Machine Learning

Computer Vision

Salymbekov University

Miss Aliia Beishenalieva

aliya.beiwenalieva@gmail.com

How the pixel information is used in CV?

- **Canny Edge Detection**

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

img = cv.imread('messi5.jpg', cv.IMREAD_GRAYSCALE)
assert img is not None, "file could not be read, check with os.path.exists()"
edges = cv.Canny(img,100,200)

plt.subplot(121),plt.imshow(img,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(edges,cmap = 'gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])

plt.show()
```

https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

Object detection (traditional vs state-of-the-art)

The main difference between traditional object detection and state-of-the-art techniques lies in the approach to feature extraction and the accuracy achieved. Traditional object detection relies on manually designed features and simpler models, while modern techniques (often deep learning-based) use neural networks to learn complex, high-level features from large datasets.

Object detection (traditional vs state-of-the-art)

Feature Extraction
Traditional: Relies on hand-engineered features (e.g., edges, shapes), like Haar cascades and HOG, with classification by methods like SVM.
Modern: Uses CNNs to automatically learn features from data, capturing complex patterns through hierarchical layers.

Accuracy and Robustness
Traditional: Lower accuracy, struggles with variations in object orientation, size, and lighting.
Modern: High accuracy and better robustness in complex, real-world conditions due to detailed pattern learning.

Adaptability and Flexibility
Traditional: Less adaptable, requiring feature redesign for new object types.
Modern: Easily fine-tuned for new objects with sufficient labeled data, without major architecture changes.

Object detection

Traditional (non-deep learning) computer vision methods can still be effective for object detection, especially in simpler applications or where computational resources are limited.

Traditional Object Detection Techniques
Traditional methods focus on **feature extraction and matching**, using handcrafted features rather than learning features from data.

- **Haar Cascades** are commonly used for tasks like face detection and were very popular before deep learning. They work by applying a set of “Haar-like” features (patterns of black and white areas) to detect regions in an image that match specific templates (like a face).

Object detection

❖ **Situations Where Traditional Methods Are Suitable**

Simple Objects or Shapes: When the objects are simple, consistent, or have limited variation.

Controlled Environments: Where lighting, background, and object placement are stable, such as industrial applications.

Limited Resources: When hardware or computing power is constrained, traditional methods are lightweight and faster.

Real-Time Needs with Limited Complexity: For applications needing real-time processing but with basic detection needs, traditional methods can be efficient.

Object detection

❖ **Situations Where Traditional Methods Are Suitable**

High Variability: When objects vary significantly in shape, scale, or appearance, deep learning models like YOLO or Faster R-CNN perform much better.

Complex Environments: In dynamic or cluttered scenes, deep learning models can learn to differentiate objects better.

High Accuracy Requirements: When high accuracy is essential, deep learning methods are often superior.

Ability to Use Pretrained Models: Pretrained deep learning models can save time and work well for many standard objects, as they've already learned general features.

Object detection

❑ Haar cascades

```python
import cv2
import matplotlib.pyplot as plt

# Load the pre-trained Haar cascade classifier for face detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

# Read the image
image = cv2.imread('path_to_your_image.jpg')
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # Convert to grayscale

# Detect faces in the image
faces = face_cascade.detectMultiScale(gray_image, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

# Draw rectangles around detected faces
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x + w, y + h), (255, 0, 0), 2)  # Blue rectangle with thickness 2

# Convert image color to RGB (from BGR) for displaying with matplotlib
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Display the output using matplotlib
plt.imshow(image_rgb)
plt.axis('off')  # Hide axes
plt.show()
```

Thank you for your attention

Computer Vision

Salymbekov University

Miss Aliia Beishenalieva

aliya.beiwenalieva@gmail.com