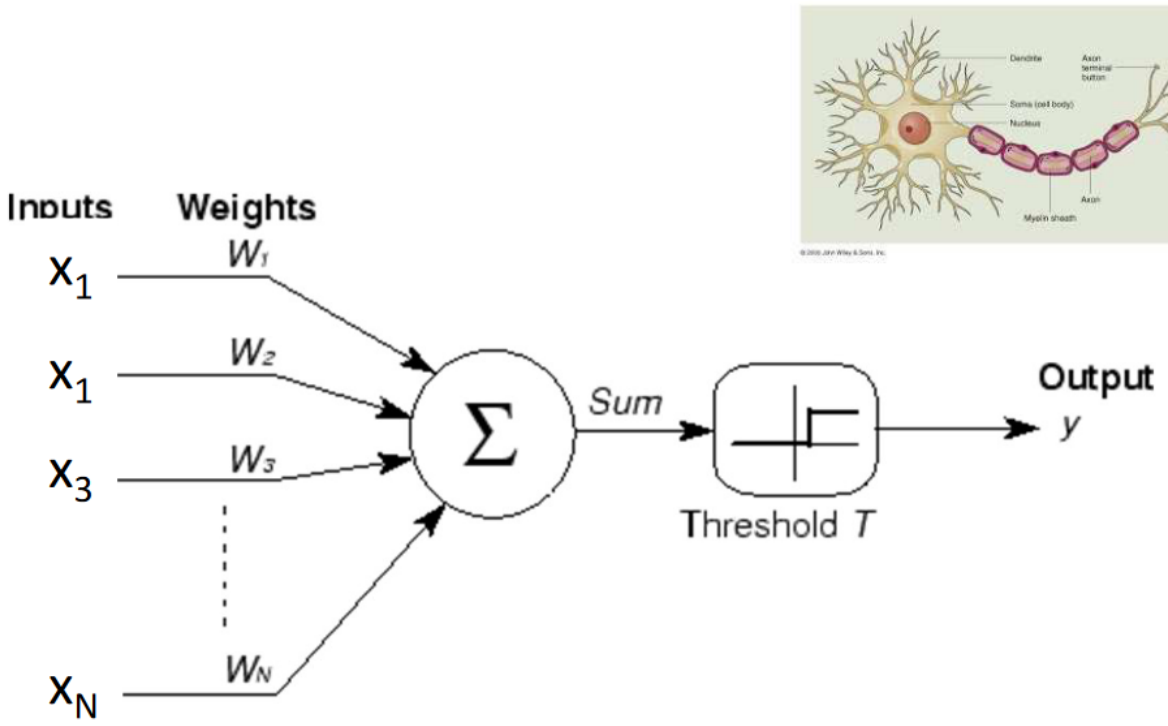
Neural networks and CNN

Biplab Banerjee

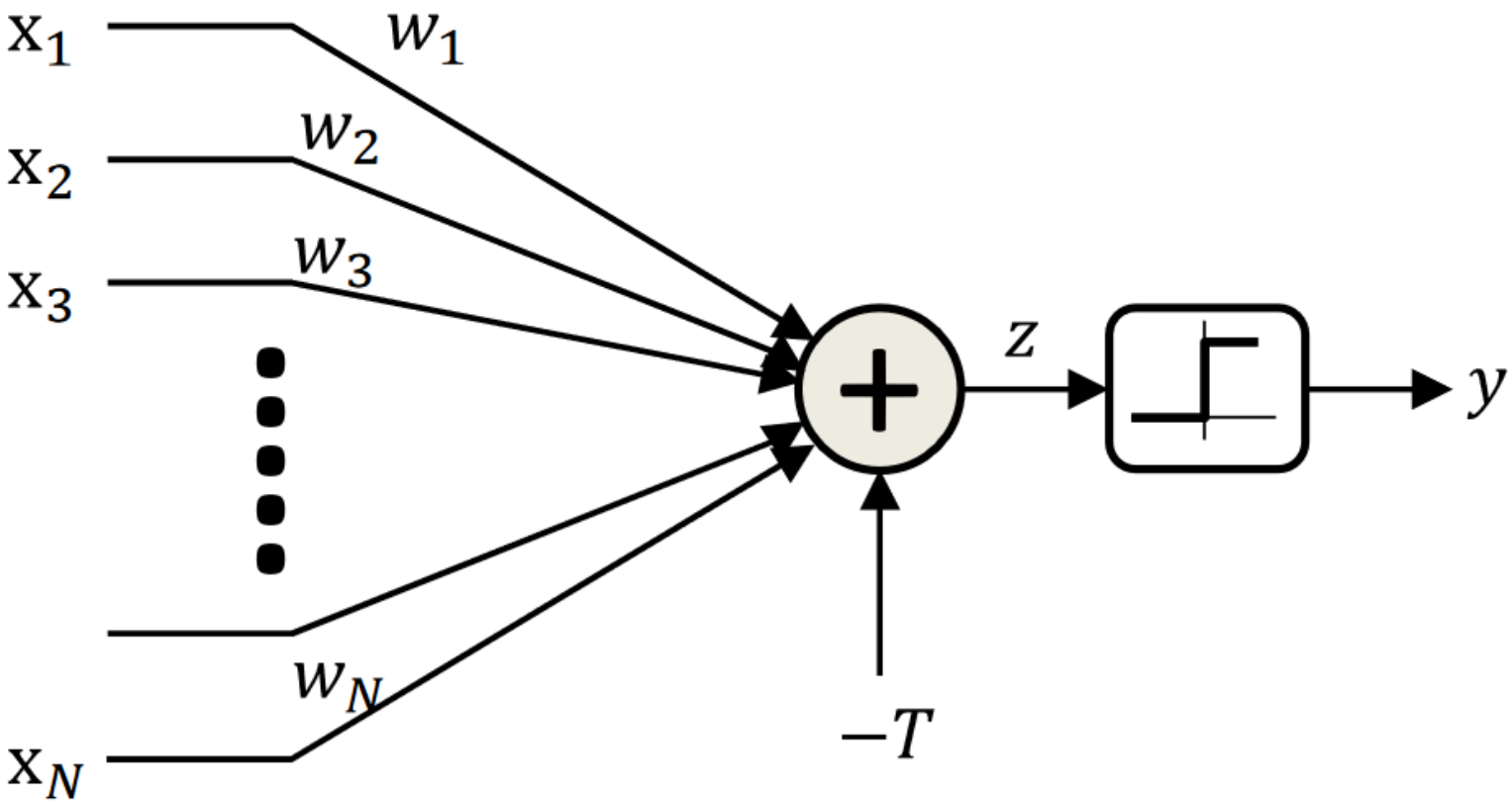
Thanks to towardsdatascience, Vincente Ordonez

Perceptron Model

Frank Rosenblatt (1957) - Cornell University

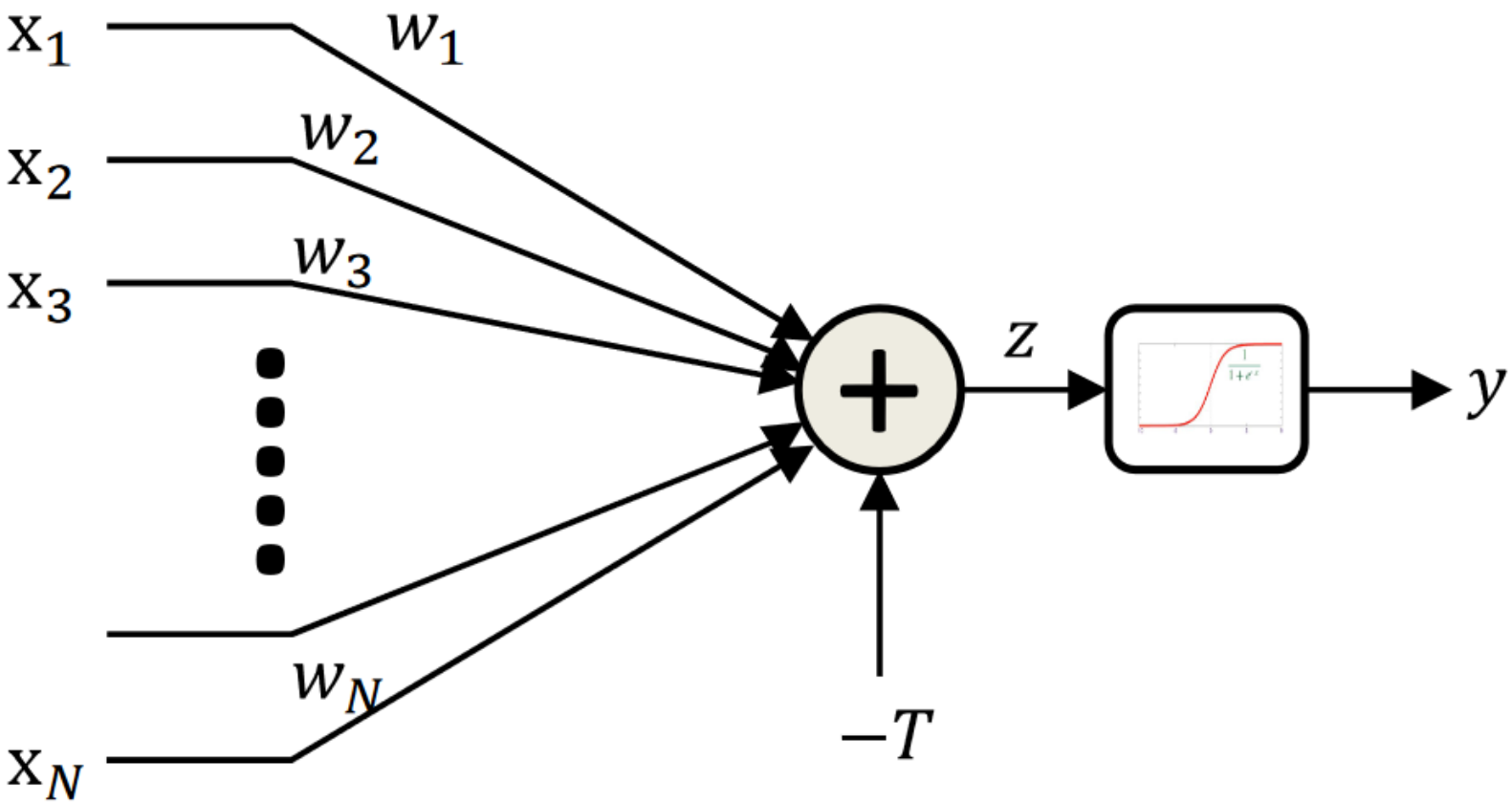


$$y = \begin{cases} 1 \; if \sum_i w_i x_i \geq T \\ 0 \; else \end{cases}$$



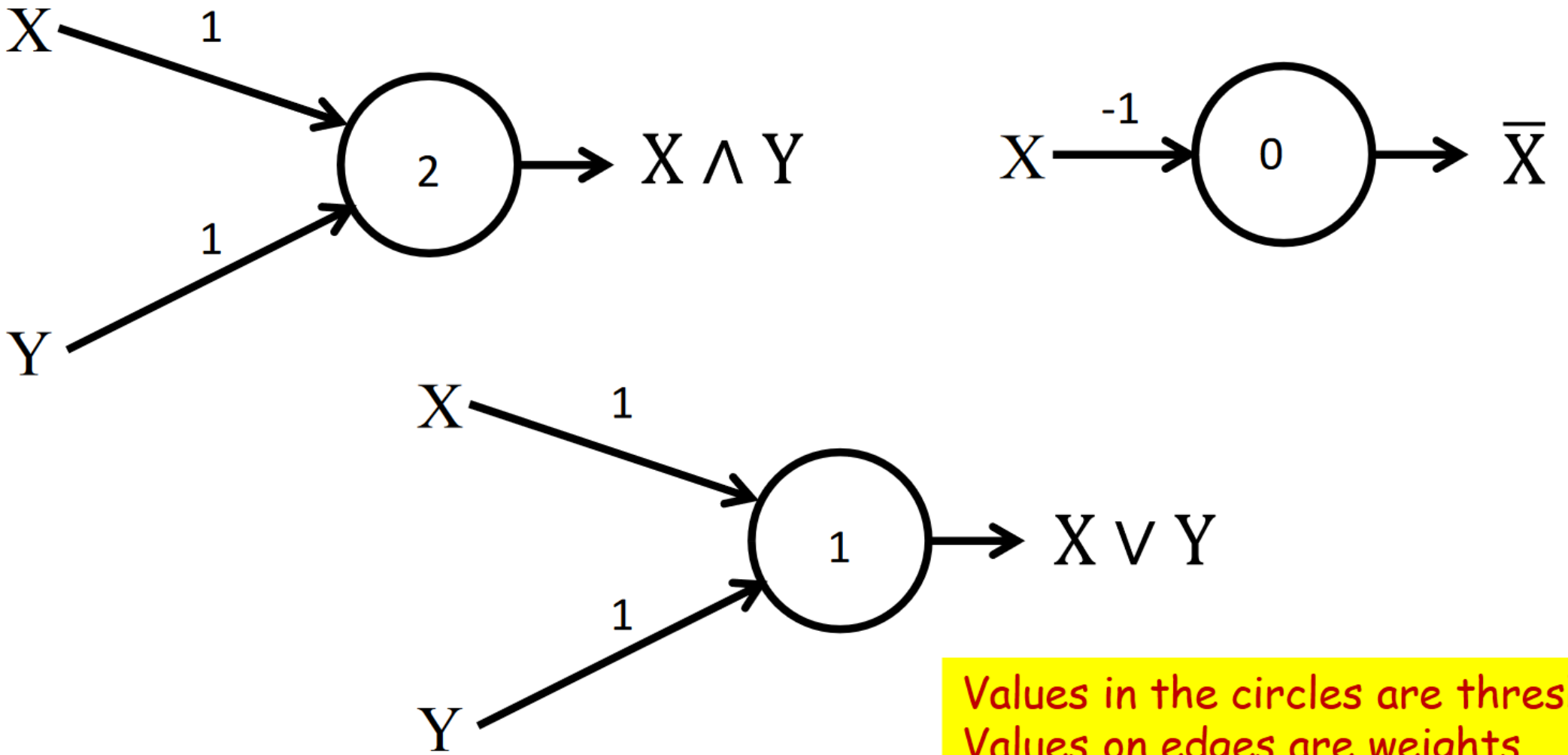
$$z = \sum_i w_i x_i - T$$

$$y = \begin{cases} 1 \; if \; z \geq 0 \\ 0 \; else \end{cases}$$



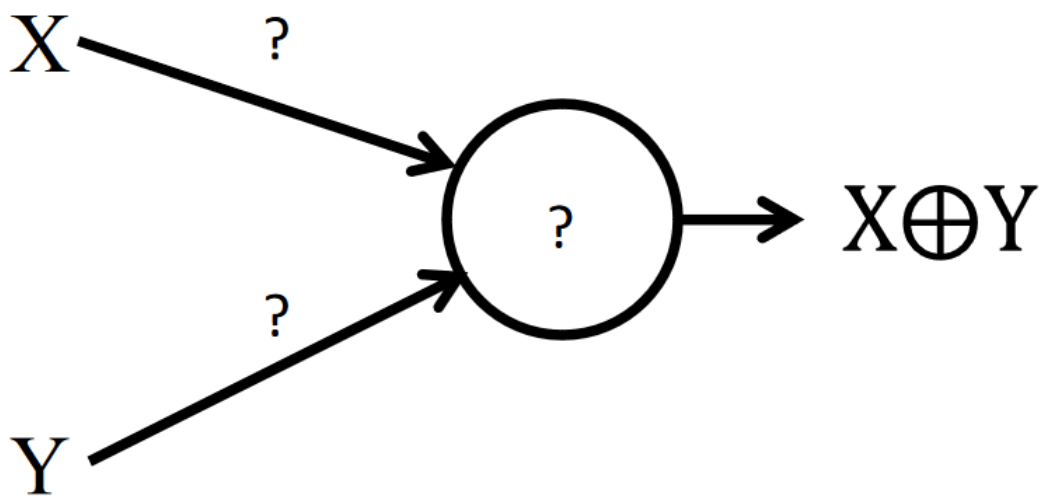
$$z = \sum_i w_i x_i - T$$

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

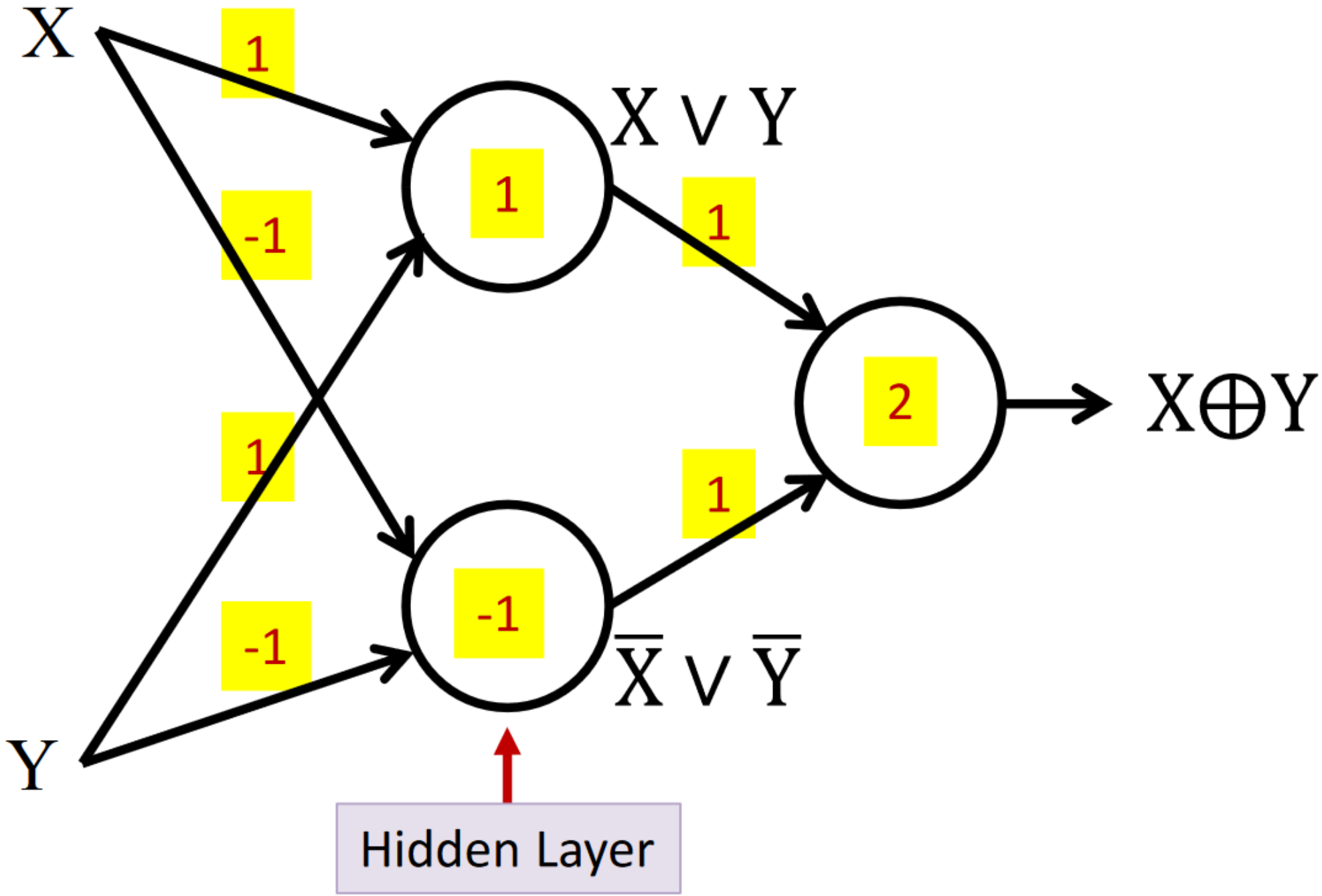


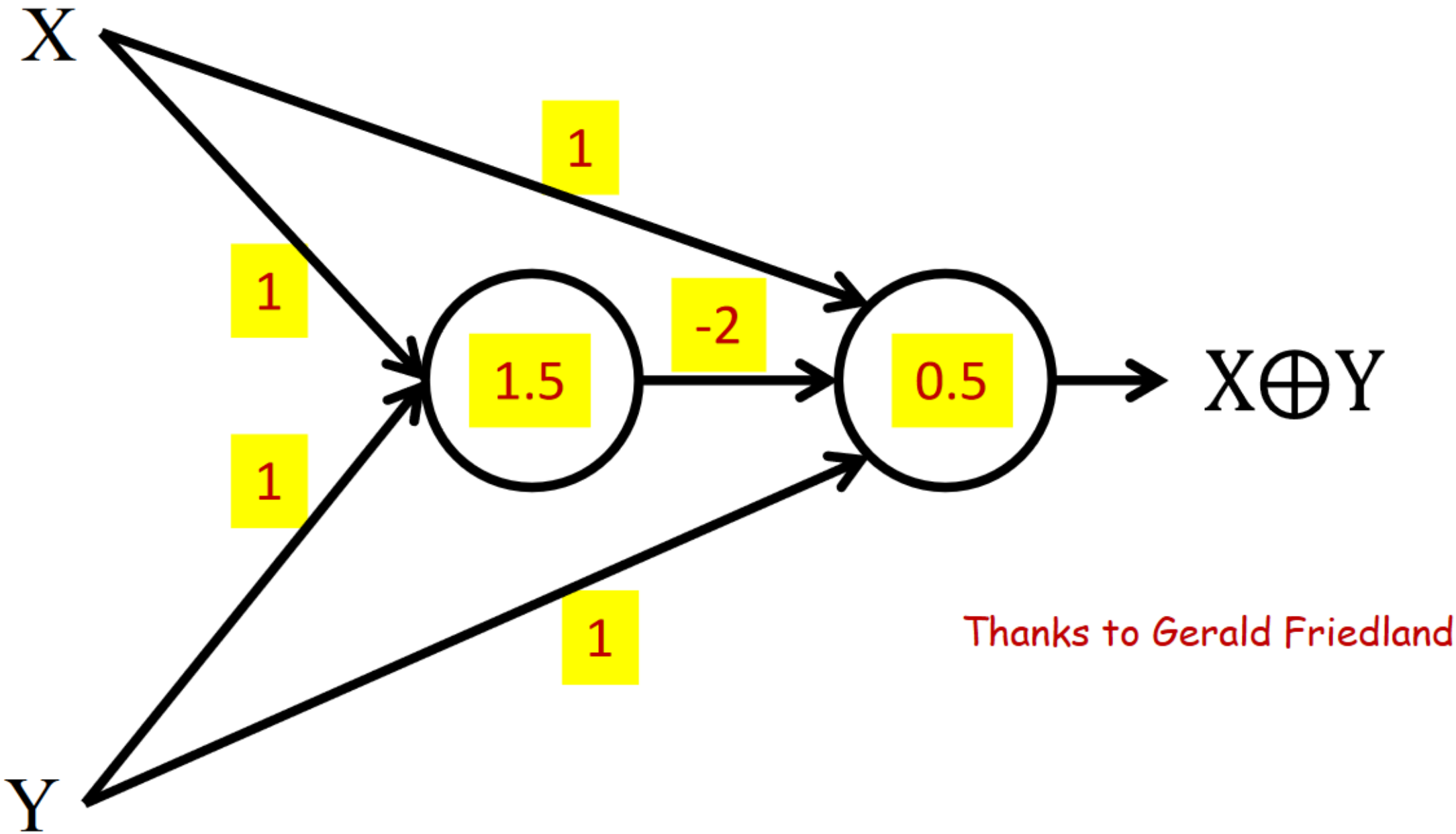
- A perceptron can model any simple binary Boolean gate

The perceptron is not enough



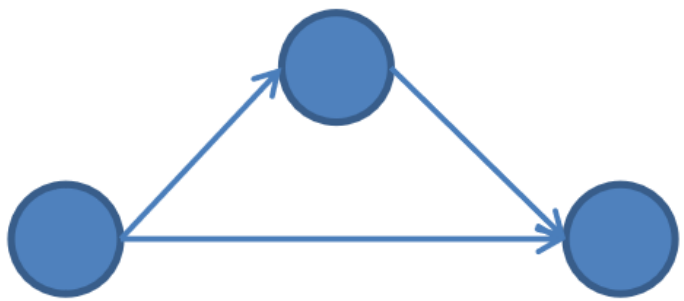
Cannot compute an XOR

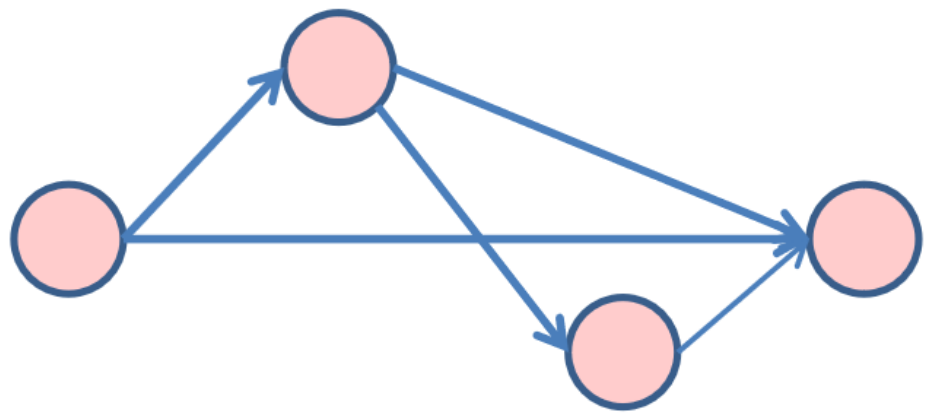

X
Y
1
-1
1
-1
1
1
-1
1
2
X ∨ Y
$\overline{X} \vee \overline{Y}$
X⊕Y
Hidden Layer



Thanks to Gerald Friedland

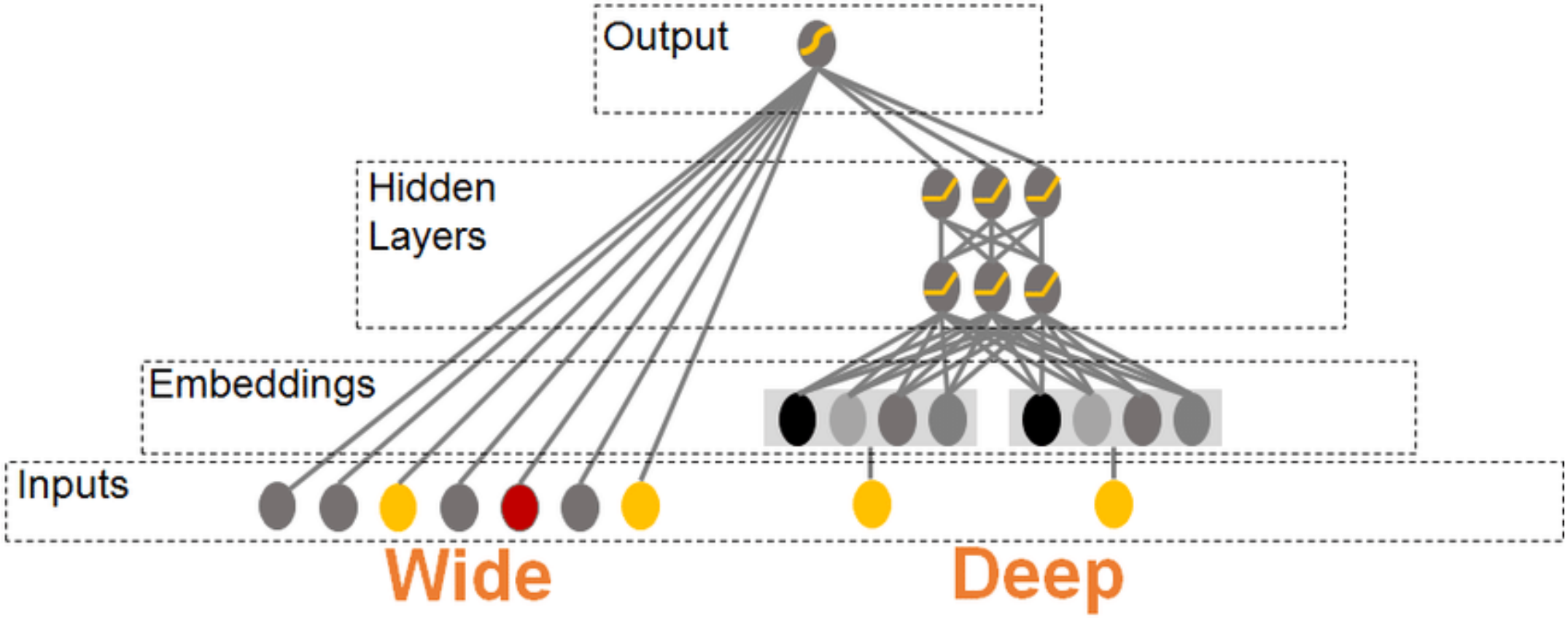
- In any directed network of computational elements with input source nodes and output sink nodes, “depth” is the length of the longest path from a source to a sink
  - A “source” node in a directed graph is a node that has only outgoing edges
  - A “sink” node is a node that has only incoming edges





- Left: Depth = 2. Right: Depth = 3

Wide vs deep networks



Wide neural networks

- With enough width, the network can memorize training examples (especially in small or moderate datasets).
- A sufficiently wide single-hidden-layer neural network with a nonlinear activation can approximate a large class of continuous functions.
- Compared to very deep architectures, wide networks rely more on ample neurons in one or two layers rather than intricate multi-layer designs.

✓A very wide network can memorize data easily, raising the likelihood of overfitting when the training set is not large or diverse.

✓Wide (shallow) networks, unlike deep architectures, do not learn hierarchically abstract features as effectively.

Memorization in wide networks

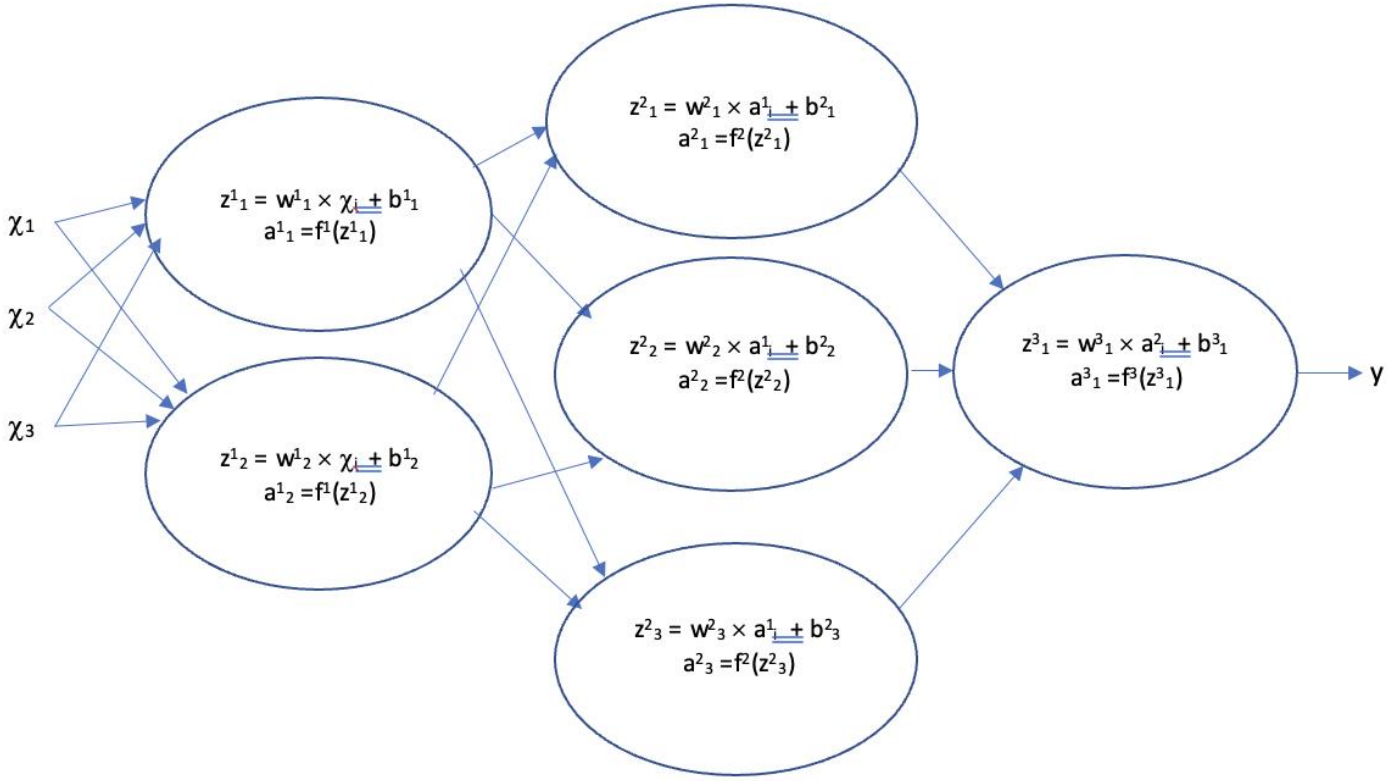
- Wide networks typically have a significantly larger number of trainable parameters. This high capacity allows them to learn detailed patterns—even idiosyncrasies of individual examples—rather than just capturing broader, generalizable trends.

- In high-dimensional parameter spaces, there are infinitely many solutions that can perfectly fit the training data. Wide networks can more readily stumble upon such solutions that memorize the training set rather than learning patterns that generalize well.

Deep neural networks

- In deep networks, lower layers tend to learn simple features (edges, corners, color blobs), while higher layers learn more complex patterns (object parts, semantic concepts). This hierarchical decomposition of features can lead to robust and generalizable representations.
- Deeper layers can gradually transform the input space into more compact, task-relevant feature spaces. This transformation helps remove irrelevant variations in the data (e.g., noise), focusing on features that matter for the target task.

Why is non-linearity important



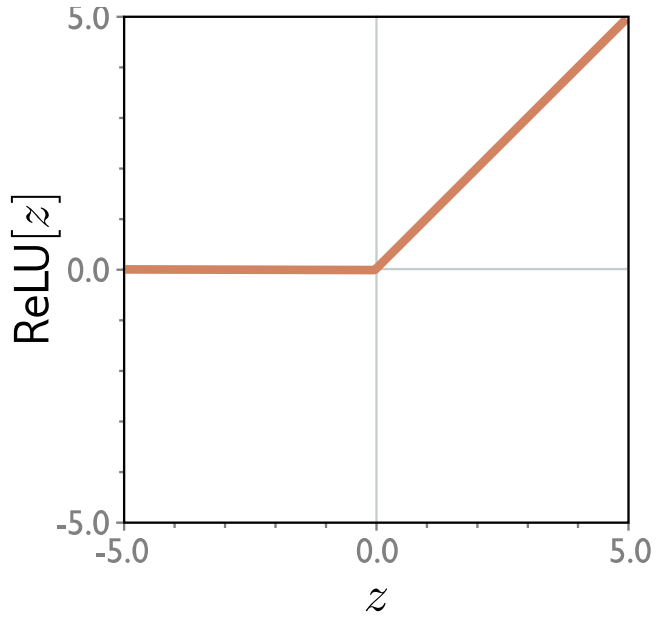
$$
\begin{aligned}
y = f_1^3(z_1^3) &= f_1^3(w_1^3 * a_1^2 + b_1^3) \\
&= f_1^3(w_1^3 * f_1^2(z_1^2) + b_1^3) \\
&= f_1^3(w_1^3 * f_1^2(w_1^2 * a_1^1 + b_1^2) + b_1^3) \\
&= f_1^3(w_1^3 * f_1^2(w_1^2 * f_1(z_1^1) + b_1^2) + b_1^3) \\
&= f_1^3(w_1^3 * f_1^2(w_1^2 * f_1(w_1^1 * x_1 + b_1^1)) + b_1^2) + b_1^3)
\end{aligned}
$$

Example shallow network

Activation function

$$y = \mathrm{f}[x, \boldsymbol{\phi}]$$
$$= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x]$$

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}.$$



Example shallow network

$$y = \mathrm{f}[x, \boldsymbol{\phi}]$$
$$= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]$$

$$\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$$

Hidden units

$$y = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x].$$

Break down into two parts:

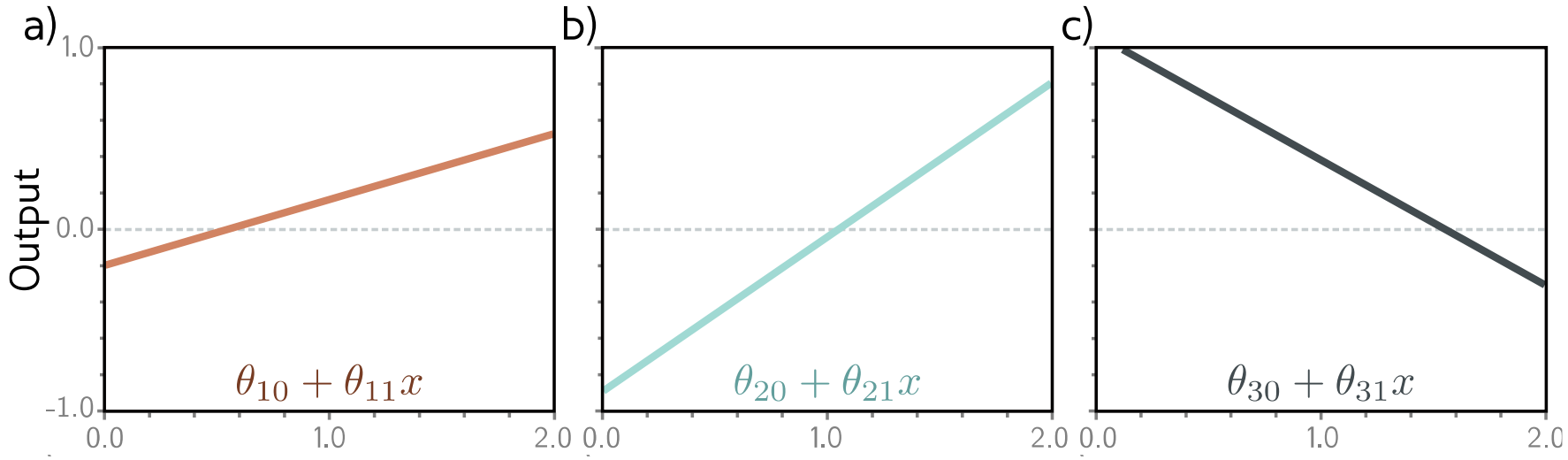
$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

where:

Hidden units

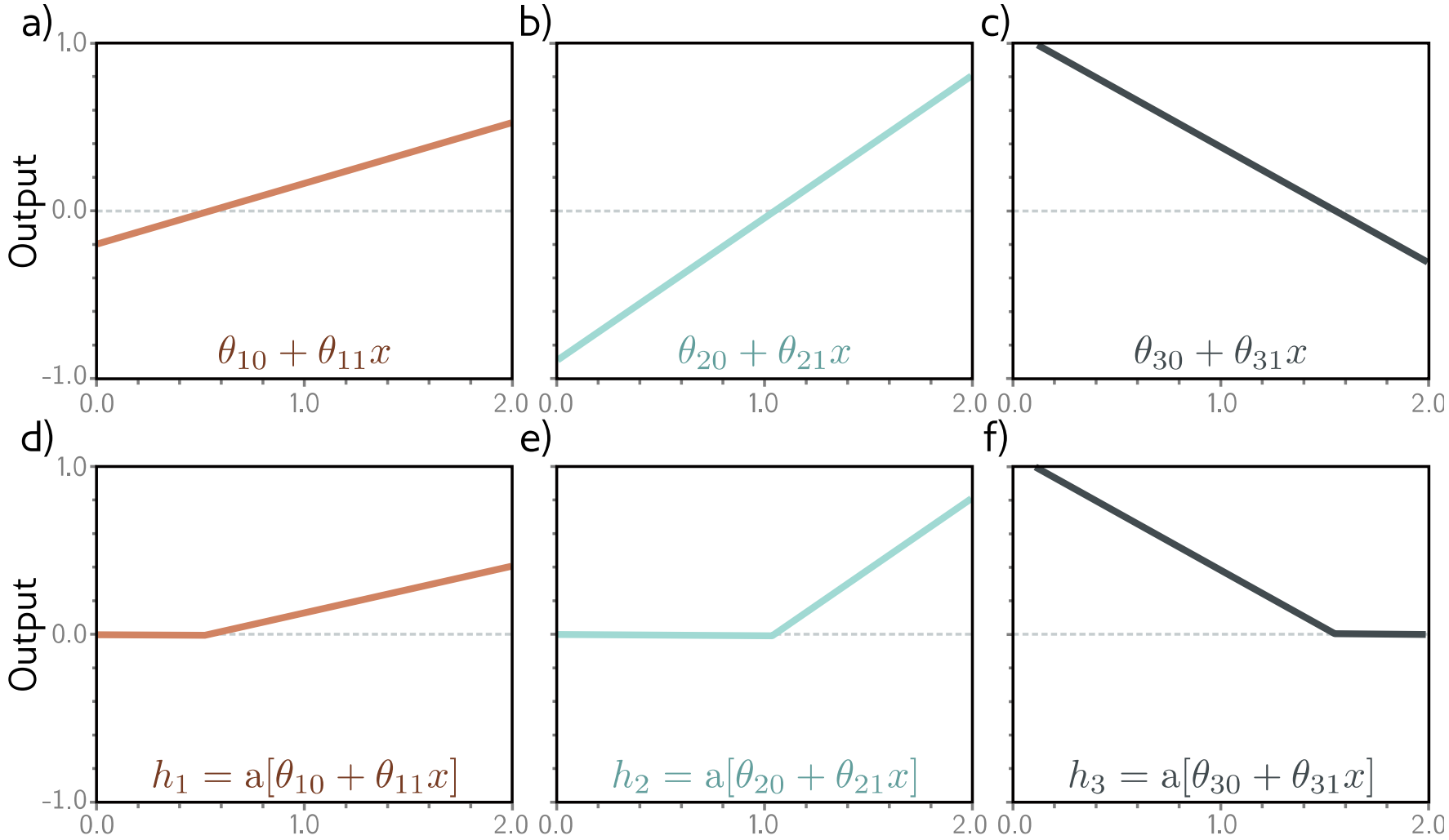
$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \end{aligned}$$

1. compute three
linear functions

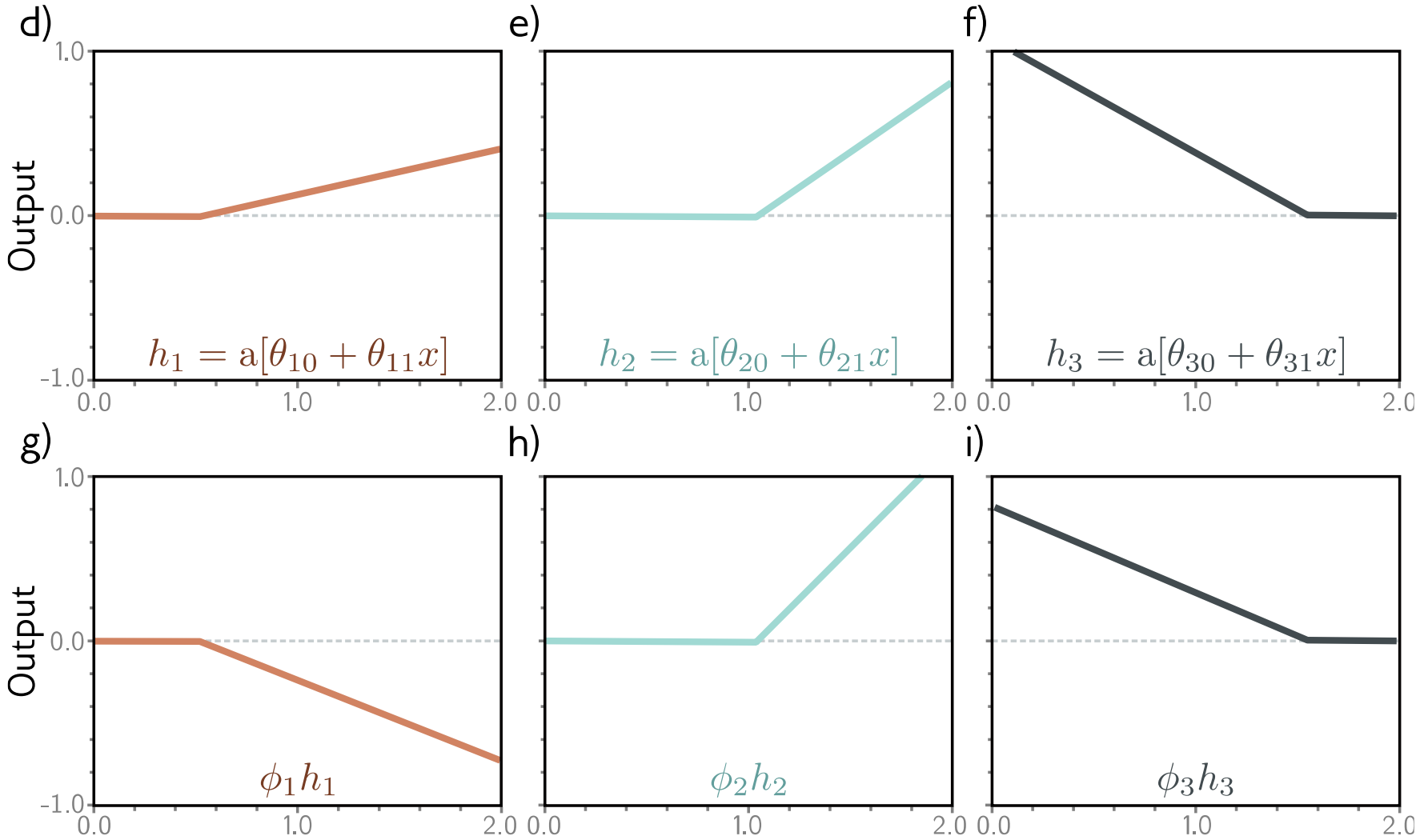


2. Pass through ReLU functions (creates hidden units)

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x],$$

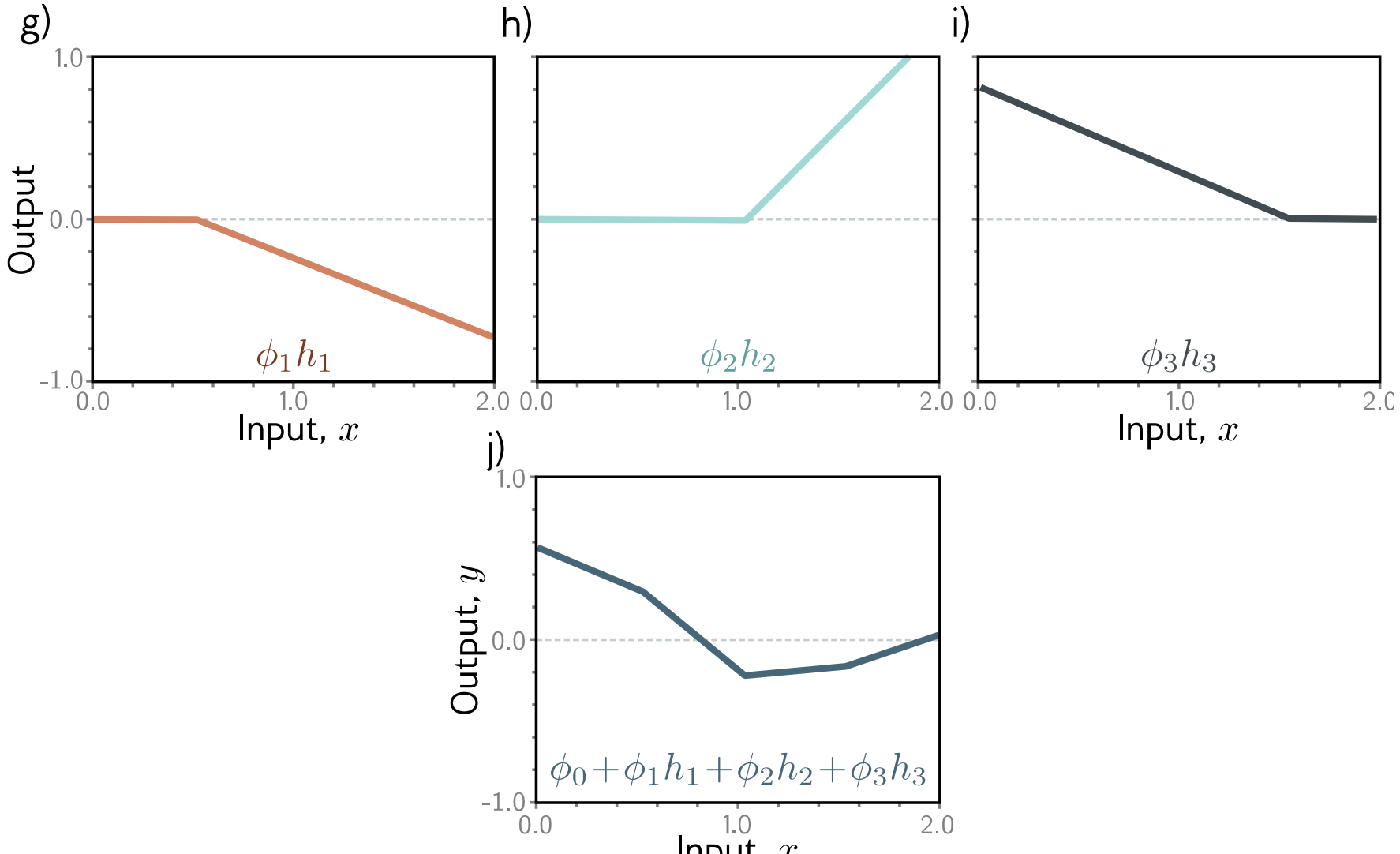


2. Weight the hidden units



4. Sum the weighted hidden units to create output

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$



With 3 hidden units:

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

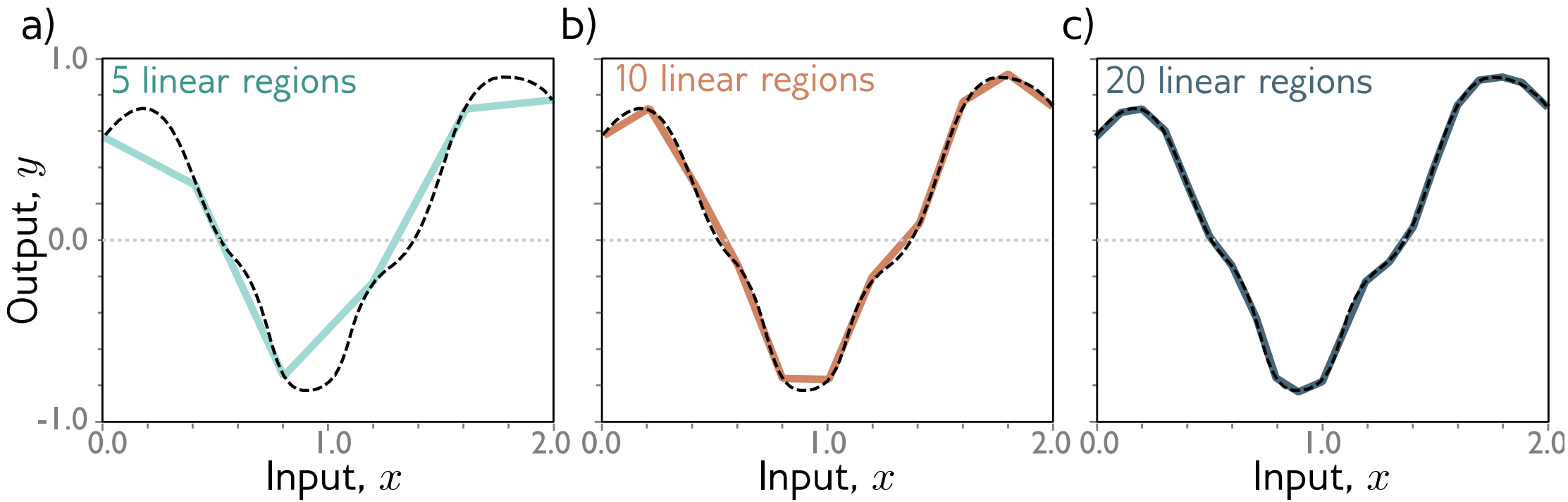
With D hidden units:

$$h_d = \mathrm{a}[\theta_{d0} + \theta_{d1}x]$$

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

With enough hidden units…

… we can describe any 1D function to arbitrary accuracy



Universal approximator theorem

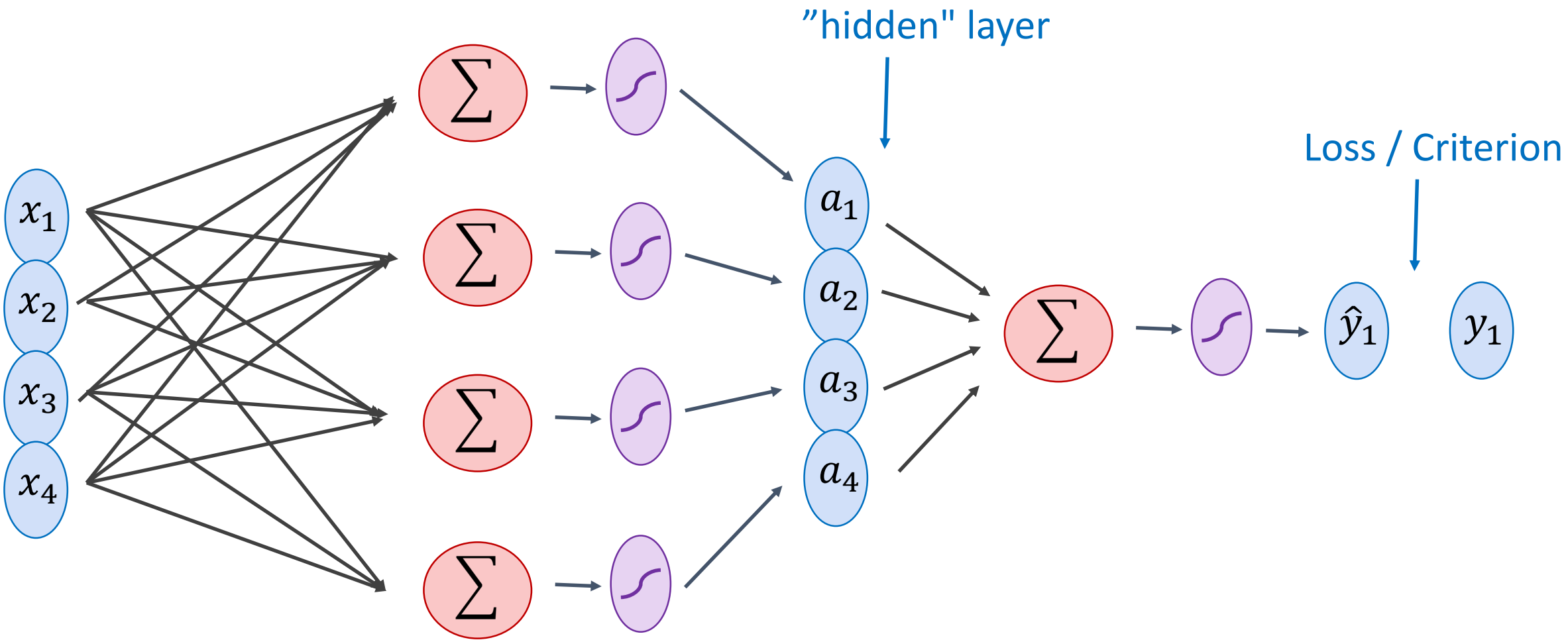
- A feed-forward neural network with at least one hidden layer, a **sufficient number** of hidden units, and a **nonlinear** activation function can approximate **any** continuous function on a compact set to an arbitrary degree of accuracy.

Theorem's Emphasis: Approximation focuses on the ability to fit a function given perfect information (i.e., a known function or infinite data).

Real-World Problem: In practical machine learning, you have finite (often noisy) training data, so the model's ability to generalize beyond that data is **not** addressed by the universal approximation theorem.

A model could "approximate" a function well in theory but still *overfit* or fail to generalize.

Two-layer Multi-layer Perceptron (MLP)



Linear Softmax

$x_i = [x_{i1} \quad x_{i2} \quad x_{i3} \quad x_{i4}]$ $\quad y_i = [1 \quad 0 \quad 0]$ $\quad \hat{y}_i = [f_c \quad f_d \quad f_b]$

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

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

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

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

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

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

Linear Softmax

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

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

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

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

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

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

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

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

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

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

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

Linear Softmax

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

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

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

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

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

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

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

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

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

Linear Softmax

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

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

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

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

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

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

$$f = softmax(g)$$

Linear Softmax

$x_i = [x_{i1} \quad x_{i2} \quad x_{i3} \quad x_{i4}]$ $y_i = [1 \quad 0 \quad 0]$ $\hat{y}_i = [f_c \quad f_d \quad f_b]$

$$f = softmax(wx^T + b^T)$$

Two-layer MLP + Softmax

$x_i = [x_{i1} \quad x_{i2} \quad x_{i3} \quad x_{i4}]$ $y_i = [1 \quad 0 \quad 0]$ $\hat{y}_i = [f_c \quad f_d \quad f_b]$

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

$$f = softmax(w_{[2]}a[1]^T + b_{[2]}^T)$$

N-layer MLP + Softmax

$x_i = [x_{i1} \quad x_{i2} \quad x_{i3} \quad x_{i4}]$ $y_i = [1 \quad 0 \quad 0]$ $\hat{y}_i = [f_c \quad f_d \quad f_b]$

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

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

...

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

...

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

How to train the parameters?

$x_i = [x_{i1} \quad x_{i2} \quad x_{i3} \quad x_{i4}]$ $y_i = [1 \quad 0 \quad 0]$ $\hat{y}_i = [f_c \quad f_d \quad f_b]$

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

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

...

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

...

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

How to train the parameters?

$x_i = [x_{i1} \quad x_{i2} \quad x_{i3} \quad x_{i4}]$ $\quad y_i = [1 \quad 0 \quad 0]$ $\quad \hat{y}_i = [f_c \quad f_d \quad f_b]$

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

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

...

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

...

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

$$l = loss(f, y)$$

We can still use SGD

We need!

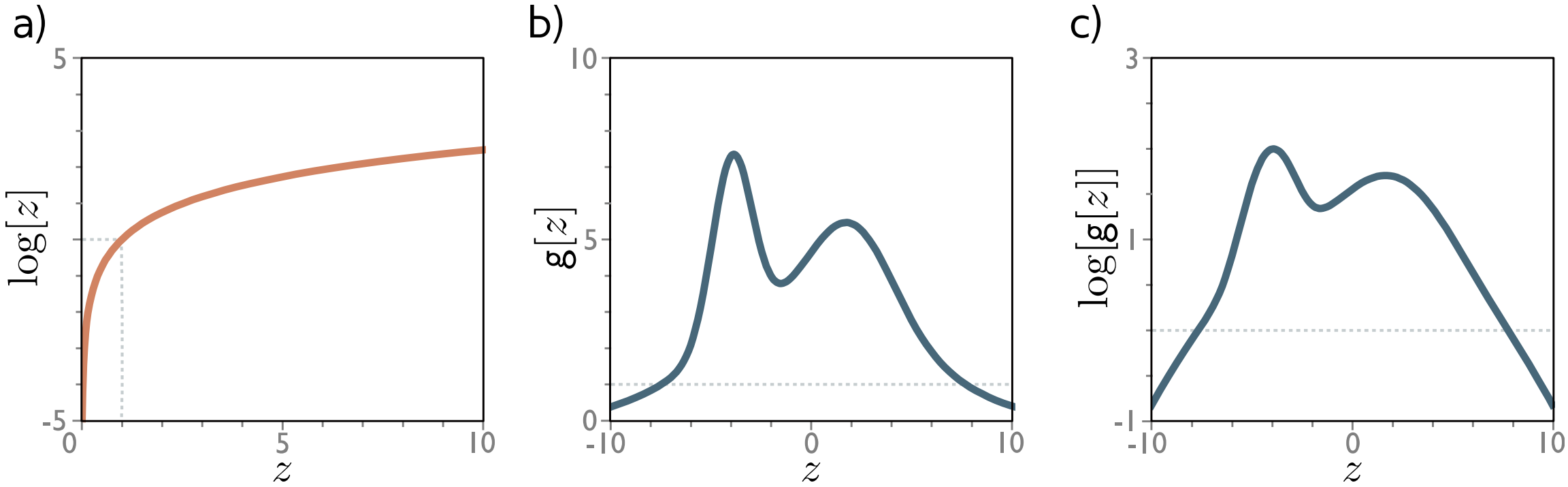
$$\frac{\partial l}{\partial w_{[k]ij}} \qquad \frac{\partial l}{\partial b_{[k]i}}$$

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

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\text{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i | \mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]) \right]$$

The log function is monotonic



Maximum of the logarithm of a function is in the same place as maximum of function

Maximum log likelihood

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\log\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\right]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\sum_{i=1}^{I} \log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\Big]\right]\end{aligned}$$

Minimizing negative log likelihood

- By convention, we minimize things (i.e., a loss)

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\sum_{i=1}^{I} \log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\Big]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\Big]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[\mathrm{L}[\boldsymbol{\phi}]\Big]\end{aligned}$$

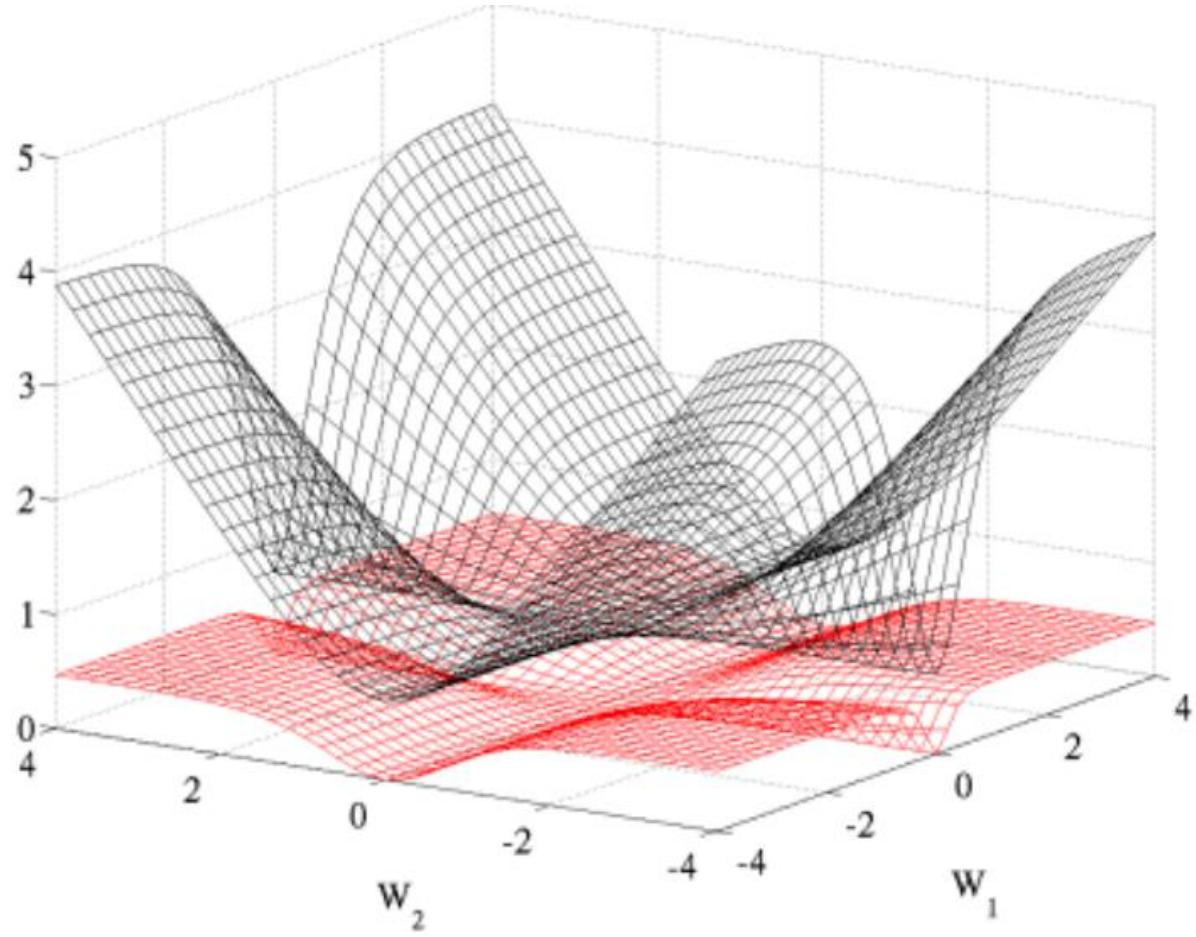


Figure 5: *Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer.*

$$D_{\mathrm{KL}}\left(\hat{p}_{\mathrm{data}} \| p_{\mathrm{model}}\right) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}}\left[\log \hat{p}_{\mathrm{data}}(\boldsymbol{x}) - \log p_{\mathrm{model}}(\boldsymbol{x})\right]$$

$$- \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}}\left[\log p_{\mathrm{model}}(\boldsymbol{x})\right]$$

$$P(y \mid x) = \begin{cases} \hat{p} & \text{if } y = 1 \\ 1 - \hat{p} & \text{if } y = 0 \end{cases}$$

$$P(y \mid x) = \hat{p}^{y}(1-\hat{p})^{1-y}$$

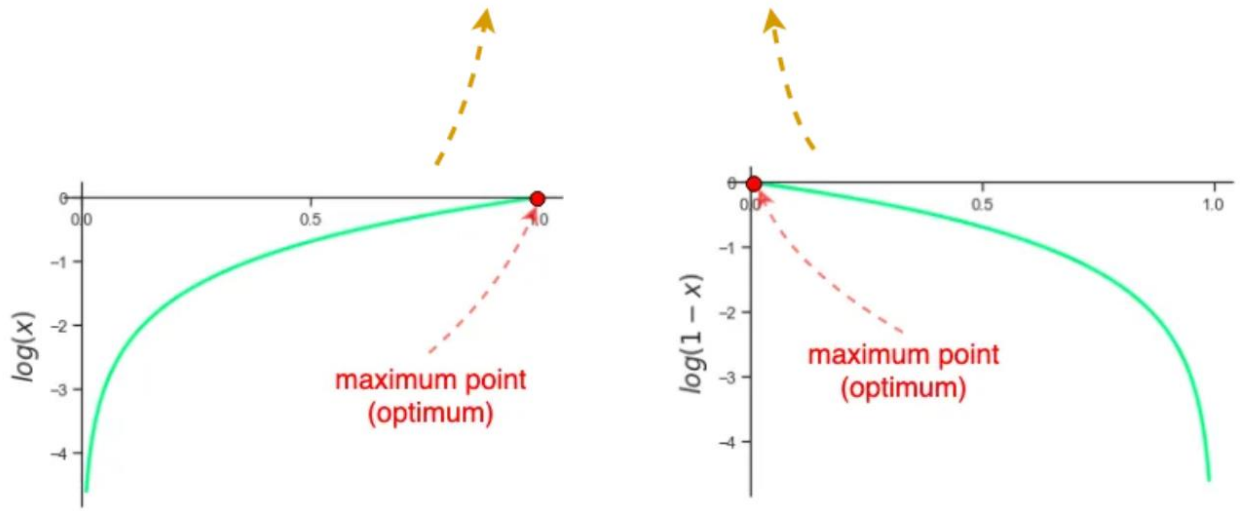
if $y = 1$ then:

$$P(y \mid x) = \hat{p}^{y}(1-\hat{p})^{1-y} = \hat{p}^{1}(1-\hat{p})^{1-1} = \hat{\mathbf{p}}$$

if $y = 0$ then:

$$P(y \mid x) = \hat{p}^{y}(1-\hat{p})^{1-y} = \hat{p}^{0}(1-\hat{p})^{1-0} = \mathbf{1} - \hat{\mathbf{p}}$$

$$\begin{aligned} \log(P(y \mid x) &= \log(\hat{p}^{y}(1-\hat{p})^{(1-y)}) \\ &= \log(\hat{p}^{y}) + \log((1-\hat{p})^{(1-y)}) \\ &= \mathbf{y}\log(\hat{\mathbf{p}}) + (\mathbf{1}-\mathbf{y})\log(\mathbf{1}-\hat{\mathbf{p}}) \end{aligned}$$



Background

A Recipe for Machine Learning

1. Given training data:

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

2. Choose each of these:

- Decision function

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

- Loss function

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

3. Define goal:

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

4. Train with SGD:

(take small steps opposite the gradient)

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

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

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

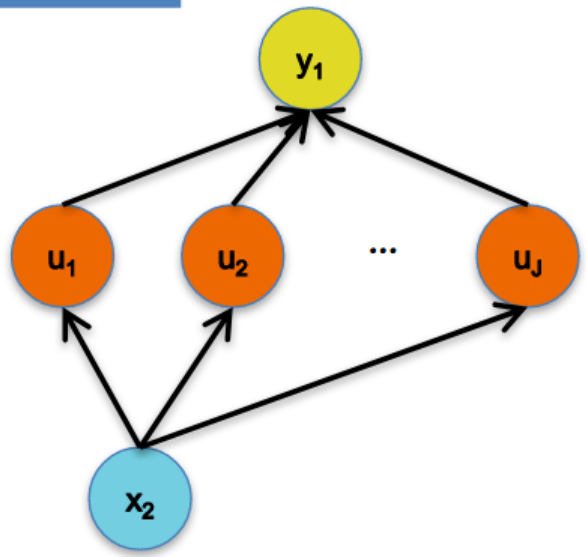
1. Finite Difference Method
   - Pro: Great for testing implementations of backpropagation
   - Con: Slow for high dimensional inputs / outputs
   - Required: Ability to call the function f(**x**) on any input **x**
2. Symbolic Differentiation
   - Note: The method you learned in high-school
   - Note: 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(**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(**x**)
   - Con: Slow for high dimensional outputs (e.g. vector-valued functions)
   - Required: Algorithm for computing f(**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(**x**)
   - Con: Slow for high dimensional inputs (e.g. vector-valued **x**)
   - Required: Algorithm for computing f(**x**)

Backpropagation – repeated application of chain rule

Given: $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

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(**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 $v_1,\ldots, v_N$
   a. Compute $u_i = g_i(v_1,\ldots, v_N)$
   b. Store the result at the node

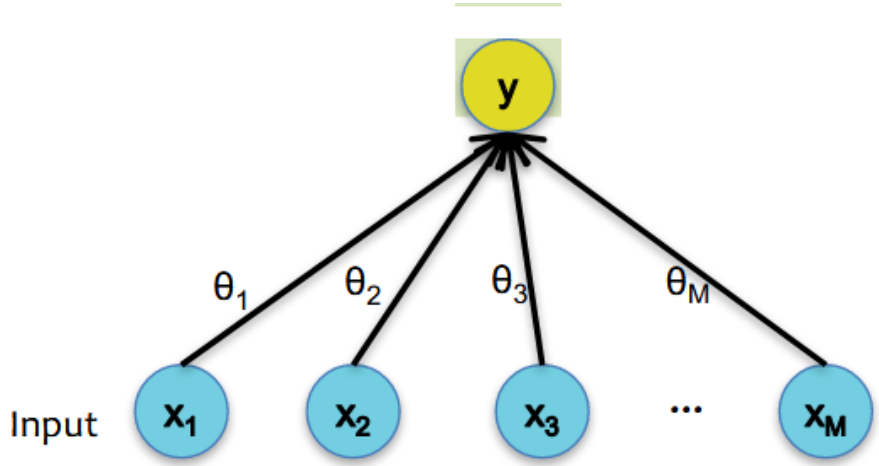
Backward Computation

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
   For variable $u_i = g_i(v_1,\ldots, v_N)$
   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	Backward
$J = cos(u)$	$\frac{dJ}{du}$ += $-sin(u)$
$u = u_1 + u_2$	$\frac{dJ}{du_1}$ += $\frac{dJ}{du}\frac{du}{du_1}$, $\frac{du}{du_1} = 1$ $\quad$ $\frac{dJ}{du_2}$ += $\frac{dJ}{du}\frac{du}{du_2}$, $\frac{du}{du_2} = 1$
$u_1 = sin(t)$	$\frac{dJ}{dt}$ += $\frac{dJ}{du_1}\frac{du_1}{dt}$, $\frac{du_1}{dt} = \cos(t)$
$u_2 = 3t$	$\frac{dJ}{dt}$ += $\frac{dJ}{du_2}\frac{du_2}{dt}$, $\frac{du_2}{dt} = 3$
$t = x^2$	$\frac{dJ}{dx}$ += $\frac{dJ}{dt}\frac{dt}{dx}$, $\frac{dt}{dx} = 2x$

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

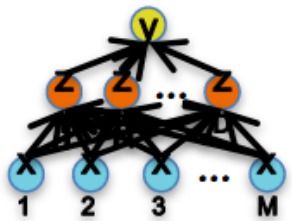
$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

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

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

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

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

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

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

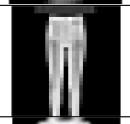
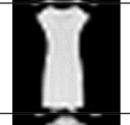
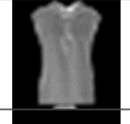
$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

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

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

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

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

Label	Description	Example
0	T-shirt/top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandal	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boot	

Normalized CNN Confusion Matrix

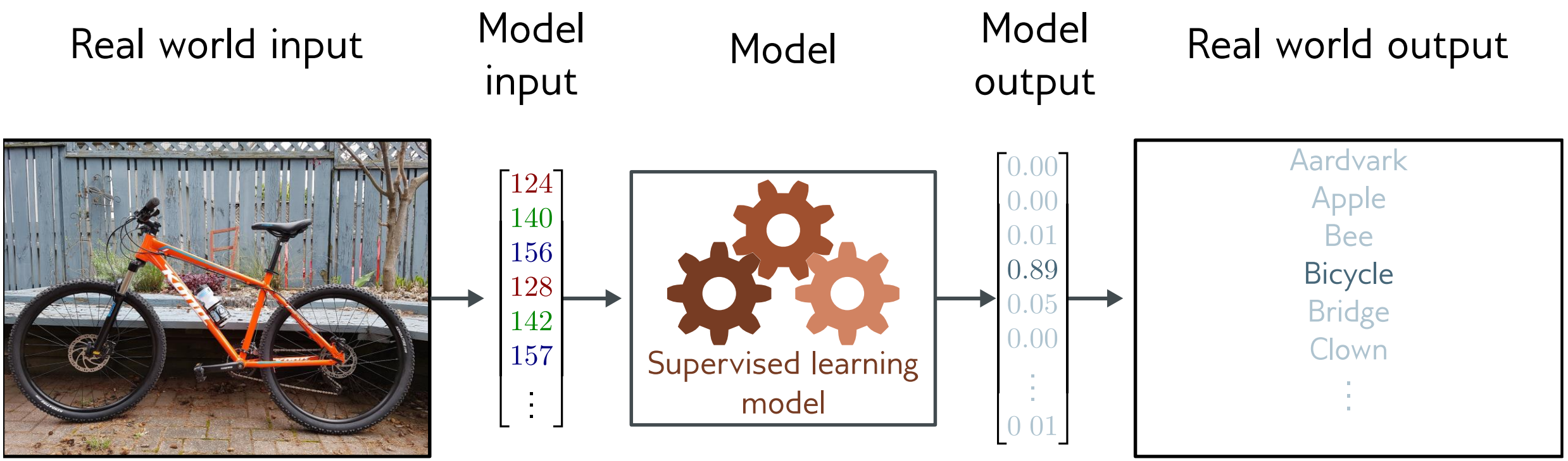
	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	0.83	0.00	0.01	0.03	0.00	0.00	0.12	0.00	0.00	0.00
Trouser	0.00	0.99	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
Pullover	0.01	0.00	0.85	0.01	0.07	0.00	0.07	0.00	0.00	0.00
Dress	0.01	0.00	0.00	0.93	0.03	0.00	0.02	0.00	0.00	0.00
Coat	0.00	0.00	0.05	0.03	0.86	0.00	0.06	0.00	0.00	0.00
Sandal	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.01	0.00	0.00
Shirt	0.08	0.00	0.05	0.02	0.07	0.00	0.78	0.00	0.00	0.00
Sneaker	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.97	0.00	0.02
Bag	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.99	0.00
Ankle boot	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.96

Normalized MLP Confusion Matrix

	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	0.84	0.00	0.01	0.03	0.00	0.00	0.11	0.00	0.01	0.00
Trouser	0.00	0.98	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00
Pullover	0.01	0.00	0.80	0.01	0.11	0.00	0.07	0.00	0.00	0.00
Dress	0.02	0.01	0.01	0.91	0.03	0.00	0.02	0.00	0.00	0.00
Coat	0.00	0.00	0.09	0.04	0.80	0.00	0.06	0.00	0.00	0.00
Sandal	0.00	0.00	0.00	0.00	0.00	0.97	0.00	0.02	0.00	0.01
Shirt	0.13	0.00	0.08	0.03	0.07	0.00	0.68	0.00	0.01	0.00
Sneaker	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.94	0.00	0.03
Bag	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.98	0.00
Ankle boot	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.04	0.00	0.92

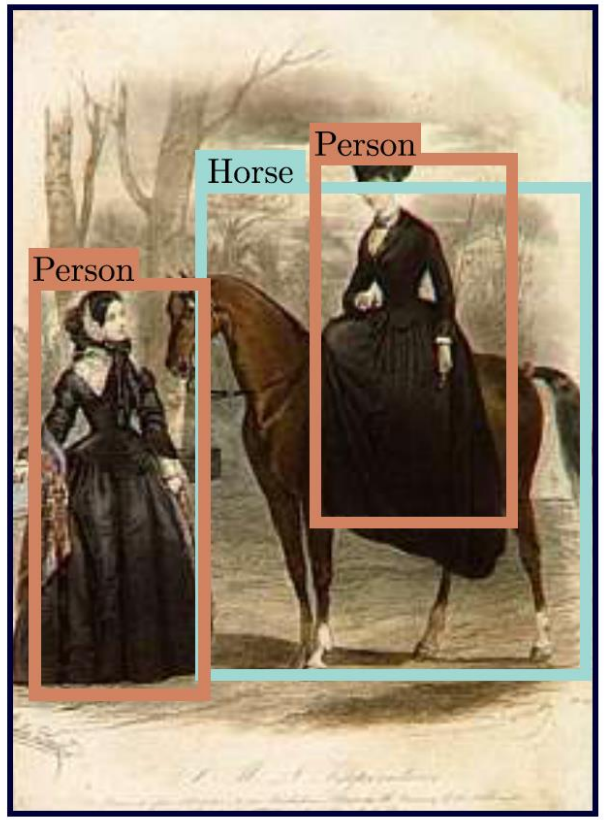
	CNN	MLP
Accuracy	0.915	0.880
Cohen's Kappa	0.906	0.868
MCC	0.906	0.869
Cross-Entropy	0.227	0.324

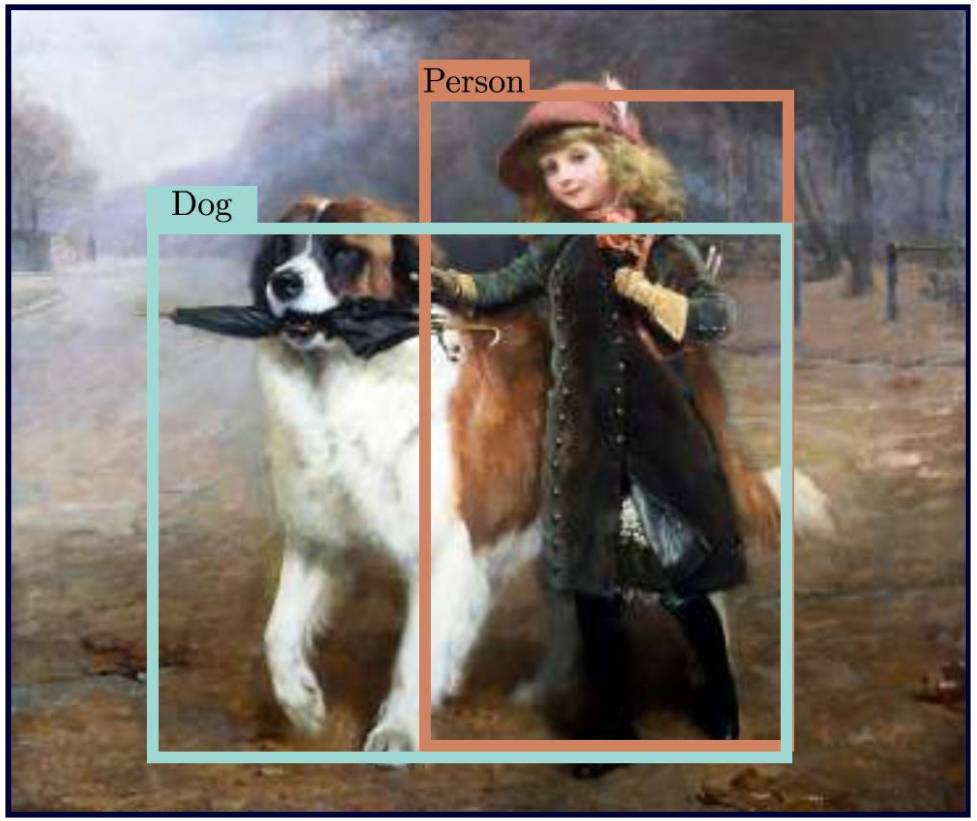
Image classification

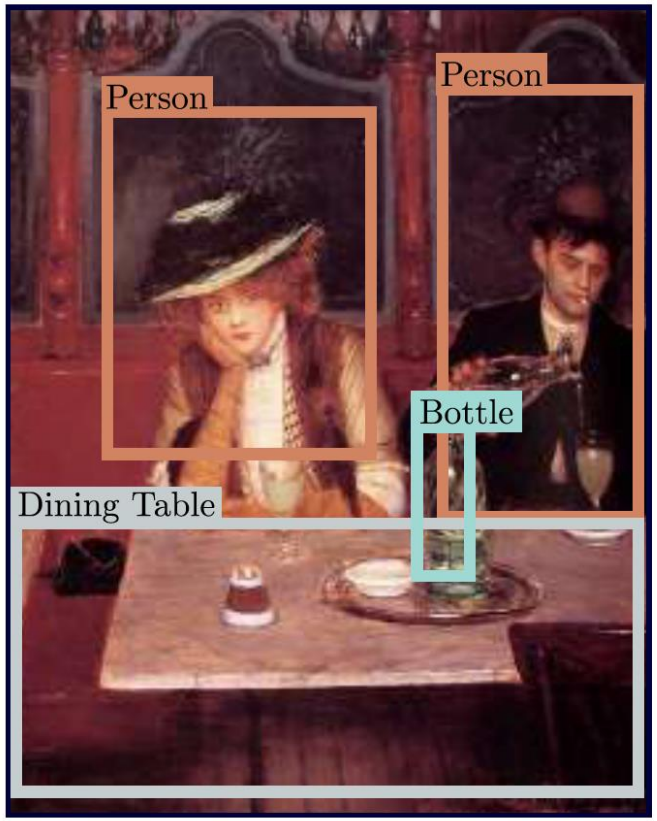


- Multiclass classification problem (discrete classes, >2 possible classes)
- Convolutional network

Object detection

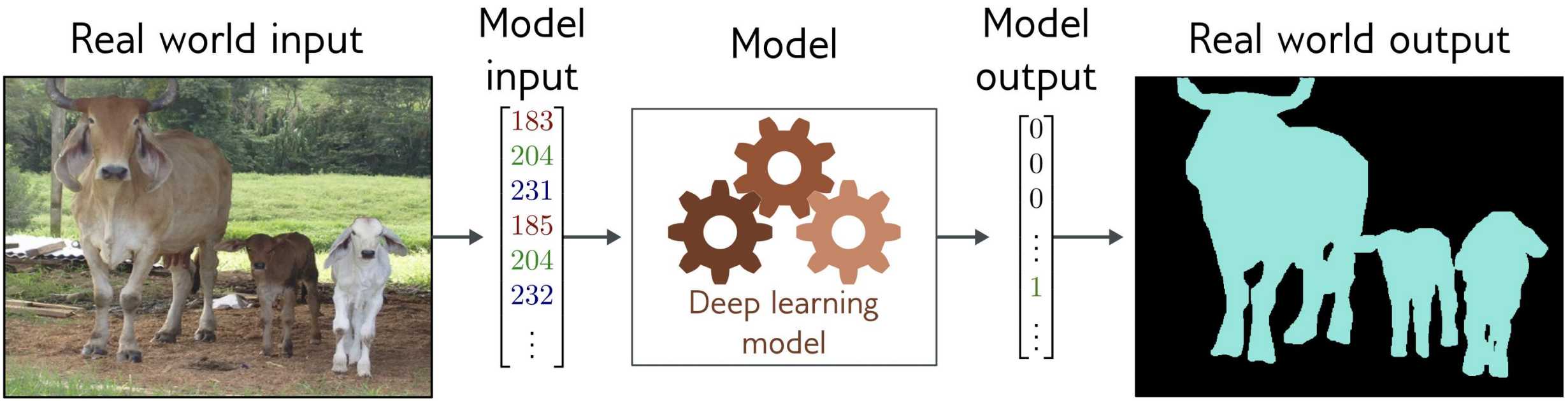






- Multivariate regression problem
- Convolutional encoder-decoder network

Image segmentation



- Multivariate binary classification problem (many outputs, two discrete classes)
- Convolutional encoder-decoder network

Invariance

- A function f[x] is invariant to a transformation t[] if:

$$\mathbf{f}[\mathbf{t}[\mathbf{x}]] = \mathbf{f}[\mathbf{x}]$$

i.e., the function output is the same even after the transformation is applied.

Invariance example

e.g., Image classification

- Image has been translated, but we want our classifier to give the same result

Equivariance

- A function f[x] is equivariant to a transformation t[] if:

$$\mathbf{f}\left[\mathbf{t}[\mathbf{x}]\right] = \mathbf{t}\left[\mathbf{f}[\mathbf{x}]\right]$$

i.e., the output is transformed in the same way as the input

Equivariance example

e.g., Image segmentation

- Image has been translated and we want segmentation to translate with it

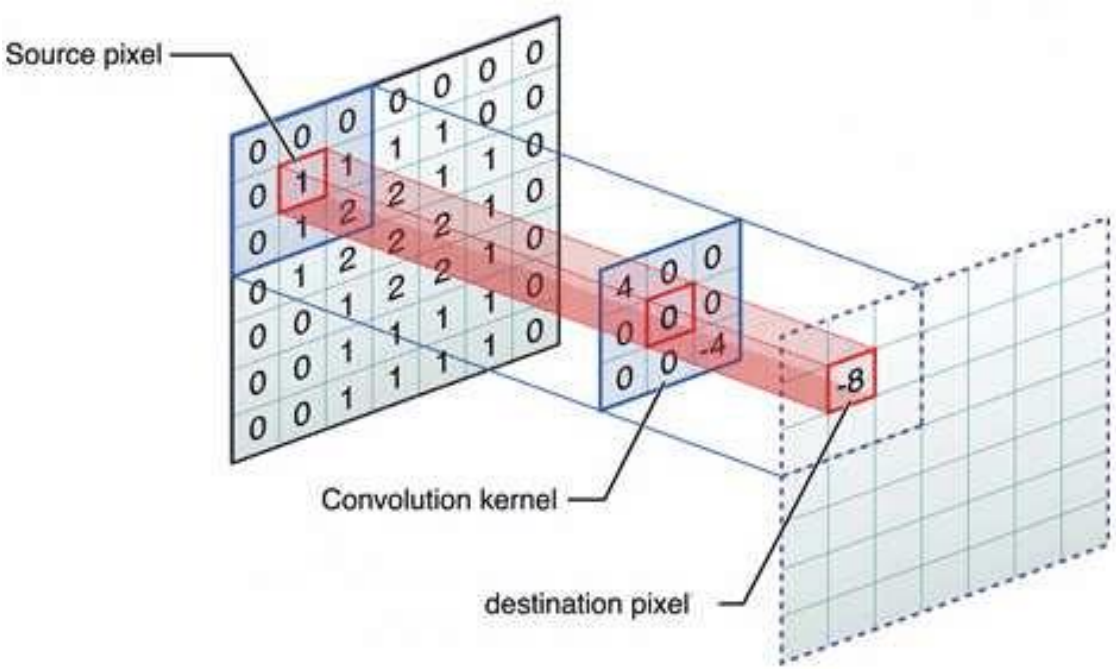


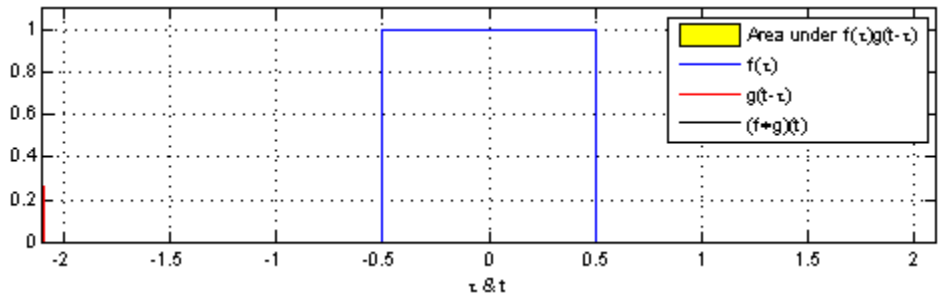

Basic building blocks of the CNN architecture

- Input layer
- Convolutional layer
- Fully connected layer
- Loss layer

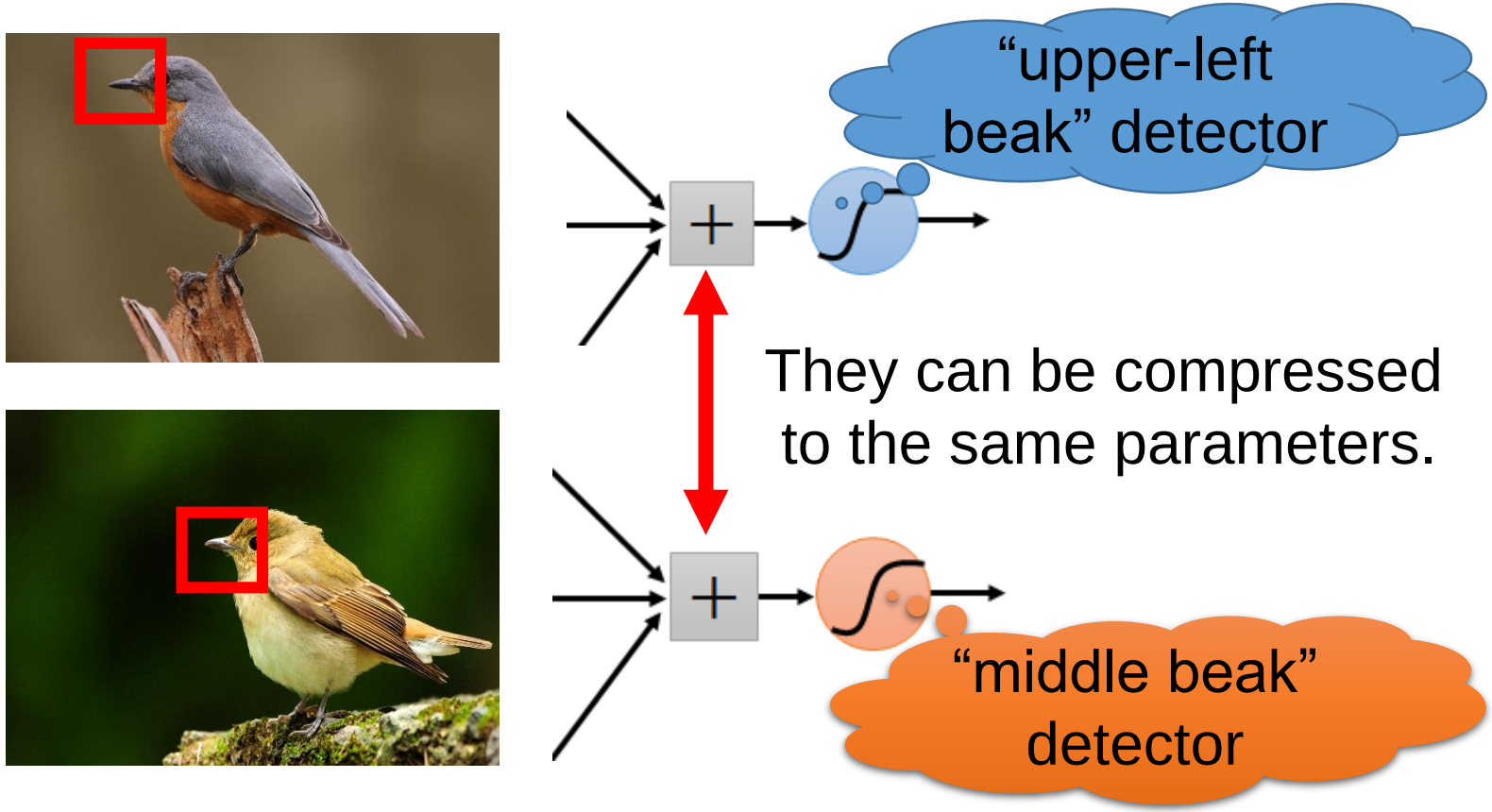
- Convolutional layer
  - Convolutional kernel
  - Pooling layer
  - Non-linearity

Convolution operation

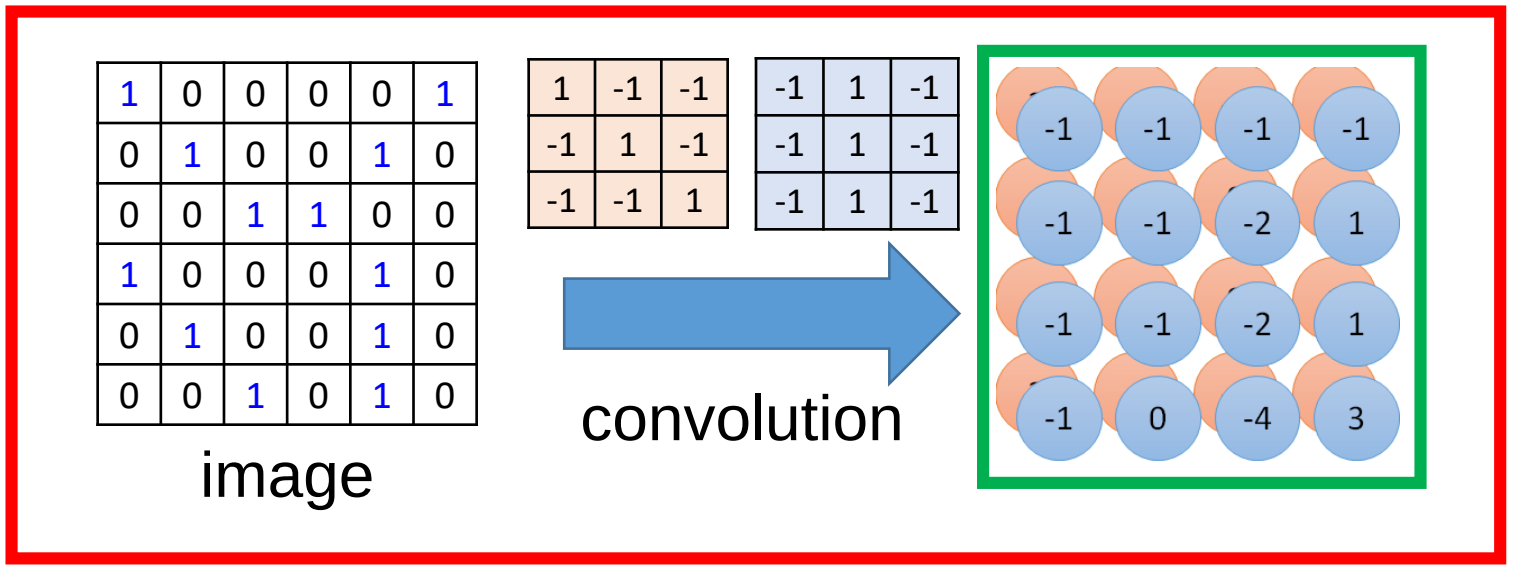




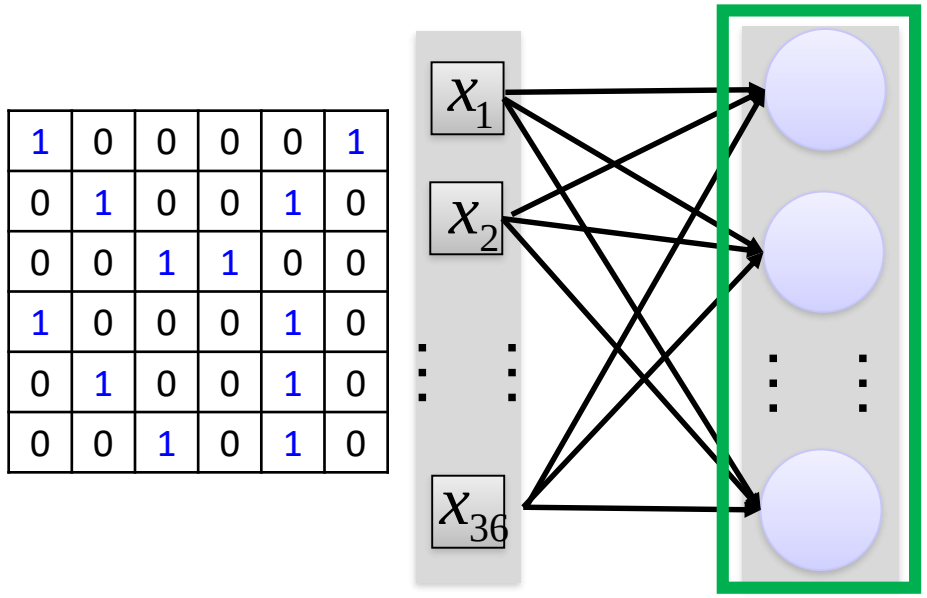
Same pattern appears in different places:
They can be compressed!
What about training a lot of such “small” detectors
and each detector must “move around”.



Convolution v.s. Fully Connected



Fully-connected



Convolution* in 1D

- Input vector **x:**

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

- Output is weighted sum of neighbors:

$$z_i = \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}$$

- Convolutional kernel or filter:

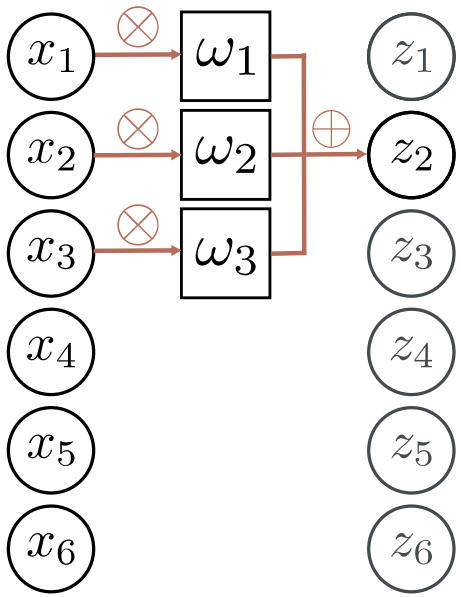
$$\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]^T$$

Kernel size = 3

* Not really technically convolution

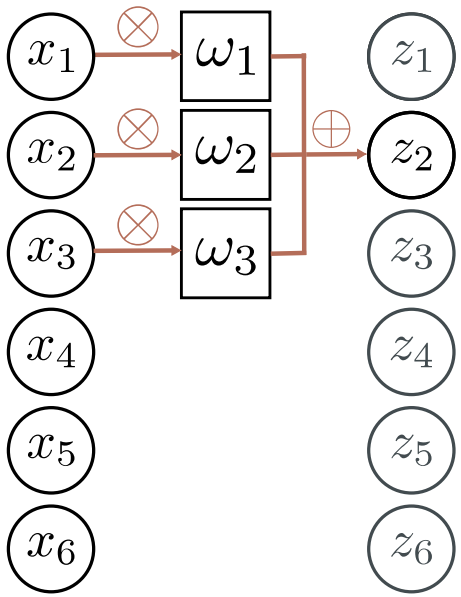
Convolution with kernel size 3

a)

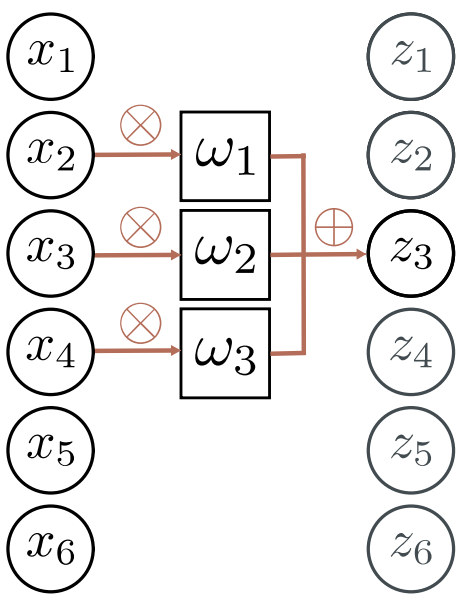


Convolution with kernel size 3

a)

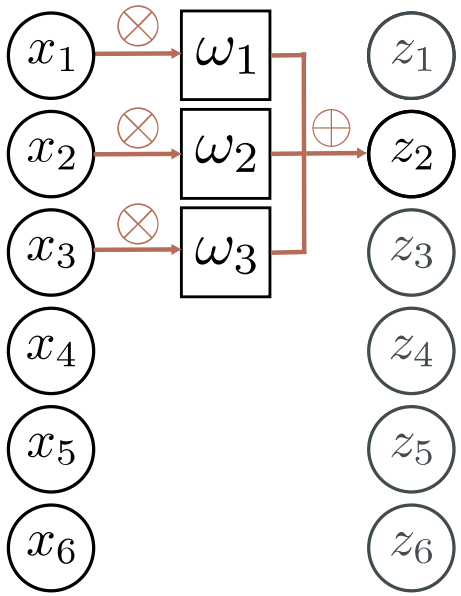


b)

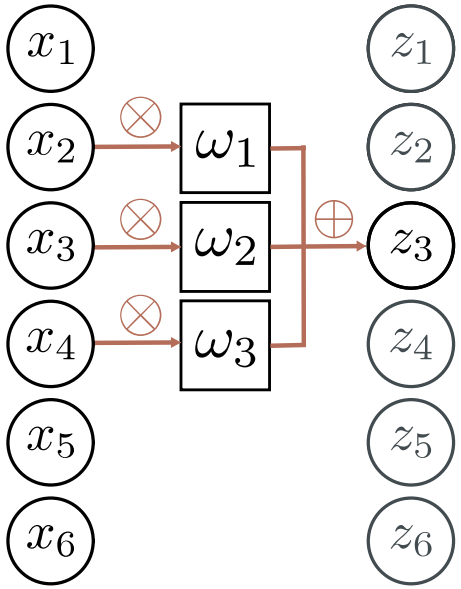


Convolution with kernel size 3

a)



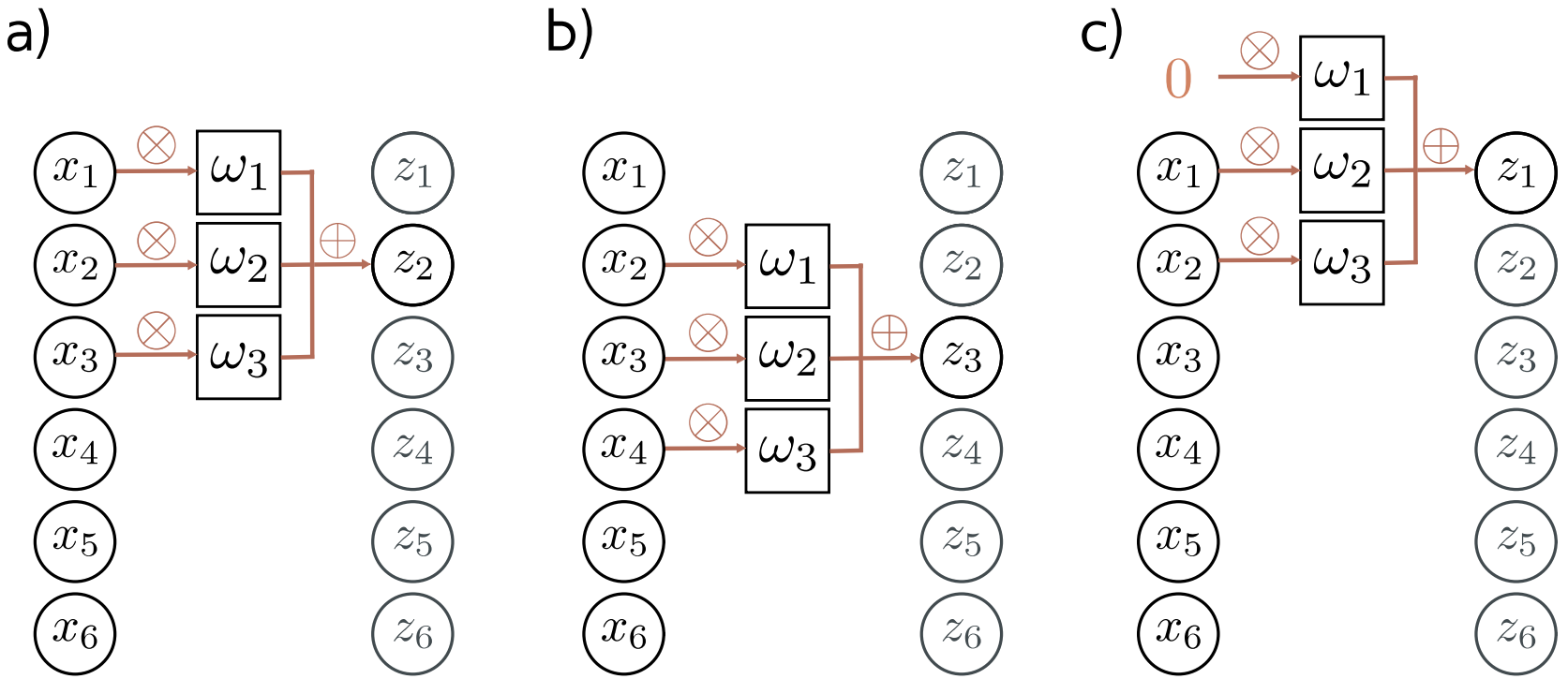
b)



Equivariant to translation of input

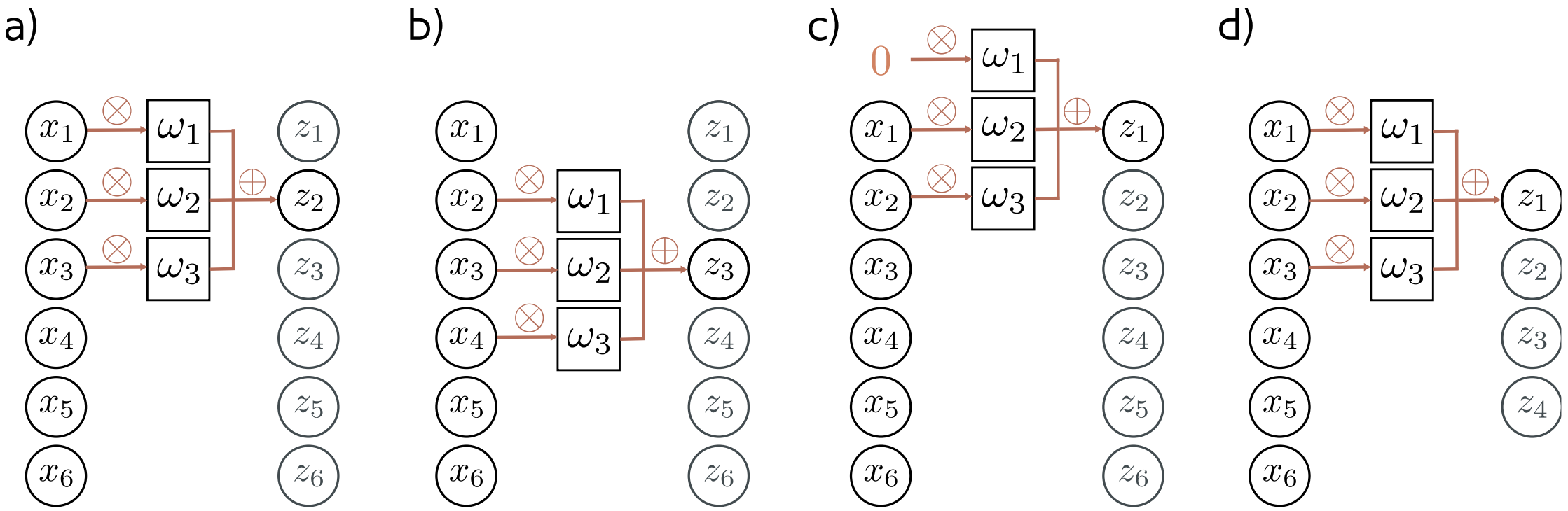
$$\mathbf{f}[\mathbf{t}[\mathbf{x}]] = \mathbf{t}[\mathbf{f}[\mathbf{x}]]$$

Zero padding



Treat positions that are beyond end of the input as zero.

“Valid” convolutions

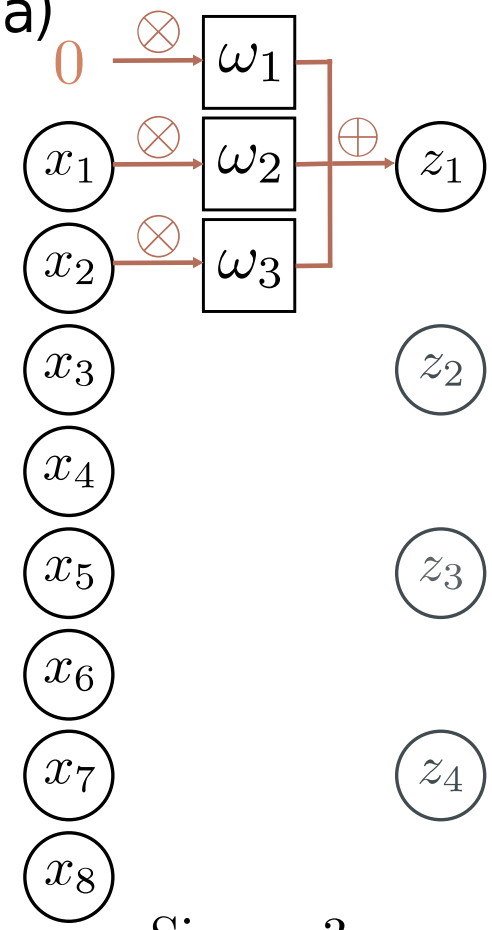


Only process positions where kernel falls in image (smaller output).

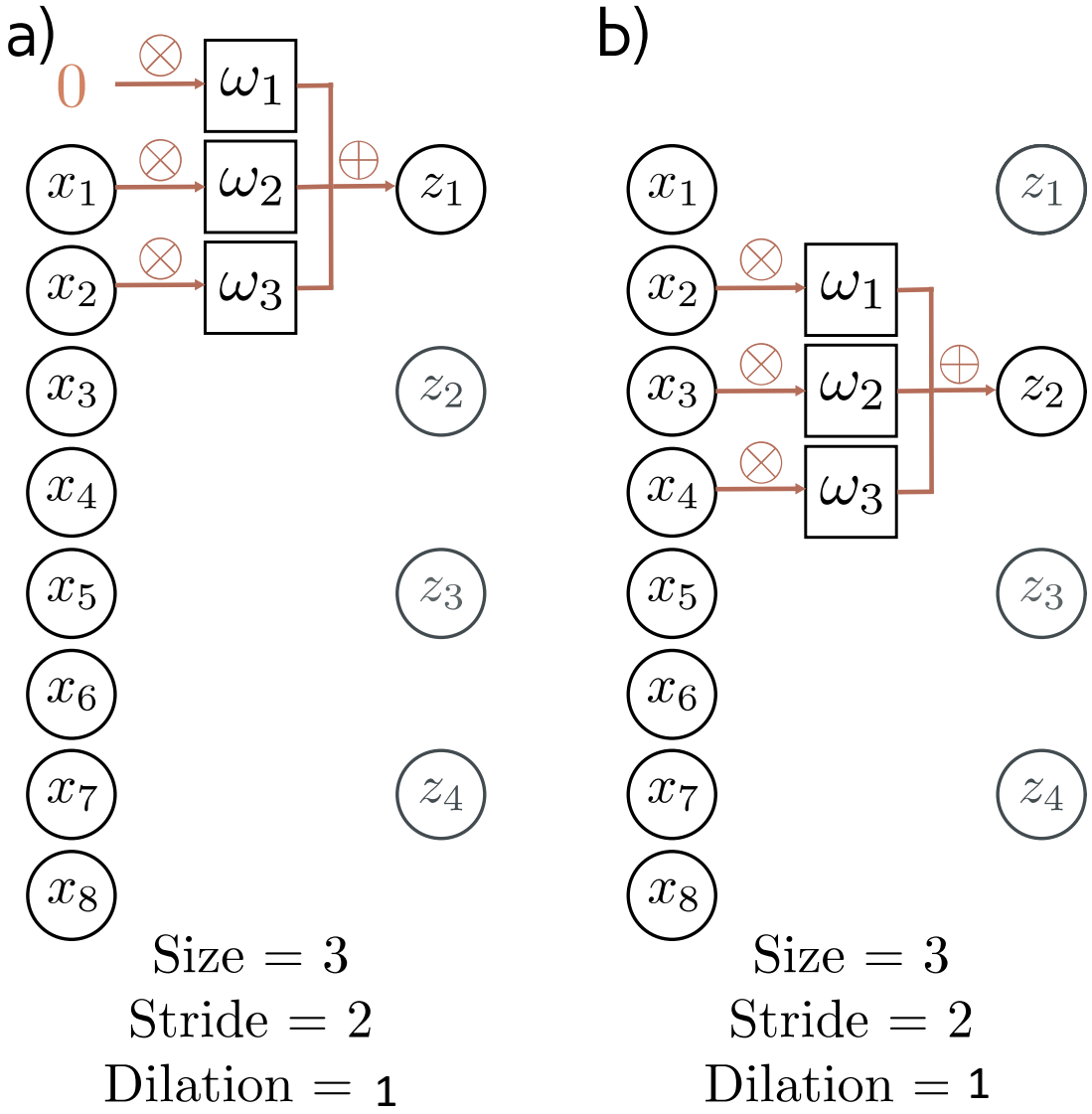
Stride, kernel size, and dilation

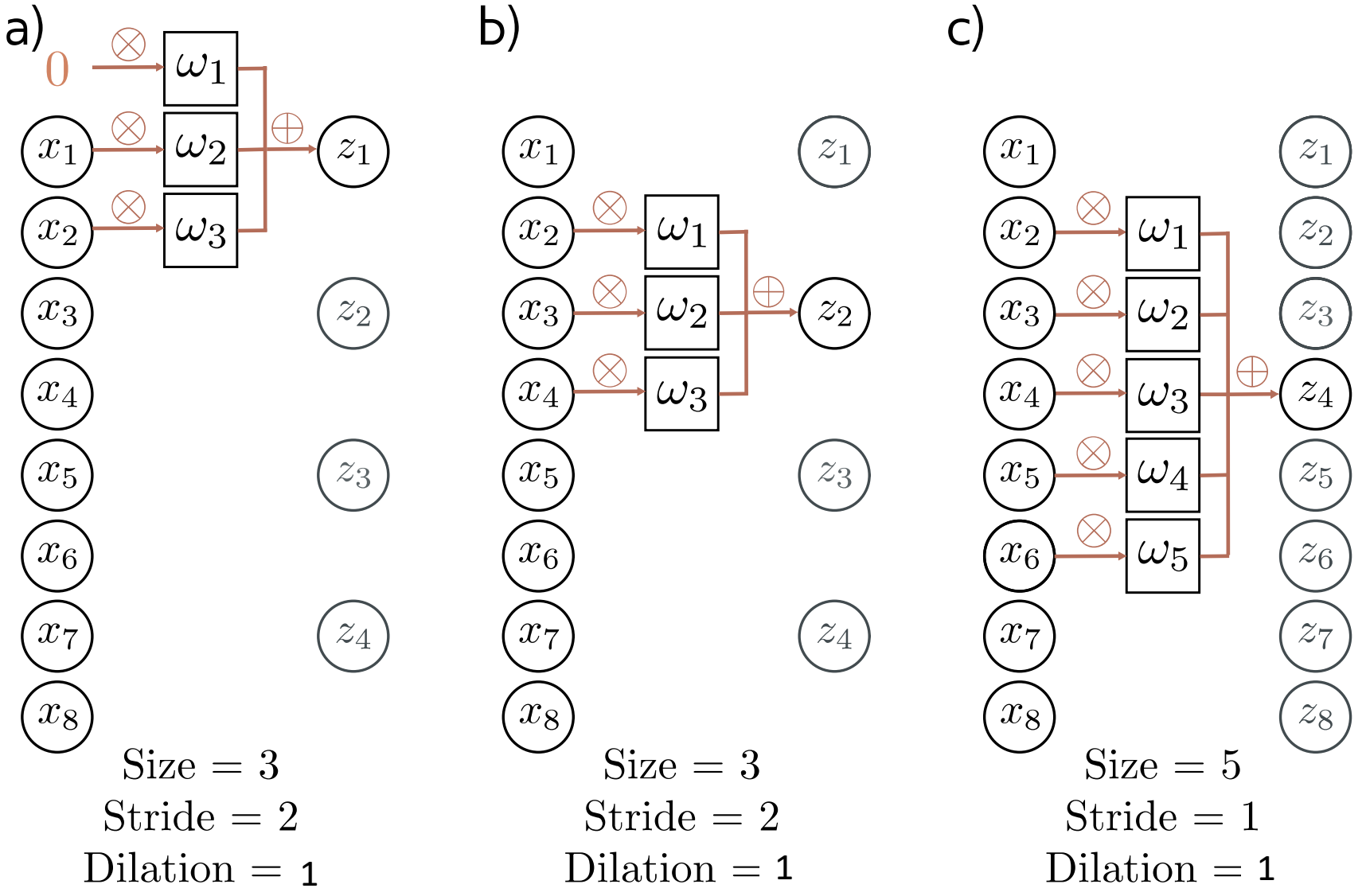
- Stride = shift by k positions for each output
  - Decreases size of output relative to input
- Kernel size = weight a different number of inputs for each output
  - Combine information from a larger area
  - But kernel size 5 uses 5 parameters
- Dilated or atrous convolutions = intersperse kernel values with zeros
  - Combine information from a larger area
  - Fewer parameters

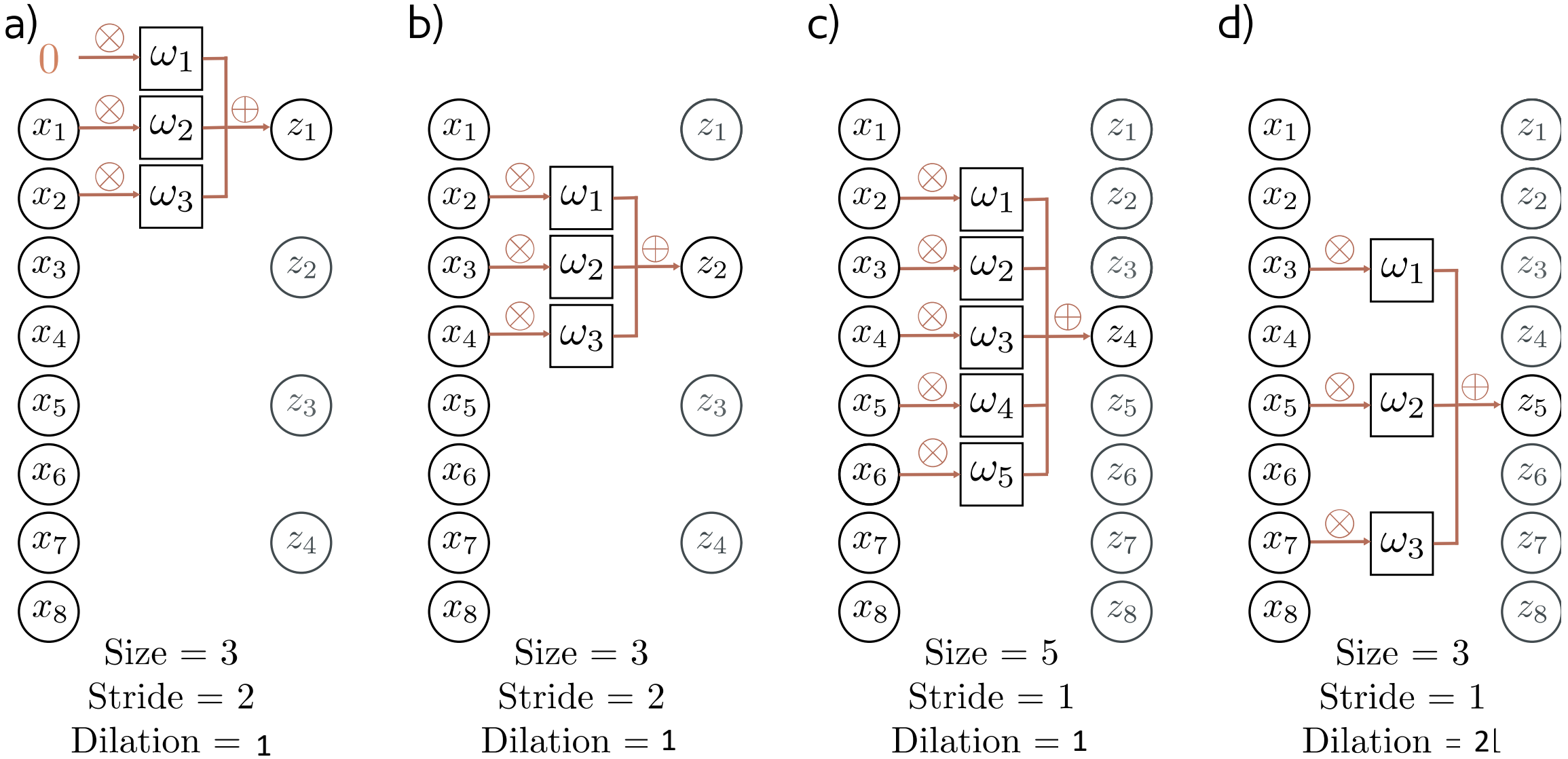
a)



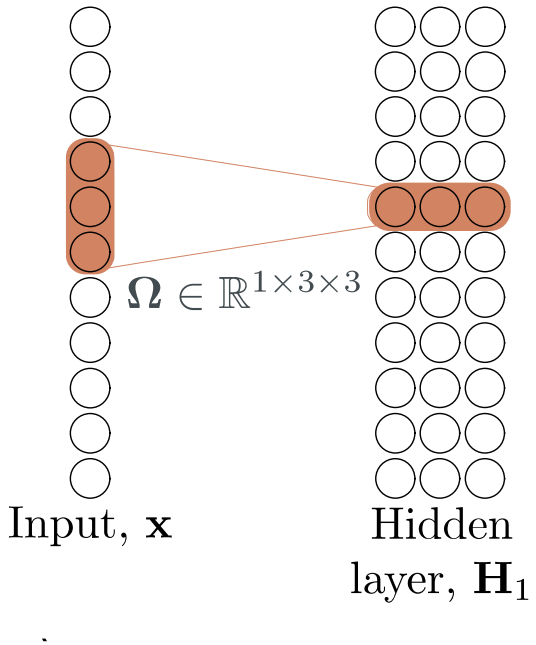
Size = 3
Stride = 2
Dilation = 1


a)
0
ω_1
ω_2
ω_3
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
z_1
z_2
z_3
z_4
Size = 3
Stride = 2
Dilation = 1
b)
ω_1
ω_2
ω_3
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
z_1
z_2
z_3
z_4
Size = 3
Stride = 2
Dilation = 1

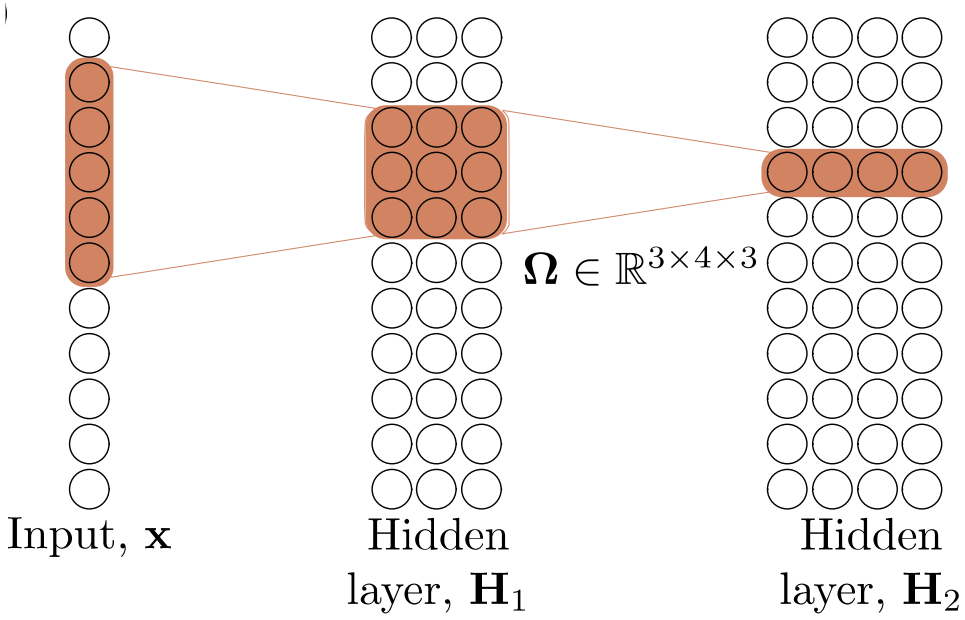

a)
0
ω_1
ω_2
ω_3
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
z_1
z_2
z_3
z_4
Size = 3
Stride = 2
Dilation = 1
b)
ω_1
ω_2
ω_3
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
z_1
z_2
z_3
z_4
Size = 3
Stride = 2
Dilation = 1
c)
ω_1
ω_2
ω_3
ω_4
ω_5
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
z_1
z_2
z_3
z_4
z_5
z_6
z_7
z_8
Size = 5
Stride = 1
Dilation = 1


a)
0
ω_1
ω_2
ω_3
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
z_1 z_2 z_3 z_4
Size = 3
Stride = 2
Dilation = 1
b)
ω_1
ω_2
ω_3
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
z_1 z_2 z_3 z_4
Size = 3
Stride = 2
Dilation = 1
c)
ω_1
ω_2
ω_3
ω_4
ω_5
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8
Size = 5
Stride = 1
Dilation = 1
d)
ω_1
ω_2
ω_3
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8
Size = 3
Stride = 1
Dilation = 2

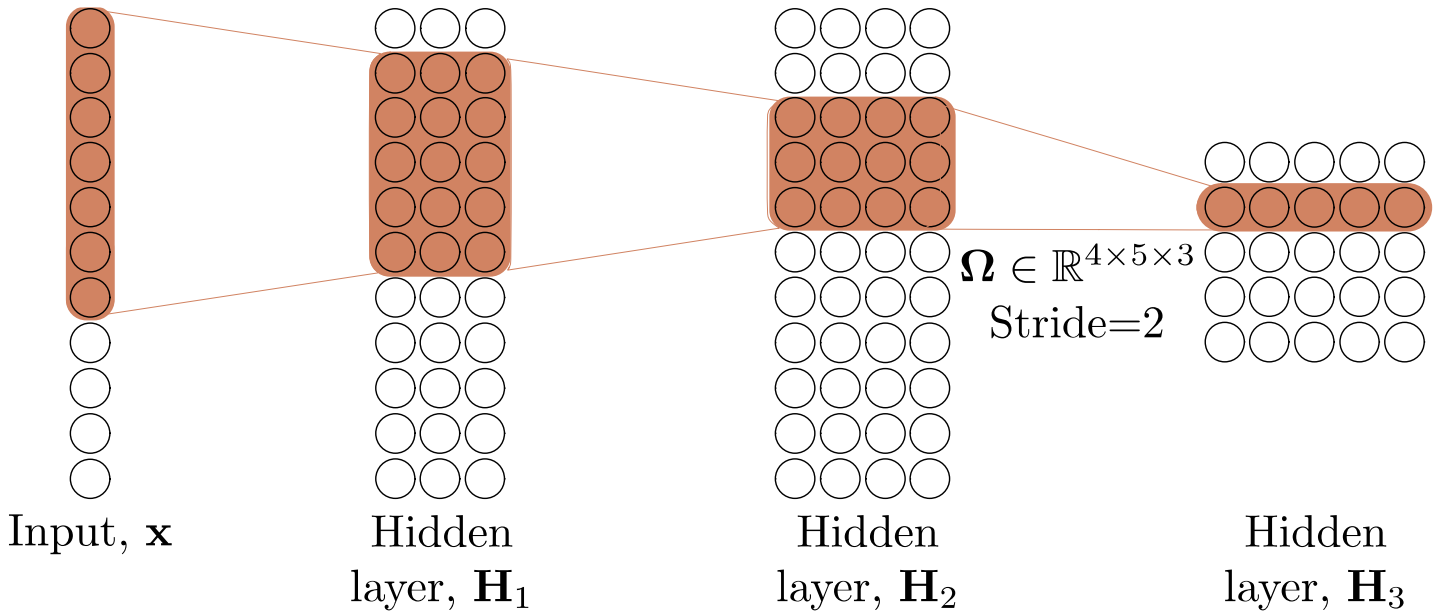
Receptive fields



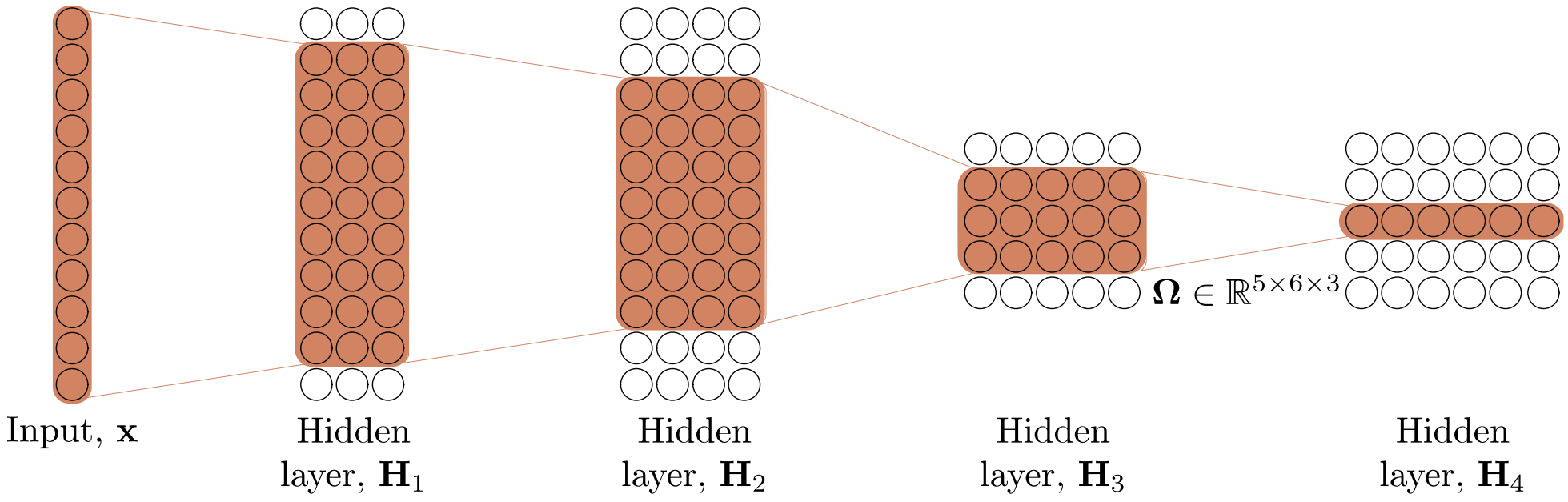
Receptive fields



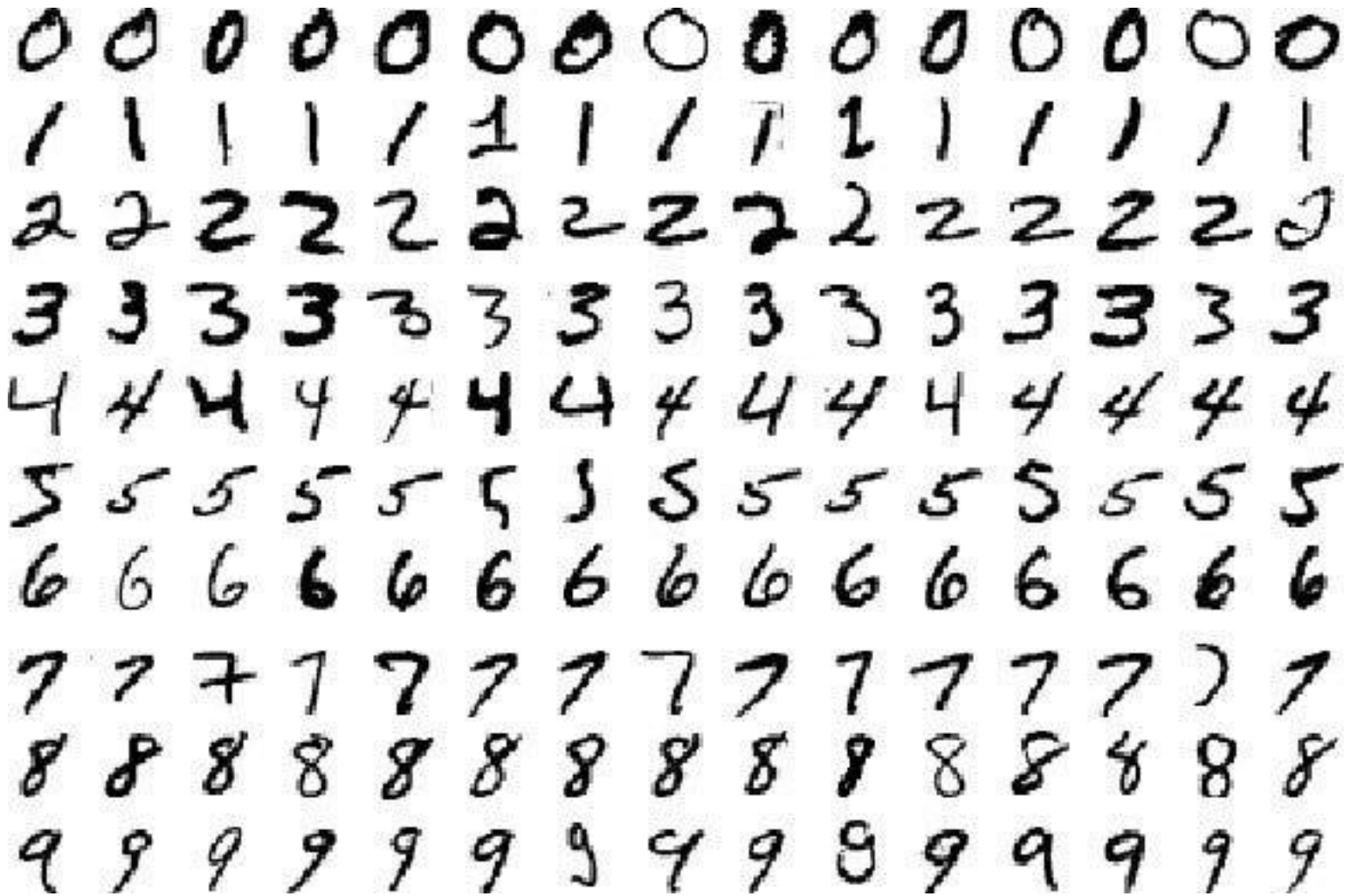
Receptive fields



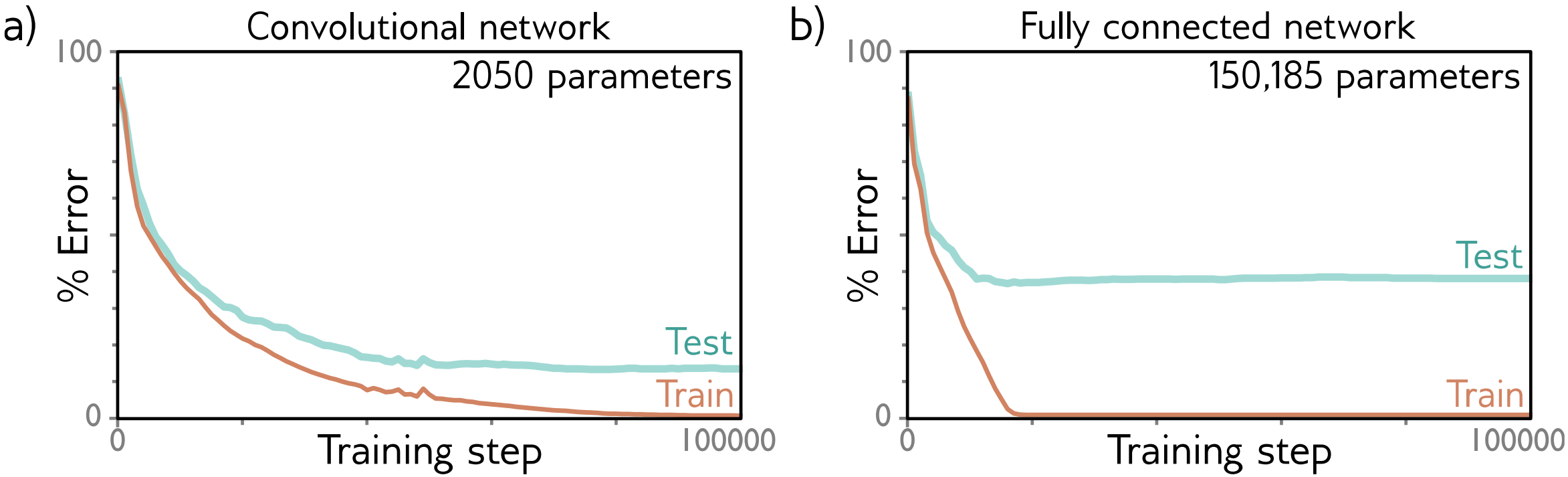
Receptive fields



MNIST digits classification



Performance



Why?

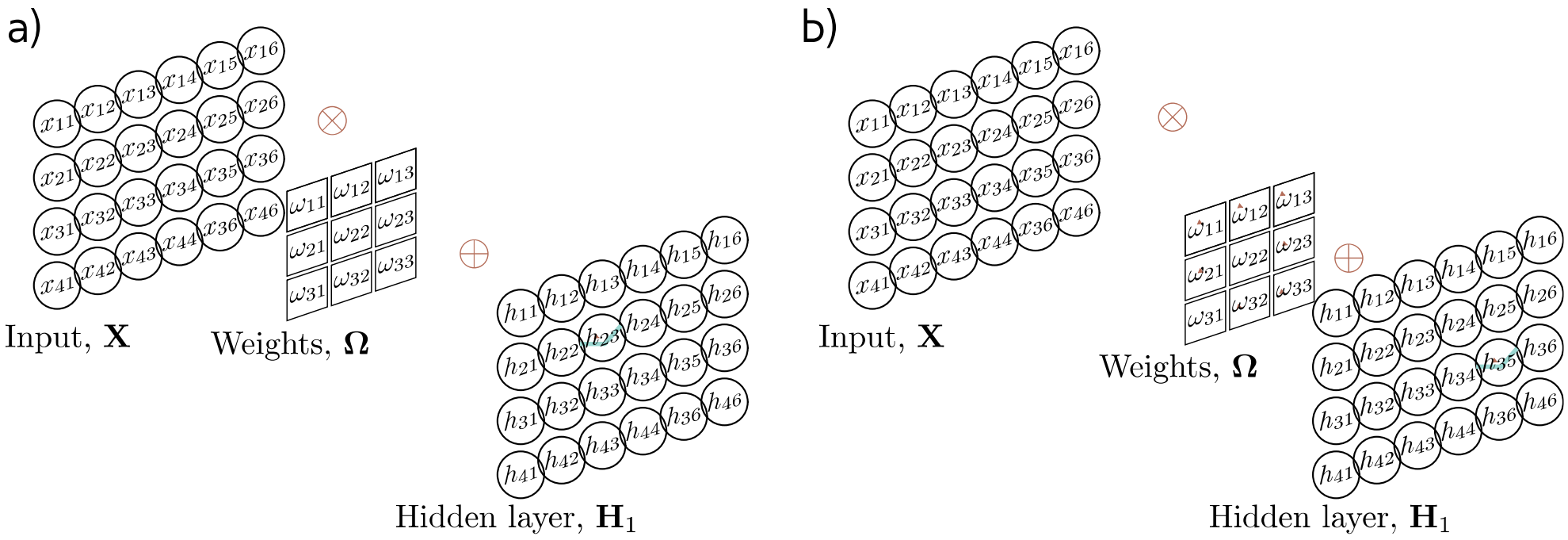
- Better inductive bias
- Forced the network to process each location similarly
- Shares information across locations
- Search through a smaller family of input/ouput mappings, all of which are plausible

2D Convolution

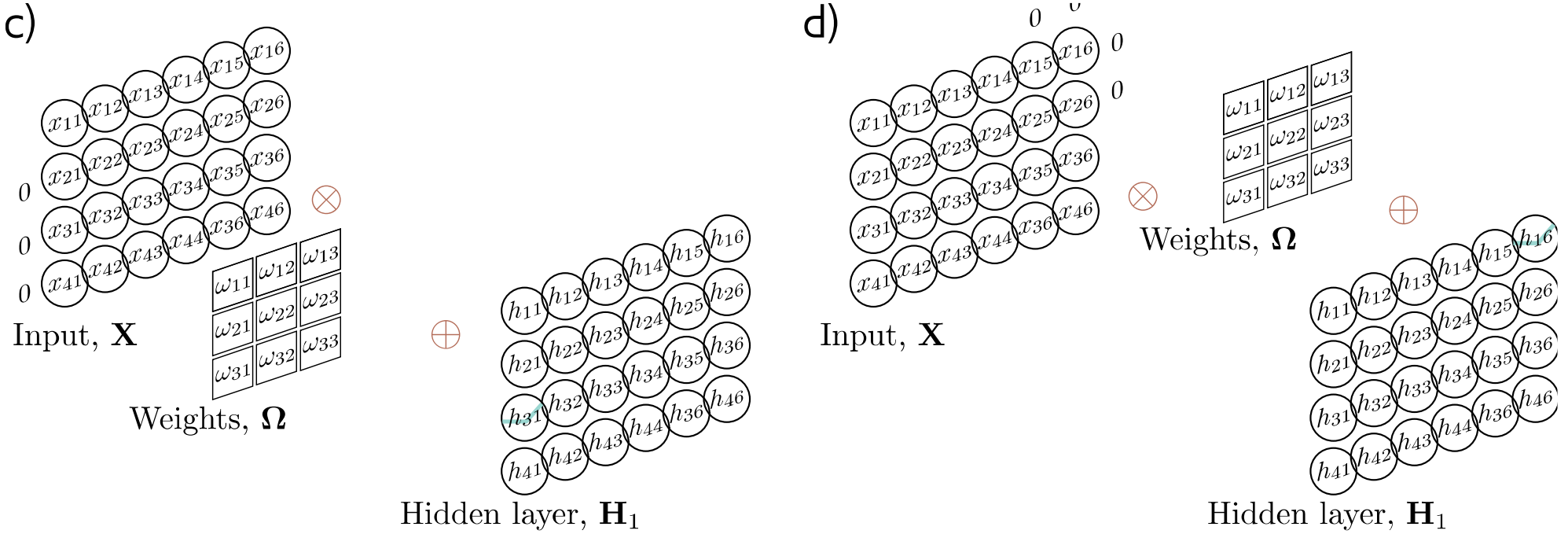
- Convolution in 2D
  - Weighted sum over a K x K region
  - K x K weights
- Build into a convolutional layer by adding bias and passing through activation function

$$h_{i,j} = \mathrm{a}\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3}\omega_{m,n}x_{i+m-2,j+n-2}\right]$$

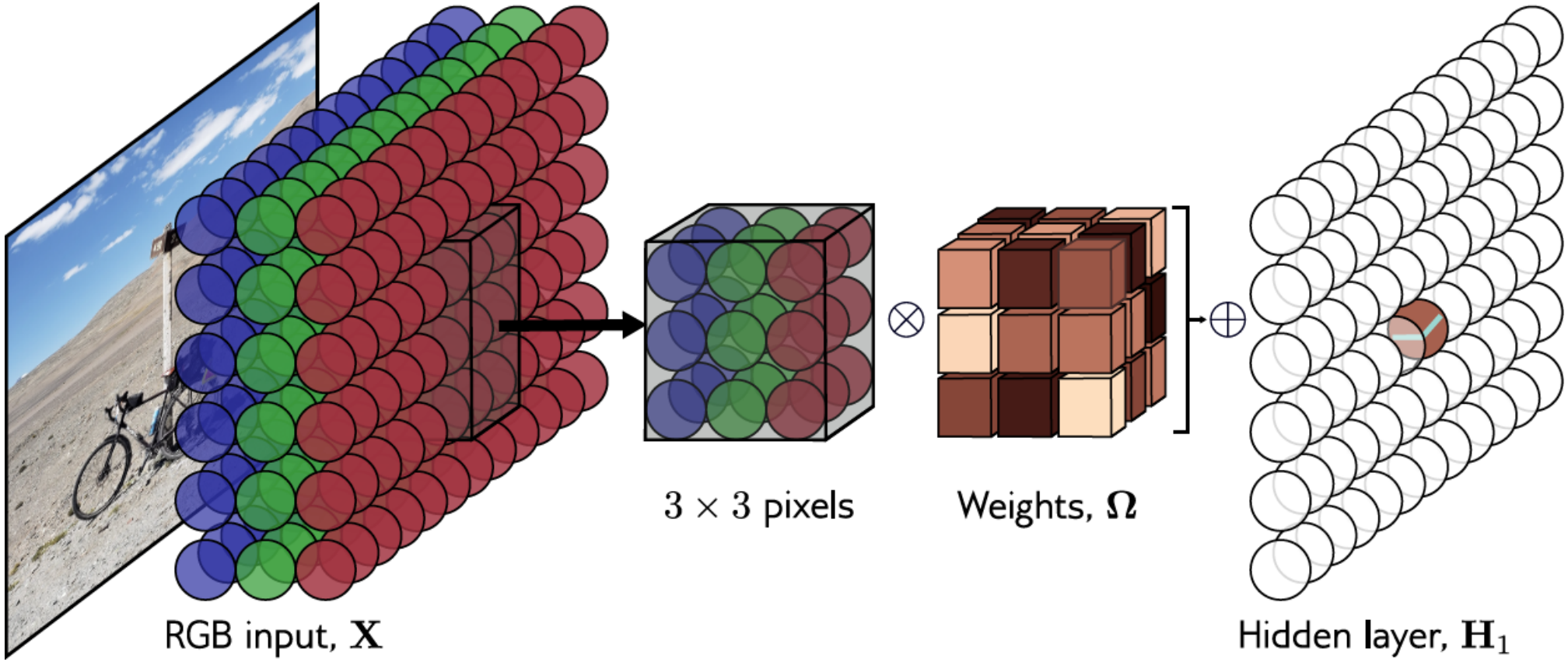
2D Convolution



2D Convolution



Channels in 2D convolution



Kernel size, stride, dilation all work as you would expect

How many parameters?

- If there are C_i input channels and kernel size K x K

$$\boldsymbol{\omega} \in \mathbb{R}^{C_i \times K \times K} \qquad \boldsymbol{\beta} \in \mathbb{R}$$

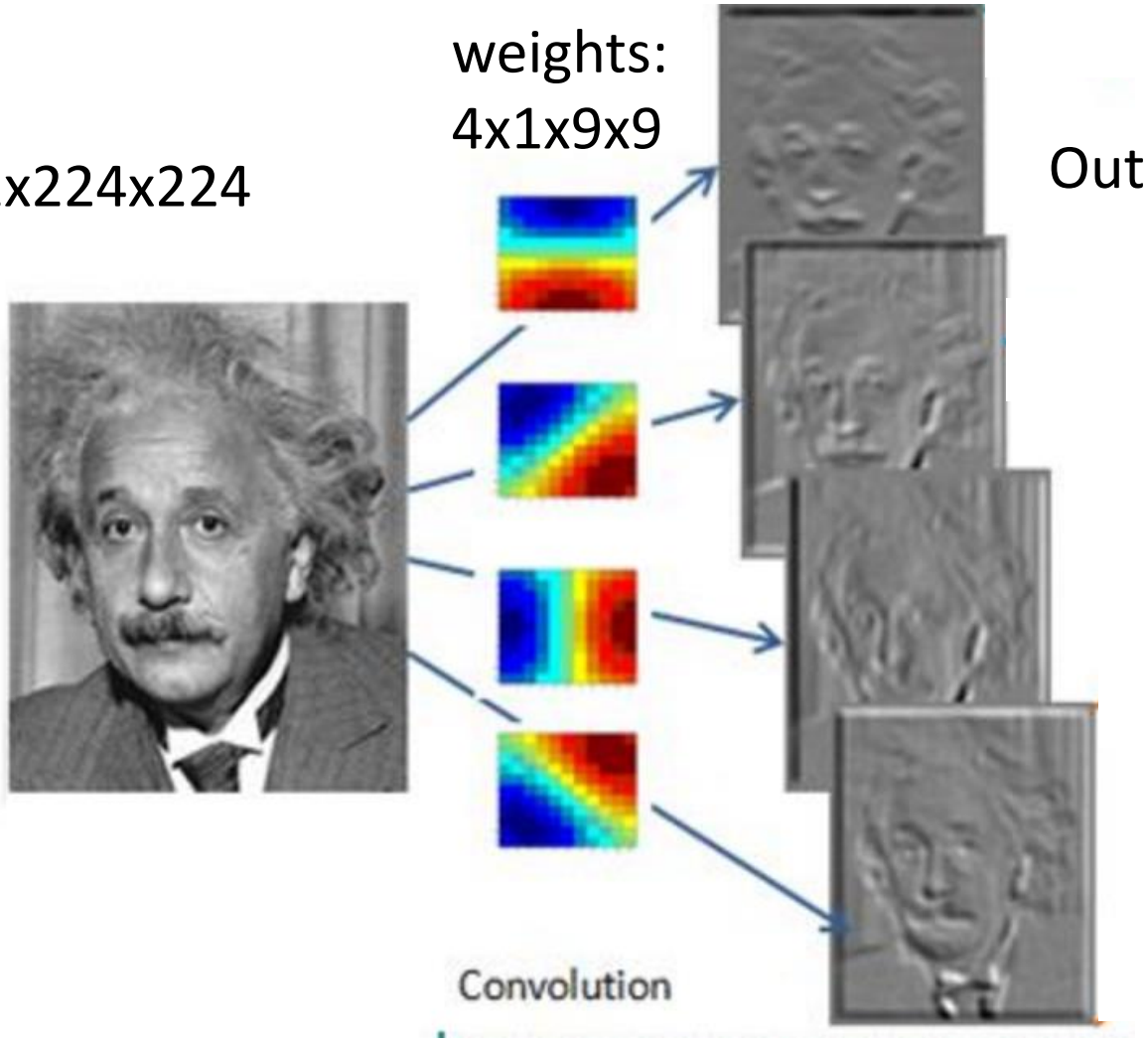
- If there are C_i input channels and C_o output channels

$$\boldsymbol{\omega} \in \mathbb{R}^{C_i \times C_o \times K \times K} \qquad \boldsymbol{\beta} \in \mathbb{R}^{C_o}$$

Convolutional Layer (with 4 filters)

Input: 1x224x224

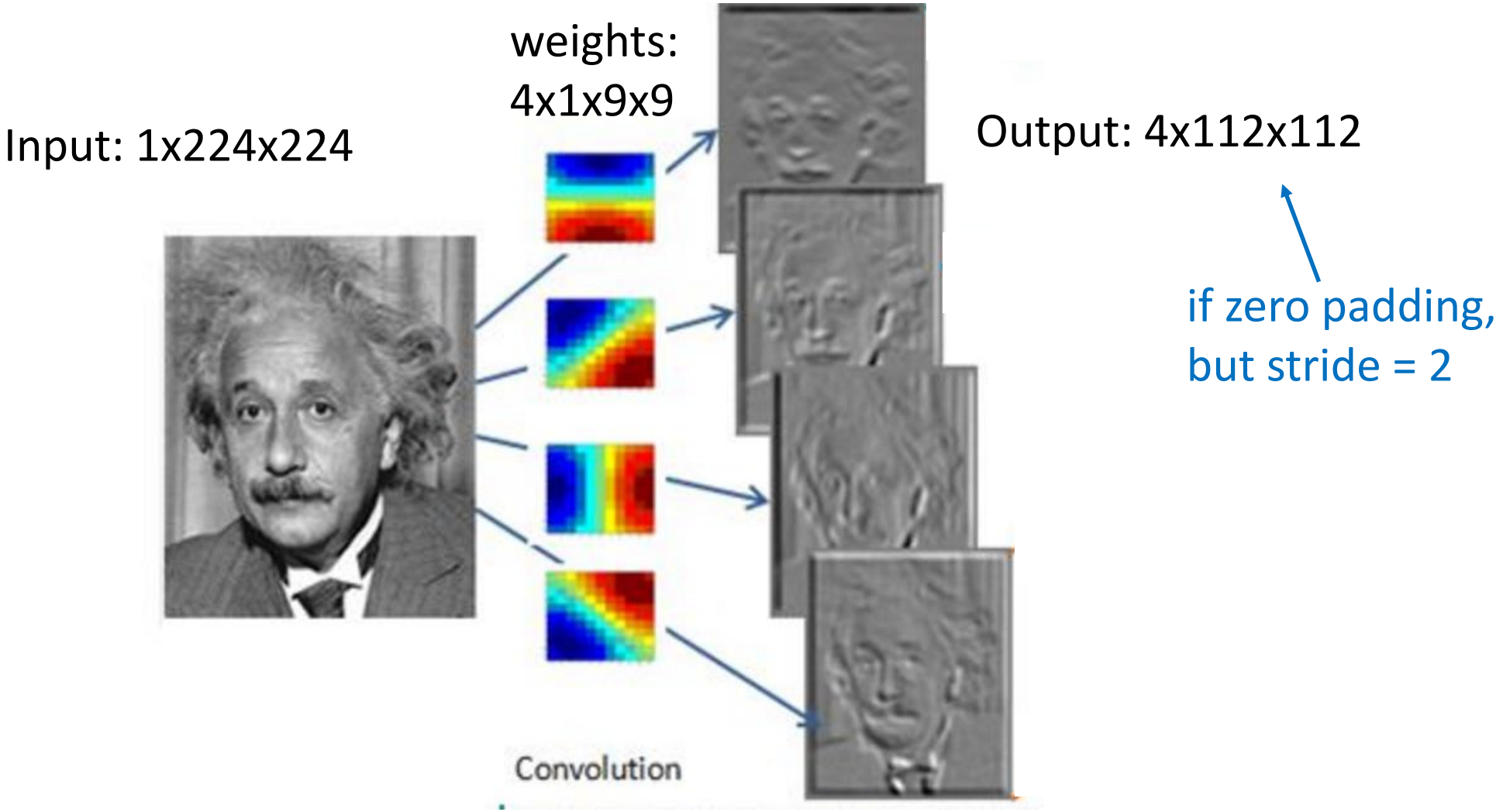
weights:
4x1x9x9



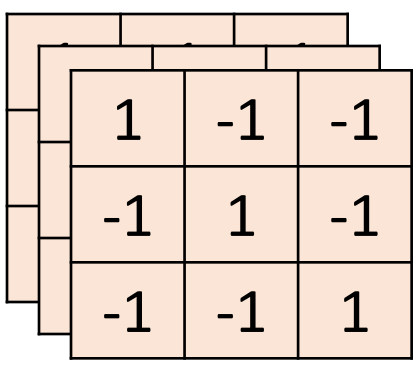
Output: 4x224x224

if zero padding,
and stride = 1

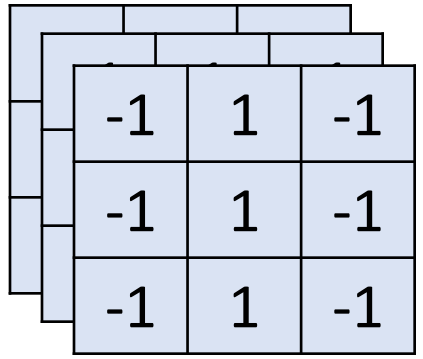
Convolutional Layer (with 4 filters)



Color image: RGB 3 channels – conv. over depth

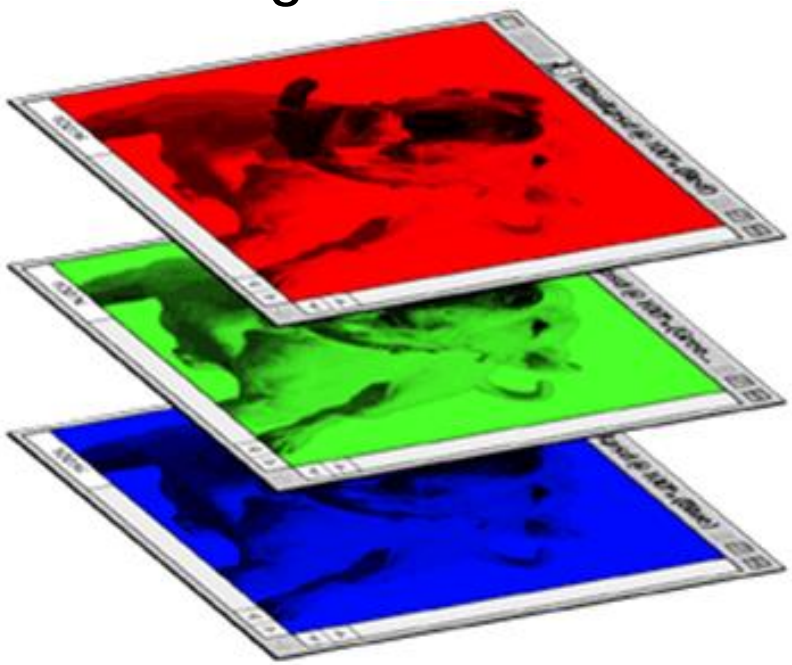


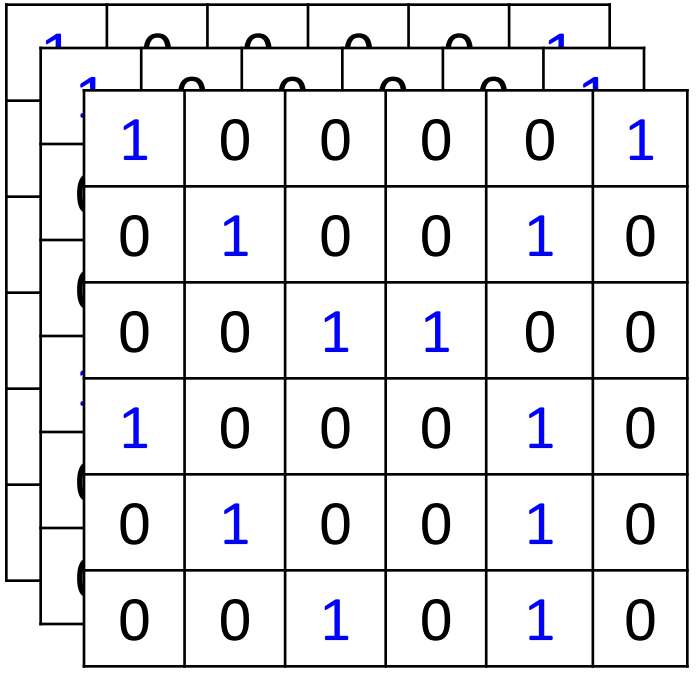
Filter 1



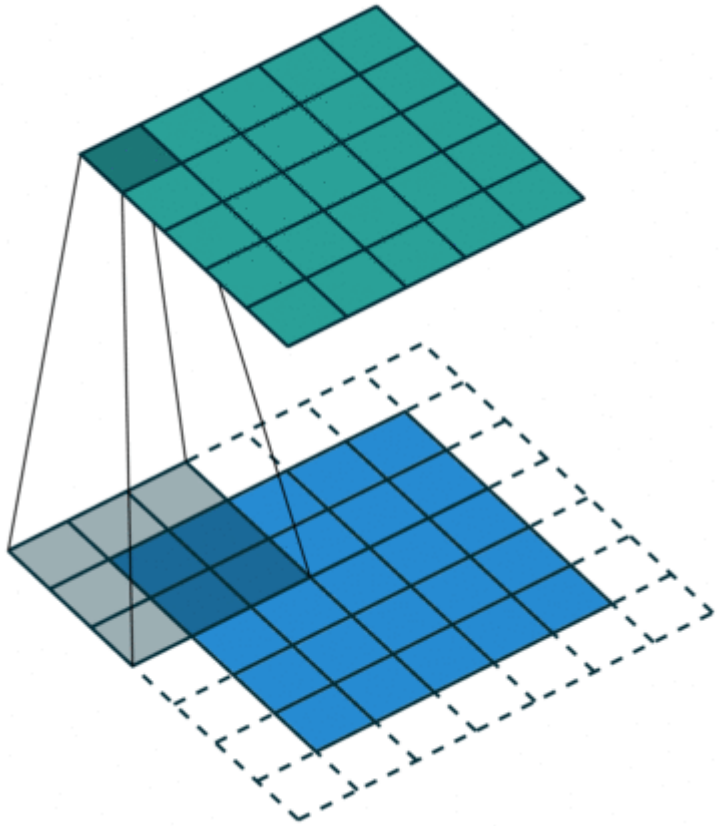
Filter 2 …3

Color image



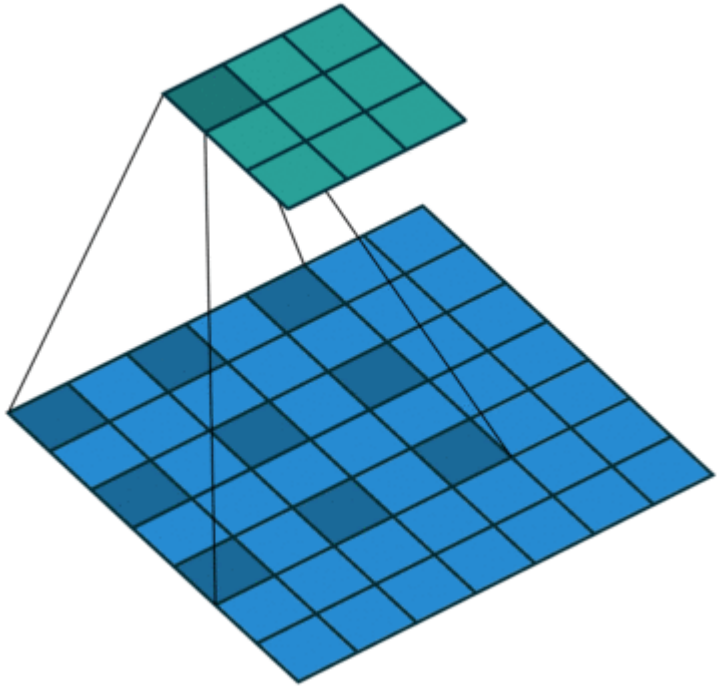


Different types of convolution



Parameters:

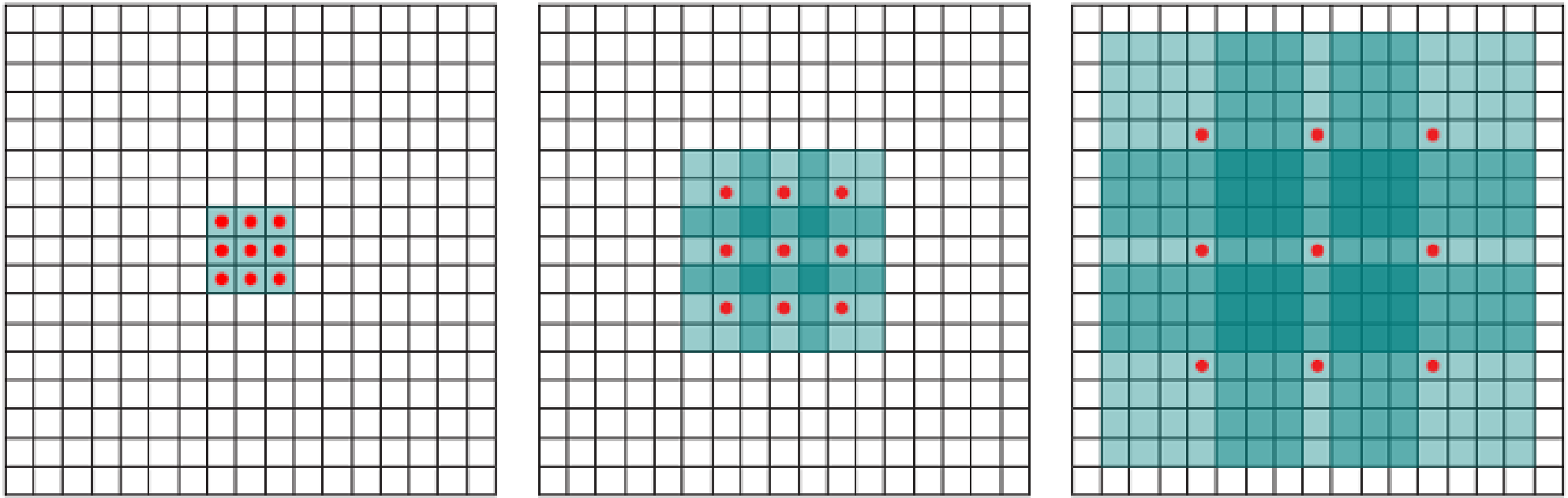
- ✓ Kernel stride
- ✓ Size
- ✓ Padding



Normal vs dialated

Dialation width = 2

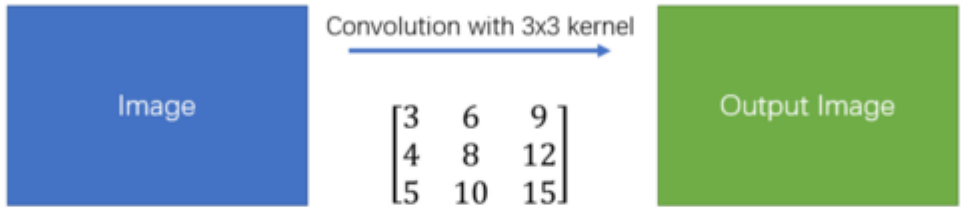
Dialated convolution



Spatially Separable convolution

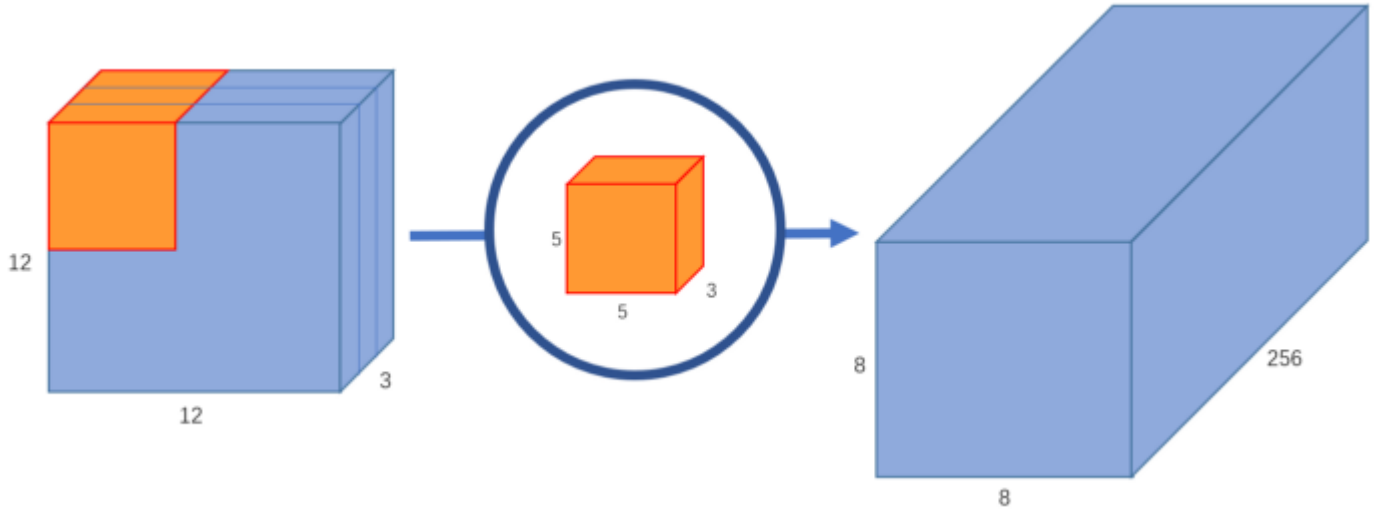
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Simple Convolution



Spatial Separable Convolution

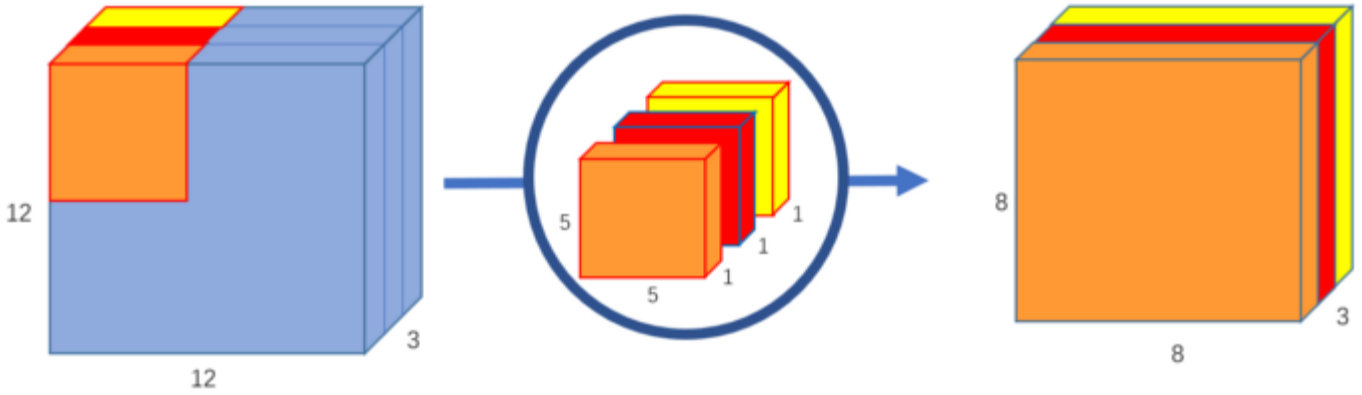
Depthwise separable convolution



Convolving by 256 5x5 kernels over the input volume

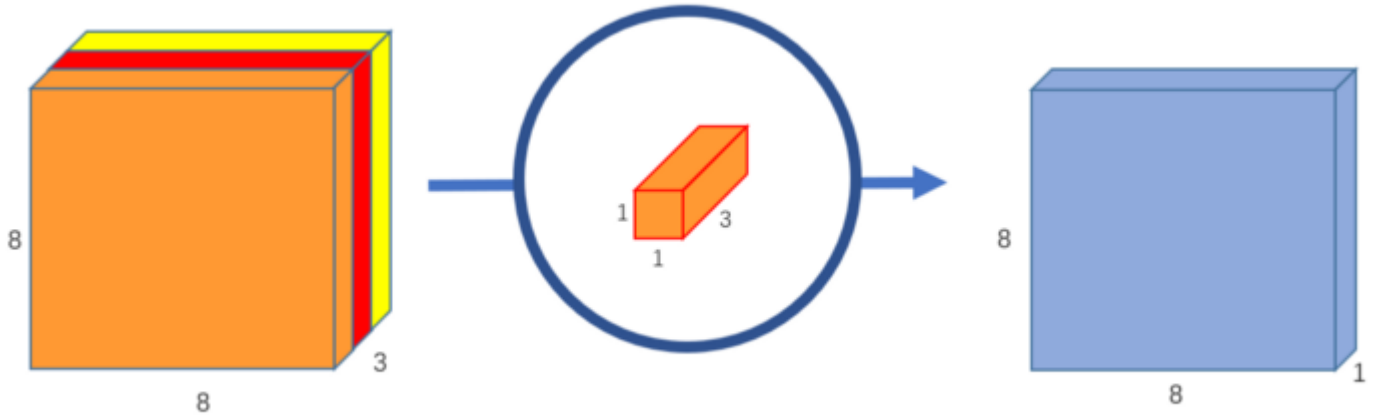
Depthwise separable convolution – step1

Along depth

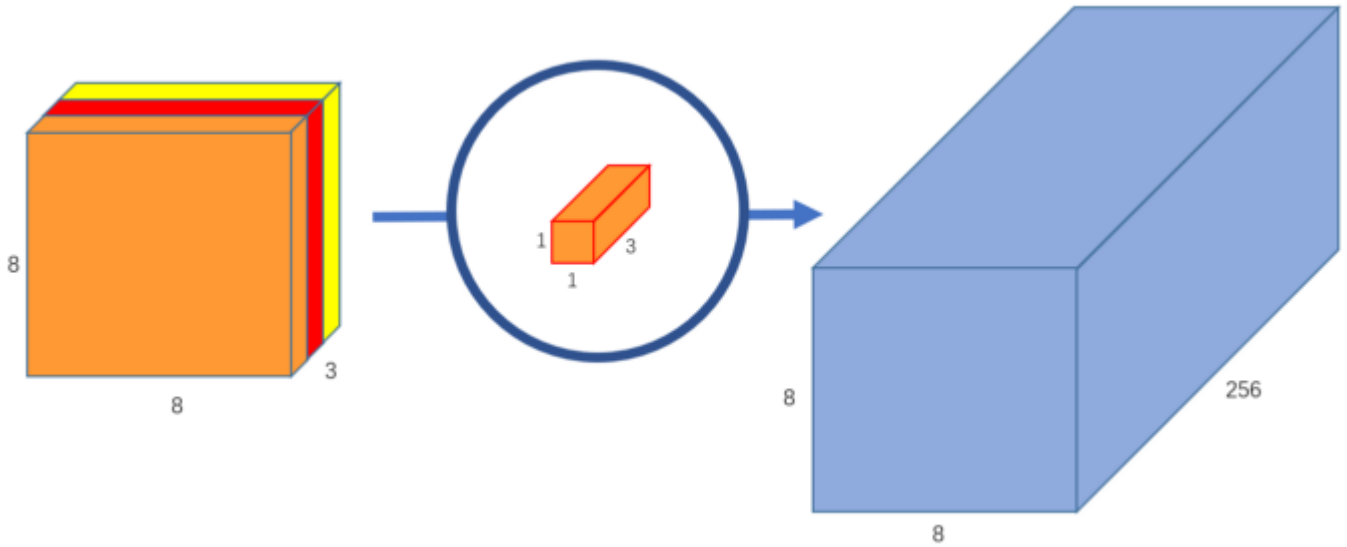


Each 5x5x1 kernel iterates 1 channel of the image (note: **1 channel**, not all channels), getting the scalar products of every 25 pixel group, giving out a 8x8x1 image. Stacking these images together creates a 8x8x3 image.

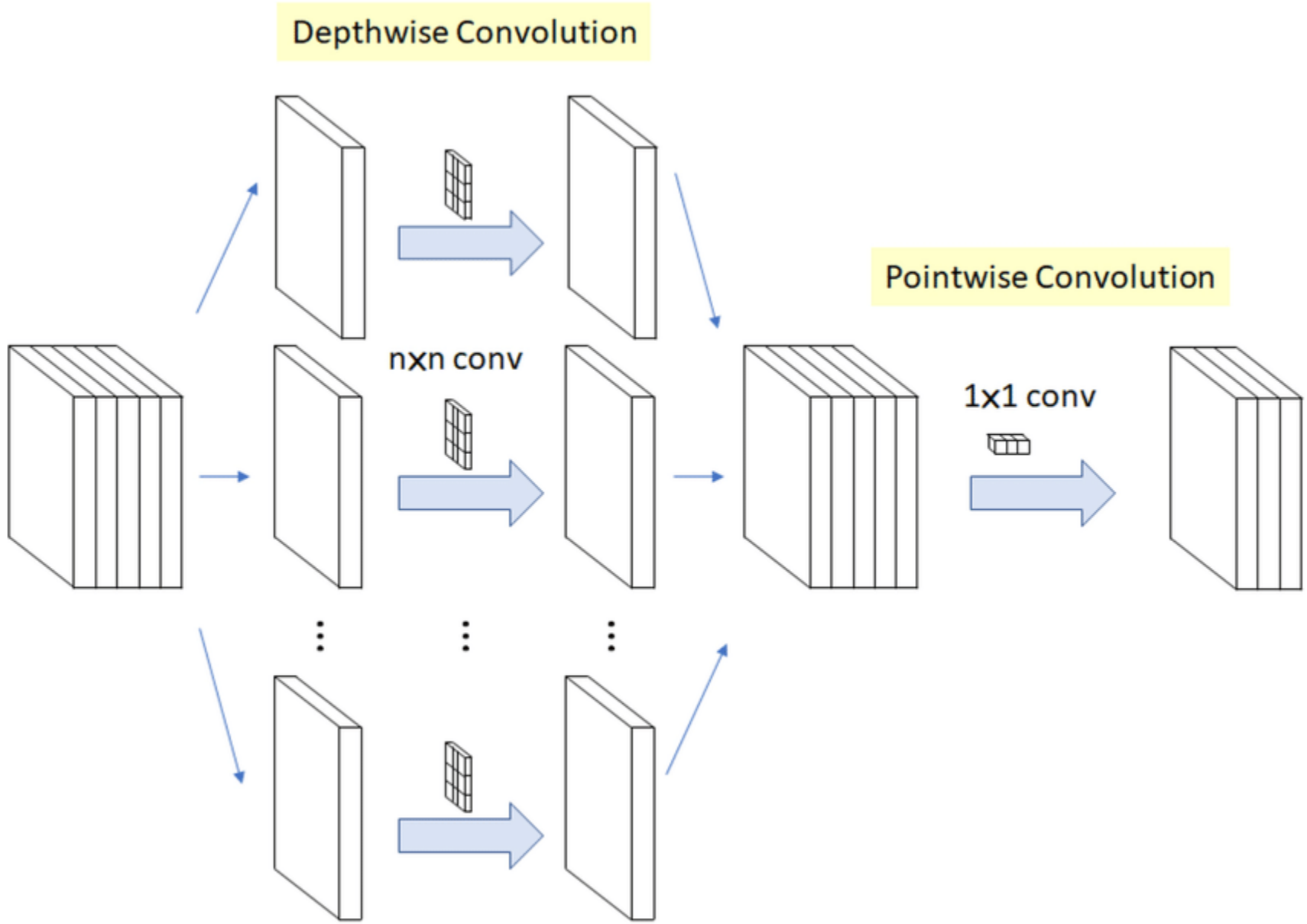
Depthwise separable convolution – step2



Pointwise



1x1 conv


Depthwise Convolution
nxn conv
Pointwise Convolution
1x1 conv

```
model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(8, 7, input_shape=(28,28,1), strides=2),
        tf.keras.layers.Conv2D(16, 5, strides=2),
        tf.keras.layers.Conv2D(32, 3),
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(loss='categorical_crossentropy', metrics=['acc'])
model.summary()
```

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_43 (Conv2D)           (None, 11, 11, 8)         400

 conv2d_44 (Conv2D)           (None, 4, 4, 16)          3216

 conv2d_45 (Conv2D)           (None, 2, 2, 32)          4640

 global_average_pooling2d_17  (None, 32)                0
  (GlobalAveragePooling2D)

 dense_17 (Dense)             (None, 10)                330

=================================================================
Total params: 8,586
Trainable params: 8,586
Non-trainable params: 0
_________________________________________________________________
```

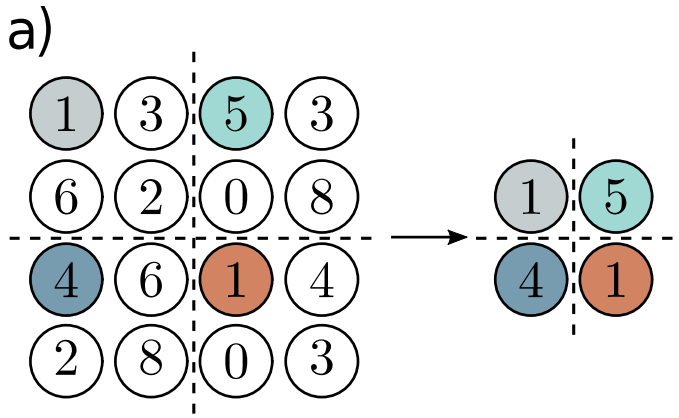
```
Test Accuracy = 0.9168999791145325
Time taken for evaluation of 10000 images = 1.43 seconds
```

```
model = tf.keras.Sequential([
        tf.keras.layers.SeparableConv2D(8, 7, input_shape=(28,28,1), strides=2),
        tf.keras.layers.SeparableConv2D(16, 5, strides=2),
        tf.keras.layers.SeparableConv2D(32, 3),
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(loss='categorical_crossentropy', metrics=['acc'])
model.summary()
```

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 separable_conv2d_9 (Separab  (None, 11, 11, 8)         65
 leConv2D)

 separable_conv2d_10 (Separa  (None, 4, 4, 16)          344
 bleConv2D)

 separable_conv2d_11 (Separa  (None, 2, 2, 32)          688
 bleConv2D)

 global_average_pooling2d_18  (None, 32)                0
  (GlobalAveragePooling2D)

 dense_18 (Dense)             (None, 10)                330

=================================================================
Total params: 1,427
Trainable params: 1,427
Non-trainable params: 0
_________________________________________________________________
```

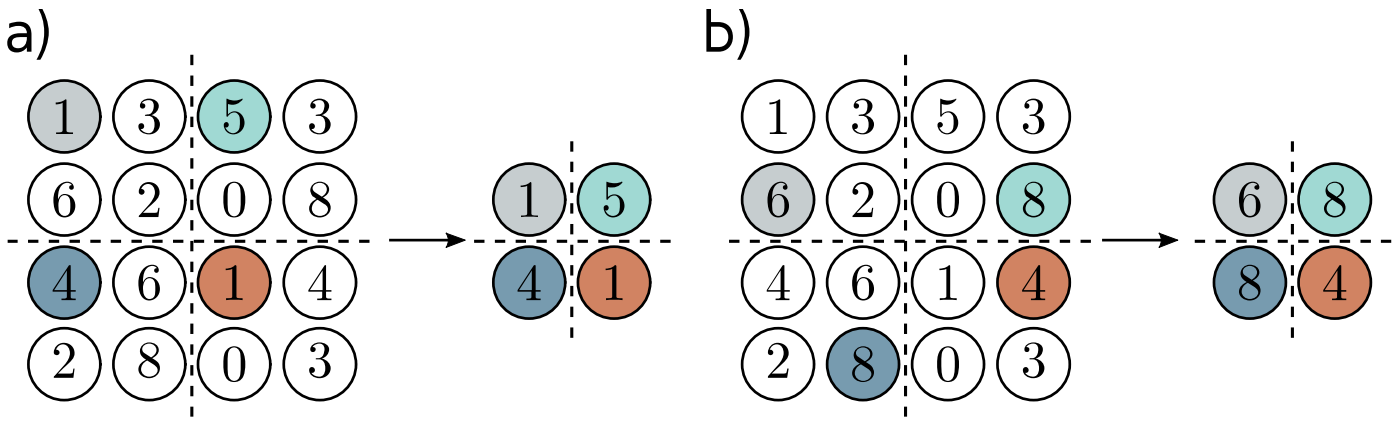
```
Test Accuracy = 0.9021000266075134
Time taken for evaluation of 10000 images = 1.11 seconds
```

Downsampling



Sample every other position (equivalent to stride two)

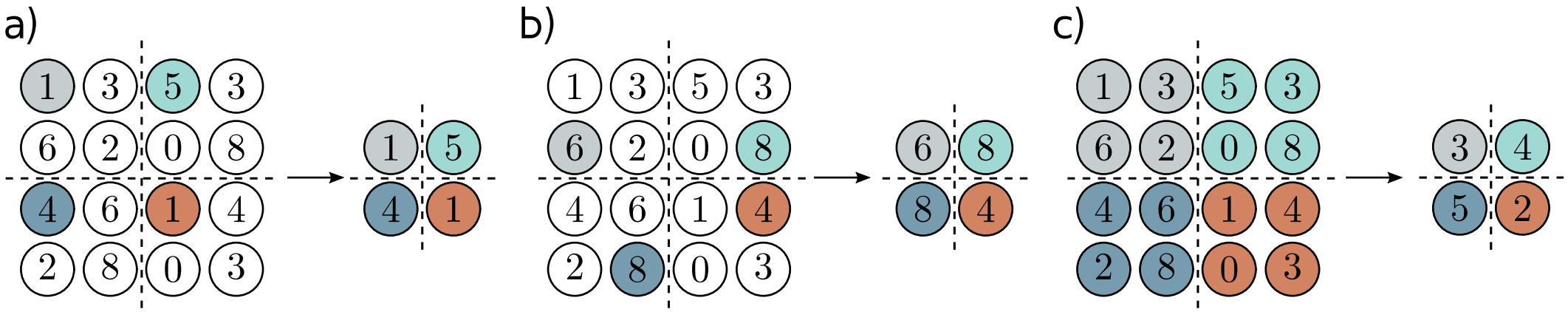
Downsampling



Sample every other position (equivalent to stride two)

Max pooling (partial invariance to translation)

Downsampling



Sample every other position (equivalent to stride two)

Max pooling (partial invariance to translation)

Mean pooling

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?

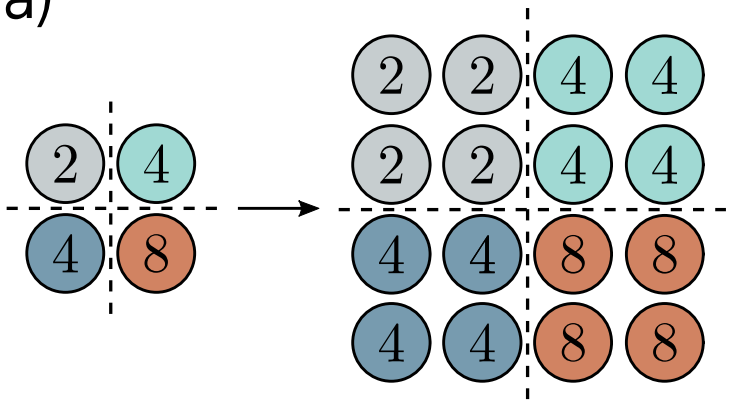
STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET

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"

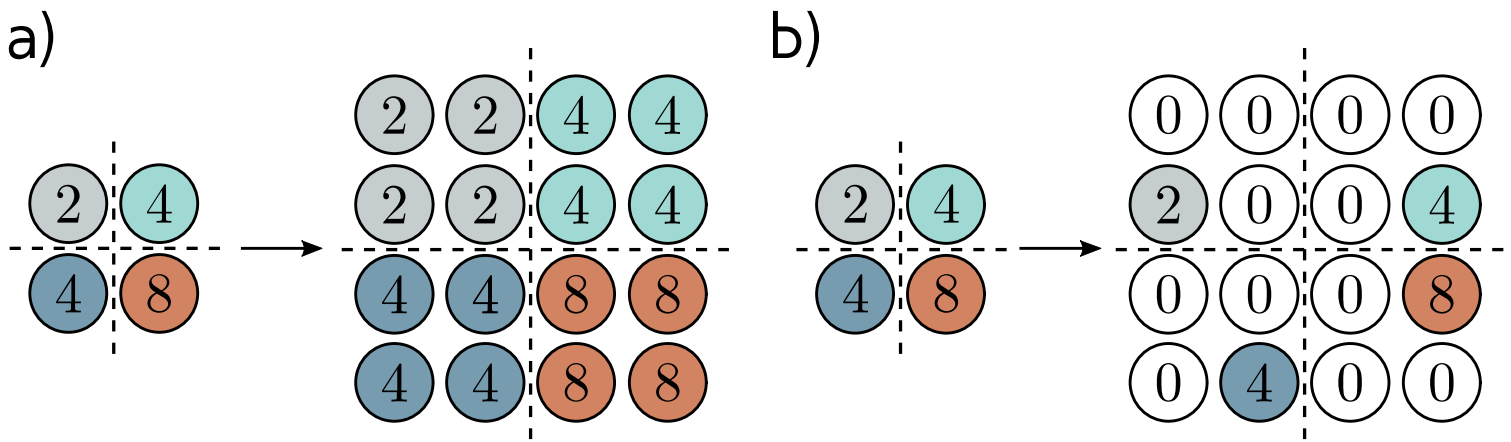
Upsampling

a)



Duplicate

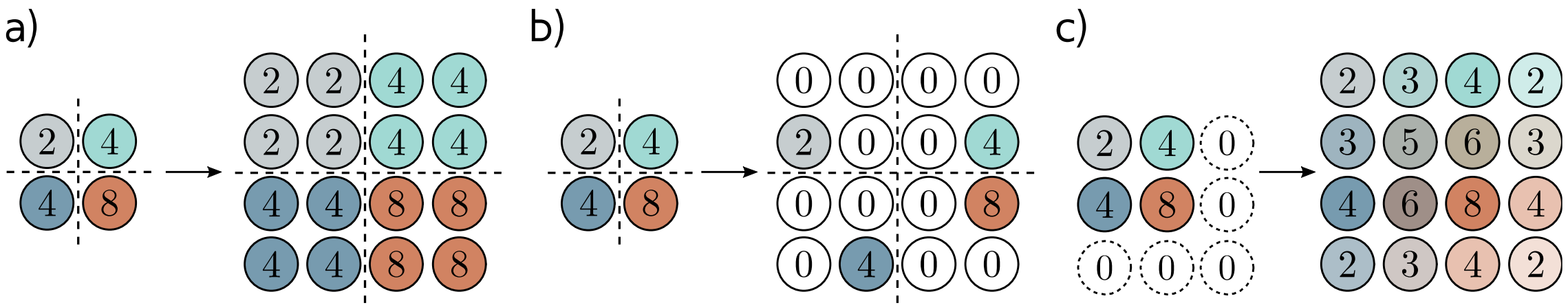
Upsampling



Duplicate

Max-upsampling

Upsampling



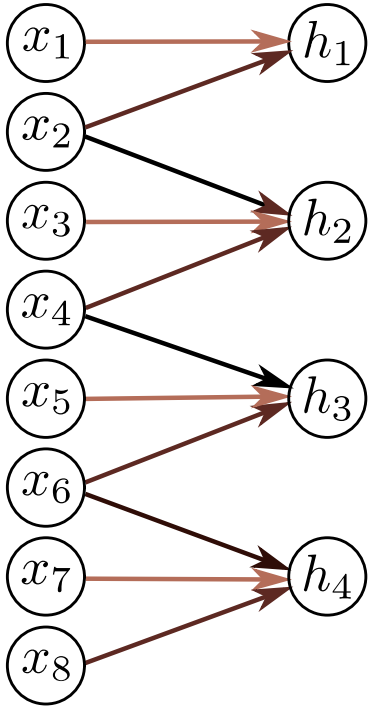
Duplicate

Max-upsampling

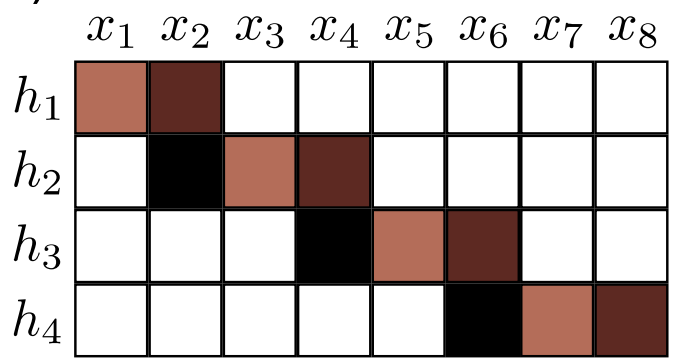
Bilinear interpolation

Transposed convolutions

a)

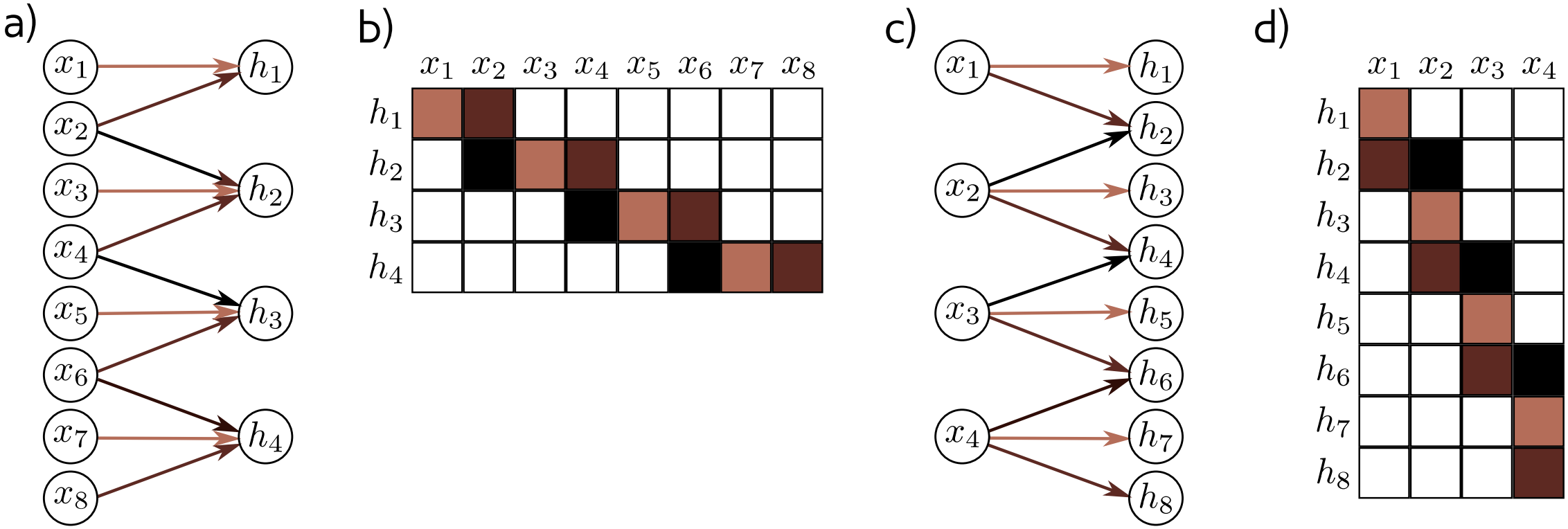


b)



Kernel size 3, Stride 2 convolution

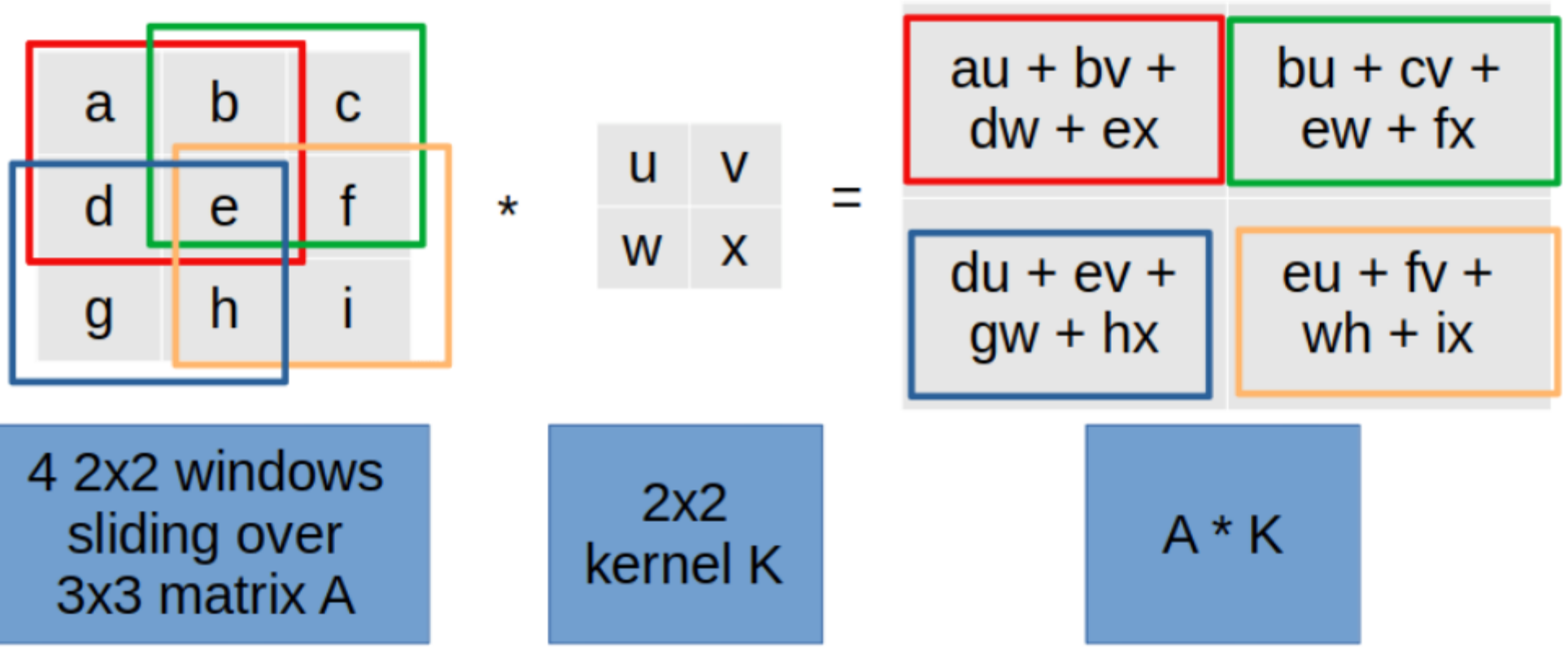
Transposed convolutions



Kernel size 3, Stride 2 convolution

Transposed convolution

Convolution as matrix vector multiplication



Let A be a matrix of size $n \times n$ and K a $k \times k$ kernel. For convenience, let $t = n - k + 1$.

To calculate the convolution $A * K$, we need to calculate the matrix-vector multiplication $Mv^T = v'^T$ where:

- M is a $t^2 \times n^2$ block matrix we get from the kernel K
- v is a row vector with the n^2 elements of A concatenated row by row
- v' is a row vector of t^2 elements which can be reshaped into a $t \times t$ matrix by splitting the row vector into t rows of t elements

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

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

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

$$K_1 = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 \\ 0 & k_{1,1} & k_{1,2} \end{bmatrix}, K_2 = \begin{bmatrix} k_{2,1} & k_{2,2} & 0 \\ 0 & k_{2,1} & k_{2,2} \end{bmatrix},$$

and get $\hat{K}$ after concatenation:

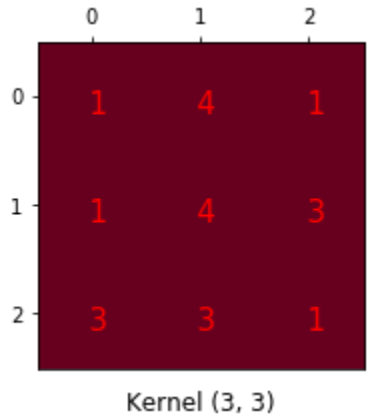
$$\hat{K} = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 \\ 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} \end{bmatrix}$$

The final step is to pad the matrix with blocks of zeros:

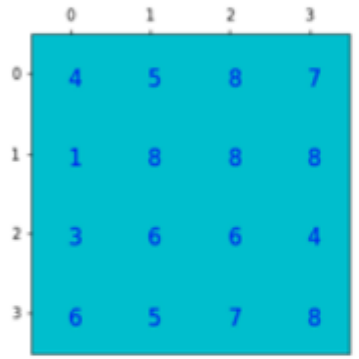
$$M = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} \end{bmatrix}$$

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.

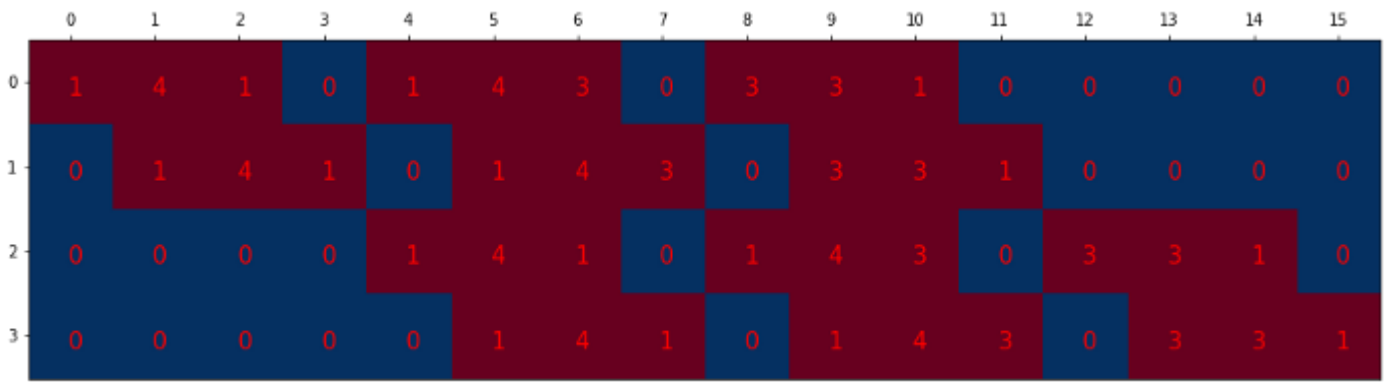
Convolution as a matrix multiplication



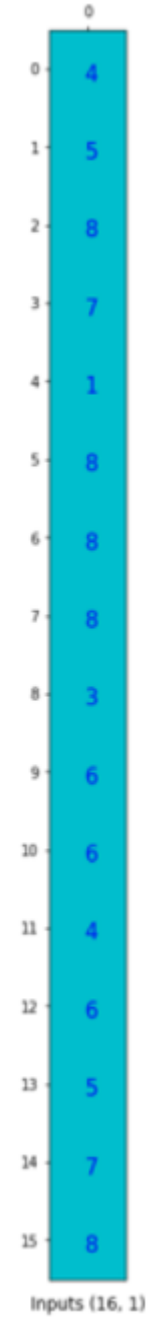
Kernel (3, 3)



Inputs (4, 4)

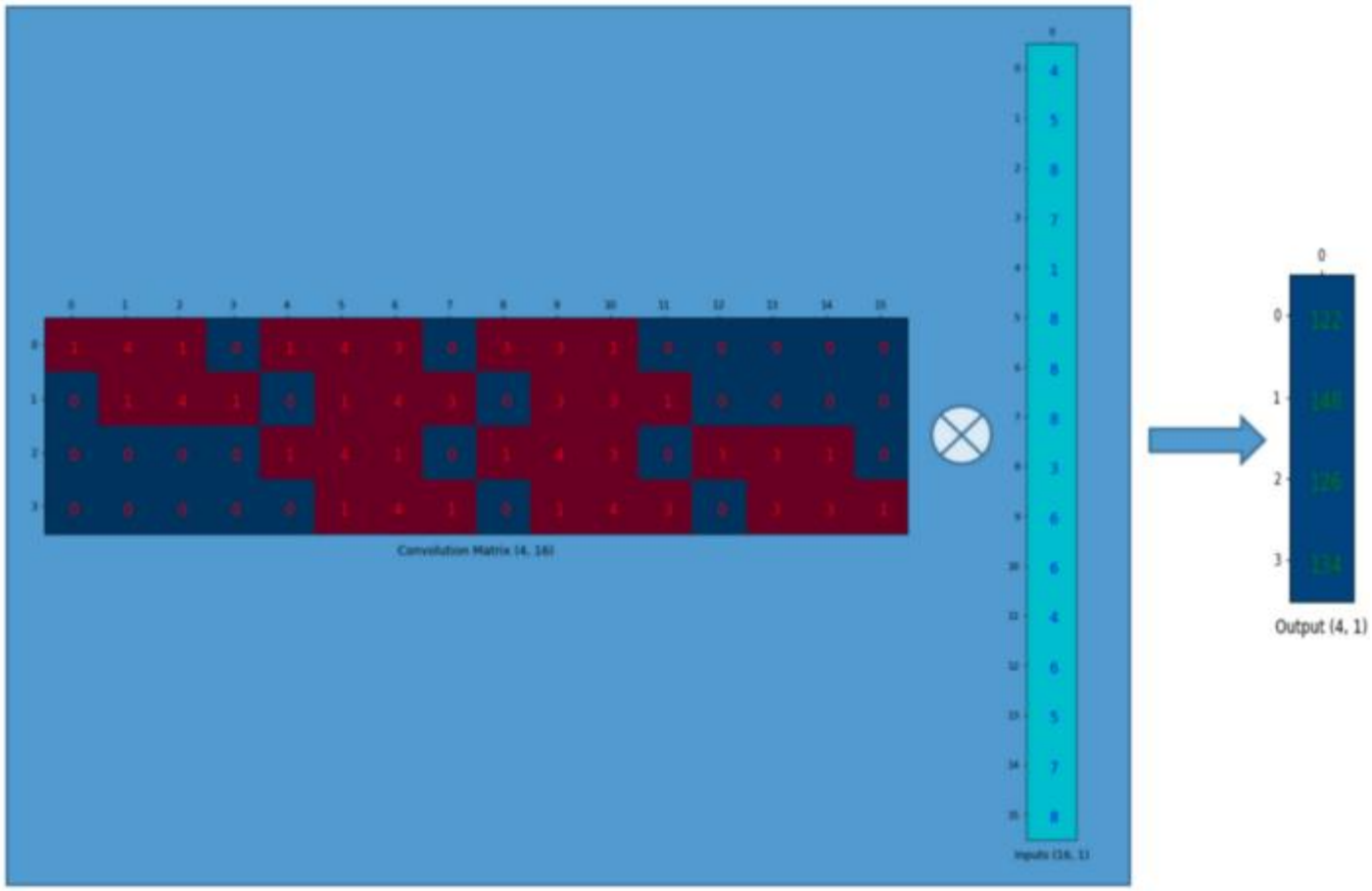


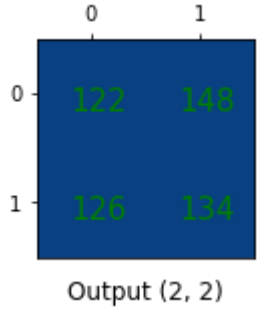
Convolution Matrix (4, 16)

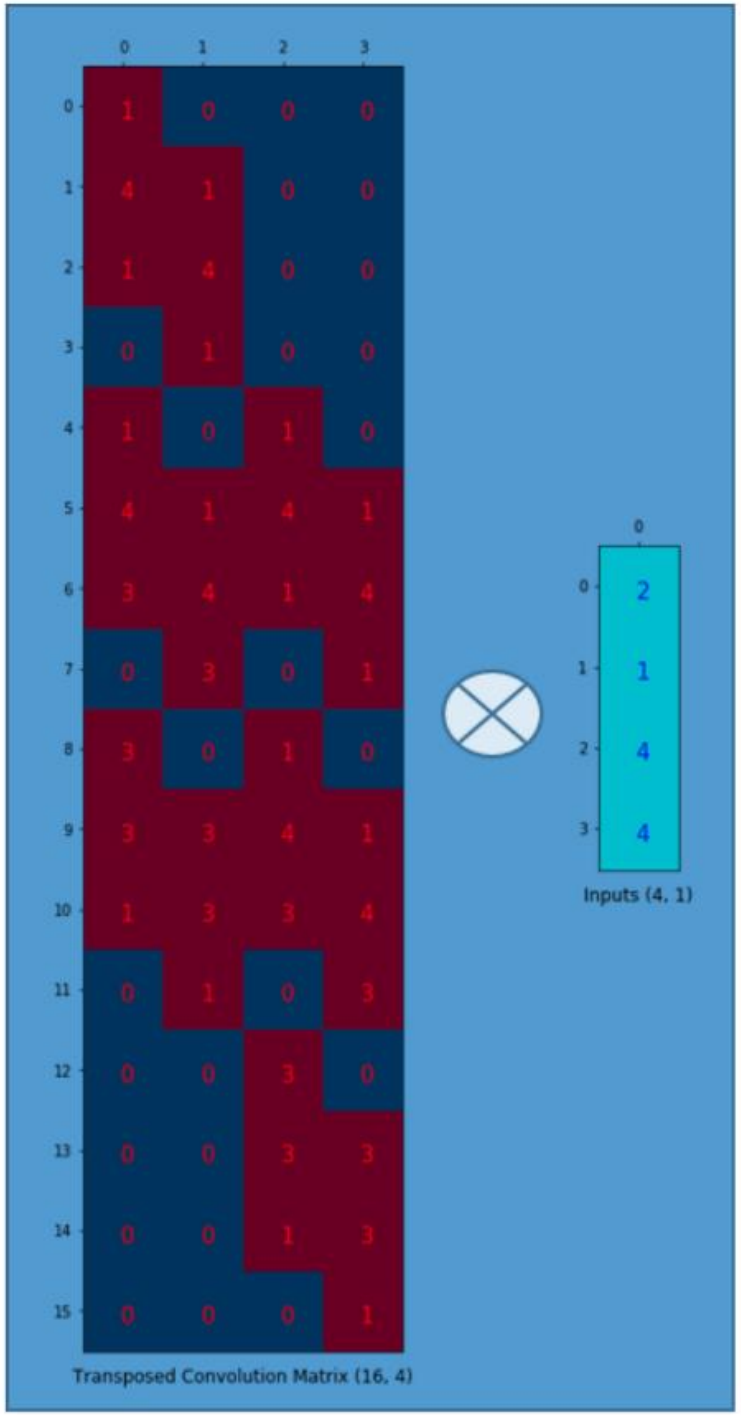


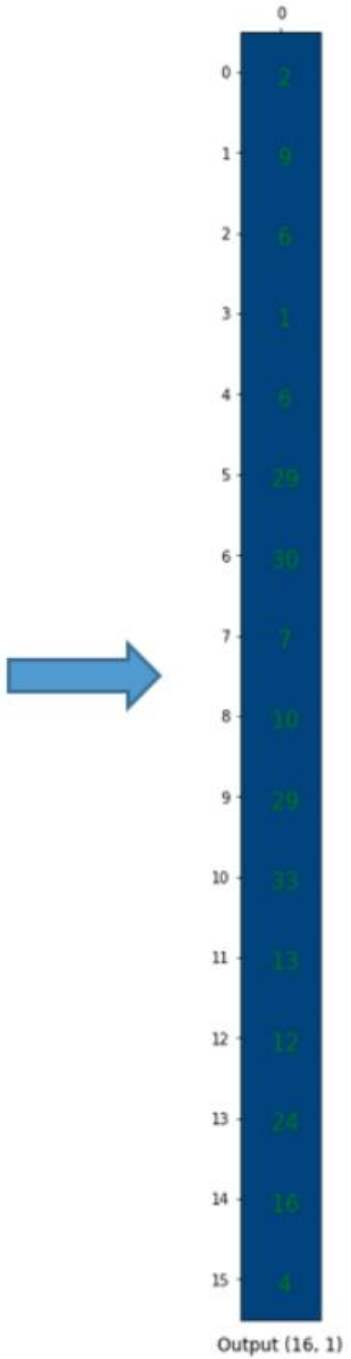
Inputs (16, 1)

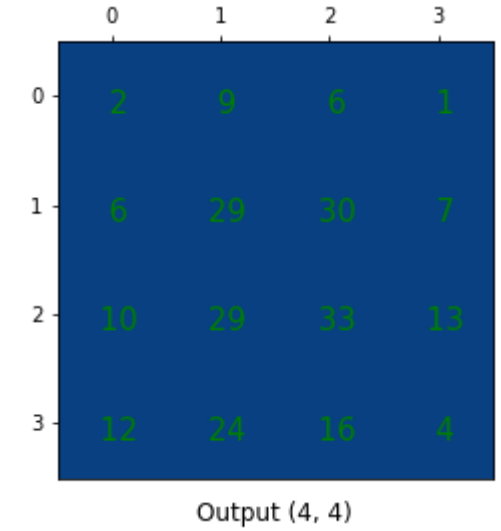
Many to one mapping – 9 values to 1 value







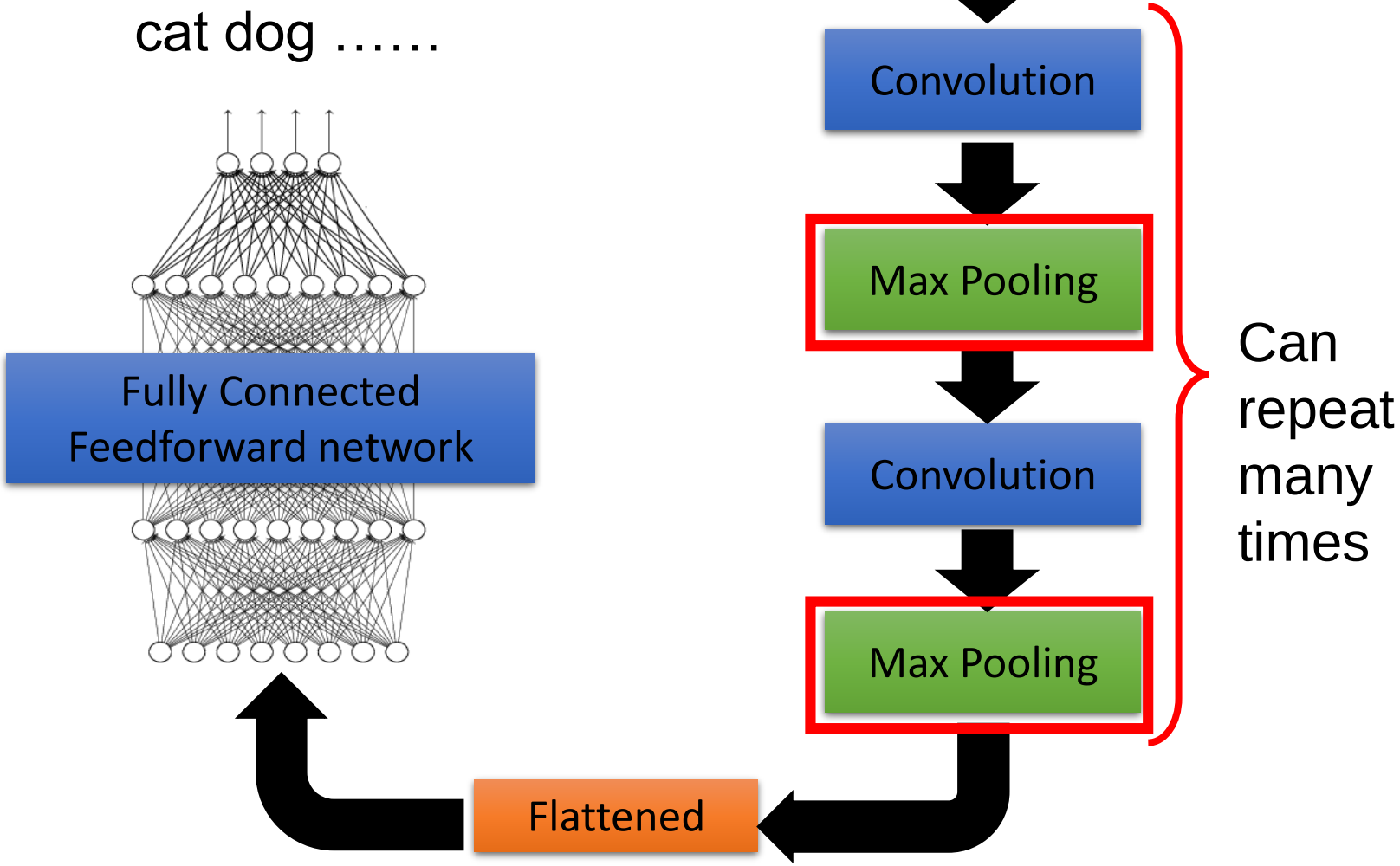




- Fractionally-strided convolution
- Deconvolution

One to many mapping

The whole CNN



The whole CNN

Convolution

Max Pooling

A new image

Convolution

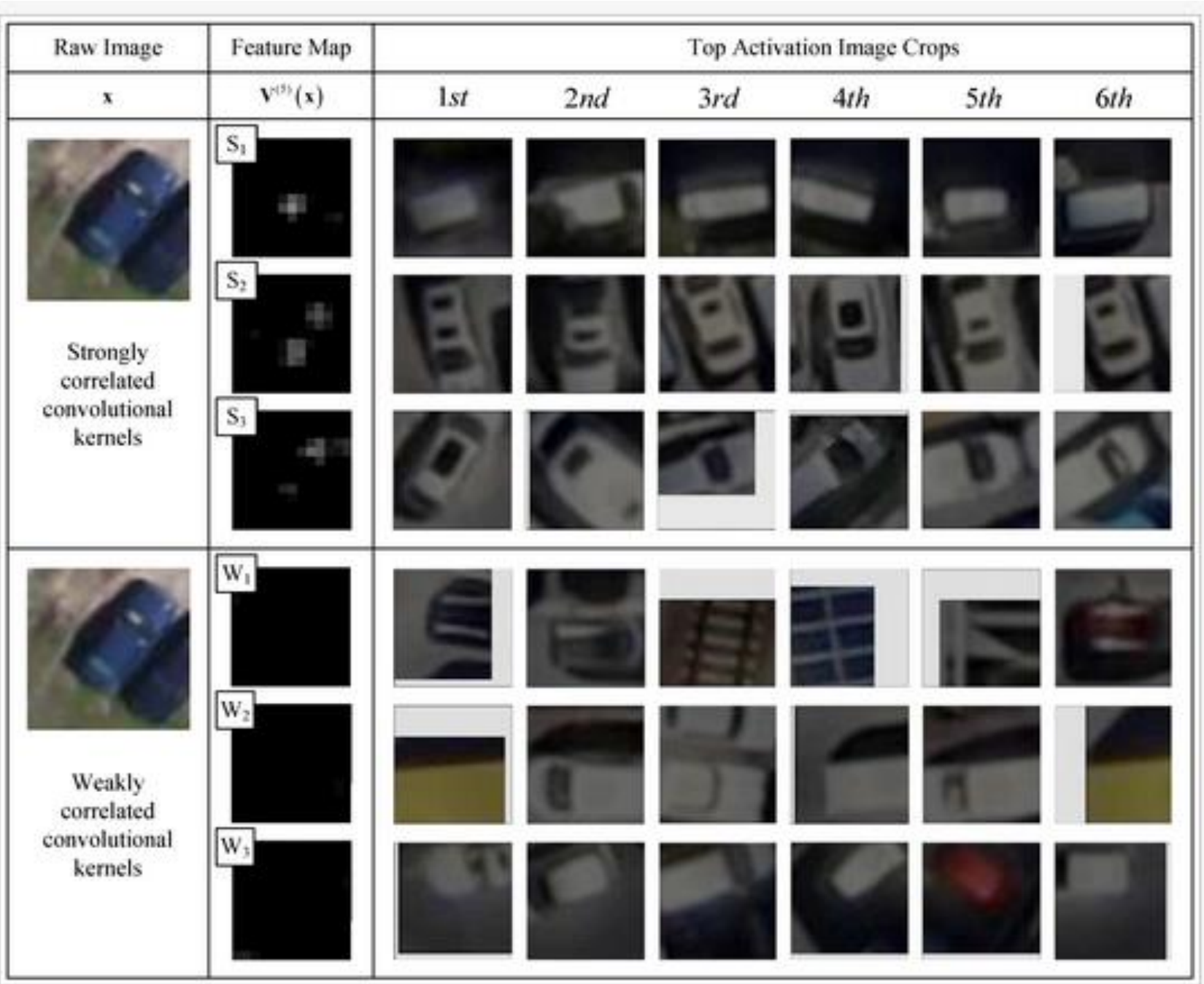
Max Pooling

A new image

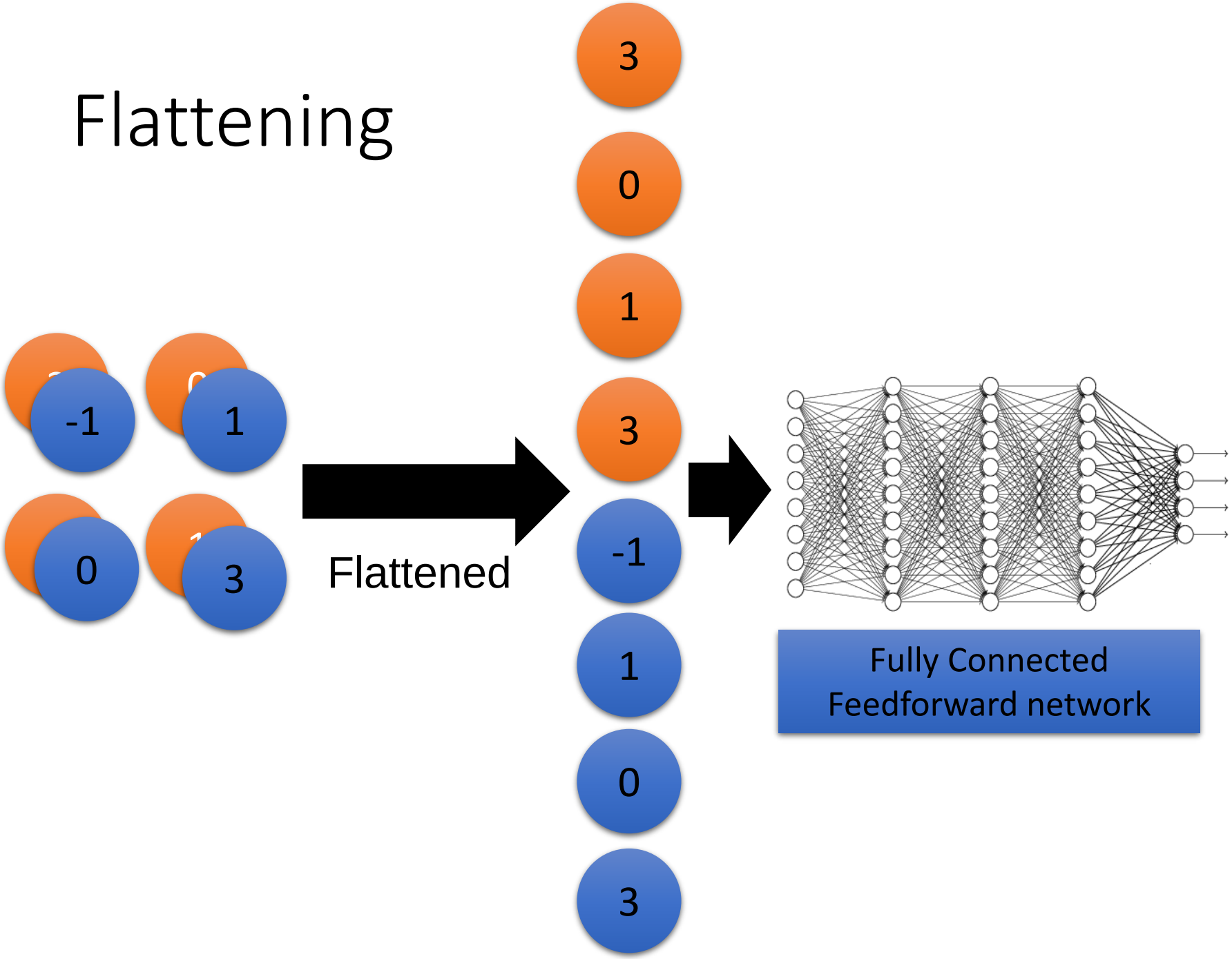
Flattened

Fully Connected Feedforward network

cat dog ……


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

Flattening


-1
1
0
3
Flattened
3
0
1
3
-1
1
0
3
Fully Connected
Feedforward network

Conv Net Topology

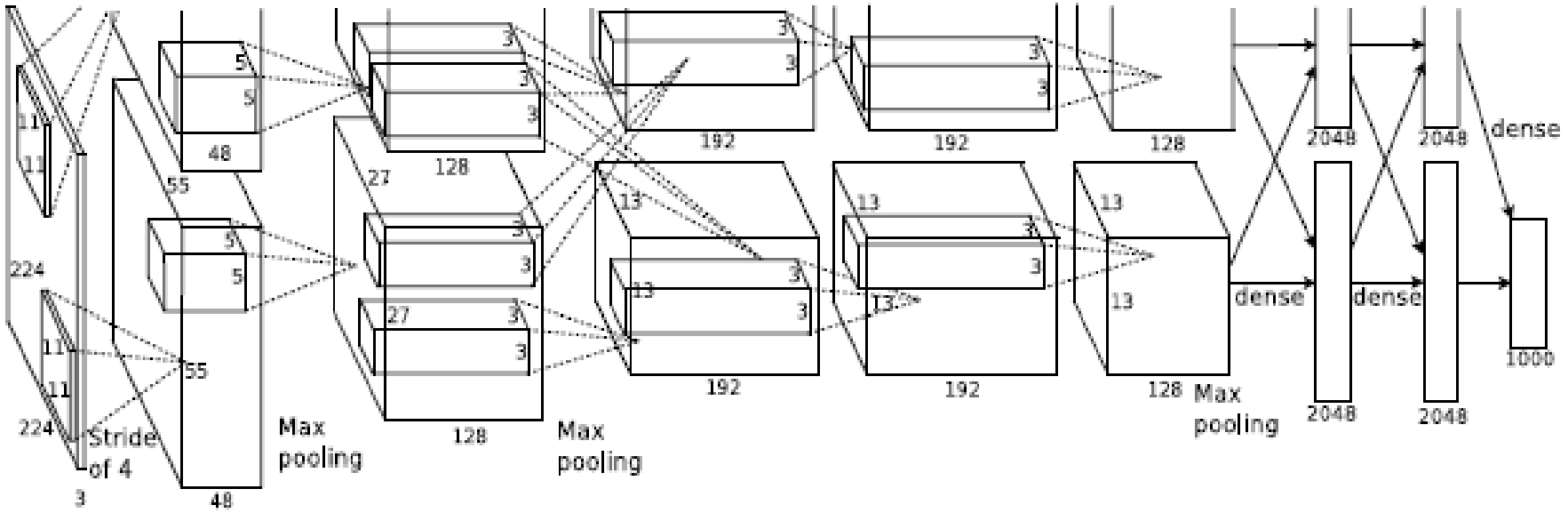
- 5 convolutional layers
- 3 fully connected layers + soft-max
- 650K neurons , 60 Mln weights

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

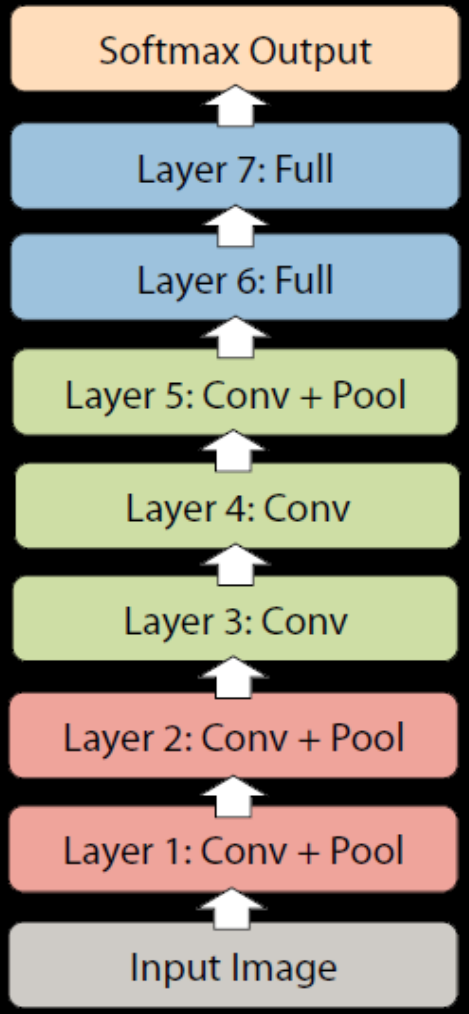
arena football
indoor american football
arena football
canadian football
american football
women's lacrosse

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?

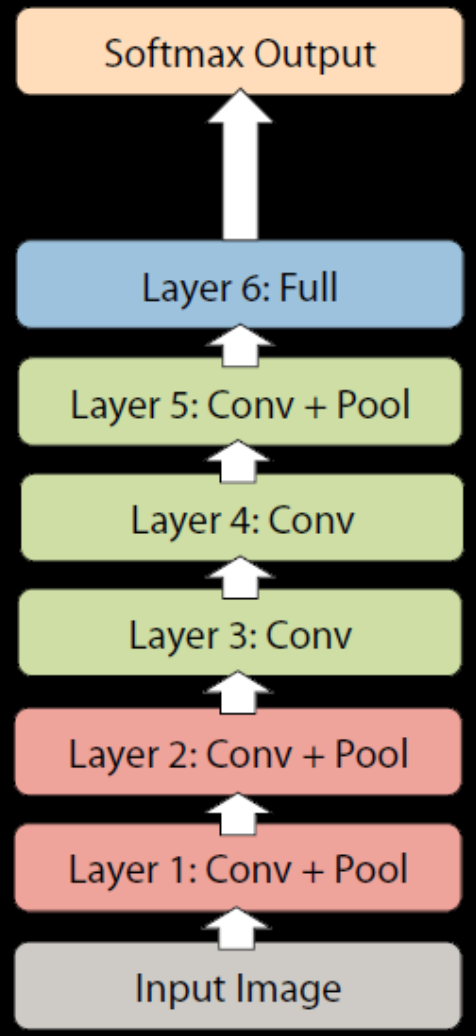
- 8 layers total
- Trained on Imagenet dataset [Deng et al. CVPR'09]
- 18.2% top-5 error
- Our reimplementation:
  18.1% top-5 error



Courtsey: ICRI

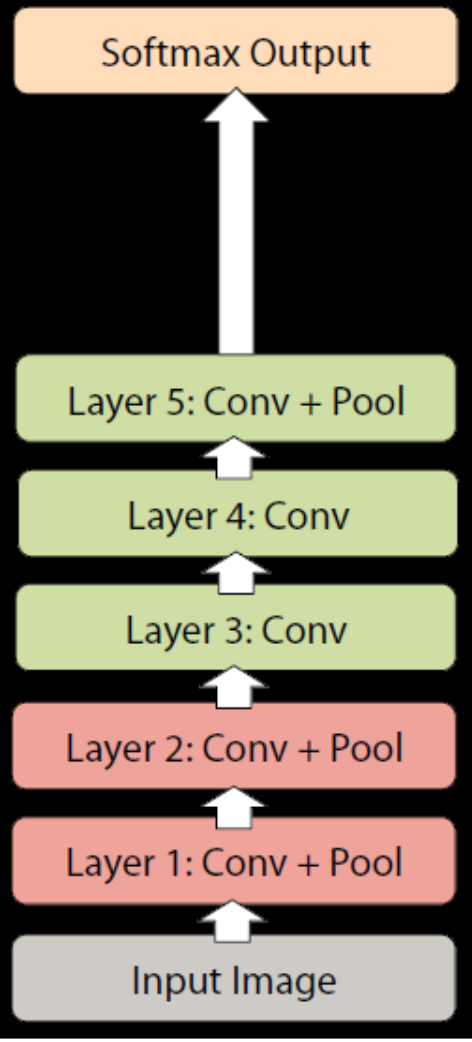
Why do we need a *deep* CNN?

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



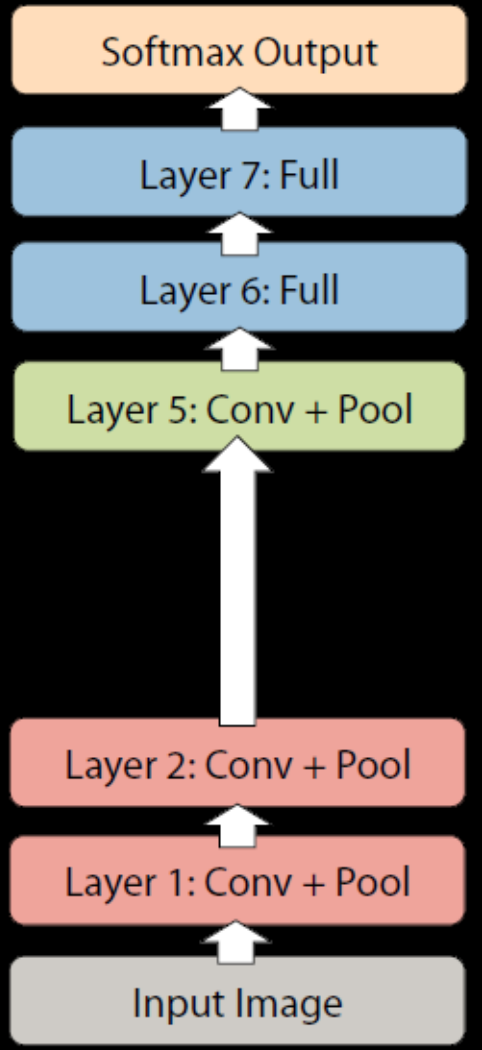
Why do we need a *deep* CNN?

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



Why do we need a *deep* CNN?

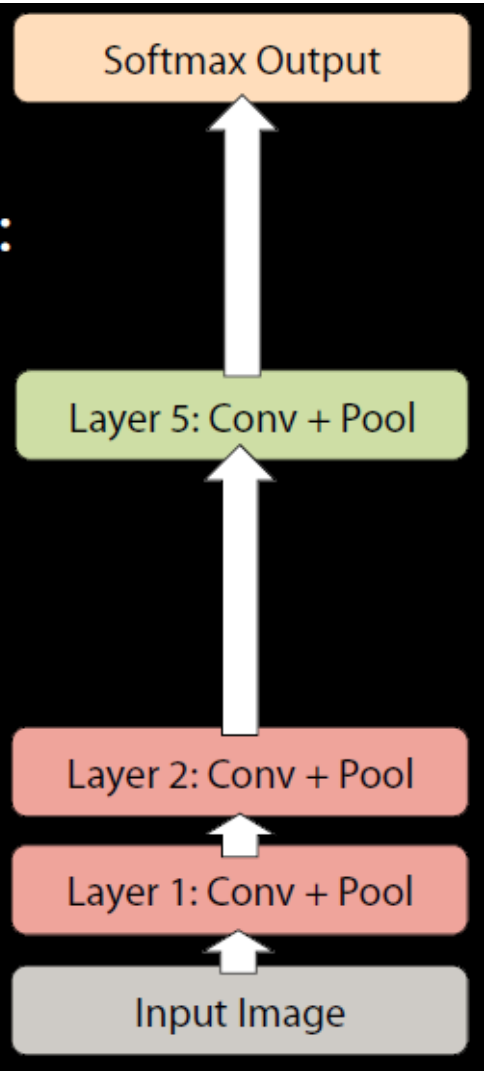
- Now try removing upper feature extractor layers:
  - Layers 3 & 4
- Drop ~1 million parameters
- 3.0% drop in performance



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