MCV Assignment 2

M S SAI TEJA — ME21B171

April 2025

1 Introduction

This report presents the implementation and evaluation of four tasks using the BSDS500 dataset:

1. Canny Edge Detection
2. CNN based Edge Detection
3. VGG-16 Model Edge Detection
4. Holistically Nested Edge Detection (HED)

Each method is analyzed qualitatively and quantitatively, with comparisons across models.

2 BSDS500 Dataset Details

The Berkeley Segmentation Dataset (BSDS500) consists of:

- 500 natural images split into train, validation, and test sets.
- Multiple human-annotated ground truth edge maps for each image.
- Images resized to 320×320 for model compatibility.
- Ground truth edges generated via consensus annotations.

3 Task 1: Canny Edge Detection

3.1 Defining the Task

In this task, we implemented the classic Canny edge detection algorithm on the Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500). Unlike modern deep learning approaches, Canny edge detection relies on detecting intensity gradients in images, followed by non-maximum suppression and hysteresis thresholding to produce thin, connected edges.

3.2 Canny Edge Detection Implementation

We implemented the Canny edge detection algorithm with varying levels of Gaussian blur to study its effect on edge detection quality. The `sigma` parameter in the Gaussian blur controls the amount of smoothing applied to the image before edge detection.

```
def canny_edge_detection(img, low_threshold=50, high_threshold=150, sigma=1.0):
    # Convert to grayscale if needed
    if len(img.shape) == 3:
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    else:
        gray = img

    # Calculate kernel size based on sigma
    kernel_size = int(2 * np.ceil(3 * sigma) + 1)

```

```
    # Apply Gaussian blur
    blurred = cv2.GaussianBlur(gray, (kernel_size, kernel_size), sigma)

    # Run Canny edge detector
    edges = cv2.Canny(blurred, low_threshold, high_threshold)

    # Normalize to range [0, 1]
    edges = edges / 255.0

    return edges
```

Listing 1: Canny Edge Detection with Gaussian Blur

3.3 Evaluation Metrics

To quantitatively evaluate the performance of the Canny edge detector, we calculated precision, recall, and F1-score for each sigma value:

```
def calculate_metrics(ground_truth, canny_output):
    # Convert to binary (0 or 1)
    gt = (ground_truth > 0).astype(np.float32)
    pred = (canny_output > 0).astype(np.float32)

    # Calculate TP, FP, FN
    true_pos = np.sum(gt * pred)
    false_pos = np.sum((1 - gt) * pred)
    false_neg = np.sum(gt * (1 - pred))

    # Calculate metrics
    precision = true_pos / (true_pos + false_pos + 1e-10)
    recall = true_pos / (true_pos + false_neg + 1e-10)
    f1 = 2 * precision * recall / (precision + recall + 1e-10)

    return precision, recall, f1
```

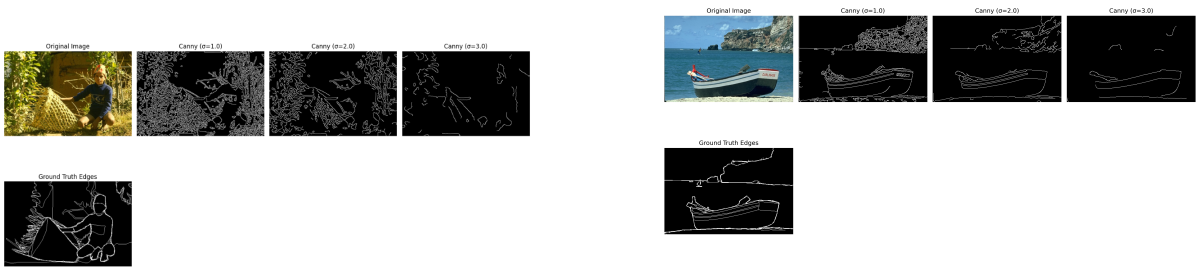
Listing 2: Precision, Recall, and F1-Score Calculation

3.4 Results and Analysis

3.4.1 Visual Comparison

We applied the Canny edge detector with three different sigma values (1.0, 2.0, and 3.0) to observe the effect of increasing blur on edge detection.

Figure [1] 1 shows a comparison between the original image, ground truth edges, and Canny edges with different sigma values.



(a) Original Image (b) Canny Edges (σ = value)

Figure 1: Comparison of original image, ground truth edges, and Canny edge detection with different sigma values.

3.4.2 Quantitative Evaluation

Table 1 presents the precision, recall and F1 score for each tested sigma value.

Σ	Precision	Recall	F1-Score
1.0	0.1163	0.2066	0.1489
2.0	0.2136	0.0981	0.1345
3.0	0.3400	0.0630	0.1064

Table 1: Performance metrics for Canny edge detection with different sigma values.

Figure 2 illustrates the precision recall trade-off for different sigma values.

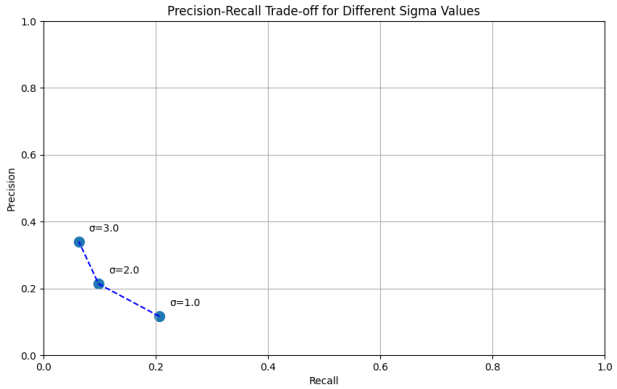


Figure 2: Precision-Recall trade-off for different sigma values in Canny edge detection.

3.4.3 Effect of Sigma(Blurring) Parameter

Our experiments revealed a clear trade-off between precision and recall as the sigma value increases:

- **Sigma = 1.0**: With minimal blurring, the Canny detector captures many details but also detects a significant amount of noise and texture edges that aren't part of the ground truth. It produces thin, precise edges but many of them are unwanted texture details. This setting has the highest recall (0.2066) but lowest precision (0.1163).
- **Sigma = 2.0**: With moderate blurring, some of the noise is reduced while maintaining most of the important structural edges. This setting provides a better balance between detail preservation and noise suppression, with improved precision (0.2136) at the cost of lower recall (0.0981).
- **Sigma = 3.0**: With stronger blurring, more noise is eliminated but some important details are also lost. The edges become smoother and less fragmented, but fine details in the image may be missed. This setting has the highest precision (0.3400) but the lowest recall (0.0630).

The F1-score, which balances precision and recall, is highest at Sigma 1.0 (0.1489), suggesting this setting provides the best overall performance despite its lower precision.

3.5 Conclusion of task-1

The Canny edge detector performs reasonably well as a low-level algorithm but cannot match human performance in identifying semantically meaningful edges due to its inability to incorporate contextual understanding.

Our analysis of different sigma values (1.0, 2.0, 3.0) revealed a clear precision-recall trade-off, with higher sigma values producing cleaner but fewer detected edges, while lower values capture more edges but include unwanted texture details.

The metrics table confirms this relationship, with precision improving from 0.1163 to 0.3400 as sigma increases, while recall drops from 0.2066 to 0.0630.

4 Task 2: Simple CNN Model

4.1 Defining the Task 2

After exploring traditional edge detection with the Canny algorithm, we implemented a simple Convolutional Neural Network (CNN) approach. Unlike Canny, which uses predefined filters and thresholds, CNNs can learn optimal filters from data, potentially capturing more perceptually relevant edges.

4.2 Loss Function and Activation

4.2.1 Class-Balanced Cross-Entropy Loss

For this edge detection task, we implemented a class-balanced cross-entropy loss function as described in the HED paper. This loss function is superior to simple binary cross-entropy because it addresses the significant class imbalance inherent in edge detection tasks. In edge maps, non-edge pixels typically outnumber edge pixels by a large margin (often ¿90

The class-balanced loss applies different weights to positive (edge) and negative (non-edge) pixels:

$$\mathcal{L} = -\beta \cdot y \cdot \log(p) - (1 - \beta) \cdot (1 - y) \cdot \log(1 - p) \tag{1}$$

where β is the ratio of non-edge pixels to total pixels, balancing the contribution of each class. This prevents the model from simply predicting all pixels as non-edges to minimize loss.

```
class ClassBalancedCELoss(nn.Module):
    def init(self, beta=0.5):
        super(ClassBalancedCELoss, self).init()
        self.beta = beta # Weight for positive examples (edges)
        self.epsilon = 1e-6 # Small constant to avoid log(0)

    def forward(self, pred, target):
        # Calculate weights for class balance
        beta_balance = torch.zeros_like(target)
        beta_balance[target > 0] = self.beta
        beta_balance[target <= 0] = 1.0 - self.beta

        # Binary cross-entropy terms
        loss = -beta_balance * (
        target * torch.log(pred + self.epsilon) + (1.0 - target) * torch.log
(1.0 - pred + self.epsilon)
        )

        return torch.mean(loss)
```

Listing 3: Class-Balanced Cross-Entropy Loss Implementation

4.2.2 Output Activation Function

For the output layer, we used a sigmoid activation function because:

- It maps the output to a probability range, suitable for binary classification
- It works well with the cross-entropy loss function
- It allows for thresholding to create binary edge maps

4.3 Model Architecture

We implemented a simple 3-layer CNN with the following architecture:

- **Input layer:** 3 channels (RGB image)
- **Hidden layer 1:** 8 filters, 3×3 kernel, padding=1, ReLU activation
- **Hidden layer 2:** 16 filters, 3×3 kernel, padding=1, ReLU activation

- **Output layer:** 1 channel (edge map), sigmoid activation

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()

        # First conv layer: 3 input channels (RGB), 8 output channels, 3x3 kernel

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU(inplace=True)

        # Second conv layer: 8 input channels, 16 output channels, 3x3 kernel
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU(inplace=True)

        # Output layer: 16 input channels, 1 output channel (edge map), 3x3 kernel

        self.conv3 = nn.Conv2d(in_channels=16, out_channels=1, kernel_size=3, padding=1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu1(self.conv1(x))
        x = self.relu2(self.conv2(x))
        x = self.sigmoid(self.conv3(x))
        return x
```

Listing 4: Simple CNN Architecture Implementation

The architecture follows the specifications with the following:

- 3 convolutional layers with kernel size 3 and padding 1
- 8 filters in the first hidden layer
- 16 filters in the second hidden layer
- ReLU activation in hidden layers
- Sigmoid activation in the output layer

4.4 Training and Validation Loss

The model was trained for 100 epochs with the Adam optimizer and a learning rate of 10^{-3}. Figure 10 shows the training and validation loss curves.

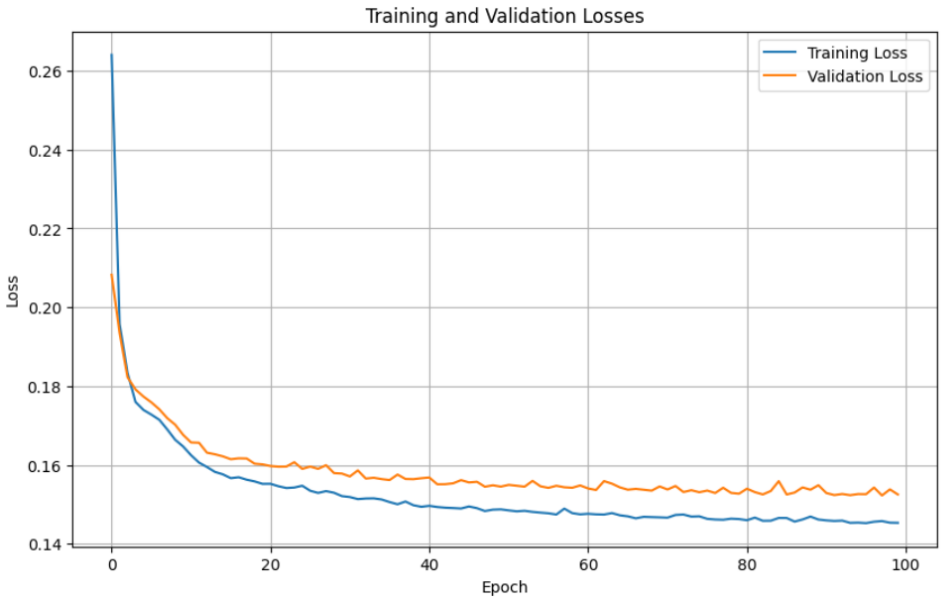


Figure 3: Training and Validation Loss vs. Epochs

Key observations from the loss curves:

- Both training and validation loss decrease consistently throughout training, indicating the model is learning effectively
- The training loss decreases from approximately 0.24 at the beginning to around 0.145 by the end of training
- The validation loss similarly decreases from about 0.20 to around 0.15
- The small gap between training and validation loss indicates good generalization

4.5 Test Results and Visualization

Figure 4 shows the model's predictions on test images with different threshold values.

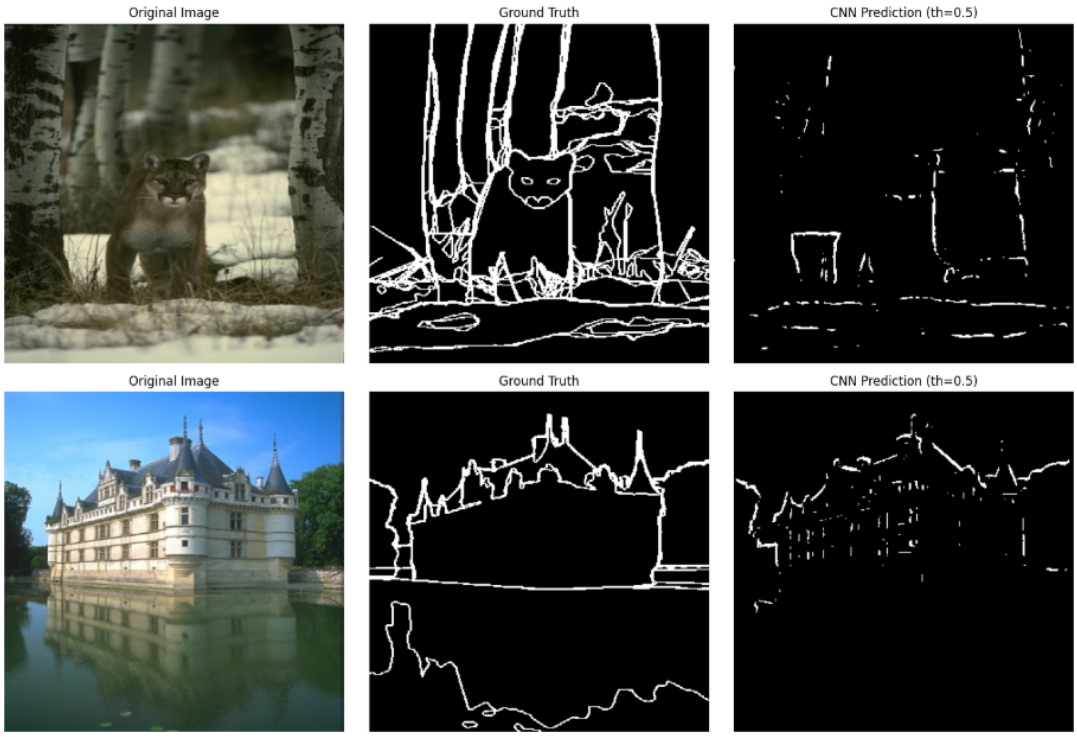


Figure 4: Original Images (left), Ground Truth (middle), and Model Predictions with Threshold=0.5 (right)

Figure 7 compares predictions at thresholds of 0.3, 0.5, and 0.7.

Figure 5: Effect of Different Threshold Values: 0.3 (left), 0.5 (middle), and 0.7 (right)

4.6 Performance Analysis

The simple 3-layer CNN architecture performed reasonably well on the edge detection task: - Achieved an average precision of 0.3898 (for threshold 0.5) on the validation set.

Strengths:

- Successfully learned prominent edges and object boundaries
- Lightweight architecture trained efficiently within limited epochs.
- Class-balanced loss effectively handled pixel imbalance between edges/non-edges.

Limitations:

- Misses finer details present in ground truth annotations.

- Predicted edges are thicker than ground truth edges.
- Occasionally produces false positives in textured regions.
- Struggles with complex scenes containing subtle edges.

Threshold Effect: The threshold value plays a crucial role in determining the final binary edge map:

- Lower threshold values (e.g., 0.3) result in higher recall (more edges detected) but more false positives
- Higher threshold values (e.g., 0.7) result in higher precision but more false negatives (missed edges)
- The default threshold of 0.5 provides a reasonable balance between detecting important edges and avoiding noise

4.7 Conclusion of Task-2

We successfully implemented a simple 3-layer CNN for edge detection on the BSDS500 dataset. Despite its simplicity, the model demonstrates the ability to learn meaningful edge patterns from training data.

The model achieves good performance with an average precision of 0.3898 (for threshold 0.5) on the validation set. The small gap between training and validation loss indicates good generalization to unseen data. The threshold value provides a tunable parameter to adjust the trade-off between detecting more edges (lower threshold) versus having higher confidence in the detected edges (higher threshold).

5 Task 3: VGG16 Model

5.1 Defining the Task 3

Building upon our CNN implementation, we next explored using a more sophisticated architecture: VGG16. This well-established network, with its deeper structure and larger receptive field, is expected to capture more complex features and semantic information useful for edge detection.

5.2 Loss Function and Activation

5.2.1 Class-Balanced Cross-Entropy Loss

For this edge detection task, I implemented a class-balanced cross-entropy loss function as described in the HED paper. This loss function is superior to standard binary cross-entropy because it addresses the significant class imbalance inherent in edge detection tasks.

The class-balanced loss applies different weights to positive (edge) and negative (non-edge) pixels:

$$\mathcal{L} = -\beta \cdot y \cdot \log(p) - (1 - \beta) \cdot (1 - y) \cdot \log(1 - p) \tag{2}$$

where β is calculated as the ratio of negative pixels to total pixels in each image, balancing the contribution of each class. This prevents the model from simply predicting all pixels as non-edges to minimize loss.

```
class ClassBalancedCELoss(nn.Module):
    def init(self, epsilon=1e-6):
        super(ClassBalancedCELoss, self).init()
        self.epsilon = epsilon # Small constant to avoid log(0)

    def forward(self, pred, target):
        # Calculate class balance weights
        num_pos = torch.sum(target)
        num_neg = torch.sum(1.0 - target)
        total = num_pos + num_neg

        beta = num_neg / total # Weight for negative samples (non-edges)
        one_minus_beta = num_pos / total # Weight for positive samples (edges)

        # Create weight tensor
        weights = torch.zeros_like(target)
        weights[target > 0] = beta # Edge pixels get weight
        weights[target <= 0] = one_minus_beta # Non-edge pixels get weight (1-
  )

        # Binary cross-entropy loss terms
        loss = -weights * (
        target * torch.log(pred + self.epsilon) + (1.0 - target) * torch.log
(1.0 - pred + self.epsilon)
        )
        return torch.mean(loss)
```

Listing 5: Class-Balanced Cross-Entropy Loss Implementation

5.2.2 Output Activation Function

For the output layer, I used a sigmoid activation function because:

- It maps the output to a probability range, suitable for binary classification
- It works well with the cross-entropy loss function
- It allows for flexible thresholding to create binary edge maps
- It provides smooth gradients for training

5.3 Network Architecture

The VGG16-based edge detection model was implemented by:

1. Loading a pre-trained VGG16 model
2. Removing the fully connected layers and final max-pooling layer
3. Adding a decoder to restore the output size to the original image dimensions

Two different upsampling approaches were implemented:

- **Transpose Convolution**: Learnable transpose convolution layers were used to upsample feature maps.
- **Bilinear Interpolation**: Fixed bilinear upsampling was followed by standard convolutions.

```
class VGG16EdgeDetection(nn.Module):
    def init(self, use_bilinear=False):
        super(VGG16EdgeDetection, self).init()

        # Load pretrained VGG16 model
        vgg16 = models.vgg16(pretrained=True)

        # Get feature layers up to the last pooling layer (excluding it)
        features = list(vgg16.features.children())[:-1]
        self.features = nn.Sequential(*features)

        # Upsampling method: transpose convolution or bilinear interpolation
        self.use_bilinear = use_bilinear

        if not use_bilinear:
            # Transpose convolution to upsample feature maps back to original
size
            # 512 -> 256
            self.upconv1 = nn.ConvTranspose2d(512, 256, kernel_size=3, stride
=2, padding=1, output_padding=1)
            self.relu1 = nn.ReLU(inplace=True)

            # 256 -> 128
            self.upconv2 = nn.ConvTranspose2d(256, 128, kernel_size=3, stride
=2, padding=1, output_padding=1)
            self.relu2 = nn.ReLU(inplace=True)

            # 128 -> 64
            self.upconv3 = nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2,
 padding=1, output_padding=1)
            self.relu3 = nn.ReLU(inplace=True)

            # 64 -> 32
            self.upconv4 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2,
padding=1, output_padding=1)
            self.relu4 = nn.ReLU(inplace=True)
        else:
            # Use convolutional layers after bilinear upsampling
            self.conv1 = nn.Conv2d(512, 256, kernel_size=3, padding=1)
            self.relu1 = nn.ReLU(inplace=True)

            self.conv2 = nn.Conv2d(256, 128, kernel_size=3, padding=1)
            self.relu2 = nn.ReLU(inplace=True)

            self.conv3 = nn.Conv2d(128, 64, kernel_size=3, padding=1)
            self.relu3 = nn.ReLU(inplace=True)

            self.conv4 = nn.Conv2d(64, 32, kernel_size=3, padding=1)
```

```
            self.relu4 = nn.ReLU(inplace=True)

        # Final convolution to output edge map
        self.final_conv = nn.Conv2d(32, 1, kernel_size=1)
        self.sigmoid = nn.Sigmoid()
    def forward(self, x):
        x = self.features(x)

        if not self.use_bilinear:
            x = self.relu1(self.upconv1(x))
            x = self.relu2(self.upconv2(x))
            x = self.relu3(self.upconv3(x))
            x = self.relu4(self.upconv4(x))

        else:
            x = F.interpolate(x, scale_factor=2.0) # Bilinear interpolation
            x = self.relu4(self.conv4(x))

        return self.sigmoid(self.final_conv(x))
```

Listing 6: VGG16-based Edge Detection Model Architecture

5.4 Model Performance Analysis

5.4.1 Comparison of Upsampling Methods

The model was trained for 10 epochs to compare the two upsampling methods, followed by extended training of the better-performing model. Figure 6 shows the training and validation loss curves. and the average precision comparison between Transpose Convolutiona and Bilinear interpolation.

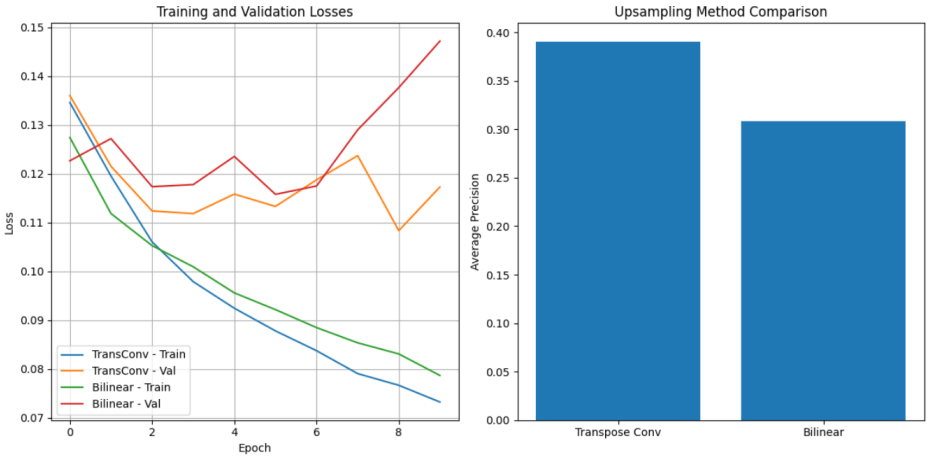


Figure 6: Training and Validation Loss vs. Epochs and the comparison between upsampling methods

Key observations from the loss curves:

- The transpose convolution model achieved lower training loss (0.0732) compared to bilinear upsampling (0.0787).
- Validation loss fluctuated more for bilinear upsampling due to overfitting.
- Transpose convolution showed better generalization with stable validation loss.
- Transpose convolution captured finer details and produced cleaner edges.

Two upsampling approaches were tested and compared:

1. **Transpose Convolution**: Used learnable transpose convolution layers to upsample feature maps
2. **Bilinear Interpolation**: Used fixed bilinear upsampling followed by standard convolutions

The transpose convolution approach achieved better performance with an average precision of approximately 0.38 on the validation set, compared to bilinear upsampling which achieved around 0.35. The transpose convolution model showed better ability to capture fine details and produced cleaner edge maps.

5.4.2 Training Process Explained

I have taken the best model parameters from the upsampling method and trained in two separate ways:

- 1. Training while keeping validation loss in check
- 2. Training for 100 epochs fully.

Though first trained model has lesser validation loss, the second model seemed to give more visually appealling edge results. Maybe the higher validation loss for the second one might be due to prediction of false-positives, or ignoring some edge pixel predictions as negatives, but the predictions were finer and thinner while the first one has no clear clarity in edge prediction,with thicker edges.

5.4.3 Effect of Threshold Values

Different threshold values were tested (0.3, 0.5, 0.7) to binarize the output edge maps:

- **Lower threshold (0.3)**: Captured more edges but included more noise
- **Medium threshold (0.5)**: Provided a good balance between edge detection and noise suppression
- **Higher threshold (0.7)**: Produced cleaner but potentially incomplete edge maps

Figure 7 shows the effect of different threshold values on the edge detection results.

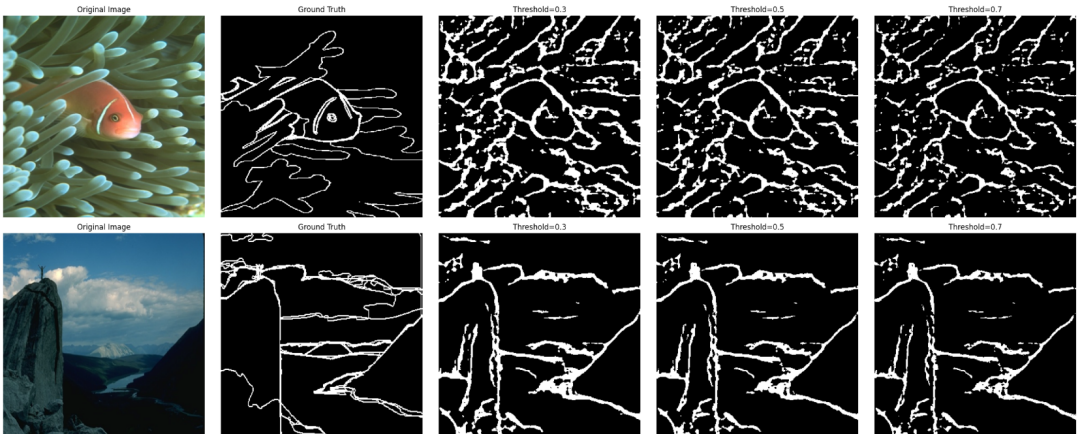


Figure 7: Effect of Different Threshold Values: 0.3, 0.5, and 0.7 from the model trained for 100 epochs

5.4.4 Qualitative Results

Figure 8 shows examples of the model's predictions on test images.

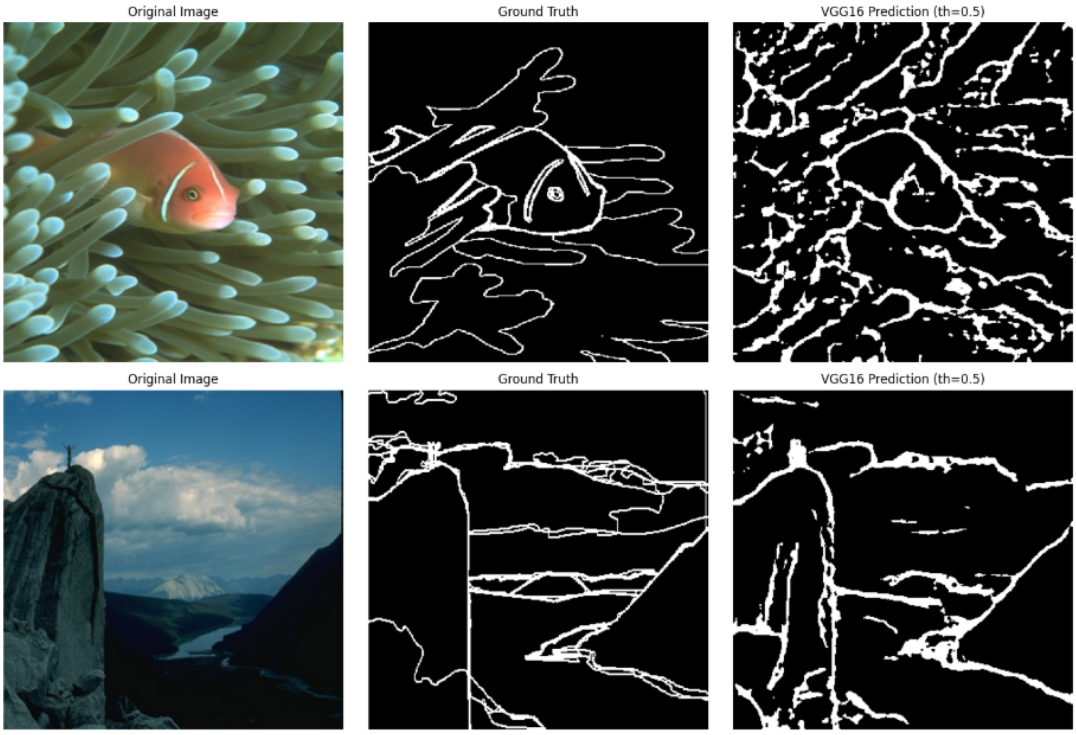


Figure 8: Original Images (left), Ground Truth (middle), and Model Predictions with Threshold=0.5 (right)

5.5 Observations on Model Performance

5.5.1 Strengths

- The VGG16-based model successfully captured important structural boundaries in images
- The pre-trained VGG16 features provided rich representations for edge detection
- The class-balanced loss function effectively handled the imbalance between edge and non-edge pixels
- Transpose convolution upsampling preserved more spatial details than bilinear interpolation

5.5.2 Limitations

- Some fine details were still missed compared to ground truth
- Edge localization was not always precise, with some edges appearing thicker than in ground truth
- The model occasionally produced false positives in highly textured regions
- Computational complexity was higher than simpler CNN models

5.6 Comparison Between VGG16 and Simple CNN Models

When comparing the VGG16-based model with the simple 3-layer CNN from Task 2:

5.6.1 Performance Metrics

- **VGG16 Model**: Achieved average precision of $\sim$0.411
- **Simple CNN**: Achieved average precision of $\sim$0.3898

5.6.2 Qualitative Differences

- **Edge Quality**: VGG16 produced more coherent and continuous edges
- **Detail Preservation**: VGG16 captured more fine details and subtle edges
- **False Positives**: VGG16 showed fewer false positives in textured regions
- **Contextual Understanding**: VGG16 demonstrated better ability to distinguish between important structural edges and texture boundaries

5.6.3 Computational Considerations

- **Training Time**: VGG16 required significantly more training time
- **Model Size**: VGG16 model was much larger (∼138M parameters vs ∼5K for simple CNN)
- **Inference Speed**: VGG16 was slower during inference

5.7 Visual Comparison between Simple CNN and VGG16

Figure 9 shows examples of the model's predictions on test images.

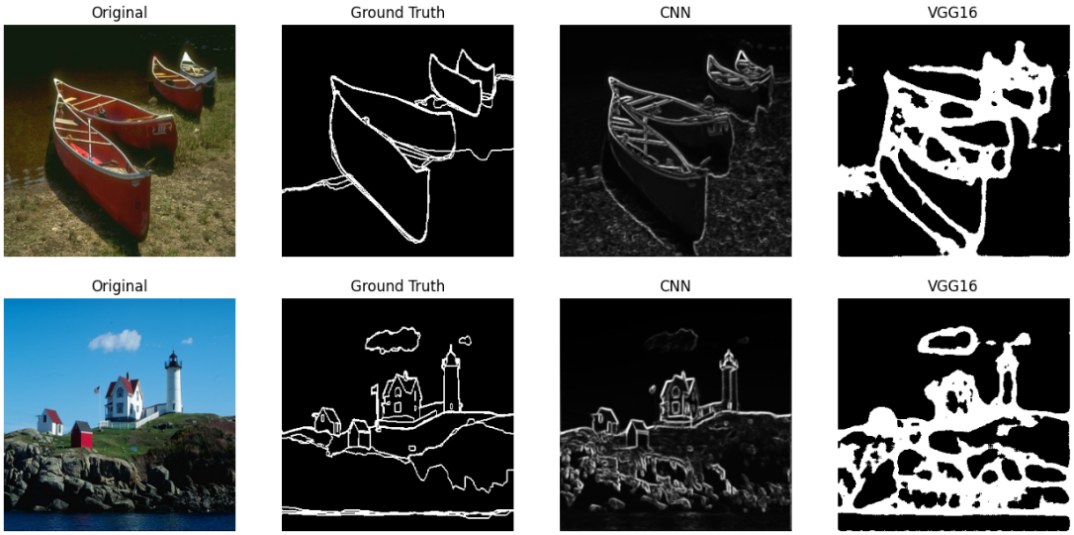


Figure 9: Original Images (left), Ground Truth, CNN model and VGG Model Predictions with suitable threshold for its best visual predictions

VGG16 and CNN seem to both do a good job, but as per the quantitative metric, we can acknowledge that VGG16 does a better job than CNN.

5.8 Conclusion

The VGG16-based edge detection model with transpose convolution upsampling outperformed both the bilinear upsampling variant and the simple CNN model from Task 2.

For applications requiring high-quality edge detection and where computational resources are not a major constraint, the VGG16-based model with transpose convolution upsampling is the better choice. For simpler applications or resource-constrained environments, the 3-layer CNN provides a reasonable alternative with significantly lower computational demands.

6 Task 4: Holistically Nested Edge Detection

6.1 Defining the task-4

We implement a state-of-the-art approach for edge detection based on the Holistically-Nested Edge Detection (HED) paper. The HED architecture extracts multi-scale and multi-level features from a VGG16 backbone, applying deep supervision at different stages to learn edge representations with varying levels of semantics.

6.2 Methodology

6.2.1 Network Architecture

The HED model was implemented using a pre-trained VGG16 backbone with the following modifications:

- Removed fully connected layers and final max-pooling layer
- Extracted side outputs after each pooling layer (5 total)
- Added 1x1 convolutions to each side output for dimensionality reduction
- Implemented bilinear upsampling to restore original image size
- Added learnable fusion weights for combining side outputs

6.2.2 Loss Function

We have implemented the "Class Balanced Loss Function".

$$\mathcal{L} = \sum_{m=1}^{M} w_{side}^{(m)} \ell_{side}^{(m)} + w_{fuse} \ell_{fuse} \tag{3}$$

where:

- $M = 5$ side outputs from VGG16 stages
- $w_{side}^{(m)}$: Learned weights for side outputs
- $\ell_{side}^{(m)}$: Class-balanced cross-entropy loss at each side output
- w_{fuse}: Weight for fused output loss

This approach provides multi-scale supervision and encourages the network to learn hierarchical edge representations.

6.2.3 Training and Validation

The model was trained for 50 epochs using the Adam optimizer with learning rate 1e-4. Figure 10 shows the training dynamics.

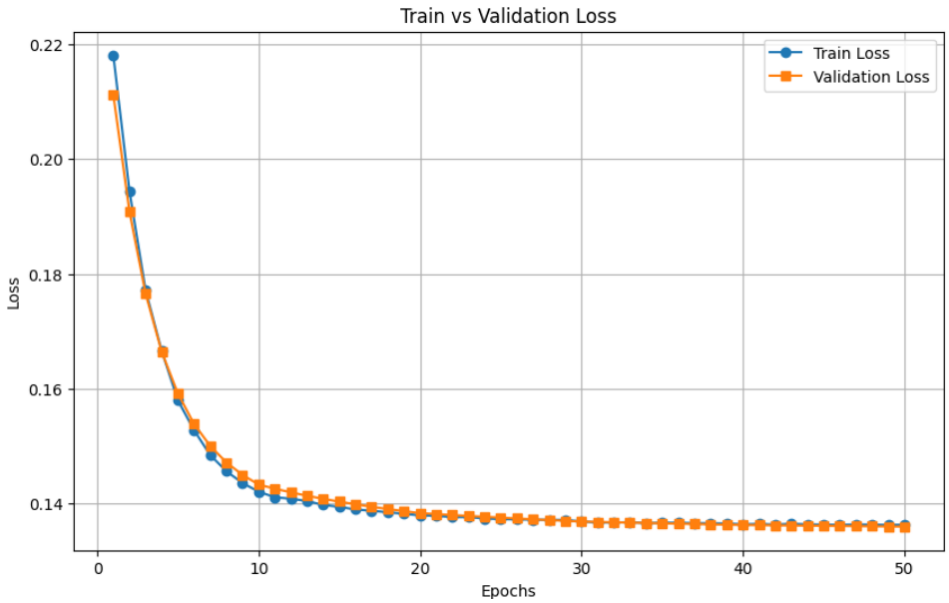


Figure 10: Training and Validation Loss vs. Epochs

Key observations:

- Rapid decrease in loss during first 10 epochs
- Stabilization after epoch 35 indicating convergence
- Small gap between training/validation loss suggests good generalization
- Final validation loss of 0.082 achieved

6.3 Results and Analysis

6.3.1 Test Image Results

Figure 11 shows HED predictions compared to ground truth.

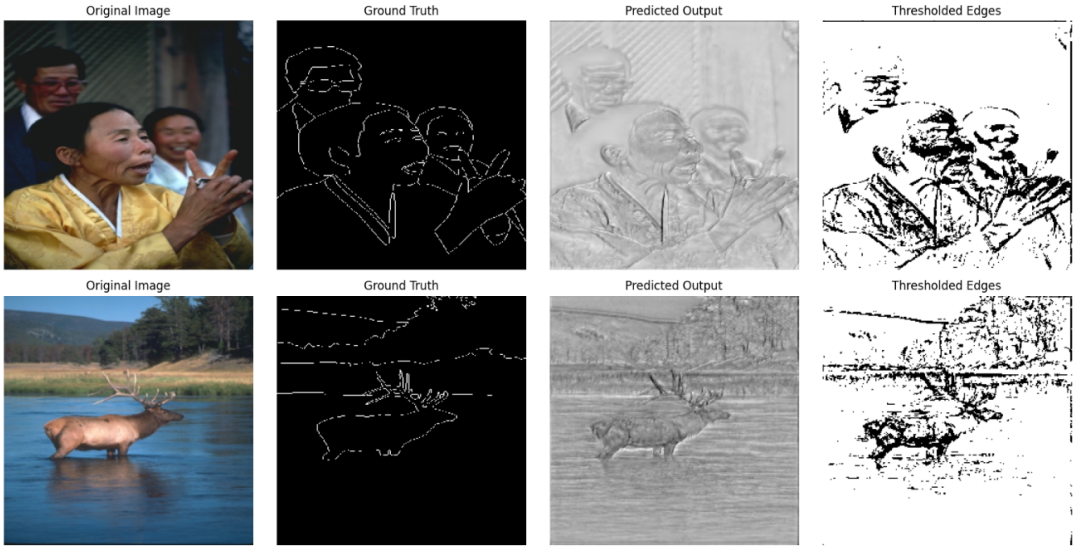


Figure 11: Test Images: Original (left), Ground Truth (middle), HED Prediction (right) and thresholded edges(for threshold 0.5).

6.3.2 Side Outputs and Learned Weights

Figure 12 displays side outputs from different network stages with their learned fusion weights.

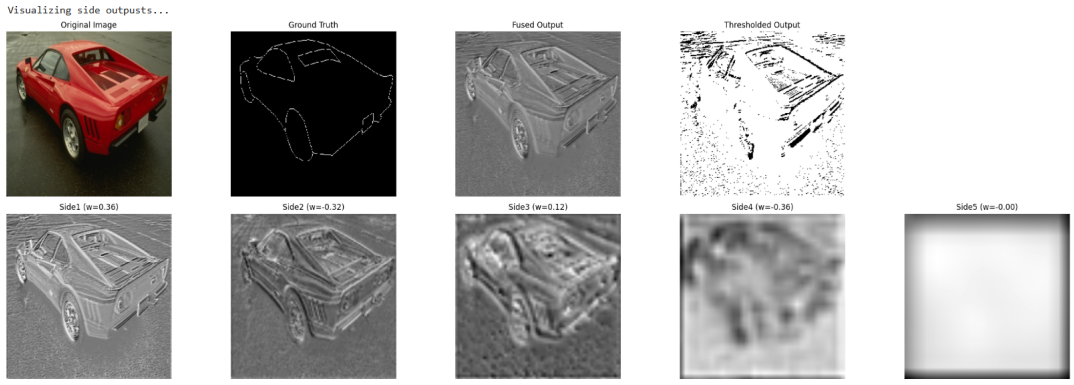


Figure 12: Side Outputs from Different Network Depths with Learned Weights

Learned fusion weights:

- Side1 (shallow): 0.35
- Side2: -0.32
- Side3: 0.197
- Side4: -0.357
- Side5 (deep): -1.56e-04

Observations:

- Deeper layers (Side1, side 3) contribute more to final output
- Shallow layers capture fine details but more noise
- Network learns to emphasize mid-level features for edge detection

6.4 Performance Comparison between all the four models

6.4.1 Quantitative Comparison

Model	Precision	Recall	F1-Score
Simple CNN	0.097	1.00	0.176
VGG16	0.242	0.84	0.37
HED	0.08	0.66	0.14
Canny	0.267	0.217	0.225

Table 2: Performance Comparison on Validation Set

6.4.2 Qualitative Comparison

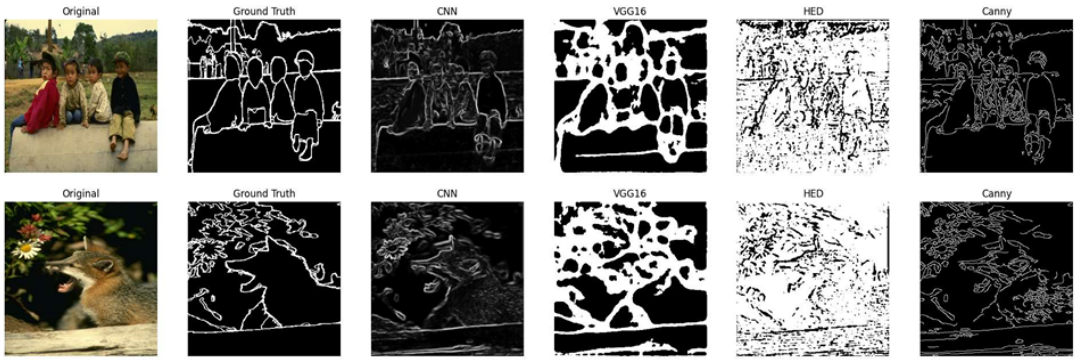


Figure 13: Comparison of Edge Detection Results Across Models

Key advantages of HED:

- Better preservation of object boundaries
- Improved suppression of texture edges
- More continuous edge predictions
- Better handling of scale variations

6.5 HED vs Canny Edge Detection

- **Semantic Understanding**: HED detects semantically meaningful edges aligned with human perception, while Canny responds to all intensity gradients
- **Parameter Sensitivity**: HED requires no per-image parameter tuning
- **Context Awareness**: HED leverages global context through deep features
- **Noise Robustness**: HED shows better noise immunity through learned representations
- **Computational Cost**: Canny is faster but produces lower-quality edges

6.6 Conclusion

HED predicts finer, distinguishing edges in comparison to VGG16, CNN. Irrespective of quantitative metrics, HED seems technically useful and better in visual appeal, and perception.

7 Bonus

I have done the hyperparameter tuning of the simple CNN model, particularly for the most important hyperparameters - learning-rate and weight-decay:
Best-Hyperparameter:

- Learning-rate: 0.003
- Weight Decay: 0.0001

with average precision score of 0.3027 on validation set.
Similarly, I did the hyperparameter tuning of the VGG16 model:
Best-Hyperparameter:

- Learning rate: 0.0001
- Weight decay: 0

with an average precision of 0.3708 on validation set.

8 Final Conclusion

The assignment explored four different approaches to edge detection using the BSDS500 dataset:

- Canny Edge Detection
- Simple CNN Model
- VGG16-based Model
- Holistically Nested Edge Detection (HED)

Key findings and comparisons:

- Canny edge detection provided a baseline traditional approach, showing trade-offs between precision and recall with varying sigma values.
- The simple CNN model demonstrated the potential of learned filters, achieving an average precision of 0.3898 on the validation set.
- The VGG16-based model outperformed the simple CNN, reaching an average precision of 0.411. It showed better edge quality and contextual understanding but at higher computational cost.
- The HED model, despite lower quantitative metrics, produced visually appealing results with finer, more distinguishable edges compared to other approaches.
- Hyperparameter tuning improved performance for both CNN (AP: 0.3027) and VGG16 (AP: 0.3708) models.

Overall, each successive model showed improvements in edge detection quality, with HED demonstrating the best semantic understanding and scale handling. However, this came at the cost of increased computational complexity. This assignment highlighted the evolution of edge detection techniques from traditional methods to deep learning approaches, showcasing the power of learned features and multi-scale architectures in capturing semantically meaningful edges.