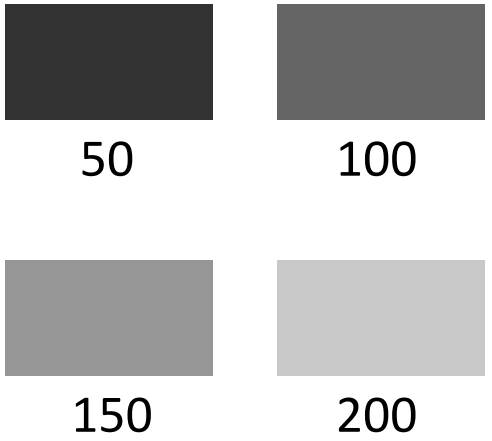
Images and Convolutions

Slide Courtesy- Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

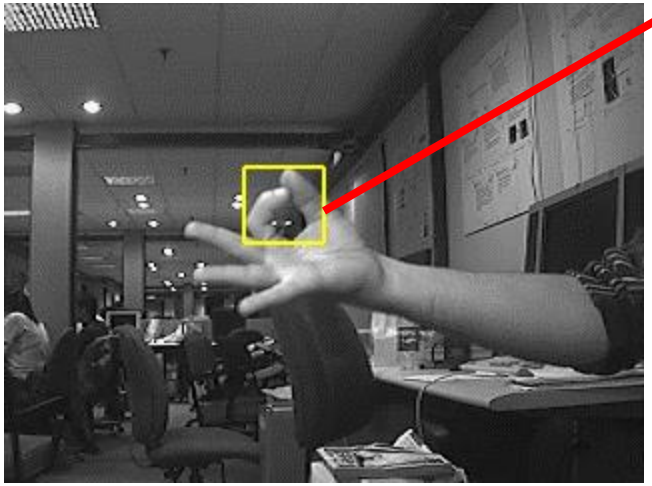
Grayscale Images

- A grayscale image is a 2D array of intensity values.
  - Dimensions: rows x columns.
- A pixel is an image location, specified by two coordinates: row and column.
- The value of every pixel is an 8-bit unsigned integer (from 0 to 255) specifying how “white” that pixel is.
  - 0 is the darkest black.
  - 255 is the brightest white.
  - Here are the “colors” (shades of gray, really) corresponding to some other values:



Grayscale Images

- To see the pixelated nature of images, you can zoom in on a small part of an image.
  - Then, the individual pixel squares become noticeable.



Color Images

- A color image is a 3D array of intensity values.
  - Dimensions: rows x columns x 3.
- The color of every pixel is an **RGB value**:
  - three 8-bit unsigned integers (from 0 to 255) specifying how “red”, “green”, and “blue” that pixel is.
  - This is why we often refer to color images as RGB images.
- Here are some examples of colors, and the RGB values they correspond to:

50, 50, 50

200, 200, 200

255, 0, 0

0, 255, 0

0, 0, 255

200, 200, 100

204,193,218

255, 192, 0

Pixelated Nature of Color Images

- Again, we can zoom in on a small part of the image to see the individual pixel squares.

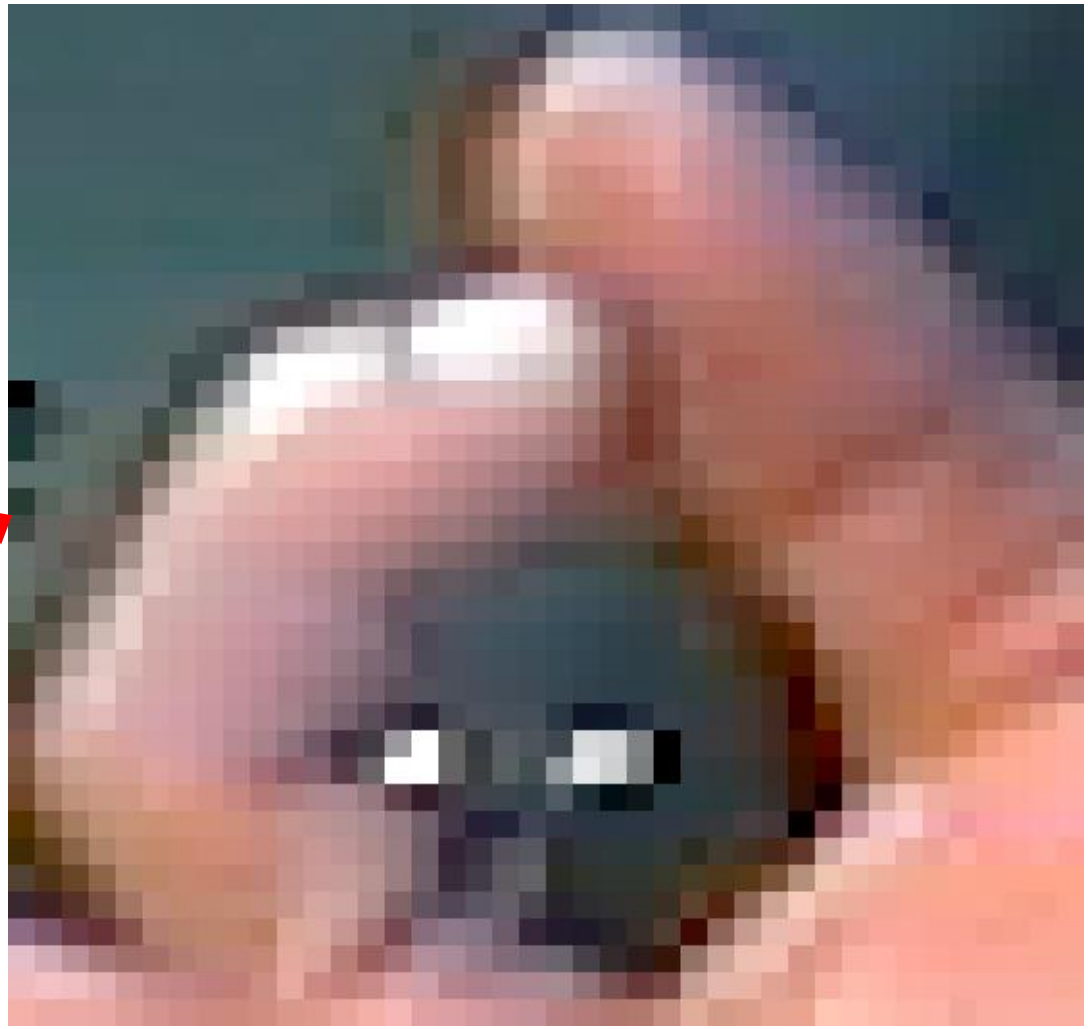


Image Representations

- Pixel values are typically stored using 8-bit unsigned integers, whose values are between 0 and 255.
  - 1 value per pixel for grayscale images.
  - 3 values per pixel for RGB images.
- However, when working with images, we will encounter and use other representations.
  - Oftentimes image values are converted to floating point numbers in the [0,1] range, to make them appropriate inputs for neural networks.
  - Sometimes image values can be 16-bit integers, if they are produced from cameras with the appropriate “dynamic range”.
  - Depth cameras often produce RGBD images, where the fourth value specifies the depth of the pixel.
  - In graphics, a fourth value is used to specify the transparency of each pixel.

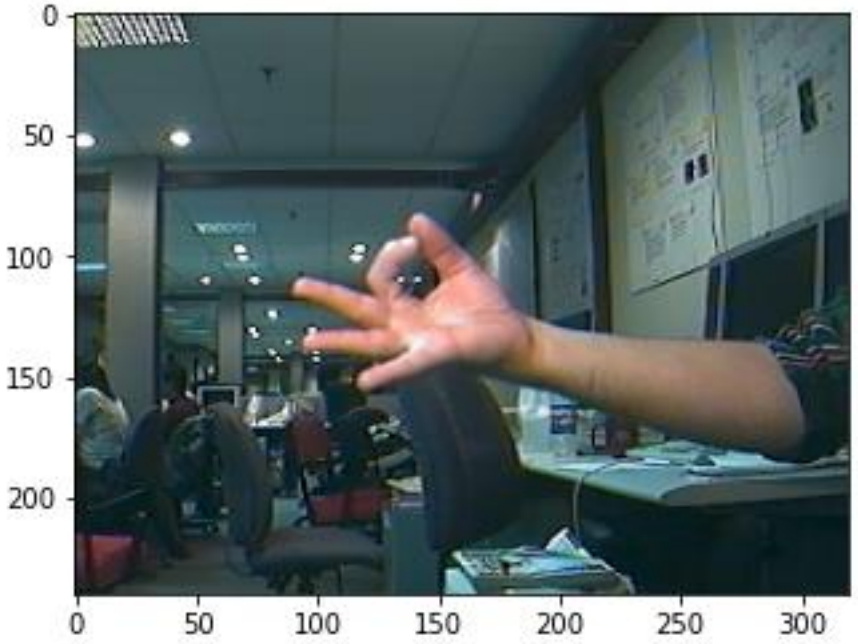
Images in Python

```
import numpy as np
import matplotlib.pyplot as plt

img = plt.imread('hand1_gray.png')
plt.imshow(img)
```

- There are many Python libraries that facilitate various aspects of working with images.
  - OpenCV, PIL, matplotlib, …

Images in Python

```
import numpy as np
import matplotlib.pyplot as plt

img = plt.imread('hand1_color.png')
plt.imshow(img)
```



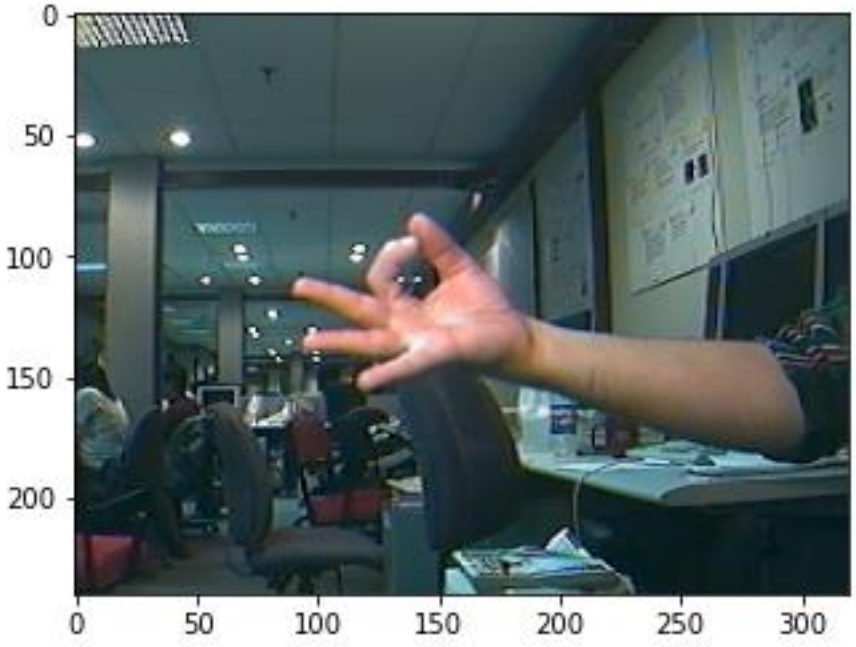
- If you execute this code:
  - imread() reads the image from the file, and returns a numpy array.
  - imshow() displays the image as shown on the right.
  - Note that (0,0) is the top left corner.

Images in Python

```
print("image dimensions:", img.shape)
mn = np.min(img)
mx = np.max(img)
print("range: [%.4f, %.4f]" % (mn, mx))
print("data type:", img.dtype)
```

Output:

```
image dimensions: (240, 320, 4)
range: [0.0000, 1.0000]
data type: float32
```



- Four values per pixel: (red, green, blue, transparency).
- Values are floats between 0 and 1.
- To avoid bugs, always verify the format of the images you use.

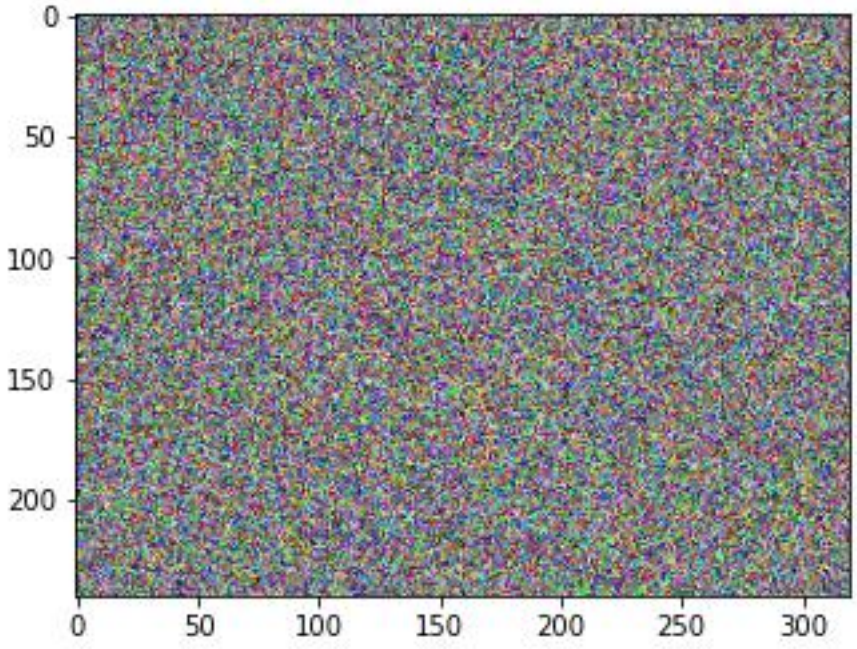
Image File Formats

- There many file formats for image files.
- The most common:
  - JPEG (extension .jpg)
  - GIF (extension .gif)
  - PNG (extension .png)
- JPEG and GIF use **lossy compression**: when saving an image in those formats, some pixel values are changed, in order to obtain a smaller file size.
  - When you save an image and you load it back, the values are a bit different.
- PNG uses **lossless compression:** it still tries to reduce the file size, if possible, but it will preserve the original pixel values.
  - When you save an image and you load it back, the values are the same.

Creating a Random Image

```
ri = np.random.randint(0, 256, (240,320,3), dtype='uint8')
plt.imshow(ri)
```

- This code creates a random RGB image, to use for testing different image formats.
  - Size: 240 rows, 320 columns.
- Note that we specify uint8 as the data type.
  - ri contains unsigned 8-bit integers between 0 and 255.



Saving as JPEG, and Reading Again

```
plt.imsave("random.jpg", ri)
r2 = plt.imread("random.jpg")
mn = np.min(r2)
mx = np.max(r2)
print("dtype:", r2.dtype, ". shape:", r2.shape)
print("min value = %.4f, max_value = %.4f" % (mn, mx))

indices = np.nonzero(ri != r2)
t = np.prod(ri.shape)
n = indices[0].shape[0]
print("jpg: %d pixels same, %d different" % (t-n, n))
print("mean difference: %.4f" % (np.mean(abs(ri-r2))))
```

Output:

```
dtype: uint8 . shape: (240, 320, 3)
min value = 0.0000, max_value = 255.0000
jpg: 1764 values same, 228636 different
mean difference: 127.0737
```

JPEG: Observations

```
plt.imsave("random.jpg", ri)
r2 = plt.imread("random.jpg")
mn = np.min(r2)
mx = np.max(r2)
print("dtype:", r2.dtype, ". shape:", r2.shape)
print("min value = %.4f, max_value = %.4f" % (mn, mx))
```

Output:

```
dtype: uint8 . shape: (240, 320, 3)
min value = 0.0000, max_value = 255.0000
jpg: 1764 values same, 228636 different
mean difference: 127.0737
```

- This code verifies the type of image that we loaded.
- Each pixel has 3 values, as expected.
- The values are of type uint8, between 0 and 255.

JPEG: Observations

```
indices = np.nonzero(ri != r2)
t = np.prod(ri.shape)
n = indices[0].shape[0]
print("jpg: %d pixels same, %d different" % (t-n, n))
print("mean difference: %.4f" % (np.mean(abs(ri-r2))))
```

Output:

```
dtype: uint8 . shape: (240, 320, 3)
min value = 0.0000, max_value = 255.0000
jpg: 1764 values same, 228636 different
mean difference: 127.0737
```

- This code checks the differences between the original image and the image we loaded from the file.
- We see that most pixel values are different.
  - The average difference is around 127, which is relatively high.
  - The randomness of the image is a very bad fit for the JPEG format.

Saving as GIF, and Reading Again

```
plt.imsave("random.gif", ri)
r2 = plt.imread("random.gif")
mn = np.min(r2)
mx = np.max(r2)
print("dtype:", r2.dtype, ". shape:", r2.shape)
print("min value = %.4f, max_value = %.4f" % (mn, mx))

r2b = r2[:,:,0:3]
indices = np.nonzero(ri != r2b)
t = np.prod(ri.shape)
n = indices[0].shape[0]
print("gif: %d pixels same, %d different" % (t-n, n))
print("mean difference: %.4f" % (np.mean(abs(ri-r2[:,:,0:3]))))
```

Output:

```
dtype: uint8 . shape: (240, 320, 4)
min value = 6.0000, max_value = 255.0000
gif: 5279 values same, 225121 different
mean difference: 122.5283
```

GIF: Observations

```
plt.imsave("random.gif", ri)
r2 = plt.imread("random.gif")
mn = np.min(r2)
mx = np.max(r2)
print("dtype:", r2.dtype, ". shape:", r2.shape)
print("min value = %.4f, max_value = %.4f" % (mn, mx))
```

Output:

```
dtype: uint8 . shape: (240, 320, 4)
min value = 6.0000, max_value = 255.0000
gif: 5279 values same, 225121 different
mean difference: 122.5283
```

- This code verifies the type of image that we loaded.
- The values are of type uint8, between 0 and 255.
- Each pixel has **four values**, in the JPEG case there were **three values**.
  - The fourth value is transparency, always equal to 255.
  - An example why it is important to doublecheck the image type.

GIF: Observations

```
r2b = r2[:,:,0:3]
indices = np.nonzero(ri != r2b)
t = np.prod(ri.shape)
n = indices[0].shape[0]
print("gif: %d pixels same, %d different" % (t-n, n))
print("mean difference: %.4f" % (np.mean(abs(ri-r2[:,:,0:3]))))
```

Output:

```
dtype: uint8 . shape: (240, 320, 4)
min value = 6.0000, max_value = 255.0000
gif: 5279 values same, 225121 different
mean difference: 122.5283
```

- This code checks the differences between the original image and the image we loaded from the file.
- (As in the JPEG case) We see that most pixel values are different.
  - The average difference is around 123, similar to the JPEG case where the average difference was around 127.

Saving as PNG, and Reading Again

```
plt.imsave("random.png", ri)
r2f = plt.imread("random.png")
mn = np.min(r2)
mx = np.max(r2)
print("dtype:", r2.dtype, ". shape:", r2.shape)
print("min value = %.4f, max_value = %.4f" % (mn, mx))

r2i = np.round(r2[:,:,0:3]*255)
indices = np.nonzero(ri != r2i)
t = np.prod(ri.shape)
n = indices[0].shape[0]
print("png: %d pixels same, %d different" % (t-n, n))
print("mean difference: %.4f" % (np.mean(abs(ri-r2i[:,:,0:3]))))
```

Output:

```
dtype: float32 . shape: (240, 320, 4)
min value = 0.0000, max_value = 1.0000
png: 230400 values same, 0 different
mean difference: 0.0000
```

PNG: Observations

```
plt.imsave("random.png", ri)
r2f = plt.imread("random.png")
mn = np.min(r2)
mx = np.max(r2)
print("dtype:", r2.dtype, ". shape:", r2.shape)
print("min value = %.4f, max_value = %.4f" % (mn, mx))
```

Output:

```
dtype: float32 . shape: (240, 320, 4)
min value = 0.0000, max_value = 1.0000
png: 230400 values same, 0 different
mean difference: 0.0000
```

- This code verifies the type of image that we loaded.
- The values are of type float32, between 0 and 1.
  - **This is different than the uint8 type we got in the JPEG and GIF cases.**
- Each pixel has **four values**. The fourth value is transparency, always equal to 1.0.
  - This matches the GIF case, and **but not the JPEG case**.

PNG: Observations

```
r2i = np.round(r2[:,:,0:3]*255)
indices = np.nonzero(ri != r2i)
t = np.prod(ri.shape)
n = indices[0].shape[0]
print("png: %d pixels same, %d different" % (t-n, n))
print("mean difference: %.4f" % (np.mean(abs(ri-r2i[:,:,0:3]))))
```

Output:

```
dtype: float32 . shape: (240, 320, 4)
min value = 0.0000, max_value = 1.0000
png: 230400 values same, 0 different
mean difference: 0.0000
```

- (Unlike the JPEG and GIF cases) Since the loaded image r2 has float32 values, we convert them to integers before comparing with the original image ri.
- (Unlike the JPEG and GIF cases) All pixel values are the same.

Comments on Image Types

- Your code can get images from many different sources.
- The specific image representation can depend on multiple things:
- File format.
- Functions and APIs that you use to read the image. Different APIs may produce different image types when reading the same file.
  - In Python's matplotlib package, **imread('random.png')** produces an image of float32 values, with four values per pixel.
  - In Matlab, **imread('random.png')** produces an image of uint8 values, with three values per pixel.
- Possible parameters that these functions take as arguments can also influence the image representation.

Comments on Image Types

- Incorrect assumptions about the image representation are a common source of bugs.
- Your code could be doing operations on uint8 numbers instead of float numbers, getting wrong results due to overflow.
- Your code may assume that the values are between 0 and 1, but they are between 0 and 255 (leading to numerical problems during training).

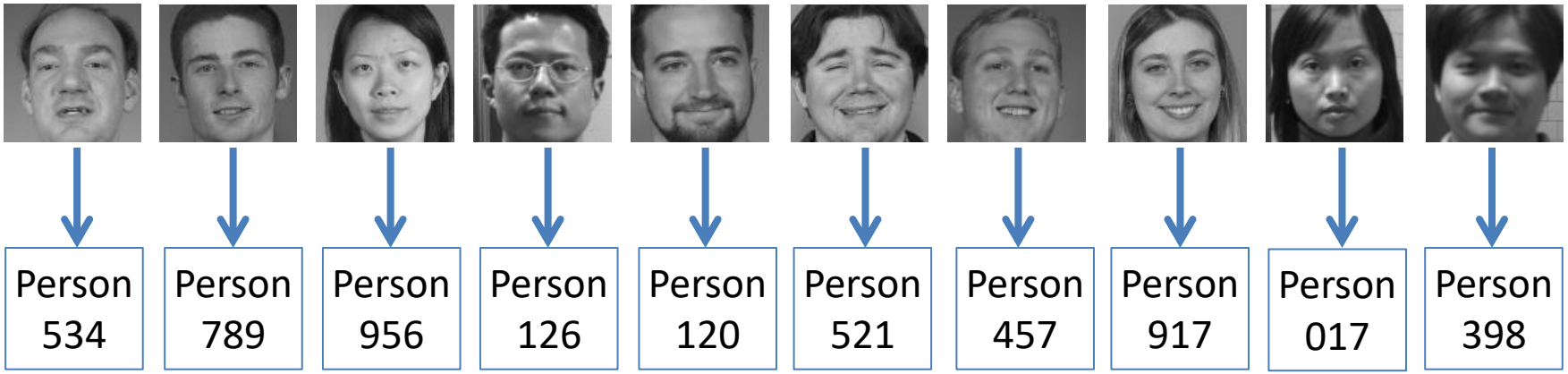
Comments on Image Types

- Your code may assume that your grayscale image has one value per pixel, but it may have 3 or 4 values per pixel.
- Displaying an image may require the image to have values of a certain type, and within a certain range. For example, matplotlib.pyplot.imread:
  - Works fine with uint 8 images with values in the 0-255 range.
  - Works fine with float images with values in the 0.0-1.0 range.
  - **Does not work without extra parameters** with float images with values in the 0-255 range.
- Bottom line: always doublecheck image type, shape, range.

Image Classification

- Goal: recognize the category that each image belongs to.
- Example: face recognition
  - Input: an image of a face.
  - Output: the ID of the person.

example inputs

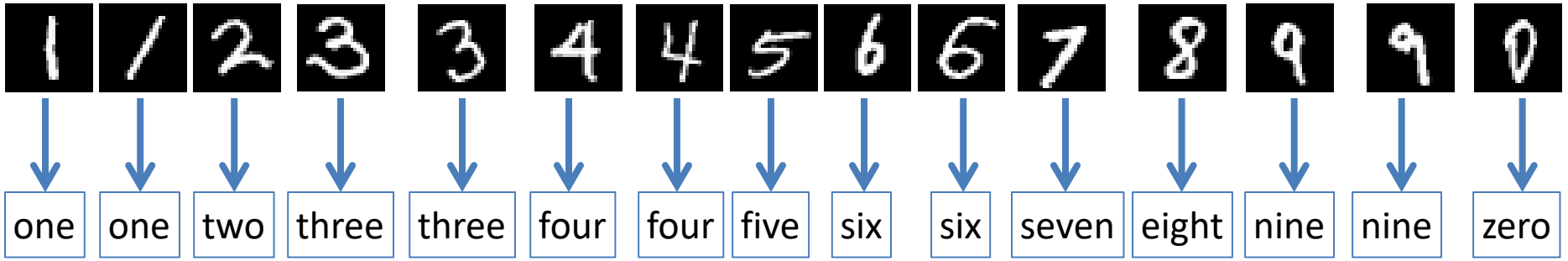


desired outputs (class labels)

Image Classification

- Example: The MNIST dataset of handwritten digits.
  - Input: a 28x28 grayscale image showing a single handwritten digit.
  - Output: the class label (10 possible cases, from zero to nine).
  - 60,000 training images.
  - 10,000 test images.

example inputs



desired outputs (class labels)

Loading the MNIST Dataset

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

dataset = tf.keras.datasets.mnist
(training_set, test_set) = dataset.load_data()
(training_images, training_labels) = training_set
(test_images, test_labels) = test_set
```

- The MNIST dataset is preloaded in Keras.
- The code above loads the dataset and creates four arrays:
  - training_images, training_labels
  - test_images, test_labels

Format of the Data

```
print("training images shape:", training_images.shape, "")
print("training images dtype:", training_images.dtype, "")
print("training labels shape:", training_labels.shape, "")

print("test images shape:", test_images.shape, "")
print("test images dtype:", test_images.dtype, "")
print("test labels shape:", test_labels.shape, "")
```

Output:

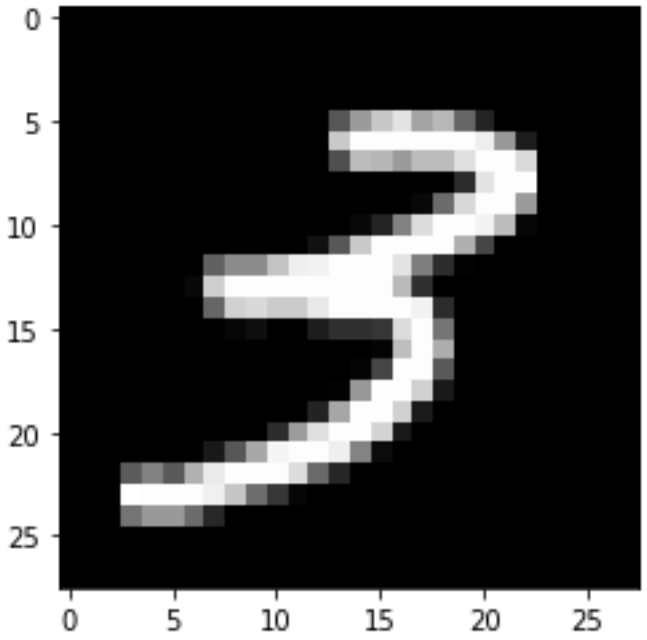
```
training images shape: (60000, 28, 28)
training images dtype: uint8
training labels shape: (60000,)
test images shape: (10000, 28, 28)
test images dtype: uint8
test labels shape: (10000,)
```

Visualizing Examples

```
training_index = 581
example_image = training_images[training_index]
label = training_labels[training_index]
plt.imshow(example_image, cmap='gray')
print("dtype:", example_image.dtype)
print("shape:", example_image.shape)
print("class label:", label)
```

Output:

```
dtype: uint8
shape: (28, 28)
class label: 3
```



Building a Neural Network

```
model = tf.keras.Sequential([
   tf.keras.layers.Flatten(input_shape=(28, 28)),
   tf.keras.layers.Dense(512, activation='tanh'),
   tf.keras.layers.Dense(10, activation="softmax")
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

- We are building a 3-layer model.
- Notice the “Flatten” layer.
  - Input images are 28x28.
  - The Flatten layer flattens its 2D input into a 1D array of length 784.

Model Summary

model.summary()

Output:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 flatten_1 (Flatten)          (None, 784)               0
 dense_2 (Dense)              (None, 512)               401920
 dense_3 (Dense)              (None, 10)                5130

=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

Training and Testing

```
training_inputs = training_images / 255.0
test_inputs = test_images / 255.0

model.fit(training_inputs, training_labels, epochs=15)
test_loss, test_acc = model.evaluate(test_inputs,  test_labels, verbose=0)

print('\nTest accuracy: %.2f%%' % (test_acc * 100))
```

Output:

```
Epoch 1/15
1875/1875 [==============================] - 5s 3ms/step -
loss: 0.2696 - accuracy: 0.9210
...
Epoch 15/15
1875/1875 [==============================] - 4s 2ms/step -
loss: 0.0072 - accuracy: 0.9977

Test accuracy: 98.14%
```

Training and Testing

```
training_inputs = training_images / 255.0
test_inputs = test_images / 255.0

model.fit(training_inputs, training_labels, epochs=15)
test_loss, test_acc = model.evaluate(test_inputs,  test_labels, verbose=0)

print('\nTest accuracy: %.2f%%' % (test_acc * 100))
```

- Notice that we do data normalization before training the model.
- We divide training and test inputs by 255, to make sure that their ranges are between 0 and 1.

```
(training_number, rows, cols) = training_images.shape
(test_number, rows, cols) = test_images.shape

model = tf.keras.Sequential([
    tf.keras.layers.Dense(512, activation='tanh'),
    tf.keras.layers.Dense(10, activation="softmax")
])
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])

training_inputs = training_images / 255.0
training_inputs = training_inputs.reshape(training_number, rows*cols)
test_inputs = test_images / 255.0
test_inputs = test_inputs.reshape(test_number, rows*cols)

model.fit(training_inputs, training_labels, epochs=15)
test_loss, test_acc = model.evaluate(test_inputs,  test_labels, verbose=0)
```

Instead of using a “Flatten” layer, we can vectorize the data explicitly:

Convolution

- Convolution is an extremely common image operation.
- This operation requires an image and a **kernel**.
- The kernel is also called a "**filter**", and convolution is also called a "filtering" operation.
  - Note: if you are familiar with convolutions from signal processing, here we define convolutions slightly differently (filters are not reversed, convolutions and filtering are treated as the same thing).
- Typically the kernel has an odd number of rows and an odd number of columns.
  - This is not strictly necessary, but it makes the notation and code more convenient.
- The kernel specifies the convolutional function.
  - Similar to how a matrix specifies a linear function.
- The image is the input to the convolution function.

Example With Numbers

Input image: 7x9

58	**17**	**75**	9	51	54	21	18	91
6	**65**	**19**	93	52	36	31	23	98
24	**74**	**69**	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

Result: 7x9

	1841							

- For the time being, we postpone the discussion of how to compute values at the boundaries (top and bottom row, left and right column).
- We assume that the filter has an odd number of rows, and an odd number of columns.
  - This way, the center of a filter corresponds to a specific position on the filter (position 2,2 for a 3x3 filter).

Example With Numbers

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(2,2)?
  - Align the filter with the image, so that the center of the filter aligns with position (2,2) of the image.
- We must sum up all these values:

58*5	17*4	75*1
6*2	65*1	19*2
24*6	74*9	69*7

Result: 7x9

	1841							

- result(2,2): 1841
- Alternatively, we can say that result(2,2) is a dot product.
  - We vectorize the image window and the kernel.
  - We take the dot product between those two vectors.

Example With Numbers

Input image: 7x9

58	**17**	**75**	**9**	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(2,3)?

Result: 7x9

	1841	**?**						

Example With Numbers

Input image: 7x9

58	**17**	**75**	**9**	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(2,3)?
  - Align the filter with the image, so that the center of the filter aligns with position (2,3) of the image.
- We must sum up all these values:

17*5	75*4	9*1
65*2	19*1	93*2
74*6	69*9	78*7

Result: 7x9

	1841	**2340**						

- result(2,3): 2340

Example With Numbers

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	**65**	**19**	**49**	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(3,3)?
  - Align the filter with the image, so that the center of the filter aligns with position (3,3) of the image.
- We must sum up all these values:

65*5	19*4	93*1
74*2	69*1	78*2
65*6	19*9	49*7

Result: 7x9

	1841	2340						
		1771						

- result(3,3): 1771

Example With Numbers

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	**65**	**19**	**49**	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(i,j)?
  - Align the filter with the image, so that the center of the filter aligns with position (i,j) of the image.
  - Multiply each value of the filter with the corresponding value in the image.
  - Sum up the results.

Result: 7x9

	1841	2340	2387	2477	2368	1825	1541	
	1503	**1771**	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

Result: 7x9

	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

- What do we do at position 1, 1?
  - According to previous instructions: “Align the filter with the image, so that the center of the filter aligns with position (1,1) of the image.”
  - Problem: some part of the filter does not align with any values in the image.
- The same issue arises at the top and bottom row, and the left and right column.
  - If the filter had size 5x7, we would have this problem at the top 2 rows, bottom 2 rows, left 3 columns, right 3 columns.

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

Result: 7x9

	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

- Main answer: **we do not care about boundary values**.
- A convolution is usually an intermediate step.
  - Some other module will use the convolution result to do something.
  - It is up to that other module to make sure that it ignores boundary values.
- In practice, since we do not care, we can follow various conventions, see next slides.

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- Option 1: just use image values that align with filter values.
- So, for position (1,1) of the result, we sum up:

	58*1	17*2
	6*9	65*7

Result: 7x9

601								
	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- Option 1: just use image values that align with filter values.
- So, for position (1,2) of the result, we sum up:

58*2	17*1	75*2
6*6	65*9	19*7

Result: 7x9

601	1037							
	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- Option 1: just use image values that align with filter values.
- So, for position (1,2) of the result, we sum up:

58*2	17*1	75*2
6*6	65*9	19*7

Result: 7x9

601	1037	1339	1576	1455	1051	821	1321	1147
1119	1841	2340	2387	2477	2368	1825	1541	1258
1040	1503	1771	2014	2765	2229	1619	1089	939
1405	1831	1888	2059	2407	2619	2722	2093	1264
787	1693	1879	1668	1971	2067	1817	1378	1053
745	1674	1793	2000	2127	1997	1641	1718	1352
128	491	773	886	702	731	618	502	482

- This way we can fill up all boundary values in the result.

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- Option 2: just put zeros at the boundary of the result.
- If aligning the center of the filter with location (i,j) gets part of the filter out of bounds, then we put a zero at result[i,j].

Result: 7x9

0	0	0	0	0	0	0	0	0
0	1841	2340	2387	2477	2368	1825	1541	0
0	1503	1771	2014	2765	2229	1619	1089	0
0	1831	1888	2059	2407	2619	2722	2093	0
0	1693	1879	1668	1971	2067	1817	1378	0
0	1674	1793	2000	2127	1997	1641	1718	0
0	0	0	0	0	0	0	0	0

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Kernel: 3x3

5	4	1
2	1	2
6	9	7

- Option 3: remove the boundary altogether.
  - Then, the result will have a smaller size than the original image.

Result: 5x7

1841	2340	2387	2477	2368	1825	1541
1503	1771	2014	2765	2229	1619	1089
1831	1888	2059	2407	2619	2722	2093
1693	1879	1668	1971	2067	1817	1378
1674	1793	2000	2127	1997	1641	1718

W = 3

$$\text{Kernel} = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} = \begin{bmatrix}\frac{1}{9} & \frac{1}{9} & \frac{1}{9}\\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9}\\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9}\end{bmatrix}$$

$$= \begin{bmatrix}0.111 & 0.111 & 0.111\\ 0.111 & 0.111 & 0.111\\ 0.111 & 0.111 & 0.111\end{bmatrix}$$

Example: Blurring

- To blur an image, we replace each pixel value with a weighted average of values in the neighborhood of the pixel.
  - Simplest: W x W neighborhood, all weights equal.

img

```
img = plt.imread('images/roofs1.jpg')
# create a WxW matrix of ones
W = 3
kernel = np.ones((W, W))

# divide values so they sum up to 1.
kernel = kernel / kernel.sum()
result = convolve2Db(img, kernel)
```

blurred, W=3

Example: Blurring

- To blur an image, we replace each pixel value with a weighted average of values in the neighborhood of the pixel.
  - Simplest: W x W neighborhood, all weights equal.

img

```
img = plt.imread('images/roofs1.jpg')
# create a WxW matrix of ones
W = 7
kernel = np.ones((W, W))

# divide values so they sum up to 1.
kernel = kernel / kernel.sum()
result = convolve2Db(img, kernel)
```

blurred, W=7

Example: Blurring

- To blur an image, we replace each pixel value with a weighted average of values in the neighborhood of the pixel.
  - Simplest: W x W neighborhood, all weights equal.

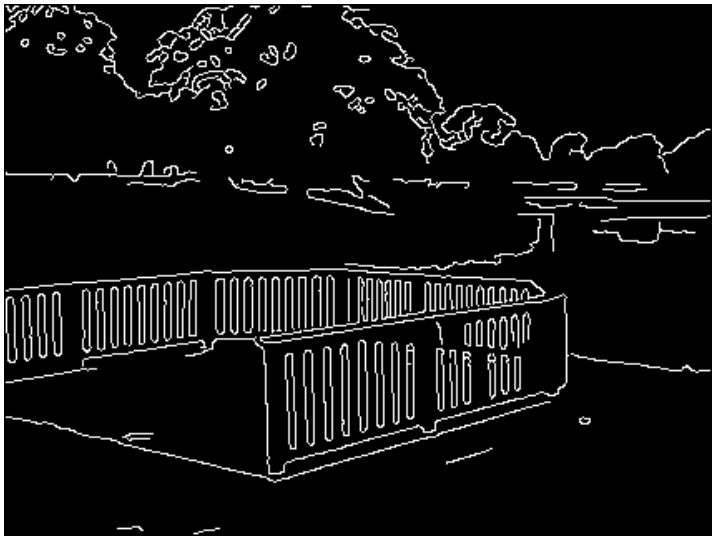
- As the kernel size gets larger, the result looks more blurry.

`img`

`blurred, W=11`

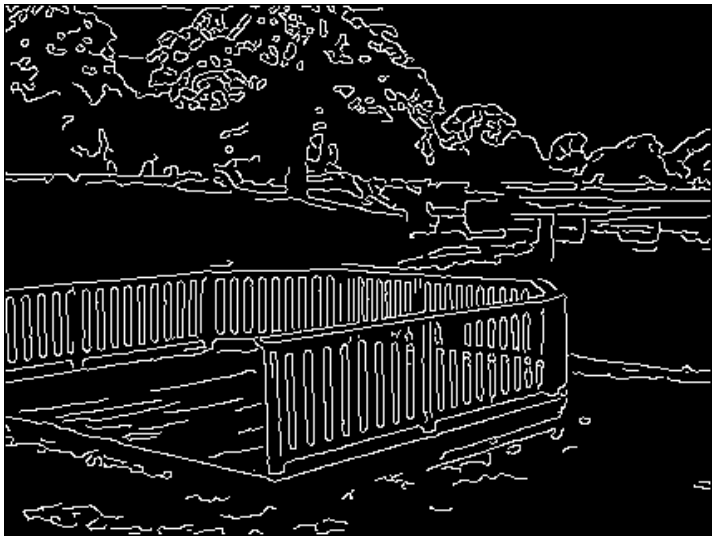
Example: Edges

- An edge pixel is a pixel at a "boundary" of an image part
- There is no single definition for what is a "boundary".
  - Breaking up an image into parts can be done in many ways.



fewer edges

input

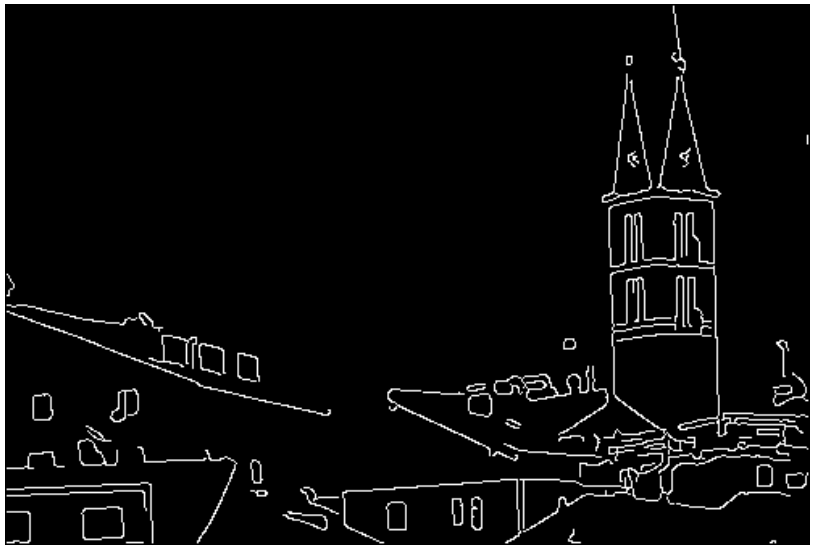


more edges

Example: Edges

- An edge pixel is a pixel at a “boundary” of an image part
- There is no single definition for what is a “boundary”.
  - Breaking up an image into parts can be done in many ways.

fewer edges



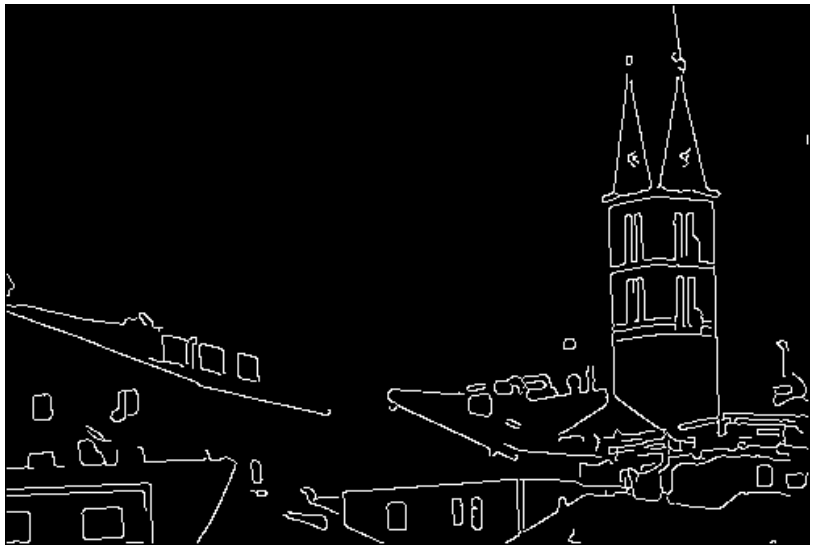
input

more edges

Example: Edges

- Producing such edge images is a computer vision topic.
- However, some preprocessing steps involve convolutions, and we can see how they work.

fewer edges



input

more edges

Vertical Edges

```
dx = np.array([[-1, 0, 1]])
img_dx = convolve2D(img, dx)
abs_dx = np.abs(img_dx)
show_image(abs_dx)
```

- What pixel (r,c) would get the highest value if we convolve with [-1,0,1]?
- A pixel such that the value at (r,c-1) is as low as possible, and the value at (r,c+1) is as high as possible.
- So, a high value in the result indicates a boundary between a darker region on the left and a brighter region on the right.

input

output

Vertical Edges

```
dx = np.array([[-1, 0, 1]])
img_dx = convolve2D(img, dx)
abs_dx = np.abs(img_dx)
show_image(abs_dx)
```

- What pixel (r,c) would get the lowest value if we convolve with [-1,0,1]?
- A pixel such that the value at (r,c-1) is as high as possible, and the value at (r,c+1) is as low as possible.
- So, a high negative value in the result indicates a boundary between a brighter region on the left and a darker region on the right.

input

output

Vertical Edges

```
dx = np.array([[-1, 0, 1]])
img_dx = convolve2D(img, dx)
abs_dx = np.abs(img_dx)
show_image(abs_dx)
```

- So, the absolute value of the result is high if the pixel is at a boundary between a region on the left and a region on the right with a high difference in intensity.
- This is why we take the absolute value.

input

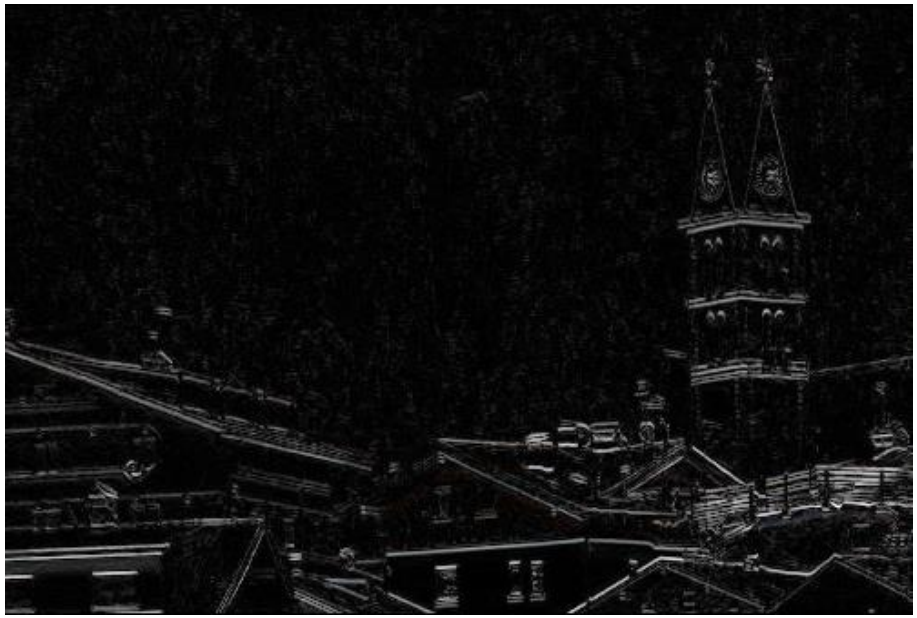
output

Horizontal Edges

```
dy = np.array([[-1], [0], [1]])
img_dy = convolve2D(img, dy)
abs_dy = np.abs(img_dy)
show_image(abs_dy)
```

- Here we convolve with a column vector: $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$
- The result shows high values for horizontal edges, i.e., for pixel is at a boundary between a region on the top and a region on the bottom with a high difference in intensity.

input



output