ICRAR

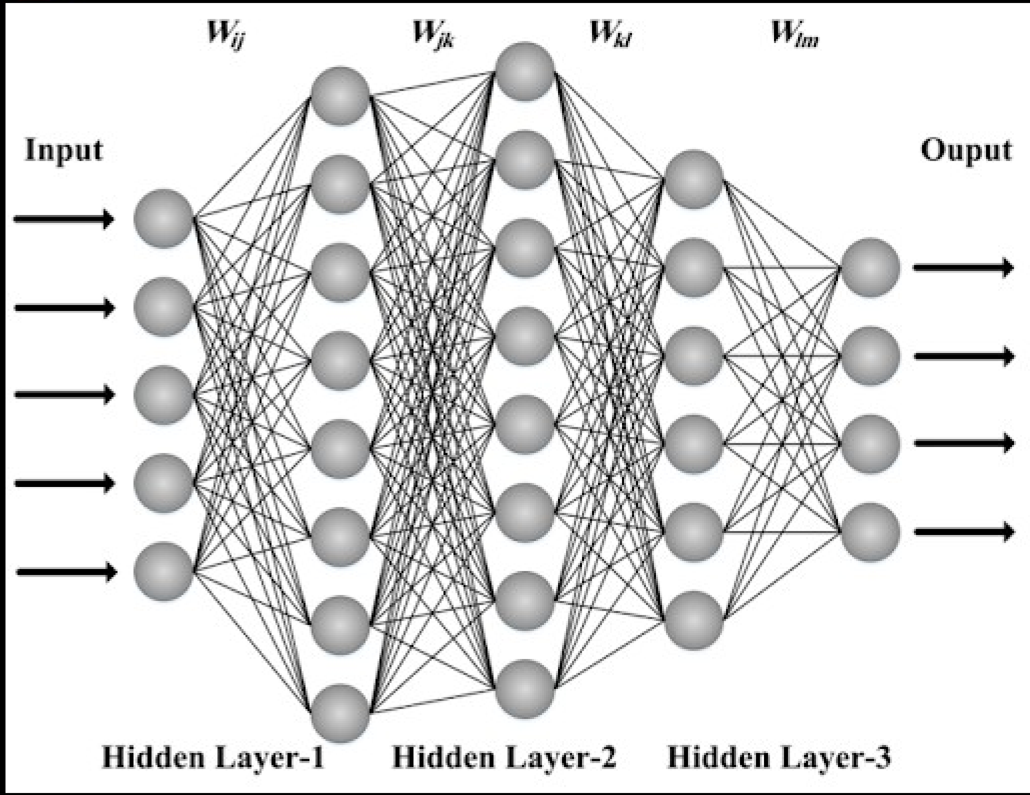
International Centre for Radio Astronomy Research

DL basics and (some) good practices: Dense, Convolutions (order!)

Foivos I. Diakogiannis

Curtin University
THE UNIVERSITY OF WESTERN AUSTRALIA
Government of Western Australia
Department of the Premier and Cabinet
Office of Science

Starting with something you are (probably) familiar with: the `Sequential()` model



Pytorch

```python
# Example of using Sequential
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )
```

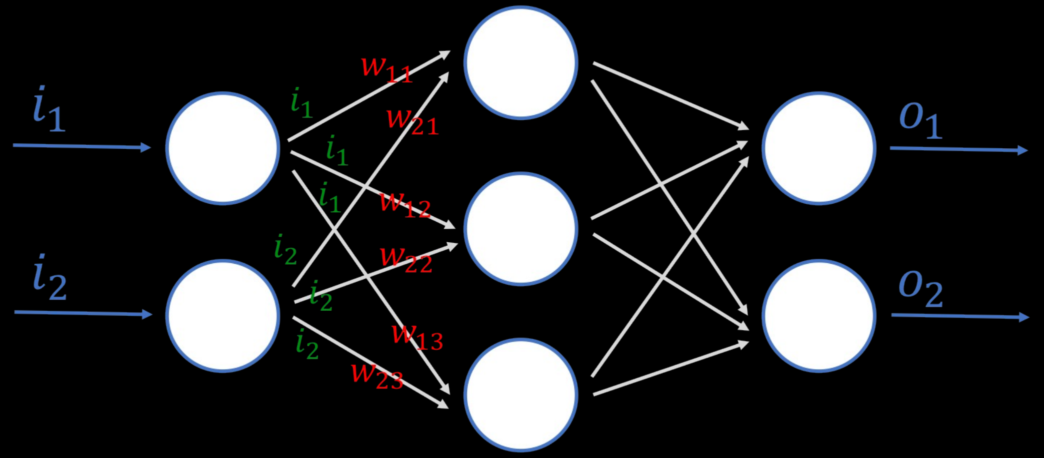
TF 2.0

```python
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

mxnet

```python
num_hidden = 64
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_outputs))
```

Dense/Linear layer basics



$$\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} (w_{11} \times i_1) + (w_{21} \times i_2) \\ (w_{12} \times i_1) + (w_{22} \times i_2) \\ (w_{13} \times i_1) + (w_{23} \times i_2) \end{bmatrix}$$

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b,$$

LINEAR

CLASS `torch.nn.Linear(in_features: int, out_features: int, bias: bool = True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

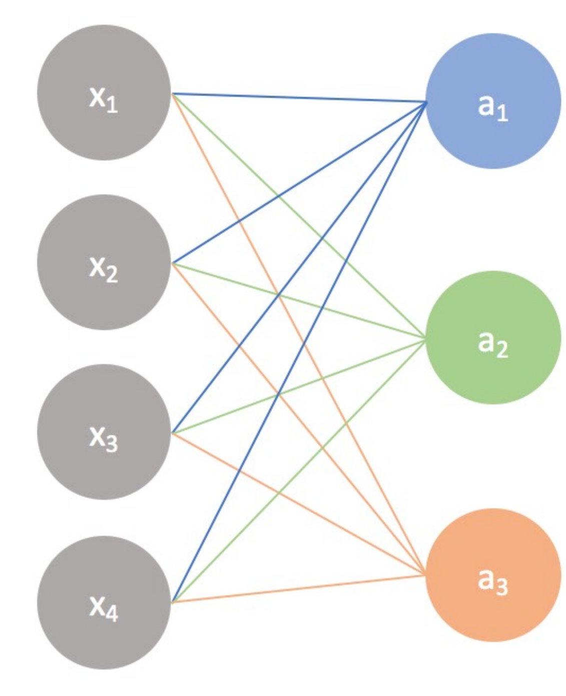
Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

Image credit: https://sausheong.github.io/posts/how-to-build-a-simple-artificial-neural-network-with-go/

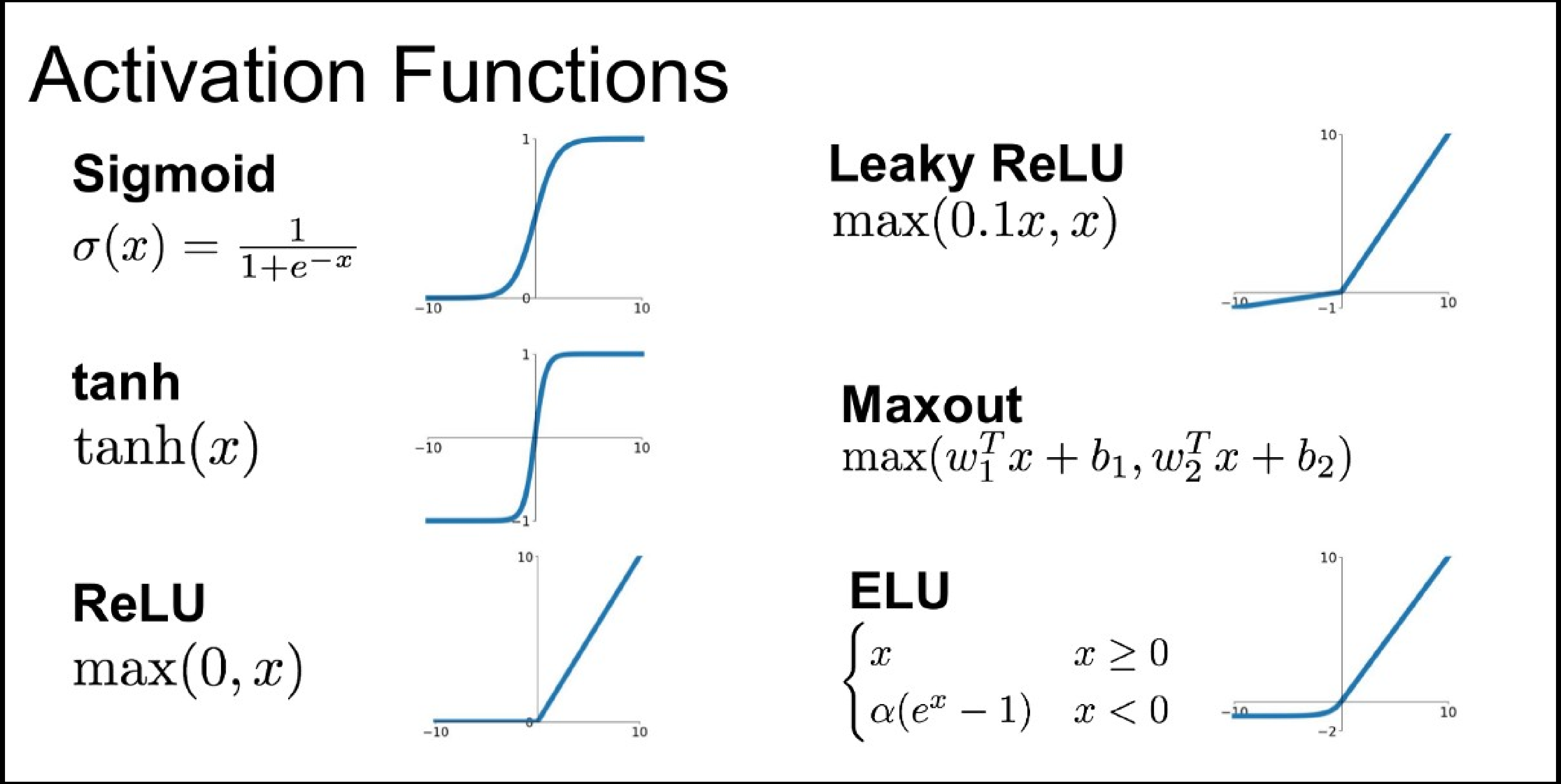
Dense/Linear layer basics

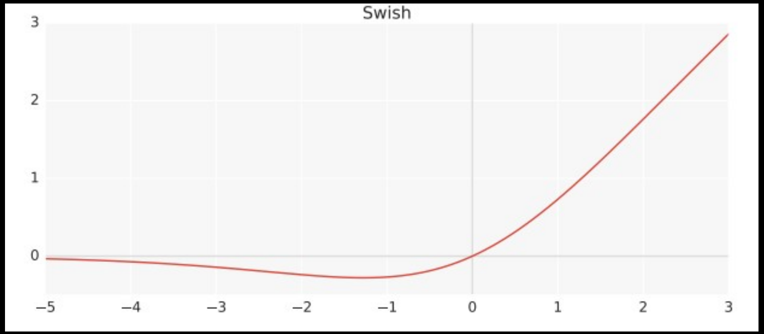


$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow[\text{activation}]{} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Image credit: https://www.jeremyjordan.me/intro-to-neural-networks/

some non-linear activation functions





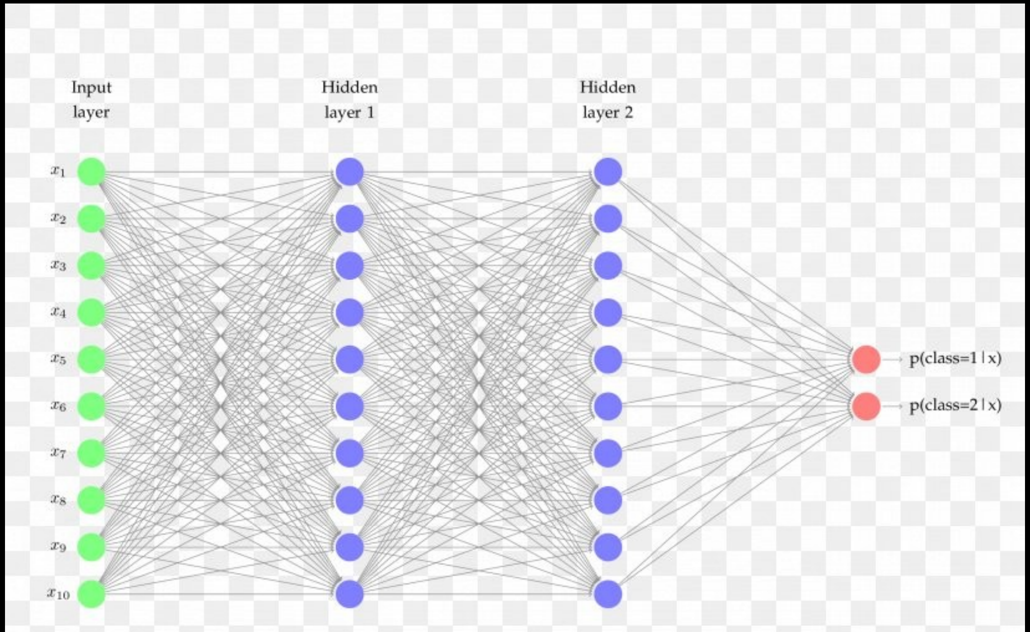

```python
import torch

class Swish(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self,input):
        return input*torch.sigmoid(input)
```

Image credit: https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

WHERE (experimenal observations)**:**

- ReLU and variants – middle layers
- Sigmoid/Softmax/tanh – last layer (output range [0,1] or [-1,1])

Multi-Layer Perceptron (MLP or Dense NN): the Sequential model

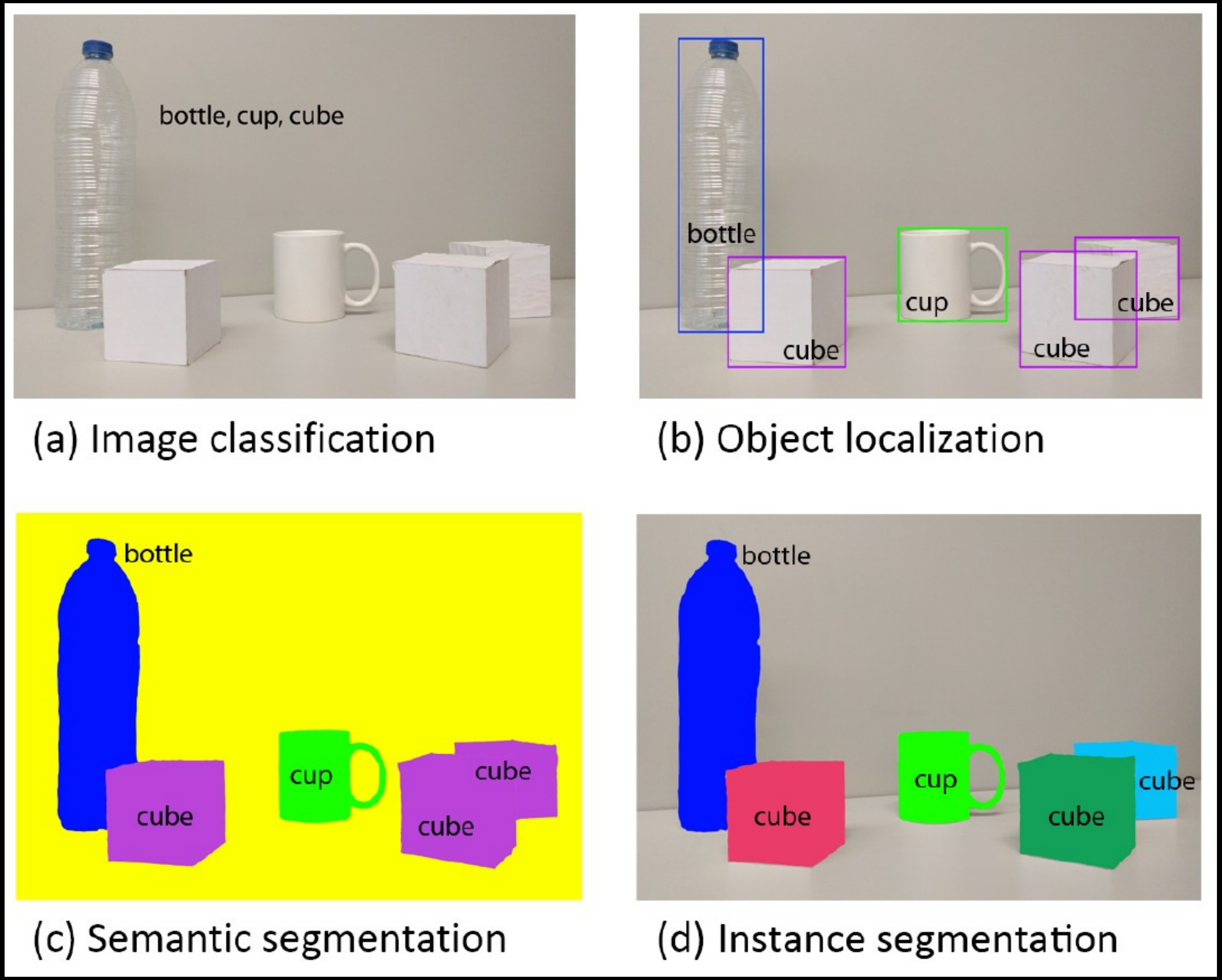


WHEN TO USE DENSE?: usually, when the input is **unordered** data (tabular data) or at least when we don't know the order.

```
import torch

net = torch.nn.Sequential(torch.nn.Linear(in_features=10,out_features=128),
                          torch.nn.ReLU(),
                          torch.nn.Linear(in_features=128,out_features=128),
                          torch.nn.ReLU(),
                          torch.nn.Linear(in_features=128,out_features=2),
                          torch.nn.Softmax(dim=-1)) # Last activation, make it probability: p1+p2=1.0

# A random vector of batch size: 5, with dimensionality 10
xx = torch.rand(5,10)
out = net(xx)
print (out.shape)
```

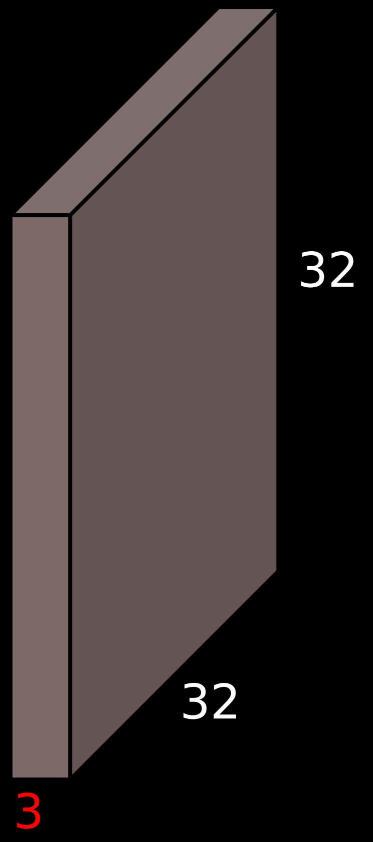
```
torch.Size([5, 2])
```

Some types of problems for ConvNets

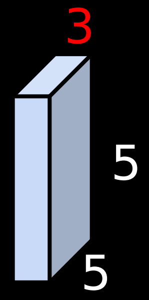


Convolution basics

32x32x3 image



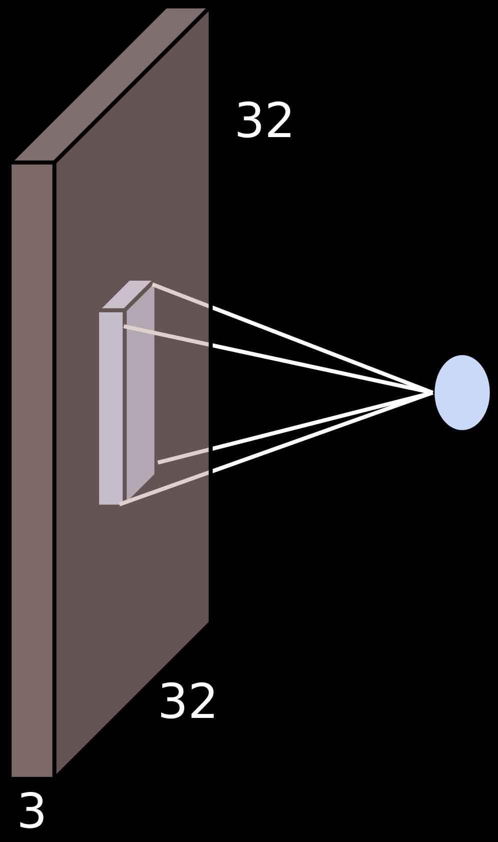
5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

credit: cs231 stanford

Convolution basics

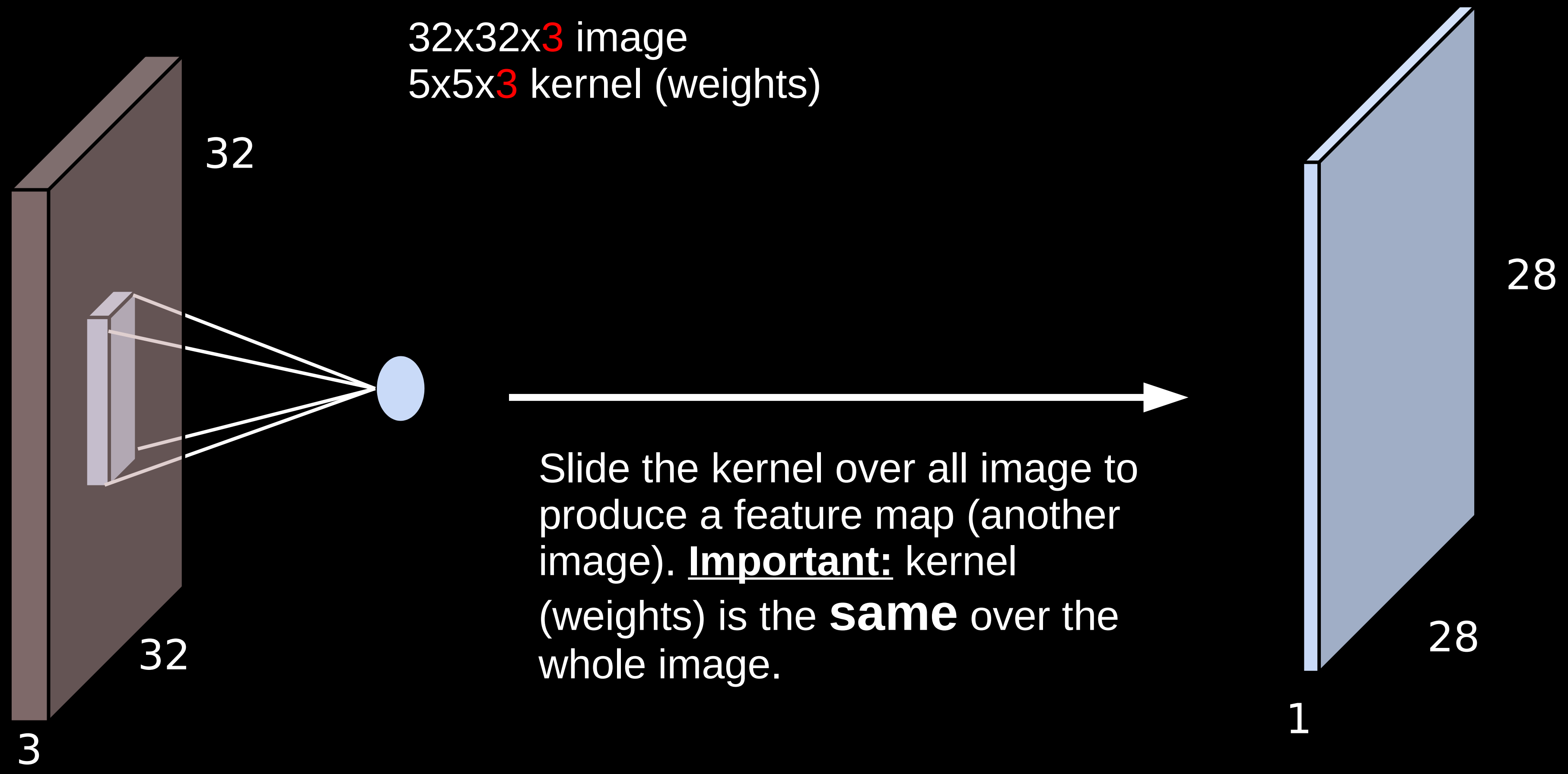


Output: 1 number, the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

credit: cs231 stanford

Convolution basics

32x32x3 image
5x5x3 kernel (weights)

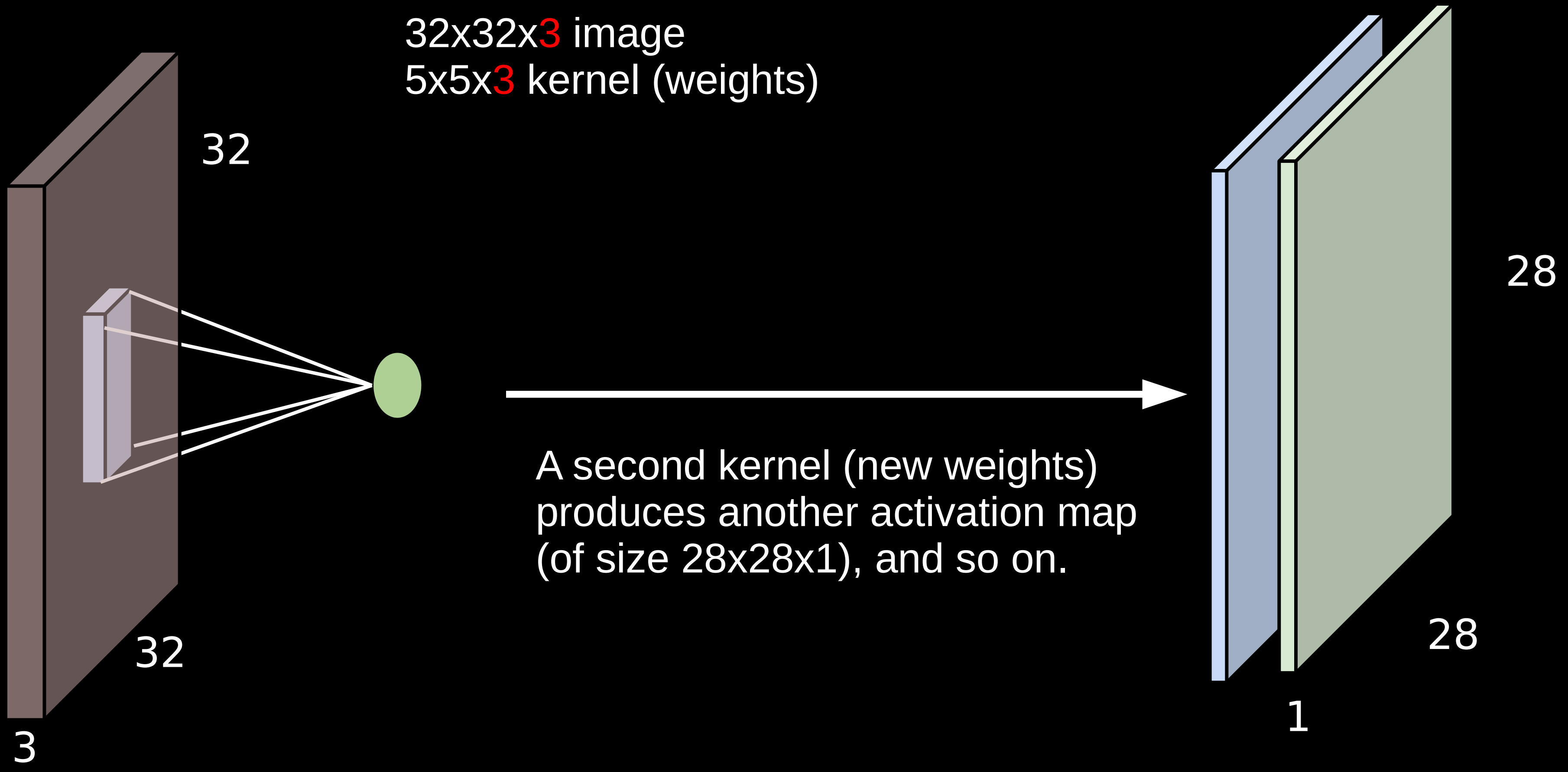


Slide the kernel over all image to produce a feature map (another image). **<u>Important:</u>** kernel (weights) is the **same** over the whole image.

credit: cs231 stanford

Convolution basics

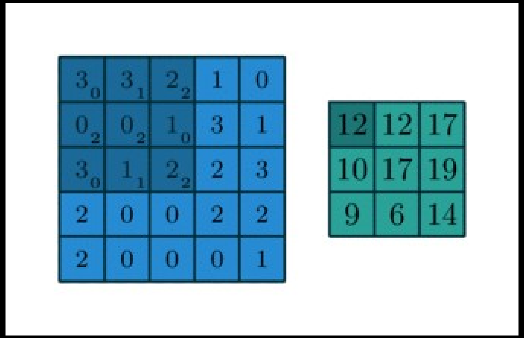
32x32x3 image
5x5x3 kernel (weights)



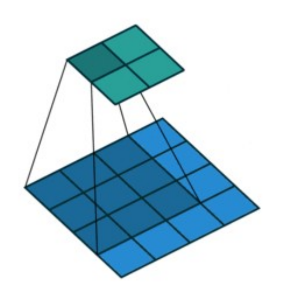
A second kernel (new weights) produces another activation map (of size 28x28x1), and so on.

credit: cs231 stanford

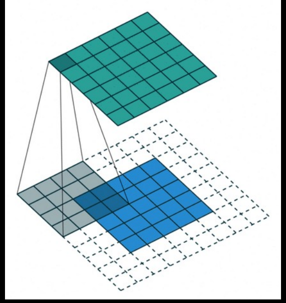
Convolution … not so basics



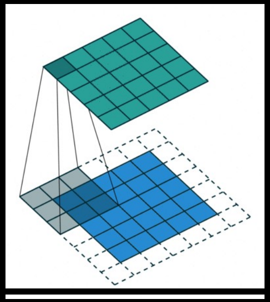
k=3,p=0,s=1,d=1



k=4,p=2,s=1,d=1



pad="same"
k=3,p=1,s=1,d=1



k=3,p=1,s=2,d=1

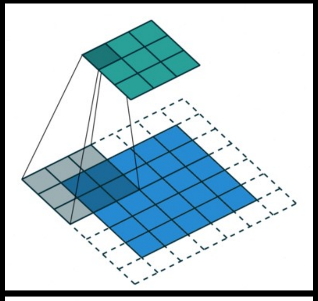


Image as data: (vector) values ***ordered on a coordinate grid***

x1	x2
x3	x4

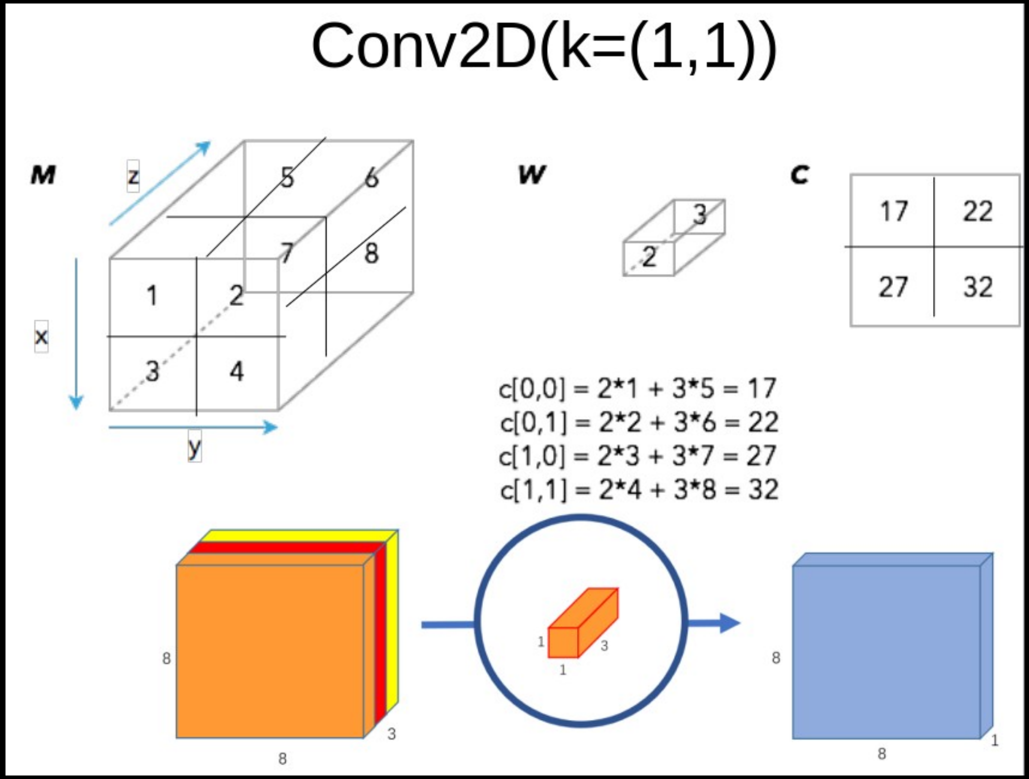
k1	k2
k3	k4

→ x1 k1 + … + x4 k4

(x1,x2,x3,x4)

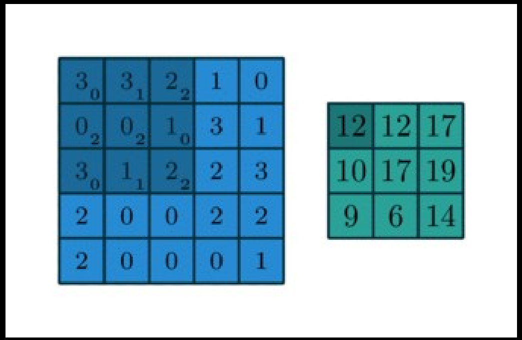
k1
k2
k3
k4

→ x1*k1+...+x4*k4

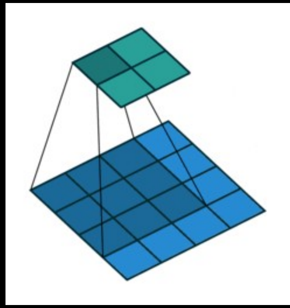


Convolution … not so basics: image as a data structure

(1.2,0.4,1.1)	(1.2,2.4,4.1)	(1.2,0.4,1.1)	(1.7,0.4,1.1)
(0.25,0.45,1.1)	(3.2,0.4,1.1)	(1.2,0.4,1.1)	(0.52,0.6,1.1)
(6.2,0.64,1.11)	(5.2,0.72,0.5)	(1.2,0.4,1.1)	(2.2,0.74,1.1)
(0.21,0.41,1.12)	(7.2,0.4,1.1)	(1.2,0.4,1.1)	(0.22,0.84,1.3)



k=3,p=0,s=1,d=1



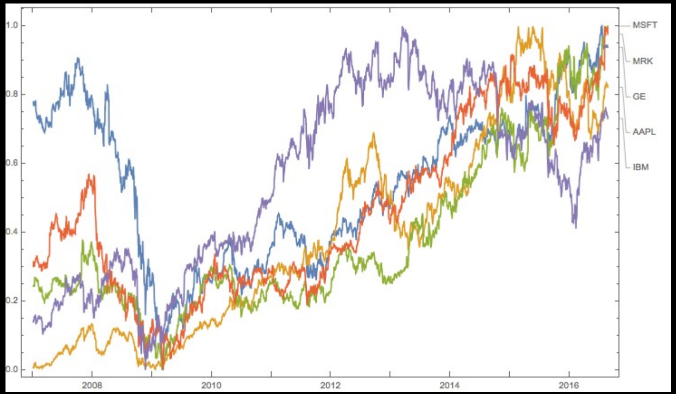
Take away message:

1. Convolutional Neural Networks are MLPs applied repeatedly to ordered subsamples of the data (pixels). The output is ordered on the same (or similar) spatial grid, preserving the original orientation.

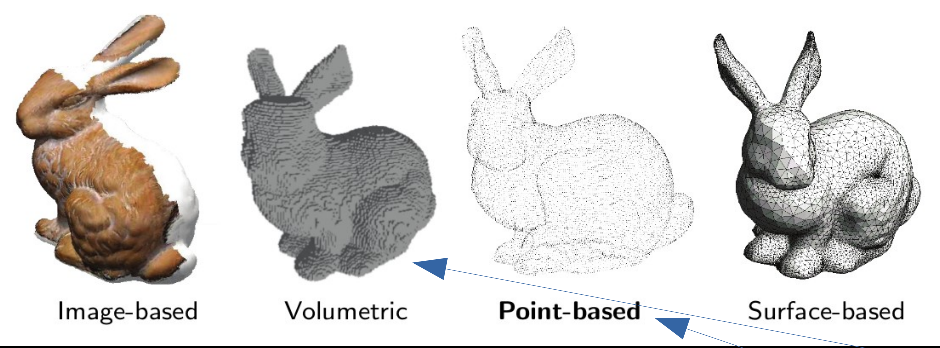
2. CNNs can be applied wherever we have a data structure that consists of a “coordinate system” and “objects” in locations of the coordinate system (e.g. images, volumes, time series, arbitrary graphs).

Ordered datasets - convolution examples

Time series
(classification/regression)

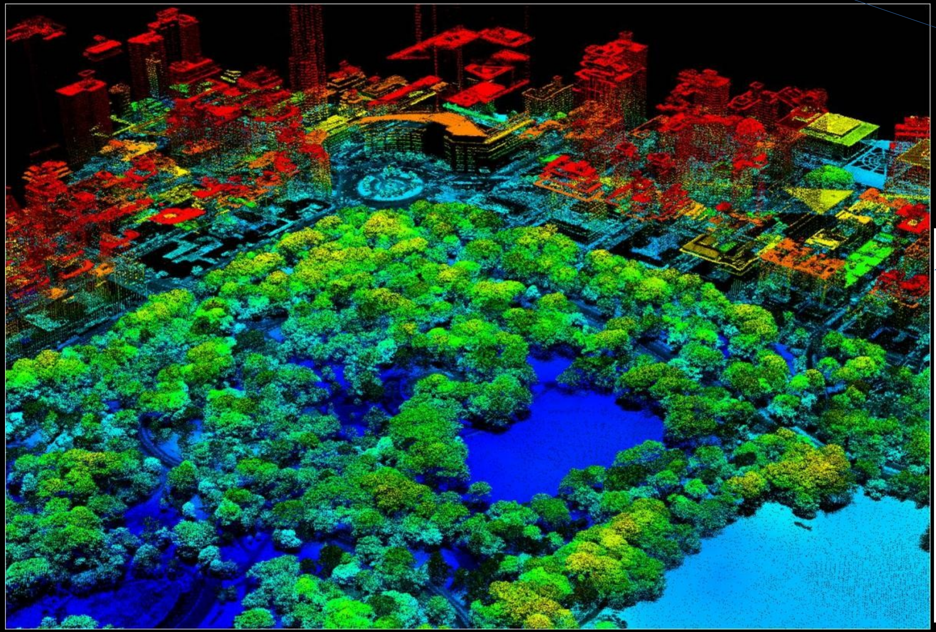


Conv2D | Conv3D | Conv3D, PointNet | Conv2D, Spectral CNN, **Graph CNN**



Point cloud, graphs: open problems, some solutions

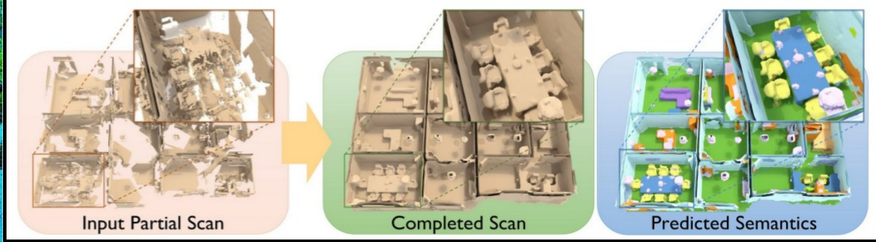
Learning on 3D volumes



LIDAR Cloud point

- precision agriculture,
- cities 3D reconstruction,
- self driving cars
- ...

VOXELS

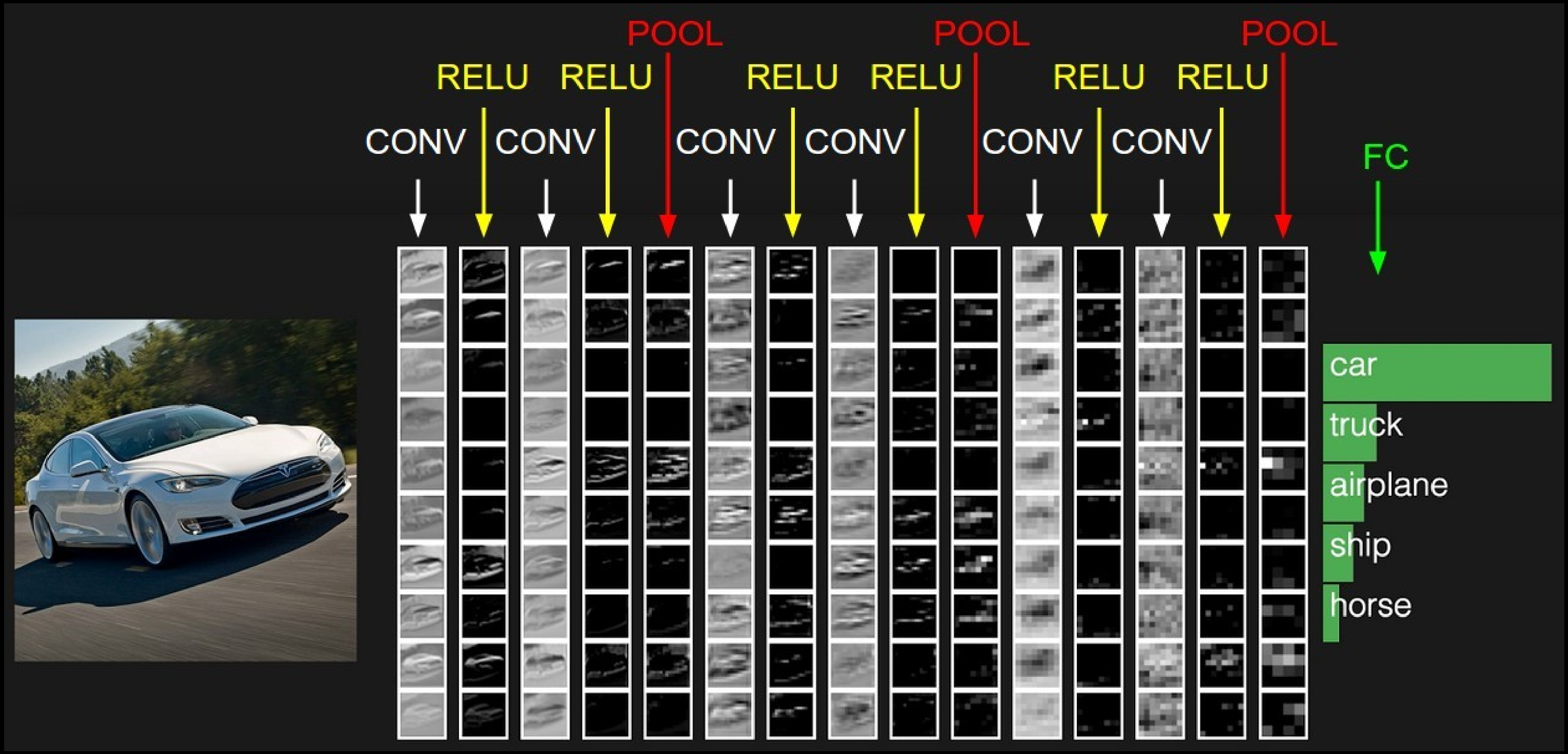


ScanComplete: Large-Scale Scene Completion and Semantic Segmentation for 3D Scans

Angela Dai[1,3,5] Daniel Ritchie[2] Martin Bokeloh[3] Scott Reed[4] Jürgen Sturm[3] Matthias Nießner[5]
[1]Stanford University [2]Brown University [3]Google [4]DeepMind [5]Technical University of Munich

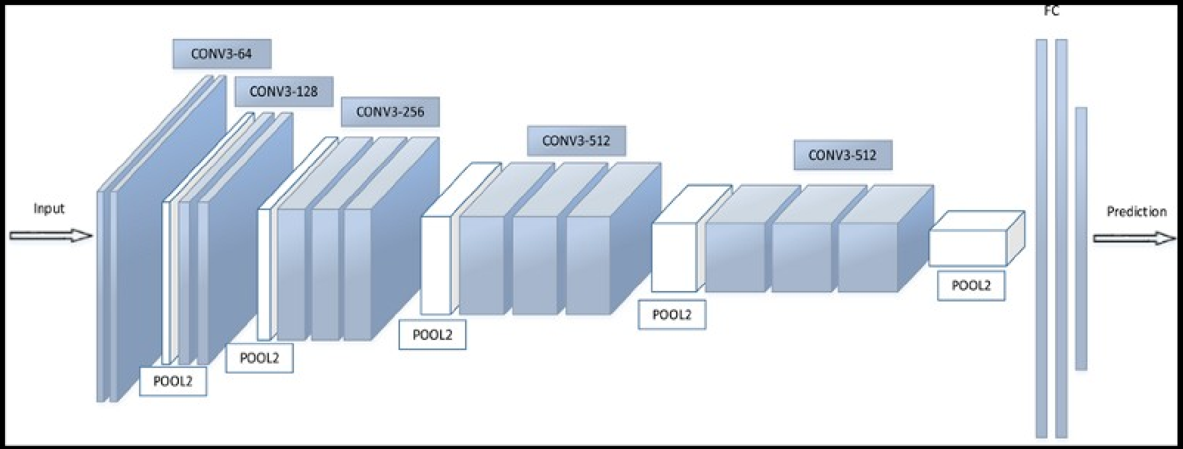
ConvNets: what do they “*see*”?

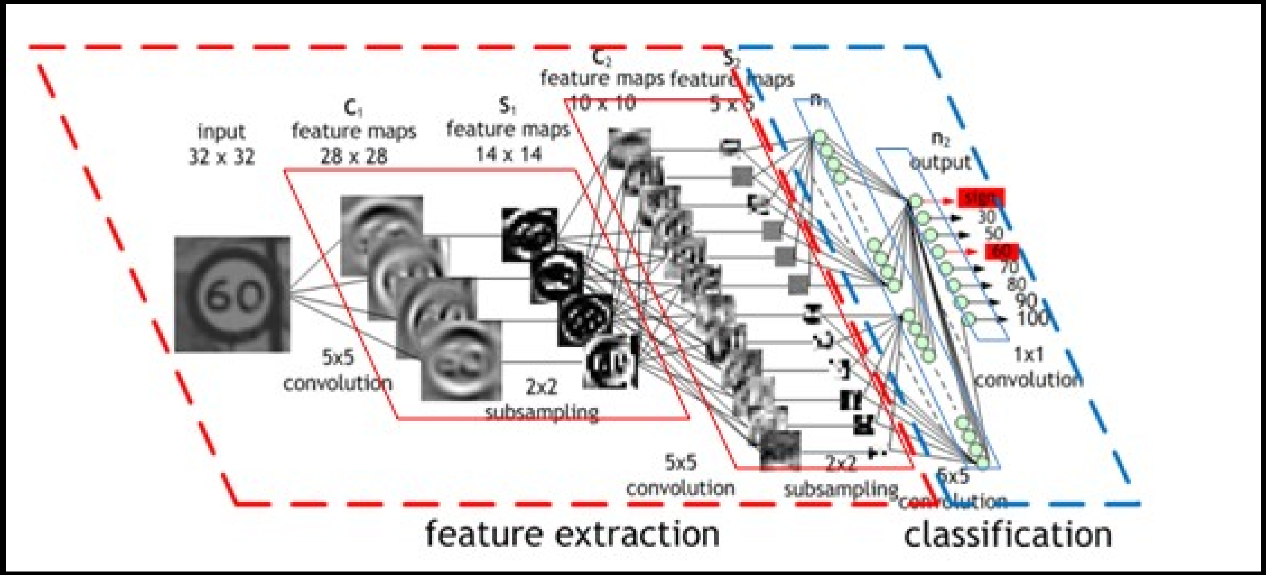
hierarchical features



ConvNets: what do they “*see*”?

Conv nets learn hierarchical features: they summarize important features that distinguish between classes





ConvNets: what do they “*see*”?

Visualizing and Understanding Convolutional Networks

Matthew D. Zeiler — ZEILER@CS.NYU.EDU
Dept. of Computer Science, Courant Institute, New York University

Rob Fergus — FERGUS@CS.NYU.EDU
Dept. of Computer Science, Courant Institute, New York University

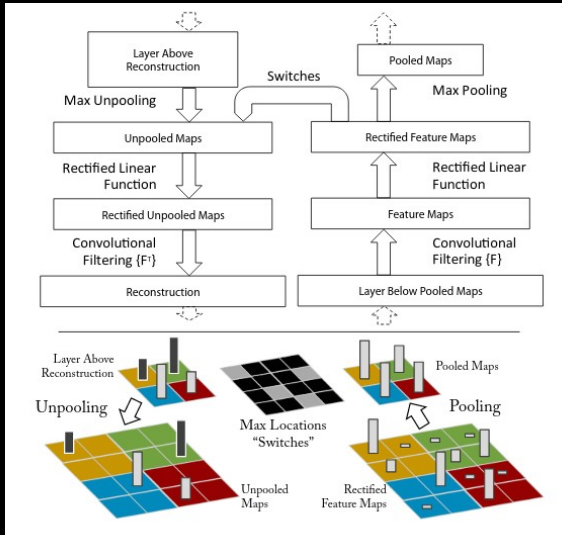
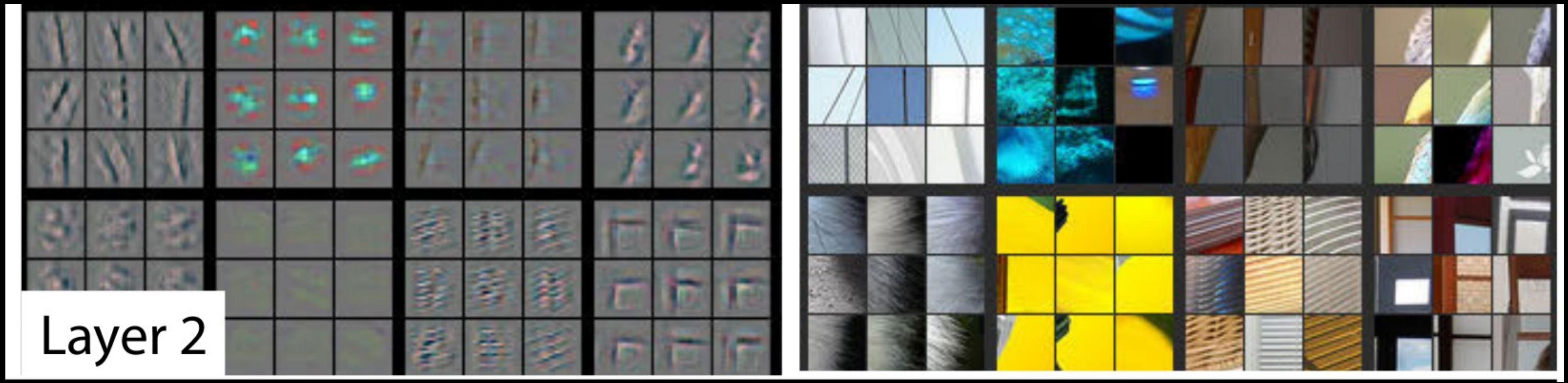
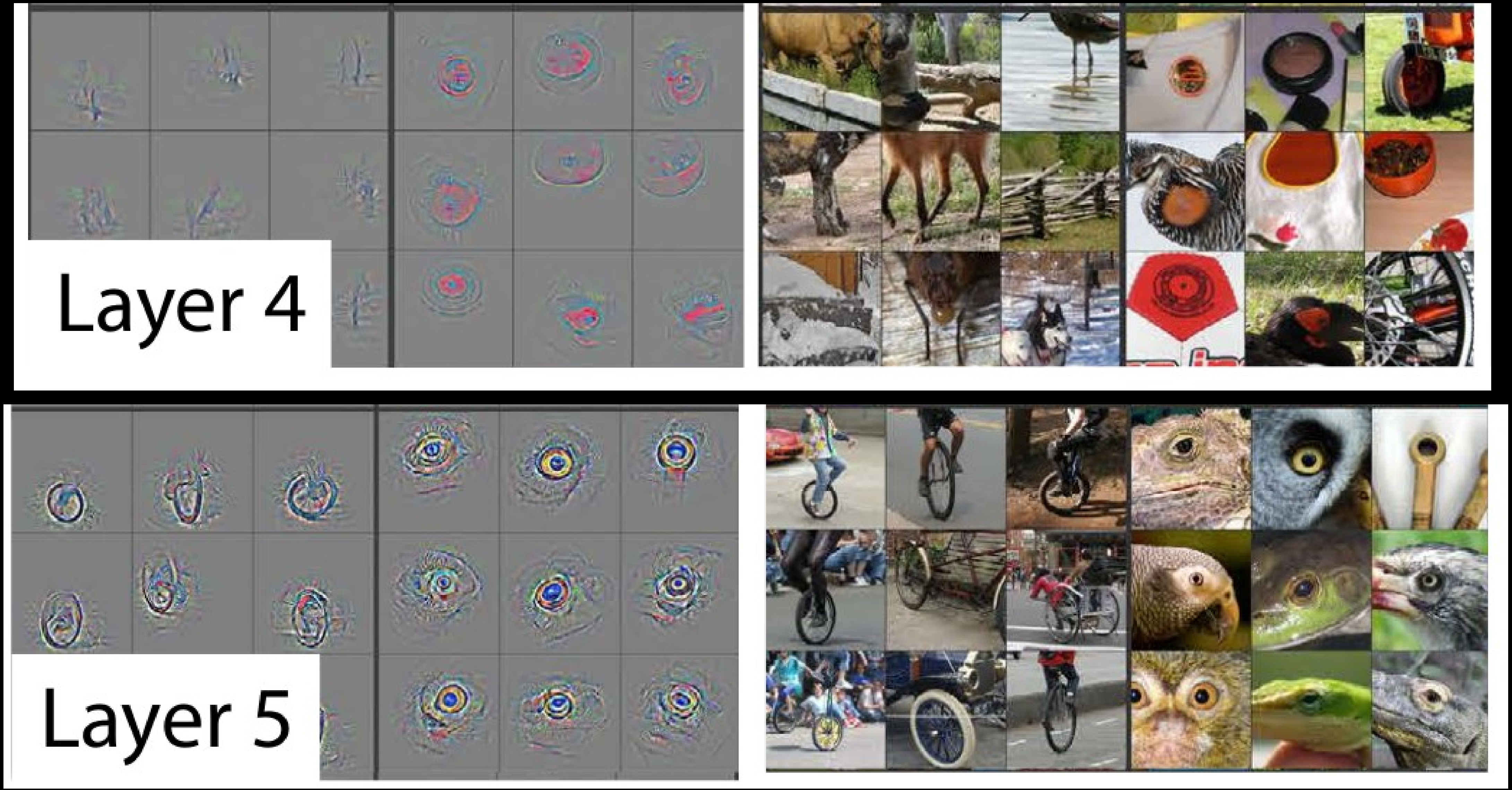


Figure 1. Top: A deconvnet layer (left) attached to a convnet layer (right). The deconvnet will reconstruct an approximate version of the convnet features from the layer beneath. Bottom: An illustration of the unpooling operation in the deconvnet, using *switches* which record the location of the local max in each pooling region (colored zones) during pooling in the convnet.



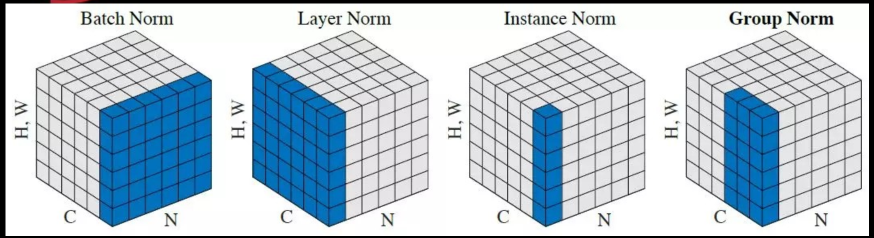
Credit Zeiler et al. 2013 (https://arxiv.org/abs/1311.2901)

ConvNets: what do they "*see*"?



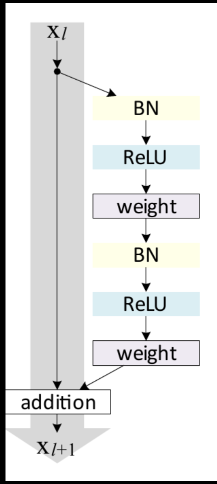
Credit Zeiler et al. 2013 (https://arxiv.org/abs/1311.2901)

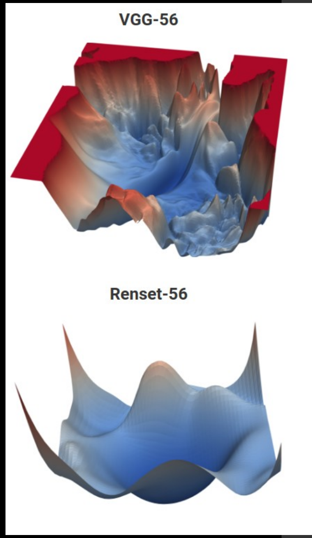
Practicalities 1: Normed Convolutions/ Residual Networks



Normed convolutions inside ConvNets (avoid in last layer), ***usually*** avoid them in GANs (Generator)

```
class Conv2DNormed(HybridBlock):
    """
        Convenience wrapper layer for 2D convolution followed by a normalization layer
        All other keywords are the same as gluon.nn.Conv2D
    """

    def __init__(self,  channels, kernel_size, strides=(1, 1),
                 padding=(0, 0), dilation=(1, 1),   activation=None,
                 weight_initializer=None,  in_channels=0, _norm_type = 'BatchNorm',
                 norm_groups=None, axis =1 , groups=1, **kwards):
        super().__init__(**kwards)

        self.conv2d = gluon.nn.Conv2D(channels, kernel_size = kernel_size,
                                      strides= strides,
                                      padding=padding,
                                      dilation= dilation,
                                      activation=activation,
                                      use_bias=False,
                                      weight_initializer = weight_initializer,
                                      groups=groups,
                                      in_channels=0)

        self.norm_layer = get_norm(_norm_type, axis=axis, norm_groups= norm_groups)

    def forward(self,input):

        x = self.conv2d(input)
        x = self.norm_layer(x)

        return x
```






```
class ResNet_v2_block(HybridBlock):
    """
    ResNet v2 building block. It is built upon the assumption of ODD kernel
    """
    def __init__(self, _nfilters,_kernel_size=(3,3),_dilation_rate=(1,1),
                 _norm_type='BatchNorm', norm_groups=None, ngroups=1, **kwards):
        super().__init__(**kwards)

        self.nfilters = _nfilters
        self.kernel_size = _kernel_size
        self.dilation_rate = _dilation_rate

        # Ensures padding = 'SAME' for ODD kernel selection
        p0 = self.dilation_rate[0] * (self.kernel_size[0] - 1)/2
        p1 = self.dilation_rate[1] * (self.kernel_size[1] - 1)/2
        p = (int(p0),int(p1))


        self.BN1 = get_norm(_norm_type, norm_groups=norm_groups )
        self.conv1 = gluon.nn.Conv2D(self.nfilters,kernel_size = self.kernel_size,
                padding=p,dilation=self.dilation_rate,use_bias=False,groups=ngroups)
        self.BN2 = get_norm(_norm_type, norm_groups= norm_groups)
        self.conv2 = gluon.nn.Conv2D(self.nfilters,kernel_size = self.kernel_size,
                padding=p,dilation=self.dilation_rate,use_bias=True, groups=ngroups)


    def forward(self,_input_layer):

        x = self.BN1(_input_layer)
        x = mx.npx.relu(x)
        x = self.conv1(x)

        x = self.BN2(x)
        x = mx.npx.relu(x)
        x = self.conv2(x)

        return x
```

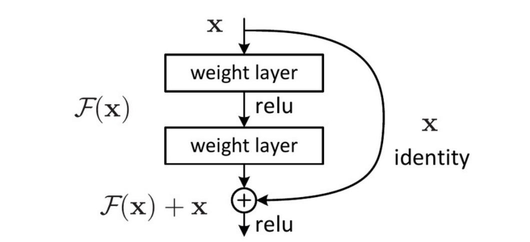
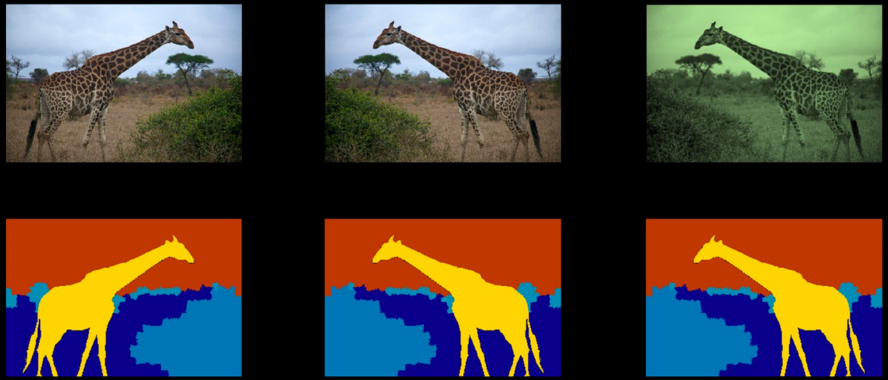


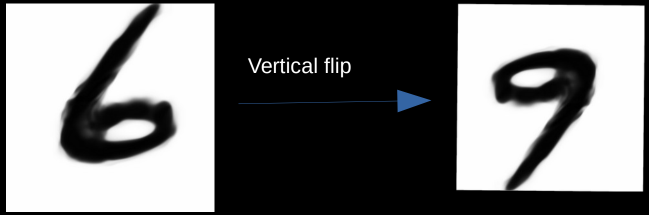
Figure 2. Residual learning: a building block.

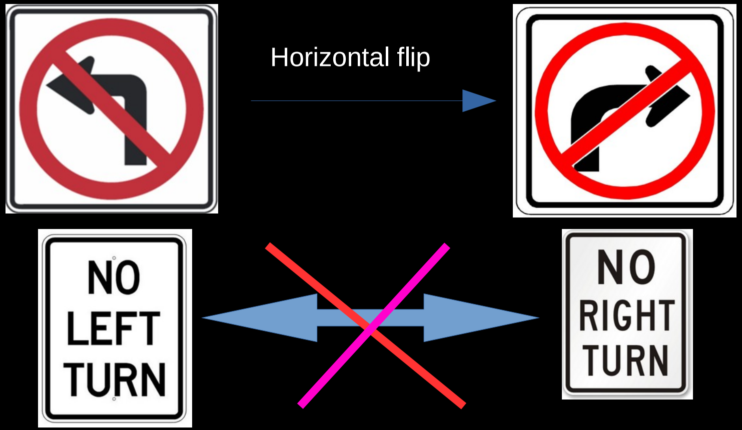
Practicalities 2: Data augmentation <==> PRIOR information

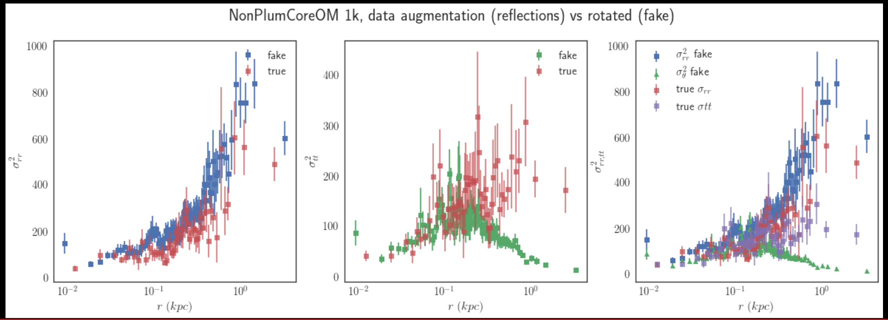
Bayesian prior: encode in the likelihood prior information (conditions) for the problem.
Deep learning: encode the prior information to your data (e.g. symmetries)

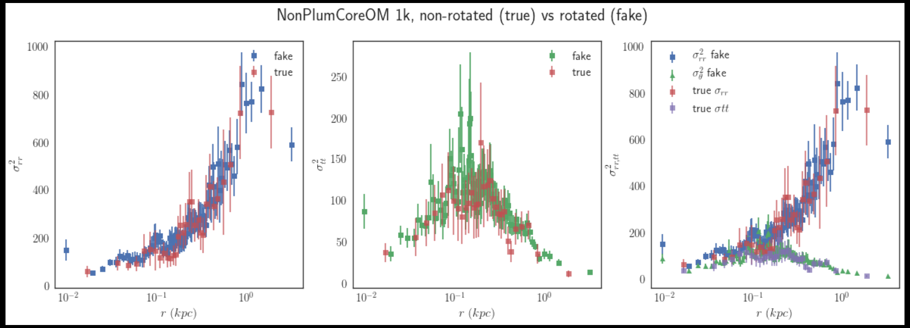


MAYDAY: make sure your transformation does not alter the content (label / regression value) of the input



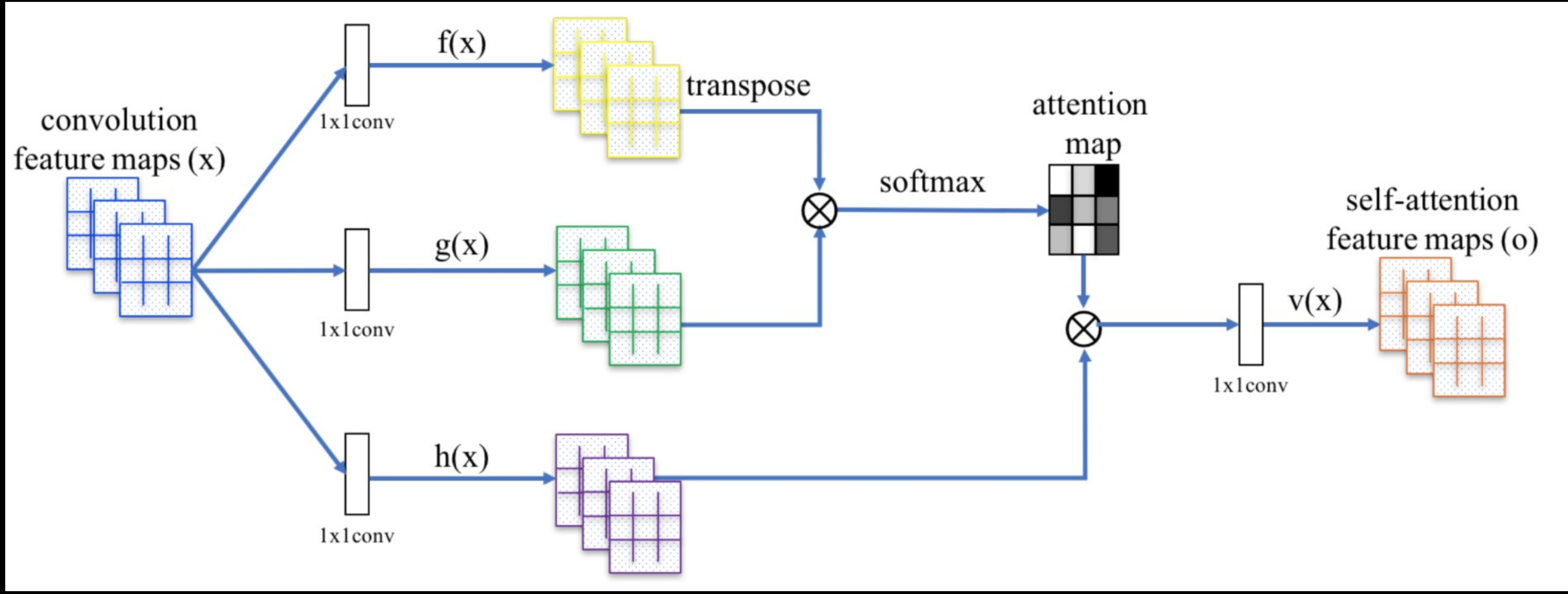






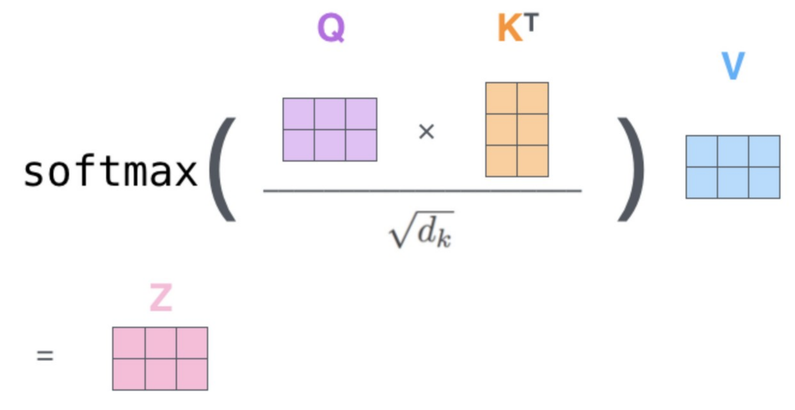
Practicalities 3: If possible, use (~~self~~) Attention (with caution)

Attention (NLP) Vaswani et al. (2017)



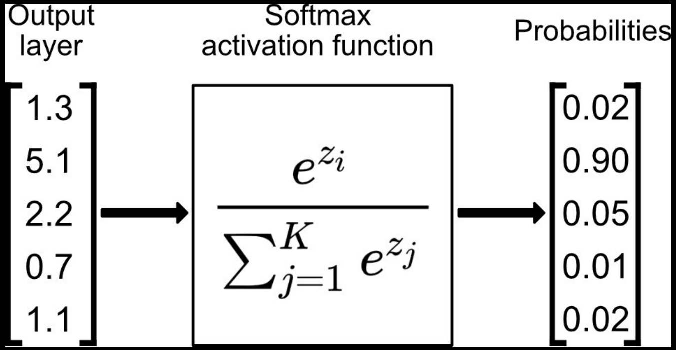
$$\mathbf{o} = \mathrm{softmax}\left(\frac{\mathbf{q} \circ_1 \mathbf{k}}{\sqrt{d}}\right), \qquad \in \Re^{C_q \times C}$$

$$\mathrm{Att}(\mathbf{q}, \mathbf{k}, \mathbf{v}) = \mathbf{o} \circ_2 \mathbf{v}, \qquad \in \Re^{C_q \times H \times W}$$



softmax (z): z → pseudo probs (their sum = 1.0)

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$



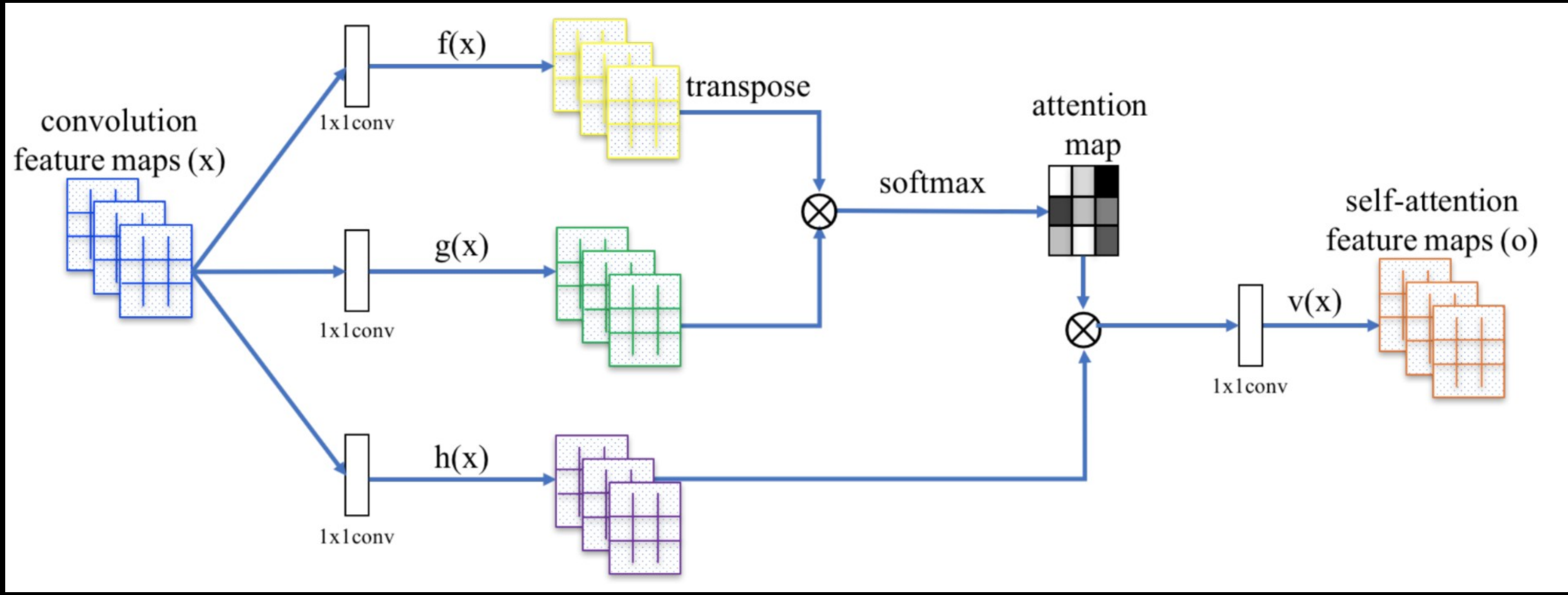
$$\mathbf{q} \circ_1 \mathbf{k} \equiv \sum_{jk} q_{ijk} k_{ljk} \in \Re^{C_q \times C}$$

$$\mathbf{o} \equiv \mathrm{softmax}\,(\mathbf{q} \circ_1 \mathbf{k}) \equiv o_{il}$$

$$\mathrm{Att}(\mathbf{q}, \mathbf{k}, \mathbf{v}) \equiv \mathrm{Att}_{ikj} \equiv \mathbf{o} \circ_2 \mathbf{v} \equiv \sum_{l} o_{il} v_{lkj} \in \Re^{C_q \times H \times W}$$

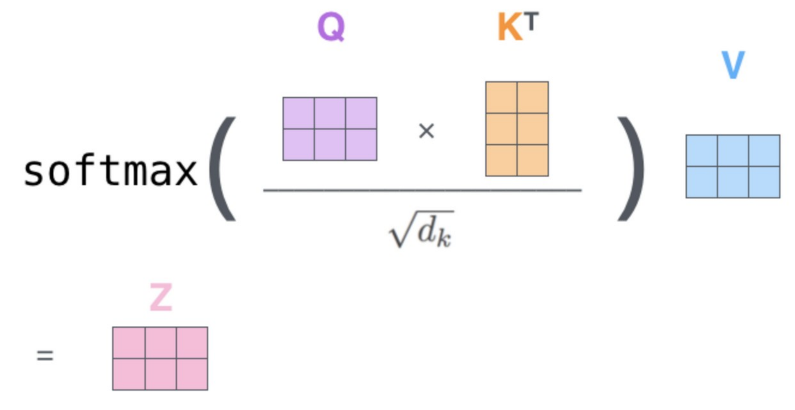
Practicalities 3: If possible, use (~~self~~) Attention (with caution)

Attention (NLP) Vaswani et al. (2017)

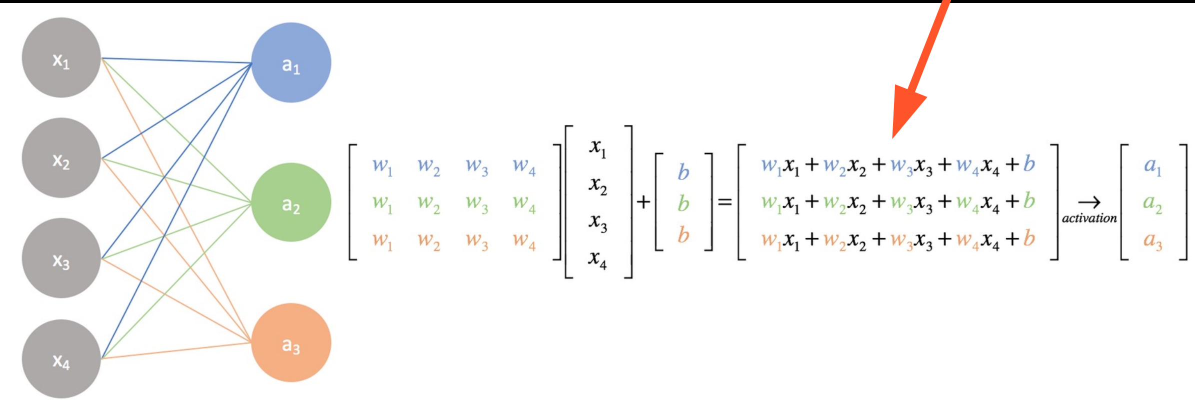


$$\mathbf{o} = \mathrm{softmax}\left(\frac{\mathbf{q} \circ_1 \mathbf{k}}{\sqrt{d}}\right), \qquad \in \Re^{C_q \times C}$$

$$\mathrm{Att}(\mathbf{q}, \mathbf{k}, \mathbf{v}) = \mathbf{o} \circ_2 \mathbf{v}, \qquad \in \Re^{C_q \times H \times W}$$



Information gets “washed away” in sums of many terms. Attention addresses this problem.



$$\mathbf{q} \circ_1 \mathbf{k} \equiv \sum_{jk} q_{ijk} k_{ljk} \in \Re^{C_q \times C}$$

$$\mathbf{o} \equiv \mathrm{softmax}(\mathbf{q} \circ_1 \mathbf{k}) \equiv o_{il}$$

$$\mathrm{Att}(\mathbf{q}, \mathbf{k}, \mathbf{v}) \equiv \mathrm{Att}_{ikj} \equiv \mathbf{o} \circ_2 \mathbf{v} \equiv \sum_{l} o_{il} v_{lkj} \in \Re^{C_q \times H \times W}$$

Practicalities 3: If possible, use (~~self~~) Attention (with caution)

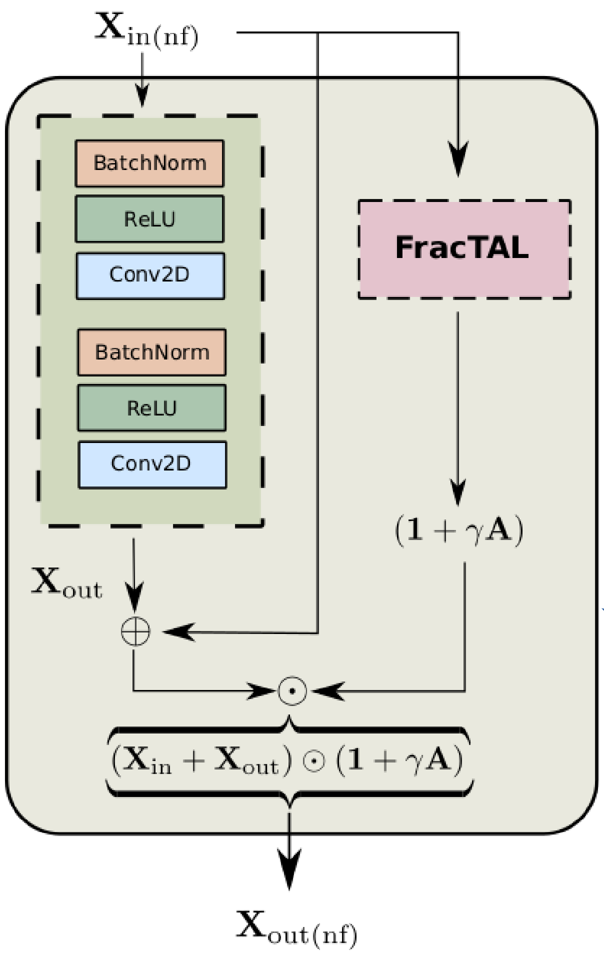


Figure 4: The FracTAL Residual unit. This building block demonstrates the fusion of the residual block with self FracTAL evaluated from the input features.

How you will decide to fuse the Attention (emphasis) with the 1D/2D/3D input layer, can make (**or break!**) your model.

Looking for change? Roll the Dice and demand Attention

Foivos I. Diakogiannis[a,b,1], François Waldner[c], Peter Caccetta[b]

[a] *ICRAR, the University of Western Australia*
[b] *Data61, CSIRO, Floreat WA*
[c] *CSIRO Agriculture & Food, St Lucia, QLD, Australia*

```
In [1]: for _ in range(4):
   ...:     print("Thank you!!!")
```