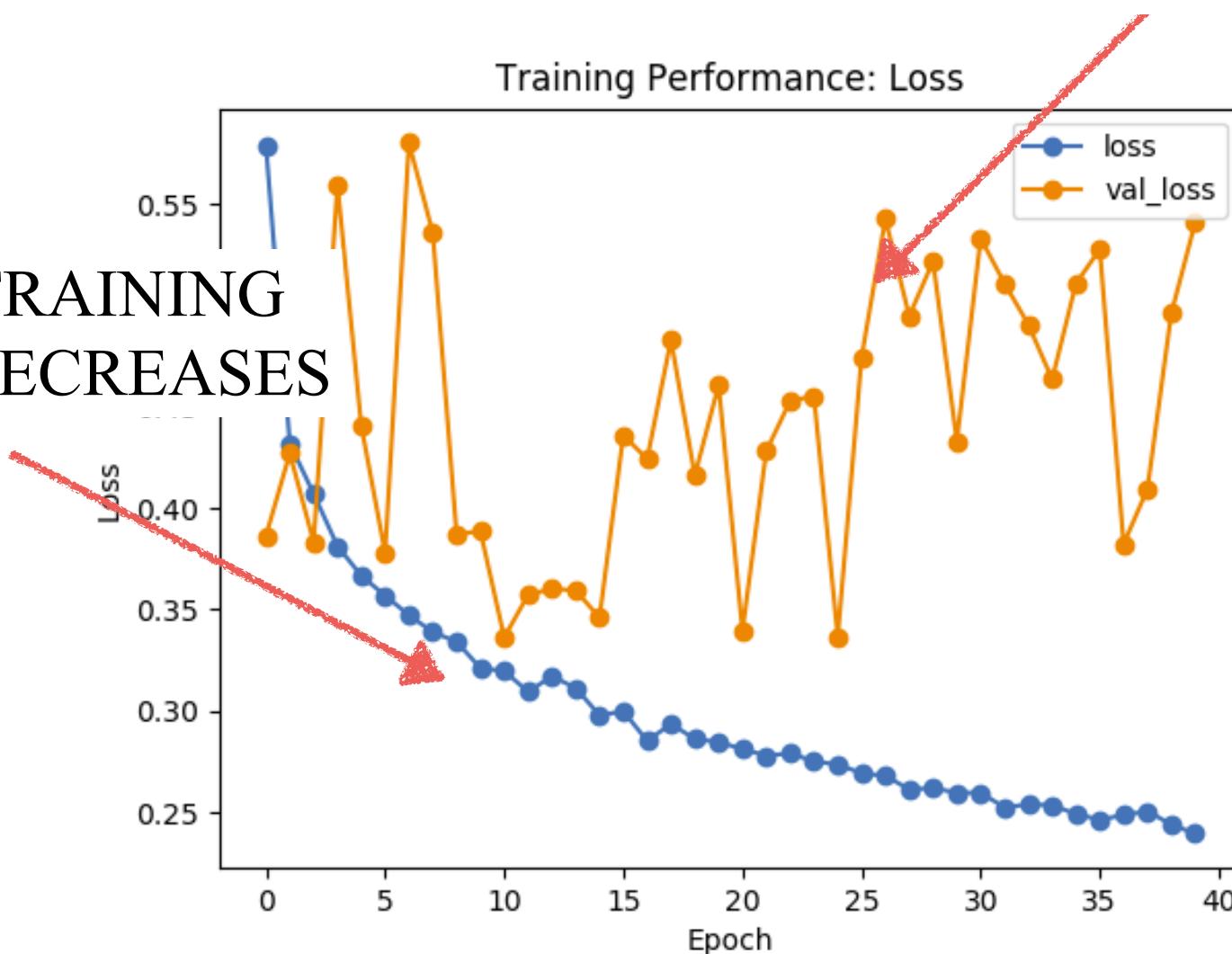
THE PROBLEMS OF GOING “TOO DEEP”

- DEEP NETWORKS ARE MORE DIFFICULT TO OPTIMIZE
- NEED MORE DATA - MORE SUBJECT TO OVERFITTING
- AND ALSO NEED MORE TIME ...

OVER-FITTING

THE TEST STAYS CONSTANT
OR INCREASES

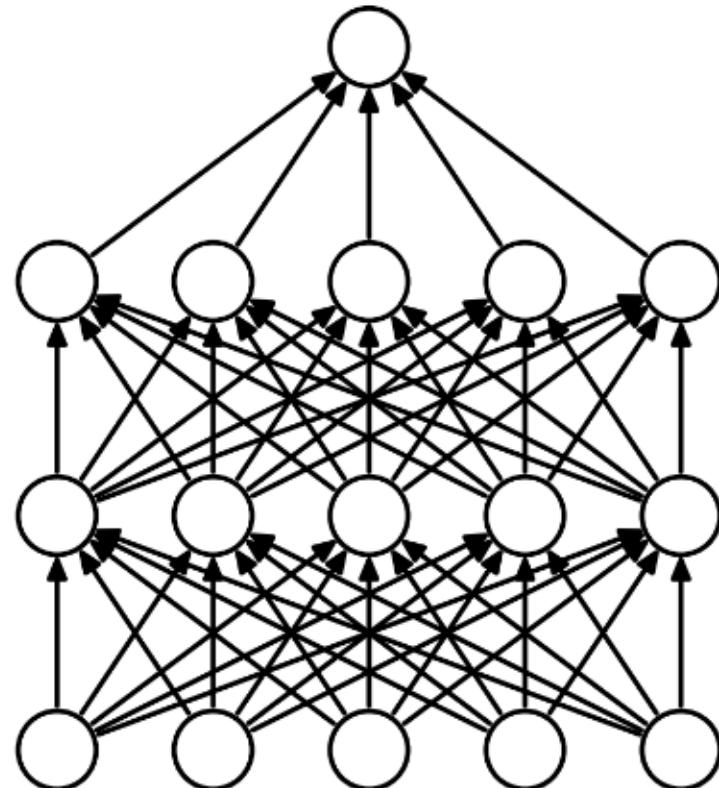
THE TRAINING
LOSS DECREASES



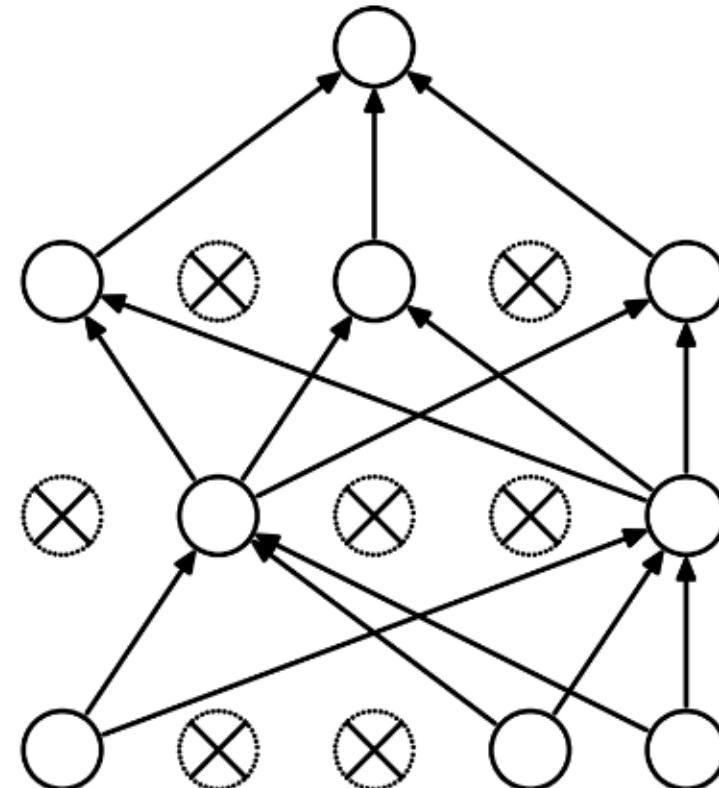
DROPOUT

[Hinton+12]

- THE IDEA IS TO REMOVE NEURONS RANDOMLY DURING THE TRAINING
- ALL NEURONS ARE PUT BACK DURING THE TEST PHASE



(a) Standard Neural Net



(b) After applying dropout.

DROPOUT

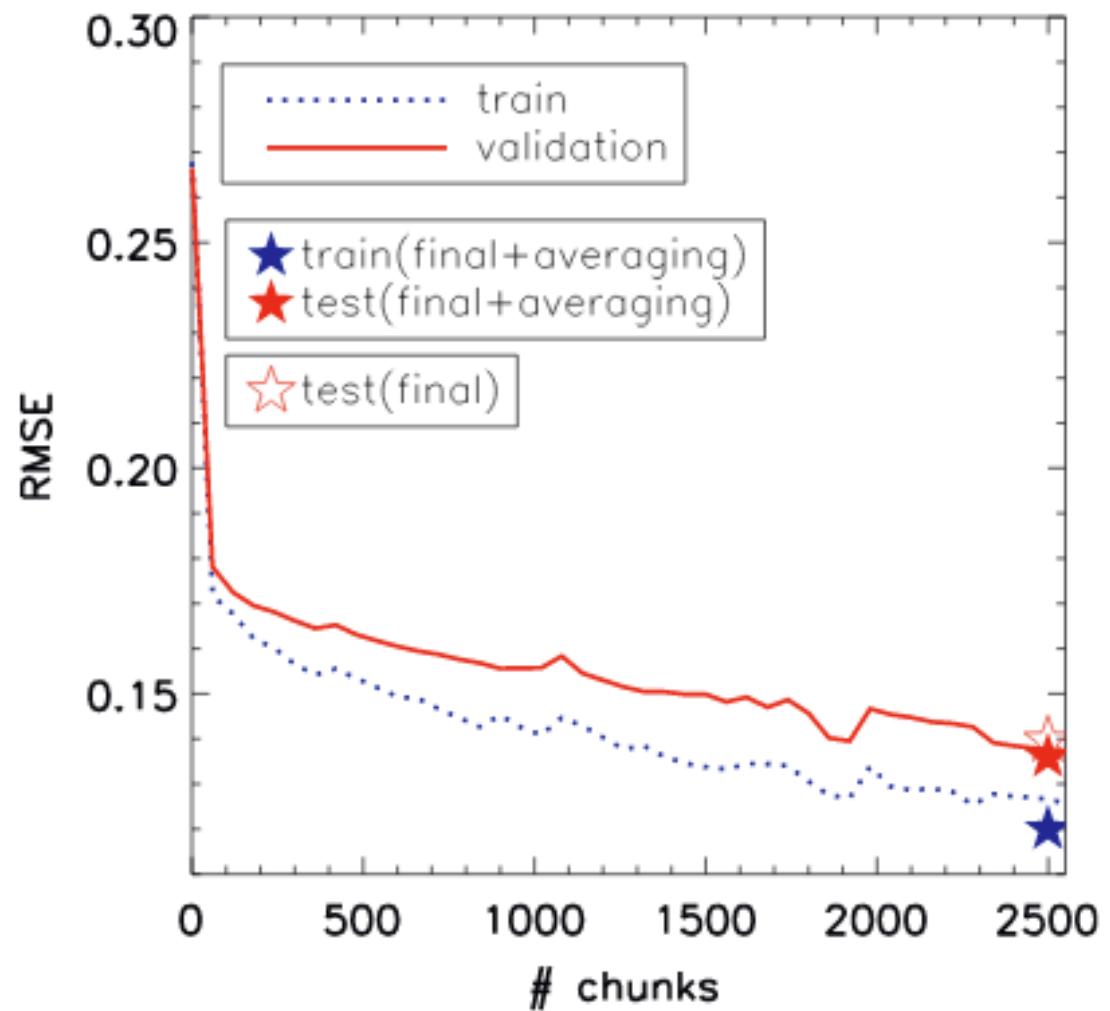
WHY DOES IT WORK?

1. SINCE NEURONS ARE REMOVED RANDOMLY, IT AVOIDS CO-ADAPTATION AMONG THEMSELVES

2. DIFFERENT SETS OF NEURONS WHICH ARE SWITCHED OFF, REPRESENT A DIFFERENT ARCHITECTURE AND ALL THESE DIFFERENT ARCHITECTURES ARE TRAINED IN PARALLEL. FOR N NEURONS ATTACHED TO DROPOUT, THE NUMBER OF SUBSET ARCHITECTURES FORMED IS 2^N . SO IT AMOUNTS TO PREDICTION BEING AVERAGED OVER THESE ENSEMBLES OF MODELS.

DROPOUT

WITH A LITTLE BIT
OF DROPOUT



IMPLEMENTATION IN KERAS / TENSORFLOW

```
#===== Model definition=====

#Convolutional Layers

model = Sequential()
model.add(Convolution2D(32, 6, 6, border_mode='same',
                       input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Convolution2D(64, 5, 5, border_mode='same'))
model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(128, 2, 2, border_mode='same'))
model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(128, 3, 3, border_mode='same'))
model.add(Activation('relu'))

model.add(Dropout(0.25))

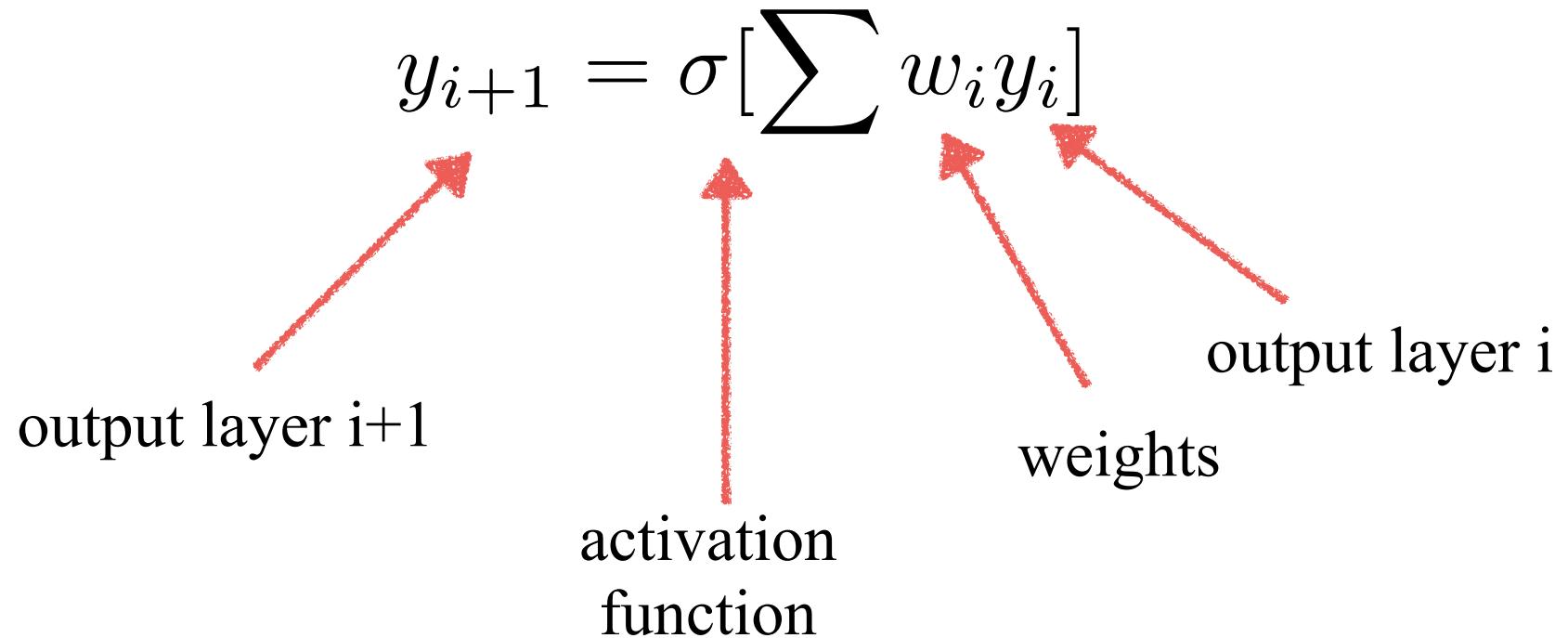
#Fully Connected start here
#-----#
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(.5))
model.add(Dense(1, init='uniform', activation='sigmoid'))

print("Compilation...")

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

VANISHING / EXPLODING GRADIENT PROBLEM

REMEMBER THAT:



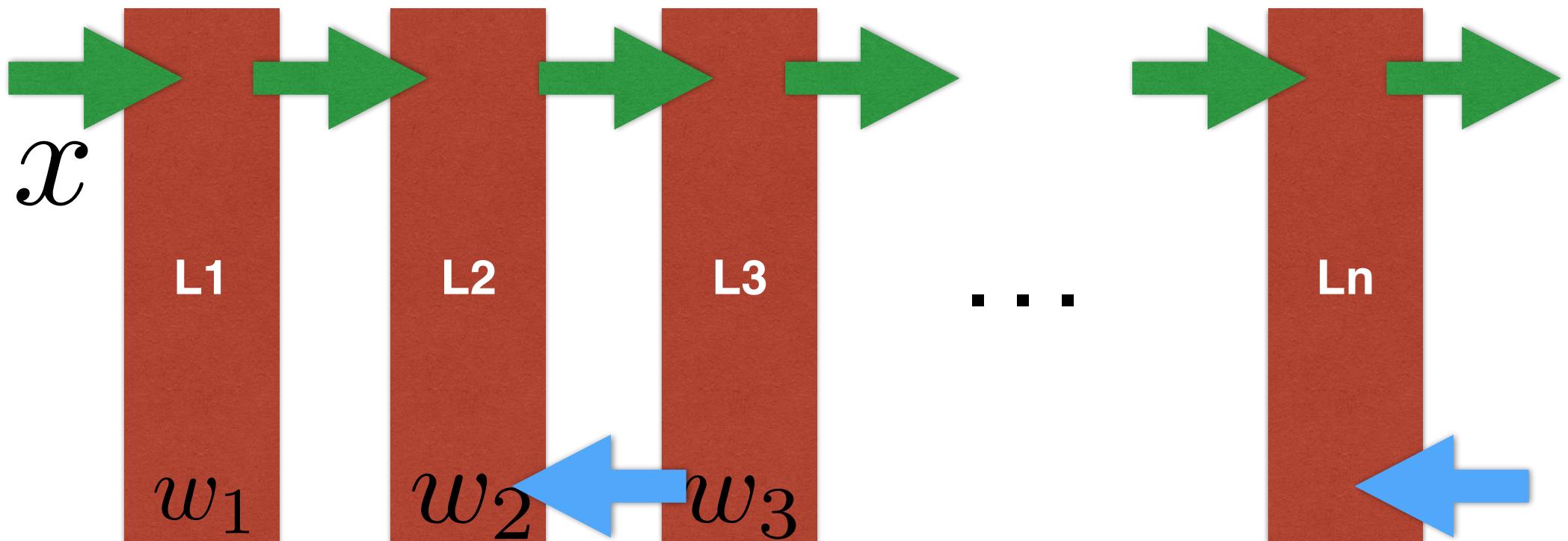
VANISHING / EXPLODING GRADIENT PROBLEM

WITH MANY LAYERS:

$$y_n = \sigma \left(\dots \sigma \left(\dots \sigma \left(\sum w_0 x \right) \right) \right)$$

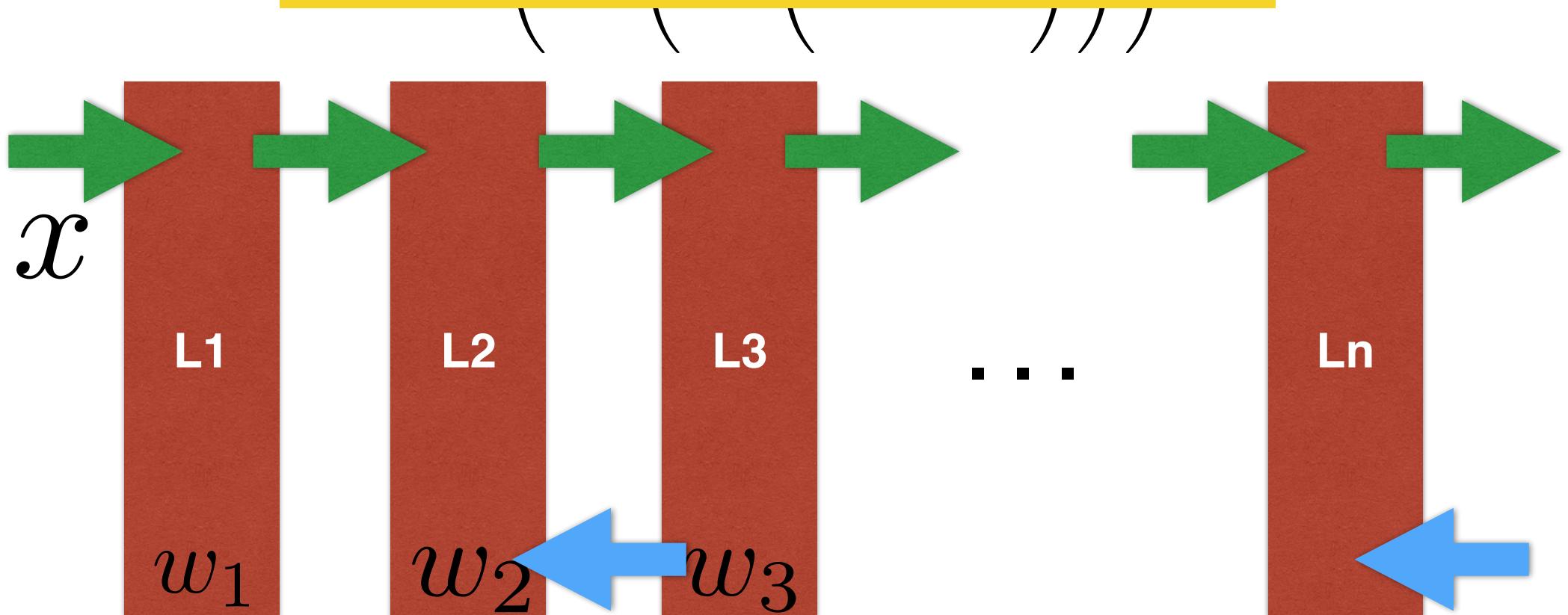
VANISHING/EXPLODING GRADIENT PROBLEM

$$y_n = \sigma\left(\dots\sigma\left(\dots\sigma\left(\sum w_0x\right)\right)\right)$$

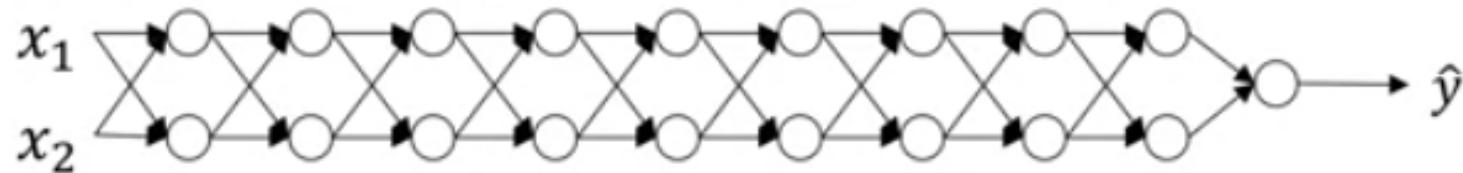


VANISHING/EXPLODING GRADIENT PROBLEM

TRAINING BECOMES UNSTABLE
VERY SLOW OR NO CONVERGENCE



VANISHING/EXPLODING GRADIENT PROBLEM



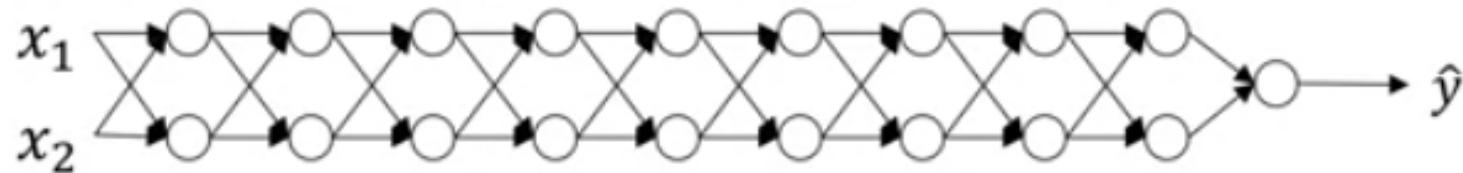
IF WE ASSUME AN IDENTITY ACTIVATION FUNCTION:

$$\hat{y} = x \prod_n w_i$$

with:

$$w_i = \begin{pmatrix} w_i^0 & 0 \\ 0 & w_i^1 \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

VANISHING/EXPLODING GRADIENT PROBLEM



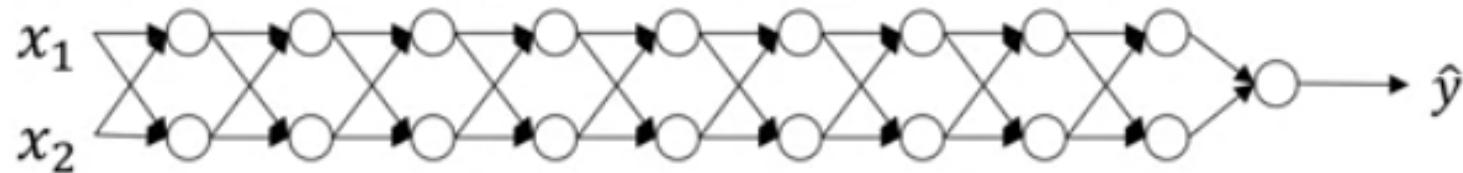
$$w_i = \begin{pmatrix} w_i^0 & 0 \\ 0 & w_i^1 \end{pmatrix} \quad \hat{y} = x \prod_n w_i$$

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{IF WEIGHTS ARE ALL INITIALIZED TO VALUES } \ll 1:$$

$$w_i^L \rightarrow 0$$

VANISHING GRADIENT

VANISHING/EXPLODING GRADIENT PROBLEM



$$w_i = \begin{pmatrix} w_i^0 & 0 \\ 0 & w_i^1 \end{pmatrix} \quad \hat{y} = x \prod_n w_i$$

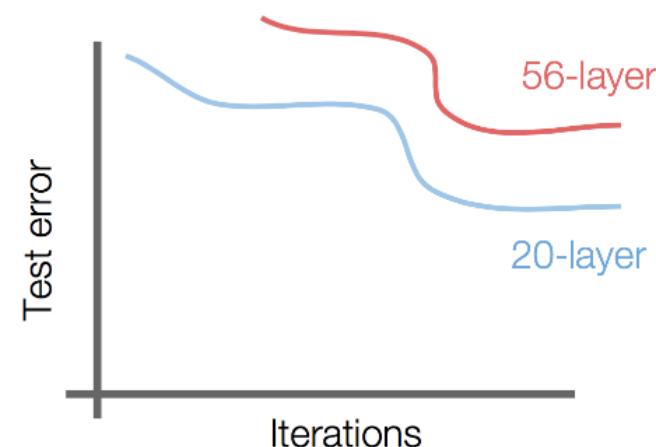
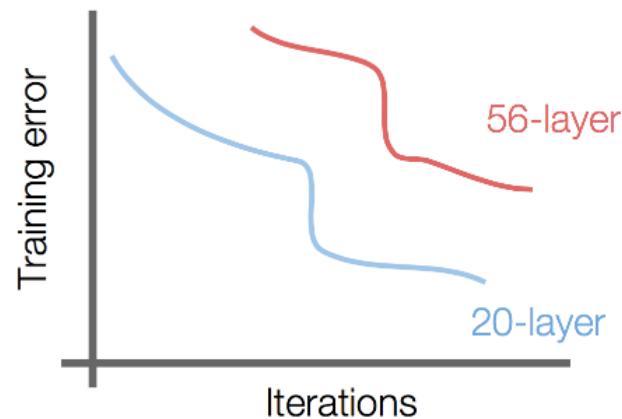
$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{IF WEIGHTS ARE ALL INITIALIZED TO VALUES } > 1:$$

$$w_i^L \rightarrow \infty$$

EXPLODING GRADIENT

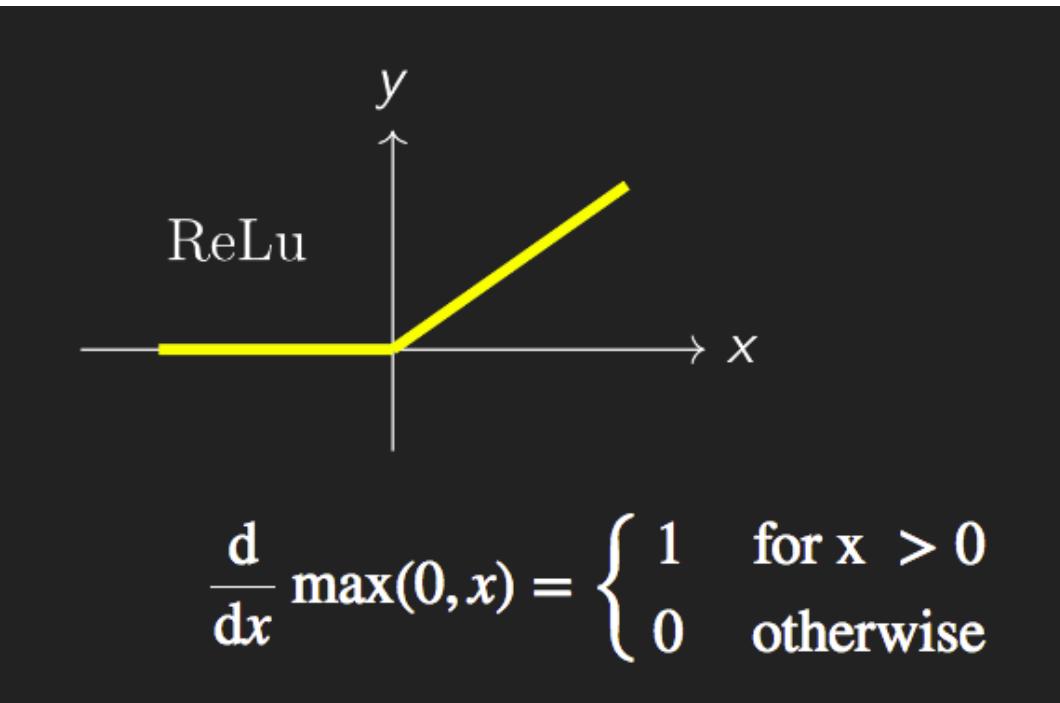
VANISHING/EXPLODING GRADIENT PROBLEM

**TRAINING BECOMES UNSTABLE
VERY SLOW OR NO CONVERGENCE**



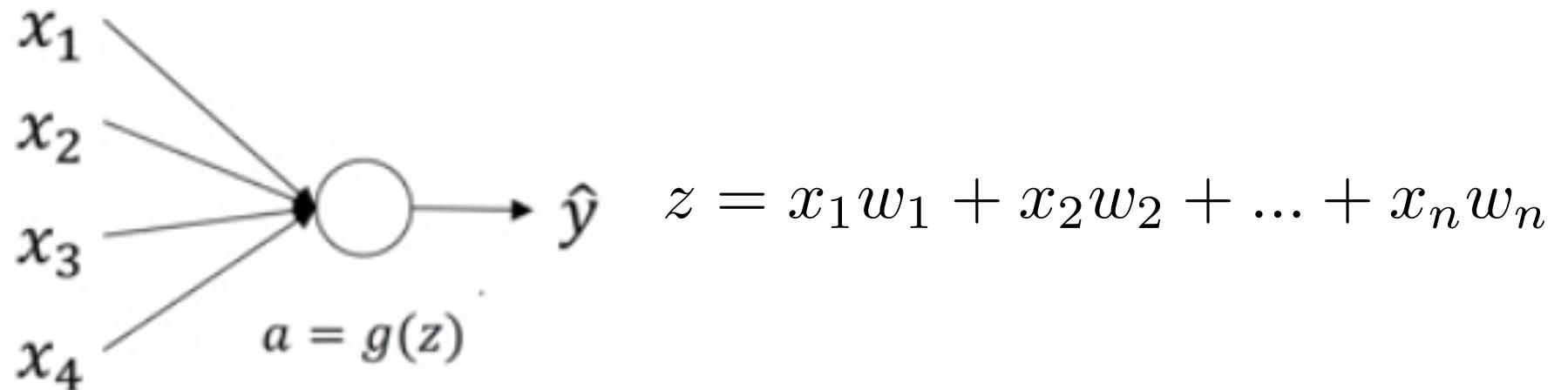
VANISHING GRADIENT PROBLEM

THE ReLU ACTIVATION FUNCTION HELPS PREVENTING VANISHING GRADIENTS

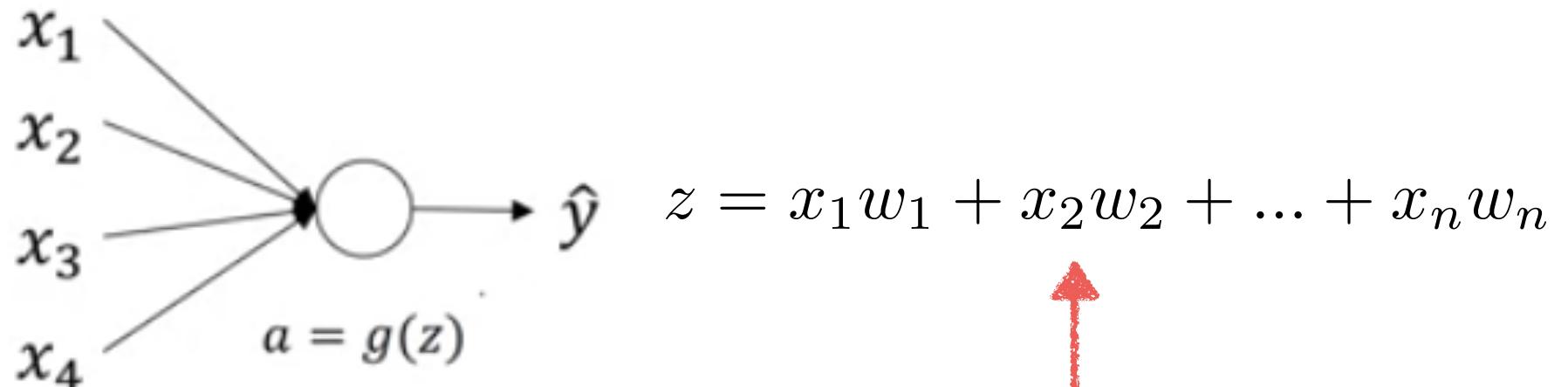


MOST WIDELY USED
IN DEEP NETWORKS

WEIGHT INITIALIZATION IS A KEY POINT...

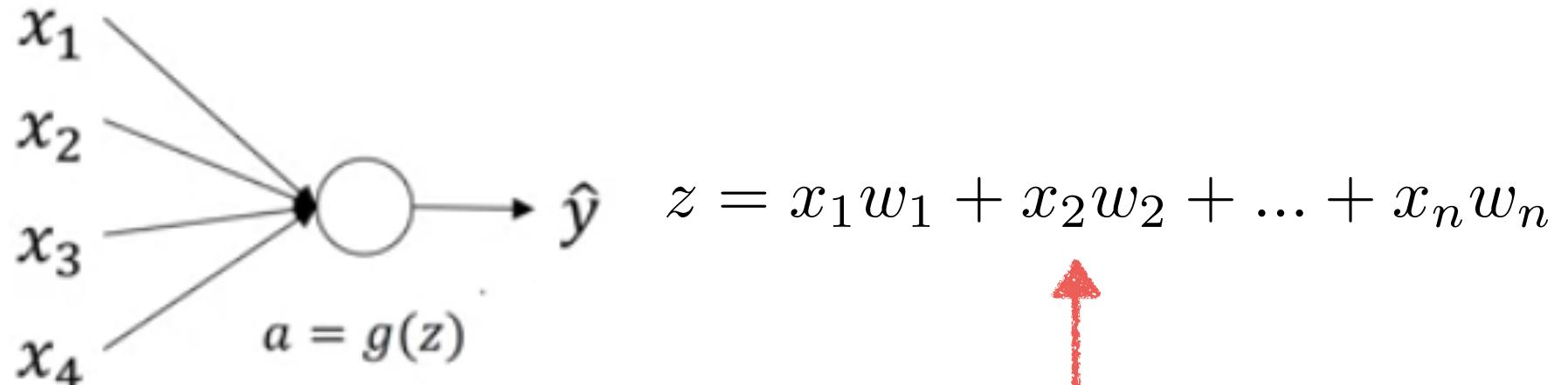


WEIGHT INITIALIZATION IS A KEY POINT...



THE LARGER n , THE SMALLER
WEIGHTS SHOULD BE...

WEIGHT INITIALIZATION IS A KEY POINT...



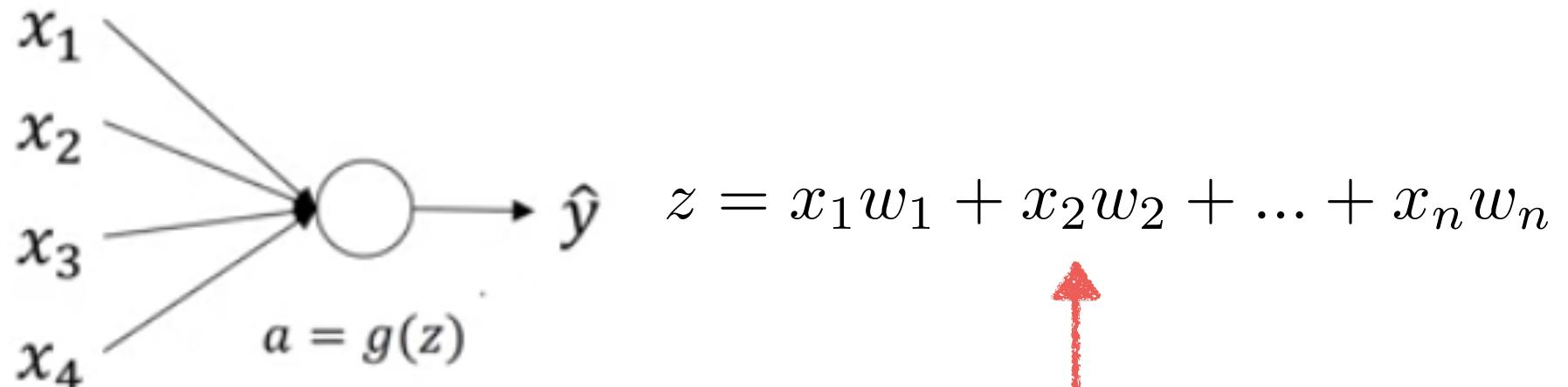
THE LARGER n , THE SMALLER
WEIGHTS SHOULD BE...

ONE SIMPLE SOLUTION:

$$\sigma^2(w_i) = \frac{1}{n}$$

number
of
inputs

WEIGHT INITIALIZATION IS A KEY POINT...



THE LARGER n , THE SMALLER
WEIGHTS SHOULD BE...

ONE SIMPLE SOLUTION:

$$\sigma^2(w_i) = \frac{1}{n}$$

number
of
inputs

WEIGHT INITIALIZATION IS A KEY POINT...

For ReLU activation functions we typically use:

$$\sigma^2(w_i) = \frac{2}{n}$$

[He initialization, He+15]

WEIGHT INITIALIZATION IS A KEY POINT...

IMPLEMENTATION IN KERAS:

```
initialization = 'he_normal'  
act = 'relu'  
  
model = Sequential()  
model.add(Convolution2D(depth_, conv_size, conv_size, activation=act, border_mode='same',  
name = "conv%i"%(layer_n), init=initialization, W_constraint=constraint))
```

WEIGHT INITIALIZATION IS A KEY POINT...

IMPLEMENTATION IN KERAS:

```
initialization = 'he_normal'  
act = 'relu'  
  
model = Sequential()  
model.add(Convolution2D(depth_, conv_size, conv_size, activation=act, border_mode='same',  
name = "conv%i"%(layer_n), init=initialization, W_constraint=constraint))
```

MANY OTHER INITIALIZATIONS AVAILABLE:

keras.initializers



<https://keras.io/initializers/>

BATCH NORMALIZATION

[SZEGEDY+15]

ANOTHER SOLUTION TO KEEP REASONABLE VALUES OF
THE ACTIVATIONS IN DEEP NETWORKS

BATCH NORMALIZATION PREVENTS LOW OR LARGE
VALUES BY RE-NORMALIZING THE VALUES BEFORE
ACTIVATION FOR EVERY BATCH

$$\hat{y}_i = \gamma \frac{y_i - E(y_i)}{\sigma(y_i)} + \beta$$

INPUT NORMALIZED INPUT SCATTER

BATCH NORMALIZATION

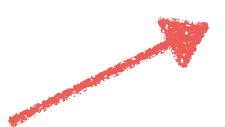
[SZEGEDY+15]

BATCH NORMALIZATION SPEEDS UP AND STABILIZES TRAINING

AS FOR THE DROPOUT, THERE IS A DIFFERENT BEHAVIOR BETWEEN TRAINING AND TESTING

$$\hat{y}_i = \gamma \frac{y_i - E(y_i)}{\sigma(y_i)} + \beta$$

INPUT 

NORMALIZED INPUT 

SCATTER 

BATCH NORMALIZATION

[SZEGEDY+15]

IN KERAS, IT IS IMPLEMENTED AS AN ADDITIONAL LAYER

BatchNormalization

[\[source\]](#)

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True, beta_initializer
```

Batch normalization layer (Ioffe and Szegedy, 2014).

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Arguments