VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 2: Edge Detection

Course: Digital Image and Video Processing

Supervisors:

Prof. Dr. Ly Quoc Ngoc

MSc. Nguyen Manh Hung

MSc. Pham Thanh Tung

Student:

23127266 - Nguyen Anh Thu

December 5, 2025

Contents

List of Figures

List of Tables

Chapter I: Introduction

1. Overview

In this assignment, I implement several edge detection techniques using NumPy. Color images $f(x, y) \in \mathbb{R}^3$ are first converted to grayscale $I(x, y)$, and all algorithms operate on this single-channel representation.

Traditional operators are reimplemented from their mathematical formulations and applied through discrete convolution on $I(x, y)$. A basic neural network model is also constructed to perform learning-based edge detection.

Finally, the results from the NumPy implementations and the neural network are compared with PyTorch's built-in functions to observe differences between classical methods, data-driven models, and optimized library routines.

1.1. Definition

What is an edge?

An edge in a digital image is a set of pixels where the image intensity changes abruptly. These pixels mark locations of significant intensity variation, often caused by sudden changes in illumination, surface orientation, or object structure. [1, p. 81]

How to detect an edge?

Edges are formed from pixels with derivative values that exceed a preset threshold. Thus, an edge is a 'local' concept that is based on a measure of intensity-level discontinuity at a point. [1, p. 81]

1.2. Problem Statement

Input. A grayscale digital image

$$f(x, y) : \Omega \subset \mathbb{Z}^2 \to [0, 255],$$

which maps each pixel coordinate (x, y) to an intensity value. *(This project uses grayscale images for processing.)*

Goal. Identify locations where $f(x, y)$ exhibits significant intensity discontinuities.

Output. An edge map

$$E(x, y) : \Omega \to \{0, 1\},$$

defined as

$$E(x, y) = \begin{cases} 1, & \text{if } g(x, y) > T, \\ 0, & \text{otherwise,} \end{cases}$$

where

$$g(x, y) = \mathcal{D}[f(x, y)]$$

is the edge response produced by an edge operator $\mathcal{D}$.

Problem. Design or select an operator $\mathcal{D}$ such that the resulting map $E(x, y)$ accurately marks true intensity discontinuities, achieves good spatial localization, and remains robust to noise.

1.3. Approaches

In this project, I implement the two main approaches:

- Traditional methods
- Deep learning-based methods

Traditional methods are the classical edge detection methods that are based on mathematical operators. Deep learning-based methods are the modern edge detection methods that are based on deep learning models. Each approach has several techniques as the figure I.1 below.

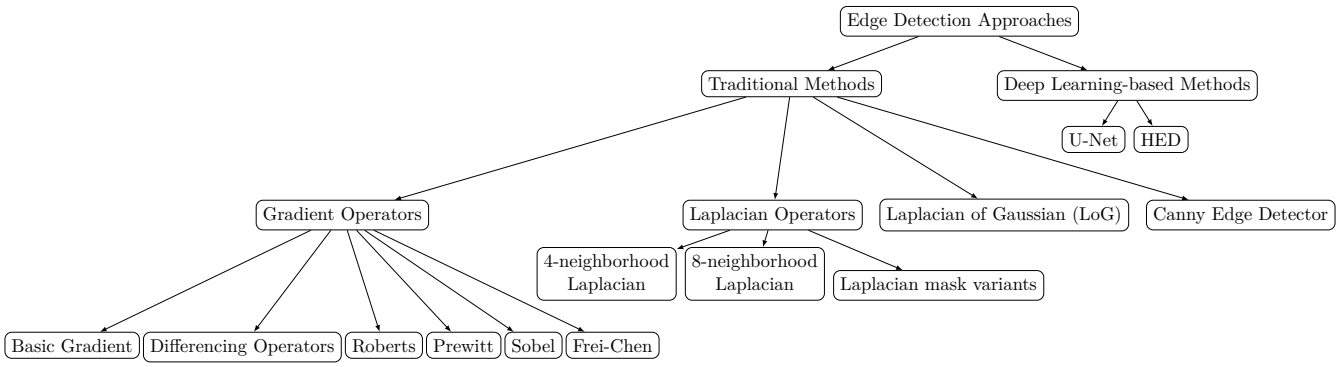


Figure I.1: Overview of Edge Detection Approaches

2. Objectives

The main objectives of this assignment are:

- Implement several edge detection techniques using NumPy.
- Compare the results from the NumPy implementations and the neural network with PyTorch's built-in functions.
- Train a basic neural network model to perform learning-based edge detection. Using dataset from Kaggle.
- Evaluate the performance of the edge detection techniques with metrics: mAP, ODS, OIS.

3. Program Structure

The program works directly with image files as input. Each image is loaded into a NumPy array representing pixel values in either grayscale or RGB format.

3.1. Input

- image file (PNG, JPG, JPEG).

3.2. Output

- saved output image results (PNG)

Chapter II: Implementation

1. Traditional Approaches

1.1. Overview

Traditional edge detection methods are built upon fundamental image derivatives and filtering operators. These techniques rely on detecting intensity changes in local neighborhoods to estimate the presence and direction of edges. Unlike modern learning-based methods, classical operators derive edges directly from mathematical approximations of the image gradient or the Laplacian.

Gradient-based methods estimate edges by computing the spatial derivatives of an image. Given a grayscale image $f(x, y)$, the first-order partial derivatives are approximated as:

$$\nabla f(x,y) = \begin{bmatrix} f_x(x,y) \\ \\ f_y(x,y) \end{bmatrix}, \qquad e(x,y) = \sqrt{f_x^2(x,y) + f_y^2(x,y)}.$$

An edge corresponds to locations where the magnitude $e(x, y)$ is large, and the edge orientation is:

$$\phi(x,y) = \arctan\left(\frac{f_y(x,y)}{f_x(x,y)}\right).$$

In what follows, I present all classical operators derived from finite-difference approximations of these derivatives.

1.2. Gradient Operators

1.2.1 Overview

Gradient-based edge detectors approximate the first-order derivatives of the image intensity function. Edges are identified as locations where the gradient magnitude is large, typically

corresponding to rapid intensity changes.

1.2.2 Basic Gradient Operator

a) Theory. The basic gradient operator computes the first-order derivative by simple forward or backward differences. It identifies edges as high local intensity changes.

b) Method. Using simple differencing:

$$f_x(x,y) \approx f(x+1,y) - f(x,y), \quad f_y(x,y) \approx f(x,y+1) - f(x,y).$$

Edge magnitude:

$$e(x,y) = \sqrt{f_x^2(x,y) + f_y^2(x,y)}.$$

c) Algorithm.

```
Function basic_gradient(image f):
    For each pixel (x, y):
        fx = f(x+1, y) - f(x, y)
        fy = f(x, y+1) - f(x, y)
        e(x, y) = sqrt(fx^2 + fy^2)
    Return edge magnitude e
```

1.2.3 Differencing Operators

a) Theory. Differencing improves localization of edges, especially for ramp edges, by using symmetric differences. This centered difference reduces bias and detects the true center of gradient transitions.

b) Method.

$$f_x(x,y) \approx f(x,y+1) - f(x,y-1), \qquad f_y(x,y) \approx f(x+1,y) - f(x-1,y).$$

c) Algorithm.

```
Function central_difference_operator(image f):
    For each pixel (x, y):
        fx = f(x, y+1) - f(x, y-1)
```

```
        fy = f(x+1, y) - f(x-1, y)
        e(x, y) = sqrt(fx^2 + fy^2)
    Return edge magnitude e
```

1.2.4 Roberts Operator

a) Theory. Roberts uses diagonal gradients to capture edges with minimum neighborhood (actually 2×2), making it sensitive but noisy.

b) Method.

$$f_x(x,y) \approx f(x,y) - f(x+1,y+1), \qquad f_y(x,y) \approx f(x,y+1) - f(x+1,y).$$

c) Algorithm.

```
Function roberts_operator(image f):
    For each pixel (x, y):
        fx = f(x, y) - f(x+1, y+1)
        fy = f(x, y+1) - f(x+1, y)
        e(x, y) = sqrt(fx^2 + fy^2)
    Return edge magnitude e
```

1.2.5 Prewitt Operator

a) Theory. Prewitt smooths the image in one direction and differentiates in the other. It approximates derivatives using uniform weights.

b) Method.

$$W_x = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \qquad W_y = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

$$f_x = f * W_x, \qquad f_y = f * W_y.$$

c) Algorithm.

```
Function prewitt_operator(image f):
    Compute fx = convolution(f, Wx)
    Compute fy = convolution(f, Wy)
    For each pixel (x, y):
        e(x, y) = sqrt(fx(x, y)^2 + fy(x, y)^2)
    Return edge magnitude e
```

1.2.6 Sobel Operator

a) Theory. Sobel improves over Prewitt by applying a stronger smoothing (central weighting) to reduce noise effects.

b) Method.

$$W_x = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \qquad W_y = \frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}.$$

c) Algorithm.

```
Function sobel_operator(image f):
    Compute fx = convolution(f, Wx)
    Compute fy = convolution(f, Wy)
    For each pixel (x, y):
        e(x, y) = sqrt(fx(x, y)^2 + fy(x, y)^2)
    Return edge magnitude e
```

1.2.7 Frei–Chen Operator

a) Theory. Frei–Chen is a more "rotationally symmetric" operator than Sobel, using a normalization factor $k = \sqrt{2}$ for isotropic response.

b) Method.

$$W_x = \begin{bmatrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & 0 \\ -1 & -\sqrt{2} & -1 \end{bmatrix}, \qquad W_y = \begin{bmatrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 1 & 0 & -1 \end{bmatrix}.$$

c) Algorithm.

```
Function frei_chen_operator(image f):
    Compute fx = convolution(f, Wx)
    Compute fy = convolution(f, Wy)
    For each pixel (x, y):
        e(x, y) = sqrt(fx(x, y)^2 + fy(x, y)^2)
    Return edge magnitude e
```

1.3. Laplacian Operators

Laplacian-based edge detection relies on the second-order derivatives of the image. The Laplacian highlights regions where the intensity curvature changes abruptly, corresponding to zero-crossings typically associated with edges [2]. For a continuous intensity function $f(x, y)$, the Laplacian is

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

In discrete images, the operator is approximated using masks derived from second-order finite differences.

1.3.1 4-neighborhood Laplacian

a) Theory.

The 4-neighborhood Laplacian estimates curvature using the four adjacent pixels (up, down, left, right). It corresponds to the discrete approximation

$$\nabla^2 f(x, y) \approx f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y).$$

b) Method.

This yields the convolution mask:

$$W_4 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

[1]

Applying the mask:

$$g(x, y) = (f * W_4)(x, y).$$

Edges are located at zero-crossings or magnitude peaks of g.

c) Algorithm.

```
Function laplacian_4neighbor(image f):
    Compute g = convolution(f, W4)
    Optionally detect zero-crossings in g
    Return g
```

1.3.2 8-neighborhood Laplacian

a) Theory.

The 8-neighborhood Laplacian includes diagonal neighbors, producing a more isotropic approximation of curvature. The discrete operator is:

$$\nabla^2 f(x, y) \approx \sum_{\text{8 neighbors}} f(x_i, y_i) - 8f(x, y).$$

b) Method.

The corresponding mask is:

$$W_8 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

[1]

$$g(x, y) = (f * W_8)(x, y).$$

Compared to the 4-neighborhood version, this mask produces stronger responses, especially for diagonal structures.

c) Algorithm.

```
Function laplacian_8neighbor(image f):
    Compute g = convolution(f, W8)
    If binary edges required:
        Detect zero-crossings of g
    Return g
```

1.3.3 Laplacian Mask Variants

a) Theory.

Laplacian masks can be modified to adjust sensitivity, noise amplification, or edge thickness. Variants arise from different discrete approximations of second derivatives or by adding smoothing to reduce noise sensitivity.

The variants presented in the lecture (p.44–48) apply different weighting distributions to the second derivative stencil.

b) Method.

Common Laplacian variants include:

$$W_{v1} = \begin{bmatrix} 0 & -\frac{3}{8}h & \frac{3}{8}h & 0 \end{bmatrix},$$

$$W_{v2} = \begin{bmatrix} 0 & -\frac{3}{16}h & 0 & \frac{3}{16}h & 0 \end{bmatrix},$$

$$W_{v3} = \begin{bmatrix} 0 & -\frac{h}{8} & -\frac{h}{8} & 0 & \frac{h}{8} & \frac{h}{8} & 0 \end{bmatrix},$$

$$W_{v4} = \begin{bmatrix} 0 & -\frac{h}{16} & -\frac{h}{8} & -\frac{h}{16} & 0 & \frac{h}{16} & \frac{h}{8} & \frac{h}{16} & 0 \end{bmatrix}.$$

(Here $h = b - a$ denotes the local intensity step size, following the finite-difference derivation in the lecture slides.)

These variants allow tuning of directional sensitivity and transition sharpness while preserving the second-derivative nature of the operator.

c) Algorithm.

```
Function laplacian_variant(image f, mask Wv):
    Compute g = convolution(f, Wv)
    If binary edges required:
        Detect zero-crossings or apply thresholding
    Return g
```

1.3.4 Laplacian of Gaussian (LoG)

a) Theory.

LoG detects edges by computing the second derivative of the image after Gaussian smoothing. The core mathematics:

$$\mathrm{LoG}(x,y) = \nabla^2(G_\sigma * f)(x,y) = f * \left(\nabla^2 G_\sigma\right),$$

where

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right),$$

$$\nabla^2 G_\sigma(x,y) = \left(\frac{x^2+y^2-2\sigma^2}{\sigma^4}\right)\frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right).$$

Edges appear at zero-crossing points of the LoG image:

$$L(x,y) = 0 \quad \text{and sign changes in the neighborhood.}$$

b) Method.

Step 1: Gaussian smoothing Smooth the input image with a Gaussian kernel:

$$S = G_\sigma * f.$$

Step 2: Apply LoG kernel Instead of computing Laplacian(S), I use the derivative of Gaussian:

$$L(x, y) = (f * \nabla^2 G_\sigma)(x, y).$$

The LoG kernel is computed dynamically from the formula above. For discrete implementation, the kernel size is typically 6σ rounded to the nearest odd number. A sample discrete 5×5 approximation:

$$LoG \approx \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}.$$

Step 3: Zero-crossing detection At a point (x, y):

$$\text{Edge}(x, y) = \begin{cases} 1, & \exists (i, j) \in \mathcal{N} : L(x, y)\, L(x + i, y + j) < 0, \\ 0, & \text{otherwise.} \end{cases}$$

With 4-neighborhood:

$$\mathcal{N} = \{(1, 0), (0, 1), (-1, 0), (0, -1)\}.$$

The final result:

$$E(x, y) = \text{Edge}(x, y).$$

c) Algorithm.

```
Function LoG_edge_detection(f, sigma):

    # Step 1: Gaussian smoothing
    S = G_sigma * f

```

```
    # Step 2: Apply LoG filter
    LoG_kernel = Laplacian(G_sigma)
    L = f * LoG_kernel

    # Step 3: Zero-crossings
    For each pixel (x,y):
        For each neighbor (i,j) in N:
            If L(x,y) * L(x+i, y+j) < 0:
                E(x,y) = 1
                break

    Return E
```

1.4. Canny

1.4.1 Theory

The Canny detector identifies edges by optimizing the signal-to-noise ratio, edge localization accuracy, and the single-edge response criterion. Mathematically, the process consists of Gaussian smoothing, gradient computation, non–maximum suppression, double thresholding, and edge linking through hysteresis

1.4.2 Method

Step 1: Gaussian smoothing Smooth the input image to reduce noise before computing derivatives:

$$f_s(x, y) = f(x, y) * G_\sigma(x, y),$$
$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right).$$

Step 2: Compute gradient magnitude and direction Use Sobel filters or derivative masks:

$$f_x = f_s * W_x, \qquad f_y = f_s * W_y,$$

$$M(x,y) = \sqrt{f_x^2 + f_y^2},$$

$$\theta(x,y) = \tan^{-1}\left(\frac{f_y}{f_x}\right).$$

Step 3: Non–maximum suppression Retain only local maxima along the gradient direction:

$$M_{\text{thin}}(x,y) = \begin{cases} M(x,y), & \text{if } M(x,y) \text{ is maximal along } \theta(x,y), \\ 0, & \text{otherwise.} \end{cases}$$

Step 4: Double thresholding Apply two thresholds T_{high} and T_{low}:

$$E(x,y) = \begin{cases} \text{strong edge}, & M_{\text{thin}}(x,y) \geq T_{high}, \\ \text{weak edge}, & T_{low} \leq M_{\text{thin}}(x,y) < T_{high}, \\ 0, & M_{\text{thin}}(x,y) < T_{low}. \end{cases}$$

Step 5: Hysteresis edge tracking Retain a weak edge if it connects to at least one strong edge in the 8-neighborhood:

$$E_{\text{final}}(x,y) = \begin{cases} 1, & \text{if weak edge and } \exists (i,j) \in N_8 : E(i,j) = \text{strong}, \\ 1, & \text{if strong edge}, \\ 0, & \text{otherwise.} \end{cases}$$

[2, 3].

1.4.3 Algorithm

```
Function Canny_edge_detection(f, sigma, T_low, T_high):

```

```
    # Step 1: Gaussian smoothing
    f_s = G_sigma * f

    # Step 2: Gradient magnitude and direction
    fx = f_s * Wx
    fy = f_s * Wy
    M  = sqrt(fx^2 + fy^2)
    theta = atan2(fy, fx)

    # Step 3: Non-maximum suppression
    M_thin = suppress_nonmax(M, theta)

    # Step 4: Double threshold
    Label pixels as strong, weak, or zero using T_high and T_low

    # Step 5: Hysteresis tracking
    For each weak pixel (x,y):
        If any 8-neighbor is strong:
            E_final(x,y) = 1
        Else:
            E_final(x,y) = 0

    Return E_final
```

2. Deep Learning Approaches

2.1. BIPEDv2 Dataset

a) Description.

BIPEDv2 is a high-resolution benchmark for edge detection in natural outdoor scenes. The dataset consists of a set of RGB images $I_i \in \mathbb{R}^{1280\times720\times3}$, each associated with a fine-grained binary edge label map $G_i \in \{0,1\}^{1280\times720}$, denoting the presence (1) or absence (0) of edge at each pixel.

b) Statistics.

BIPEDv2 contains 250 manually labeled outdoor images (resolution 1280×720). The annotations are curated by expert annotators, removing redundant or spurious boundaries and ensuring highly accurate ground truth contours. The dataset is divided into 200 images for training and 50 images for testing. It is publicly available for reproducible benchmarking of edge detection methods.

2.2. Model Architecture

a) U-Net Architecture.

The edge detection model employs a U-Net architecture, which consists of an encoder-decoder structure with skip connections. The encoder progressively downsamples the input image to extract hierarchical features, while the decoder upsamples these features to reconstruct the edge map at the original resolution.

The model takes an RGB image $I \in \mathbb{R}^{3\times H\times W}$ as input and produces edge probability logits $L \in \mathbb{R}^{1\times H\times W}$. The architecture comprises:

- **Encoder**: Four downsampling blocks that reduce spatial dimensions while increasing feature channels ($32 \to 64 \to 128 \to 256$);
- **Decoder**: Three upsampling blocks with skip connections that restore spatial resolution while reducing channels;
- **Output layer**: A 1×1 convolution that maps features to edge logits.

Each encoder block applies max pooling followed by two consecutive convolutions with batch normalization and ReLU activation. Each decoder block performs bilinear upsampling, concatenates with the corresponding encoder feature map via skip connections, and applies two convolutions.

b) Forward Pass.

The forward pass of the model can be described as:

$$x_1 = \text{DoubleConv}(I, 32)$$
$$x_2 = \text{Down}(x_1, 64)$$
$$x_3 = \text{Down}(x_2, 128)$$
$$x_4 = \text{Down}(x_3, 256)$$
$$y_1 = \text{Up}(x_4, x_3)$$
$$y_2 = \text{Up}(y_1, x_2)$$
$$y_3 = \text{Up}(y_2, x_1)$$
$$L = \text{Conv}_{1\times1}(y_3)$$

where x_i represents encoder features at different scales, y_i represents decoder features, and L is the output logit map.

2.3. Data Preprocessing

a) Dataset Loading.

The dataset loader parses the list file to extract image pairs. For each entry, it constructs paths to RGB images and corresponding edge maps based on the split (train or test). Training images are located in `imgs/train/rgbr/real/`, while test images are in `imgs/test/rgbr/`. Edge maps follow the same structure under `edge_maps/`. All images are resized to $(320, 320)$ pixels.

b) Data Augmentation.

During training, images and edge maps are randomly flipped horizontally and vertically with probability 0.5 each. RGB images are resized using bilinear interpolation, while edge maps use nearest-neighbor interpolation to preserve binary structure. Edge maps are binarized at threshold 0.5 after conversion to tensors.

2.4. Loss Function and Metrics

a) Loss Function.

The model is trained using binary cross-entropy loss with logits (BCEWithLogitsLoss), which combines sigmoid activation and binary cross-entropy loss for numerical stability. Given predicted logits $L \in \mathbb{R}^{B\times1\times H\times W}$ and ground truth edge maps $G \in \{0, 1\}^{B\times1\times H\times W}$, the loss is computed as:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[G_i \log(\sigma(L_i)) + (1 - G_i)\log(1 - \sigma(L_i))\right]$$

where $\sigma(\cdot)$ denotes the sigmoid function, $N = B \times H \times W$ is the total number of pixels, and $G_i \in \{0, 1\}$ represents the ground truth label at pixel i.

b) Evaluation Metric.

Intersection over Union (IoU) is used as the primary evaluation metric. For predicted probability map $P \in [0, 1]^{B\times1\times H\times W}$ and ground truth $G \in \{0, 1\}^{B\times1\times H\times W}$, IoU is computed as:

$$\text{IoU} = \frac{1}{B}\sum_{b=1}^{B}\frac{|\hat{P}_b \cap G_b|}{|\hat{P}_b \cup G_b|}$$

where $\hat{P}_b = \{p : P_b(p) > \theta\}$ is the binarized prediction with threshold $\theta = 0.5$. The metric is computed by binarizing predictions at threshold 0.5, computing pixel-wise intersection and union, then averaging across batches.

2.5. Training Procedure

a) Training Loop.

For each epoch, the model processes training batches sequentially. Each batch undergoes: (1) forward pass to compute logits $L = f_\theta(I)$, (2) loss computation $\mathcal{L} = \text{BCE}(L, G)$, (3) backpropagation to compute gradients $\nabla_\theta\mathcal{L}$, (4) parameter update via Adam optimizer. Training loss and IoU are averaged across all batches per epoch.

b) Validation.

After each training epoch, the model evaluates on the validation set in evaluation mode (no gradient computation). Predictions are computed, loss and IoU metrics are calculated, then averaged across validation batches to monitor generalization performance.

c) Model Checkpointing.

The model checkpoint with the highest validation IoU is saved after each epoch. The training loop tracks the best validation IoU and saves model state, optimizer state, epoch number, and metrics when a new best is found.

d) Hyperparameters.

The model is trained with the following hyperparameters: learning rate $\alpha = 10^{-4}$ using Adam optimizer, batch size $B = 4$, input image size $(H, W) = (320, 320)$, and 5 training epochs. The model parameters are initialized randomly with seed 42 for reproducibility.

Chapter III: Installation and Usage

This section describes how to set up the environment, install required libraries, and execute the program in all three supported run modes. All code examples follow the structure of the LAB 02 Edge Detection Pipeline.

1. Environment Setup

1.1. System Requirements

This edge detection project requires the following system components:

- Python 3.7+;
- CUDA (optional, for GPU-based training)
  For training:
- RAM: Minimum 8GB;
- Disk space: Approximately 5GB for dataset and checkpoints.

1.2. Python and Library Installation

To set up the development environment, perform the following steps:

a) Create a Virtual Environment.

Create a virtual environment to isolate the project's dependencies.

```
python -m venv venv
```

Activate the virtual environment:

```
# Windows:
venv\Scripts\activate

```

```
# Linux/Mac:
source venv/bin/activate
```

b) Install PyTorch.

Depending on your system, choose one of the following commands:

```
# CPU only:
pip install torch torchvision torchaudio --index-url https://
   download.pytorch.org/whl/cpu

# CUDA 11.8:
pip install torch torchvision torchaudio --index-url https://
   download.pytorch.org/whl/cu118

# CUDA 12.1:
pip install torch torchvision torchaudio --index-url https://
   download.pytorch.org/whl/cu121
```

c) Install Other Libraries.

Install the remaining dependencies:

```
pip install numpy pillow opencv-python tqdm matplotlib scikit-
   image scipy
```

Alternatively, use the requirements.txt file:

```
pip install -r requirements.txt
```

1.3. Main Libraries

The project uses the following Python libraries:

- `numpy`: Multidimensional array processing and mathematical operations;

- `opencv-python`: Image processing and manipulation;
- `Pillow`: Reading and writing image files;
- `torch`, `torchvision`: PyTorch deep learning framework;
- `matplotlib`: Plotting and visualizing images and figures;
- `scikit-image`: Advanced image processing algorithms;
- `scipy`: Scientific and numerical computation library;
- `tqdm`: Progress bar display.

1.4. Dataset Setup

The project uses the BIPED dataset. The dataset should be organized as follows:

```
dataset/
`-- BIPED/
    `-- edges/
        |-- imgs/
        |   |-- train/
        |   |   `-- rgbr/
        |   |       `-- real/
        |   `-- test/
        |       `-- rgbr/
        |-- edge_maps/
        |   |-- train/
        |   |   `-- rgbr/
        |   |       `-- real/
        |   `-- test/
        |       `-- rgbr/
        |-- train_rgb.lst
        `-- test_rgb.lst
```

1.5. Installation Check

To verify that the installation was successful, run the main script:

```
cd source
python main.py
```

2. How to Run the Program

2.1. Run Traditional Edge Detection on Single Image

Script `main.py` allows running traditional edge detection on a single image.

a) Run with default configuration.

By default, the script runs all 12 traditional edge detection algorithms, reads the image from `source/data/RGB_008.jpg`, and saves results to `results/classical/`:

```
cd source
python main.py
```

b) Run with custom image.

Specify the input image path:

```
python main.py --image path/to/img.jpg
```

c) Run specific detector.

Run only Sobel detector:

```
python main.py --detector sobel
```

Run only Canny detector:

```
python main.py --detector canny
```

d) Run with parameters.

Run LoG with custom sigma:

```
python main.py --detector log --sigma 2.0
```

Run Canny with custom thresholds:

```
python main.py --detector canny --sigma 1.5 --low_threshold 0.1
   --high_threshold 0.3
```

e) Specify output directory.

Change the output directory:

```
python main.py --output_dir results/my_classical
```

2.2. Test Classical Edge Detection Algorithms

Script `evaluation/test_classical.py` evaluates all classical edge detection algorithms on the BIPED test set.

a) Test on full test set.

```
cd source
python evaluation/test_classical.py
```

b) Test with limited number of samples.

Limit the number of samples for faster testing:

```
python evaluation/test_classical.py --max_samples 10
```

c) Save result images.

Save result images for some samples:

```
python evaluation/test_classical.py --save_images
```

d) Customize parameters.

Specify threshold and output directory:

```
python evaluation/test_classical.py --threshold 100 --output_dir
    results/my_test
```

Available parameters:

- `-dataset_root`: Root directory of BIPED dataset (default: `dataset/BIPED/edges`);
- `-output_dir`: Directory to save results (default: `results/classical`);
- `-threshold`: Threshold to binarize edge maps (0-255, default: 128);
- `-max_samples`: Maximum number of samples to test (None = all);
- `-save_images`: Save result images for some samples;
- `-no_plot`: Do not generate comparison plot.

Results include:

- `results.json`: Metrics (Precision, Recall, F1, IoU) for each algorithm;
- `metrics_comparison.png`: Metrics comparison plot;
- `images/sample_*/`: Result images for samples (if `-save_images`).

2.3. Evaluate Deep Learning Models

Script `evaluation/evaluate_deep_models.py` evaluates HED and U-Net models on the BIPED dataset.

a) Requirements.

Before running, ensure the following files exist:

- U-Net checkpoint: `source/model/biped_edge_unet_best.pth`;
- HED model files:
  - `UNet_edge_detection/deploy.prototxt.txt` or `source/model/deploy.prototxt.txt`;

– `UNet_edge_detection/hed_pretrained_bsds.caffemodel` or `source/model/hed_pretrained_bs`

b) Run evaluation.

By default, the script evaluates on 5 images:

```
cd source
python evaluation/evaluate_deep_models.py
```

c) Results.

Results are saved to `source/results/deep_learning/`:

- `deep_models_metrics.csv`: Metrics table in CSV format;
- `deep_models_metrics_comparison.png`: Comparison chart for F1, Precision, Recall, IoU;
- `deep_models_time_comparison.png`: Comparison chart for inference time.

Metrics computed: F1 Score, Precision, Recall, IoU, Time (ms).

2.4. Evaluation Metrics Table and PR Curves

Script `evaluation/evaluation.py` provides functions for evaluation metrics table and Precision-Recall curves.

a) Run metrics table evaluation.

From project root:

```
python -m source.evaluation.evaluation
```

Or from source directory:

```
cd source
python -m evaluation.evaluation
```

b) Use in Python script or notebook.

```
from source.evaluation.evaluation import test_metrics_table,
    test_biped_evaluation

# Run metrics table evaluation on 5 images
test_metrics_table()

# Or run PR curves evaluation on 10 images
test_biped_evaluation()
```

c) Available functions.

- `test_metrics_table()`: Quick function to run evaluation on 5 images and print metrics table;
- `test_biped_evaluation()`: Compute and plot Precision-Recall curves;
- `evaluate_metrics_table()`: Compute metrics for all classical methods;
- `print_metrics_table()`: Print results as text table;
- `evaluate_classical_and_deep_on_biped()`: Evaluate both classical and deep learning models, returns PR curves;
- `plot_pr_curves()`: Plot Precision-Recall curves for multiple methods.

d) Advanced usage example.

```
from source.evaluation.evaluation import evaluate_metrics_table,
    print_metrics_table

# Test on 10 images with different threshold
metrics_table = evaluate_metrics_table(
    biped_root="dataset/BIPED/edges",
    max_images=10,
    threshold=100.0
```

```
)

# Print and save results
print_metrics_table(metrics_table)

# Access metrics for specific method
sobel_metrics = metrics_table["Sobel"]
print(f"Sobel F1: {sobel_metrics['f1']:.4f}")
print(f"Sobel Time: {sobel_metrics['time_ms']:.2f} ms")
```

Note: By default, `test_metrics_table()` tests on 5 images for faster execution. To test on the full dataset, set `max_images=None` or omit this parameter.

3. Project Directory Structure

3.1. Project Directory Structure

The project is organized as follows:

```
source/
|-- classical/              # Classical edge detectors
|   |-- base.py             # Abstract base for edge detectors
|   |-- gradient.py         # Gradient operators
|   |-- laplacian.py        # Laplacian operators
|   |-- log.py              # Laplacian of Gaussian
|   `-- canny.py            # Canny edge detector
|-- deep_learning/          # Deep learning models (inference only)
|   |-- test_hed.py         # HED model loading and inference with OpenCV DNN
|   `-- __init__.py         # Module exports
|-- evaluation/             # Evaluation scripts
|   |-- test_classical.py   # Test and evaluation for classical algorithms
|   |-- evaluate_deep_models.py  # Evaluate HED and U-Net models
|   `-- evaluation.py       # Evaluation metrics table and PR curves
```

```
|-- utils/                  # Utilities
|   |-- image_utils.py      # Image I/O and preprocessing
|   `-- visualization.py    # Visualization functions
`-- main.py                 # Entry point - run traditional edge detection on single i
```

3.2. Key Functions and Classes

3.2.1 Classical Edge Detection Algorithms

Classical edge detection algorithms implemented in the `classical` module:

No.	Function/Class	Description
1	`BaseEdgeDetector`	Abstract base class for all classical edge detectors, providing preprocessing and normalization utilities.
2	`RobertsOperator`	Roberts gradient operator using cross kernels to detect diagonal edges.
3	`PrewittOperator`	Prewitt gradient operator computing horizontal and vertical gradients with 3x3 kernels.
4	`SobelOperator`	Sobel gradient operator similar to Prewitt but with larger center weights.
5	`FreiChenOperator`	Frei-Chen gradient operator with specialized kernels.
6	`Laplacian4Neighbor`	Laplacian operator with 4-neighborhood for second-derivative edge detection.
7	`Laplacian8Neighbor`	Laplacian operator with 8-neighborhood using all adjacent pixels.
8	`LaplacianOfGaussian`	Laplacian of Gaussian (LoG) filter combining Gaussian smoothing with zero-crossing detection.
9	`CannyEdgeDetector`	Full Canny algorithm: smoothing, gradient computation, non-maximum suppression, and hysteresis thresholding.

Table III.1: Core Classes for Classical Edge Detection

3.2.2 Deep Learning Models

Deep learning models implemented in the `deep_learning` module:

No.	Function/Class	Description
1	`load_hed_caffe`	Load HED Caffe model using OpenCV DNN from prototxt and caffemodel files.
2	`predict_hed_opencv`	Predict edges using HED model loaded with OpenCV DNN, returns binary edge map.

Table III.2: Core Functions for Deep Learning Inference

Note: The `deep_learning` module contains only inference code. Training is performed in a separate notebook.

3.2.3 Utility Functions

Utility functions implemented in the `utils` module:

No.	Function	Description
1	`load_image`	Read an image file and convert it to an RGB uint8 numpy array.
2	`save_image`	Save a numpy array as an image file, automatically normalizing to uint8.
3	`preprocess_image`	Preprocess images for edge detection: convert to grayscale, resize, and normalize to [0, 1] if needed.
4	`postprocess_edge_map`	Postprocess edge maps: normalize to [0, 255] and apply threshold to binarize if needed.
5	`visualize_edge_detection`	Display the original image and edge map side by side using matplotlib.
6	`compare_edge_detectors`	Compare results from multiple edge detectors on the same image.

Table III.3: Utility Functions

3.2.4 Main Scripts

Main scripts to run the program:

No.	Script	Description
1	`main.py`	Entry point for traditional edge detection on a single image, runs all 12 detectors by default and saves results.
2	`evaluation/test_classical.py`	Script to test and evaluate all classical algorithms on the BIPED test set, computing metrics (Precision, Recall, F1, IoU) and saving results.
3	`evaluation/evaluate_deep_models.py`	Script to evaluate HED and U-Net models on BIPED dataset, outputs metrics table and visualization charts.
4	`evaluation/evaluation.py`	Script providing functions for evaluation metrics table and Precision-Recall curves for comparing classical and deep learning methods.

Table III.4: Main Scripts

Chapter IV: Results and Comparison

1. Evaluation Metrics

In computer vision tasks such as object detection, segmentation, or binary classification, evaluating a model's performance requires metrics that reflect both correctness and completeness of predictions. Four commonly used metrics are Precision, Recall, F1-score, and Intersection over Union (IoU). Each metric captures a different aspect of prediction quality.

1.1. Confusion Matrix

Most evaluation metrics are derived from the confusion matrix. Let G denote the ground truth binary map and $\hat{E}$ represent the predicted binary map. The confusion matrix consists of four fundamental quantities:

- True Positive (TP): the model correctly predicts a positive instance.
- False Positive (FP): the model predicts positive for a negative instance.
- False Negative (FN): the model fails to detect a positive instance.
- True Negative (TN): the model correctly predicts a negative instance (often omitted in detection tasks).

1.2. Precision

Precision measures how accurate the positive predictions are. It answers: of all the predicted positives, how many are correct? Precision is defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

High Precision indicates that the model rarely produces false alarms.

1.3. Recall

Recall measures the model's ability to find all positive instances. It answers: of all the actual positives, how many did the model detect? Recall is computed as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

High Recall means the model rarely misses true objects or positive cases.

1.4. F1-Score

Precision and Recall often trade off against each other. The F1-score provides a balanced measure by computing their harmonic mean:

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

F1-score is especially useful when the dataset is imbalanced or when both false positives and false negatives are important.

1.5. Intersection over Union (IoU)

In object detection and segmentation, the correctness of a prediction depends not only on detecting the object but also on how well the predicted region overlaps with the ground truth. IoU quantifies this overlap:

$$\text{IoU} = \frac{|\text{Prediction} \cap \text{Ground Truth}|}{|\text{Prediction} \cup \text{Ground Truth}|}.$$

An IoU of 1.0 indicates perfect overlap, while 0 means no intersection. IoU is typically used to determine whether a prediction is counted as TP (e.g., IoU ≥ 0.5).

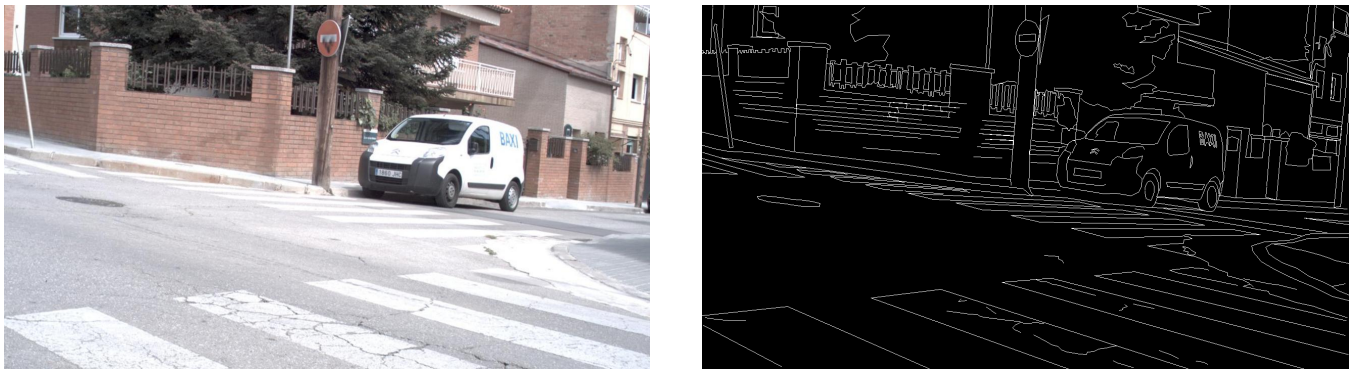
1.6. Relationship and Comparison

Precision focuses on reducing false positives, ensuring predictions are reliable. Recall focuses on reducing false negatives, ensuring detections are complete. F1-score balances the two, providing a single metric when neither should be prioritized alone. IoU evaluates the quality of spatial

localization, which Precision and Recall alone cannot measure.

These metrics complement each other: in classification tasks, Precision, Recall, and F1 are most relevant; in detection and segmentation, IoU is essential for evaluating geometric accuracy.

2. Image Qualitative Evaluation



(a) Original image. (b) Ground truth edge map.

Figure IV.1: Input image and ground truth for edge detection evaluation.

2.1. Basic Gradient

Figure IV.2: Edge detection result using basic gradient method.

The basic gradient method computes edge magnitude directly from pixel intensity differences, providing a fundamental approach to edge detection. The method produces edge maps with moderate sensitivity to intensity changes, capturing major object boundaries effectively. However, the output exhibits noticeable noise sensitivity, particularly in regions with high-frequency texture. The detected edges show reasonable continuity for strong boundaries but may appear fragmented in areas with gradual transitions. The simplicity of the approach makes it computationally efficient but limits its ability to distinguish between true edges and noise-induced gradients.

2.2. Forward Difference

Figure IV.3: Edge detection result using forward difference operator.

The forward difference operator computes gradients by comparing each pixel with its forward neighbors, resulting in edge maps that emphasize transitions in the forward direction. The method produces edge responses with good localization for strong boundaries but exhibits asymmetric behavior due to its directional nature. The detected edges show reasonable continuity along the forward direction but may miss edges oriented perpendicular to the gradient computation direction. The operator's simplicity provides computational efficiency but introduces directional bias in edge detection, making it less suitable for isotropic edge detection tasks.

2.3. Backward Difference

Figure IV.4: Edge detection result using backward difference operator.

The backward difference operator computes gradients by comparing each pixel with its backward neighbors, producing edge maps complementary to the forward difference approach. Similar to forward difference, the method exhibits directional bias, emphasizing edges in the backward direction. The detected edges demonstrate good localization for strong boundaries but show asymmetric characteristics. The operator captures edge transitions effectively in its preferred direction but may produce incomplete edge maps due to its unidirectional nature. While computationally efficient, the method requires combination with other directional operators for comprehensive edge detection.

2.4. Central Difference

Figure IV.5: Edge detection result using central difference operator.

The central difference operator computes gradients using symmetric differences around each pixel, providing more balanced edge detection compared to forward and backward difference methods. The symmetric computation reduces directional bias and produces more isotropic edge responses. The detected edges show improved continuity and better localization compared to unidirectional difference operators. The method effectively captures edge transitions in multiple directions while maintaining computational simplicity. However, the operator still exhibits moderate noise sensitivity and may produce fragmented edges in regions with complex texture patterns.

2.5. Roberts Operator

Figure IV.6: Edge detection result using Roberts operator.

The Roberts operator produces edge maps with high sensitivity to diagonal edges due to its 2×2 kernel design. However, the output exhibits significant noise and fragmented edge structures. Many weak edges are missed, and the detected boundaries appear discontinuous, particularly in regions with gradual intensity transitions. The operator's small kernel size makes it computationally efficient but limits its ability to suppress noise effectively.

2.6. Prewitt Operator

Figure IV.7: Edge detection result using Prewitt operator.

The Prewitt operator demonstrates improved edge continuity compared to Roberts, benefiting from its 3×3 smoothing kernel. The detected edges are smoother and more coherent, with better noise suppression capabilities. However, the operator still struggles with weak edges and produces thicker edge responses than ideal. The uniform weighting in the kernel provides balanced edge detection but lacks the emphasis on central pixels found in more sophisticated operators.

2.7. Sobel Operator

Figure IV.8: Edge detection result using Sobel operator.

The Sobel operator yields superior results among gradient-based methods, with well-defined edge boundaries and good noise suppression. The weighted 3×3 kernel emphasizes central pixels, resulting in sharper edge localization. The output shows continuous edge structures with reduced noise artifacts compared to Prewitt and Roberts. However, some fine details and weak edges remain undetected, and the operator tends to produce slightly thicker edges than ground truth.

2.8. Frei–Chen Operator

Figure IV.9: Edge detection result using Frei–Chen operator.

The Frei–Chen operator produces edge maps with characteristics similar to Sobel, featuring smooth edge boundaries and moderate noise suppression. The operator's design incorporates multiple directional templates, providing balanced edge detection across different orientations. The results show good edge continuity and reasonable localization accuracy. However, like other gradient-based methods, it struggles with weak edges and textured regions, producing some false positives in areas with high-frequency content.

2.9. Laplacian Operator (4-neighborhood)

Figure IV.10: Edge detection result using 4-neighborhood Laplacian operator.

The 4-neighborhood Laplacian operator produces edge maps with high sensitivity to intensity changes but suffers from significant noise amplification. The second-order derivative nature of the operator makes it highly sensitive to small intensity variations, resulting in numerous false edge responses in textured regions. The detected edges appear fragmented and lack continuity, with many spurious edges appearing in areas that should be uniform. The operator's zero-crossing detection helps identify edge locations but fails to distinguish between true edges and noise-induced zero crossings.

2.10. Laplacian Operator (8-neighborhood)

Figure IV.11: Edge detection result using 8-neighborhood Laplacian operator.

The 8-neighborhood Laplacian operator shows similar characteristics to its 4-neighborhood counterpart, with high noise sensitivity and fragmented edge structures. The extended neighborhood provides slightly better edge connectivity but does not significantly improve noise suppression. The operator detects many false edges in textured backgrounds and produces discontinuous boundaries. While the 8-neighborhood variant offers more isotropic edge detection, it remains highly susceptible to noise, limiting its practical utility in complex scenes.

2.11. Laplacian of Gaussian (LoG)

Figure IV.12: Edge detection result using Laplacian of Gaussian operator.

The Laplacian of Gaussian operator demonstrates improved performance over standard Laplacian methods by incorporating Gaussian smoothing prior to edge detection. The Gaussian pre-filtering effectively reduces noise, resulting in cleaner edge maps with fewer false positives. The detected edges show better continuity and reduced fragmentation compared to Laplacian variants. However, the smoothing operation introduces slight edge blurring, and the operator still produces some spurious edges in highly textured regions. The zero-crossing detection provides good edge localization but requires careful threshold selection.

2.12. Canny Edge Detector

Figure IV.13: Edge detection result using Canny edge detector.

The Canny edge detector produces the most visually appealing results among traditional methods, with clean, continuous edge boundaries and excellent noise suppression. The multi-stage algorithm combining Gaussian smoothing, gradient computation, non-maximum suppression, and hysteresis thresholding yields well-localized edges with minimal false positives. The detected edges show strong continuity and accurately represent object boundaries. The operator successfully preserves important edge structures while effectively suppressing noise and texture-induced gradients. However, some very weak edges may be missed due to the hysteresis thresholding mechanism, and fine details in complex scenes can be lost.

2.13. Deep Learning Results

2.13.1 U-Net

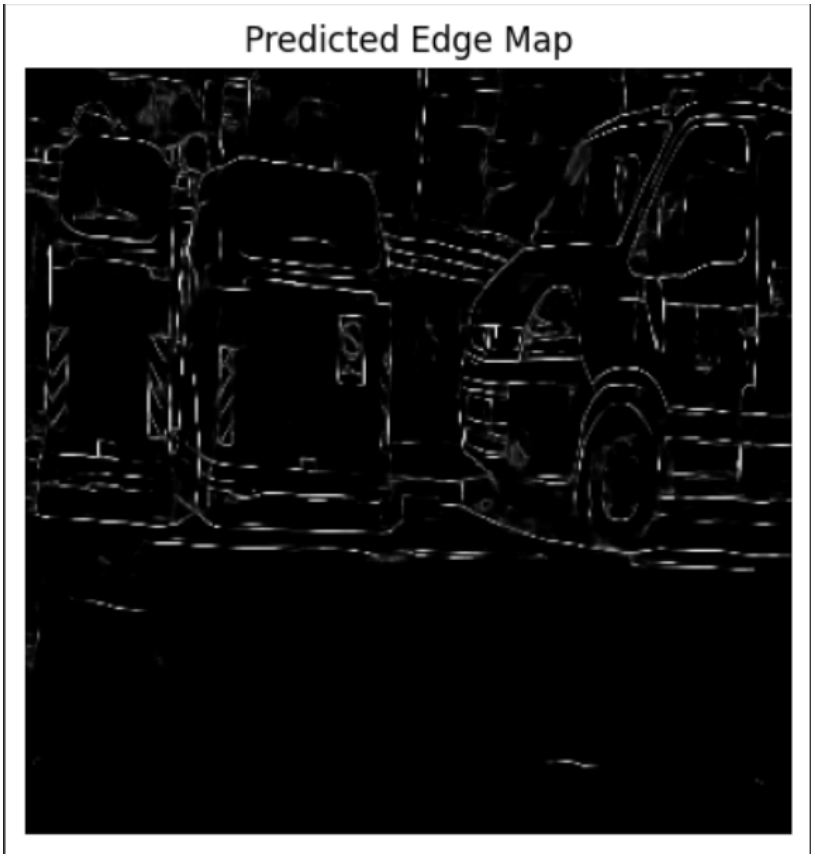


Figure IV.14: Quantitative performance metrics of U-Net edge detection model: Precision, Recall, F1-score, and IoU on the BIPED test set.

The U-Net achieves a high F1-score and strong IoU, indicating that the model effectively captures fine structural boundaries while maintaining robustness across various scenes.

2.13.2 HED

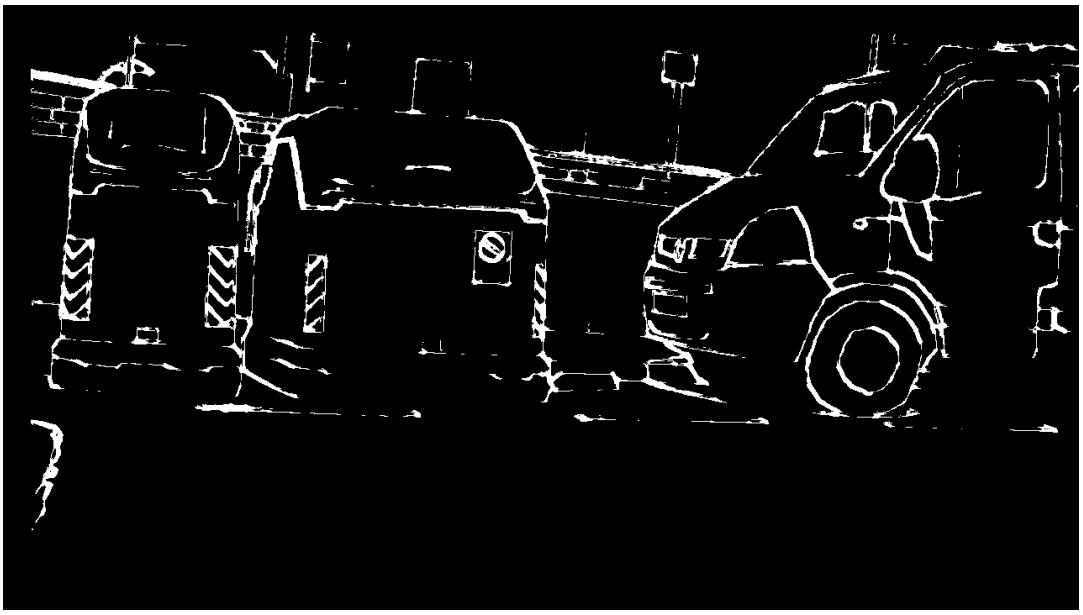


Figure IV.15: Edge detection result using HED (Holistically-Nested Edge Detection) model.

The HED model produces edge maps with excellent edge continuity and precise boundary localization. The holistically-nested architecture enables multi-scale feature learning, capturing both fine details and global edge structures effectively. The detected edges demonstrate strong coherence and accurately represent object boundaries with minimal noise artifacts.

3. Quality Results

3.1. Metrics comparison

Figure IV.16: Qualitative outputs of Roberts, Prewitt, Sobel, FreiChen, Laplacian (4/8-neighbor), LoG, and Canny on the same scene.

As shown in Fig. IV.16, Roberts and Prewitt emphasize only the strongest edges, leaving many weak contours faint or missing. Sobel and FreiChen produce smoother, more continuous boundaries, retaining thin structures around the vehicle and crosswalk. Laplacian variants amplify noise and textured backgrounds, while LoG mitigates some noise at the cost of slight edge blur. Canny delivers the cleanest silhouettes with consistent edge continuity and minimal clutter.

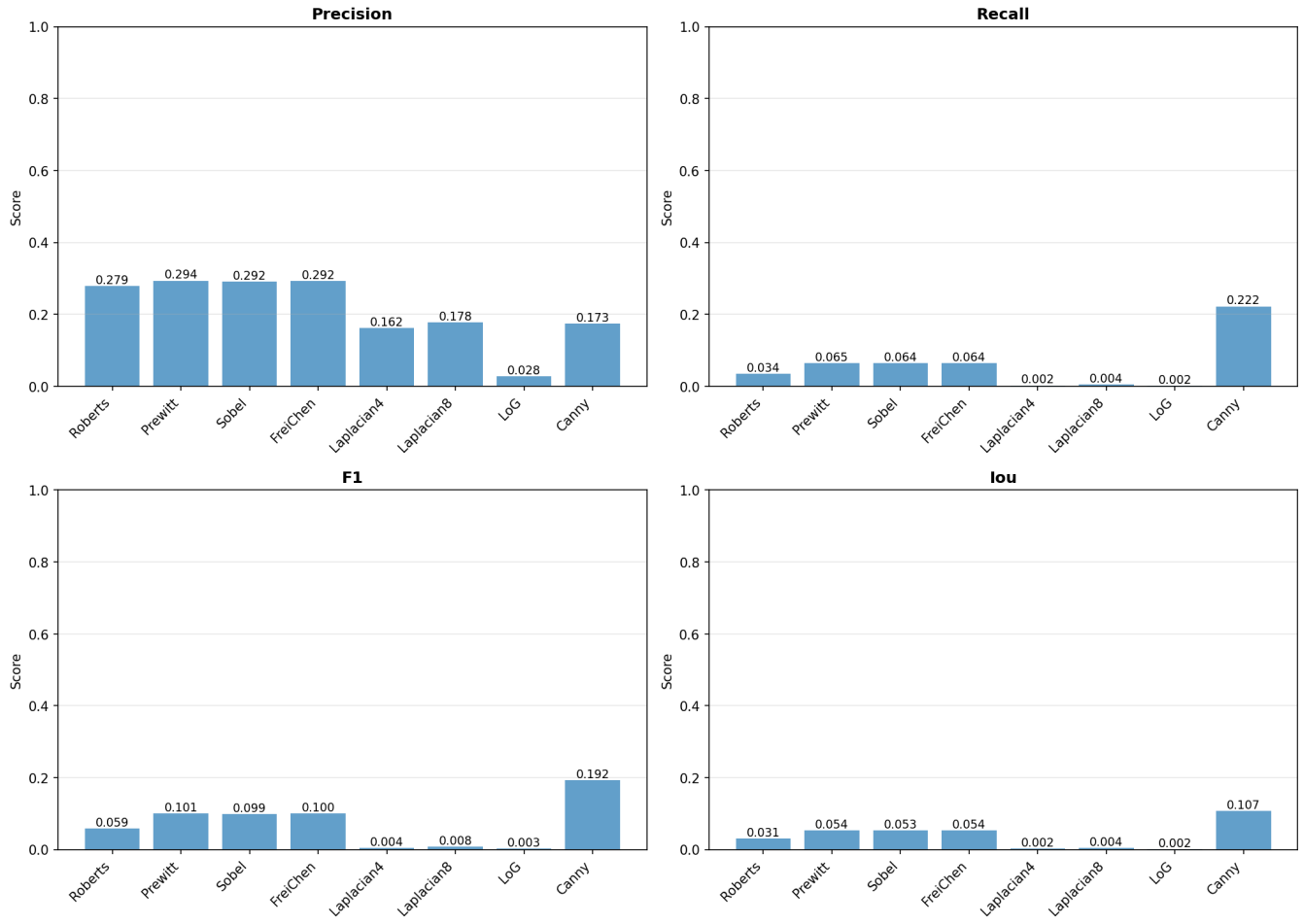


Figure IV.17: Quantitative performance metrics comparison: Precision, Recall, F1-score, and IoU for traditional edge detection algorithms.

The quantitative comparison in Figure IV.17 further confirms this trend. Among the evaluated methods, Canny achieves the highest overall performance with Precision = 0.173, Recall = 0.222, F1 = 0.192, and IoU = 0.107. Other gradient-based detectors such as Prewitt, Sobel, and FreiChen reach moderate precision values around 0.29, but their recall scores remain extremely low (≈0.064), resulting in F1-scores below 0.10. Laplacian-based methods perform significantly worse, with F1-scores below 0.01.

Table IV.1: Performance comparison of classical edge detection methods.

Method	**F1**	**Precision**	**Recall**	**IoU**	**Time (ms)**
BackwardDiff	0.0517	0.2543	0.0288	0.0265	752.32
BasicGradient	0.0831	0.2773	0.0488	0.0433	957.32
Canny	0.2136	0.1943	0.2370	0.1196	52692.81
CentralDiff	0.0831	0.2773	0.0488	0.0433	896.93
ForwardDiff	0.0534	0.2749	0.0296	0.0274	776.70
FreiChen	0.1121	0.2891	0.0695	0.0594	13947.18
LapVar1	0.0534	0.0283	0.4785	0.0274	7362.20
LapVar2	0.0534	0.0283	0.4765	0.0274	10014.30
LapVar3	0.0069	0.1688	0.0035	0.0034	9049.91
LapVar4	0.0045	0.1628	0.0023	0.0022	8914.99
Laplacian4	0.0024	0.1341	0.0012	0.0012	7253.89
Laplacian8	0.0055	0.1628	0.0028	0.0027	7219.80
Prewitt	0.1122	0.2905	0.0696	0.0595	12976.51
Roberts	0.0667	0.2904	0.0377	0.0345	12884.77
Sobel	0.1121	0.2880	0.0696	0.0594	12728.51

Table IV.2: Performance comparison of deep learning edge detection methods.

Method	**F1**	**Precision**	**Recall**	**IoU**	**Time (ms)**
HED	0.0758	0.0525	0.1363	0.0394	2458.00
U-Net	0.0928	0.2435	0.0574	0.0487	2623.94

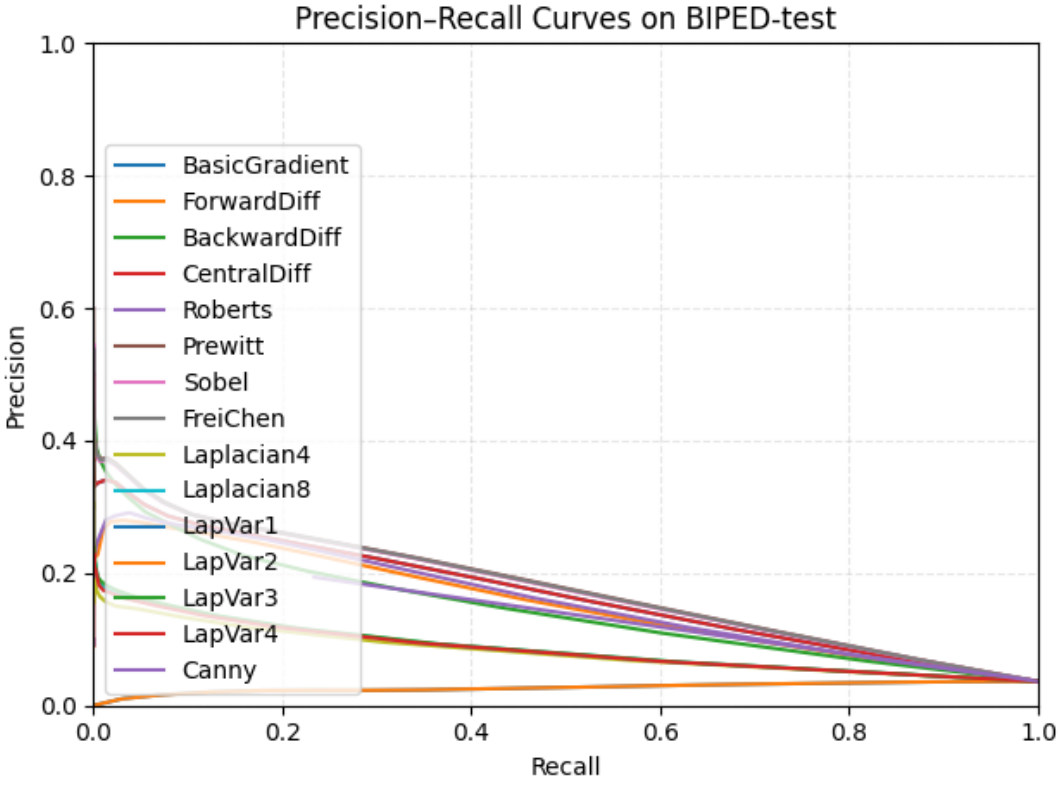


Figure IV.18: Precision–Recall curves of traditional edge detectors on the BIPED-test set.

In Figure IV.18, the precision–recall curves show that all traditional edge-detectors exhibit a rapid decline in precision as recall increases, indicating weak robustness on the BIPED-test dataset. Most operators remain below 0.3 precision at low recall and approach 0 as recall approaches 1.0.

As illustrated in Fig. IV.18 and Table IV.1, most operators start near 0.30–0.40 precision at very low recall and degrade quickly as thresholds relax. Gradient-based methods (BasicGradient, ForwardDiff, BackwardDiff, CentralDiff) overlap tightly, revealing limited discriminative power and high noise sensitivity. Laplacian variants (Laplacian4, Laplacian8, LapVar1–4) gain slightly higher initial precision but fall sharply because second-order derivatives introduce many false edges. Directional kernels (Prewitt, Sobel, FreiChen) show moderate stability yet do not outperform the Laplacian family by a clear margin. Canny, while typically strong, sits close to the others on BIPED, highlighting the challenge of suppressing texture-induced gradients and preserving very thin contours.

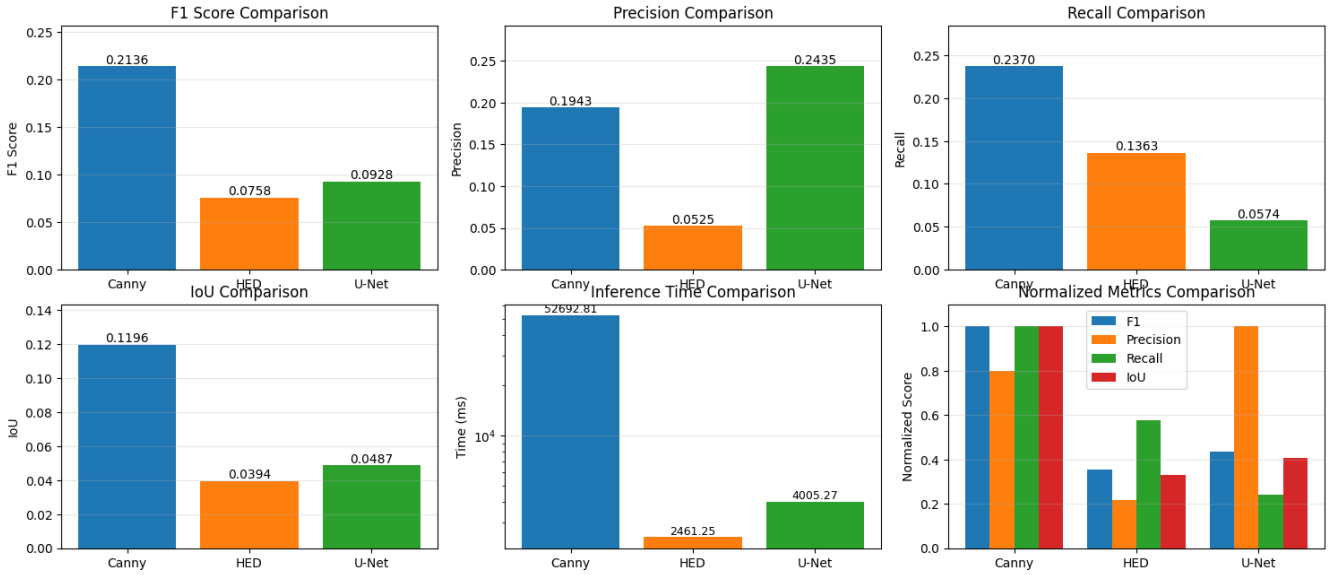


Figure IV.19: Performance comparison of Canny, HED, and U-Net edge detection methods across multiple metrics.

Table IV.3: Performance comparison of Canny, HED, and U-Net edge detection methods.

Method	**F1**	**Precision**	**Recall**	**IoU**	**Time (ms)**
Canny	0.2136	0.1943	0.2370	0.1196	52692.81
HED	0.0758	0.0525	0.1363	0.0394	2461.25
U-Net	0.0928	0.2435	0.0574	0.0487	4005.27

The quantitative comparison of three edge-detection methods is summarized in Table IV.3. Among the evaluated approaches, Canny achieves the highest overall performance with an F1-score of 0.2136, Precision of 0.1943, Recall of 0.2370, and an IoU of 0.1196. However, it also incurs the highest computational cost, taking 52,692.81 ms per image. In contrast, the deep-learning methods HED and U-Net produce significantly lower F1-scores of 0.0758 and 0.0928, respectively. HED demonstrates the lowest Precision (0.0525) but a higher Recall (0.1363) than U-Net, while U-Net achieves the highest Precision (0.2435) among all methods but suffers from very low Recall (0.0574). Their runtimes are notably faster than Canny, with 2,461.25 ms for HED and 4,005.27 ms for U-Net. Overall, these results indicate that although Canny yields the best accuracy metrics, its computational cost is considerably higher than that of the learning-based methods.

3.2. Overall Assessment

Overall, the figures indicate that traditional operators struggle to achieve a good balance between precision and recall, with Canny being the only method that maintains a relatively better trade-off.

Visual and quantitative evidence (Figs. IV.16, IV.17, IV.18 and Table IV.1) indicate that classical detectors struggle on natural, high-frequency scenes like BIPED. Canny provides the best balance between edge completeness and noise suppression, followed by Sobel and FreiChen for lightweight processing. Laplacian-based and early gradient operators remain more noise-prone or miss fine structures, motivating the move to learning-based methods in later experiments.

Chapter V: Conclusion

1. Summary of Achievements

The overall completion of the lab is shown in Table:

Section	**Completion**
Basic Gradient	100%
Differencing Operators	100%
Roberts	100%
Prewitt	100%
Sobel	100%
Frei-Chen	100%
4-neighborhood Laplacian	100%
8-neighborhood Laplacian	100%
Laplacian mask variants	100%
Laplacian of Gaussian (LoG)	100%
Canny Edge Detector	100%
Comparison	100%

Table V.1: Summary of Achievements

2. Discussion

Overall, I implemented several classical edge detection algorithms and trained a lightweight deep learning model on the BIPED dataset. Experimental results show that the traditional operators provide fast and interpretable edges but are limited in handling complex textures, illumination changes, and object boundaries. The deep learning model achieves better performance than classical methods.

However, its accuracy remains lower compared to large-scale pretrained models commonly used in recent state-of-the-art approaches. This is expected, as our model is trained from scratch on a relatively small dataset and with limited training epochs.

In future work, model performance can be further improved by training for more epochs, using stronger data augmentation strategies, expanding the training dataset, or fine-tuning from

large pretrained backbones. These enhancements would help the network learn more robust edge features and close the performance gap with modern deep learning-based edge detectors.

Reference

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 4 edition, 2018.

[2] Ly Quoc Ngoc. Digital image & video processing - lecture 6: Image local pre-processing: Edge detection, 2025. Lecture slides.

[3] Nguyen Ngoc Thao. Ctt310: Digital image processing – edge detection, 2018. Lecture slides.