Coded Deep Depth Estimation [70pts]

This is the part 2 of the project 2. Part 1 is this project is present here.

The aim of this section is two folds:

1. Given All in focus (AiF) or pinhole model RGB images and Ground Truth Depth maps, train your monocular depth estimation method using NYUv2 and UMDCodedVO-LivingRoom dataset only and test them on out of domain ICL-NUIM and UMDCodedVO-DiningRoom datasets. ((Link to test set) [https://drive.google.com/drive/folders/12U8BH-AWUA4DgbOValO-_hNI_Z9RgMXr?usp=sharing]) [25pts]

2. Given an Coded RGB images and Ground Truth Depth maps, train your monocular depth estimation method using NYUv2 and UMDCodedVO-LivingRoom dataset only and test them on out of domain ICL-NUIM and UMDCodedVO-DiningRoom datasets. (Use the code for Part 1 to generate coded images for ICL-NUIM and UMDCodedVO-DiningRoom datasets) [25pts]

Evaluate and compare results on testing dataset for both the above mentioned monocular depth methods. Use Rel-Abs and RMSE metric for quantitative evaluation. Use 'viridis' or 'plasma' color scheme for qualitative evaluation link. Note: Both qualitative and quantitative evaluation must be with metric units i.e. from 0m to 6m. [20pts]

Note: Use of A100 is highly recommended

A. Import Necessary Libraries

```
# Import necessary libraries
import os  # For setting environment variables and interacting with the operating system
import torch  # PyTorch for deep learning computations
import torch.nn as nn  # Neural network modules from PyTorch
os.environ["OPENCV_IO_ENABLE_OPENEXR"] = "1"  # Enable OpenEXR format support in OpenCV
import cv2  # OpenCV for computer vision tasks
import numpy as np  # For numerical operations
import torchvision.transforms as transforms  # For data augmentation and transformation
from torch.utils.data import DataLoader, ConcatDataset, Dataset  # PyTorch utilities for data handling
from tqdm import tqdm  # Progress bar library to monitor loops
import time  # To track time during execution

# Environment Setup

# Select device for computation, CUDA if available, otherwise CPU
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)  # Print the selected device (either 'cuda' or 'cpu')
```

```
cuda
```

B. Data Handling

(Nothing to modify here)

```
# Data Loader class for handling image and depth pairs
class ImageDepthDataset(Dataset):
    def __init__(self, path: str, codedDir: str = "Coded", cache: bool
= True, is_blender: bool = False,
                 image_size=(480, 640), scale_factor: float = 5000,
limit=None):
        """
        Initializes the dataset by loading image and depth file pairs.

        Parameters:
        - path: Directory where the dataset is stored.
        - codedDir: Directory within the path that contains the coded
images (default: "Coded").
        - cache: Whether to load and store the data in memory (True)
or process on-the-fly (False).
        - is_blender: Flag to indicate whether the depth data is from
Blender (EXR format) or not (PNG format).
        - image_size: Desired size of the images after cropping
(default: (480, 640)).
        - scale_factor: Scale factor to adjust depth values (default:
5000).
        - limit: Maximum number of image-depth pairs to load. If None,
loads all available files.
        """
        self.path = path  # Path to the dataset
        self.is_blender = is_blender  # Whether the depth images are
from Blender
        self.transform =
transforms.Compose([transforms.CenterCrop(image_size)])  # Image
transformation (cropping)
        self.data = []  # Container to store processed or unprocessed
file paths/data
        self.scale_factor = scale_factor  # Scaling factor to adjust
depth values
        self.limit = limit if limit else -1  # Limit on the number of
files to load, if provided
        dir_path = os.path.join(path, codedDir)  # Path to the
directory containing coded images
        print(codedDir, is_blender, scale_factor)  # Print dataset
configuration for debugging
```

```
        # Get list of coded image files (sorted) from the directory,
apply limit if set
        files = sorted([p for p in os.listdir(dir_path) if
p.endswith(".png")])[:self.limit]

        # Iterate through the files and append their coded and depth
file paths to self.data
        for file in files:
            coded_file = os.path.join(path, codedDir, file)  # Full
path to the coded image
            # Path to corresponding depth file (EXR for Blender,
otherwise PNG)
            depth_file = os.path.join(path, "depth",
file.replace(".png", ".exr") if is_blender else file)

            # If caching is enabled, process and store the data in
memory
            if cache:
                self.data.append(self.process(coded_file, depth_file))
            # Otherwise, store the file paths for on-the-fly
processing
            else:
                self.data.append((coded_file, depth_file))

        self.cache = cache  # Whether the dataset is cached in memory
or not
        self.len = len(self.data)  # Number of files loaded
        print(self.len)  # Print the number of files loaded for
debugging

    def process(self, coded_file: str, depth_file: str):
        """
        Processes an image-depth pair by loading, transforming, and
scaling them.

        Parameters:
        - coded_file: Path to the coded image file.
        - depth_file: Path to the corresponding depth image file.

        Returns:
        A dictionary with the transformed and scaled image and depth.
        """
        # Load coded image as a tensor (permute to change dimensions
from HxWxC to CxHxW)
        coded = torch.from_numpy(cv2.imread(coded_file)).permute(2, 0,
1)
        # print(f"Coded Image Shape (before transform):
{coded.shape}")
        # Process depth based on the is_blender flag (EXR or PNG
```

```
format)
        if self.is_blender:
            # Load depth from EXR format (for Blender data)
            raw_depth = cv2.imread(depth_file, cv2.IMREAD_ANYCOLOR |
cv2.IMREAD_ANYDEPTH)
            metric_depth = torch.from_numpy(raw_depth[:, :, 0])  #
Extract depth channel
        else:
            # Load depth and apply scale factor (for non-Blender data)
            metric_depth = torch.from_numpy(cv2.imread(depth_file,
cv2.IMREAD_ANYCOLOR | cv2.IMREAD_ANYDEPTH) / self.scale_factor)
        # print(f"Depth Image Shape (before transform):
{metric_depth.shape}")
        # Return a dictionary with processed image and depth data
        return {
            "image_id": coded_file.split('/')[-1][:-4],  # Extract
image ID from file name
            "Coded": self.transform(coded.to(torch.float32)) / 255.0,
# Normalize the coded image
            "Depth": self.transform(metric_depth.to(torch.float32))  #
Transform the depth image
        }

    def __len__(self):
        """
        Returns the number of samples in the dataset.
        """
        return self.len

    def __getitem__(self, idx):
        """
        Retrieves a single sample from the dataset by index.

        Parameters:
        - idx: Index of the sample to retrieve.

        Returns:
        The processed image-depth pair if cached, otherwise processes
on the fly.
        """
        # If cached, return the pre-processed data
        if self.cache:
            return self.data[idx]
        # Otherwise, process the data on-the-fly and return
        else:
            return self.process(*self.data[idx])
```

C. Network architecture goes here:

Follow network details from here

```
import torch
import torch.nn as nn
import torchvision.transforms.functional as TF  # Import transforms
for cropping

class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3,
padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3,
padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.block(x)

class U_Net(nn.Module):
    def __init__(self, img_ch=3, output_ch=1):
        super(U_Net, self).__init__()
        self.encoder1 = ConvBlock(img_ch, 64)
        self.encoder2 = ConvBlock(64, 128)
        self.encoder3 = ConvBlock(128, 256)
        self.encoder4 = ConvBlock(256, 512)
        self.encoder5 = ConvBlock(512, 1024)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.upconv5 = nn.ConvTranspose2d(1024, 512, kernel_size=2,
stride=2)
        self.decoder5 = ConvBlock(1024, 512)

        self.upconv4 = nn.ConvTranspose2d(512, 256, kernel_size=2,
stride=2)
        self.decoder4 = ConvBlock(512, 256)

        self.upconv3 = nn.ConvTranspose2d(256, 128, kernel_size=2,
stride=2)
        self.decoder3 = ConvBlock(256, 128)

        self.upconv2 = nn.ConvTranspose2d(128, 64, kernel_size=2,
```

```
stride=2)
        self.decoder2 = ConvBlock(128, 64)

        # self.upconv1 = nn.ConvTranspose2d(64, 64, kernel_size=2,
stride=2)
        self.decoder1 = ConvBlock(128, 64)

        self.final_conv = nn.Conv2d(64, output_ch, kernel_size=1)

    def forward(self, x):
        # Encoder
        # print(f"Input shape: {x.shape}")
        enc1 = self.encoder1(x)
        # print(f"After encoder1: {enc1.shape}")# (1, 64, 480, 640)
        enc2 = self.encoder2(self.pool(enc1))
        # print(f"After encoder2: {enc2.shape}")# (1, 128, 240, 320)
        enc3 = self.encoder3(self.pool(enc2))
        # print(f"After encoder3: {enc3.shape}")# (1, 256, 120, 160)
        enc4 = self.encoder4(self.pool(enc3))
        # print(f"After encoder4: {enc4.shape}")# (1, 512, 60, 80)
        enc5 = self.encoder5(self.pool(enc4))
        # print(f"After encoder5: {enc5.shape}")# (1, 1024, 30, 40)

        # Decoder
        dec5 = self.upconv5(enc5)
        # print(f"After up5: {dec5.shape}")# (1, 512, 60, 80)
        dec5 = torch.cat((enc4, dec5), dim=1)
        # print(f"After skip: {dec5.shape}")# Skip connection (1,
1024, 60, 80)
        dec5 = self.decoder5(dec5)
        # print(f"After dec5: {dec5.shape}")# (1, 512, 60, 80)

        dec4 = self.upconv4(dec5)
        # print(f"After up4: {dec4.shape}")# (1, 256, 120, 160)
        dec4 = torch.cat((enc3, dec4), dim=1)
        # print(f"After skip: {dec4.shape}")# Skip connection (1, 512,
120, 160)
        dec4 = self.decoder4(dec4)
        # print(f"After dec4: {dec4.shape}")# (1, 256, 120, 160)

        dec3 = self.upconv3(dec4)
        # print(f"After up: {dec3.shape}")# (1, 128, 240, 320)
        dec3 = torch.cat((enc2, dec3), dim=1)
        # print(f"After skip: {dec3.shape}")# Skip connection (1, 256,
240, 320)
        dec3 = self.decoder3(dec3)  # (1, 128, 240, 320)
        # print(f"After dec3: {dec3.shape}")

        dec2 = self.upconv2(dec3)
        # print(f"After up2: {dec2.shape}")# (1, 64, 480, 640)
```

```
        dec2 = torch.cat((enc1, dec2), dim=1)
        # print(f"After skip: {dec2.shape}")# Skip connection (1, 128, 
480, 640)
        # dec2 = self.decoder2(dec2)  # (1, 64, 480, 640)
        # print(f"After dec2: {dec2.shape}")

        # dec1 = self.upconv1(dec2)  # (1, 64, 960, 1280)
        # print(f"After up: {dec1.shape}")
        # Apply center cropping to match the size of x (original 
image)
        # dec1 = TF.center_crop(dec1, x.shape[2:])  # Use center crop 
to match the original input size
        # dec1 = torch.cat((x, dec1), dim=1)  # Skip connection with 
the original image
        # print(f"After skip: {dec1.shape}")
        dec1 = self.decoder1(dec2)
        # print(f"After dec1: {dec1.shape}")# (1, 64, 480, 640)

        out = self.final_conv(dec1)  # (1, output_ch, 480, 640)

        return out
```

D. Define Experiment/Training details here:

```
# Define Experiment Configurations (from config.py)
class Experiment:
    def __init__(self, config_name):
        """
        Initializes the experiment based on the configuration name.

        Parameters:
        - config_name: The name of the experiment configuration to 
use.

        This constructor sets up the model, training parameters, 
datasets, and loss function
        based on the specific configuration.
        """
        if config_name == "MetricWeightedLossBlenderNYU":
            # Set the model architecture (e.g., U-Net) for this 
experiment
            self.model = U_Net
            # Defome Number of training epochs
            self.epochs = 45
            # Define Batch size for training
            self.batch_size = 1
            # Define Learning rate for the optimizer
            self.learning_rate = 0.0001
            # Modify the list of training datasets with specific 
configurations
```

```
            self.train_datasets = [
                {"nyu_data": ["rgb", 1000, False]},  # Dataset name:
Coded Images Subfolder, scale factor, is_blender flag
                {"LivingRoom1": ["rgb", 1, True]}  #  Dataset name:
Coded Images Subfolder, scale factor, is_blender flag
            ]

            # Modify the list of test datasets with specific
configurations
            self.test_datasets = [
                # Test dataset configuration examples:
                # {"Corridor": ["Codedphasecam-27Linear", 1, True]},
                {"DiningRoom": ["Codedphasecam-27Linear-New", 1,
True]}  # Dataset for testing
                # Additional test datasets can be added here
            ]
            # Set the loss function to the custom weighted mean
squared error (MSE) loss
            self.loss_fn = self.my_loss
        # Add more configurations as needed for different experiments

    def my_loss(self, output, target):
        """
        Write a loss function.

        Parameters:
        - output: The predicted output from the model.
        - target: The ground truth depth values.

        Returns:
        The loss value.
        """
        weighted_depth = 2**(0.3*target)
        weighted_mse = weighted_depth * (output - target) ** 2

    # Reduce the loss (either using mean or sum)
        loss = torch.mean(weighted_mse)
        # Calculate the loss based on target values
        return loss

# Helper Functions

def count_parameters(model):
    """
    Counts the number of trainable parameters in a PyTorch model.

    Parameters:
    - model: The PyTorch model whose parameters are to be counted.

    Returns:
```

```
    The total number of trainable parameters in the model.
    """
    return sum(p.numel() for p in model.parameters() if
p.requires_grad)

def init_weights(net, init_type="normal", gain=0.02):
    """
    Initializes the weights of the network layers based on the
specified initialization type.

    Parameters:
    - net: The neural network (PyTorch model) whose weights are to be
initialized.
    - init_type: The type of weight initialization ('normal',
'xavier', 'kaiming', or 'orthogonal').
    - gain: A scaling factor for the initialization (applies to
certain initialization methods).

    This function defines an internal function `init_func` that is
applied to each layer of the network.
    The weights of convolutional and linear layers are initialized
based on the `init_type`,
    while batch normalization layers have their weights and biases
initialized separately.
    """

    def init_func(m):
        """
        Applies the initialization function to each layer `m` in the
network.

        This function checks the type of layer (Conv, Linear,
BatchNorm2d) and applies the
        appropriate initialization method to its weights and biases.
        """
        classname = m.__class__.__name__  # Get the class name of the
layer
        if hasattr(m, "weight") and (classname.find("Conv") != -1 or
classname.find("Linear") != -1):
            # If the layer has a 'weight' attribute and is a Conv or
Linear layer
            if init_type == "normal":
                nn.init.normal_(m.weight.data, 0.0, gain)  # Normal
distribution initialization
            elif init_type == "xavier":
                nn.init.xavier_normal_(m.weight.data, gain=gain)  #
Xavier initialization
            elif init_type == "kaiming":
                nn.init.kaiming_normal_(m.weight.data, a=0,
mode="fan_in")  # Kaiming (He) initialization
```

```
            elif init_type == "orthogonal":
                nn.init.orthogonal_(m.weight.data, gain=gain)  #
Orthogonal initialization

            # If the layer has a bias term, initialize it to 0
            if hasattr(m, "bias") and m.bias is not None:
                nn.init.constant_(m.bias.data, 0.0)

        # If the layer is a BatchNorm2d layer, initialize its weight
to 1 and bias to 0
        elif classname.find("BatchNorm2d") != -1:
            nn.init.normal_(m.weight.data, 1.0, gain)  # Initialize
BatchNorm
```

E: Training Loop Goes here:

```
# Training Loop Function
def train(model, dataloader, test_loader, optimizer, criterion,
epochs, checkpoint_path):
    """
    Trains the model over multiple epochs and saves the best-
performing model based on the loss.

    Parameters:
    - model: The neural network model to be trained.
    - dataloader: DataLoader for the training data.
    - test_loader: DataLoader for the test/validation data (if
evaluation is done during training).
    - optimizer: Optimizer for updating the model's weights.
    - criterion: Loss function used for training (e.g., MSELoss).
    - epochs: Number of epochs to train the model.
    - checkpoint_path: Path to save model checkpoints during training.
    """

    epoch_start = 0  # Starting epoch (useful for resuming training)
    best_loss = float('inf')  # Initialize best loss with a very large
value for comparison

    # Load checkpoint if it exists (for resuming training from the
last saved state)
    if os.path.exists(checkpoint_path):
        checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(checkpoint['model_state_dict'])  # Load
model weights
        epoch_start = checkpoint['epoch']  # Load the last epoch
completed
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
# Load optimizer state

    # Main training loop
```

```
    for epoch in range(epoch_start, epochs):
        print("epoch: ", epoch)
        model.train()  # Set model to training mode
        total_loss = 0  # Reset total loss for the current epoch

        # Loop over each batch in the dataloader
        for batch in dataloader:
            optimizer.zero_grad()  # Reset the gradients from the
previous step
            inputs, targets = batch['Coded'].to(device),
batch['Depth'].to(device)  # Move input and target to the device
(GPU/CPU)
            # print("Input",inputs.shape)
            # print("Target",targets.shape)
            outputs = model(inputs)  # Forward pass: compute model
predictions
            targets = targets.unsqueeze(1)  # Add channel dimension to
targets (for compatibility with model output)
            loss = criterion(outputs, targets)  # Compute the loss
between outputs and targets
            loss.backward()  # Backpropagation: compute gradients
            optimizer.step()  # Update model weights based on computed
gradients
            total_loss += loss.item()  # Accumulate the loss for the
current batch

        # Calculate the average loss for the epoch
        avg_loss = total_loss / len(dataloader)
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss}")

        # Save model checkpoint after each epoch
        torch.save({
            'epoch': epoch,  # Save the current epoch
            'model_state_dict': model.state_dict(),  # Save model's
state (weights)x``
            'optimizer_state_dict': optimizer.state_dict(),  # Save
optimizer's state
            'loss': avg_loss,  # Save the average loss for this epoch
        }, checkpoint_path)

        # Save the best model checkpoint if the current epoch has the
lowest loss so far
        if avg_loss < best_loss:
            best_loss = avg_loss  # Update best loss
            best_checkpoint_path = checkpoint_path.replace('.pt',
'_best.pt')  # Create a file name for the best checkpoint
            torch.save({
                'epoch': epoch,  # Save the current epoch
                'model_state_dict': model.state_dict(),  # Save the
model's state (weights)
```

```
                'optimizer_state_dict': optimizer.state_dict(),  #
Save the optimizer's state
                'loss': avg_loss,  # Save the average loss for this
epoch
            }, best_checkpoint_path)  # Save the best model as a
separate file
            print(f"New best model saved with loss {best_loss:.4f}")

        # Uncomment this block if you want to evaluate the model after
every few epochs
        # This block will run the validation step every 5 epochs and
print the test loss
        # if epoch % 5 == 0:
        #   test_l1, test_l1_under3 = validate(model, test_loader,
nn.L1Loss(), device)
        #   print(f"Test L1 error: {test_l1}")
        #   print(f"Test L1 error for depth < 3m: {test_l1_under3}")
```

F: Evaluation Metrics:

```
import torch
import numpy as np
import time
import os
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Define metrics
def abs_rel(pred, target):
    return torch.mean(torch.abs(target - pred) / target)

def rmse(pred, target):
    return torch.sqrt(torch.mean((target - pred) ** 2))

# Convert tensor to numpy
def to_numpy(img: torch.Tensor):
    return np.clip(img.detach().cpu().numpy(), 0, 6)  # Clip to range
[0m, 6m]

# Evaluate Function
def evaluate(model, dataloader, device, output_dir,
color_map='viridis'):
    model.eval()  # Set model to evaluation mode

    abs_rel_errors = []
    threshold_accs = []
    rms_errors = []
    inference_times = []
```

```
    # Ensure output directory exists
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for i, batch in enumerate(dataloader):
        # Get the input image and ground truth depth from the batch
        input_image = batch['Coded'].to(device)
        true_depth = batch['Depth'].to(device)

        # Start inference timer
        start_time = time.time()

        # Forward pass through the model to get predicted depth
        with torch.no_grad():
            predicted_depth = model(input_image)

        # Stop inference timer and calculate inference time
        end_time = time.time()
        inference_times.append(end_time - start_time)

        # Compute metrics
        abs_rel_errors.append(abs_rel(predicted_depth,
true_depth).item())
        rms_errors.append(rmse(predicted_depth, true_depth).item())

        # Convert the predicted depth to numpy, clip, and prepare for
color mapping
        pred_depth_np = to_numpy(predicted_depth[0, 0])  # Convert the
first image in batch

        # Plot with color map and add color bar
        plt.figure(figsize=(6, 6))
        img = plt.imshow(pred_depth_np, cmap=cm.get_cmap(color_map),
vmin=0, vmax=6)
        cbar = plt.colorbar(img, fraction=0.046, pad=0.04)
        cbar.set_label('Depth (meters)')
        plt.axis('off')

        # Save image with color bar as PNG
        plt.savefig(os.path.join(output_dir, f'pred_depth_{i}.png'),
bbox_inches='tight', pad_inches=0)
        plt.close()

    # Calculate average metrics
    avg_abs_rel = np.mean(abs_rel_errors)
    avg_rmse = np.mean(rms_errors)
    avg_inference_time = np.mean(inference_times)
    fps = 1.0 / avg_inference_time if avg_inference_time > 0 else
float('inf')
```

```python
    average_metrics = {
        'Abs-Rel': avg_abs_rel,
        'RMSE': avg_rmse,
        'FPS': fps
    }

    return average_metrics
```

G: Putting everything together:

```python
# Main Execution
def main(config_name, dataset_path, test_dataset_path):
    """
    Main function to run the experiment.

    Parameters:
    - config_name: Name of the experiment configuration to use.
    - dataset_path: Path to the training dataset.
    - test_dataset_path: Path to the test dataset.

    This function loads the datasets, initializes the model, and
evaluates it on the test data.
    """

    # Load the experiment configuration (model, datasets, loss
function, etc.)
    experiment = Experiment(config_name)

    # Define the checkpoint path where the model is saved
    checkpoint_path = f'C:\Adv Computer Vision\P2\model_checkpoints\
model_sect1_{experiment.epochs}'

    # Load training datasets using the configurations defined in the
experiment
    # Each dataset is created using ImageDepthDataset and stored in
train_datasets list
    train_datasets = [
        ImageDepthDataset(
            os.path.join(dataset_path, list(data_d.keys())[0]),  #
Path to the dataset
            codedDir=list(data_d.values())[0][0],  # Directory
containing coded images
            cache=False,  # Whether to cache the dataset in memory
            scale_factor=list(data_d.values())[0][1],  # Scale factor
for depth images
            is_blender=list(data_d.values())[0][2],  # Whether the
data is from Blender (EXR format)
            limit=1000  # Limit the number of images to load
(optional)
        )
```

```
        for data_d in experiment.train_datasets  # Loop through the
training datasets in the experiment config
    ]

    # Create a DataLoader to iterate over the combined training
datasets (shuffled)
    train_loader = DataLoader(ConcatDataset(train_datasets),
batch_size=experiment.batch_size, shuffle=True)

    # Load test datasets using the configurations defined in the
experiment
    test_datasets = [
        ImageDepthDataset(
            os.path.join(test_dataset_path, list(data_d.keys())[0]),
# Path to the dataset
            codedDir=list(data_d.values())[0][0],  # Directory
containing coded images
            cache=False,  # Whether to cache the dataset in memory
            scale_factor=list(data_d.values())[0][1],  # Scale factor
for depth images
            is_blender=list(data_d.values())[0][2]  # Whether the data
is from Blender (EXR format)
        )
        for data_d in experiment.test_datasets  # Loop through the
test datasets in the experiment config
    ]

    # Create a DataLoader to iterate over the combined test datasets
(not shuffled)
    test_loader = DataLoader(ConcatDataset(test_datasets),
batch_size=1, shuffle=False)

    # Initialize the model from the experiment configuration and move
it to the specified device (CPU/GPU)
    model = experiment.model().to(device)

    # Initialize model weights
    init_weights(model)
    parameters =   model.parameters()
    # Define the optimizer (Adam in this case) with the learning rate
from the experiment config
    # Optimizer goes here. Adam optimizer is recommended.
    optimizer = torch.optim.Adam(parameters,lr=0.0001)

    print("In train loader - Batches:", len(train_loader), "Samples:
", len(train_loader) * experiment.batch_size)

    # Uncomment the line below to start training the model (currently
commented for evaluation only)
    # train(model, train_loader, test_loader, optimizer,
```

```
experiment.loss_fn, experiment.epochs, checkpoint_path)

    # Load the pre-trained model from the checkpoint
    model.load_state_dict(torch.load(checkpoint_path,
map_location=device)['model_state_dict'])

    # Loop through each test dataset for evaluation
    for data_d in experiment.test_datasets:
        print(f"Evaluating for dataset: {list(data_d.keys())[0]}")

        # Load the evaluation dataset using the same configuration as
the test datasets
        eval_datasets = ImageDepthDataset(
            os.path.join(test_dataset_path, list(data_d.keys())[0]),
# Path to the dataset
            codedDir=list(data_d.values())[0][0],  # Directory
containing coded images
            cache=False,  # Whether to cache the dataset in memory
            scale_factor=list(data_d.values())[0][1],  # Scale factor
for depth images
            is_blender=list(data_d.values())[0][2]  # Whether the data
is from Blender (EXR format)
        )

        # Create a DataLoader for the evaluation dataset (batch size
16, no shuffle)
        eval_loader = DataLoader(eval_datasets, batch_size=16,
shuffle=False)

        # Define the output directory for saving predicted depth
images
        output_dir =
f'./CodedVO_pred_Sect1_{experiment.epochs}_new_colormap' +
list(data_d.keys())[0]
        os.makedirs(os.path.join(output_dir, "pred_depth"),
exist_ok=True)  # Create output directory if it doesn't exist
        print("outputdir",test_loader)
        # Evaluate the model on the test data and compute metrics
        metrics = evaluate(model, test_loader, device, output_dir)

        # Print evaluation metrics
        print(metrics)

# If running from command-line, the main function is executed with the
given configuration
if __name__ == "__main__":
    main('MetricWeightedLossBlenderNYU', './train/', './UMD-CodedVO-
dataset')
```

```
rgb False 1000
1000
rgb True 1
1000
Codedphasecam-27Linear-New True 1
999

<>:18: SyntaxWarning: invalid escape sequence '\A'
<>:18: SyntaxWarning: invalid escape sequence '\A'
C:\Users\sktha\AppData\Local\Temp\ipykernel_31912\1093368701.py:18:
SyntaxWarning: invalid escape sequence '\A'
  checkpoint_path = f'C:\Adv Computer Vision\P2\model_checkpoints\
model_sect1_{experiment.epochs}'

In train loader - Batches: 2000 Samples:  2000

C:\Users\sktha\AppData\Local\Temp\ipykernel_31912\1093368701.py:68:
FutureWarning: You are using `torch.load` with `weights_only=False`
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(checkpoint_path,
map_location=device)['model_state_dict'])

Evaluating for dataset: DiningRoom
Codedphasecam-27Linear-New True 1
999
outputdir <torch.utils.data.dataloader.DataLoader object at
0x000002453535D010>

C:\Users\sktha\AppData\Local\Temp\ipykernel_31912\1394826309.py:58:
MatplotlibDeprecationWarning: The get_cmap function was deprecated in
Matplotlib 3.7 and will be removed in 3.11. Use
``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap()``
or ``pyplot.get_cmap()`` instead.
  img = plt.imshow(pred_depth_np, cmap=cm.get_cmap(color_map), vmin=0,
vmax=6)

{'Abs-Rel': np.float64(0.45316312734309855), 'RMSE':
np.float64(1.1641531132004999), 'FPS': np.float64(348.4987330448161)}
```

Submission Guidelines:

1. Compiled version of the colab file with results (Feel free to use two colab files if you wish, one for AiF training/testing and other one for coded and comparison)
2. PDF Report (Combined with Part 1)
3. Output predicted depth maps on testing dataset (ICL and DiningRoom) of at least 5 images each must be present in your report.
4. Compare extensively on both AiF and Coded Images Training/Test both qualitatively and quantitatively. Make sure they are in the metric units and not normalized.

Note: You should see reasonable results after 15 epochs but almost as good results as in the CodedVO paper by 40 epochs.

The following are the out of domain testing on an image from UMDCodedVO-DiningRoom dataset at different epochs:

GT Depth Image:

Depth Prediction after 5 epochs:

Depth Prediction after 15 epochs:

Depth Prediction after 30 epochs:

Depth Prediction after 45 epochs: