

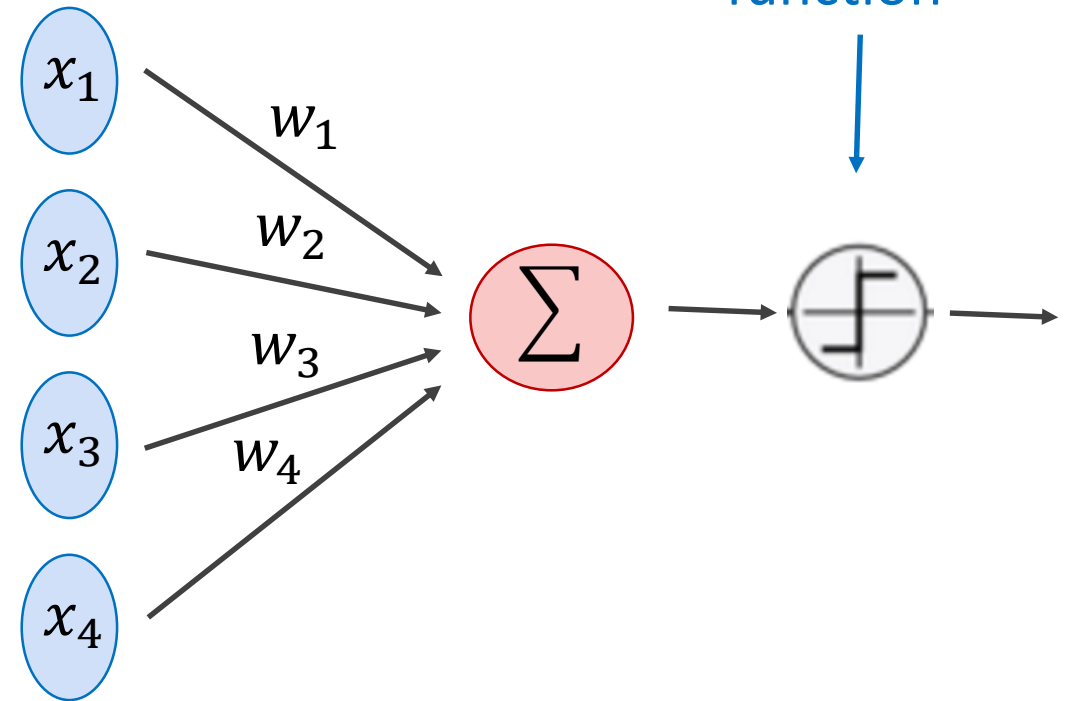
Neural networks and CNN

Biplab Banerjee

Perceptron Model

Frank Rosenblatt (1957) - Cornell University

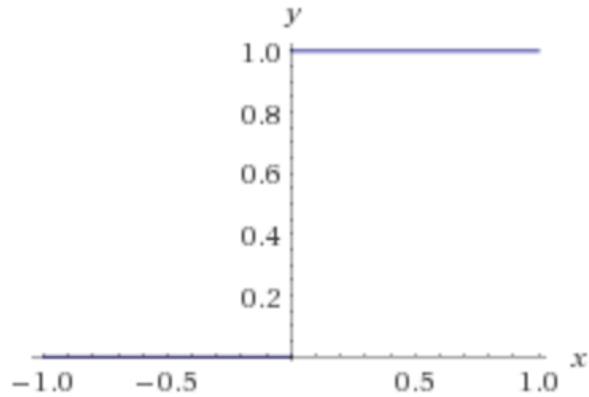
$$f(x) = \begin{cases} 1, & \text{if } \sum_{i=0}^n w_i x_i + b > 0 \\ 0, & \text{otherwise} \end{cases}$$



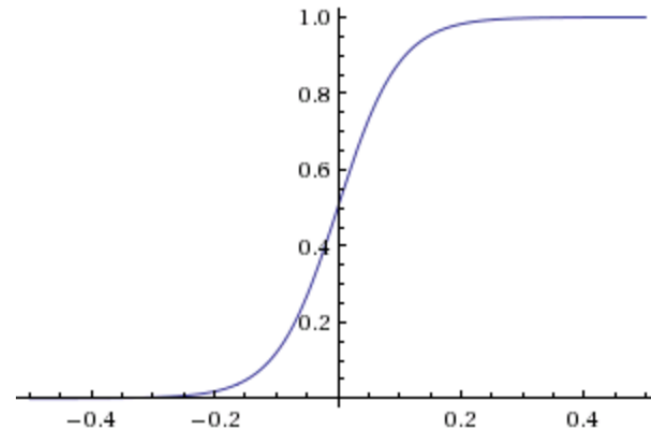
More: <https://en.wikipedia.org/wiki/Perceptron>

Activation Functions

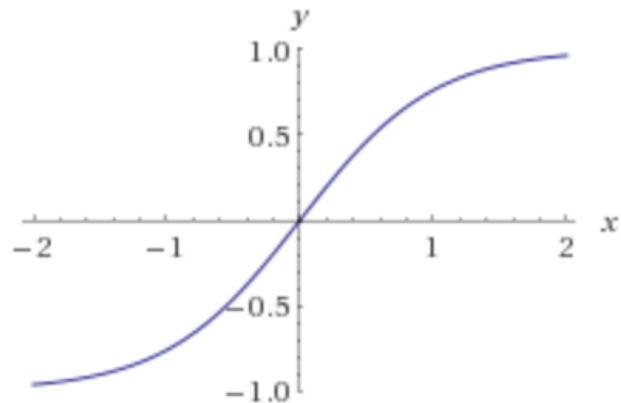
Step(x)



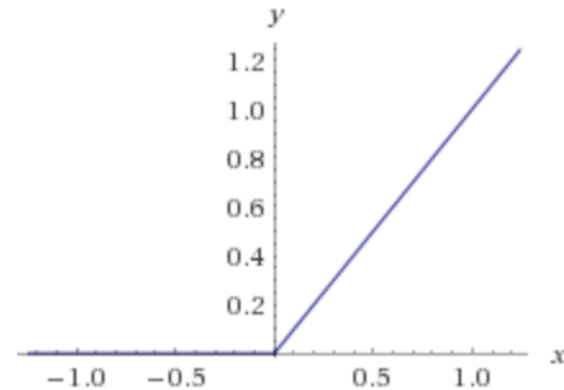
Sigmoid(x)



Tanh(x)



$\text{ReLU}(x) = \max(0, x)$



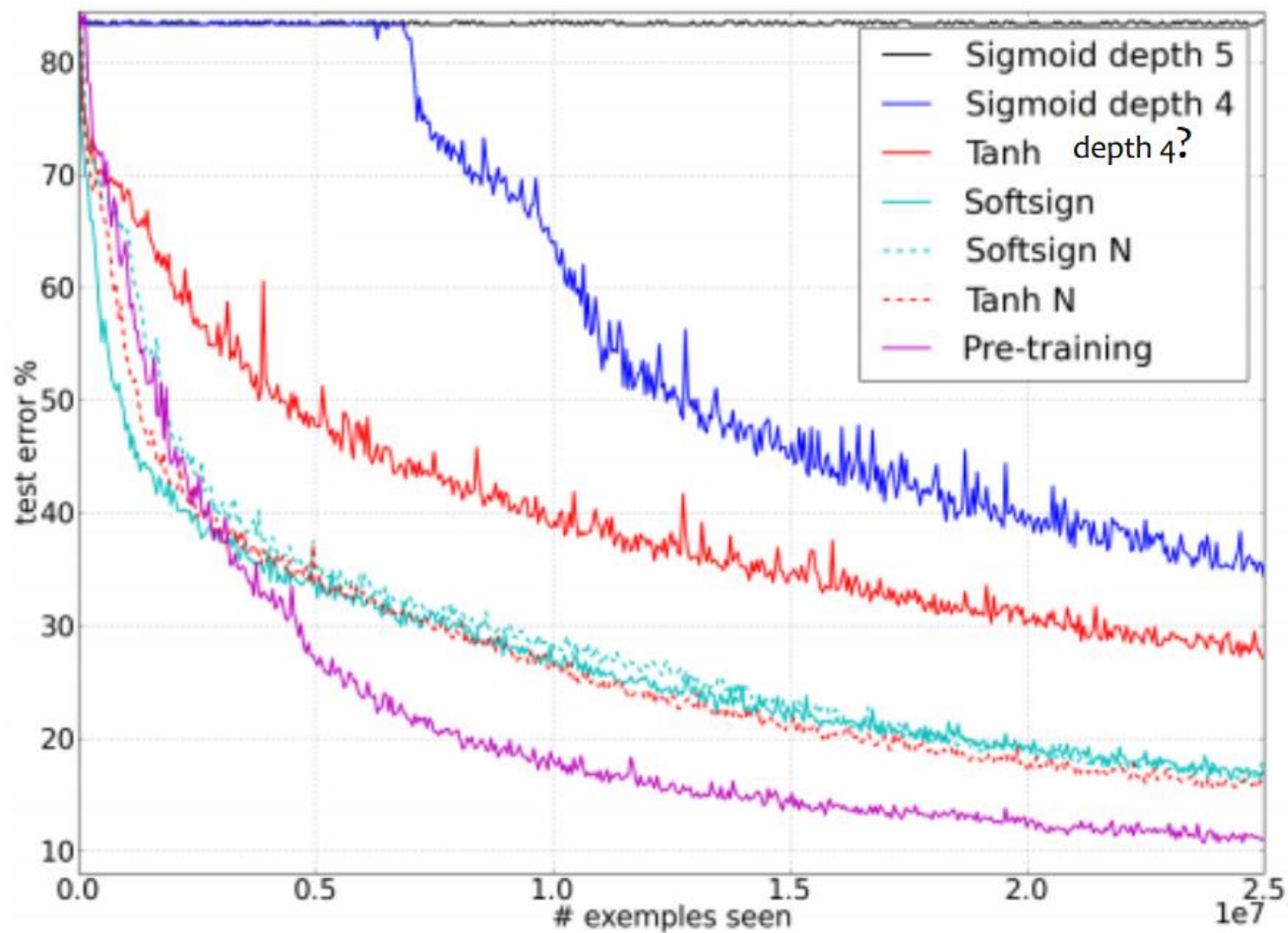
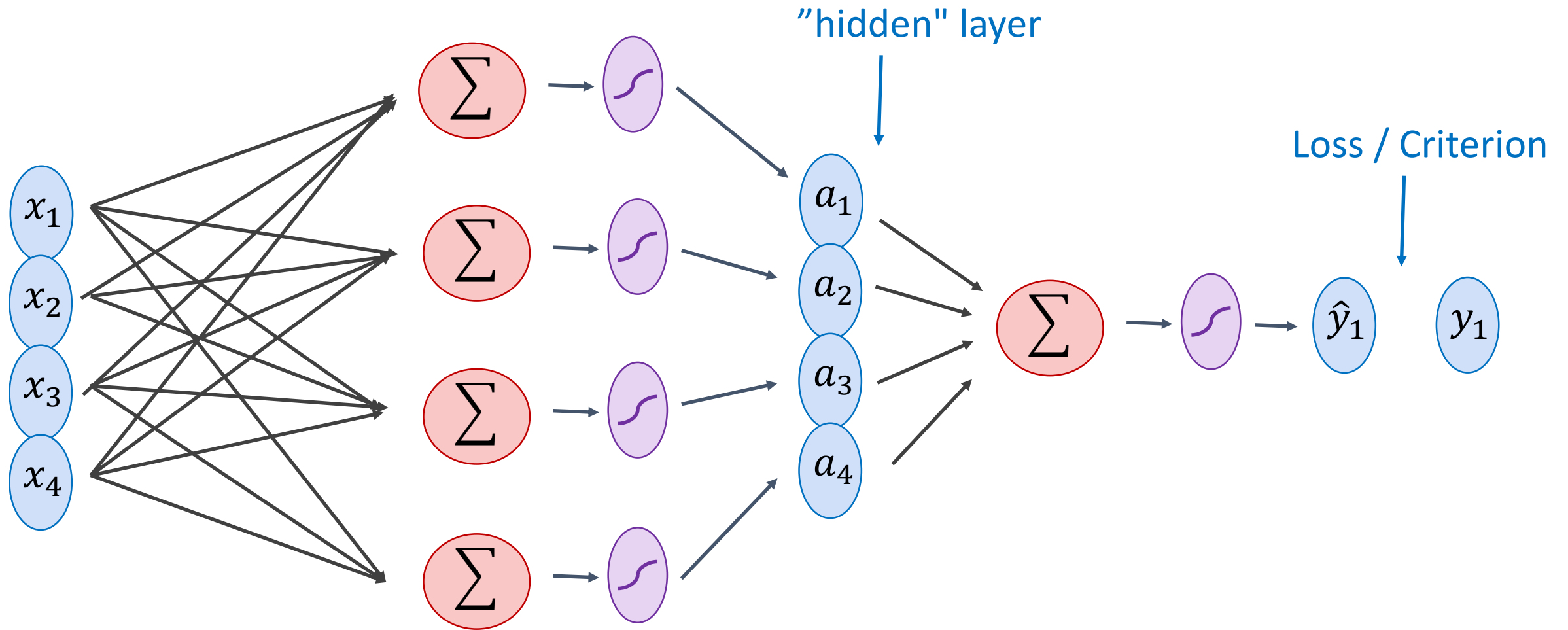


Figure from Glorot & Bentio (2010)

Two-layer Multi-layer Perceptron (MLP)



Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g_c = w_{c1}x_{i1} + w_{c2}x_{i2} + w_{c3}x_{i3} + w_{c4}x_{i4} + b_c$$

$$g_d = w_{d1}x_{i1} + w_{d2}x_{i2} + w_{d3}x_{i3} + w_{d4}x_{i4} + b_d$$

$$g_b = w_{b1}x_{i1} + w_{b2}x_{i2} + w_{b3}x_{i3} + w_{b4}x_{i4} + b_b$$

$$f_c = e^{g_c} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_d = e^{g_d} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_b = e^{g_b} / (e^{g_c} + e^{g_d} + e^{g_b})$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g_c = w_{c1}x_{i1} + w_{c2}x_{i2} + w_{c3}x_{i3} + w_{c4}x_{i4} + b_c$$

$$g_d = w_{d1}x_{i1} + w_{d2}x_{i2} + w_{d3}x_{i3} + w_{d4}x_{i4} + b_d$$

$$g_b = w_{b1}x_{i1} + w_{b2}x_{i2} + w_{b3}x_{i3} + w_{b4}x_{i4} + b_b$$

$$W = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix}$$

$$b = [b_c \ b_d \ b_b]$$

$$f_c = e^{g_c} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_d = e^{g_d} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_b = e^{g_b} / (e^{g_c} + e^{g_d} + e^{g_b})$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g = wx^T + b^T$$

$$w = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix}$$

$$b = [b_c \ b_d \ b_b]$$

$$f_c = e^{g_c} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_d = e^{g_d} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_b = e^{g_b} / (e^{g_c} + e^{g_d} + e^{g_b})$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g = wx^T + b^T$$

$$w = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix}$$

$$b = [b_c \ b_d \ b_b]$$

$$f = \text{softmax}(g)$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$f = \text{softmax}(wx^T + b^T)$$

Two-layer MLP + Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$a_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T)$$

$$f = \text{softmax}(w_{[2]}x^T + b_{[2]}^T)$$

N-layer MLP + Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$a_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T)$$

$$a_2 = \text{sigmoid}(w_{[2]}a_1^T + b_{[2]}^T)$$

...

$$a_k = \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}^T)$$

...

$$f = \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}^T)$$

How to train the parameters?

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$a_1 = \textit{sigmoid}(w_{[1]}x^T + b_{[1]}^T)$$

$$a_2 = \textit{sigmoid}(w_{[2]}a_1^T + b_{[2]}^T)$$

...

$$a_k = \textit{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}^T)$$

...

$$f = \textit{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}^T)$$

How to train the parameters?

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$a_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T)$$

$$a_2 = \text{sigmoid}(w_{[2]}a_1^T + b_{[2]}^T)$$

...

$$a_k = \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}^T)$$

...

$$f = \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}^T)$$

$$l = \text{loss}(f, y)$$

We can still use SGD

We need!

$$\frac{\partial l}{\partial w_{[k]ij}}$$

$$\frac{\partial l}{\partial b_{[k]i}}$$

- Regression:
 - Use the same objective as Linear Regression
 - Quadratic loss (i.e. mean squared error)
- Classification:
 - Use the same objective as Logistic Regression
 - Cross-entropy (i.e. negative log likelihood)
 - This requires probabilities, so we add an additional “softmax” layer at the end of our network

	Forward	Backward
Quadratic	$J = \frac{1}{2}(y - y^*)^2$	$\frac{dJ}{dy} = y - y^*$
Cross Entropy	$J = y^* \log(y) + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = y^* \frac{1}{y} + (1 - y^*) \frac{1}{y - 1}$

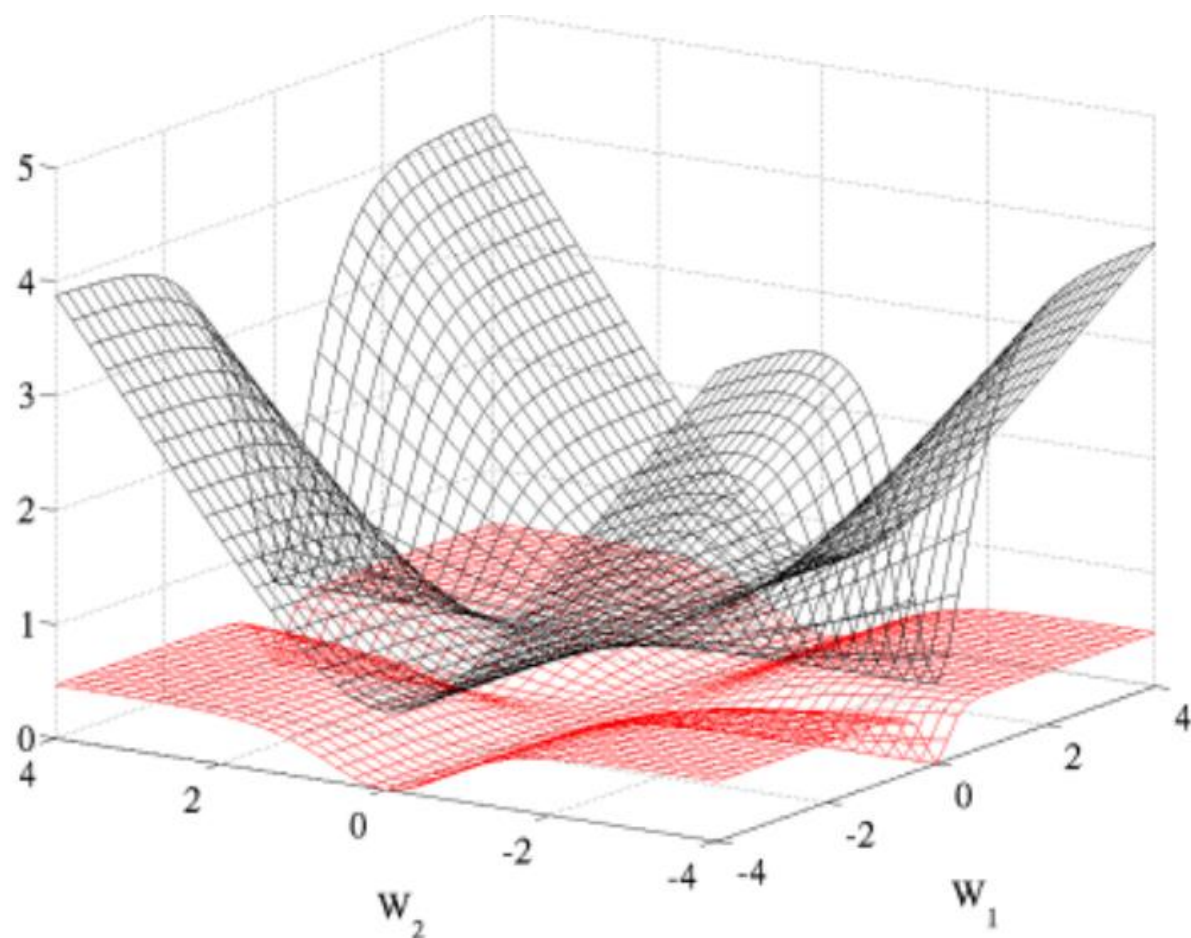


Figure 5: Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer.

Background

A Recipe for Machine Learning

1. Given training data:

$$\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$$

2. Choose each of these:

– Decision function

$$\hat{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x}_i)$$

– Loss function

$$\ell(\hat{\mathbf{y}}, \mathbf{y}_i) \in \mathbb{R}$$

3. Define goal:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

4. Train with SGD:

(take small steps
opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

1. Finite Difference Method

- Pro: Great for testing implementations of backpropagation
- Con: Slow for high dimensional inputs / outputs
- Required: Ability to call the function $f(\mathbf{x})$ on any input \mathbf{x}

2. Symbolic Differentiation

- Note: The method you learned in high-school
- Note: Used by Mathematica / Wolfram Alpha / Maple
- Pro: Yields easily interpretable derivatives
- Con: Leads to exponential computation time if not carefully implemented
- Required: Mathematical expression that defines $f(\mathbf{x})$

3. Automatic Differentiation - Reverse Mode

- Note: Called *Backpropagation* when applied to Neural Nets
- Pro: Computes partial derivatives of one output $f(\mathbf{x})_i$ with respect to all inputs x_j in time proportional to computation of $f(\mathbf{x})$
- Con: Slow for high dimensional outputs (e.g. vector-valued functions)
- Required: Algorithm for computing $f(\mathbf{x})$

4. Automatic Differentiation - Forward Mode

- Note: Easy to implement. Uses dual numbers.
- Pro: Computes partial derivatives of all outputs $f(\mathbf{x})_i$ with respect to one input x_j in time proportional to computation of $f(\mathbf{x})$
- Con: Slow for high dimensional inputs (e.g. vector-valued \mathbf{x})
- Required: Algorithm for computing $f(\mathbf{x})$

Given $f : \mathbb{R}^A \rightarrow \mathbb{R}^B, f(\mathbf{x})$

Compute $\frac{\partial f(\mathbf{x})_i}{\partial x_j} \forall i, j$

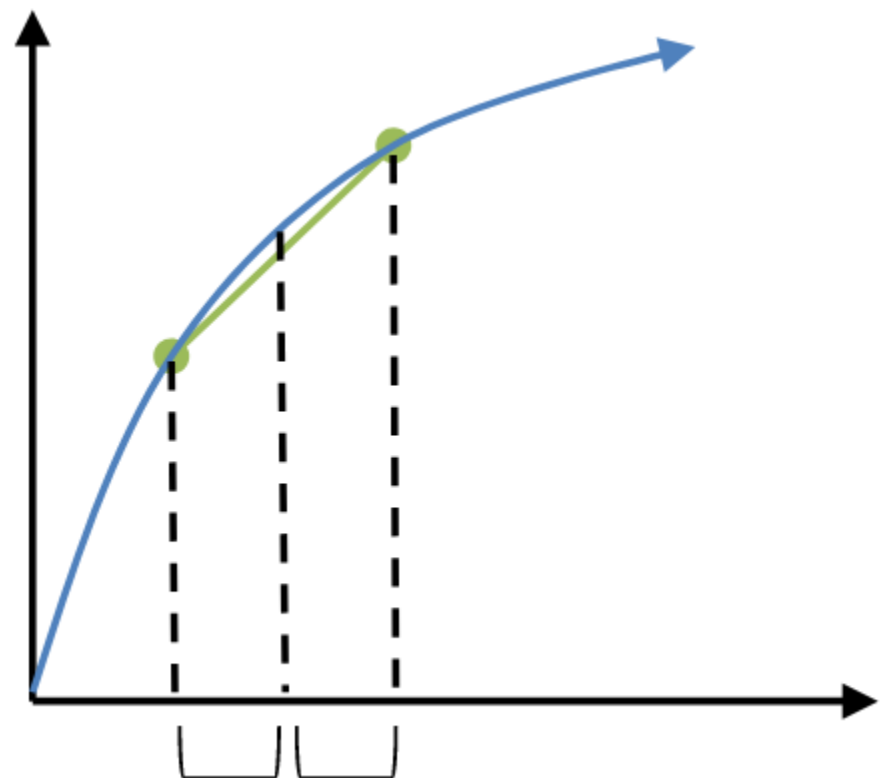
The centered finite difference approximation is:

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{(J(\boldsymbol{\theta} + \epsilon \cdot \mathbf{d}_i) - J(\boldsymbol{\theta} - \epsilon \cdot \mathbf{d}_i))}{2\epsilon}$$

where \mathbf{d}_i is a 1-hot vector consisting of all zeros except for the i th entry of \mathbf{d}_i , which has value 1.

Notes:

- Suffers from issues of floating point precision, in practice
- Typically only appropriate to use on small examples with an appropriately chosen epsilon

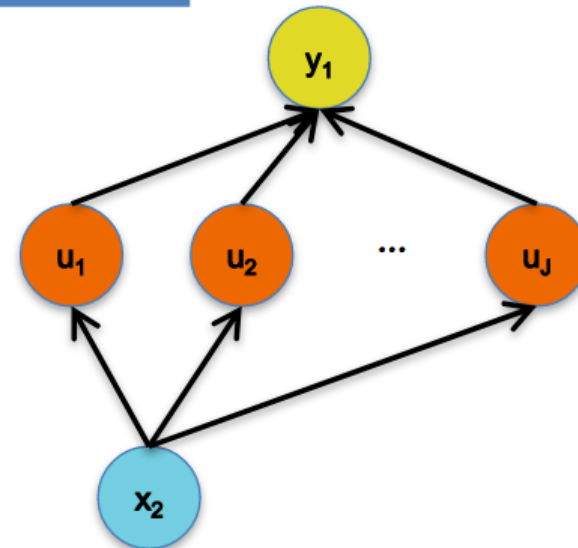


Backpropagation – repeated application of chain rule

Given: $y = g(u)$ and $u = h(x)$.

Chain Rule:

$$\frac{dy_i}{dx_k} = \sum_{j=1}^J \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$



Forward Computation

1. Write an **algorithm** for evaluating the function $y = f(\mathbf{x})$. The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.
For variable u_i with inputs v_1, \dots, v_N
 - a. Compute $u_i = g_i(v_1, \dots, v_N)$
 - b. Store the result at the node

Backward Computation

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
For variable $u_i = g_i(v_1, \dots, v_N)$
 - a. We already know dy/du_i
 - b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j)$
(Choice of algorithm ensures computing (du_i/dv_j) is easy)

Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward

$$J = \cos(u)$$

$$u = u_1 + u_2$$

$$u_1 = \sin(t)$$

$$u_2 = 3t$$

$$t = x^2$$

Backward

$$\frac{dJ}{du} += -\sin(u)$$

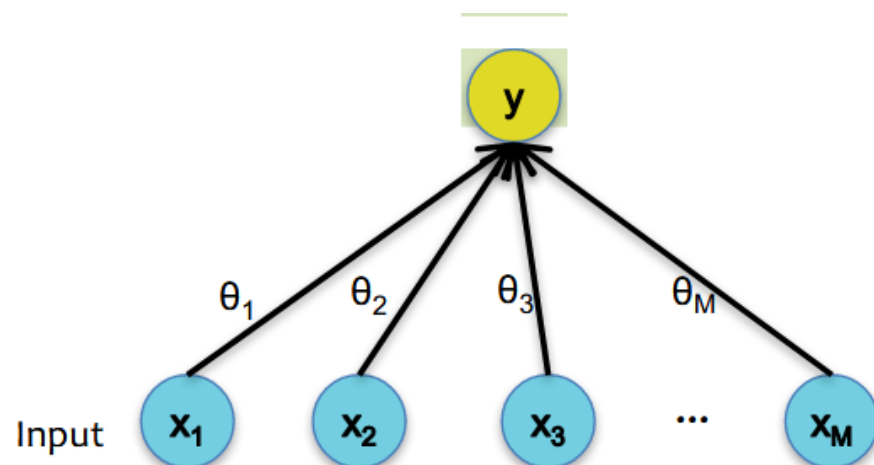
$$\frac{dJ}{du_1} += \frac{dJ}{du} \frac{du}{du_1}, \quad \frac{du}{du_1} = 1 \qquad \frac{dJ}{du_2} += \frac{dJ}{du} \frac{du}{du_2}, \quad \frac{du}{du_2} = 1$$

$$\frac{dJ}{dt} += \frac{dJ}{du_1} \frac{du_1}{dt}, \quad \frac{du_1}{dt} = \cos(t)$$

$$\frac{dJ}{dt} += \frac{dJ}{du_2} \frac{du_2}{dt}, \quad \frac{du_2}{dt} = 3$$

$$\frac{dJ}{dx} += \frac{dJ}{dt} \frac{dt}{dx}, \quad \frac{dt}{dx} = 2x$$

Logistic Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^D \theta_j x_j$$

Backward

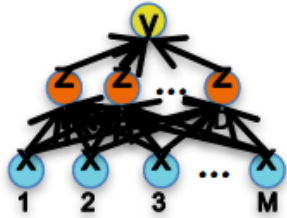
$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{da} = \frac{dJ}{dy} \frac{dy}{da}, \quad \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

$$\frac{dJ}{d\theta_j} = \frac{dJ}{da} \frac{da}{d\theta_j}, \quad \frac{da}{d\theta_j} = x_j$$

$$\frac{dJ}{dx_j} = \frac{dJ}{da} \frac{da}{dx_j}, \quad \frac{da}{dx_j} = \theta_j$$

Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^D \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^M \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy} \frac{dy}{db}, \quad \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j}, \quad \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db} \frac{db}{dz_j}, \quad \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j}, \quad \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}}, \quad \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{da_j} \frac{da_j}{dx_i}, \quad \frac{da_j}{dx_i} = \sum_{j=0}^D \alpha_{ji}$$

Two-layer Neural Network – Forward Pass

```
# Setup the input variable x.
img, label = trainset[0]
x = img.view(1, 1 * 28 * 28)

# Setup the number of inputs, hidden neurons, and outputs.
nInputs = 1 * 28 * 28
nHidden = 512
nOutputs = 10

# Create the model here.
linear_fn1 = toynn_Linear(nInputs, nHidden)
relu_fn = toynn_ReLU()
linear_fn2 = toynn_Linear(nHidden, nOutputs)

# Make predictions.
x = linear_fn1.forward(x)
x = relu_fn.forward(x)
x = linear_fn2.forward(x)

# Show the prediction scores for each class.
# Yes, pytorch tensors already come with a softmax function.
# We need it here because we hard-coded the softmax inside
# the loss function.
print(x.softmax(dim = 1))
```

Two-layer Neural Network – Backward Pass

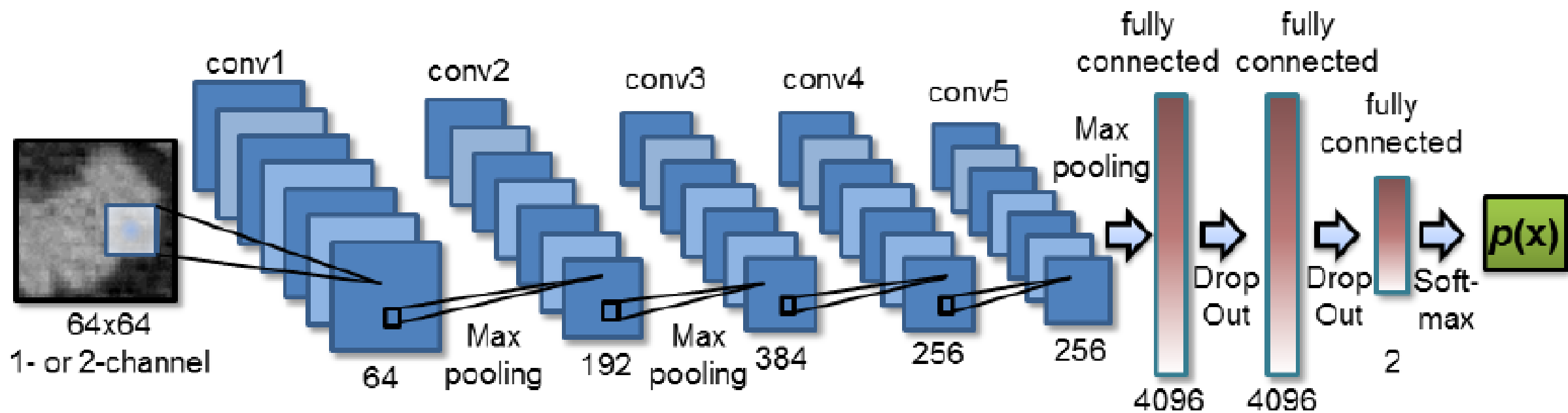
```
# Create the model here.
linear_fn1 = toynn_Linear(nInputs, nHidden)
relu_fn = toynn_ReLU()
linear_fn2 = toynn_Linear(nHidden, nOutputs)
loss_fn = toynn_CrossEntropyLoss()

# Make predictions (forward pass).
a = linear_fn1.forward(x)
z = relu_fn.forward(a)
yhat = linear_fn2.forward(z)

# Compute loss.
loss = loss_fn.forward(yhat, label)
yhat_grads = loss_fn.backward(yhat, label)

# Compute gradients (backward pass).
z_grads = linear_fn2.backward(z, yhat_grads)
a_grads = relu_fn.backward(a, z_grads)
x_grads = linear_fn1.backward(x, a_grads)

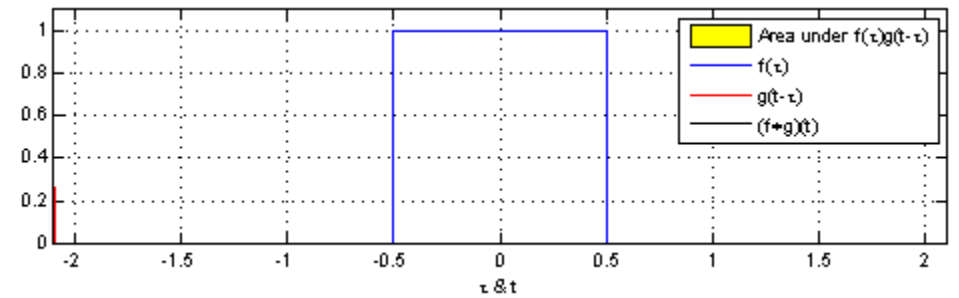
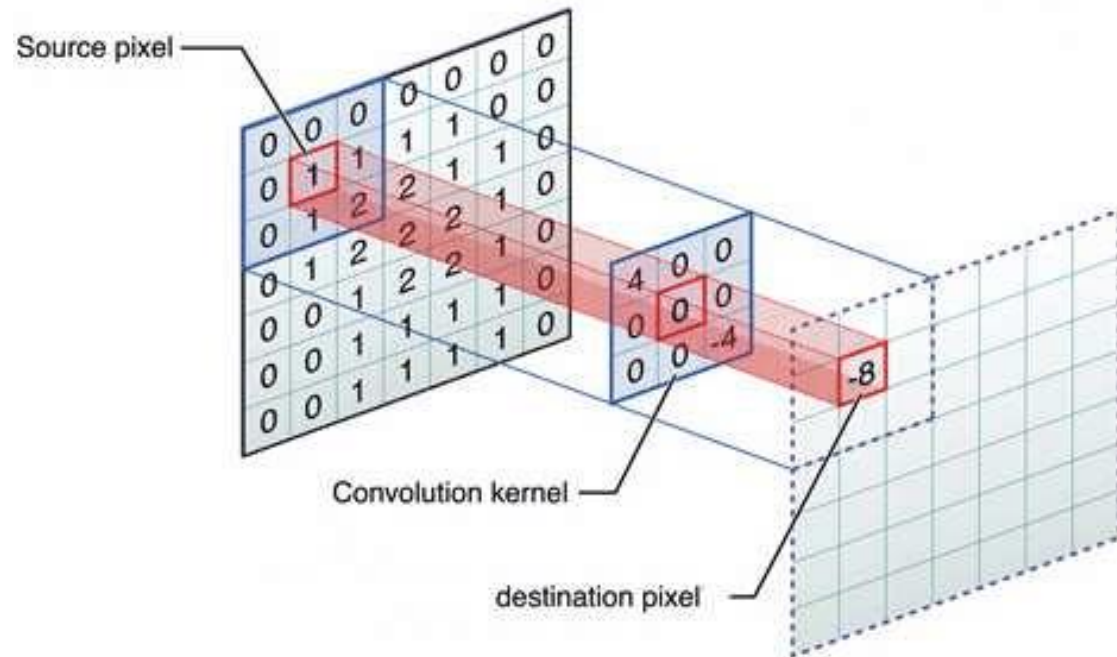
# Update parameters:
learningRate = 0.2
linear_fn1.weight.add_(-learningRate, linear_fn1.weight_grads)
linear_fn1.bias.add_(-learningRate, linear_fn1.bias_grads)
linear_fn2.weight.add_(-learningRate, linear_fn2.weight_grads)
linear_fn2.bias.add_(-learningRate, linear_fn2.bias_grads)
```



Basic building blocks of the CNN architecture

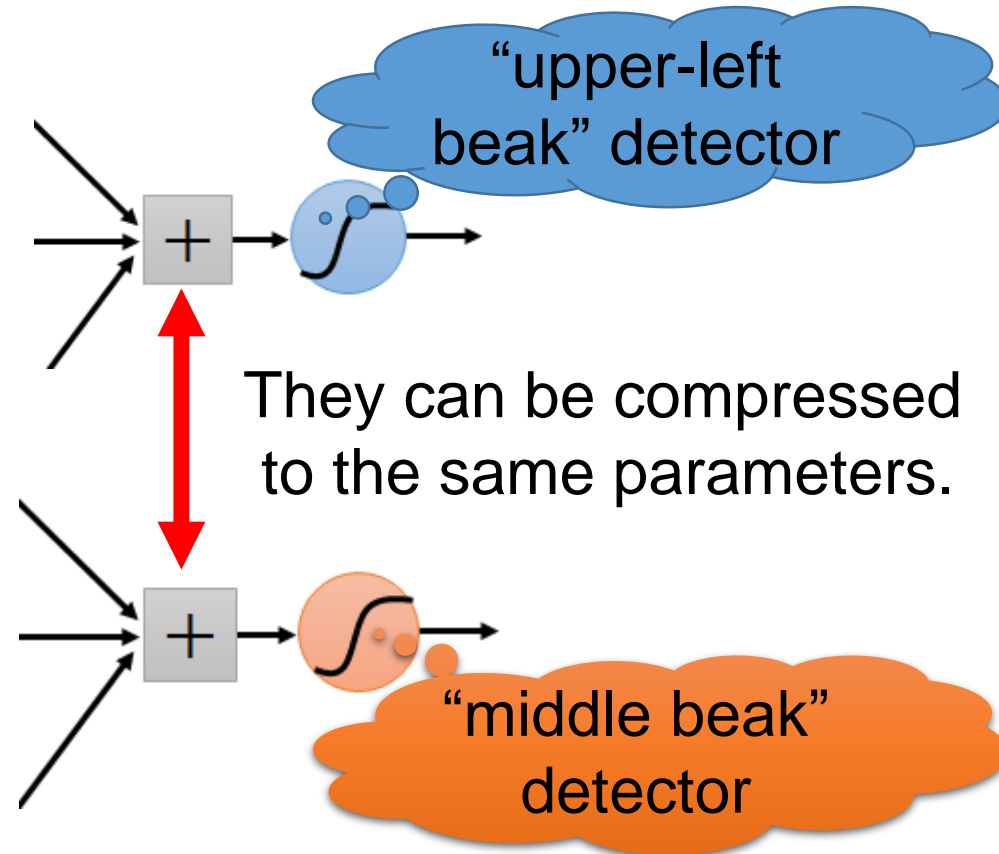
- Input layer
 - Convolutional layer
 - Fully connected layer
 - Loss layer
-
- Convolutional layer
 - Convolutional kernel
 - Pooling layer
 - Non-linearity

Convolution operation

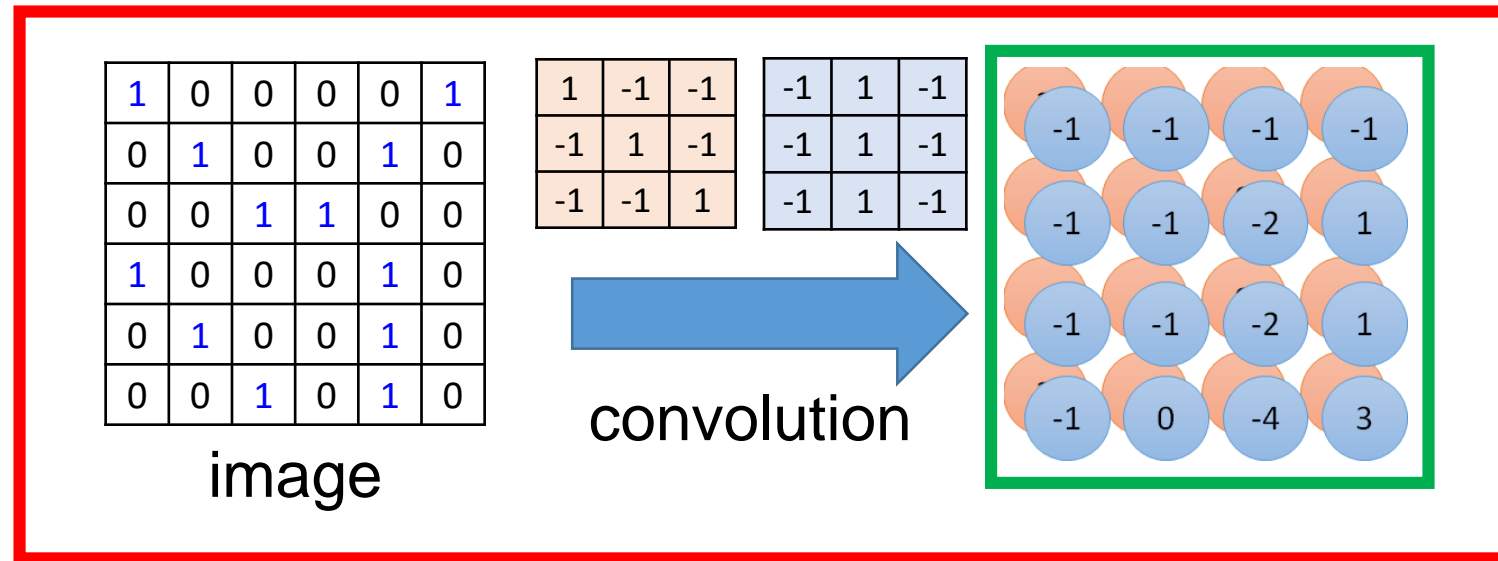


Same pattern appears in different places:
They can be compressed!

What about training a lot of such “small” detectors
and each detector must “move around”.

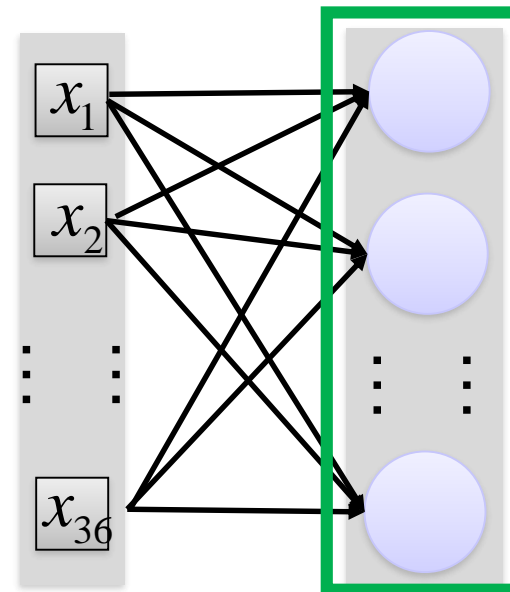


Convolution v.s. Fully Connected

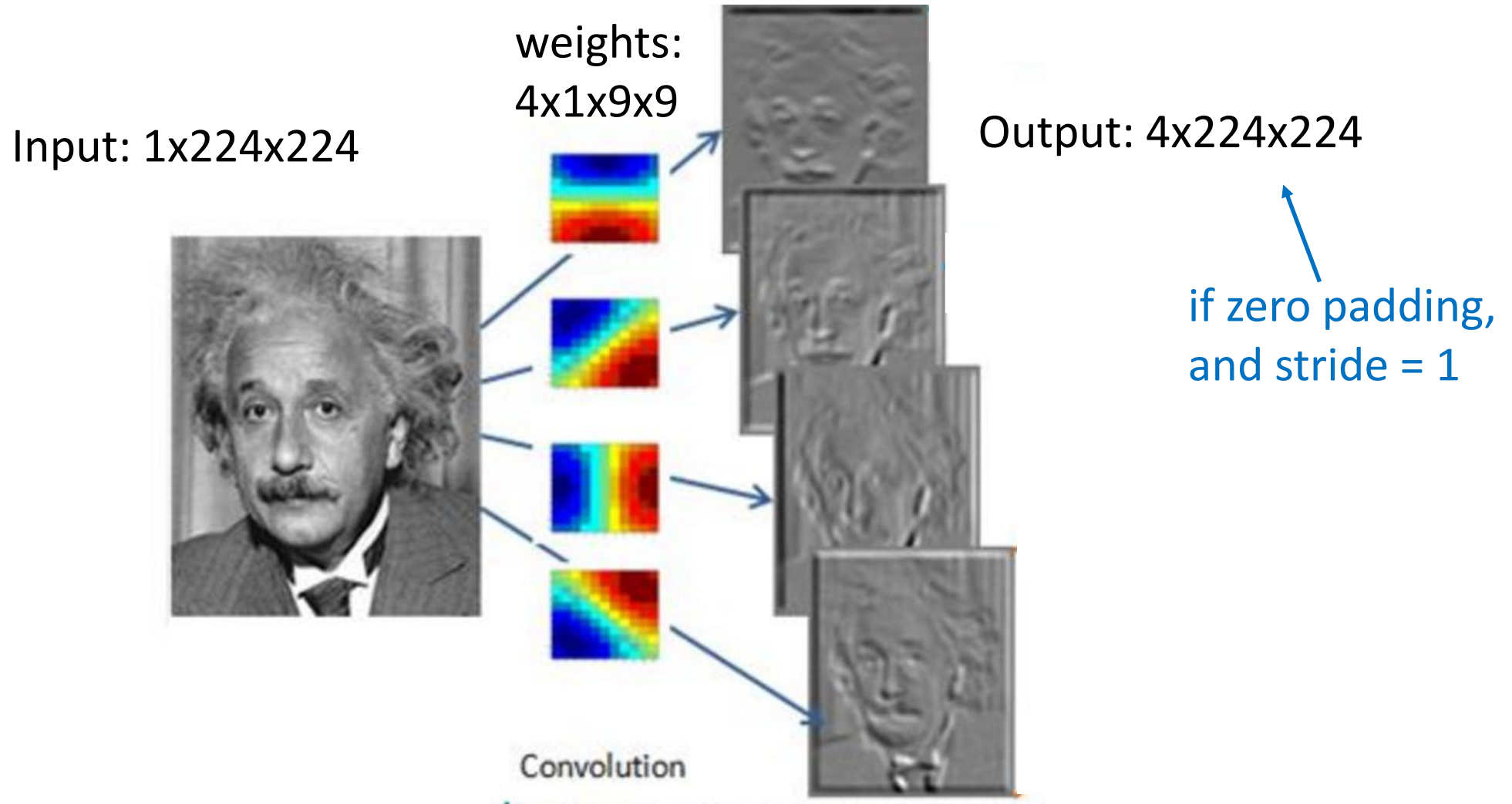


Fully-
connected

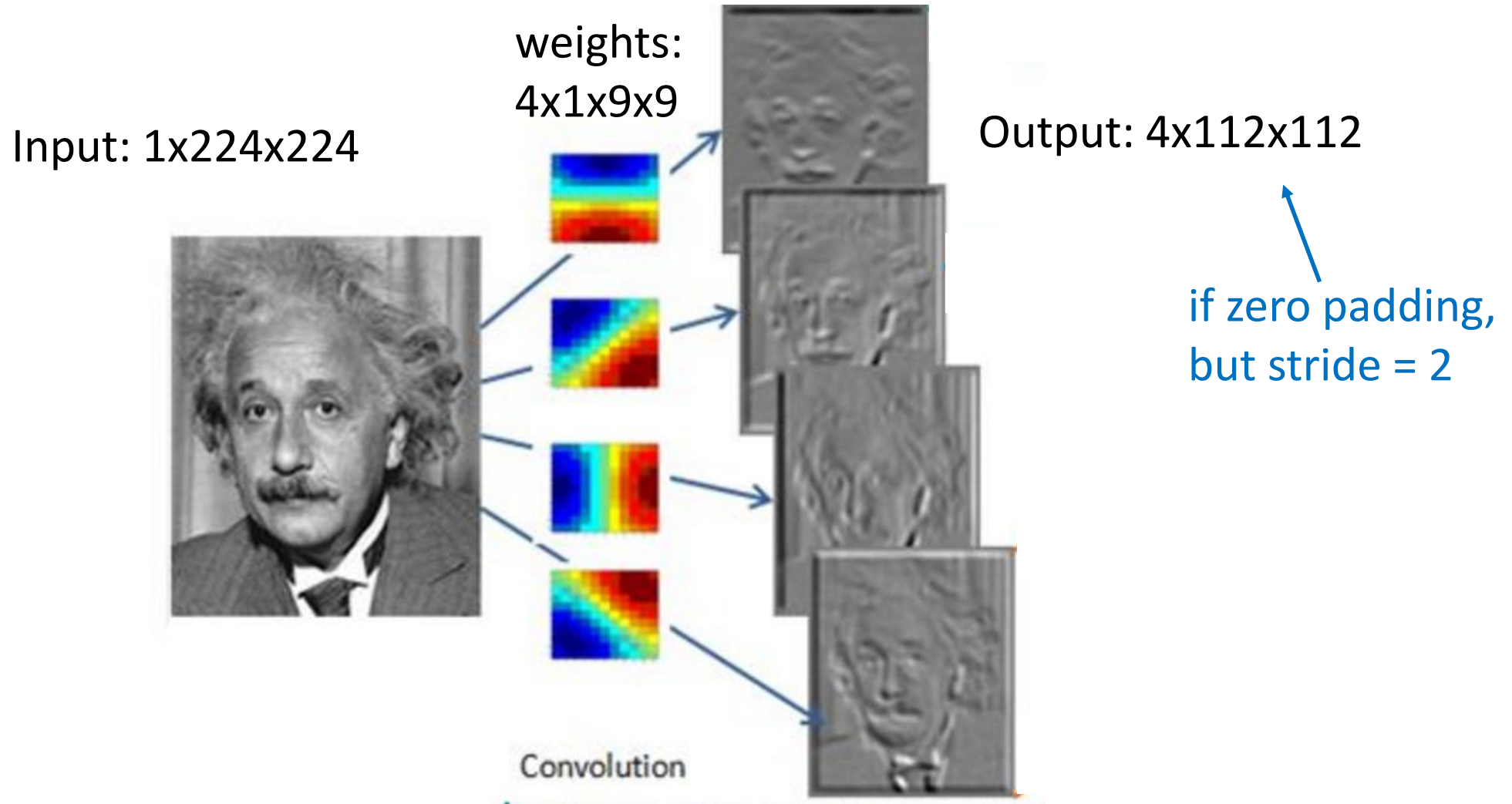
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0



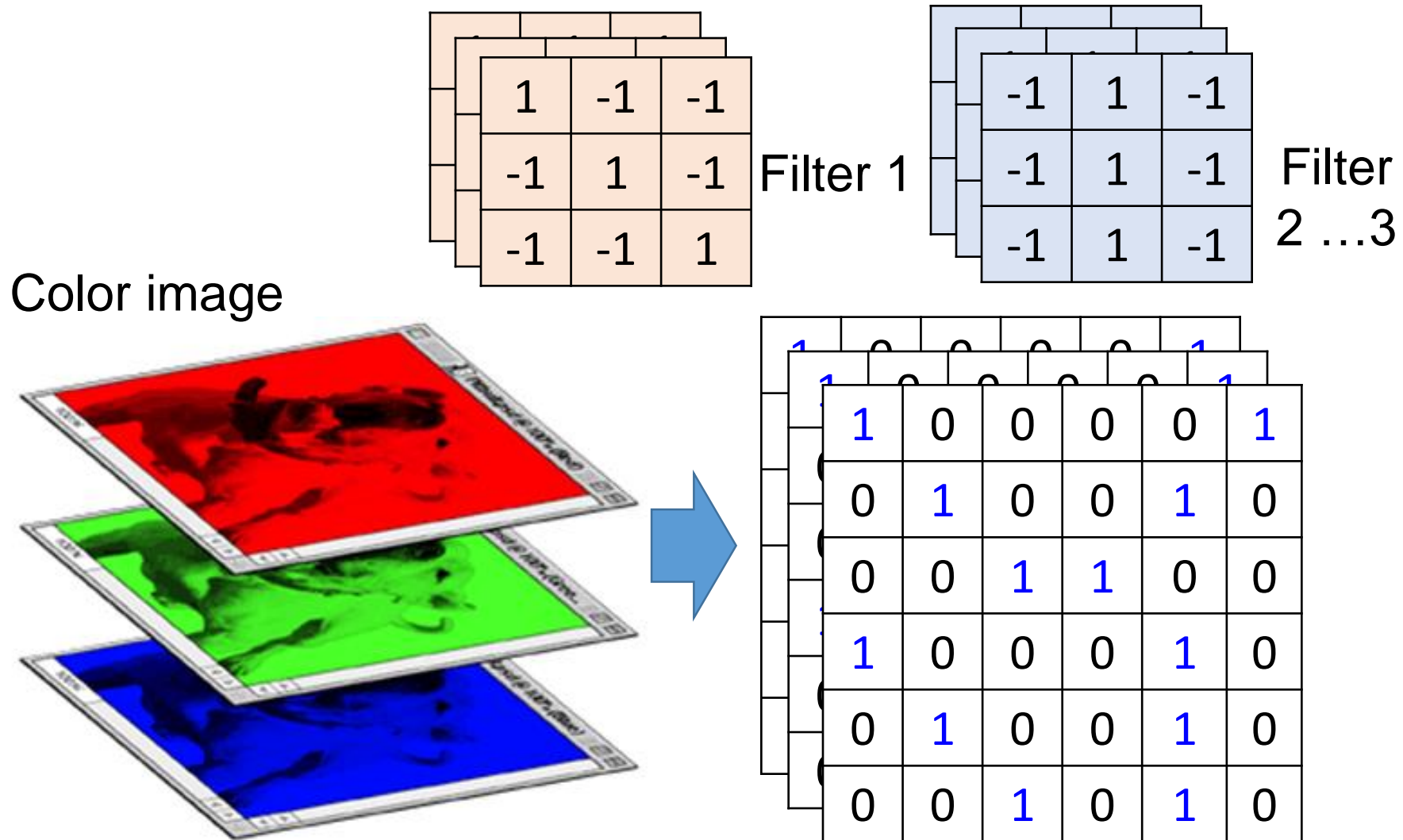
Convolutional Layer (with 4 filters)



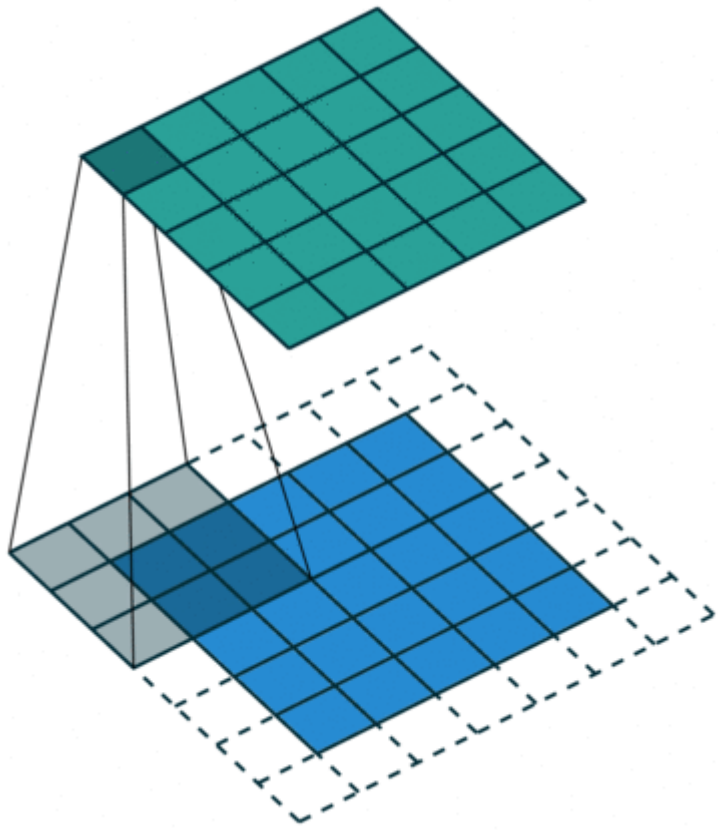
Convolutional Layer (with 4 filters)



Color image: RGB 3 channels – conv. over depth



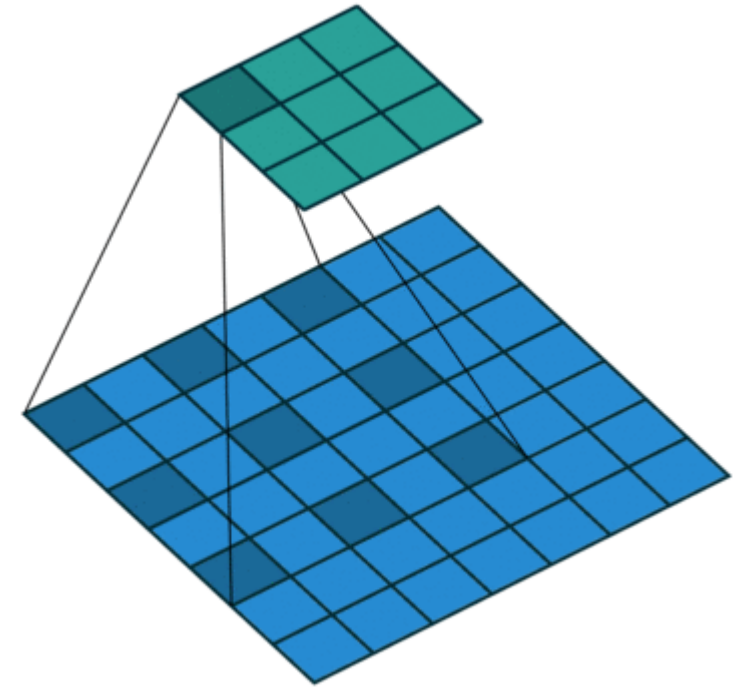
Different types of convolution



Parameters:

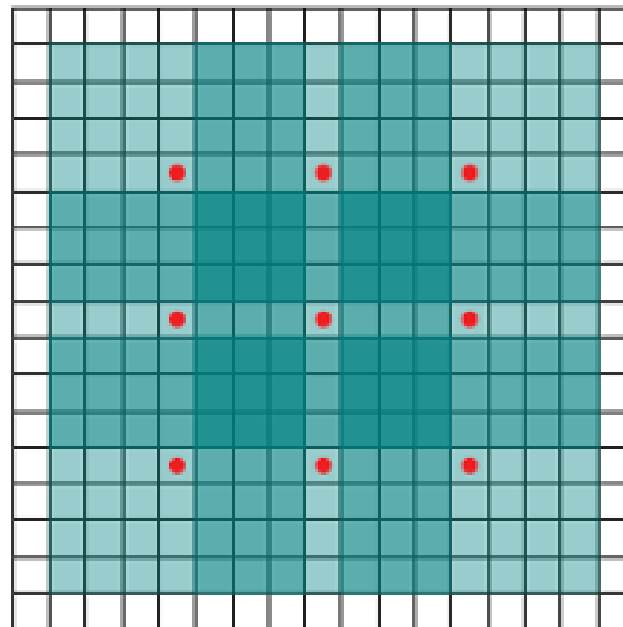
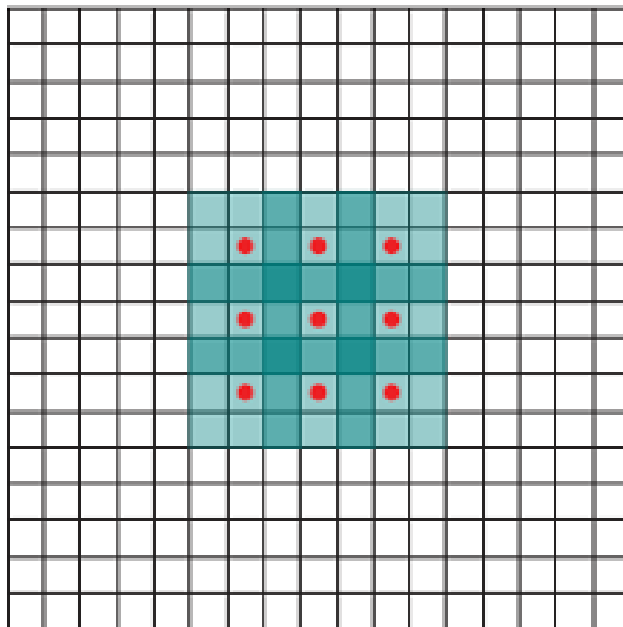
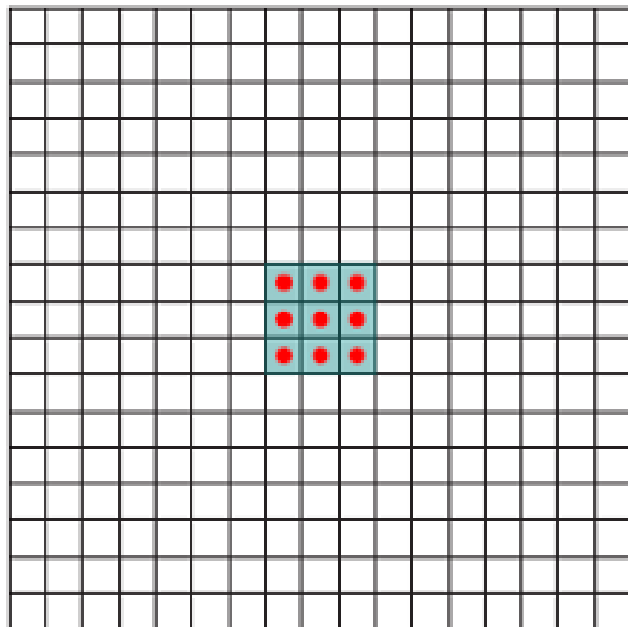
- ✓ Kernel stride
- ✓ Size
- ✓ Padding

Normal vs dialated



Dilation width = 2

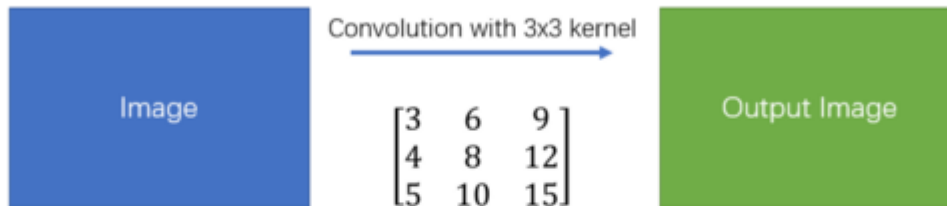
Dilated convolution



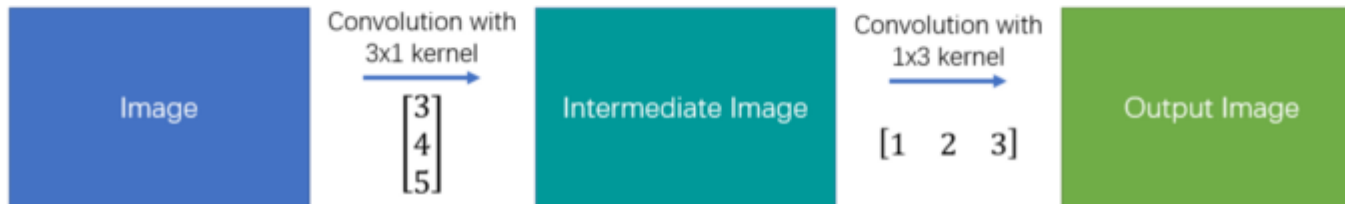
Spatially Separable convolution

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

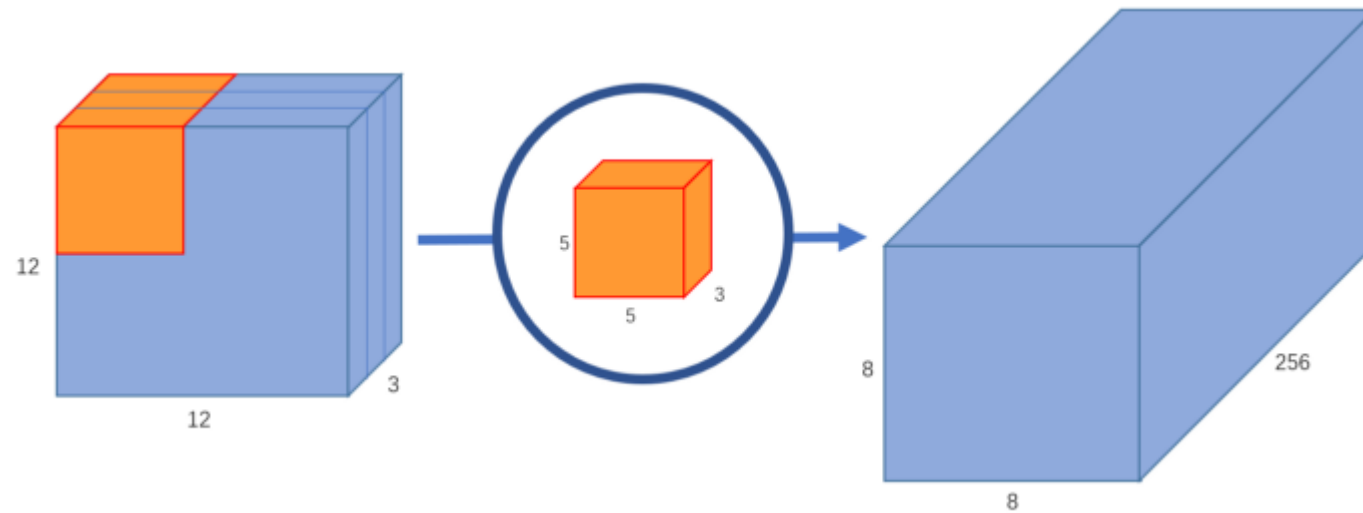
Simple Convolution



Spatial Separable Convolution



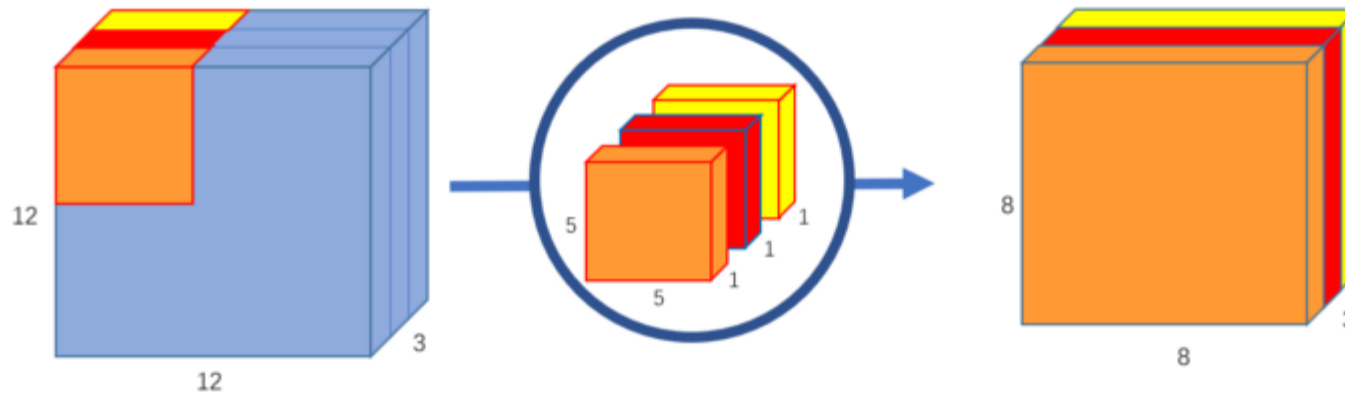
Depthwise separable convolution



Convolving by 256 5x5 kernels over the input volume

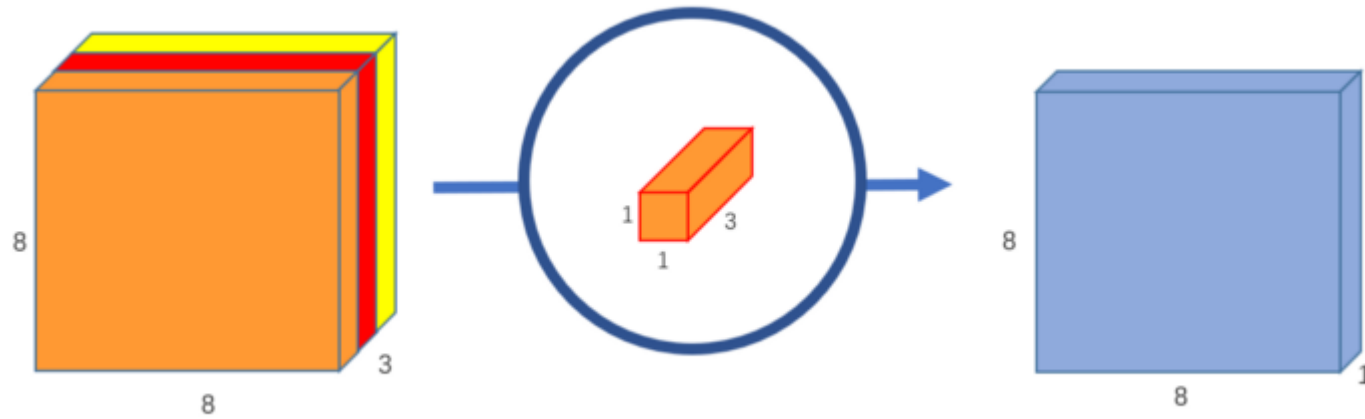
Depthwise separable convolution – step1

Along depth

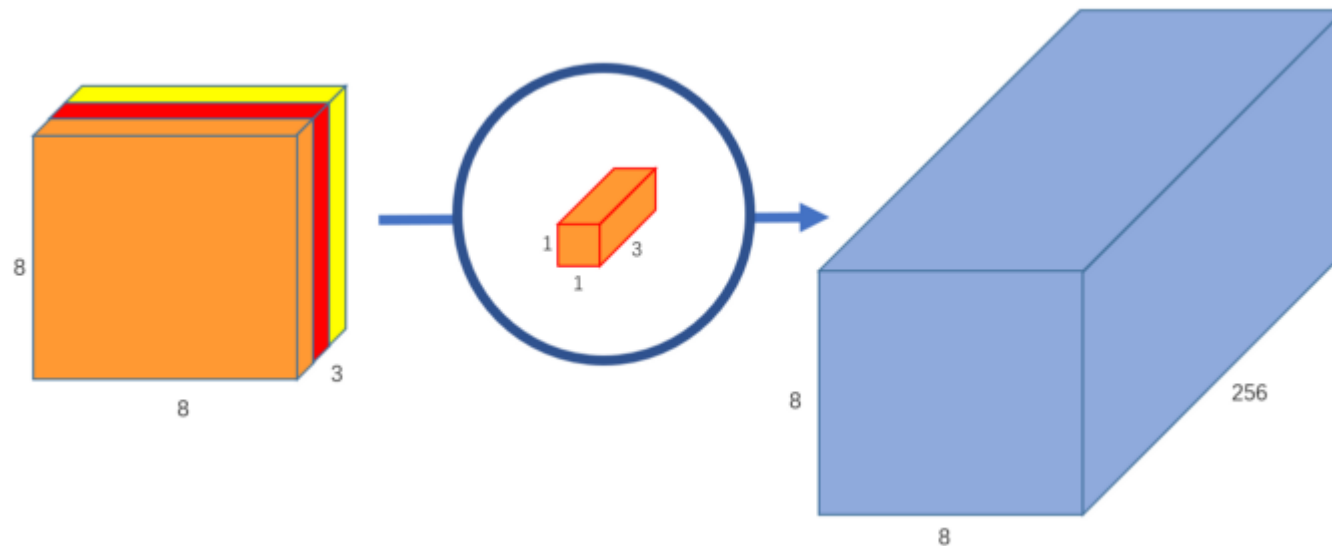


Each 5x5x1 kernel iterates 1 channel of the image (note: **1 channel**, not all channels), getting the scalar products of every 25 pixel group, giving out a 8x8x1 image. Stacking these images together creates a 8x8x3 image.

Depthwise separable convolution – step2



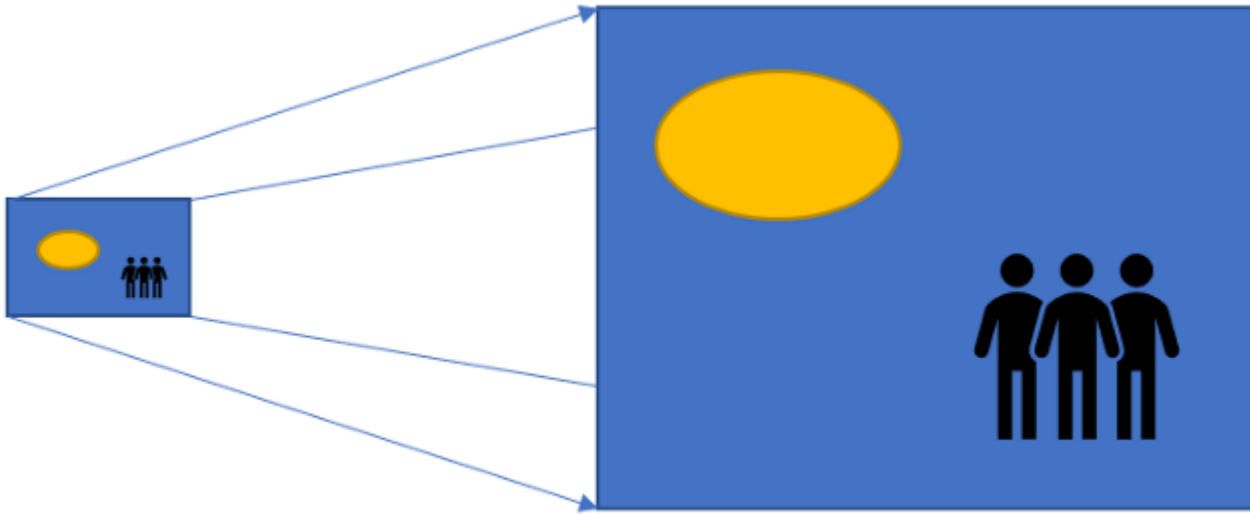
Pointwise



Let's calculate the number of multiplications the computer has to do in the original convolution. There are 256 $5 \times 5 \times 3$ kernels that move 8×8 times. That's $256 \times 3 \times 5 \times 5 \times 8 \times 8 = 1,228,800$ multiplications.

What about the separable convolution? In the depthwise convolution, we have 3 $5 \times 5 \times 1$ kernels that move 8×8 times. That's $3 \times 5 \times 5 \times 8 \times 8 = 4,800$ multiplications. In the pointwise convolution, we have 256 $1 \times 1 \times 3$ kernels that move 8×8 times. That's $256 \times 1 \times 1 \times 3 \times 8 \times 8 = 49,152$ multiplications. Adding them up together, that's 53,952 multiplications.

Transpose convolution



- Nearest neighbor interpolation
- Bi-linear interpolation
- Bi-cubic interpolation

$$\begin{pmatrix} x1 & x2 & x3 \\ x4 & x5 & x6 \\ x7 & x8 & x9 \end{pmatrix} * \begin{pmatrix} k1 & k2 \\ k3 & k4 \end{pmatrix}$$

Here is a constructed matrix with a vector:

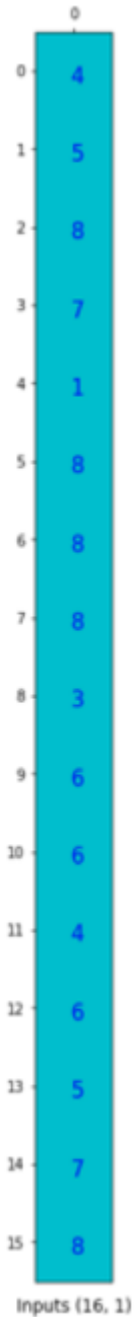
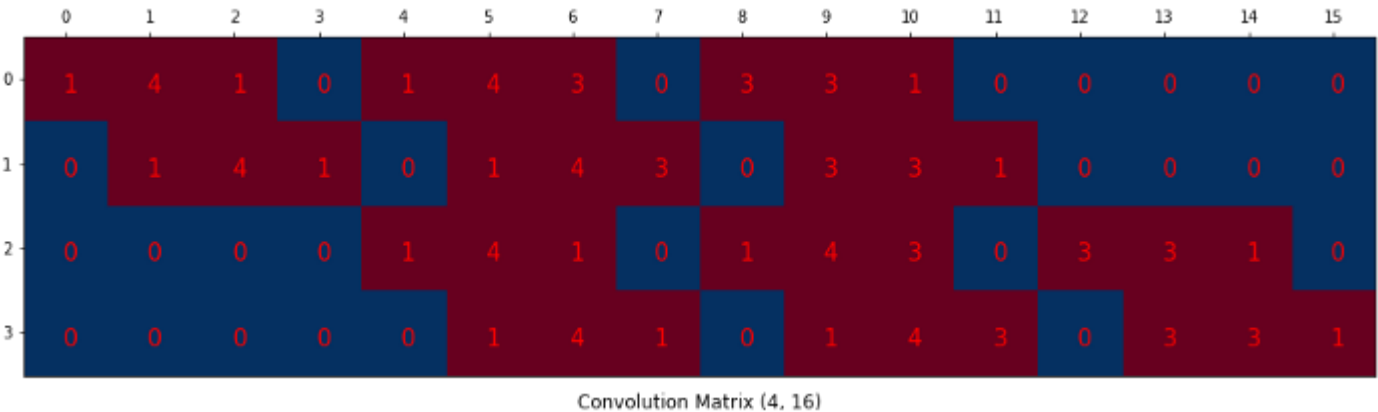
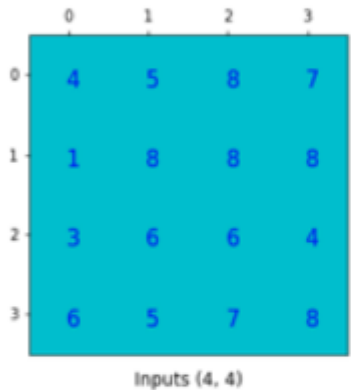
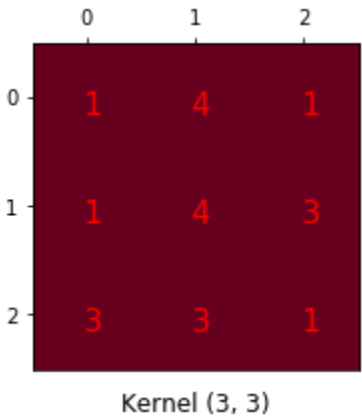
$$\begin{pmatrix} k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 & 0 \\ 0 & k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 \\ 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 & 0 \\ 0 & 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 \end{pmatrix} \cdot \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \\ x9 \end{pmatrix}$$

which is equal to

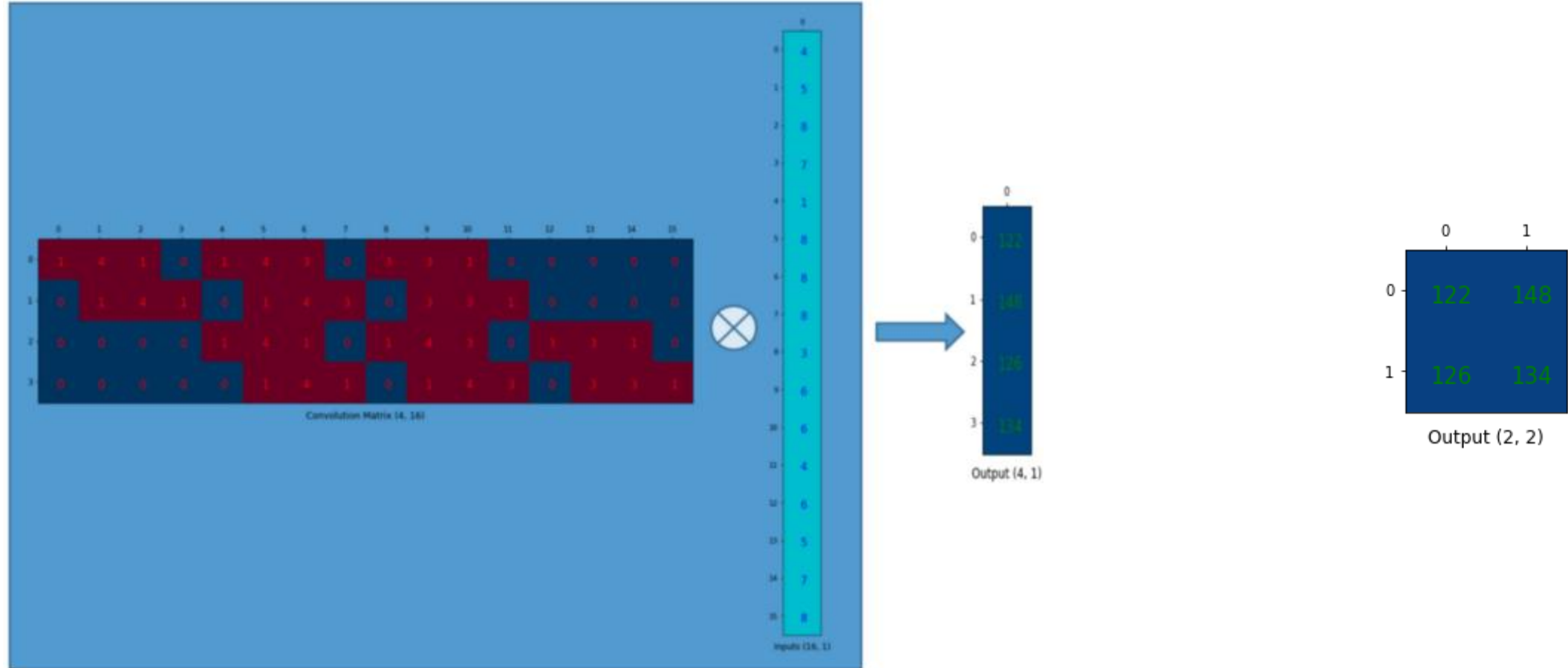
$$\begin{pmatrix} k1 x1 + k2 x2 + k3 x4 + k4 x5 \\ k1 x2 + k2 x3 + k3 x5 + k4 x6 \\ k1 x4 + k2 x5 + k3 x7 + k4 x8 \\ k1 x5 + k2 x6 + k3 x8 + k4 x9 \end{pmatrix}$$

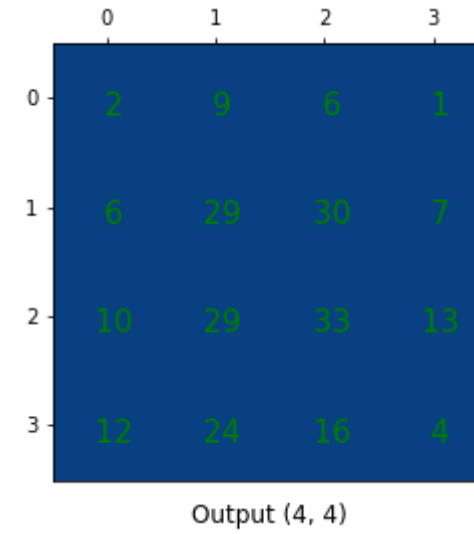
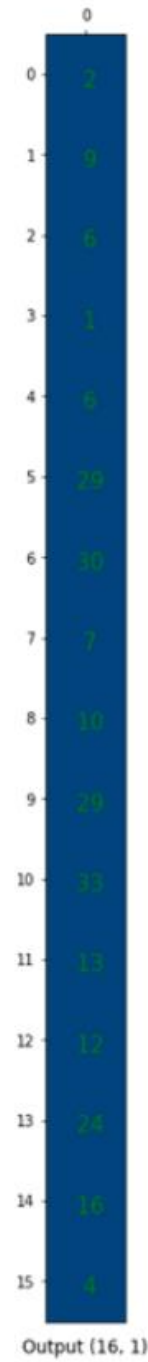
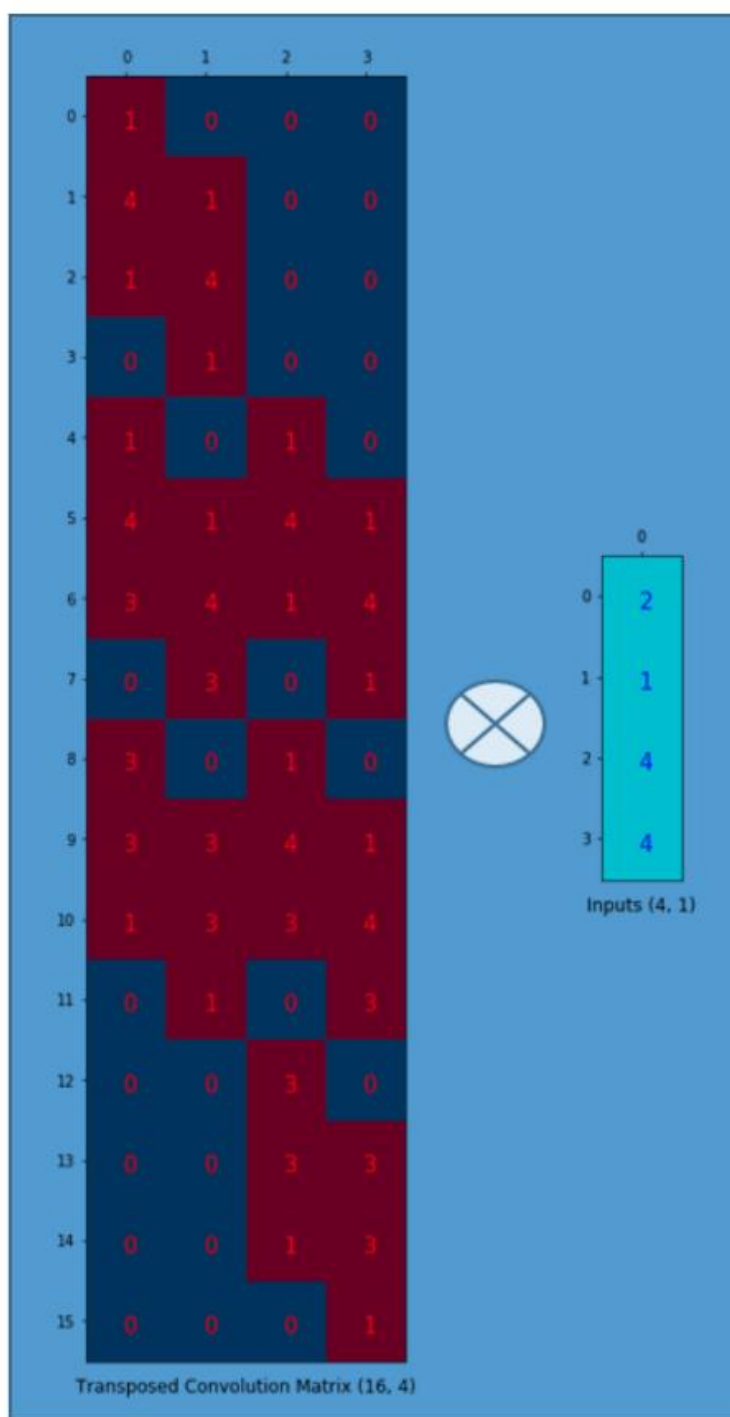
Suppose we have a 4x4 matrix and apply a convolution operation on it with a 3x3 kernel, with no padding, and with a stride of 1. As shown further below, the output is a 2x2 matrix.

Convolution as a matrix multiplication



Many to one mapping – 9 values to 1 value

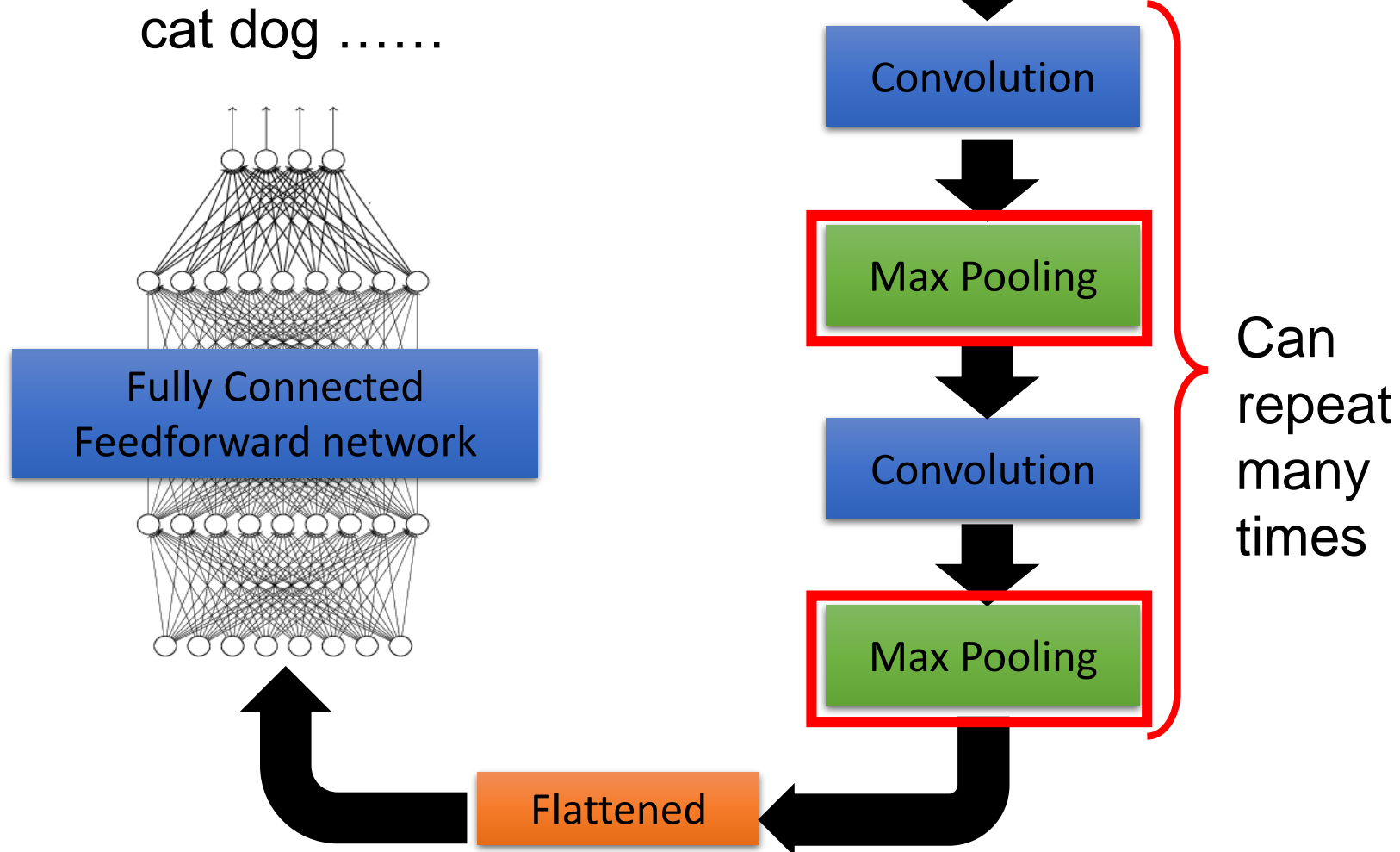




- Fractionally-strided convolution
- Deconvolution

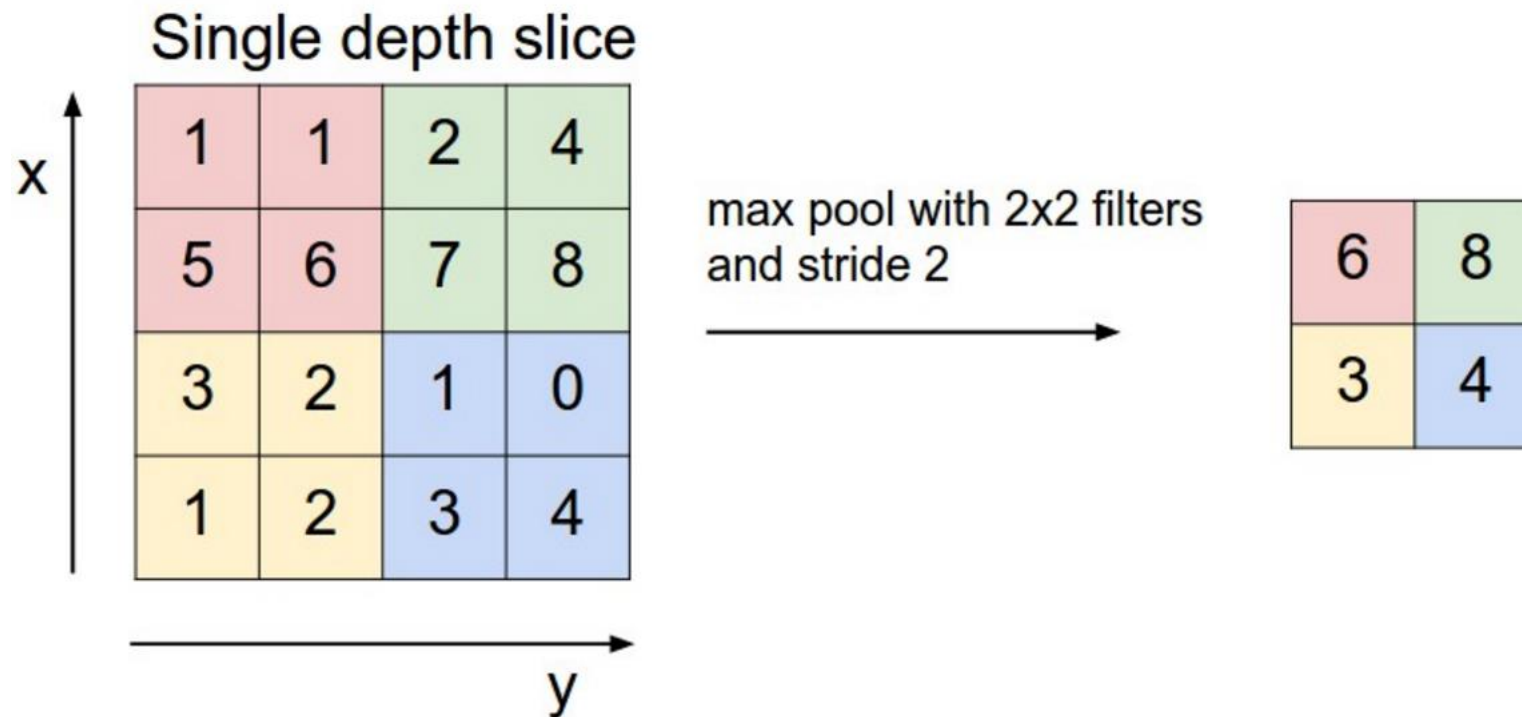
One to many mapping

The whole CNN



Pooling

- Down-sample the image – controls the parameters of the CNN model



Why Pooling

- Subsampling pixels will not change the object

bird



Subsampling

bird



- ✓ We can subsample the pixels to make image smaller
- ✓ fewer parameters to characterize the image

Pooling or strided convolution?

STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET

Jost Tobias Springenberg*, Alexey Dosovitskiy*, Thomas Brox, Martin Riedmiller

Department of Computer Science

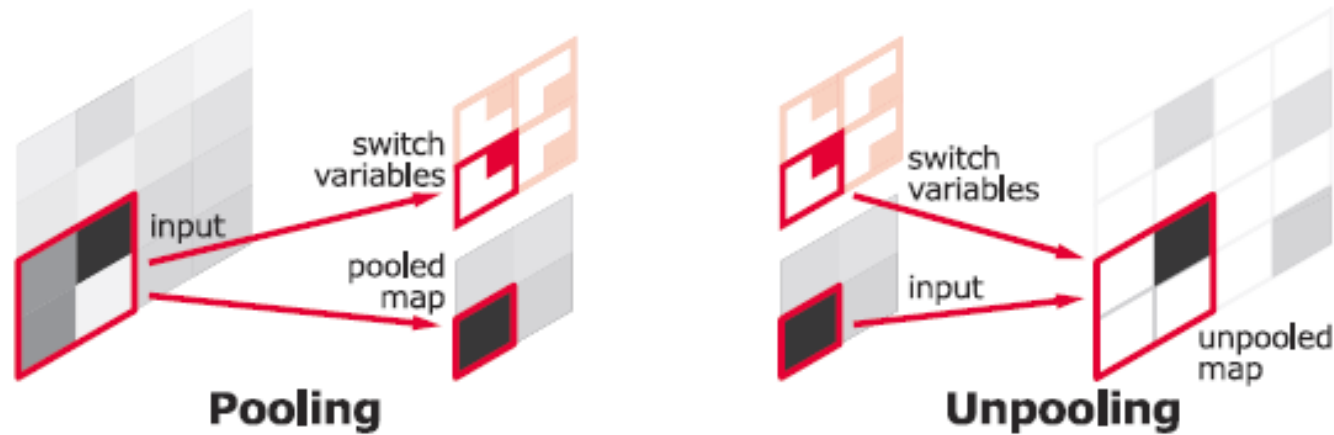
University of Freiburg

Freiburg, 79110, Germany

`{springj, dosovits, brox, riedmiller}@cs.uni-freiburg.de`

"when pooling is replaced by an additional convolution layer
with stride $r = 2$ performance stabilizes and even improves on the base model"

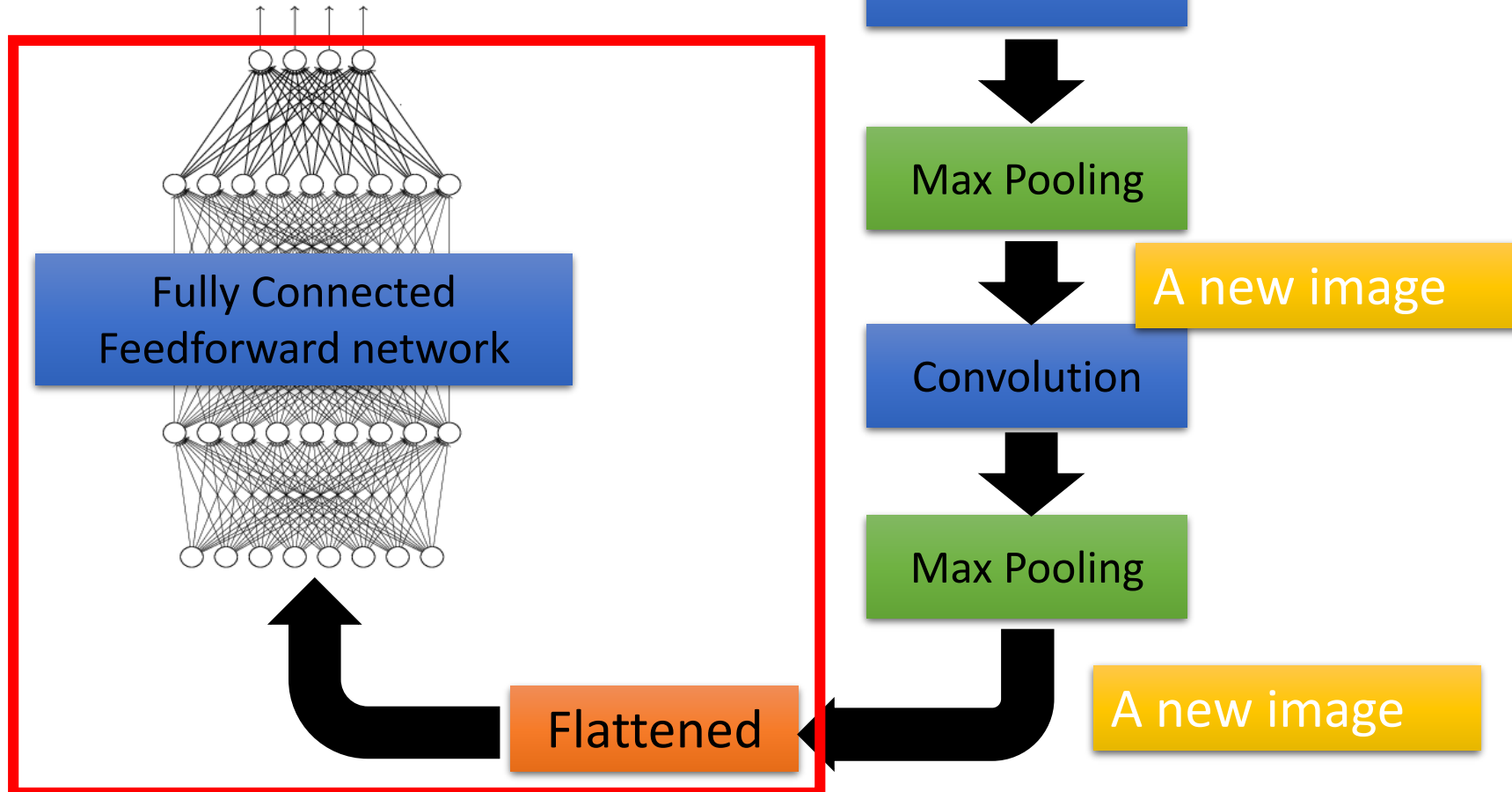
Unpool


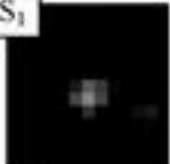


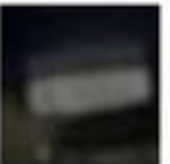


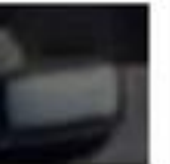
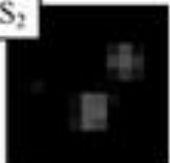






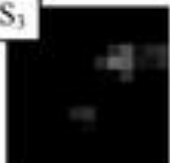


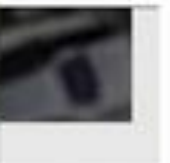

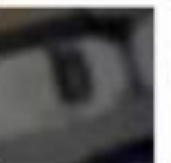
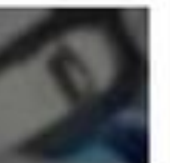





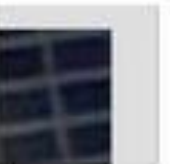

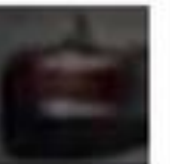

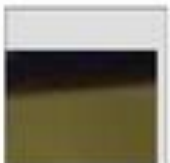









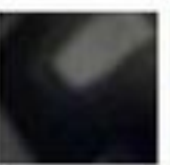




Unpooling: In the convnet, the max pooling operation is non-invertible, however we can obtain an approximate inverse by recording the locations of the maxima within each pooling region in a set of switch variables. In the deconvnet, the unpooling operation uses these switches to place the reconstructions from the layer above into appropriate locations, preserving the structure of the stimulus. See Fig. 1(bottom) for an illustration of the procedure.

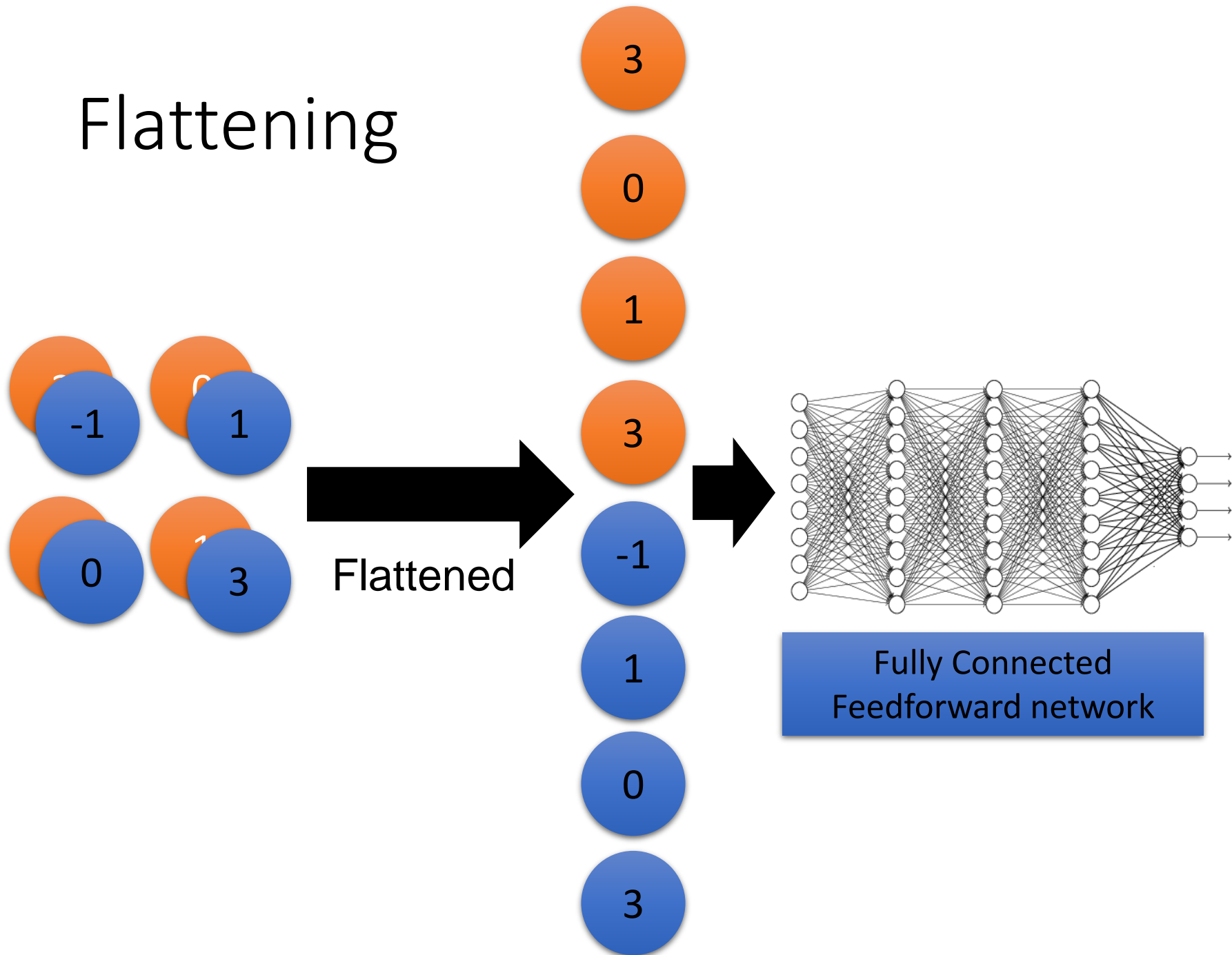
The whole CNN

cat dog



Raw Image	Feature Map	Top Activation Image Crops					
x	$V^{(j)}(x)$	1st	2nd	3rd	4th	5th	6th
 Strongly correlated convolutional kernels	S_1 						
	S_2 						
	S_3 						
 Weakly correlated convolutional kernels	W_1 						
	W_2 						
	W_3 						

Flattening



Conv Net Topology

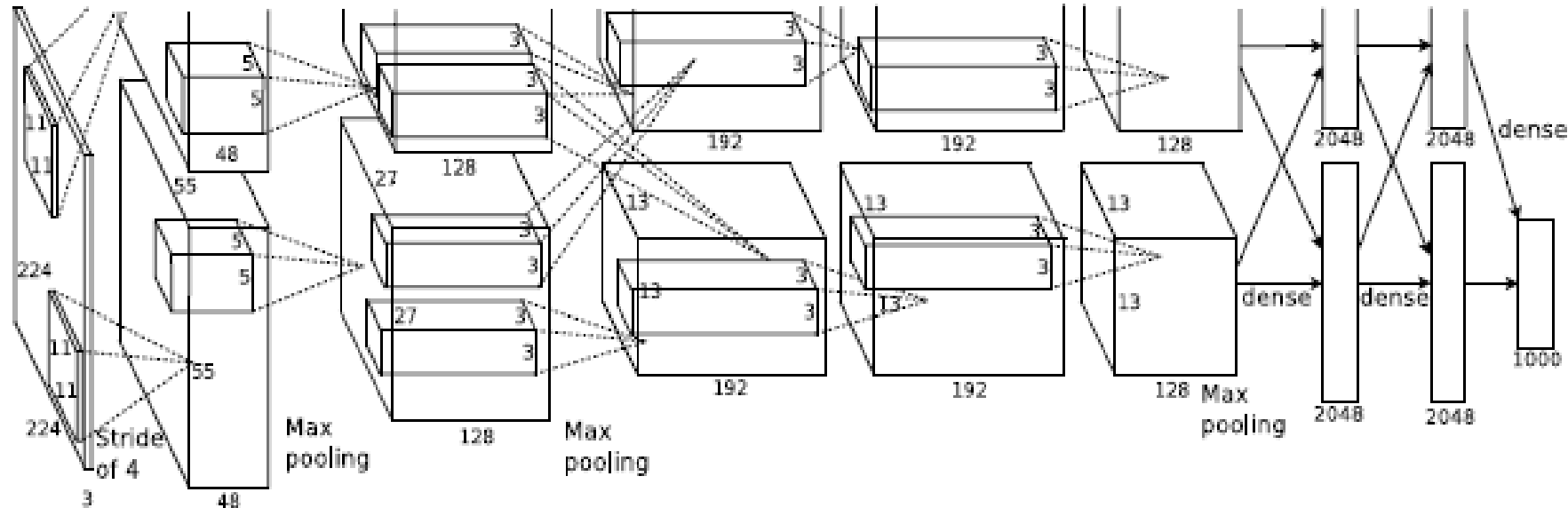
- 5 convolutional layers
- 3 fully connected layers + soft-max
- 650K neurons , 60 Mln weights

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca





track cycling
cycling
track cycling
road bicycle racing
marathon
ultramarathon



ultramarathon
ultramarathon
half marathon
running
marathon
inline speed skating



heptathlon
heptathlon
decathlon
hurdles
pentathlon
sprint (running)



bikejoring
mushing
bikejoring
harness racing
skijoring
carting



longboarding
longboarding
aggressive inline skating
freestyle scootering
freeboard (skateboard)
sandboarding



ultimate (sport)
ultimate (sport)
hurling
flag football
association football
rugby sevens



demolition derby
demolition derby
monster truck
mud bogging
motocross
grand prix motorcycle racing



telemark skiing
snowboarding
telemark skiing
nordic skiing
ski touring
skijoring



whitewater kayaking
whitewater kayaking
rafting
kayaking
canoeing
adventure racing



arena football
indoor american football
arena football
canadian football
american football
women's lacrosse



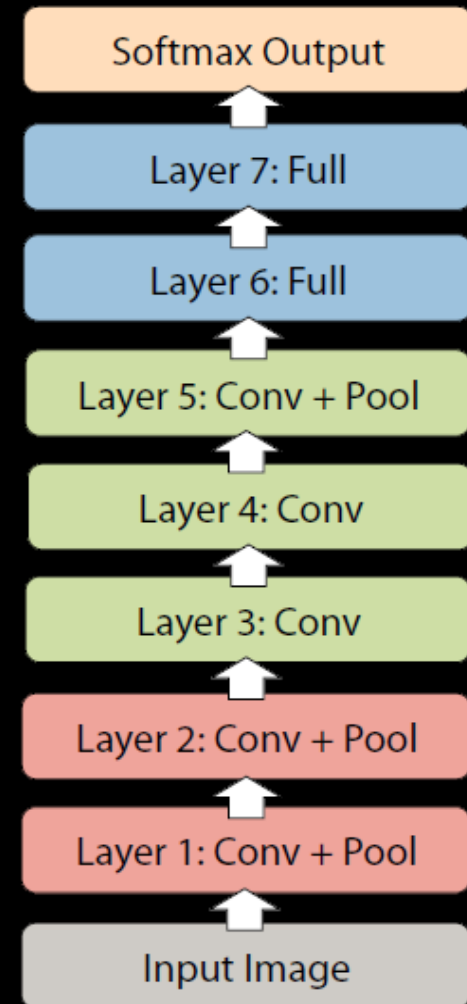
reining
barrel racing
rodeo
reining
cowboy action shooting
bull riding



eight-ball
nine-ball
blackball (pool)
trick shot
eight-ball
straight pool

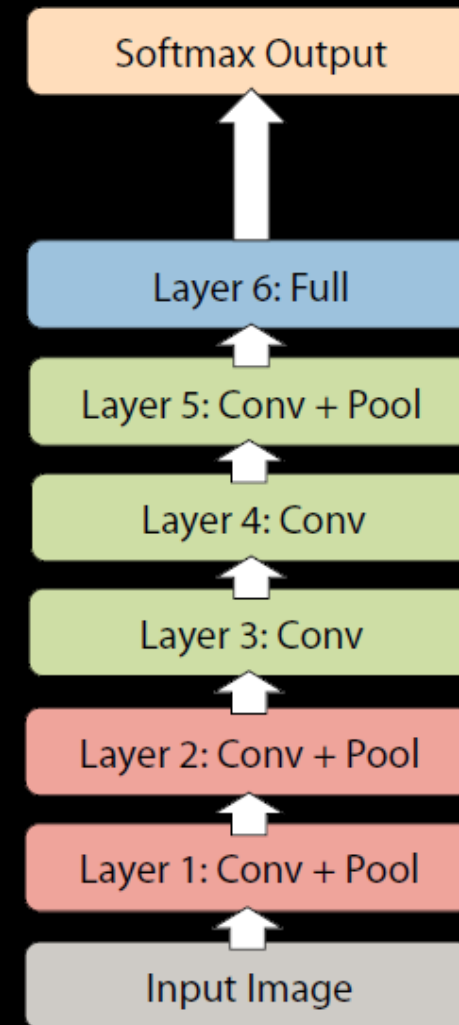
Why do we need a deep CNN?

- 8 layers total
- Trained on Imagenet dataset [Deng et al. CVPR'09]
- 18.2% top-5 error
- Our reimplementation:
18.1% top-5 error



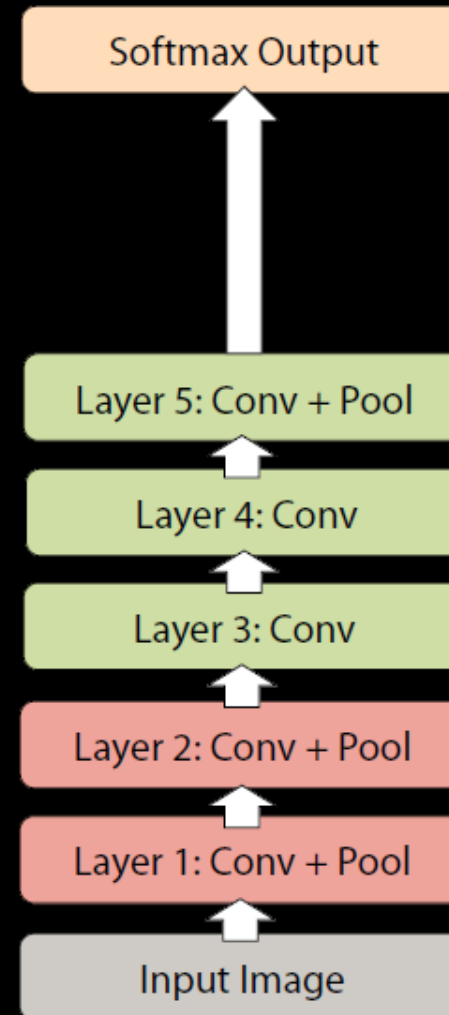
Why do we need a *deep* CNN?

- Remove top fully connected layer
– Layer 7
- Drop 16 million parameters
- Only 1.1% drop in performance!



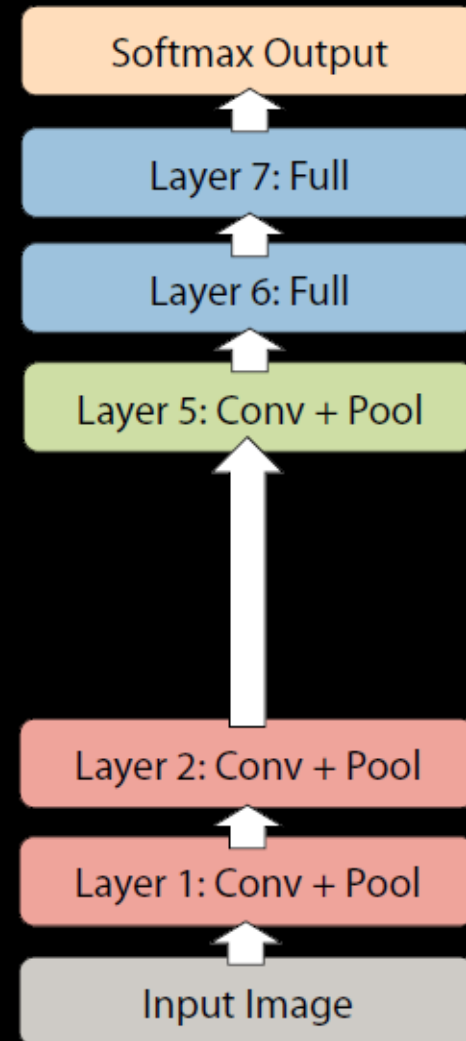
Why do we need a *deep* CNN?

- Remove both fully connected layers
 - Layer 6 & 7
- Drop ~50 million parameters
- 5.7% drop in performance



Why do we need a *deep* CNN?

- Now try removing upper feature extractor layers:
 - Layers 3 & 4
- Drop ~1 million parameters
- 3.0% drop in performance



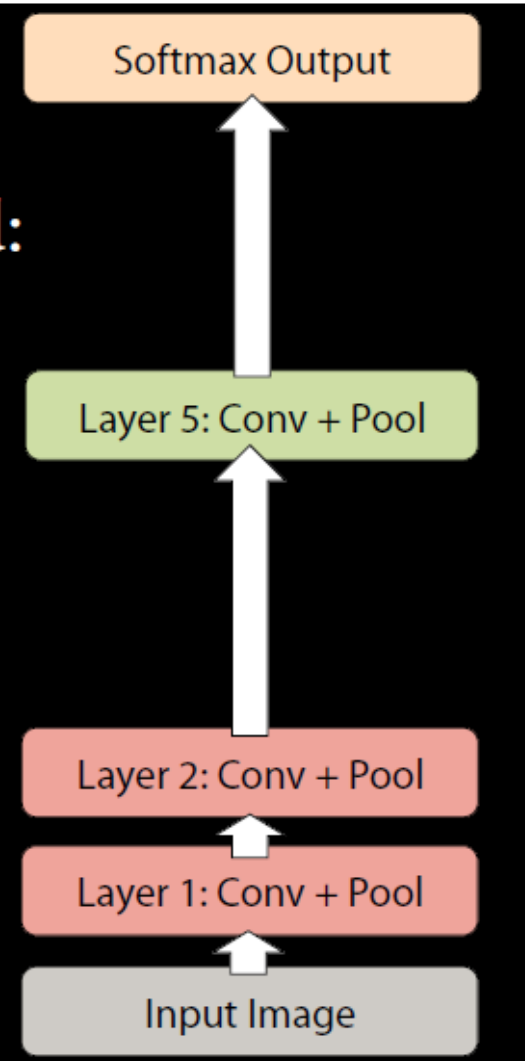
Why do we need a *deep* CNN?

- Now try removing upper feature extractor layers & fully connected:
 - Layers 3, 4, 6, 7

- Now only 4 layers

- 33.5% drop in performance

→ Depth of network is key



Suggested reading

A guide to convolution arithmetic for deep learning

Vincent Dumoulin^{1★} and Francesco Visin^{2★†}

★MILA, Université de Montréal

†AIRLab, Politecnico di Milano

January 12, 2018