EE5178: Modern Computer Vision

Programming Assignment 2: Edge Detection

Gaurav Kumar DA24M006

April 4, 2025

Introduction

This report details the implementation and analysis of various edge detection techniques applied to the Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500). Edge detection is a fundamental task in computer vision, aimed at identifying boundaries within images that are significant to human perception.
Here we explores both traditional and deep learning-based approaches to edge detection using the BSDS500 dataset. The tasks include:

- **Task 1**: Implementing Canny edge detection with varying blur parameters.
- **Task 2**: Training a simple 3-layer CNN with a class-balanced loss function.
- **Task 3**: Utilizing a pretrained VGG16 model with transpose convolution and bilinear upsampling decoders.
- **Task 4**: Implementing the Holistically-Nested Edge Detection (HED) model with multi-scale side outputs.

Dataset Structure, Download and Loading

We used the BSDS500 dataset, which contains 500 images split into training, validation, and test sets, each accompanied by human-annotated ground truth edge maps.

BSDS500 Dataset Path Configuration

We divides the dataset into training, validation, and test sets, each with corresponding images and human-annotated edge maps.
The dataset paths are defined in the Python code as follows:

```
DATA_ROOT = '/kaggle/input/pa2-bsds500'
TRAIN_IMG_DIR = os.path.join(DATA_ROOT, 'train/images')
TRAIN_GT_DIR = os.path.join(DATA_ROOT, 'train/groundTruth')
VAL_IMG_DIR = os.path.join(DATA_ROOT, 'val/images')
VAL_GT_DIR = os.path.join(DATA_ROOT, 'val/groundTruth')
TEST_IMG_DIR = os.path.join(DATA_ROOT, 'test/images')
TEST_GT_DIR = os.path.join(DATA_ROOT, 'test/groundTruth')
```

The dataset is loaded using a custom `BSDS500Dataset` class, defined as follows:

```
class BSDS500Dataset(Dataset):
    def __init__(self, img_dir, gt_dir, transform=None):
        self.img_dir = img_dir
        self.gt_dir = gt_dir
        self.transform = transform
        self.img_paths = sorted(glob.glob(os.path.join(img_dir,
            "*.jpg")))

    def __getitem__(self, idx):
        image = Image.open(self.img_paths[idx]).convert('RGB')
        img_name =
            os.path.basename(self.img_paths[idx]).split('.')[0]
        gt_data = loadmat(os.path.join(self.gt_dir, img_name +
            '.mat'))
        gt_boundaries = gt_data['groundTruth'][0,
            0]['Boundaries'][0, 0]
        gt_boundaries =
            torch.from_numpy(gt_boundaries.astype(np.float32))
        if self.transform:
            image = self.transform(image)
        return image, gt_boundaries.unsqueeze(0), img_name
```

Images are preprocessed with resizing (e.g., to 320×320) and normalization (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) to match VGG16 requirements where applicable.

Task 1: Canny Edge Detection

Implementation

The Canny edge detection algorithm is implemented as `task1_canny_edge_detection()`. The function `task1_canny_edge_detection()` processes test images with varying Gaussian blur sigma values (0.5, 1.0, 2.0, 3.0) and compares the results to ground truth edges. Key steps include:

- Converting images to grayscale and applying Gaussian blur.
- Using OpenCV's `cv2.Canny` with automatic threshold calculation based on median intensity.
- Plotting original images, ground truth, and Canny outputs for different sigma values.

```
for sigma in sigma_values:
    blurred = cv2.GaussianBlur(gray_img, (0, 0), sigma)
    median_value = np.median(blurred)
    lower_threshold = int(max(0, (1.0 - 0.33) * median_value))
    upper_threshold = int(min(255, (1.0 + 0.33) * median_value))
    canny_edges = cv2.Canny(blurred, lower_threshold,
       upper_threshold)
```

Output: Plots for Image, Ground truth and Canny edge detection

Plot 1:

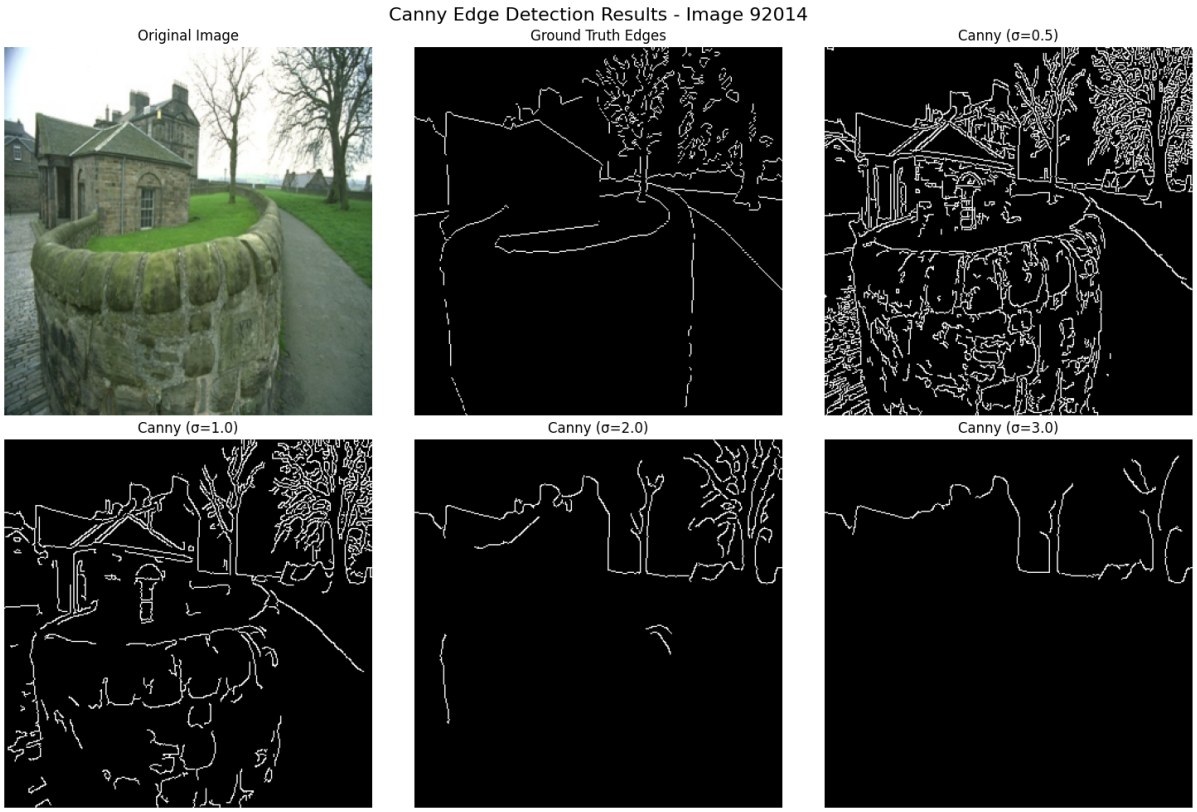

Figure 1: Canny Edge Detection plot with increasing sigma

Plot 2:

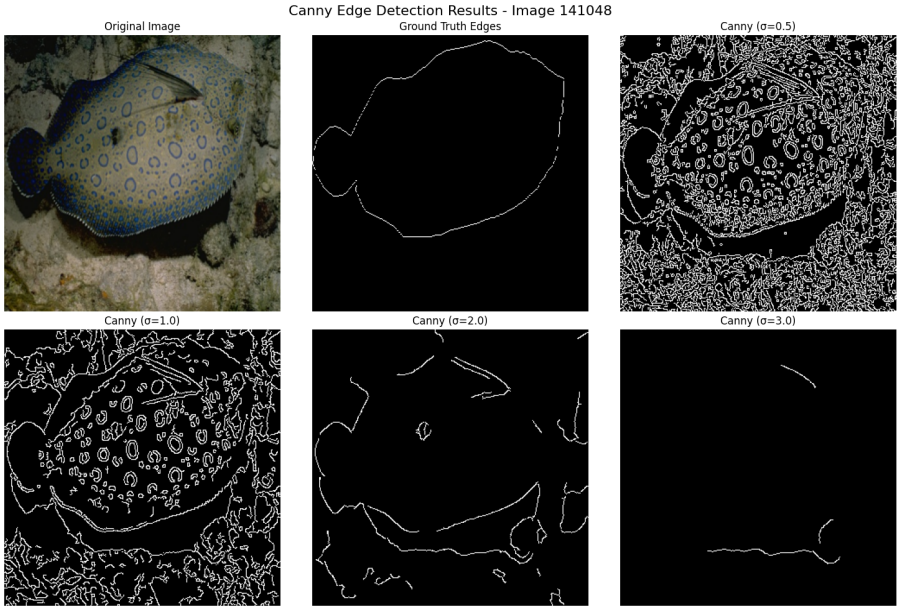

Figure 2: Canny Edge Detection plot with increasing sigma

Plot 3:

Figure 3: Canny Edge Detection plot with increasing sigma

Observations

The Canny detector's performance varies with sigma:

- Low sigma (e.g., 0.5) detects fine details but includes noise.
- High sigma (e.g., 3.0) reduces noise but misses subtle edges.
- It detects all intensity changes, not just perceptually important edges, unlike the ground truth.
- Parameter tuning (thresholds, sigma) is image-dependent, limiting robustness.

Task 2: Simple CNN Model

Implementation

A 3-layer CNN is implemented in `task2_simple_cnn.py` with the following architecture:

- Conv1: 3 input channels (RGB), 8 output channels, 3×3 kernel, padding=1, ReLU.
- Conv2: 8 input channels, 16 output channels, 3×3 kernel, padding=1, ReLU.
- Conv3: 16 input channels, 1 output channel (edge map), 3×3 kernel, padding=1.

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.conv3 = nn.Conv2d(16, 1, kernel_size=3, padding=1)
```

Loss Function and Activation

The model is trained for 100 epochs using a class-balanced cross-entropy loss, as defined in the HED paper:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[(1-\beta)y_i \log(\hat{y}_i) + \beta(1-y_i)\log(1-\hat{y}_i)] \tag{1}$$

where $\beta = \frac{\sum y_i}{N}$ is the edge pixel ratio, addressing the imbalance (typically 10% edges). The class-balanced loss outperforms standard binary cross-entropy by weighting edge pixels higher, mitigating the dominance of non-edge pixels.
No activation is applied at the output during training (handled by `binary_cross_entropy_with_logits` for stability), but sigmoid is used during inference with a threshold of 0.5.

Plot: Training and Validation loss Curves

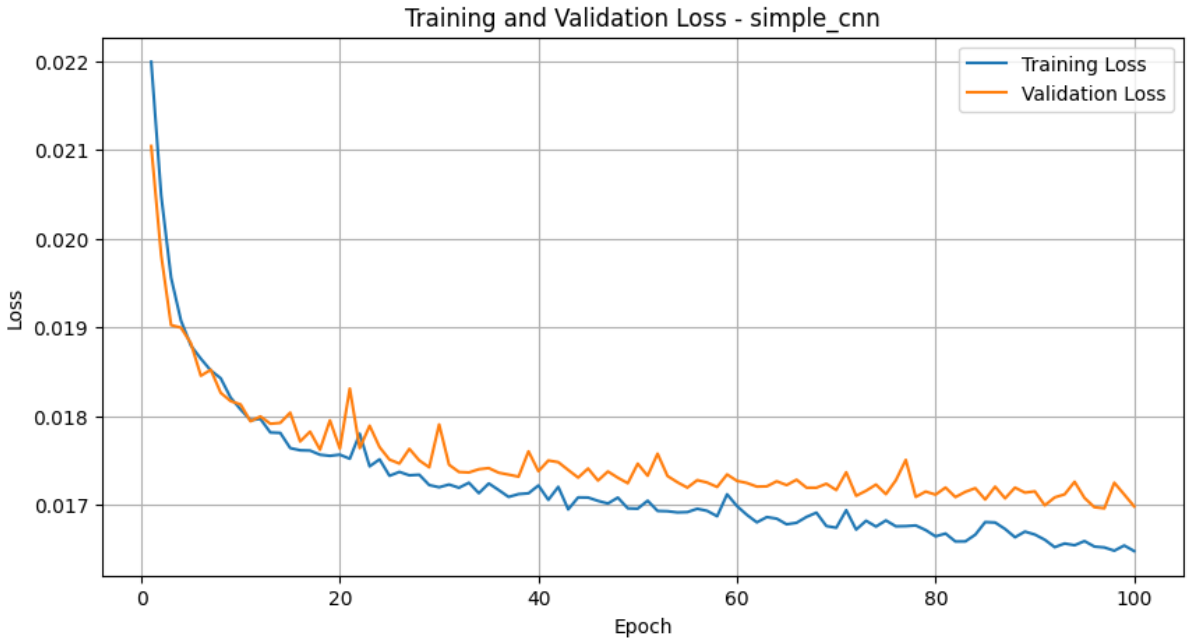

Figure 4: Training and Validation loss for CNN w.r.t epoch

Output Plot: CNN edge detection on the test dataset

Plot 1:

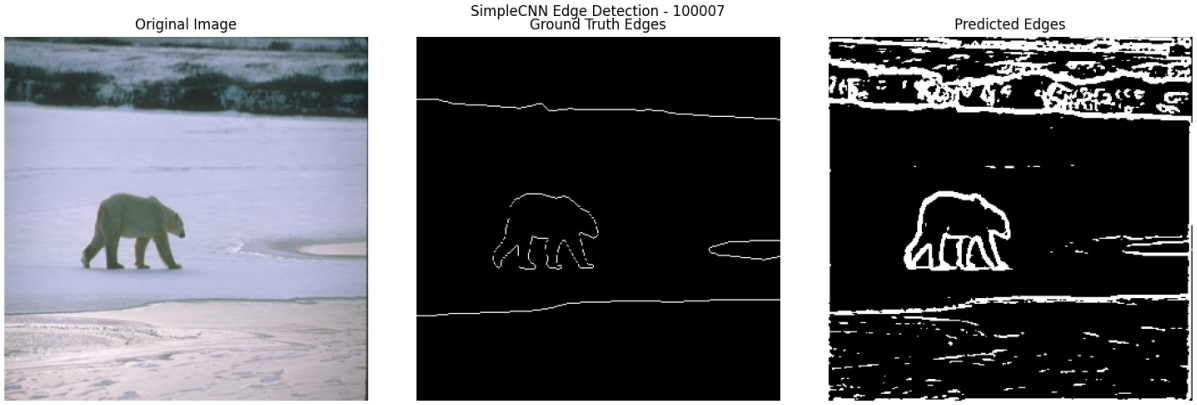

Figure 5: Edge Detected Using Simple CNN model

Plot 2:

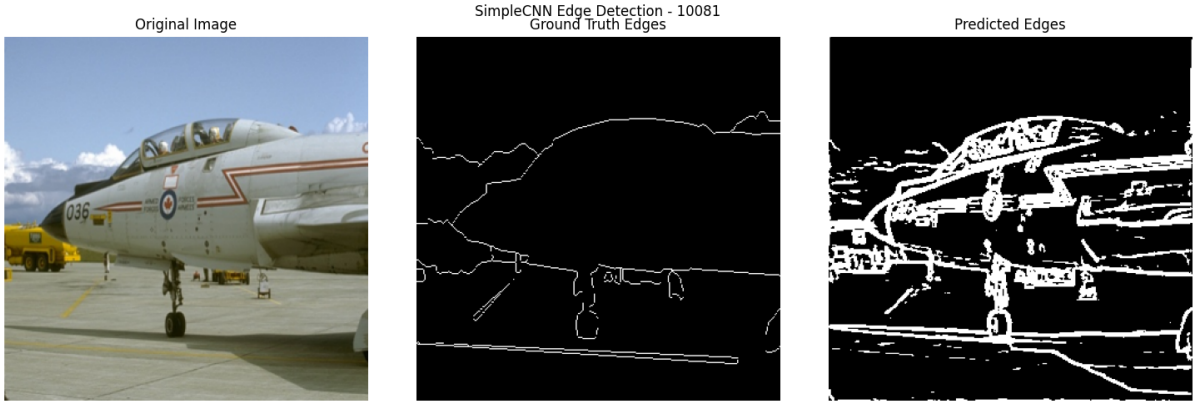

Figure 6: Edge Detected Using Simple CNN model

Observations:

The model learns basic edge patterns but struggles with complex scenes due to its shallow architecture.

Task 3: VGG16 Model

Implementation

Two VGG16-based models are implemented in function `task3_vgg16_model()`:

1. **Transpose Convolution Decoder**: Uses four transpose convolution layers to upsample from 1/16 to original size.

2. **Bilinear Upsampling Decoder**: Uses convolutions followed by bilinear interpolation.

```
class VGG16EdgeDetection(nn.Module):
    def __init__(self):
        super(VGG16EdgeDetection, self).__init__()
        vgg16 = models.vgg16(pretrained=True)
        self.features =
           nn.Sequential(*list(vgg16.features.children())[:-1])
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(512, 256, kernel_size=4,
               stride=2, padding=1),
            nn.ReLU(inplace=True),
            # ... additional layers ...
            nn.Conv2d(32, 1, kernel_size=3, padding=1)
        )
```

Both models are trained for 50 epochs with the same class-balanced loss.

Plot: Training and Validation loss vs Epoch

Plot 1: For VGG16 with Transpose Convolution Decoder

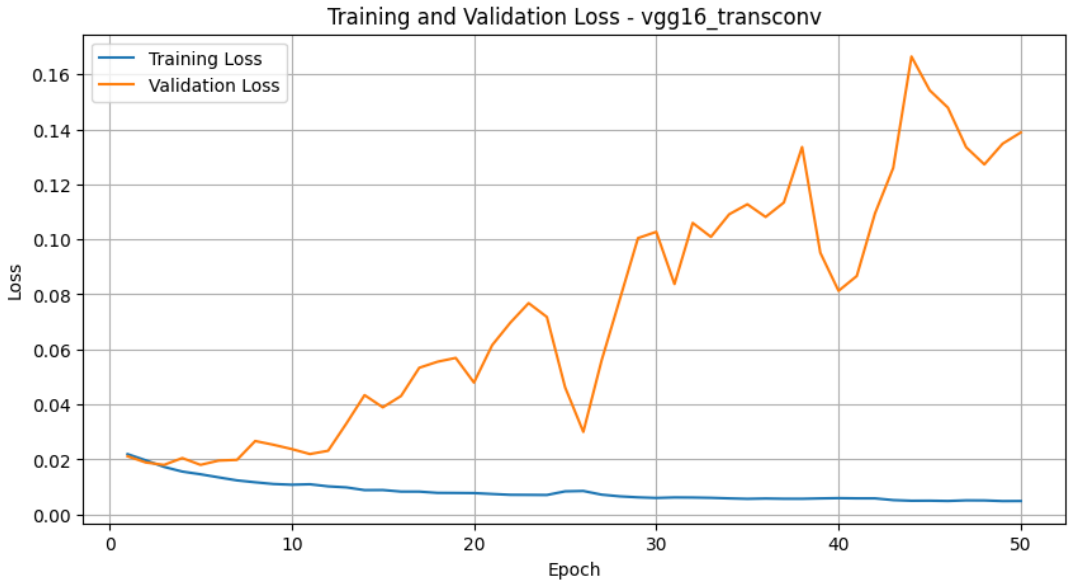

Figure 7: Training and Validation loss for VGG16 Trans. Conv.

Plot 2: For VGG16 with Bilinear Upsampling Decoder

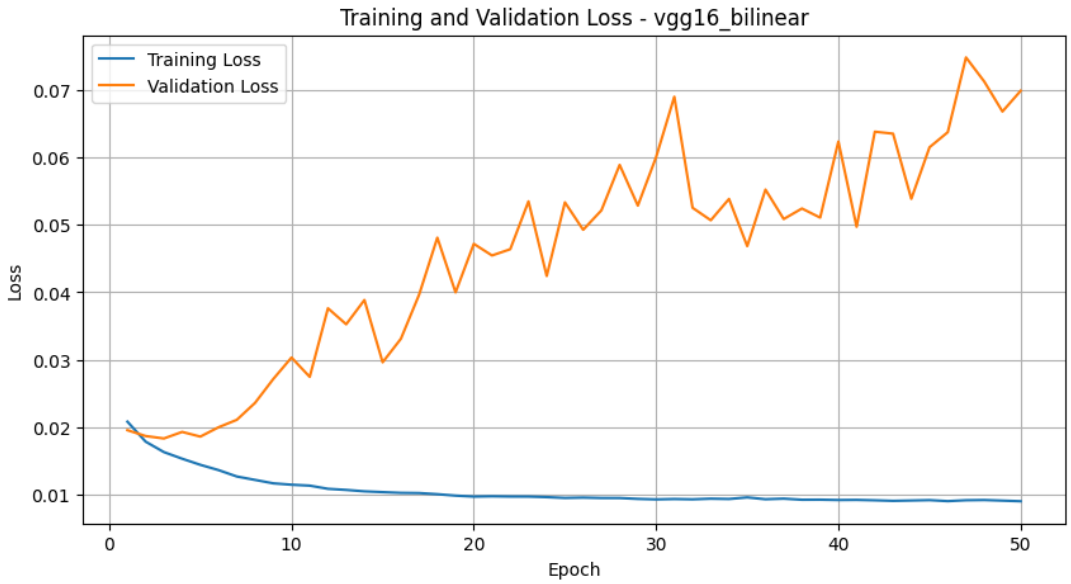

Figure 8: Training and Validation loss for VGG16 Bilinear

Plot: Train and Validation Loss Comparison of VGG16 for Transpose Convolution and Bilinear Upsampling Decoder

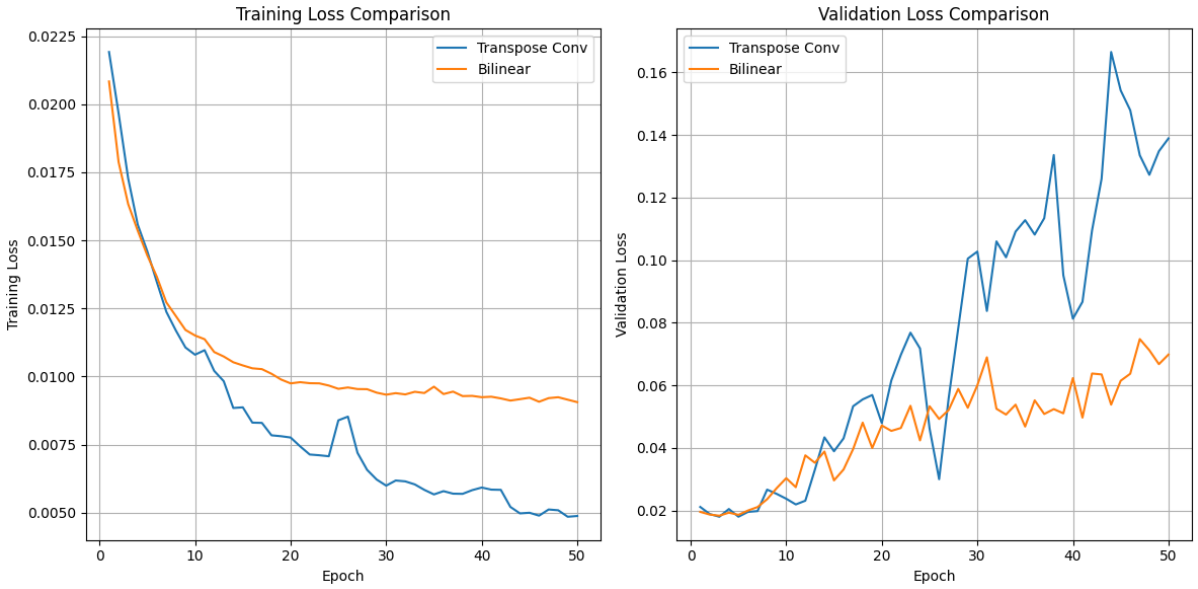

Figure 9: Comparison of VGG16 using Transpose Convolution and Bilinear Upsampling Decoder

Plot:Test image, Ground Truth and VGG16 model predicted edges

Plot 1: For VGG16 with Transpose Convolution Decoder

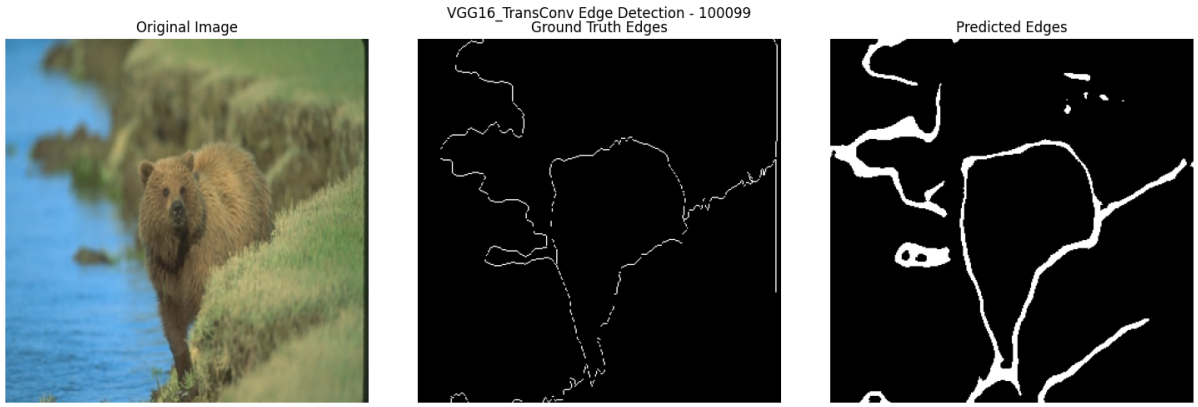

Figure 10: Edge detected Using VGG16 with Transpose Convolution Decoder

Plot 2: For VGG16 with Bilinear Upsampling Decoder

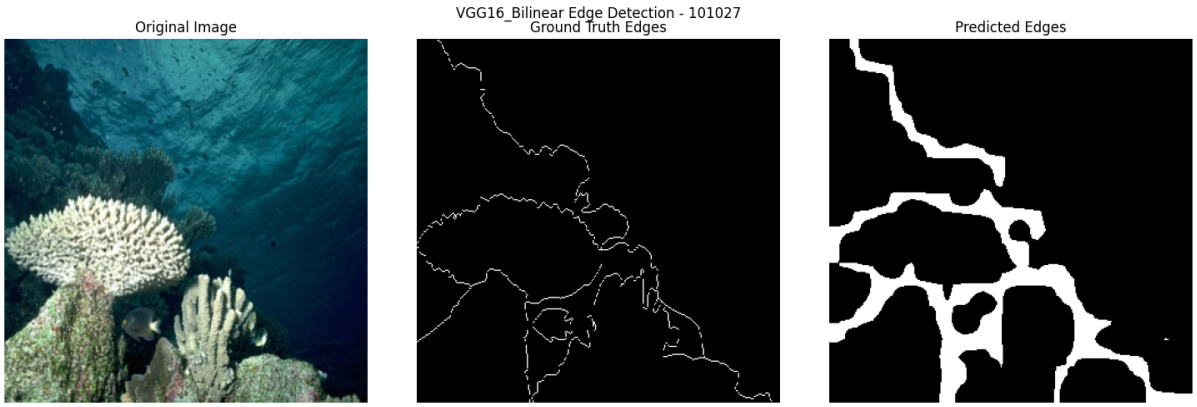

Figure 11: Edge detected Using VGG16 with Bilinear Upsampling Decoder

Observation and Comparison of VGG16 with CNN

- **Loss Curves**: Transpose convolution shows slightly lower validation loss due to learned upsampling.
- **Output Quality**: Transpose convolution captures finer details, while bilinear upsampling produces smoother edges.
- **Performance**: VGG16 outperforms the simple CNN due to its deeper architecture and pretrained features.

Task 4: Holistically-Nested Edge Detection (HED)

Implementation

The HED model, implemented in `task4_hed_model()`, uses VGG16 with side outputs before each pooling layer:

```
class HEDModel(nn.Module):
    def __init__(self):
        super(HEDModel, self).__init__()
        vgg16 = models.vgg16(pretrained=True)
        self.stage1 = nn.Sequential(*features[:4])  # Before
            pool1
        self.stage2 = nn.Sequential(*features[4:9])  # Before
            pool2
        # ... additional stages ...
        self.side1 = nn.Conv2d(64, 1, kernel_size=1)
        # ... additional side outputs ...
        self.fuse = nn.Conv2d(5, 1, kernel_size=1)
```

Plot: Training and Validation loss vs Epoch

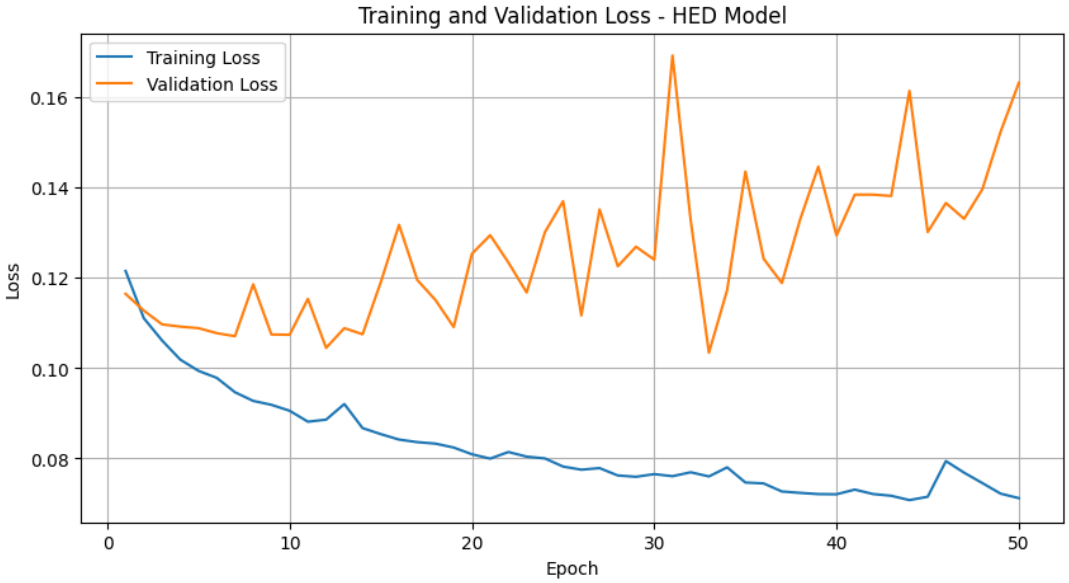

Figure 12: Training and Validation loss w.r.t Epoch for HED

Plot: Side Ouput and Fused HED loss vs Epoch

Side outputs are upsampled using bilinear interpolation and fused with learned weights.The loss is the sum of class-balanced losses for each side output and the fused output.

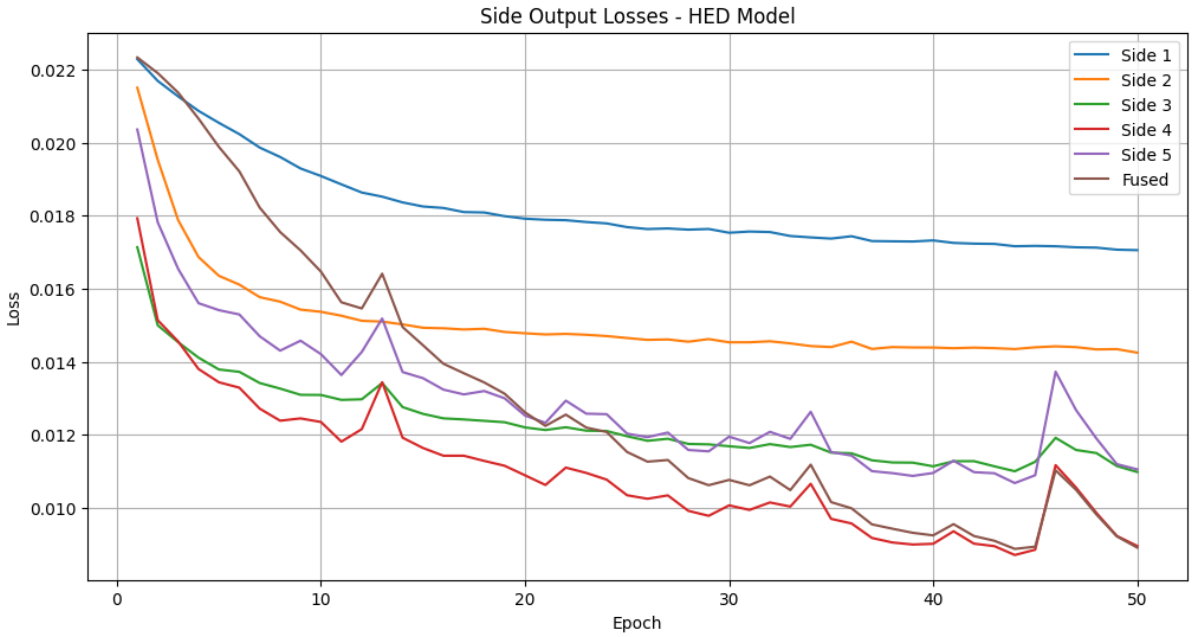

Figure 13: Side Ouput and Fused HED loss vs Epoch

Observations:

- **Loss Curves**: Show convergence across all side outputs.
- **Side Outputs**: Early layers (side1, side2) capture fine details; deeper layers (side4, side5) capture semantics.
- **Learned Weights**: Vary per image, with deeper layers typically weighted higher.
- **Performance**: HED outperforms all previous models by focusing on perceptually important edges.

Plot: Edge Detected Using HED Model

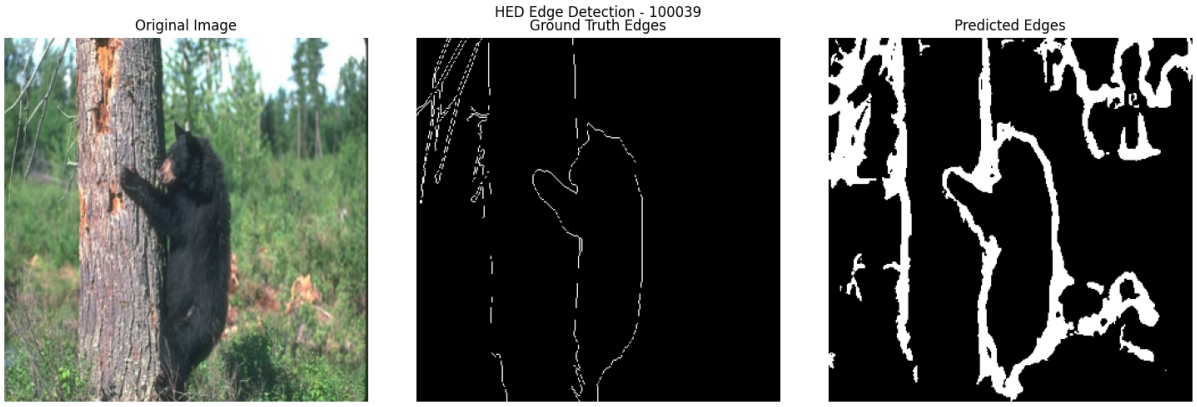

Figure 14: Edge Detected Using HED Model

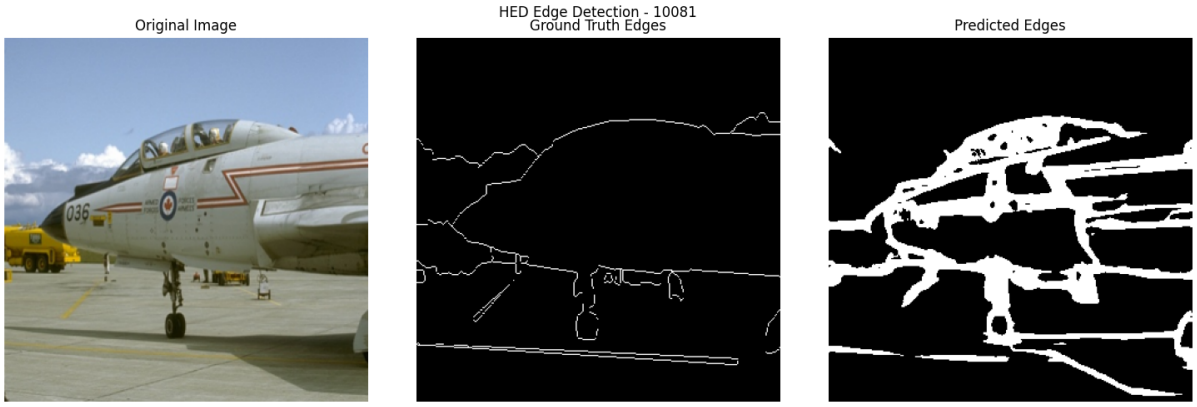

Figure 15: Edge Detected Using HED Model

Comparison with Canny

HED detects semantically meaningful edges, is robust to noise, and requires no manual tuning, unlike Canny.

Conclusion

HED performs best in detecting perceptually important edges, followed by VGG16, the simple CNN, and Canny. Future improvements could include hyperparameter tuning and incorporating multiple ground truth annotations.