EE5178 - MODERN COMPUTER VISION

PROGRAMMING ASSIGNMENT - 2

(Edge Detection)

Satheesh D M

MA24M023

31/03/25

Contents

1 Objective

Not all edges are equally important to the human eye. For segmentation, we need to detect images through edges that humans perceive as important. So, the aim of this assignment is to predict the edge map of a given image, which is close to the human annotation (important to human beings). The Berkeley Segmentation Data Set (BSDS500) is used in this exercise.

2 Data Set and the Edge map

The BSDS500 data set has 500 images that contain annotated edge and segmentation mappings. This includes 200 train images, 100 validation set images, and 200 test set images. However, the test image mappings are available as '`.mat`' format. And this has multiple edge boundary and segmentation mappings. For consistency throughout his assignment, the second edge boundary mapping out of the six available mappings for any given image is used as the label. The edge map is a binary-colored image (white for the edge).

The following code snippet is used for the edge extraction.

```
import scipy.io
import numpy as np
from PIL import Image
import os


def extract_and_save_boundaries(input_dir, output_dir):
    """
    Args:
        input_dir (str): input file containing .mat files
        output_dir (str): output directory to save the extracted boundaries as .jpg
    files
    """
    # Create output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    # Iterate over all .mat files in the input directory
```

```
    for filename in os.listdir(input_dir):
        if filename.endswith(".mat"):
            file_path = os.path.join(input_dir, filename)
            try:
                # Load .mat file
                data = scipy.io.loadmat(file_path)

                # Extract boundary for iteration 0
                boundary = data["groundTruth"][0][0][0, 0]["Boundaries"]

                # Convert to image
                boundary_img = Image.fromarray(
                    (boundary * 255).astype(np.uint8)
                )

                # Save using the same name with PNG extension
                output_path = os.path.join(
                    output_dir, f"{os.path.splitext(filename)[0]}.jpg"
                )
                boundary_img.save(output_path)

                print(f"Saved: {output_path}")
            except Exception as e:
                print(f"Error processing {filename}: {e}")


# Convert and save boundaries for each dataset
extract_and_save_boundaries(
    "archive/ground_truth/test", "archive/ground_truth_boundaries/test"
)
extract_and_save_boundaries(
    "archive/ground_truth/train", "archive/ground_truth_boundaries/train"
)
extract_and_save_boundaries(
    "archive/ground_truth/val", "archive/ground_truth_boundaries/val"
)
```

3 Task 1: Canny Edge Detection

For the code, refer to '`01.Canny-Edge-Detection.ipynb`' in the submission.

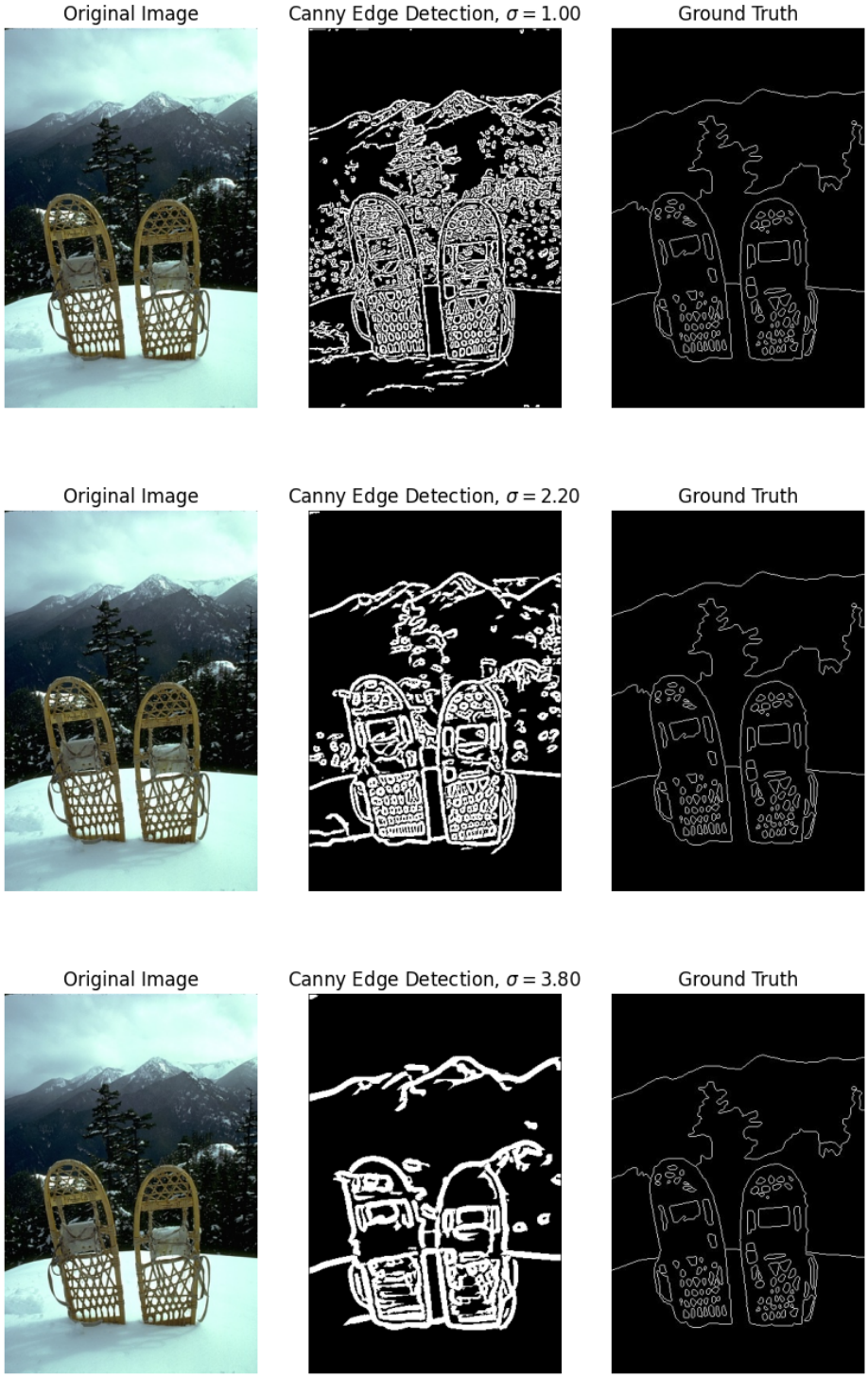
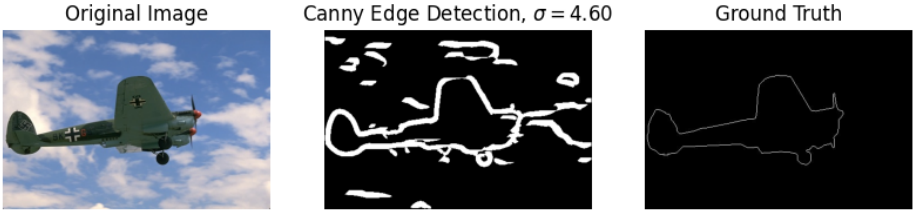
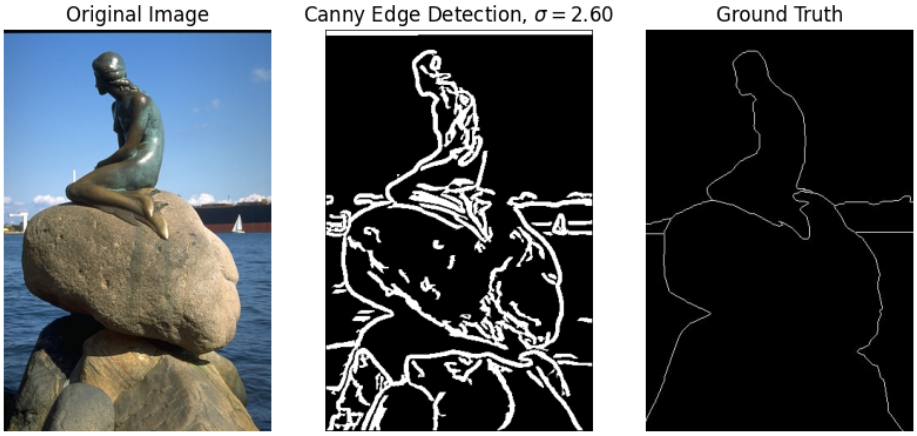
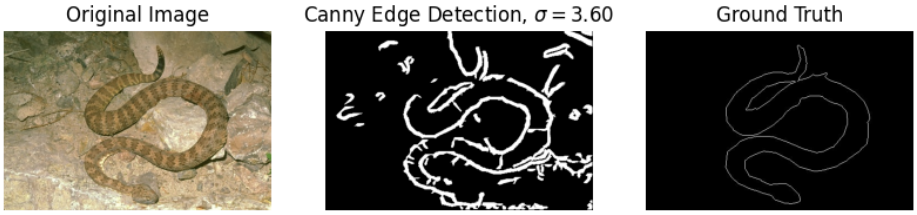


Figure 3.1: This figure shows the effect of the blurring parameter (σ) on the edge maps while using Canny edge detector. With increase in σ, the minor details are lost in the edge map.

Sample mapping that were reasonably close to the ground truth with appropriate σ values.



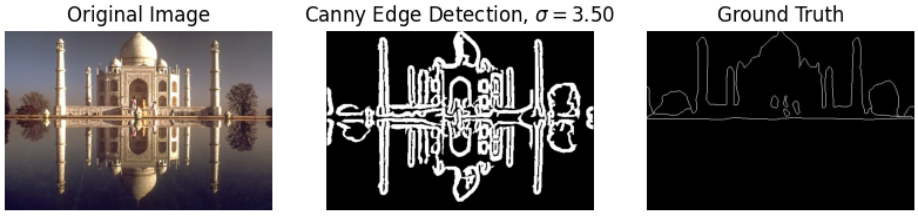


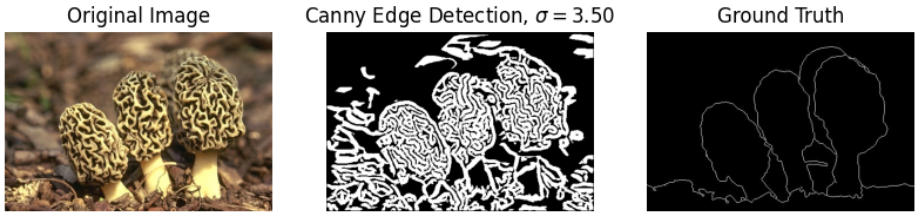


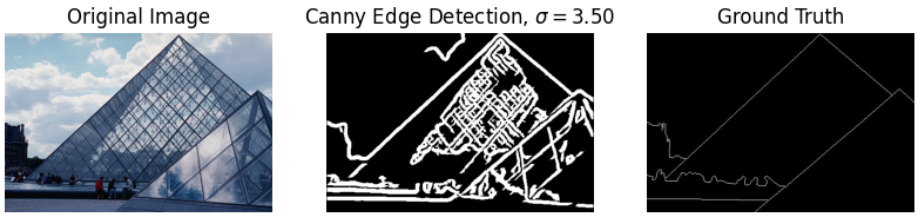
Does the canny detector work well for edge detection? Why/Why not?

- Canny edge detector works well for being a simple derivative of Gaussian-based filtering followed by non maximal suppression and hysteresis thresholding.
- Advantage of Canny detector is that it is a simple rule based filter. It can be applied to any image of any dimension (H $\times$ W).
- Disadvantages:
    1. Since Canny uses Gaussian smoothing, it can struggle with images containing a lot of noise.
    2. Too much noise reduction can blur edges, while too little can leave unwanted edge.
- In general Canny edge detector is not the ideal edge detection technique.

Some sample highlight were Canny filter failed miserably:







4 Task 2: Simple CNN Model

Dataloaders were created for the BSDS dataset's train, validation and test sets. The following CNN was trained, for full code refer to '`02.CNN-Edge-Detection.ipynb`' in the submission.

```
import torch.nn as nn
import torch


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

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

Training Details :

- Total epochs = 100
- Optimizer = Adam
- Learning rate = 0.001
- Regularization = None
- Loss function = Custom Class balanced binary cross entropy loss function from the HED paper.

Class balanced binary cross entropy loss :

This loss function is introduced because the ultimate goal is to do a binary classification on a pixel level to predict weather the given pixel is an edge or not an edge. But the positive (edge) class to the negtive (non-edge) class imbalance in any given image will be very high ($\approx 1:9$).

In the HED paper the authors introduced a class-balancing weight β on a per-pixel term basis. Index j is over the image spatial dimensions of image X. Then they used this class-balancing weight as a simple way to offset this imbalance between edge and non-edge classes. Specifically, they define the following class-balanced cross-entropy loss function as,

$$\mathcal{L}_{\text{class balanced}} = -\beta \sum_{j \in Y_+} \log(\Pr(y_j = 1|X, W)) - (1-\beta) \sum_{j \in Y_-} \log(\Pr(y_j = 0|X, W)) \tag{4.1}$$

where, $Y = (y_j, j = 1, 2, \ldots, |X|)$, $j_j \in \{0, 1\}$ is the edge map. X is the given image. $\beta = |Y_-|/|Y|$, $(1-\beta) = |Y_+|/|Y|$. And $|Y_-|$ and $|Y_+|$ denote the non edge and edge ground truth label sets. W denote the parameters.

Output layer activation : Sigmoid

From the code it may appear that the output layer has no activation. It is true. The network outputs the logits only. But the custom loss function defined in the code uses the inbuilt Binary cross entropy loss with logits function from torch library. It takes inputs as logits and then apply **sigmoid activation** and calculates the BCE loss. It is done in this way because this way of implementation has better numerical stability. Sigmoid activation is applied because, BCE looks for probability for loss calculation.

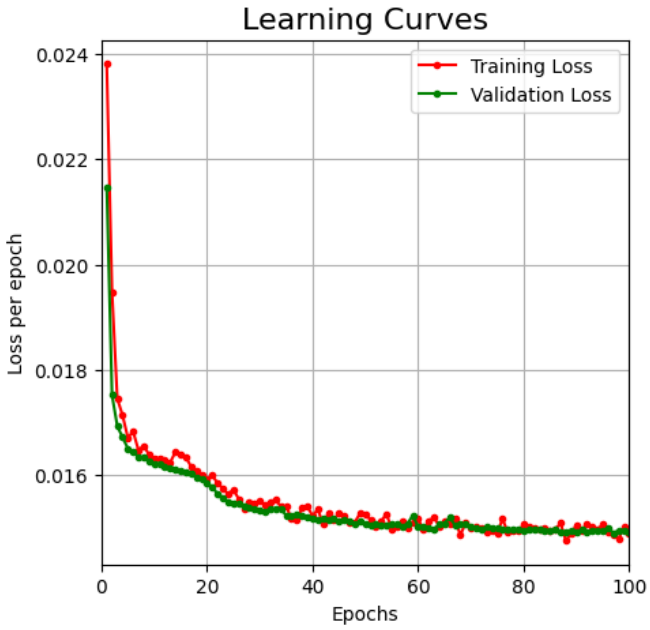


Figure 4.1: CNN model learning curves. Tried different approaches (adjusting learning rate, optimizer) but validation loss was always greater than the training loss but very close to it. This indicates the model is generalizing (might be an underfit), but not reached optimal state yet.

Comments on performance of the CNN model:

This CNN model performed better than the canny edge detector. It got a little bit closer to the ground truth. But there are some issues.

- Patchy edges. The image requires post processing for extracting better edges.
- Though better than Canny the problem of thresholding a lot while binarizing removed edges, while too little left unwanted patchy edges still persists.

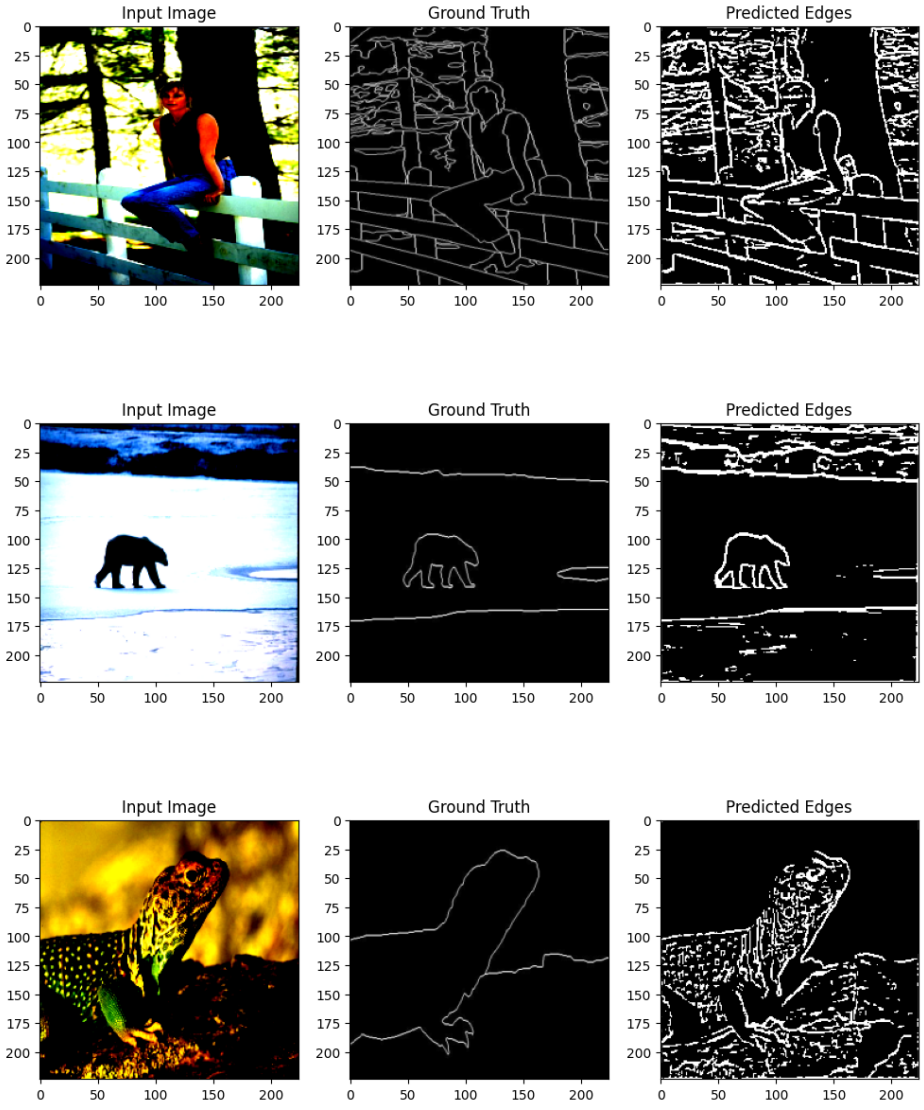


Figure 4.2: Sample CNN output plots.

5 Task 3: VGG16 Model

Dataloaders were created for the BSDS dataset's train, validation and test sets. The following VGG16 based network was trained, for full code refer to '`03.VGG-Edge-Detection.ipynb`' in the submission.

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

        # Encoder (VGG16)
        vgg16 = models.vgg16(weights=models.VGG16_Weights.DEFAULT)
        self.encoder = nn.Sequential(
            *list(vgg16.features.children())[:-1]
        )  # (512 channels)

        # Decoder
        self.deconv1 = nn.ConvTranspose2d(
            512, 256, kernel_size=3, stride=2, padding=1, output_padding=1
        )
        self.deconv2 = nn.ConvTranspose2d(
            256, 128, kernel_size=3, stride=2, padding=1, output_padding=1
        )
        self.final_conv = nn.ConvTranspose2d(128, 1, kernel_size=1)
        self.upsample = nn.Upsample(
            size=(224, 224), mode="bilinear", align_corners=True
        )
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.encoder(x)

        x = self.deconv1(x)
        x = self.relu(x)

        x = self.deconv2(x)
        x = self.relu(x)

```

```
        x = self.final_conv(x)
        x = self.upsample(x)

        return x
```

Training Details :

- Total epochs = 50
- Optimizer = Adam
- Learning rate = 0.00001
- Regularization = None
- Loss function = Custom Class balanced binary cross entropy loss function from the HED paper.
- Output activation = (Sigmoid) but in shown code it seems to be None.

The explanation for using sigmoid activation in the output and using the class balanced binary cross entropy is the same as explained in the previous CNN section. And it is the same throughout this assignment even in the next Hollistic nested edge detctor architecture also.

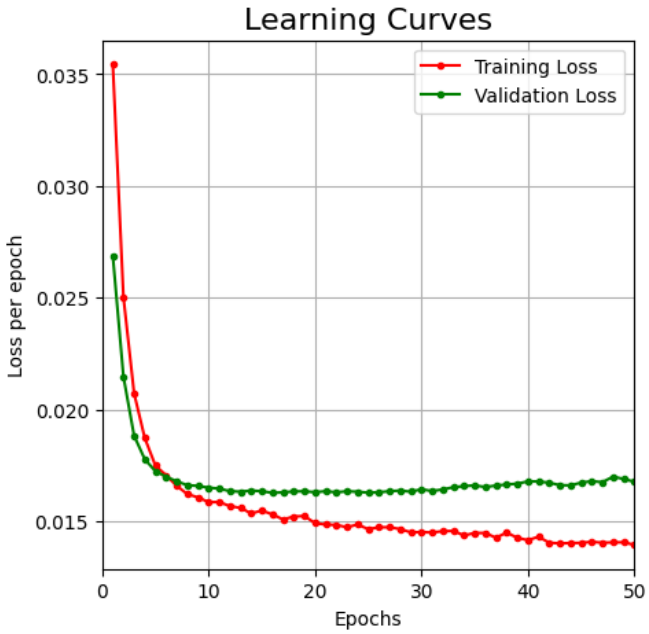


Figure 5.1: VGG16 based encoder decoder edge detection model's learning curves.

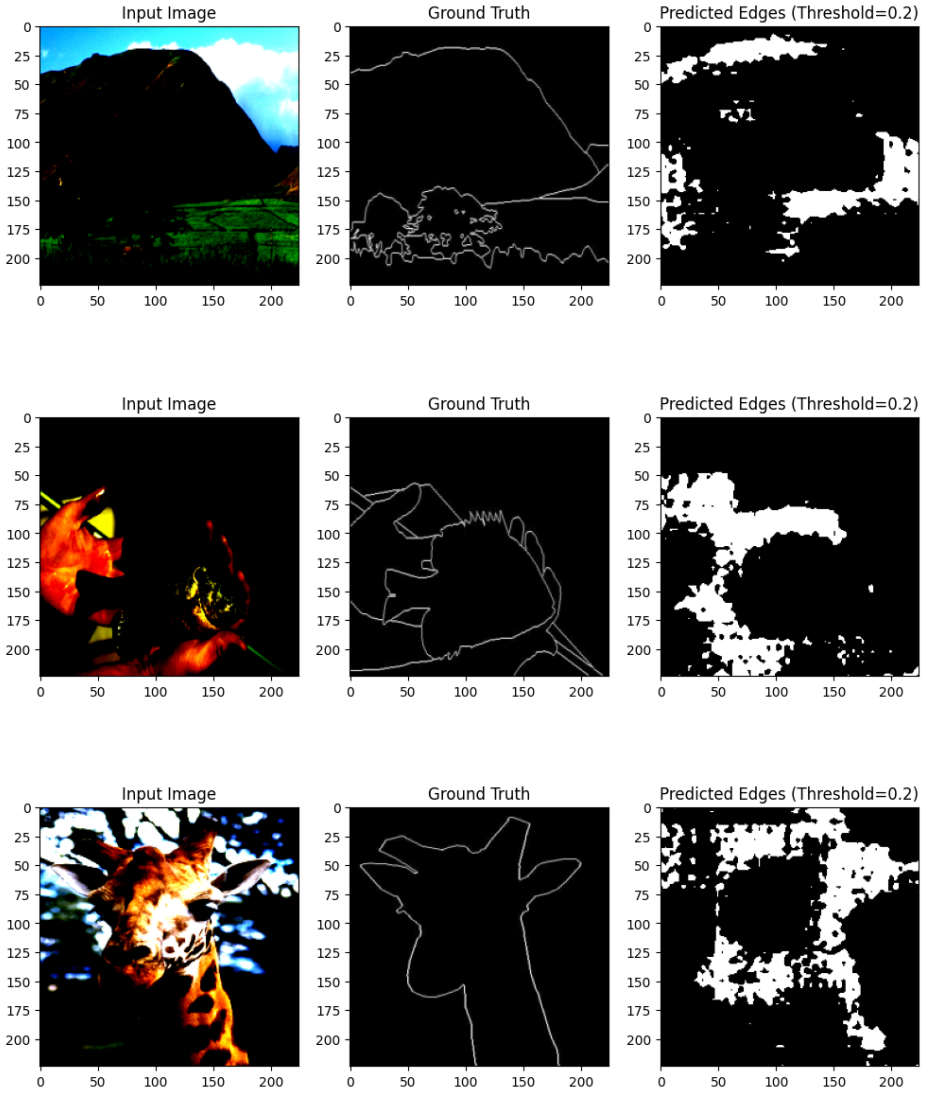


Figure 5.2: Sample VGG16 based encoder decoder edge detection model's output plots.

Comments on performance of the VGG16 based model:

This CNN model performed poorly than the canny edge detector.

- Patchy edges with checker board pattern.
- Changing threshold values did not help in getting rid of the patchy checker board pattern.
- The ConvTranspose2d layers use a kernel size = 3, stride=2 with padding=1 and output padding=1. This causes uneven overlap of pixels, leading to artifacts like checkerboard patterns.
- This architecture is used since it was communicated as a hint in the email to the entire class.

CNN & VGG16 based model output comparison: (Clearly CNN is better)

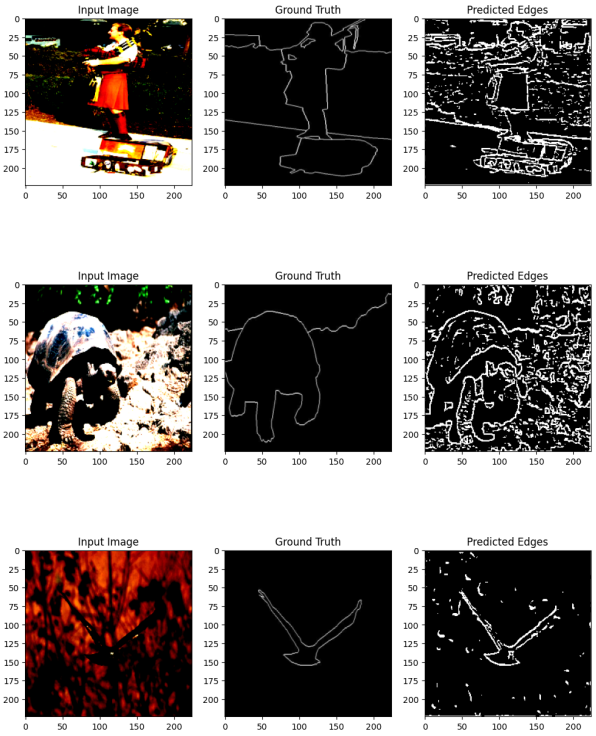


Figure 5.3: CNN model's output plots.

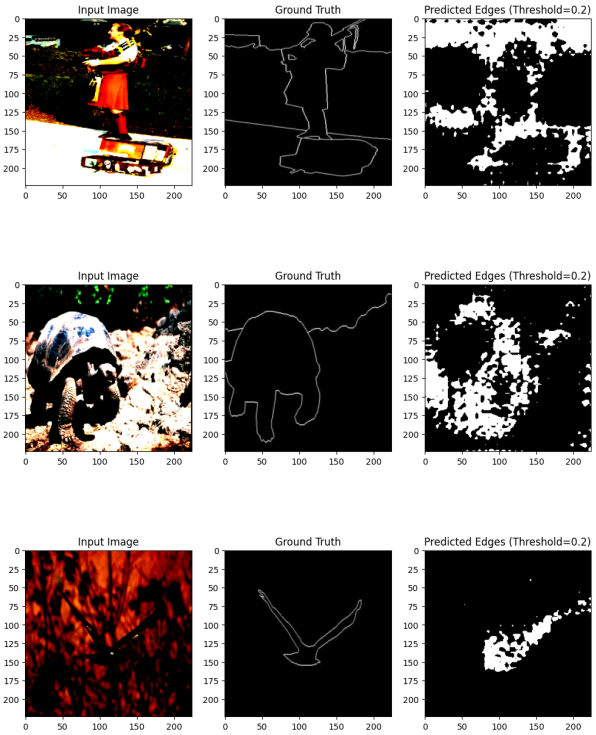


Figure 5.4: VGG16 based encoder decoder edge detection model's output plots.

6 Task 4 Holistically Nested Edge Detection

Dataloaders were created for the BSDS dataset's train, validation and test sets. The following HED network was trained, for full code refer to '`04.HED-Edge-Detection.ipynb`' in the submission.

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

        # Load VGG16 as base network
        vgg16 = models.vgg16(weights=models.VGG16_Weights.DEFAULT)
        features = list(vgg16.features.children())

        # Encoder (VGG16 backbone, WITHOUT last maxpool)
        self.conv1 = nn.Sequential(*features[:5])
        self.conv2 = nn.Sequential(*features[5:10])
        self.conv3 = nn.Sequential(*features[10:17])
        self.conv4 = nn.Sequential(*features[17:24])
        self.conv5 = nn.Sequential(*features[24:29])

        # Side output layers (1x1 conv to get single-channel logits)
        self.side1 = nn.Conv2d(64, 1, kernel_size=1)
        self.side2 = nn.Conv2d(128, 1, kernel_size=1)
        self.side3 = nn.Conv2d(256, 1, kernel_size=1)
        self.side4 = nn.Conv2d(512, 1, kernel_size=1)
        self.side5 = nn.Conv2d(512, 1, kernel_size=1)

        # Learnable fusion weights
        self.weights = nn.Parameter(torch.ones(5, dtype=torch.float32))

    def forward(self, x):
        img_size = x.shape[2:]

        # Forward pass through VGG16 layers
        c1 = self.conv1(x)
        c2 = self.conv2(c1)
        c3 = self.conv3(c2)
```

```
        c4 = self.conv4(c3)
        c5 = self.conv5(c4)

        # Compute side outputs (raw logits)
        s1 = self.side1(c1)
        s2 = self.side2(c2)
        s3 = self.side3(c3)
        s4 = self.side4(c4)
        s5 = self.side5(c5)

        # Upsample side outputs to match input size
        s1 = F.interpolate(
            s1, size=img_size, mode="bilinear", align_corners=True
        )
        s2 = F.interpolate(
            s2, size=img_size, mode="bilinear", align_corners=True
        )
        s3 = F.interpolate(
            s3, size=img_size, mode="bilinear", align_corners=True
        )
        s4 = F.interpolate(
            s4, size=img_size, mode="bilinear", align_corners=True
        )
        s5 = F.interpolate(
            s5, size=img_size, mode="bilinear", align_corners=True
        )

        # Normalize the weights using softmax (to ensure non-negative weights)
        normalized_weights = F.softmax(self.weights, dim=0)

        # Final fused output using learnable weighted sum
        fused = sum(
            w * s for w, s in zip(normalized_weights, [s1, s2, s3, s4, s5])
        )

        return s1, s2, s3, s4, s5, fused
```

Training Details :

- Total epochs = 100
- Optimizer = Adam
- Learning rate = 0.00001
- Regularization = None
- Loss function = Custom Class balanced binary cross entropy loss function from the HED paper.
- Output activation = (Sigmoid) but in shown code it seems to be None.

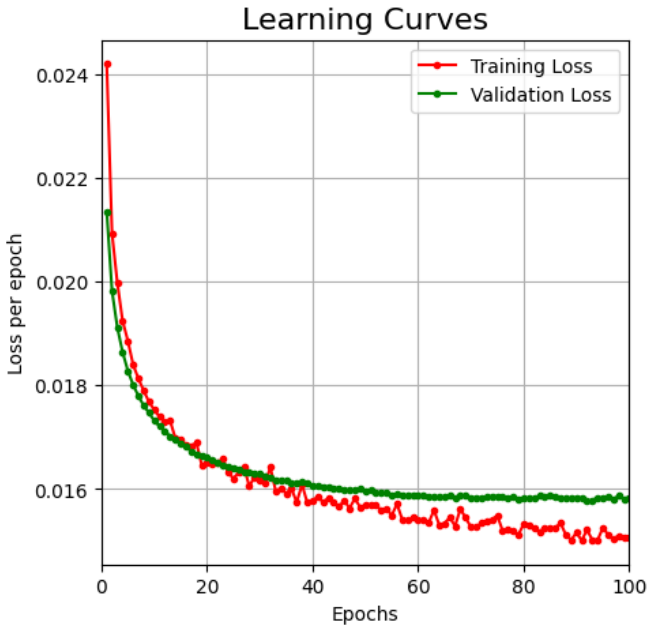
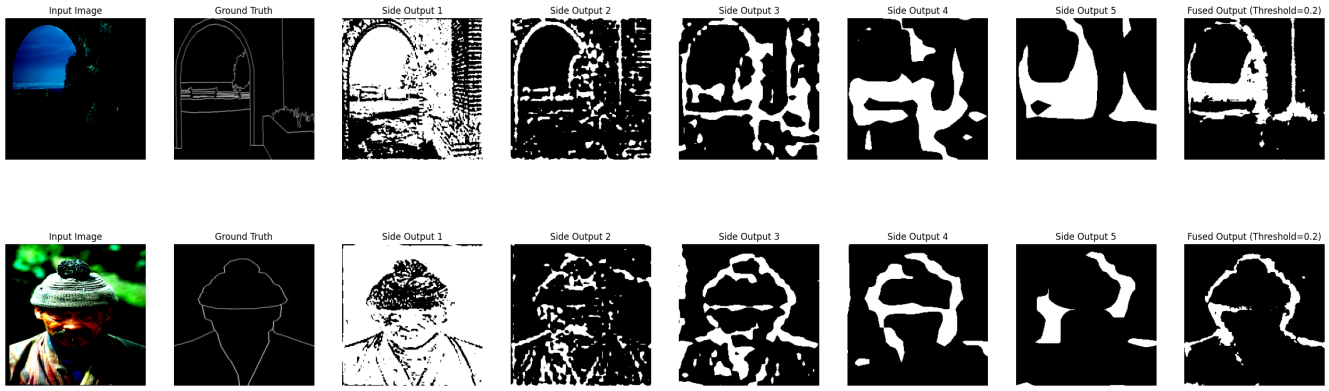


Figure 6.1: Hollistic nesed Edge Detection model's learning curves.



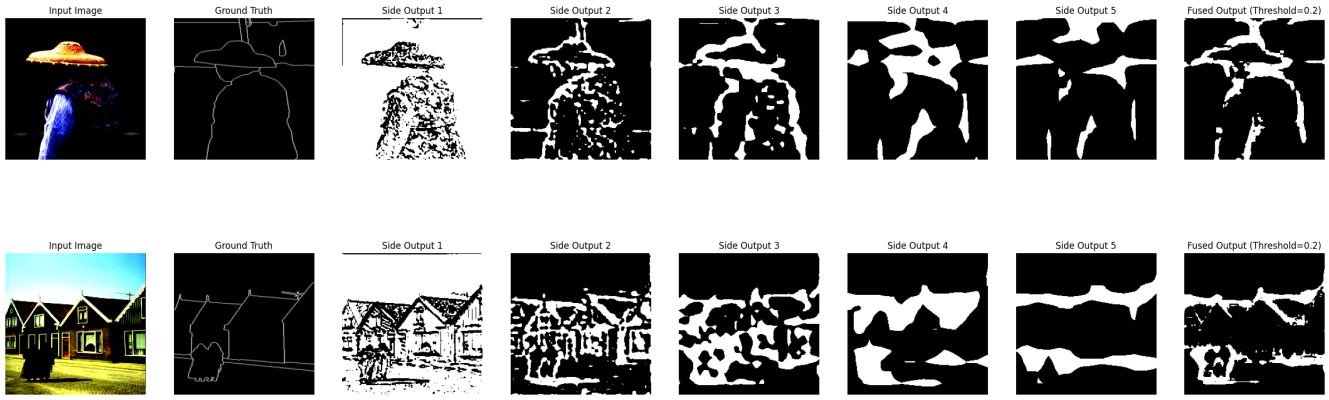


Figure 6.2: HED model's output plots.

Obersvations:

- Each side output is learning an edge map of the given input at various scales.
- The final fused output is a weighted sum whose weights are learned while training the model.
- My final edge map is very patchy and not very clear. I tried different threshold values. I have shown the plots for the appropriately good threshold value.
- My observation is that the model can be trained even better. From the learning curve, it becomes clear that the model did not converge to an optimal state properly within 100 epochs. The validation loss did not increase back showing overfitting but it was continuously maintained in a plateau.
- Maybe adjusting the learning rate according to the error surface by a learning rate scheduler should improve the model's performance. I am submitting this model due to the time constraints.

Comparions:

- Compared to VGG16 based model HED is performing very good. Though patchy, no checker board pattern and the final edge map is comparable with the ground truth.
- Compared to CNN model HED in principle and in my case also gets edges that are important to human only (close to ground truth).
- Overall HED got close to the annotated edge maps and outperformed all the other 3 models studied in this assignment Canny detector, CNN detector and VGG16 based detector.