Computer Vision

Bogdan Alexe

bogdan.alexe@fmi.unibuc.ro

University of Bucharest, 2nd semester, 2023-2024

Course structure

1. **Features and filters: low-level vision**

   Linear filters, color, texture, edge detection

2. **Grouping and fitting: mid-level vision**

   Fitting curves and lines, robust fitting, RANSAC, Hough transform, segmentation

3. **Multiple views**

   Local invariant feature and description, epipolar geometry and stereo, object instance recognition

4. **Object Recognition: high – level vision**

   Object classification, object detection, part based models, bovw models

1. **Video understanding**

   Object tracking, background subtraction, motion descriptors, optical flow

Low-Level: Resizing

Low-Level: Resizing

nearest neighbor interpolation

bilinear interpolation

Low-Level: Grayscale

Low-Level: Exposure

Low-Level: Hue

Low-Level: Saturation

Low-Level: Edges

Low-Level: Oriented Gradients

Low-Level: Oriented Gradients

Low-Level: Segmentation (color)

Low-Level Vision

Photo manipulation
- Size
- Color
- Exposure

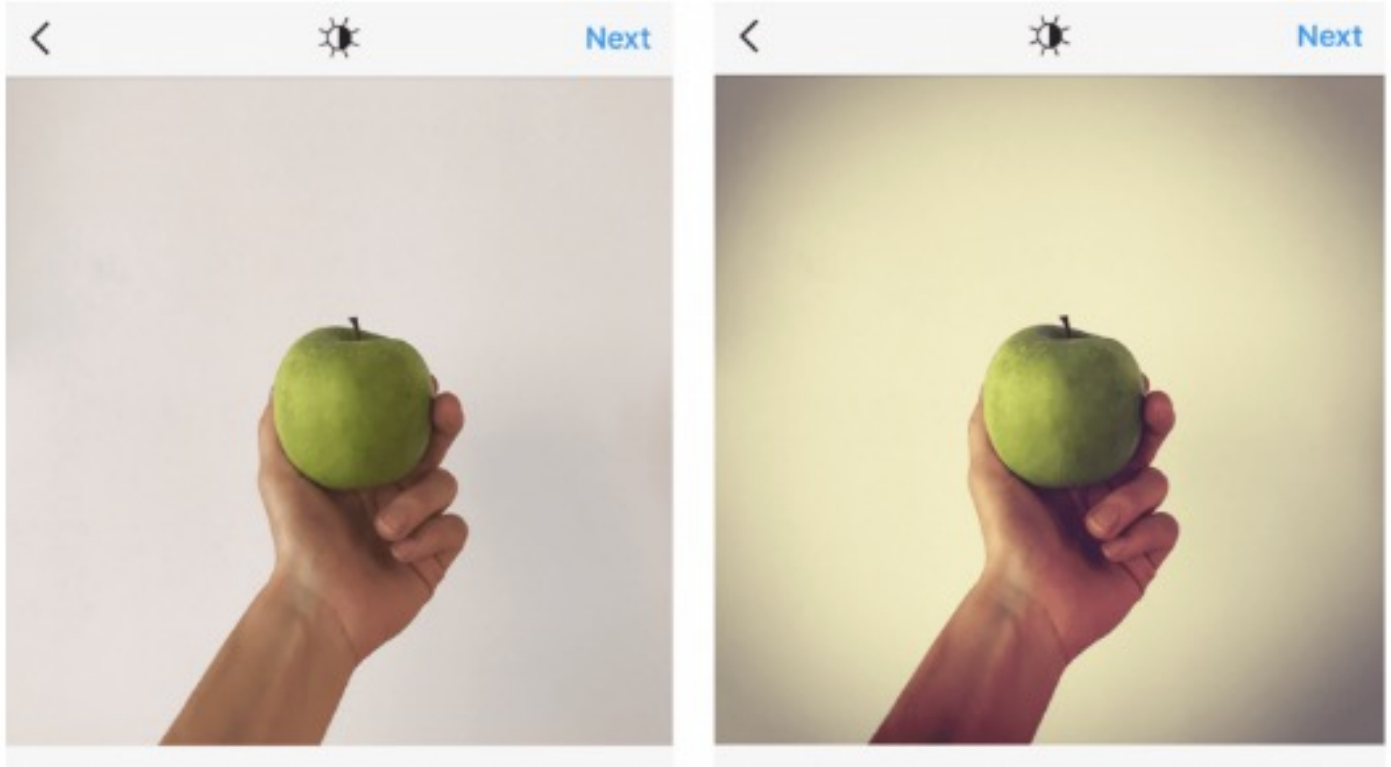
Feature extraction
- Edges
- Oriented gradients
- Segments

Low level vision is exciting!!!

Instagram filter #10: X-Pro II

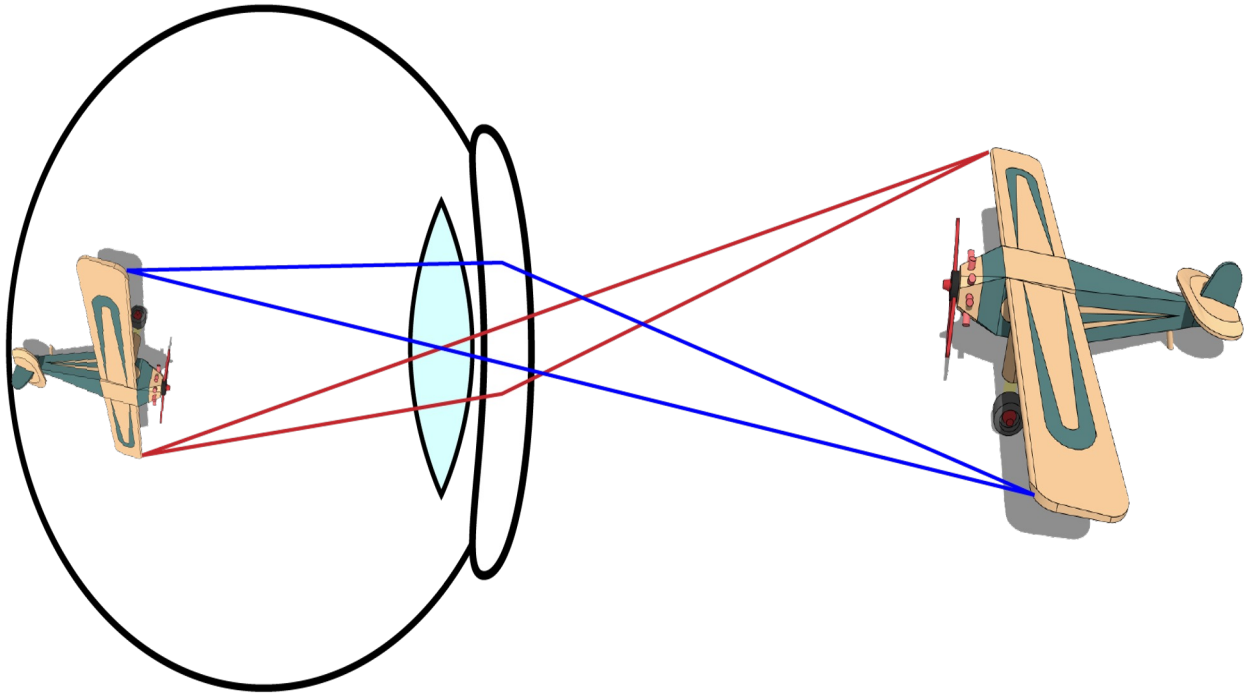
Last but not least, X-Pro II. This filter is the most high-contrast and probably the least discreet on the list: it adds a LOAD of shadow and darkness, not to mention a hefty vignette, which shades away the edges of the photo. X-PRO II is based on a photo-developing technique called “cross-processing”, where photos are processed in a chemical solution for different types of film. This is one of Instagram’s oldest filters, and was intended to be used to correct the below-average camera phones of 2010, when Instagram had just launched. Surprisingly, it is still one of the most popular filters on the platform 🤦‍♀️

Good for: Turning normal photos into very intense ones

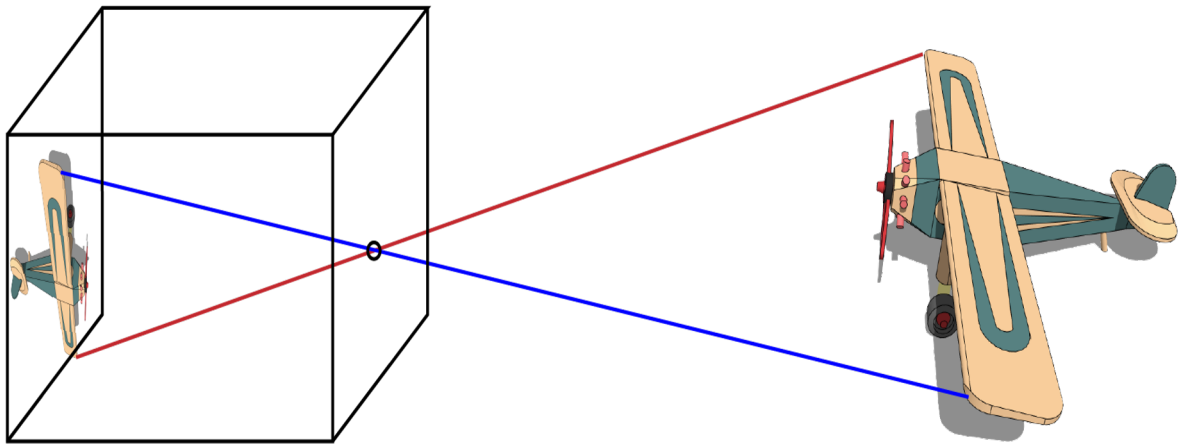


What is an image?

Eyes: projection onto retina

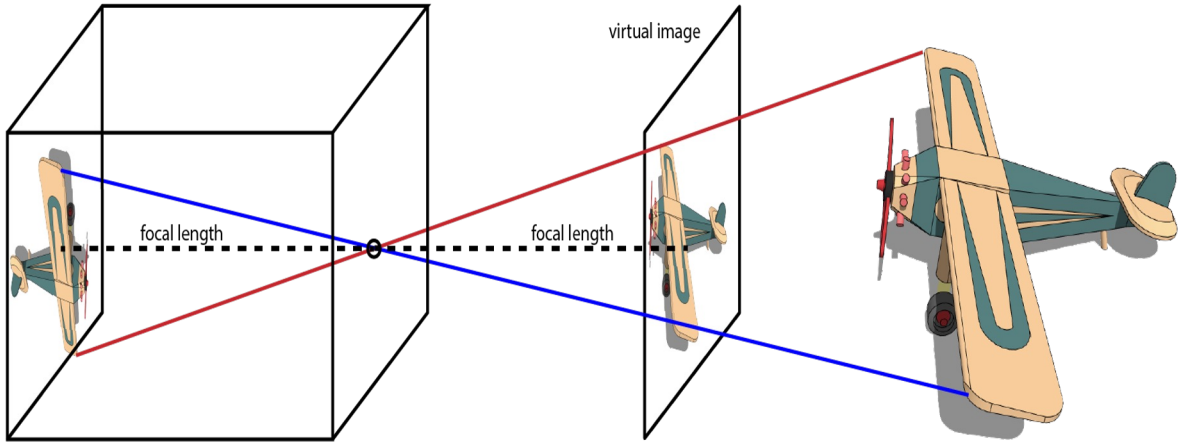


Model: pinhole camera



- camera aperture is described as a point
- no lenses are used to focus light

Model: pinhole camera



- the image plane is located at distance f (focal length) from the origin O (where the camera aperture is located)

Image: 2D projection of the 3D world

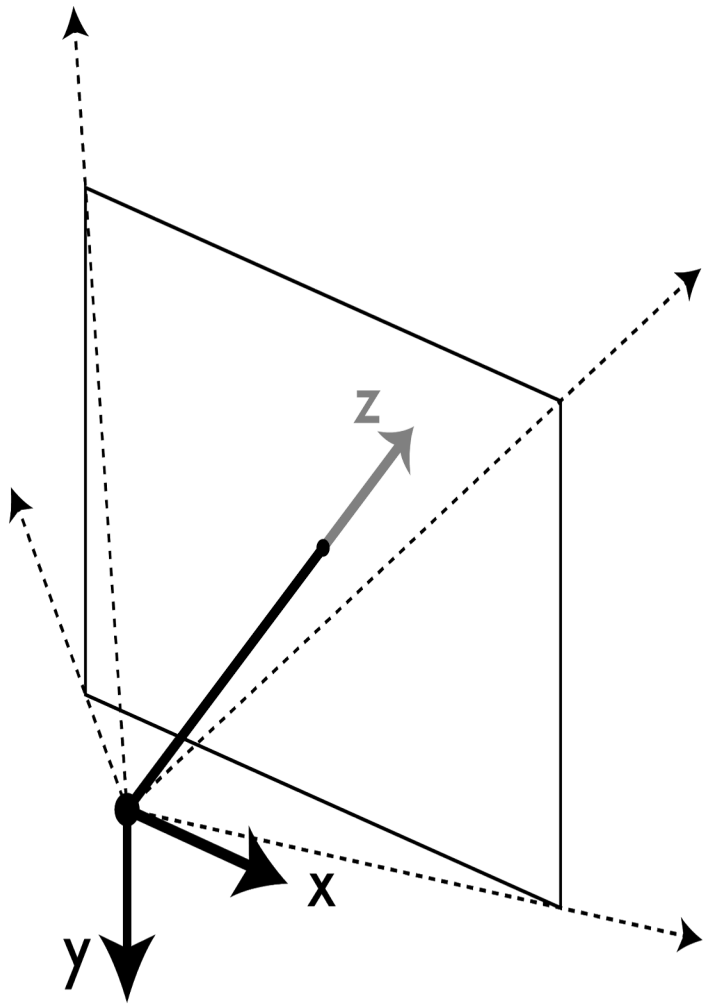


Image: 2D projection of the 3D world

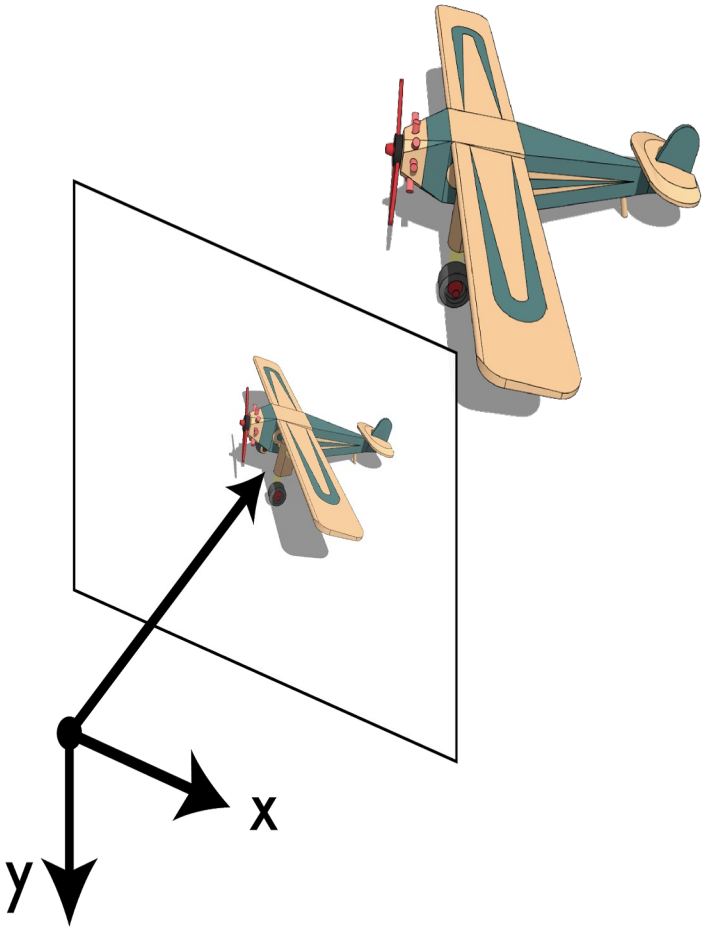


Image: 2D projection of the 3D world

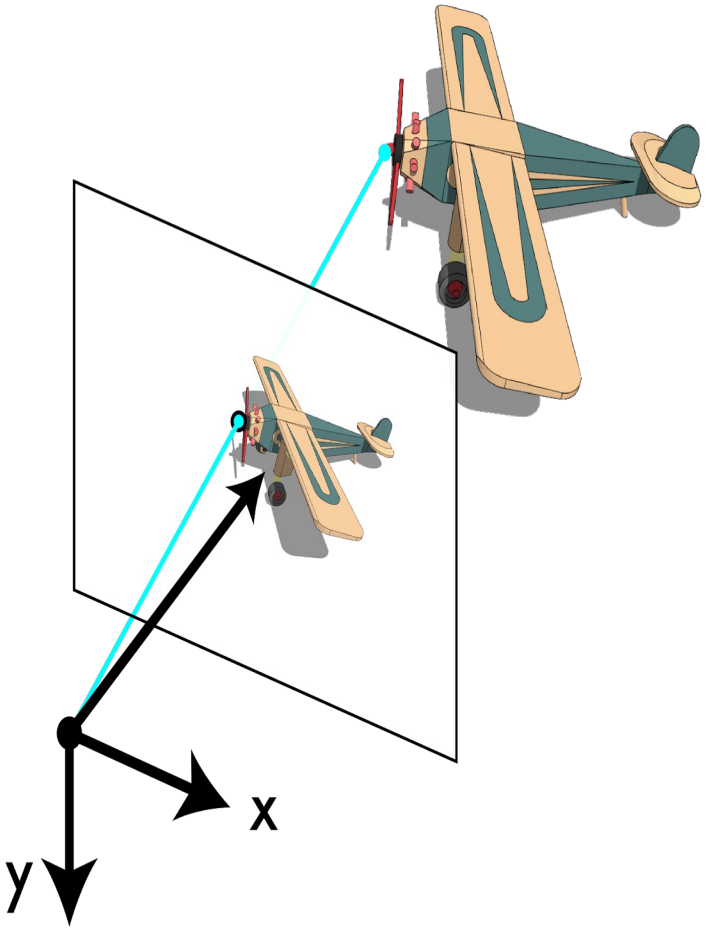


Image: 2D projection of the 3D world

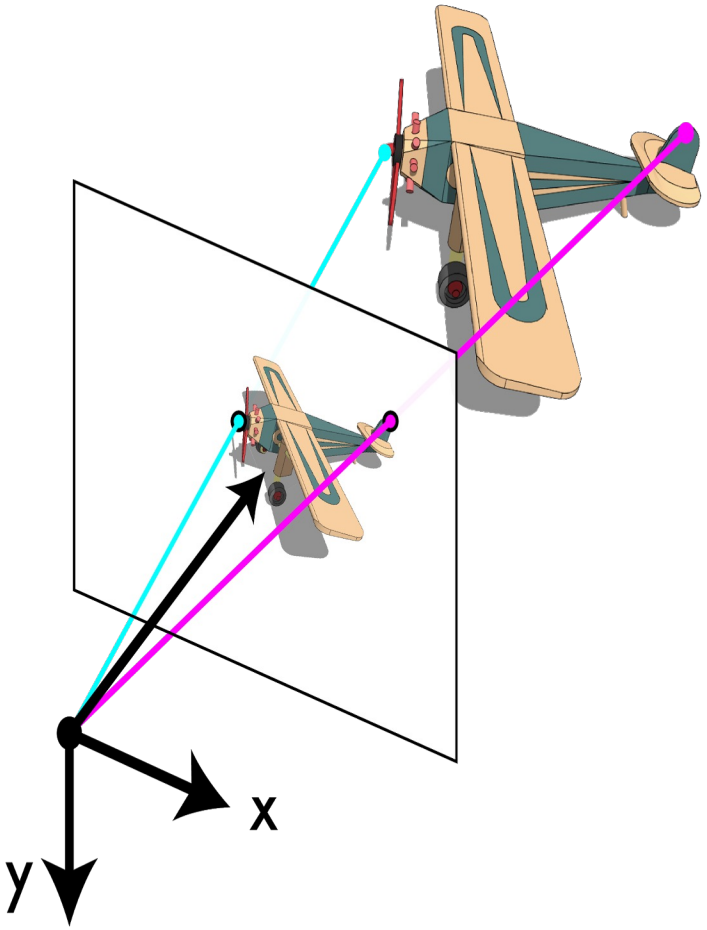


Image: 2D projection of the 3D world

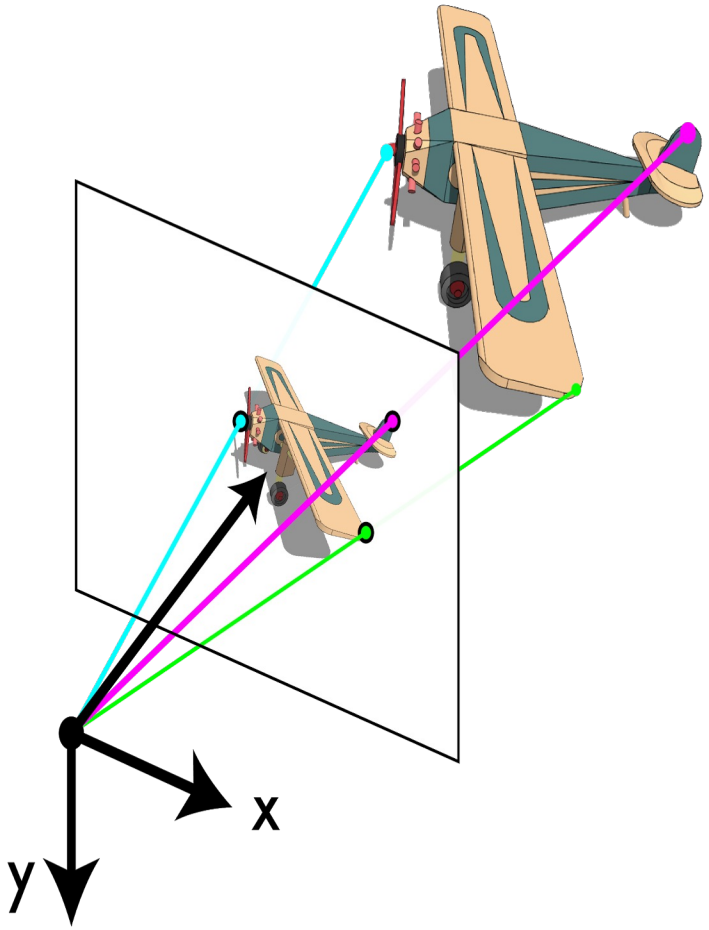
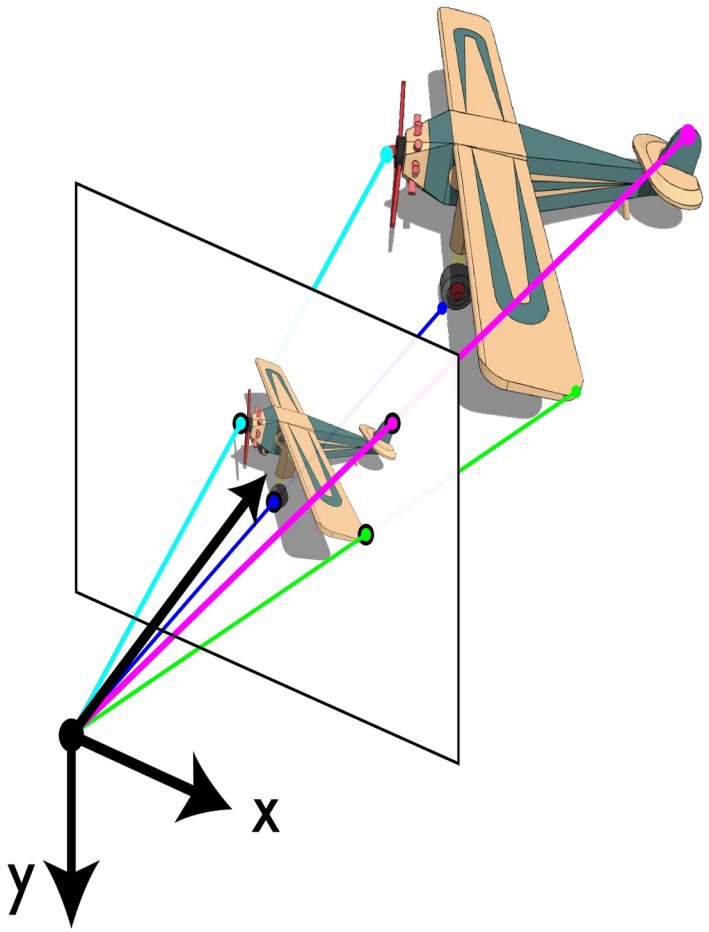
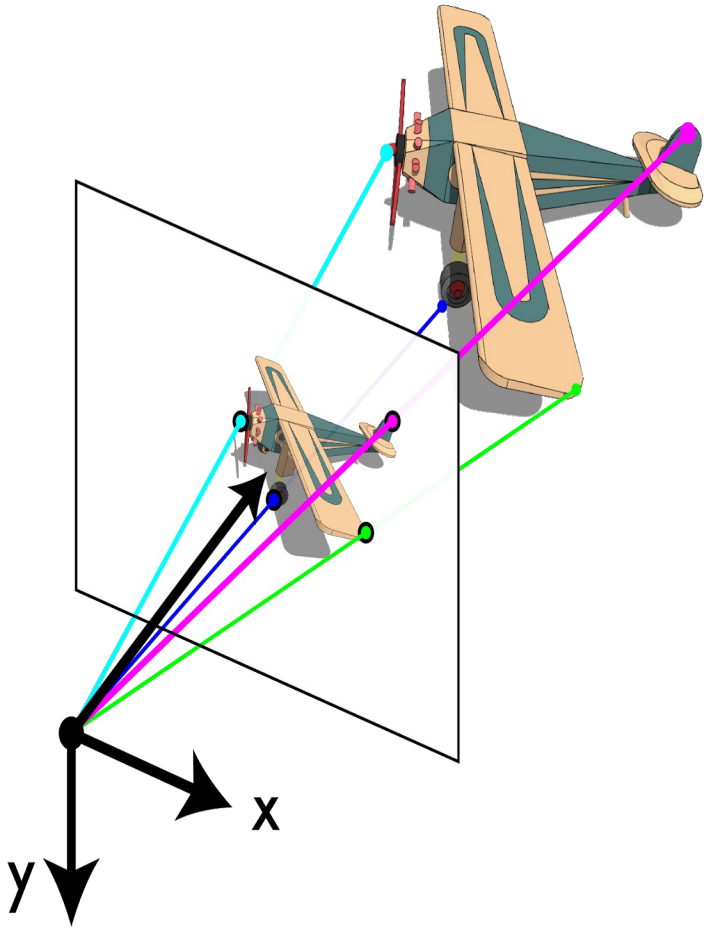


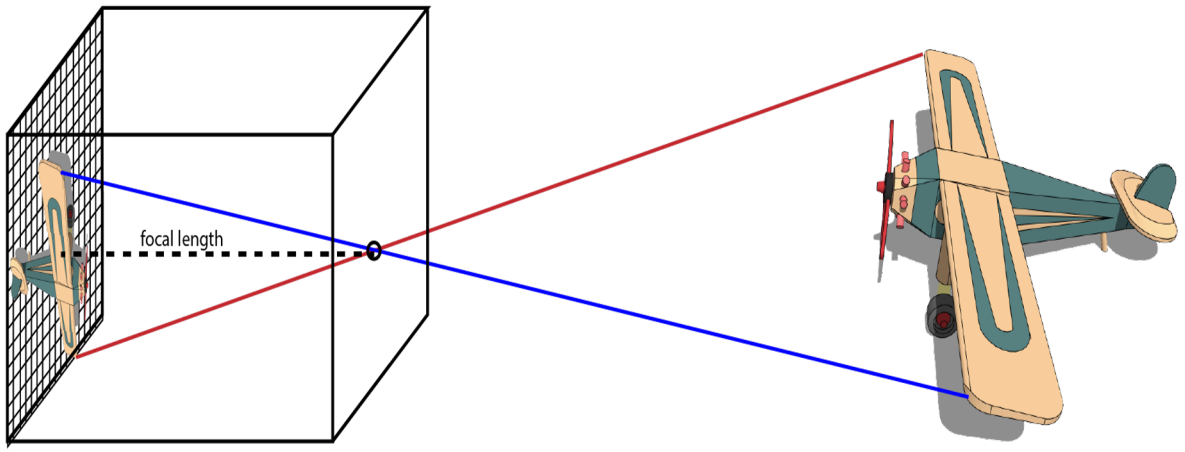
Image: 2D projection of the 3D world



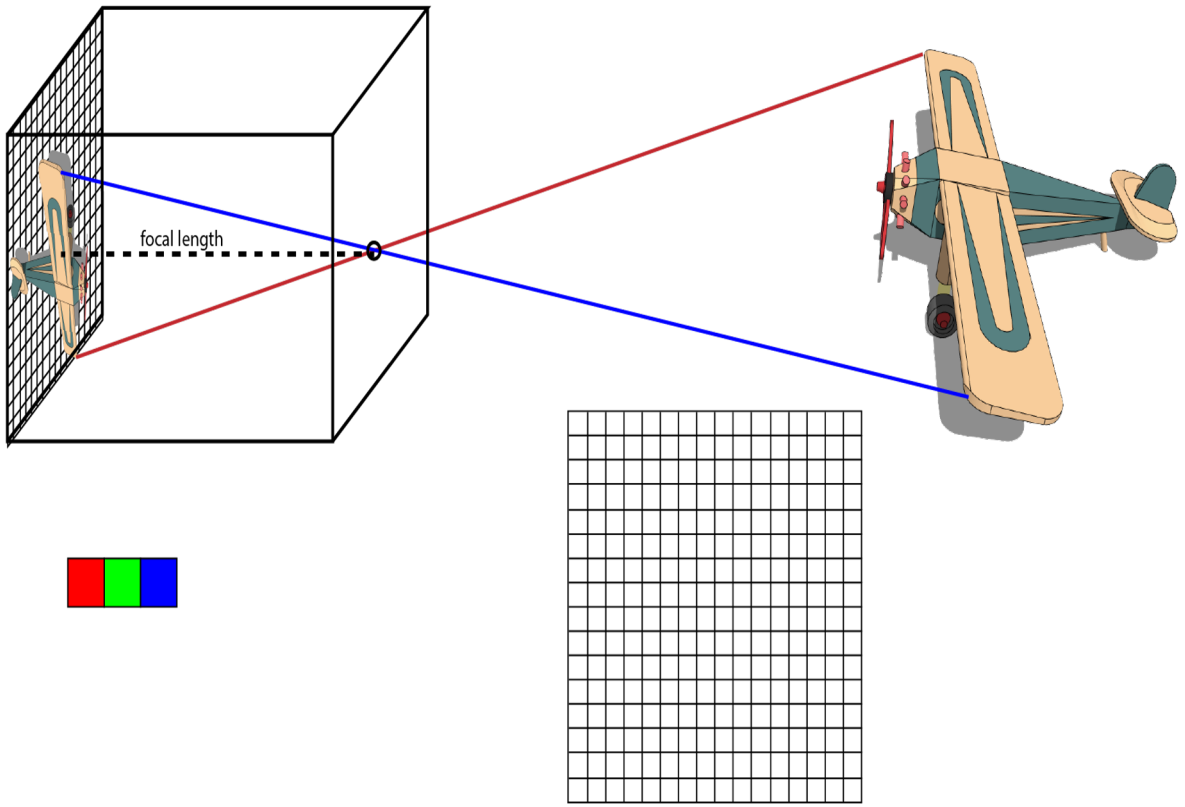
At each point we record incident light



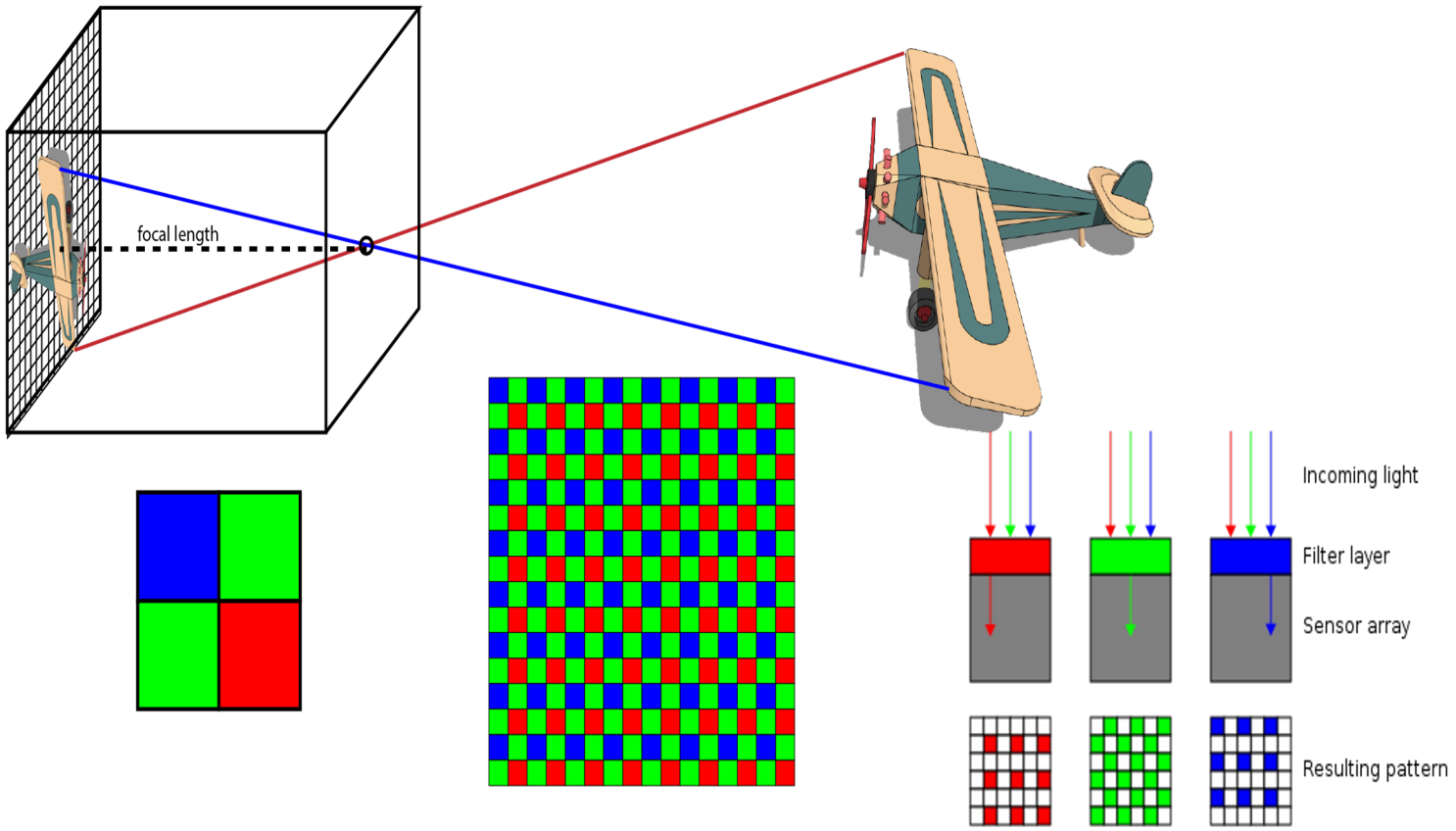
At each point we record incident light



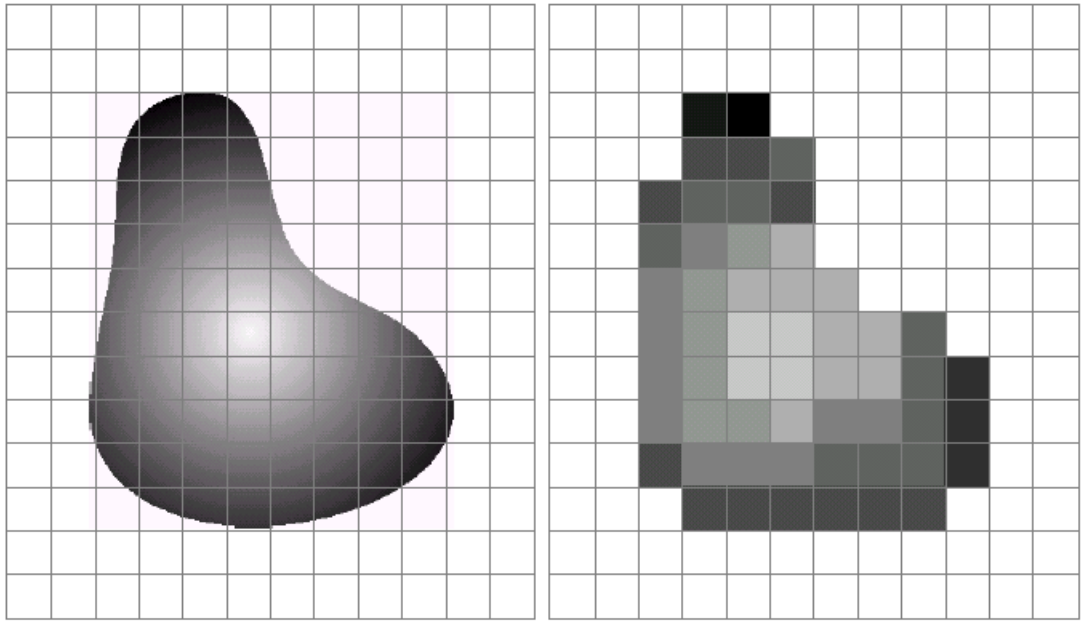
How do we record color?



Bayer pattern for CMOS sensors

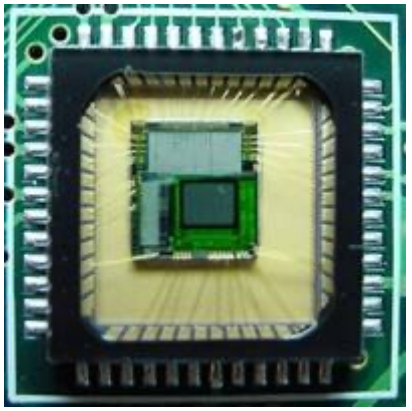


Sensor Array



Continuous image projected onto a sensor array

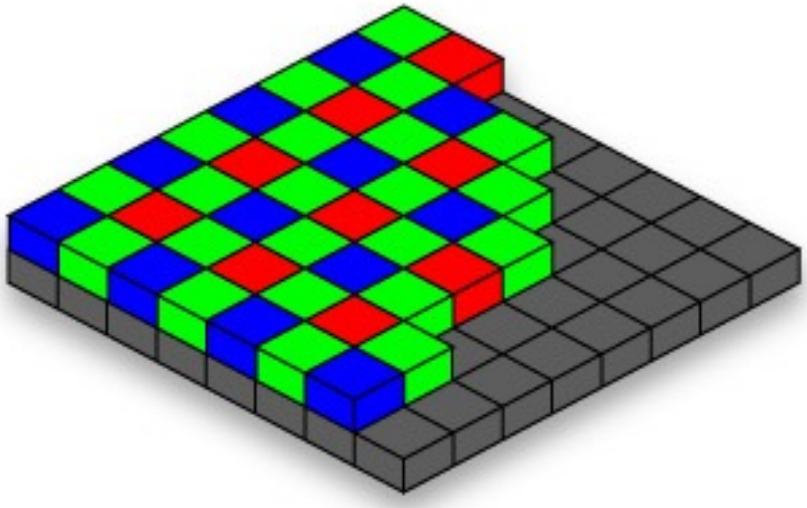
Result of image sampling and quantization



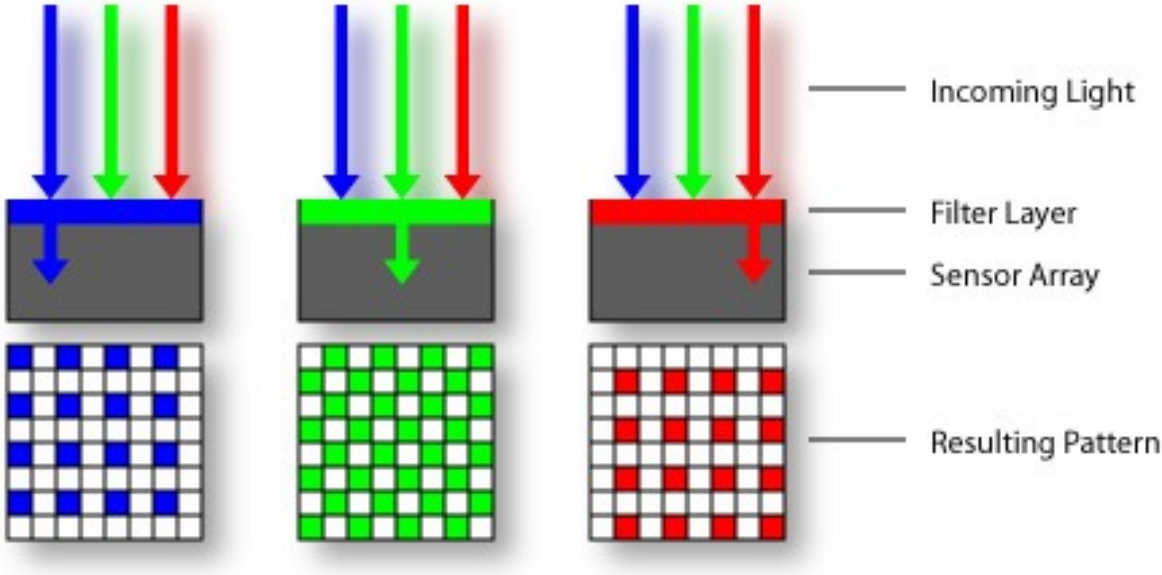
CMOS sensor

Each sensor cell records amount of light coming in at a small range of orientations

Color Sensing: Bayer Grid



Estimate RGB at each cell from neighboring values



http://en.wikipedia.org/wiki/Bayer_filter

Lab 1- demosaicing an image with the Bayer pattern

Demosaicing

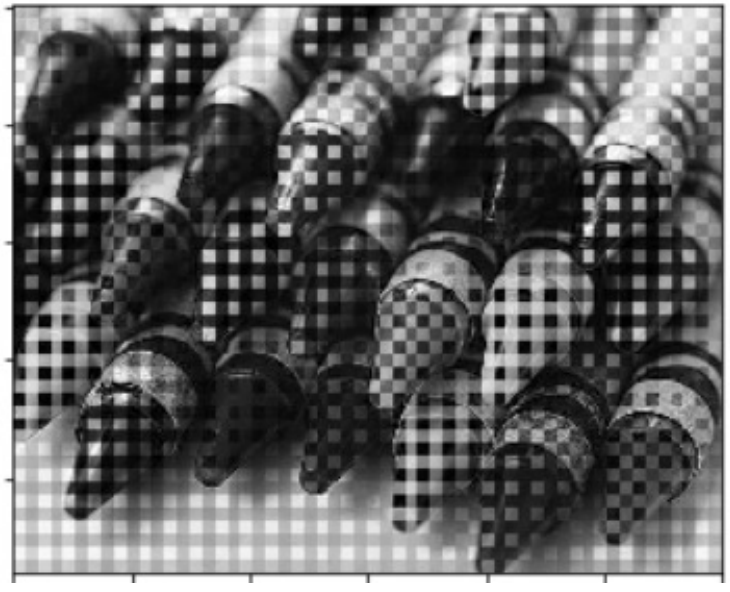
In this assigment, we are going to 'demosaic' an image encoded with the Bayer Pattern. There are some cameras that use the Bayer Pattern in order to save an image. Using this encoding only 50% of green pixels, 25% of red pixels and 25% of blue pixels are kept. The Bayer encoding takes a RBG image and encodes it as in the bellow image.

'rggb'

'bggr'

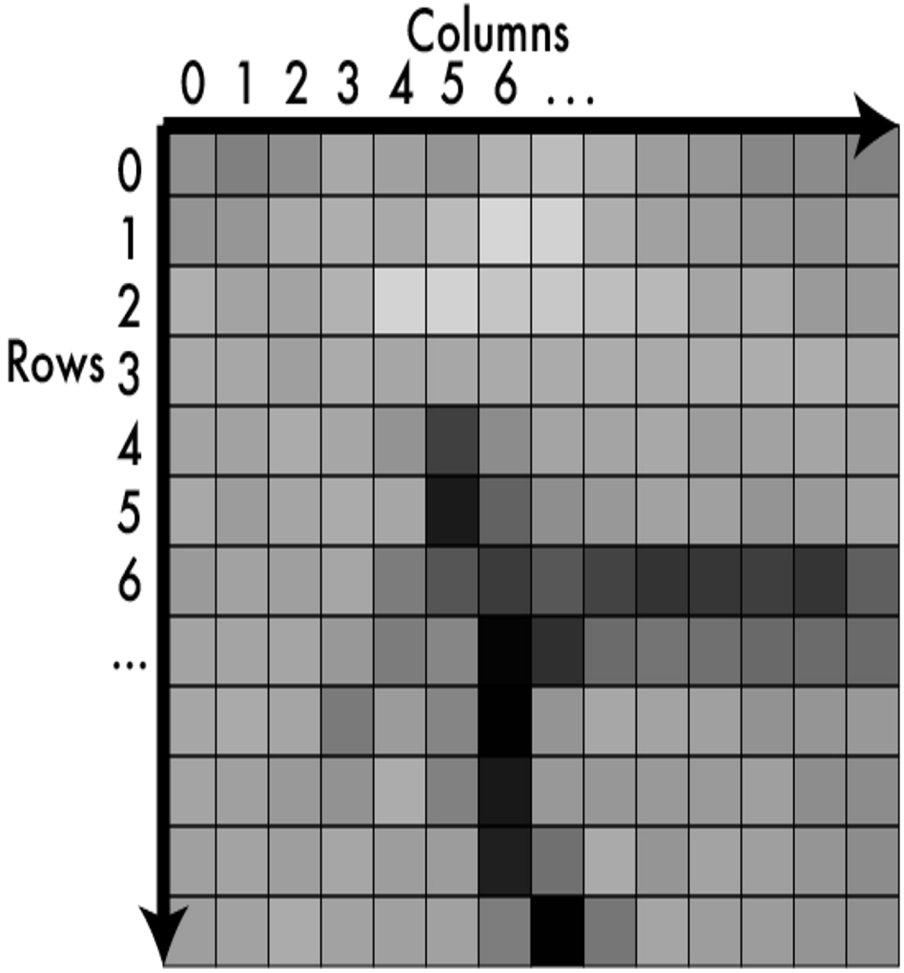
'gbrg'

'grbg'

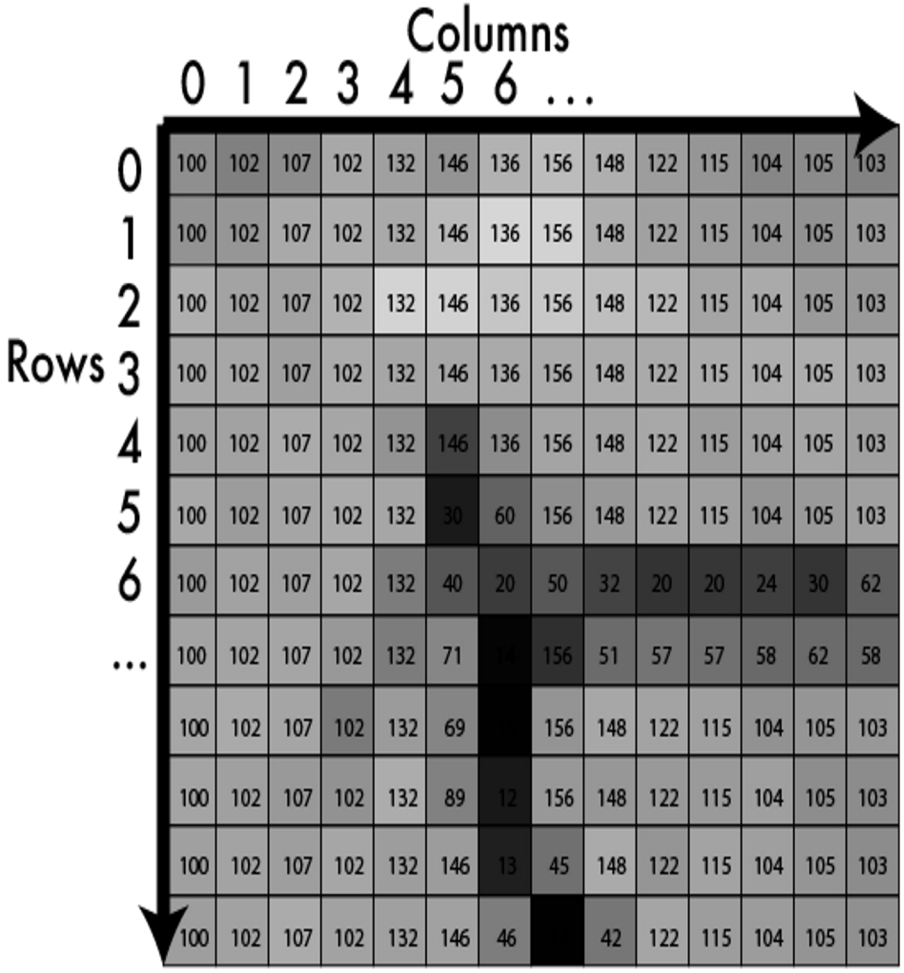


demosaicing

An image is a matrix of light

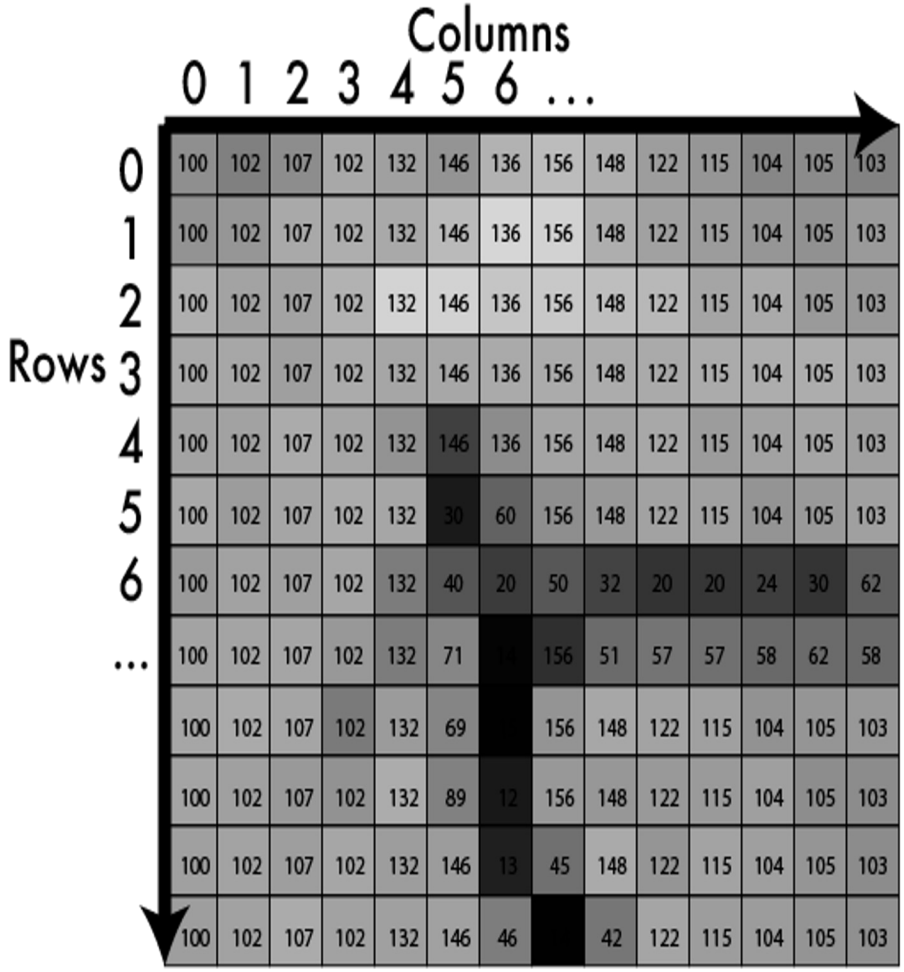


Values in matrix = how much light

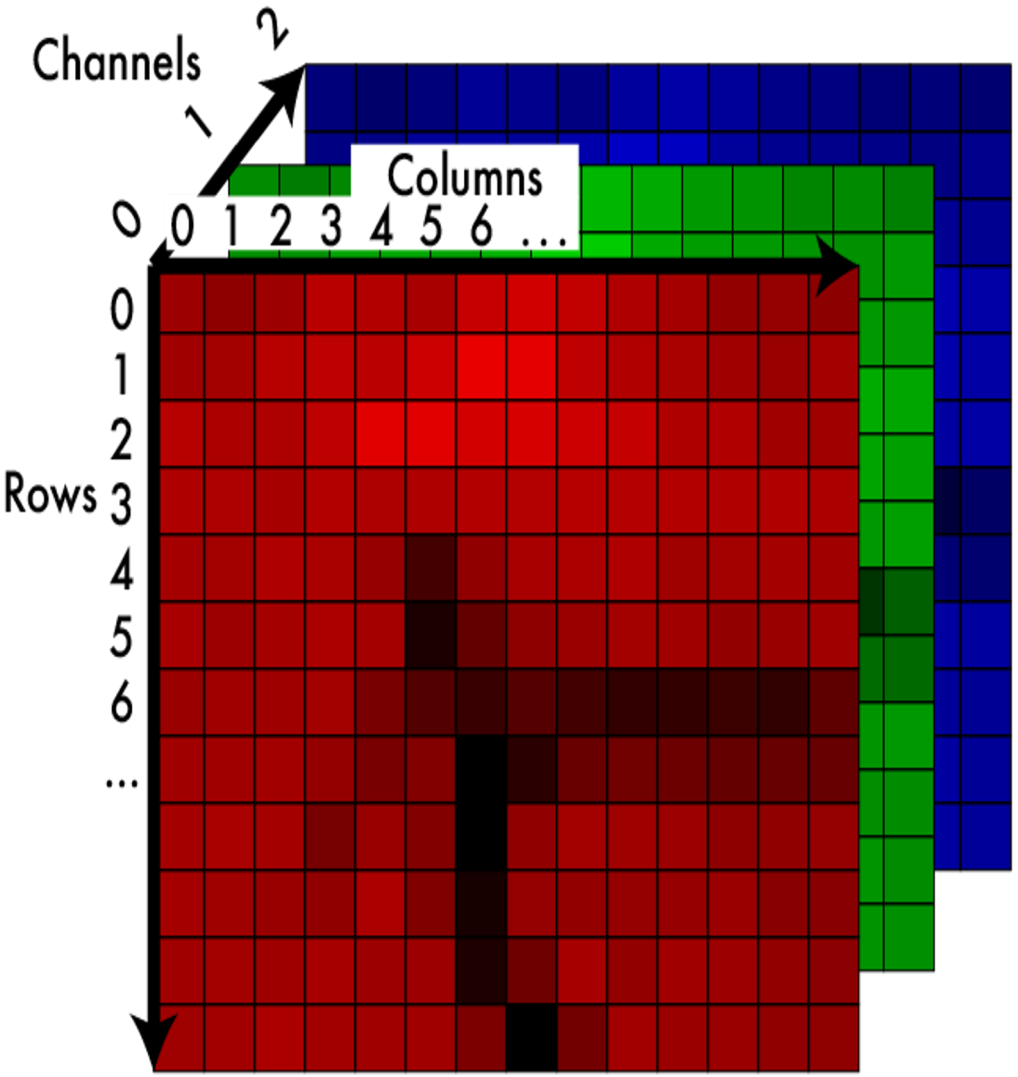


Values in matrix = how much light

- Higher = more light
- Lower = less light
- Bounded
  - No light = 0
  - Sensor/device limit = max
  - Typical ranges:
    - [0-255], fit into a byte
    - [0-1], floating point
- Called pixels

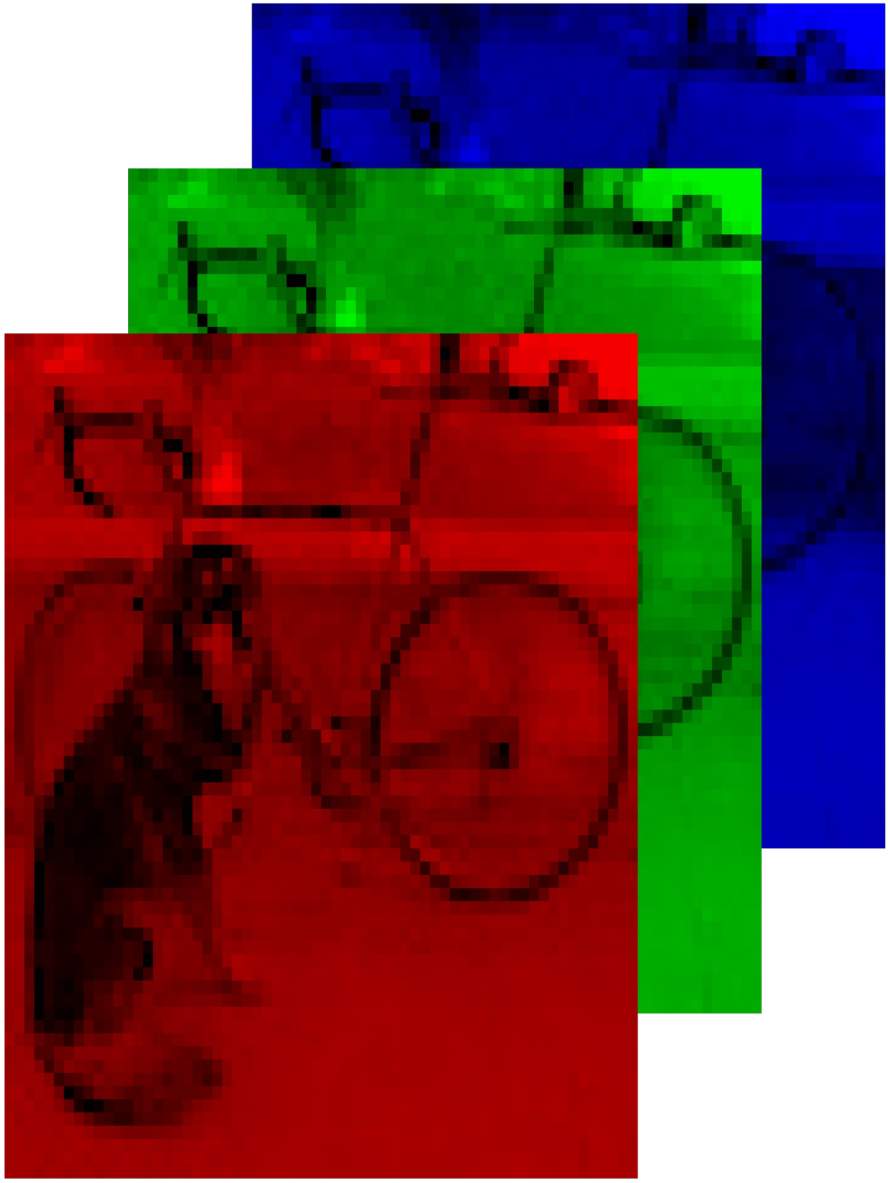


Color image: 3d tensor in colorspace

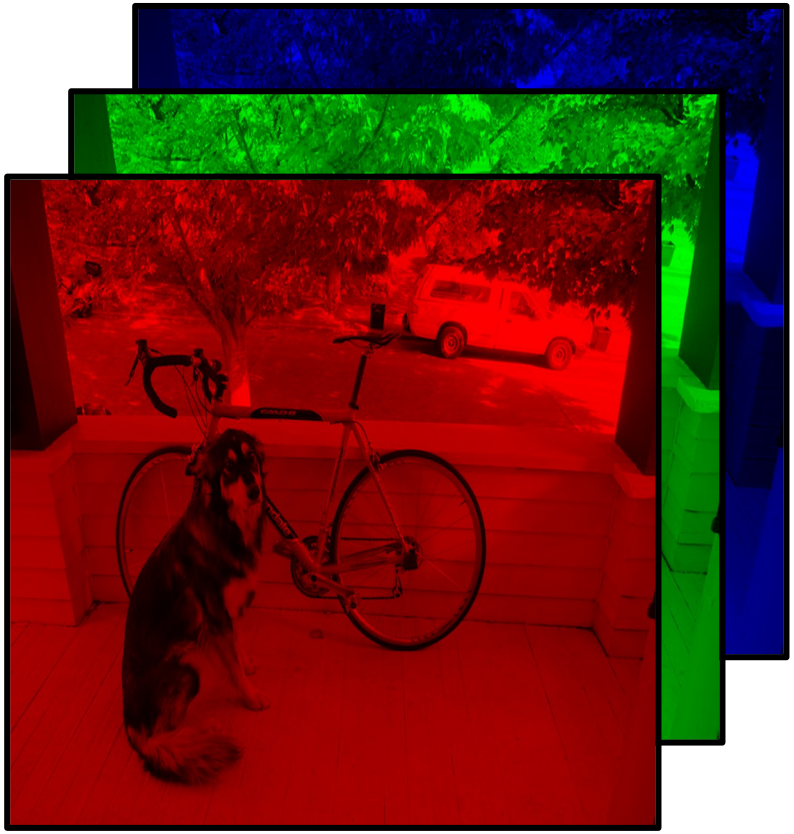


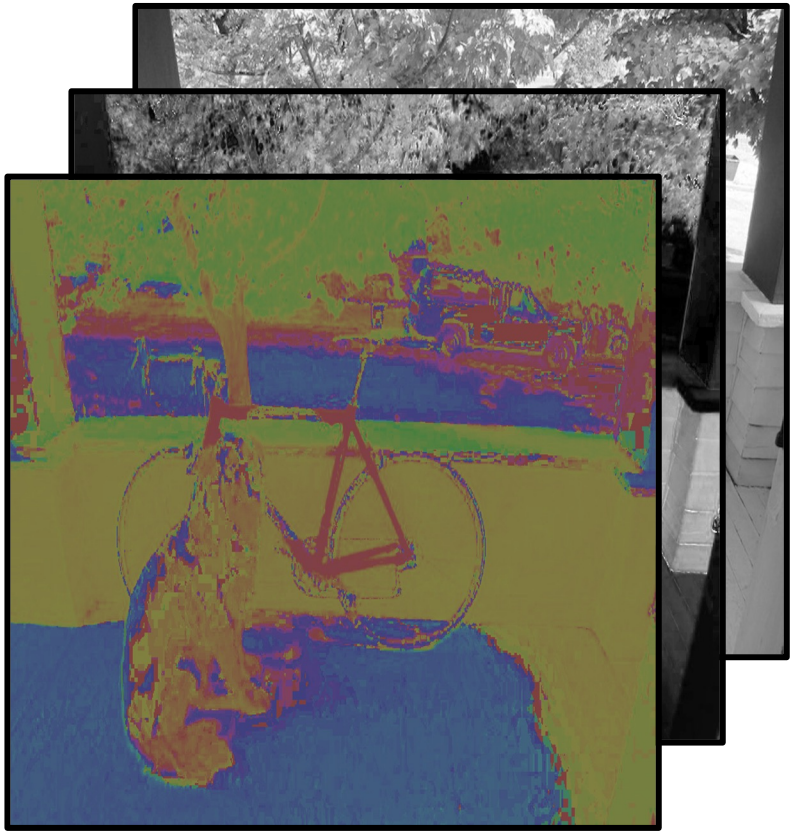
RGB information in separate “channels”

- each channel encodes one primary, use RGB for now.
- we can match “real” colors using a mix of primaries.
- adding the light produced from each primary mimics the original color.



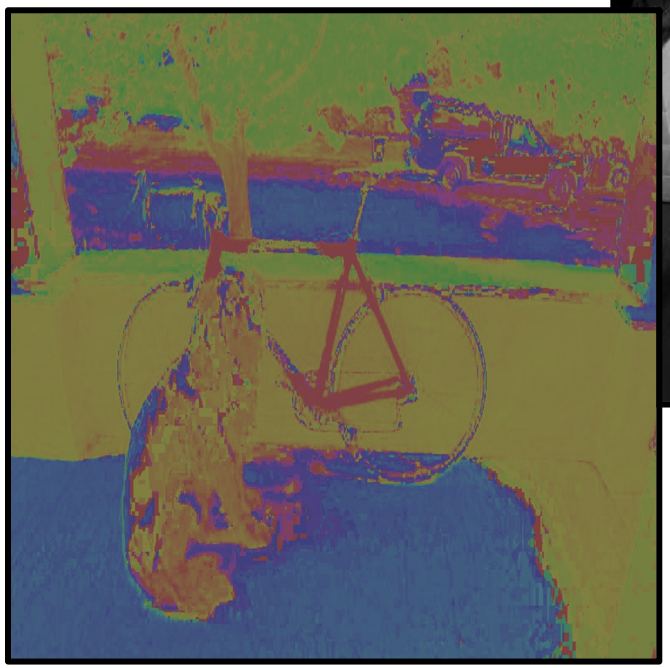
HSV Color image: 3d tensor, different info

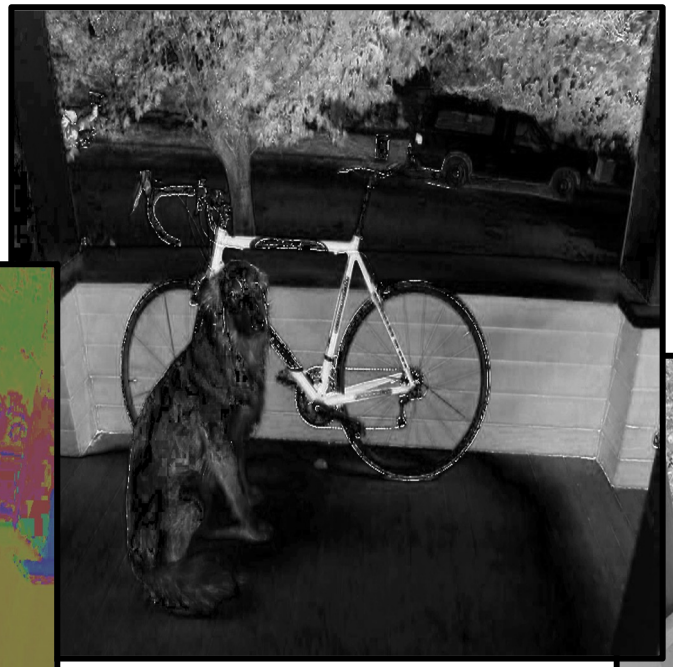




Saturation

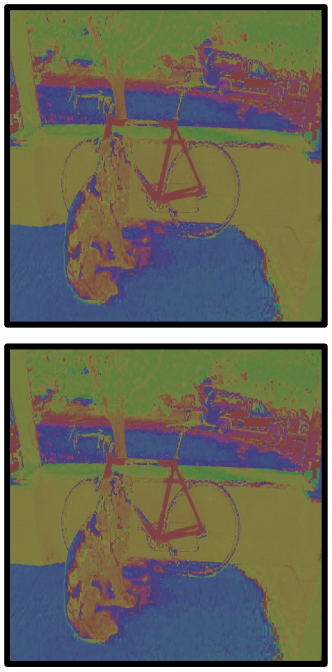
Hue





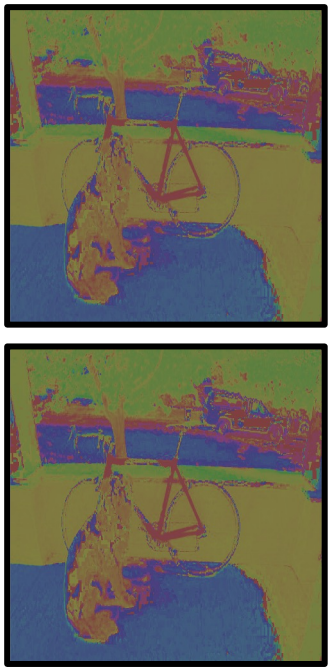
Value

More saturation = intense colors



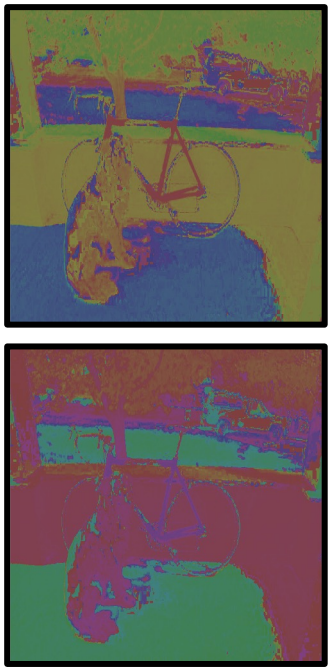
2x

More value = lighter image



2x

Shift hue = shift colors



-.2

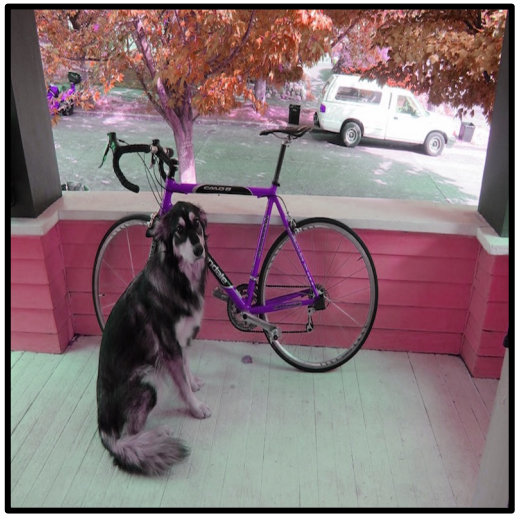


Image interpolation and resizing

An image is kinda like a function

An image is a mapping from indices to pixel values:

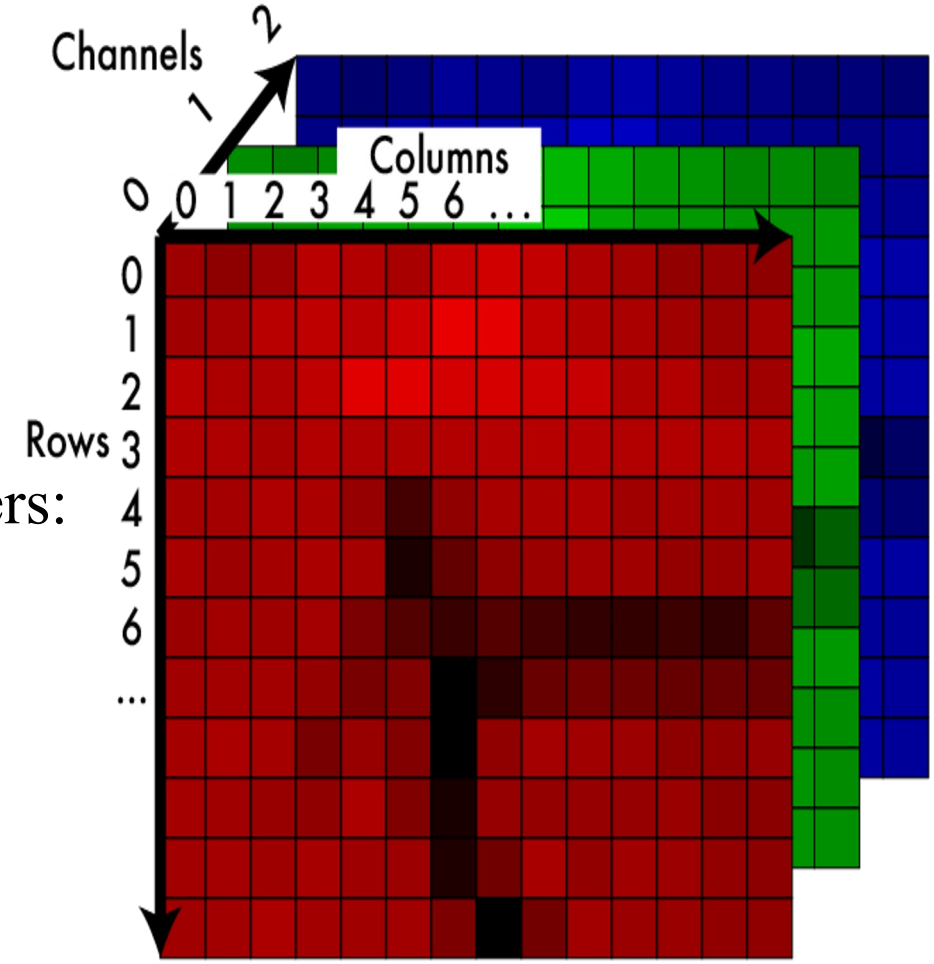
$$\text{Im} : I \times I \times I \rightarrow R$$

height — indices of rows

width — indices of columns

channel

value

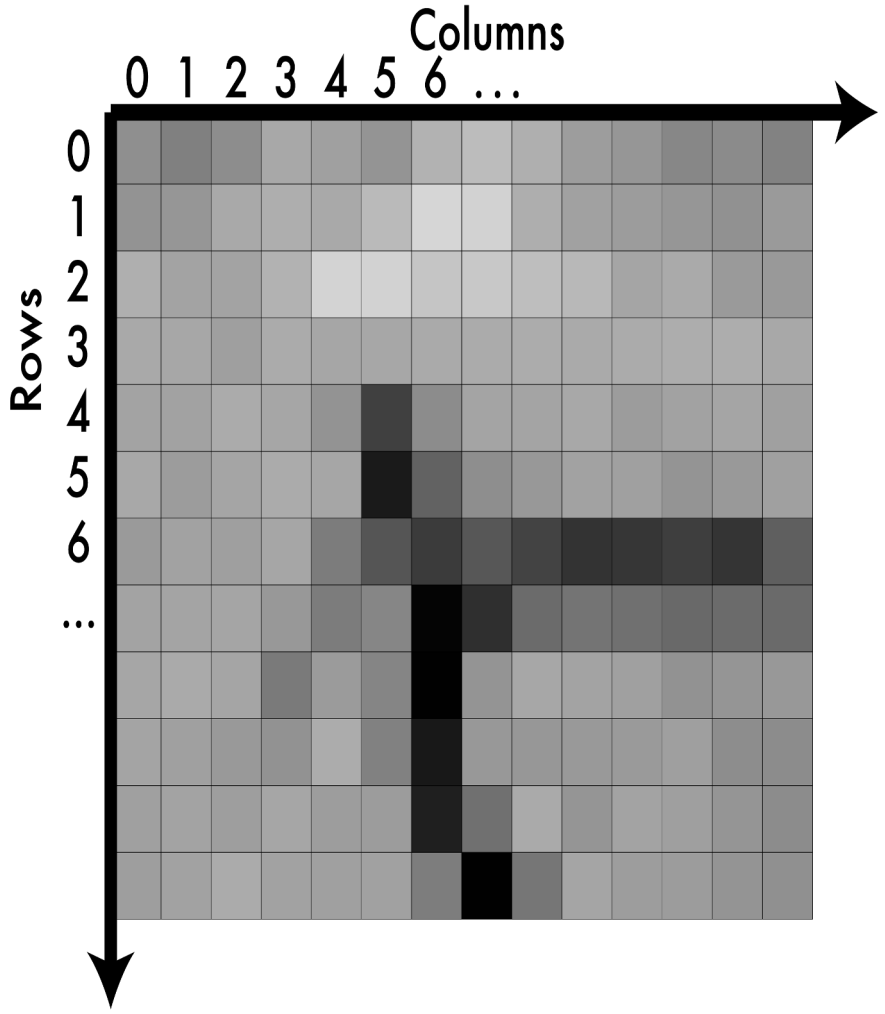


We may want to pass in non-integers:

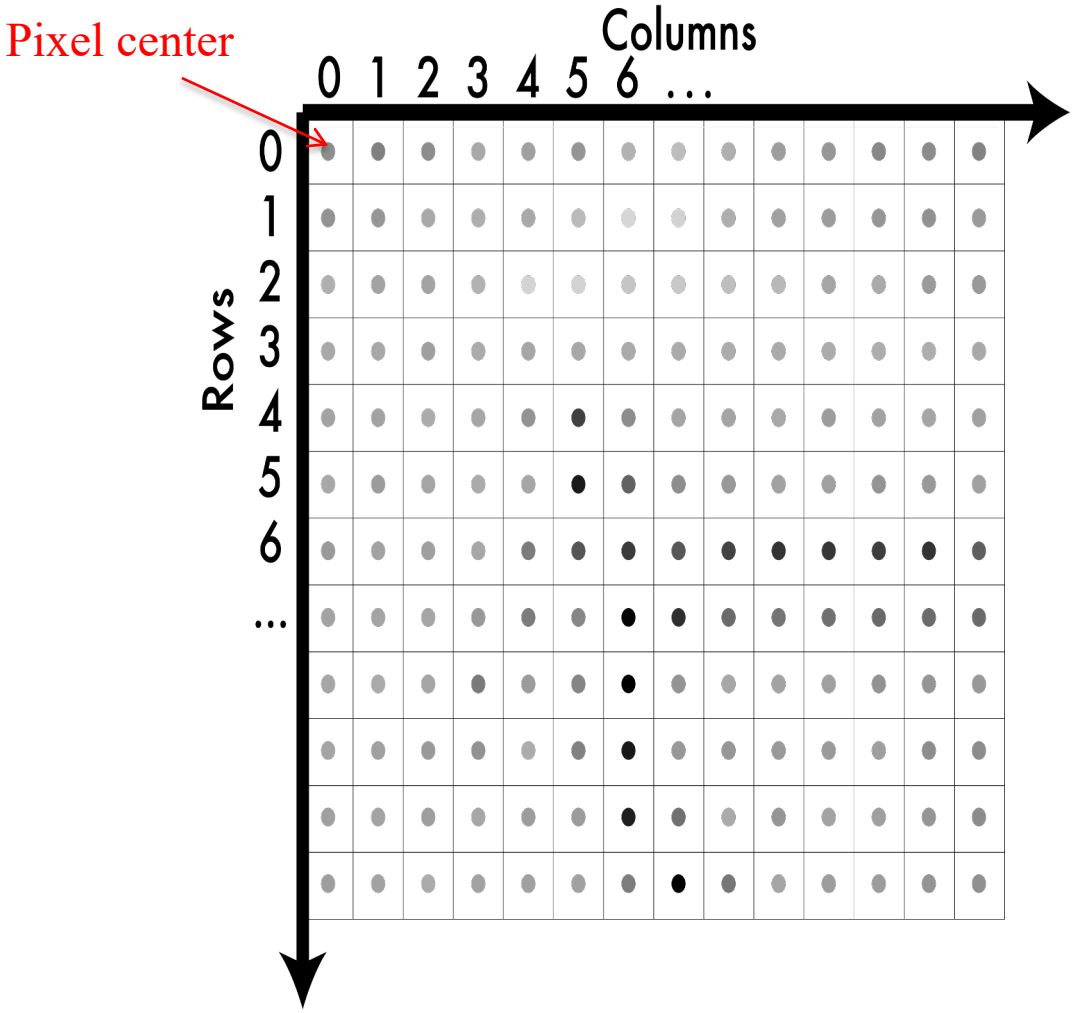
$$\text{Im}' : R \times R \times I \rightarrow R$$

height — width — channel — value

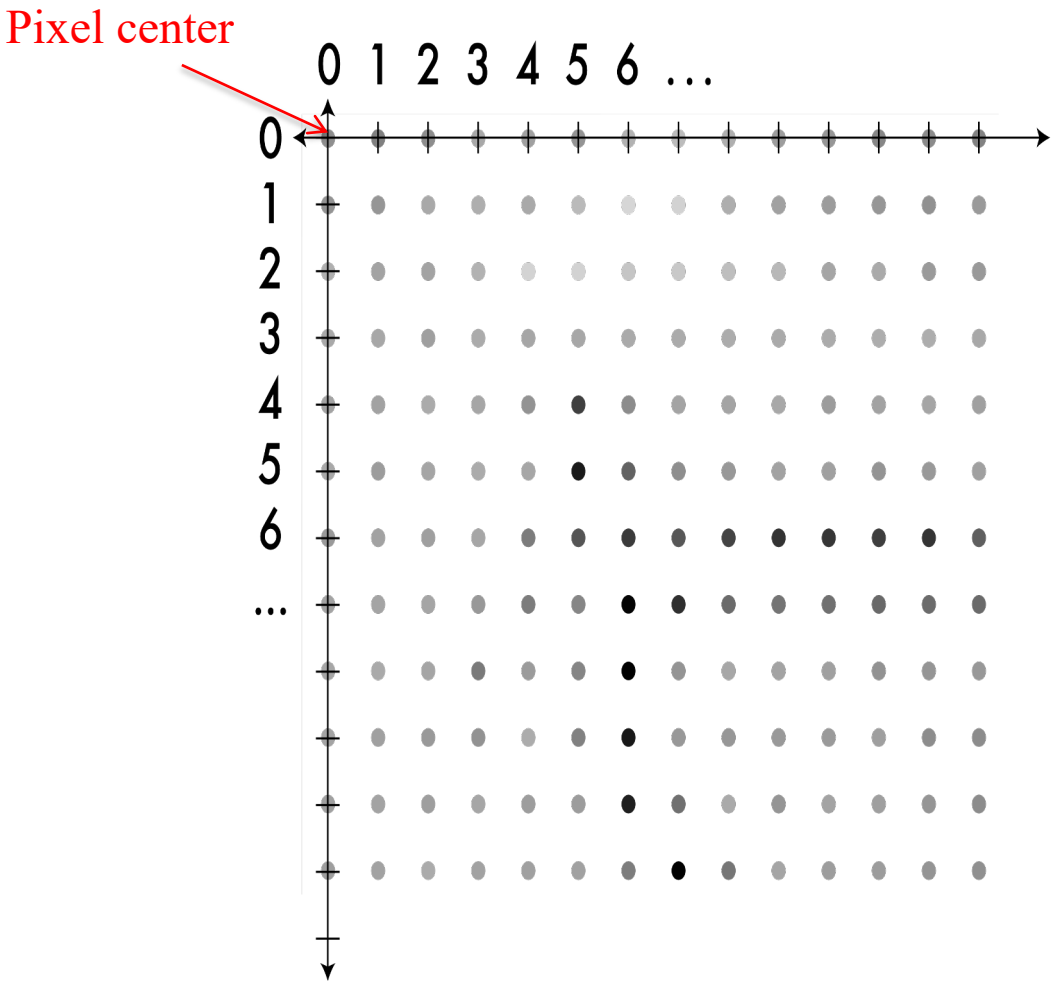
A note on coordinates in images



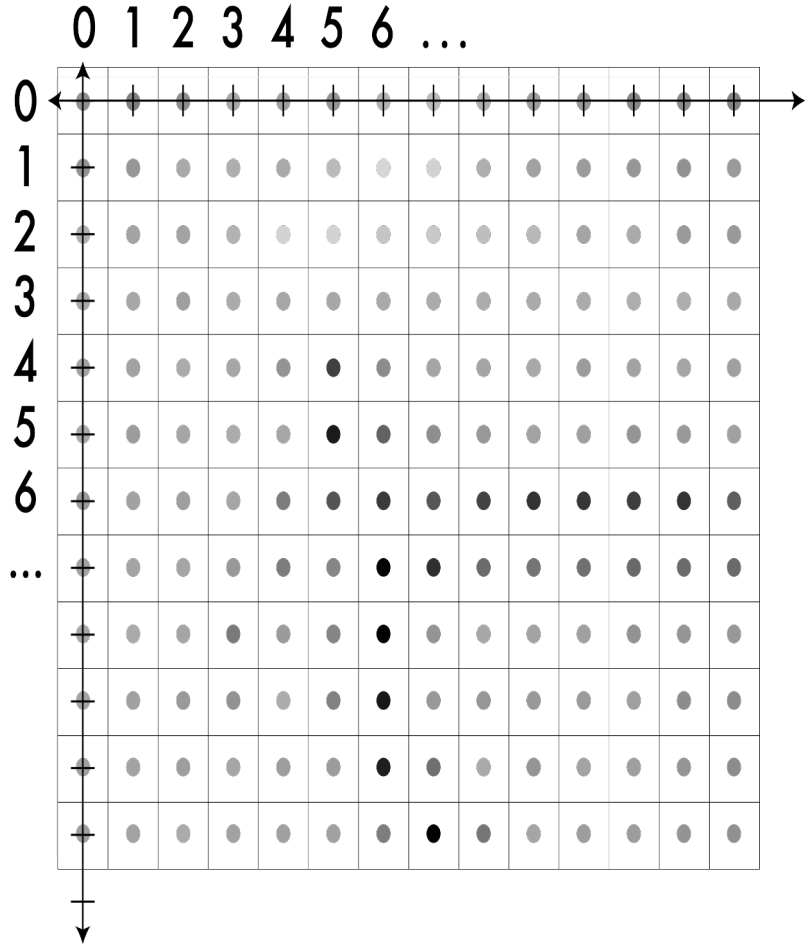
A note on coordinates in images



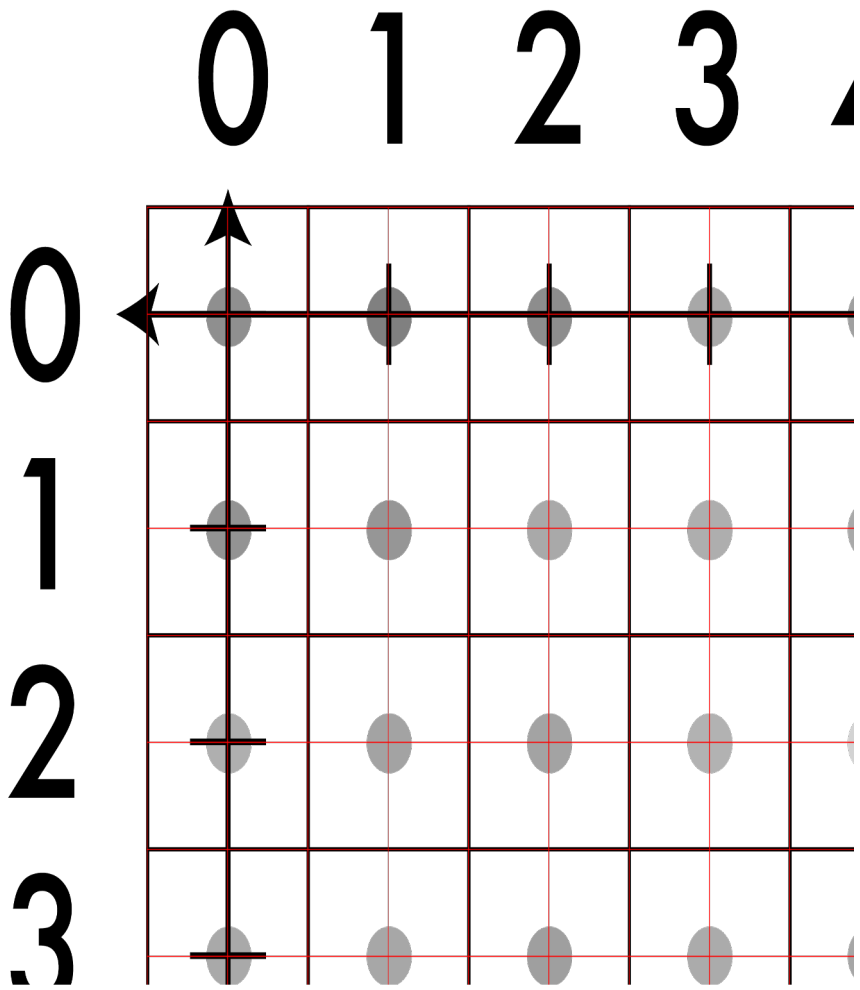
A note on coordinates in images



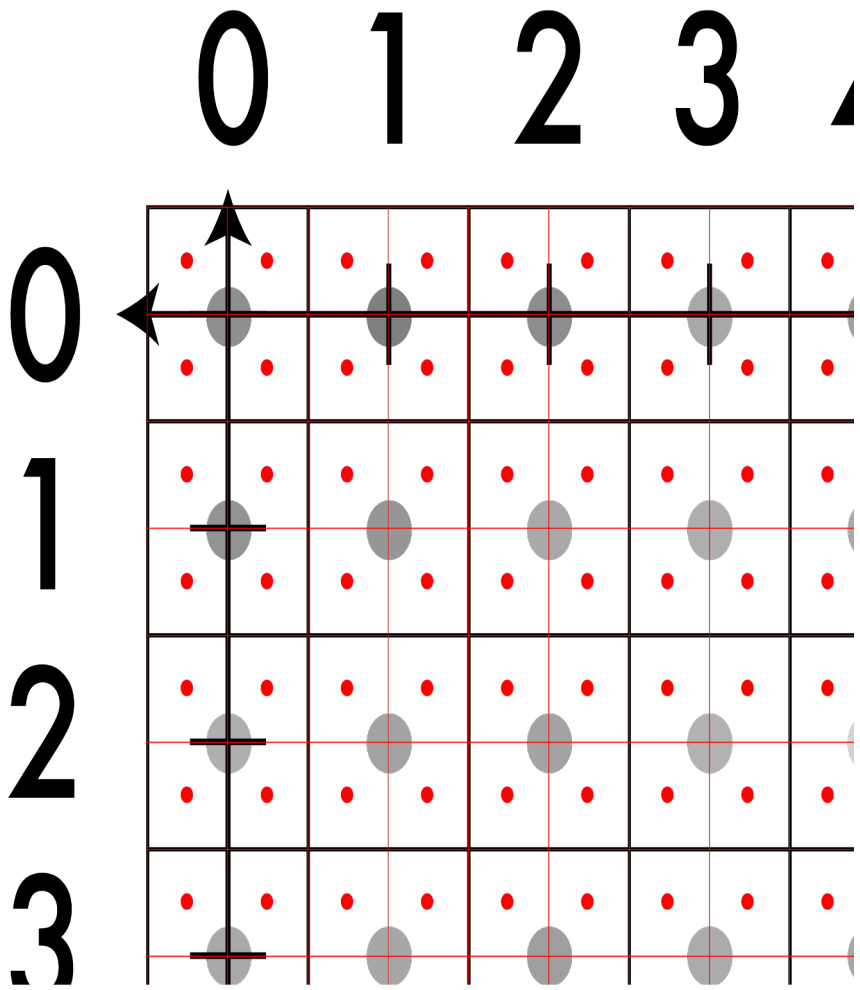
A note on coordinates in images



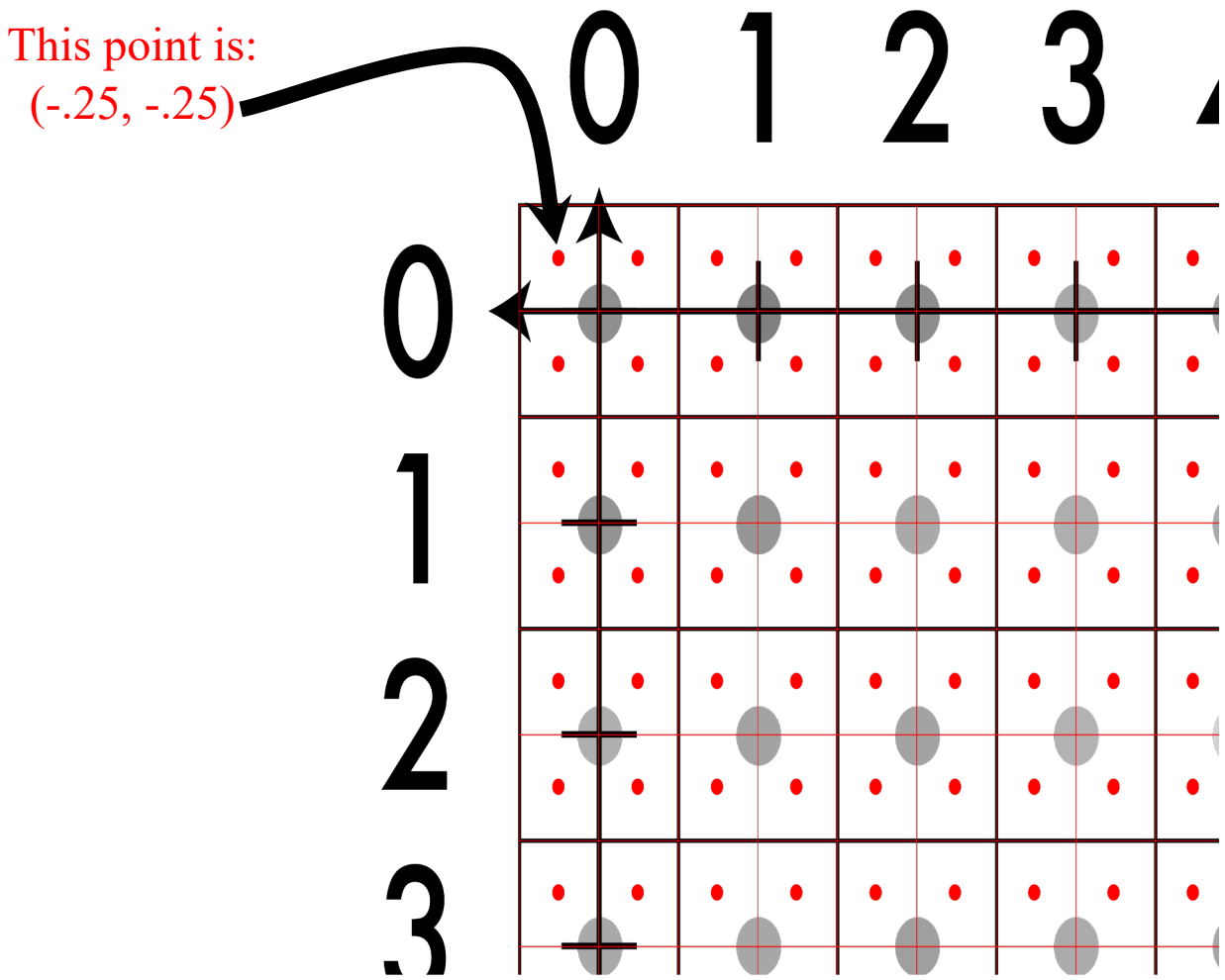
A note on coordinates in images



A note on coordinates in images



A note on coordinates in images



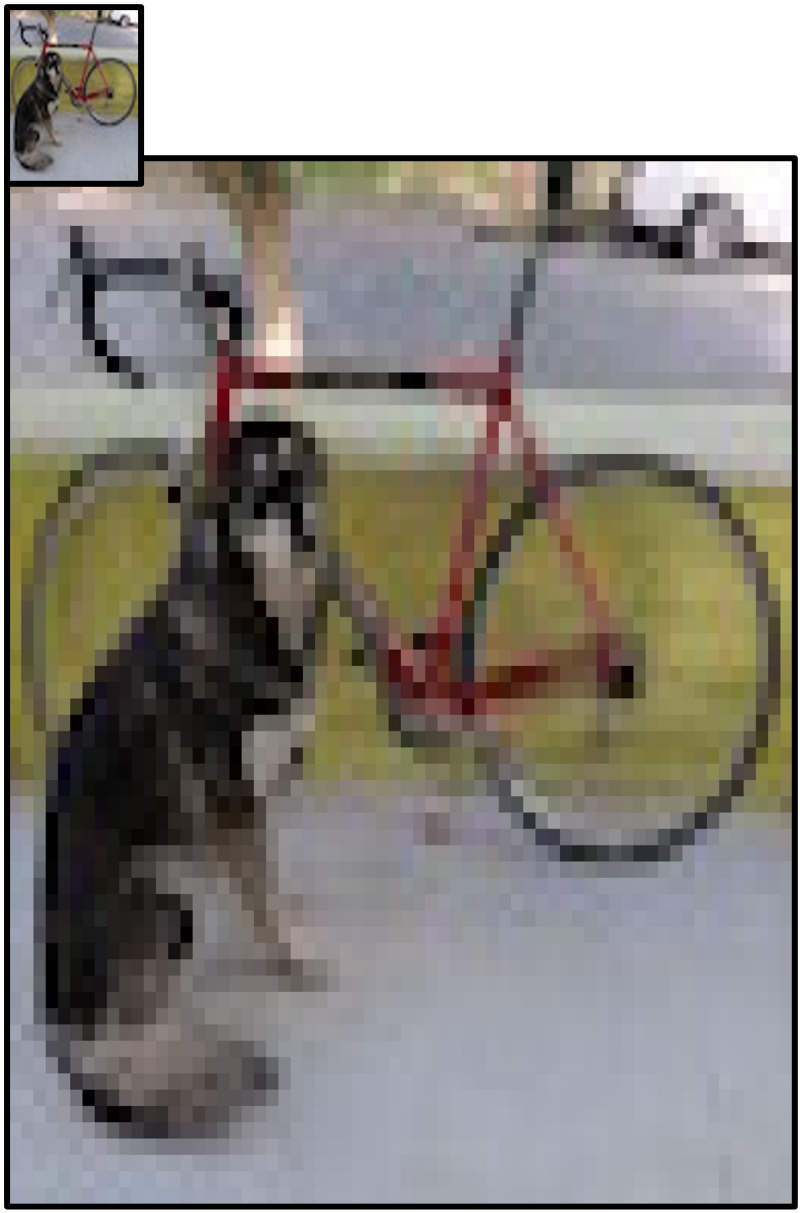
Nearest neighbor interpolation

$$\text{Im} : \underset{\text{height}}{I} \times \underset{\text{width}}{I} \times \underset{\text{channel}}{I} \rightarrow \underset{\text{value}}{R}$$

$$\text{Im}' : \underset{\text{height}}{R} \times \underset{\text{width}}{R} \times \underset{\text{channel}}{I} \rightarrow \underset{\text{value}}{R}$$

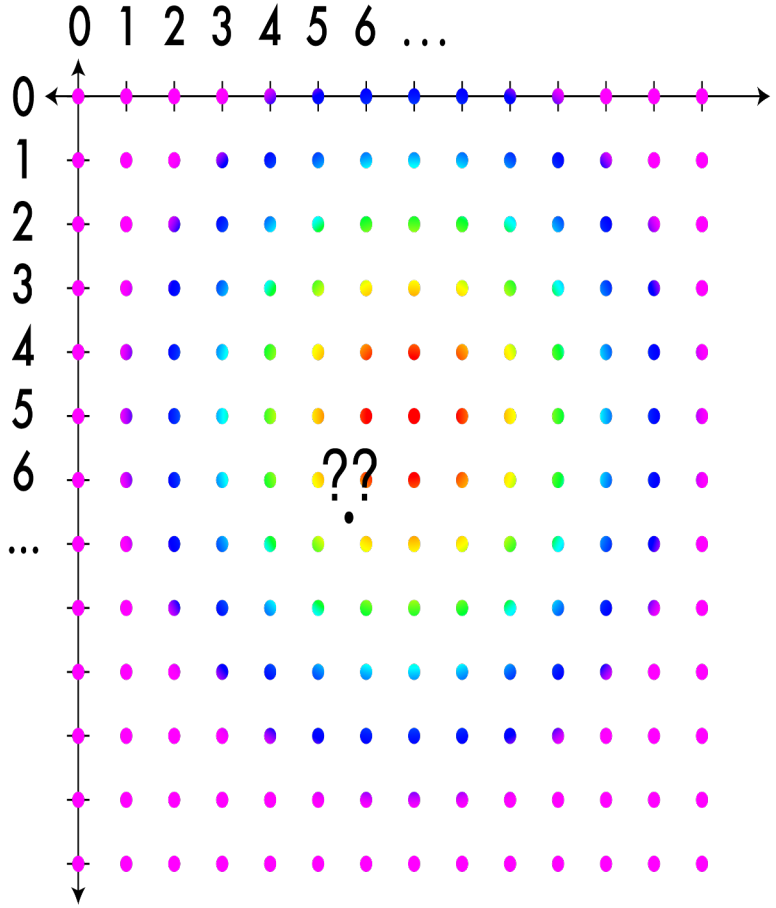
$$\text{Im}'(y, x, c) = \text{Im}(round(y), round(x), c)$$

- Looks blocky



Bilinear interpolation

Find the closest pixels in a box



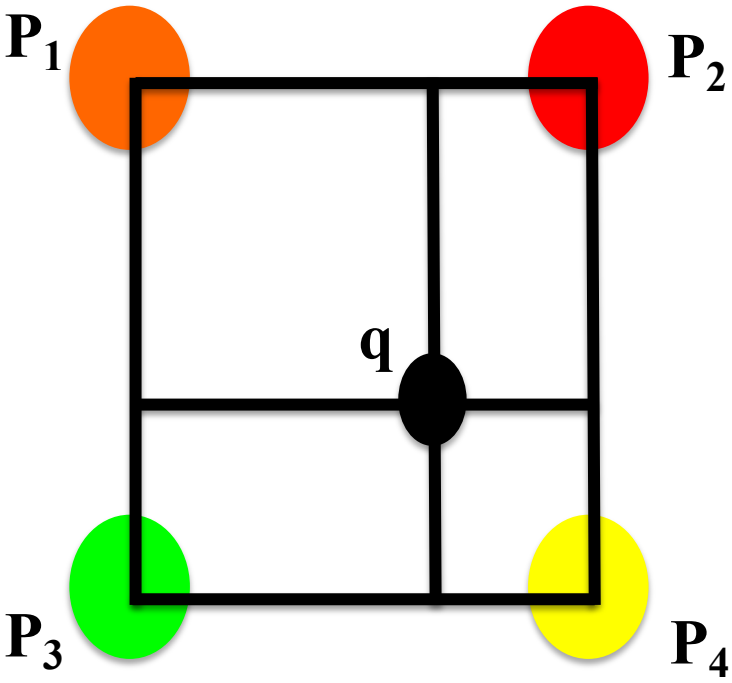
Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

Closer pixels should have a higher coefficient (weight)



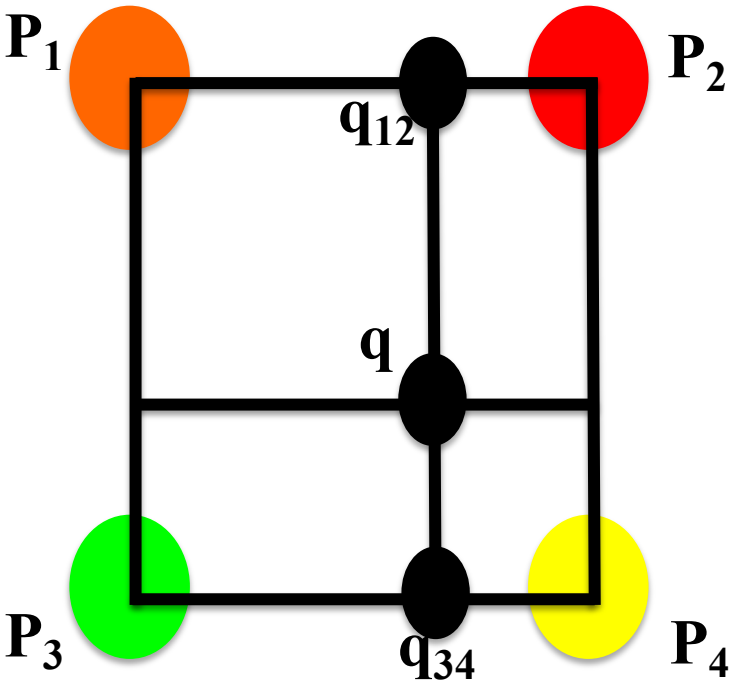
Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

Closer pixels should have a higher coefficient (weight)

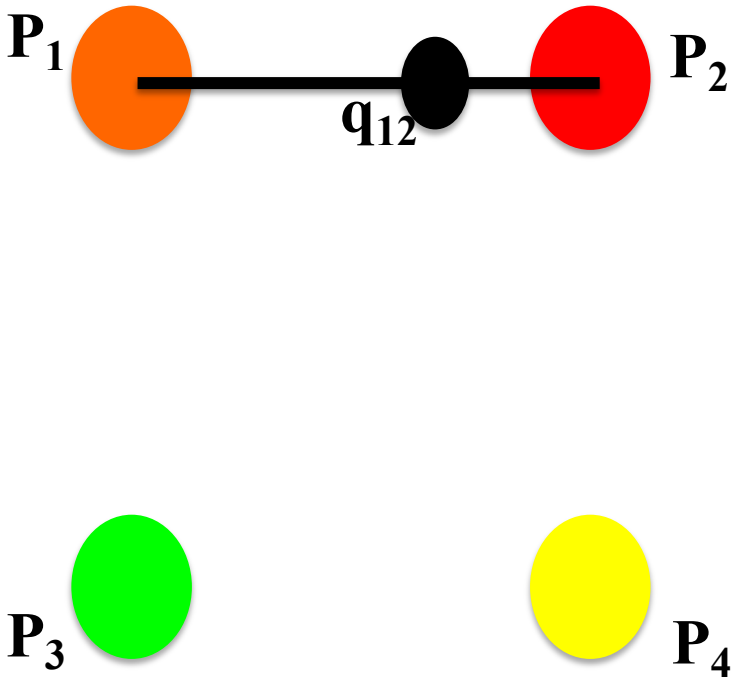


Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$



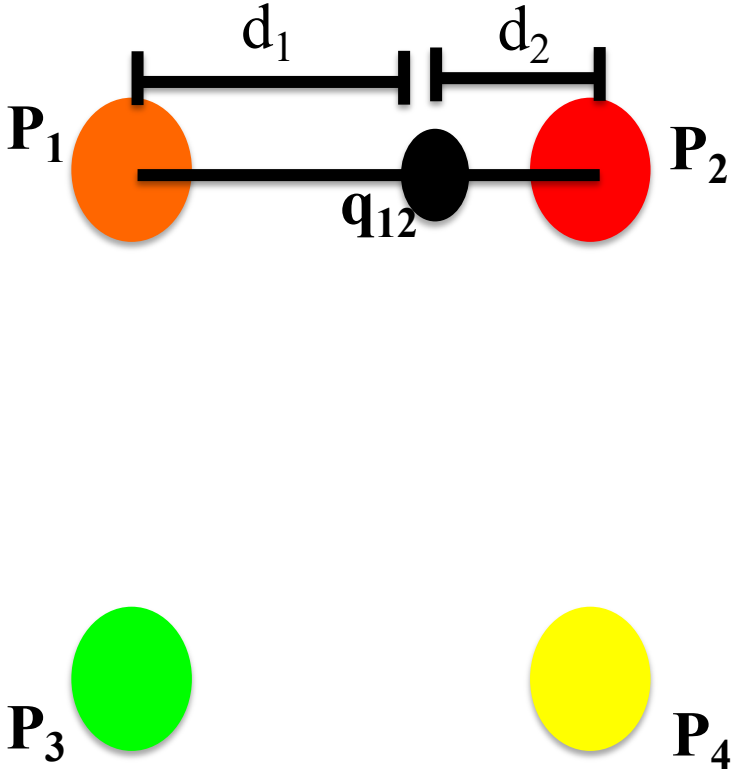
q_{12} between P_1 and P_2

Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

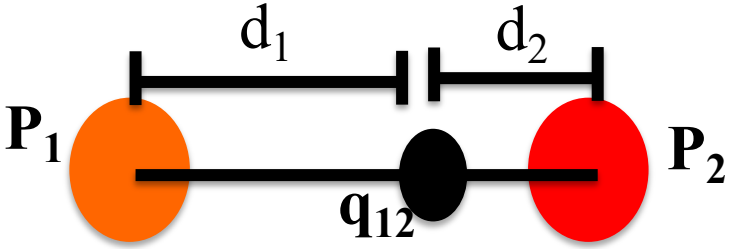


q_{12} between P_1 and P_2

Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:



$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

q_{12} between P_1 and P_2

$$q_{12} = \frac{d_2}{d_1 + d_2} P_1 + \frac{d_1}{d_1 + d_2} P_2$$

But $d_1 + d_2 = 1$, so:

$$q_{12} = d_2 P_1 + d_1 P_2$$

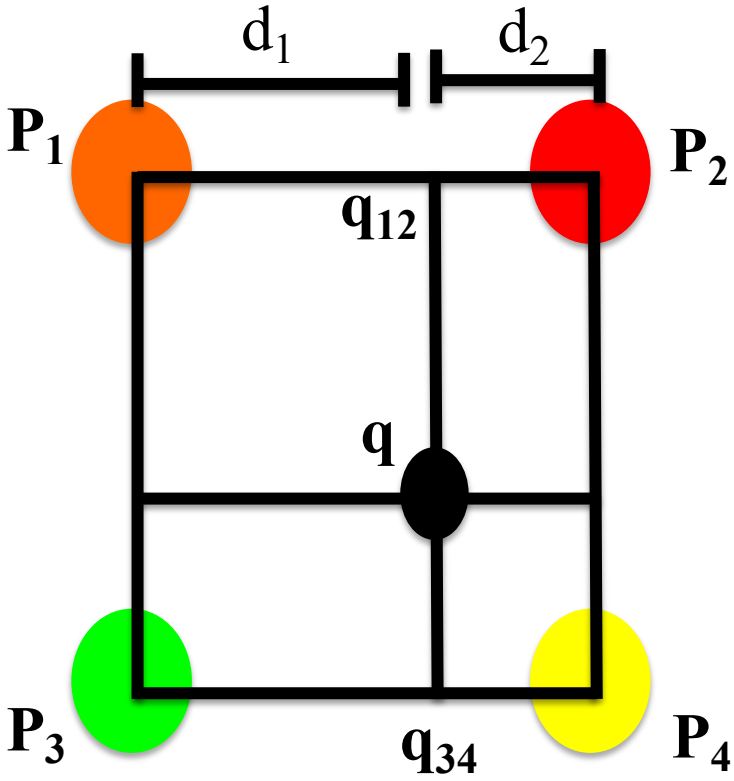
Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

$$q_{12} = d_2 P_1 + d_1 P_2$$
$$q_{34} = d_2 P_3 + d_1 P_4$$



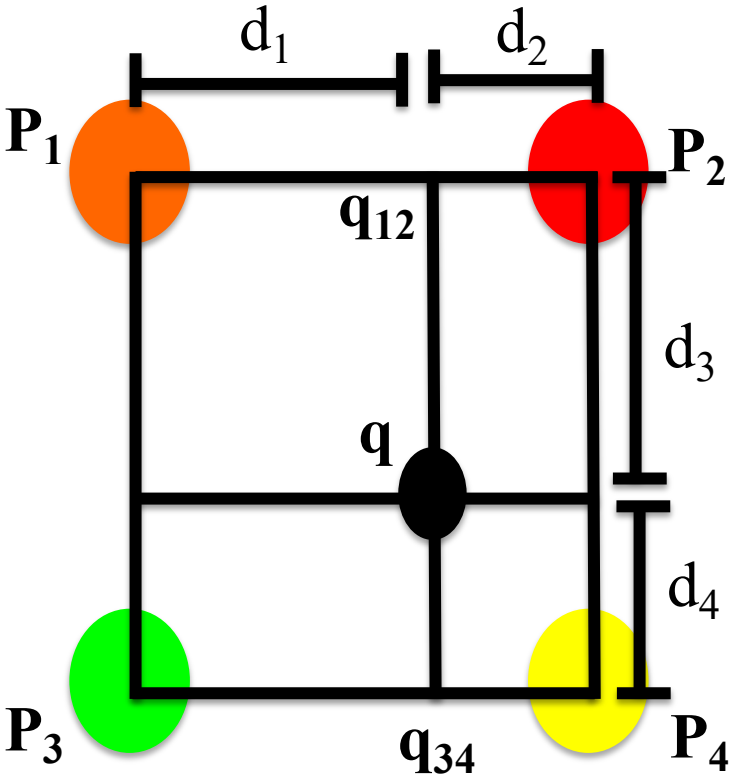
Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

$$q_{12} = d_2 P_1 + d_1 P_2$$
$$q_{34} = d_2 P_3 + d_1 P_4$$



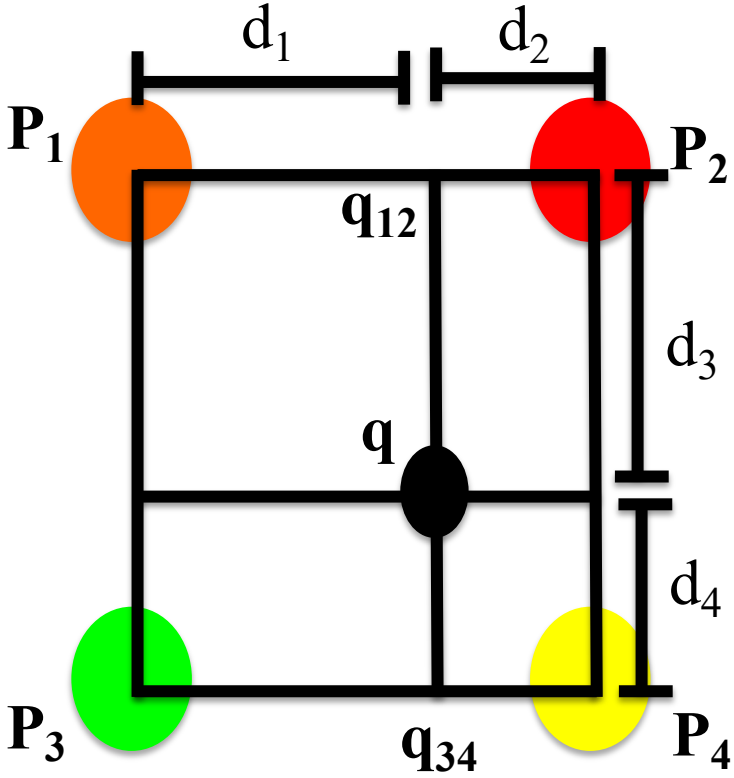
Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

$$q_{12} = d_2 P_1 + d_1 P_2$$
$$q_{34} = d_2 P_3 + d_1 P_4$$



Pixel q between q_{12} and q_{34}, so:

$$q = d_4 q_{12} + d_3 q_{34} = d_4 (d_2 P_1 + d_1 P_2) + d_3 (d_2 P_3 + d_1 P_4)$$

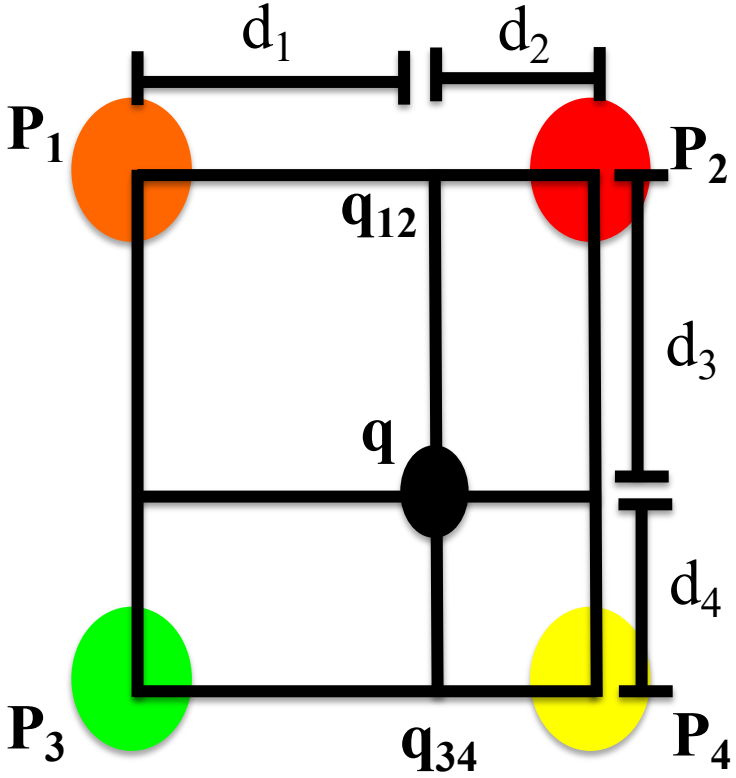
Bilinear interpolation

Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

$$q_{12} = d_2 P_1 + d_1 P_2$$
$$q_{34} = d_2 P_3 + d_1 P_4$$



Pixel q between q_{12} and q_{34}, so:

$$q = d_4 q_{12} + d_3 q_{34} = d_4 (d_2 P_1 + d_1 P_2) + d_3 (d_2 P_3 + d_1 P_4)$$

So, $\alpha_1 = d_2 d_4, \alpha_2 = d_1 d_4, \alpha_3 = d_2 d_3, \alpha_4 = d_1 d_3$

Bilinear interpolation

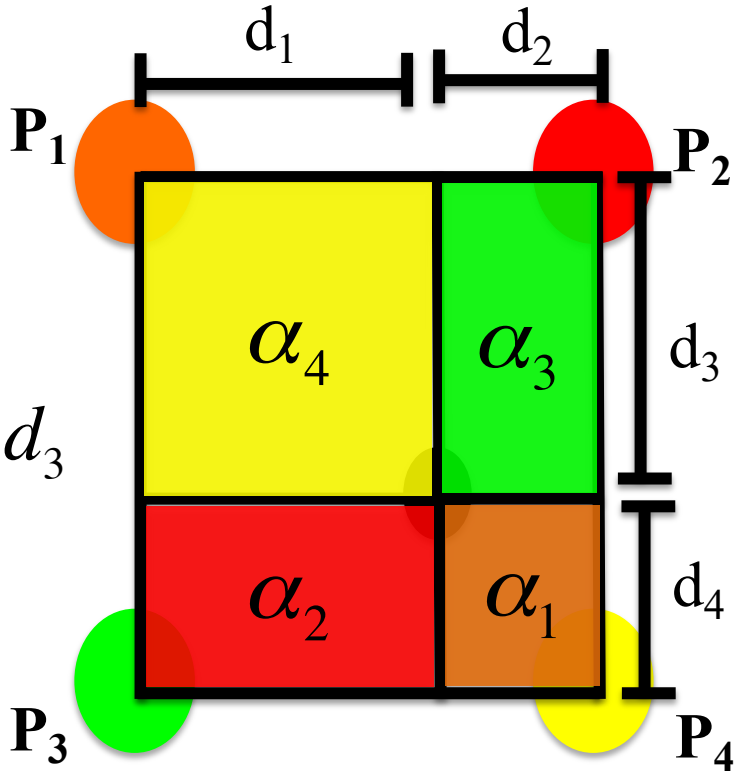
Find the closest pixels in a box

Determine coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ such that:

$$q = \alpha_1 P_1 + \alpha_2 P_2 + \alpha_3 P_3 + \alpha_4 P_4$$

$$\alpha_1 = d_2 d_4, \alpha_2 = d_1 d_4, \alpha_3 = d_2 d_3, \alpha_4 = d_1 d_3$$

α_i - area of the opposite rectangle to pixel P_i

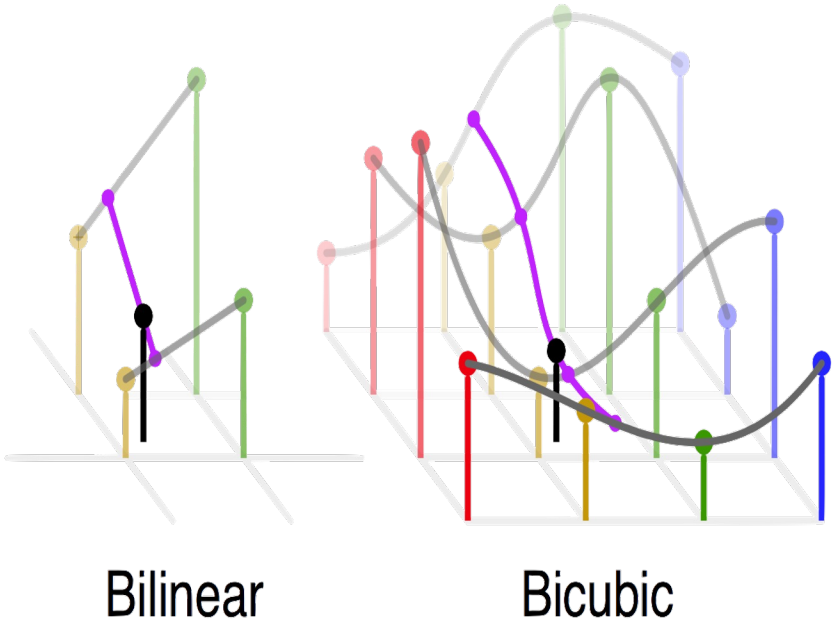


Bilinear interpolation

- Smoother than NN
- More complex
  - 4 lookups
  - some math
- Often the right tradeoff of speed vs final result

Bicubic interpolation

- A cubic interpolation of 4 cubic interpolations
- Smoother than bilinear, no “star”
- 16 nearest neighbors
- Use 4 points to fit 3rd order poly:
  - $f(x) = a + bx + cx^2 + dx^3$
- Interpolate along axis
- Fit another poly to interpolated values
- More complex
- Maybe better?

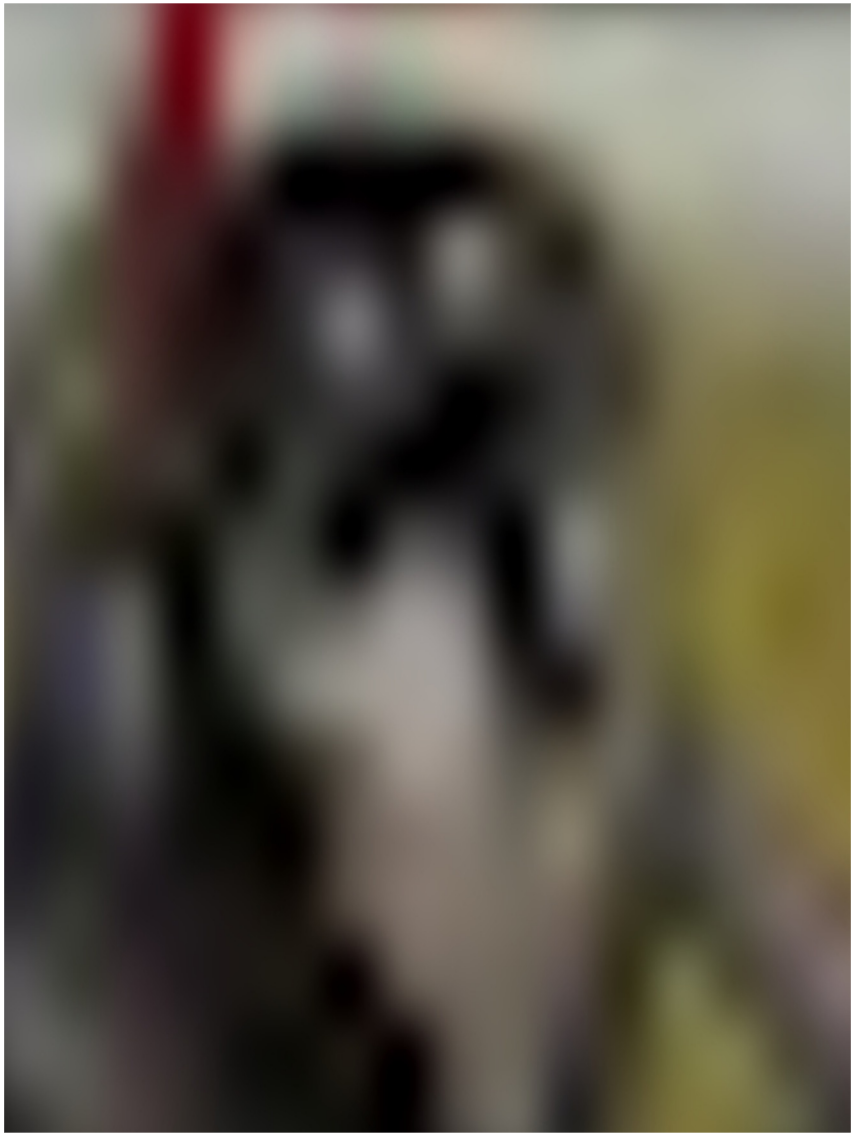


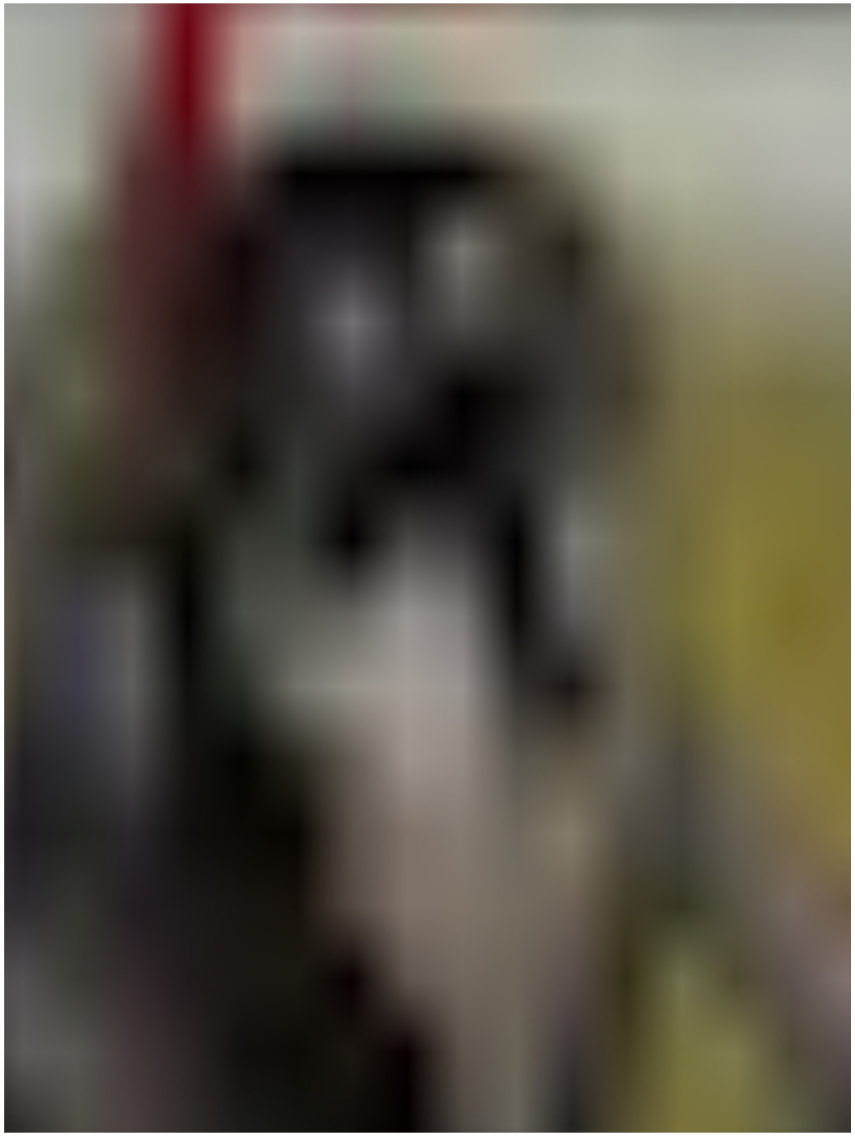
Bicubic vs bilinear

bicubic

bilinear

Bicubic vs bilinear





Using interpolation for Image Resizing

Image resizing!

Say we want to increase the size of an image…

This is a beautiful image of a sunset… it's just very small…

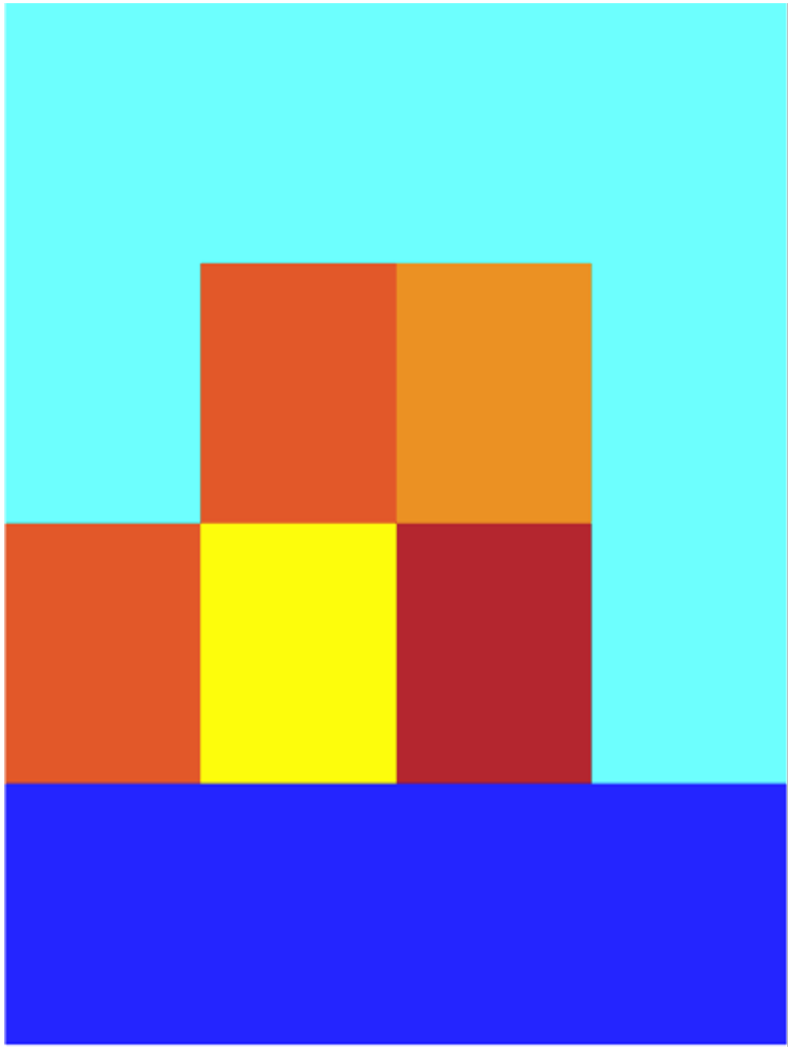
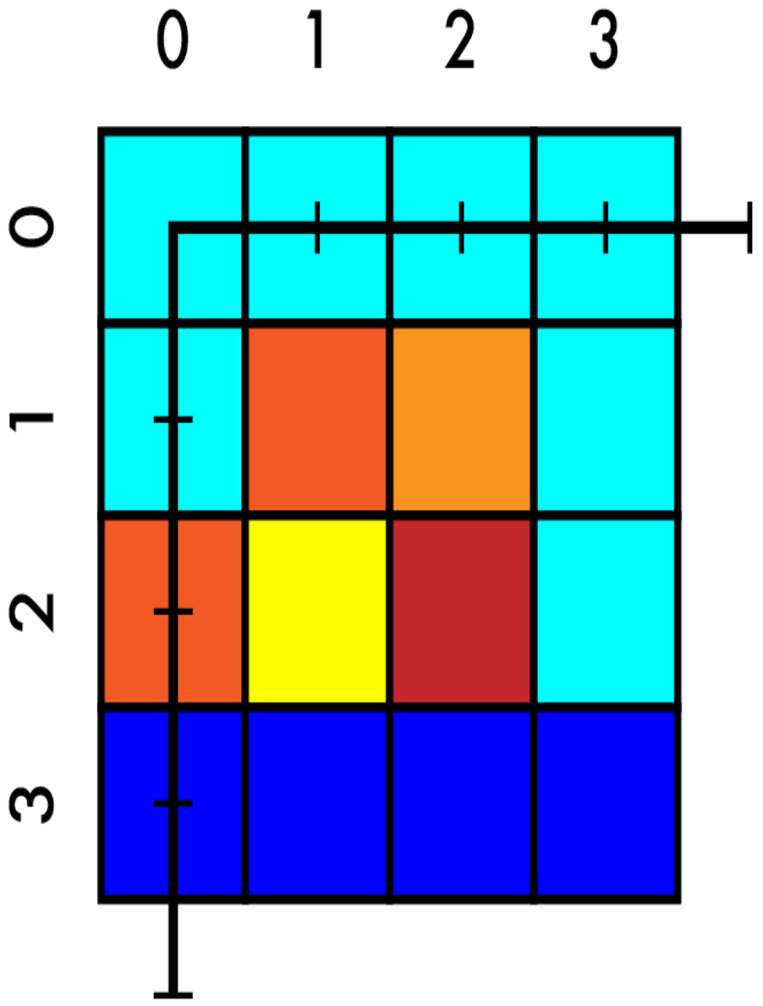


Image resizing!

Say we want to increase the size of an image…

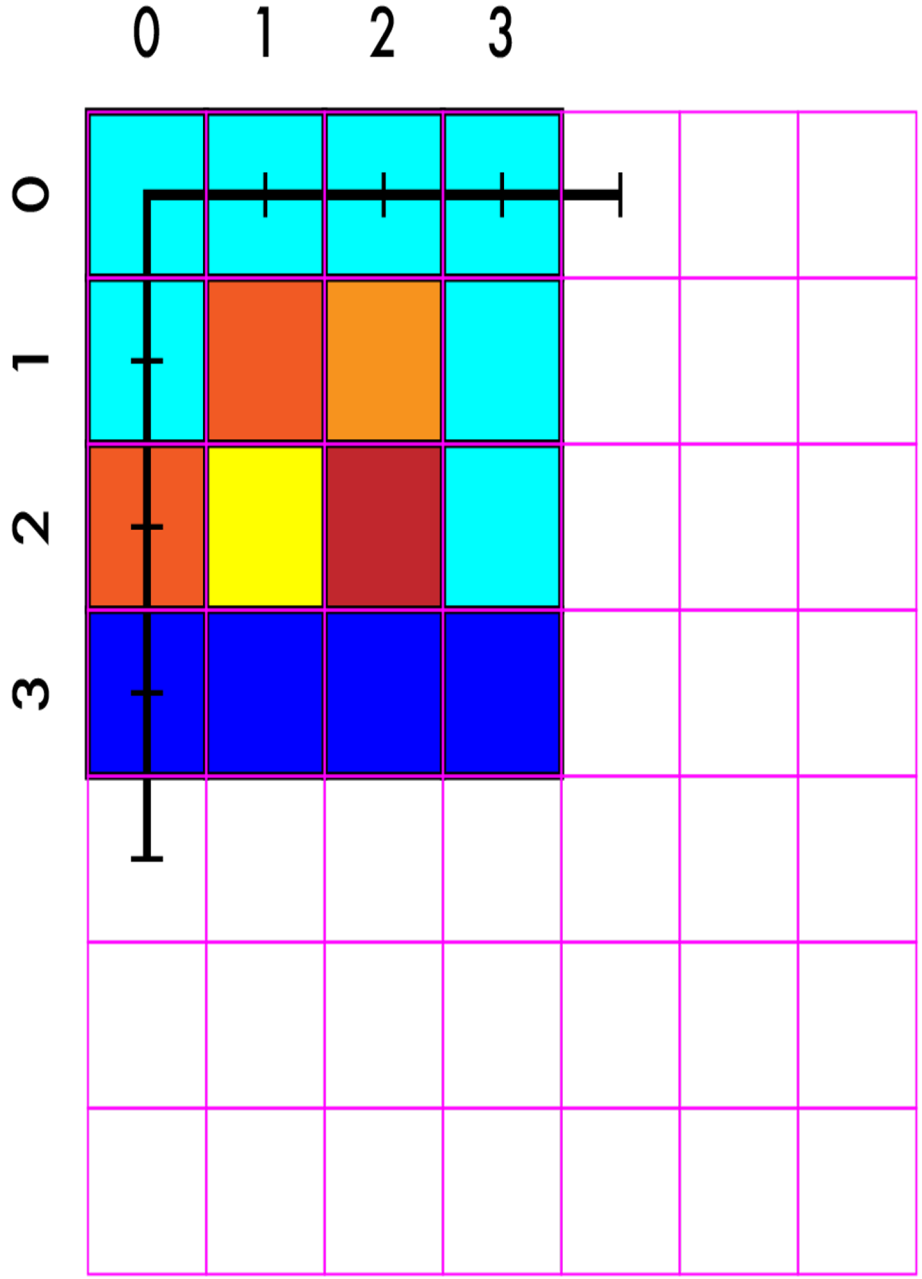
This is a beautiful image of a sunset… it's just very small…

Say we want to increase size 4 x 4 to 7 x 7



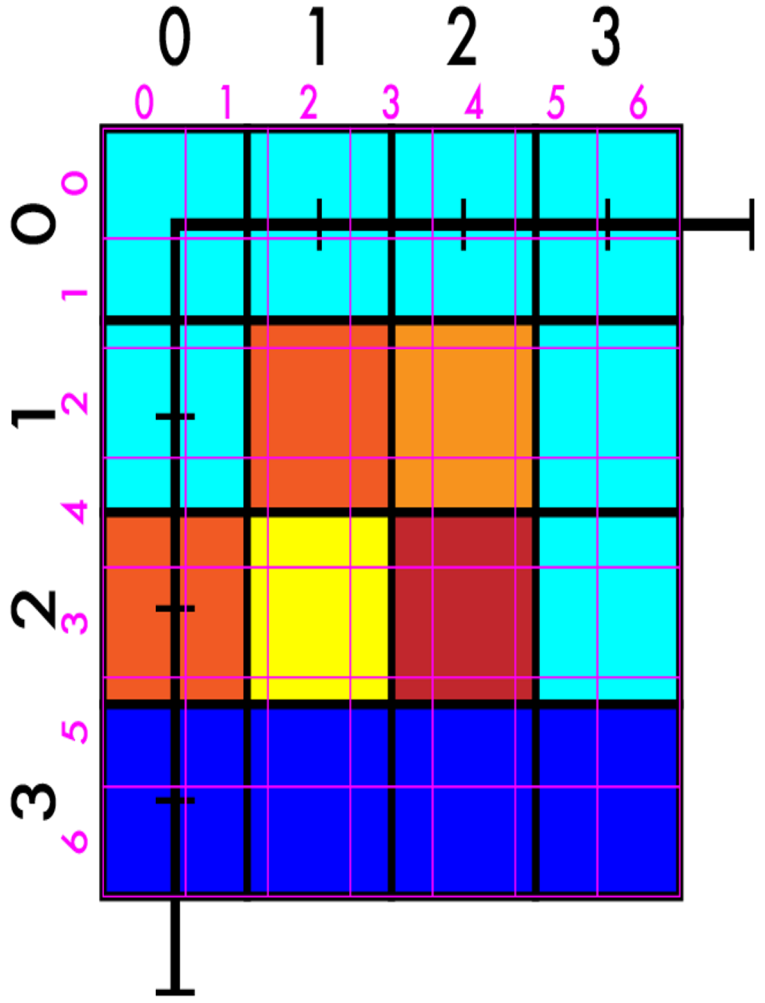
Resize 4 x 4 to 7 x 7

- Create our new image



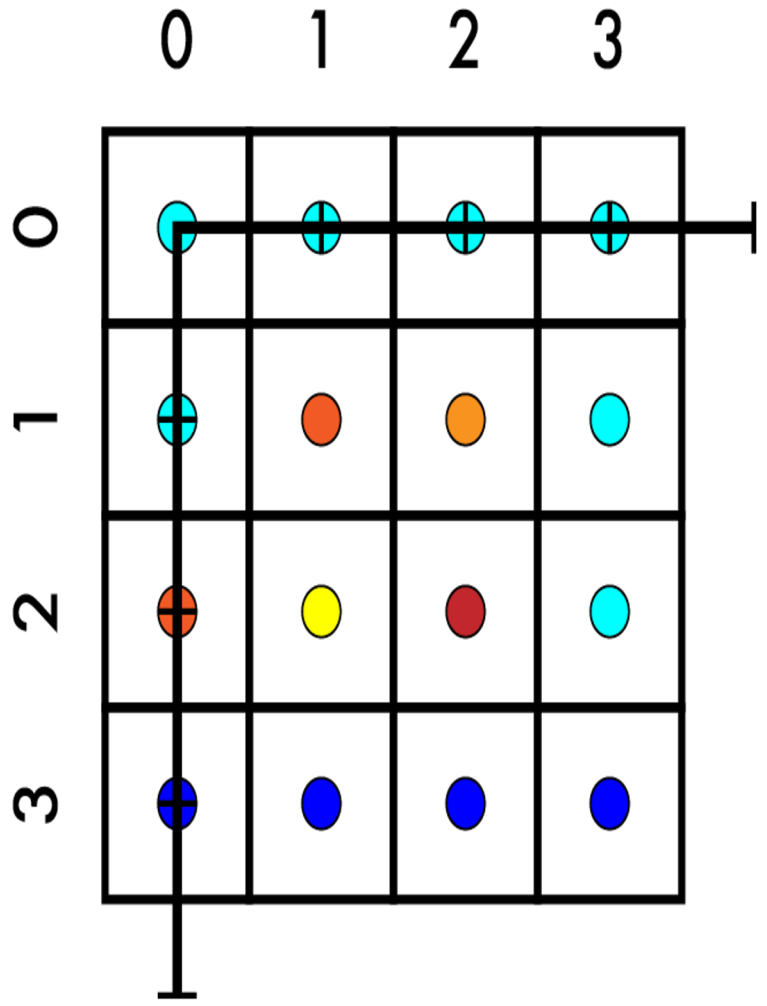
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates



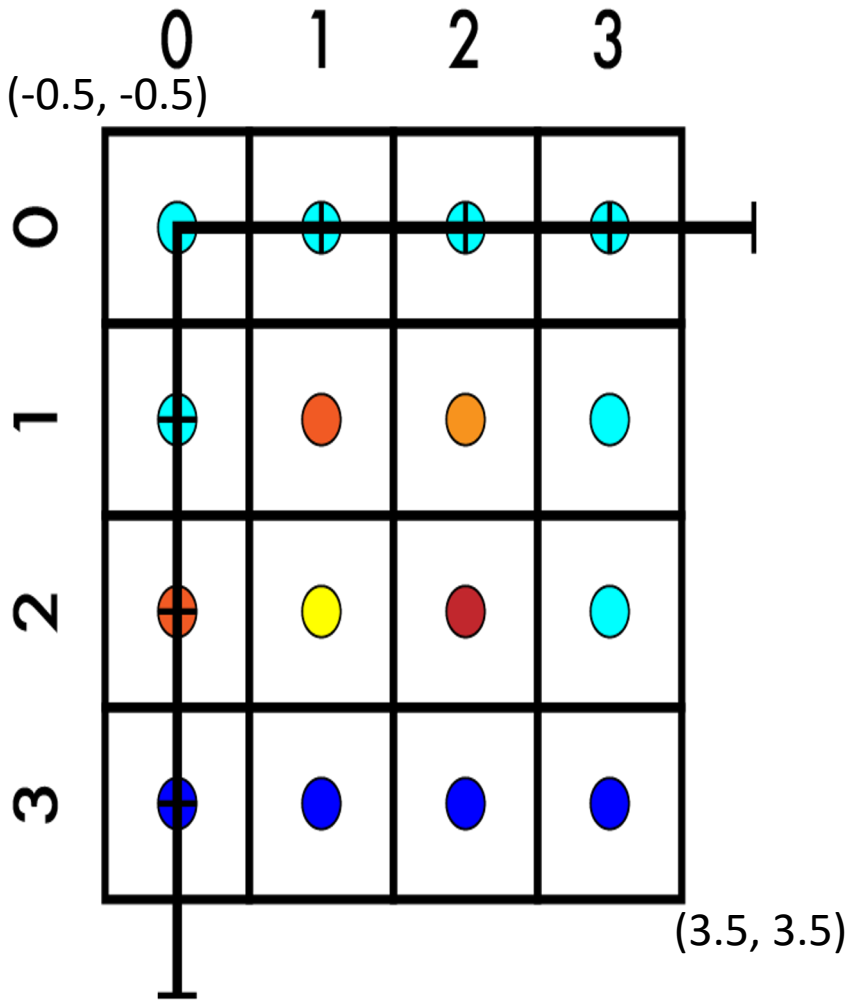
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates



Resize 4×4 to 7×7

- Create our new image
- Match up coordinates

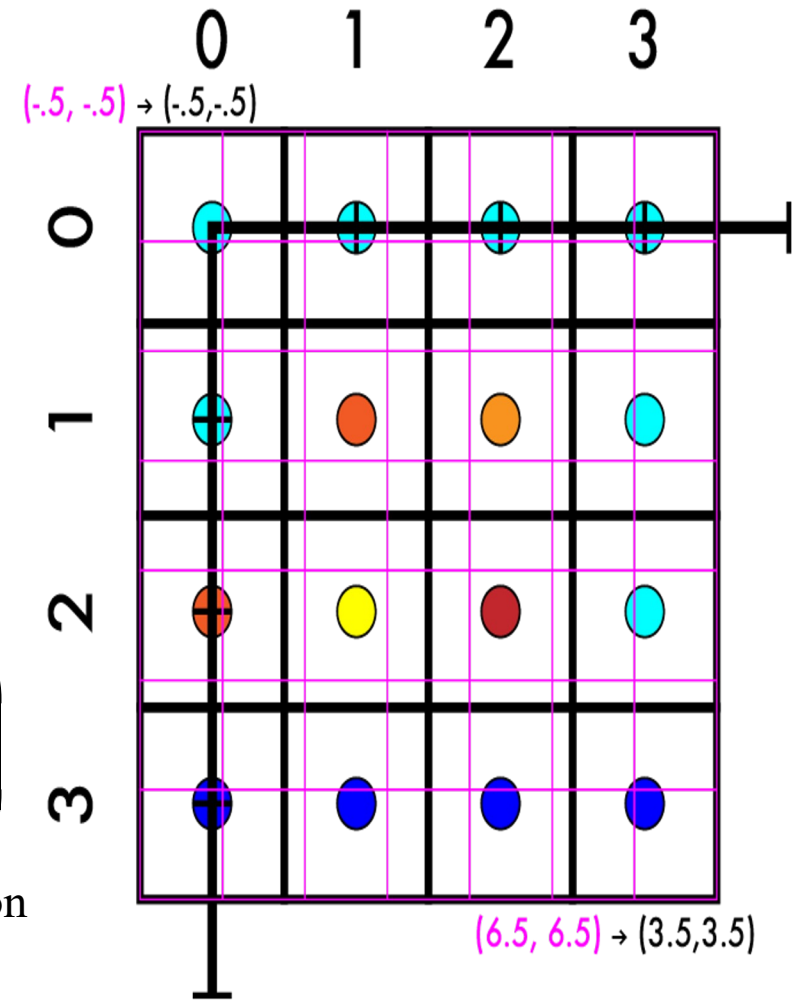


Match up coordinates

- Find the 2D transform that maps the left corner of the first image to the left corner of the second image + the right corner of the first image to the right corner of the second image (or the other way round)

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

scaling translation



Match up coordinates

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

scaling translation

$$\begin{pmatrix} -0.5 \\ -0.5 \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} -0.5 \\ -0.5 \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Left corner 1st image

Left corner 2nd image

$$\begin{pmatrix} 3.5 \\ 3.5 \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} 6.5 \\ 6.5 \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Right corner 1st image

Right corner 2nd image

$$\begin{cases} -s_x*0.5 + t_x = -0.5 \quad (1) \\ -s_y*0.5 + t_y = -0.5 \quad (2) \\ s_x*6.5 + t_x = 3.5 \quad (3) \\ s_y*6.5 + t_y = 3.5 \quad (4) \end{cases}$$

4 equations with 4 unknowns

Match up coordinates

$$\begin{cases} -s_x*0.5 + t_x = -0.5 & (1) \\ -s_y*0.5 + t_y = -0.5 & (2) \\ s_x*6.5 + t_x = 3.5 & (3) \\ s_y*6.5 + t_y = 3.5 & (4) \end{cases}$$

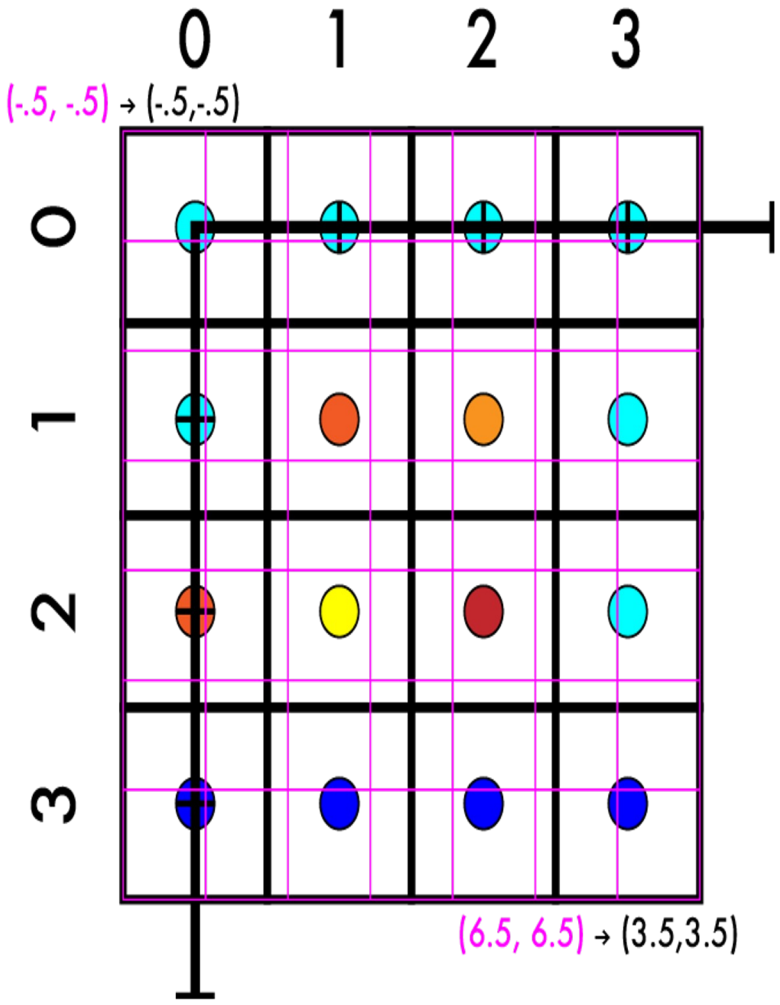
(3) – (1): $7*s_x = 4$, so $s_x = 4/7$

from (1): $t_x = -0.5 + 4/7*0.5$, so $t_x = -3/14$

(4) – (2): $4*s_y = 7$, so $s_y = 4/7$

from (2): $t_y = -0.5 + 4/7*0.5$, so $t_y = -3/14$

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \frac{4}{7} & 0 \\ 0 & \frac{4}{7} \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} -\frac{3}{14} \\ -\frac{3}{14} \end{pmatrix}$$

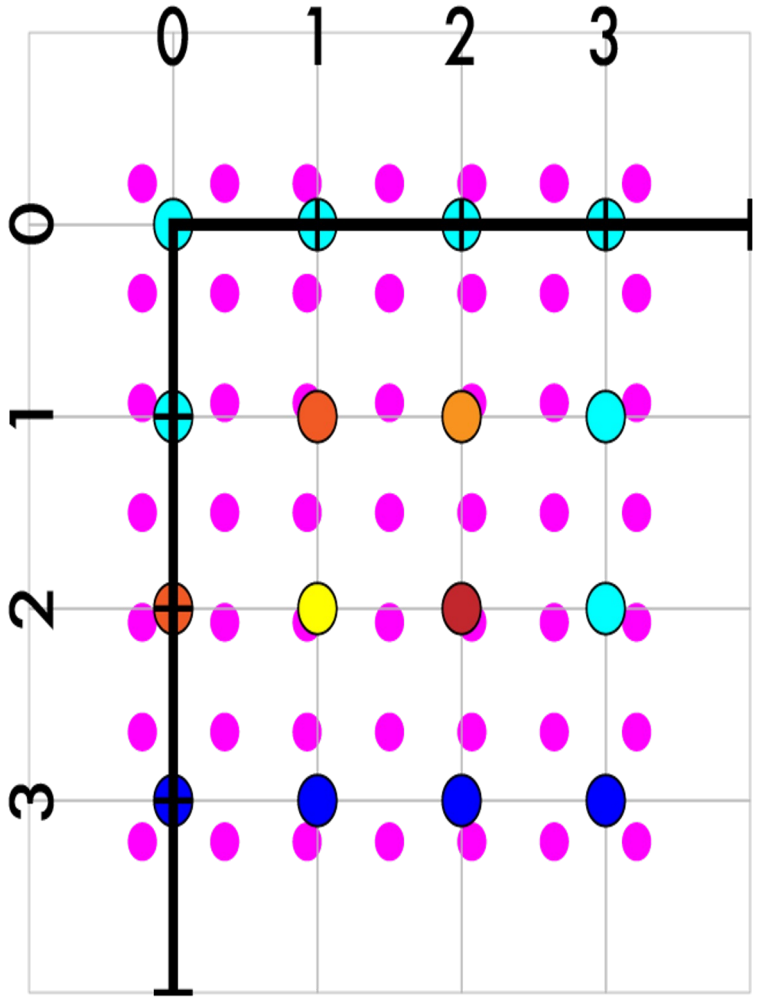


Resize 4×4 to 7×7

- Create our new image
- Match up coordinates

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \frac{4}{7} & 0 \\ 0 & \frac{4}{7} \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} -\frac{3}{14} \\ -\frac{3}{14} \end{pmatrix}$$

- Iterate over new points

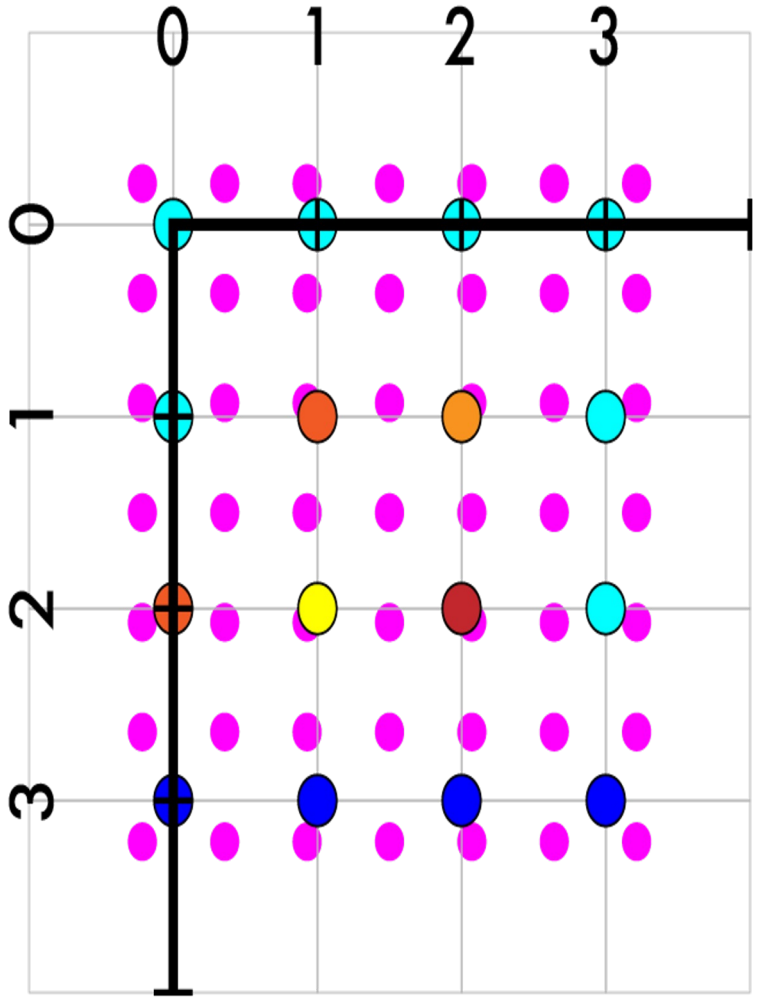


Resize 4×4 to 7×7

- Create our new image
- Match up coordinates

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \frac{4}{7} & 0 \\ 0 & \frac{4}{7} \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} -\frac{3}{14} \\ -\frac{3}{14} \end{pmatrix}$$

- Iterate over new points
  - Map to old coordinates

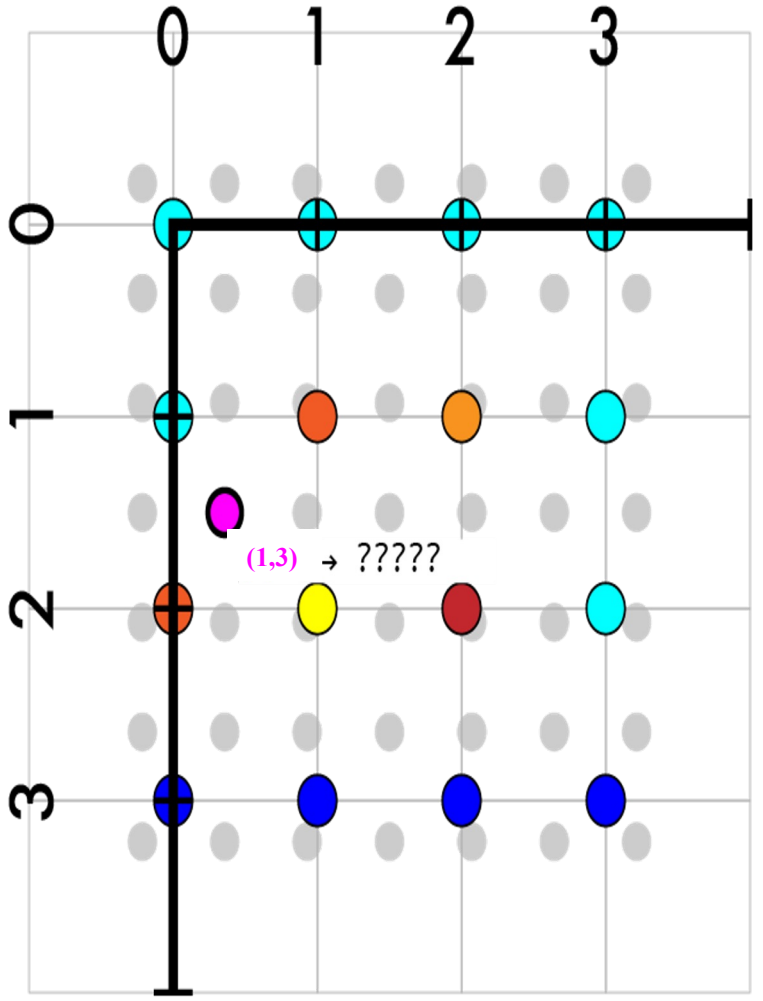


Resize 4×4 to 7×7

- Create our new image
- Match up coordinates

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \frac{4}{7} & 0 \\ 0 & \frac{4}{7} \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} -\frac{3}{14} \\ -\frac{3}{14} \end{pmatrix}$$

- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)



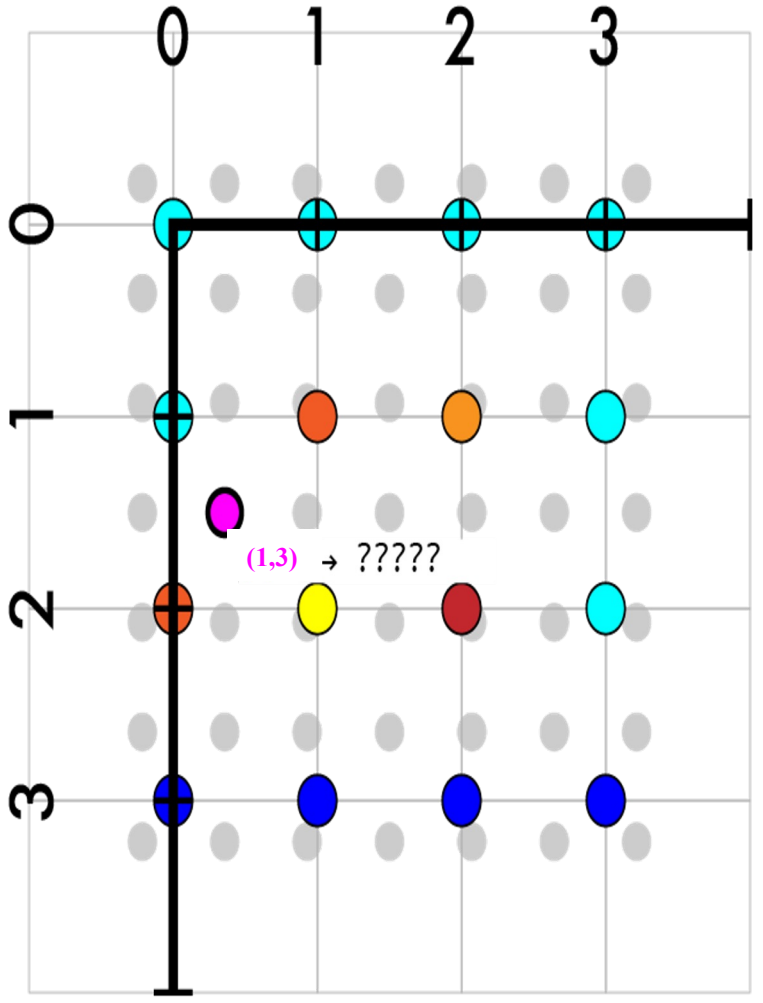
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \frac{4}{7} & 0 \\ 0 & \frac{4}{7} \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} -\frac{3}{14} \\ -\frac{3}{14} \end{pmatrix}$$

- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \frac{4}{7} & 0 \\ 0 & \frac{4}{7} \end{pmatrix} \begin{pmatrix} 1 \\ 3 \end{pmatrix} + \begin{pmatrix} -\frac{3}{14} \\ -\frac{3}{14} \end{pmatrix} = \begin{pmatrix} \frac{5}{14} \\ \frac{21}{14} \end{pmatrix}$$

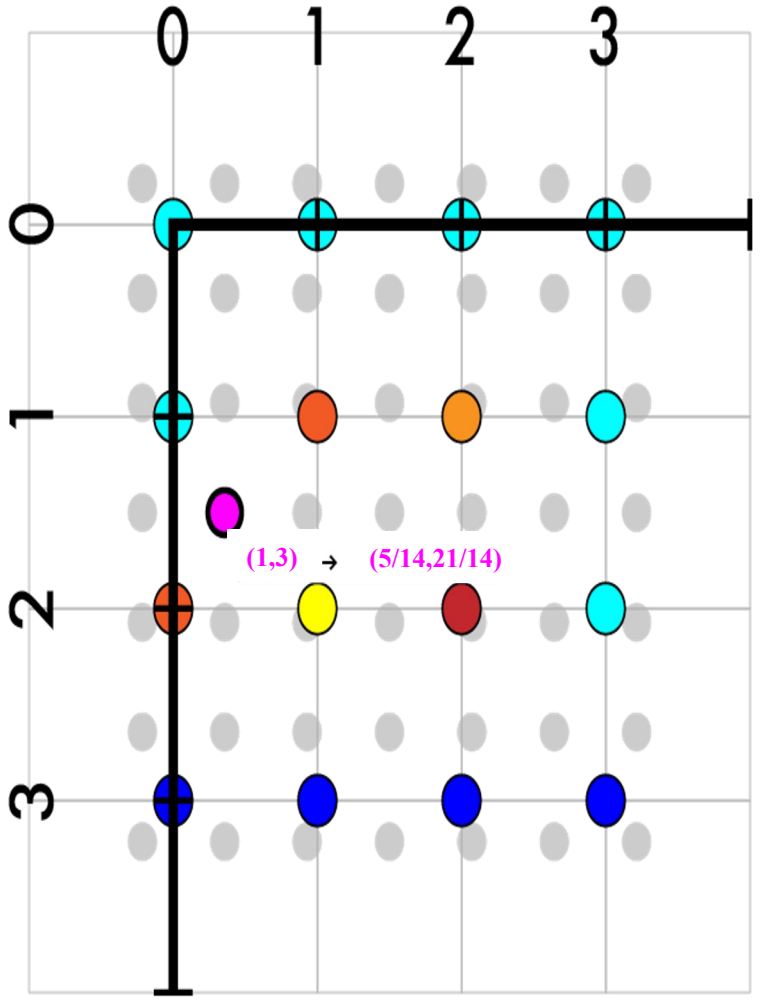


Resize 4×4 to 7×7

- Create our new image
- Match up coordinates

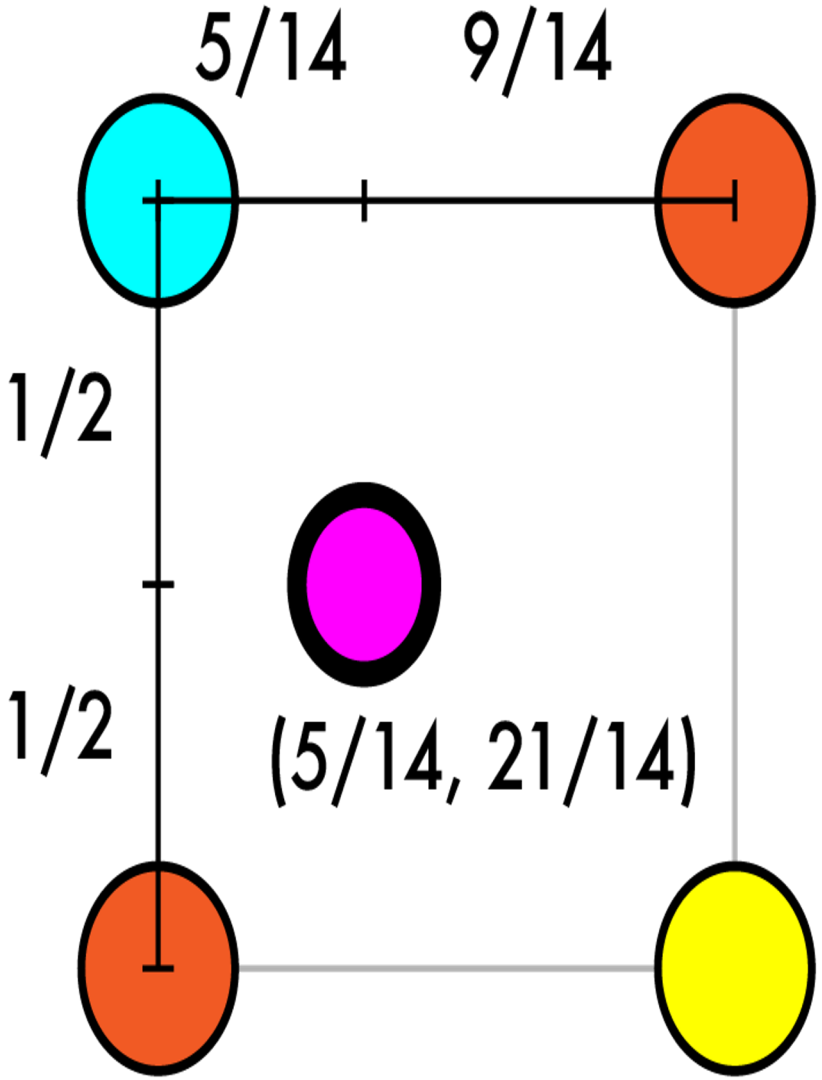
$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \frac{4}{7} & 0 \\ 0 & \frac{4}{7} \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} -\frac{3}{14} \\ -\frac{3}{14} \end{pmatrix}$$

- Iterate over new points
  - Map to old coordinates
  - Take point (1,3) (new image)
  - Map it to (5/14, 21/14) (old image)
  - Interpolate old values (bilinear)



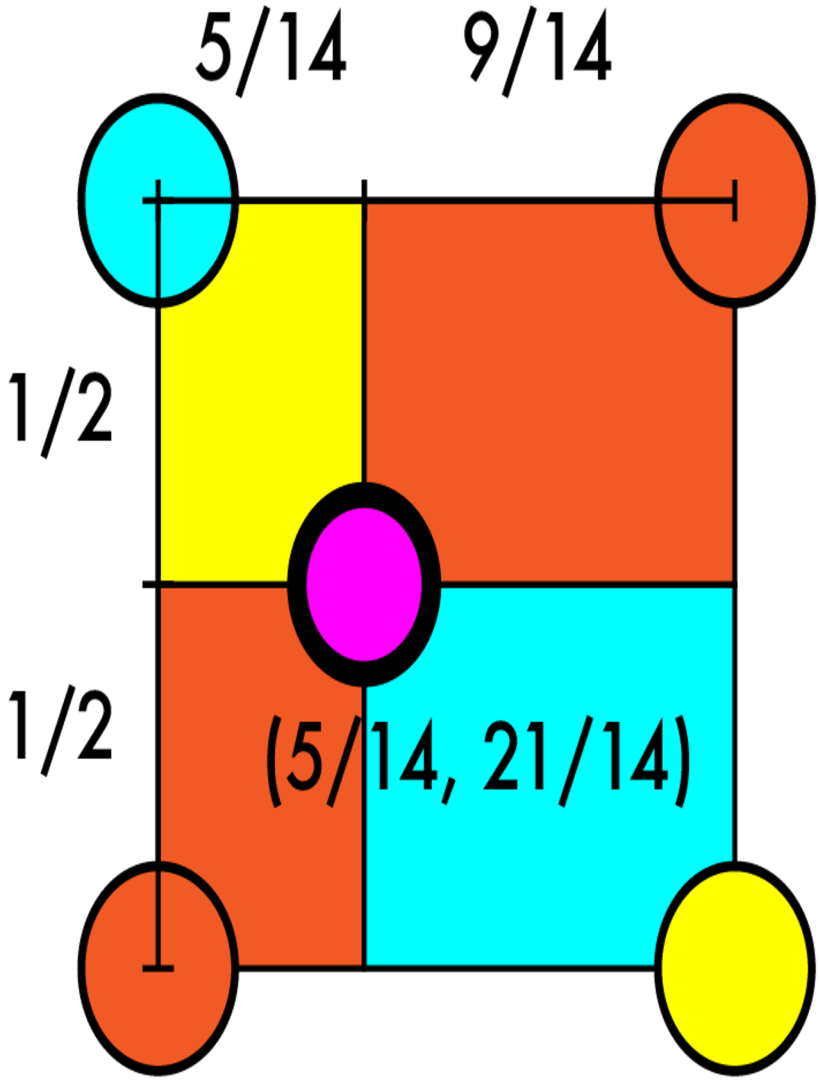
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values



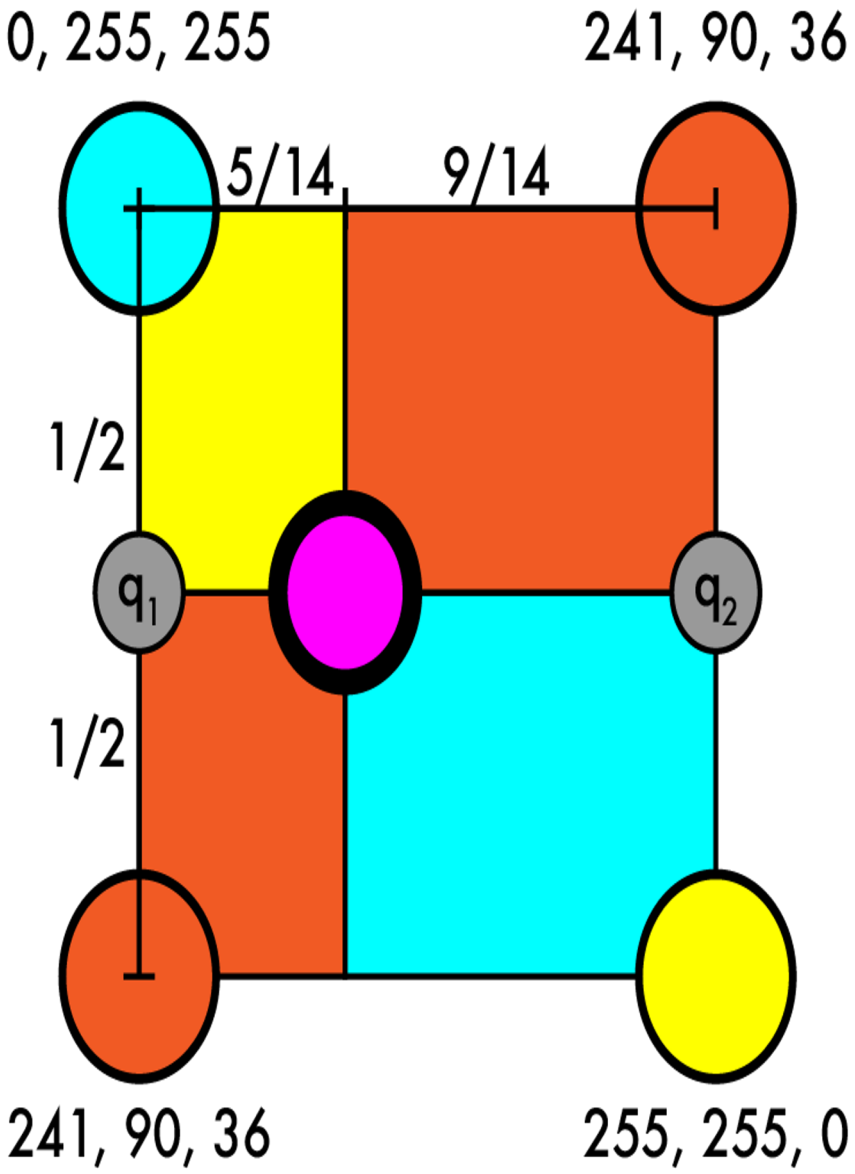
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - Size of opposite rectangles



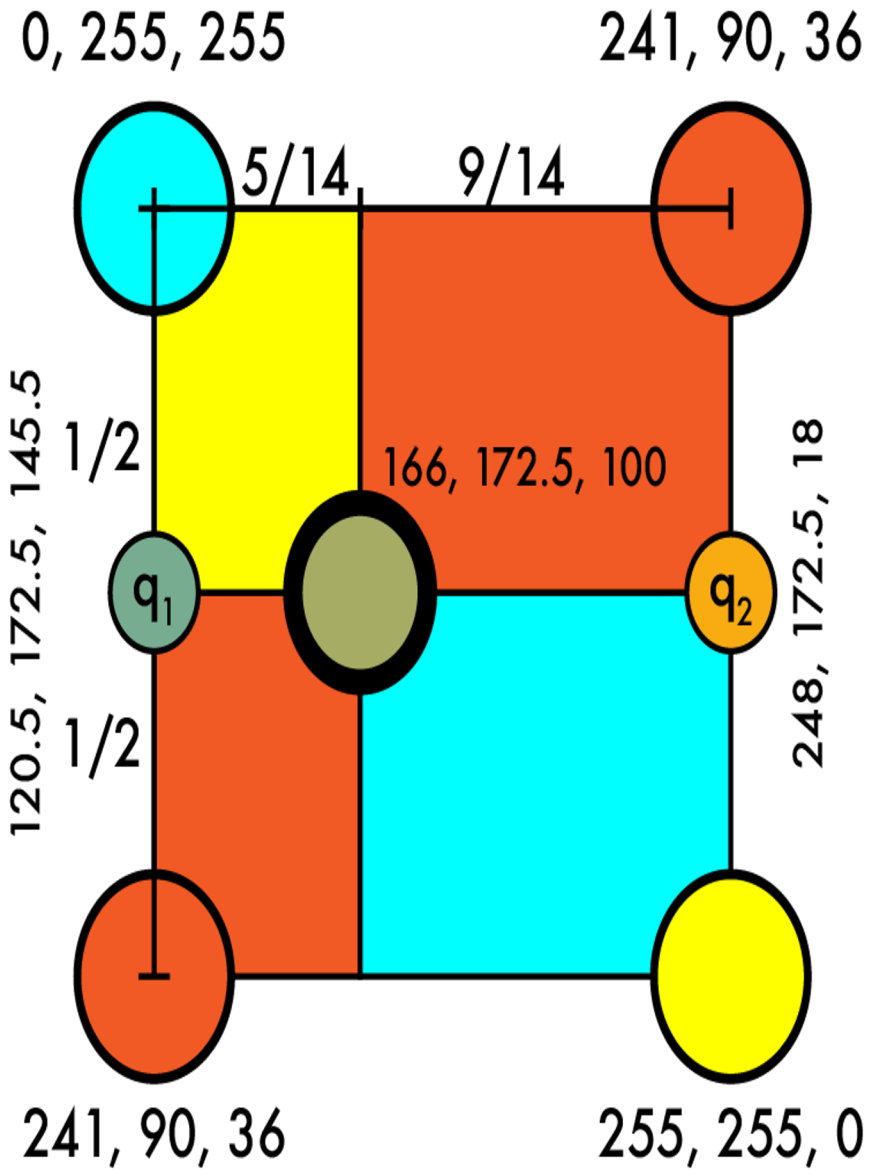
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - Size of opposite rectangles
    - OR find q1 and q2, then interpolate between them



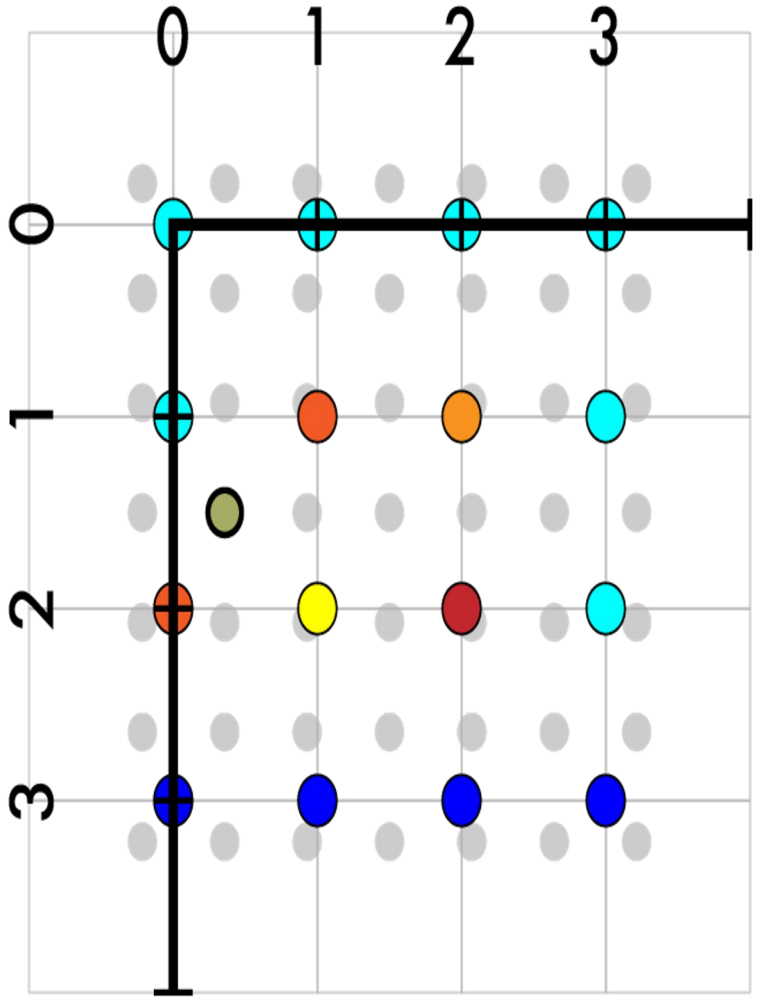
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - Size of opposite rectangles
    - OR find q1 and q2, then interpolate between them
      - q1 = (120.5, 172.5, 145.5)
      - q2 = (248, 172.5, 18)
      - q = (166, 172.5, 100)



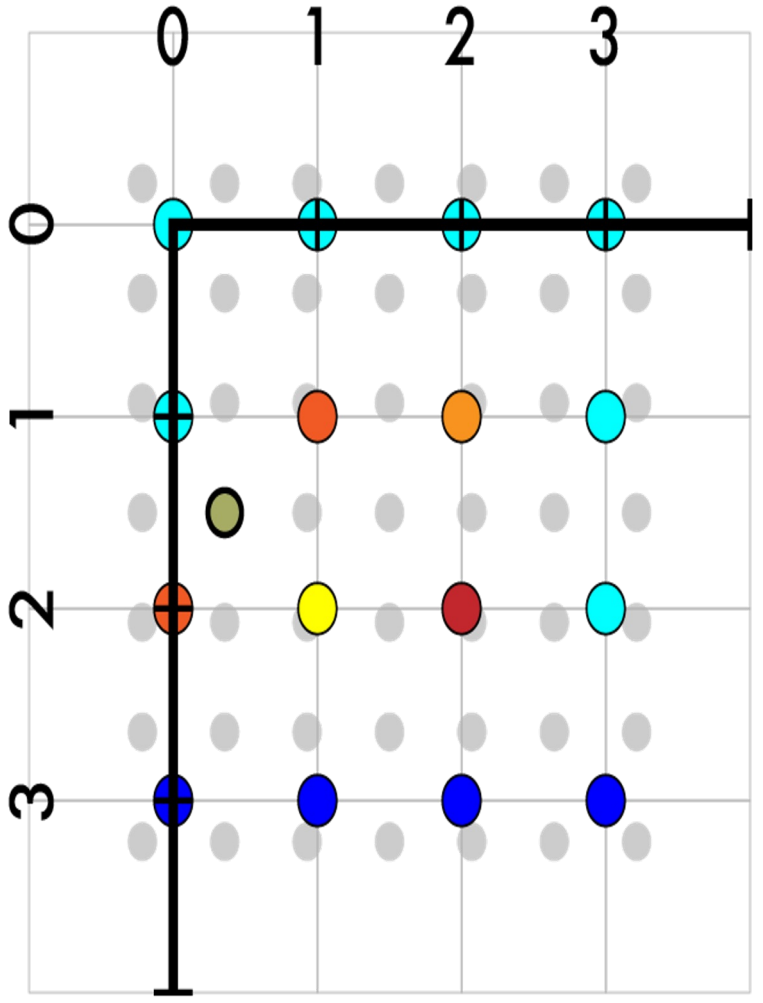
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - q = (166, 172.5, 100)



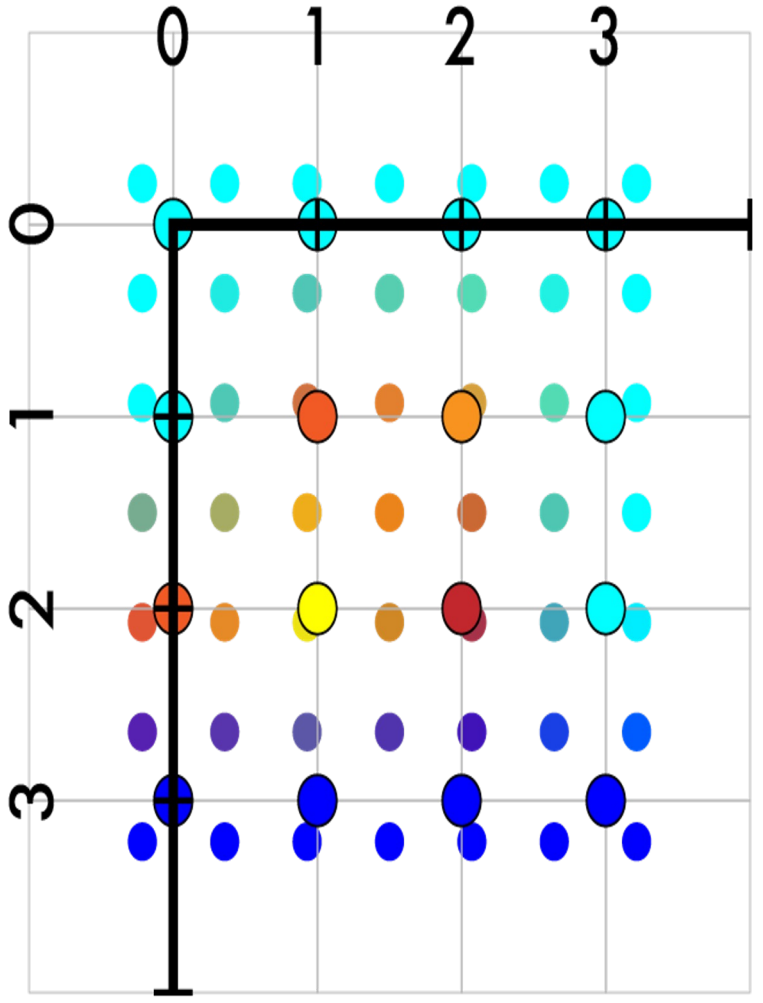
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - q = (166, 172.5, 100)
- Fill in the rest



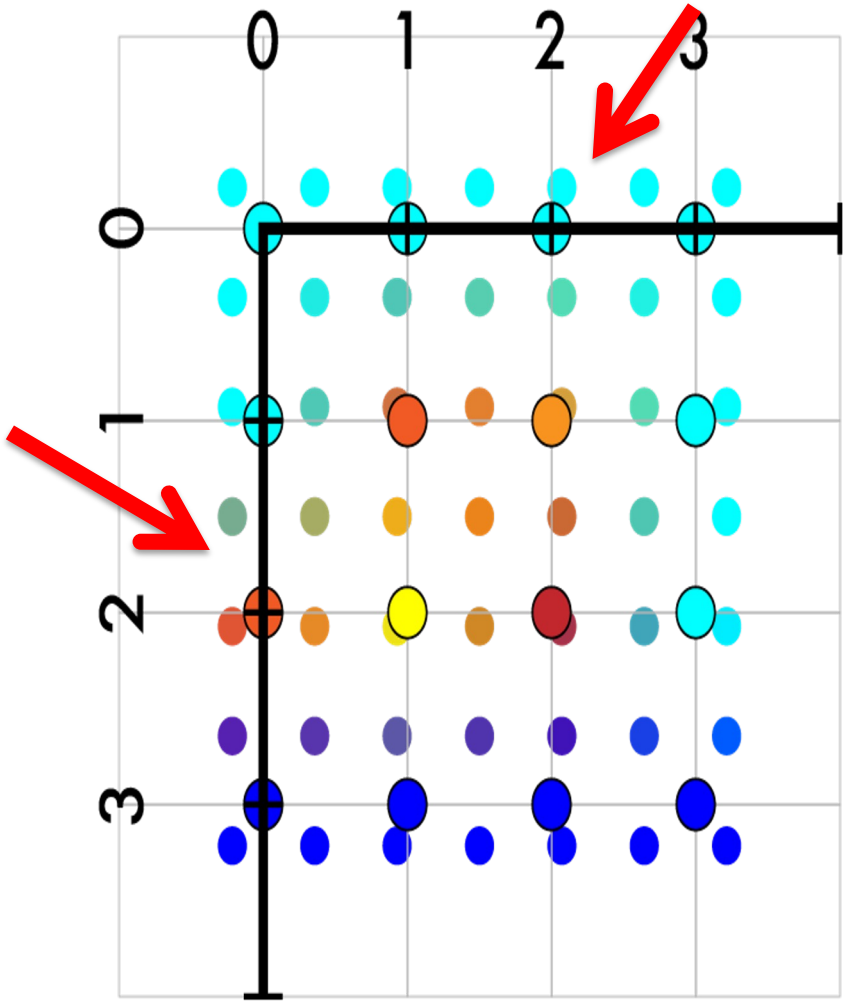
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - q = (166, 172.5, 100)
- Fill in the rest



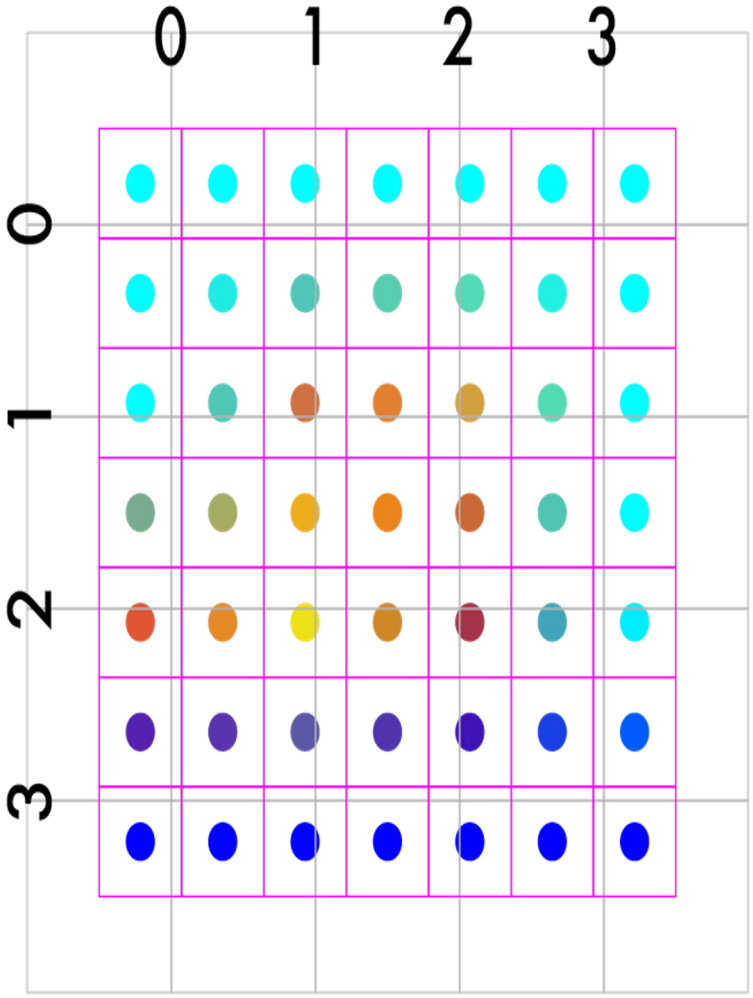
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - q = (166, 172.5, 100)
- Fill in the rest
  - On outer edges use padding



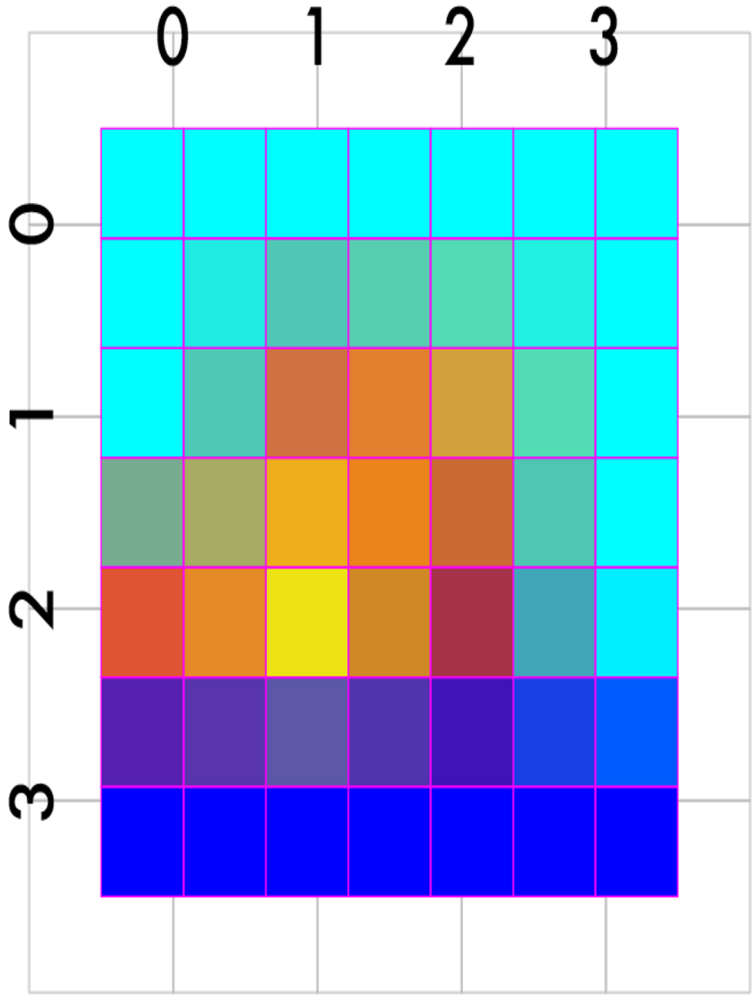
Resize 4×4 to 7×7

- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - q = (166, 172.5, 100)
- Fill in the rest

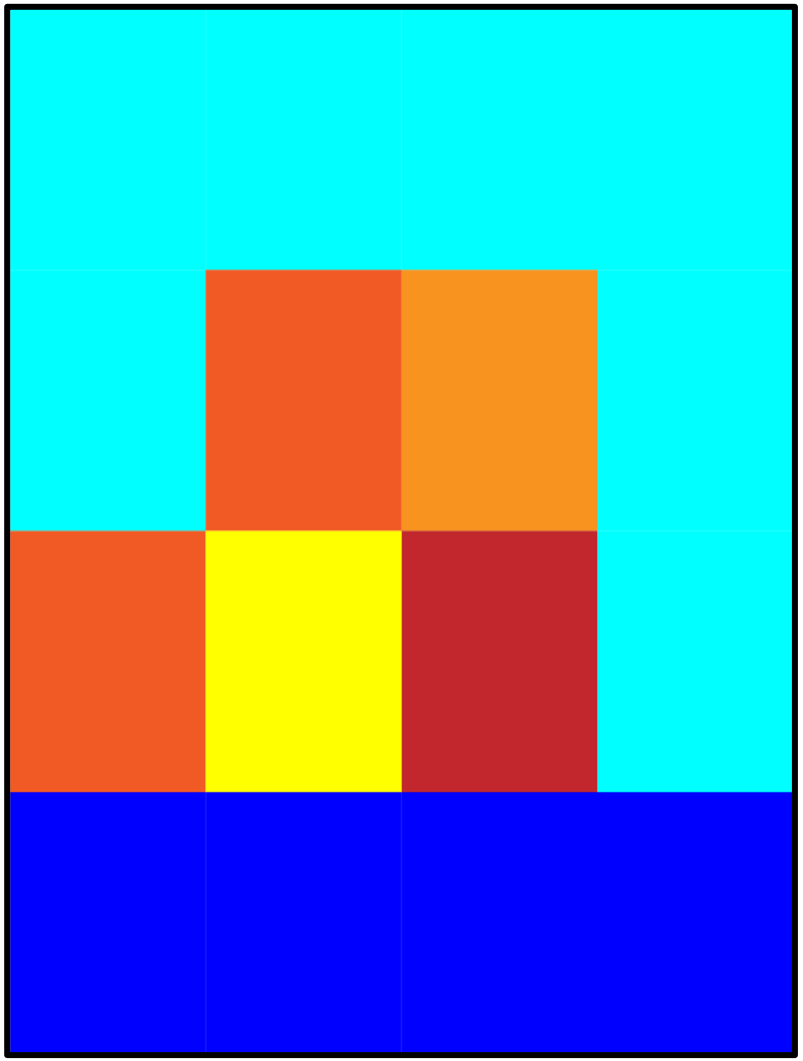


Resize 4×4 to 7×7

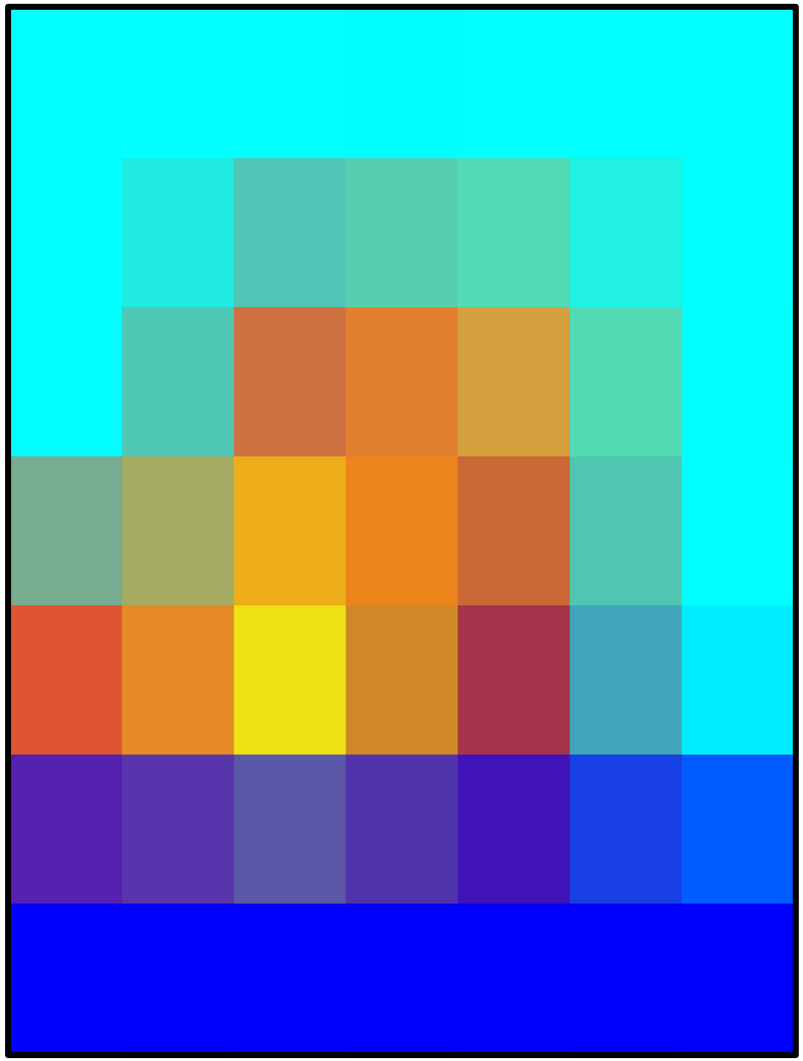
- Create our new image
- Match up coordinates
- Iterate over new points
  - Map to old coordinates
  - Take point (1,3)
  - Map it to (5/14, 21/14)
  - Interpolate old values
    - q = (166, 172.5, 100)
- Fill in the rest



Final result!

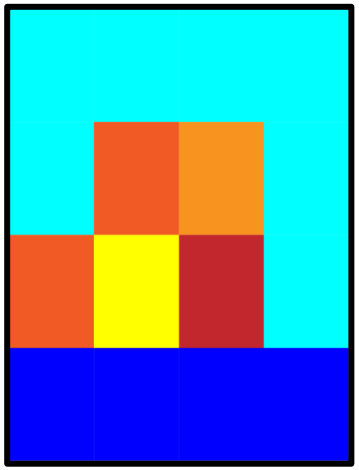


Original image 4 × 4

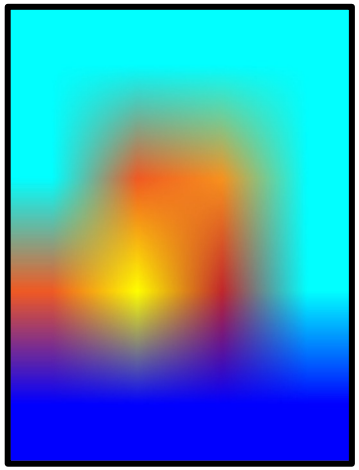


Resized image 7 × 7 using bilinear interpolation

Different scales

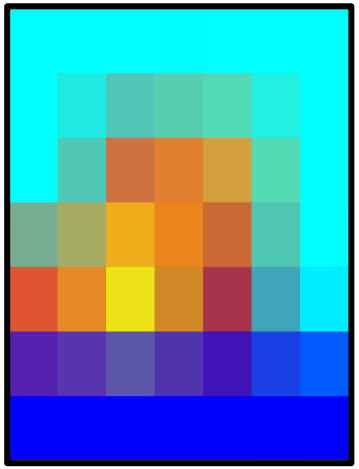


Original image 4 × 4

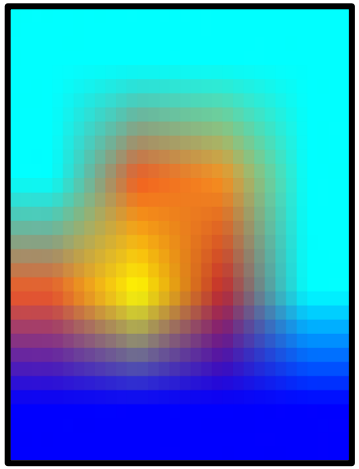


Resized image 256 × 256
using bilinear interpolation

→ Star like pattern

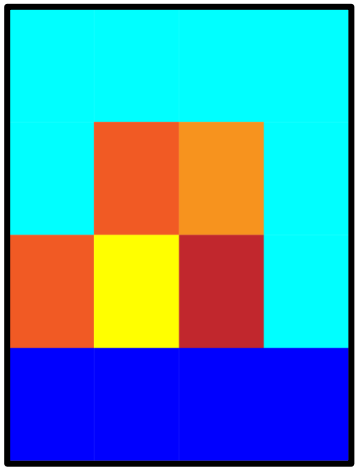


Resized image 7 × 7 using
bilinear interpolation

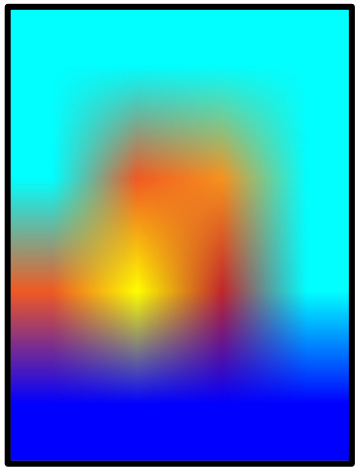


Resized image 32 × 32
using bilinear interpolation

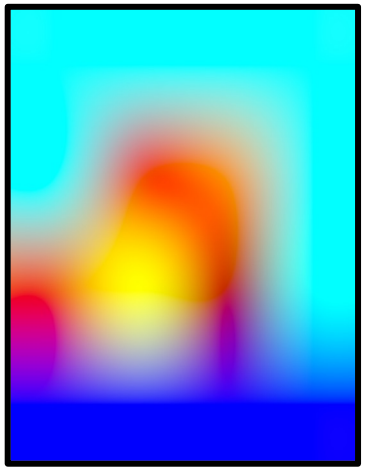
Different methods



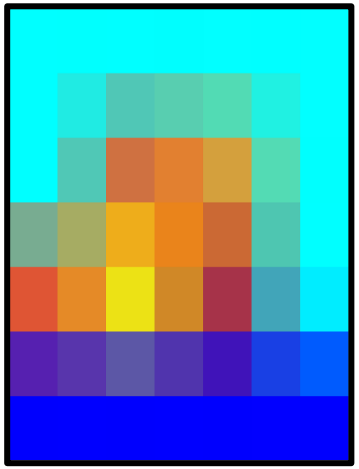
Original image 4 × 4



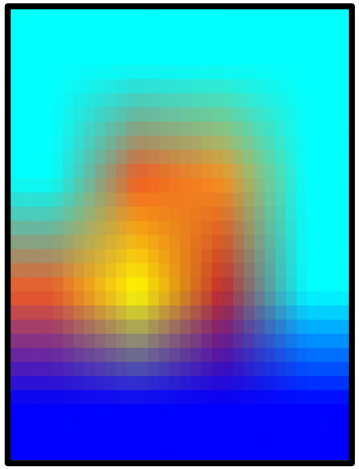
Resized image 256 × 256
using bilinear interpolation



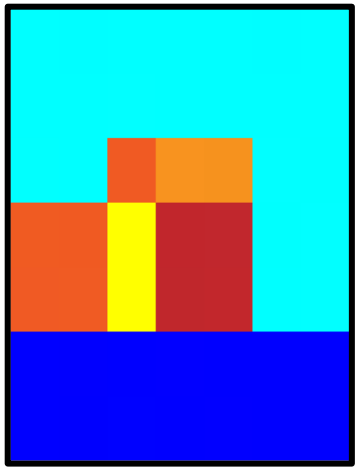
Resized image 256 × 256
using bicubic interpolation



Resized image 7 × 7 using
bilinear interpolation



Resized image 32 × 32
using bilinear interpolation



Resized image 7 × 7 using
nearest neighbor interpolation

Want to make image smaller

From 448×448 to 64×64

nearest neighbor
interpolation

bilinear interpolation

From 448×448 to 64×64

nearest neighbor
interpolation

bilinear interpolation

From 448×448 to 64×64

nearest neighbor
interpolation

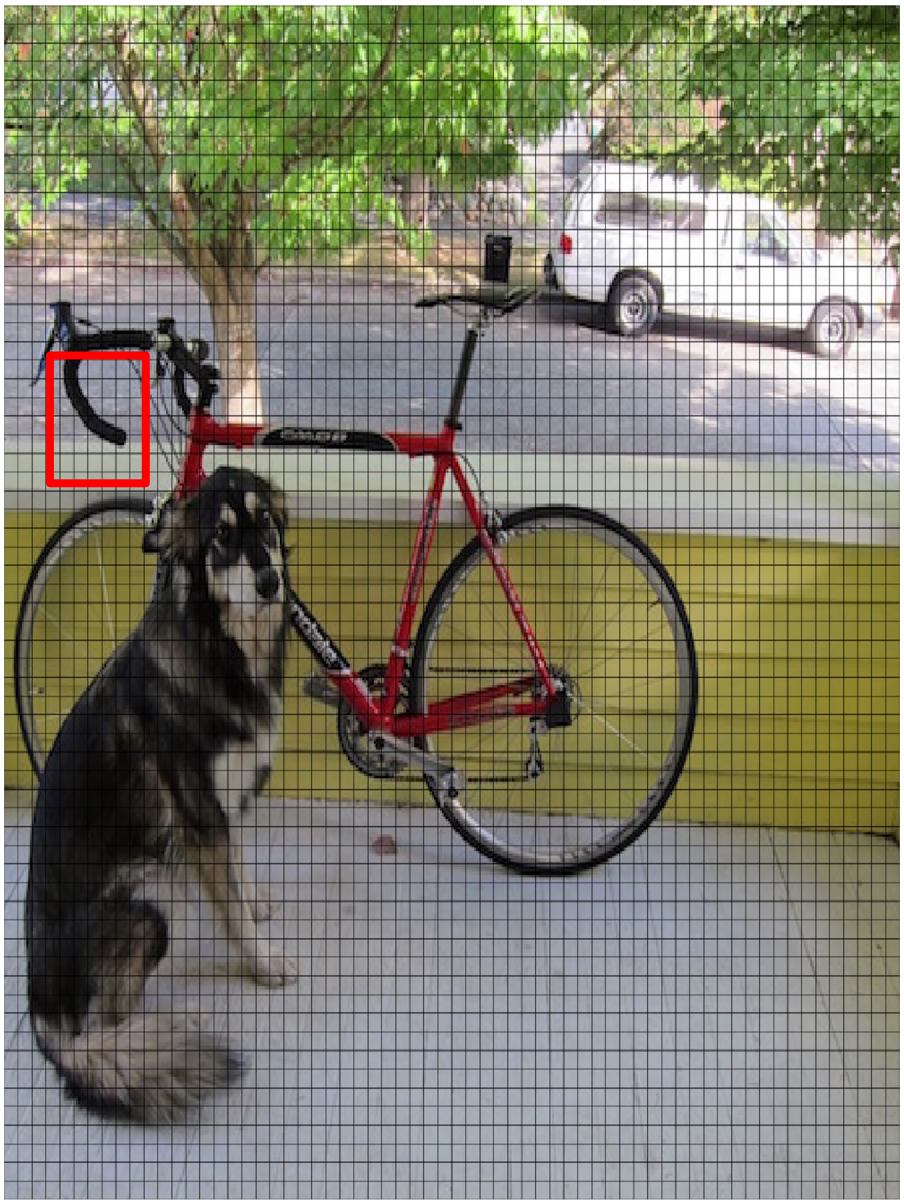
bilinear interpolation

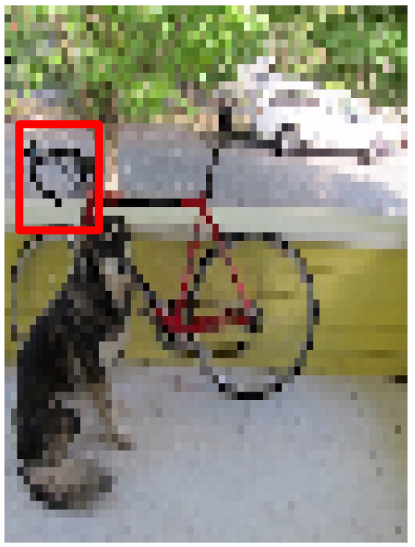
From 448×448 to 64×64

nearest neighbor
interpolation

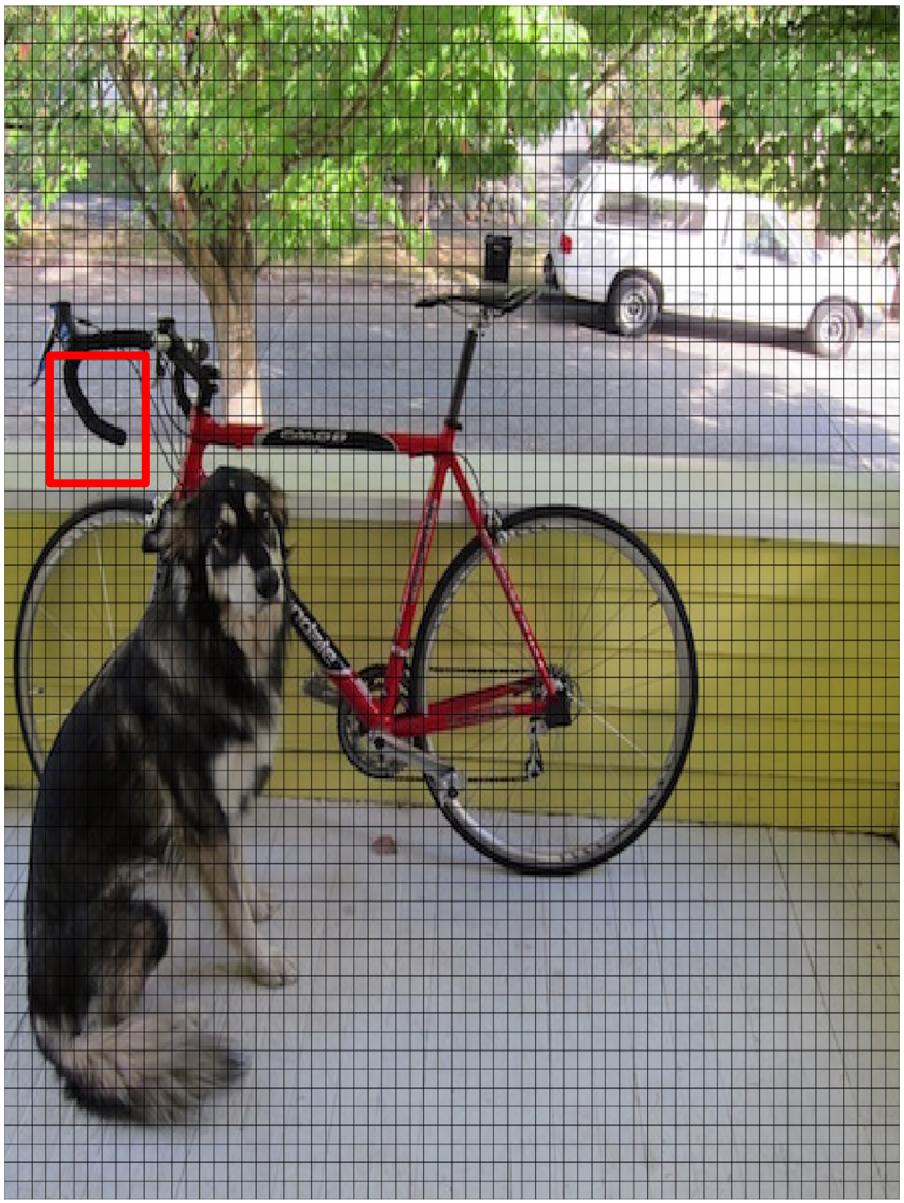
bilinear interpolation

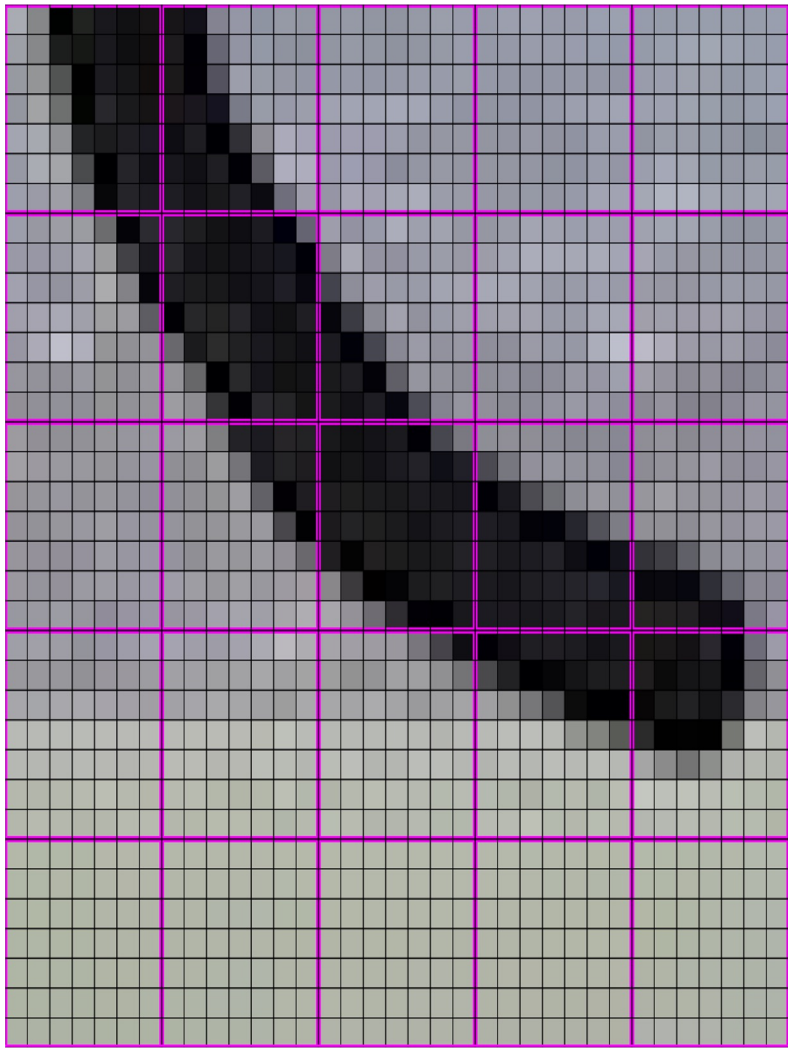
From 448×448 to 64×64



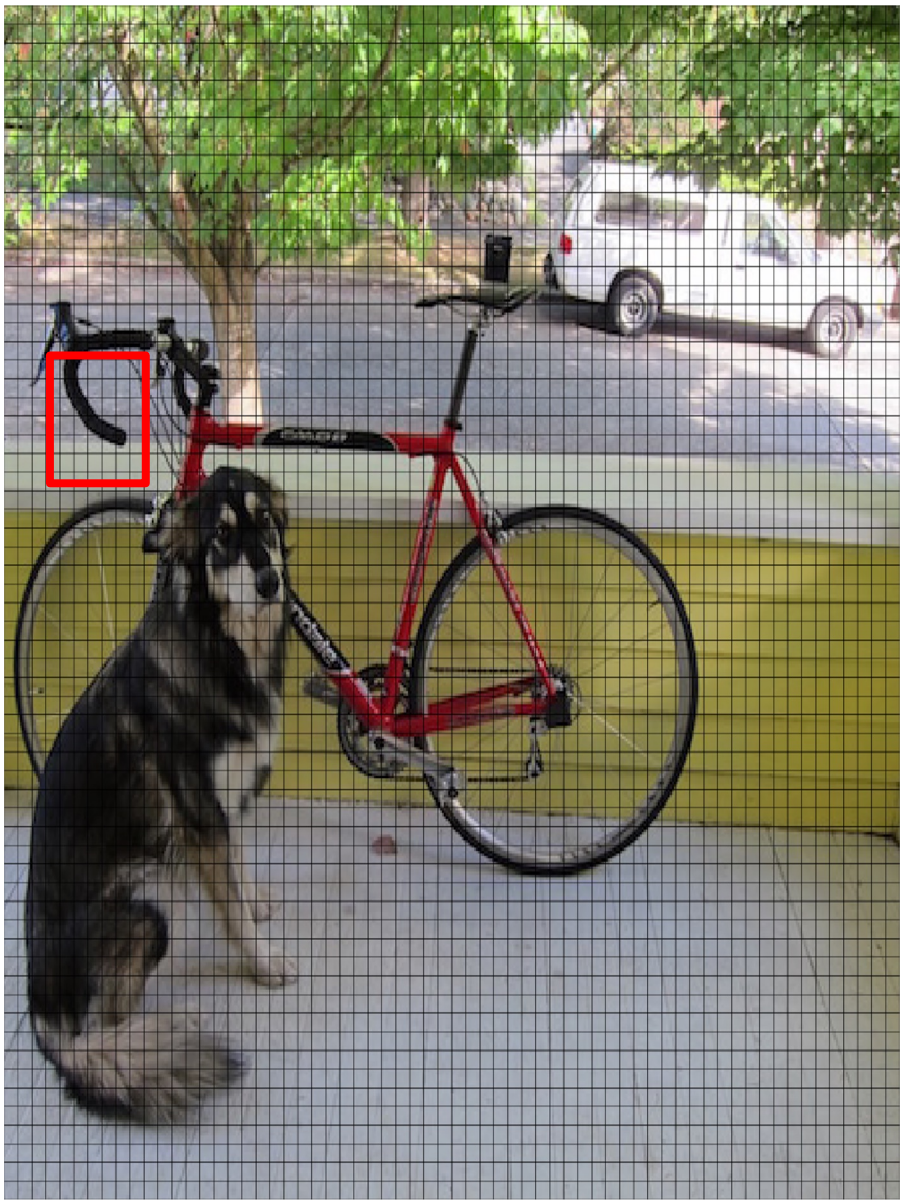


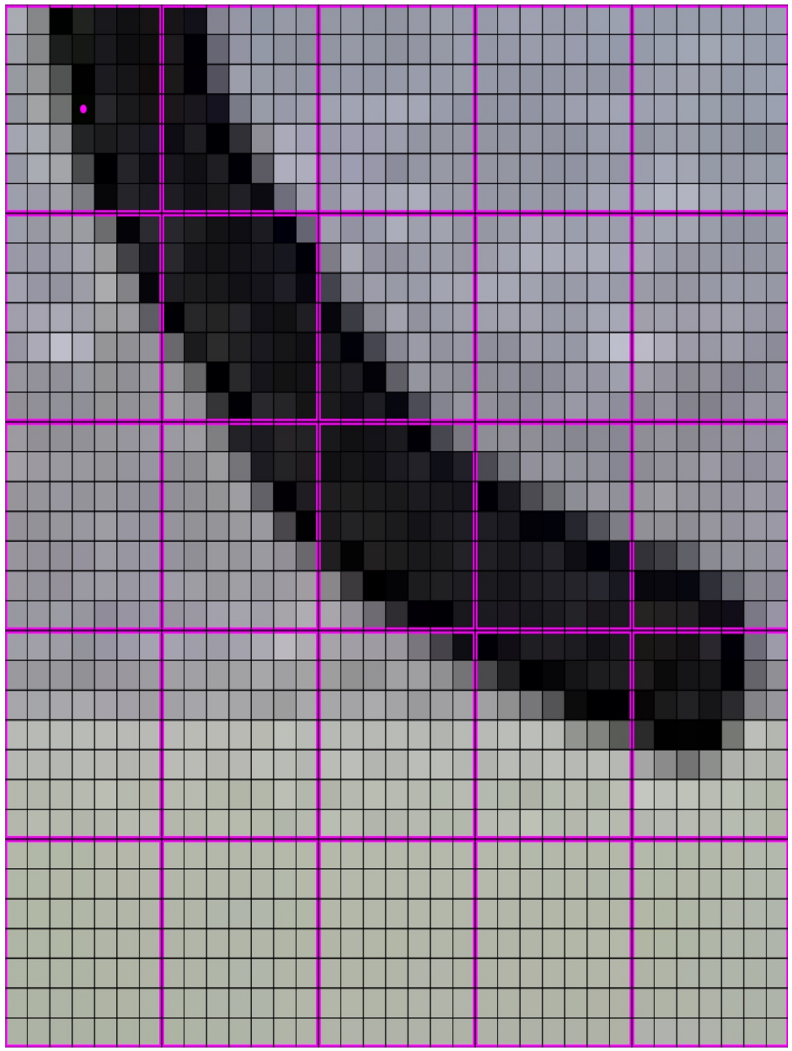
From 448×448 to 64×64



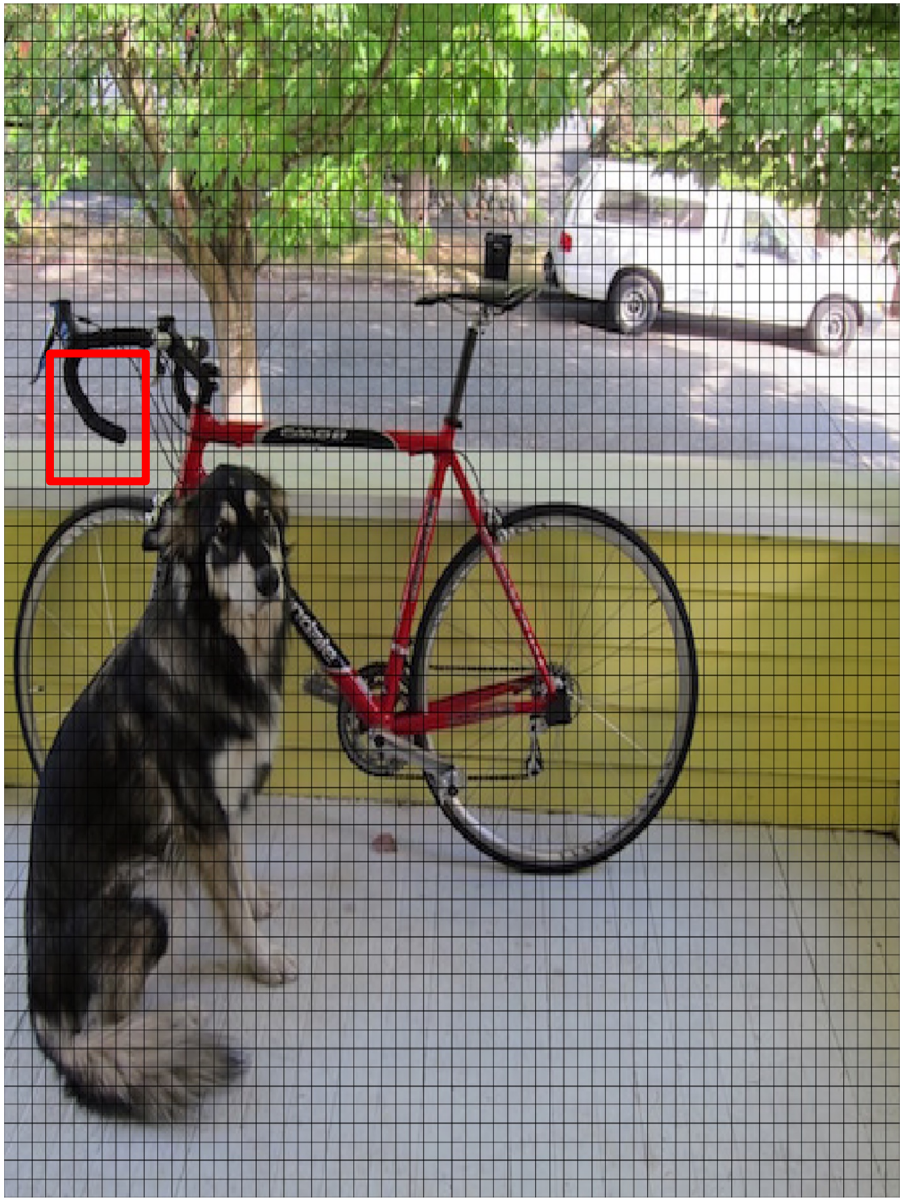


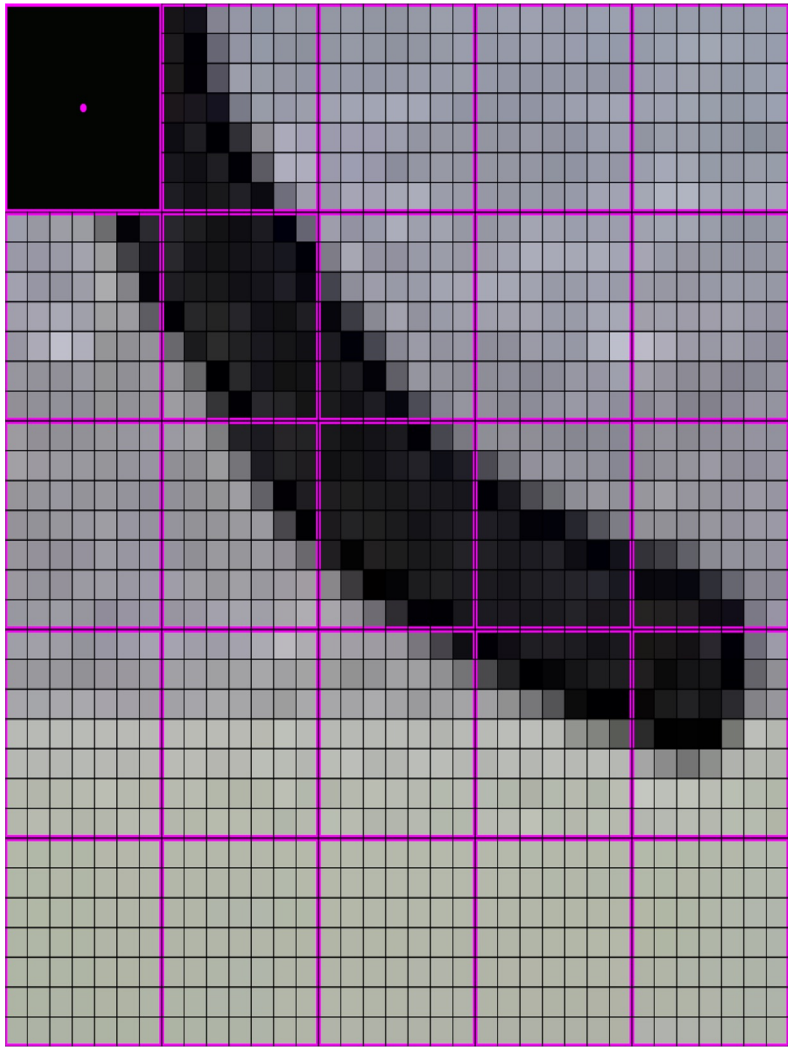
From 448×448 to 64×64



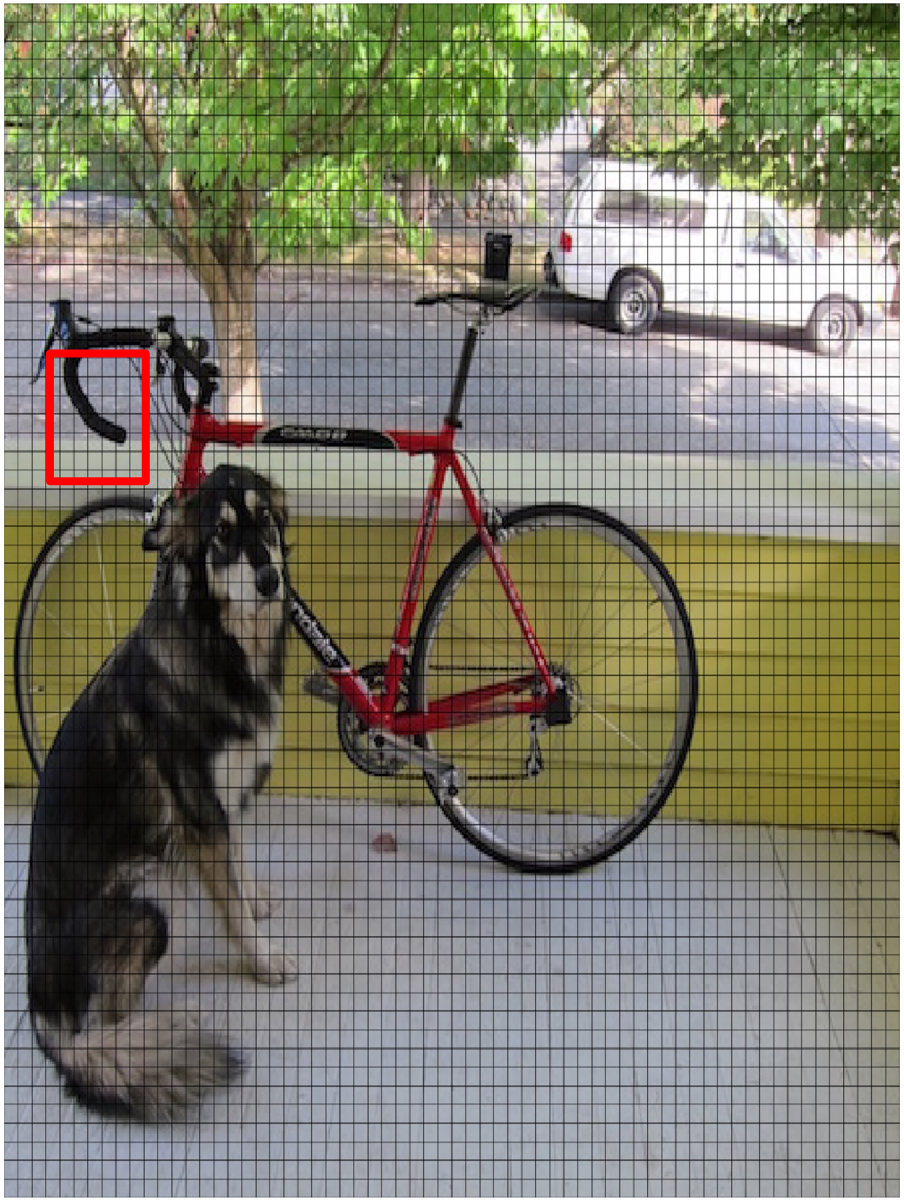


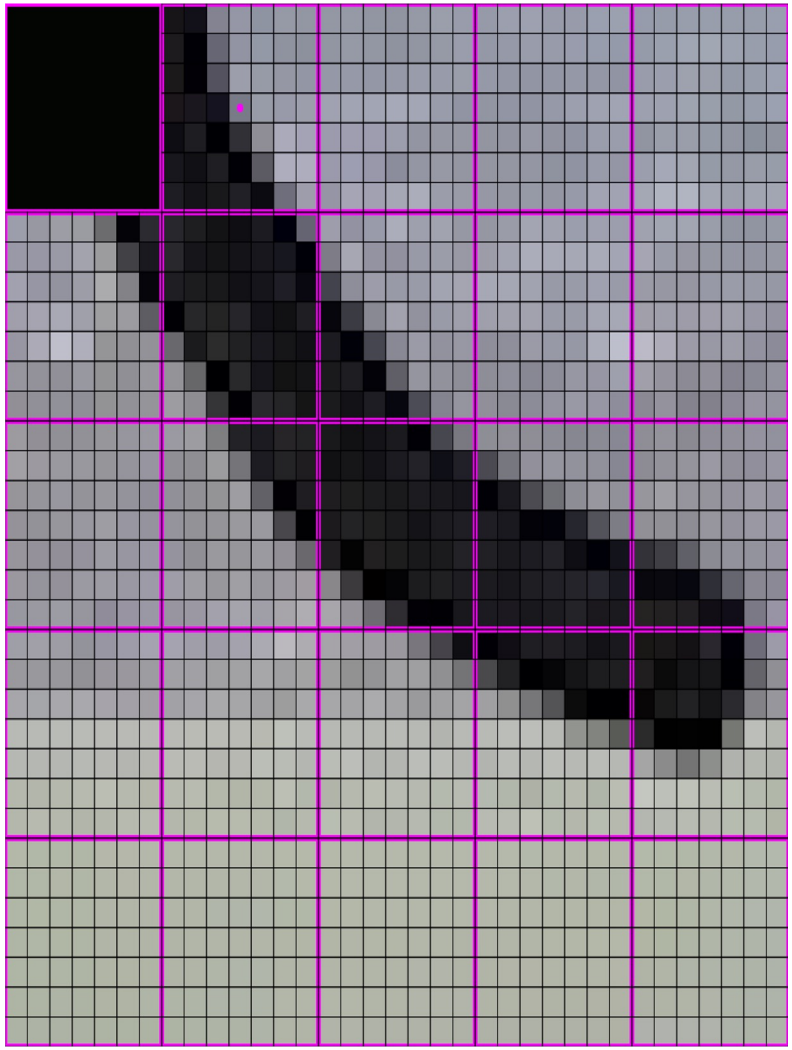
From 448×448 to 64×64



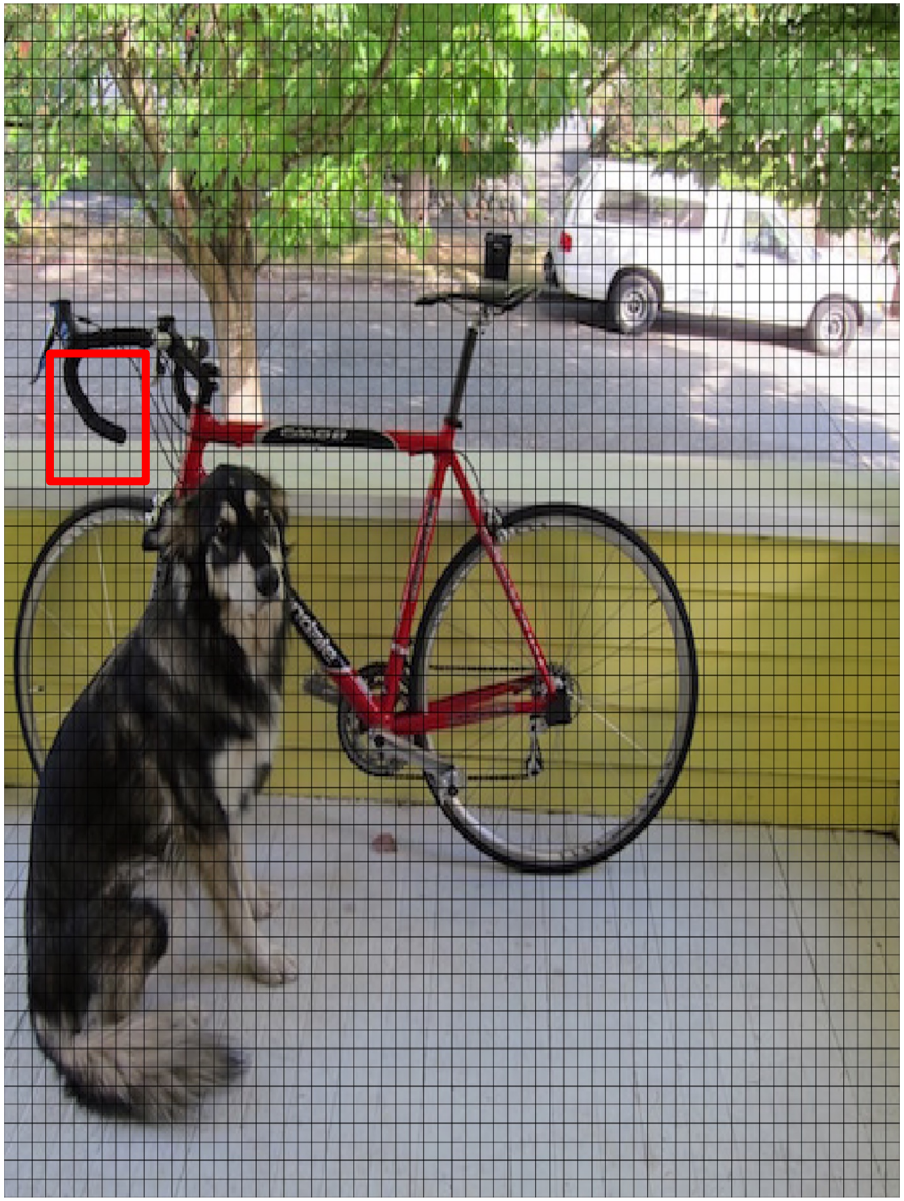


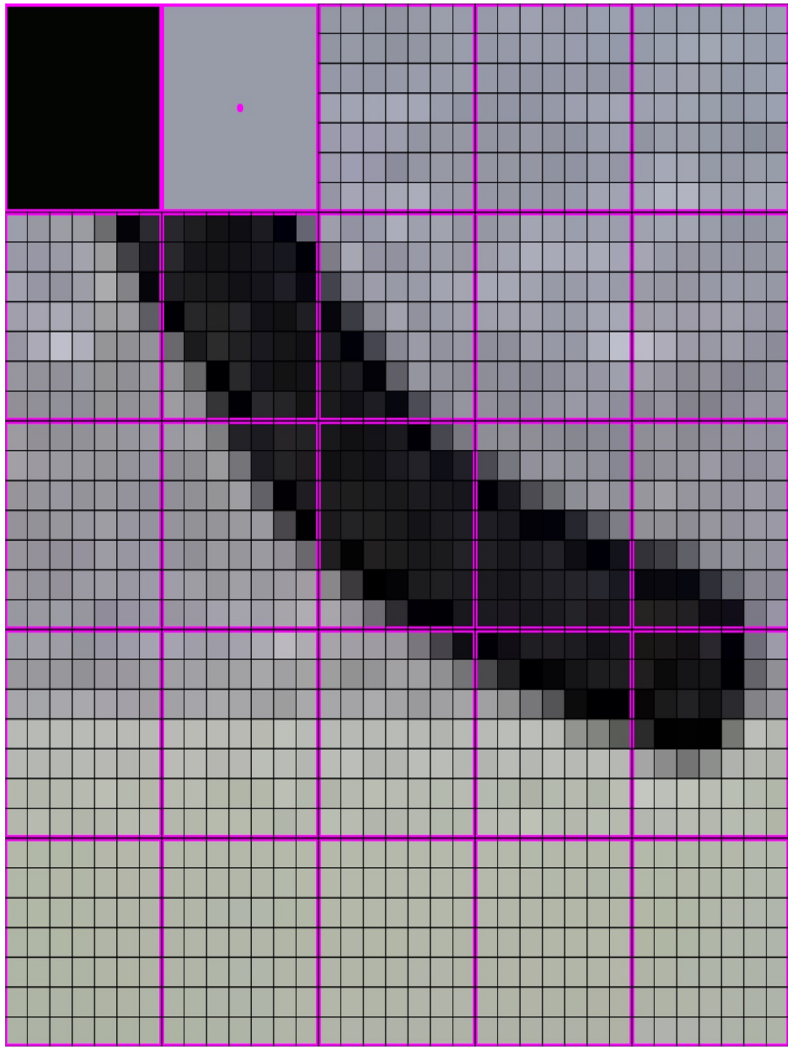
From 448×448 to 64×64



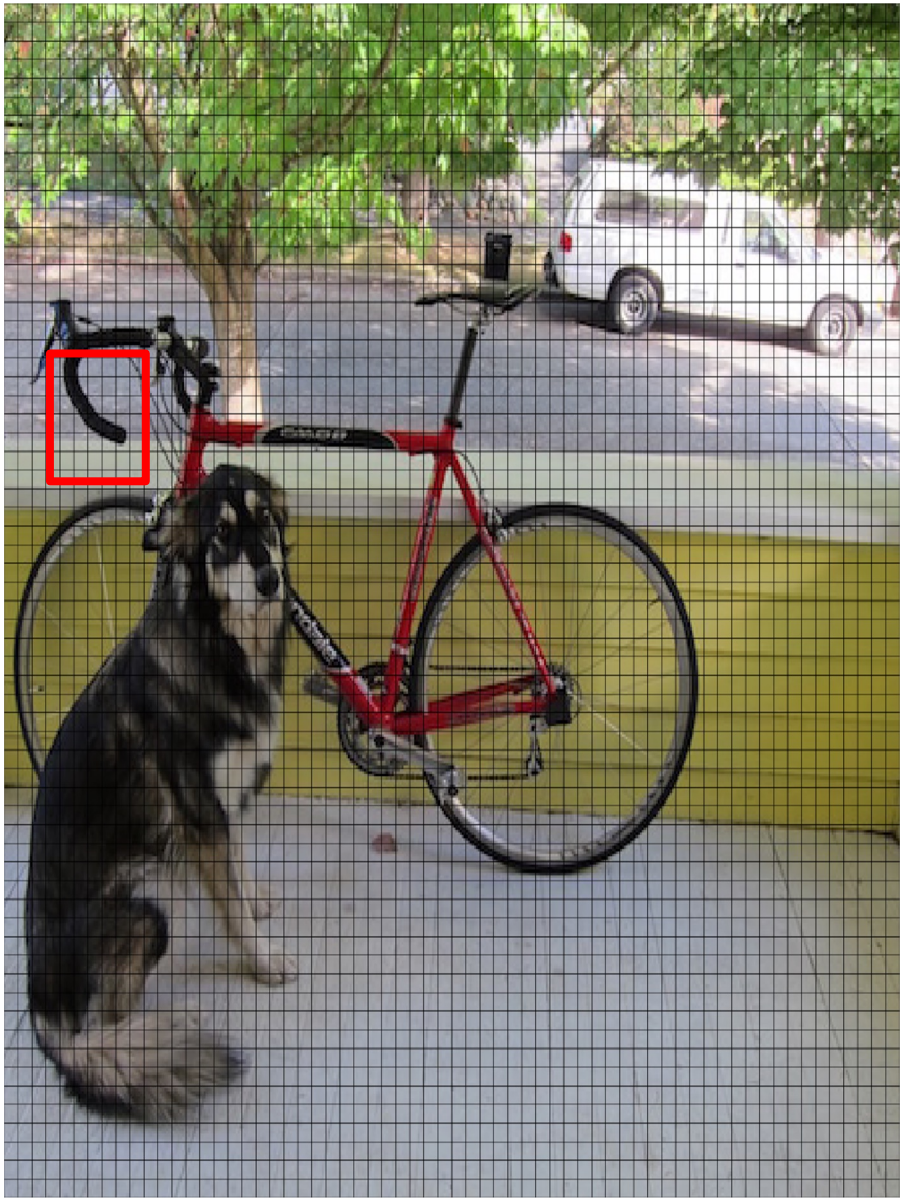


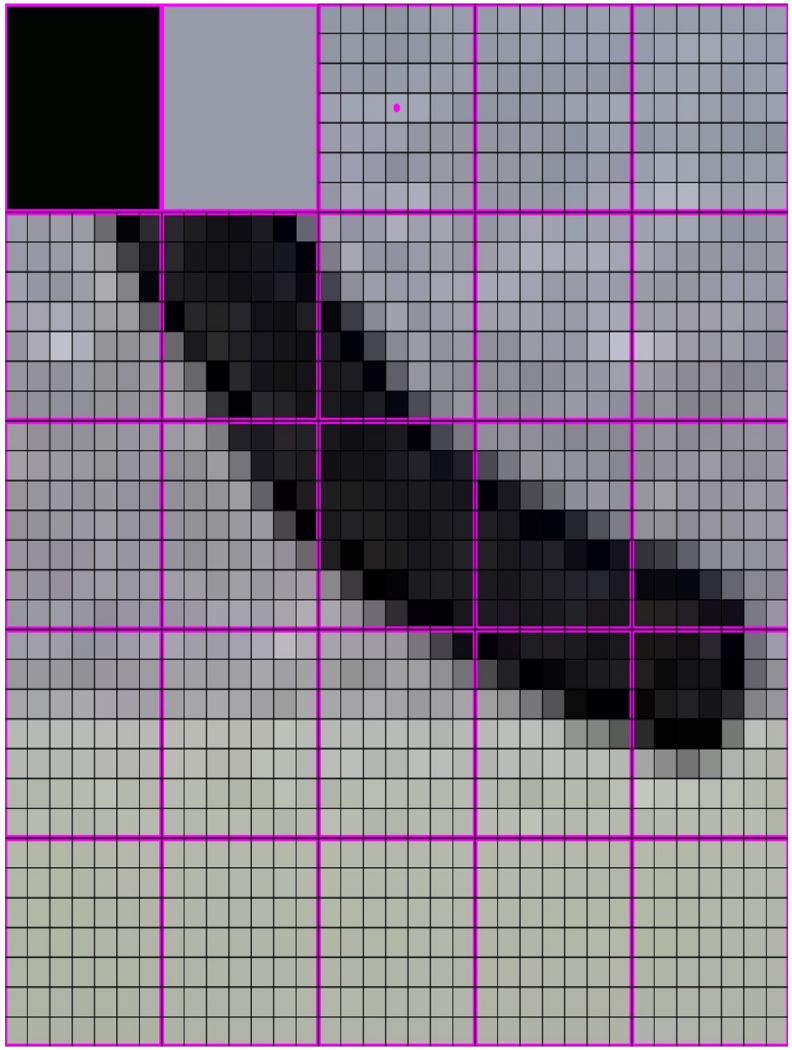
From 448×448 to 64×64



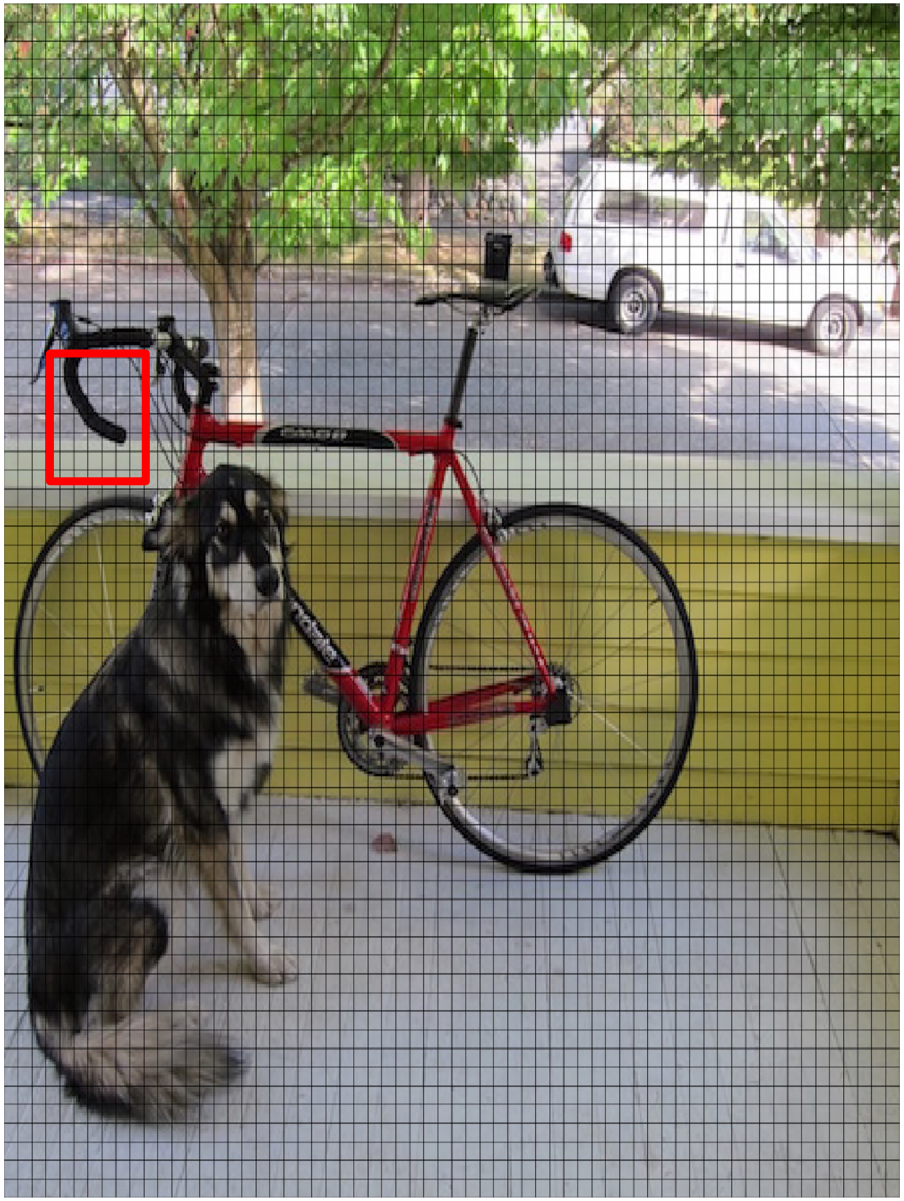


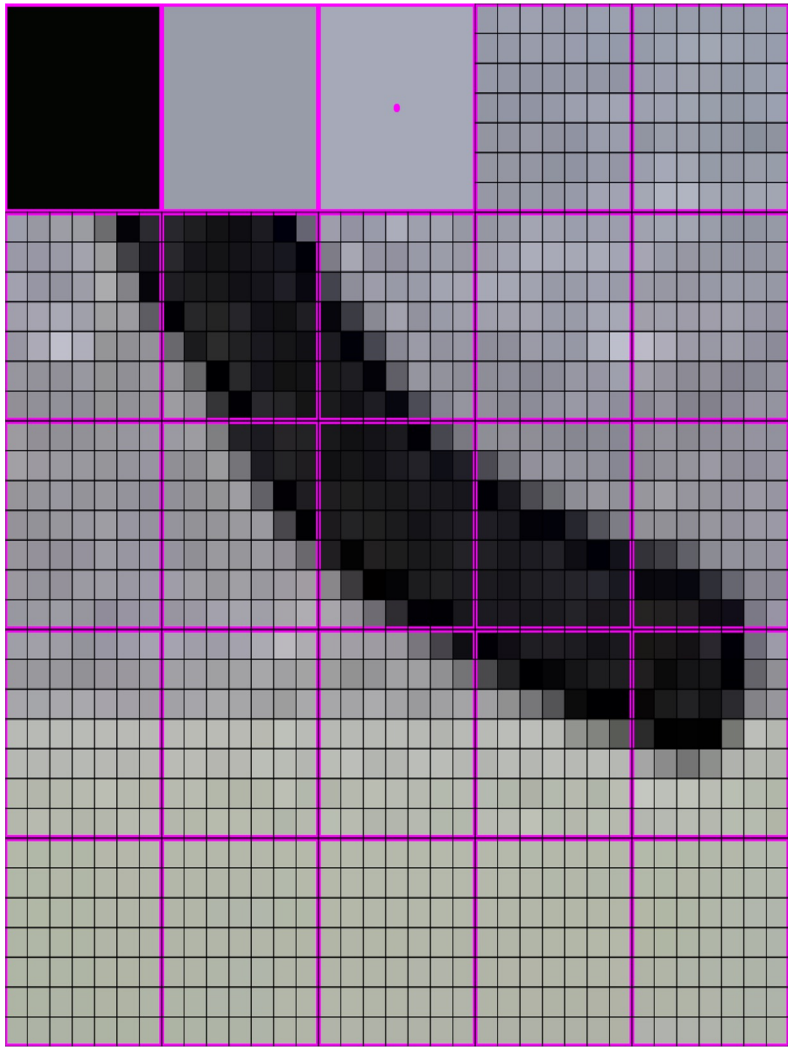
From 448×448 to 64×64



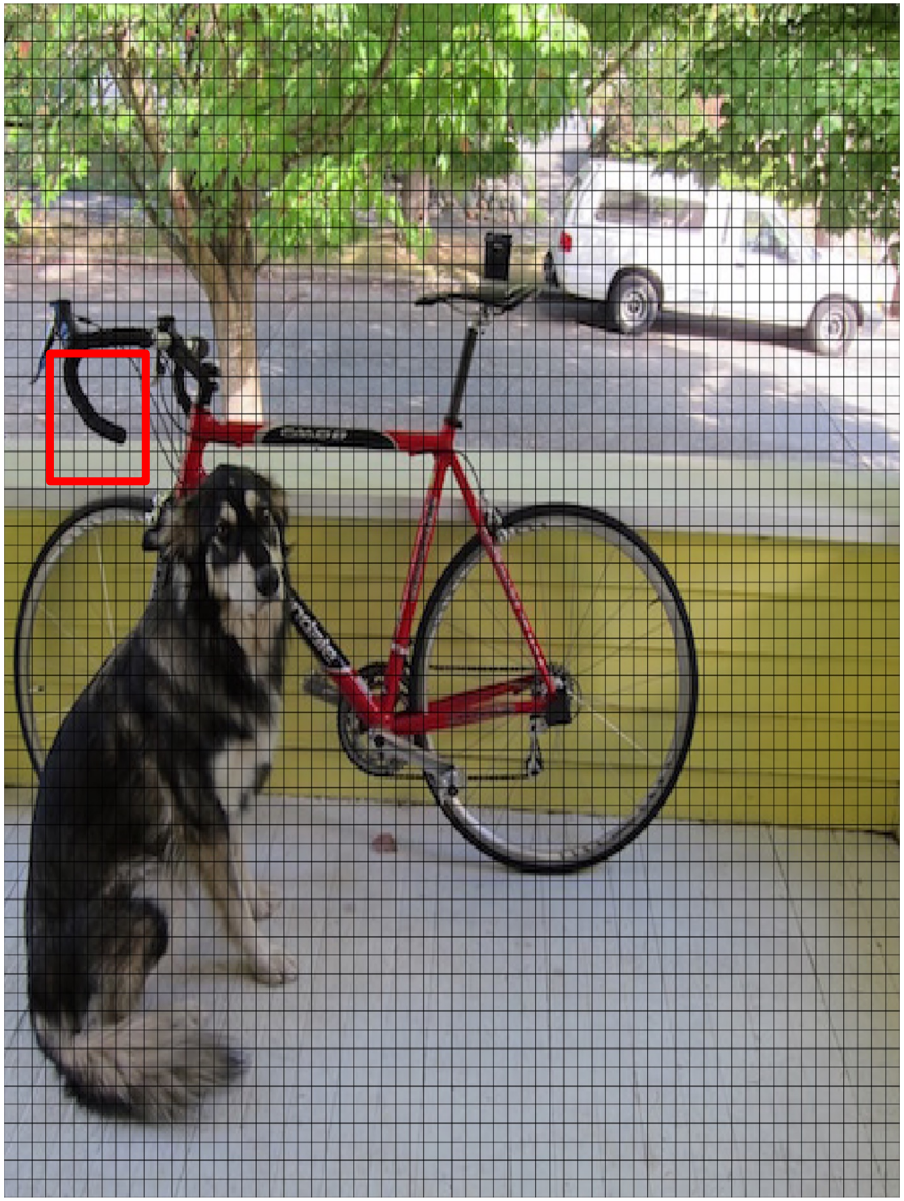


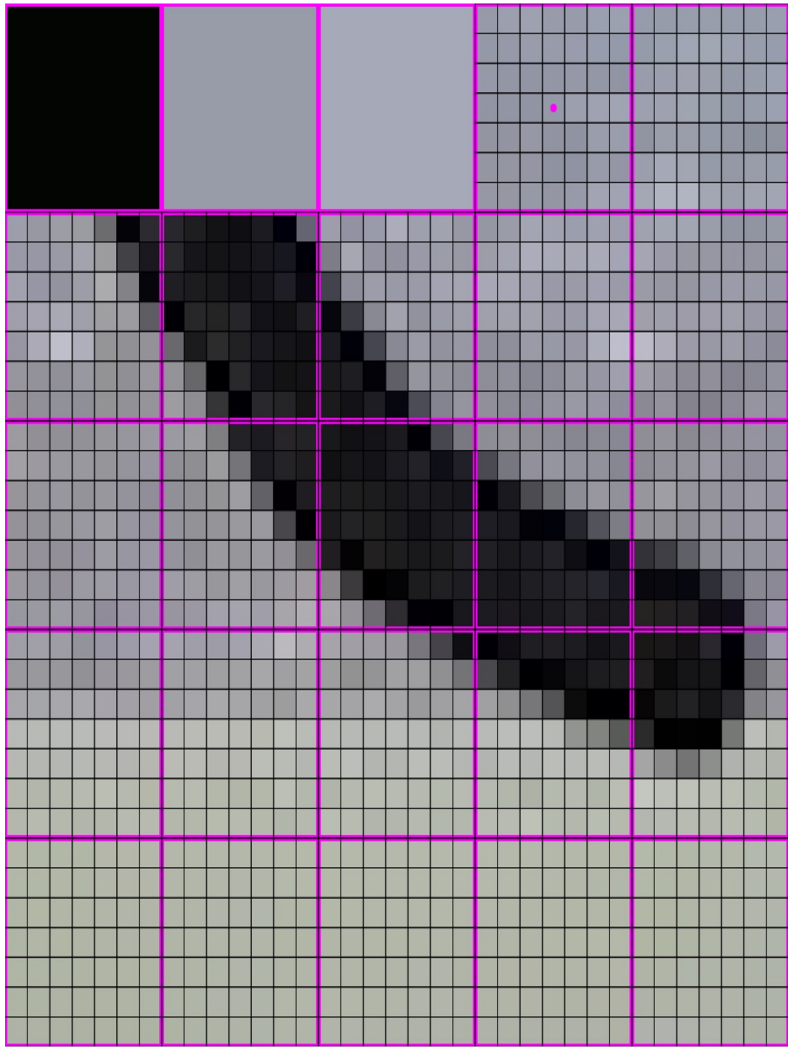
From 448×448 to 64×64



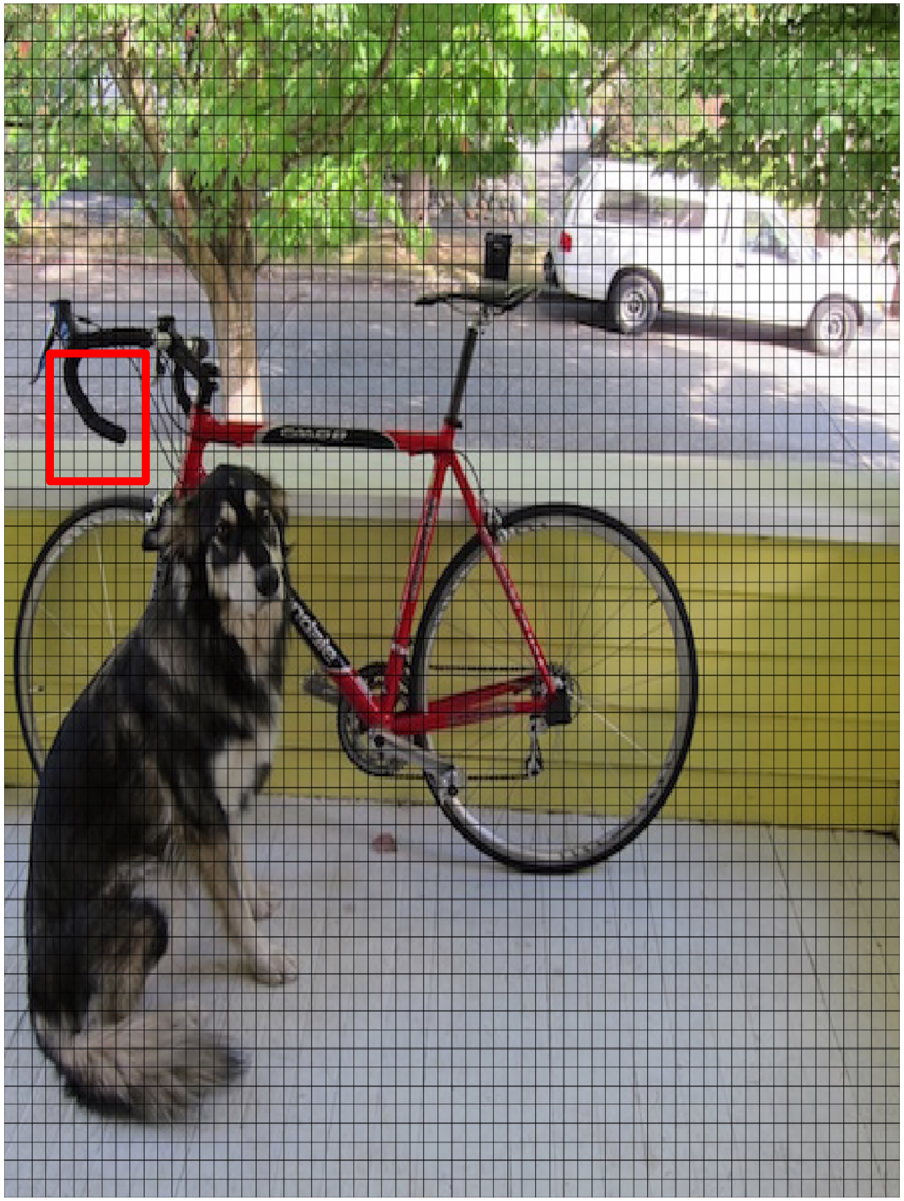


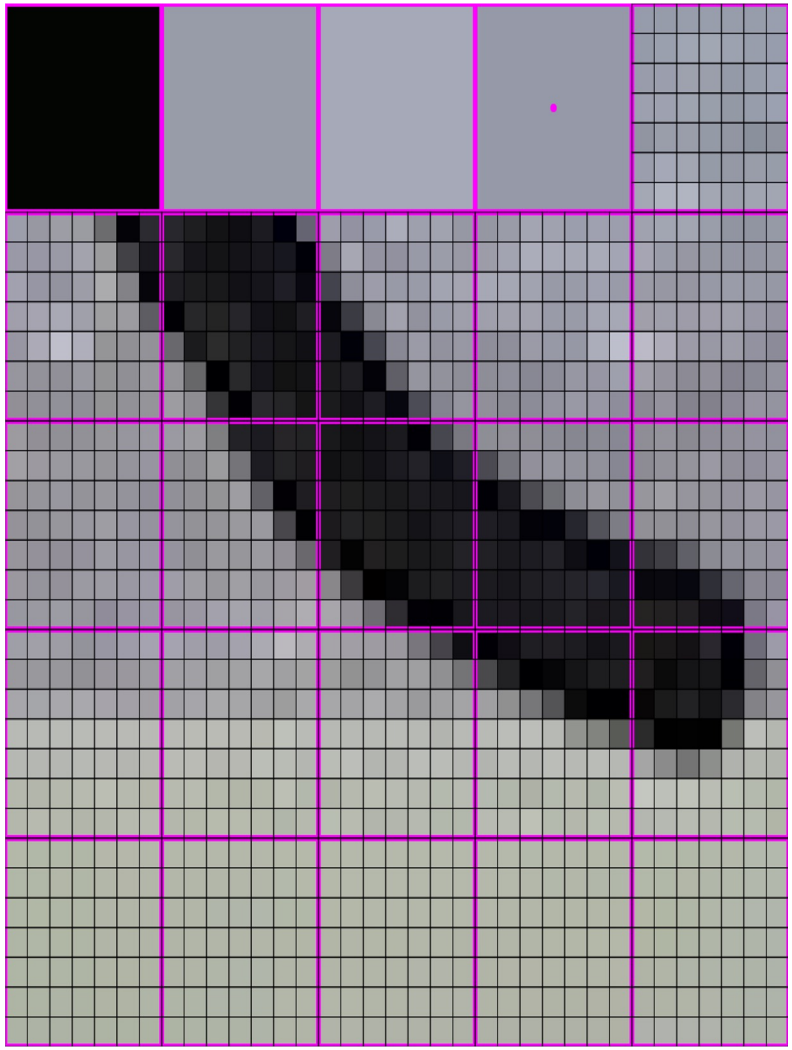
From 448×448 to 64×64



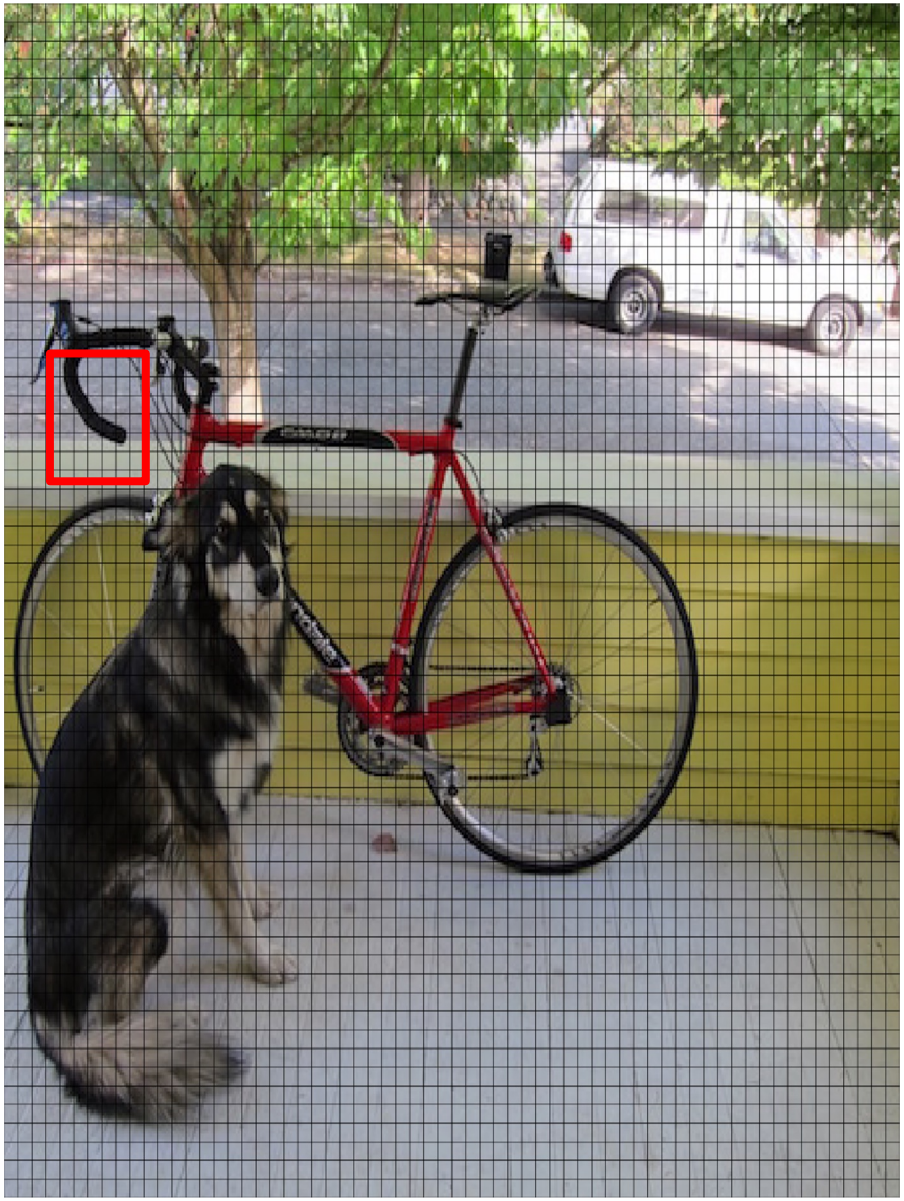


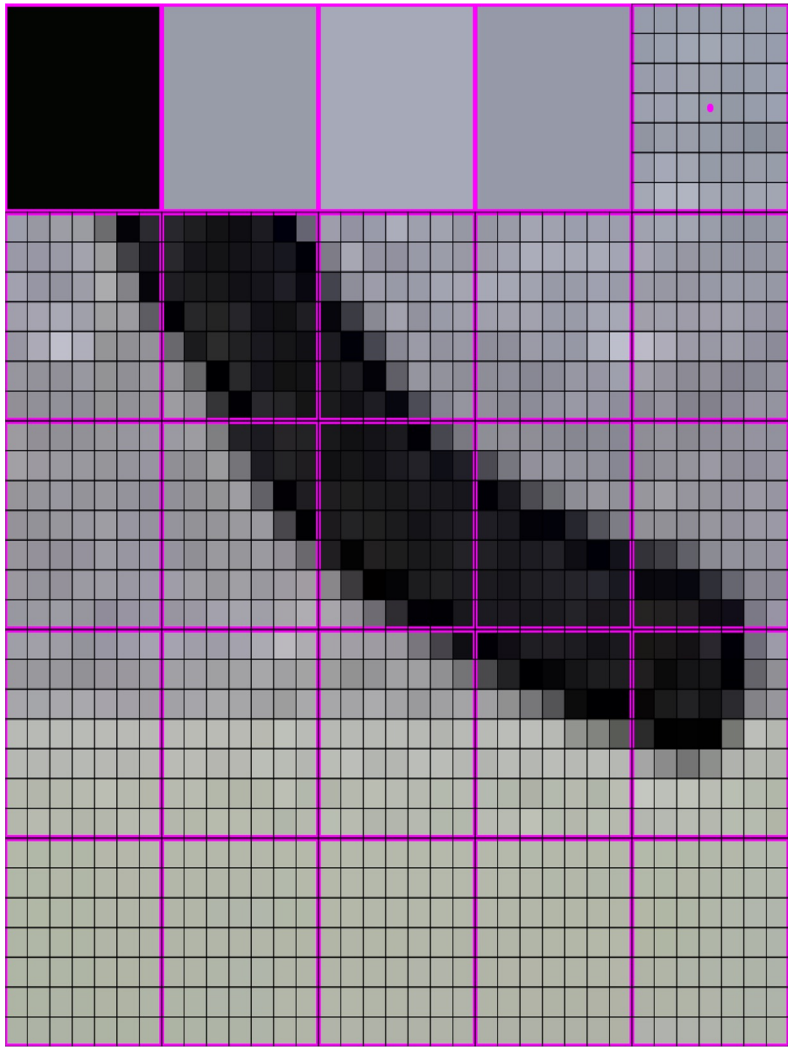
From 448×448 to 64×64



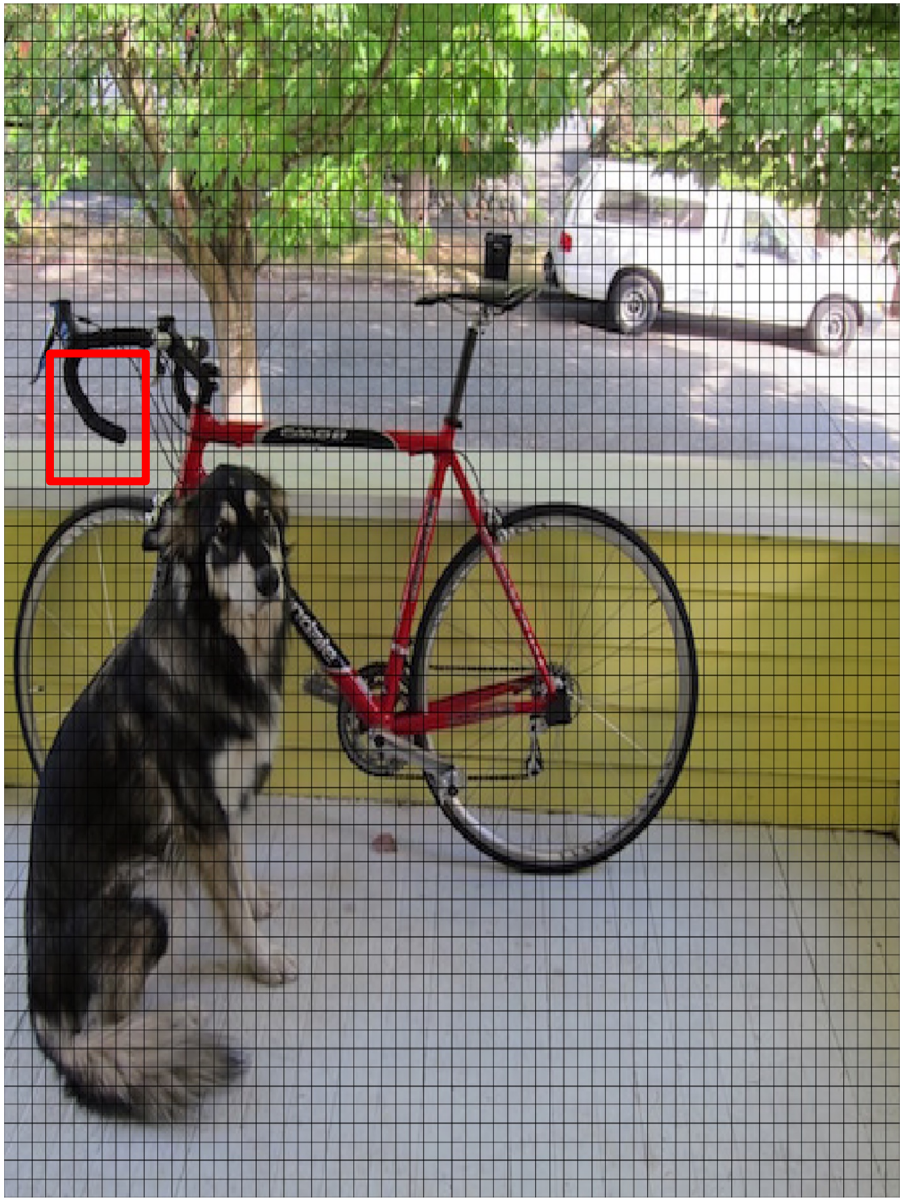


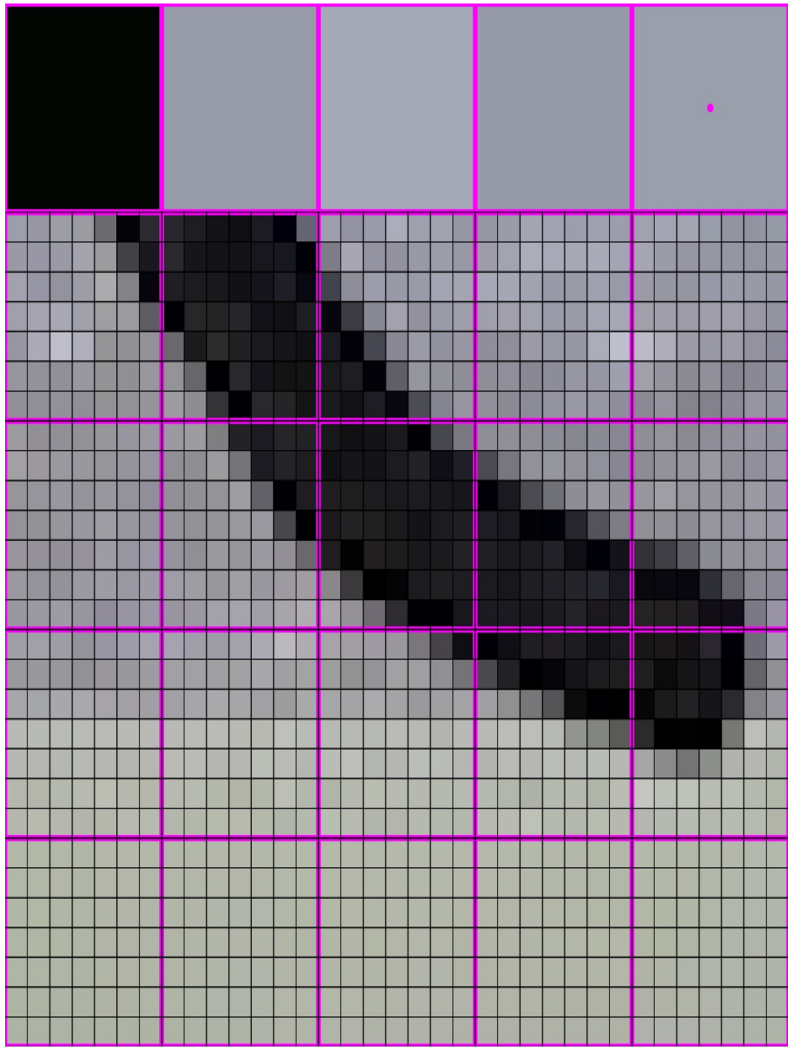
From 448×448 to 64×64



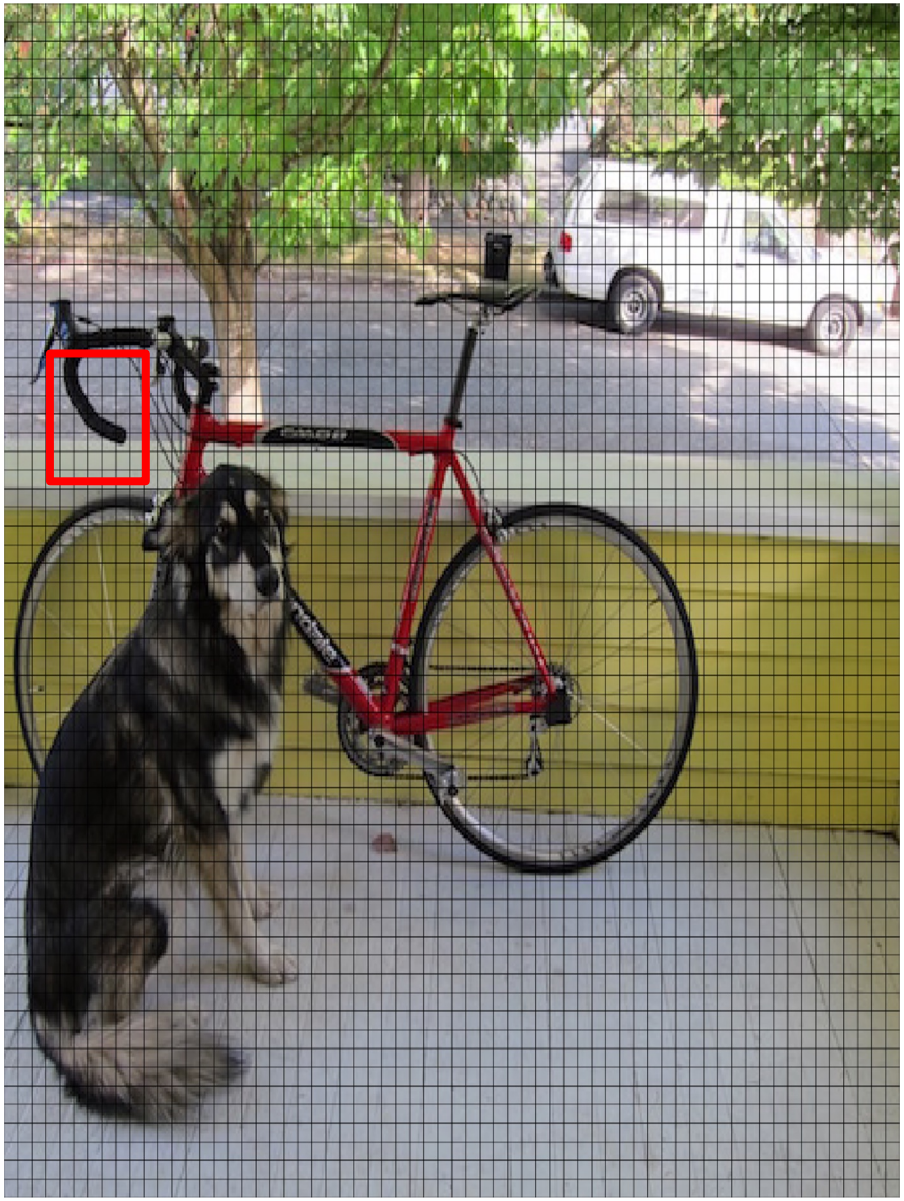


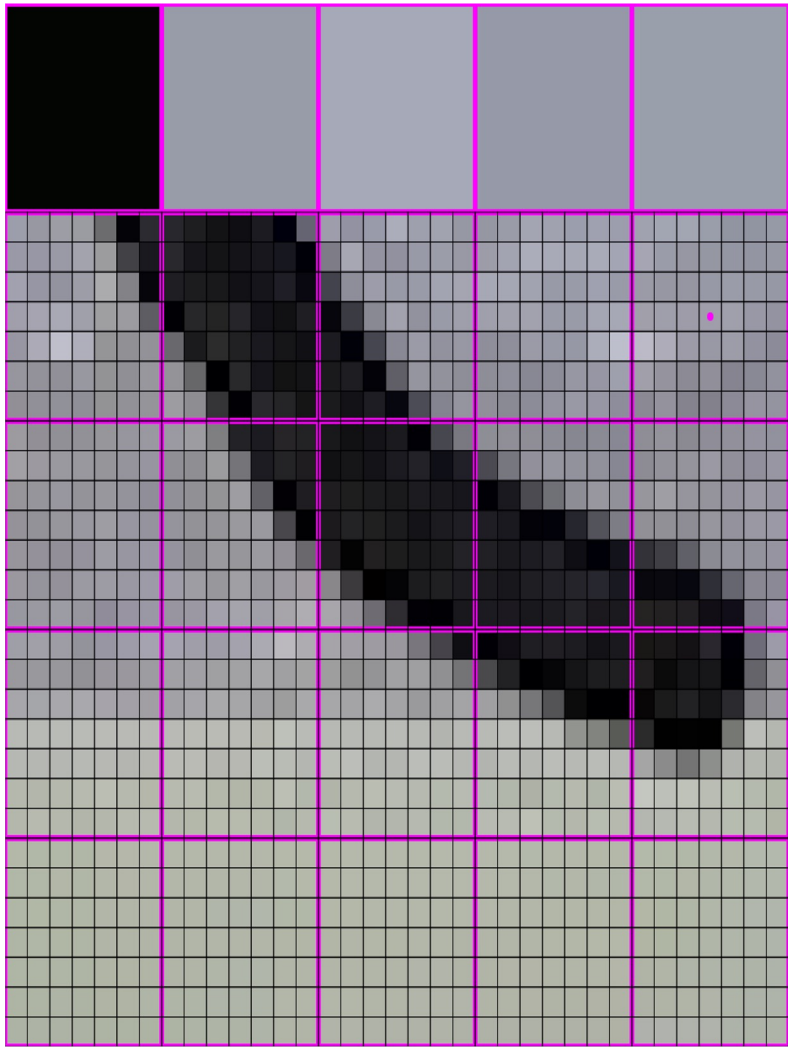
From 448×448 to 64×64



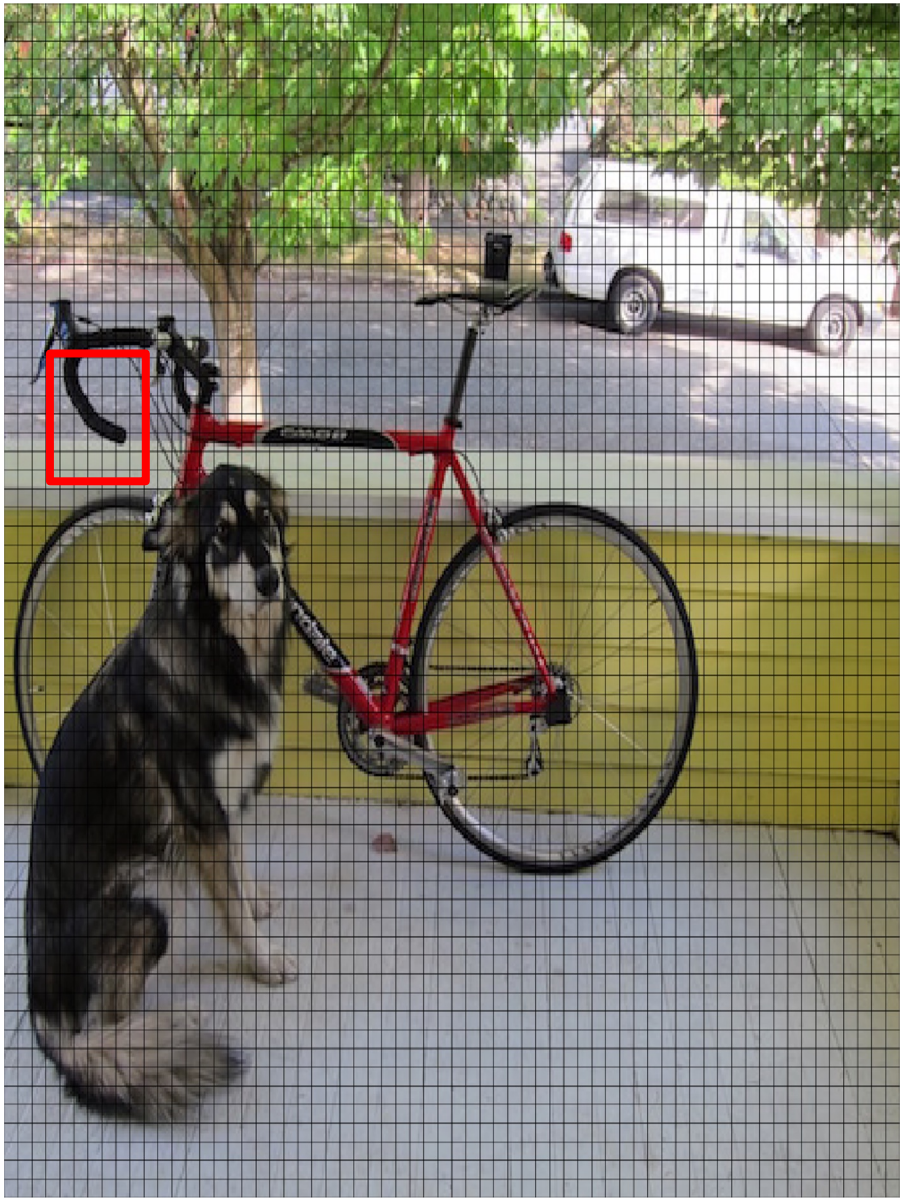


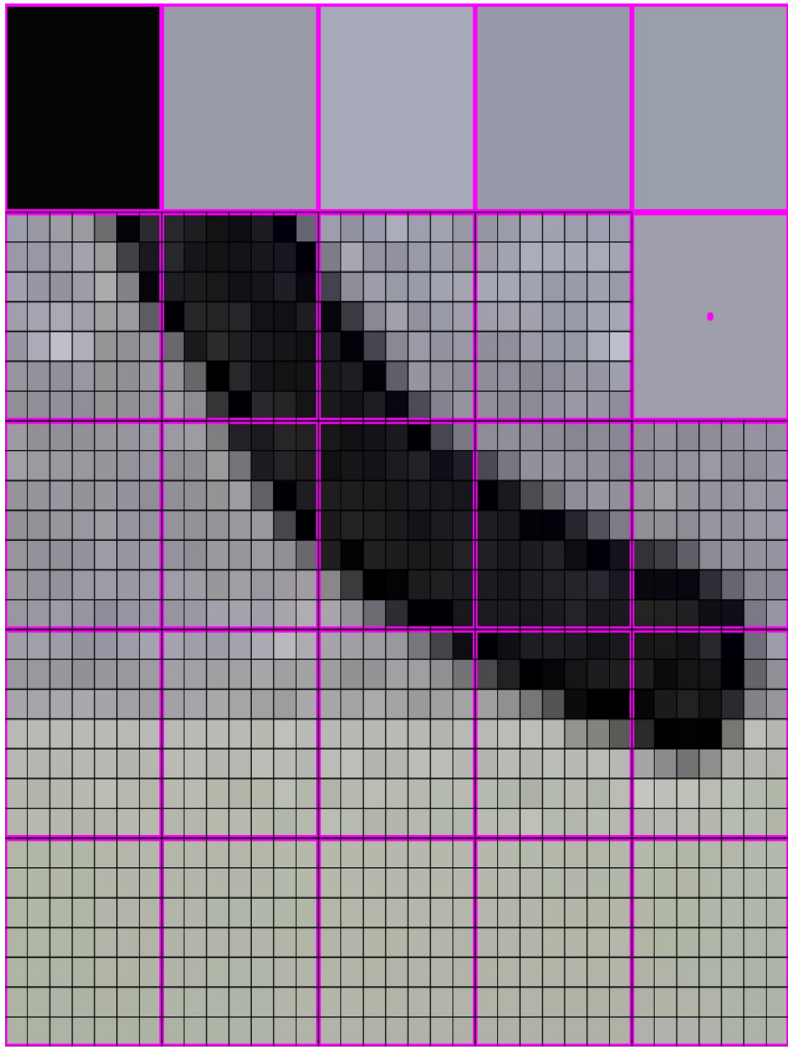
From 448×448 to 64×64



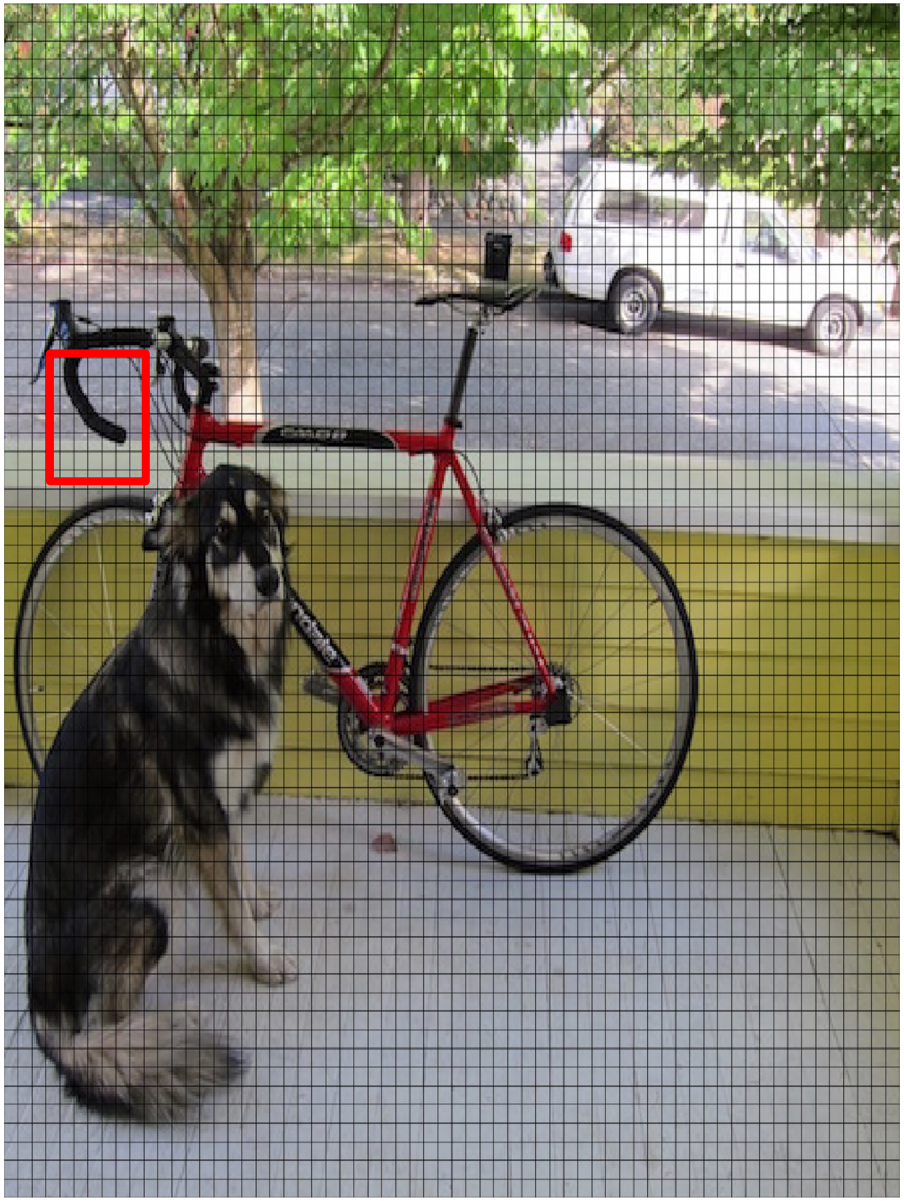


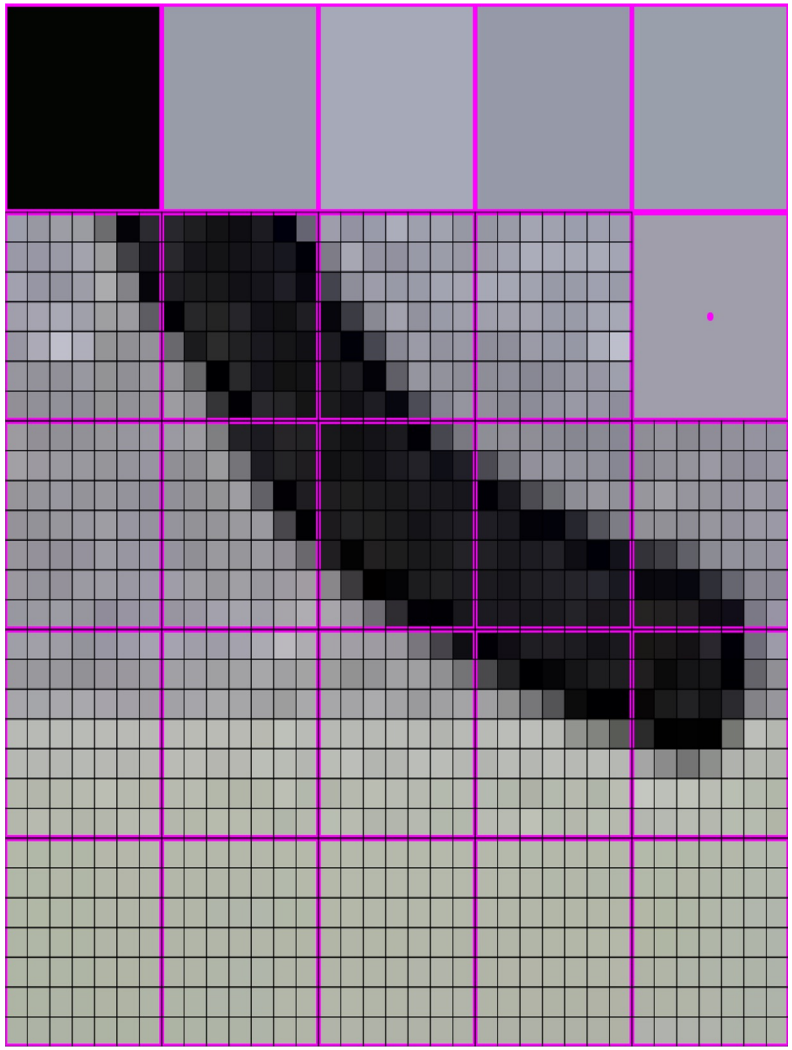
From 448×448 to 64×64



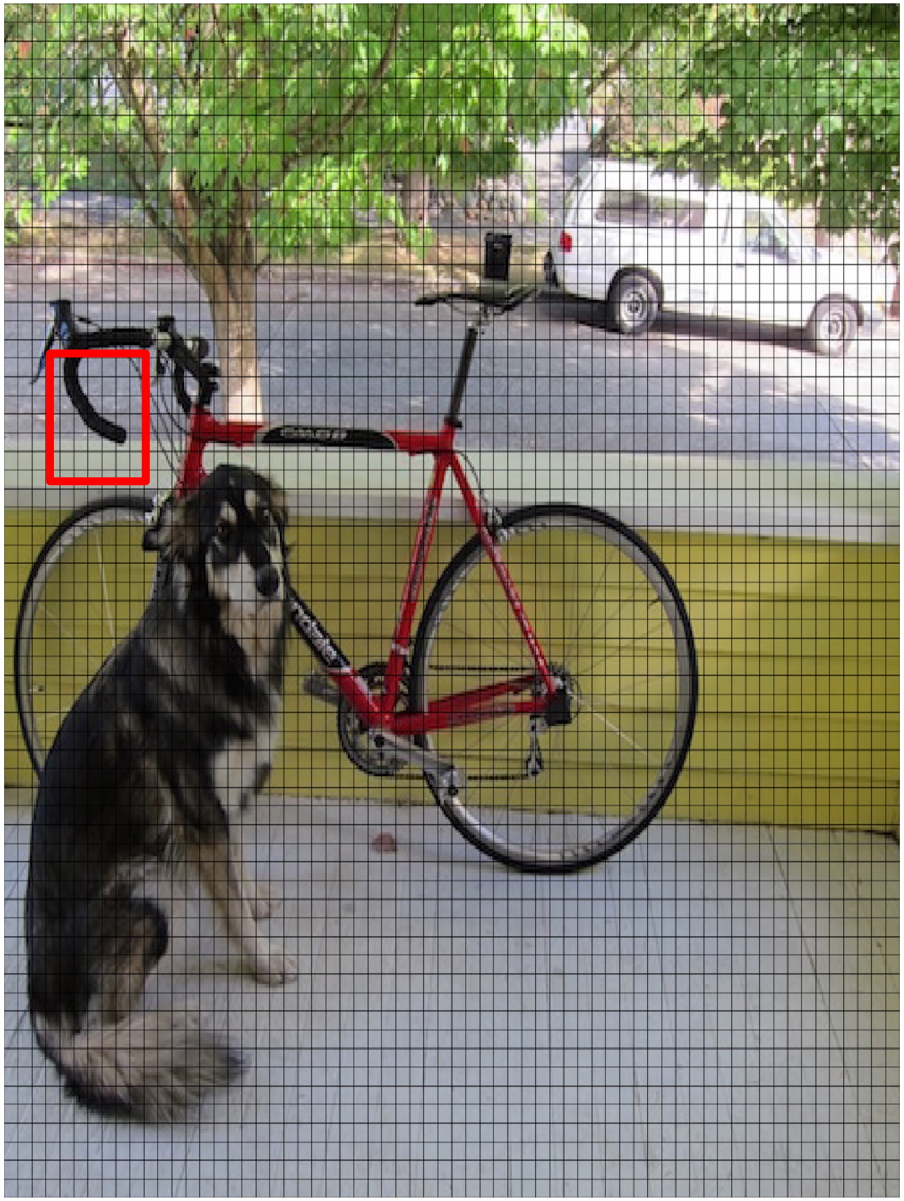


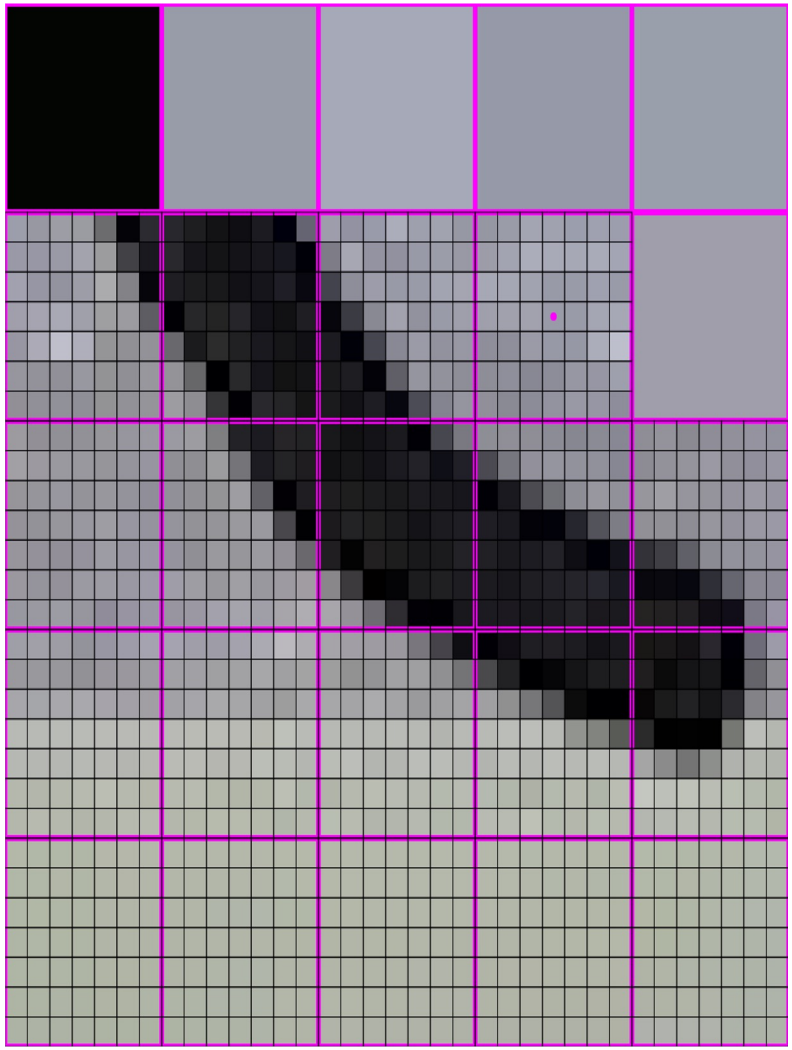
From 448×448 to 64×64



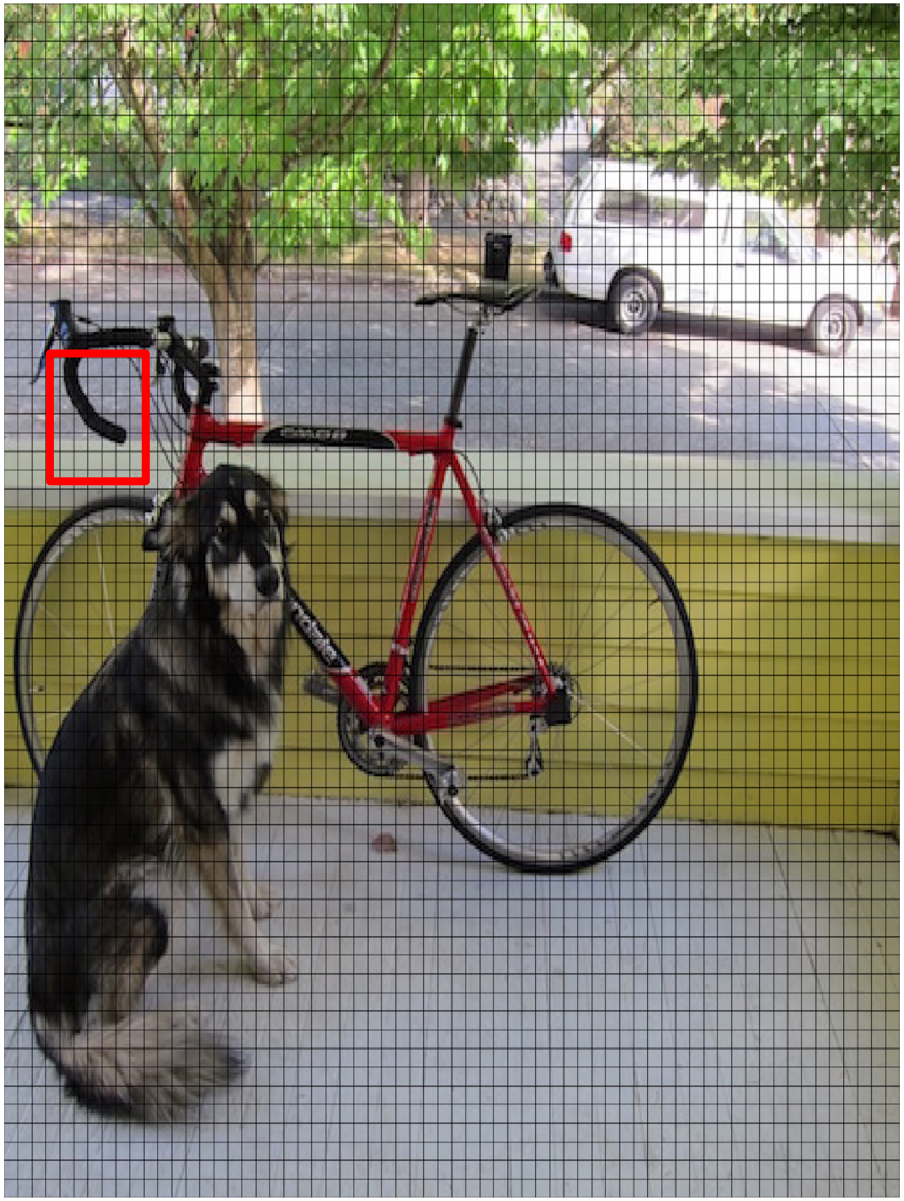


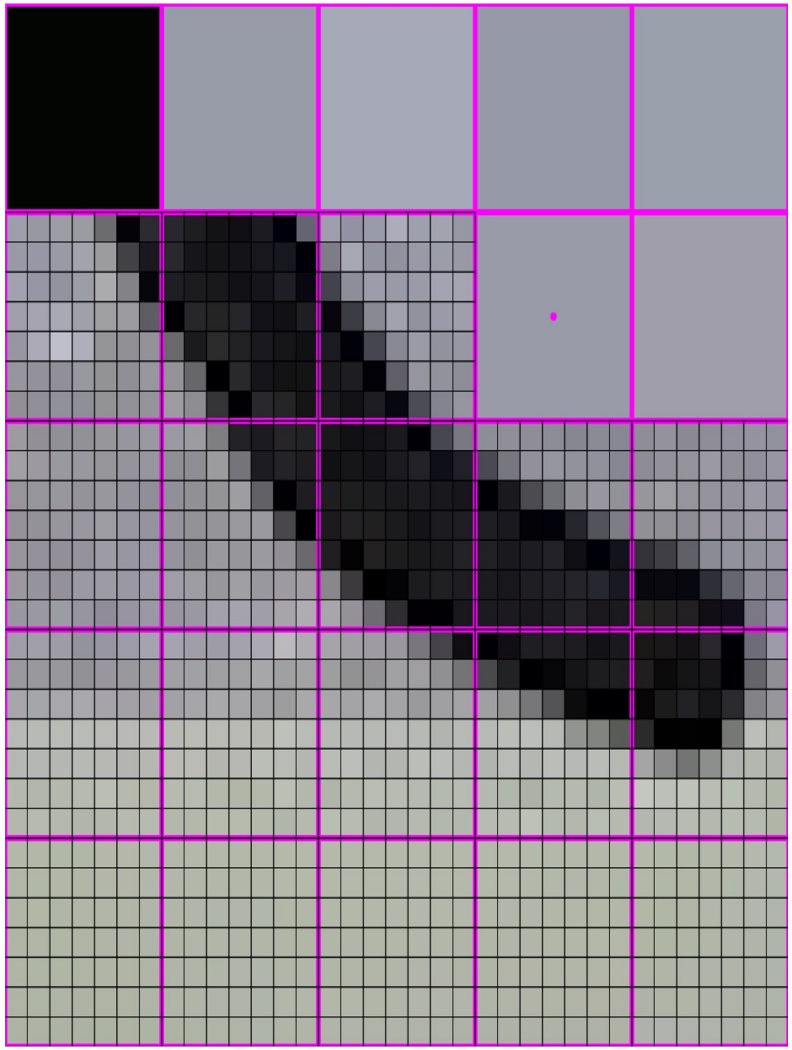
From 448×448 to 64×64



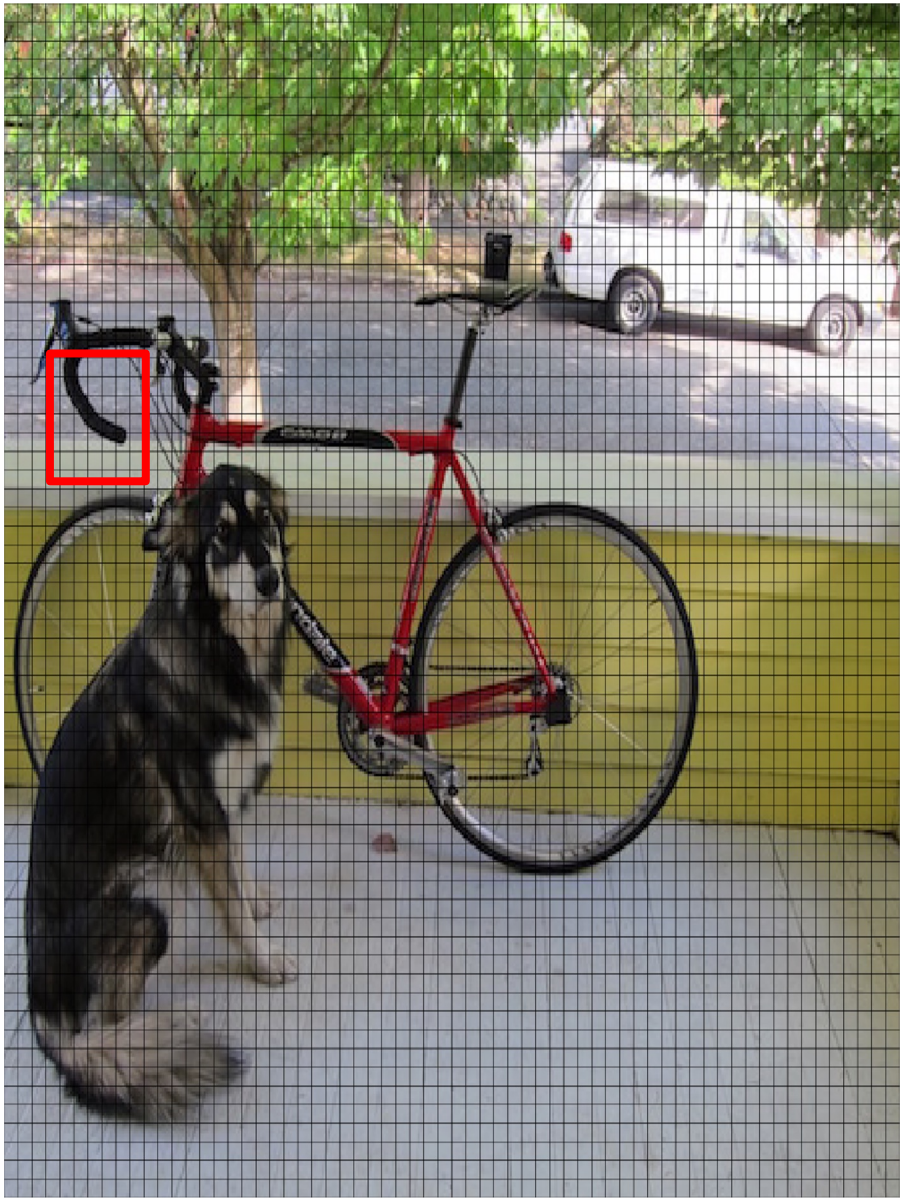


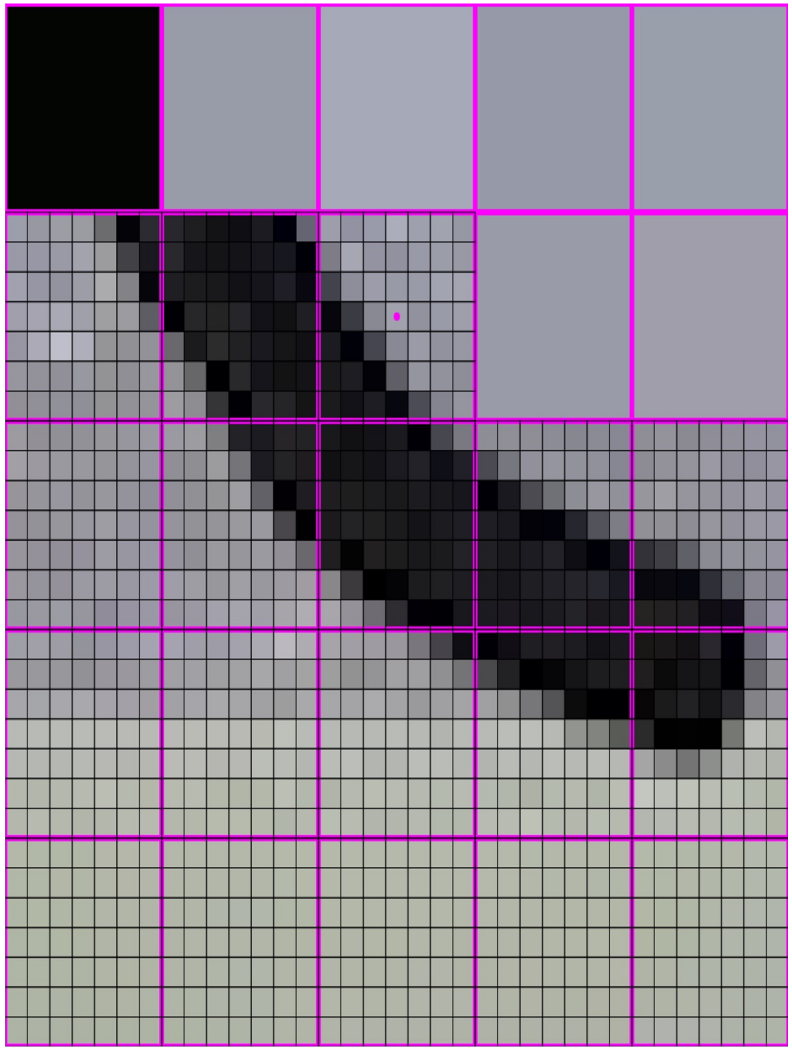
From 448×448 to 64×64



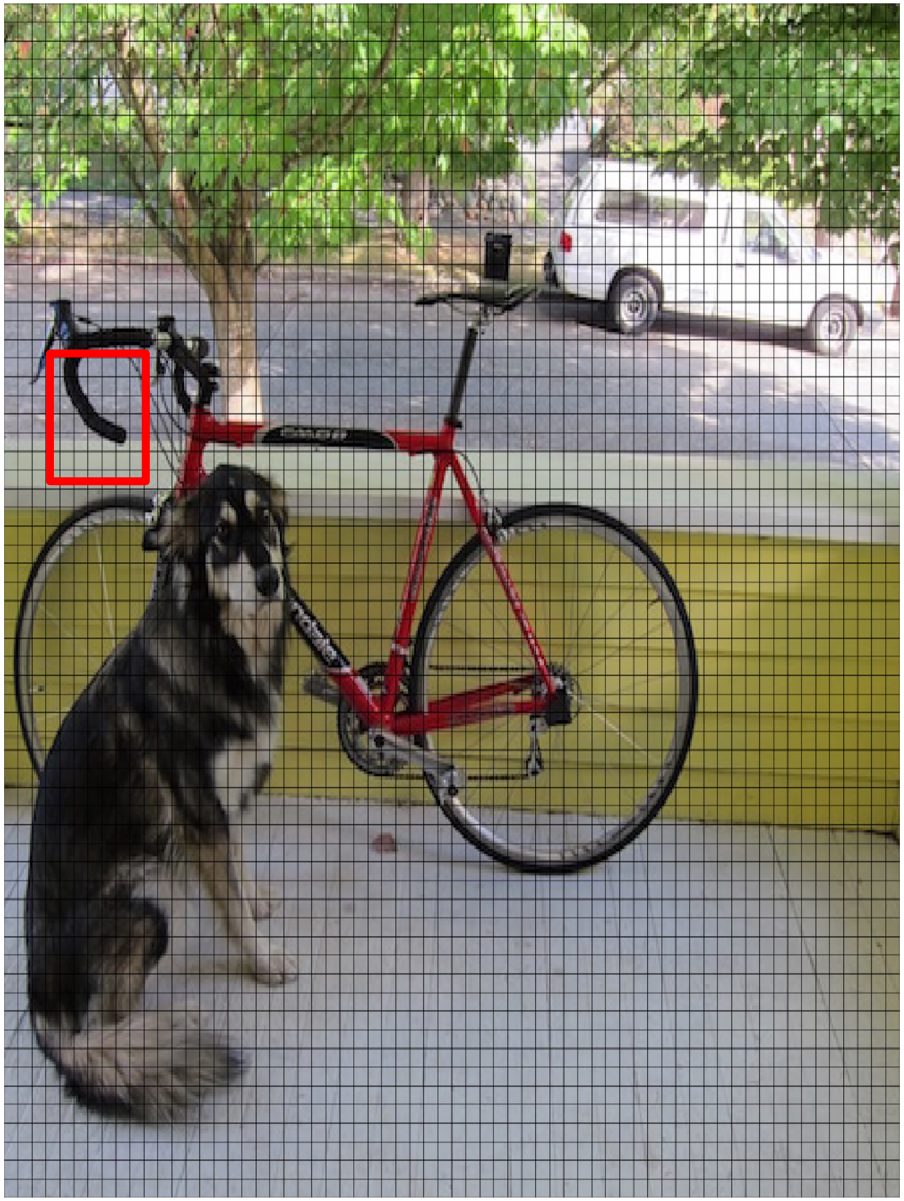


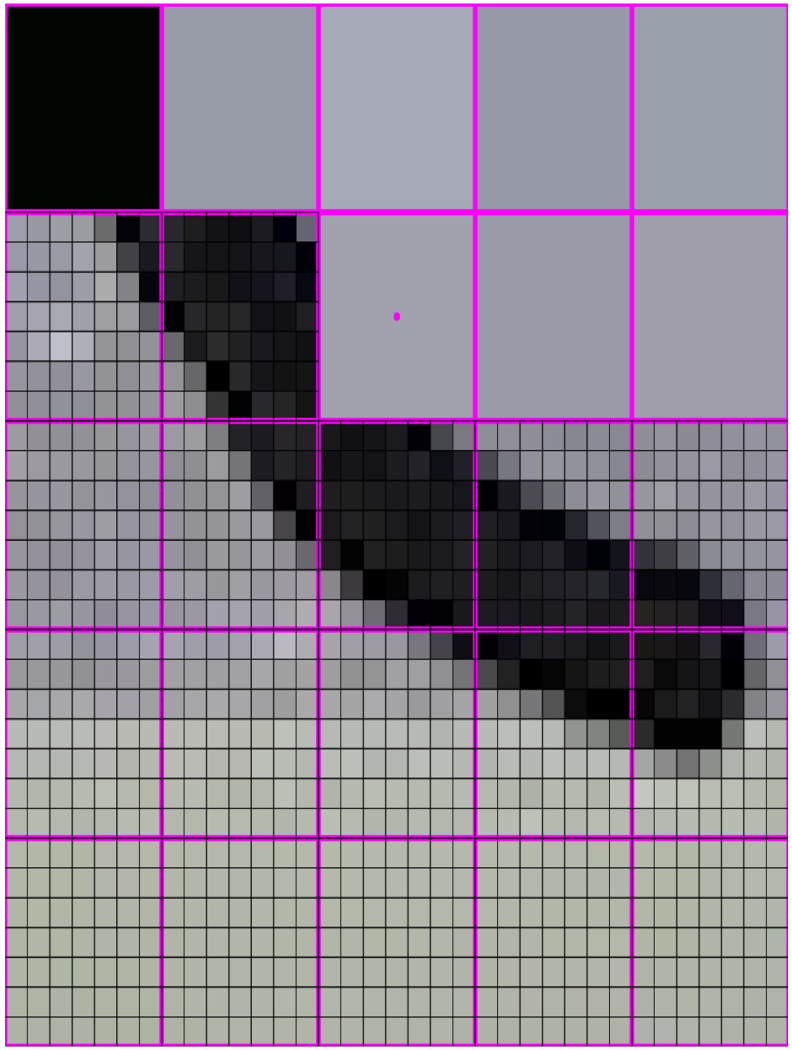
From 448×448 to 64×64



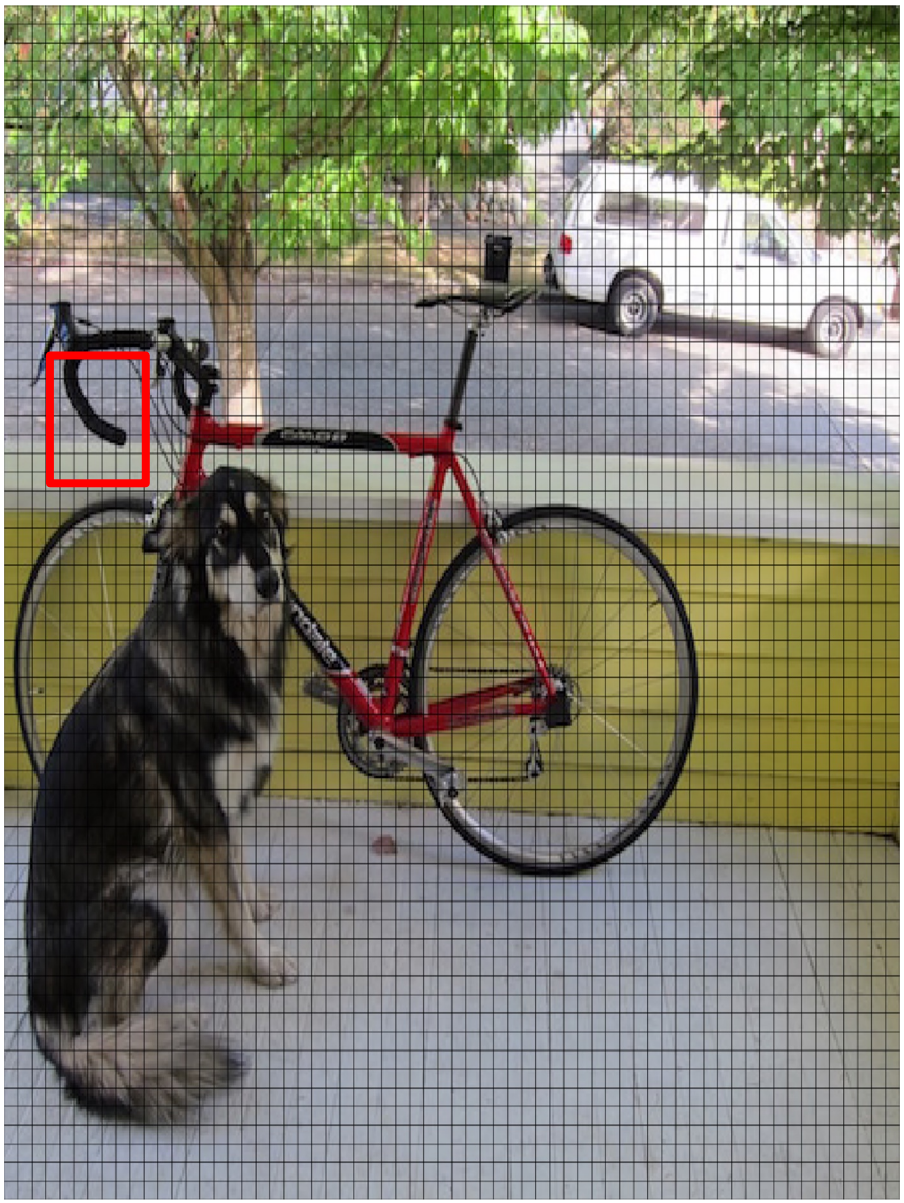


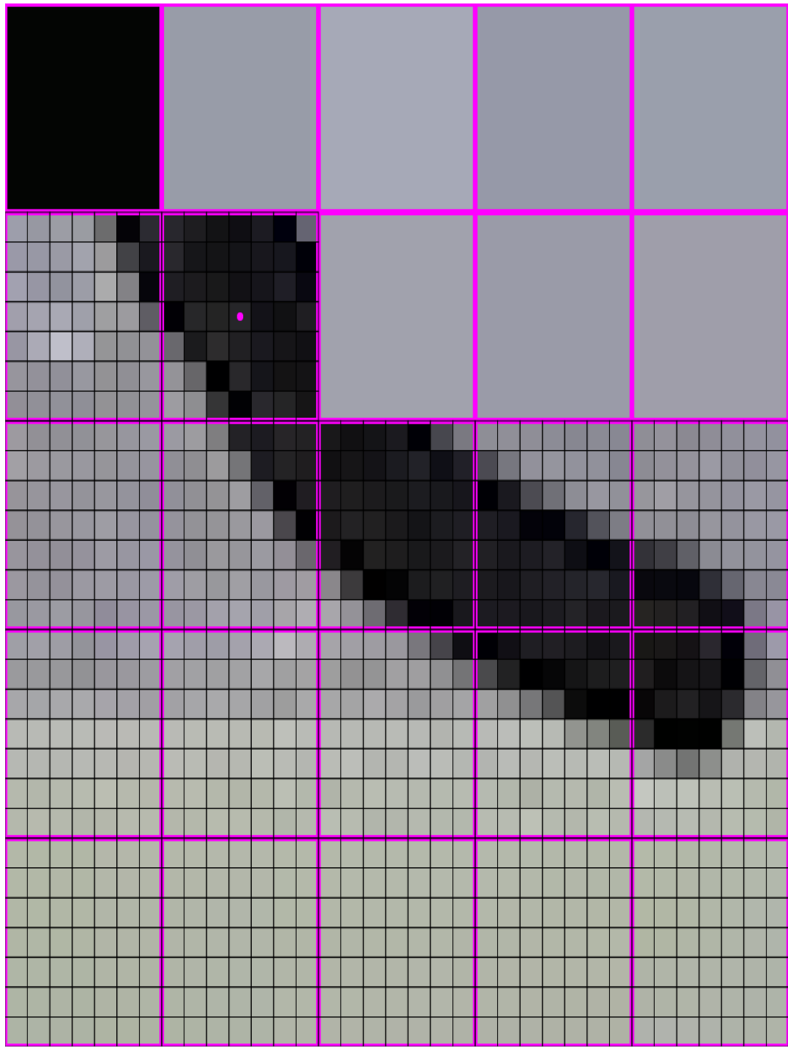
From 448×448 to 64×64



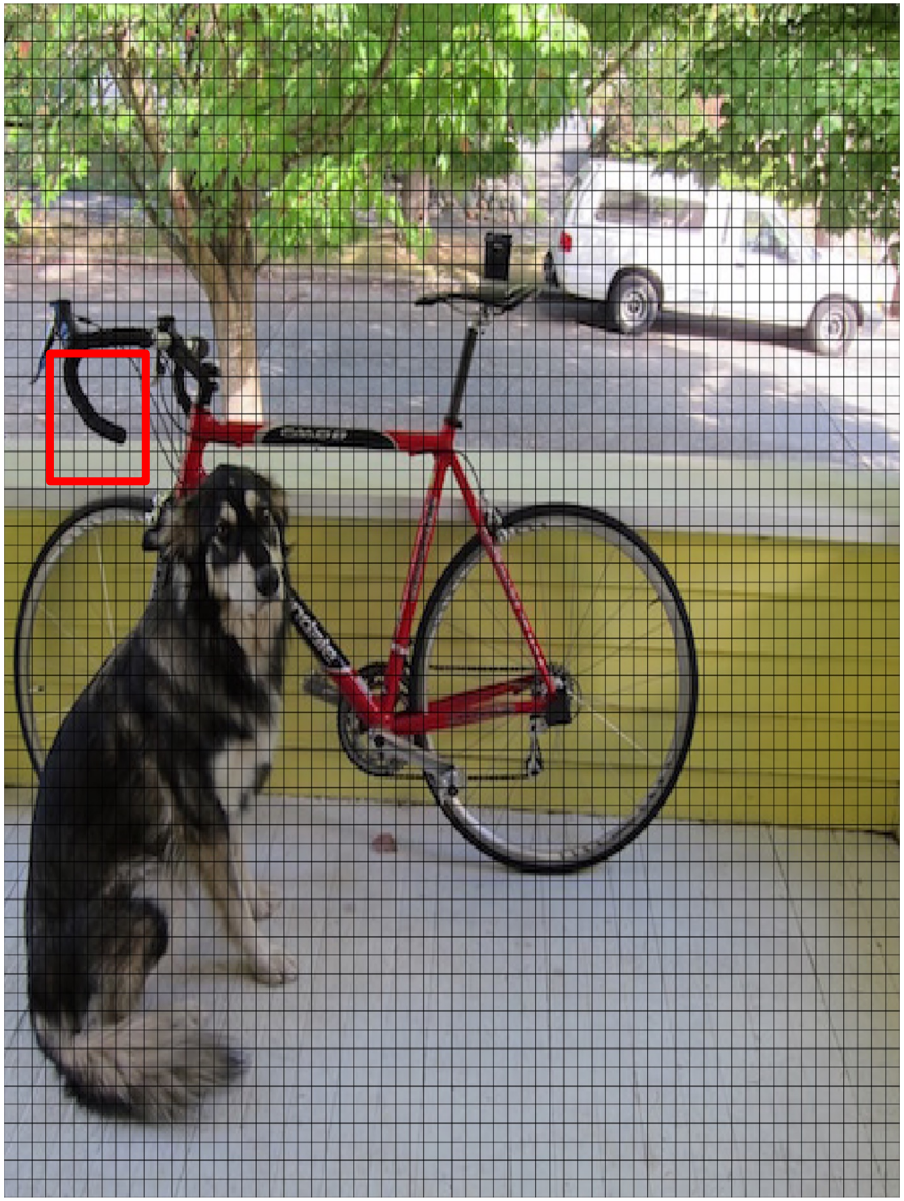


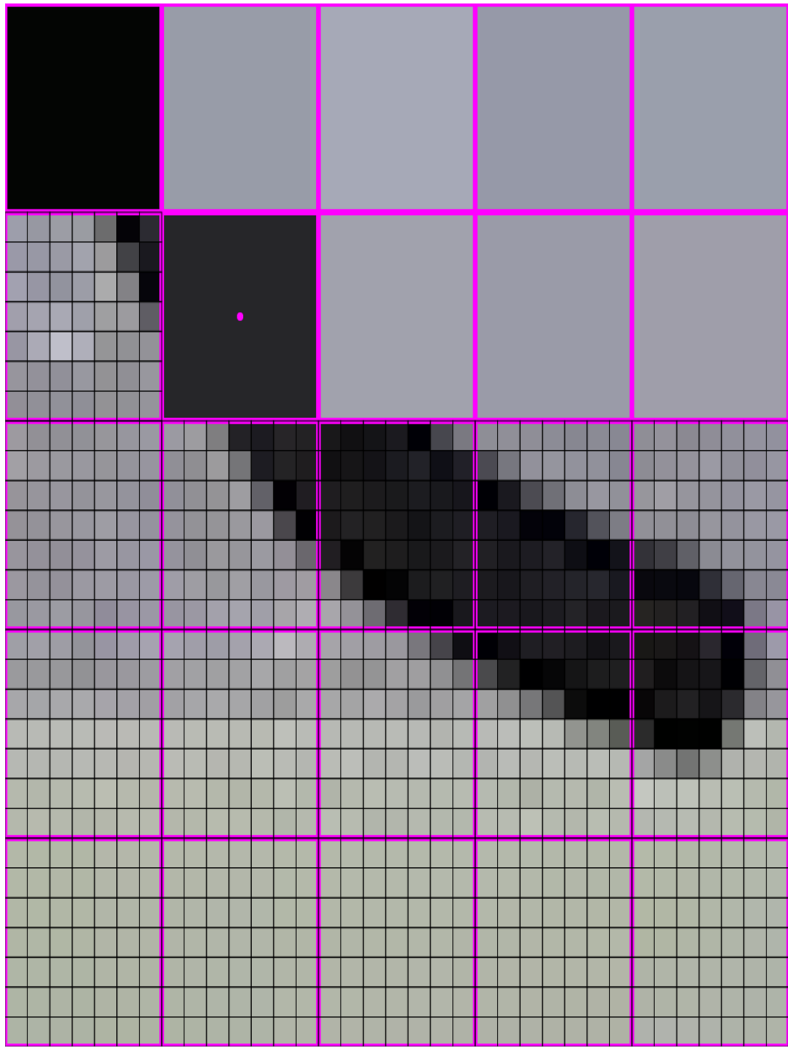
From 448×448 to 64×64



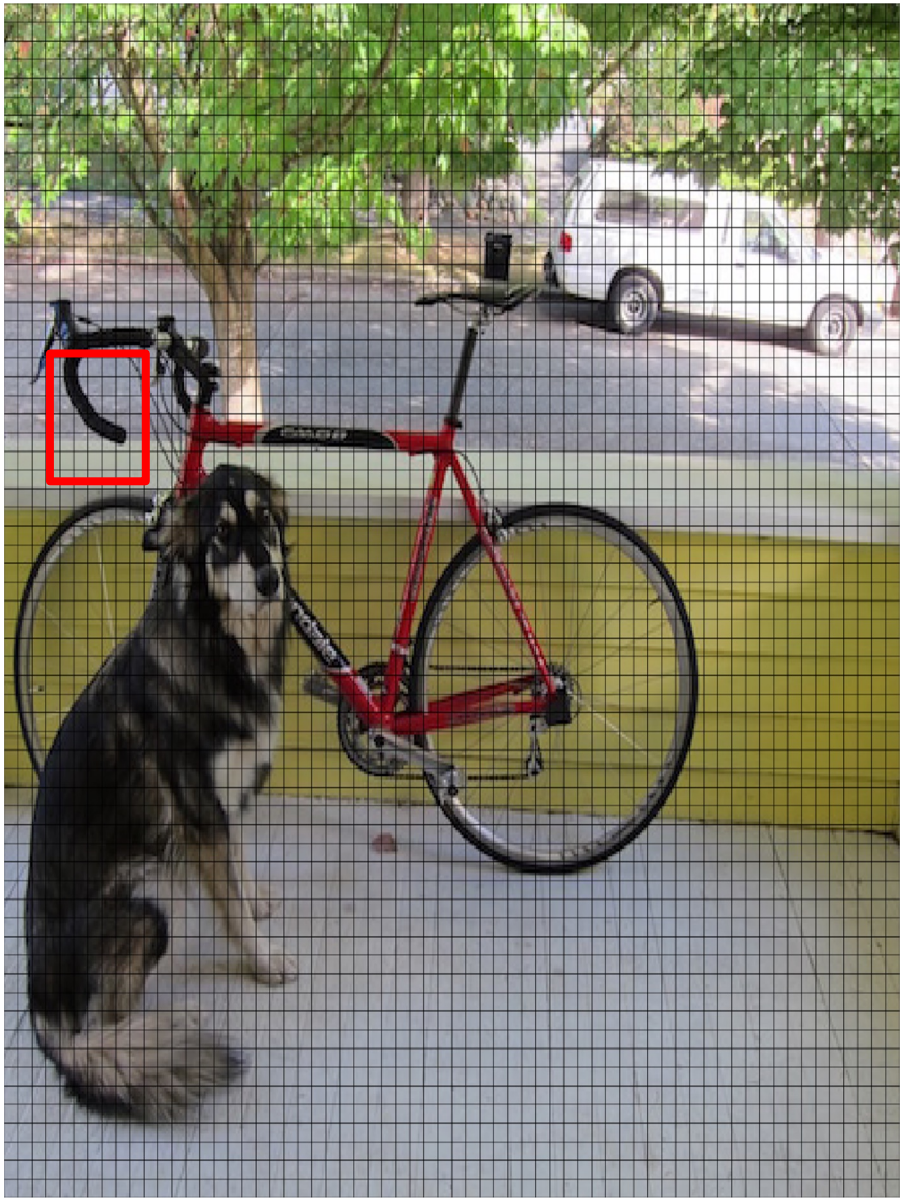


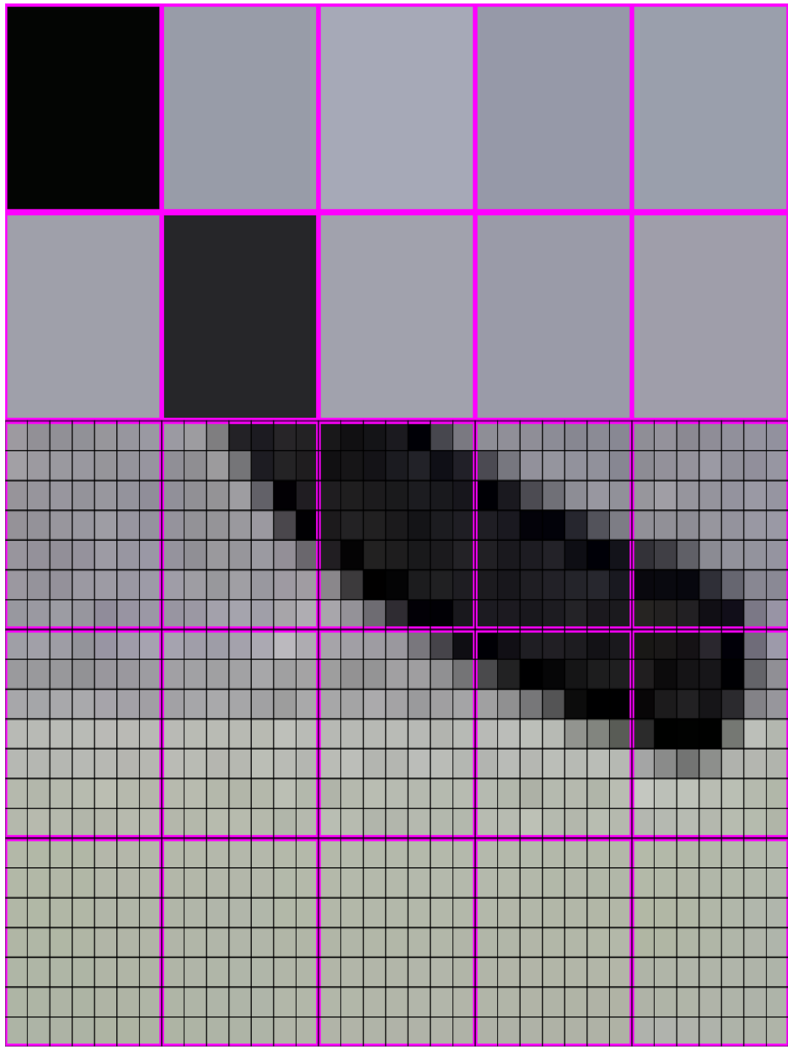
From 448×448 to 64×64



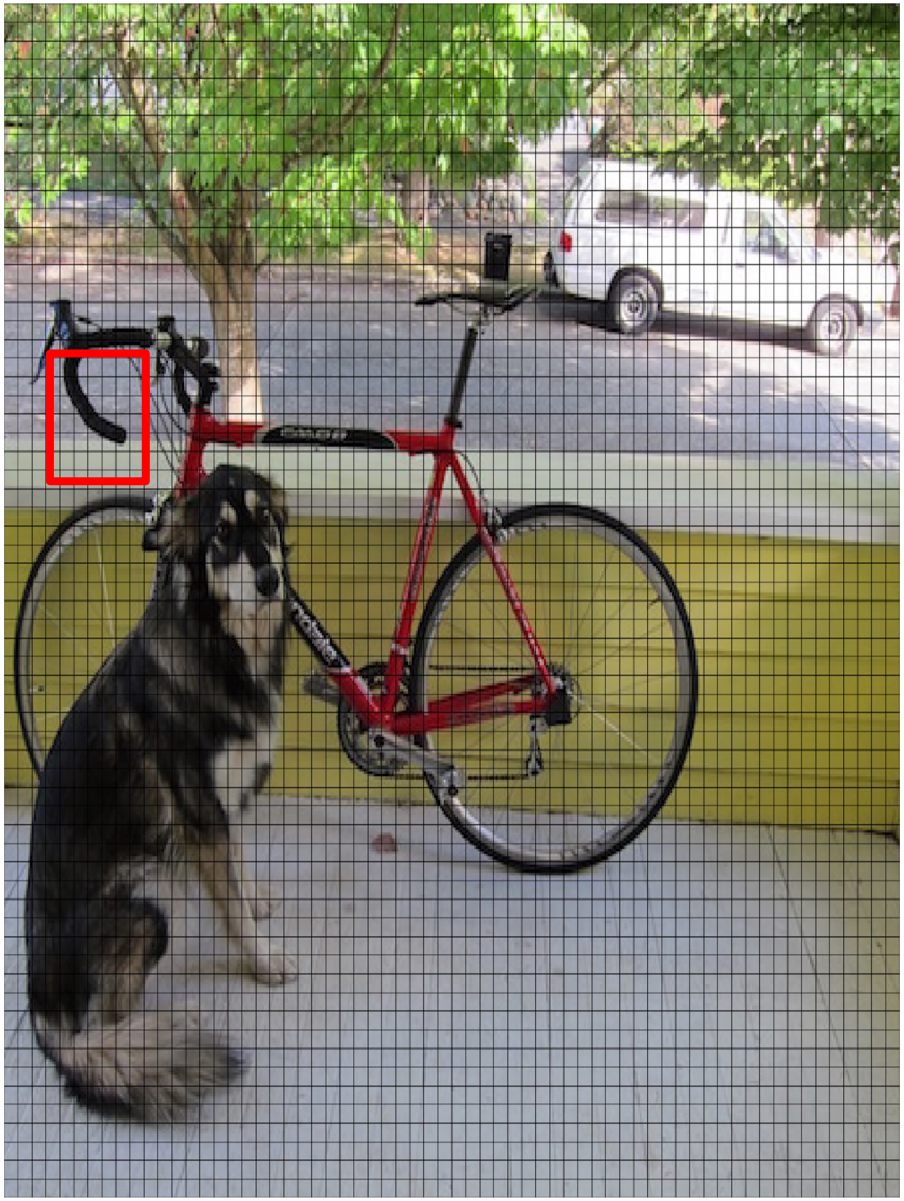


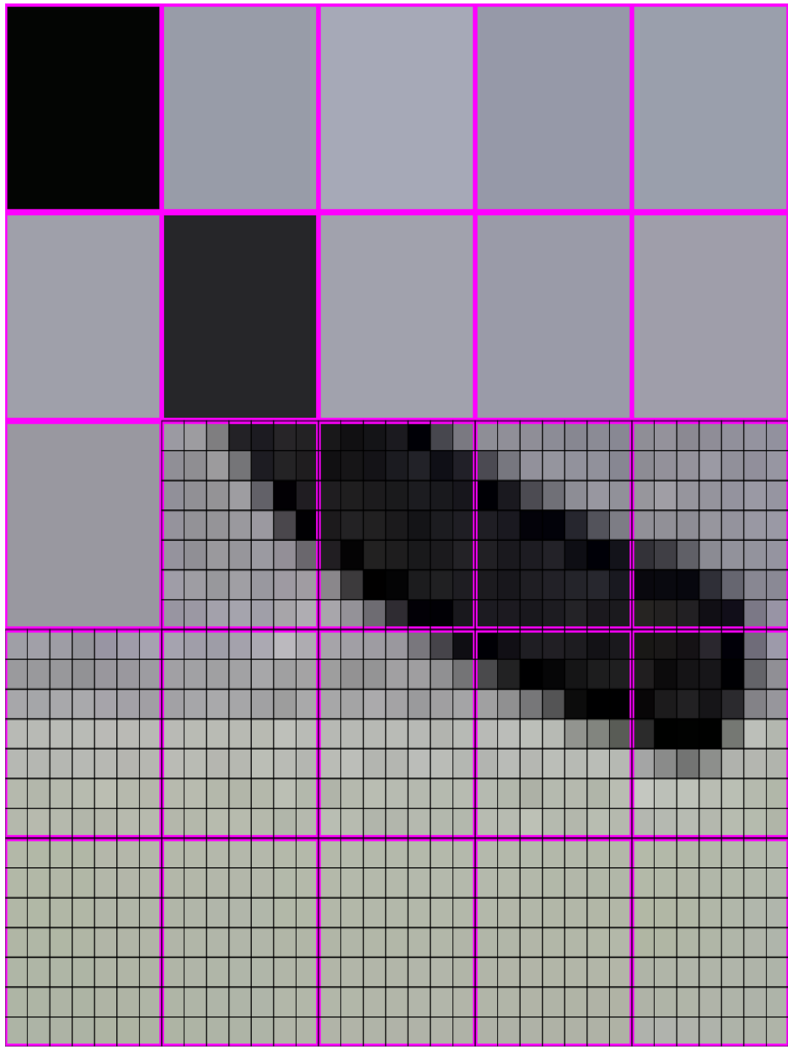
From 448×448 to 64×64



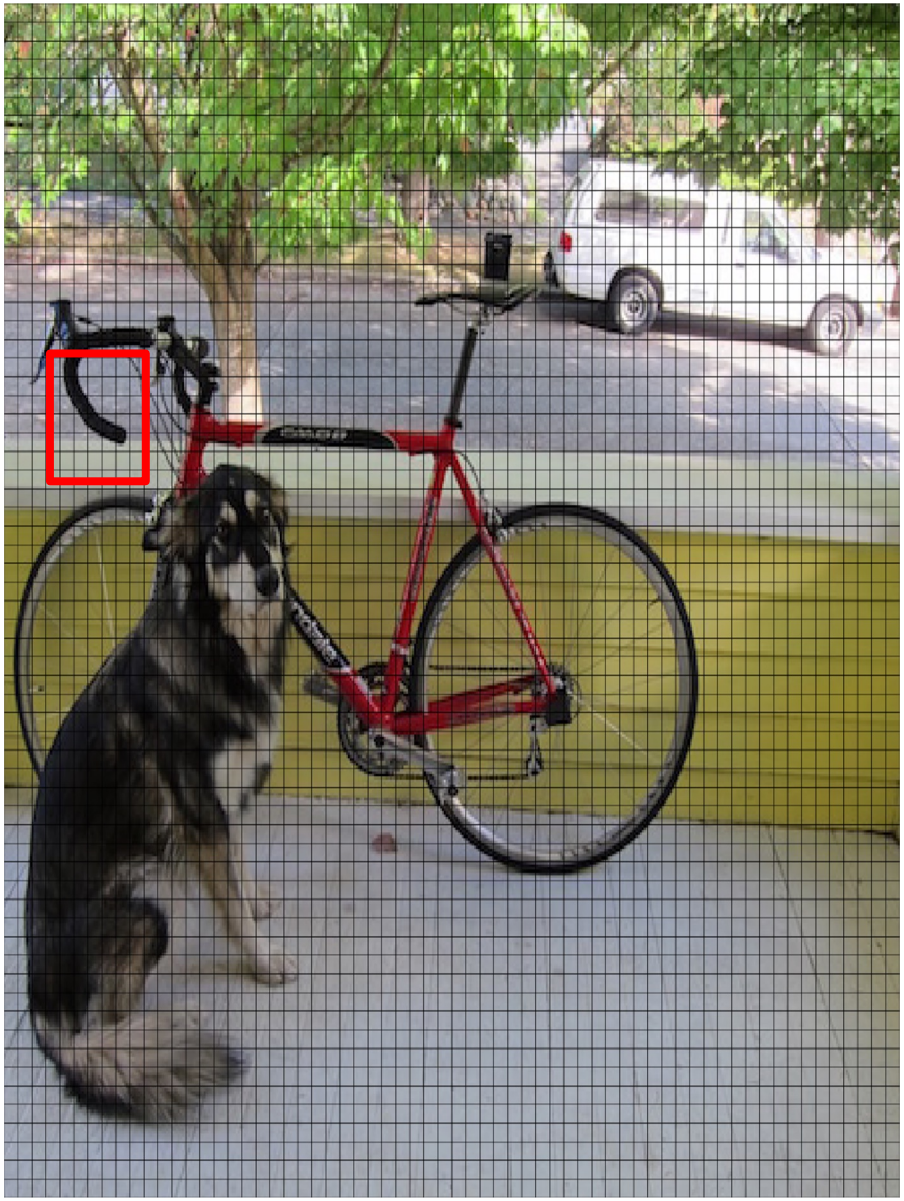


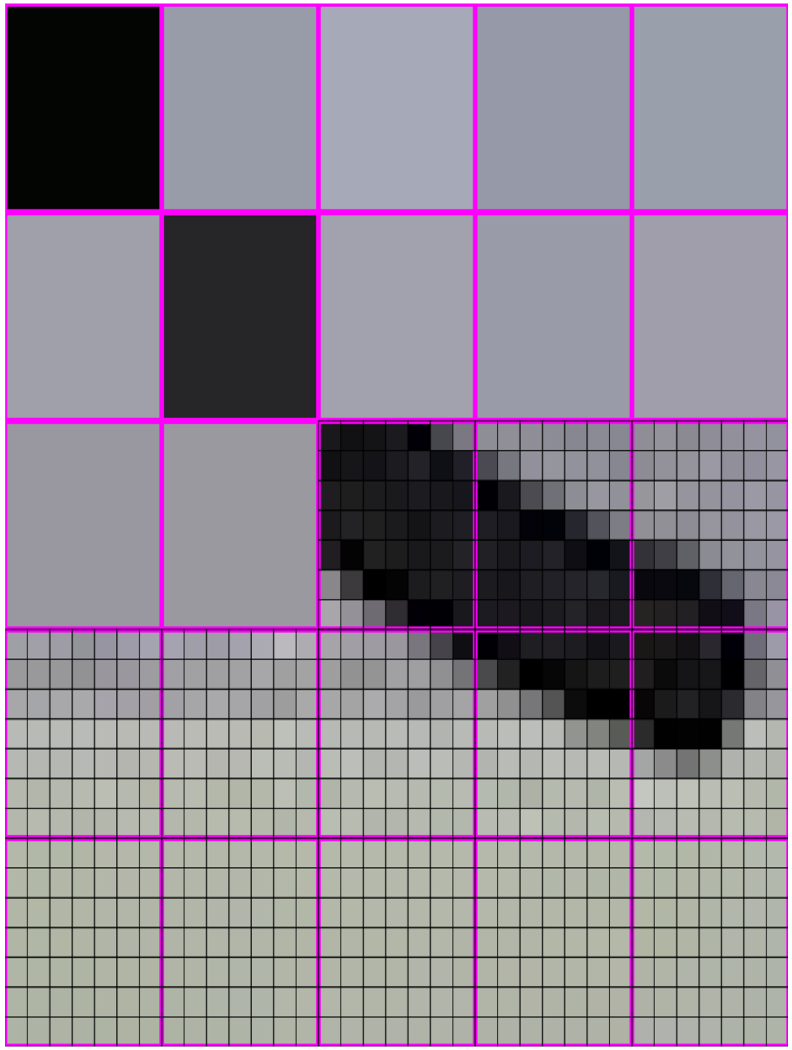
From 448×448 to 64×64



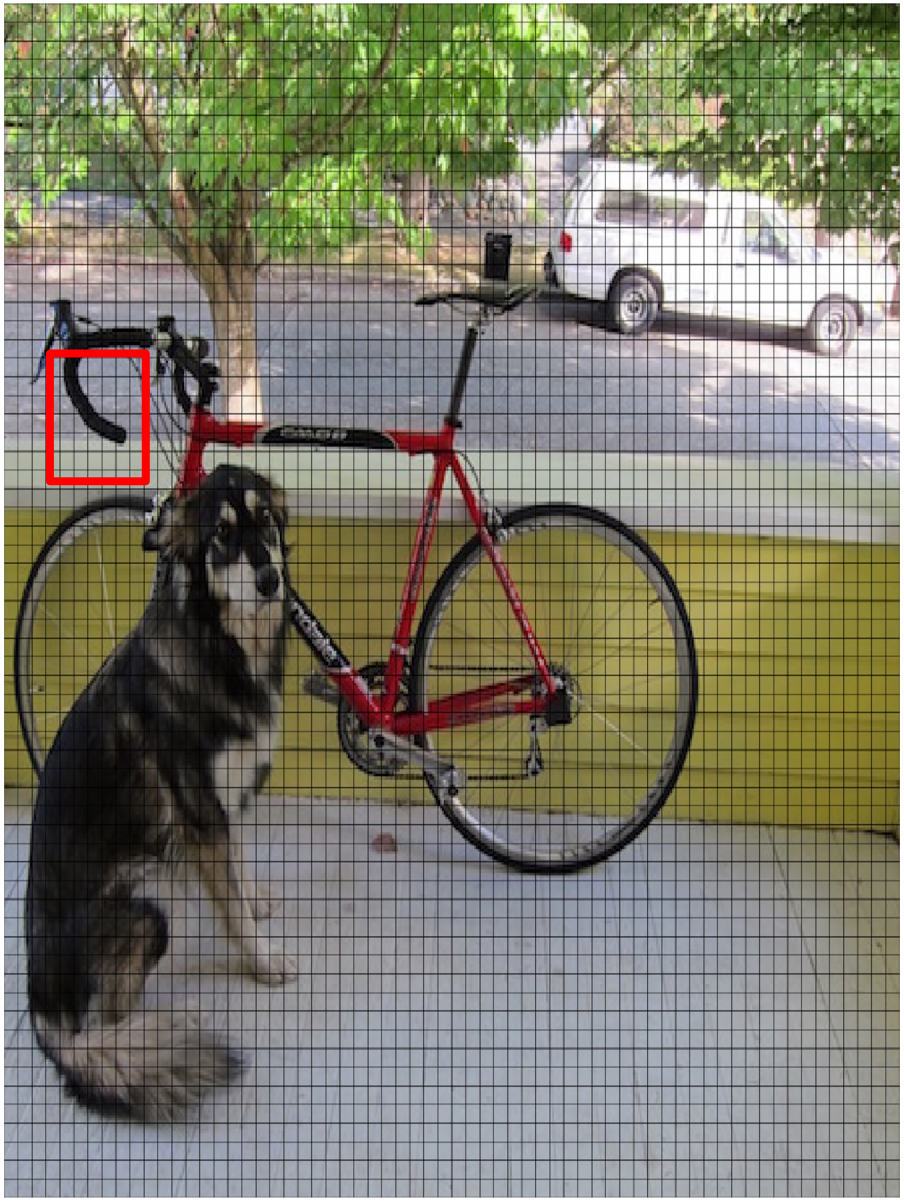


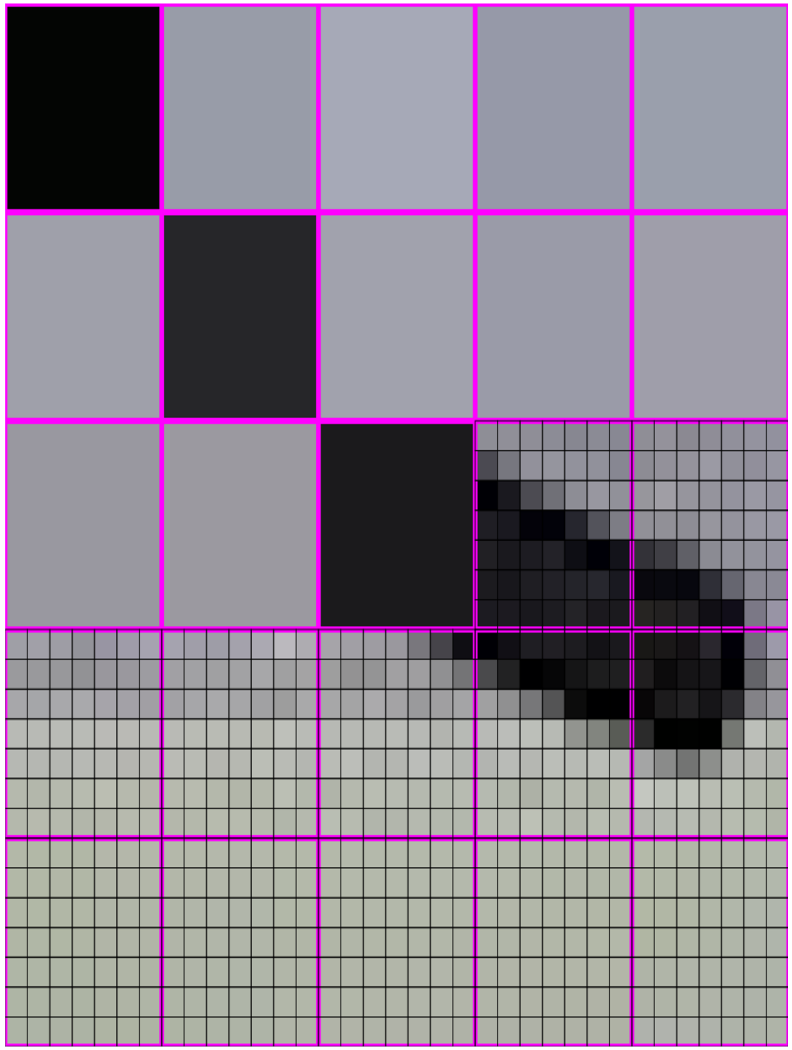
From 448×448 to 64×64



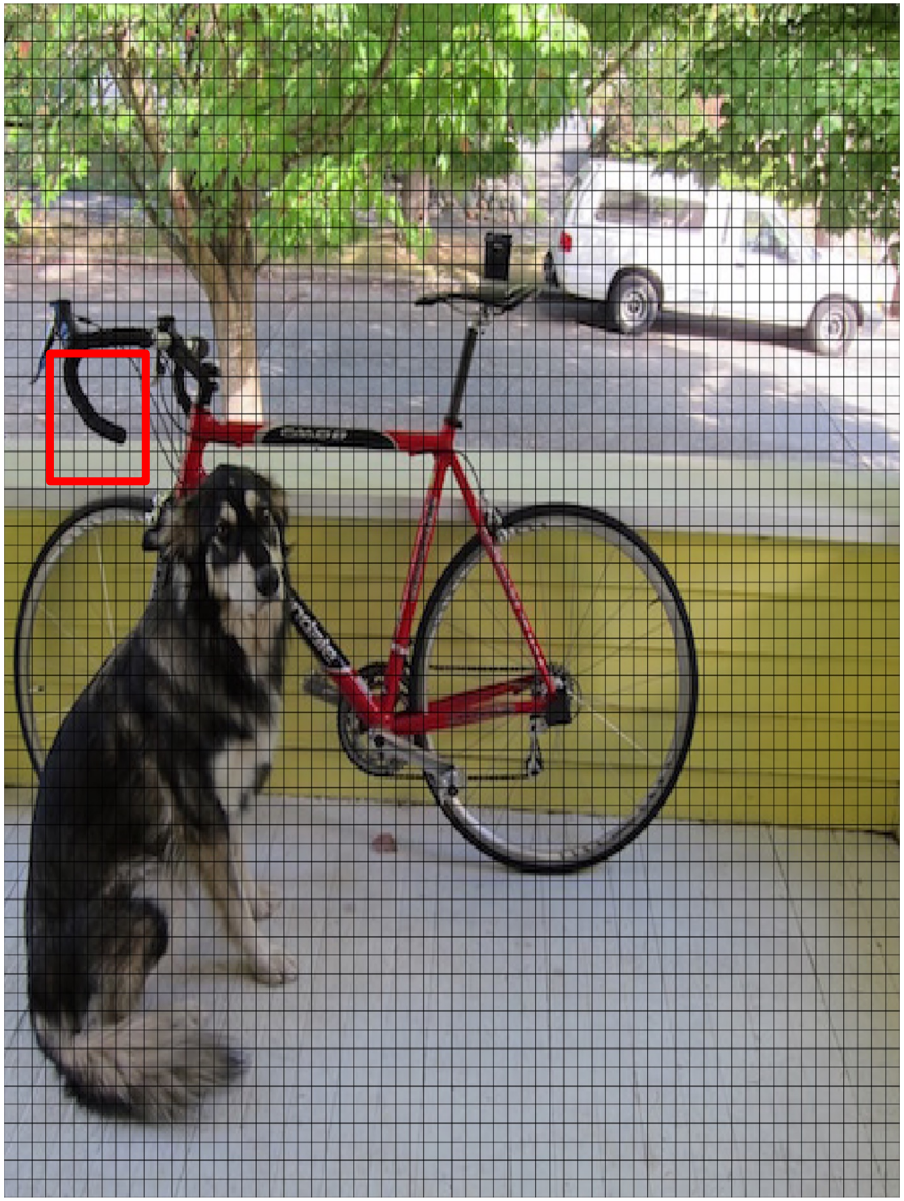


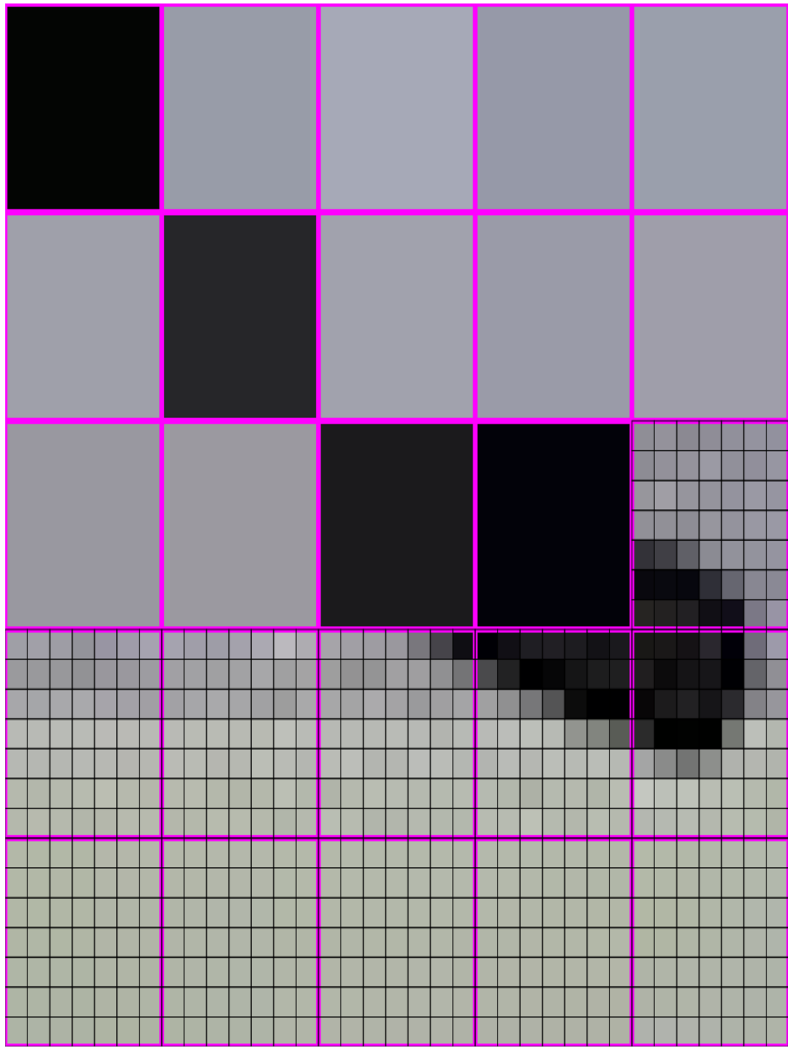
From 448×448 to 64×64



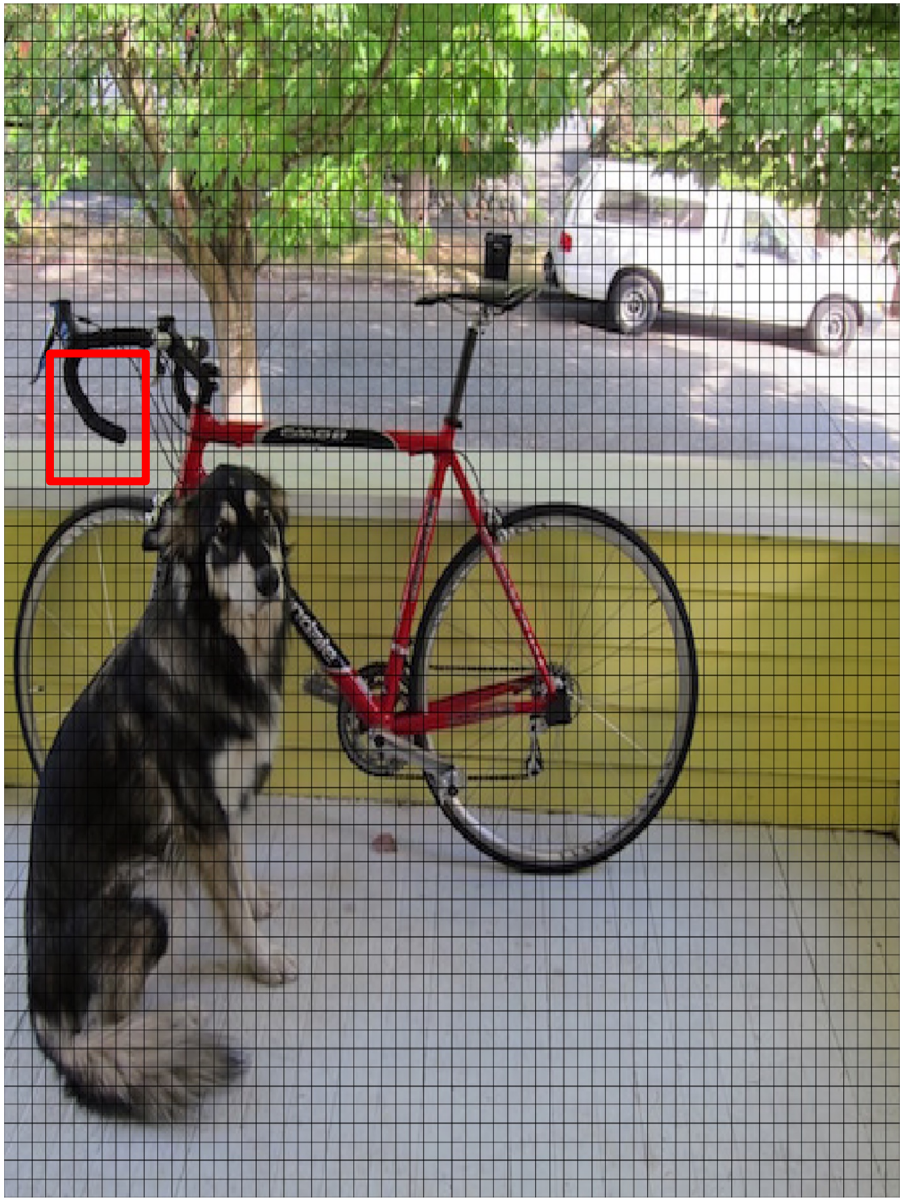


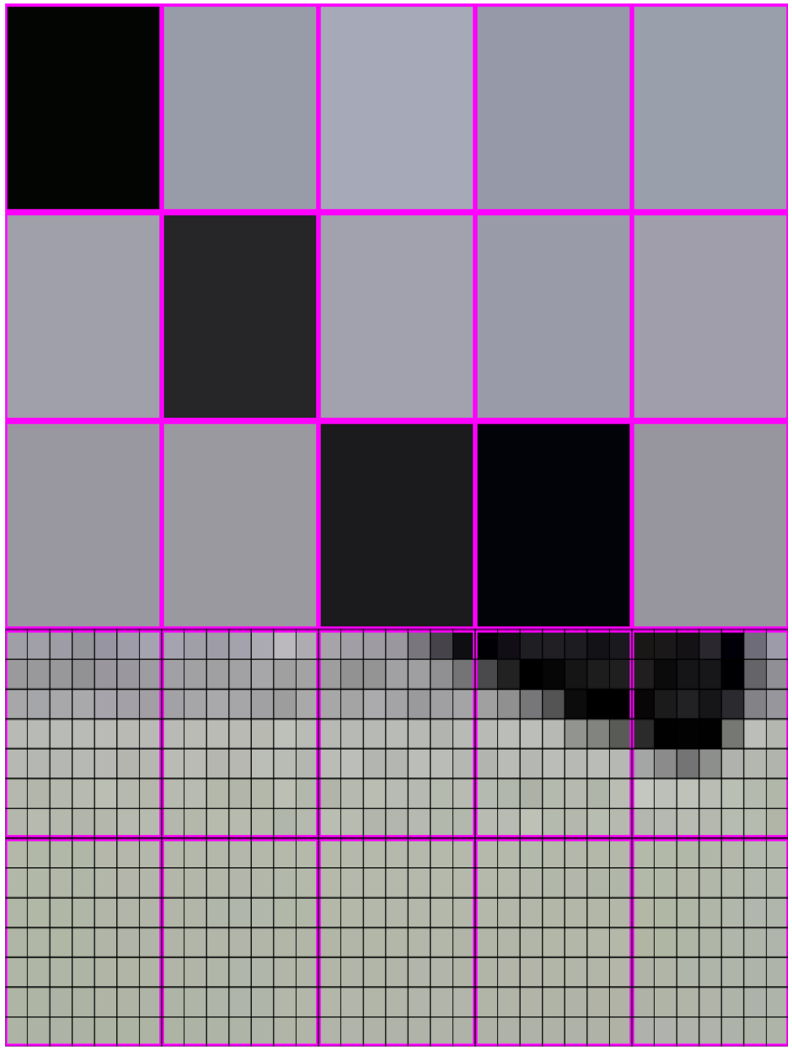
From 448×448 to 64×64



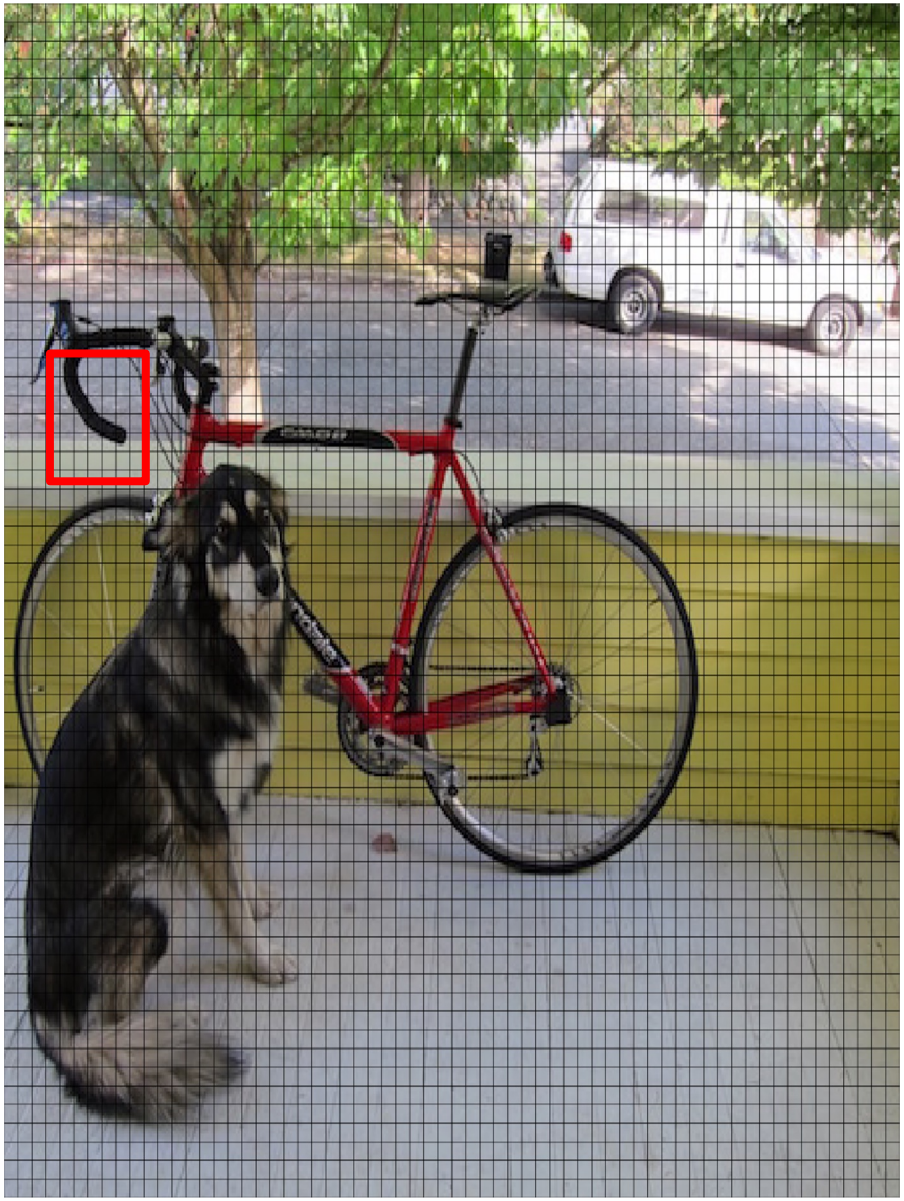


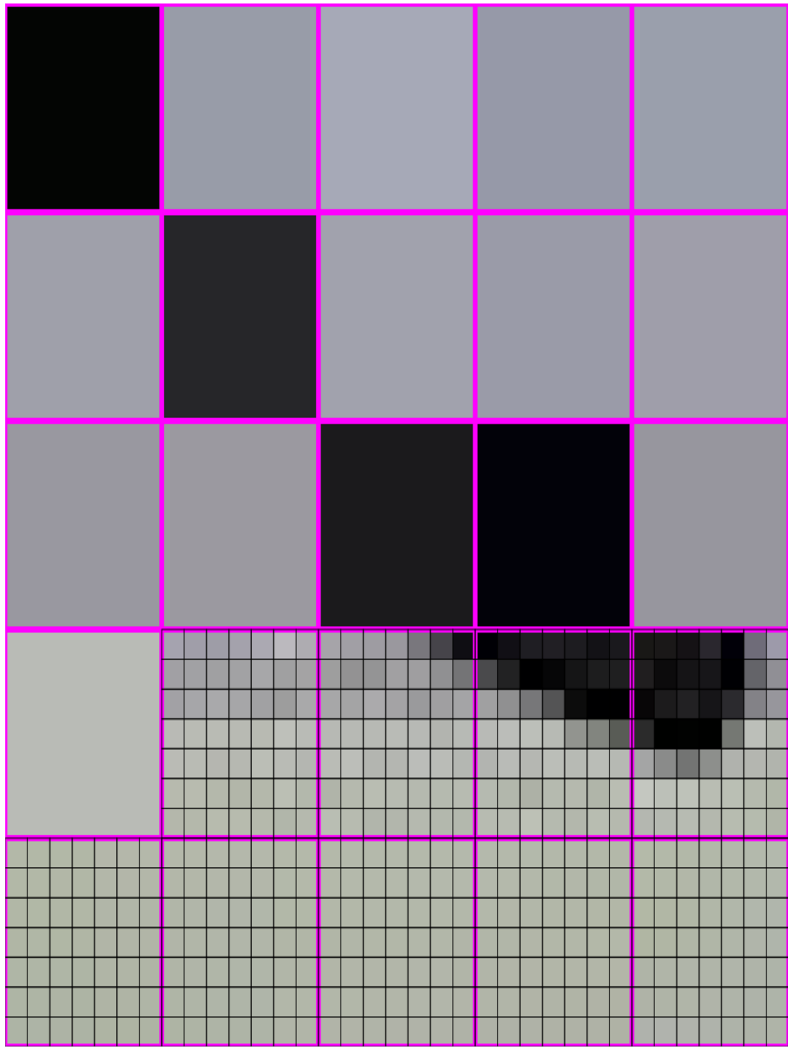
From 448×448 to 64×64



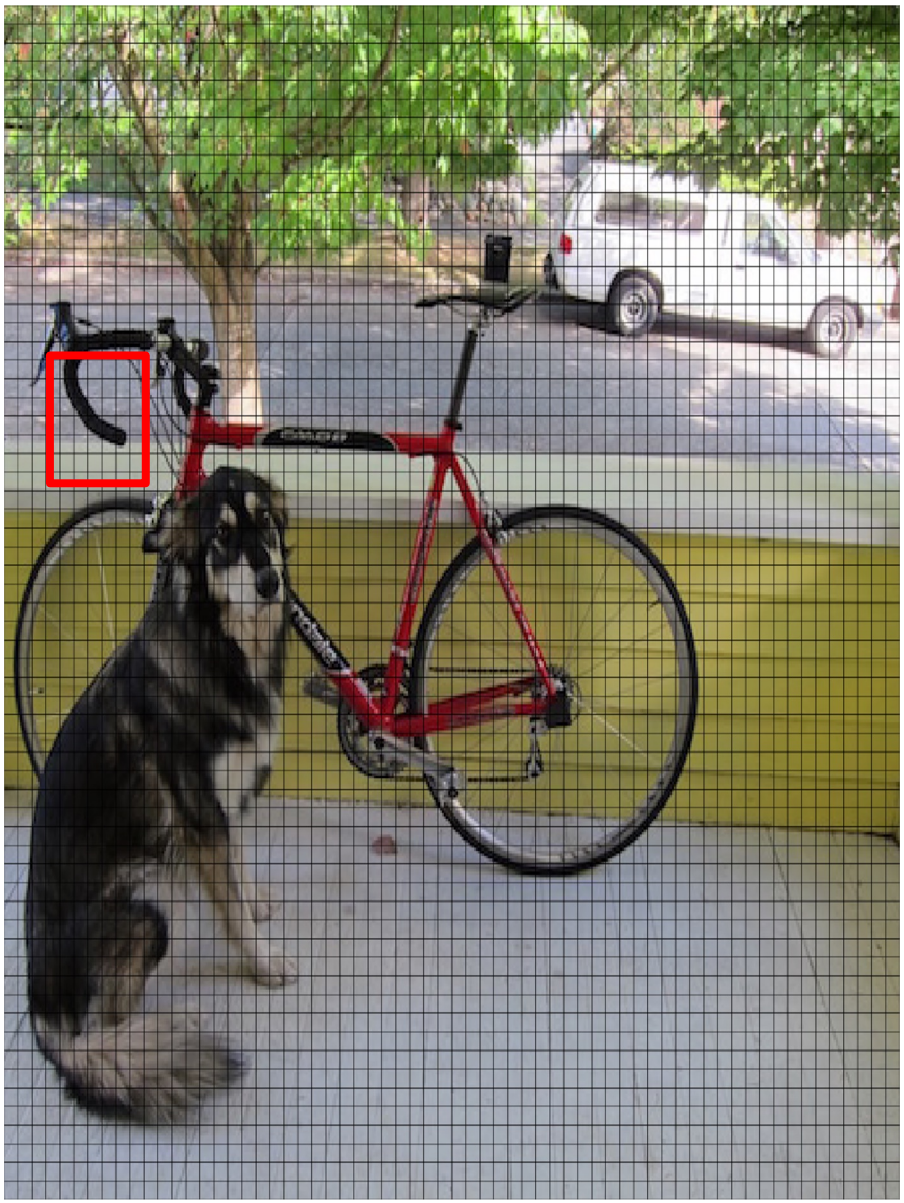


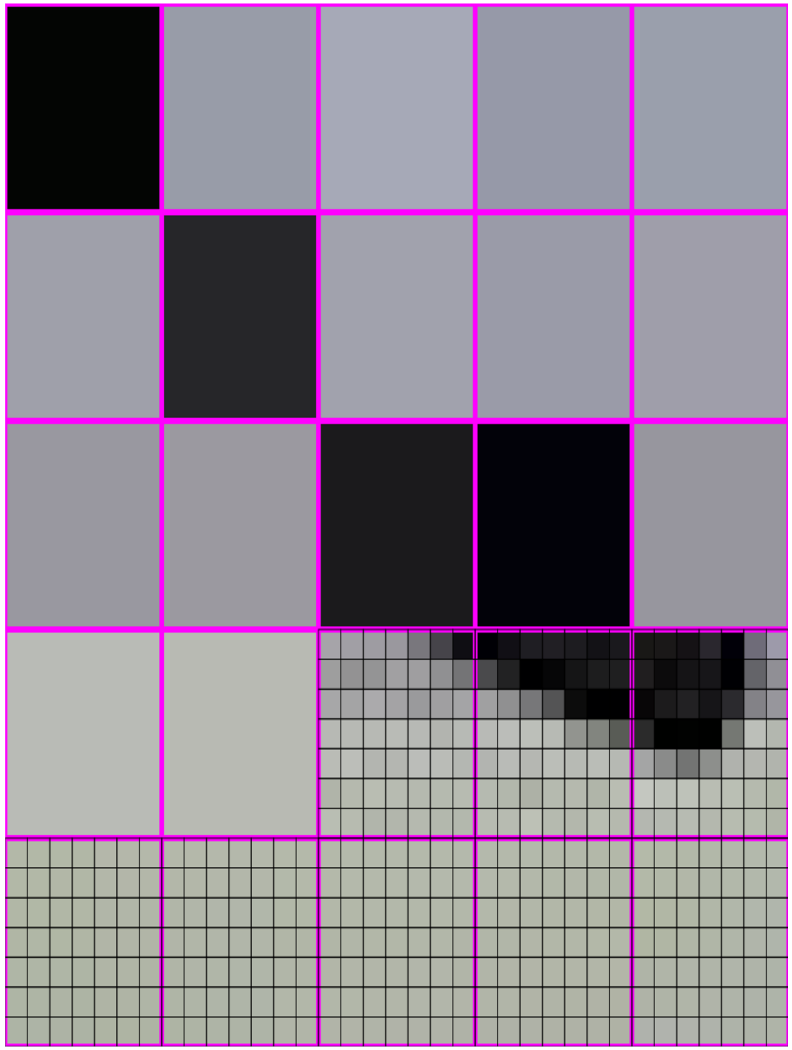
From 448×448 to 64×64



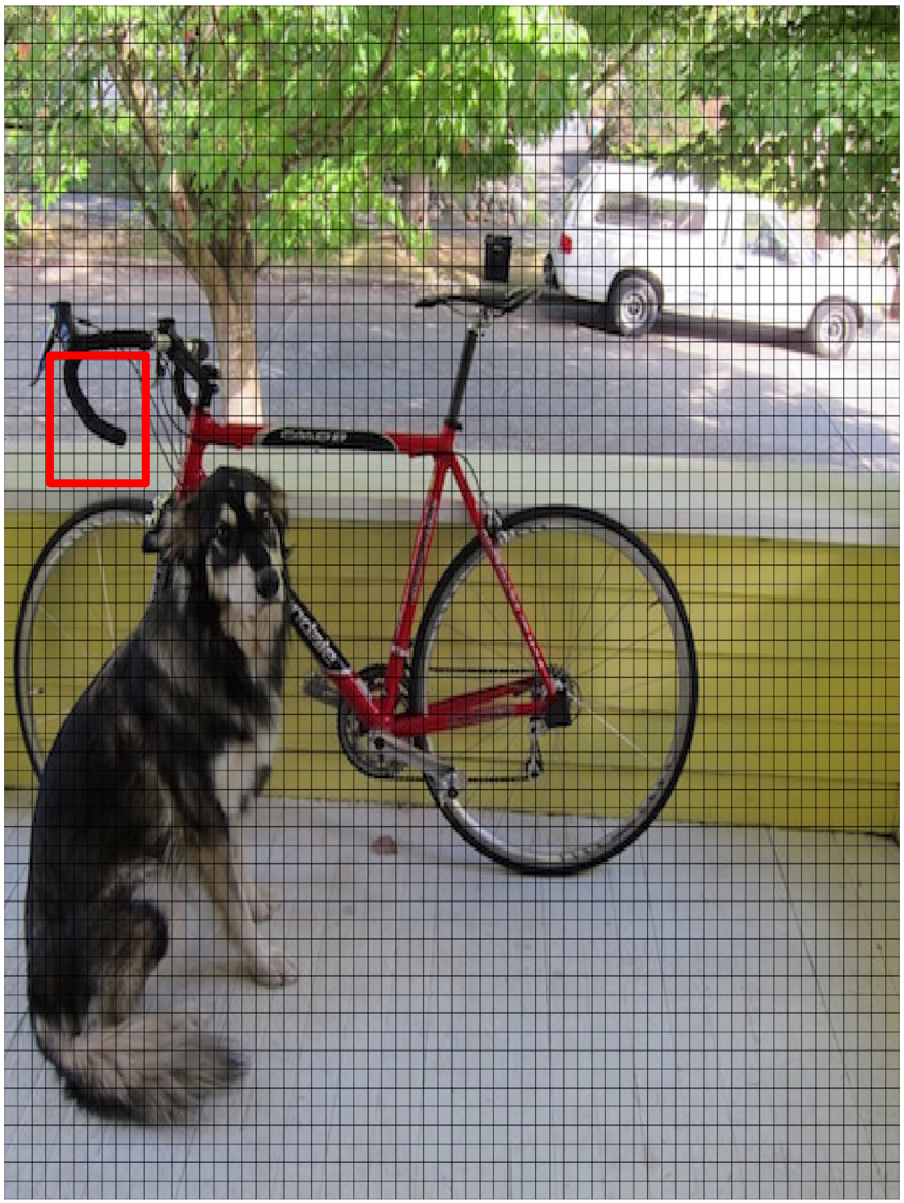


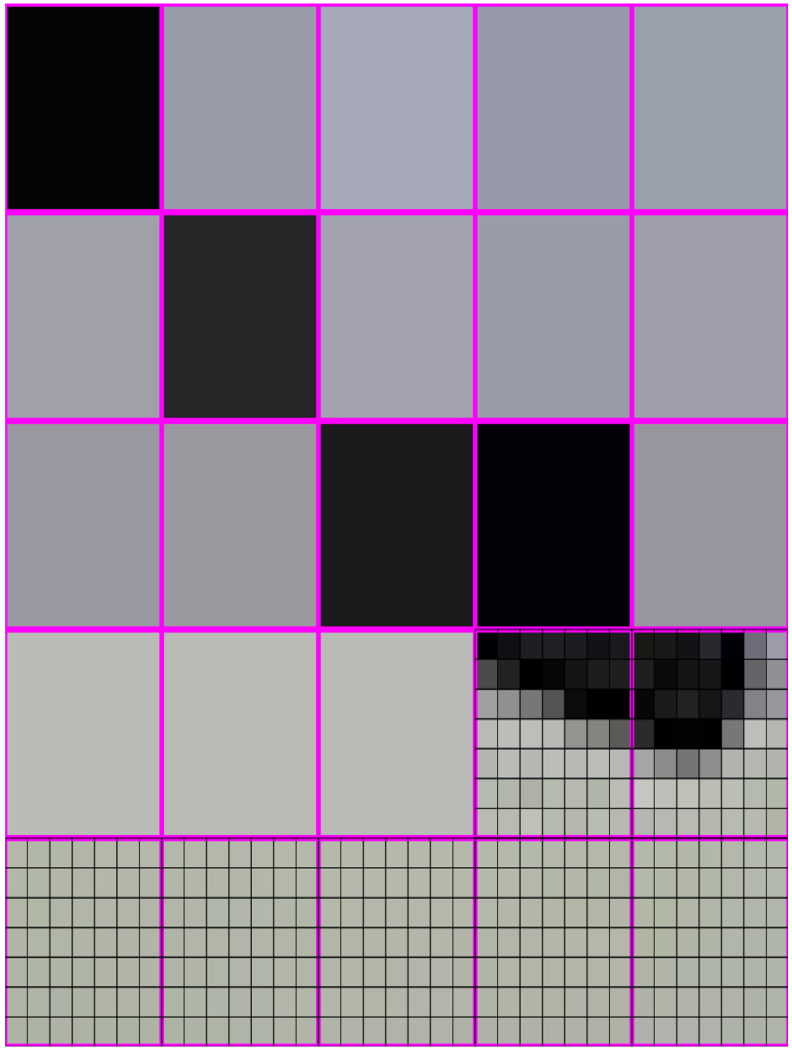
From 448×448 to 64×64



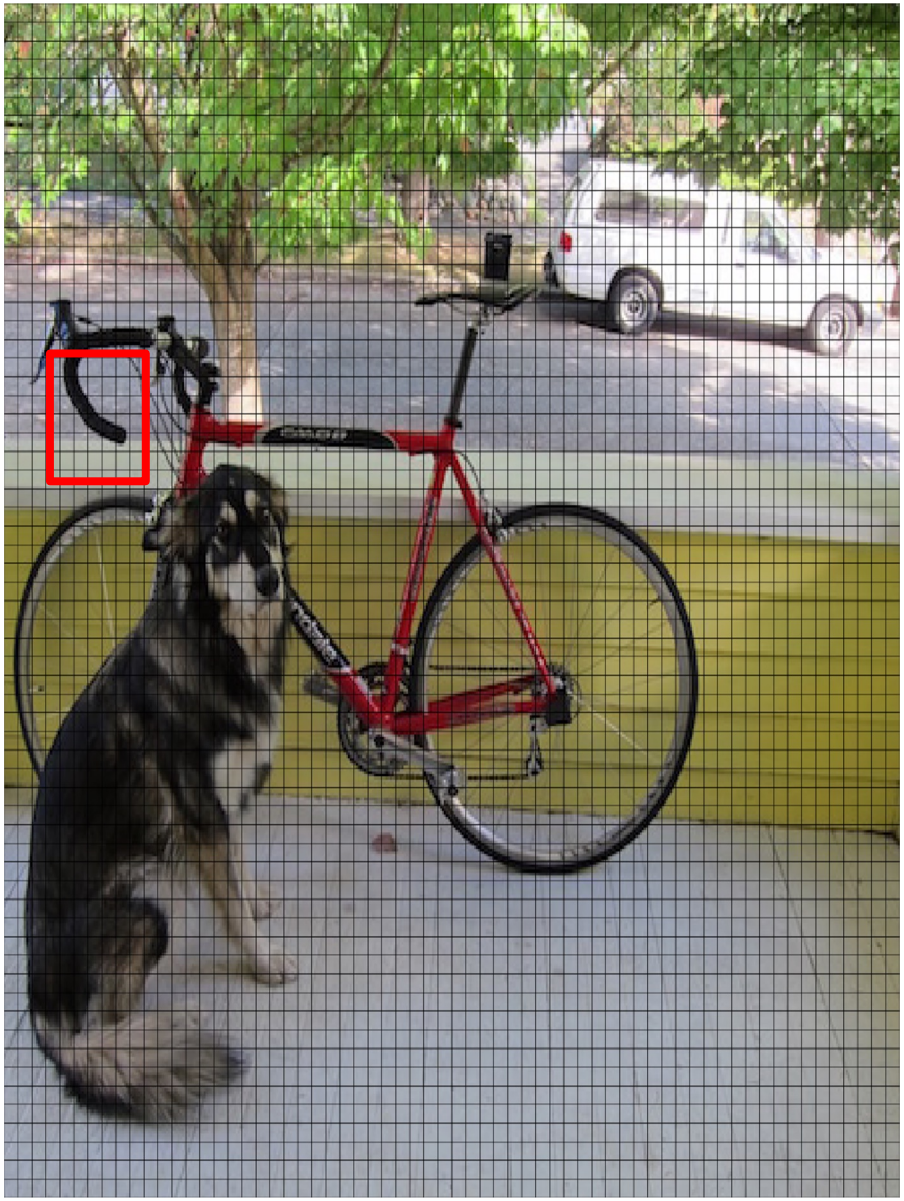


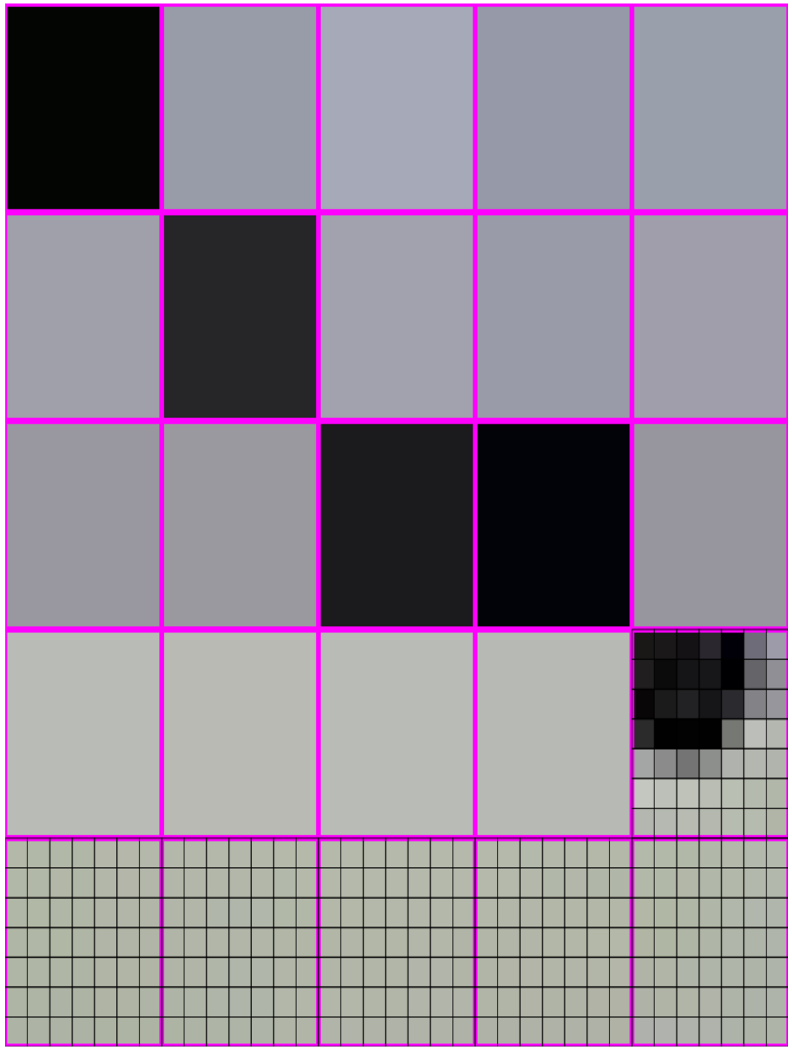
From 448×448 to 64×64



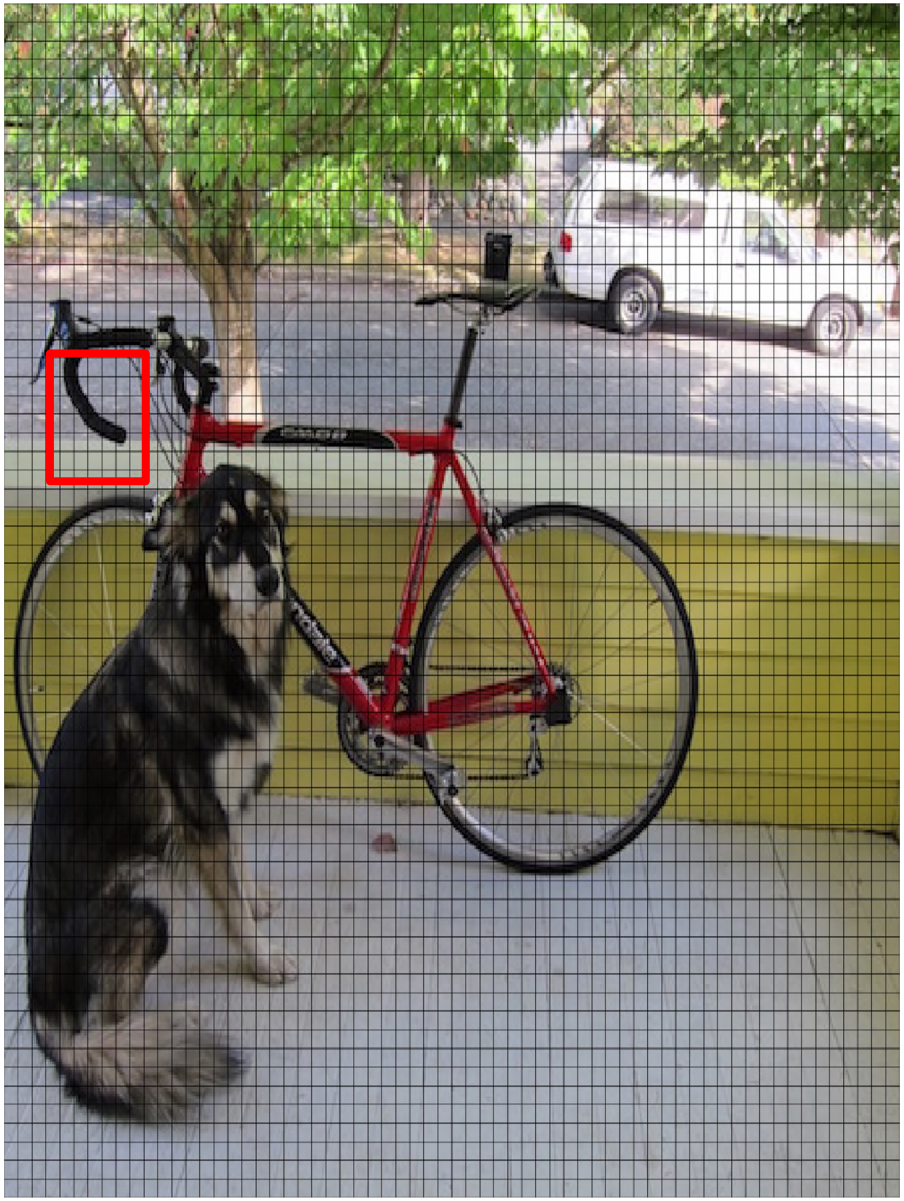


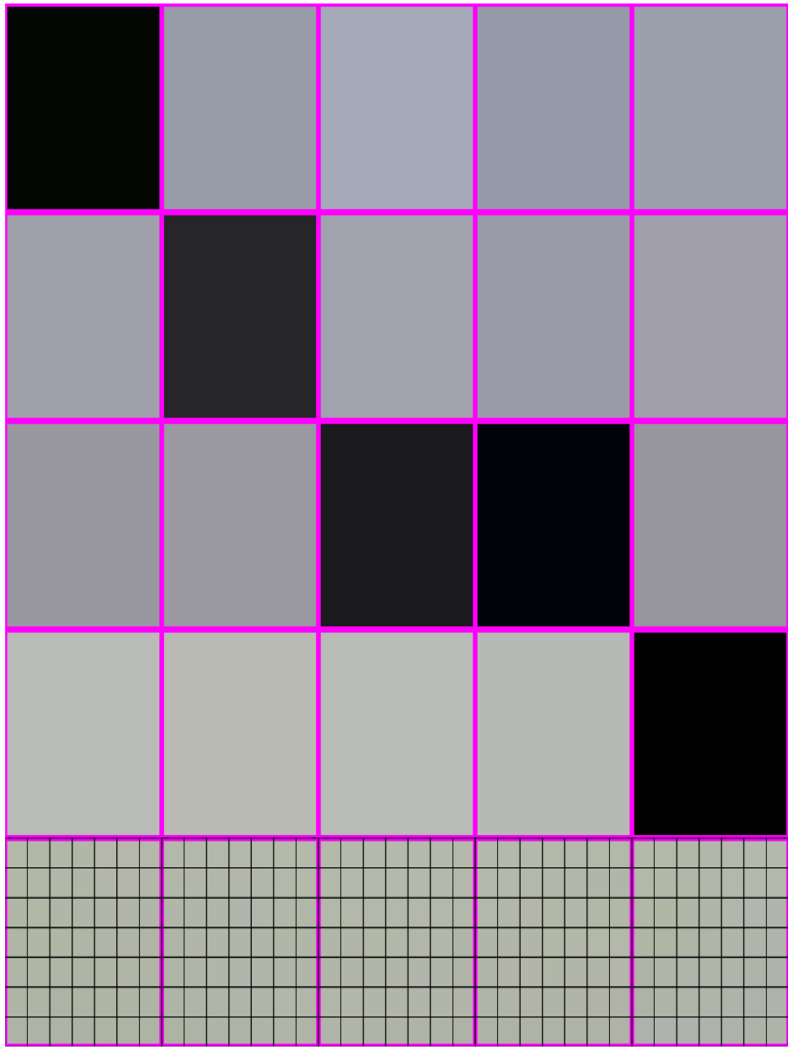
From 448×448 to 64×64



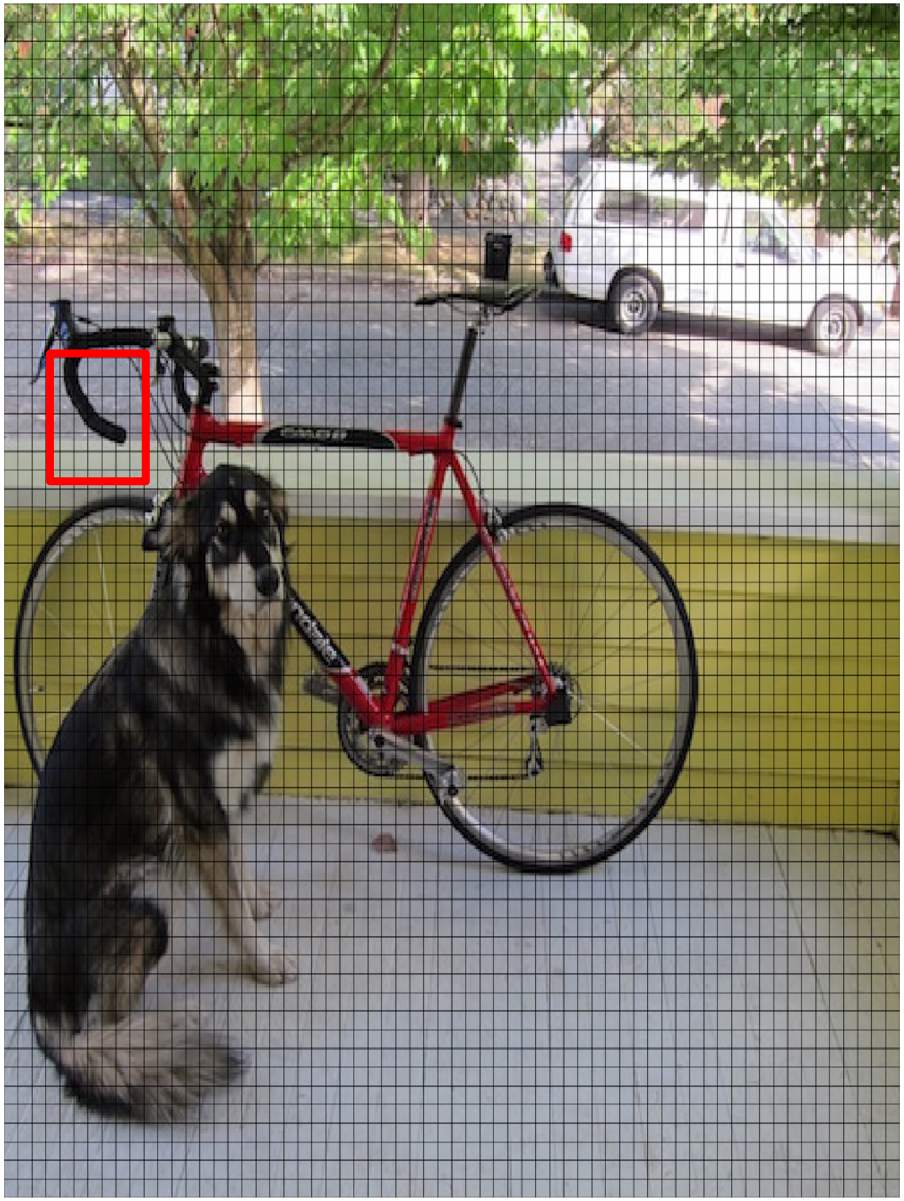


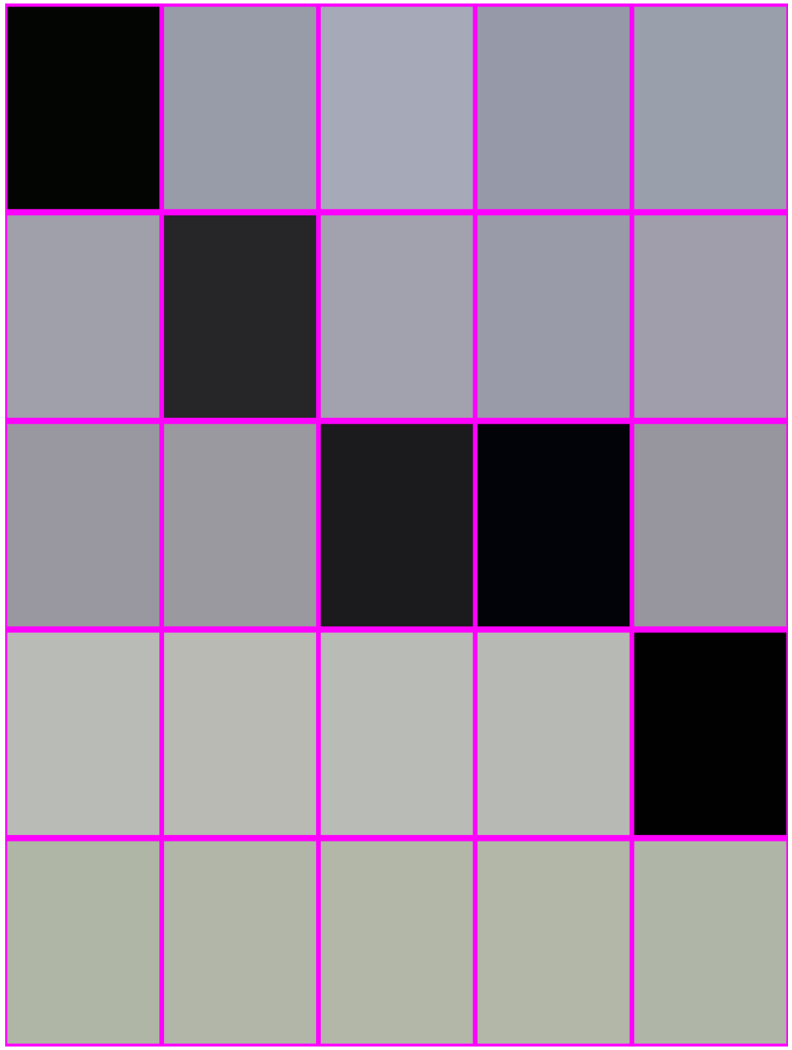
From 448×448 to 64×64



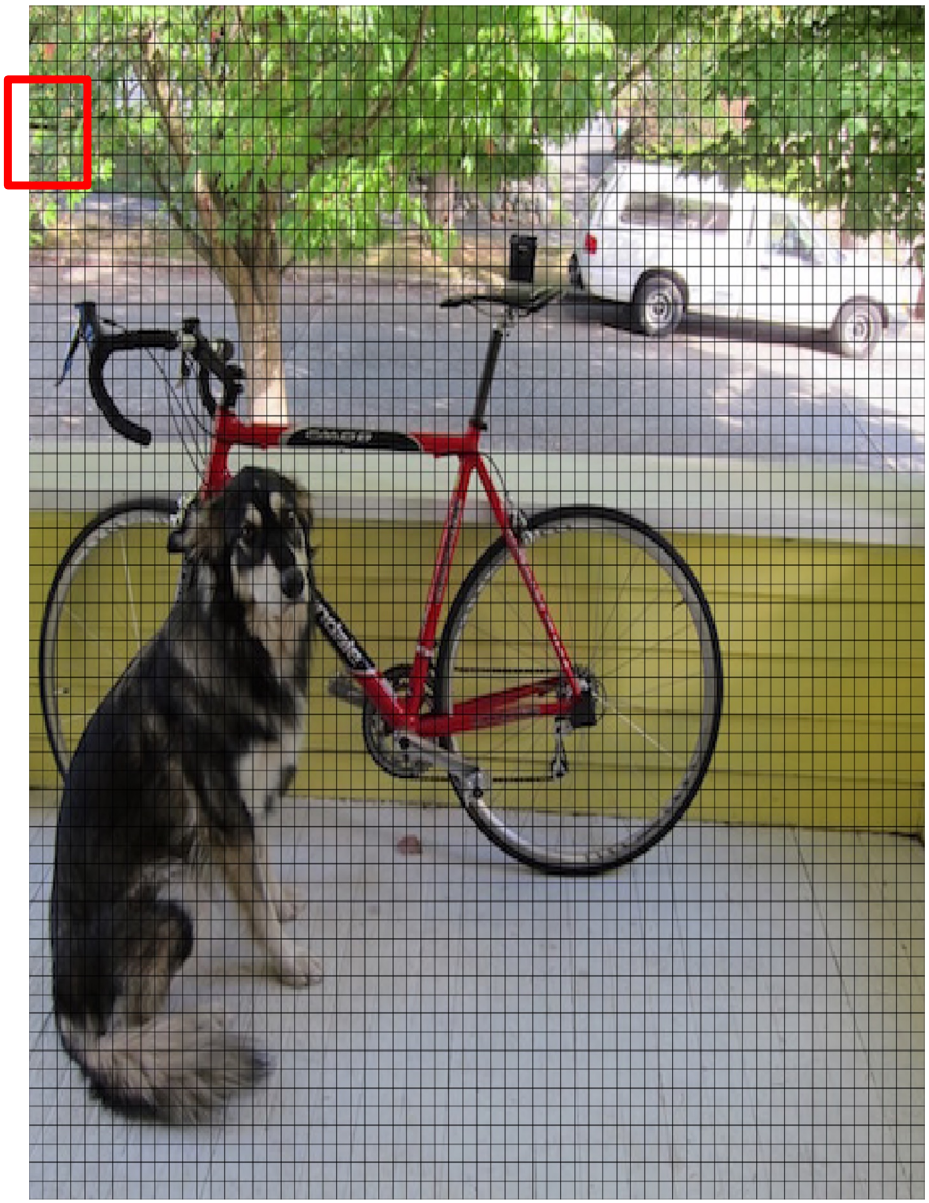


From 448×448 to 64×64

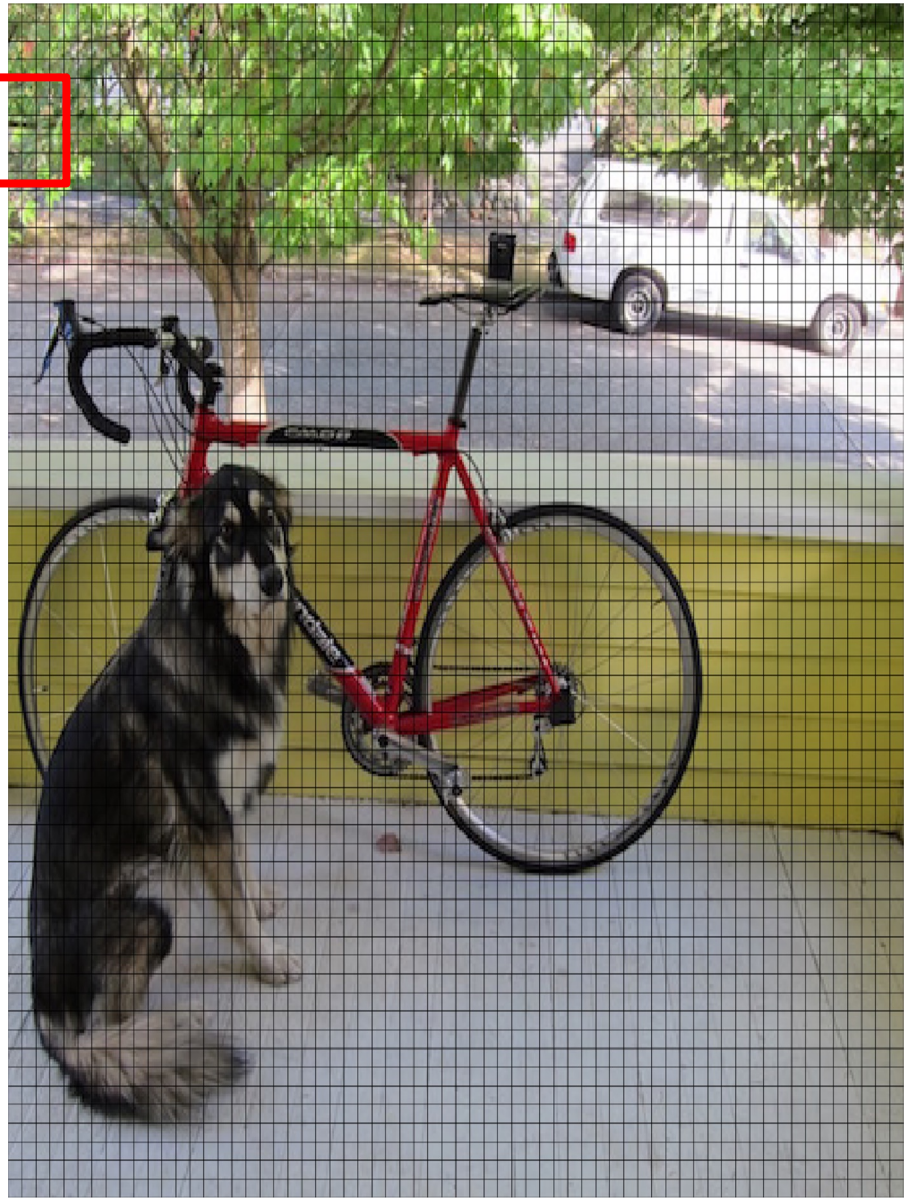


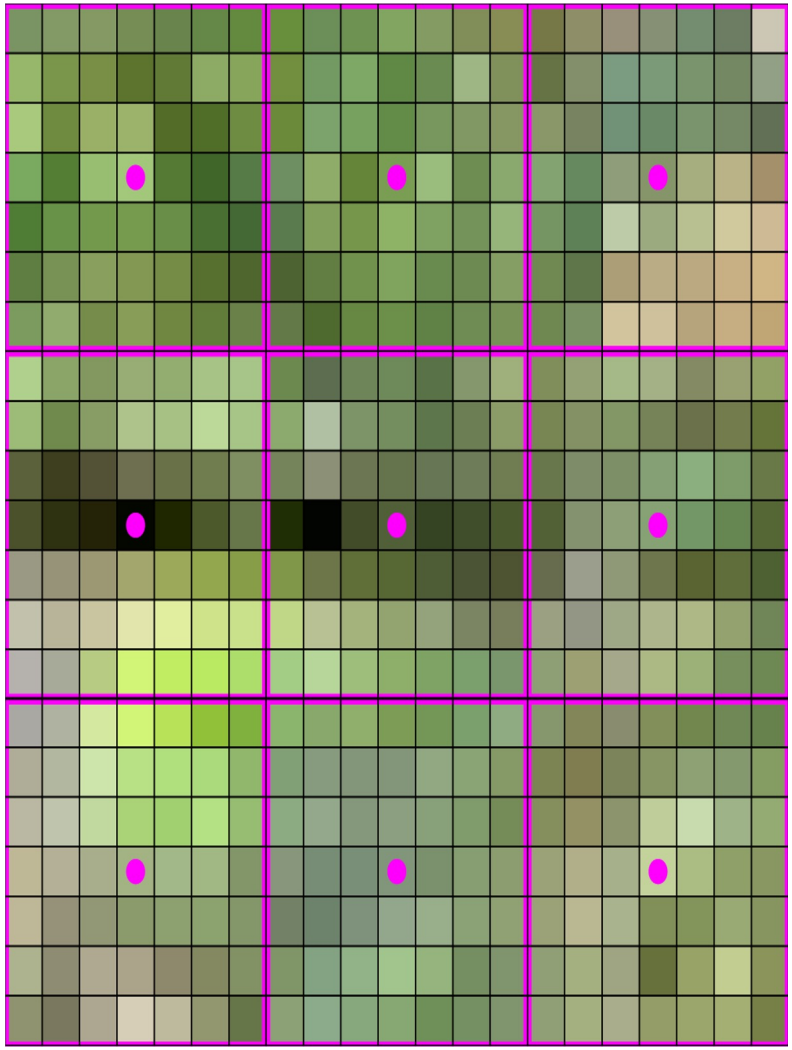


From 448×448 to 64×64

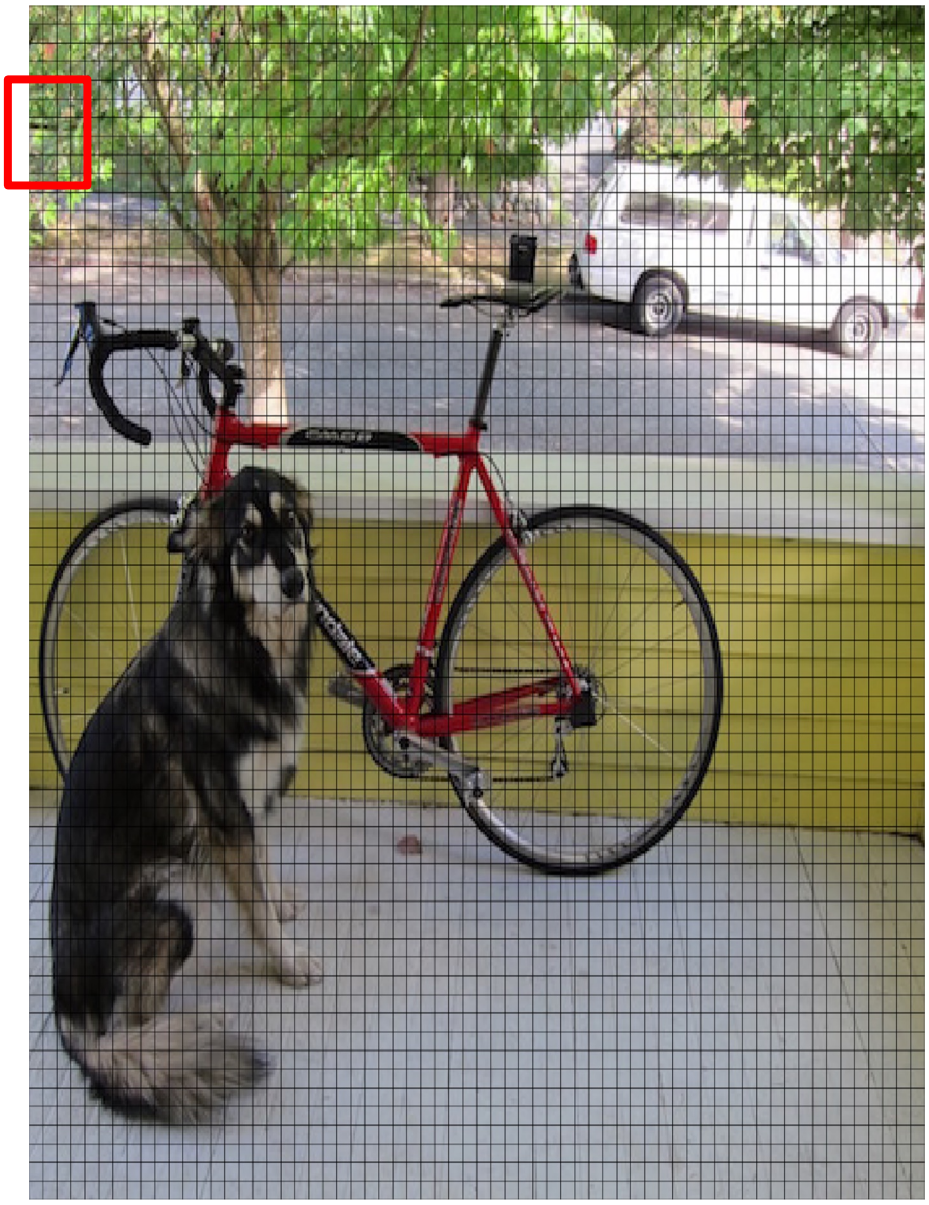


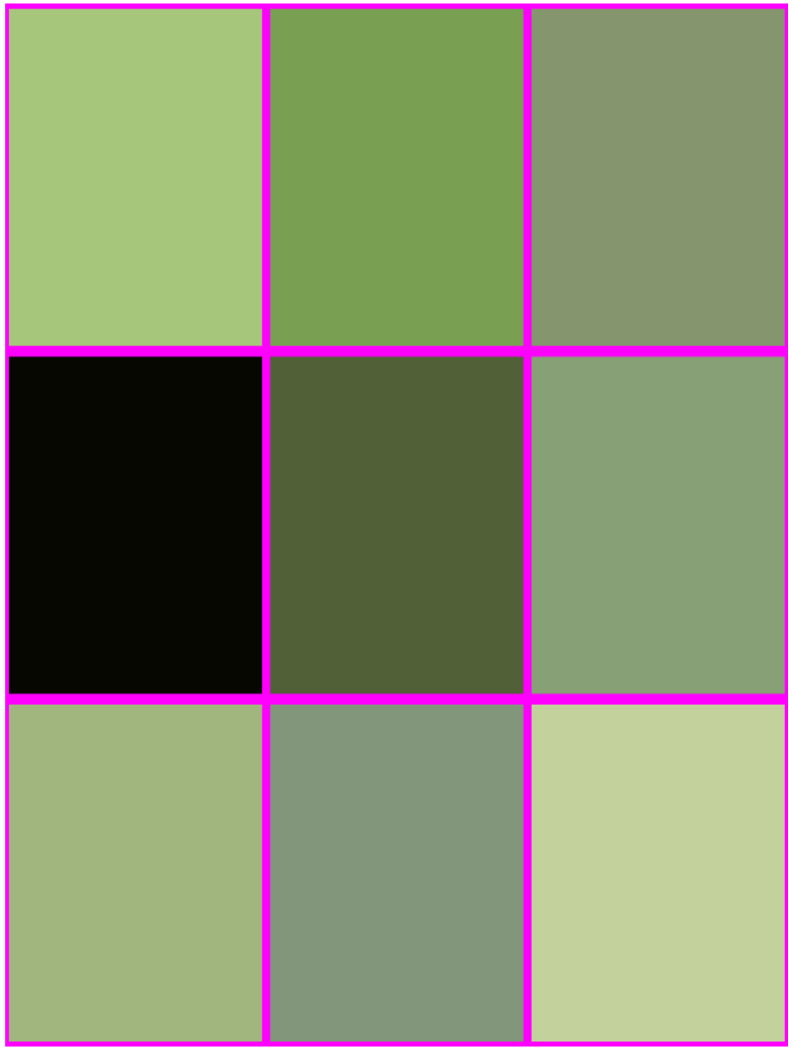
From 448×448 to 64×64



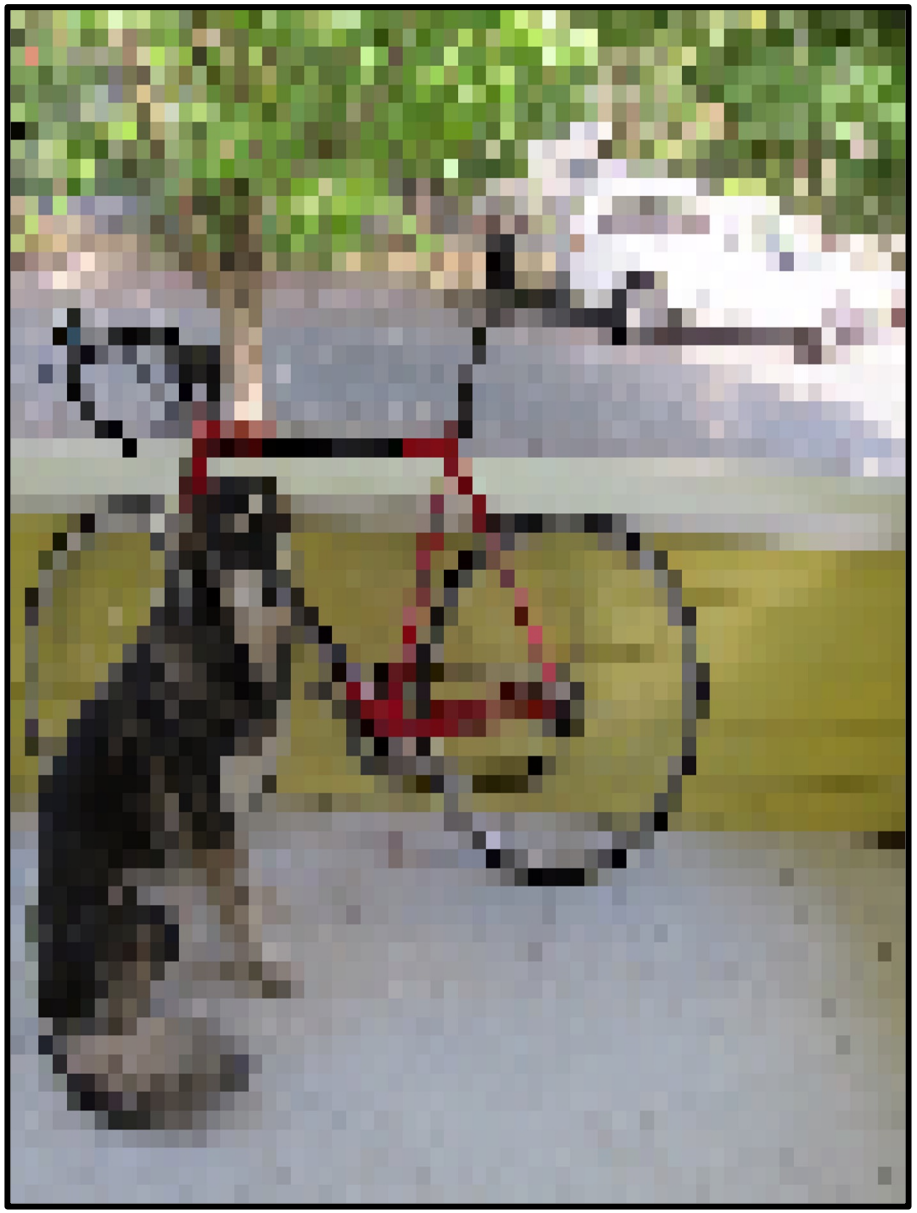


From 448×448 to 64×64





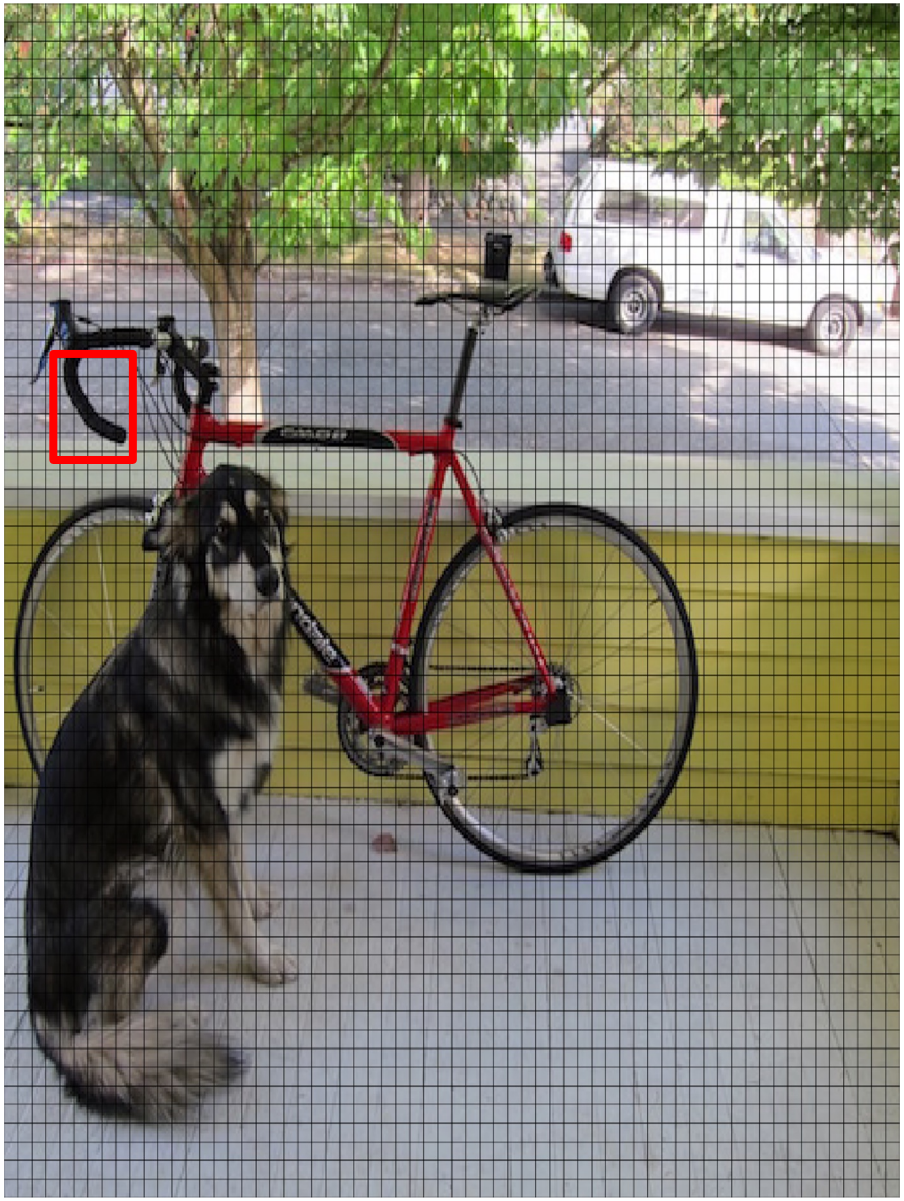
IS THIS ALL THERE IS??



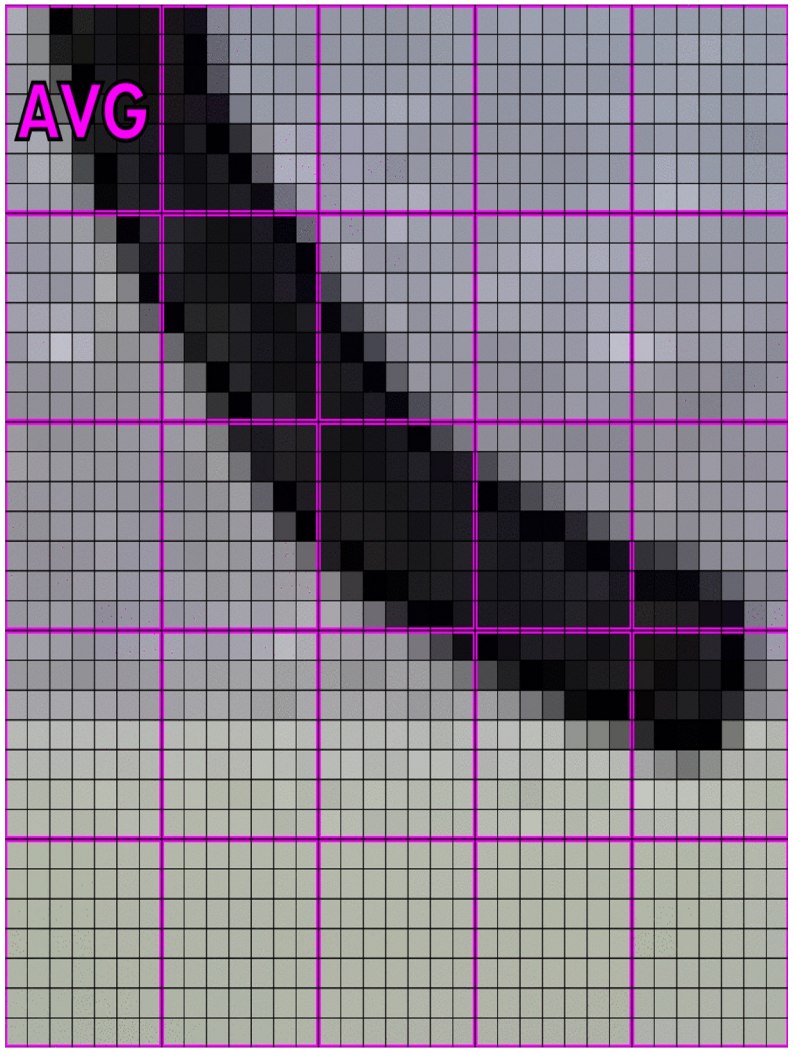
THERE IS A BETTER WAY!

OBTAIN MUCH BETTER RESULTS!

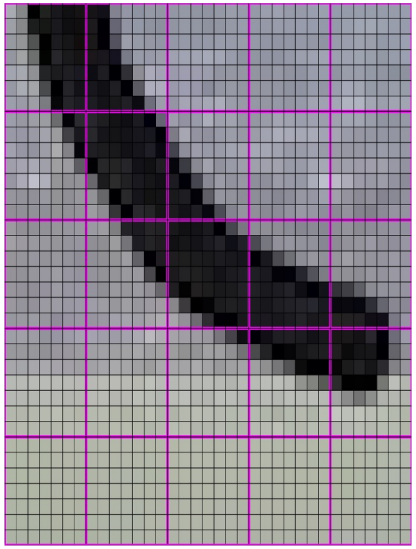
How to do it?



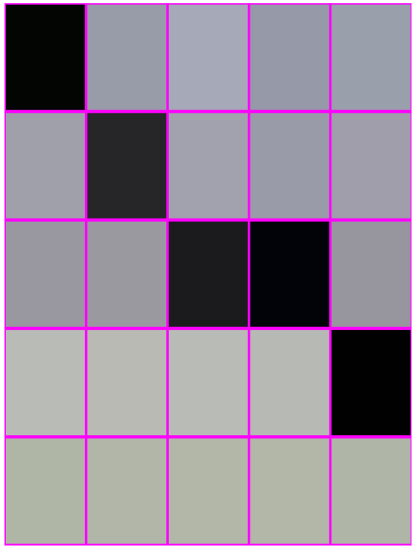
How to do it? Averaging!

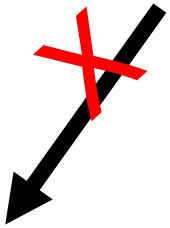


How to do it ? Averaging!

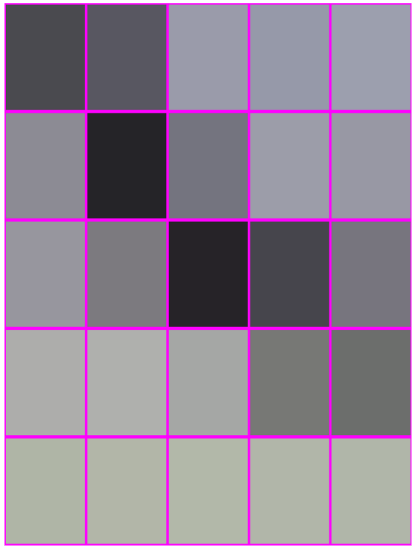


“interpolation”

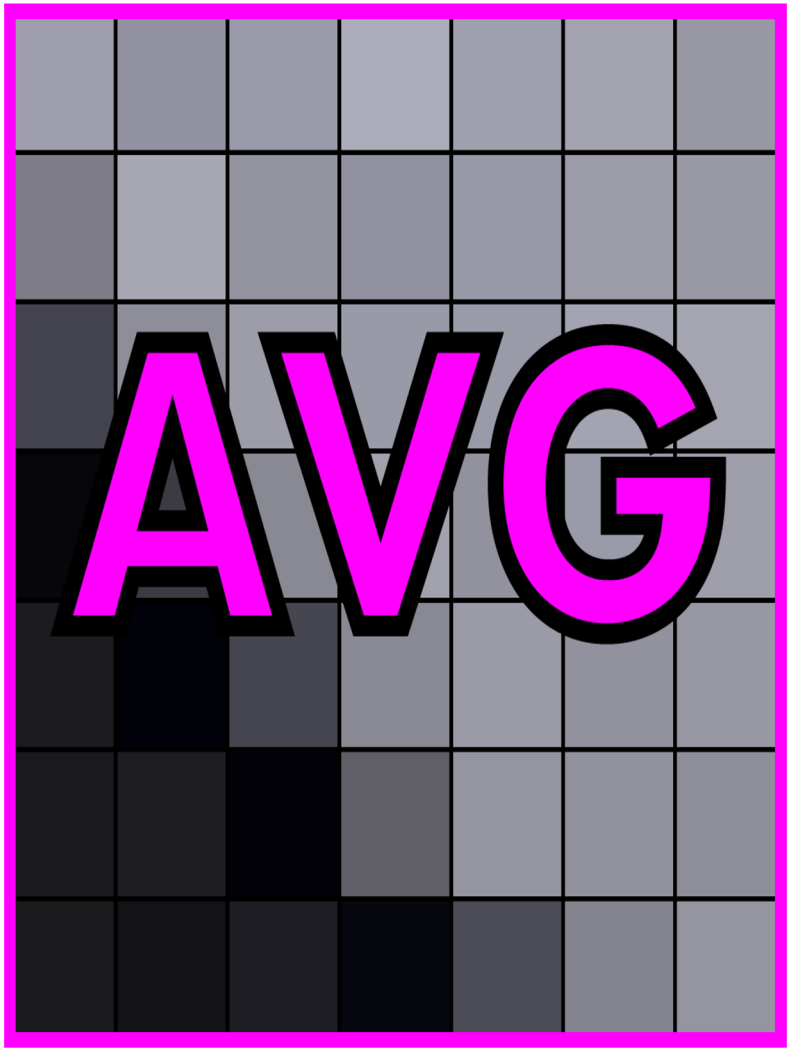


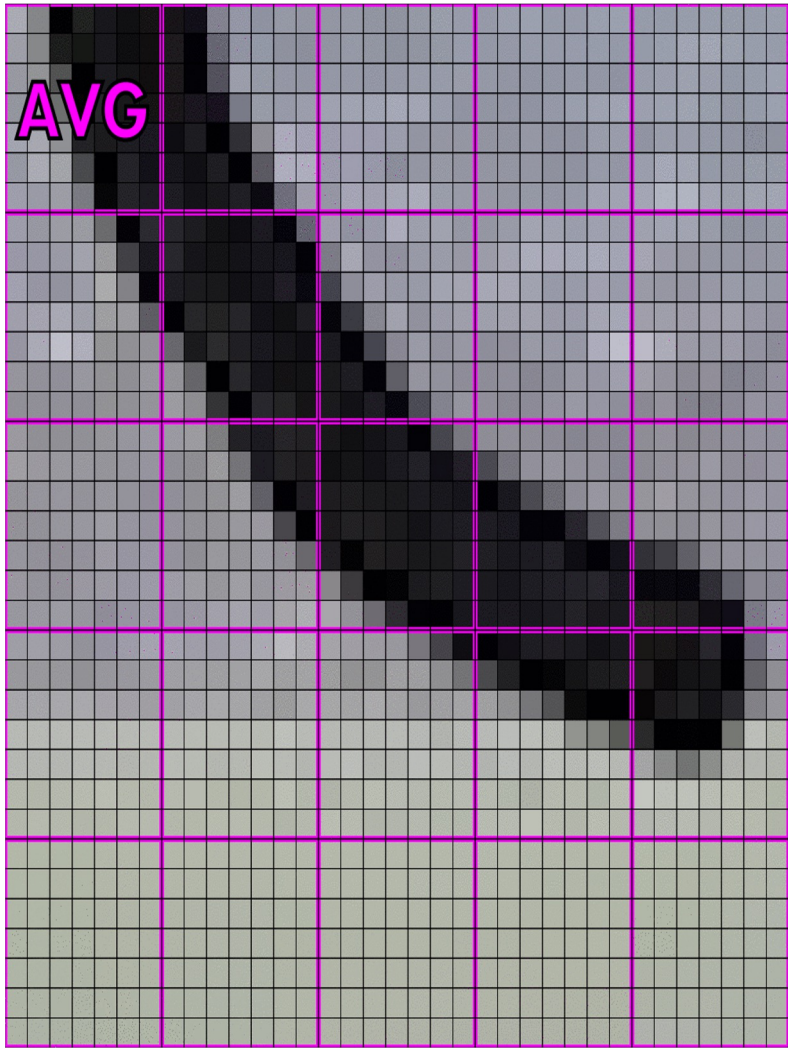


averaging

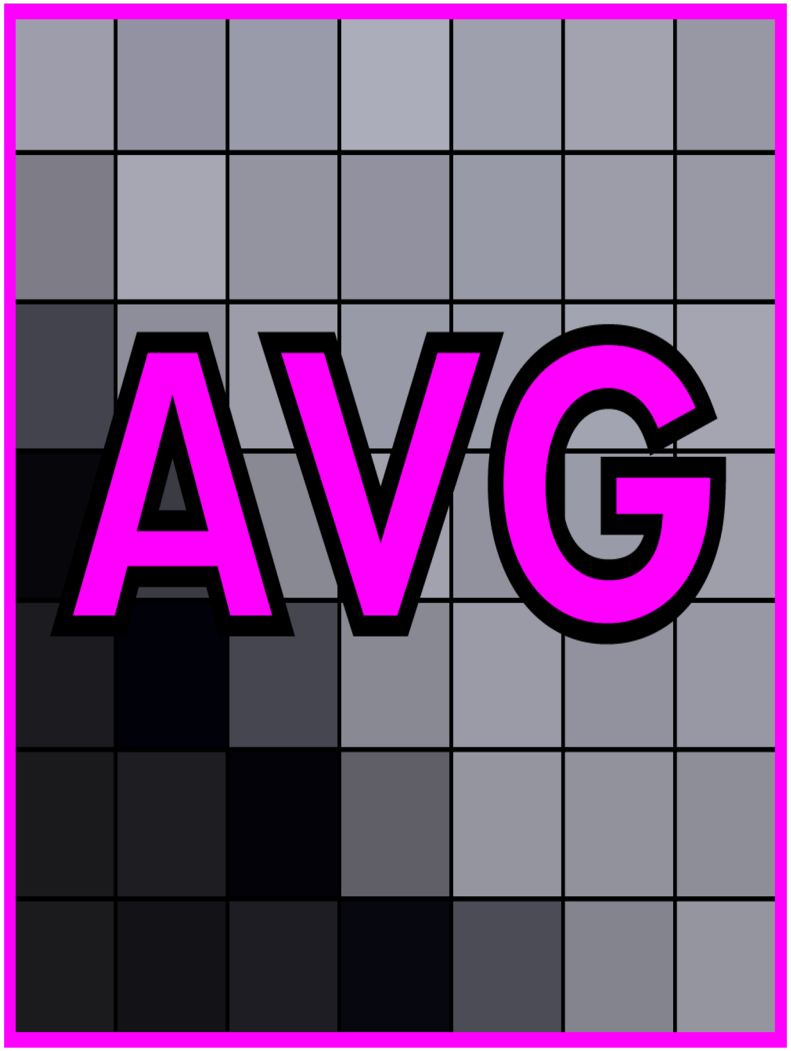


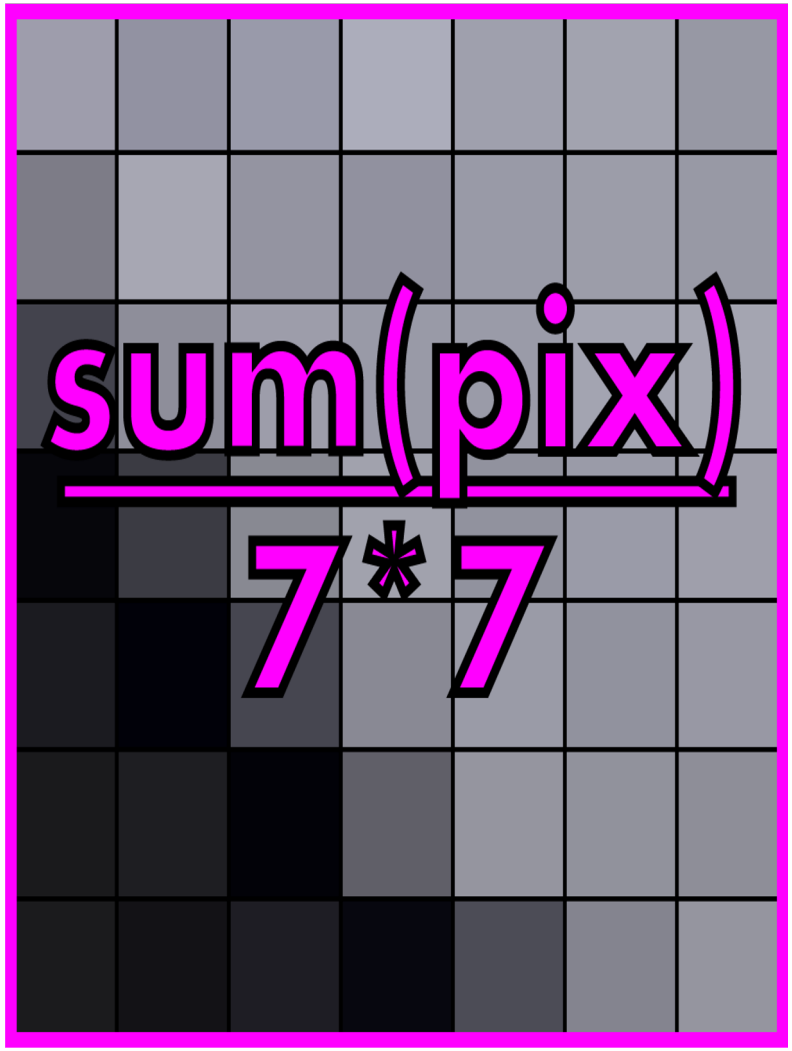
What is averaging?



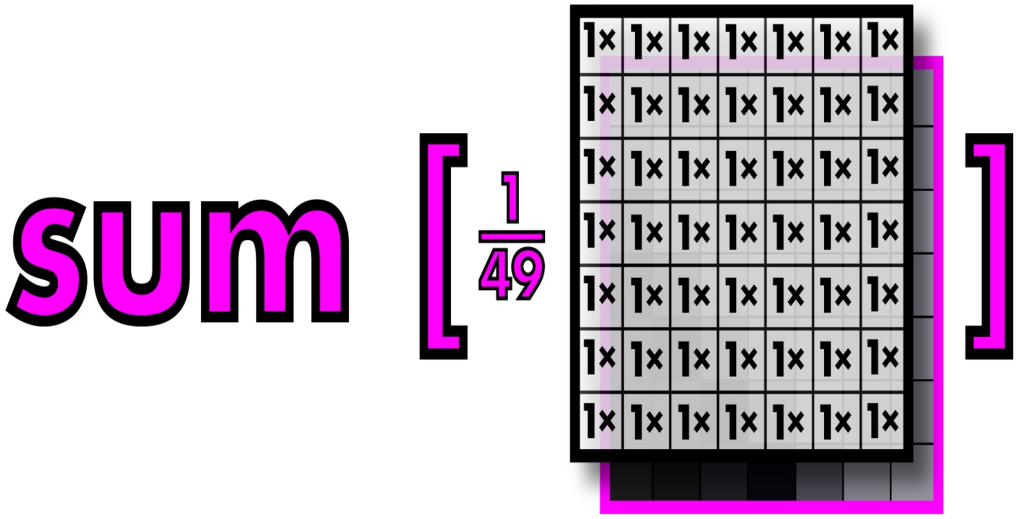


What is averaging? A weighted sum



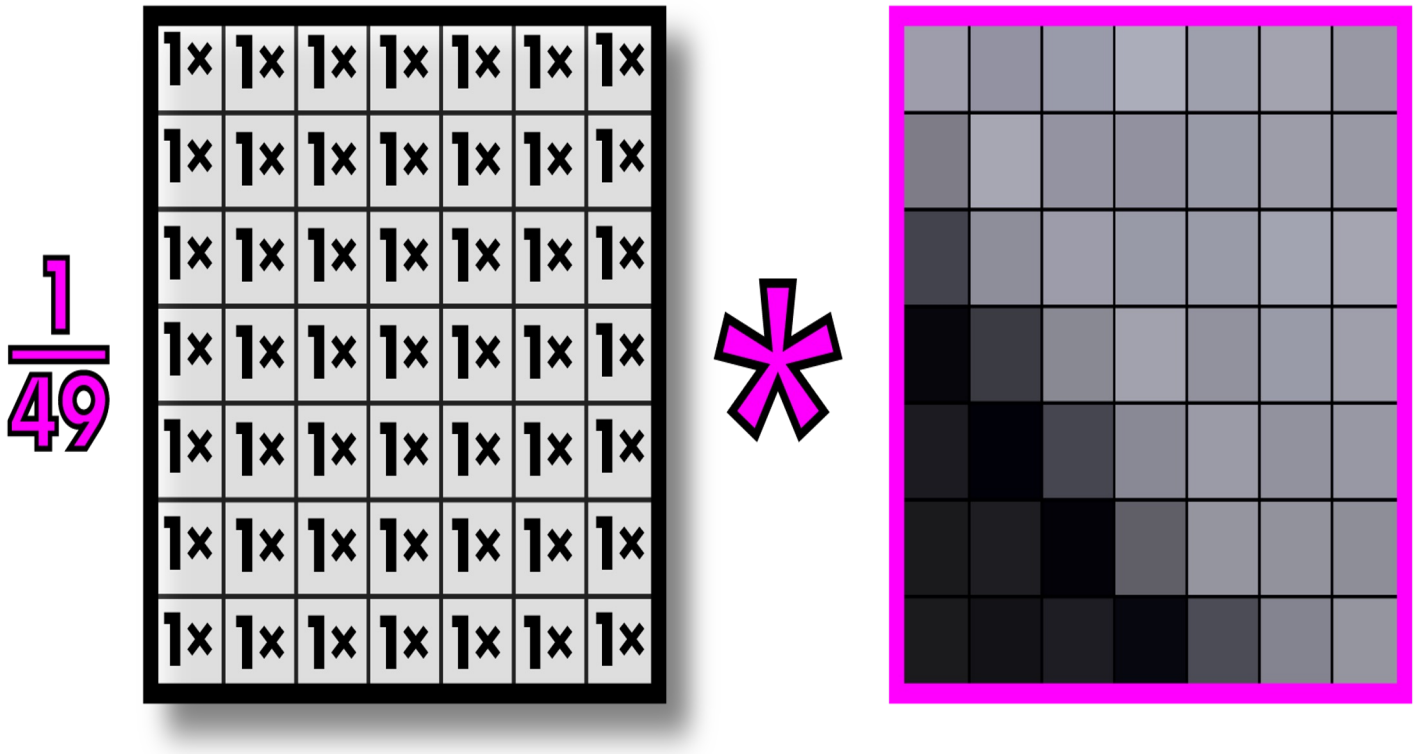


What is averaging? A weighted sum



Call this operation "*convolution*"

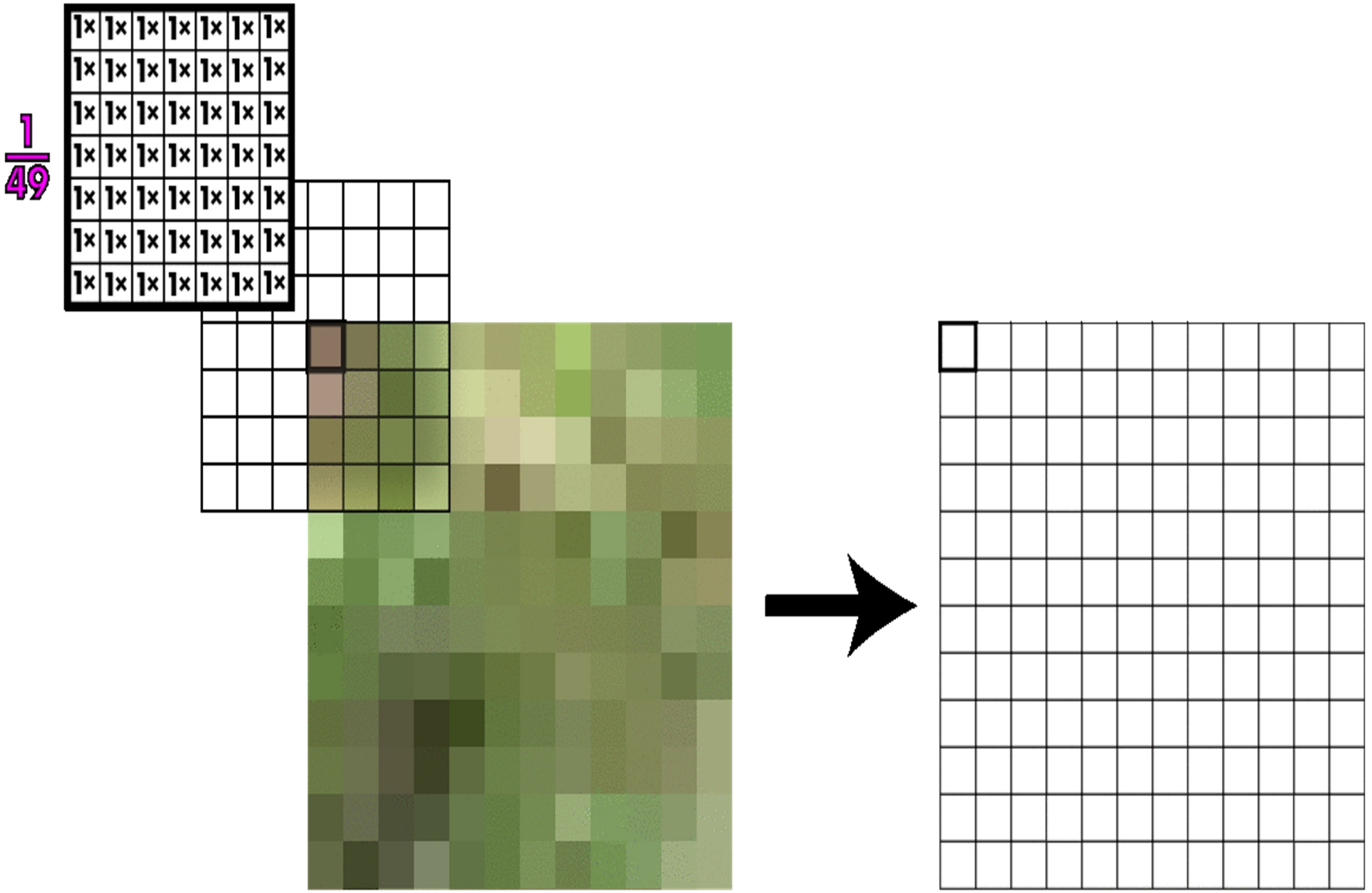
Filter or *kernel*



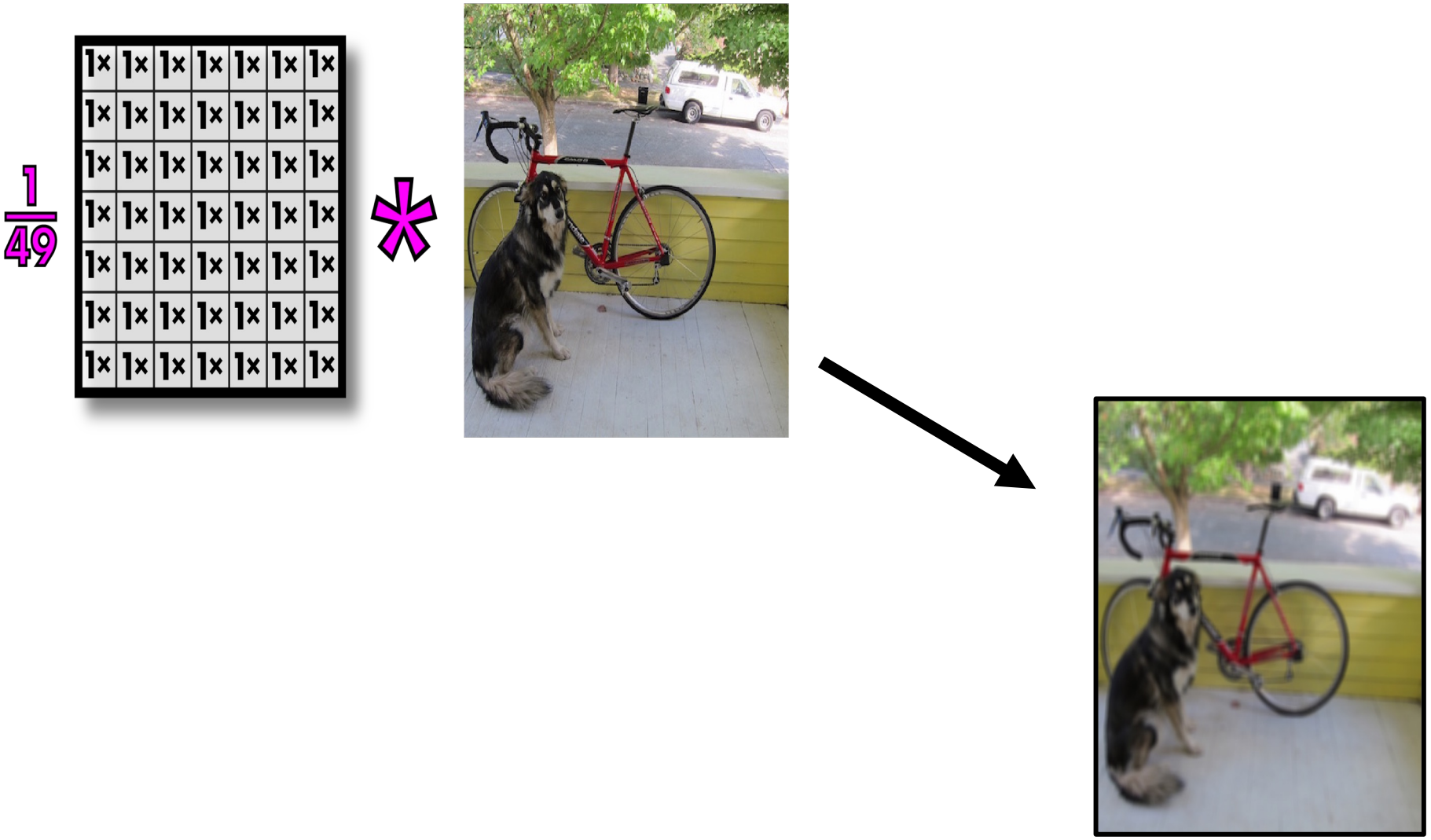
Convolutions on larger images

$$\frac{1}{49}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} * \quad 

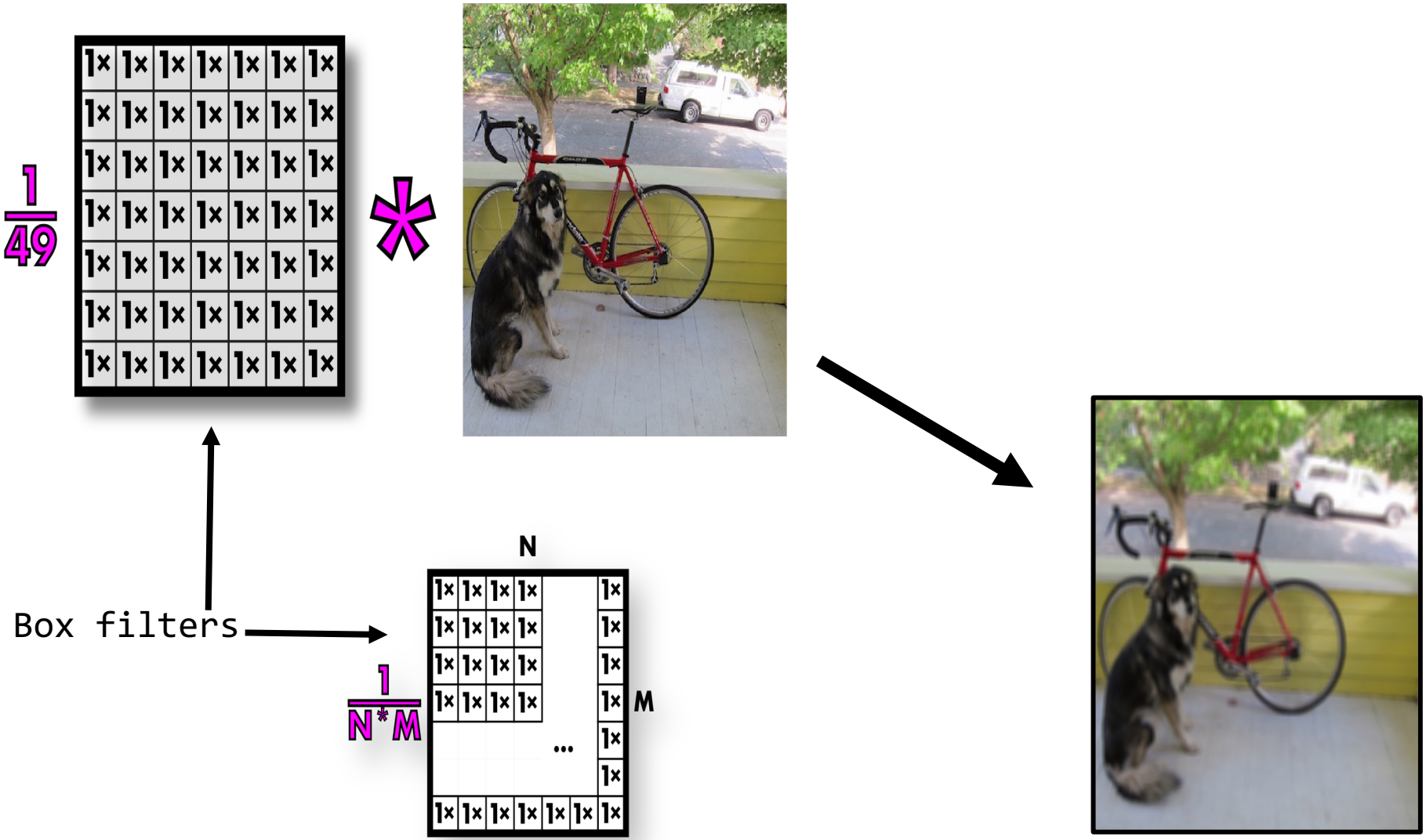
Kernel slides across image



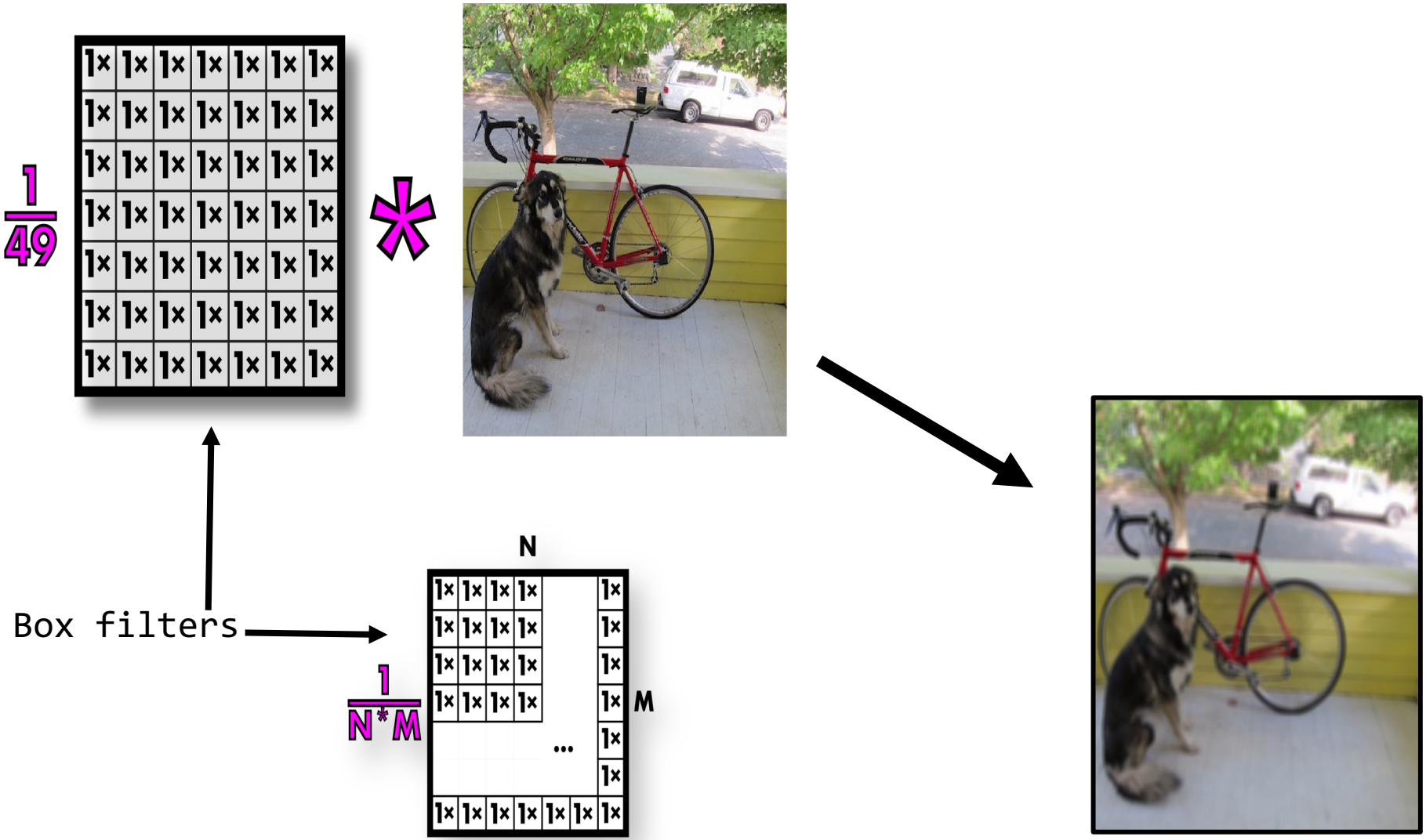
Convolutions on larger images



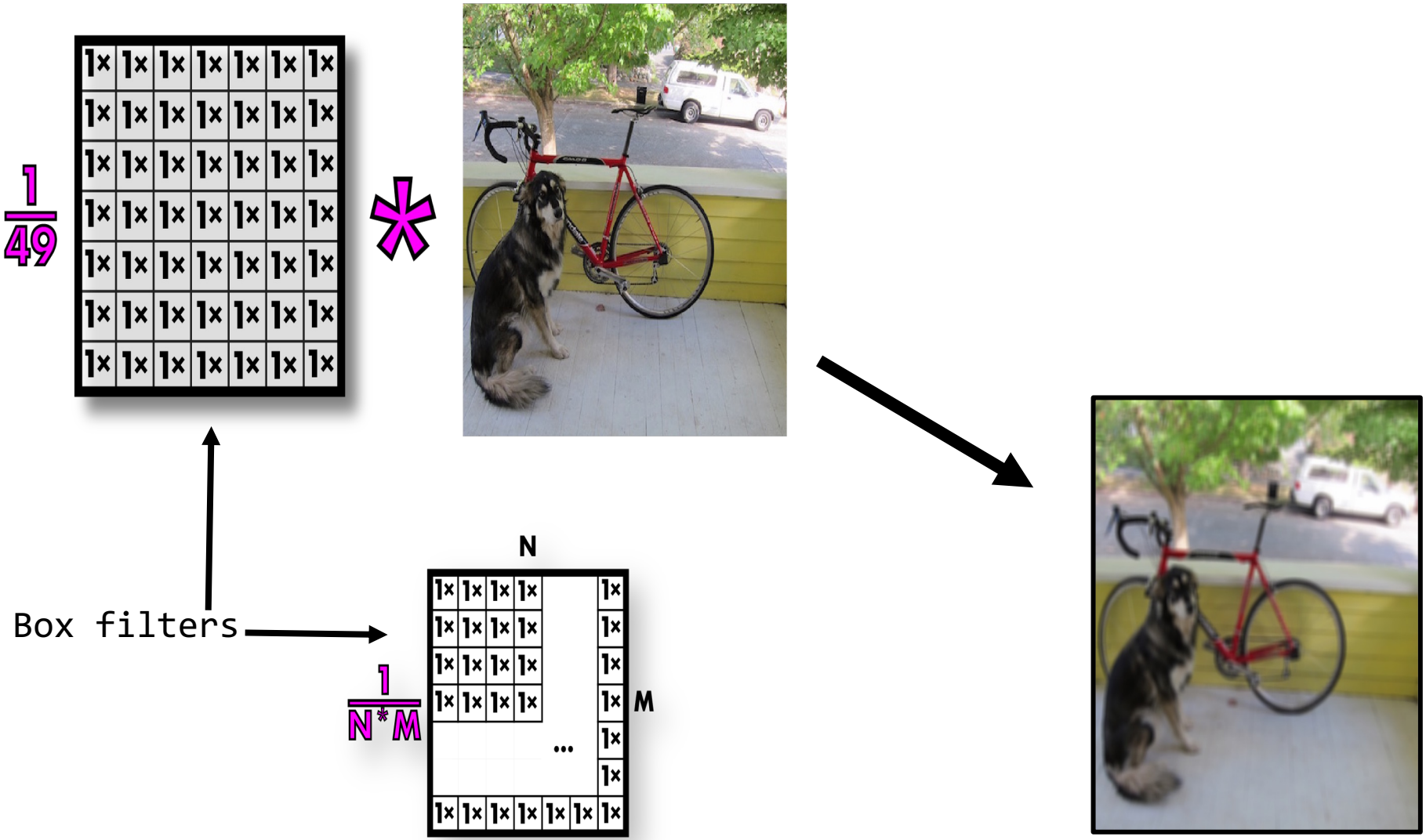
This is called box filter



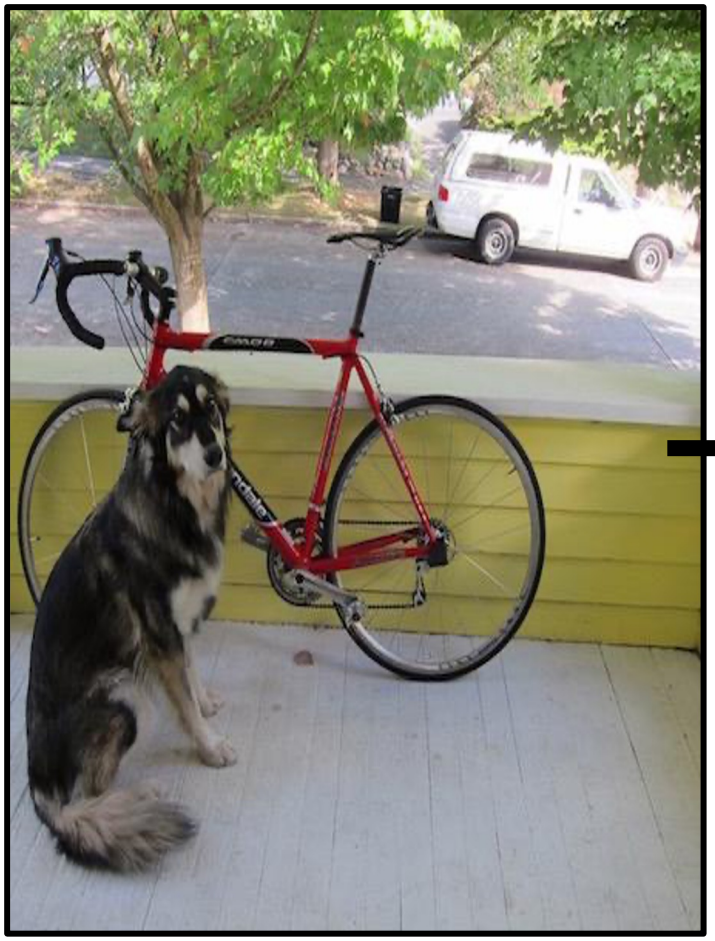
Box filters smooth image

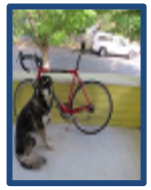


Box filters smooth image



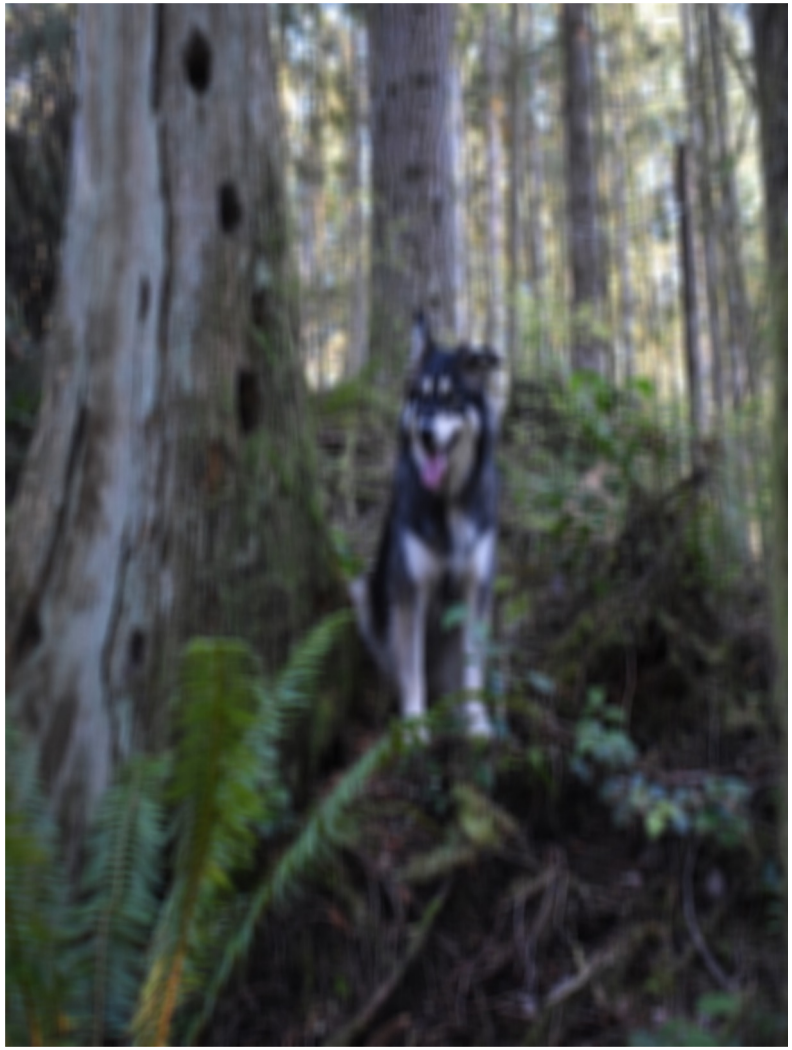
Now we resize our smoothed image



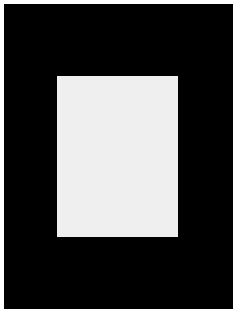


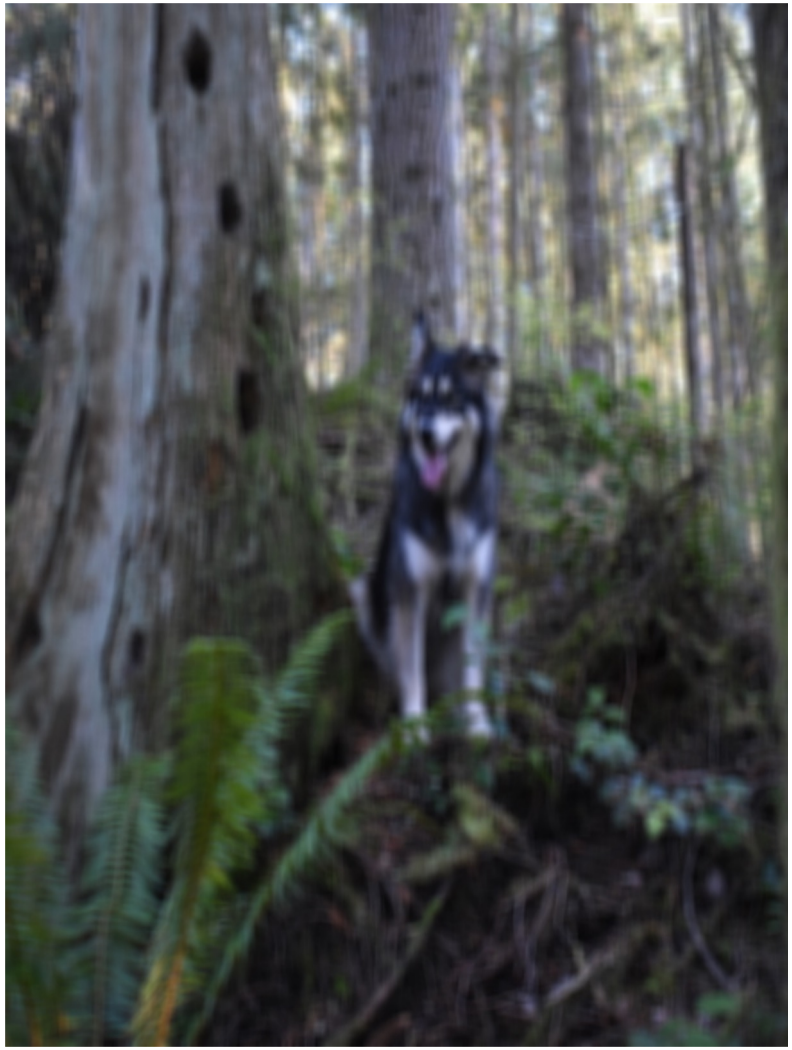
So much better!

Box filters have artifacts

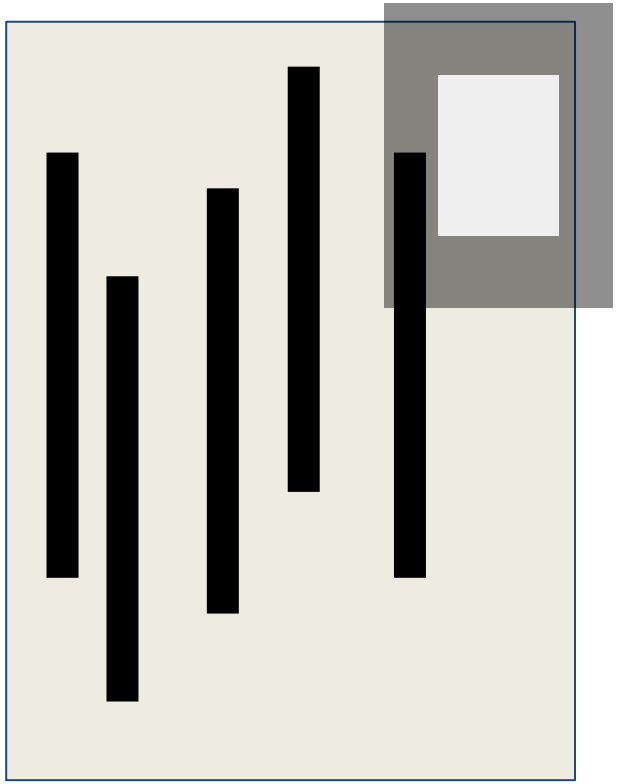


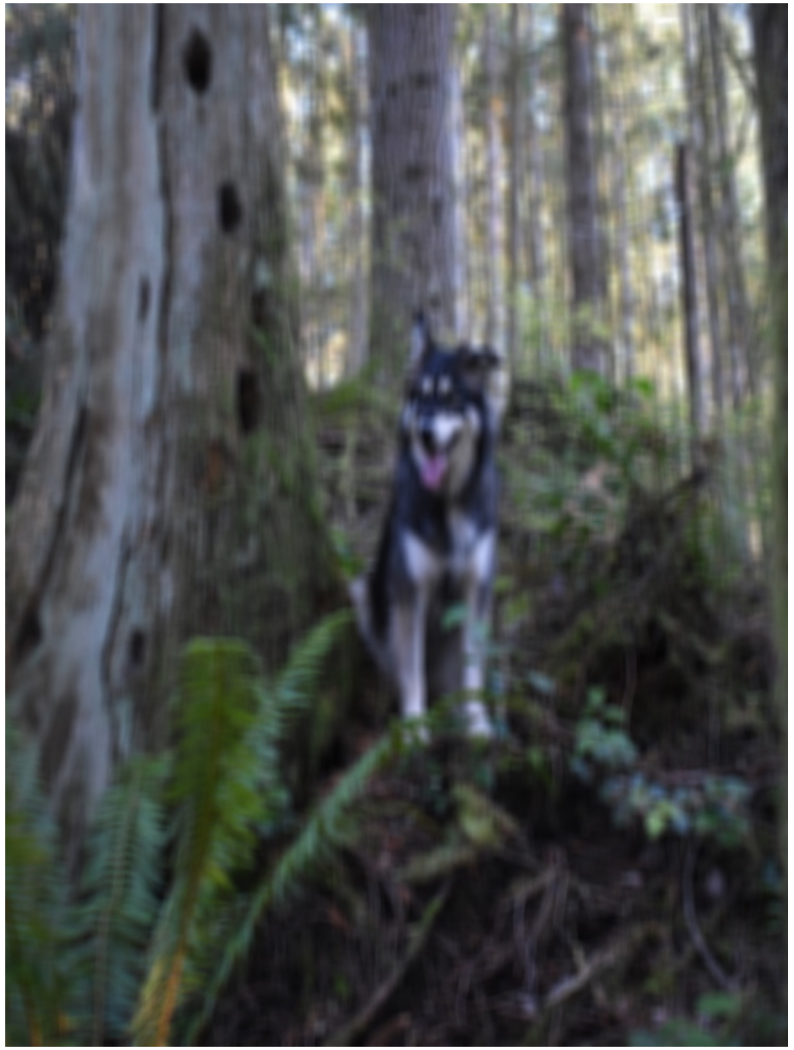
Box filters: vertical + horizontal streaking



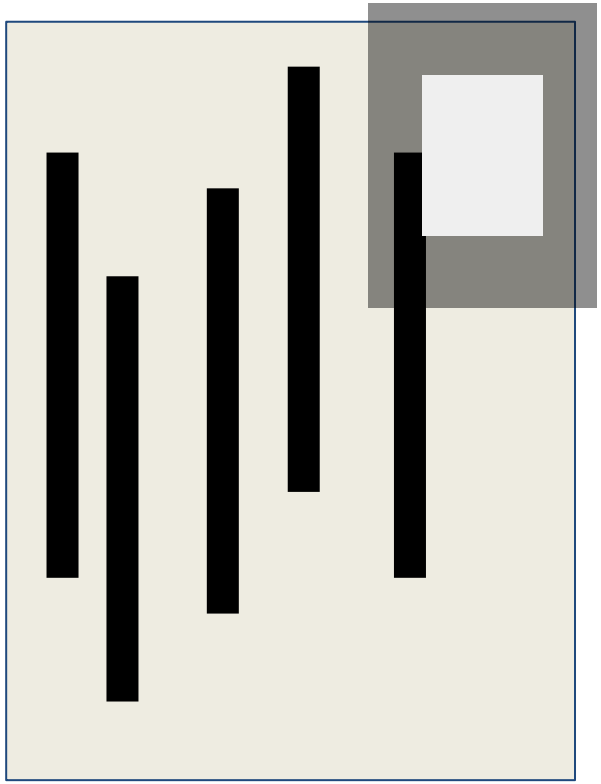


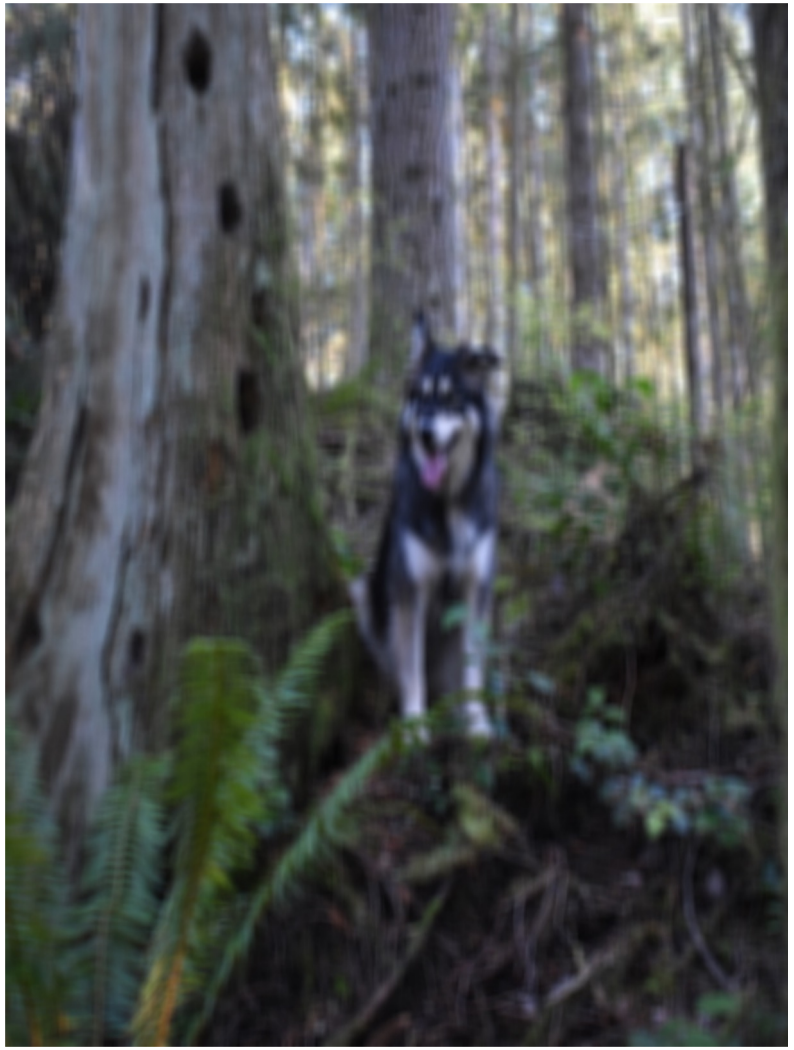
Box filters: vertical + horizontal streaking



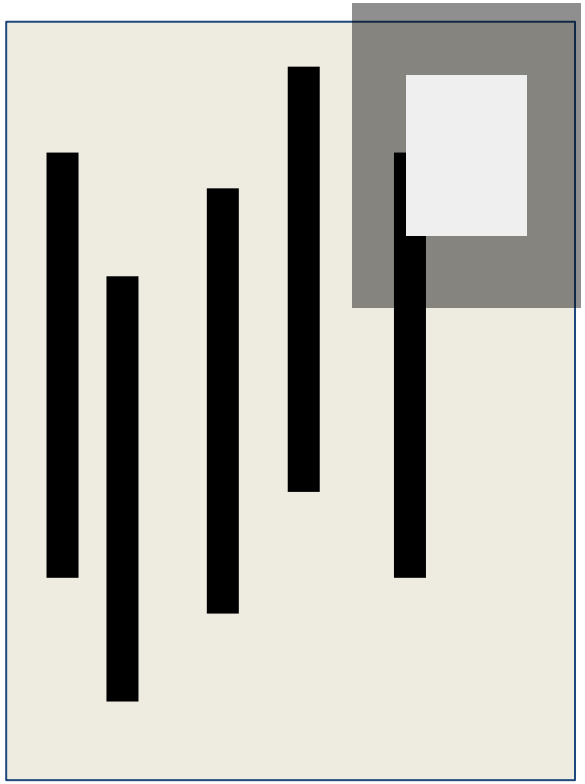


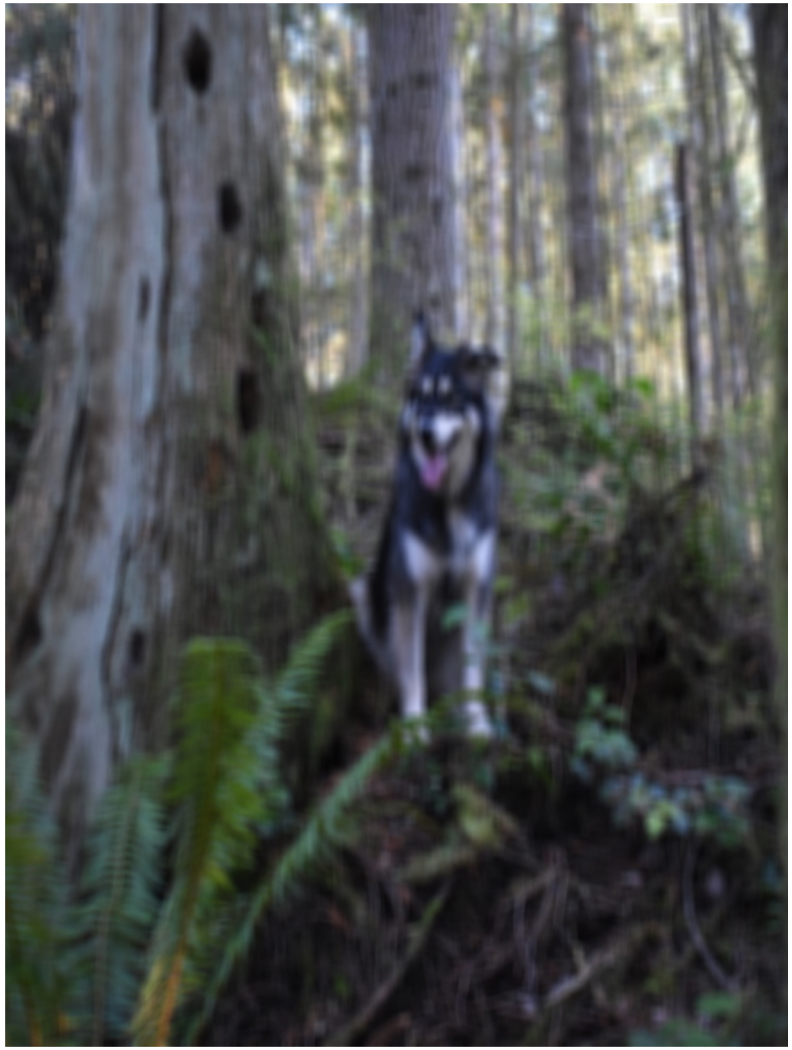
Box filters: vertical + horizontal streaking



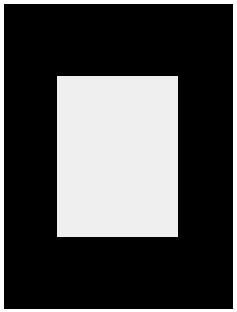


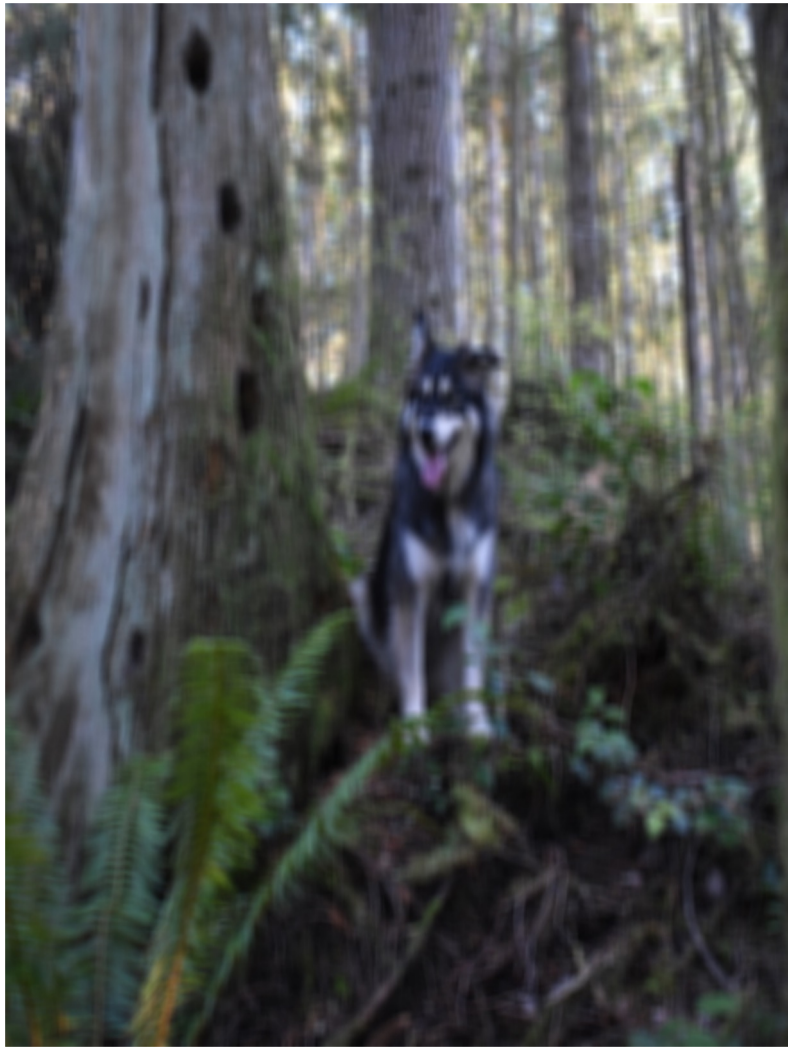
Box filters: vertical + horizontal streaking





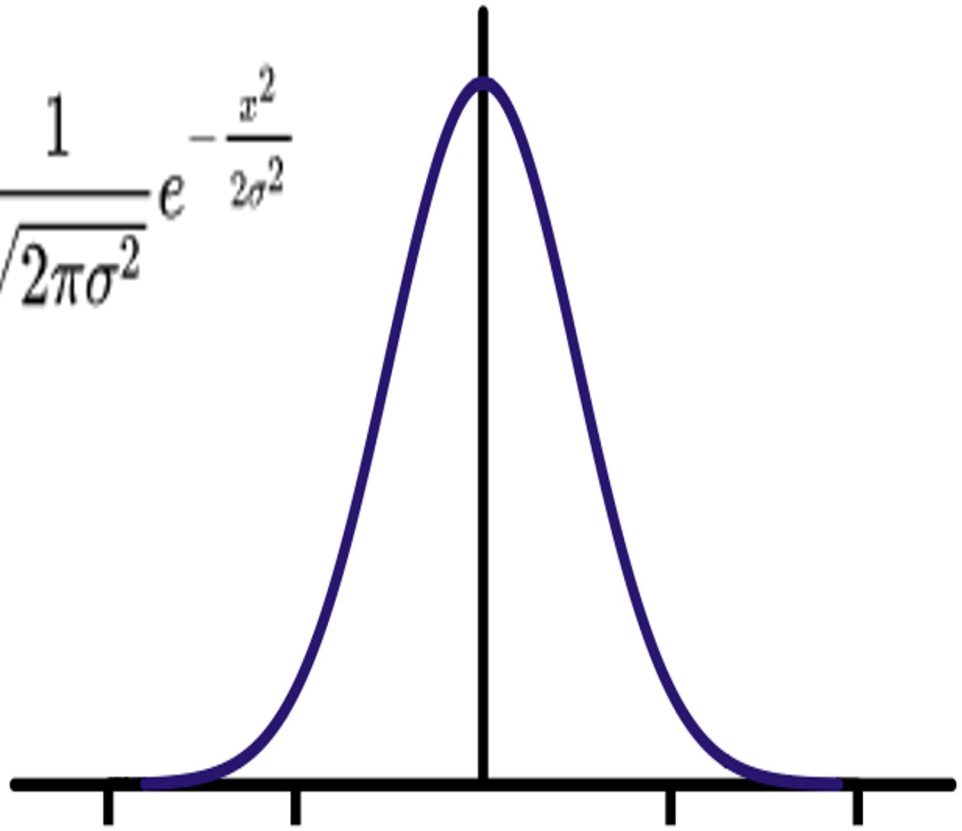
We want a smoothly weighted kernel





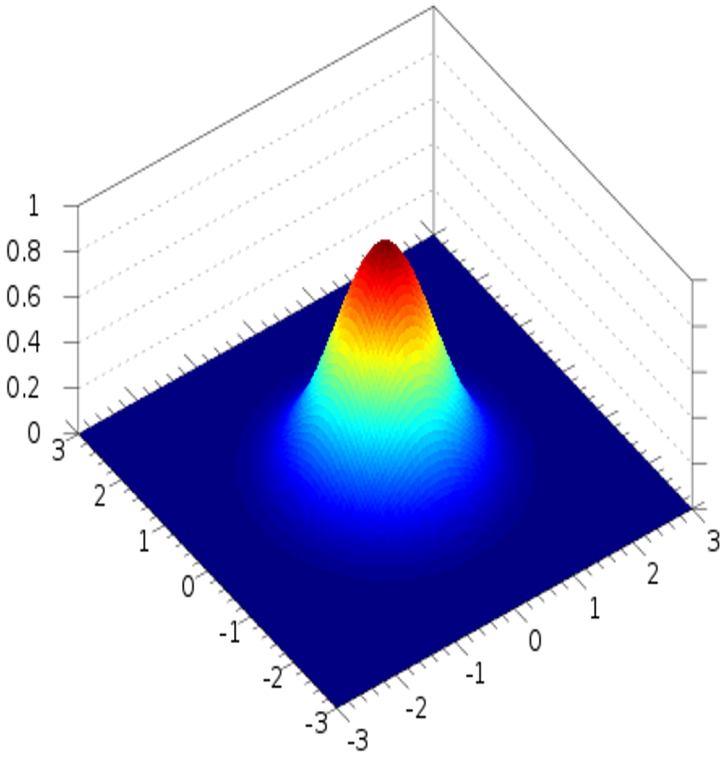
Gaussians

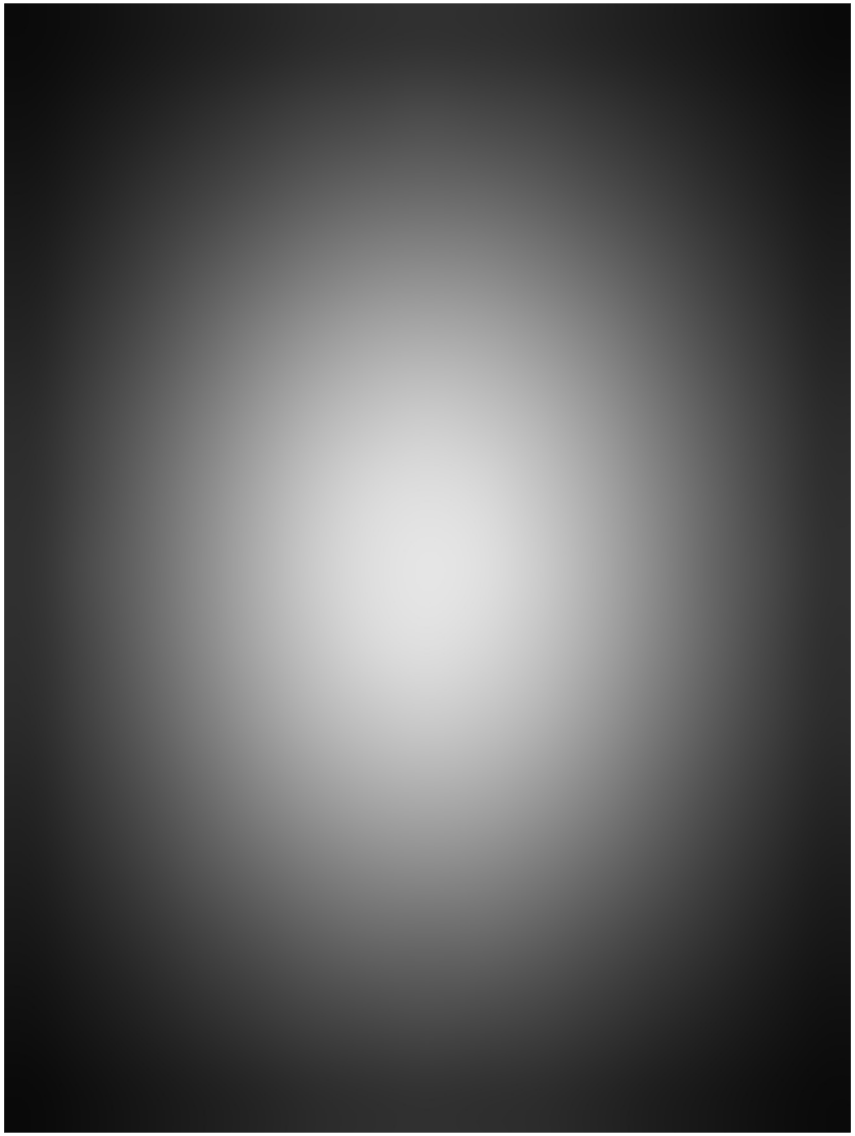
$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$



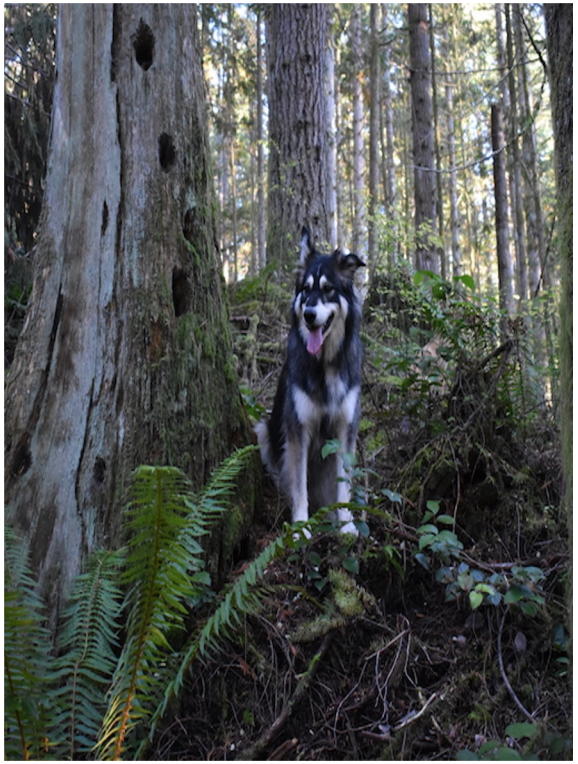
2D Gaussian

$$g(x,y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

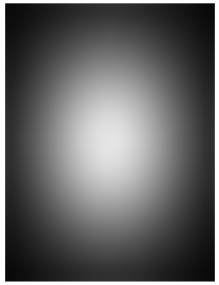




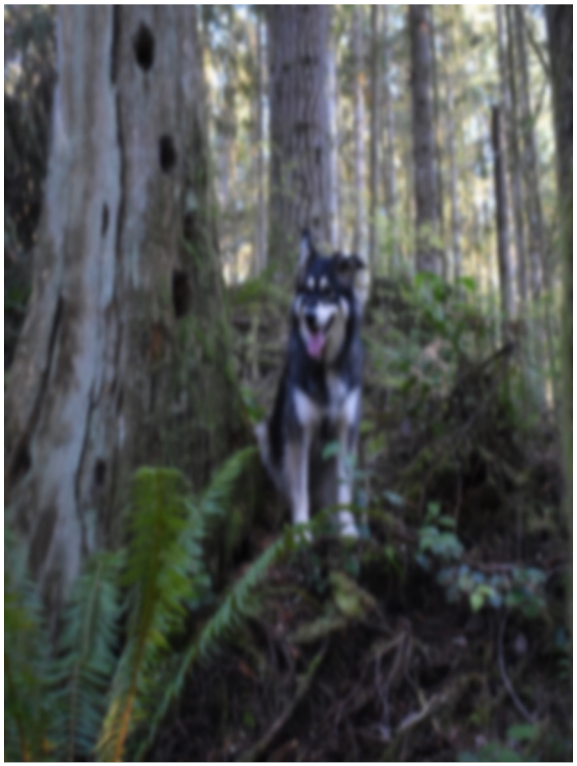
Better smoothing with Gaussians



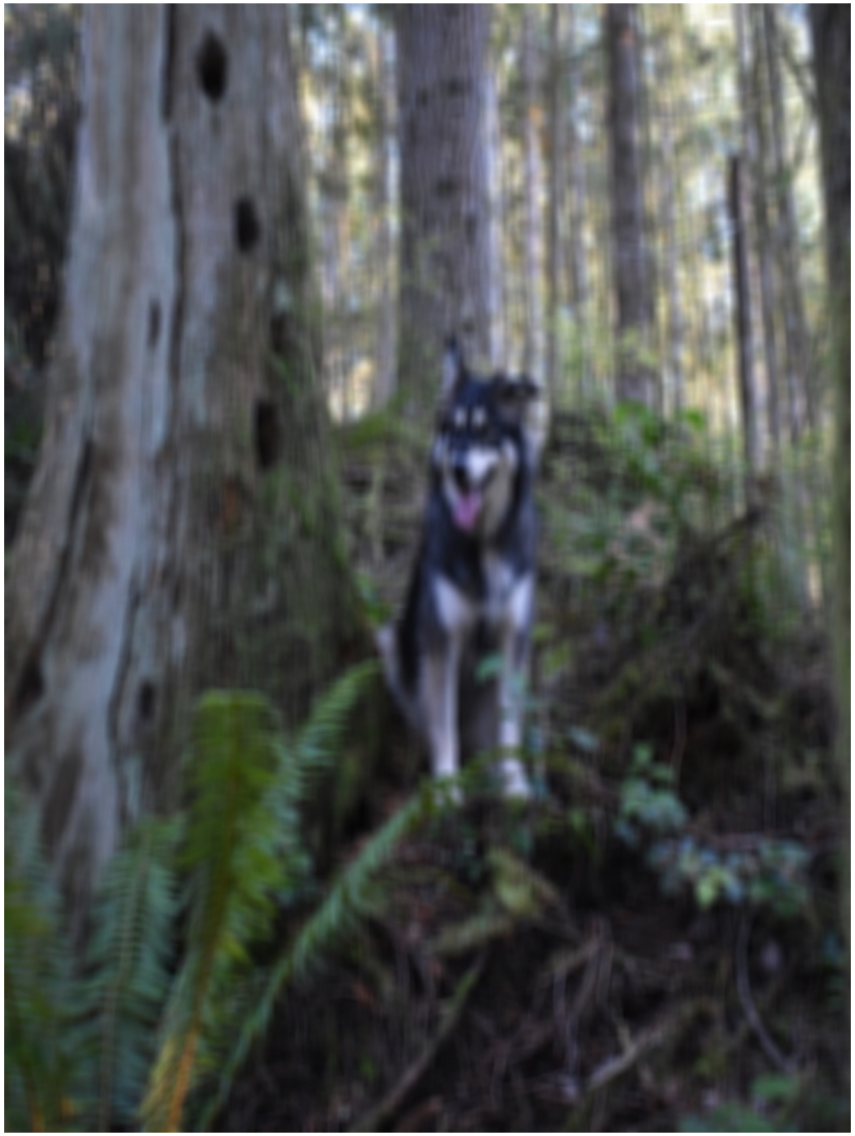
*



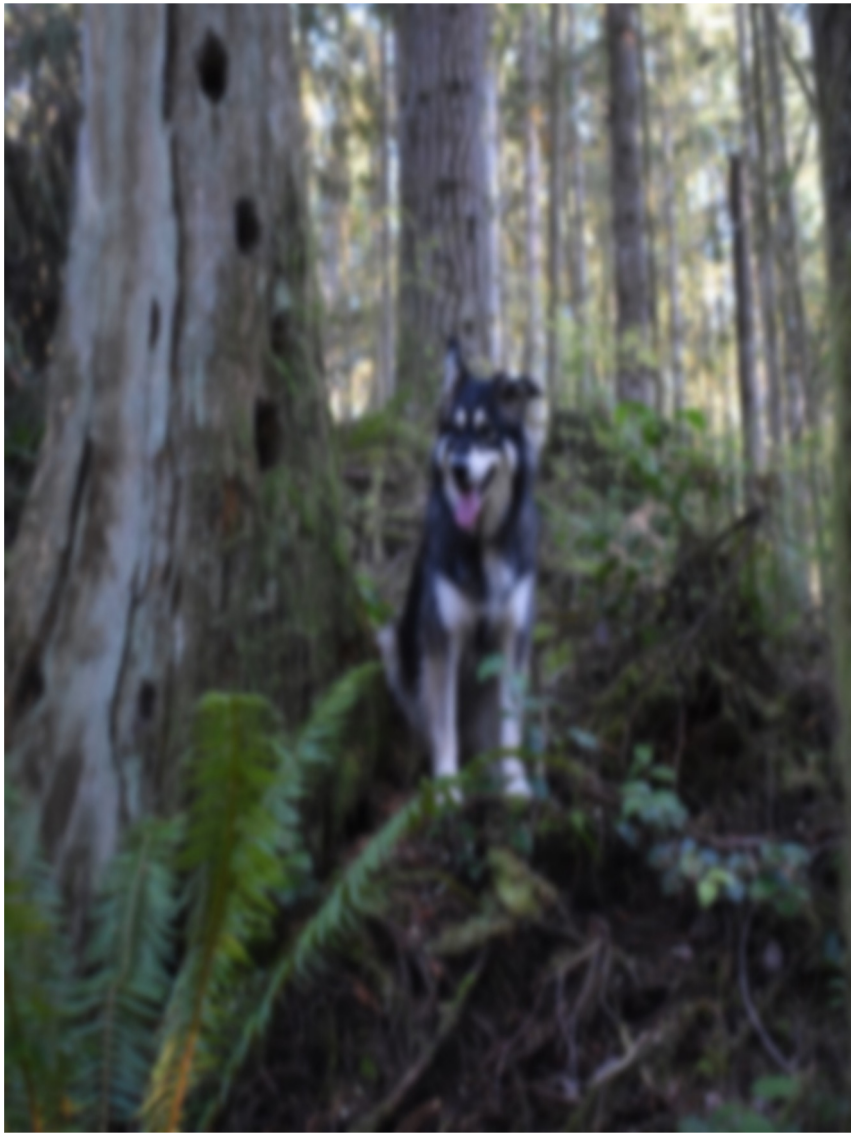
=



Better smoothing with Gaussians



Smoothed with box filter



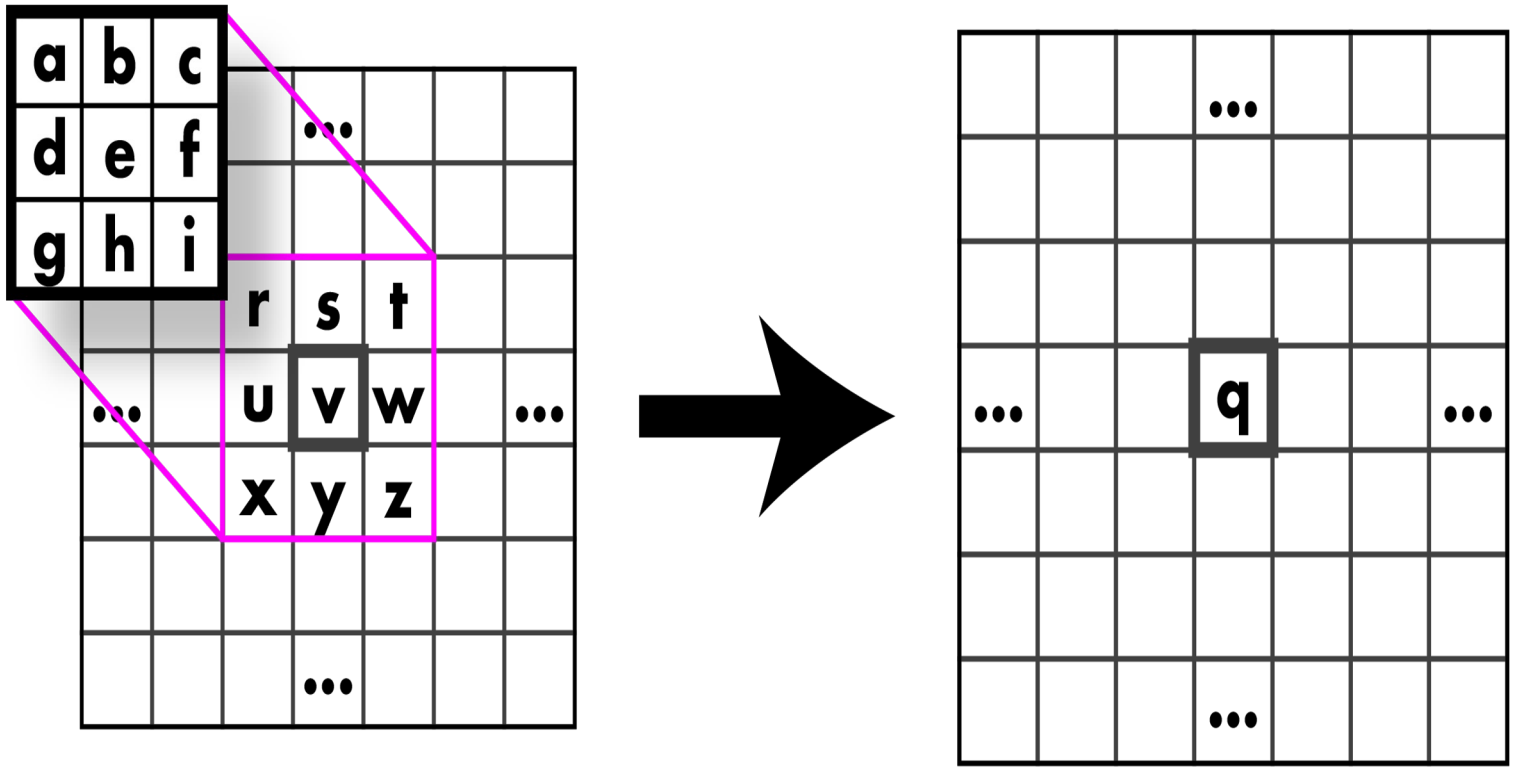
Smoothed with Gaussian filter

Better smoothing with Gaussians

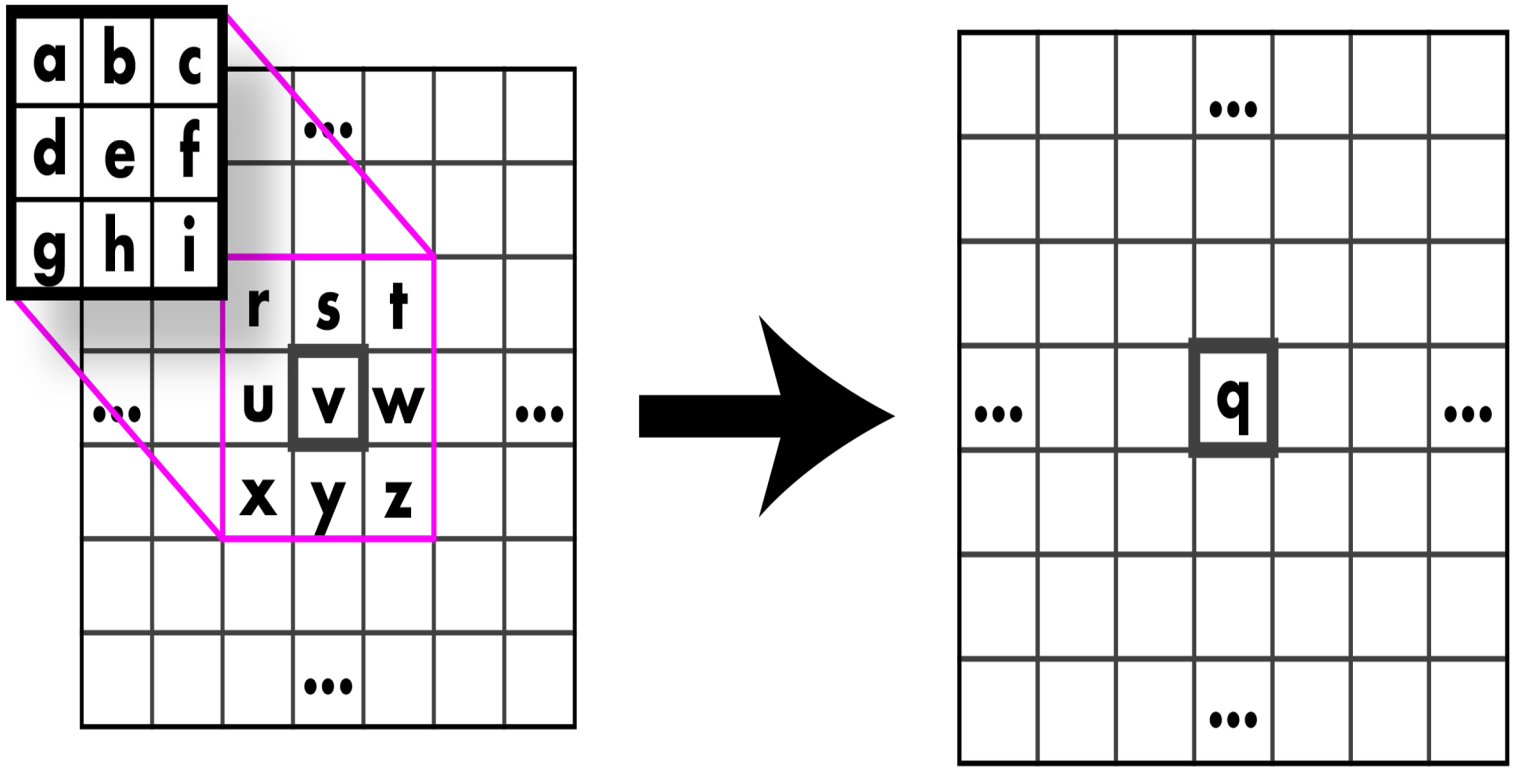
Smoothed with box filter

Smoothed with Gaussian filter

Convolution

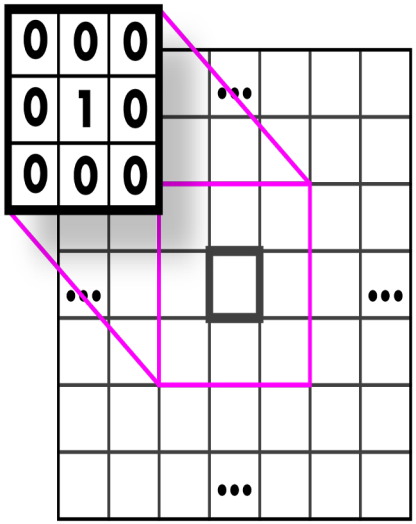


Convolution

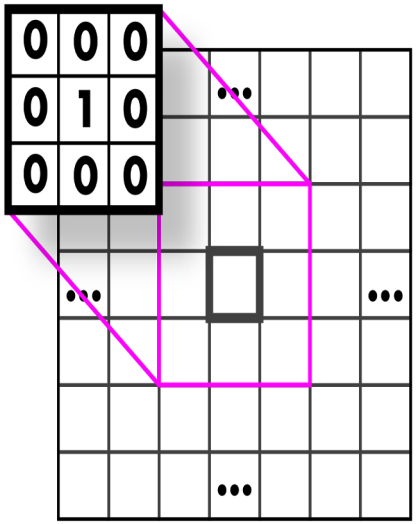


$$q = a \times r + b \times s + c \times t + d \times u + e \times v + f \times w + g \times x + h \times y + i \times z$$

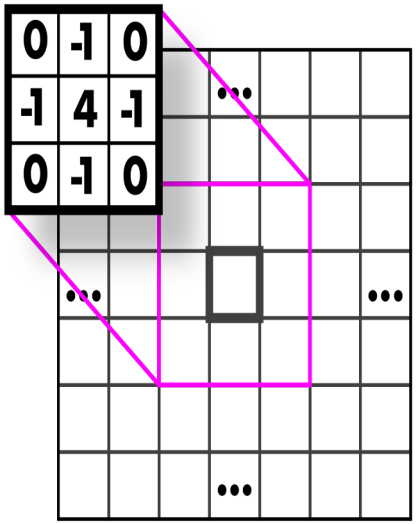
Guess that kernel!



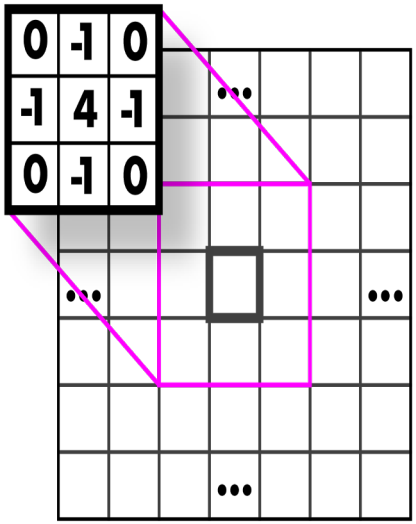
Identity Kernel: Does nothing!

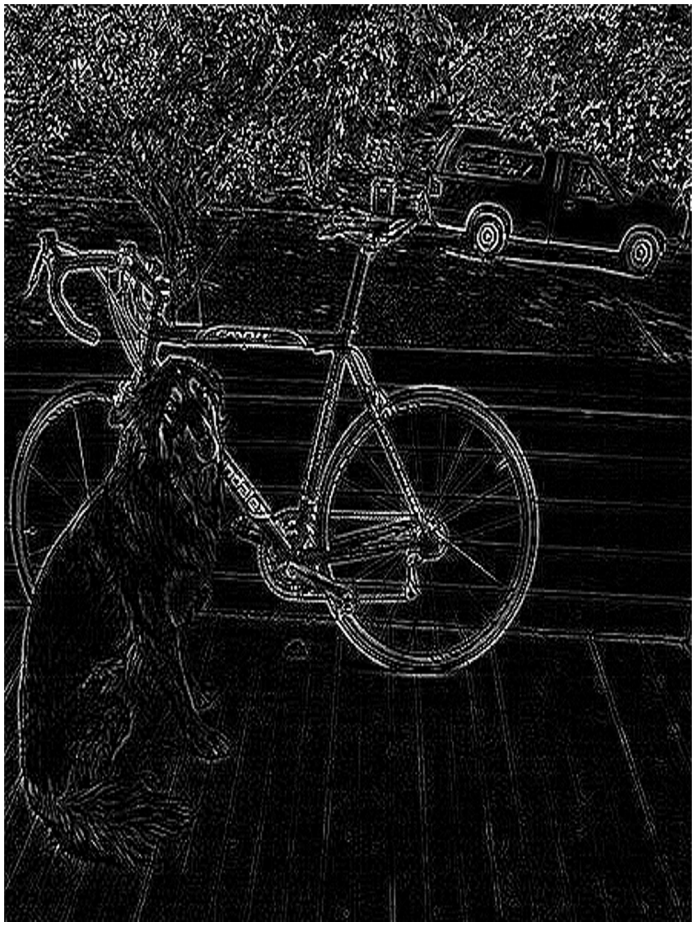


Guess that kernel!

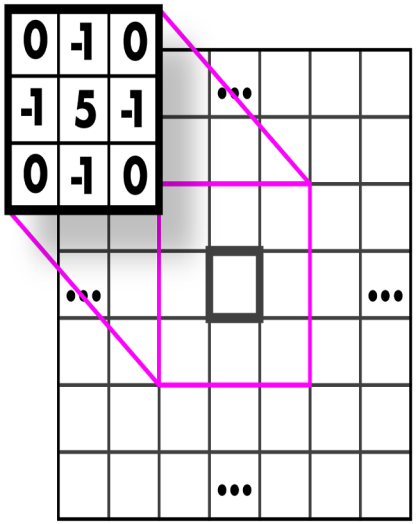


Highpass Kernel: finds edges

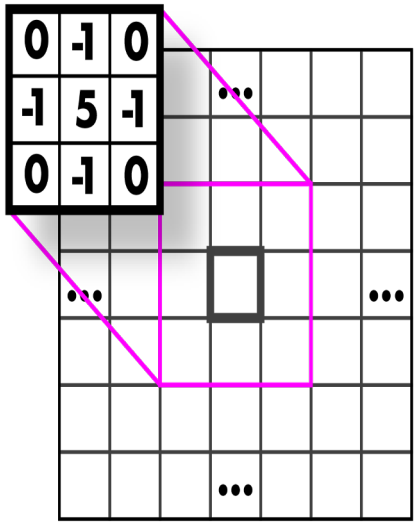


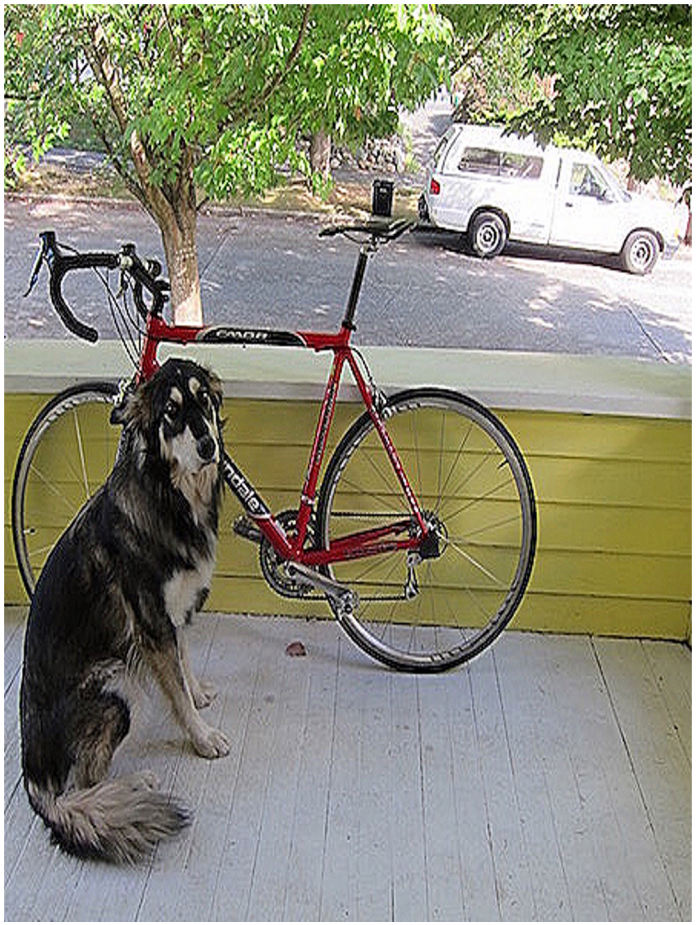


Guess that kernel!



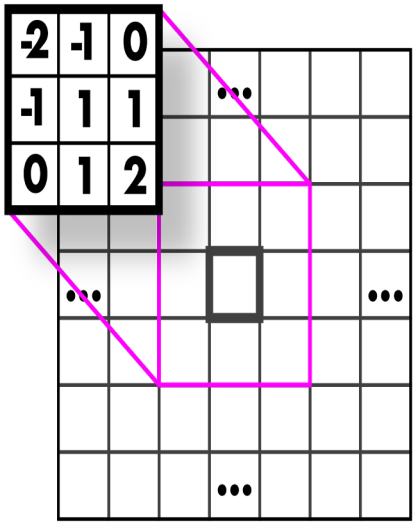
Sharpen Kernel: sharpens!



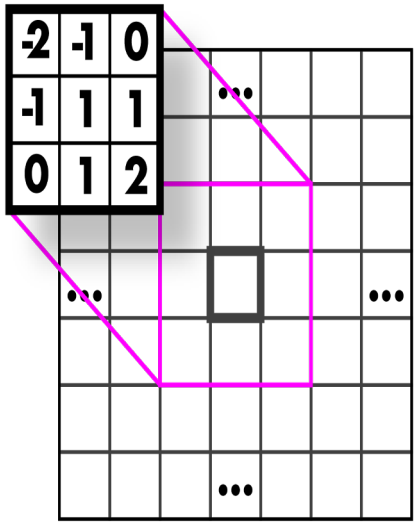


Note: sharpen = highpass + identity!

Guess that kernel!

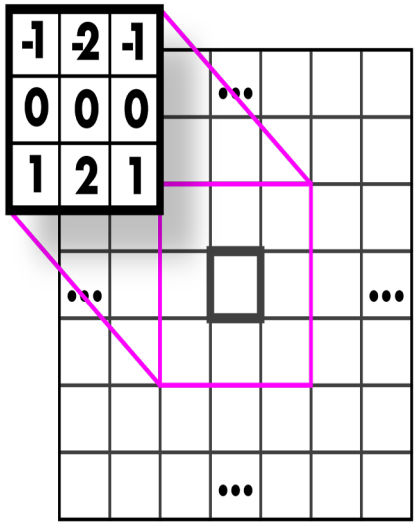


Emboss Kernel: stylin’

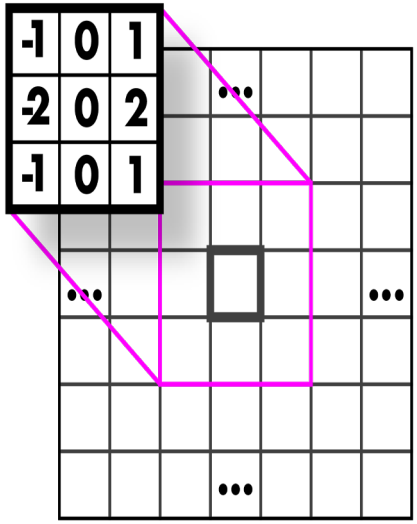


Guess those kernels!

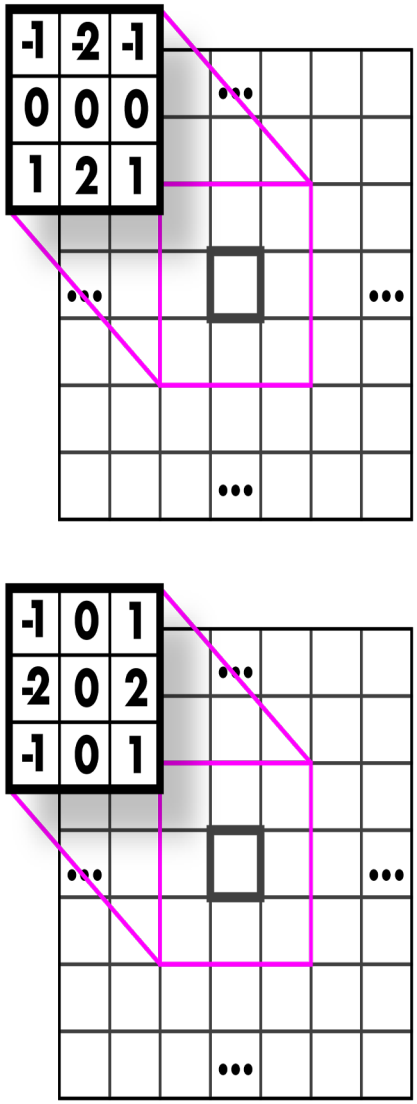
-1	-2	-1
0	0	0
1	2	1

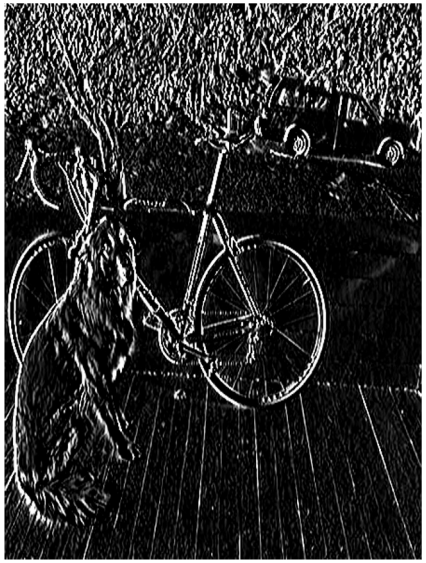


-1	0	1
-2	0	2
-1	0	1

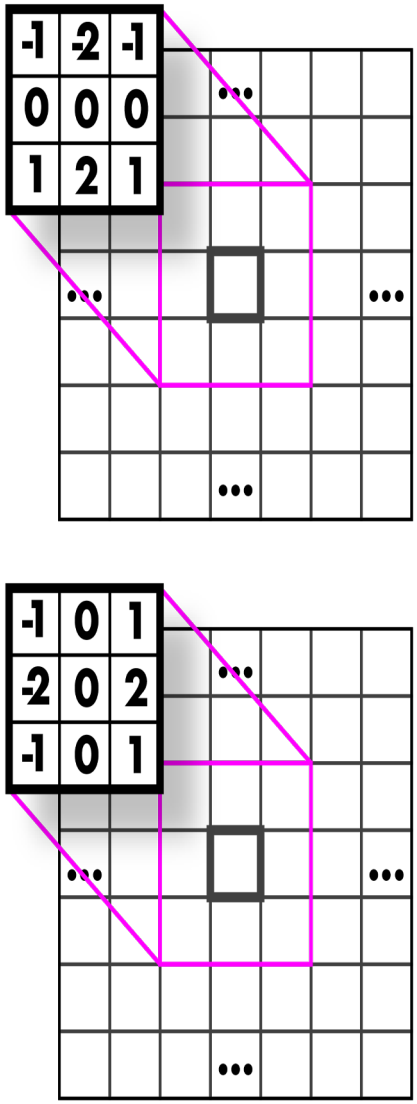


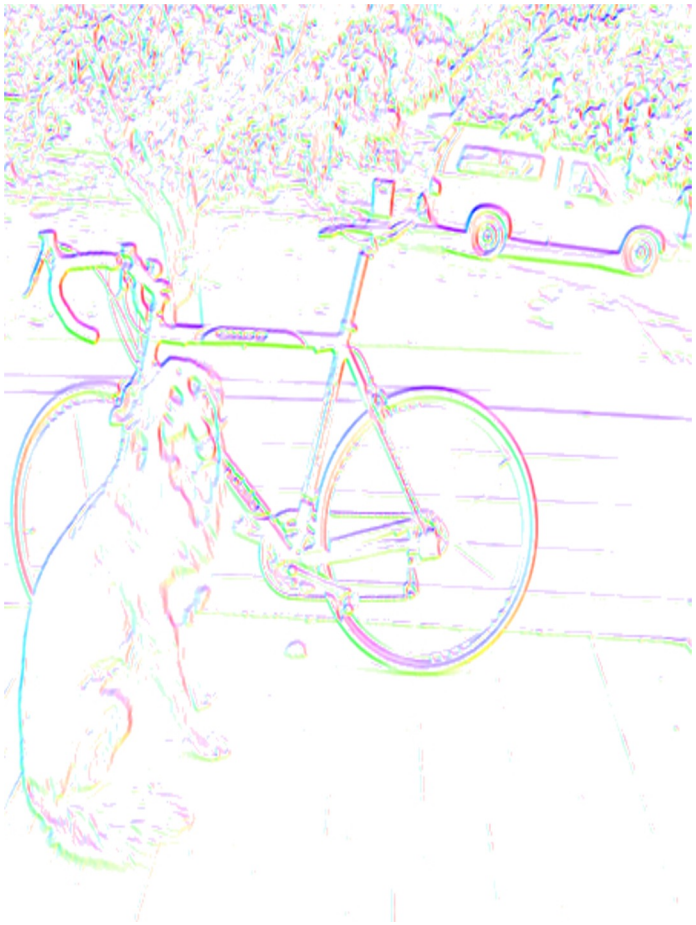
Sobel Kernels: edges and…





Sobel Kernels: edges and gradient!





Sobel Kernels: edges and gradient!