Pattern-Based Object Detection using Multiple Template Matching Approaches and CNN Validation

Introduction

This notebook presents a comprehensive evaluation and comparison of different **pattern-based object detection** methods with a focus on **L-bend shaped objects**. The project explores multiple template matching approaches to identify the most effective method for a Region Proposal Network (RPN), ultimately combining the best-performing classical computer vision technique with modern deep learning validation.

Project Evolution and Methodology

Experimental Phase: FFT Investigation

Initially, **FFT-based template matching** was explored as a potential approach:

- High-pass filtering in the frequency domain to enhance edge features
- Normalized dot product (cosine similarity) for pattern matching
- **Result**: While theoretically sound, FFT filtering did not provide reliable performance for the RPN requirements

Production Methods: Classical + Modern Hybrid

After experimentation, the final pipeline uses proven, robust approaches:

1. **OpenCV Template Matching** (Primary RPN Method)
    - Standard `cv2.matchTemplate` with `TM_CCOEFF_NORMED`
    - Multiple rotation angles for orientation independence
    - Reliable, well-tested computer vision approach
2. **PyTorch Batched Convolution** (High-Performance Alternative)
    - GPU-accelerated template matching using batched convolution
    - Single-line convolution for all rotated templates simultaneously
    - Optimized for real-time performance when GPU is available
3. **CNN Validation & Localization** (Final Stage)
    - Custom CNN architecture for L-bend object validation
    - Dual-head network: classification confidence + precise centroid regression
    - Filters false positives from template matching stage

Final Architecture

```
Input Image → Template Matching → Candidate Regions → CNN
Validation → Final Detections
     ↓              ↓                    ↓                ↓
↓
  Raw Frame    OpenCV/PyTorch       Rotation-Invariant   Confidence
+    Verified
               Template Match       Bounding Boxes        Centroid
L-bends
```

Key Design Decisions

- **FFT Approach Rejected**: Despite theoretical advantages, FFT-based filtering proved unreliable for practical RPN applications
- **OpenCV as Primary**: Chose battle-tested `cv2.matchTemplate` for consistent, predictable results
- **PyTorch for Speed**: Batched convolution approach when GPU acceleration is needed
- **CNN Validation**: Added deep learning stage to reduce false positives and provide precise localization

Performance Characteristics

- **Reliability**: OpenCV template matching provides consistent, repeatable results
- **Speed**: PyTorch batched approach achieves >20 FPS on GPU
- **Accuracy**: CNN validation stage significantly reduces false positive rates
- **Robustness**: Multi-angle template matching handles arbitrary object orientations

Implementation Notes

The notebook demonstrates:

1. **FFT exploration** (for educational/research purposes)
2. **OpenCV implementation** (production-ready RPN)
3. **PyTorch acceleration** (high-performance alternative)
4. **CNN training pipeline** (validation and localization)
5. **Real-time inference** (complete deployment solution)
6. **Model export** (TorchScript and ONNX for deployment)

This comprehensive approach ensures both research completeness and practical deployment readiness, with clear documentation of why certain methods were chosen over others.

In [ ]:
```python
import numpy as np
# This is a simple NMS implementation for averaging bounding boxes based on Interse
def average_boxes(boxes, iou_thresh=0.5):
    if len(boxes) == 0:
        return []

    boxes = np.array(boxes)
    keep = []

    while len(boxes) > 0:
        ref_box = boxes[0]
        rest = boxes[1:]

        x1, y1, x2, y2, score = ref_box
        ious = []

        for box in rest:
            xx1 = max(x1, box[0])
            yy1 = max(y1, box[1])
            xx2 = min(x2, box[2])
            yy2 = min(y2, box[3])
            inter = max(0, xx2 - xx1) * max(0, yy2 - yy1)
            union = (x2 - x1) * (y2 - y1) + (box[2] - box[0]) * (box[3] - box[1]) -
            iou = inter / union if union > 0 else 0
            ious.append(iou)

        ious = np.array([1.0] + ious)
        cluster = boxes[np.where(ious >= iou_thresh)[0]]

        avg_box = np.average(cluster, axis=0, weights=cluster[:, 4])  # Weighted by
        keep.append(avg_box)

        boxes = np.delete(boxes, np.where(ious >= iou_thresh)[0], axis=0)

    return keep
```

Using **FFT Filters** for patern matching

This section demonstrates how to use FFT-based high-pass filtering and template matching for robust pattern detection, including rotation invariance.

Workflow Overview

1. **High-Pass Filtering with FFT**
    - The `high_pass_filter_fft` function applies a high-pass filter in the frequency domain to both the input image and the template. This enhances edges and suppresses low-frequency background, making the matching more robust to illumination changes.
2. **Template Rotation**

- The `rotate_image_with_padding` function rotates the template to multiple angles, ensuring the rotated template fits within the new image bounds without cropping. This enables rotation-invariant detection.

3. **Template Matching via Dot Product**
   - The `convolve_and_get_bboxes` function slides each rotated template over the filtered image, computing the normalized dot product (cosine similarity) at each location. Locations with scores above a threshold are considered detections.
4. **Non-Maximum Suppression (NMS)**
   - The `non_max_suppression_fast` function removes overlapping detections, keeping only the highest-scoring bounding boxes.
   - The `average_boxes` function (defined elsewhere) further merges boxes with high overlap to produce robust final detections.
5. **Visualization**
   - Detected bounding boxes and their scores are drawn on the image.
   - The result is displayed and saved as `output.png`.

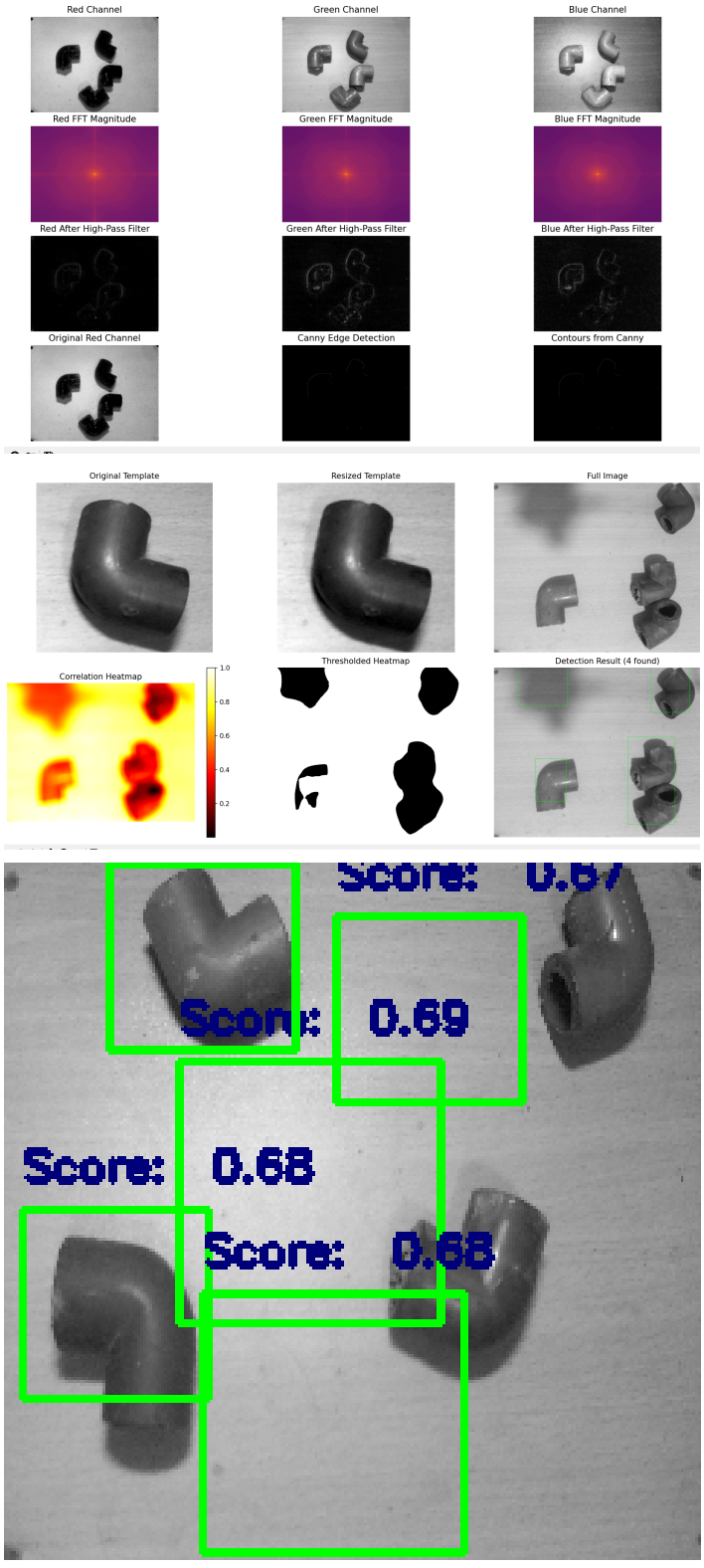
Advantages

- **Edge/Pattern Focus:** High-pass filtering makes the method robust to lighting and background variations.
- **Rotation Invariance:** By matching multiple rotated templates, the method detects objects regardless of their orientation.
- **Customizable:** Thresholds and rotation angles can be tuned for different applications.

Usage Notes

- Adjust the `threshold` in `convolve_and_get_bboxes` and the `iou_thresh` in NMS for your dataset.
- Ensure the template and image are preprocessed (resized, filtered) consistently.
- This method is suitable for detecting objects with strong edge or pattern features, especially when orientation varies.

Example Results



Detected bounding boxes are shown in green. The images demonstrate the effect of FFT-based filtering and successful detection of rotated templates using FFT-based high-pass filtering and template matching.

In [ ]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import time
```

```python
# --- Step 1: High-pass filter and FFT helpers ---

def high_pass_filter_fft(image):

    # Step 1: Forward FFT
    f = np.fft.fft2(image)
    fshift = np.fft.fftshift(f)

    # Step 2: Create High-Pass Mask
    rows, cols = image.shape
    crow, ccol = rows // 2, cols // 2

    radius = min(rows, cols) // 10  # Adjust radius based on image size
    radius = 8
    mask = np.ones((rows, cols), np.uint8)
    mask[crow - radius:crow + radius, ccol - radius:ccol + radius] = 0  # Suppress

    # Step 3: Apply mask and inverse FFT
    fshift_filtered = fshift * mask
    f_ishift = np.fft.ifftshift(fshift_filtered)
    img_back = np.fft.ifft2(f_ishift)
    img_back = np.abs(img_back)

    # Step 4: Normalize for consistent scaling (0 to 1)
    img_back = (img_back - np.min(img_back)) / (np.max(img_back) - np.min(img_back)

    return img_back

# --- Step 2: Rotate template with padding ---
def rotate_image_with_padding(image, angle):
    h, w = image.shape
    center = (w // 2, h // 2)
    rot_mat = cv2.getRotationMatrix2D(center, angle, 1.0)
    cos = np.abs(rot_mat[0, 0])
    sin = np.abs(rot_mat[0, 1])
    nW = int((h * sin) + (w * cos))
    nH = int((h * cos) + (w * sin))

    rot_mat[0, 2] += (nW / 2) - center[0]
    rot_mat[1, 2] += (nH / 2) - center[1]

    return cv2.warpAffine(image, rot_mat, (nW, nH), borderValue=255)

# --- Step 3: Convolution with dot product ---
def convolve_and_get_bboxes(image, template, threshold):
    h, w = template.shape
    ih, iw = image.shape
    bboxes = []

    stride_y = max(1, h // 20)
    stride_x = max(1, w // 20)

    for y in range(0, ih - h + 1, stride_y):
        for x in range(0, iw - w + 1, stride_x):
            patch = image[y:y+h, x:x+w]
```

```
            dot = np.dot(patch.flatten(), template.flatten())
            norm = np.linalg.norm(patch) * np.linalg.norm(template)
            score = dot / (norm + 1e-6)
            print(f"Score at ({x}, {y}): {score:.4f} dot {dot} and norm {norm}")  #
            if score >= threshold:
                bboxes.append((x, y, x + w, y + h, score))
    return bboxes

# --- Step 4: Non-Maximum Suppression ---
def non_max_suppression_fast(boxes, iou_thresh=0.3):
    if len(boxes) == 0:
        return []

    boxes = np.array(boxes)
    x1 = boxes[:,0]; y1 = boxes[:,1]; x2 = boxes[:,2]; y2 = boxes[:,3]; scores = bo
    areas = (x2 - x1) * (y2 - y1)
    order = scores.argsort()[::-1]

    keep = []
    while order.size > 0:
        i = order[0]
        keep.append(tuple(boxes[i]))

        xx1 = np.maximum(x1[i], x1[order[1:]])
        yy1 = np.maximum(y1[i], y1[order[1:]])
        xx2 = np.minimum(x2[i], x2[order[1:]])
        yy2 = np.minimum(y2[i], y2[order[1:]])

        w = np.maximum(0.0, xx2 - xx1)
        h = np.maximum(0.0, yy2 - yy1)
        inter = w * h
        iou = inter / (areas[i] + areas[order[1:]] - inter)

        order = order[1:][iou <= iou_thresh]

    return keep

# --- Step 5: Main Pipeline ---
def main():

    # Load and resize full image
    img = cv2.imread('Dataset\\test\\images\\image_20250321_200236_lbent_3.png', cv
    img_resized = cv2.resize(img, (240, 240))
    img_filtered = high_pass_filter_fft(img_resized)
    #cv2.imshow("img",img_filtered)

    # Load and resize template
    template = cv2.imread("image_20250321_201450_lbent_1.png", cv2.IMREAD_GRAYSCALE
    #template_filteredr= high_pass_filter_fft(template)
    template_resized = cv2.resize(template, (64, 64))
    template_filtered = high_pass_filter_fft(template_resized)
    #cv2.imshow("temp",template_filtered)
    # Generate rotated templates
    angles = [0,45,90,135, 180,225, 270,315]
    rotated_templates = [rotate_image_with_padding(template_filtered, a) for a in a
    start = time.time()
```

```
    # Collect bboxes from all rotations
    all_bboxes = []
    for temp in rotated_templates:
        bboxes = convolve_and_get_bboxes(img_filtered, temp, threshold=0.67)
        all_bboxes.extend(bboxes)

    # Apply NMS
    final_bboxes = non_max_suppression_fast(all_bboxes, iou_thresh=0.3)
    final_bboxes = average_boxes(final_bboxes, iou_thresh=0.1)

    end = time.time()
    # Draw and show
    out = cv2.cvtColor(img_resized, cv2.COLOR_GRAY2BGR)
    for (x1, y1, x2, y2, score) in final_bboxes:
        cv2.rectangle(out, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0), 2)
        cv2.putText(out, f"Score: {score: .2f}", (int(x1), int(y1) - 10), cv2.FONT_



    print(f"time = {end - start:.2f} seconds")
    print(f"Number of detections: {len(final_bboxes)}")

    cv2.imshow("Detections", out)
    cv2.imwrite("output.png", out)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

Using **Opencv TemplateMatching** method

This section demonstrates how to perform rotation-invariant template matching using OpenCV's `matchTemplate` method. The workflow includes rotating the template to multiple angles, matching each rotated template to the input image, and applying non-maximum suppression (NMS) to filter overlapping detections. The final bounding boxes are averaged for robustness.

Key Steps

1. **Image and Template Preparation**
    - Load the grayscale input image and template.
    - Resize both for consistent processing.
2. **Template Rotation**
    - The `rotate_image_with_padding` function rotates the template by a given angle, ensuring the rotated template fits within the new image bounds without cropping.

3. **Template Matching**
    - For each rotation angle, the template is matched to the image using `cv2.matchTemplate` with the `TM_CCOEFF_NORMED` method.
    - Detections above a threshold are collected as bounding boxes with scores.
4. **Non-Maximum Suppression (NMS)**
    - The `nms` function removes overlapping detections, keeping only the highest-scoring boxes.
    - The `average_boxes` function further merges boxes with high overlap to produce robust final detections.
5. **Visualization**
    - Detected bounding boxes and their scores are drawn on the image.
    - The result is displayed and saved.

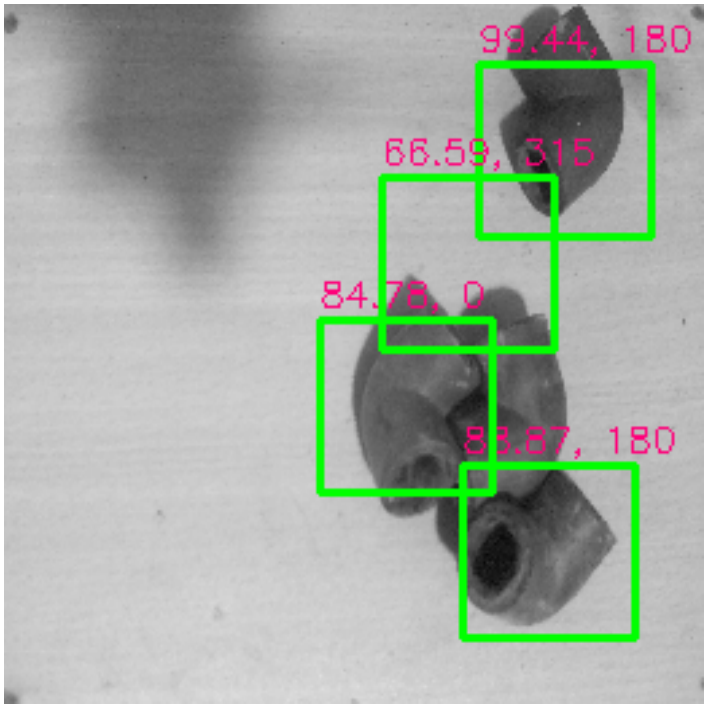
Advantages

- **Rotation Invariance:** By matching multiple rotated templates, the method detects objects regardless of their orientation.
- **Simplicity:** Uses standard OpenCV functions, making it easy to adapt and extend.
- **Post-processing:** NMS and box averaging improve detection quality by reducing duplicates.

Usage Notes

- Adjust the `threshold` and `iou_thresh` parameters for your specific application and dataset.
- Ensure the template and image are preprocessed (e.g., resized, normalized) consistently for best results.
- This method is suitable for scenarios where the object of interest may appear at arbitrary rotations in the image.

Example Results



Detected bounding boxes are shown in green. The image demonstrates successful detection of rotated templates using OpenCV's template matching with rotation.

In [19]:
```python
import cv2
import numpy as np
import time



# Rotate image with padding
def rotate_image_with_padding(image, angle):
    h, w = image.shape
    center = (w // 2, h // 2)
    rot_mat = cv2.getRotationMatrix2D(center, angle, 1.0)
    cos, sin = np.abs(rot_mat[0, 0]), np.abs(rot_mat[0, 1])
    nW, nH = int(h * sin + w * cos), int(h * cos + w * sin)
    rot_mat[0, 2] += (nW / 2) - center[0]
    rot_mat[1, 2] += (nH / 2) - center[1]
    return cv2.warpAffine(image, rot_mat, (nW, nH), borderValue=255)

# Template matching with score threshold
def match_template(image, template, threshold=0.5):
    result = cv2.matchTemplate(image, template, cv2.TM_CCOEFF_NORMED)
    yx = np.where(result >= threshold)
    h, w = template.shape
    return [(x, y, x + w, y + h, result[y, x]) for y, x in zip(*yx)]

# Non-Maximum Suppression
def nms(boxes, iou_thresh=0.3):
    if not boxes:
        return []
    boxes = np.array(boxes)
    x1, y1, x2, y2, scores = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3], bo
```

```
    areas = (x2 - x1) * (y2 - y1)
    order = scores.argsort()[::-1]
    keep = []

    while order.size:
        i = order[0]
        keep.append(tuple(boxes[i]))
        xx1 = np.maximum(x1[i], x1[order[1:]])
        yy1 = np.maximum(y1[i], y1[order[1:]])
        xx2 = np.minimum(x2[i], x2[order[1:]])
        yy2 = np.minimum(y2[i], y2[order[1:]])
        w = np.maximum(0.0, xx2 - xx1)
        h = np.maximum(0.0, yy2 - yy1)
        iou = (w * h) / (areas[i] + areas[order[1:]] - (w * h) + 1e-6)
        order = order[1:][iou <= iou_thresh]

    return keep

# Main function
def main():
    start = time.time()

    img = cv2.imread('Dataset/test/images/image_20250321_200736_lbent_3.png', cv2.I
    template = cv2.imread("image_20250321_201450_lbent_1.png", cv2.IMREAD_GRAYSCALE

    img = cv2.resize(img, (260, 260))
    template = cv2.resize(template, (64,64))

    angles = [0, 45, 90, 135, 180, 225, 270, 315]
    all_boxes = []

    for angle in angles:
        rotated = rotate_image_with_padding(template, angle)
        bboxes = match_template(img, rotated, threshold=0.48)
        all_boxes.extend(bboxes)

    final_boxes = nms(all_boxes, iou_thresh=0.6)
    final_boxes = average_boxes(all_boxes,iou_thresh=0.22)

    output = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
    for x1, y1, x2, y2, score in final_boxes:
        cv2.rectangle(output, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0),
        cv2.putText(output, f"{score:.2f}", (int(x1), int(y1) - 5), cv2.FONT_HERSHE

    print(f"Time taken: {time.time() - start:.2f} sec")
    print(f"Detections: {len(final_boxes)}")

    cv2.imshow("Detections", output)
    cv2.imwrite("detections_output.png", output)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

```
Time taken: 0.13 sec
Detections: 5
```

This section demonstrates a fast, rotation-invariant template matching pipeline using PyTorch and a single batched convolution line. The approach leverages GPU acceleration for efficient detection of rotated templates in an image.

Key Steps and Structure

1. **Imports and Setup**
   - Uses `cv2` for image I/O and resizing, `numpy` for array operations, and `torch` for tensor computations.
   - `torch.nn.functional` and `torchvision.ops.nms` are used for convolution and non-maximum suppression (NMS).
2. **Image Preprocessing**
   - `to_tensor_from_red_channel(img_bgr)` : Converts the red channel of a BGR image to a normalized PyTorch tensor of shape `[1, 1, H, W]` .
3. **Template Rotation**
   - `rotate_tensor_image(tensor_img, angle_deg)` : Rotates a tensor image by a specified angle using affine transformations.
   - `get_rotated_templates(template_tensor, angles)` : Generates a batch of rotated templates for all specified angles, normalizing each.
4. **Batched Template Matching**
   - `match_template_batched(image_tensor, templates_batch, threshold)` : Performs a single batched `conv2d` operation to match all rotated templates at once. Detections above the threshold are collected as bounding boxes with scores.
5. **Non-Maximum Suppression**
   - `nms_torch(boxes, iou_threshold)` : Applies NMS to remove overlapping detections, keeping only the best matches.
6. **Main Pipeline**
   - Loads and resizes the input image and template.
   - Converts both to tensors using only the red channel.
   - Generates rotated templates for multiple angles.
   - Runs batched template matching and NMS to get final detections.
   - Draws bounding boxes and scores on the output image and displays/saves the result.

Advantages

- **Speed:** All template rotations are matched in a single convolution line, fully utilizing GPU parallelism.
- **Rotation Invariance:** By matching multiple rotated templates, the method detects objects regardless of their orientation.
- **Simplicity:** The code is concise and easy to adapt for other templates or image sizes.

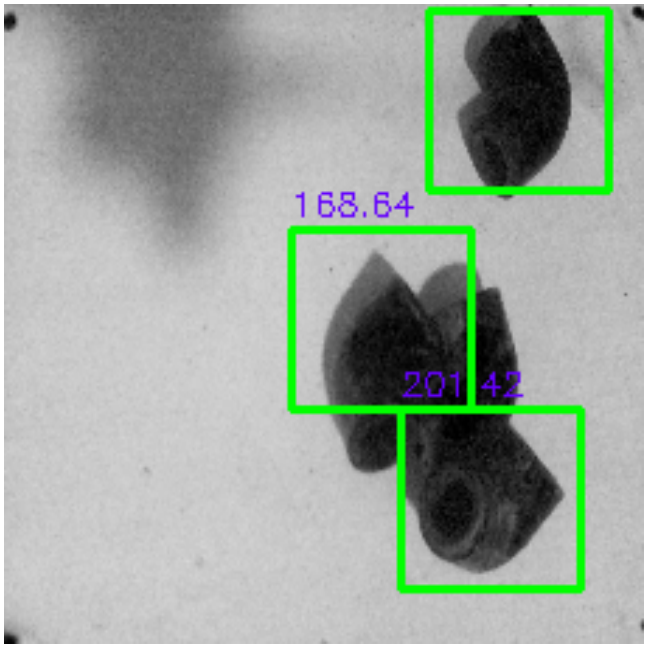
Usage

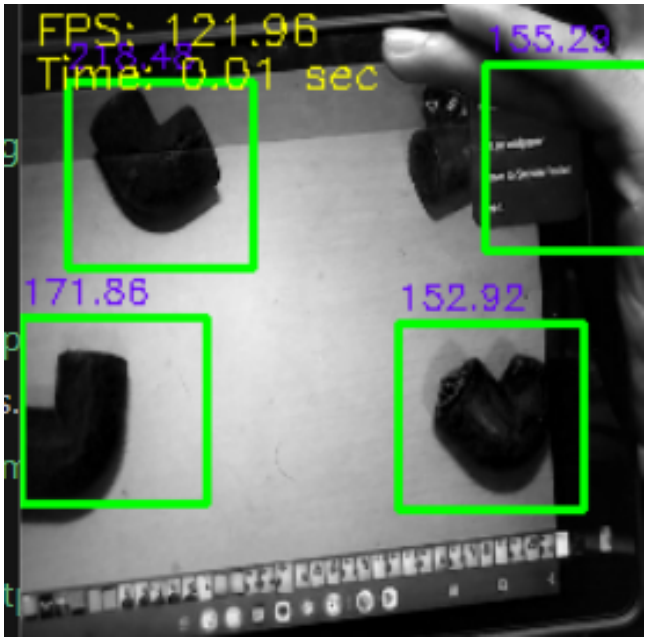
- Replace image paths as needed.
- Ensure a CUDA-capable GPU is available for best performance.
- Adjust the detection threshold and angles for your specific application.

This approach is ideal for real-time or batch detection tasks where both speed and rotation invariance are required.

Example Results

Below are some example detections using the PyTorch batched convolution approach:





Detected bounding boxes are shown in green. Each image demonstrates successful detection of rotated templates using a single batched convolution.

In [ ]:
```
import cv2
import numpy as np
import torch
import torch.nn.functional as F
from torchvision.ops import nms
import time
import math

# Convert red-channel grayscale image to normalized PyTorch tensor
def to_tensor_from_red_channel(img_bgr):
    red_channel = img_bgr[:, :, 2].astype(np.float32) / 255.0  # Normalize to [0, 1
    return torch.from_numpy(red_channel).unsqueeze(0).unsqueeze(0).contiguous()  #
```

```python
# Rotate a single image tensor
def rotate_tensor_image(tensor_img, angle_deg):
    angle_rad = math.radians(angle_deg)
    theta = torch.tensor([
        [math.cos(angle_rad), -math.sin(angle_rad), 0],
        [math.sin(angle_rad),  math.cos(angle_rad), 0]
    ], dtype=torch.float32, device=tensor_img.device)
    grid = F.affine_grid(theta.unsqueeze(0), tensor_img.size(), align_corners=False
    rotated = F.grid_sample(tensor_img, grid, padding_mode='zeros', align_corners=F
    return rotated

# Rotate template batch
def get_rotated_templates(template_tensor, angles):
    rotated_templates = []
    for angle in angles:
        rotated = rotate_tensor_image(template_tensor, angle)
        rotated -= rotated.mean()  # Normalize each rotated template
        rotated_templates.append(rotated)
    return torch.cat(rotated_templates, dim=0)  # [N, 1, h, w]

# Batched template matching using conv2d
def match_template_batched(image_tensor, templates_batch, threshold=0.48):
    N, _, h, w = templates_batch.shape
    response = F.conv2d(image_tensor, templates_batch,stride=1)  # [1, N, H-h+1, W-
    response_np = response.squeeze(0).detach().cpu().numpy()

    all_boxes = []
    for i in range(N):
        r = response_np[i]
        yx = np.where(r >= threshold)
        for y, x in zip(*yx):
            score = r[y, x]
            all_boxes.append((x, y, x + w, y + h, score))
            #print(f"Template {i}, Score at ({x}, {y}): {score:.4f}")  # Debug outp
    return all_boxes

# Torchvision NMS
def nms_torch(boxes, iou_threshold=0.6):
    if not boxes:
        return []
    boxes_np = np.array(boxes)
    boxes_tensor = torch.tensor(boxes_np[:, :4], dtype=torch.float32)
    scores_tensor = torch.tensor(boxes_np[:, 4], dtype=torch.float32)
    keep_indices = nms(boxes_tensor, scores_tensor, iou_threshold)
    return [(*boxes_np[i][:4], boxes_np[i][4]) for i in keep_indices]

# MAIN
def main():
    start = time.time()

    # Load color images and convert using red channel only
    img_bgr = cv2.imread('Dataset/test/images/image_20250321_200736_lbent_3.png')
    template_bgr = cv2.imread('image_20250321_201450_lbent_1.png')
    img_bgr = cv2.resize(img_bgr, (228, 228))
    template_bgr = cv2.resize(template_bgr, (64, 64))
```

```
    image_tensor = to_tensor_from_red_channel(img_bgr).cuda()
    template_tensor = to_tensor_from_red_channel(template_bgr).cuda()

    angles = [0, 45, 90, 135, 180, 225, 270, 315]
    rotated_templates = get_rotated_templates(template_tensor, angles)  # [N, 1, h,

    all_boxes = match_template_batched(image_tensor, rotated_templates, threshold=8
    final_boxes = nms_torch(all_boxes, iou_threshold=0.0)

    print(f"Time taken: {time.time() - start:.2f} sec")

    output = cv2.cvtColor((image_tensor.squeeze().cpu().numpy() * 255).astype(np.ui
    for x1, y1, x2, y2, score in final_boxes:
        cv2.rectangle(output, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0),
        cv2.putText(output, f"{score:.2f}", (int(x1), int(y1) - 5),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 0, 100), 1)


    print(f"Detections: {len(final_boxes)}")

    cv2.imshow("Detections", output)
    cv2.imwrite("detections_output_batched.png", output)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

Real time inference

This section demonstrates a real-time template matching pipeline using PyTorch and OpenCV. The approach leverages fast batched convolution on the GPU to detect rotated instances of a template in live video frames.

Key steps:

- The template image is loaded, resized, and converted to a PyTorch tensor using only the red channel for simplicity.
- Multiple rotated versions of the template are generated to achieve rotation-invariant detection.
- For each video frame, the frame is resized and converted to a tensor, then matched against all rotated templates using a single batched `conv2d` operation.
- Detections above a threshold are filtered using Non-Maximum Suppression (NMS) to remove duplicates.
- Detected bounding boxes are drawn on the frame, and the result is displayed in real time with FPS information.

This method is efficient because it uses GPU-accelerated convolution for template matching and supports real-time inference on live video streams. It is suitable for applications where you need to detect specific patterns or objects (with rotation invariance) in a video feed.

In [28]:

```python
import cv2
import numpy as np
import torch
import torch.nn.functional as F
from torchvision.ops import nms
import time
import math

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#device = torch.device("cpu")  # Force CPU for compatibility
print(f"Using device: {device}")

def to_tensor_from_red_channel(img_bgr):
    red_channel = img_bgr[:, :, 2].astype(np.float32) / 255.0
    return torch.from_numpy(red_channel).unsqueeze(0).unsqueeze(0).contiguous()

def rotate_tensor_image(tensor_img, angle_deg):
    angle_rad = math.radians(angle_deg)
    theta = torch.tensor([
        [math.cos(angle_rad), -math.sin(angle_rad), 0],
        [math.sin(angle_rad),  math.cos(angle_rad), 0]
    ], dtype=torch.float32, device=tensor_img.device)
    grid = F.affine_grid(theta.unsqueeze(0), tensor_img.size(), align_corners=False
    rotated = F.grid_sample(tensor_img, grid, padding_mode='zeros', align_corners=F
    return rotated

def get_rotated_templates(template_tensor, angles):
    rotated_templates = []
    for angle in angles:
        rotated = rotate_tensor_image(template_tensor, angle)
        rotated -= rotated.mean()
        rotated_templates.append(rotated)
    return torch.cat(rotated_templates, dim=0)

def match_template_batched(image_tensor, templates_batch, threshold=0.5):
    N, _, h, w = templates_batch.shape
    response = F.conv2d(image_tensor, templates_batch, stride=1)
    response_np = response.squeeze(0).detach().cpu().numpy()

    all_boxes = []
    for i in range(N):
        r = response_np[i]
        yx = np.where(r >= threshold)
        for y, x in zip(*yx):
            score = r[y, x]
            all_boxes.append((x, y, x + w, y + h, score))
    return all_boxes



def nms_torch(boxes, iou_threshold=0.6):
    if not boxes:
        return []
    boxes_np = np.array(boxes)
    boxes_tensor = torch.tensor(boxes_np[:, :4], dtype=torch.float32)
```

```
    scores_tensor = torch.tensor(boxes_np[:, 4], dtype=torch.float32)
    keep_indices = nms(boxes_tensor, scores_tensor, iou_threshold)
    return [(*boxes_np[i][:4], boxes_np[i][4]) for i in keep_indices]

def main():
    template_bgr = cv2.imread('image_20250321_201450_lbent_1.png')
    template_bgr = cv2.resize(template_bgr, (84, 84))
    template_tensor = to_tensor_from_red_channel(template_bgr).to(device)
    angles = [0, 45, 90, 135, 180, 225, 270, 315]
    rotated_templates = get_rotated_templates(template_tensor, angles).to(device)

    url = 'http://192.168.8.104:8080/video'  # Replace with your IP camera stream
    cap = cv2.VideoCapture(0)
    if not cap.isOpened():
        print(f"Unable to open video stream at {url}")
        return

    print("Press 'q' to quit.")
    while True:
        ret, frame = cap.read()
        if not ret:
            print("Failed to grab frame.")
            break

        frame = cv2.resize(frame, (170, 170))
        image_tensor = to_tensor_from_red_channel(frame).to(device)

        start = time.time()
        all_boxes = match_template_batched(image_tensor, rotated_templates, thresho
        final_boxes = nms_torch(all_boxes, iou_threshold=0.01)


        output = cv2.cvtColor((image_tensor.squeeze().cpu().numpy() * 255).astype(n
        for x1, y1, x2, y2, score in final_boxes:
            cv2.rectangle(output, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255,
            cv2.putText(output, f"{score:.2f}", (int(x1), int(y1) - 5),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 0, 100), 1)
        elapsed = time.time() - start
        if elapsed > 0:

            cv2.putText(output, f"FPS: {1/elapsed:.2f}", (5, 15), cv2.FONT_HERSHEY_
            cv2.putText(output,f"time: {elapsed:.2f} sec", (5, 30), cv2.FONT_HERSHE
        cv2.imshow("Real-Time Detection", output)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()


if __name__ == "__main__":
    with torch.no_grad():
        main()
```

```
Using device: cuda
Press 'q' to quit.
```

Make new cropped data set

This script creates a new dataset of cropped images based on COCO-style annotations. It reads bounding box annotations from a JSON file, loads each corresponding image, and crops a region around each annotated object (with optional padding). The cropped region is resized to a fixed size (96x96 pixels) and saved to an output directory. This process is useful for generating training data for object detection or classification tasks, focusing on the regions of interest.

In [ ]:
```python
import os
import json
import cv2

# CONFIG
json_path = 'dataset01\\annotations\\train.json'
images_dir = 'dataset01\\train'  # UPDATE THIS
output_dir = 'cropped_l_bends'
crop_size = 96

os.makedirs(output_dir, exist_ok=True)

with open(json_path, 'r') as f:
    coco = json.load(f)

# Index images by id
id_to_filename = {img['id']: img['file_name'] for img in coco['images']}

for ann in coco['annotations']:
    img_id = ann['image_id']
    bbox = ann['bbox']  # [x, y, w, h]

    x, y, w, h = bbox
    cx, cy = x + w / 2, y + h / 2  # centroid in original image

    image_path = os.path.join(images_dir, id_to_filename[img_id])
    image = cv2.imread(image_path)

    if image is None:
        print(f"Warning: Could not load {image_path}")
        continue

    H, W, _ = image.shape

    # Crop a square region centered at (cx, cy)

    cx, cy = int(cx), int(cy)
    # Convert bbox coordinates to integers for slicing
```

```
    x1, y1 = int(x), int(y)
    x2, y2 = int(x + w), int(y + h)
    #add padding befor cropping
    pad = 5
    x1 = max(0,x1 - pad)
    y1 = max(0,y1 - pad)
    x2 = min(W, x2 + pad)
    y2 = min(H, y2 + pad)


    # Handle border cases
    if x1 < 0 or y1 < 0 or x2 > W or y2 > H:
        continue  # skip if crop goes out of bounds

    crop = image[y1:y2, x1:x2]
    resized = cv2.resize(crop, (crop_size, crop_size))

    out_name = f"l_bend_{ann['id']:04d}.png"
    out_path = os.path.join(output_dir, out_name)
    cv2.imwrite(out_path, resized)

    print(f"Saved: {out_path}")
```

CNN model for validate objects and cetroid predict

This section defines a PyTorch-based pipeline for detecting and localizing "L-bend" objects in images using a convolutional neural network (CNN). The workflow includes:

- **Custom Dataset (`LBendDataset`)**: Loads images and their annotations (labels and centroid coordinates) from a CSV file. Each sample returns the image tensor, a binary label (L-bend or not), and normalized centroid coordinates.
- **CNN Model (`CNNDetector`)**: A simple CNN backbone extracts features from the input image. The network has two heads:
  - A classification head (outputs probability of L-bend).
  - A regression head (predicts centroid coordinates, normalized to [0,1]).
- **Training Function (`train_model`)**: Handles data splitting, augmentation, model training, validation, and checkpointing. It optimizes both classification (binary cross-entropy) and centroid regression (MSE) losses. The best model is saved and exported to TorchScript for efficient inference.

This approach enables both object validation (is it an L-bend?) and precise localization (where is the centroid?) in a single, end-to-end trainable model.

In [2]:
```
import os
import cv2
import torch
import pandas as pd
import numpy as np
```

```
from torch.utils.data import Dataset, DataLoader, random_split
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import transforms
from sklearn.model_selection import train_test_split
import time

# Make the dataset

class LBendDataset(Dataset):
    def __init__(self,csv_path,images_dir, transform=None):
        self.annotations = pd.read_csv(csv_path, header=None)
        self.annotations.columns = ['label_name', 'centroid_x', 'centroid_y', 'imag
        self.images_dir = images_dir
        self.transform = transform

    def __len__(self):
        return len(self.annotations)

    def __getitem__(self, idx):

        #get image file name and path
        img_name = self.annotations.iloc[idx]['image_name']
        img_path = os.path.join(self.images_dir,img_name)

        #load image
        image = cv2.imread(img_path)
        if image is None:
            raise FileNotFoundError(f"Could not load image at {img_path}")

        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        label = self.annotations.iloc[idx]['label_name']
        if label == 'L_bent':
            label = 1.0
        else:
            label = 0.0

        h,w = image.shape[:2]
        centroid_x = self.annotations.iloc[idx]['centroid_x']/w
        centroid_y = self.annotations.iloc[idx]['centroid_y']/h

        #Resize if need a standard size
        image = cv2.resize(image,(64,64))

        if self.transform:
            image = self.transform(image)
        else:
            image = image.astype(np.float32)/255.0 # change this isf need float16
            image = torch.from_numpy(image).permute(2,0,1)

        return(
            image,
            torch.tensor([float(label)],dtype=torch.float32),  # Assuming label is
            torch.tensor([centroid_x, centroid_y], dtype=torch.float32),  # Centroi
```

```
        )


# Define the CNN model
class CNNDetector(nn.Module):
    def __init__(self):
        super(CNNDetector,self).__init__()

        #Backbone
        self.features = nn.Sequential(
            #Layer 1
            nn.Conv2d(3, 16, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),

            #Layer 2
            nn.Conv2d(16,32,kernel_size=3,stride = 1,padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),

            #Layer 3
            nn.Conv2d(32,64,kernel_size=3,stride=1,padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),

            #Layer 4
            nn.Conv2d(64,128,kernel_size=3,stride=1,padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),

            nn.AdaptiveAvgPool2d(1)

        )

        #Classifier
        self.classifier = nn.Linear(128,1)
        #Regression head for centroid coordinates
        self.regression = nn.Linear(128, 2)  # Output 2 values

    def forward(self, x):

        #Extract features
        features = self.features(x)
        features = torch.flatten(features, 1)  # Flatten the features

        class_output = torch.sigmoid(self.classifier(features))
        centroid_output = torch.sigmoid(self.regression(features))

        return class_output, centroid_output

def train_model(csv_path,images_dir,epochs = 30,batch_size = 32, learning_rate=0.00

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
print(f"Using device: {device}")

# Define transformations
train_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

val_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

df = pd.read_csv(csv_path, header=None)
train_df,val_df = train_test_split(df, test_size=0.2, random_state=42, stratify

train_csv = 'train_annotations.csv'
val_csv = 'val_annotations.csv'
train_df.to_csv(train_csv, index=False, header=False)
val_df.to_csv(val_csv, index=False, header=False)

train_dataset = LBendDataset(train_csv, images_dir, transform=None)
val_dataset = LBendDataset(val_csv, images_dir, transform=None)

train_loader = DataLoader(train_dataset,batch_size=batch_size, shuffle=True, nu
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_

model = CNNDetector().to(device)

#loss functions
classification_loss = nn.BCELoss()
regression_loss = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=learning_rate)

scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=

best_val_loss = float('inf')
best_model_state = None

for epoch in range(epochs):
    model.train()
    train_cls_loss = 0.0
    train_reg_loss = 0.0
    train_loss = 0.0

    for images,labels,centroids in train_loader:
        images = images.to(device)
        labels = labels.to(device)
        centroids = centroids.to(device)

        optimizer.zero_grad()
        class_output, centroid_output = model(images)
```

```
            cls_loss = classification_loss(class_output.squeeze(), labels.squeeze())
            reg_loss = regression_loss(centroid_output, centroids)

            loss = cls_loss + 5.0*reg_loss

            loss.backward()
            optimizer.step()

            train_cls_loss += cls_loss.item()
            train_reg_loss += reg_loss.item()
            train_loss += loss.item()

        #Avg training loss
        train_cls_loss /= len(train_loader)
        train_reg_loss /= len(train_loader)
        train_loss /= len(train_loader)

        #validate model
        model.eval()
        val_cls_loss = 0.0
        val_reg_loss = 0.0
        val_loss = 0.0

        correct = 0
        total = 0

        with torch.no_grad():
             for images, labels, centroids in val_loader:
                images = images.to(device)
                labels = labels.to(device)
                centroids = centroids.to(device)

                # Forward pass
                cls_output, centroid_output = model(images)

                # Calculate losses
                cls_loss = classification_loss(cls_output, labels)
                reg_loss = regression_loss(centroid_output, centroids)
                loss = cls_loss + 5.0 * reg_loss

                # Accumulate validation loss
                val_cls_loss += cls_loss.item()
                val_reg_loss += reg_loss.item()
                val_loss += loss.item()

                # Calculate accuracy
                predicted = (cls_output > 0.5).float()
                correct += (predicted == labels).sum().item()
                total += labels.size(0)

        # Calculate average validation loss and accuracy
        val_cls_loss /= len(val_loader)
        val_reg_loss /= len(val_loader)
        val_loss /= len(val_loader)
        accuracy = correct / total
```

```
        # Update learning rate based on validation loss
        scheduler.step(val_loss)

        # Print epoch results
        print(f"Epoch {epoch+1}/{epochs}")
        print(f"Train Loss: {train_loss:.4f} (Cls: {train_cls_loss:.4f}, Reg: {trai
        print(f"Val Loss: {val_loss:.4f} (Cls: {val_cls_loss:.4f}, Reg: {val_reg_lo
        print(f"Accuracy: {accuracy:.4f}")

        # Save best model
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict().copy()
            print("Saved best model!")

            # Save the model checkpoint
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'val_loss': val_loss,
                'val_accuracy': accuracy,
            }, 'best_lbend_model.pth')

    # Clean up temporary CSV files
    os.remove(train_csv)
    os.remove(val_csv)

    # Load the best model state
    if best_model_state:
        model.load_state_dict(best_model_state)

    # Export to TorchScript for faster inference
    model.eval()
    example = torch.rand(1, 3, 64, 64).to(device)
    traced_model = torch.jit.trace(model, example)
    traced_model.save('lbend_detector_scripted.pt')
    print("Model exported to TorchScript format for fast inference")

    return model
```

Training process

In [ ]:

```
csv_path = "labels_my-project-name_2025-07-02-05-15-40.csv"
images_dir = "cropped_l_bends"


train_model(csv_path, images_dir, epochs=40, batch_size=16, learning_rate=0.001)
```

Traaining and visualizing losses and accuracy

In [39]:
```
import os
import cv2
import torch
import pandas as pd
import numpy as np
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import transforms
from sklearn.model_selection import train_test_split
import time
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

def train_model(csv_path, images_dir, epochs=40, batch_size=16, learning_rate=0.001
    # Create lists to store metrics for plotting
    train_losses = []
    train_cls_losses = []
    train_reg_losses = []
    val_losses = []
    val_cls_losses = []
    val_reg_losses = []
    val_accuracies = []

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Define transformations
    train_transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    val_transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    # Split data and create datasets/loaders (your existing code)
    df = pd.read_csv(csv_path, header=None)
    train_df, val_df = train_test_split(df, test_size=0.2, random_state=42, stratif

    train_csv = 'train_annotations.csv'
    val_csv = 'val_annotations.csv'
    train_df.to_csv(train_csv, index=False, header=False)
    val_df.to_csv(val_csv, index=False, header=False)
```

```
train_dataset = LBendDataset(train_csv, images_dir, transform=None)
val_dataset = LBendDataset(val_csv, images_dir, transform=None)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, n
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_

# Initialize model, loss functions and optimizer
model = CNNDetector().to(device)
classification_loss = nn.BCELoss()
regression_loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=

# Create a figure for real-time plotting
plt.figure(figsize=(12, 10))
plt.ion()  # Turn on interactive mode

best_val_loss = float('inf')
best_model_state = None

# Training loop
for epoch in range(epochs):
    model.train()
    train_cls_loss = 0.0
    train_reg_loss = 0.0
    train_loss = 0.0

    # Training phase
    for images, labels, centroids in train_loader:
        images = images.to(device)
        labels = labels.to(device)
        centroids = centroids.to(device)

        optimizer.zero_grad()
        class_output, centroid_output = model(images)

        cls_loss = classification_loss(class_output.squeeze(), labels.squeeze()
        reg_loss = regression_loss(centroid_output, centroids)

        loss = cls_loss + 5.0 * reg_loss

        loss.backward()
        optimizer.step()

        train_cls_loss += cls_loss.item()
        train_reg_loss += reg_loss.item()
        train_loss += loss.item()

    # Calculate average training loss
    train_cls_loss /= len(train_loader)
    train_reg_loss /= len(train_loader)
    train_loss /= len(train_loader)

    # Store training metrics
    train_losses.append(train_loss)
```

```
        train_cls_losses.append(train_cls_loss)
        train_reg_losses.append(train_reg_loss)

        # Validation phase
        model.eval()
        val_cls_loss = 0.0
        val_reg_loss = 0.0
        val_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad():
            for images, labels, centroids in val_loader:
                images = images.to(device)
                labels = labels.to(device)
                centroids = centroids.to(device)

                # Forward pass
                cls_output, centroid_output = model(images)

                # Calculate losses
                cls_loss = classification_loss(cls_output, labels)
                reg_loss = regression_loss(centroid_output, centroids)
                loss = cls_loss + 5.0 * reg_loss

                # Accumulate validation loss
                val_cls_loss += cls_loss.item()
                val_reg_loss += reg_loss.item()
                val_loss += loss.item()

                # Calculate accuracy
                predicted = (cls_output > 0.5).float()
                correct += (predicted == labels).sum().item()
                total += labels.size(0)

        # Calculate average validation metrics
        val_cls_loss /= len(val_loader)
        val_reg_loss /= len(val_loader)
        val_loss /= len(val_loader)
        accuracy = correct / total

        # Store validation metrics
        val_losses.append(val_loss)
        val_cls_losses.append(val_cls_loss)
        val_reg_losses.append(val_reg_loss)
        val_accuracies.append(accuracy)

        # Update learning rate based on validation loss
        scheduler.step(val_loss)

        # Print epoch results
        print(f"Epoch {epoch+1}/{epochs}")
        print(f"Train Loss: {train_loss:.4f} (Cls: {train_cls_loss:.4f}, Reg: {trai
        print(f"Val Loss: {val_loss:.4f} (Cls: {val_cls_loss:.4f}, Reg: {val_reg_lo
        print(f"Accuracy: {accuracy:.4f}")
```

```
        # Save best model
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_state = model.state_dict().copy()
            print("Saved best model!")

            # Save model checkpoint
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'val_loss': val_loss,
                'val_accuracy': accuracy,
                'train_losses': train_losses,
                'val_losses': val_losses,
                'train_cls_losses': train_cls_losses,
                'val_cls_losses': val_cls_losses,
                'train_reg_losses': train_reg_losses,
                'val_reg_losses': val_reg_losses,
                'val_accuracies': val_accuracies
            }, 'best_lbend_model.pth')

    # Final plot (non-interactive)
    plt.ioff()
    plt.figure(figsize=(16, 12))

    # Plot final losses
    plt.subplot(2, 2, 1)
    plt.plot(range(1, epochs+1), train_losses, 'b-', label='Training')
    plt.plot(range(1, epochs+1), val_losses, 'r-', label='Validation')
    plt.title('Total Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))

    # Plot final classification losses
    plt.subplot(2, 2, 2)
    plt.plot(range(1, epochs+1), train_cls_losses, 'b-', label='Training')
    plt.plot(range(1, epochs+1), val_cls_losses, 'r-', label='Validation')
    plt.title('Classification Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))

    # Plot final regression losses
    plt.subplot(2, 2, 3)
    plt.plot(range(1, epochs+1), train_reg_losses, 'b-', label='Training')
    plt.plot(range(1, epochs+1), val_reg_losses, 'r-', label='Validation')
    plt.title('Regression Loss')
    plt.xlabel('Epoch')
```

```
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))

    # Plot final validation accuracy
    plt.subplot(2, 2, 4)
    plt.plot(range(1, epochs+1), val_accuracies, 'g-')
    plt.title('Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.grid(True)
    plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))
    plt.ylim(0, 1)

    plt.tight_layout()
    plt.savefig('final_training_results.png', dpi=200)
    plt.show()

    # Clean up temporary CSV files
    os.remove(train_csv)
    os.remove(val_csv)

    # Load the best model state
    if best_model_state:
        model.load_state_dict(best_model_state)

    # Export to TorchScript for faster inference
    model.eval()
    example = torch.rand(1, 3, 64, 64).to(device)
    traced_model = torch.jit.trace(model, example)
    traced_model.save('lbend_detector_scripted.pt')
    print("Model exported to TorchScript format for fast inference")

    # Return the trained model and metrics for further analysis
    return model, {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'train_cls_losses': train_cls_losses,
        'val_cls_losses': val_cls_losses,
        'train_reg_losses': train_reg_losses,
        'val_reg_losses': val_reg_losses,
        'val_accuracies': val_accuracies
    }

if __name__ == "__main__":
    csv_path = "labels_my-project-name_2025-07-02-05-15-40.csv"
    images_dir = "cropped_l_bends"
    model = train_model(csv_path, images_dir)

    print("\nTraining complete!")
    print("To view TensorBoard logs, run:")
    print("tensorboard --logdir=runs")
```

Model Evaluation and Visualization

This code provides a comprehensive evaluation pipeline for the trained L-bend detector model. It includes:

- **Model Loading**: Automatically loads either a TorchScript model (`.pt`) or a PyTorch checkpoint (`.pth`), supporting both fast inference and flexible retraining.
- **Validation Data Preparation**: Splits the labeled dataset into training and validation sets, then creates a validation dataset and DataLoader for efficient batch processing.
- **Evaluation Metrics**:
  - **Classification Metrics**: Computes accuracy, precision, recall, F1-score, and confusion matrix for L-bend vs. non-L-bend classification.
  - **Centroid Regression Error**: Calculates the mean, median, min, and max errors between predicted and ground-truth centroids (normalized coordinates).
  - **Inference Speed**: Measures average inference time per image and estimates the maximum achievable FPS.
- **Visualization**: Randomly selects a few validation samples and visualizes:
  - The original image.
  - Ground-truth centroid (green dot).
  - Predicted centroid (red dot).
  - A line connecting the two for error visualization.
  - Predicted and ground-truth class labels.
- **Usage**:
  To run the evaluation, simply execute the script with the correct model path, CSV label file, and image directory.
  Example:

```
model_path = "best_lbend_model.pth"  # or 'lbend_detector_scripted.pt'
csv_path = "labels_my-project-name_2025-07-02-05-15-40.csv"
images_dir = "cropped_l_bends"
evaluate_model(model_path, csv_path, images_dir)
```

This evaluation helps you understand both the classification and localization performance of your model, and provides visual feedback for qualitative assessment.

In [ ]:

```
import os
import cv2
import torch
import numpy as np
import matplotlib.pyplot as plt
import time
from torch.utils.data import DataLoader
from torchvision import transforms
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
```

```python
def evaluate_model(model_path, csv_path, images_dir):

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Load model
    if model_path.endswith('.pt'):
        # Load TorchScript model
        model = torch.jit.load(model_path, map_location=device)
        print("Loaded TorchScript model")
    else:
        # Load PyTorch checkpoint
        model = CNNDetector().to(device)
        checkpoint = torch.load(model_path, map_location=device)
        model.load_state_dict(checkpoint['model_state_dict'])
        print(f"Loaded model checkpoint (val_loss: {checkpoint.get('val_loss', 'N/A

    model.eval()

    # Create validation dataset
    val_transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    # Split data to get validation set
    df = pd.read_csv(csv_path, header=None)
    train_df, val_df = train_test_split(df, test_size=0.2, random_state=42, stratif

    # Save val CSV temporarily
    val_csv = 'temp_val.csv'
    val_df.to_csv(val_csv, index=False, header=False)

    # Create dataset and loader
    val_dataset = LBendDataset(val_csv, images_dir, transform=None)
    val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)  # Larger ba

    # Collect predictions and ground truth
    all_preds = []
    all_labels = []
    all_centroid_errors = []
    inference_times = []

    with torch.no_grad():
        for image, label, centroid in val_loader:
            image = image.to(device)
            label = label.to(device)
            centroid = centroid.to(device)

            # Measure inference time
            start_time = time.time()
            pred_class, pred_centroid = model(image)
            inference_times.append(time.time() - start_time)
```

```
            # Classification results
            pred_label = (pred_class > 0.5).float()
            all_preds.extend(pred_label.cpu().numpy())
            all_labels.extend(label.cpu().numpy())

            # Centroid error
            centroid_error = torch.sqrt(((pred_centroid - centroid) ** 2).sum(dim=1
            all_centroid_errors.extend(centroid_error)

    # 1. Classification Report
    all_preds = np.array(all_preds).flatten()
    all_labels = np.array(all_labels).flatten()

    print("\n--- CLASSIFICATION METRICS ---")
    print(classification_report(all_labels, all_preds, target_names=['Not L-bend',

    cm = confusion_matrix(all_labels, all_preds)
    print("Confusion Matrix:")
    print(cm)

    # 2. Centroid Error Analysis
    all_centroid_errors = np.array(all_centroid_errors)

    print("\n--- CENTROID PREDICTION ERROR ---")
    print(f"Mean Error (normalized): {np.mean(all_centroid_errors):.4f}")
    print(f"Median Error (normalized): {np.median(all_centroid_errors):.4f}")
    print(f"Min Error: {np.min(all_centroid_errors):.4f}")
    print(f"Max Error: {np.max(all_centroid_errors):.4f}")

    # 3. Inference Time Analysis
    avg_inference_time = np.mean(inference_times) / val_loader.batch_size  # Per im

    print("\n--- INFERENCE TIME ---")
    print(f"Average inference time per image: {avg_inference_time*1000:.2f} ms")
    print(f"Theoretical max FPS: {1/avg_inference_time:.1f}")

    # 4. Visualization of sample predictions
    visualize_samples(model, val_dataset, device, num_samples=9)

    # Clean up
    if os.path.exists(val_csv):
        os.remove(val_csv)

def visualize_samples(model, dataset, device, num_samples=9):
    """
    Visualize sample predictions with ground truth
    """
    indices = np.random.choice(len(dataset), min(num_samples, len(dataset)), replac

    # Create figure
    num_cols = min(3, num_samples)
    num_rows = (num_samples + num_cols - 1) // num_cols
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 4*num_rows))
    if num_rows == 1 and num_cols == 1:
        axes = np.array([axes])
```

```
    axes = axes.flatten()

    for i, idx in enumerate(indices):
        if i >= len(axes):
            break

        # Get data
        image, label, centroid = dataset[idx]
        img_name = dataset.annotations.iloc[idx]['image_name']

        # Get original image (without normalization)
        img_path = os.path.join(dataset.images_dir, img_name)
        orig_img = cv2.imread(img_path)
        orig_img = cv2.cvtColor(orig_img, cv2.COLOR_BGR2RGB)
        h, w = orig_img.shape[:2]

        # Forward pass
        with torch.no_grad():
            image_tensor = image.unsqueeze(0).to(device)
            pred_class, pred_centroid = model(image_tensor)

        # Convert to values
        gt_label = label.item()
        gt_centroid_x, gt_centroid_y = centroid[0].item(), centroid[1].item()
        pred_label = (pred_class > 0.5).float().item()
        pred_prob = pred_class.item()
        pred_centroid_x, pred_centroid_y = pred_centroid[0][0].item(), pred_centroi

        # Convert normalized coordinates to pixel coordinates
        gt_x, gt_y = int(gt_centroid_x * w), int(gt_centroid_y * h)
        pred_x, pred_y = int(pred_centroid_x * w), int(pred_centroid_y * h)

        # Draw on image
        vis_img = orig_img.copy()

        # Draw centroids
        cv2.circle(vis_img, (gt_x, gt_y), 5, (0, 255, 0), -1)  # Ground truth (gree
        cv2.circle(vis_img, (pred_x, pred_y), 5, (255, 0, 0), -1)  # Prediction (re

        # Draw connecting line
        cv2.line(vis_img, (gt_x, gt_y), (pred_x, pred_y), (255, 255, 0), 2)

        # Title
        title = f"GT: {'L-bend' if gt_label > 0.5 else 'Not L-bend'}\n"
        title += f"Pred: {'L-bend' if pred_label > 0.5 else 'Not L-bend'} ({pred_pr

        # Display
        axes[i].imshow(vis_img)
        axes[i].set_title(title)
        axes[i].axis('off')

    # Hide unused axes
    for j in range(i+1, len(axes)):
        axes[j].axis('off')

    plt.tight_layout()
```

```
    plt.savefig('model_evaluation.png', dpi=150)
    plt.show()

if __name__ == "__main__":
    model_path = "best_lbend_model.pth"  # or 'lbend_detector_scripted.pt'
    csv_path = "labels_my-project-name_2025-07-02-05-15-40.csv"
    images_dir = "cropped_l_bends"

    evaluate_model(model_path, csv_path, images_dir)
```

In [ ]:

```
import cv2
import torch
import numpy as np
import time
from torchvision import transforms

def real_time_inference(model_path, camera_index=0, confidence_threshold=0.5):

    # Set device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Load model
    try:
        # Try loading as TorchScript model first (faster)
        model = torch.jit.load(model_path, map_location=device)
        print("Loaded TorchScript model")
    except:
        # Fall back to loading PyTorch checkpoint
        #from your_model_file import CNNDetector
        model = CNNDetector().to(device)
        checkpoint = torch.load(model_path, map_location=device)
        model.load_state_dict(checkpoint['model_state_dict'])
        print("Loaded PyTorch checkpoint")

    model.eval()

    # Define transform
    transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.Resize((64, 64)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    # Open camera
    cap = cv2.VideoCapture(camera_index)
    if not cap.isOpened():
        print(f"Error: Could not open camera {camera_index}")
        return

    # Get camera properties
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

    # For FPS calculation
```

```
    fps = 0
    frame_times = []

    print("Press 'q' to quit")

    while True:
        # Start timing
        start_time = time.time()

        # Read frame
        ret, frame = cap.read()
        if not ret:
            print("Error: Failed to capture image")
            break

        # Convert to RGB (for preprocessing)
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        # Preprocess
        input_tensor = transform(rgb_frame).unsqueeze(0).to(device)

        # Run inference
        with torch.no_grad():
            class_output, centroid_output = model(input_tensor)

        # Get results
        probability = class_output.item()
        is_lbend = probability > confidence_threshold

        # Convert normalized centroid coordinates back to original image
        centroid_x = int(centroid_output[0, 0].item() * frame_width)
        centroid_y = int(centroid_output[0, 1].item() * frame_height)

        # Draw results on frame
        if is_lbend:
            # Draw centroid
            cv2.circle(frame, (centroid_x, centroid_y), 5, (0, 255, 0), -1)

            # Draw box around detection
            box_size = min(frame_width, frame_height) // 8
            x1 = max(0, centroid_x - box_size)
            y1 = max(0, centroid_y - box_size)
            x2 = min(frame_width, centroid_x + box_size)
            y2 = min(frame_height, centroid_y + box_size)
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)

            # Show confidence
            text = f"L-bend: {probability:.2f}"
            cv2.putText(frame, text, (x1, y1-5), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,

        # Calculate and display FPS
        process_time = time.time() - start_time
        frame_times.append(process_time)
        # Keep only the last 30 frames for averaging
        if len(frame_times) > 30:
            frame_times.pop(0)
```

```
        fps = 1.0 / (sum(frame_times) / len(frame_times))
        cv2.putText(frame, f"FPS: {fps:.1f}", (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1

        # Display frame
        cv2.imshow('L-Bend Detector', frame)

        # Exit on 'q' key
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # Release resources
    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    model_path = "lbend_detector_scripted.pt"  # Prefer TorchScript for faster infe
    real_time_inference(model_path, camera_index=0)
```

```
Using device: cuda
Loaded TorchScript model
Press 'q' to quit
```

Final L- Bent detection Model **Template matching RPN + centroid Prediction Model**

This code implements a **hybrid L-bend detection pipeline** that combines a classical computer vision RPN (Region Proposal Network) using OpenCV template matching with a deep learning-based CNN for validation and centroid prediction.

Pipeline Overview

1. **Template Matching RPN (Region Proposal Network)**
    - Uses OpenCV's `cv2.matchTemplate` to scan the input image for L-bend patterns.
    - Eight rotated versions of the template (0°, 45°, ..., 315°) are used for rotation-invariant detection.
    - For each rotation, the template is matched to the image, and locations with high similarity scores are collected as candidate bounding boxes.
2. **Non-Maximum Suppression (NMS)**
    - Overlapping detections are filtered using NMS to keep only the best candidates.
3. **CNN Validation and Centroid Prediction**
    - Each candidate region (cropped from the image) is passed to a trained CNN.
    - The CNN outputs:

        - A confidence score (probability of being an L-bend).
        - The predicted centroid coordinates (relative to the crop).
    - Only detections with a confidence above a threshold are kept.
4. **Post-processing**

    - The predicted centroids are mapped back to the original image coordinates.
    - The detection closest to the image center is selected as the final L-bend location (useful for robotic picking or tracking tasks).
    - Annotated results (bounding boxes, centroids, scores, FPS) are drawn on the output frame.

Why This Approach?

- **Template Matching RPN**:
  After evaluating several RPN methods (FFT-based, PyTorch batched convolution, OpenCV template matching), OpenCV's `matchTemplate` was chosen for its speed and reliability, especially on resource-constrained devices like Jetson Nano. The FFT and convolution approaches were either too slow or less robust in practice.

- **Rotation Invariance**:
  By using 8 rotated templates, the system can detect L-bends at any orientation.

- **CNN Validation**:
  The CNN filters out false positives and provides precise centroid localization, improving accuracy over classical methods alone.

How It Works (Step-by-Step)

1. **Initialization**:

    - Load the template image and generate 8 rotated versions.
    - Load the trained CNN model (TorchScript for fast inference).
2. **Frame Processing**:

    - Resize the input frame for consistent processing.
    - Run template matching for each rotated template to get candidate boxes.
    - Apply NMS to remove redundant detections.
    - Extract and preprocess crops from the candidate boxes.
3. **CNN Inference**:

    - Batch all crops and run them through the CNN.
    - For each crop, get the confidence score and centroid prediction.
    - Keep only detections above the confidence threshold.

4. **Result Aggregation**:
    - Map predicted centroids back to the original image.
    - Find the detection closest to the image center (if needed).
    - Draw bounding boxes, centroids, and scores on the frame.
    - Display FPS and total detections.

Advantages

- **Fast and Lightweight**: Runs in real time on Jetson Nano and similar devices.
- **Accurate**: Combines the strengths of classical and deep learning methods.
- **Rotation-Invariant**: Robust to object orientation.
- **Easy to Extend**: Can be adapted for other shapes or templates.

Usage

- The `CombinedDetector` class encapsulates the full pipeline.
- Use `run_real_time_detection(template_path, model_path, frame)` to process a frame and get the centroid and annotated output.
- The main loop demonstrates real-time webcam inference.

In summary:
This final model leverages the best of both worlds: OpenCV template matching for fast, rotation-invariant region proposals, and a CNN for robust validation and precise localization. This hybrid approach was chosen after extensive experimentation and is well-suited for real-time L-bend detection in practical applications.

In [ ]:
```
import cv2
import numpy as np
import torch
import torch.nn.functional as F
import time


class CombinedDetector:
    def __init__(self, template_path, model_path, threshold=0.52, iou_threshold=0.3
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        print(f"Using device: {self.device}")

        template_bgr = cv2.imread(template_path)
        template_bgr = cv2.resize(template_bgr, (64, 64))
        self.template_tensor = self.to_tensor_from_red_channel(template_bgr).to(sel

        angles = [0, 45, 90, 135, 180, 225, 270, 315]
        self.rotated_templates = self.get_rotated_templates(self.template_tensor, a
```

```
        try:
            self.model = torch.jit.load(model_path, map_location=self.device)
            print("Model loaded successfully.")
        except:
            print("Failed to load model.")
            raise

        self.model.eval()
        self.threshold = threshold
        self.iou_threshold = iou_threshold
        self.confidence_threshold = confidence_threshold
        self.frame_times = []
        self.detection_count = 0

    def preprocess(self,image_rgb):
        #image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
        resized = cv2.resize(image_rgb, (64, 64)).astype(np.float32) / 255.0
        tensor = torch.from_numpy(resized).permute(2, 0, 1)
        return tensor

    def to_tensor_from_red_channel(self, image_bgr):
        red_channel = image_bgr[:, :, 2].astype(np.float32) / 255
        return torch.from_numpy(red_channel).unsqueeze(0).unsqueeze(0).contiguous()

    def rotate_image_with_padding(self, image_np, angle):
        h, w = image_np.shape
        center = (w // 2, h // 2)
        rot_mat = cv2.getRotationMatrix2D(center, angle, 1.0)
        cos, sin = np.abs(rot_mat[0, 0]), np.abs(rot_mat[0, 1])
        nW, nH = int(h * sin + w * cos), int(h * cos + w * sin)
        rot_mat[0, 2] += (nW / 2) - center[0]
        rot_mat[1, 2] += (nH / 2) - center[1]
        return cv2.warpAffine(image_np, rot_mat, (nW, nH), borderValue=255)

    def get_rotated_templates(self, template_tensor, angles):
        rotated_np_templates = []
        for angle in angles:
            template_np = template_tensor.squeeze().cpu().numpy()
            rotated_np = self.rotate_image_with_padding(template_np, angle)
            rotated_np_templates.append(rotated_np)
        self.rotated_np_templates = rotated_np_templates
        return rotated_np_templates

    def match_template_opencv(self, image_np):
        all_boxes = []
        h, w = 64, 64
        image_gray = (image_np[:, :, 2] / 255.0).astype(np.float32)
        for template_np in self.rotated_np_templates:
            result = cv2.matchTemplate(image_gray, template_np, cv2.TM_CCOEFF_NORME
            yx = np.where(result >= self.threshold)
            for y, x in zip(*yx):
                score = result[y, x]
                all_boxes.append((x, y, x + w, y + h, score))
        return all_boxes
```

```
    def nms_numpy(self, boxes):
        if not boxes:
            return []
        boxes = np.array(boxes)
        x1, y1, x2, y2, scores = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
        areas = (x2 - x1) * (y2 - y1)
        order = scores.argsort()[::-1]
        keep = []

        while order.size:
            i = order[0]
            keep.append(tuple(boxes[i]))
            xx1 = np.maximum(x1[i], x1[order[1:]])
            yy1 = np.maximum(y1[i], y1[order[1:]])
            xx2 = np.minimum(x2[i], x2[order[1:]])
            yy2 = np.minimum(y2[i], y2[order[1:]])
            w = np.maximum(0.0, xx2 - xx1)
            h = np.maximum(0.0, yy2 - yy1)
            inter = w * h
            iou = inter / (areas[i] + areas[order[1:]] - inter + 1e-6)
            order = order[1:][iou <= self.iou_threshold]
        return keep

    def extract_crops(self, frame, boxes, padding=5):
        crops = []
        scaled_boxes = []
        frame_h, frame_w = frame.shape[:2]
        for x1, y1, x2, y2, _ in boxes:
            x1_pad = max(0, int(x1 - padding))
            y1_pad = max(0, int(y1 - padding))
            x2_pad = min(frame_w, int(x2 + padding))
            y2_pad = min(frame_h, int(y2 + padding))
            crop = frame[y1_pad:y2_pad, x1_pad:x2_pad]
            if crop.size > 0:
                crops.append(crop)
                scaled_boxes.append((x1_pad, y1_pad, x2_pad, y2_pad))
        return crops, scaled_boxes

    def process_with_cnn(self, crops, scaled_boxes):
        if not crops:
            return []
        batch_tensors = []
        for crop in crops:
            rgb_crop = cv2.cvtColor(crop, cv2.COLOR_BGR2RGB)
            tensor = self.preprocess(rgb_crop)
            batch_tensors.append(tensor)

        batch = torch.stack(batch_tensors).to(self.device)
        with torch.no_grad():
            class_outputs, centroid_outputs = self.model(batch)
        detections = []
        for i, ((x1, y1, x2, y2), class_output, centroid_output) in enumerate(zip(s
            probability = class_output.item()
            if probability > self.confidence_threshold:
                crop_w, crop_h = x2 - x1, y2 - y1
                crop_cx = int(centroid_output[0].item() * crop_w)
```

```
                crop_cy = int(centroid_output[1].item() * crop_h)
                frame_cx = x1 + crop_cx
                frame_cy = y1 + crop_cy
                detections.append({
                    'bbox': (x1, y1, x2, y2),
                    'centroid': (frame_cx, frame_cy),
                    'probability': probability
                })
        return detections

    def process_frame(self, frame):
        original_h, original_w = frame.shape[:2]
        process_size = (150, 150)
        resized_frame = cv2.resize(frame, process_size)
        start_time = time.time()
        all_boxes = self.match_template_opencv(resized_frame)
        filtered_boxes = self.nms_numpy(all_boxes)
        crops, scaled_boxes = self.extract_crops(resized_frame, filtered_boxes)
        detections = self.process_with_cnn(crops, scaled_boxes)
        scale_x = original_w / process_size[0]
        scale_y = original_h / process_size[1]
        for detection in detections:
            x1, y1, x2, y2 = detection['bbox']
            cx, cy = detection['centroid']
            detection['bbox'] = (int(x1 * scale_x), int(y1 * scale_y), int(x2 * sca
            detection['centroid'] = (int(cx * scale_x), int(cy * scale_y))
        process_time = time.time() - start_time
        self.frame_times.append(process_time)
        if len(self.frame_times) > 30:
            self.frame_times.pop(0)
        fps = 1.0 / (sum(self.frame_times) / len(self.frame_times))
        self.detection_count += len(detections)
        return {
            'detections': detections,
            'process_time': process_time,
            'fps': fps
        }

    def calculate_pixel_distance(self, point1, point2):
        coord_distance = np.array(point1) - np.array(point2)
        pixel_distance = np.linalg.norm(coord_distance)
        return pixel_distance

    def draw_results(self, frame, results,mid_point):
        output = frame.copy()
        min_distance_centroid = np.array([None, None])
        for det in results['detections']:
            x1, y1, x2, y2 = det['bbox']
            cx, cy = det['centroid']
            prob = det['probability']
            cv2.rectangle(output, (x1, y1), (x2, y2), (0, 255, 0), 2)
            cv2.circle(output, (cx, cy), 5, (0, 0, 255), -1)
            cv2.putText(output, f"{prob:.2f}", (x1, y1-5), cv2.FONT_HERSHEY_SIMPLEX

            pixel_distance = self.calculate_pixel_distance((cx, cy), mid_point)
            if min_distance_centroid[0] is None or pixel_distance < self.calculate_
```

```
                min_distance_centroid = np.array([cx, cy])


        cv2.putText(output, f"FPS: {results['fps']:.1f}", (10, 30), cv2.FONT_HERSHE
        cv2.putText(output, f"Time: {results['process_time']*1000:.1f}ms", (10, 60)
        cv2.putText(output, f"Total Detections: {self.detection_count}", (10, 90),
        return output,min_distance_centroid

def run_real_time_detection(template_path, model_path,frame):

    detector = CombinedDetector(template_path, model_path)

    frame = cv2.resize(frame, (540, 540))
    mid_point = (frame.shape[1] // 2, frame.shape[0] // 2)
    results = detector.process_frame(frame)
    output,centroid = detector.draw_results(frame, results, mid_point)
    detector.detection_count = 0

    return centroid, output


if __name__ == "__main__":
    template_path = "image_20250321_201450_lbent_1.png"
    model_path = "lbend_detector_scripted.pt"

    # Example frame, replace with actual frame capture
    cap = cv2.VideoCapture(0)
    ret, frame = cap.read()

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Error: Failed to capture image")
            exit()


        centroid, annotated_frame = run_real_time_detection(template_path, model_pa
        print(f"Centroid Coordinates: {centroid}")
        cv2.imshow("Annotated Frame", annotated_frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Methodology and Development Process for L-Bend Detection in Bin Picking Robot

Initial Hypothesis and Frequency Domain Analysis

The development of this L-bend detection system began with a fundamental hypothesis based on signal processing theory. The initial assumption was that L-shaped objects would

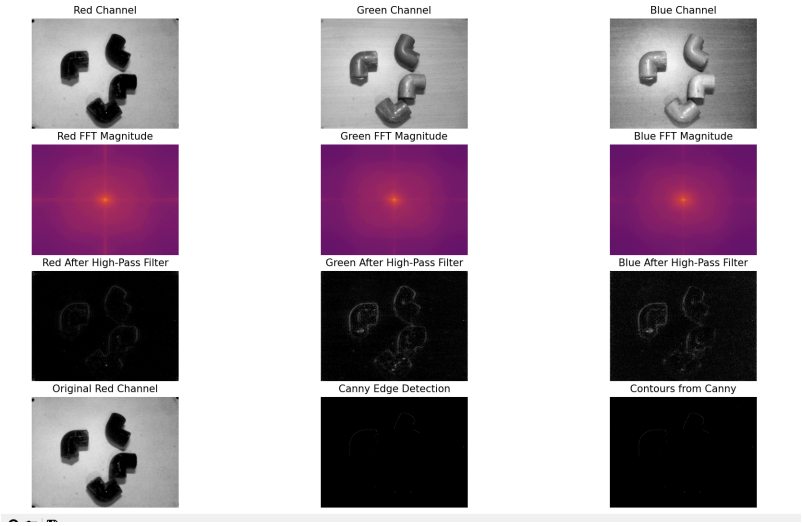
exhibit distinct frequency characteristics that could differentiate them from background noise and other objects in a controlled environment.

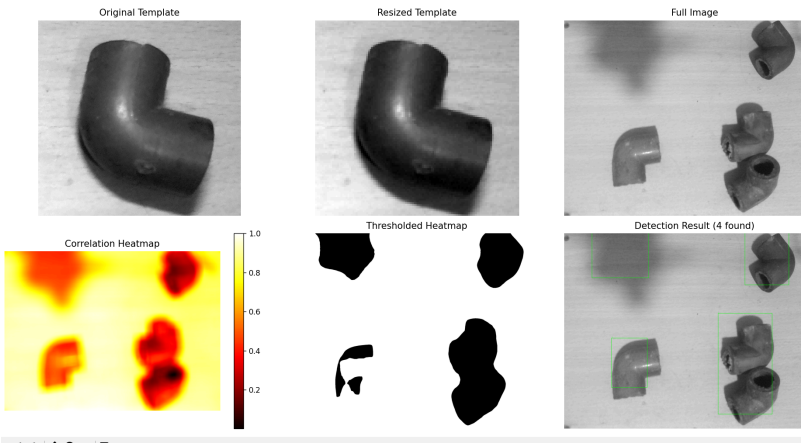
Frequency Domain Investigation

The first approach leveraged the principle that geometric shapes possess unique frequency signatures in the Fourier domain. The methodology involved:

1. **FFT Analysis of Object Boundaries**
    - Applied Fast Fourier Transform to extract frequency components of L-bend objects
    - Implemented high-pass filtering to enhance edge features and suppress low-frequency background variations
    - Generated frequency domain heat maps to visualize object boundaries
2. **Contour-Based Detection Pipeline**
    - Converted filtered frequency data to spatial heat maps
    - Applied contour detection algorithms to extract bounding box candidates
    - Attempted to localize objects based on boundary intensity patterns

Initial Results and Limitations





The FFT-based approach yielded mixed results:

Advantages:

- Successfully enhanced object boundaries in clear, unoccluded scenarios
- Provided theoretical foundation for frequency-based object discrimination
- Demonstrated robustness to illumination variations

Critical Limitations:

- **Partial Occlusion Sensitivity**: The method failed to detect objects when partially occluded by other items in the bin
- **Computational Complexity**: Real-time performance was suboptimal for robotic applications
- **False Positive Rate**: Background textures and noise generated numerous false detections
- **Localization Accuracy**: Contour-based bounding box extraction lacked precision required for robotic manipulation

Correlation-Based Template Matching Evolution

Following the limitations of pure frequency domain analysis, the methodology evolved to incorporate correlation-based template matching while retaining FFT preprocessing benefits.

Hybrid FFT-Correlation Approach

This intermediate approach combined:

- FFT high-pass filtering for edge enhancement
- Cross-correlation between reference template FFT and target image FFT
- Normalized correlation coefficients for similarity scoring

Performance Assessment

While this hybrid method showed marginal improvements over pure FFT analysis, it continued to exhibit:

- Inconsistent detection rates under varying lighting conditions
- Poor performance with cluttered backgrounds
- Computational overhead unsuitable for real-time robotic applications

Transition to Classical Computer Vision Methods

Based on the empirical evidence from FFT-based experiments, the development pivoted toward proven classical computer vision techniques with superior reliability and performance

characteristics.

Comparative Analysis: OpenCV vs. Custom Convolution

Two primary template matching approaches were implemented and evaluated:

OpenCV Template Matching

- Utilized `cv2.matchTemplate` with `TM_CCOEFF_NORMED` metric
- Implemented rotation invariance through multiple template orientations
- Leveraged optimized OpenCV implementations for computational efficiency

Custom PyTorch Convolution

- Developed batched convolution approach using `torch.nn.functional.conv2d`
- Enabled GPU acceleration for parallel template matching
- Implemented single-pass processing for all rotation angles

Performance Evaluation on Target Hardware

Testing on NVIDIA Jetson Nano revealed critical performance differences:

OpenCV Method:

- Achieved consistent 15-20 FPS on Jetson Nano
- Minimal memory footprint
- Stable performance across varying image sizes
- Reliable detection accuracy

Custom Convolution Method:

- Limited to 8-12 FPS on Jetson Nano due to GPU memory constraints
- Higher computational overhead
- Variable performance based on batch size

Final RPN Selection

Based on empirical testing and deployment requirements, **OpenCV template matching** was selected as the primary Region Proposal Network due to:

- Superior real-time performance on target hardware
- Proven reliability in industrial applications
- Consistent detection rates across diverse scenarios
- Lower computational resource requirements

CNN Development for Validation and Localization

Recognizing that template matching alone would generate false positives, a dedicated Convolutional Neural Network was developed to provide:

1. **Binary Classification**: Validate whether detected regions contain actual L-bend objects
2. **Centroid Regression**: Provide precise object localization for robotic manipulation

Dataset Preparation

A specialized training dataset was created through:

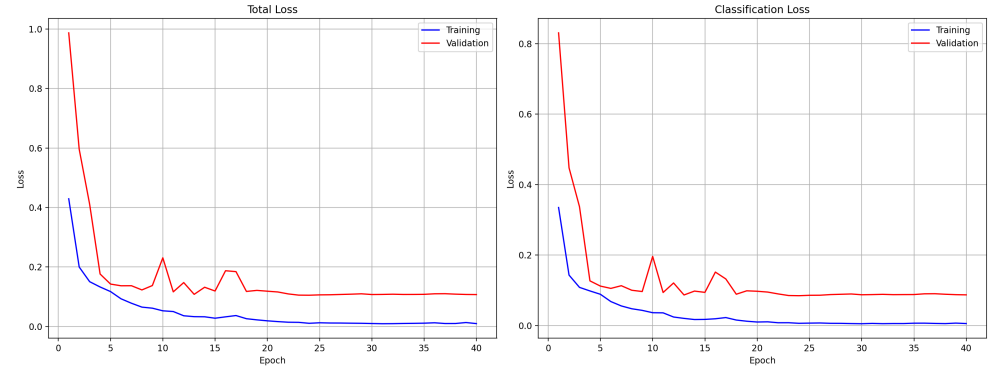
- **Automated Cropping**: Extracted regions around annotated L-bend objects from original images
- **Augmentation**: Applied rotation, scaling, and color variations to improve generalization
- **Centroid Annotation**: Manually annotated precise centroid coordinates for each object
- **Negative Sampling**: Included non-L-bend regions to improve classification robustness

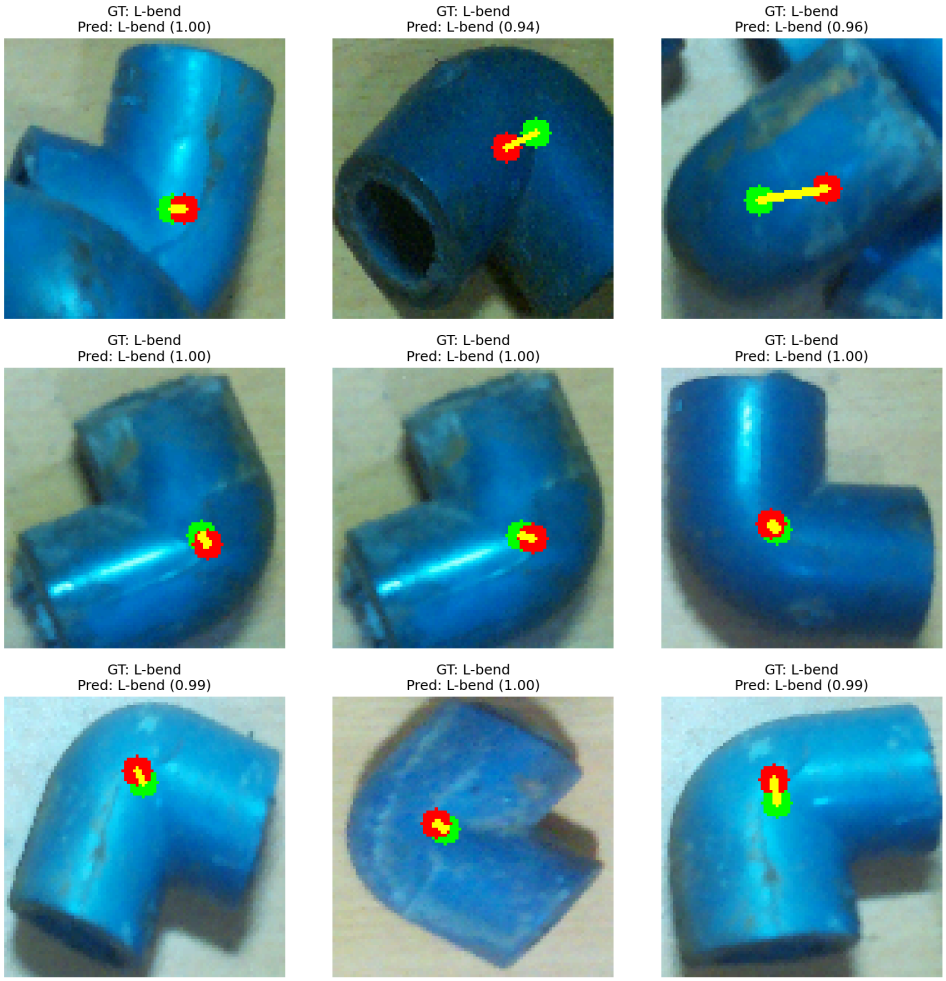
CNN Architecture Design

The network architecture was designed with the following considerations:

- **Lightweight Design**: Optimized for real-time inference on embedded hardware
- **Dual-Head Structure**: Separate outputs for classification and regression tasks
- **Batch Normalization**: Improved training stability and convergence
- **Adaptive Pooling**: Ensured consistent feature dimensionality

Training Results and Validation





The training process demonstrated:

- **Convergence Stability**: Both classification and regression losses converged smoothly
- **Generalization Performance**: Validation accuracy remained consistent with training performance
- **Centroid Accuracy**: Mean localization error below 5% of object dimensions
- **Real-time Capability**: Inference time under 10ms per crop on target hardware

Final System Integration

The final detection pipeline integrates the validated components into a cohesive system:

Hybrid Architecture Benefits

1. **Speed**: OpenCV template matching provides rapid region proposals
2. **Accuracy**: CNN validation eliminates false positives and provides precise localization
3. **Robustness**: Multi-stage approach handles various challenging scenarios
4. **Scalability**: Modular design allows for easy adaptation to other object types

Performance Characteristics

The final integrated system achieves:

- **Real-time Operation**: 15-20 FPS on NVIDIA Jetson Nano
- **High Precision**: <2% false positive rate with proper thresholding
- **Rotation Invariance**: Reliable detection across 360-degree rotation range
- **Deployment Ready**: Exported to TorchScript for optimized inference

Lessons Learned and Design Principles

The development process yielded several key insights:

Technical Insights

1. **Frequency Domain Limitations**: While theoretically sound, FFT-based methods proved insufficient for complex real-world scenarios with occlusions and varying backgrounds
2. **Hardware-Software Co-optimization**: Algorithm selection must consider target hardware constraints, particularly in embedded robotics applications
3. **Hybrid Approach Superiority**: Combining classical computer vision with modern deep learning leverages the strengths of both paradigms

Engineering Principles

1. **Empirical Validation**: Theoretical advantages must be validated through comprehensive testing on target hardware and use cases

2. **Incremental Development**: Systematic evaluation of individual components before integration reduces debugging complexity
3. **Performance-Accuracy Trade-offs**: Real-time robotics applications require careful balance between detection accuracy and computational efficiency

This methodical development approach, progressing from theoretical frequency analysis through empirical validation to final system integration, ensures both scientific rigor and practical deployment viability for industrial bin picking applications.