National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Ayush Kumar Shrivas
Student ID:x23331666

School of Computing
National College of Ireland

Supervisor: Yalemisew Abgaz

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ayush Kumar Shrivas
Student ID:	x23331666
Programme:	Data Analytics
Year:	2024-2025
Module:	MSc Research Project
Supervisor:	Yalemisew Abgaz
Submission Due Date:	11/08/2025
Project Title:	Configuration Manual
Word Count:	1033
Page Count:	12

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Ayush Kumar Shrivas
Date:	11th August 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Ayush Kumar Shrivas
x23331666

1 Introduction

The Configuration manual serves the purpose of documenting the technical details of the research project. It includes information about the tools, code outline and the software environment used for this study. It also highlights the specific requirements and fine-tuning in the code through snippets of the implemented code. This manual can be used for reproducing similar work in the future to extend the research scope. Section 2 discusses the hardware specifications and software environment used for executing the project. Section 3 includes information about the data collection and layout. Section 4 contains details about the installation's dependencies and library imports for the code base. Section 5 discusses the data loading process and Section 6 discusses the Data cleaning and processing performed for the dataset. Section 7 shows the code snippets of the 5 implemented models in the study. Section 8 mentions the training performed for the dataset and Section 9 discusses the testing and evaluation done in the research.

2 Environment

Figure 1 shows the hardware details of the system used in the study of this project. The system is a Lenovo IdeaPad S540-14IML, with an Intel i5-10210U (1.6–2.11 GHz) processor having 8 GB RAM and 477 GB storage. Graphics memory is 2 GB (integrated) as shown.

To aid the system computation and increase the speed of execution, use of Google Colab [1] was done as an integrated development environment.

3 Data Collection

Figure 2 shows the dataset location and structure. The cityscapes_data/ contains two folders: train/ (2975 files) and val/ (500 files). Each file is a side-by-side pair: left = camera image, right = segmentation map.

4 Library and Package installation

Figure 3 shows all the python packages installed in Colab. These cover PyTorch, data processing/visualization, and ready-made segmentation architectures. Figure 4 shows the

[1] Google Colab .

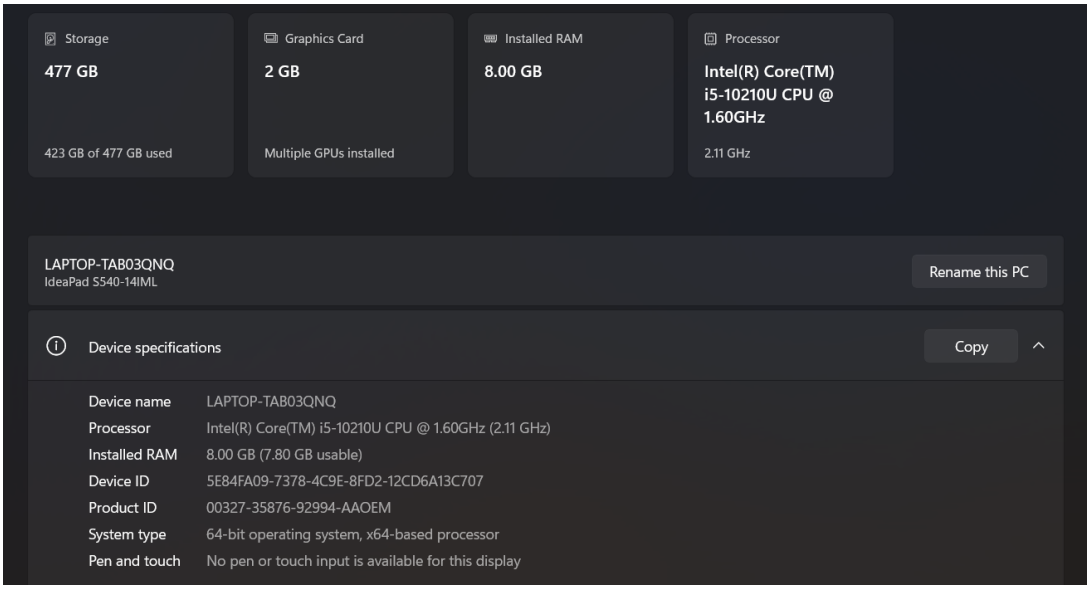


Figure 1: System Configuration

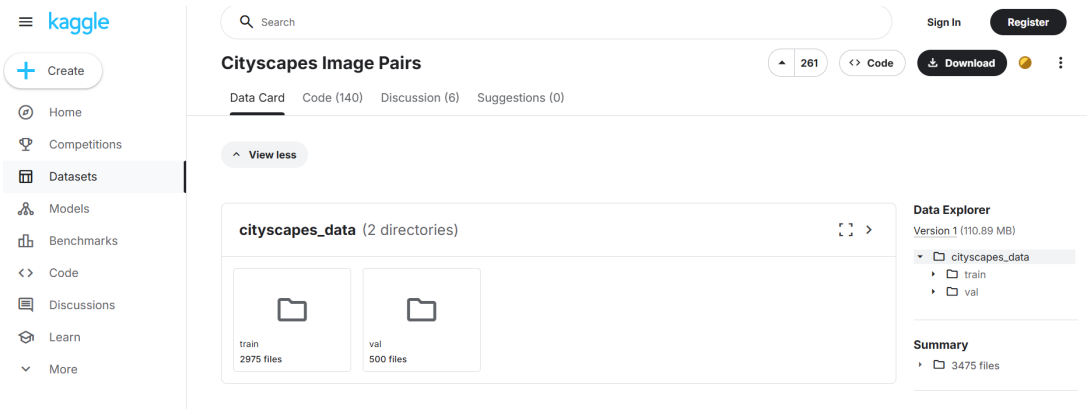


Figure 2: Dataset Location

code imports standard libs (os, glob, zipfile, cv2, numpy), PyTorch core/APIs, tqdm, sklearn metrics, matplotlib, and segmentation_models_pytorch as smp. It sets the compute device: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') This makes the rest of the code run on GPU if available, else CPU.

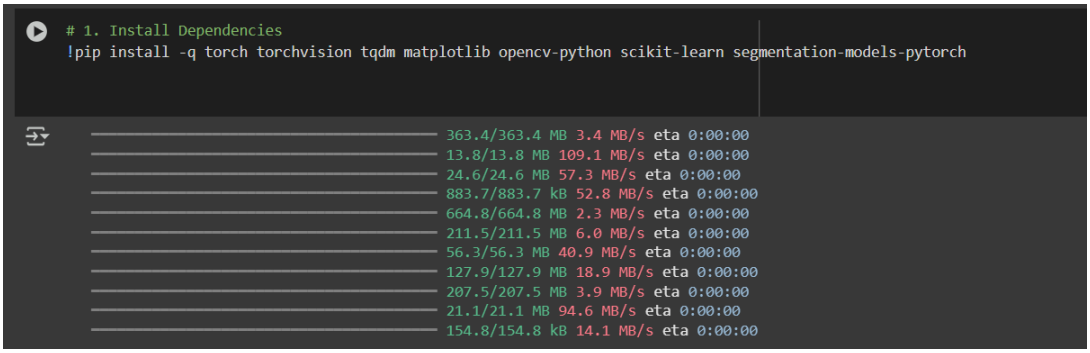


Figure 3: Installed Packages

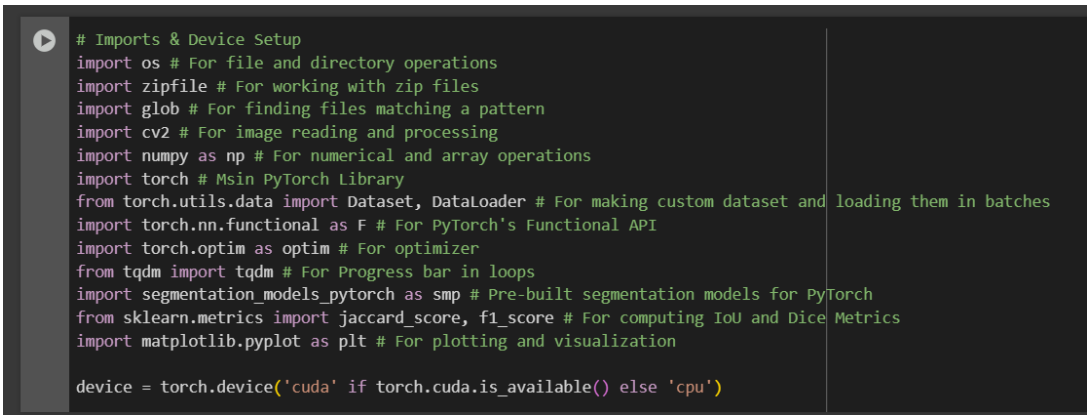


Figure 4: Imported Libraries

5 Data Loading

Figure 5 shows the zipped dataset uploaded via google.colab.files.upload(). It extracts the dataset to /content/data. This makes the dataset accessible in the Colab filesystem.

6 Data Cleaning and Processing

Custom data generator classes were formed to load and prepare the data for each model. Figure 6 shows the Dataset classes (PyTorch) wherein constant image sizes (full: 512×256; model input: 256×256) are loaded. SegmentationDataset(Dataset) is declared for train/val to store file paths and target resize. Also, __len__ is implemented to report dataset size.

Figure 7 shows Dataset classes (PyTorch) wherein SegmentationDataset.__getitem__ helps read an image path with OpenCV. The pair is split into left (camera) and right

```python
# Upload & Unzip Dataset
from google.colab import files
uploaded = files.upload() # Open file uploaf dialog and wait for uplaod
zip_path = next(iter(uploaded)) # Get thr uploaded file name

# Unzip Dataset
extract_root = '/content/data' # Set directory where to extract files
os.makedirs(extract_root, exist_ok=True)
with zipfile.ZipFile(zip_path, 'r') as zf: # Open the Zip file
    zf.extractall(extract_root) # Extract all contents to ectract_root folder

data_root = None
for root_dir, dirs, _ in os.walk(extract_root): # Walking through all subfolders in extract_root
    if 'cityscapes_data' in dirs:
        data_root = os.path.join(root_dir, 'cityscapes_data')
        break
if data_root is None:
    raise ValueError(
        f"Could not find 'cityscapes_data' under {extract_root}. Available: {os.listdir(extract_root)}"
    )

print("Folders present in the dataset:", os.listdir(data_root)) # List the folder present in the dataset.
```

Figure 5: Loading Dataset

```python
# Dataset Class Definitions
import torch # Import PyTorch Library
from torch.utils.data import Dataset, DataLoader # Import Dataset and DataLoader classes
import cv2 # Import OpenCV for image reading and processing
import numpy as np # Import numpy for numerical operations
import glob # Import glob for finding files with patterns

# Define full image size and resized shape
IMG_WIDTH_FULL = 512
IMG_HEIGHT_FULL = 256
IMG_SIZE = (256, 256)

# Dataset class for training/validation
class SegmentationDataset(Dataset):
    def __init__(self, image_paths, img_size=IMG_SIZE): # Initialization: save image paths and desired image size
        self.paths = image_paths # Store list of image file paths
        self.img_size = img_size # Store the resize dimensions

    def __len__(self): return len(self.paths) # Return total number of images in dataset
```

Figure 6: Dataset classes(a)

(segmentation) halves. It normalizes to [0,1], resizes both to the model input size, converts to torch.float32 tensors with shape [C,H,W] and returns (camera_tensor, mask_tensor). TestDataset is handled in a similar manner, but returns (camera_tensor, file_path) for inference/visualization.

```
    def __getitem__(self, idx):
        path = self.paths[idx] # Get image path by index
        img = cv2.imread(path)  # Read the image using OpenCV
        img = cv2.resize(img, (IMG_WIDTH_FULL, IMG_HEIGHT_FULL))  # Resize to original shape
        cam = img[:, :IMG_WIDTH_FULL//2] / 255.0  # Left half - camera view
        seg = img[:, IMG_WIDTH_FULL//2:] / 255.0  # Right half - segmentation map
        cam = cv2.resize(cam, self.img_size)  # Resize halves to model input size
        seg = cv2.resize(seg, self.img_size)
        cam_t = torch.tensor(cam, dtype=torch.float32).permute(2,0,1) # Convet camera view to tensor and change shape to [C, H, W]
        seg_t = torch.tensor(seg, dtype=torch.float32).permute(2,0,1) # Convet segmentation map to tensor and change shape to [C, H, W]
        return cam_t, seg_t # Return camera image, segmentation mask as tensors

# Dataset for testing
class TestDataset(Dataset):
    def __init__(self, image_paths, img_size=IMG_SIZE): # Initialization save image paths and size
        self.paths = image_paths # Store list of image file paths
        self.img_size = img_size # Store the resize dimensions
    def __len__(self): return len(self.paths) # Return total number of images in dataset
    def __getitem__(self, idx):
        path = self.paths[idx] # Get image path by index
        img = cv2.imread(path) # Read the image using OpenCV
        img = cv2.resize(img, (IMG_WIDTH_FULL, IMG_HEIGHT_FULL)) # Resize to Original shape
        cam = img[:, :IMG_WIDTH_FULL//2] / 255.0 # Take left half, normalize to [0, 1]
        cam = cv2.resize(cam, self.img_size) # Resize camera image to model input size
        cam_t = torch.tensor(cam, dtype=torch.float32).permute(2,0,1) # Convert to tensor and change shape to [C, H, W]
        return cam_t, path # Return Camera Image tensor, file path
```

Figure 7: Dataset classes(b)

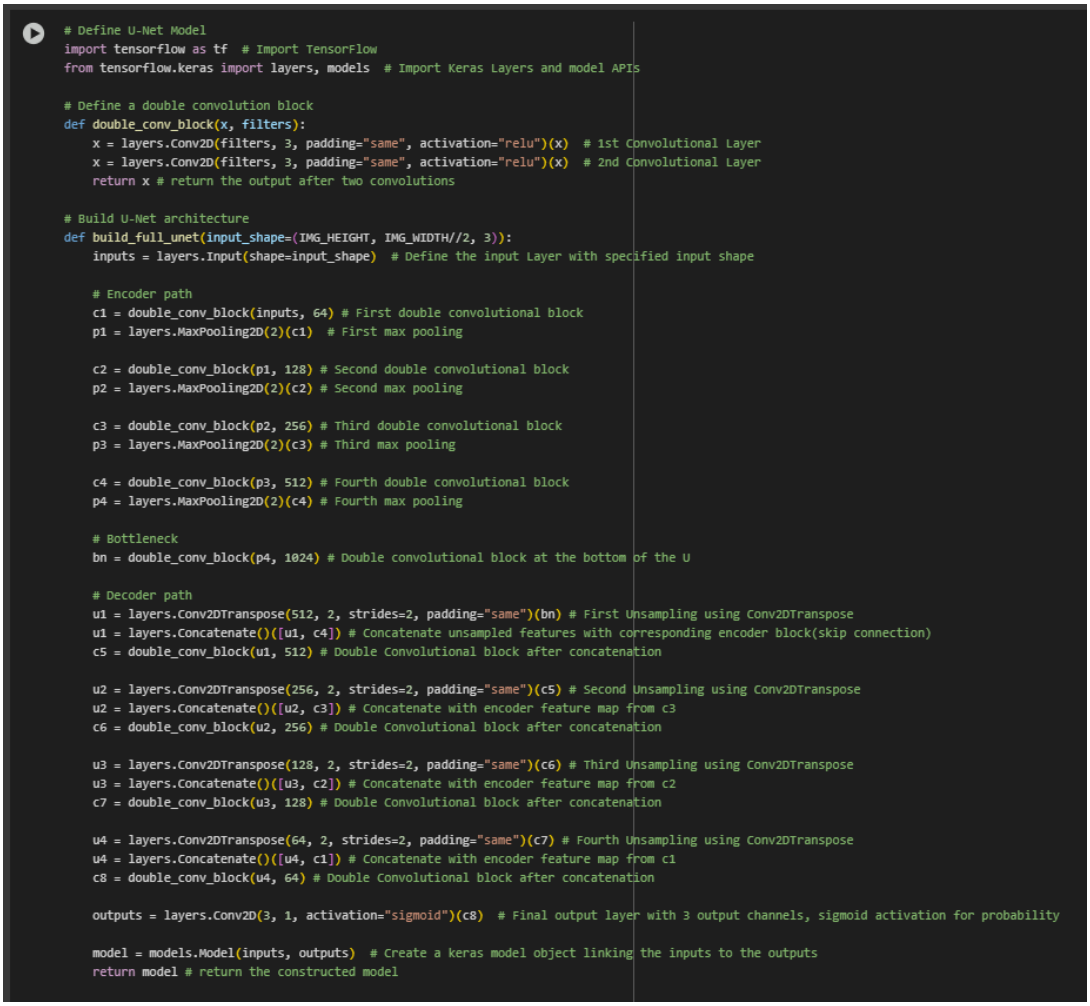
Figure 8 shows collected file paths built DataLoaders (PyTorch). Glob is used to gather *.png/*.jpg/*.jpeg under train/, val/, and testing/ (if present). DataLoaders with batch_size=8, shuffle=True for train, and num_workers=2 are created with the purpose of efficient mini-batch loading for training/validation/testing.

```
# Load image paths and prepare DataLoaders
glob_ext = ('*.png', '*.jpg', '*.jpeg') # List of file extensions to look for (PNG, JPG, JPEG images)
train_paths, val_paths, test_paths = [], [], [] # Create empty lists to hold paths for train, val, and test images

# Collect all image paths
for ext in glob_ext:
    train_paths += glob.glob(os.path.join(data_dir, "train", ext)) # Finding all training images with this extension and add to list
    val_paths   += glob.glob(os.path.join(data_dir, "val", ext)) # Find all validation images with this extension and add to list
    test_paths  += glob.glob(os.path.join(data_dir, "testing", ext)) # Find all test images with this extension and add to list

print(f"Found {len(train_paths)} train, {len(val_paths)} val, {len(test_paths)} test images.") # Print number of images found in each set

# Raise error if no training images
if not train_paths:
    raise FileNotFoundError("No training images found.")

# Create DataLoaders
torch_kwargs = {'batch_size': 8, 'shuffle': True, 'num_workers': 2} # Set Common Dataloader options (batch size, shuffle, workers)
train_loader = DataLoader(SegmentationDataset(train_paths), **torch_kwargs) # Create DataLoader for training data
val_loader   = DataLoader(SegmentationDataset(val_paths), batch_size=8, shuffle=False, num_workers=2) # Create DataLoader for validation data
test_loader  = DataLoader(TestDataset(test_paths), batch_size=8, shuffle=False, num_workers=2) # Create Dataloader for test data
```

Figure 8: Build DataLoaders

7 Implementation of Models

1. U-Net model definition (TensorFlow/Keras) Figure 9 defines a double_conv_block and build_full_unet() in Keras. Classic encoder–decoder with skips; output layer has 3 channels with sigmoid.



```python
# Define U-Net Model
import tensorflow as tf  # Import TensorFlow
from tensorflow.keras import layers, models  # Import Keras Layers and model APIs

# Define a double convolution block
def double_conv_block(x, filters):
    x = layers.Conv2D(filters, 3, padding="same", activation="relu")(x)  # 1st Convolutional Layer
    x = layers.Conv2D(filters, 3, padding="same", activation="relu")(x)  # 2nd Convolutional Layer
    return x # return the output after two convolutions

# Build U-Net architecture
def build_full_unet(input_shape=(IMG_HEIGHT, IMG_WIDTH//2, 3)):
    inputs = layers.Input(shape=input_shape)  # Define the input Layer with specified input shape

    # Encoder path
    c1 = double_conv_block(inputs, 64) # First double convolutional block
    p1 = layers.MaxPooling2D(2)(c1)  # First max pooling

    c2 = double_conv_block(p1, 128) # Second double convolutional block
    p2 = layers.MaxPooling2D(2)(c2) # Second max pooling

    c3 = double_conv_block(p2, 256) # Third double convolutional block
    p3 = layers.MaxPooling2D(2)(c3) # Third max pooling

    c4 = double_conv_block(p3, 512) # Fourth double convolutional block
    p4 = layers.MaxPooling2D(2)(c4) # Fourth max pooling

    # Bottleneck
    bn = double_conv_block(p4, 1024) # Double convolutional block at the bottom of the U

    # Decoder path
    u1 = layers.Conv2DTranspose(512, 2, strides=2, padding="same")(bn) # First Unsampling using Conv2DTranspose
    u1 = layers.Concatenate()([u1, c4]) # Concatenate unsampled features with corresponding encoder block(skip connection)
    c5 = double_conv_block(u1, 512) # Double Convolutional block after concatenation

    u2 = layers.Conv2DTranspose(256, 2, strides=2, padding="same")(c5) # Second Unsampling using Conv2DTranspose
    u2 = layers.Concatenate()([u2, c3]) # Concatenate with encoder feature map from c3
    c6 = double_conv_block(u2, 256) # Double Convolutional block after concatenation

    u3 = layers.Conv2DTranspose(128, 2, strides=2, padding="same")(c6) # Third Unsampling using Conv2DTranspose
    u3 = layers.Concatenate()([u3, c2]) # Concatenate with encoder feature map from c2
    c7 = double_conv_block(u3, 128) # Double Convolutional block after concatenation

    u4 = layers.Conv2DTranspose(64, 2, strides=2, padding="same")(c7) # Fourth Unsampling using Conv2DTranspose
    u4 = layers.Concatenate()([u4, c1]) # Concatenate with encoder feature map from c1
    c8 = double_conv_block(u4, 64) # Double Convolutional block after concatenation

    outputs = layers.Conv2D(3, 1, activation="sigmoid")(c8)  # Final output layer with 3 output channels, sigmoid activation for probability

    model = models.Model(inputs, outputs)  # Create a keras model object linking the inputs to the outputs
    return model # return the constructed model
```

Figure 9: UNet Model

2. Metrics + SegNet model (TensorFlow/Keras) Figure 10 shows implemented custom metrics like Dice coefficient, IoU, and pixel accuracy (Keras/TensorFlow). It defines a SegNet-style encoder–decoder and compiles it with Adam and binary_crossentropy, tracking the above metrics and calls model.summary() to print the architecture.

```python
# Build SegNet Model and evaluation metrics
import tensorflow as tf # Import TensorFlow
from tensorflow.keras import layers, models # Import Keras Layers/Models modules

# Define Dice coefficient metric for segmentation accuracy
def dice_coef(y_true, y_pred, smooth=1e-6):
    y_true_f = tf.keras.backend.flatten(y_true) # Flatten ground truth and prediction arrays to 1D
    y_pred_f = tf.keras.backend.flatten(y_pred)
    intersection = tf.reduce_sum(y_true_f * y_pred_f) # Calculate intersection between true and predicted masks
    return (2. * intersection + smooth) / (tf.reduce_sum(y_true_f) + tf.reduce_sum(y_pred_f) + smooth) # Return Dice Score using intersection and sums

# Define Intersection over union (IoU) Metric
def iou_coef(y_true, y_pred, smooth=1e-6):
    y_true_f = tf.keras.backend.flatten(y_true) # Flatten ground truth and prediction arrays to 1D
    y_pred_f = tf.keras.backend.flatten(y_pred)
    intersection = tf.reduce_sum(y_true_f * y_pred_f) # Calculate intersection between true and predicted masks
    union = tf.reduce_sum(y_true_f) + tf.reduce_sum(y_pred_f) - intersection # Calculate union between true and predicted masks
    return (intersection + smooth) / (union + smooth) # Return IoU Score

# Define Pixel Accuracy metrics
def pixel_accuracy(y_true, y_pred):
    # Convert prediction to binary using a threshold of 0.5
    y_pred_bin = tf.cast(y_pred > 0.5, tf.float32)
    y_true_bin = tf.cast(y_true > 0.5, tf.float32)
    return tf.reduce_mean(tf.cast(tf.equal(y_true_bin, y_pred_bin), tf.float32)) # Return Comparison between predicted and True pixels and take mean

# Define Function to build a SegNet Model
def build_segnet(input_shape=(IMG_HEIGHT, IMG_WIDTH//2, 3)):
    inp = layers.Input(shape=input_shape) # Input layer for the image
    # Encoder
    x = layers.Conv2D(64,3,padding='same',activation='relu')(inp) # First Convolutional Layer
    x = layers.Conv2D(64,3,padding='same',activation='relu')(x) # Second Convolutional Layer
    x = layers.MaxPooling2D()(x) # Downsample (Max Pooling)
    x = layers.Conv2D(128,3,padding='same',activation='relu')(x) # Next Convolutional Layer
    x = layers.Conv2D(128,3,padding='same',activation='relu')(x) # Next Convolutional Layer
    x = layers.MaxPooling2D()(x)# Downsample (Max Pooling)
    x = layers.Conv2D(256,3,padding='same',activation='relu')(x) # Next Convolutional Layer
    x = layers.Conv2D(256,3,padding='same',activation='relu')(x) # Next Convolutional Layer
    x = layers.MaxPooling2D()(x) # Downsample (Max Pooling)
    # Decoder
    x = layers.UpSampling2D()(x) # Unsample to increase spatial resolution
    x = layers.Conv2D(256,3,padding='same',activation='relu')(x) # Convolutional Layer
    x = layers.Conv2D(256,3,padding='same',activation='relu')(x) # Convolutional Layer
    x = layers.UpSampling2D()(x) # Unsample
    x = layers.Conv2D(128,3,padding='same',activation='relu')(x) # Convolutional Layer
    x = layers.UpSampling2D()(x) # Unsample
    x = layers.Conv2D(64,3,padding='same',activation='relu')(x) # Convolutional Layer
    out = layers.Conv2D(3,1,activation='sigmoid')(x) # Output Layer: 3 channels, sigmoid activation
    return models.Model(inp, out) # Build and Return the Model

# Build the SegNet Model
model = build_segnet()

# Compile the model with Adam Optimizer, binary crossentropy loss, and Evaluation Metrics
model.compile(
    optimizer='adam', # Use Adam optimizer
    loss='binary_crossentropy', # Use Binary crossentropy loss
    metrics=[
        pixel_accuracy, # Track pixel accuracy
        iou_coef, # Track Intersection over Union
        dice_coef # Track Dice Coefficient
    ]
)
model.summary() # Print summary of the model architecture
```

Figure 10: SegNet Model

3. PSPNet Model Definition (TensorFlow/Keras) Figure 11 shows the implemented PSPNet (Pyramid Scene Parsing Network) for semantic segmentation where ResizeLayer class resizes pooled feature maps back to target dimensions and pyramid_pooling_block() performs pooling at multiple bin sizes (e.g., 1×1, 2×2, 3×3, 6×6) to capture multi-scale context, then upsamples and concatenates them. Also, build_pspnet() builds the encoder, applies pyramid pooling, then a decoder with upsampling and convolution layers.

4. Load Pre-trained SegFormer from Hugging Face Figure 12 demonstrates the in-

```python
# PSPNet Model Definition
import tensorflow as tf  # import TensorFlow
from tensorflow.keras import layers, models  # import Keras layers and models

class ResizeLayer(tf.keras.layers.Layer):
    def __init__(self, target_height, target_width):
        super().__init__()  # call base constructor
        self.target_height = target_height  # store target height
        self.target_width  = target_width   # store target width

    def call(self, inputs):
        return tf.image.resize(inputs,
                               [self.target_height, self.target_width],
                               method='bilinear') # resize inputs to target dimensions

def pyramid_pooling_block(x, bin_sizes):  # define PSPNet's pyramid pooling
    concat_list = [x]  # list to gather pooled features
    h, w = x.shape[1], x.shape[2]  # get spatial dimensions
    for bs in bin_sizes:  # for each pyramid bin size
        pool = layers.AveragePooling2D(
            pool_size=(h//bs, w//bs),  # define pool size
            strides=(h//bs, w//bs),     # define strides
            padding='same')(x)          # apply average pooling
        pool = layers.Conv2D(64, 1, padding='same', activation='relu')(pool)  # 1x1 conv
        pool = ResizeLayer(h, w)(pool)  # upsample back to original size
        concat_list.append(pool)  # append pooled feature
    return layers.Concatenate()(concat_list)  # concatenate all features

def build_pspnet(input_shape=(256, 256, 3)):  # build PSPNet model
    inputs = layers.Input(shape=input_shape)  # define input layer
    # Encoder
    x = layers.Conv2D(64, 3, padding='same', activation='relu')(inputs)  # convolutional Layer
    x = layers.MaxPooling2D((2,2))(x)  # downsample by 2
    x = layers.Conv2D(128, 3, padding='same', activation='relu')(x)  # convolutional Layer
    x = layers.MaxPooling2D((2,2))(x)  # downsample
    x = layers.Conv2D(256, 3, padding='same', activation='relu')(x)  # convolutional Layer
    # Pyramid Pooling
    ppm = pyramid_pooling_block(x, bin_sizes=[1,2,3,6])  # apply pooling block
    x = layers.Conv2D(256, 3, padding='same', activation='relu')(ppm)  # convolutional Layer
    # Decoder
    x = layers.UpSampling2D((2,2), interpolation='bilinear')(x)  # upsample
    x = layers.Conv2D(128, 3, padding='same', activation='relu')(x)  # convolutional Layer
    x = layers.UpSampling2D((2,2), interpolation='bilinear')(x)  # upsample back
    x = layers.Conv2D(64, 3, padding='same', activation='relu')(x)  # convolutional Layer
    outputs = layers.Conv2D(3, 1, activation='sigmoid')(x)  # final convolutional Layer for segmentation mask
    return models.Model(inputs, outputs)  # return model instance

model = build_pspnet()  # instantiate model

# Evaluation Metrics
# Dice coefficient
def dice_coef(y_true, y_pred, smooth=1e-6):
    y_true_f = tf.reshape(y_true, [-1]) # Flatten both ground truth and predicted masks to 1D arrays
    y_pred_f = tf.reshape(y_pred, [-1])
    y_pred_bin = tf.cast(y_pred_f > 0.5, tf.float32) # Convert predicted mask to binary using a threshold of 0.5
    intersection = tf.reduce_sum(y_true_f * y_pred_bin) # Calculate intersection between true and predicted binary masks
    return (2. * intersection + smooth) / (tf.reduce_sum(y_true_f) + tf.reduce_sum(y_pred_bin) + smooth) # Calculate Dice Coefficient Score

# Pixel accuracy
def pixel_acc(y_true, y_pred):
    y_true_bin = tf.cast(y_true > 0.5, tf.float32) # Flatten both ground truth and predicted masks to 1D arrays
    y_pred_bin = tf.cast(y_pred > 0.5, tf.float32)
    correct = tf.cast(tf.equal(y_true_bin, y_pred_bin), tf.float32) # Check where prediction matches ground truht (pixel-wise)
    return tf.reduce_mean(correct) # Return mean accuracy across all pixels

# Mean IoU (Intersection over Union)
def mean_iou_custom(y_true, y_pred, smooth=1e-6):
    y_true_bin = tf.cast(y_true > 0.5, tf.float32) # Convert true mask to binary
    y_pred_bin = tf.cast(y_pred > 0.5, tf.float32) # Convert predicted mask to binary
    ious = [] # List to store IoU for each channel/class
    for i in range(3):  # for each channel
        yt = tf.reshape(y_true_bin[..., i], [-1]) # Flatten each channel to 1D
        yp = tf.reshape(y_pred_bin[..., i], [-1])
        intersection = tf.reduce_sum(yt * yp) # Compute intersection for this channel
        union = tf.reduce_sum(yt) + tf.reduce_sum(yp) - intersection # Compute union for this channel
        ious.append((intersection + smooth) / (union + smooth)) # Calculate IoU and add to list
    return tf.add_n(ious) / 3.0 # Return the mean IoU across all channels

# Compile the model with Adam optimizer, Binary crossentropy, and Evaluation Metrics
model.compile(
    optimizer='adam', # Use Adam optimizer
    loss='binary_crossentropy', # Use Binary Crossentropy Loss Function
    metrics=[pixel_acc, mean_iou_custom, dice_coef] # Evaluation Metrics
)

model.summary() # Display summary of the model architecture
```

Figure 11: PSPNet Model

stalled transformers library. SegformerForSemanticSegmentation and SegformerImageProcessor are imported and nvidia/segformer-b0-finetuned-ade-512-512 pretrained model is loaded. Image processor (no rescaling, resizing, or normalization here) are configured and model instance with num_labels=3 for segmentation output are created.

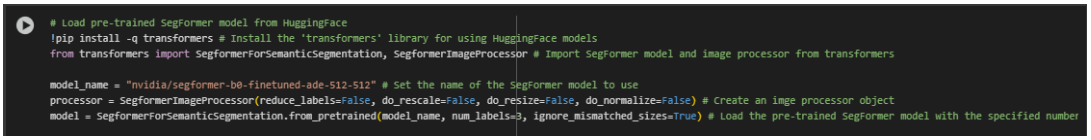


Figure 12: SEGFormer Model

5. Model Initialization (PyTorch SMP) Figure 13 shows Segmentation Models PyTorch (smp) library to create a U-Net model with encoder backbone as mit_b2 (Mix Transformer from SegFormer family) with Pre-trained ImageNet weights for the encoder as it Input channels - 3 (RGB images) and Output classes - 3 segmentation classes defining optimizer as AdamW with learning rate 1e-4.

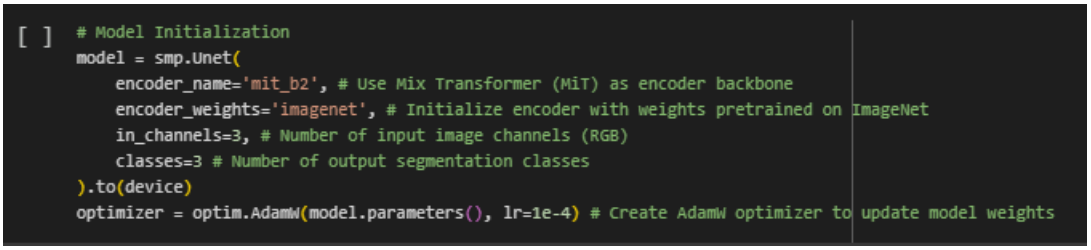


Figure 13: UNETr Model

8 Training and Evaluation Functions

1. Training and Evaluation Functions (PyTorch): Figure 14 shows the code maintains a metrics_history dictionary to store accuracy, IoU, and Dice for both train/val. The function train_one_epoch() loops over batches in training DataLoader, moves images/masks to device, predicts masks, computes binary cross-entropy loss and thresholds predictions at 0.5 to get binary masks. It also calculates Pixel Accuracy, IoU, and Dice for the epoch and stores results in metrics_history. The function evaluate() is similar to training loop but without gradient updates and runs on validation set to compute the same metrics.

2. Training Loop Execution: Figure 15 shows the training loop with 20 epochs set. The code loops over each epoch, running train_one_epoch() then evaluate() for validation metrics, then prints epoch number and metrics to track performance over time.

```python
# Metrics History
metrics_history = {
    'train_pixel_acc': [],
    'val_pixel_acc': [],
    'train_iou': [],
    'val_iou': [],
    'train_dice': [],
    'val_dice': [],
}
# Function to train the model for one epoch
def train_one_epoch():
    model.train()   # Set the model to training mode
    total_loss = 0.0   # Initialize total loss for this epoch
    correct_pixels, total_pixels = 0, 0   # Counters for pixel accuracy calculation
    ious, dices = [], []   # Lists to store IoU and Dice scores

    for imgs, masks in tqdm(train_loader, desc='Training'):   # Loop over each batch with progress bar
        imgs = imgs.to(device)   # Move images to the selected device (GPU or CPU)
        masks = masks.to(device) # Move masks to the selected device
        preds = model(imgs)      # Forward pass: get model predictions for this batch
        preds = F.interpolate(preds, size=masks.shape[2:], mode='bilinear', align_corners=False)  # Resize predictions to match masks
        loss = F.binary_cross_entropy_with_logits(preds, masks)   # Compute binary cross-entropy loss with logits
        optimizer.zero_grad()   # Clear previous gradients before backward pass
        loss.backward()         # Backpropagation: compute gradients
        optimizer.step()        # Update model weights using optimizer
        total_loss += loss.item()   # Add batch loss to running total

        bin_preds = (torch.sigmoid(preds) > 0.5)   # Apply sigmoid and threshold to get binary predictions
        correct_pixels += (bin_preds == (masks > 0.5)).sum().item()   # Update correct pixel count
        total_pixels += masks.numel()  # Update total number of pixels considered

        # Calculate metrics for each sample in batch
        for p, t in zip(bin_preds.cpu(), masks.cpu()):
            p_flat = p.flatten().numpy().astype(int)   # Flatten prediction and convert to numpy int
            t_flat = t.flatten().numpy().astype(int)   # Flatten mask and convert to numpy int
            ious.append(jaccard_score(t_flat, p_flat, average='micro'))   # Compute IoU
            dices.append(f1_score(t_flat, p_flat, average='micro'))       # Compute Dice score

    avg_loss = total_loss / len(train_loader)   # Compute average loss per batch
    pixel_acc = correct_pixels / total_pixels   # Compute pixel accuracy for entire epoch
    mean_iou = np.mean(ious)                    # Compute mean IoU across all samples
    mean_dice = np.mean(dices)                  # Compute mean Dice score across all samples

    print(f"Training Loss: {avg_loss:.4f}, Pixel Accuracy: {pixel_acc:.4f}, Mean IoU: {mean_iou:.4f}, Dice Score: {mean_dice:.4f}")
    # Print metrics summary for this epoch

    # Save train metrics
    metrics_history['train_pixel_acc'].append(pixel_acc)
    metrics_history['train_iou'].append(mean_iou)
    metrics_history['train_dice'].append(mean_dice)
    return pixel_acc, mean_iou, mean_dice


def evaluate():
    model.eval()   # Set the model to evaluation mode (disable dropout, batchnorm updates, etc.)
    correct_pixels, total_pixels = 0, 0   # Counters for pixel accuracy
    ious, dices = [], []   # Lists to accumulate IoU and Dice for all validation samples

    with torch.no_grad():
        for imgs, masks in tqdm(val_loader, desc='Validation'):  # Loop over validation batches with progress bar
            imgs = imgs.to(device)    # Move images to device
            masks = masks.to(device)  # Move masks to device
            preds = model(imgs)       # Forward pass: get model predictions
            preds = F.interpolate(preds, size=masks.shape[2:], mode='bilinear', align_corners=False)  # Resize predictions
            bin_preds = (torch.sigmoid(preds) > 0.5)   # Binarize predictions

            correct_pixels += (bin_preds == (masks > 0.5)).sum().item()   # Update correct pixel count
            total_pixels += masks.numel()                                 # Update total pixel count

            for p, t in zip(bin_preds.cpu(), masks.cpu()):
                p_flat = p.flatten().numpy().astype(int)   # Flatten prediction and convert to numpy int
                t_flat = t.flatten().numpy().astype(int)   # Flatten mask and convert to numpy int
                ious.append(jaccard_score(t_flat, p_flat, average='micro'))   # Compute IoU
                dices.append(f1_score(t_flat, p_flat, average='micro'))       # Compute Dice score

    pixel_acc = correct_pixels / total_pixels  # Compute overall pixel accuracy
    mean_iou = np.mean(ious)                   # Compute mean IoU across all validation samples
    mean_dice = np.mean(dices)                 # Compute mean Dice score across all validation samples

    print(f"Pixel Accuracy: {pixel_acc:.4f}, Mean IoU: {mean_iou:.4f}, Dice Score: {mean_dice:.4f}")
    # Print metrics summary for validation

    # Save validation metrics
    metrics_history['val_pixel_acc'].append(pixel_acc)
    metrics_history['val_iou'].append(mean_iou)
    metrics_history['val_dice'].append(mean_dice)
    return pixel_acc, mean_iou, mean_dice
```

Figure 14: Training and Evaluation Functions

```python
# Run Training Loop
epochs = 20 # Set the total number of epochs
for epoch in range(1, epochs+1): # Loop through each epoch
    print(f"\n=== Epoch {epoch}/{epochs} ===") # Print the current epoch number
    train_pixel_acc, train_iou, train_dice = train_one_epoch() # Train the model for one epoch
    val_pixel_acc, val_iou, val_dice = evaluate() # Evaluate the model on the validation data after each epoch
```

Figure 15: Training Loop Execution

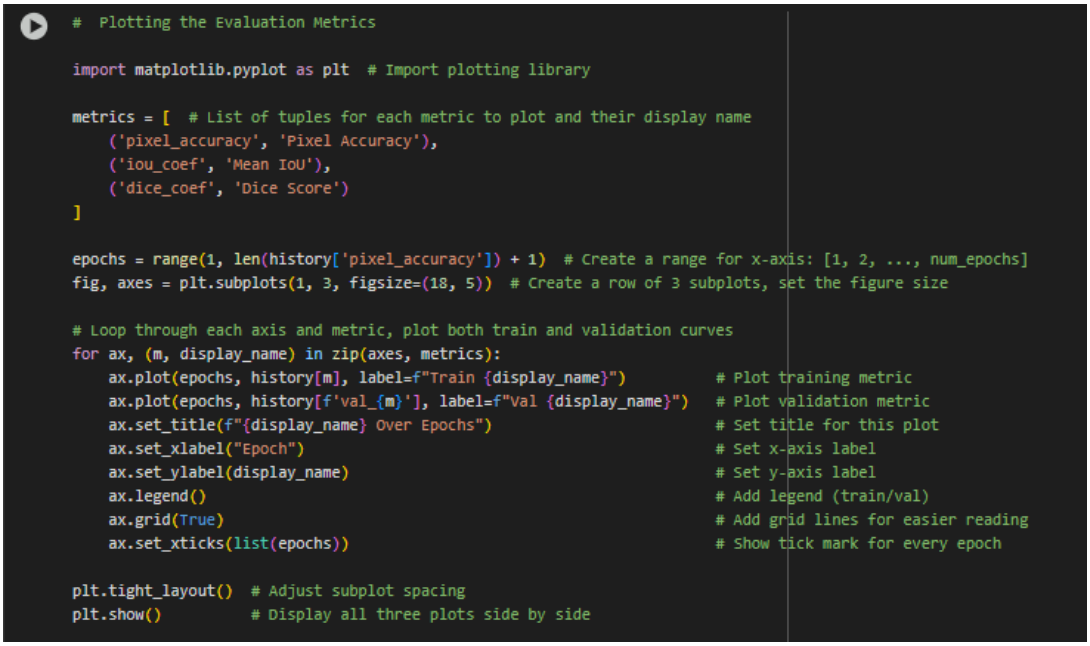
9 Testing and Evaluation

1. Testing Visualization: Figure 16 shows the code that puts model in eval() mode, iterates over the test DataLoader, generates predictions, resizes them to match original size, applies sigmoid + threshold, displays side-by-side plots of - Left: Original camera image, Right: Predicted segmentation mask and then stops after showing 5 examples.

```python
# Testing & Visualization
model.eval() # Set the model to evaluation mode
shown = 0 # Counter for displayed images

with torch.no_grad():
    for cam in test_loader:
        cam = cam.to(device)
        preds = model(cam)  # Get model prediction
        preds = F.interpolate(preds, size=cam.shape[2:], mode='bilinear', align_corners=False)  # Resize output
        pr_mask = torch.sigmoid(preds)[0].cpu().numpy().transpose(1,2,0)  # Postprocess mask
        orig_cam = cam[0].cpu().permute(1,2,0).numpy()  # Original camera image for display

        # Show side-by-side input and predicted mask
        fig, axs = plt.subplots(1,2,figsize=(8,4))
        axs[0].imshow(orig_cam); axs[0].set_title('Camera'); axs[0].axis('off')
        axs[1].imshow(pr_mask); axs[1].set_title('Prediction'); axs[1].axis('off')
        plt.tight_layout(); plt.show()

        shown += 1               # Increment the shown counter
        if shown >= 5: break     # Stop after 5 images
```

Figure 16: Testing & Visualization

2. Plotting Evaluation Metrics: Figure 17 shows the code where it uses Matplotlib to plot Pixel Accuracy, Mean IoU, and Dice Score for both train and validation sets. It creates subplots for each metric with legends, grid lines, and epoch ticks. It helps visually assess training progress and overfitting/underfitting of the results.



```python
#  Plotting the Evaluation Metrics

import matplotlib.pyplot as plt  # Import plotting library

metrics = [  # List of tuples for each metric to plot and their display name
    ('pixel_accuracy', 'Pixel Accuracy'),
    ('iou_coef', 'Mean IoU'),
    ('dice_coef', 'Dice Score')
]

epochs = range(1, len(history['pixel_accuracy']) + 1)  # Create a range for x-axis: [1, 2, ..., num_epochs]
fig, axes = plt.subplots(1, 3, figsize=(18, 5))  # Create a row of 3 subplots, set the figure size

# Loop through each axis and metric, plot both train and validation curves
for ax, (m, display_name) in zip(axes, metrics):
    ax.plot(epochs, history[m], label=f"Train {display_name}")          # Plot training metric
    ax.plot(epochs, history[f'val_{m}'], label=f"Val {display_name}")   # Plot validation metric
    ax.set_title(f"{display_name} Over Epochs")                         # Set title for this plot
    ax.set_xlabel("Epoch")                                              # Set x-axis label
    ax.set_ylabel(display_name)                                         # Set y-axis label
    ax.legend()                                                         # Add legend (train/val)
    ax.grid(True)                                                       # Add grid lines for easier reading
    ax.set_xticks(list(epochs))                                         # Show tick mark for every epoch

plt.tight_layout()  # Adjust subplot spacing
plt.show()          # Display all three plots side by side
```

Figure 17: Plotting Evaluation Metrics