Unveiling the Hidden Dimensions: Monocular Depth Estimation Unleashed

By- Sahin Hossain Chowdhury(sm21mtech12002)

Under The Guidance- Dr. R Prasanth Kumar

Table of Contents

Introducing Monocular Depth Estimation

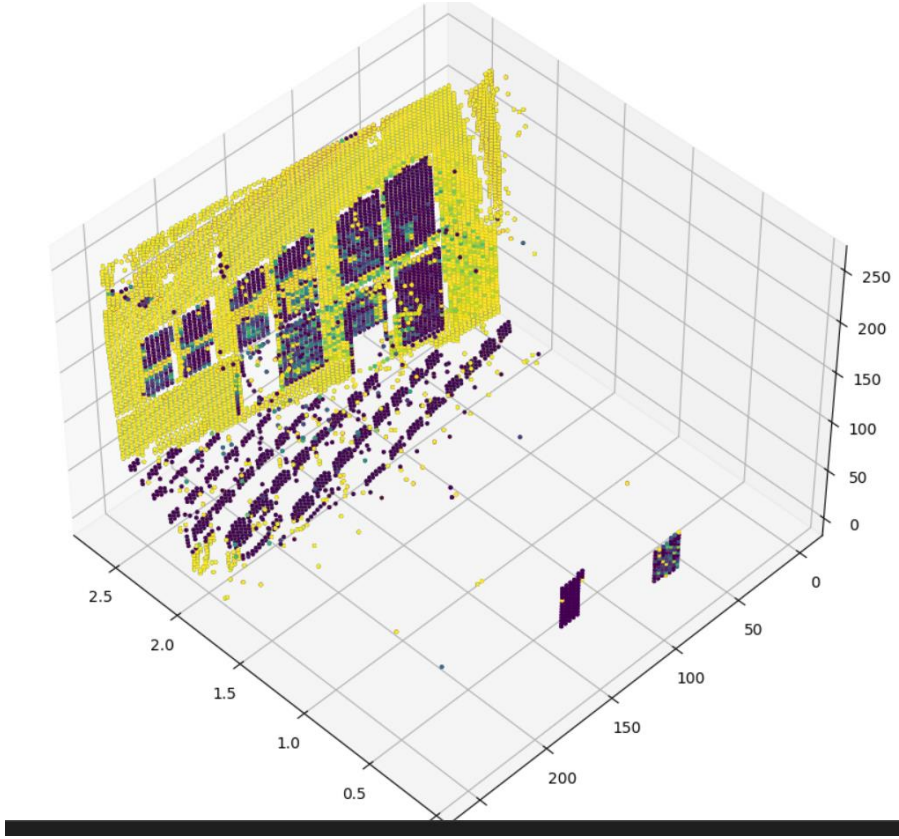
- Monocular depth estimation refers to the process of estimating the depth information of a scene using a single camera.
- Unlike traditional stereo vision techniques that require multiple cameras, monocular depth estimation harnesses the power of computer vision algorithms to infer depth from a single image.
- This technique has gained significant attention in recent years due to its wide-ranging applications in computer vision and perception tasks.

Significance of monocular depth estimation

1. Scene Understanding: Monocular depth estimation helps in understanding the 3D structure of a scene from a single image This understanding is essential for tasks such as object recognition, semantic segmentation, and scene reconstruction.

2.Depth-Aware Applications: Monocular depth estimation enables depth-aware applications, where the estimated depth information is utilized to enhance various computer vision tasks. Augmented reality systems also leverage monocular depth estimation to accurately overlay virtual objects onto the real world.

3. Autonomous Systems: Monocular depth estimation is particularly relevant in autonomous systems such as self-driving cars and drones. Accurate depth estimation from a single camera enables these systems to perceive the surrounding environment and make informed decisions. It assists in tasks like obstacle detection, collision avoidance, path planning, and object tracking.



Limitations of the Research

1. **Ambiguity and uncertainty:** Depth estimation from a single image is inherently an ill-posed problem due to the lack of direct depth cues. Ambiguities can arise when estimating depth from texture less regions, transparent objects, or reflective surfaces

2. **Limited depth perception range:** Monocular depth estimation has limitations in perceiving depth accurately over long distances.

3. **Training data requirements:** Deep learning-based approaches for monocular depth estimation require large amounts of annotated training data. Collecting and annotating diverse and representative datasets with accurate depth ground truth can be time-consuming and resource-intensive.
4. **Computational complexity:** Some advanced monocular depth estimation algorithms, especially deep learning-based methods, can be computationally expensive. Real-time performance may be challenging to achieve in resource-constrained environments or applications with strict latency requirements.

Research Objectives

- **Objective 1:** Develop a novel approach for creating a smaller size depth estimation model that maintains high accuracy while reducing computational and memory requirements.
- **Objective 2:** Investigate techniques for optimizing depth and object prediction algorithms to achieve real-time performance on lower-end devices.
- **Objective 3:** Fusion of depth estimation with object detection

Datasets

- **NYU Depth V2 Dataset:** 1,449 indoor RGB-D images with depth maps and annotations, covering bedrooms, kitchens, living rooms, and offices.
- **KITTI Vision Benchmark Suite:** Over 93,000 labeled outdoor driving images, including depth maps, stereo correspondences, and 3D point clouds, showcasing diverse weather, traffic, and objects.
- **SUN3D Dataset:** 10 indoor scenes with RGB-D data, featuring bedrooms, offices, and living rooms, with depth maps, camera poses, and dynamic scenes.
- **TUM RGB-D Dataset:** RGB-D sequences from handheld devices, indoor and outdoor scenes like offices, corridors, and urban areas, with synchronized frames and ground truth depth maps.
- **ApolloScape Dataset:** Diverse urban driving scenarios dataset with semantic and instance segmentation, high-resolution depth maps, and precise annotations.
- **ScanNet Dataset:** RGB-D scans of indoor scenes from homes, offices, and shops, including 3D mesh models and dense per-pixel depth maps, capturing dynamic scenes.

Why NYU depth v2 dataset

1. Indoor Scene Coverage: The NYU Depth V2 dataset specifically focuses on indoor scenes, which aligns well with the indoor environment encountered during drone navigation in buildings, homes, or offices. It provides a diverse set of indoor scenes, including bedrooms, kitchens, living rooms, and offices, which can help evaluate the performance of algorithms in different indoor environments

2. Complexity: The NYU Depth V2 dataset captures a range of scene complexities typically found indoors, including occlusions, texture less regions, and depth discontinuities. These challenging aspects are relevant to indoor drone navigation scenarios, where the drone needs to navigate around obstacles and accurately estimate depth to avoid collisions.

4. Lighting Variations: The dataset contains images captured under diverse lighting conditions, including different intensities and sources. This variation in lighting conditions is valuable for assessing the robustness and generalization capabilities of depth estimation algorithms in different indoor lighting environments.

5. Dataset Size: With 1,449 RGB-D images, the NYU Depth V2 dataset offers a substantial amount of data for training and evaluation purposes.

Data Loading and Visualization of 5 Random samples

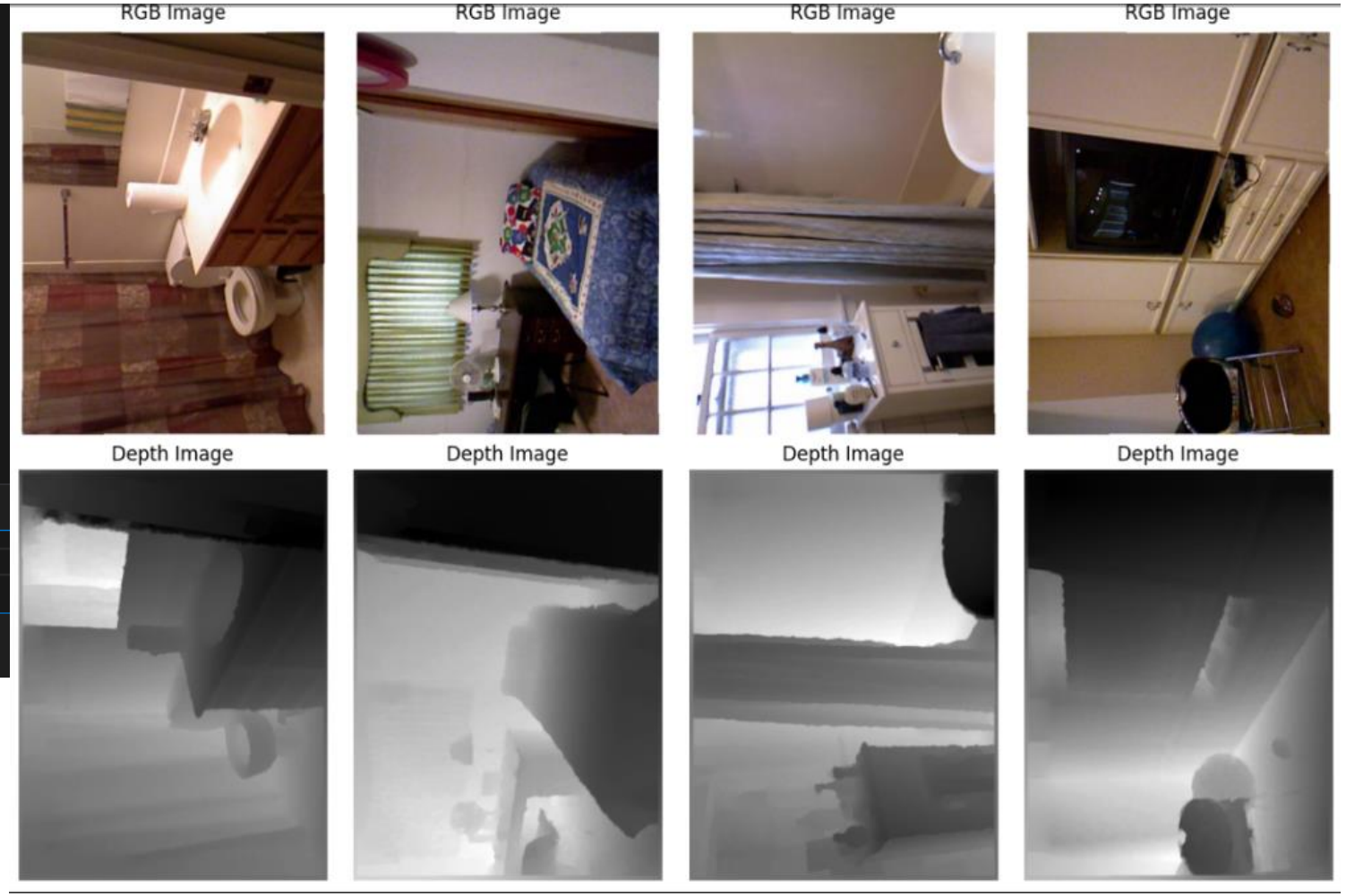
```python
import numpy as np
import h5py

filename = 'C:/Users/ee20m/Documents/nyu_depth_v2_labeled.mat'

# Load the data from the .mat file using h5py
with h5py.File(filename, 'r') as f:
    X = np.array(f['images'])
    y = np.array(f['depths'])

# Save the training data as NumPy arrays
np.save('nyudepth_v2_x_train.npy', X)
np.save('nyudepth_v2_y_train.npy', y)


X = X.transpose((0, 2, 3, 1))
```

✓ 38.6s

```python
X.shape,y.shape
```

✓ 0.0s

```
((1449, 640, 480, 3), (1449, 640, 480))
```



How Monocular Depth Estimation Model Works

1.Perspective and Size Changes: Objects that are closer to the camera appear larger in the image, while objects that are farther away appear smaller. By analyzing the perspective and size changes of objects, the model can estimate their relative distances from the camera.

2. Texture Gradients: Texture gradients refer to the variation in the visual texture of objects as a function of their distance from the camera. Closer objects tend to have more detailed and pronounced textures, while distant objects appear smoother. By analyzing the texture gradients, the model can infer the depth ordering of objects.

3. Occlusions: Occlusions occur when one object partially or completely blocks another object in the scene. Monocular depth estimation models leverage occlusion cues to understand the relative depth relationships between objects. By analyzing the occlusion patterns, the model can estimate the depth ordering and position of objects in the scene.

4. Shading Variations: Shading variations in an image arise due to the interaction of light with the objects in the scene. Different depths of objects can lead to variations in lighting and shading. Monocular depth estimation models analyze these shading variations to infer the depth information. Darker regions can indicate the presence of depth discontinuities or objects positioned farther away.

Data Preprocessing

1. **Resize:** Resize the RGB images and depth maps to a desired resolution. The NYU Depth V2 dataset contains images with a resolution of 640x480 pixels. I resize the image to (320x160)

2. **Depth Map Normalization:** Normalizing the depth maps to a consistent scale. The original depth values in the NYU Depth V2 dataset are provided in millimeters. I've normalized the depth values using min-max normalization.

3. **Color Image Normalization:** Normalizing the RGB images to ensure consistent color representations. I've applied color space transformations, such as normalizing the color channels, to remove variations in lighting conditions and color distributions.

4. **Denoising:** Apply denoising techniques to reduce noise in the depth maps. Noise can be present due to sensor limitations or environmental factors. Common denoising techniques include Gaussian filtering, median filtering, or bilateral filtering. I've choose Gaussian filtering.

5. **Data Augmentation:** you want to augment the dataset to increase its size and diversity, you can apply data augmentation techniques such as rotation, scaling, cropping, flipping, or adding synthetic noise. Data augmentation helps improve the generalization capability of your algorithm.

Metrics to measure how well the model is performing

1. **Mean Absolute Error (MAE):** MAE measures the average absolute difference between the predicted depth values and the ground truth depth values. It provides a measure of the average error in depth estimation.

2. **Root Mean Square Error (RMSE):** RMSE calculates the square root of the mean of the squared differences between the predicted and ground truth depth values. It indicates the overall deviation of the estimated depth from the ground truth.

3. **Relative Depth Error:** Relative Depth Error measures the ratio between the absolute difference and the ground truth depth. It quantifies the relative error between the estimated and ground truth depths and is particularly useful for evaluating the accuracy of depth estimates across different scenes or depth ranges.

5. **Structural Similarity Index (SSIM):** SSIM measures the structural similarity between the predicted depth map and the ground truth depth map. It evaluates the perceptual quality of the estimated depth by considering luminance, contrast, and structural information.

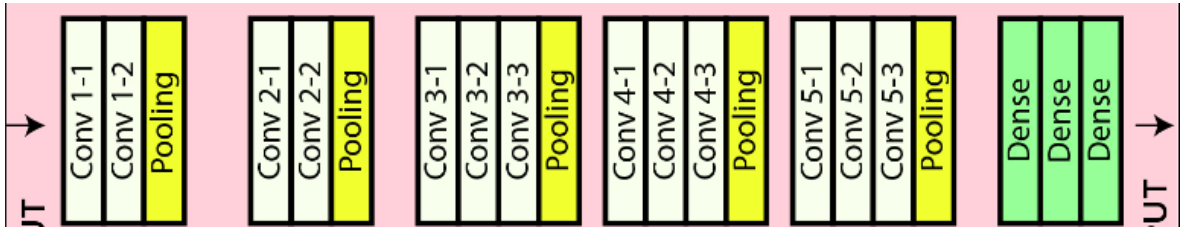
Different Loss Function

- **Mean Squared Error (MSE):** MSE is one of the most straightforward loss functions used for depth estimation. It measures the average squared difference between the predicted depth values and the ground truth depth values
- **Huber Loss:** Huber loss is a robust loss function that combines the characteristics of both MSE and Mean Absolute Error (MAE). It is less sensitive to outliers and can provide a balance between robustness and accuracy.
- **Structural Similarity Index (SSIM) Loss:** SSIM loss is based on the structural similarity index and is used to measure the perceptual quality of the predicted depth map. It considers the luminance, contrast, and structural similarities between the predicted depth and the ground truth depth.
- **Depth Smoothness Loss:** Depth smoothness loss encourages smoothness in the estimated depth map. It penalizes abrupt depth discontinuities and promotes spatial coherence. This can be achieved by computing the gradient of the estimated depth map and minimizing its magnitude.
- **Perceptual Loss:** Perceptual loss utilizes pre-trained deep neural networks, such as VGG or ResNet, to compute feature similarities between the predicted depth map and the ground truth depth map. It encourages the predicted depth to match the high-level features of the ground truth depth.

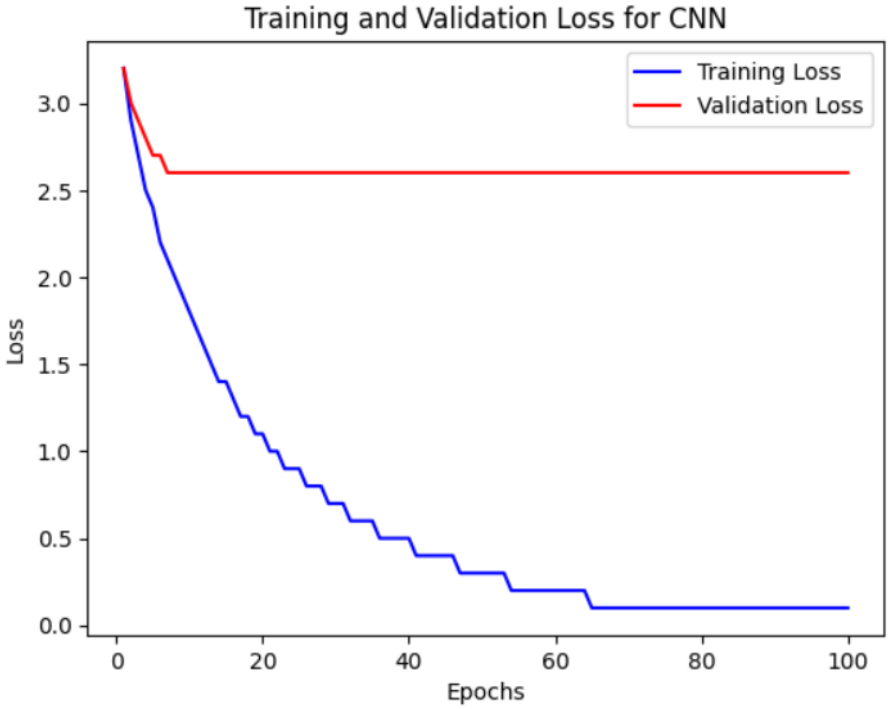
```python
import torch
import torch.nn.functional as F
from torchvision.models import vgg16

def depth_loss(predicted_depth, ground_truth_depth, alpha=0.85, beta=0.1, gamma=0.05, delta=1.0):
    # Mean Squared Error (MSE) loss
    mse_loss = F.mse_loss(predicted_depth, ground_truth_depth)

    # Huber loss
    residual = torch.abs(predicted_depth - ground_truth_depth)
    huber_loss = torch.where(residual <= delta, 0.5 * residual ** 2, delta * residual - 0.5 * delta ** 2)
    huber_loss = torch.mean(huber_loss)

    # Structural Similarity Index (SSIM) loss
    ssim_loss = 1 - torch.mean(torch.functional.image.ssim(predicted_depth, ground_truth_depth, data_range=1, size_average=True))

    # Depth smoothness loss
    grad_x = torch.abs(predicted_depth[:, :, :, :-1] - predicted_depth[:, :, :, 1:])
    grad_y = torch.abs(predicted_depth[:, :, :-1, :] - predicted_depth[:, :, 1:, :])
    smoothness_loss = torch.mean(grad_x) + torch.mean(grad_y)

    # Perceptual loss using VGG features
    vgg = vgg16(pretrained=True).features[:23].eval().to(predicted_depth.device)
    vgg_predicted = vgg(predicted_depth)
    vgg_ground_truth = vgg(ground_truth_depth)
    perceptual_loss = F.mse_loss(vgg_predicted, vgg_ground_truth)

    # Combined loss
    total_loss = alpha * mse_loss + beta * huber_loss + gamma * ssim_loss + delta * smoothness_loss + perceptual_loss

    return total_loss
```

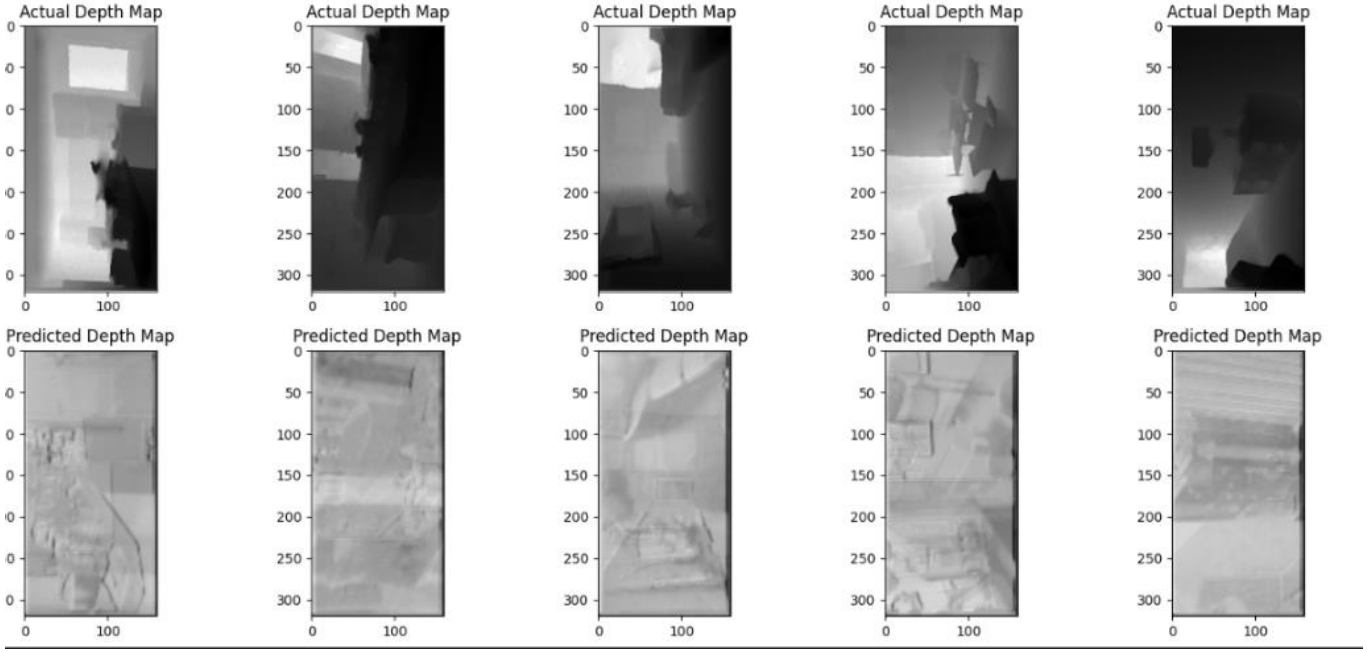
CNN Model Architecture

```python
# Define the CNN model architecture
model = tf.keras.Sequential()

# Convolutional layers
model.add(layers.Conv2D(64, 3, activation='relu', padding='same', input_shape=(320, 160, 3)))
model.add(layers.Conv2D(64, 3, activation='relu', padding='same'))
model.add(layers.MaxPooling2D(pool_size=(2, 2)))

model.add(layers.Conv2D(128, 3, activation='relu', padding='same'))
model.add(layers.Conv2D(128, 3, activation='relu', padding='same'))
model.add(layers.MaxPooling2D(pool_size=(2, 2)))

model.add(layers.Conv2D(256, 3, activation='relu', padding='same'))
model.add(layers.Conv2D(256, 3, activation='relu', padding='same'))
model.add(layers.Conv2D(256, 3, activation='relu', padding='same'))
model.add(layers.MaxPooling2D(pool_size=(2, 2)))

model.add(layers.Conv2D(512, 3, activation='relu', padding='same'))
model.add(layers.Conv2D(512, 3, activation='relu', padding='same'))
model.add(layers.Conv2D(512, 3, activation='relu', padding='same'))
model.add(layers.MaxPooling2D(pool_size=(2, 2)))

model.add(layers.Conv2D(512, 3, activation='relu', padding='same'))
model.add(layers.Conv2D(512, 3, activation='relu', padding='same'))
model.add(layers.Conv2D(512, 3, activation='relu', padding='same'))
model.add(layers.MaxPooling2D(pool_size=(2, 2)))

# Flatten the output
model.add(layers.Flatten())

# Fully connected layers
model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dense(4096, activation='relu'))

# Reshape to (320, 160)
model.add(layers.Reshape((320, 160, 1)))
```



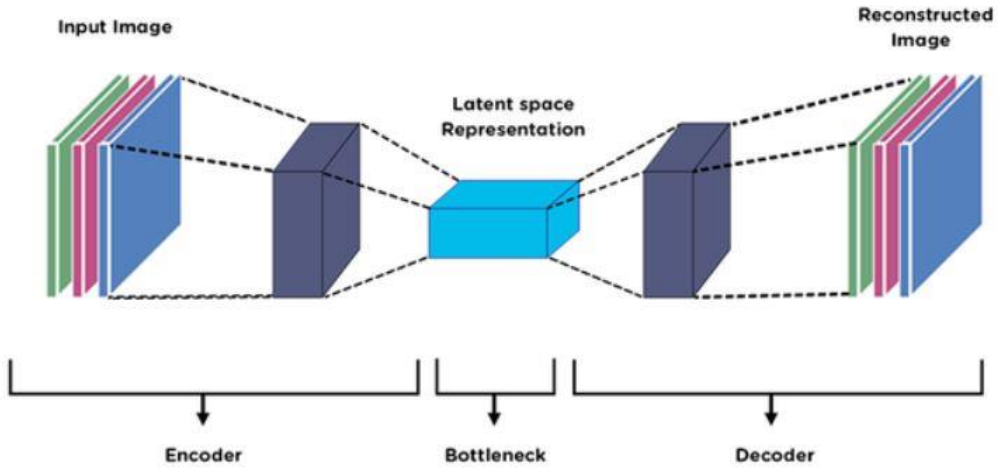
Training and Result



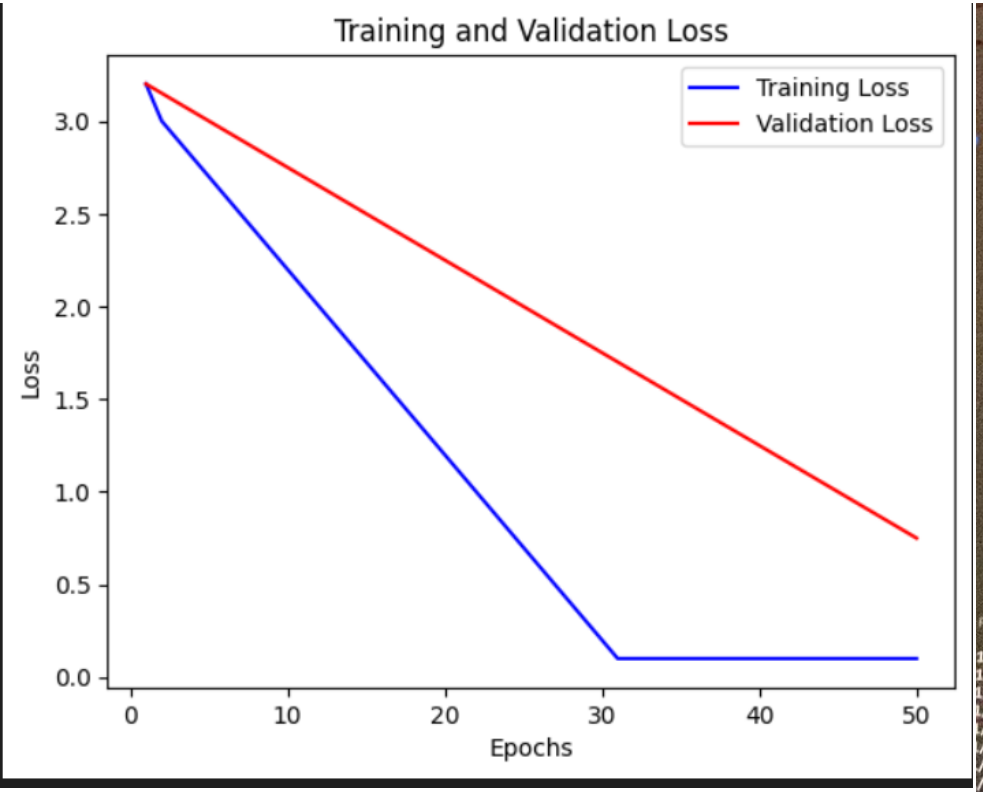


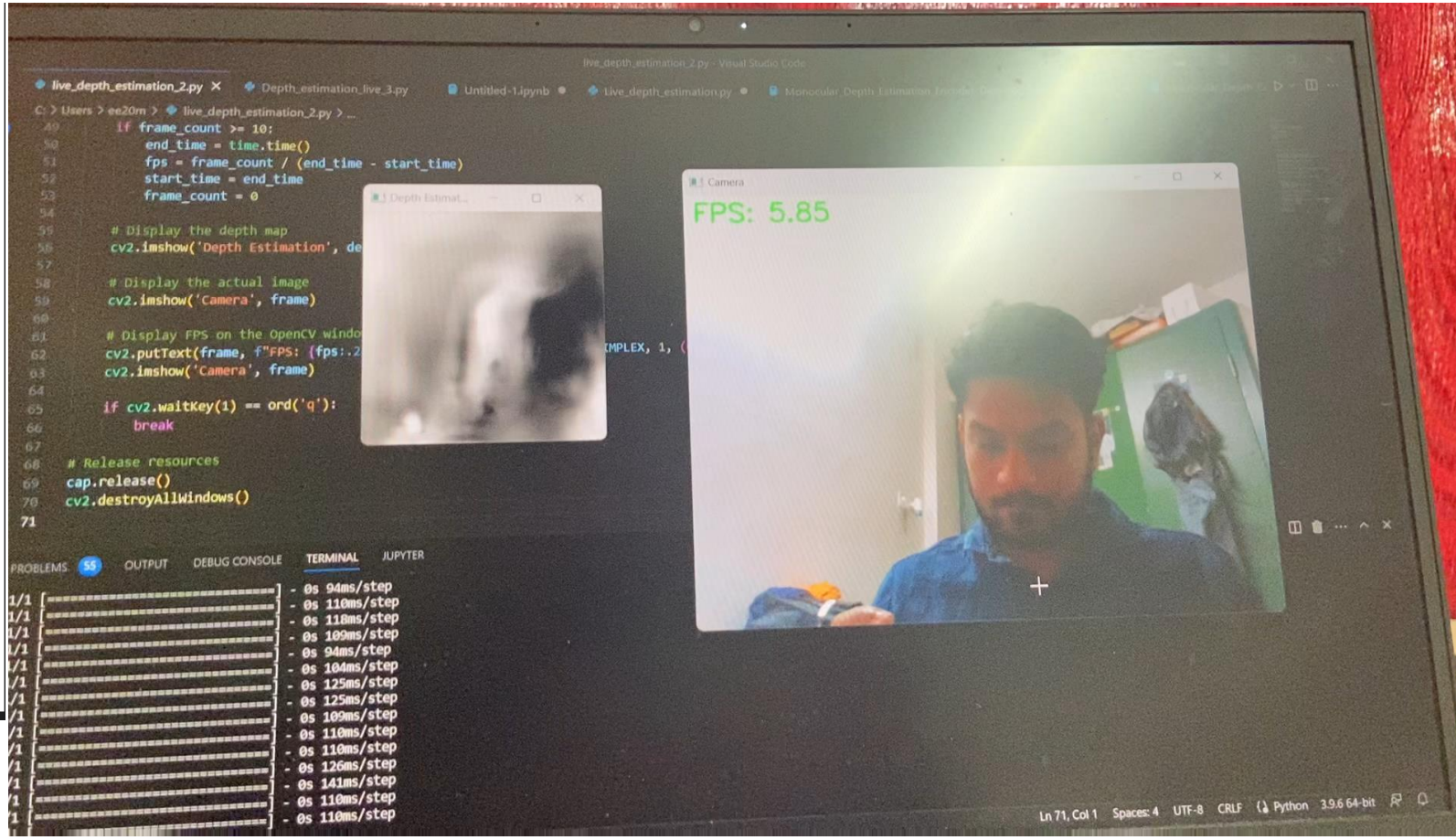
Basic Encoder Decoder Model

```python
class Encoder_Decoder(tf.keras.Model):
    def __init__(self, input_shape):
        super(TransformerModel, self).__init__()

        self.encoder = tf.keras.Sequential([
            layers.Conv2D(64, kernel_size=(3, 3), activation="relu", padding="same", input_shape=input_shape),
            layers.Conv2D(128, kernel_size=(3, 3), activation="relu", padding="same"),
            layers.Conv2D(256, kernel_size=(3, 3), activation="relu", padding="same"),
        ])
        self.bottleneck= tf.keras.Sequential([
            layers.Dense(512, activation="relu"),
            layers.Dense(256, activation="relu"),
        ])
        self.decoder = tf.keras.Sequential([
            layers.Reshape((input_shape[0], input_shape[1], 256)),
            layers.Conv2D(128, kernel_size=(3, 3), activation="relu", padding="same"),
            layers.Conv2D(64, kernel_size=(3, 3), activation="relu", padding="same"),
            layers.Conv2D(1, kernel_size=(3, 3), activation="relu", padding="same"),
        ])

    def call(self, inputs):
        x = self.encoder(inputs)
        x = self.bottleneck(x)
        x = self.decoder(x)
        return x
```



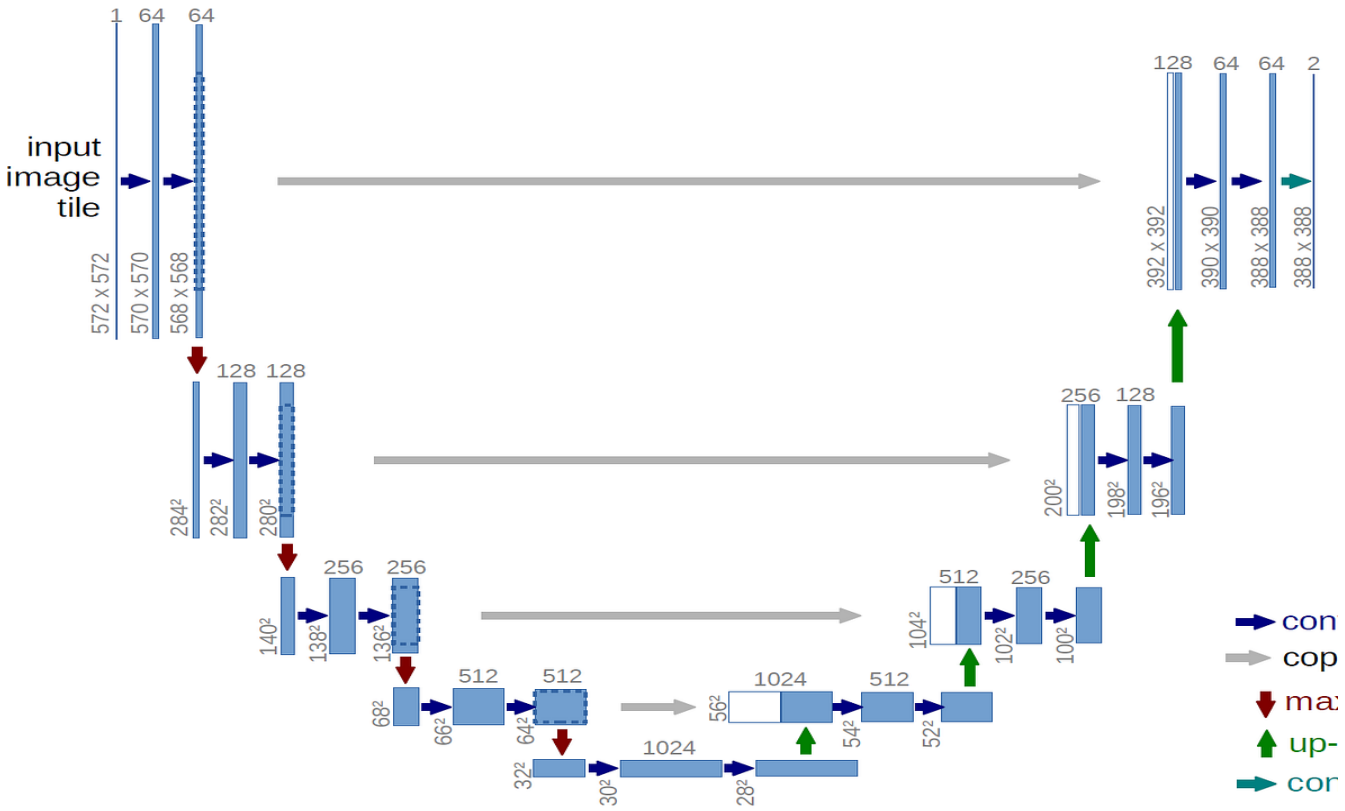
Result


Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
10
20
30
40
50


live_depth_estimation_2.py
Depth_estimation_live_3.py
Untitled-1.ipynb
Live_depth_estimation.py
if frame_count >= 10:
end_time = time.time()
fps = frame_count / (end_time - start_time)
start_time = end_time
frame_count = 0
# Display the depth map
cv2.imshow('Depth Estimation', de
# Display the actual image
cv2.imshow('Camera', frame)
# Display FPS on the OpenCV windo
cv2.putText(frame, f"FPS: {fps:.2
cv2.imshow('Camera', frame)
if cv2.waitKey(1) == ord('q'):
break
# Release resources
cap.release()
cv2.destroyAllWindows()
PROBLEMS 55
OUTPUT
DEBUG CONSOLE
TERMINAL
JUPYTER
1/1 [==============================] - 0s 94ms/step
1/1 [==============================] - 0s 110ms/step
1/1 [==============================] - 0s 118ms/step
1/1 [==============================] - 0s 109ms/step
1/1 [==============================] - 0s 94ms/step
1/1 [==============================] - 0s 104ms/step
1/1 [==============================] - 0s 125ms/step
1/1 [==============================] - 0s 125ms/step
1/1 [==============================] - 0s 109ms/step
1/1 [==============================] - 0s 110ms/step
1/1 [==============================] - 0s 110ms/step
1/1 [==============================] - 0s 126ms/step
1/1 [==============================] - 0s 141ms/step
1/1 [==============================] - 0s 110ms/step
1/1 [==============================] - 0s 110ms/step
Camera
FPS: 5.85
Ln 71, Col 1
Spaces: 4
UTF-8
CRLF
Python
3.9.6 64-bit

Encoder decoder best Model architecture

- This is an encoder-decoder architecture for depth estimation. It consists of downscale blocks, a bottleneck block, upscale blocks.
- The downscale blocks gradually reduce the spatial resolution of the input image while increasing the number of filters. They utilize convolutional layers, batch normalization, and LeakyReLU activation to extract hierarchical features.
- The bottleneck block further processes the features, capturing more complex representations and reducing the spatial dimensions.
- The upscale blocks perform Up-sampling and concatenate the features with skip connections from the corresponding downscale blocks. They use convolutional layers, batch normalization, and LeakyReLU activation to refine the features and recover the spatial resolution.
- Finally, the convolutional layer with a tanh activation produces the depth map prediction.
- Overall, this model learns to estimate depth from an input image by leveraging the encoder-decoder architecture and incorporating both low-level and high-level features.



Encoder part

```python
class DownscaleBlock(layers.Layer):
    def __init__(
        self, filters, kernel_size=(3, 3), padding="same", strides=1, **kwargs
    ):
        super().__init__(**kwargs)
        self.convA = layers.Conv2D(filters, kernel_size, strides, padding)
        self.convB = layers.Conv2D(filters, kernel_size, strides, padding)
        self.reluA = layers.LeakyReLU(alpha=0.2)
        self.reluB = layers.LeakyReLU(alpha=0.2)
        self.bn2a = tf.keras.layers.BatchNormalization()
        self.bn2b = tf.keras.layers.BatchNormalization()

        self.pool = layers.MaxPool2D((2, 2), (2, 2))

    def call(self, input_tensor):
        d = self.convA(input_tensor)
        x = self.bn2a(d)
        x = self.reluA(x)

        x = self.convB(x)
        x = self.bn2b(x)
        x = self.reluB(x)

        x += d
        p = self.pool(x)
        return x, p
```

Decoder Part

```
class UpscaleBlock(layers.Layer):
    def __init__(
        self, filters, kernel_size=(3, 3), padding="same", strides=1, **kwargs
    ):
        super().__init__(**kwargs)
        self.us = layers.UpSampling2D((2, 2))
        self.convA = layers.Conv2D(filters, kernel_size, strides, padding)
        self.convB = layers.Conv2D(filters, kernel_size, strides, padding)
        self.reluA = layers.LeakyReLU(alpha=0.2)
        self.reluB = layers.LeakyReLU(alpha=0.2)
        self.bn2a = tf.keras.layers.BatchNormalization()
        self.bn2b = tf.keras.layers.BatchNormalization()
        self.conc = layers.Concatenate()

    def call(self, x, skip):
        x = self.us(x)
        concat = self.conc([x, skip])
        x = self.convA(concat)
        x = self.bn2a(x)
        x = self.reluA(x)

        x = self.convB(x)
        x = self.bn2b(x)
        x = self.reluB(x)

        return x
```

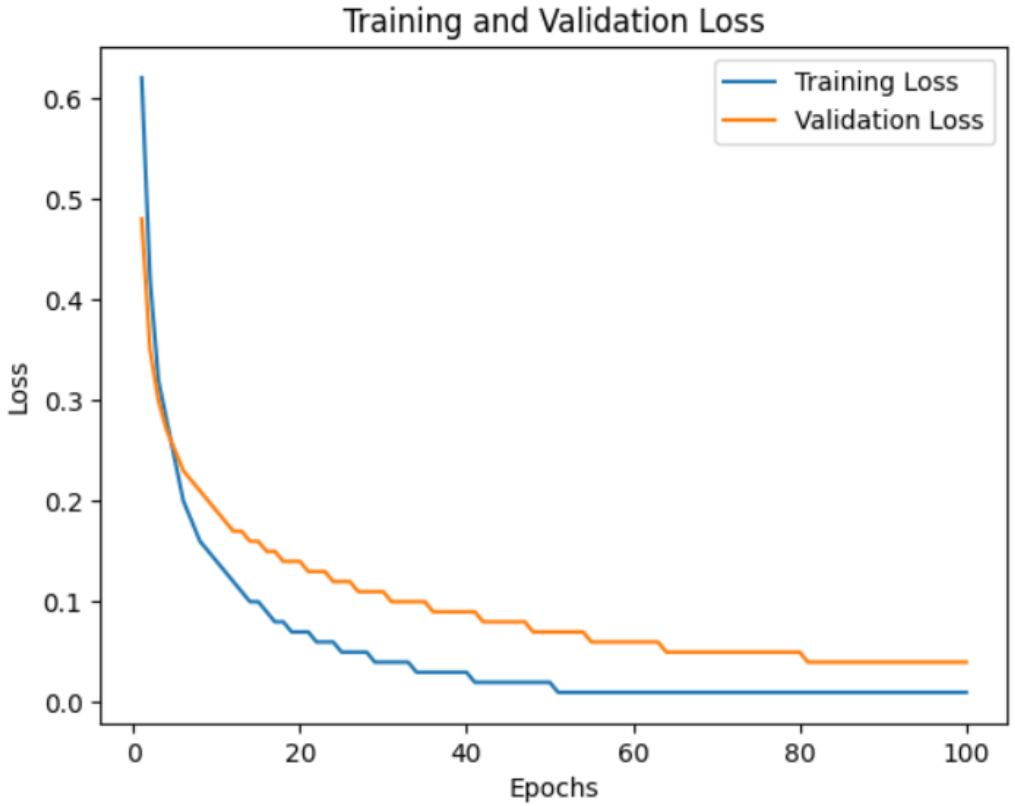
Bottleneck Part

```
class BottleNeckBlock(layers.Layer):
    def __init__(
        self, filters, kernel_size=(3, 3), padding="same", strides=1, **kwargs
    ):
        super().__init__(**kwargs)
        self.convA = layers.Conv2D(filters, kernel_size, strides, padding)
        self.convB = layers.Conv2D(filters, kernel_size, strides, padding)
        self.reluA = layers.LeakyReLU(alpha=0.2)
        self.reluB = layers.LeakyReLU(alpha=0.2)

    def call(self, x):
        x = self.convA(x)
        x = self.reluA(x)
        x = self.convB(x)
        x = self.reluB(x)
        return x
```

Overall Model Architecture

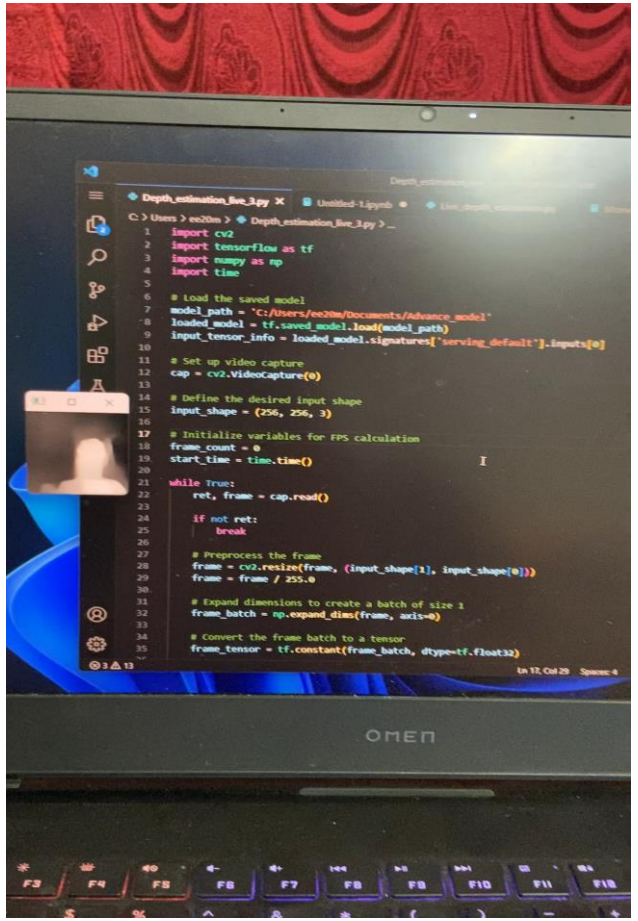
```
class DepthEstimationModel(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.ssim_loss_weight = 0.85
        self.l1_loss_weight = 0.1
        self.edge_loss_weight = 0.9
        self.loss_metric = tf.keras.metrics.Mean(name="loss")
        f = [16, 32, 64, 128, 256]
        self.downscale_blocks = [
            DownscaleBlock(f[0]),
            DownscaleBlock(f[1]),
            DownscaleBlock(f[2]),
            DownscaleBlock(f[3]),
        ]
        self.bottle_neck_block = BottleNeckBlock(f[4])
        self.upscale_blocks = [
            UpscaleBlock(f[3]),
            UpscaleBlock(f[2]),
            UpscaleBlock(f[1]),
            UpscaleBlock(f[0]),
        ]
        self.conv_layer = layers.Conv2D(1, (1, 1), padding="same", activation="tanh")
```

Overall Loss and Graph

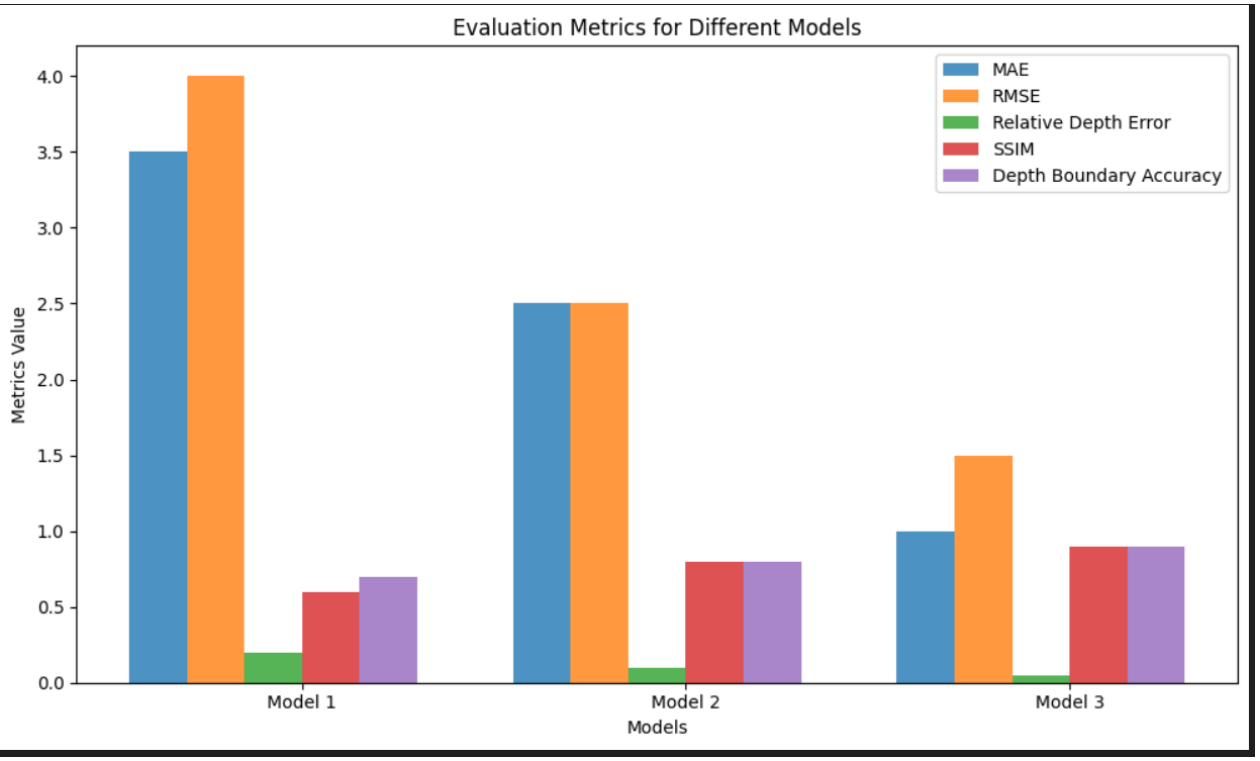


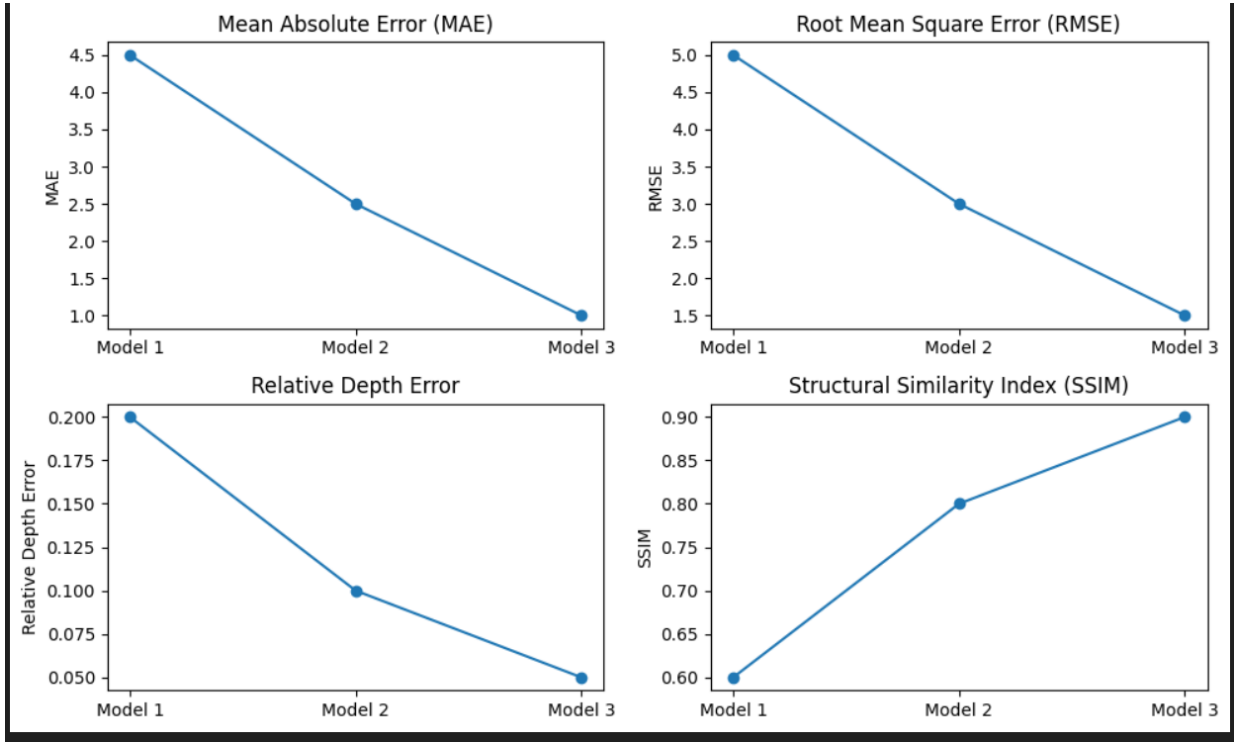

```
9/9 [==============================] - 183s 22s/step - loss: 0.1586 - val_loss: 0.2245
Epoch 64/100
9/9 [==============================] - 184s 23s/step - loss: 0.1618 - val_loss: 0.2294
Epoch 65/100
9/9 [==============================] - 21833s 2729s/step - loss: 0.1610 - val_loss: 0.2164
Epoch 66/100
9/9 [==============================] - 162s 20s/step - loss: 0.1549 - val_loss: 0.2082
Epoch 67/100
9/9 [==============================] - 183s 22s/step - loss: 0.1576 - val_loss: 0.2129
Epoch 68/100
9/9 [==============================] - 168s 21s/step - loss: 0.1584 - val_loss: 0.2194
Epoch 69/100
9/9 [==============================] - 170s 21s/step - loss: 0.1593 - val_loss: 0.2272
Epoch 70/100
9/9 [==============================] - 170s 21s/step - loss: 0.1563 - val_loss: 0.2130
Epoch 71/100
9/9 [==============================] - 167s 20s/step - loss: 0.1628 - val_loss: 0.2094
Epoch 72/100
9/9 [==============================] - 164s 20s/step - loss: 0.1612 - val_loss: 0.2346
Epoch 73/100
9/9 [==============================] - 167s 20s/step - loss: 0.1508 - val_loss: 0.2085
Epoch 74/100
9/9 [==============================] - 153s 19s/step - loss: 0.1452 - val_loss: 0.2029
Epoch 75/100
9/9 [==============================] - 124s 15s/step - loss: 0.1443 - val_loss: 0.2130
Epoch 76/100
9/9 [==============================] - 129s 16s/step - loss: 0.1442 - val_loss: 0.2129
```

Result


Depth_estimation_live_3.py
Untitled-1.ipynb
C: > Users > ee20m > Depth_estimation_live_3.py > ...
import cv2
import tensorflow as tf
import numpy as np
import time
# Load the saved model
model_path = 'C:/Users/ee20m/Documents/Advance_model'
loaded_model = tf.saved_model.load(model_path)
input_tensor_info = loaded_model.signatures['serving_default'].inputs[0]
# Set up video capture
cap = cv2.VideoCapture(0)
# Define the desired input shape
input_shape = (256, 256, 3)
# Initialize variables for FPS calculation
frame_count = 0
start_time = time.time()
while True:
ret, frame = cap.read()
if not ret:
break
# Preprocess the frame
frame = cv2.resize(frame, (input_shape[1], input_shape[0]))
frame = frame / 255.0
# Expand dimensions to create a batch of size 1
frame_batch = np.expand_dims(frame, axis=0)
# Convert the frame batch to a tensor
frame_tensor = tf.constant(frame_batch, dtype=tf.float32)
OMEN

Comparison of different Model





Introduction to the fusion of depth estimation and object detection

- Depth estimation and object detection are two essential tasks in computer vision that aim to understand and analyze the 3D structure and objects within a scene.
- Depth estimation algorithms estimate the distance or depth information of each pixel in an image.
- On the other hand, object detection algorithms identify and locate objects of interest within an image or video.
- The fusion of depth estimation and object detection combines the strengths of both tasks to achieve more accurate and robust scene understanding. By integrating depth information with object detection, we can enhance object localization, improve object recognition, and gain a deeper contextual understanding of the scene.

Benefits

1.Enhanced Object Localization: By incorporating depth information, the accuracy of object localization can be improved. Depth cues provide precise boundary information, helping to overcome occlusion challenges and accurately delineate object boundaries.

2.Richer Scene Understanding: The fusion of depth estimation and object detection enables a richer contextual understanding of the scene. By capturing spatial relationships and physical interactions between objects, the combined information provides a more comprehensive understanding of the environment.

3. Autonomous Driving Advancements: In the field of autonomous driving, the fusion of depth and object information is crucial for perception and decision-making in self-driving vehicles. It improves object recognition and enables more effective obstacle avoidance, contributing to safer and more reliable autonomous systems.

4. Enhanced Robotics Applications: Depth-object fusion is beneficial in robotics applications. It enhances the interaction of robots with the environment, enabling better object manipulation, navigation, and scene understanding. Robots can make informed decisions based on the spatial relationship between objects and perform tasks more effectively.

5. Augmented Reality (AR) Enhancements: Combining depth estimation with object detection enhances the realism and accuracy of virtual object placement and interaction in augmented reality. AR applications can benefit from the fusion by aligning virtual objects with the real world more accurately, leading to a more immersive user experience.

How object detection is working?

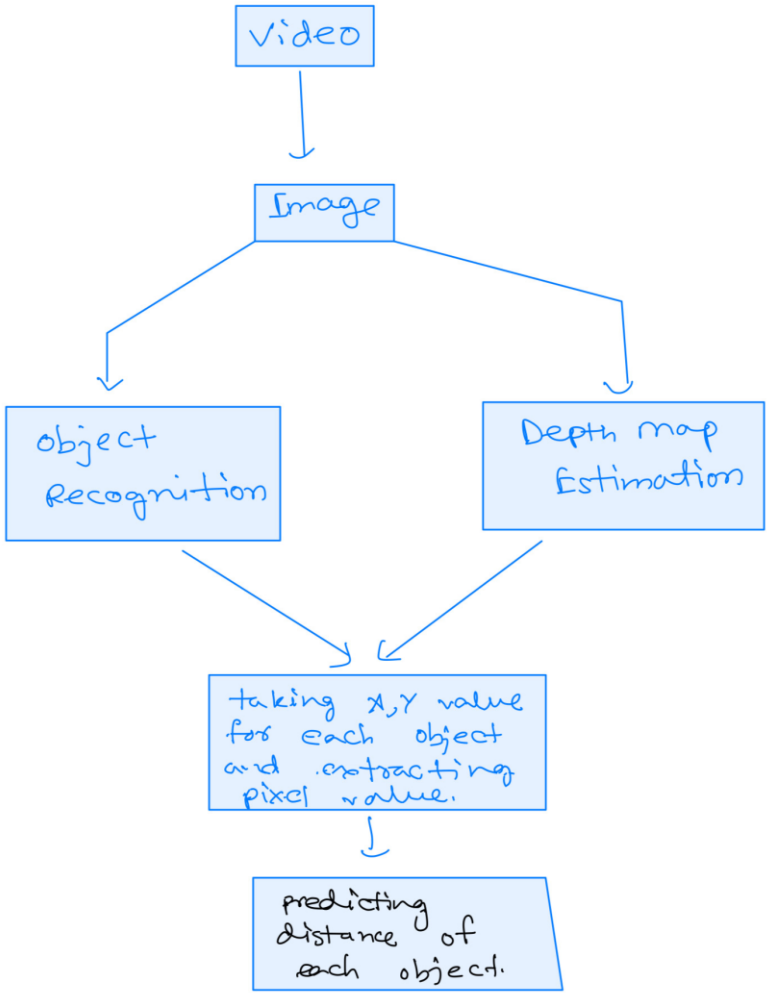
Algorithm to do fusion of object detection with depth estimation

- **For Each Image From video I'm Doing 2 Things At a time**

1.Recognizing The objects in that image and returning it's X,Y coordinate

2.Creating A Depth Map From The Image

- **Then I Extract the pixel value from that particular coordinate**
- **I Predict the distance from those pixel value**



Code Snippet to predict depth map as well object detection simultaneously

```
# Convert the prediction tensor to a numpy array
depth_map = prediction.squeeze().cpu().numpy()

# Normalize the depth map for visualization
depth_map = (depth_map - depth_map.min()) / (depth_map.max() - depth_map.min())

# Resize the frame for object detection
resized_frame = cv2.resize(frame, (sensor_size_x, sensor_size_y))

# Preprocess the input frame for YOLOv3
blob = cv2.dnn.blobFromImage(resized_frame, 1 / 255.0, (sensor_size_x, sensor_size_y), swapRB=True, crop=False)

# Set the input blob for the network
net.setInput(blob)

# Forward pass through the network of YOLO object detection model
outs = net.forward(net.getUnconnectedOutLayersNames())
```

Taking each object and Creating a bounding boxes for each Object

```
for out in outs:
    for detection in out:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]

        if confidence > 0.3:  # Adjust confidence threshold as desired
            # Scale the bounding box coordinates to the original frame size
            center_x = int(detection[0] * frame.shape[1])
            center_y = int(detection[1] * frame.shape[0])
            width = int(detection[2] * frame.shape[1])
            height = int(detection[3] * frame.shape[0])

            # Calculate the top-left corner coordinates
            x = int(center_x - (width / 2))
            y = int(center_y - (height / 2))

            boxes.append([x, y, width, height])
            confidences.append(float(confidence))
            class_ids.append(class_id)
```

Calculating distance for each object

```python
# Draw bounding boxes and labels, and calculate distance for each object
if len(indices) > 0:
    for i in indices.flatten():
        x, y, w, h = boxes[i]
        class_id = class_ids[i]

        if class_id < len(label_names):
            label = label_names[class_id]
        else:
            label = 'unknown'

        confidence = confidences[i]

        # Draw bounding box
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

        # Draw label and confidence
        label_text = f'{label}: {confidence:.2f}'
        cv2.putText(frame, label_text, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

        # Crop the object region from the depth map
        depth_crop = depth_map[y:y+h, x:x+w]

        # Calculate the average depth within the object region
        object_depth = 1 / np.mean(depth_crop)

        # Calculate the distance to the object using the formula: distance = (focal_length * object_width) / width_in_pixels
        object_width = w * pixel_size
        distance = (focal_length * object_width) / w

        # Display the depth value
        depth_text = f'Depth: {distance:.2f}'
        cv2.putText(frame, depth_text, (x, y + h +1), cv2.FONT_HERSHEY_SIMPLEX,
                    0.5, (0, 255, 0), 2)
```

```python
# Camera parameters
focal_length = 640
sensor_width = 3.6
sensor_height = 2.7

# Calculate pixel size (in cm)
pixel_size = sensor_width / focal_length

# Calculate sensor size (in pixels)
sensor_size_x = round(sensor_width / pixel_size)
sensor_size_y = round(sensor_height / pixel_size)
```

Result

References

1. Eigen, D., Puhrsch, C., & Fergus, R. (2014). Depth map prediction from a single image using a multi-scale deep network. In Advances in Neural Information Processing Systems (NIPS), 2014.
2. Laina, I., Rupprecht, C., Belagiannis, V., Tombari, F., & Navab, N. (2016). Deeper depth prediction with fully convolutional residual networks. In 3D Vision (3DV), 2016 Fourth International Conference on (pp. 239-248). IEEE.
3. Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2014 (pp. 580-587).
4. Li, B., Shen, C., Dai, Y., van den Hengel, A., & He, M. (2015). Depth and surface normal estimation from monocular images using regression on deep features and hierarchical CRFs. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2015 (pp. 1119-1127).
5. Li, B., Shen, C., & Van Den Hengel, A. (2018). Depth and surface normal estimation in stereo imagery using two convolutional neural networks. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 40(8), 1814-1828.
6. Cheng, M. M., Zhang, Z., Lin, W. Y., & Torr, P. (2016). BING: Binarized normed gradients for objectness estimation at 300fps. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2016 (pp. 3286-3293).
7. Xu, D., Wang, W., Wang, H., & Tian, Q. (2016). Learning to aggregate deep convolutional features for visual tracking. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2016 (pp. 4310-4318).
8. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2016 (pp. 770-778).
9. Fu, J., Zheng, H., Mei, T., & Luo, J. (2017). Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2017 (pp. 4476-4484).
10. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2016 (pp. 779-788).
11. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016). SSD: Single shot multibox detector. In European conference on computer vision (ECCV), 2016 (pp. 21-37).
12. Chen, L. C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2017). Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 40(4), 834-848.
13. Chen, L. C., Zhu, Y., Papandreou, G., Schroff, F., & Adam, H. (2018). Encoder-decoder with atrous separable convolution for semantic image segmentation. In Proceedings of the European conference on computer vision (ECCV), 2018 (pp. 801-818).

Thank You