CNN Architecture, OpenCV & Data Augmentation

Course:
Deep Learning with Tensorflow & Kersa 2

Developed by:
Mohammad Noorchenarboo

May 25, 2025

Current Section

Why Do We Need Convolution in Neural Networks?

Imagine trying to classify a 100x100 image with a fully connected ANN.

- **Each pixel** is treated as an independent feature.
- This results in $100 \times 100 = 10{,}000$ input features – very sparse, non-spatially aware.

Scalability Warning

Fully connected layers grow *quadratically* with input size – making them unsuitable for large images.

Problem: How do we efficiently detect local patterns (edges, textures, etc.) while preserving spatial structure?
Solution: Convolutional layers, which use local filters (kernels) and shared weights to learn spatial features.

$$Y(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(i+m, j+n) \cdot K(m,n)$$

A convolution applies a small kernel (filter) to different regions of the image, performing dot products to extract features.

Structure of a Convolutional Layer

Key Parameters:

- **Kernel Size (F)**: Size of the filter, e.g. 3×3, 5×5.
- **Stride (S)**: Number of pixels the filter moves each step.
- **Padding (P)**: Preserves dimensions using 'same' or reduces them with 'valid'.
- **Depth (D)**: Number of filters applied – outputs a volume, not a flat vector.

$$\text{Output Size} = \left\lfloor \frac{W - F + 2P}{S} \right\rfloor + 1$$

Example: 5x5 Input, 3x3 Filter, No Padding, Stride 1

- Output = $(5 - 3 + 0)/1 + 1 = 3$
- Resulting feature map = 3×3

Common Pitfall

Incorrect stride or padding can lead to mismatched input/output dimensions in deep CNN stacks.

Implementing Convolution in Keras

2D Convolution Layer in Keras

```
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(Conv2D(filters=32,
                 kernel_size=(3, 3),
                 strides=(1, 1),
                 padding='same',
                 activation='relu',
                 input_shape=(28, 28, 1)))

model.summary()
# Output shape: (None, 28, 28, 32)
```

Output Explanation

This layer outputs 32 feature maps of size 28x28, due to 'same' padding.

Why Shared Weights Matter

Traditional ANN: Every neuron has a unique set of weights.
CNN: Each kernel is used across the entire image (weight sharing).

$$\text{Total parameters} = (F \times F \times \text{Channels}) \times \text{Number of Filters}$$

Parameter Efficiency

CNNs drastically reduce the number of trainable parameters, enabling deeper models and faster training.

Example:

A 3×3 kernel on a single-channel image has $3 \times 3 = 9$ weights, regardless of image size.

Misconception

Weight sharing does not mean fewer computations – it only reduces parameters, not FLOPs (Floating-point Operations Per Second).

Summary Table: Convolutional Layers and Kernel Operations

Concept	**Explanation**
Kernel / Filter	A small weight matrix that moves across the image extracting local features.
Stride	Step size with which the kernel moves – affects output size.
Padding	'Same' keeps output size equal to input; 'Valid' reduces dimensions.
Weight Sharing	Kernel parameters reused across input space – fewer total weights.
Output Volume	For multiple filters, CNN outputs a 3D volume of feature maps.

Current Section

What Happens When Filters Don't Fit Perfectly?

Question: How do we handle cases when the filter size does not evenly divide the input image?

$$\text{Output size} = \left\lfloor \frac{W - F + 2P}{S} \right\rfloor + 1$$

Warning: Spatial Reduction

Without padding, convolving large kernels can rapidly shrink the image – making it unusable in deep CNNs.

Two padding modes commonly used:

- **'valid'**: No padding – output shrinks.
- **'same'**: Pads input so that output size equals input size.

Why Padding Matters

Padding allows deeper models by maintaining spatial resolution, especially in early layers.

Understanding ’valid’ Padding

’valid’ padding means no padding at all:

$$\text{Output size} = \left\lfloor \frac{W - F}{S} \right\rfloor + 1$$

Example: 7x7 Input, 3x3 Filter, Stride 1, ’valid’

- Output size = $\left\lfloor \frac{7-3}{1} \right\rfloor + 1 = 5$
- Output feature map = 5×5

Drawback

Each convolution shrinks the feature map, which can lead to loss of border information.

Understanding 'same' Padding

'same' padding ensures that output dimensions match input:

$$P = \left\lfloor \frac{F-1}{2} \right\rfloor \quad \text{(for odd-sized filters)}$$

Goal: Maintain output size = input size

Example: 7x7 Input, 3x3 Filter, Stride 1, 'same'

- Padding = $\left\lfloor \frac{3-1}{2} \right\rfloor = 1$
- Output = $\left\lfloor \frac{7-3+2\cdot 1}{1} \right\rfloor + 1 = 7$

Use Case

Use 'same' padding in early layers to retain spatial resolution for deep stacking.

Stride: The Step of Convolution

Stride (S) defines how far the filter jumps with each move.

- **Stride = 1:** Dense scanning of input.
- **Stride > 1:** Downsamples spatial dimensions.

$$\text{Output size} = \left\lfloor \frac{W - F + 2P}{S} \right\rfloor + 1$$

Example: 7x7 Input, 3x3 Filter, Padding = 0, Stride = 2

$$\left\lfloor \frac{7-3+0}{2} \right\rfloor + 1 = 3 \Rightarrow \text{Output size = } 3 \times 3$$

Oversized Strides

Large strides may skip important features and lead to aliasing.

Stride and Padding in Keras

Experimenting with Stride and Padding

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D

model = Sequential()

# SAME padding
model.add(Conv2D(16, (3,3), strides=(1,1), padding='same',
    input_shape=(28,28,1)))
# VALID padding
model.add(Conv2D(32, (3,3), strides=(2,2), padding='valid'))

model.summary()
# Layer 1 Output: (28, 28, 16)
# Layer 2 Output: (13, 13, 32)
```

What This Means

'SAME' preserved the input size; 'VALID' with stride 2 reduced spatial dimensions to almost half.

Summary Table: Padding and Stride

Parameter	Description
'valid' Padding	No padding; output is smaller than input.
'same' Padding	Adds padding so output size equals input size.
Stride = 1	Moves filter by 1 pixel; maximum overlap.
Stride > 1	Skips pixels; reduces spatial size.
Padding and stride interaction	Controls spatial resolution and computational load.

Current Section

Why Do We Pool in CNNs?

Motivation: After convolution, feature maps may still be large. Pooling helps:

- Reduce spatial dimensions
- Introduce spatial invariance (translation robustness)
- Control overfitting by downsampling

$$\text{Output size} = \left\lfloor \frac{W - F}{S} \right\rfloor + 1$$

Definition

Pooling is a downsampling operation applied over each feature map independently.

Types:

- Max Pooling
- Average Pooling
- Global Pooling

Max Pooling with Numerical Example

Max Pooling: Selects the maximum value from each window.
Input Feature Map (4x4):

$$\begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 2 & 1 & 0 & 3 \\ 4 & 9 & 5 & 2 \end{bmatrix}$$

2x2 Pool, Stride 2 $\Rightarrow$ **Output (2x2):**

$$\begin{bmatrix} 6 & 8 \\ 9 & 5 \end{bmatrix}$$

Interpretation

The highest value in each 2×2 block is preserved; other details are discarded.

Average Pooling with Numerical Example

Average Pooling: Computes the average of values in each region.
Same 4x4 Input:

$$\begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 2 & 1 & 0 & 3 \\ 4 & 9 & 5 & 2 \end{bmatrix}$$

2x2 Pool, Stride 2 $\Rightarrow$ **Output (2x2):**

$$\begin{bmatrix} 3.75 & 5.25 \\ 4 & 2.5 \end{bmatrix}$$

Use Case

Average pooling is often used in tasks requiring smooth spatial reduction (e.g. regression, heatmaps).

Note

Average pooling may blur out sharp features or edges.

Global Pooling Layers

Global Pooling:

- Instead of using a fixed window, apply pooling across the entire feature map.
- Reduces each feature map to a **single value**.

$$\text{Global Max Pooling:} \quad y = \max(X) \qquad \text{Global Average Pooling:} \quad y = \frac{1}{N}\sum_{i=1}^{N} x_i$$

When to Use

Used before the final classification layer to flatten feature maps without needing a dense layer.

Example:

Global max pooling on a 3×3 feature map:

$$\begin{bmatrix} 0.5 & 1.2 & 0.3 \\ 0.4 & 2.1 & 0.8 \\ 0.6 & 0.7 & 1.0 \end{bmatrix} \Rightarrow \max = 2.1$$

Pooling Layers in Keras

Using Max and Average Pooling in Keras

```
from tensorflow.keras.layers import MaxPooling2D, AveragePooling2D,
     GlobalMaxPooling2D
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(AveragePooling2D(pool_size=(2, 2), strides=2,
     padding='valid'))
model.add(GlobalMaxPooling2D())

model.summary()
# Outputs:
# MaxPooling2D -> reduces spatial dimensions by 2
# AveragePooling2D -> same reduction
# GlobalMaxPooling2D -> outputs 1 value per feature map
```

Best Practice

Use max pooling for feature extraction in classification tasks; average pooling for smoother transitions.

Summary Table: Pooling Layers

Pooling Type	Behavior and Use Case
Max Pooling	Retains the most dominant value in each region; preserves edges.
Average Pooling	Computes the mean of values; useful for smooth generalization.
Global Max Pooling	Reduces each feature map to a scalar by selecting the max value.
Global Average Pooling	Computes average of each feature map; common in modern CNNs.
Stride	Determines how much to shift the pooling window; controls downsampling.

Current Section

Why Use OpenCV in CNN Pipelines?

Problem: Deep learning models require standardized image formats and intensities.
OpenCV (cv2) is an efficient C++-backed library used for:

- Reading and displaying images
- Converting color spaces
- Resizing, normalizing, and thresholding
- Augmenting training data

Why OpenCV?

- Fast image I/O and processing
- Consistent preprocessing for training/inference
- Direct NumPy compatibility

$$\text{CNN Input:} \quad X \in \mathbb{R}^{H \times W \times C} \Rightarrow \text{Standardized via OpenCV}$$

Reading and Displaying Images

Loading Image with OpenCV

```python
import cv2

img = cv2.imread('image.jpg')          # Loads image in BGR format
cv2.imshow('Original Image', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Under the Hood

cv2.imread() returns a NumPy array of shape $(H, W, 3)$:

- 3 channels: Blue, Green, Red
- Data type: unsigned 8-bit integers, range [0, 255]

$$\text{Pixel Value at (x, y)} = \text{BGR triplet} = [B, G, R]$$

Note

Images are loaded in BGR format, not RGB – this affects color transformations.

Grayscale Conversion: What Actually Happens?

cv2.cvtColor() can convert a BGR image to grayscale:

Convert Color to Grayscale

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Mathematics Behind Grayscale

$$\text{Gray} = 0.114 \cdot B + 0.587 \cdot G + 0.299 \cdot R$$

- Based on human eye sensitivity
- Emphasizes green, suppresses blue

Grayscale Conversion: What Actually Happens?

Numerical Example

For pixel: $[B, G, R] = [60, 120, 200]$:

$$\text{Gray} = 0.114 \cdot 60 + 0.587 \cdot 120 + 0.299 \cdot 200 = 6.84 + 70.44 + 59.8 = 137.08$$

$\Rightarrow \text{Rounded} = 137$

Caution

This is not a simple average – it is a perceptually weighted transformation.

Image Resizing: Scale Matters

Resizing an Image

```
resized = cv2.resize(gray, (64, 64), interpolation=cv2.INTER_AREA)
```

How It Works

Resizing interpolates pixel values to match target dimensions $(W_{\text{target}}, H_{\text{target}})$.

$$\text{Scaling Ratio} = \frac{W_{\text{target}}}{W_{\text{orig}}} \quad \text{New Pixel} = \text{Weighted average of surrounding pixels}$$

Example:

If resizing a 128×128 to 64×64, every 2×2 pixel block becomes one.

Interpolation Tip

Use `cv2.INTER_AREA` for shrinking and `cv2.INTER_CUBIC` or `INTER_LINEAR` for enlarging.

Pixel Normalization: From 0–255 to 0–1

Pixel Value Normalization

```
norm_img = resized / 255.0
```

Why Normalize?

- Neural nets train faster with smaller input ranges.
- Keeps gradients stable during backpropagation.

Example:

If pixel intensity = 137, then:

$$\text{Normalized} = \frac{137}{255} \approx 0.537$$

Note

CNNs assume input values in $[0, 1]$ or standardized (mean = 0, std = 1).

Complete Pipeline: From BGR to Normalized Tensor

Image Preprocessing Pipeline

```
import cv2
import numpy as np

img = cv2.imread('image.jpg')                        # BGR format
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)         # Grayscale
resized = cv2.resize(gray, (64, 64))                 # Resize
norm = resized / 255.0                               # Normalize

tensor = np.expand_dims(norm, axis=-1)               # Shape: (64, 64,
    1)
```

Final Shape

CNNs expect inputs with shape (H, W, C) – for grayscale, $C = 1$.

Summary Table: OpenCV Preprocessing Techniques

Technique	Mathematical Interpretation
Grayscale Conversion	Weighted sum: $0.114B + 0.587G + 0.299R$
Resizing	Interpolated mapping from source to target pixel grid
Normalization	$I_{\text{norm}} = \frac{I}{255}$
Tensor Conversion	Ensures shape (H, W, C) by adding a channel axis

Current Section

Why Do We Augment Images in Deep Learning?

Motivation:

- CNNs can easily overfit on small datasets.
- Augmentation simulates new training samples.
- It improves generalization by exposing the model to more data variation.

Definition

Image augmentation applies transformations (e.g. rotation, scaling) to create altered versions of existing images.

$$\text{New Image} = T(I) \quad \text{where } T \text{ is a transformation matrix}$$

Warning

Augmentation only applies to training data – not to validation/test sets.

Geometric Transformations in OpenCV

Translation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Image Translation with OpenCV

```python
import cv2
import numpy as np

tx, ty = 30, 20
M = np.float32([[1, 0, tx], [0, 1, ty]])
translated = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]))
```

Numerical Example

Pixel at (50, 50) will move to $(50 + 30, 50 + 20) = (80, 70)$

Rotation in OpenCV

Rotation around the center:

$$M = \begin{bmatrix} \cos\theta & -\sin\theta & (1-\cos\theta)x_c + \sin\theta\, y_c \\ \sin\theta & \cos\theta & (1-\cos\theta)y_c - \sin\theta\, x_c \end{bmatrix}$$

Rotate an Image

```
center = (img.shape[1]//2, img.shape[0]//2)
M = cv2.getRotationMatrix2D(center, angle=45, scale=1.0)
rotated = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]))
```

Numerical Example

Rotation by 45° moves point (1,0) to $(\cos 45^\circ, \sin 45^\circ) \approx (0.707, 0.707)$

Flipping, Scaling, and Zooming

Flipping:

cv2.flip(img, 1) $\Rightarrow$ Horizontal Flip   cv2.flip(img, 0) $\Rightarrow$ Vertical Flip

Numerical Example: 2x2 Image Array

Original grayscale image:

$$\text{img} = \begin{bmatrix} 100 & 150 \\ 200 & 250 \end{bmatrix}$$

Horizontal flip (cv2.flip(img, 1)):

$$\begin{bmatrix} 150 & 100 \\ 250 & 200 \end{bmatrix}$$

Vertical flip (cv2.flip(img, 0)):

$$\begin{bmatrix} 200 & 250 \\ 100 & 150 \end{bmatrix}$$

Flipping, Scaling, and Zooming

Scaling:

Zoom matrix: $\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \end{bmatrix}$ where $s > 1$ for zoom-in, $s < 1$ for zoom-out

Zooming with Resizing

```
zoomed = cv2.resize(img, None, fx=2, fy=2,
    interpolation=cv2.INTER_NEAREST)
# Output (approximate visual result):
# [[100 100 150 150]
#  [100 100 150 150]
#  [200 200 250 250]
#  [200 200 250 250]]
```

Keras: ‘ImageDataGenerator‘ and Random Augmentation

Keras-Based Augmentation

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator  #
     Import the class for image augmentation

datagen = ImageDataGenerator(                         # Create an instance with
     the following random transforms:
    rotation_range=20,                                # Random rotation in the
         range [-20Â°, +20Â°]
    width_shift_range=0.2,                            # Random horizontal shift
         by up to 20% of image width
    height_shift_range=0.2,                           # Random vertical shift by
         up to 20% of image height
    zoom_range=0.15,                                  # Random zoom-in or
         zoom-out by up to 15%
    horizontal_flip=True,                             # Random horizontal flip
         (mirror image)
    rescale=1./255                                    # Normalize pixel values
         from [0, 255] to [0, 1]
)

aug_iter = datagen.flow(X_train, y_train, batch_size=32)
# Create a generator that yields batches of augmented images and labels
```

Keras: 'ImageDataGenerator' and Random Augmentation

How It Works

Each batch is randomly transformed using defined ranges – image dimensions remain consistent.

Example:

`rotation_range=20` allows rotation in $[-20°, +20°]$ uniformly at random.

Summary Table: Augmentation Techniques and Effects

Augmentation	**Mathematical Behavior / Purpose**
Rotation	Rotation matrix about image center
Translation	Pixel shift: $x' = x + t_x$, $y' = y + t_y$
Scaling (Zoom)	Multiplies pixel grid by scale factor s
Flipping	Flips image along axis (horizontal/vertical)
‘ImageDataGenerator‘	Randomly applies transformations per batch during training

Current Section

Why Extract Features Before CNNs?

Motivation: Before CNNs, edges and gradients were used as engineered features to detect patterns and object boundaries.

What is a Gradient?

- A gradient measures how much pixel values change across the image.
- Computed in two directions: I_x (horizontal) and I_y (vertical).
- It helps detect areas of rapid intensity change – i.e., edges.

$$\nabla I(x,y) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right] = [I_x, I_y] \quad \Rightarrow \quad \|\nabla I(x,y)\| = \sqrt{I_x^2 + I_y^2}$$

Why It Matters

The stronger the gradient, the sharper the edge. High gradient magnitude highlights image boundaries, useful in edge detection and feature maps.

Sobel Edge Detection

Sobel operator estimates image gradients in x and y directions.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Gradient magnitude: $\sqrt{(G_x * I)^2 + (G_y * I)^2}$

Numerical Insight

Gradient is strongest where pixel intensity changes rapidly in horizontal or vertical directions.

Sobel Edge Detection

Numerical Example (Manual Calculation)

Input grayscale patch:

$$I = \begin{bmatrix} 10 & 20 & 30 \\ 20 & 40 & 60 \\ 30 & 60 & 90 \end{bmatrix}$$

Sobel filters for edge detection:

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolve the 3x3 center region:

$$I_x = (-1)(10) + 0(20) + 1(30) + (-2)(20) + 0(40) + 2(60) + (-1)(30) + 0(60) + 1(90) = 160$$

$$I_y = (1)(10) + 2(20) + 1(30) - (1)(30) - 2(60) - 1(90) = -180$$

Compute edge strength:

$$\text{Magnitude} = \sqrt{160^2 + (-180)^2} \approx 240.83$$

Sobel Edge Detection

Sobel Filtering in Code (With Detailed Comments)

```
import cv2
import numpy as np

# Define a simple grayscale image manually
img = np.array([[10, 20, 30],
                [20, 40, 60],
                [30, 60, 90]], dtype=np.uint8)  # 8-bit unsigned
                    integers

# Compute gradient in X direction using Sobel operator
sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
# cv2.CV_64F means: output will be 64-bit float (to store negative
    values like -180)
# 1 means derivative in x-direction, 0 in y-direction

# Compute gradient in Y direction
sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)
# 0 in x, 1 in y means vertical gradient

# Combine gradients to compute magnitude (edge strength)
magnitude = np.sqrt(sobelx**2 + sobely**2)

# Central pixel (1,1) expected magnitude: ~240.83
```

Sobel Edge Detection

Important Detail: Why Use `cv2.CV_64F`?

The Sobel operator computes derivatives that can be negative. - If the output is stored as `np.uint8` (0 to 255), negative values are clipped to 0. - `cv2.CV_64F` tells OpenCV to use 64-bit floating point numbers, preserving negative values. This is crucial to ensure correct gradient calculation.

Classical Limitation

Hand-crafted filters like Sobel are static. They cannot learn or adapt like CNN kernels, which are optimized during training.

Sobel Edge Detection

Term / Parameter	Explanation
$\nabla I(x, y)$	Gradient of intensity: rate of change in pixel values
I_x, I_y	Approximated using convolution (e.g., Sobel filters)
`cv2.CV_64F`	Ensures floating point precision and allows negative values
`ksize=3`	3x3 kernel size used in Sobel filtering
`np.sqrt(...)`	Combines gradients to compute edge strength at each pixel

Canny Edge Detection: A Multi-Step Process

Canny Edge Detection in OpenCV

```
edges = cv2.Canny(gray, threshold1=100, threshold2=200)
# Detects edges with lower and upper thresholds
```

Thresholding Logic

- **Strong edges:** gradient magnitude > 200 ⇒ kept
- **Weak edges:** between 100 and 200 ⇒ kept only if connected to strong edge
- **Below 100:** suppressed as noise

Why Use Two Thresholds?

A single threshold would miss weak but important edges. Dual thresholds help retain edge continuity while filtering noise.

Canny Edge Detection: A Multi-Step Process

Numerical Example: Hysteresis Thresholding

Gradient magnitude image:

$$\begin{bmatrix} 80 & 120 & 210 \\ 90 & 160 & 180 \\ 50 & 190 & 220 \end{bmatrix} \quad \text{Thresholds: } T_1 = 100, \; T_2 = 200$$

Classification:

- Values > 200: **Strong** edges $\Rightarrow$ keep (e.g., 210, 220)
- Values $100 < x < 200$: **Weak** edges $\Rightarrow$ maybe keep (e.g., 120, 160, 180, 190)
- Values < 100: **Discarded** (e.g., 80, 90, 50)

Connected weak pixels (like 190 connected to 220) are preserved.

Harris Corner Detection

Corner: A pixel where intensity changes significantly in both *x* and *y* directions – unlike edges (one direction) or flat regions (no change).
Harris Response Function:

$$R = \det(M) - k \cdot (\mathrm{trace}(M))^2 \quad \text{where } M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

M is the second-moment matrix, summarizing how intensity varies in a local patch.
Small *k* (e.g., 0.04) More sensitive to detecting **corners**; tolerates edges more
Large *k* (e.g., 0.1) Penalizes edges heavily; fewer false positives, but may miss corners

Harris Corners in OpenCV

```
gray_float = np.float32(gray)  # Convert to 32-bit float (required)
harris = cv2.cornerHarris(gray_float, blockSize=2, ksize=3, k=0.04)
# blockSize: neighborhood size for corner detection
# ksize: aperture for Sobel (used to compute I_x and I_y)
# k: sensitivity to edges vs. corners (typically 0.04-0.06)
```

Harris Corner Detection

Numerical Example: Response Calculation

Let the image patch produce gradients:

$$I_x = 4, \quad I_y = 5 \Rightarrow M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} 16 & 20 \\ 20 & 25 \end{bmatrix}$$

Now compute:

$$\det(M) = (16)(25) - (20)^2 = 400 - 400 = 0$$

$$\text{trace}(M) = 16 + 25 = 41 \Rightarrow R = 0 - 0.04 \cdot 41^2 = -67.24$$

Interpretation: Since $R < 0$, this pixel is likely on an edge.

Harris Corner Detection

What R Tells Us

- $R \gg 0 \Rightarrow$ strong intensity change in both directions $\Rightarrow$ **corner**
- $R \approx 0 \Rightarrow$ no change in either direction $\Rightarrow$ **flat region**
- $R < 0 \Rightarrow$ strong change in one direction only $\Rightarrow$ **edge**

Why Use float32?

`cv2.cornerHarris()` expects floating point precision to store non-integer derivatives and intermediate values. Using `uint8` will lead to inaccurate results.

Visualizing Features and Annotations

Drawing Features on Image

```
# Mark Canny edges in red
img[edges > 100] = [0, 0, 255]

# Annotate corners
img[harris > 0.01 * harris.max()] = [0, 255, 0]

# Draw a rectangle or circle
cv2.rectangle(img, (10,10), (50,50), (255,0,0), 2)
cv2.circle(img, (60, 60), 10, (0, 255, 255), -1)
```

Why Visualize?

- Debug preprocessing steps
- Verify model interprets relevant structures
- Annotate features for inspection

Summary Table: Feature Detection Methods

Method	Mathematical Operation and Use
Sobel	Computes $\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}$ to estimate edges
Canny	Multi-step pipeline: blur, Sobel, non-max suppression, thresholding
Harris Corners	Measures intensity variation using matrix eigenvalues
Feature Visualization	Overlay results to inspect edge/corner quality and position

Current Section

Complete Workflow: From Raw Image to Prediction

Objective: Use OpenCV to preprocess images, then train a CNN on MNIST/CIFAR-10 using TensorFlow.

Pipeline Steps:

1. Load image using OpenCV
2. Preprocess: grayscale, resize, normalize
3. Reshape for CNN input
4. Define and compile CNN model
5. Train and evaluate

$$X_{\text{CNN}} = \texttt{OpenCV(img)} \rightarrow \text{resize} \rightarrow \text{normalize} \rightarrow \text{expand\_dims}$$

Target Shape

CNN expects input shape: (N, H, W, C)

Preprocessing Pipeline Using OpenCV

OpenCV Preprocessing Function

```
def preprocess_image(path, target_size=(32, 32)):
    img = cv2.imread(path)                          # Read image from file
        (in BGR format by default)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)    # Convert image to
        grayscale
    resized = cv2.resize(gray, target_size)         # Resize image to target
        dimensions (default 32x32)
    norm = resized / 255.0                          # Normalize pixel values
        to range [0, 1]
    tensor = np.expand_dims(norm, axis=-1)          # Add channel dimension:
        (H, W) -> (H, W, 1)
    return tensor                                   # Return preprocessed
        image tensor
```

Numerical Example

Original: 128x128 RGB → Grayscale → Resize to 32x32 → Normalize → Add channel axis → (32, 32, 1)

Integration Ready

This function can be used in custom datasets, not just built-in Keras ones.

MNIST CNN Model with TensorFlow

CNN for MNIST (Grayscale)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
     Dense, Dropout
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2,2)),
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D((2,2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
     loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Why It Works

- Two conv-pool blocks
- Dense + dropout layer to control overfitting
- Output: 10 classes (digits 0–9)

CIFAR-10 CNN Model (RGB)

CNN for CIFAR-10 (Color)

```
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(32, 32, 3)),
    Conv2D(32, (3,3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),
    Dropout(0.25),

    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),
    Dropout(0.25),

    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

Design Insight

Deeper than MNIST model due to complexity of color images and 10 general object classes.

Train, Evaluate, and Integrate

Compile and Train Model

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=64, epochs=10,
     validation_split=0.2)
```

Model Evaluation

Use `model.evaluate()` to test on unseen data. You can also use OpenCV-processed inputs.

OpenCV + CNN Integration

You can feed any OpenCV-processed image directly into a trained CNN:

```
x = preprocess_image('custom_image.jpg', (28,28))
x = np.expand_dims(x, axis=0)  # Add batch dim
prediction = model.predict(x)
```

Summary Table: TensorFlow + OpenCV CNN Pipeline

Step	Technique and Explanation
Load & Convert Image	OpenCV reads image in BGR; convert to grayscale or RGB
Resize & Normalize	Resize to fixed shape (28x28 or 32x32); normalize to [0,1]
CNN Model (MNIST)	2 Conv layers + 1 Dense; input shape (28,28,1)
CNN Model (CIFAR-10)	More conv/pool layers; input shape (32,32,3)
Evaluation	Predict class from OpenCV-preprocessed image