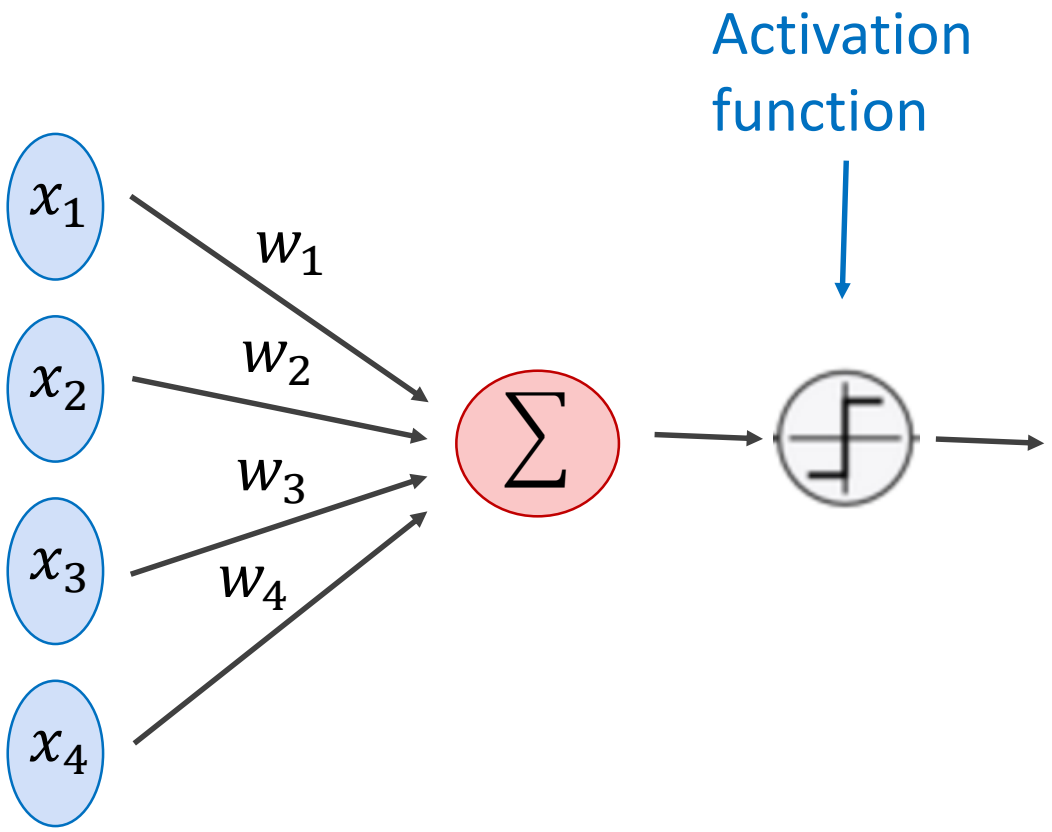
# Perceptron Model

Frank Rosenblatt (1957) - Cornell University

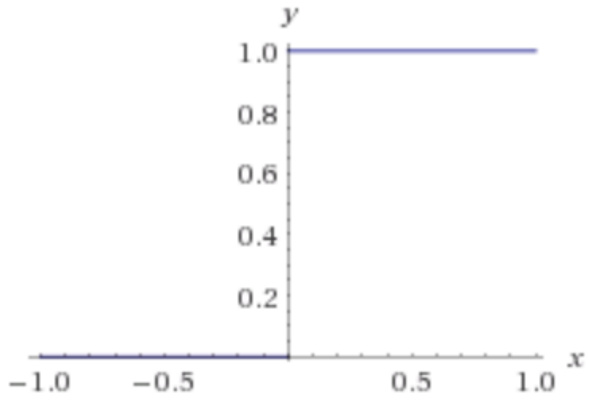
$$f(x) = \begin{cases} 1, & \text{if } \sum_{i=0}^{n} w_i x_i + b > 0 \\ 0, & \text{otherwise} \end{cases}$$



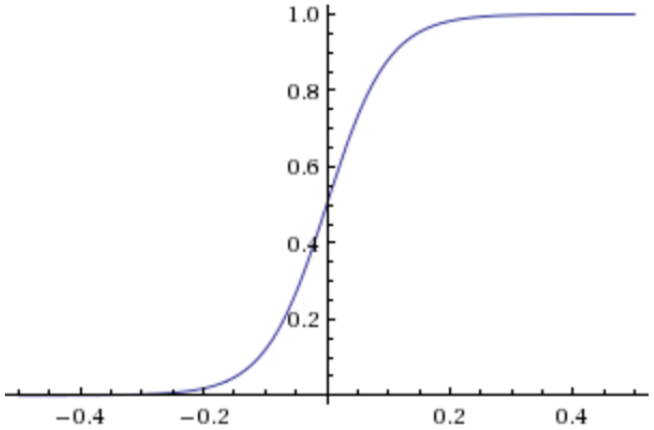

More: https://en.wikipedia.org/wiki/Perceptron

# Activation Functions

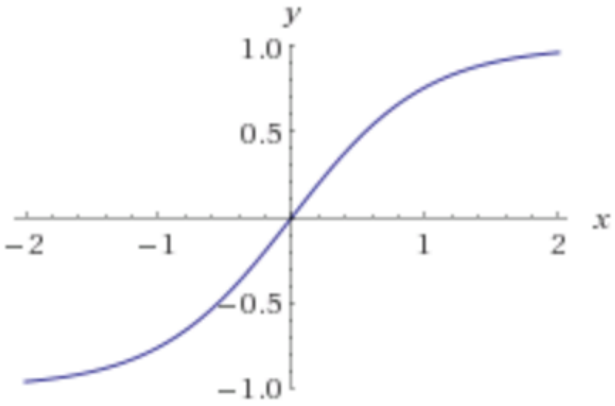
## Step(x)



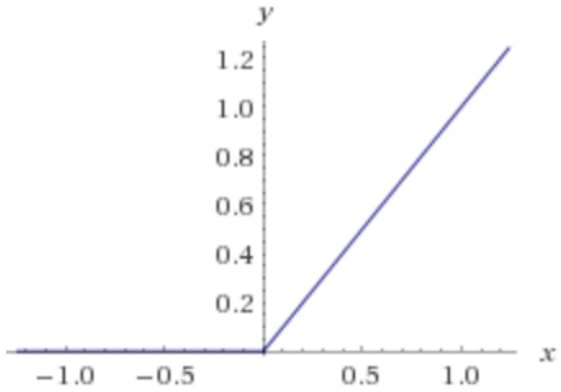

## Sigmoid(x)



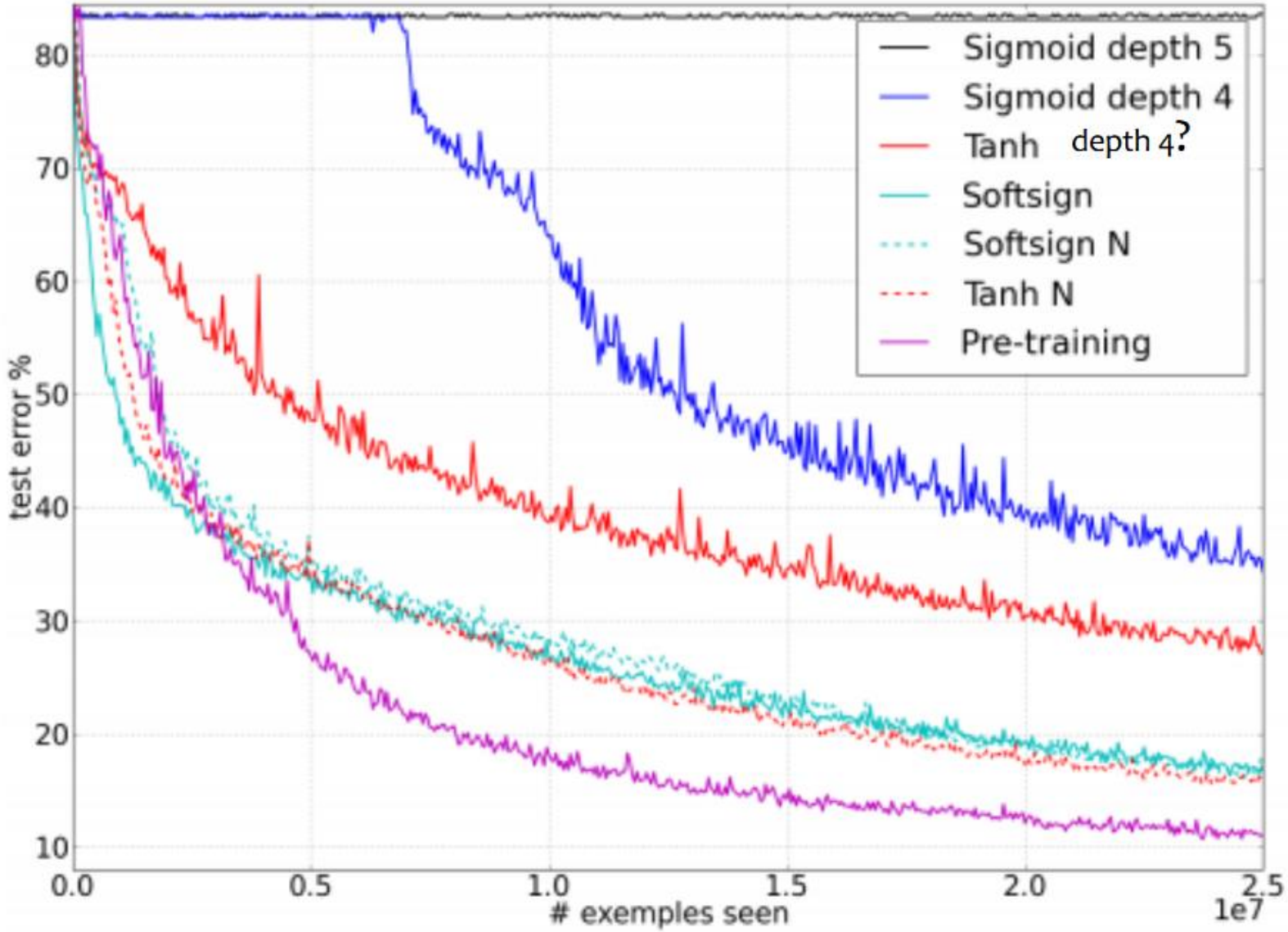

## Tanh(x)




## ReLU(x) = max(0, x)



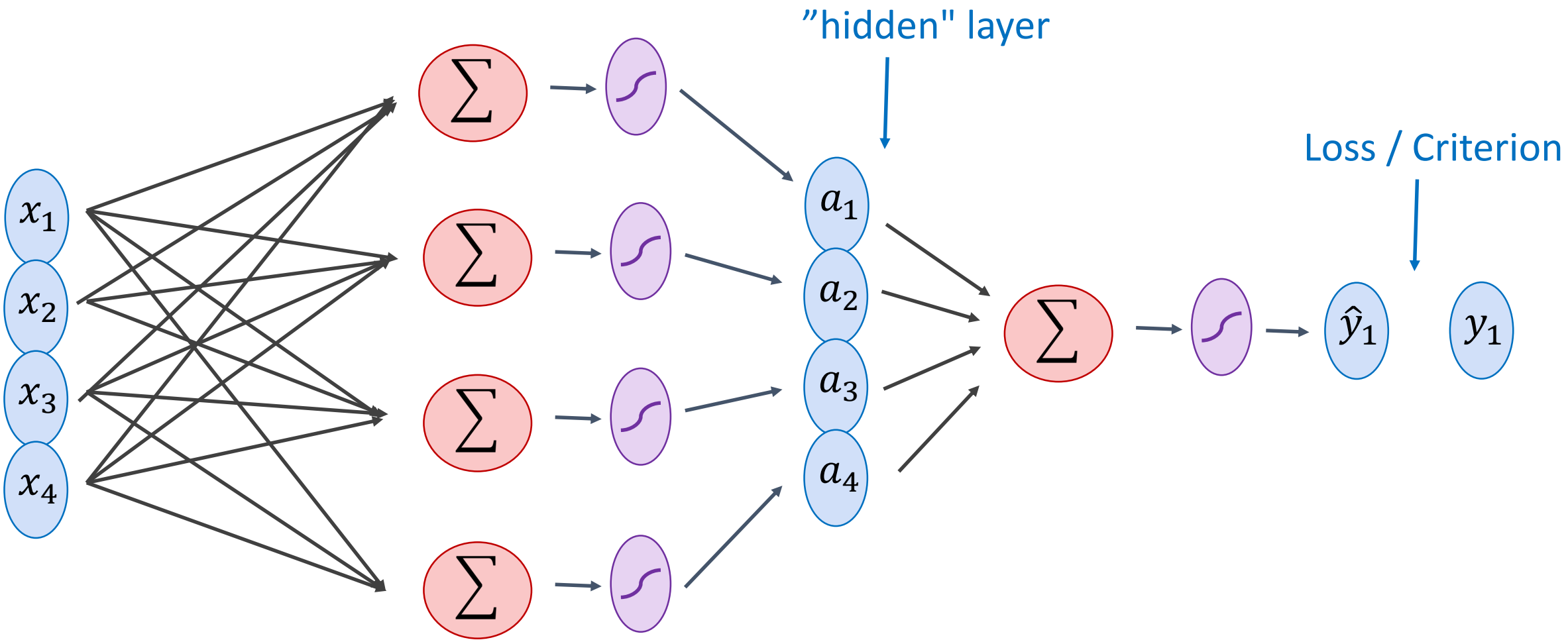
AI Stats 2010




sigmoid
vs.
tanh

Figure from Glorot & Bentio (2010)

# Two-layer Multi-layer Perceptron (MLP)



# Linear Softmax

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

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

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

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

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

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

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

# Linear Softmax

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

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

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

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

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

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

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

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

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

# 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

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

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

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

# 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?

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

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

$$a_2 = sigmoid(w_{[2]}a^T_1 + b^T_{[2]})$$

...

$$a_k = sigmoid(w_{[k]}a^T_{k-1} + b^T_{[i]})$$

...

$$f = softmax(w_{[n]}a^T_{n-1} + b^T_{[n]})$$

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

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

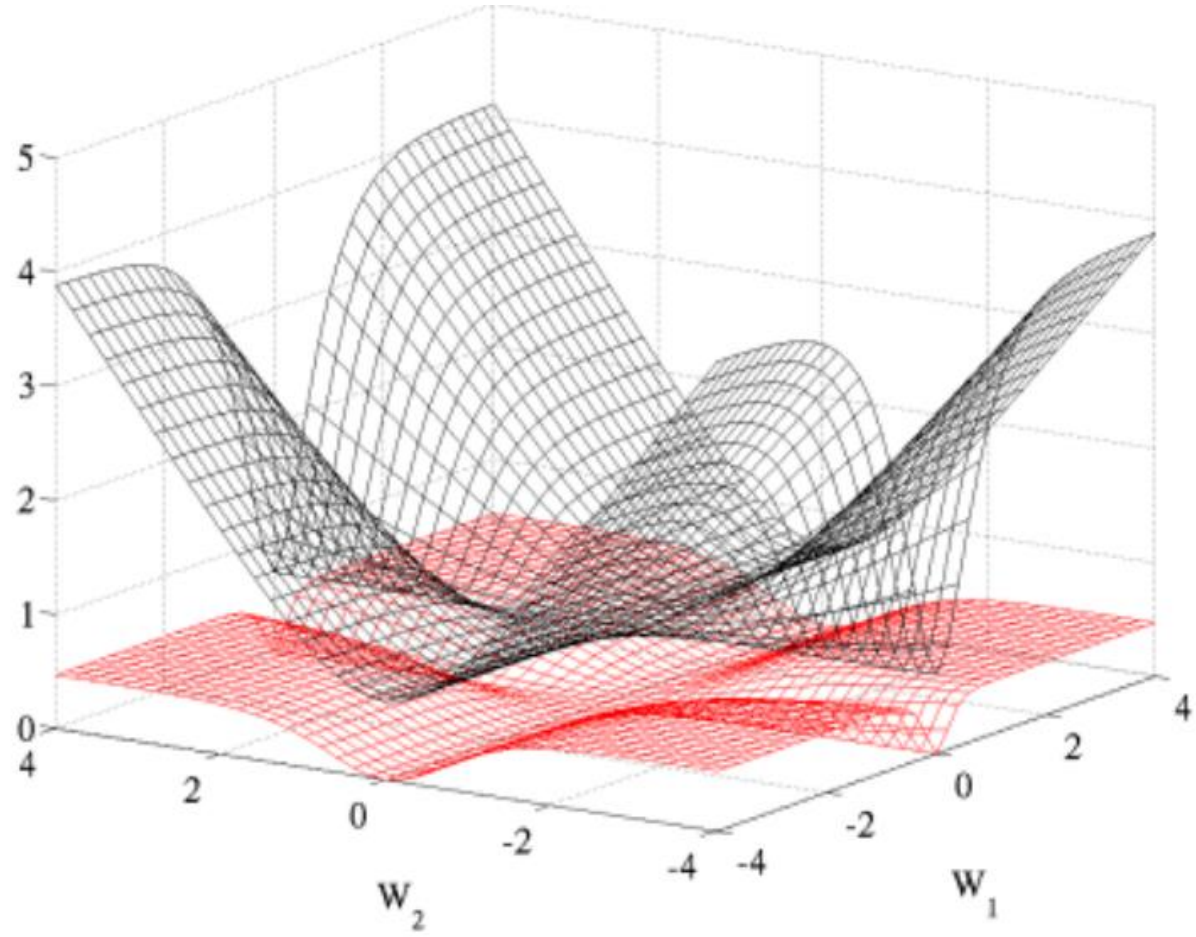



Figure 5: *Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, $W_1$ respectively on the first layer and $W_2$ on the second, output layer.*

Background

# A Recipe for Machine Learning

1. Given training data:

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

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**)

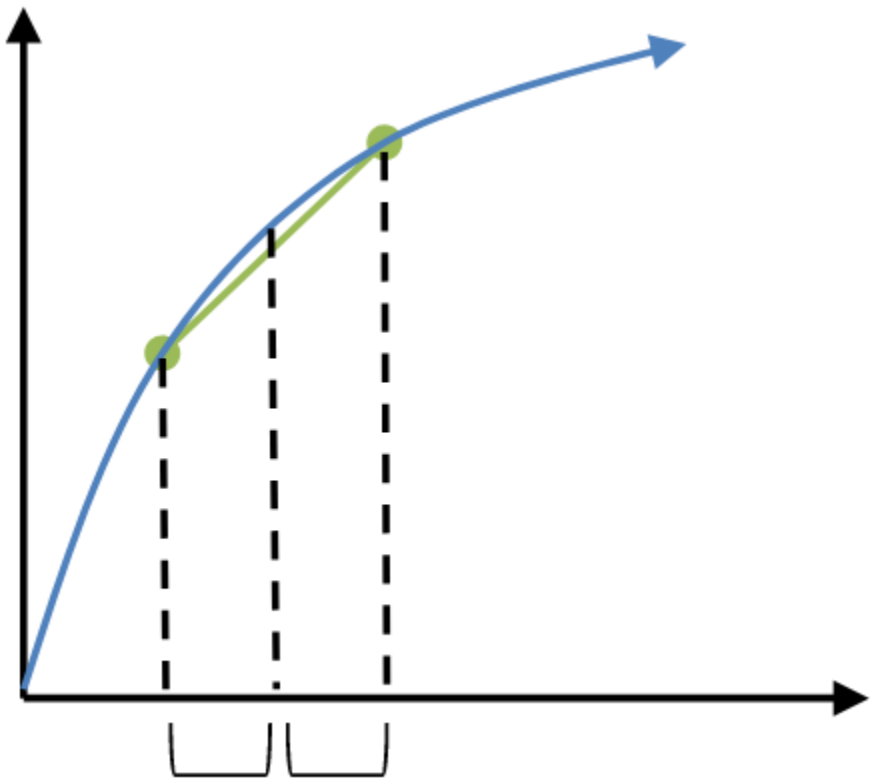
The *centered* finite difference approximation is:

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

where $\boldsymbol{d}_i$ is a 1-hot vector consisting of all zeros except for the $i$th entry of $\boldsymbol{d}_i$, 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

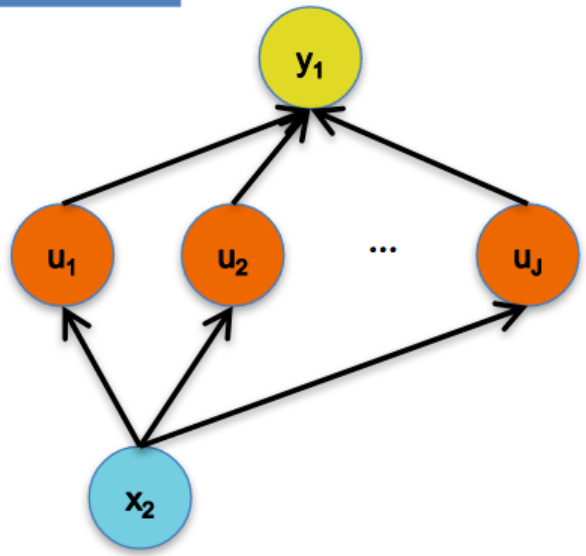


# Backpropagation – repeated application of chain rule

**Given:** $\boldsymbol{y} = g(\boldsymbol{u})$ 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

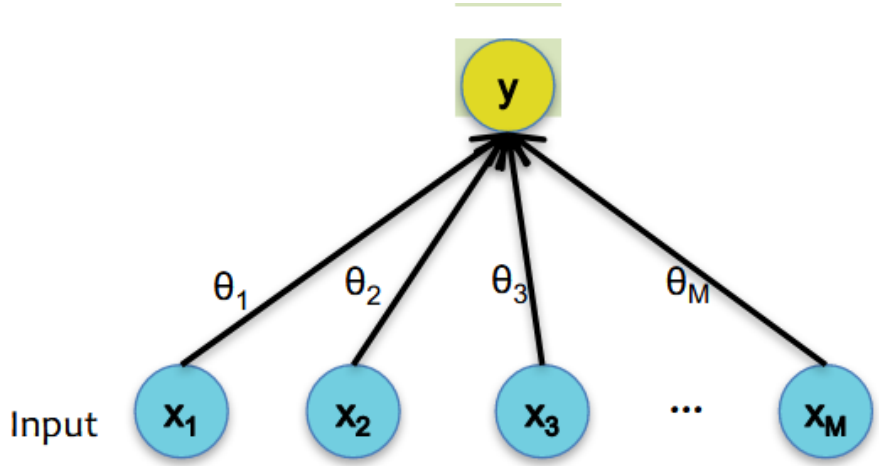
Backward Computation

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

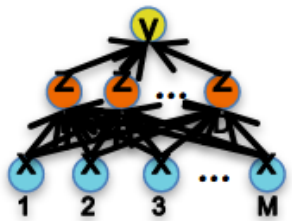
$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

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

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

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

## 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

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

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

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

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

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

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

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

# 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

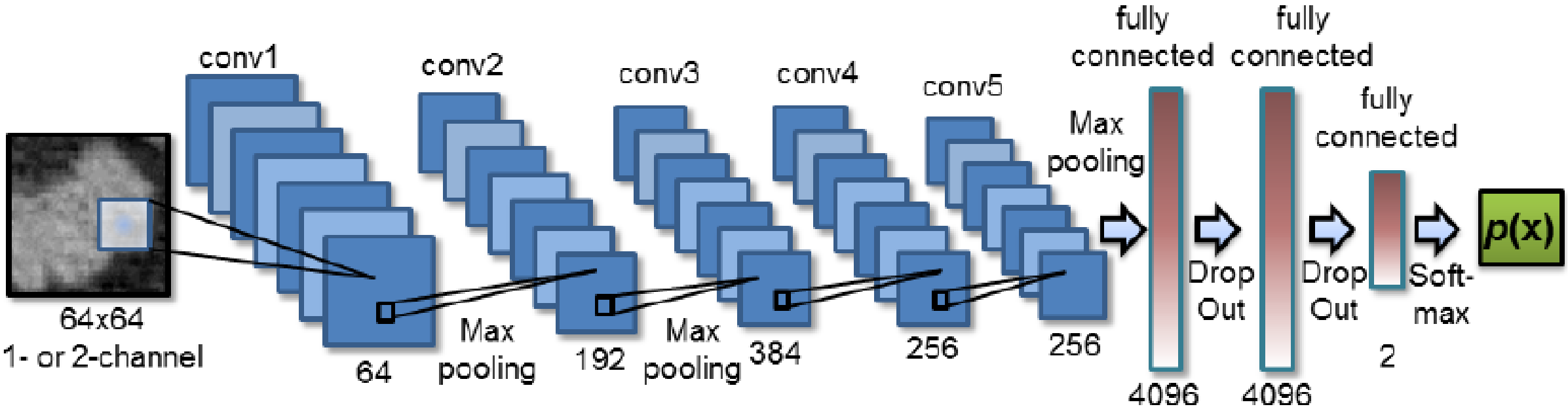
```
# Create the model here.
linear_fn1 = toynn_Linear(nInputs, nHidden)
relu_fn = toynn_ReLU()
linear_fn2 = toynn_Linear(nHidden, nOutputs)
loss_fn = toynn_CrossEntropyLoss()

# 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)
```

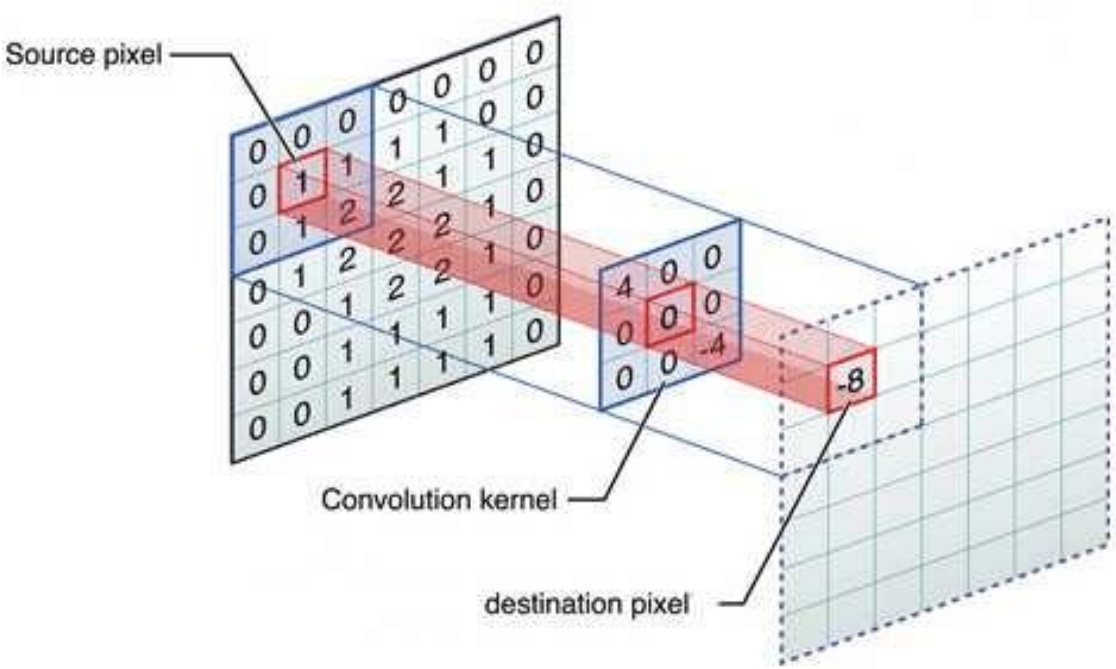

conv1
conv2
conv3
conv4
conv5
fully connected
fully connected
fully connected
Max pooling
64x64
1- or 2-channel
64
Max pooling
192
Max pooling
384
256
256
4096
Drop Out
4096
Drop Out
2
Soft-max
p(x)

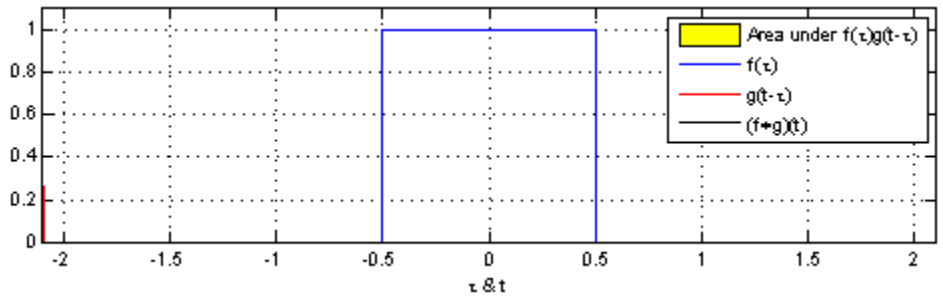
# Basic building blocks of the CNN architecture

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

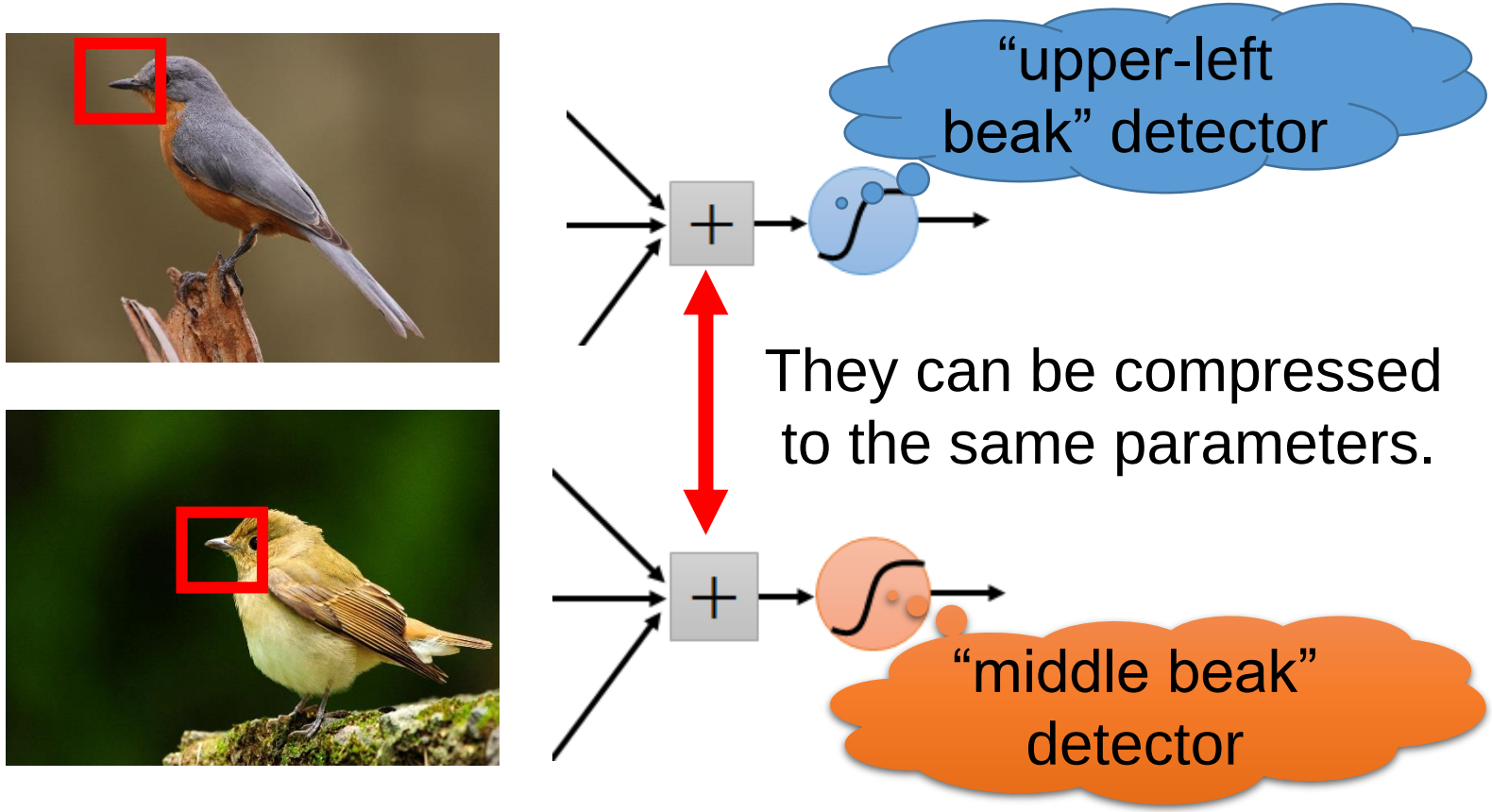
- Convolutional layer
  - Convolutional kernel
  - Pooling layer
  - Non-linearity

# Convolution operation

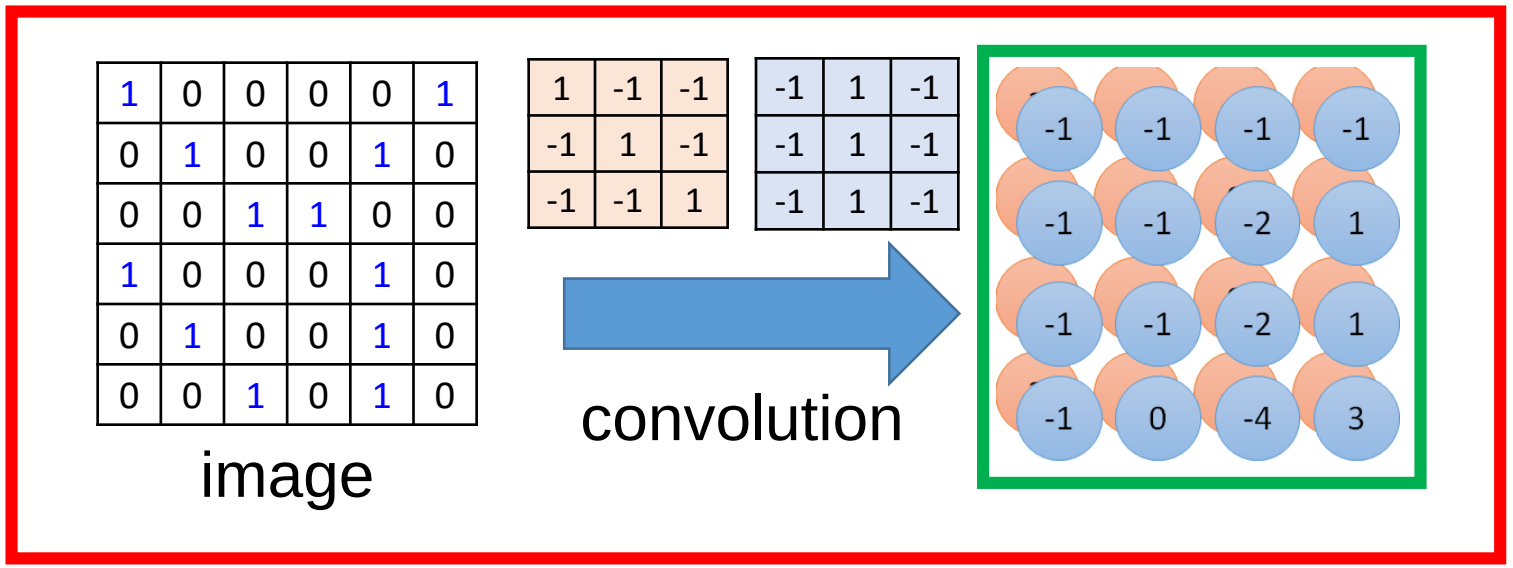


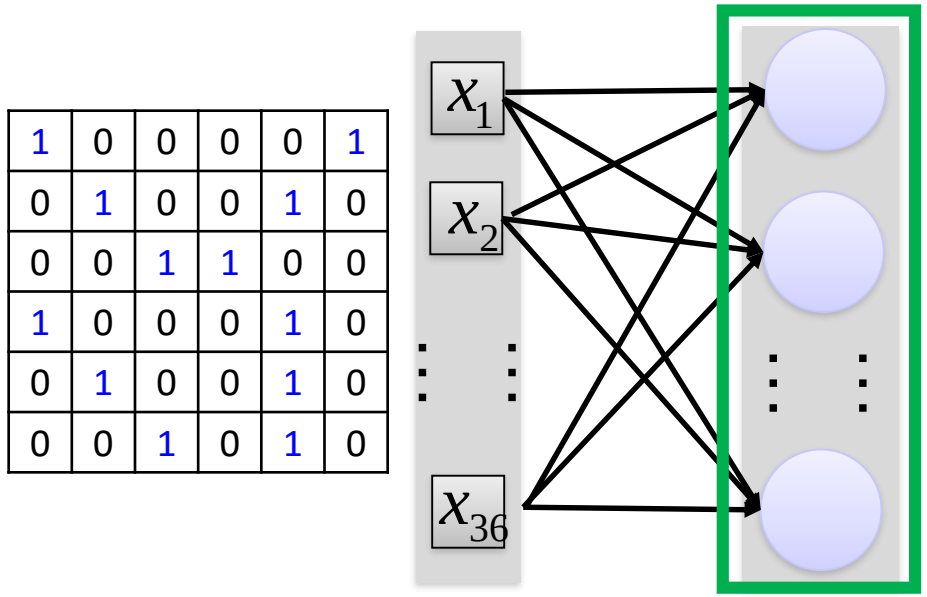
Same pattern appears in different places:
They can be compressed!
What about training a lot of such “small” detectors
and each detector must “move around”.



# Convolution v.s. Fully Connected



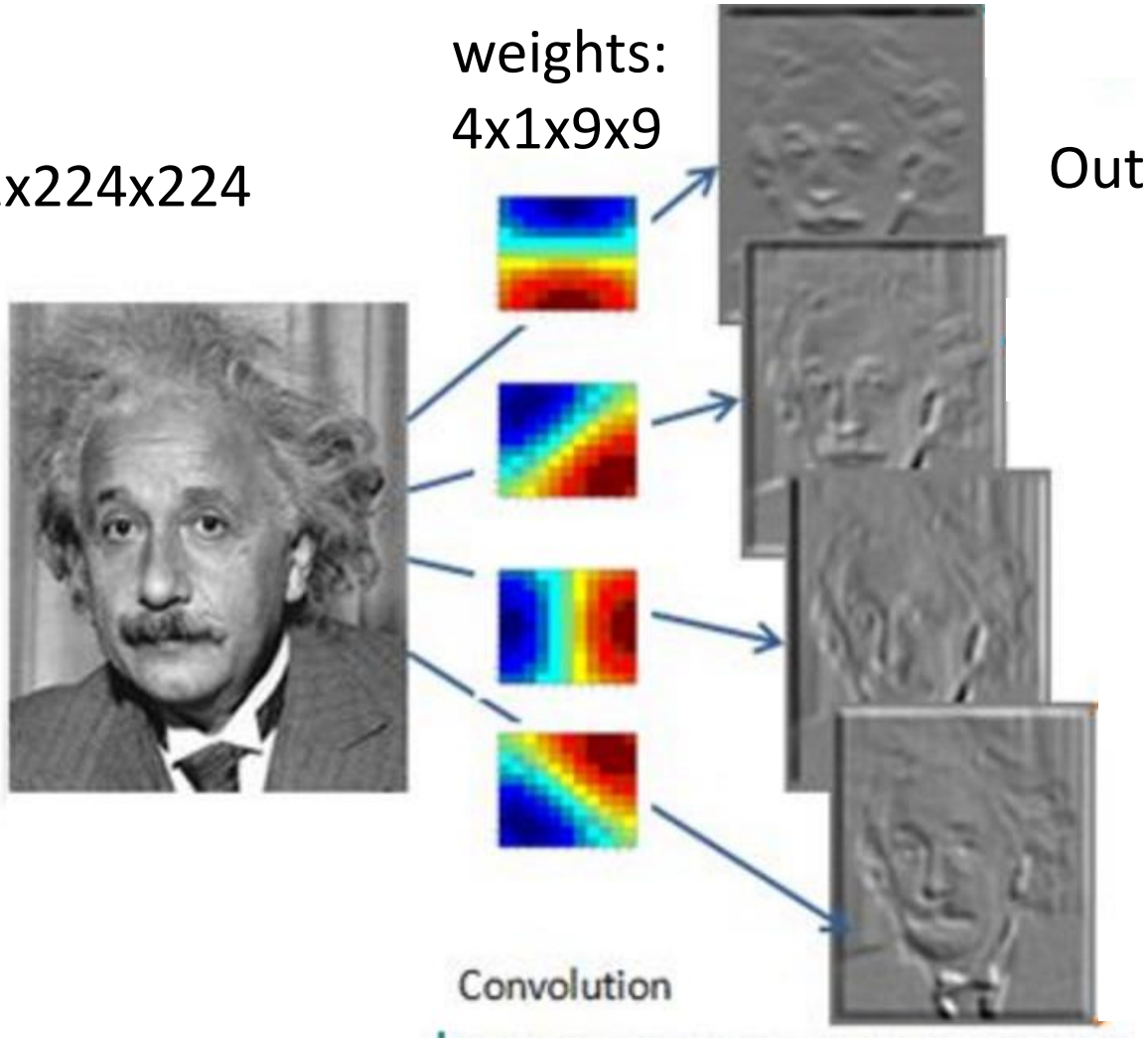

Fully-connected



# Convolutional Layer (with 4 filters)

Input: 1x224x224

weights:
4x1x9x9



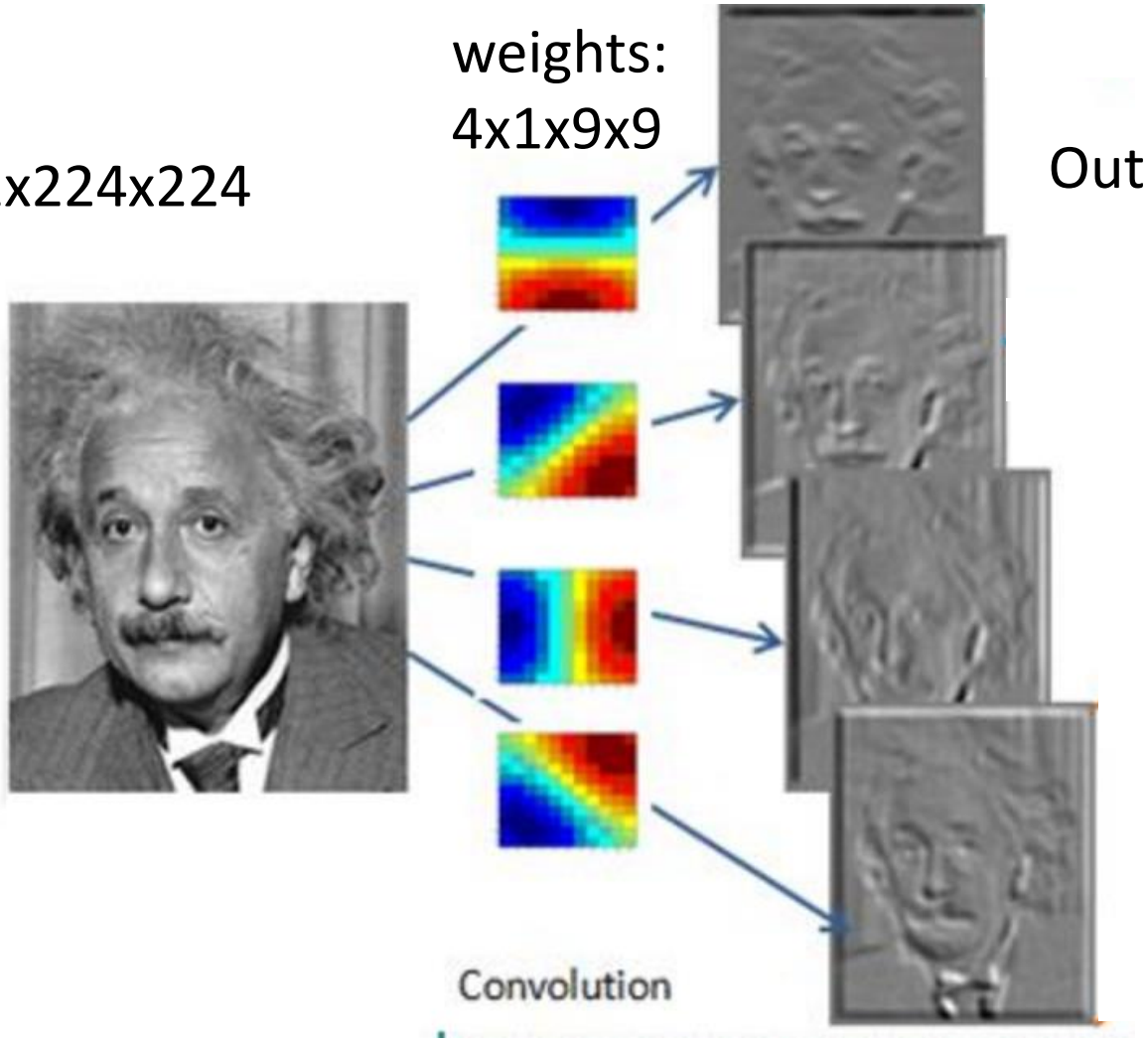

Output: 4x224x224

if zero padding,
and stride = 1

# Convolutional Layer (with 4 filters)

Input: 1x224x224

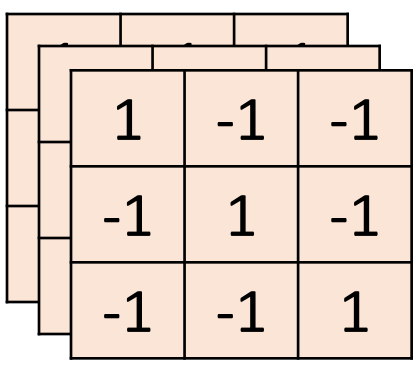
weights:
4x1x9x9



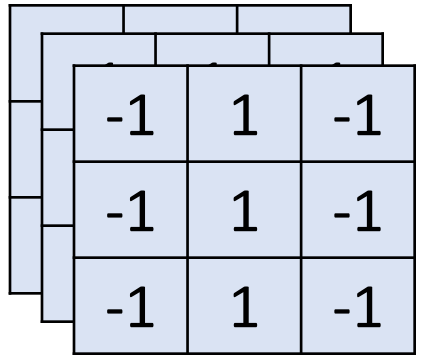

Output: 4x112x112

if zero padding,
but stride = 2

# Color image: RGB 3 channels – conv. over depth

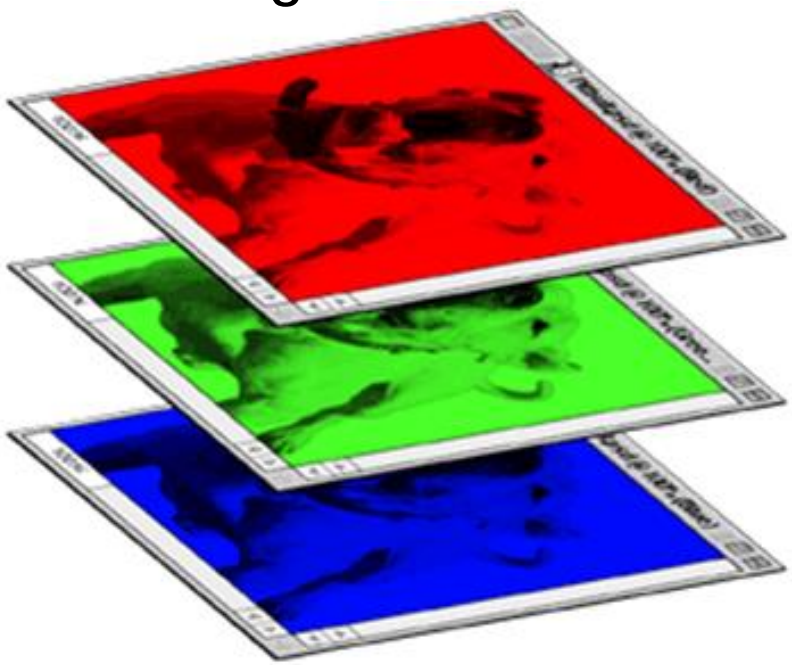


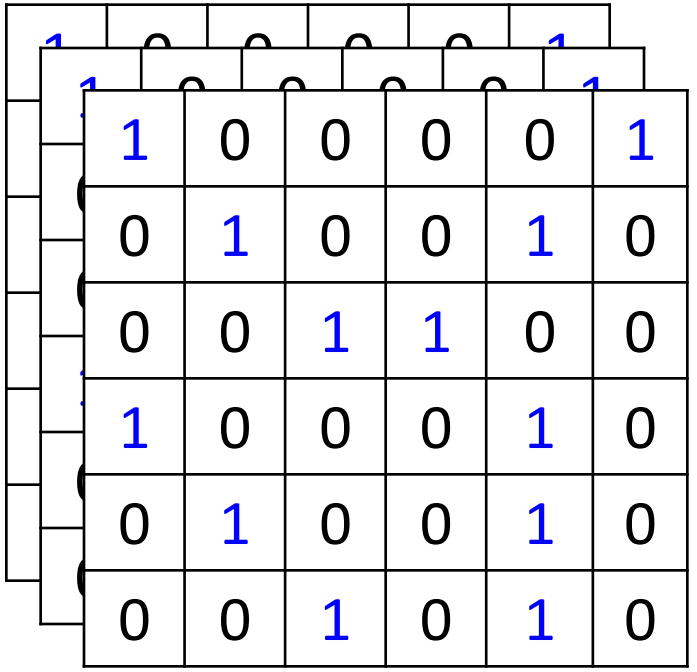

Filter 1



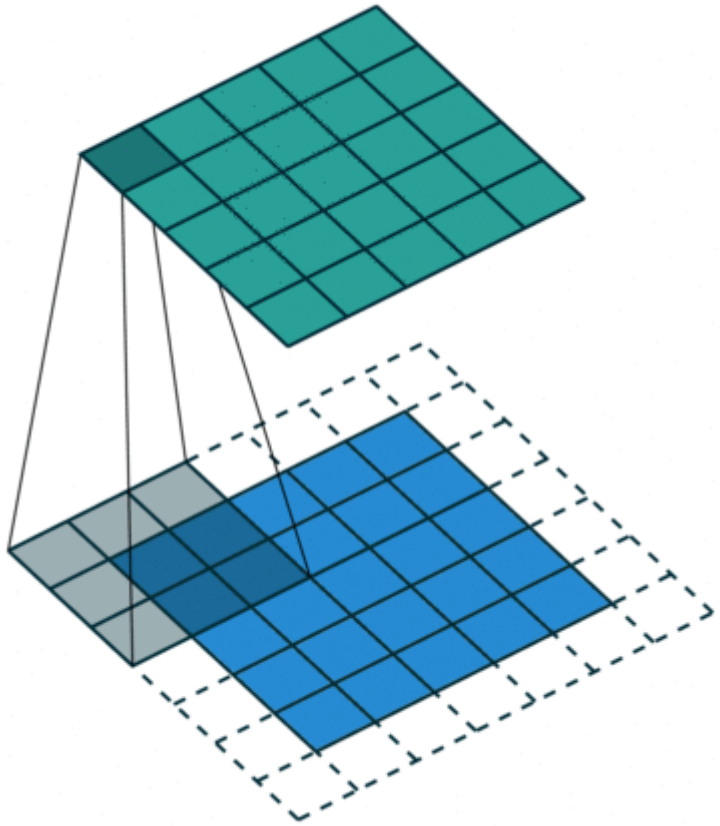

Filter 2 …3

Color image



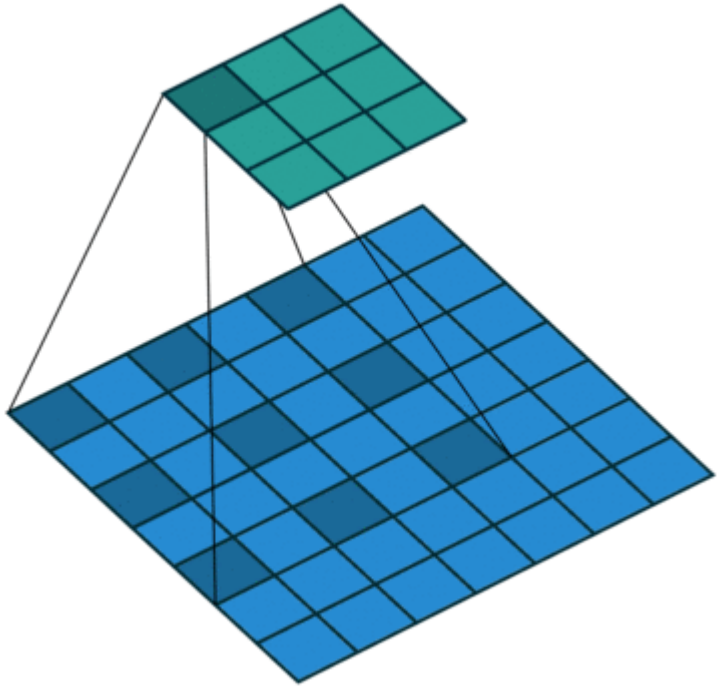


# Different types of convolution



Parameters:

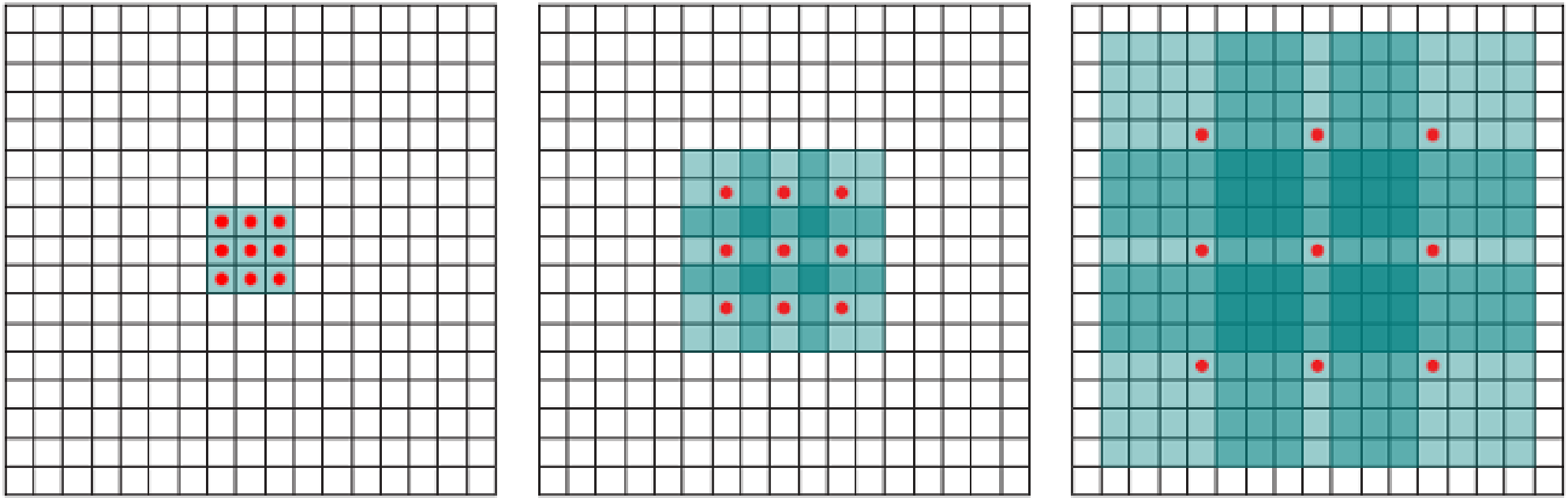
- ✓ Kernel stride
- ✓ Size
- ✓ Padding



**Normal vs dialated**

Dialation width = 2

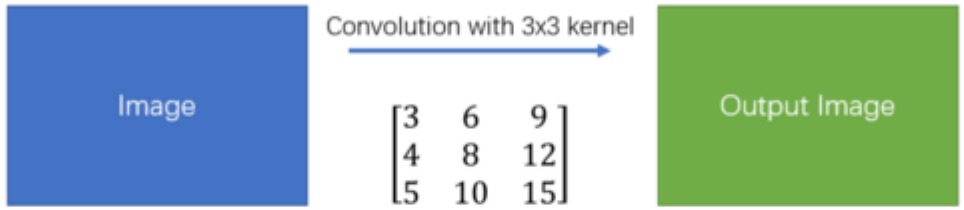
# Dialated convolution



# Spatially Separable convolution

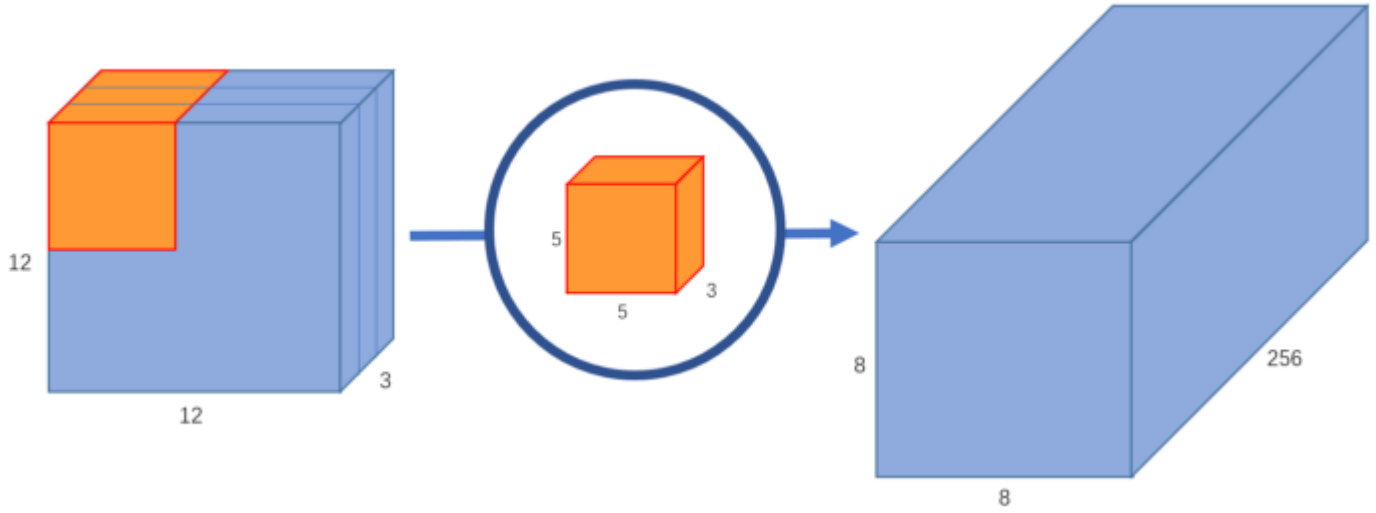
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

## Simple Convolution




## Spatial Separable Convolution

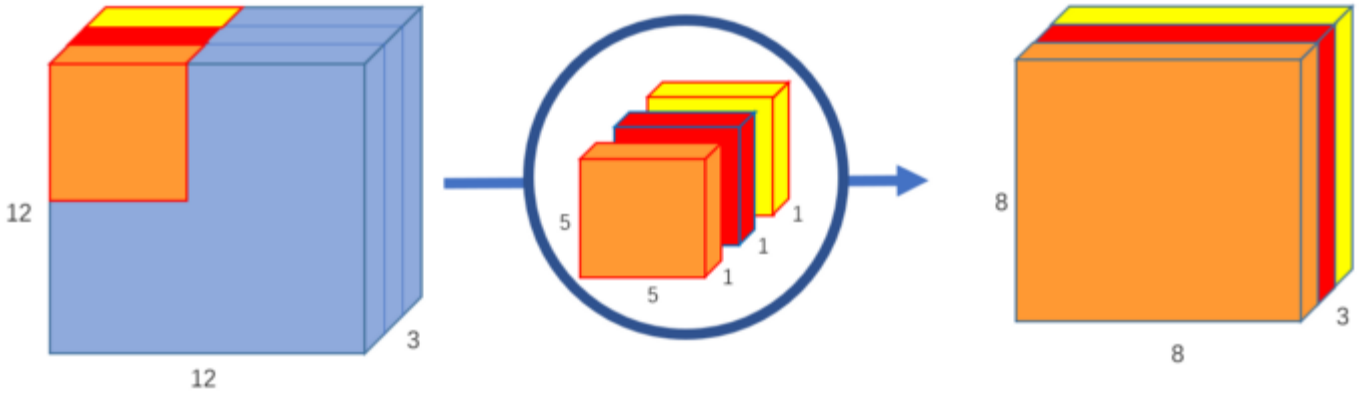
# Depthwise separable convolution




Convolving by 256 5x5 kernels over the input volume

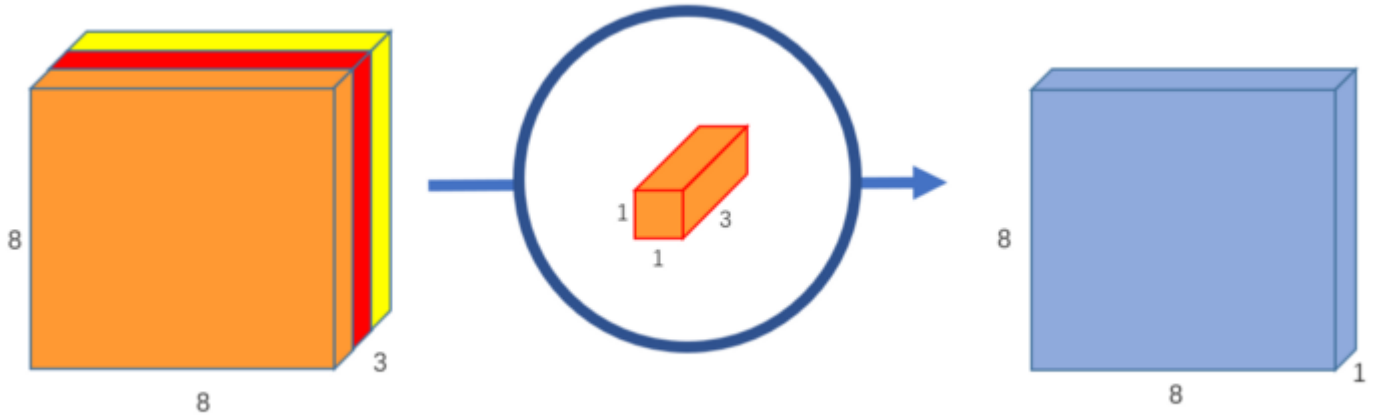
# Depthwise separable convolution – step1

Along depth

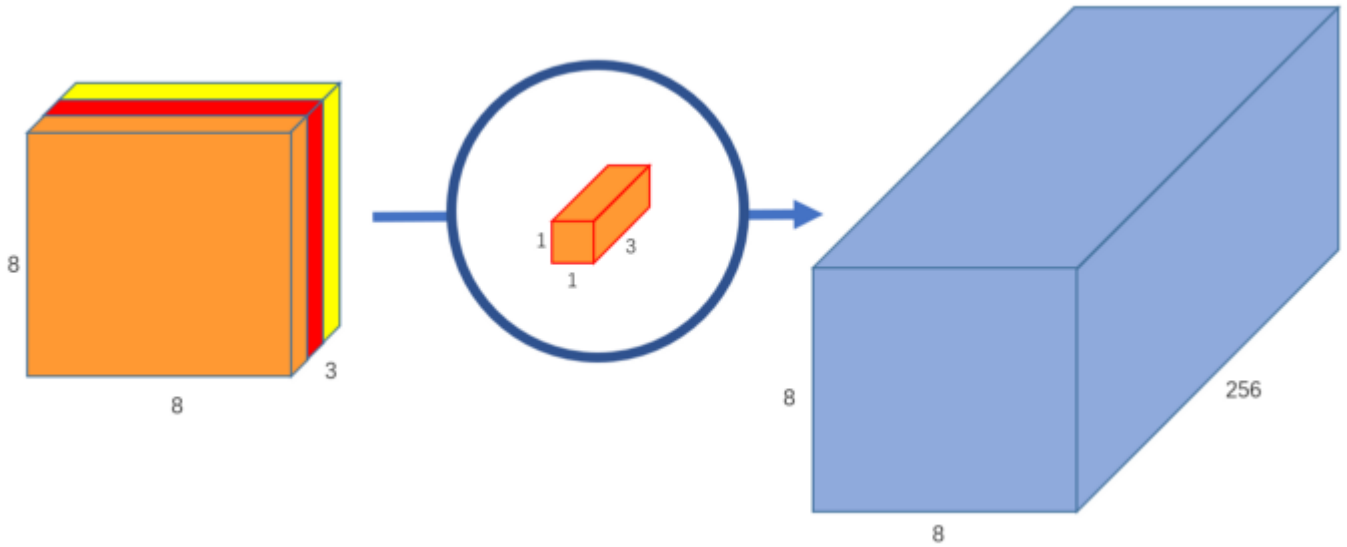



Each 5x5x1 kernel iterates 1 channel of the image (note: **1 channel**, not all channels), getting the scalar products of every 25 pixel group, giving out a 8x8x1 image. Stacking these images together creates a 8x8x3 image.

# Depthwise separable convolution – step2




Pointwise

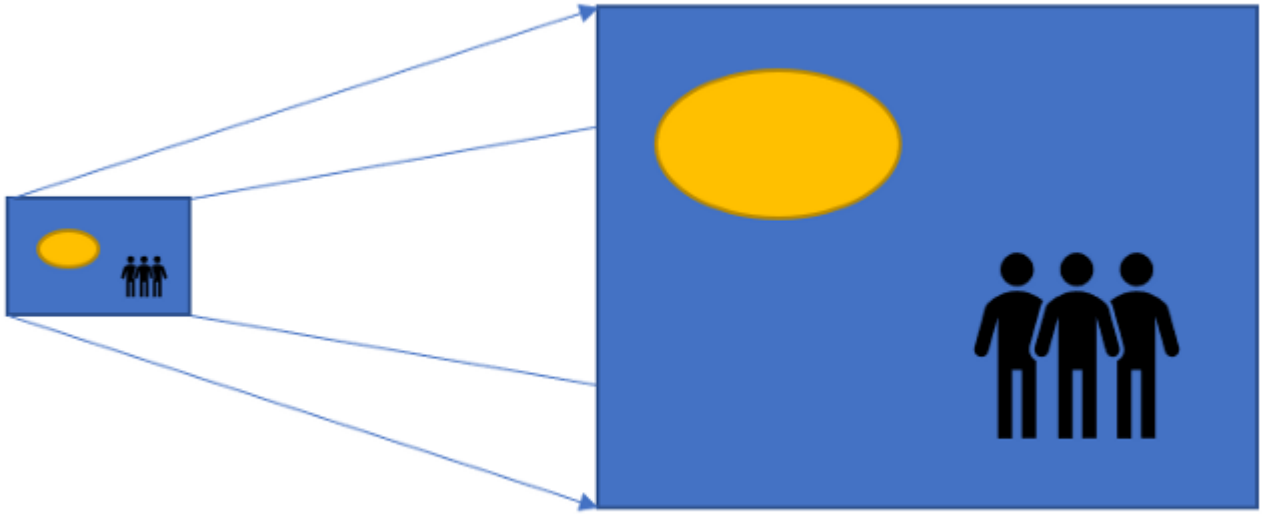



1x1 conv

Let's calculate the number of multiplications the computer has to do in the original convolution. There are 256 5x5x3 kernels that move 8x8 times. That's 256x3x5x5x8x8=1,228,800 multiplications.

What about the separable convolution? In the depthwise convolution, we have 3 5x5x1 kernels that move 8x8 times. That's 3x5x5x8x8 = 4,800 multiplications. In the pointwise convolution, we have 256 1x1x3 kernels that move 8x8 times. That's 256x1x1x3x8x8=49,152 multiplications. Adding them up together, that's 53,952 multiplications.

# Transpose convolution



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

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

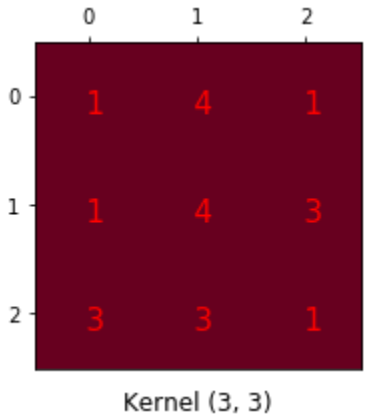
Here is a constructed matrix with a vector:

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

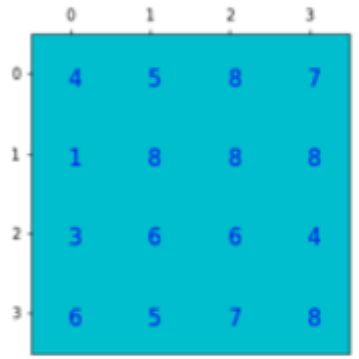
which is equal to $\begin{pmatrix} k1\,x1 + k2\,x2 + k3\,x4 + k4\,x5 \\ k1\,x2 + k2\,x3 + k3\,x5 + k4\,x6 \\ k1\,x4 + k2\,x5 + k3\,x7 + k4\,x8 \\ k1\,x5 + k2\,x6 + k3\,x8 + k4\,x9 \end{pmatrix}$.

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.

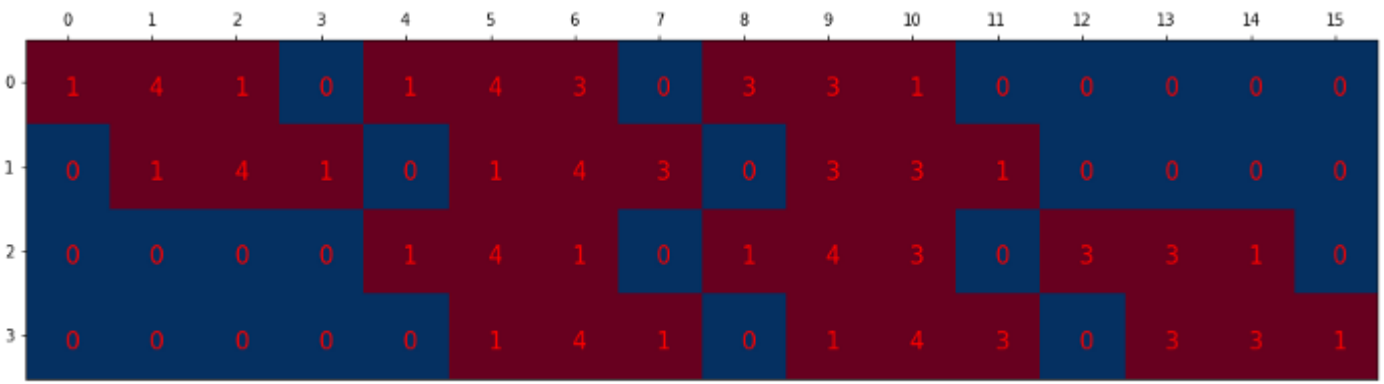
Convolution as a matrix multiplication



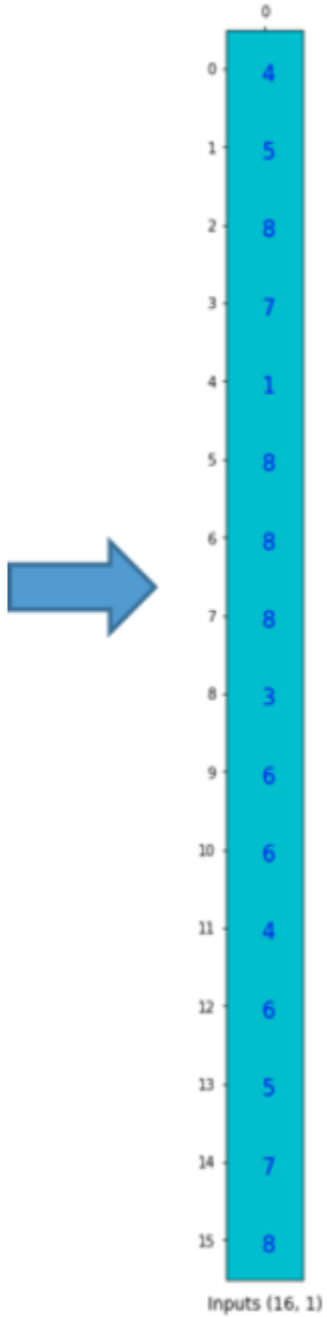
Kernel (3, 3)



Inputs (4, 4)

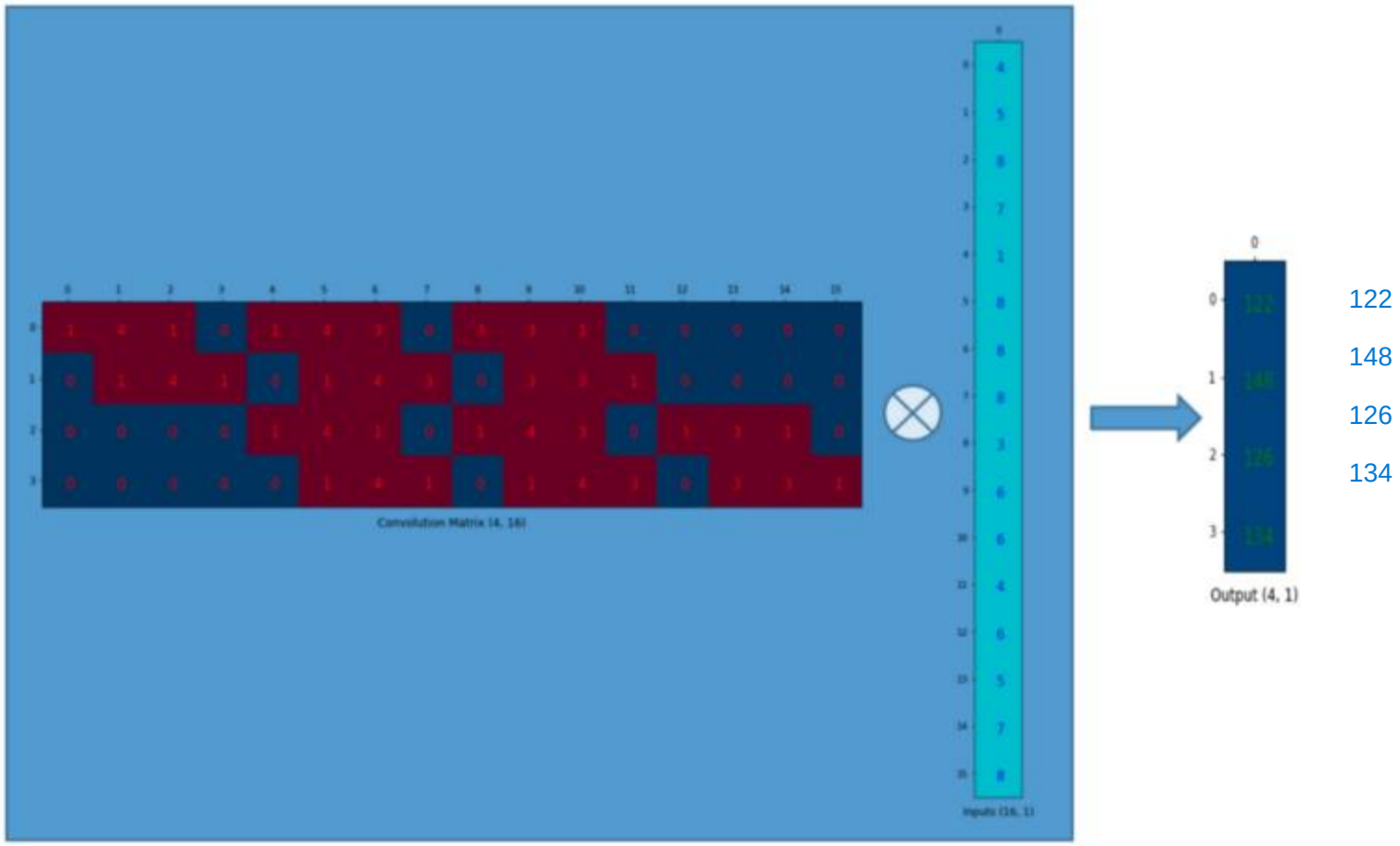


Convolution Matrix (4, 16)



Inputs (16, 1)

# Many to one mapping – 9 values to 1 value





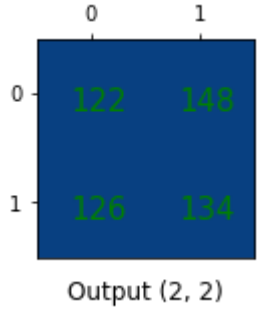


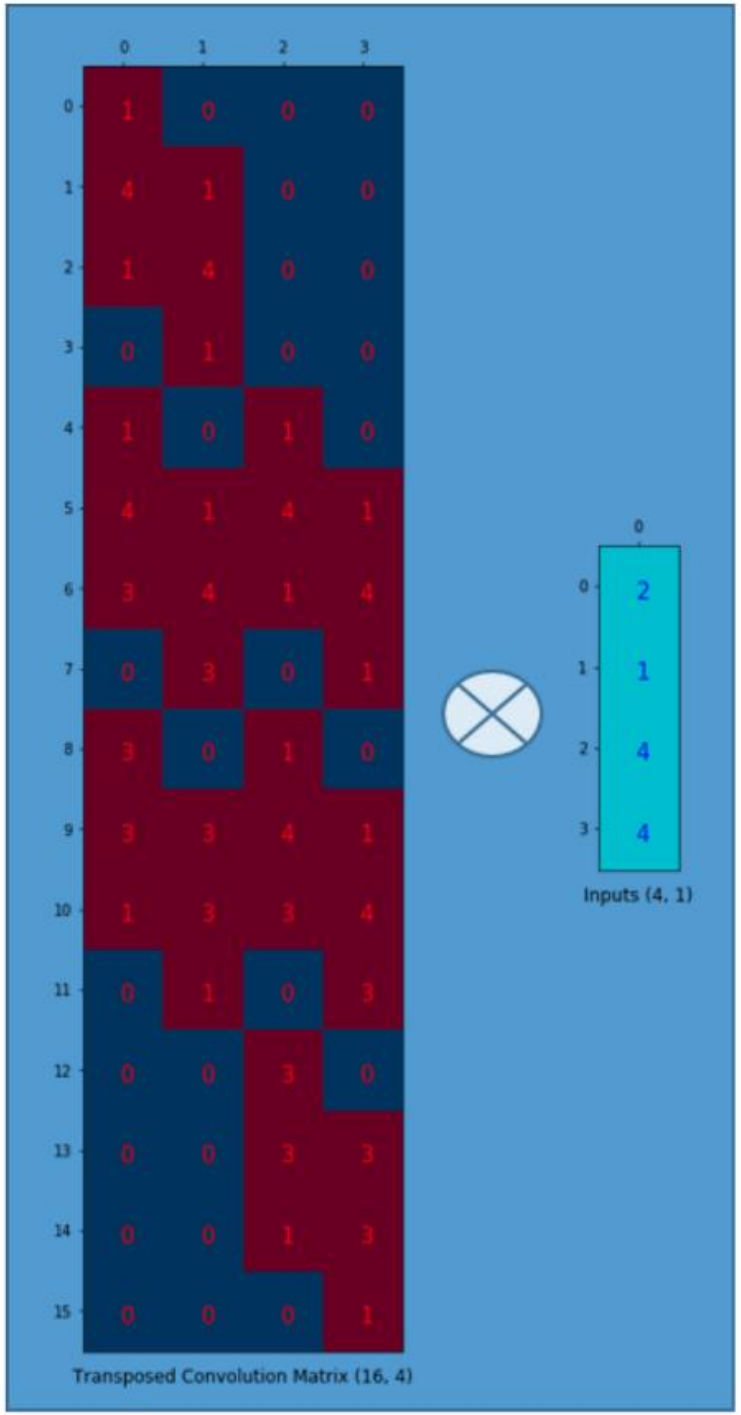
122

148

126

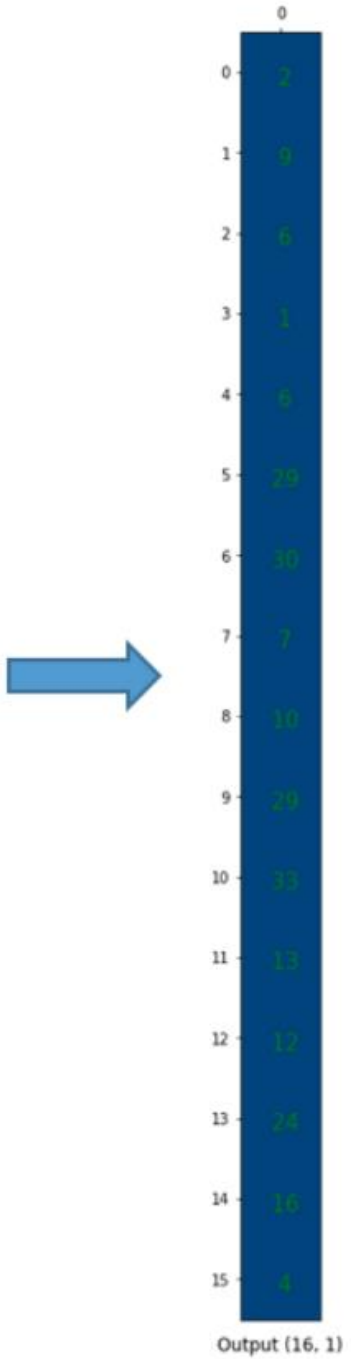
134

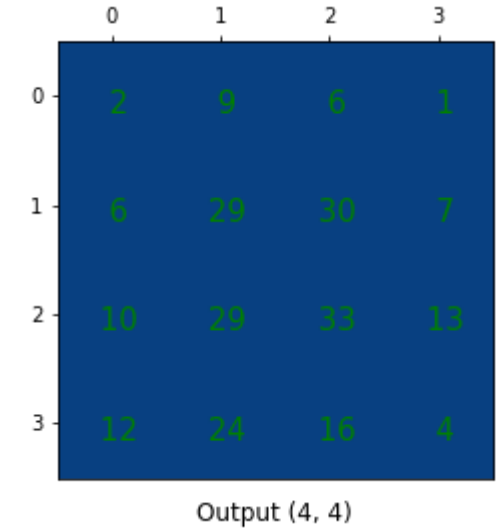








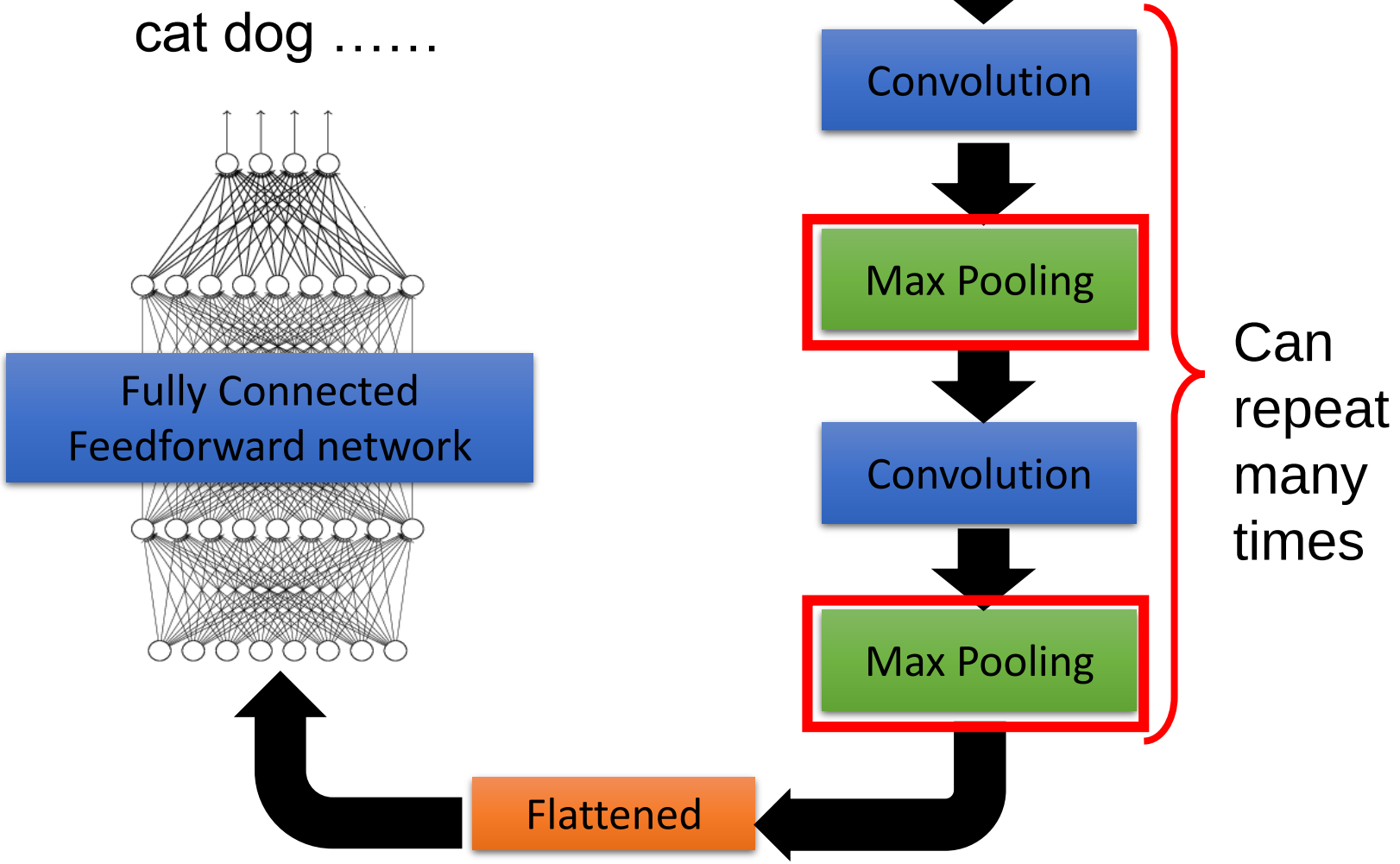



- Fractionally-strided convolution
- Deconvolution

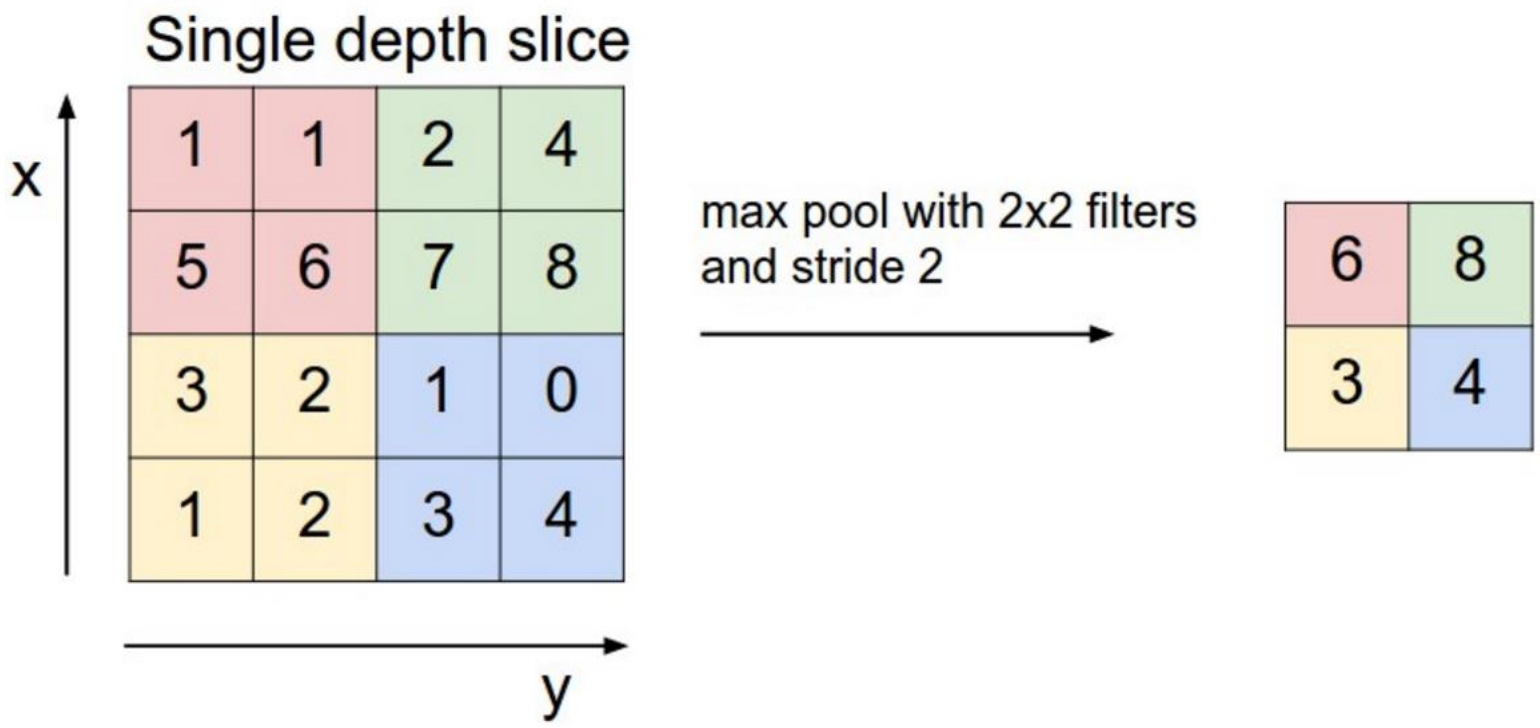
One to many mapping

# 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

Convolution

Max Pooling

A new image

Convolution

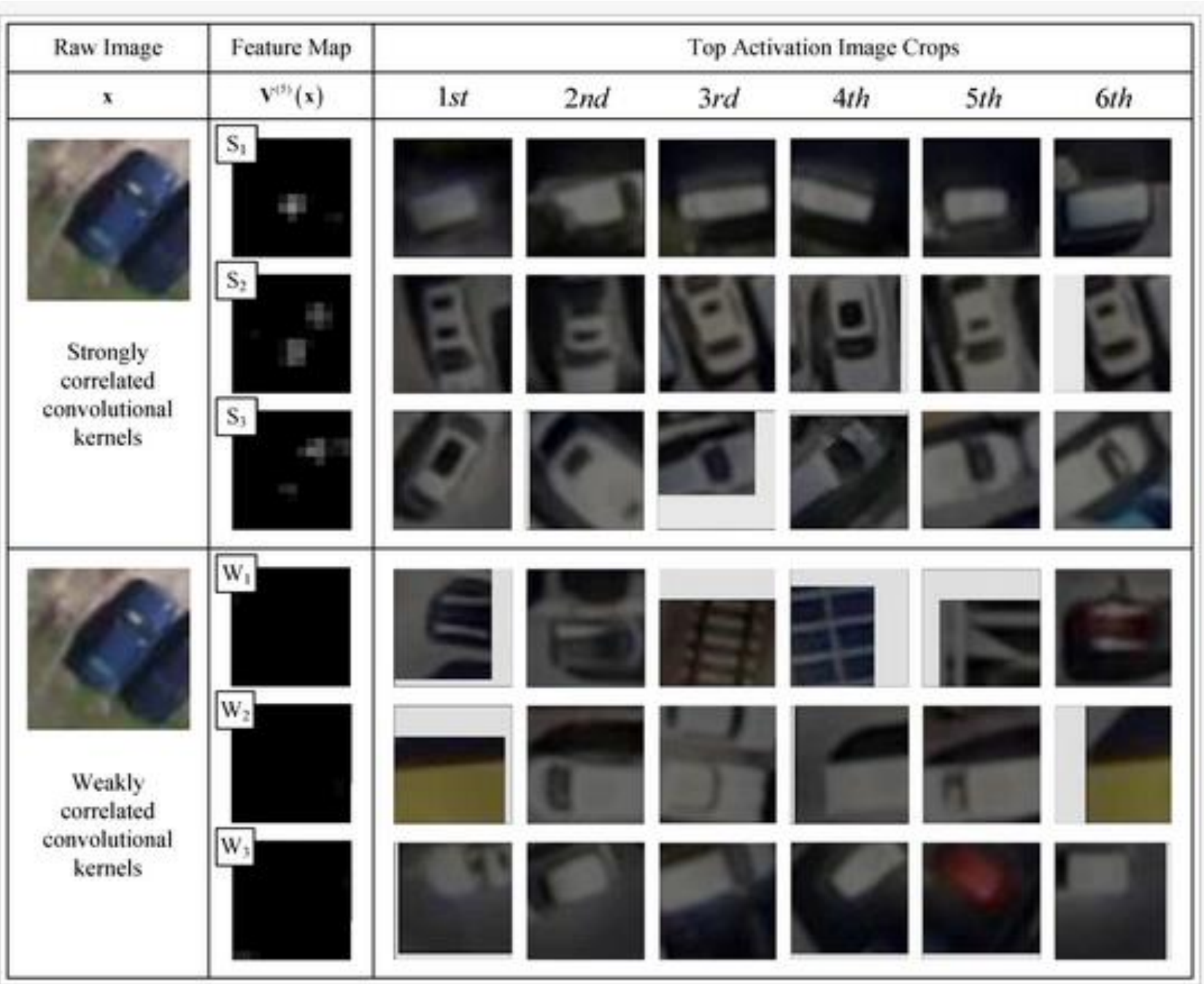
Max Pooling

A new image

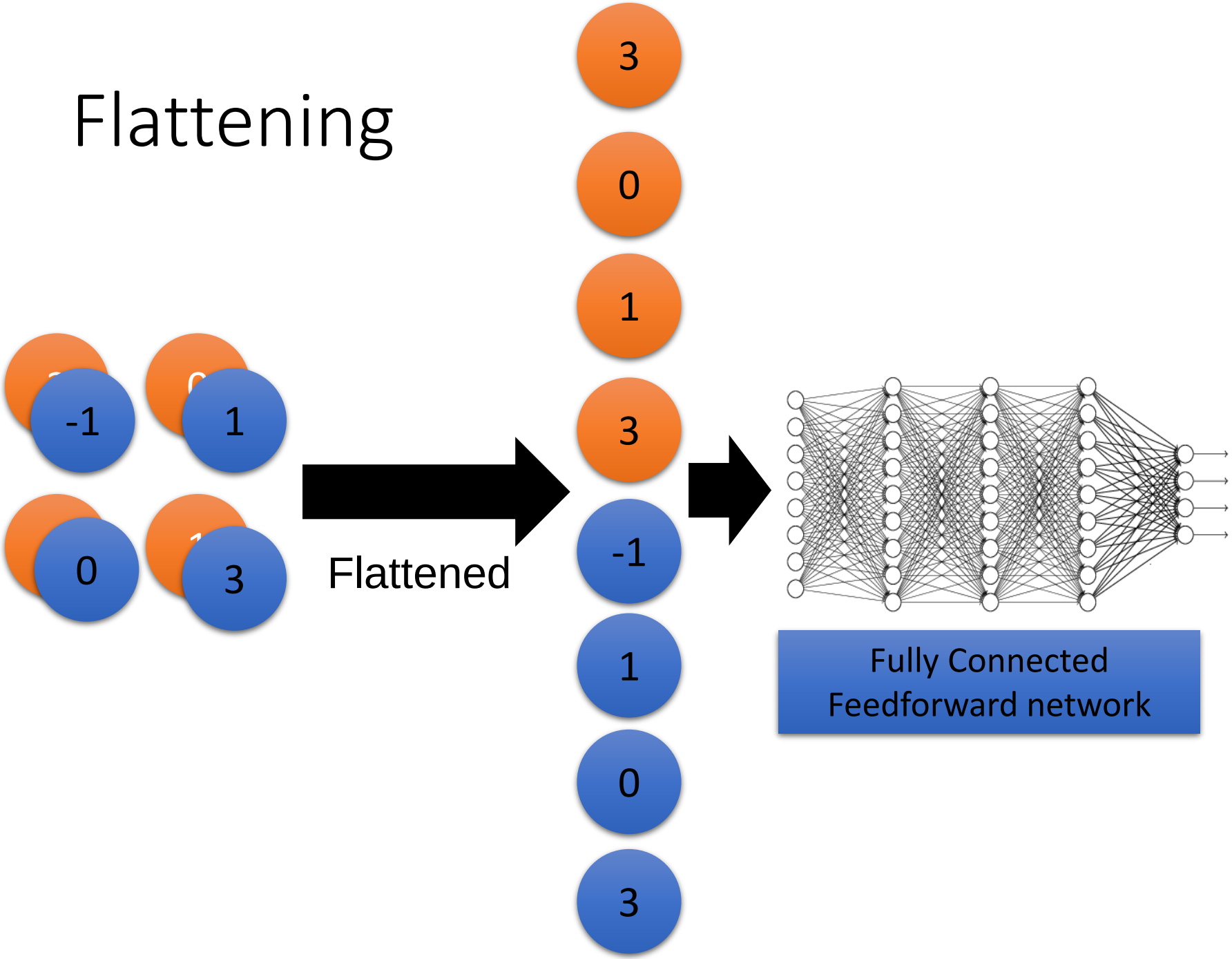
Flattened

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

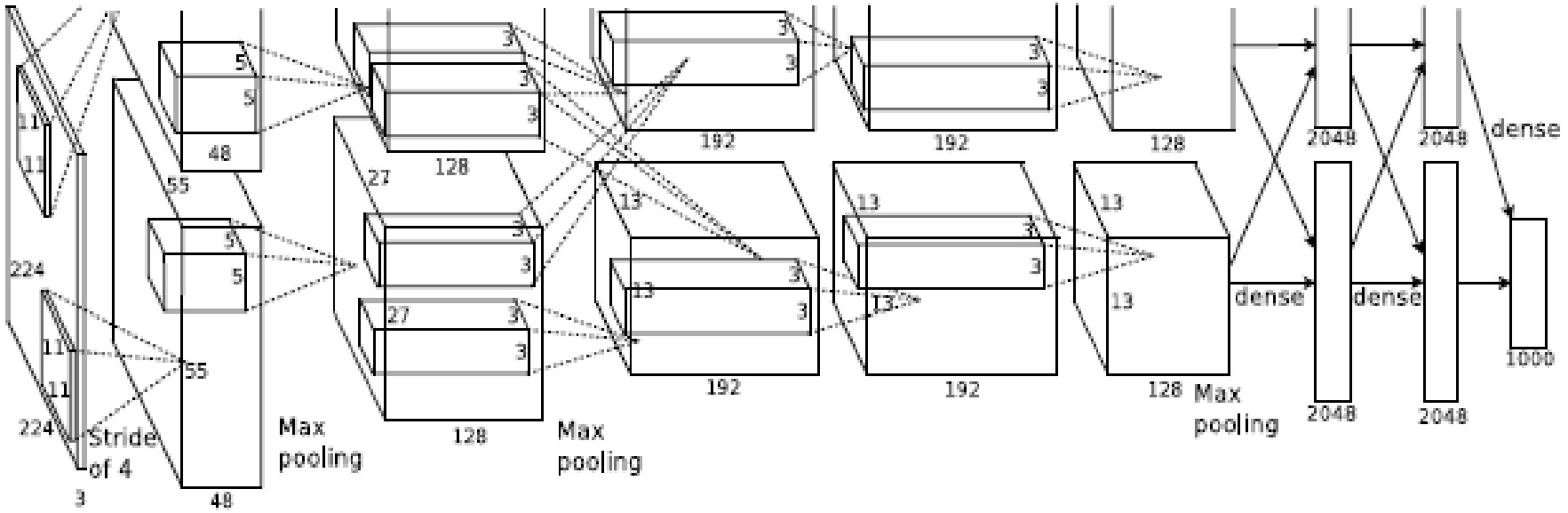
- 5 convolutional layers
- 3 fully connected layers + soft-max
- 650K neurons , 60 Mln weights

**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

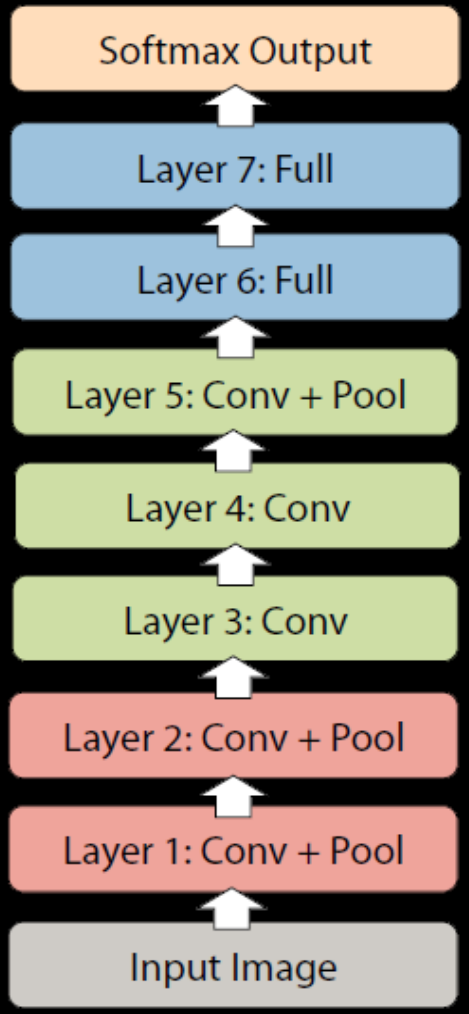
arena football
indoor american football
arena football
canadian football
american football
women's lacrosse

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?

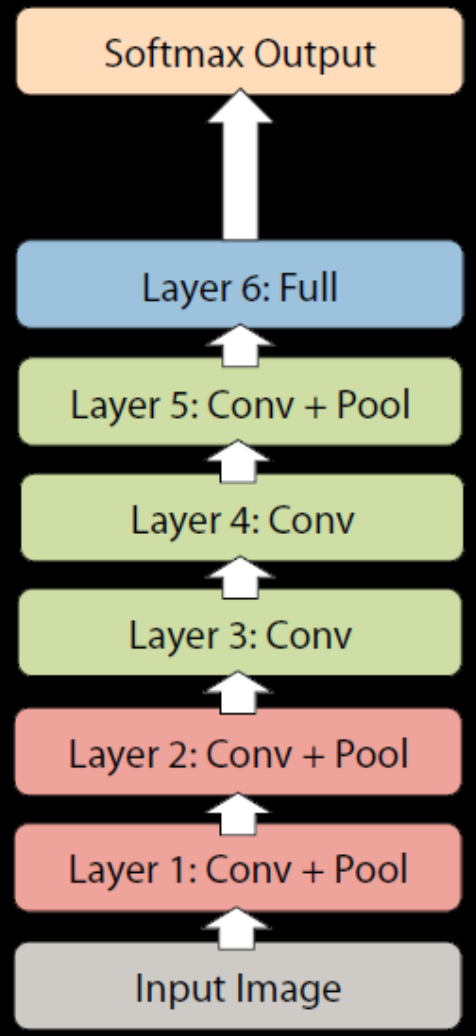
- 8 layers total
- Trained on Imagenet dataset [Deng et al. CVPR'09]
- 18.2% top-5 error
- Our reimplementation:
  18.1% top-5 error




Courtsey: ICRI

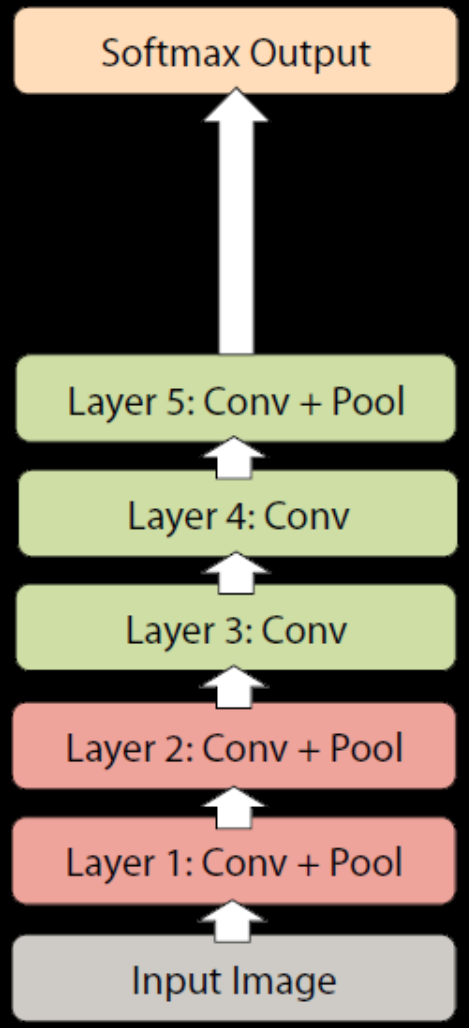
# Why do we need a *deep* CNN?

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



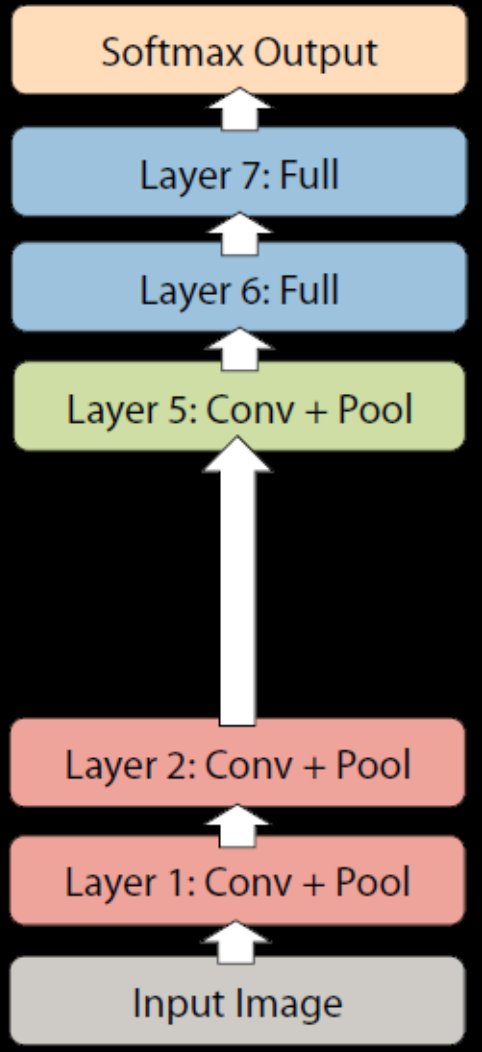
# Why do we need a *deep* CNN?

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



# Why do we need a *deep* CNN?

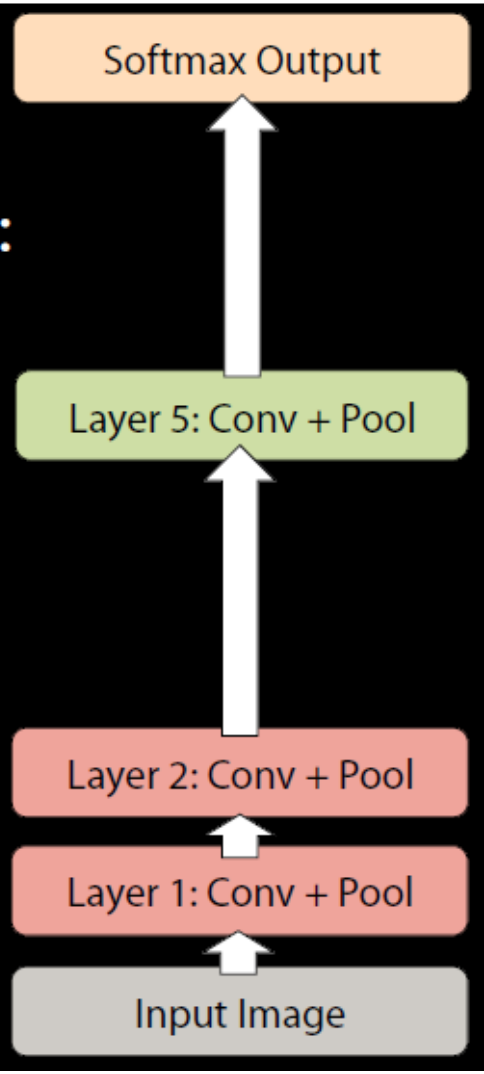
- Now try removing upper feature extractor layers:
  - Layers 3 & 4
- Drop ~1 million parameters
- 3.0% drop in performance



# 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

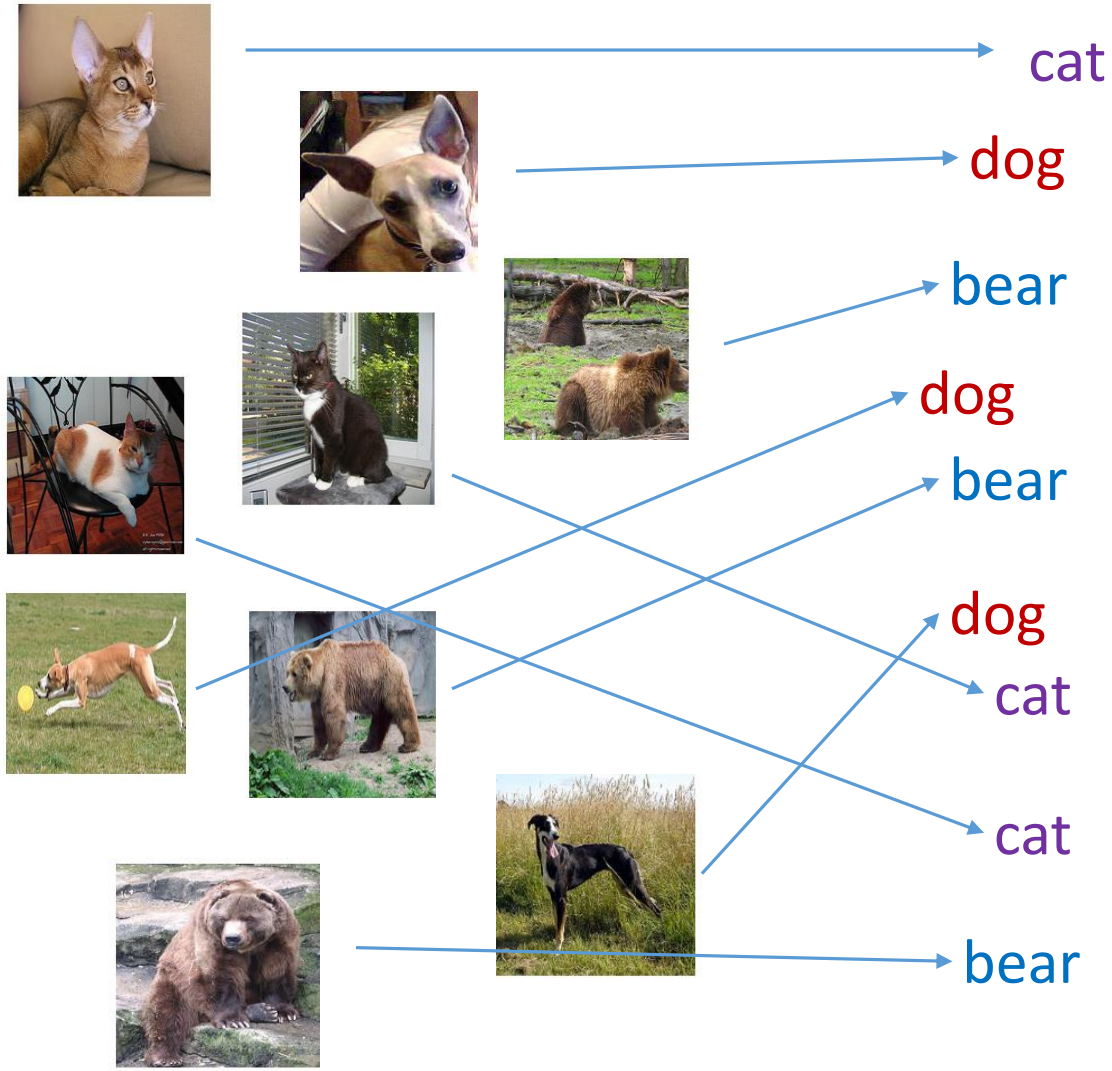
# Summary: Image Features

- The idea of low, mid, and high level features
- Largely replaced by Neural networks
- But there is a direct connection between the feature hierarchy

- Many other features proposed
  - LBP: Local Binary Patterns: Useful for recognizing faces.
  - Dense SIFT: SIFT features computed on a grid similar to the HOG features.
  - etc.

# Supervised Learning vs Unsupervised Learning

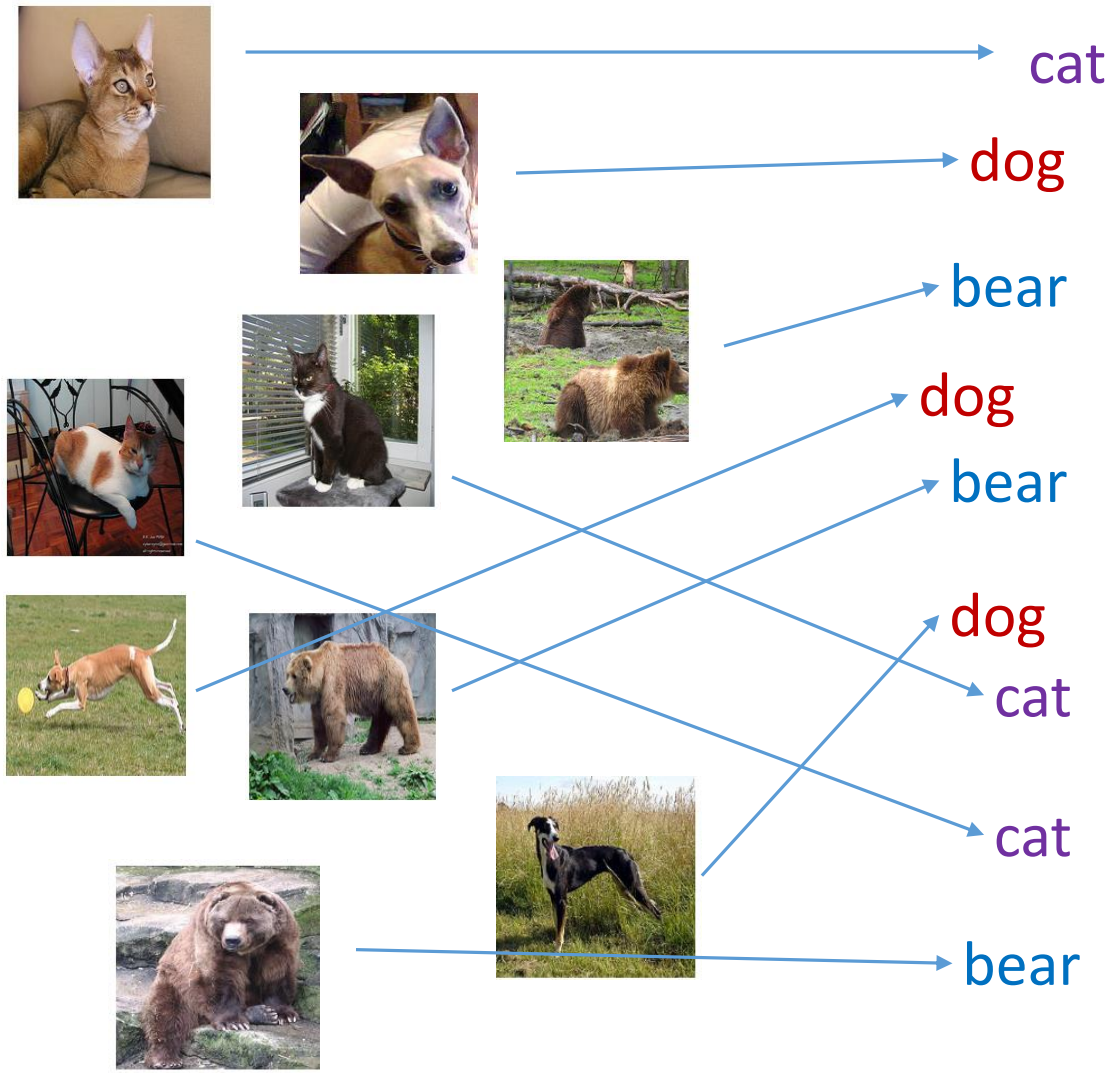
$x \rightarrow y$



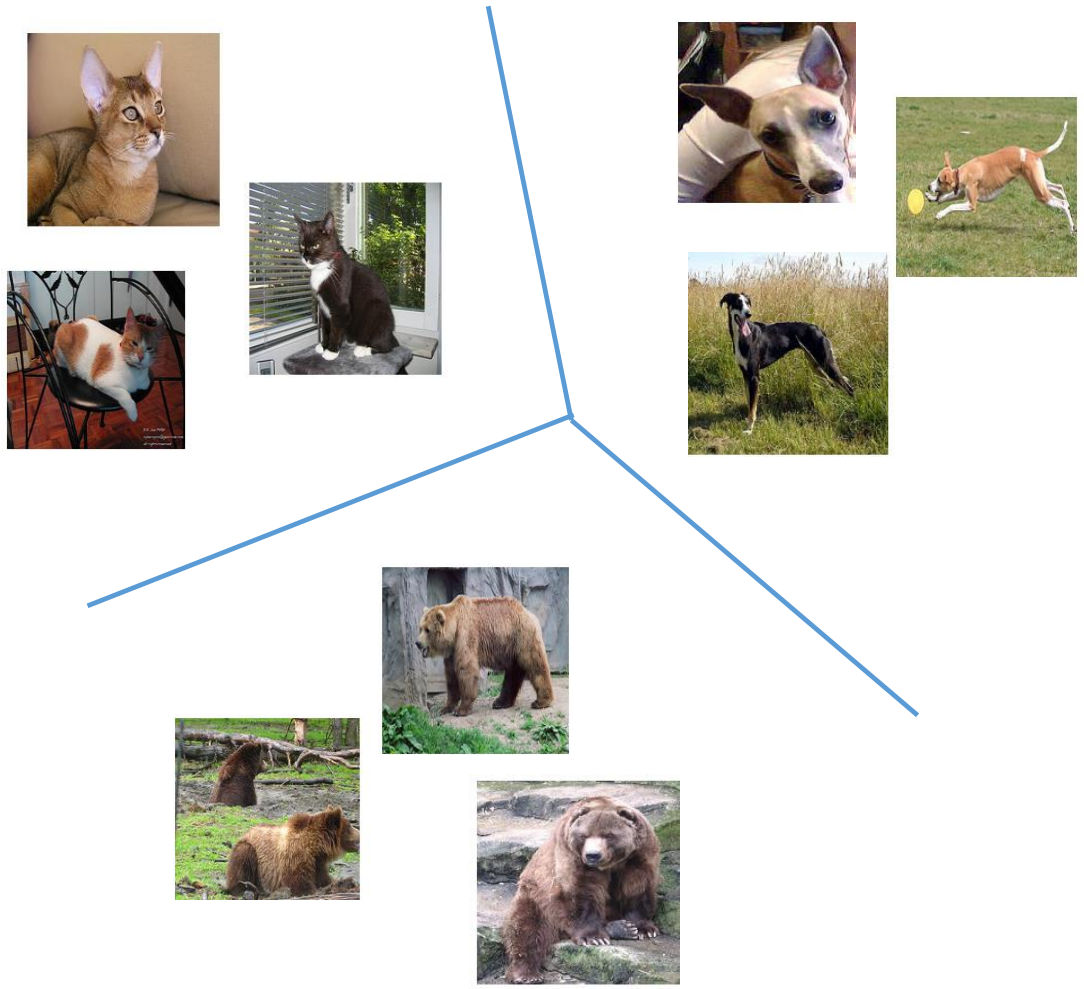

$x$

# Supervised Learning vs Unsupervised Learning

$x \rightarrow y$

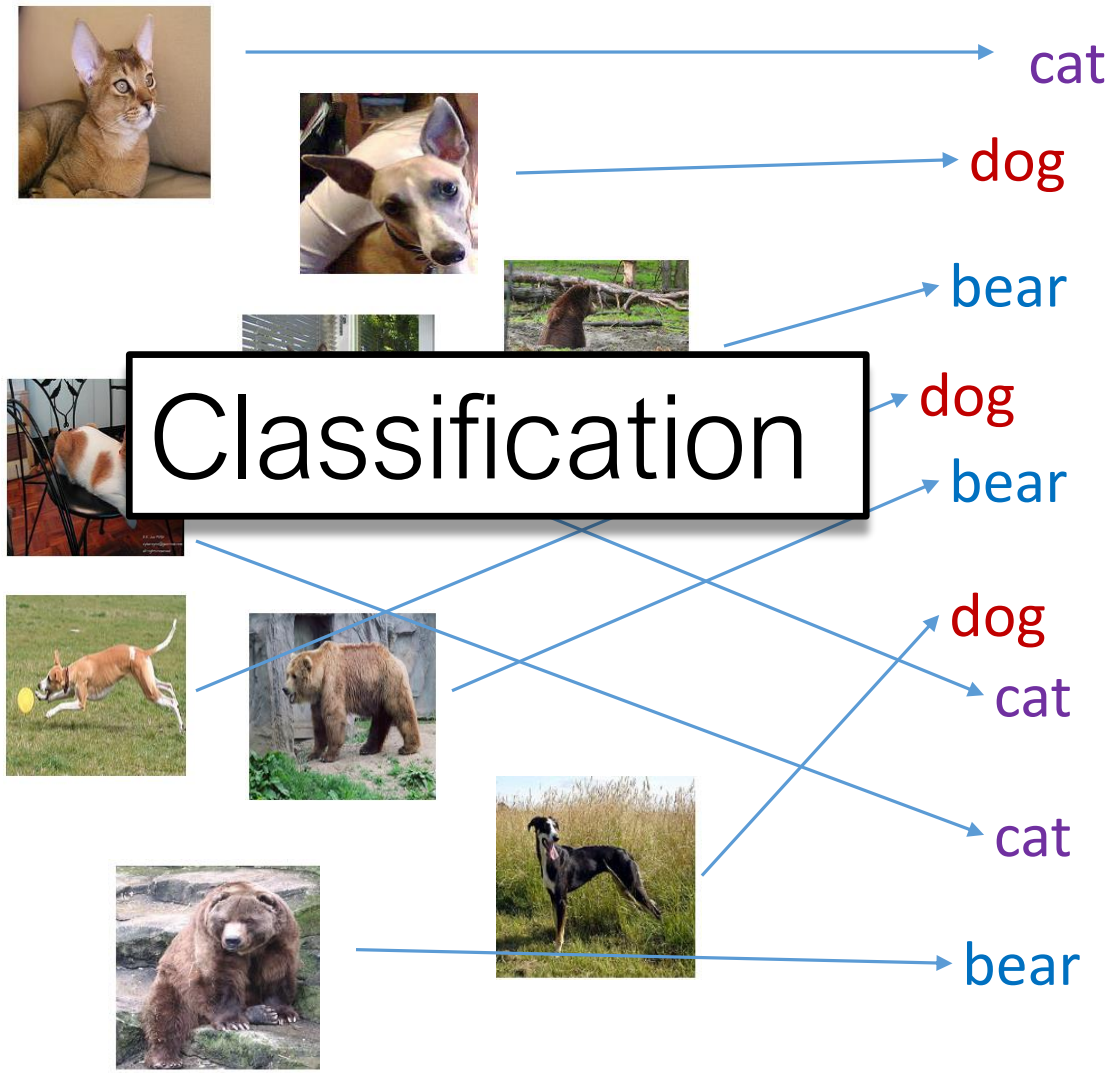



$x$



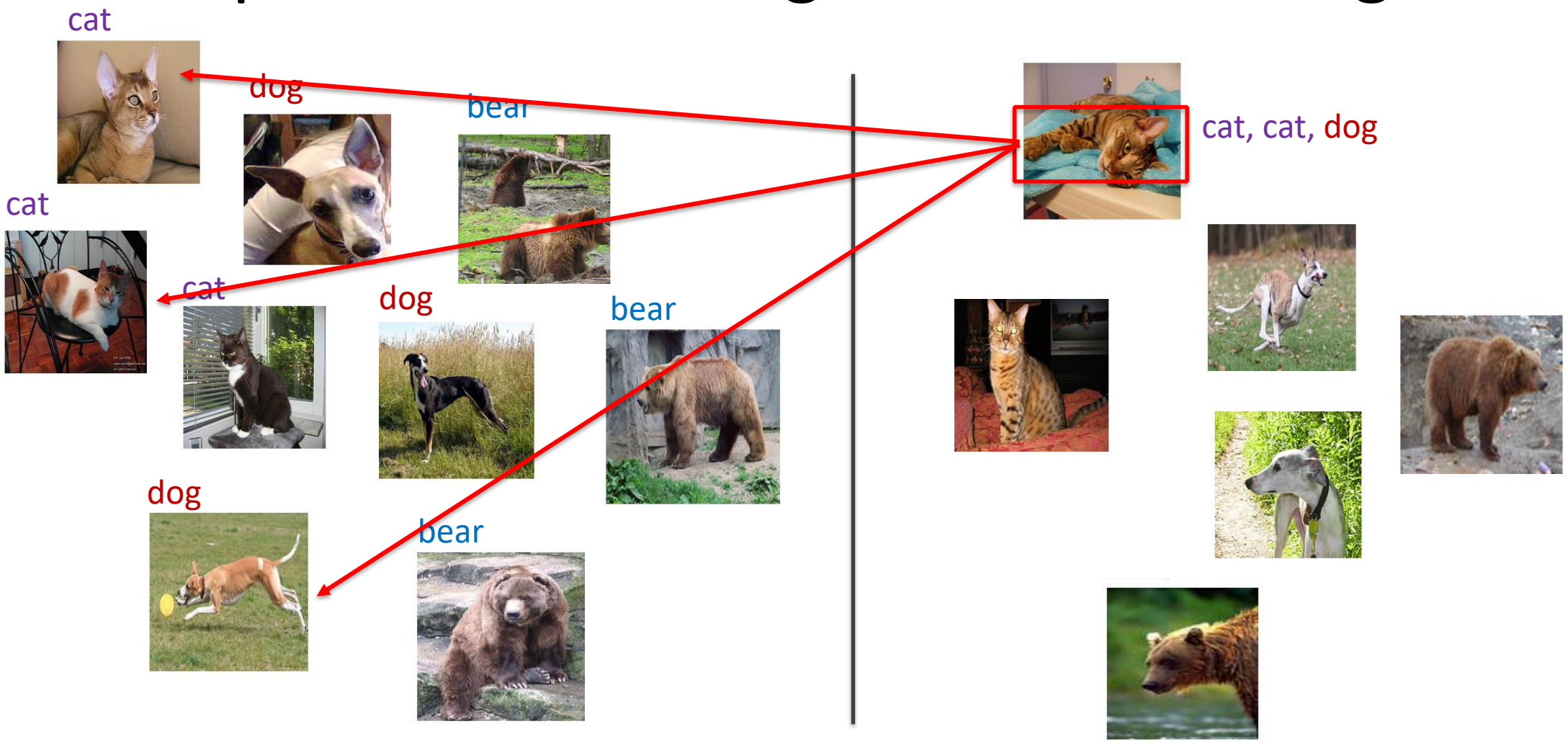
# Supervised Learning vs Unsupervised Learning

$x \rightarrow y$

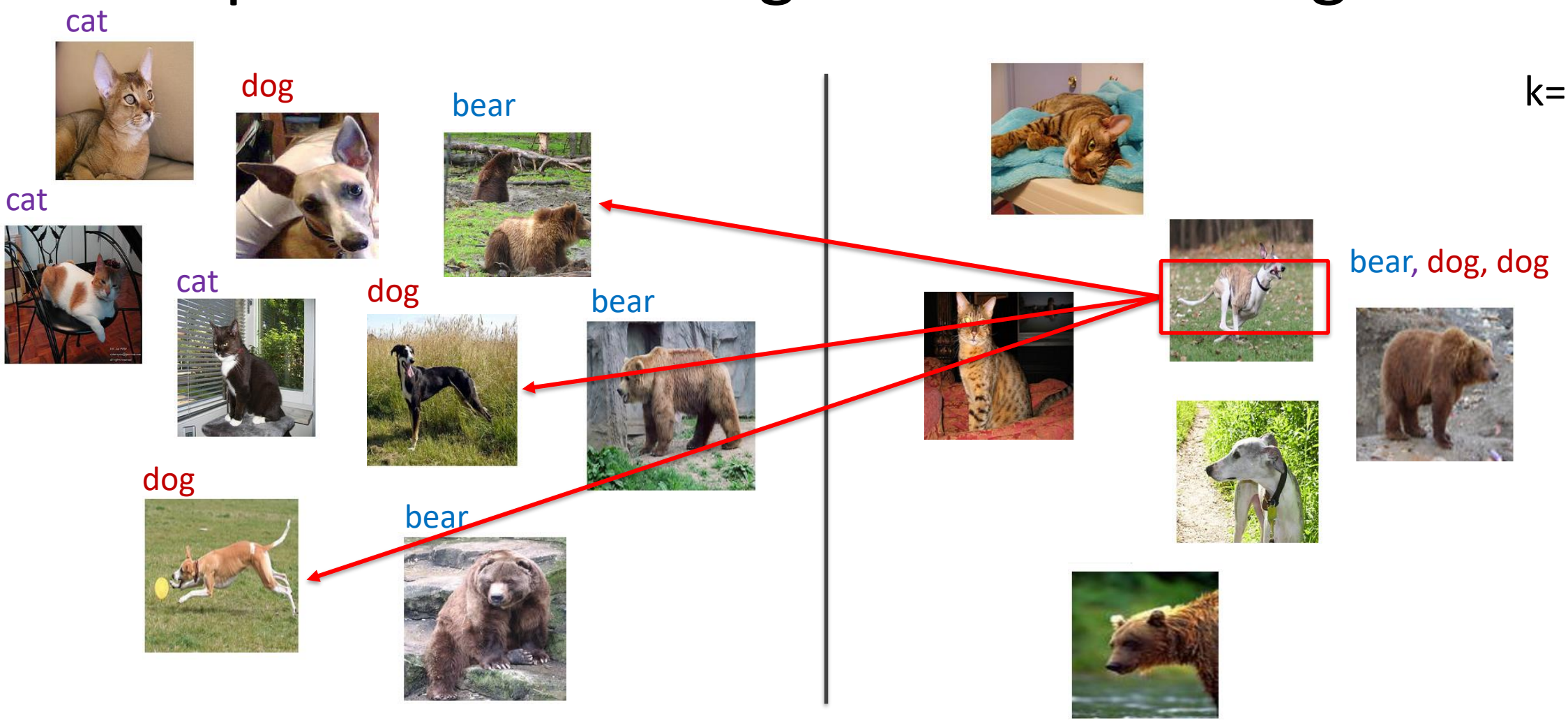



$x$

# Supervised Learning – k-Nearest Neighbors



# Supervised Learning – k-Nearest Neighbors



# Supervised Learning – k-Nearest Neighbors

- How do we choose the right K?
- How do we choose the right features?
- How do we choose the right distance metric?

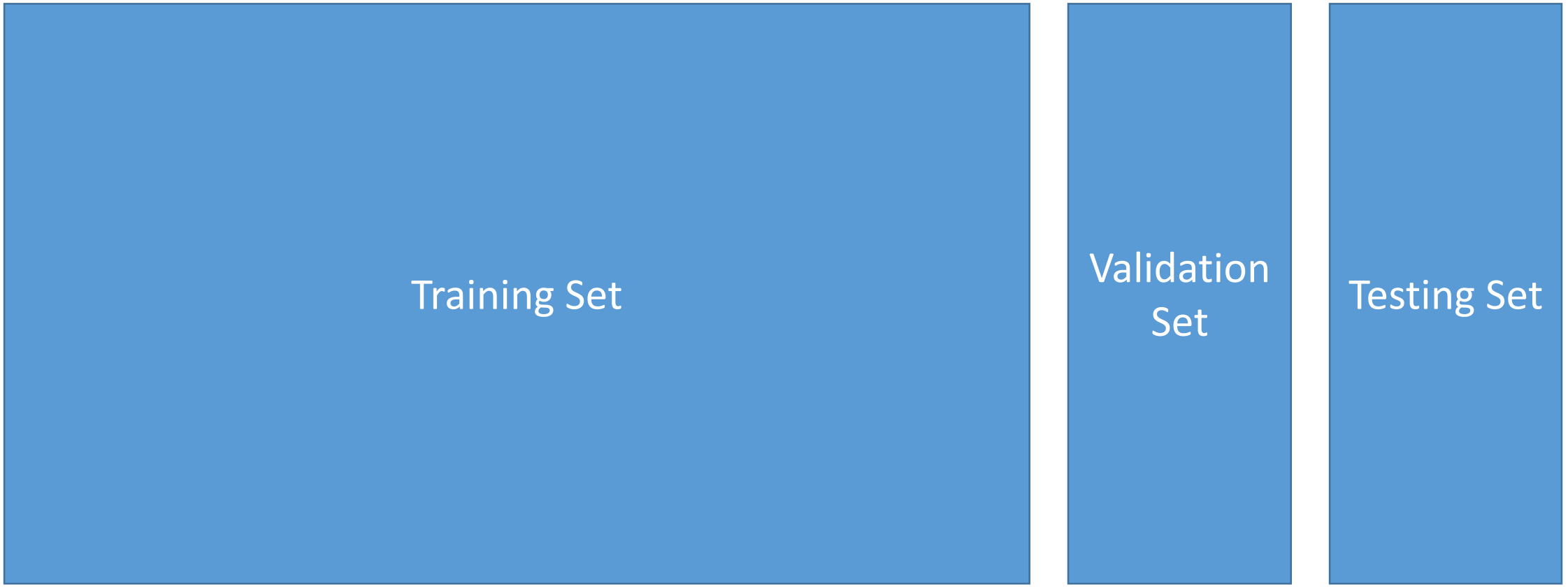
# Supervised Learning – k-Nearest Neighbors

- How do we choose the right K?
- How do we choose the right features?
- How do we choose the right distance metric?

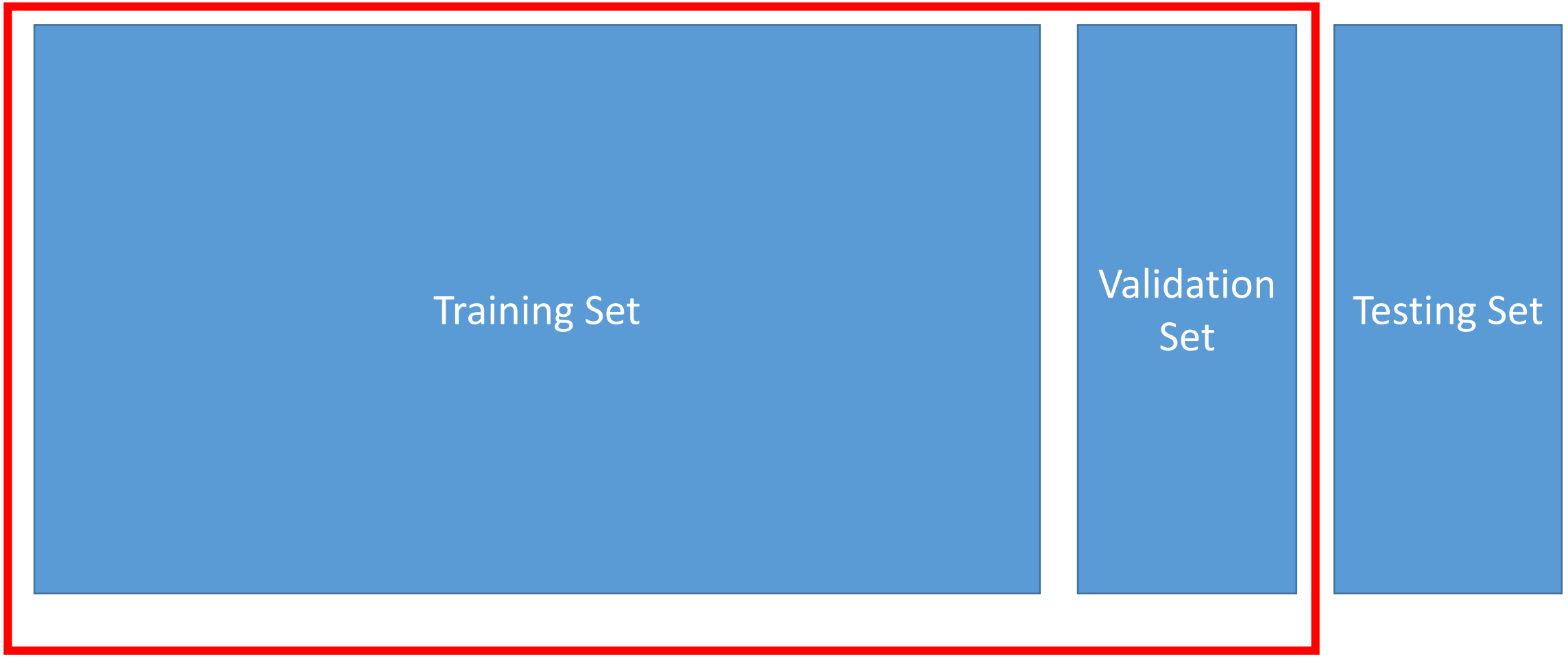
Answer: Just choose the one combination that works best!
**BUT** not on the test data.

Instead split the training data into a ”Training set” and a ”Validation set” (also called ”Development set”)

# Training, Validation (Dev), Test Sets

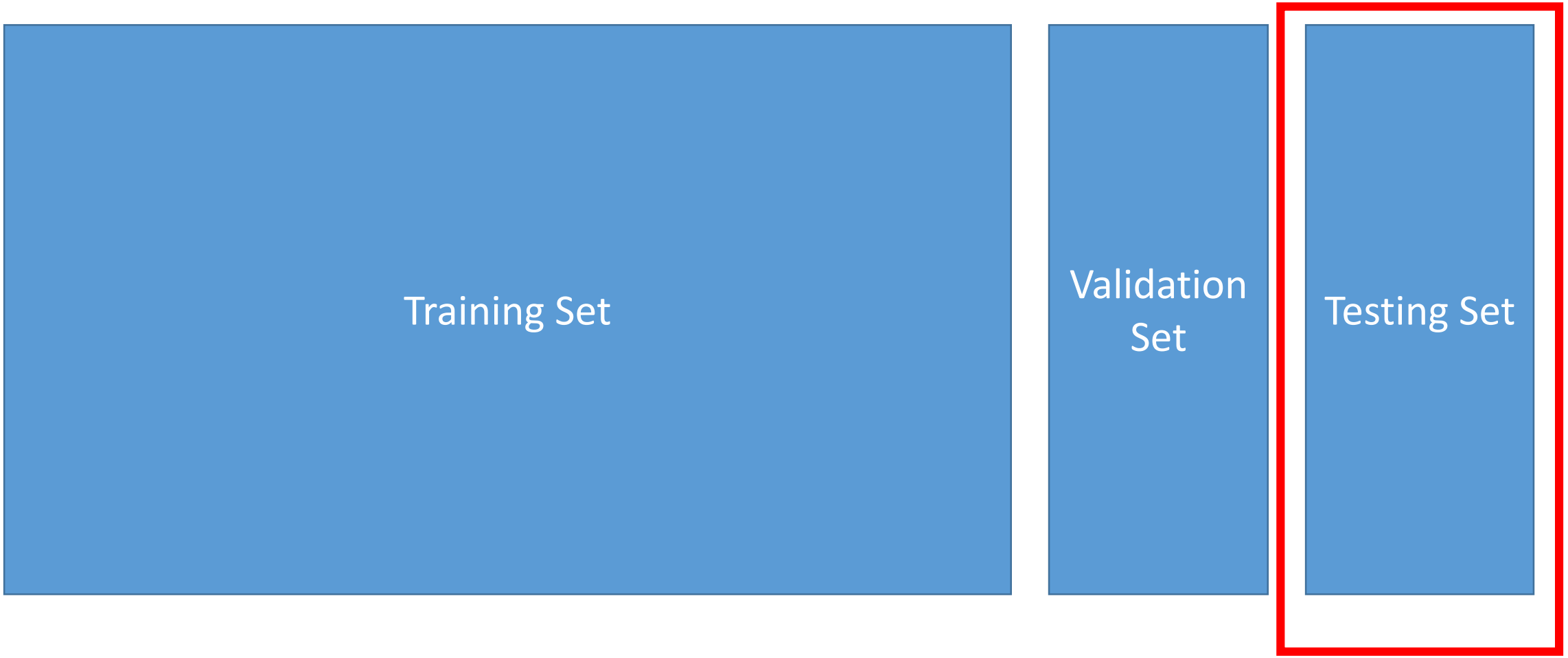


# Training, Validation (Dev), Test Sets



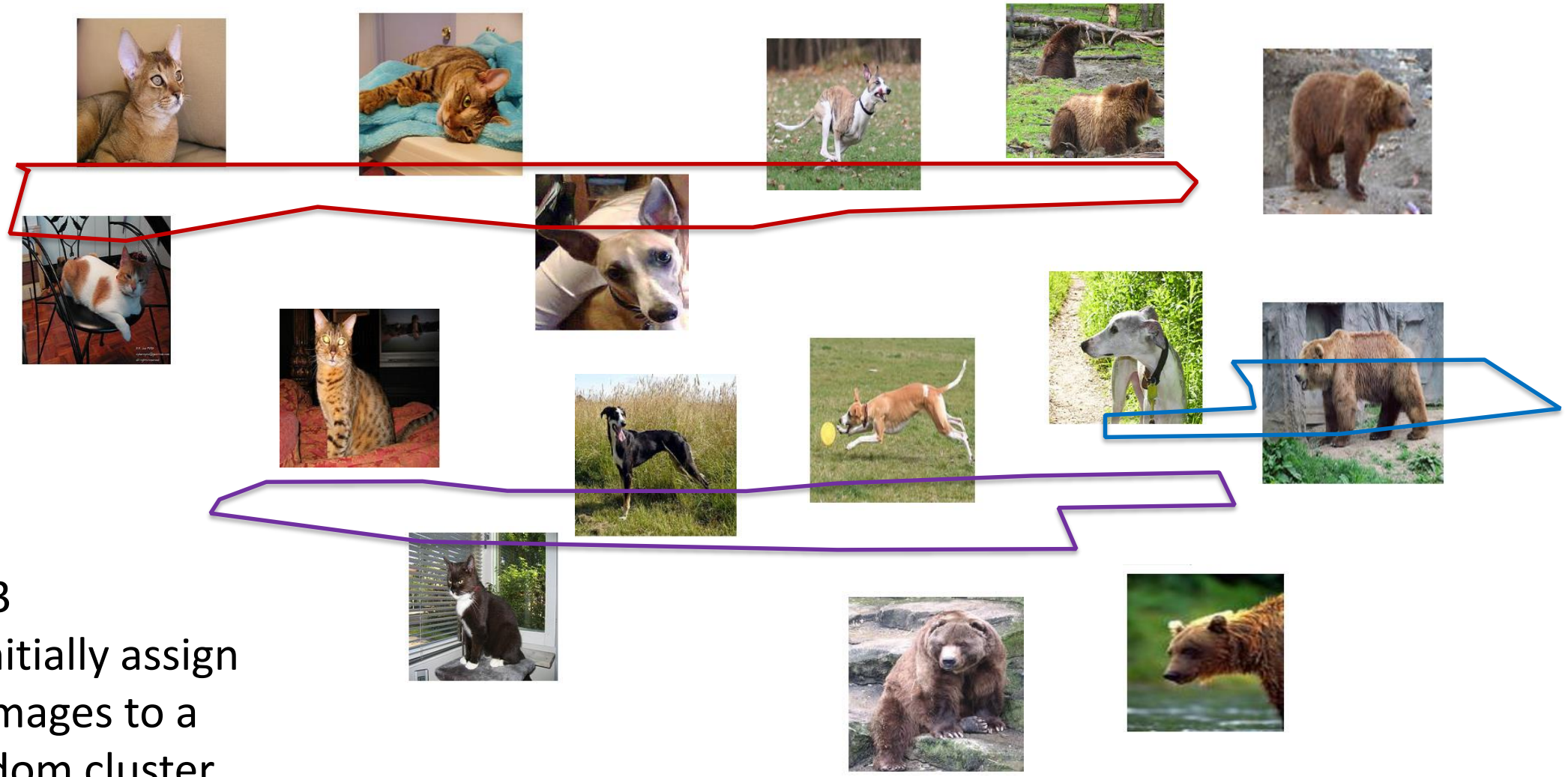

Used during development

# Training, Validation (Dev), Test Sets




Only to be used for evaluating the model at the very end of development and any changes to the model after running it on the test set, could be influenced by what you saw happened on the test set, which would invalidate any future evaluation.

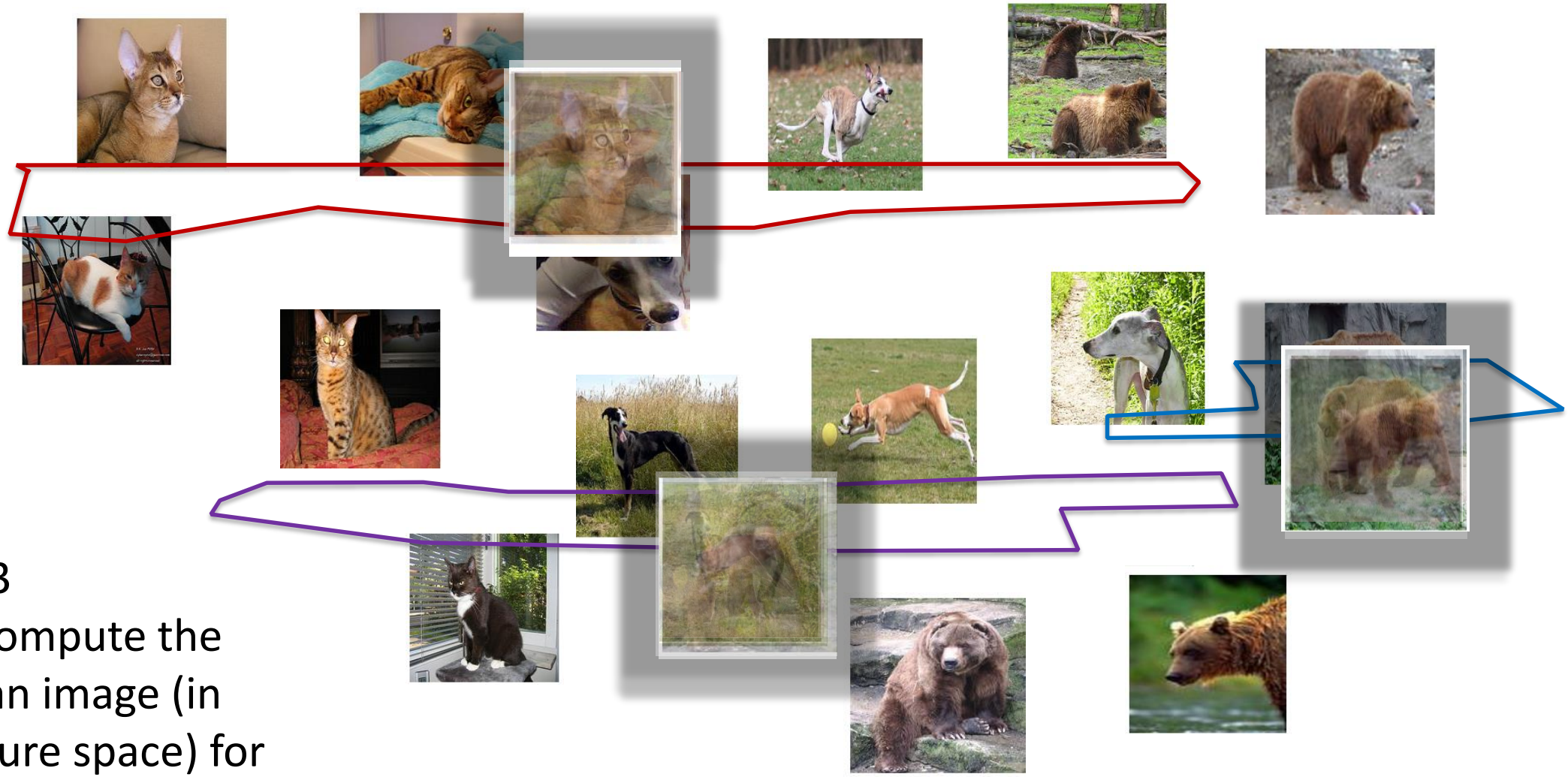
# Unsupervised Learning – k-means clustering



k = 3

1. Initially assign all images to a random cluster

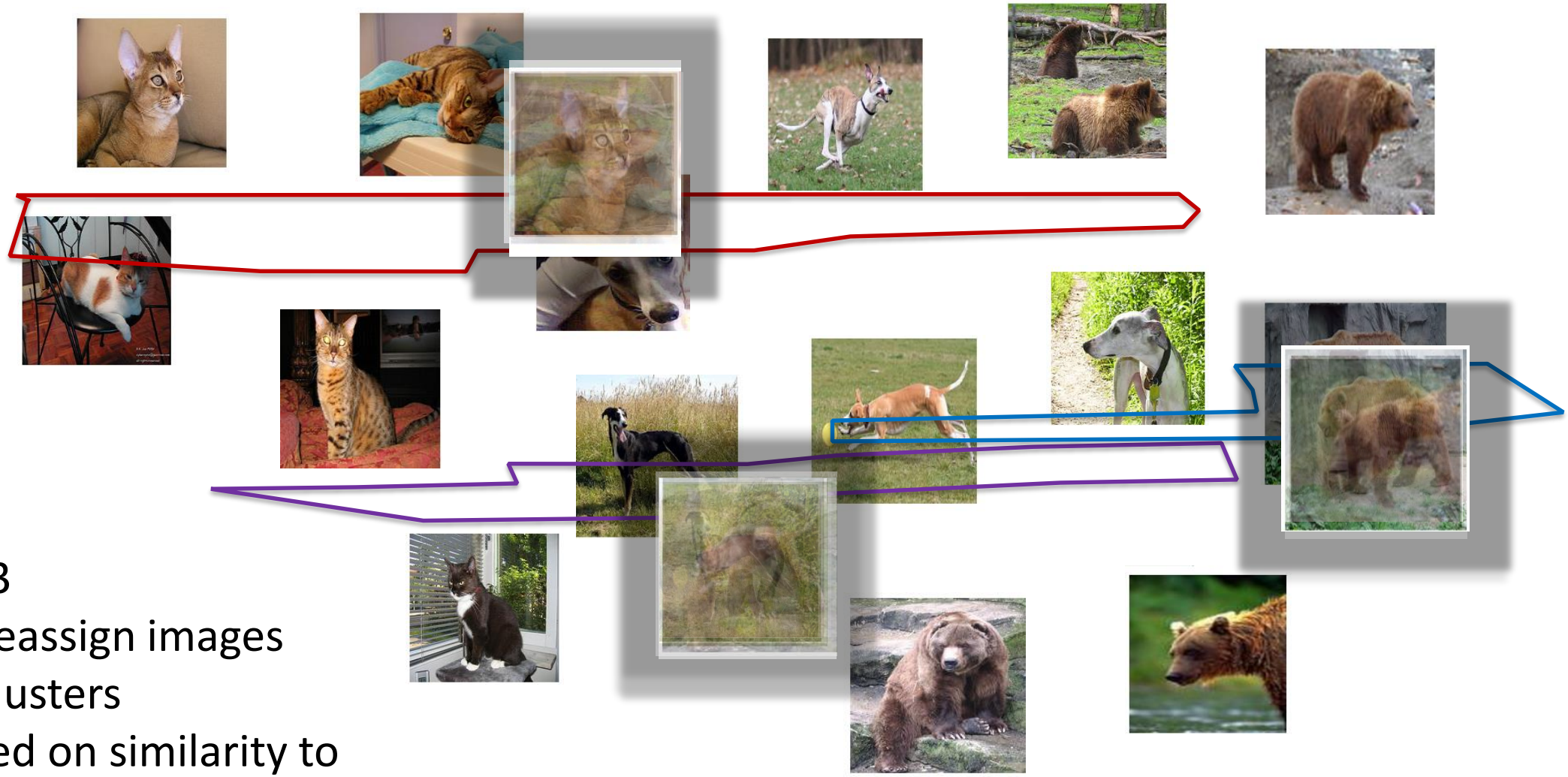
# Unsupervised Learning – k-means clustering



k = 3

2. Compute the mean image (in feature space) for each cluster

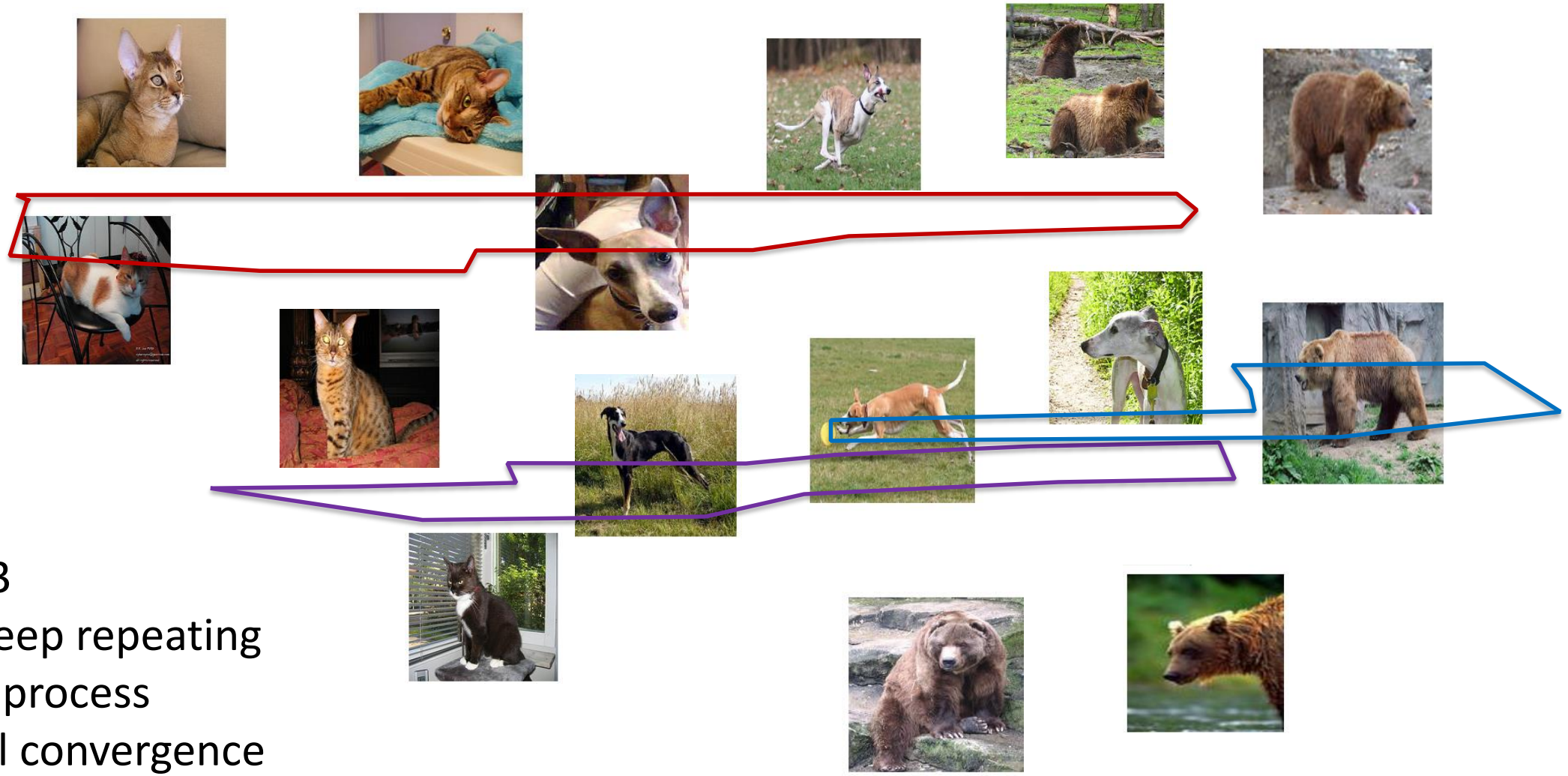
# Unsupervised Learning – k-means clustering



k = 3

3. Reassign images to clusters based on similarity to cluster means

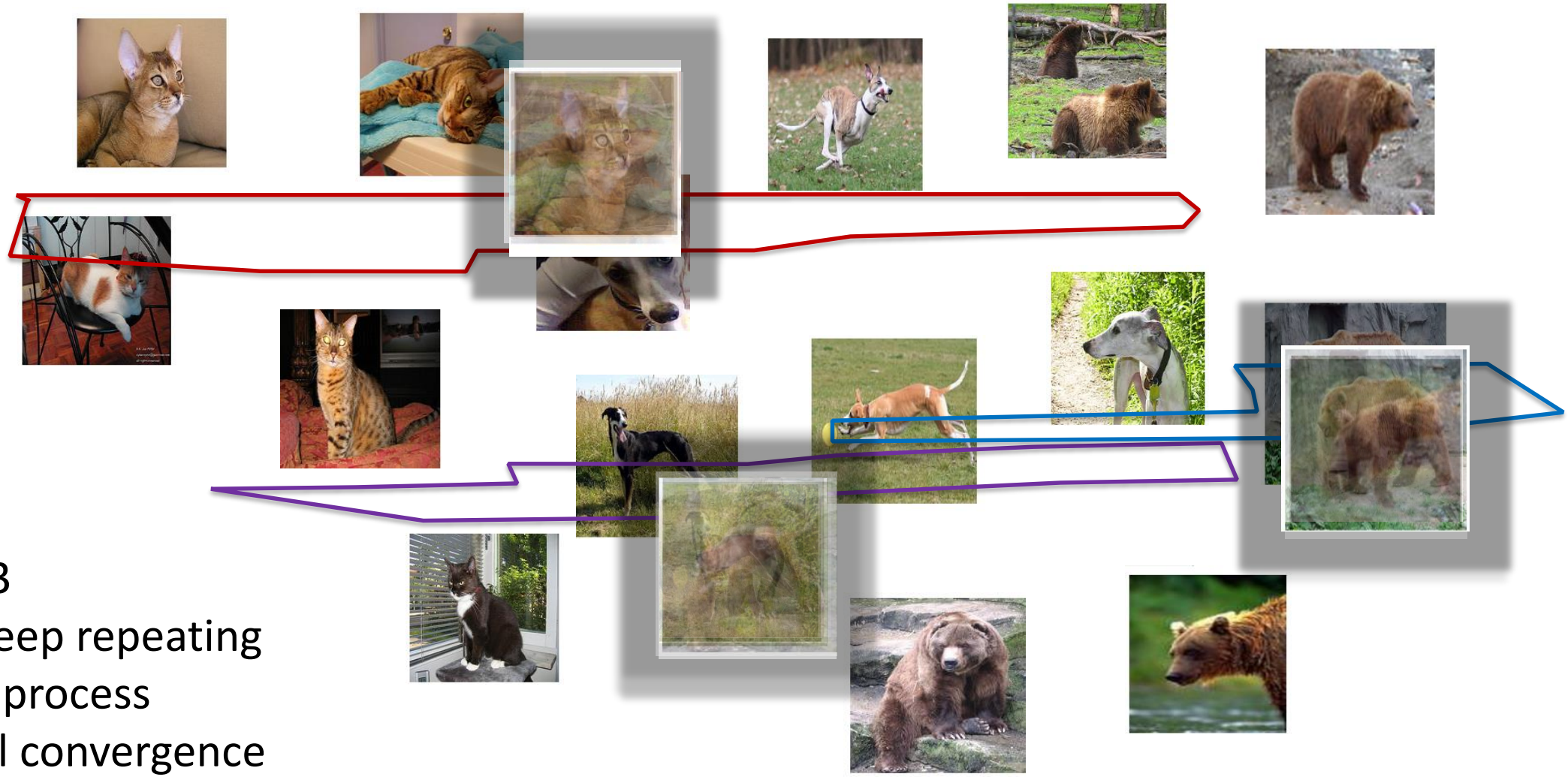
# Unsupervised Learning – k-means clustering



k = 3

4. Keep repeating this process until convergence

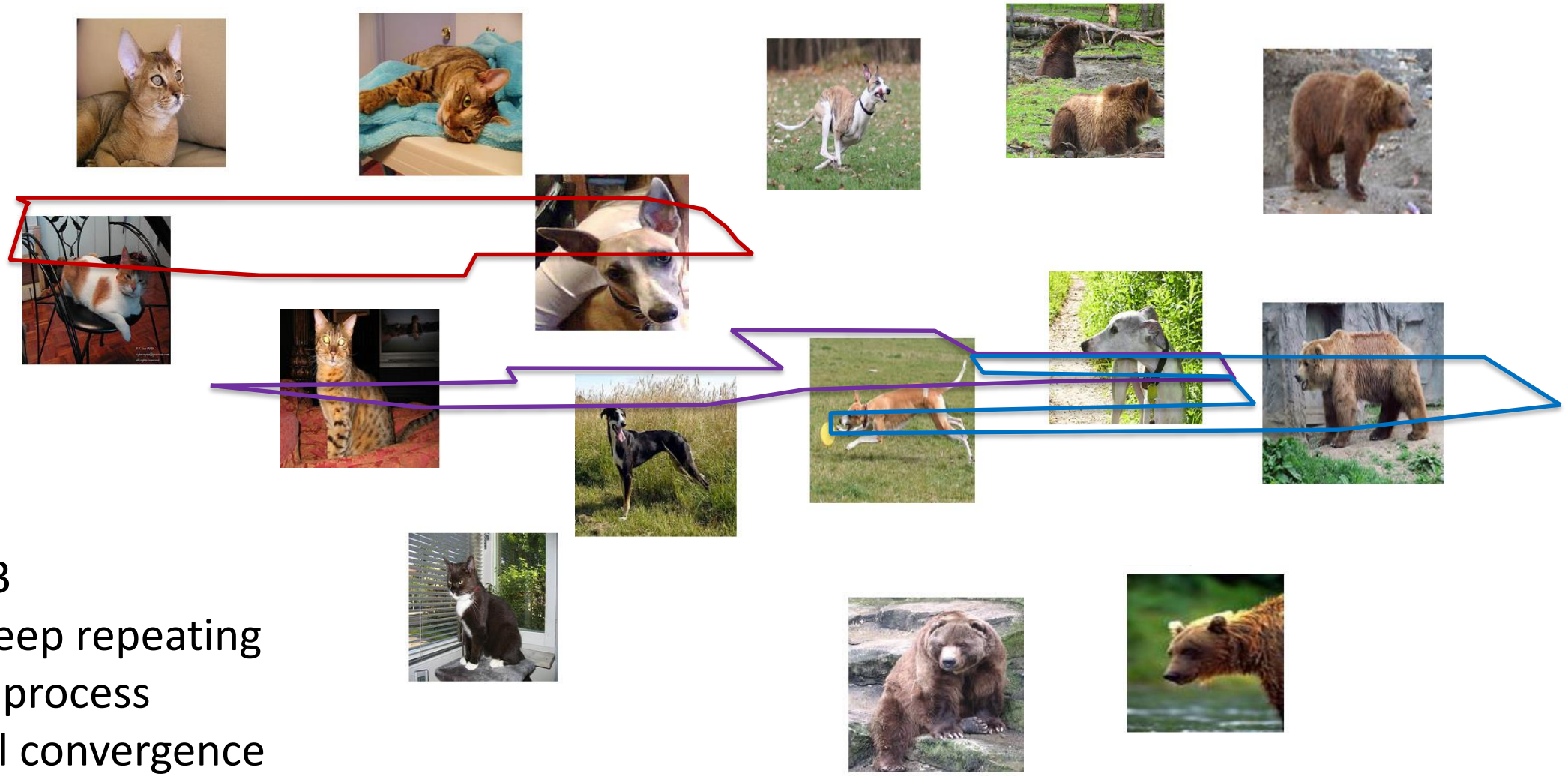
# Unsupervised Learning – k-means clustering



k = 3

4. Keep repeating this process until convergence

# Unsupervised Learning – k-means clustering



k = 3

4. Keep repeating this process until convergence

# Unsupervised Learning – k-means clustering

- How do we choose the right K?
- How do we choose the right features?
- How do we choose the right distance metric?
- How sensitive is this method with respect to the random assignment of clusters?

# Supervised Learning - Classification

## Training Data

cat

dog

bear

cat

cat

dog

bear

dog

bear

## Test Data

# Supervised Learning - Classification

Training Data

cat

dog

cat

.
.
.

bear

Test Data

.
.
.

# Supervised Learning - Classification

Training Data

$x_1 = [$  $]$ $y_1 = [\text{cat}\ ]$

$x_2 = [$  $]$ $y_2 = [\text{dog}\ ]$

$x_3 = [$  $]$ $y_3 = [\text{cat}\ ]$

.
.
.

$x_n = [$  $]$ $y_n = [\text{bear}\ ]$

# Supervised Learning - Classification

Training Data

inputs

targets / labels / ground truth

predictions

$x_1 = [x_{11} \quad x_{12} \quad x_{13} \quad x_{14}]$ $y_1 = 1$ $\hat{y}_1 = 1$

$x_2 = [x_{21} \quad x_{22} \quad x_{23} \quad x_{24}]$ $y_2 = 2$ $\hat{y}_2 = 2$

$x_3 = [x_{31} \quad x_{32} \quad x_{33} \quad x_{34}]$ $y_3 = 1$ $\hat{y}_3 = 2$

.
.
.

$x_n = [x_{n1} \quad x_{n2} \quad x_{n3} \quad x_{n4}]$ $y_n = 3$ $\hat{y}_n = 1$

We need to find a function that maps *x* and *y* for any of them.

$$\hat{y}_i = f(x_i; \theta)$$

How do we ”learn” the parameters of this function?

We choose ones that makes the following quantity small:

$$\sum_{i=1}^{n} Cost(\hat{y}_i, y_i)$$

# Supervised Learning – Linear Softmax

Training Data

inputs

targets / labels / ground truth

$x_1 = [x_{11} \quad x_{12} \quad x_{13} \quad x_{14}]$ $\quad y_1 = 1$

$x_2 = [x_{21} \quad x_{22} \quad x_{23} \quad x_{24}]$ $\quad y_2 = 2$

$x_3 = [x_{31} \quad x_{32} \quad x_{33} \quad x_{34}]$ $\quad y_3 = 1$

.
.
.

$x_n = [x_{n1} \quad x_{n2} \quad x_{n3} \quad x_{n4}]$ $\quad y_n = 3$

# Supervised Learning – Linear Softmax

Training Data

inputs

targets / labels / ground truth

predictions

$x_1 = [x_{11} \quad x_{12} \quad x_{13} \quad x_{14}]$ $y_1 = [1 \quad 0 \quad 0]$ $\hat{y}_1 = [0.85 \quad 0.10 \quad 0.05]$

$x_2 = [x_{21} \quad x_{22} \quad x_{23} \quad x_{24}]$ $y_2 = [0 \quad 1 \quad 0]$ $\hat{y}_2 = [0.20 \quad 0.70 \quad 0.10]$

$x_3 = [x_{31} \quad x_{32} \quad x_{33} \quad x_{34}]$ $y_3 = [1 \quad 0 \quad 0]$ $\hat{y}_3 = [0.40 \quad 0.45 \quad 0.15]$

.
.
.

$x_n = [x_{n1} \quad x_{n2} \quad x_{n3} \quad x_{n4}]$ $y_n = [0 \quad 0 \quad 1]$ $\hat{y}_n = [0.40 \quad 0.25 \quad 0.35]$

# Supervised Learning – Linear Softmax

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

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

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

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

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

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

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

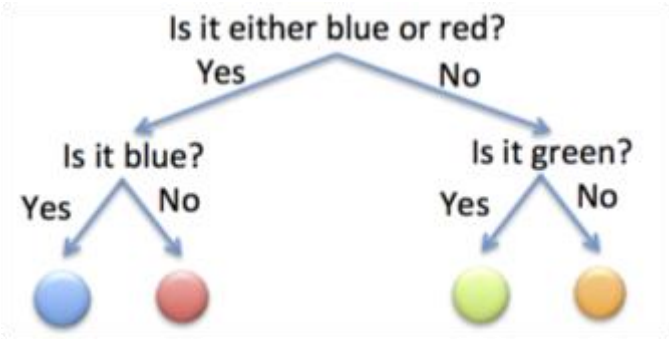
# How do we find a good w and b?

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

We need to find w, and b that minimize the following:

$$L(w,b) = \sum_{i=1}^{n}\sum_{j=1}^{3} -y_{i,j}\log(\hat{y}_{i,j}) \quad = \sum_{i=1}^{n} -\log(\hat{y}_{i,label}) \quad = \sum_{i=1}^{n} -\log f_{i,label}(w,b)$$

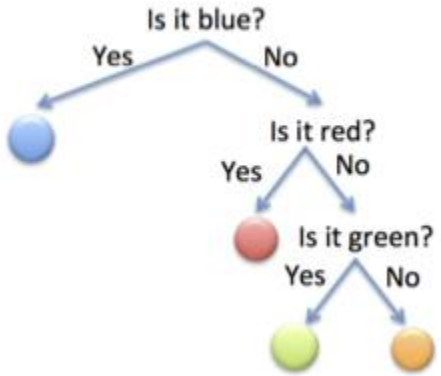
# Idea of entropy and cross-entropy




How many questions are to be
Asked to right guess the color
Of a randomly picked ball?

# Another case

Now, I will draw a coin from a bag of coins: 1/2 of them are blue, 1/4 are red, 1/8 are green, and 1/8 are orange. The previous strategy no longer is the best; because there is a fair chance to draw a blue coin, we should prioritize guessing the most likely outcome. Your optimal strategy now looks like this:



# Gradient Descent (GD)

$$L(w,b) = \sum_{i=1}^{n} -\log f_{i,label}(w,b)$$

expensive

$\lambda = 0.01$

Initialize w and b randomly

**for** e = 0, num_epochs **do**

Compute: $dL(w,b)/dw$ and $dL(w,b)/db$

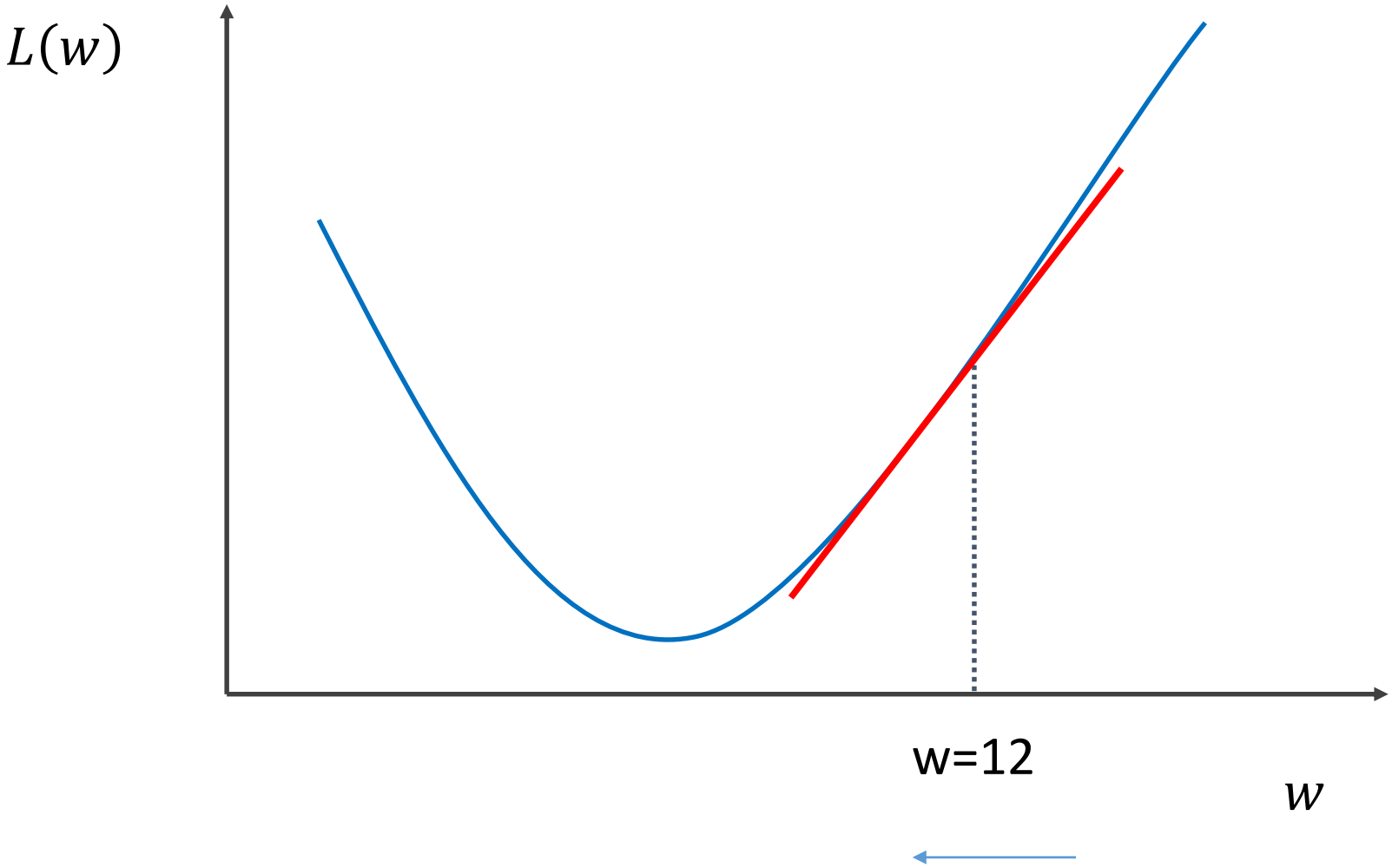
Update w: $w = w - \lambda\, dL(w,b)/dw$

Update b: $b = b - \lambda\, dL(w,b)/db$

Print: $L(w,b)$ // Useful to see if this is becoming smaller or not.

**end**

# Gradient Descent (GD) (idea)



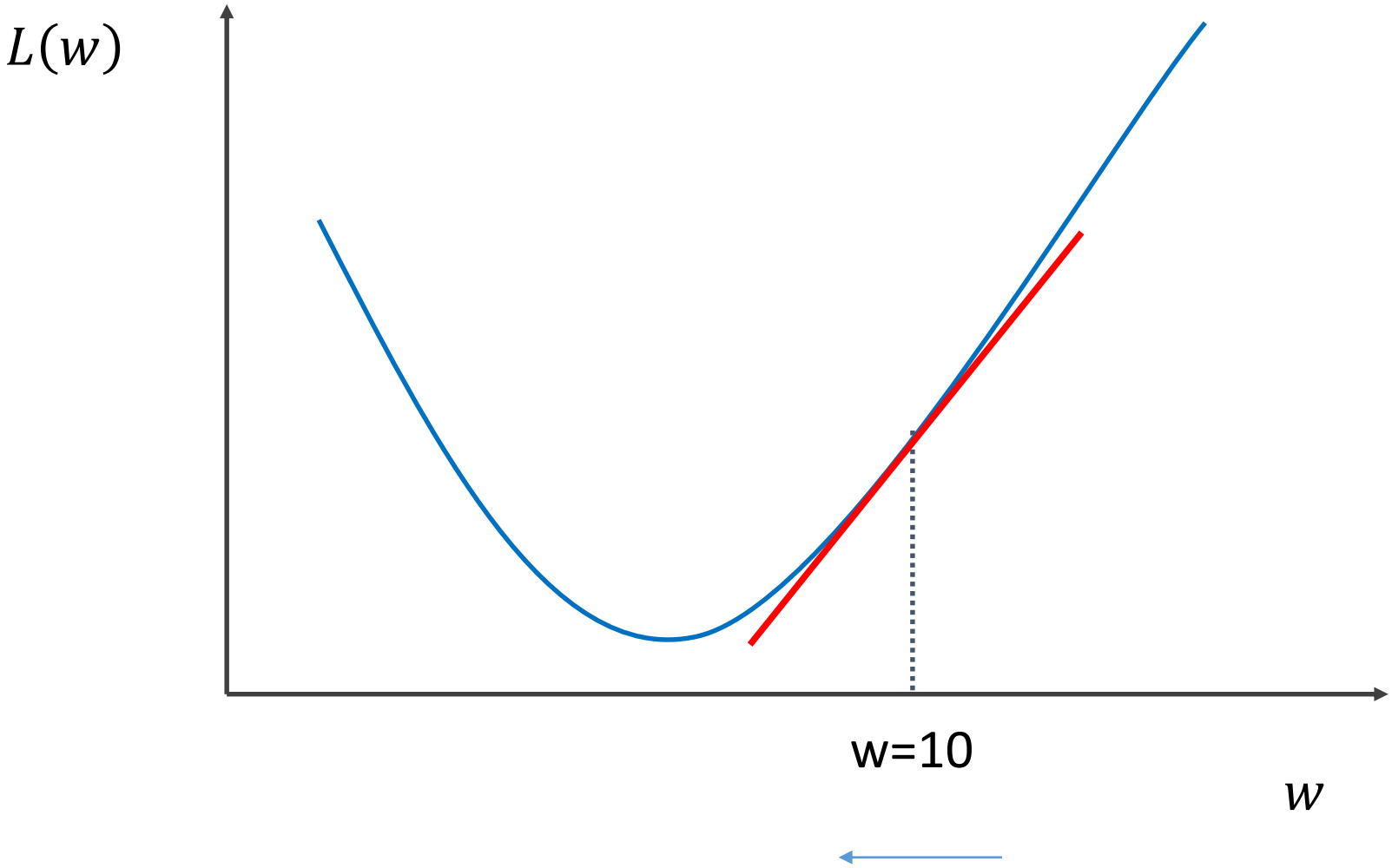

1. Start with a random value of w (e.g. w = 12)

2. Compute the gradient (derivative) of L(w) at point w = 12. (e.g. dL/dw = 6)

3. Recompute w as:

w = w – lambda * (dL / dw)

# Gradient Descent (GD) (idea)




2. Compute the gradient (derivative) of L(w) at point w = 12. (e.g. dL/dw = 6)

3. Recompute w as:

w = w – lambda * (dL / dw)

# (mini-batch) Stochastic Gradient Descent (SGD)

$\lambda = 0.01$

Initialize w and b randomly

$$l(w,b) = \sum_{i \in B} -\log f_{i,label}(w,b)$$

**for** e = 0, num_epochs **do**

**for** b = 0, num_batches **do**

Compute: $dl(w,b)/dw$ and $dl(w,b)/db$

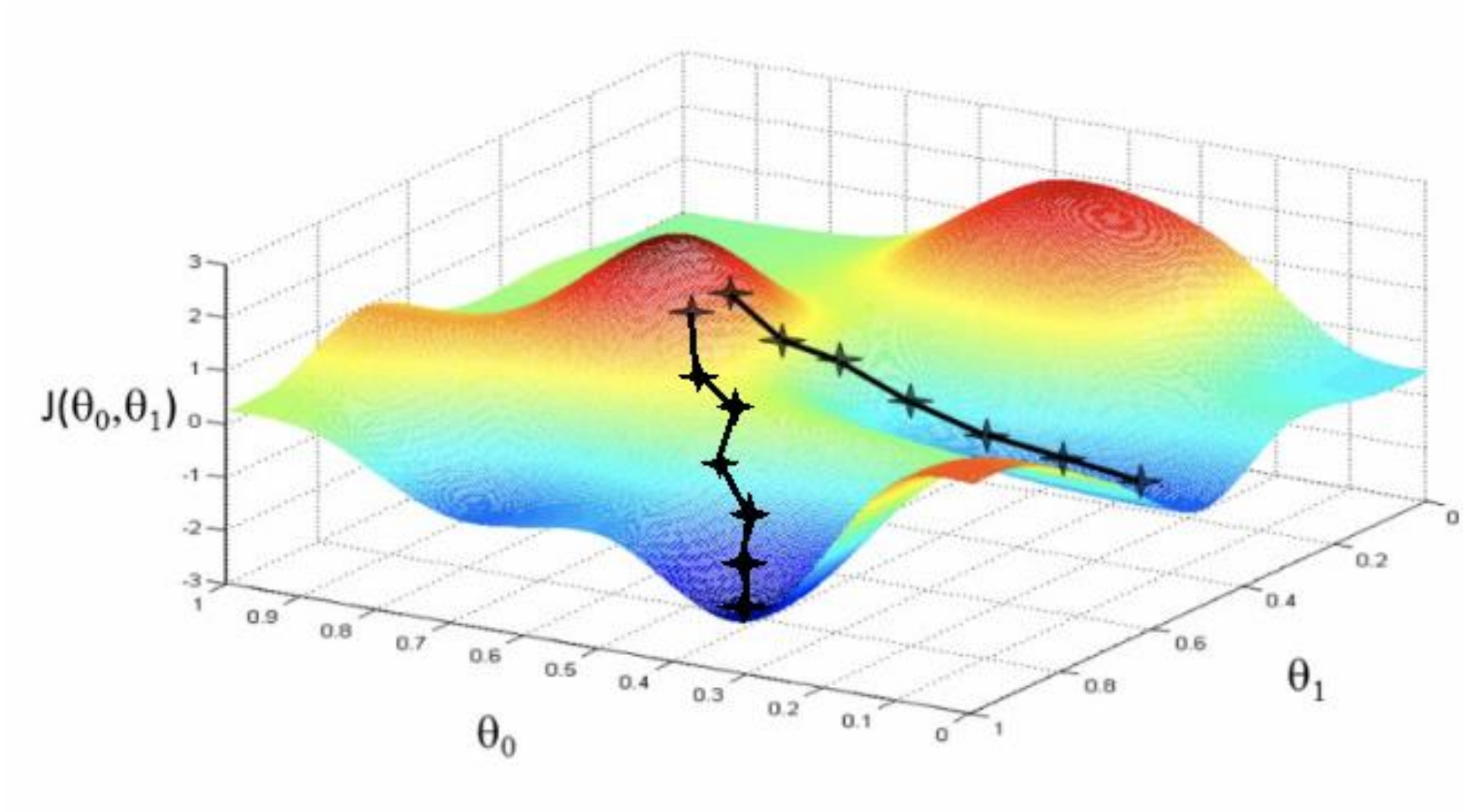
Update w: $w = w - \lambda\, dl(w,b)/dw$

Update b: $b = b - \lambda\, dl(w,b)/db$

Print: $l(w,b)$ // Useful to see if this is becoming smaller or not.

**end**

**end**




Source: Andrew Ng