Project Report

Vehicular Technology

Semantic Segmentation using FCN|U-net| Deeplabv3+

Introduction

In this project, we explore the task of semantic segmentation using the CamVid dataset, which provides urban driving scenes with pixel-wise annotations for 32 semantic classes. Semantic segmentation plays a critical role in understanding road scenes in autonomous vehicles by classifying each pixel into predefined categories such as roads, pedestrians, vehicles, trees, traffic signs, etc.

To evaluate and compare different segmentation approaches, we implement and train three widely recognized deep learning architectures:

- Fully Convolutional Network (FCN): Based on a ResNet-50 backbone, this model replaces dense layers with convolutional layers to enable pixel-level predictions while preserving spatial structure.
- U-Net with Attention Mechanism: A U-shaped architecture enhanced with attention gates to better focus on relevant regions, improving small object detection and edge precision.
- DeepLabv3+: A state-of-the-art architecture leveraging atrous spatial pyramid pooling (ASPP) and encoder-decoder structure for capturing multi-scale contextual information effectively.

Each model was evaluated on the same dataset splits and underwent similar preprocessing and augmentation pipelines to ensure fair comparison. The main performance metric used is mean Intersection over Union (mIoU).

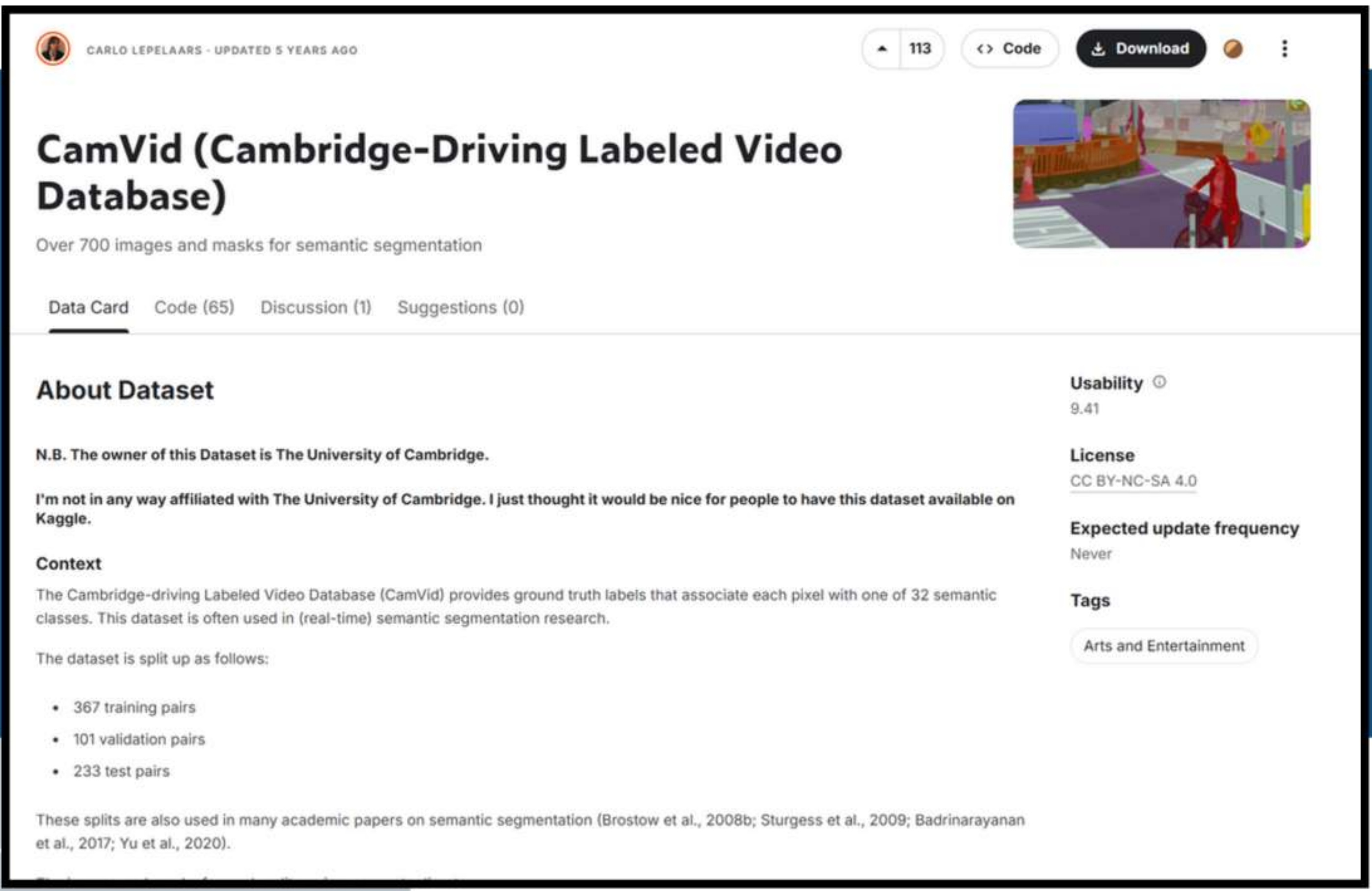
Dataset Description

- **Overview**

The dataset used for this project is the CamVid (Cambridge-driving Labeled Video Database), sourced from Kaggle. CamVid is a well-known dataset in semantic segmentation research.

It contains pixel-level annotations for 32 distinct semantic classes, covering common road scene elements such as roads, buildings, pedestrians, cars, trees, and traffic signs.

This dataset originates from video sequences captured from a moving vehicle, making it suitable for evaluating segmentation models in driving scenarios, albeit limited in temporal scope compared to more modern datasets like KITTI or BDD100K.



CamVid (Cambridge-Driving Labeled Video Database)

Over 700 images and masks for semantic segmentation

About Dataset

N.B. The owner of this Dataset is The University of Cambridge.

I'm not in any way affiliated with The University of Cambridge. I just thought it would be nice for people to have this dataset available on Kaggle.

Context

The Cambridge-driving Labeled Video Database (CamVid) provides ground truth labels that associate each pixel with one of 32 semantic classes. This dataset is often used in (real-time) semantic segmentation research.

The dataset is split up as follows:

- 367 training pairs
- 101 validation pairs
- 233 test pairs

These splits are also used in many academic papers on semantic segmentation (Brostow et al., 2008b; Sturgess et al., 2009; Badrinarayanan et al., 2017; Yu et al., 2020).

Usability
9.41

License
CC BY-NC-SA 4.0

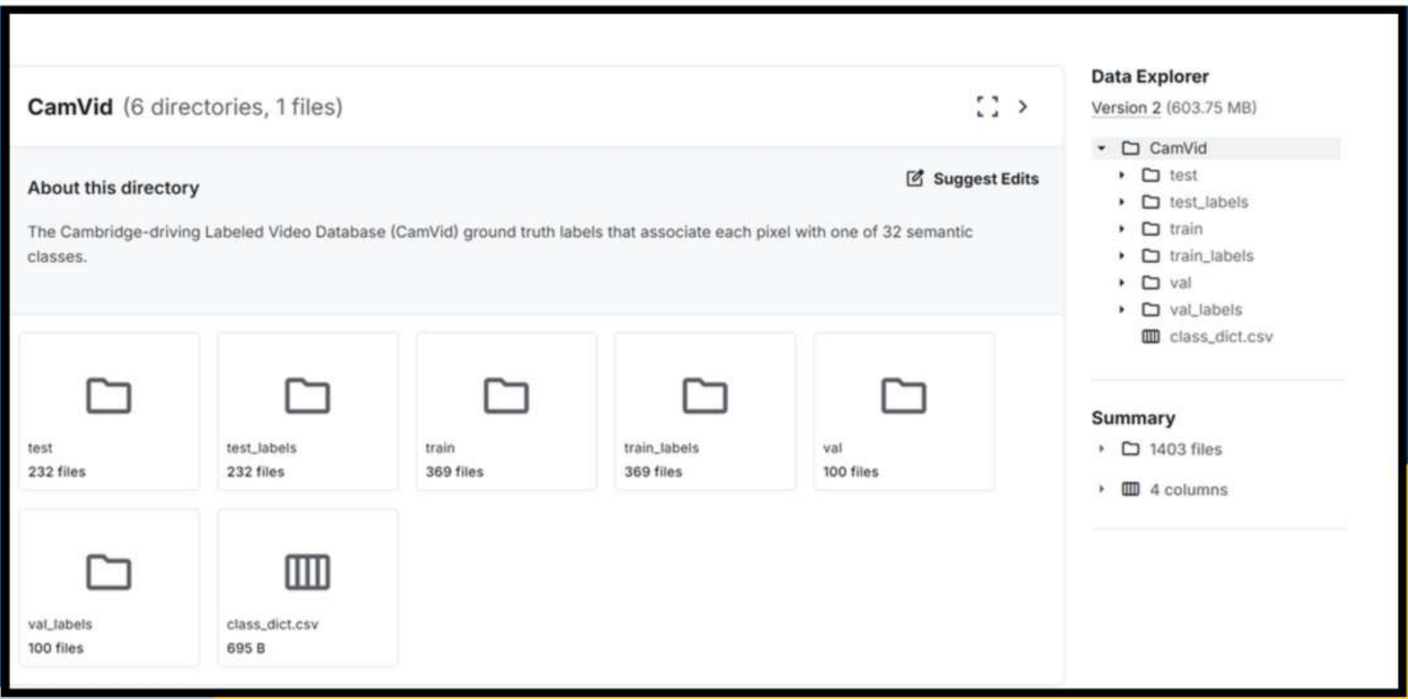
Expected update frequency
Never

Tags
Arts and Entertainment

Dataset Description

- **Structure**

1. The dataset is organized into **six main directories**:
2. **train/ and train_labels/**: 369 images and their corresponding pixel-wise label masks.
3. **val/ and val_labels/**: 100 images and masks for validation.
4. **test/ and test_labels/**: 232 test images and corresponding semantic masks.



CamVid (6 directories, 1 files)

About this directory

The Cambridge-driving Labeled Video Database (CamVid) ground truth labels that associate each pixel with one of 32 semantic classes.

test	232 files
test_labels	232 files
train	369 files
train_labels	369 files
val	100 files
val_labels	100 files
class_dict.csv	695 B

Data Explorer

Version 2 (603.75 MB)

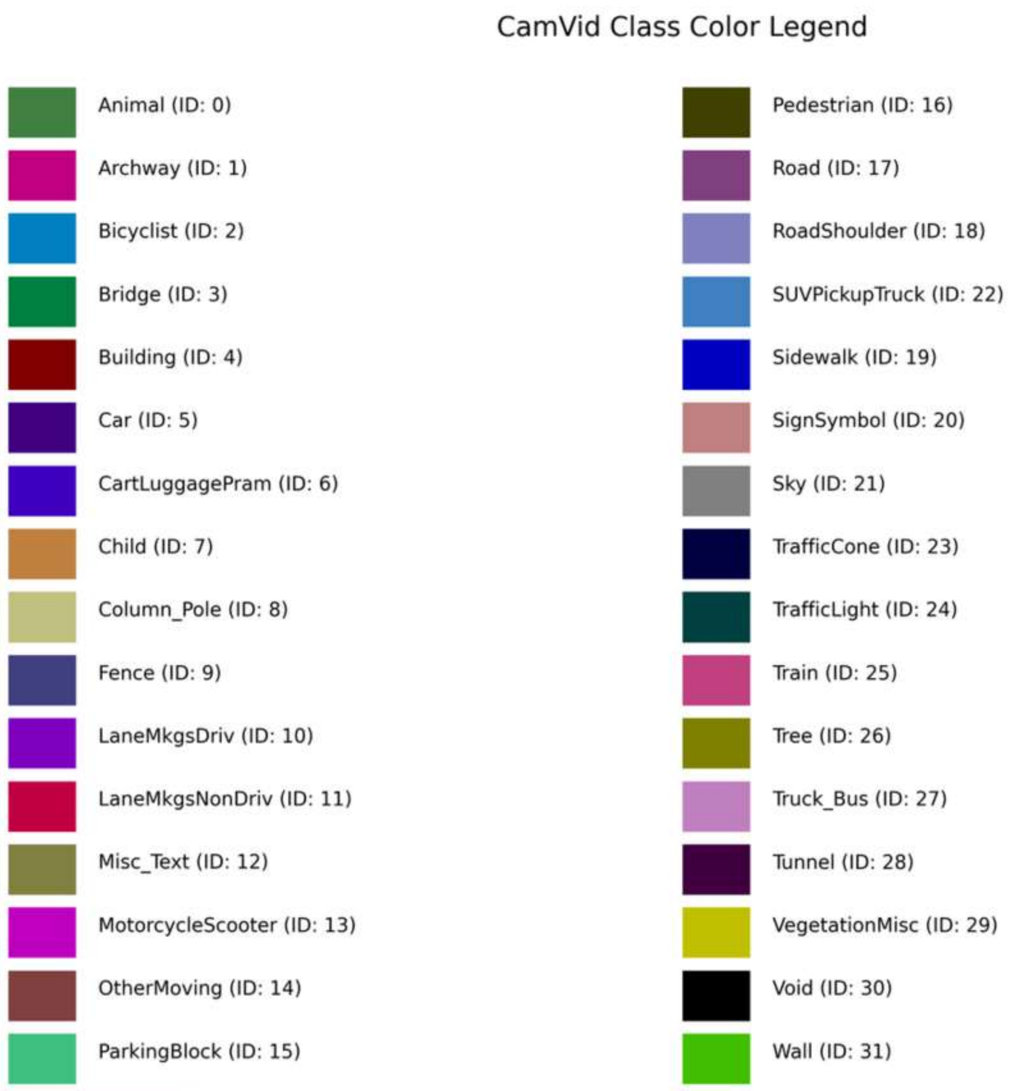
- CamVid
  - test
  - test_labels
  - train
  - train_labels
  - val
  - val_labels
  - class_dict.csv

Summary

- 1403 files
- 4 columns

Dataset Description

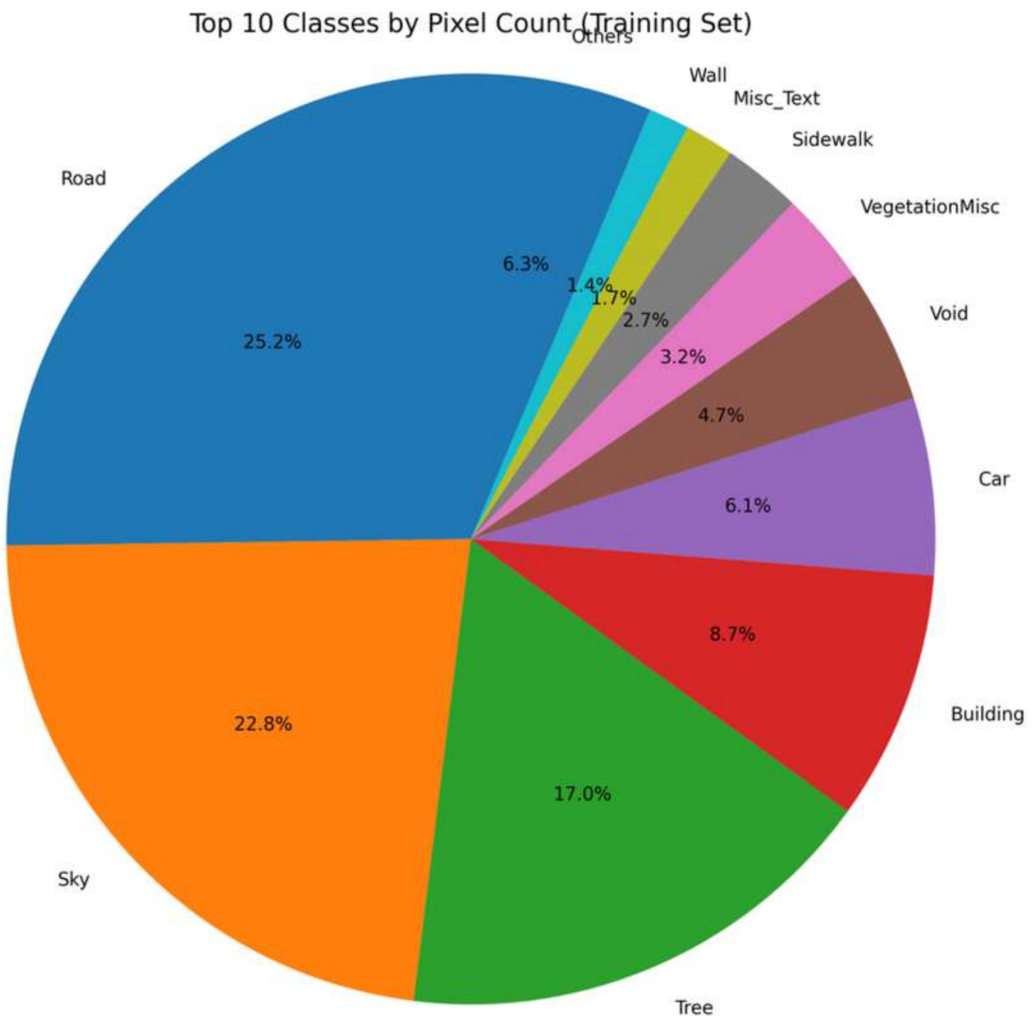
5. class_dict.csv: Contains the RGB values associated with each of the 32 classes, used to convert RGB masks into class index labels. This allwos an easy interpretation of segmentation results, allowing for quick identification of objects such as roads, vehicles, pedestrians, buildings, etc. within urban driving scenes.

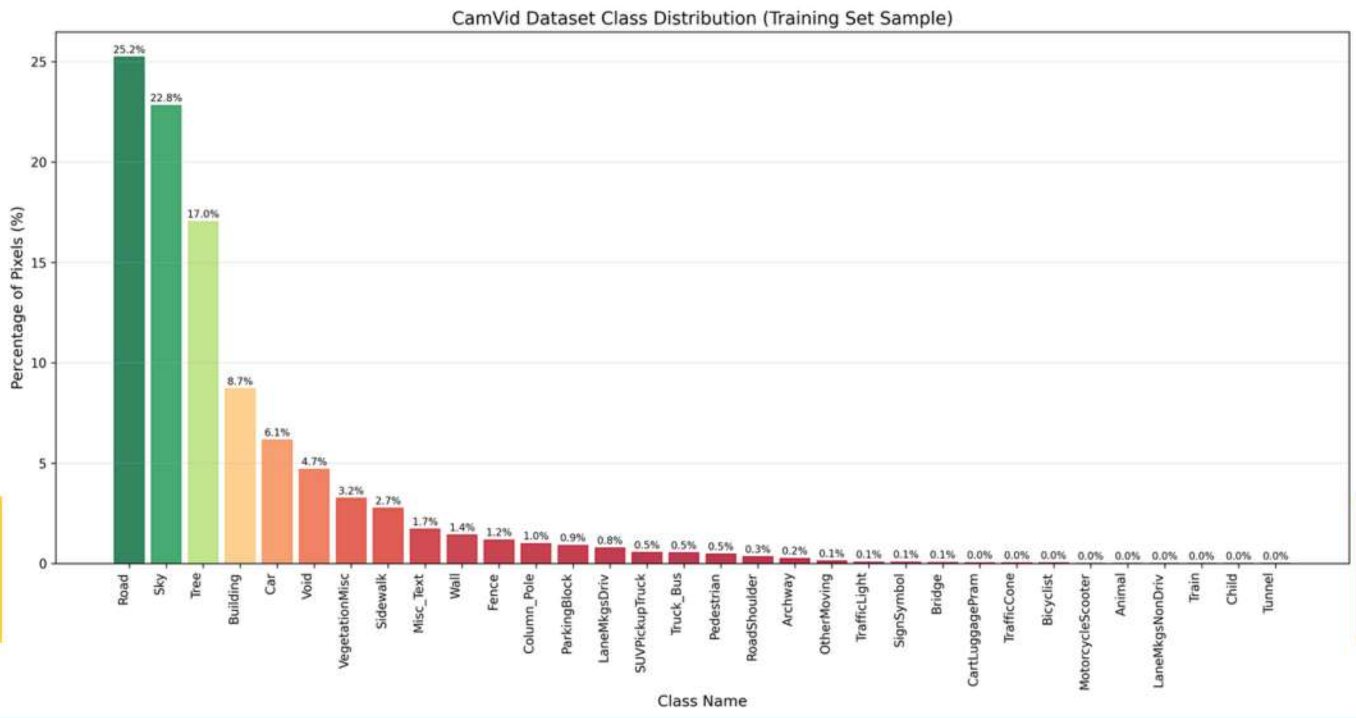


Dataset Description

- **Characteristics**

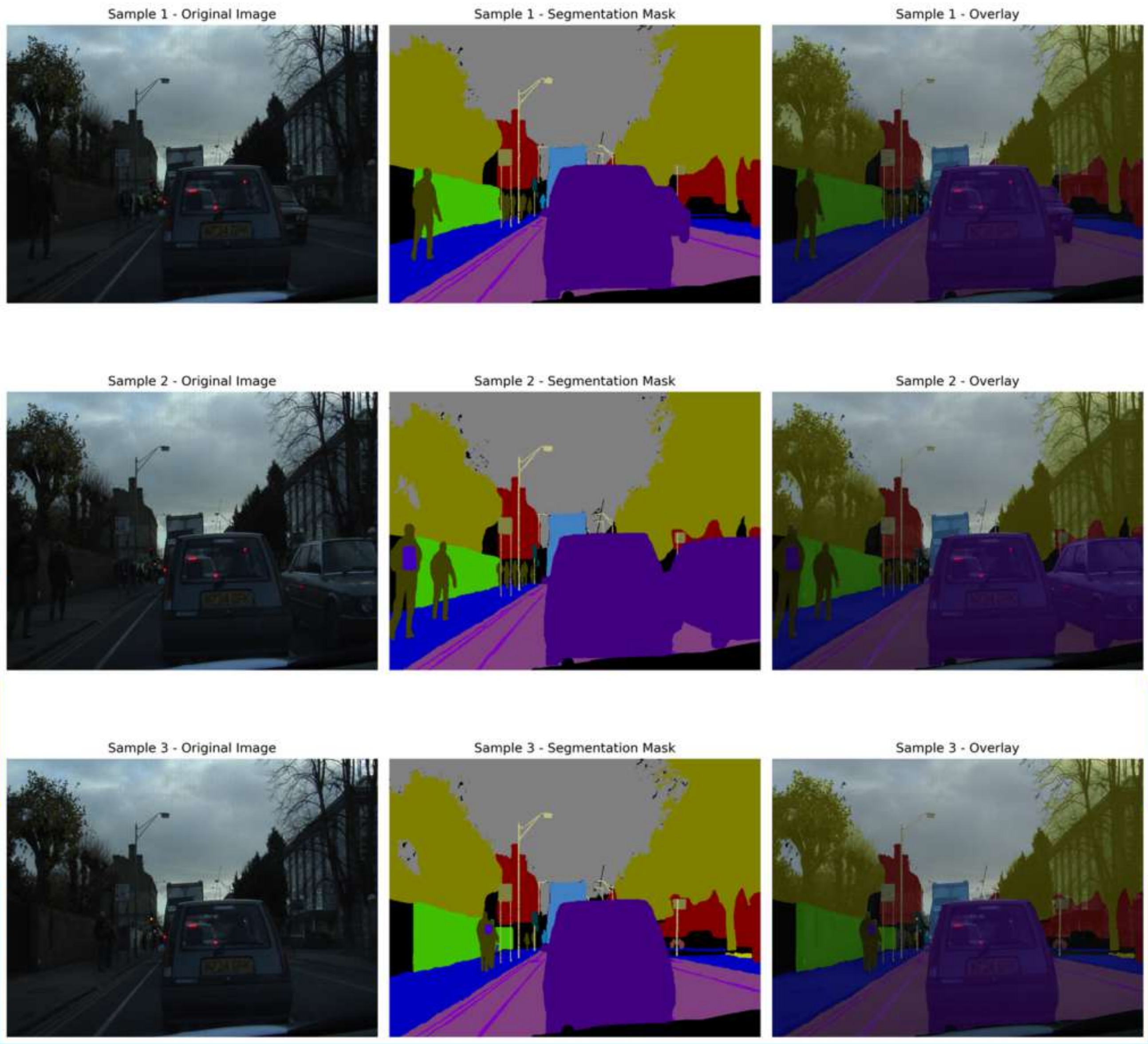
1. **Image Format**: All images are in .png format and are derived from video sequences.
2. **Scene Type**: Urban street scenes captured in daytime, with a variety of lighting and weather conditions.
3. **Frame Continuity**: Many images appear sequential (e.g., 0001TP_006690.png, 0001TP_006720.png, etc.), which allows for limited temporal analysis or tracking.
4. **Class Distribution:** The dataset is highly imbalanced, with classes like road, sky, and tree more dominant in the pixel count, while many objects, like pedestrians, traffic lights, and signs, make up only a small fraction.


Top 10 Classes by Pixel Count (Training Set)
Others
Wall
Misc_Text
Sidewalk
VegetationMisc
Void
Car
Building
Tree
Sky
Road
6.3%
1.4%
1.7%
2.7%
3.2%
4.7%
6.1%
8.7%
17.0%
22.8%
25.2%


CamVid Dataset Class Distribution (Training Set Sample)
Percentage of Pixels (%)
Class Name
0
5
10
15
20
25
Road 25.2%
Sky 22.8%
Tree 17.0%
Building 8.7%
Car 6.1%
Void 4.7%
VegetationMisc 3.2%
Sidewalk 2.7%
Misc_Text 1.7%
Wall 1.4%
Fence 1.2%
Column_Pole 1.0%
ParkingBlock 0.9%
LaneMkgsDriv 0.8%
SUVPickupTruck 0.5%
Truck_Bus 0.5%
Pedestrian 0.5%
RoadShoulder 0.3%
Archway 0.2%
OtherMoving 0.1%
TrafficLight 0.1%
SignSymbol 0.1%
Bridge 0.1%
CartLuggagePram 0.0%
TrafficCone 0.0%
Bicyclist 0.0%
MotorcycleScooter 0.0%
Animal 0.0%
LaneMkgsNonDriv 0.0%
Train 0.0%
Child 0.0%
Tunnel 0.0%

Dataset Description

- Below are a few examples of how the CamVid dataset labels each part of an urban driving scene. You can see the original image, the detailed segmentation mask, and how the two look when combined. This side-by-side view makes it easy to spot which objects and regions are recognized in each frame, and gives a clear sense of how well the pixel-wise annotations capture the complexity of real-world roads.

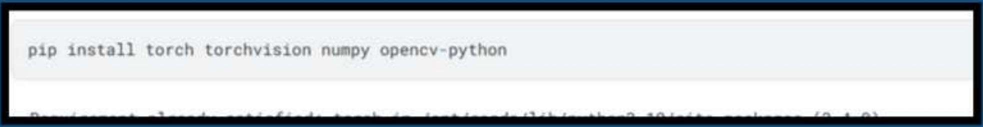

Sample 1 - Original Image
Sample 1 - Segmentation Mask
Sample 1 - Overlay
Sample 2 - Original Image
Sample 2 - Segmentation Mask
Sample 2 - Overlay
Sample 3 - Original Image
Sample 3 - Segmentation Mask
Sample 3 - Overlay

Environment and Setup

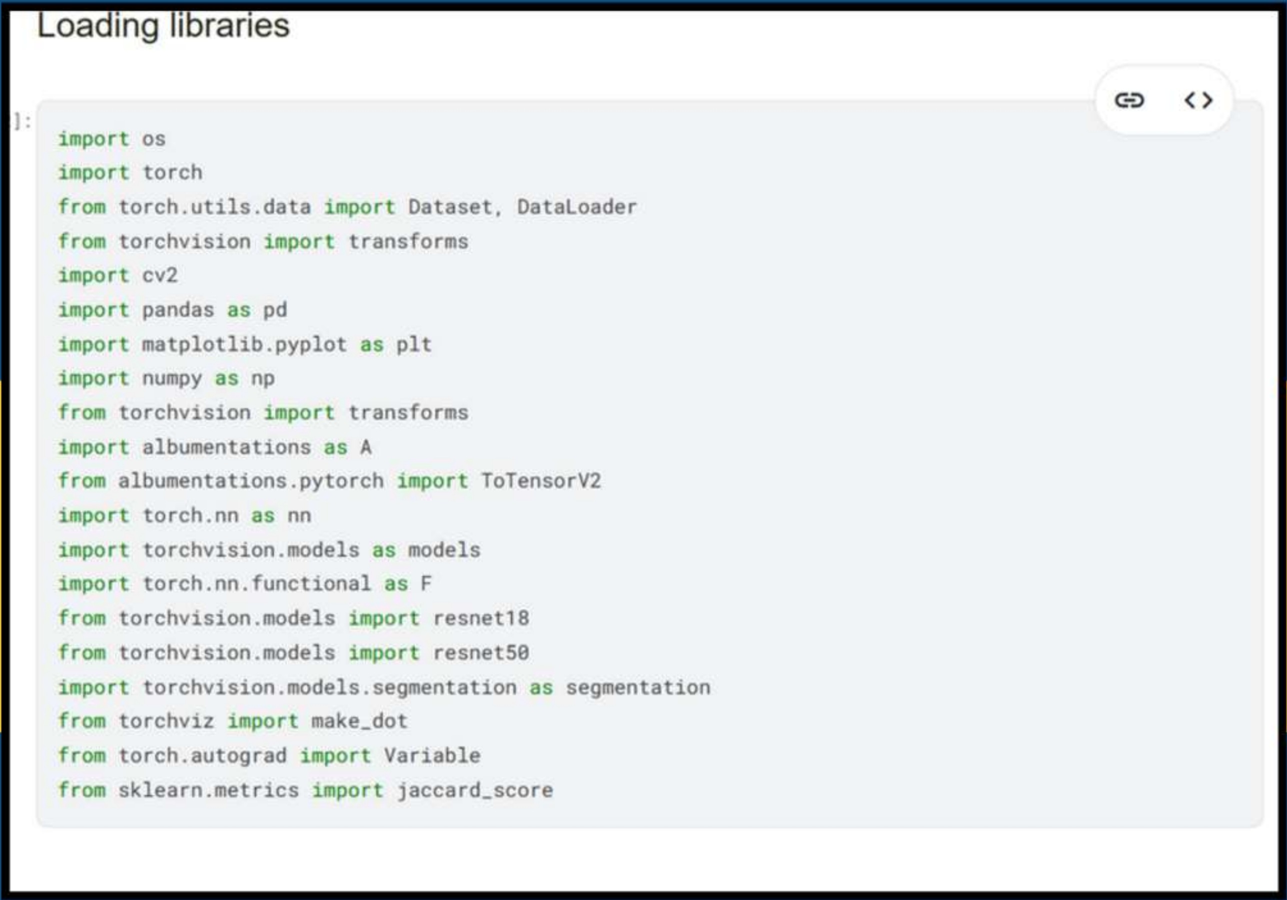
Library Installation

To support the segmentation models and data processing, the following Python libraries were used:

- **torch, torchvision**: Core deep learning frameworks for model training and evaluation.
- **albumentations**: Efficient and flexible image augmentation library.
- **opencv-python**: For reading and processing image data.
- **matplotlib, pandas, numpy**: For data handling and visualization.
- **scikit-learn**: Specifically for computing the Jaccard Index (IoU).



```
pip install torch torchvision numpy opencv-python
```



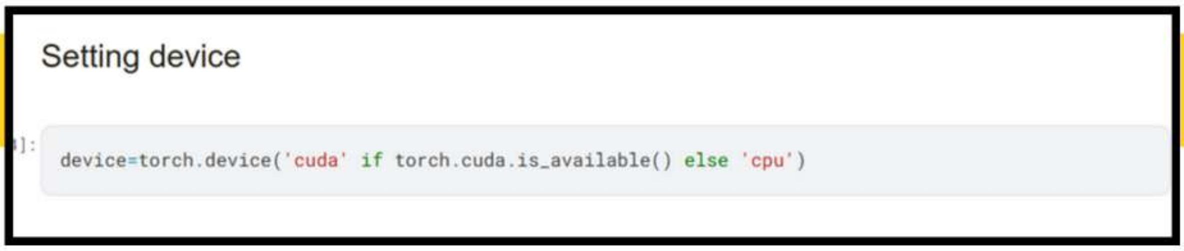
Loading libraries

```
import os
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import cv2
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from torchvision import transforms
import albumentations as A
from albumentations.pytorch import ToTensorV2
import torch.nn as nn
import torchvision.models as models
import torch.nn.functional as F
from torchvision.models import resnet18
from torchvision.models import resnet50
import torchvision.models.segmentation as segmentation
from torchviz import make_dot
from torch.autograd import Variable
from sklearn.metrics import jaccard_score
```

Environment and Setup

Hardware

Training and evaluation were conducted using:



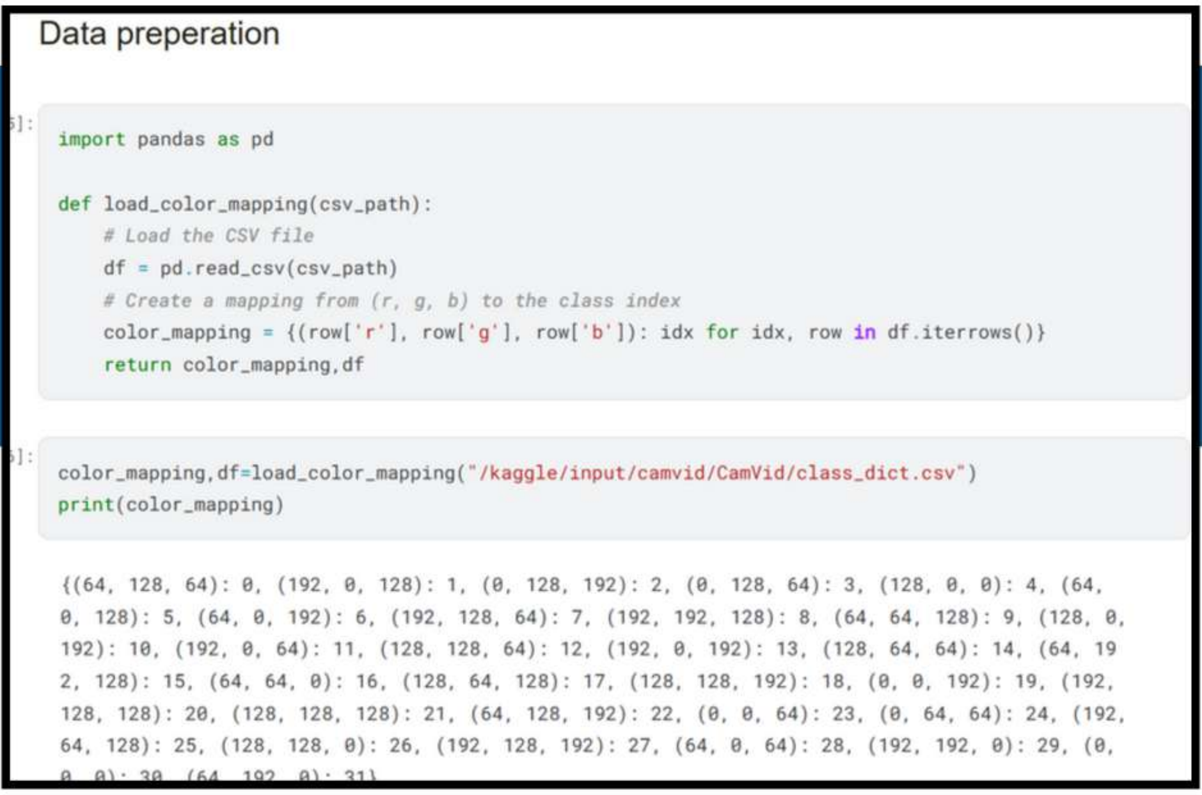
Setting device

```
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

This setup ensures the training can leverage GPU acceleration where available, significantly speeding up model convergence.

Label Color Mapping

A custom function reads the class_dict.csv file and converts RGB segmentation masks into integer class index masks:



Data preperation

```
import pandas as pd

def load_color_mapping(csv_path):
    # Load the CSV file
    df = pd.read_csv(csv_path)
    # Create a mapping from (r, g, b) to the class index
    color_mapping = {(row['r'], row['g'], row['b']): idx for idx, row in df.iterrows()}
    return color_mapping,df
```

```
color_mapping,df=load_color_mapping("/kaggle/input/camvid/CamVid/class_dict.csv")
print(color_mapping)
```

```
{(64, 128, 64): 0, (192, 0, 128): 1, (0, 128, 192): 2, (0, 128, 64): 3, (128, 0, 0): 4, (64,
0, 128): 5, (64, 0, 192): 6, (192, 128, 64): 7, (192, 192, 128): 8, (64, 64, 128): 9, (128, 0,
192): 10, (192, 0, 64): 11, (128, 128, 64): 12, (192, 0, 192): 13, (128, 64, 64): 14, (64, 19
2, 128): 15, (64, 64, 0): 16, (128, 64, 128): 17, (128, 128, 192): 18, (0, 0, 192): 19, (192,
128, 128): 20, (128, 128, 128): 21, (64, 128, 192): 22, (0, 0, 64): 23, (0, 64, 64): 24, (192,
64, 128): 25, (128, 128, 0): 26, (192, 128, 192): 27, (64, 0, 64): 28, (192, 192, 0): 29, (0,
0, 0): 30, (64, 192, 0): 31}
```

This is essential for preparing masks in a format compatible with PyTorch's cross-entropy loss.

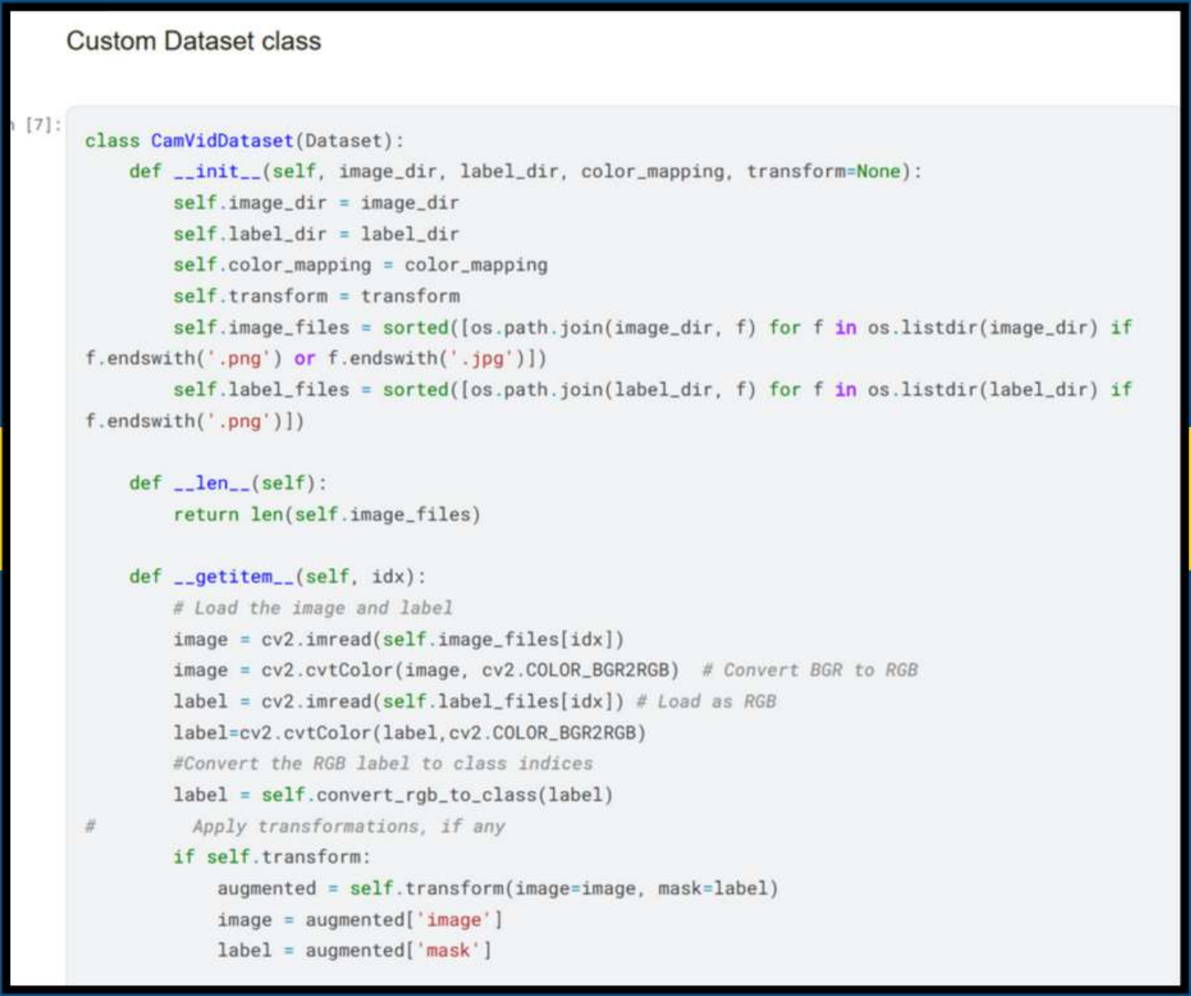
Data Preprocessing and Augmentation

Custom Dataset Class

To effectively manage the CamVid dataset for training, validation, and testing, a custom PyTorch Dataset class named CamVidDataset was implemented. This class:

- Loads RGB images and their corresponding label masks.
- Converts BGR images (as read by OpenCV) to RGB format.
- Converts label masks from RGB encoding to integer class indices using the class_dict.csv.
- Applies optional augmentations and normalization during training or testing.

This structure enables seamless integration with PyTorch's DataLoader and ensures efficient batch processing during training.



Custom Dataset class

```
class CamVidDataset(Dataset):
    def __init__(self, image_dir, label_dir, color_mapping, transform=None):
        self.image_dir = image_dir
        self.label_dir = label_dir
        self.color_mapping = color_mapping
        self.transform = transform
        self.image_files = sorted([os.path.join(image_dir, f) for f in os.listdir(image_dir) if
f.endswith('.png') or f.endswith('.jpg')])
        self.label_files = sorted([os.path.join(label_dir, f) for f in os.listdir(label_dir) if
f.endswith('.png')])

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        # Load the image and label
        image = cv2.imread(self.image_files[idx])
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB
        label = cv2.imread(self.label_files[idx]) # Load as RGB
        label=cv2.cvtColor(label,cv2.COLOR_BGR2RGB)
        #Convert the RGB label to class indices
        label = self.convert_rgb_to_class(label)
#         Apply transformations, if any
        if self.transform:
            augmented = self.transform(image=image, mask=label)
            image = augmented['image']
            label = augmented['mask']
```

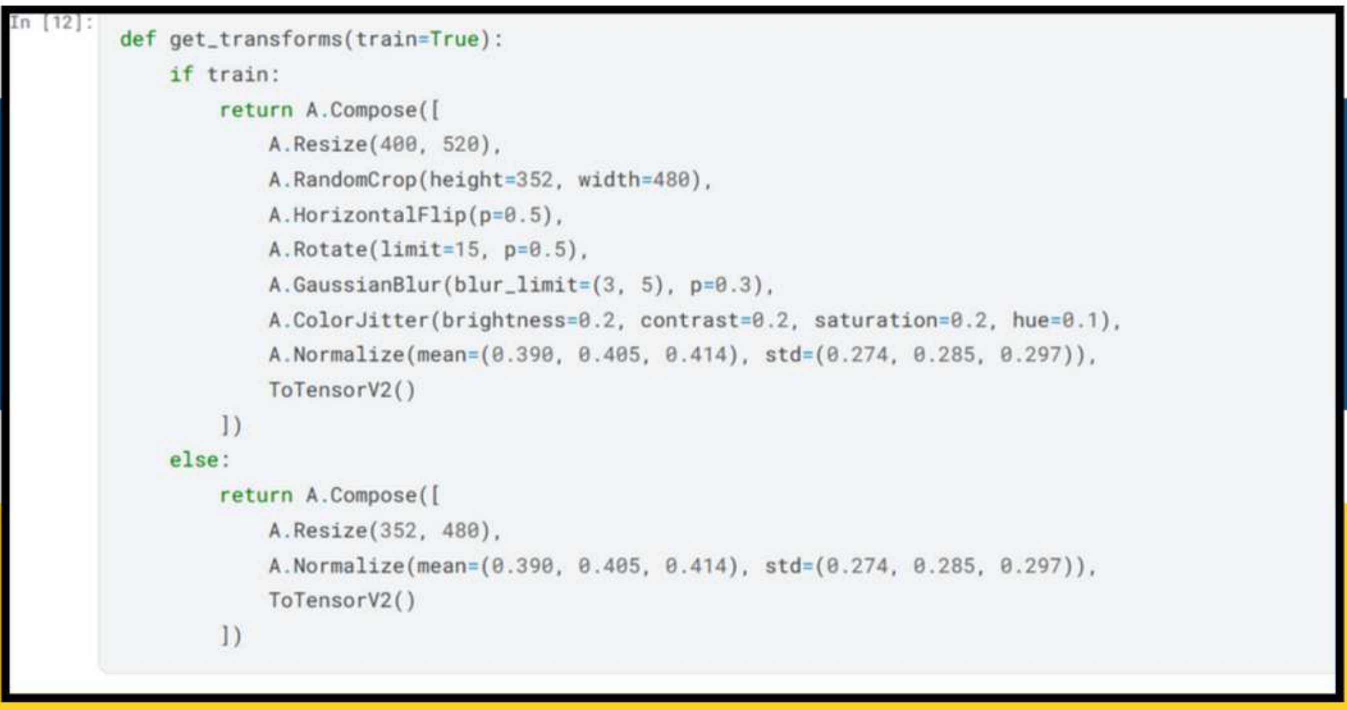
Data Preprocessing and Augmentation

Data Augmentation Pipeline

To increase the model's robustness and prevent overfitting, extensive data augmentation was applied to training samples using the Albumentations library. The augmentation techniques include:

- Resize to 400×520 followed by RandomCrop to 352×480.
- Horizontal Flip with 50% probability.
- Random Rotation within a ±15° range.
- Gaussian Blur with a small kernel.
- Color Jitter (brightness, contrast, saturation, hue) to simulate lighting variability.
- Normalization with dataset-specific mean and standard deviation.
- ToTensorV2 to convert images and labels into PyTorch tensor format.

For validation and test sets, only resizing and normalization were applied to preserve data integrity.

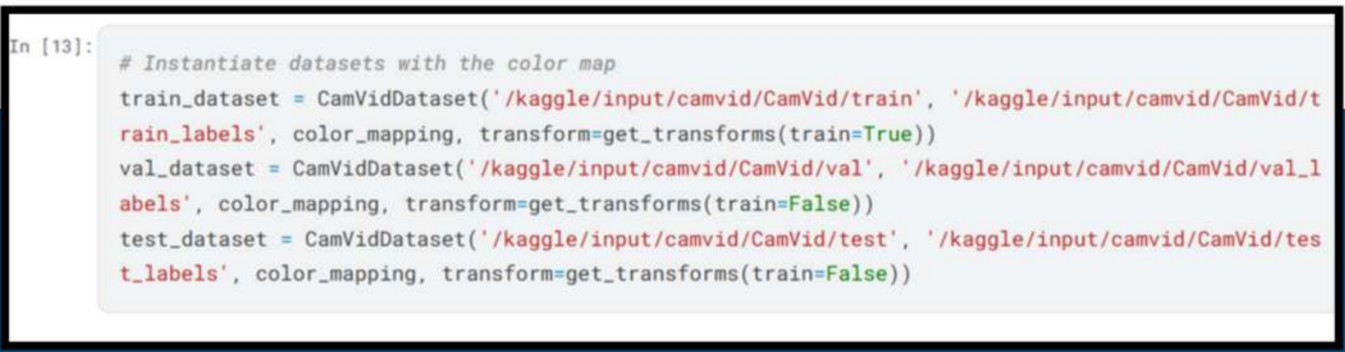


```
In [12]:
def get_transforms(train=True):
    if train:
        return A.Compose([
            A.Resize(400, 520),
            A.RandomCrop(height=352, width=480),
            A.HorizontalFlip(p=0.5),
            A.Rotate(limit=15, p=0.5),
            A.GaussianBlur(blur_limit=(3, 5), p=0.3),
            A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
            A.Normalize(mean=(0.390, 0.405, 0.414), std=(0.274, 0.285, 0.297)),
            ToTensorV2()
        ])
    else:
        return A.Compose([
            A.Resize(352, 480),
            A.Normalize(mean=(0.390, 0.405, 0.414), std=(0.274, 0.285, 0.297)),
            ToTensorV2()
        ])
```

Data Preprocessing and Augmentation

Dataset Initialization

Three datasets were created for training, validation, and testing:



```
In [13]:
# Instantiate datasets with the color map
train_dataset = CamVidDataset('/kaggle/input/camvid/CamVid/train', '/kaggle/input/camvid/CamVid/t
rain_labels', color_mapping, transform=get_transforms(train=True))
val_dataset = CamVidDataset('/kaggle/input/camvid/CamVid/val', '/kaggle/input/camvid/CamVid/val_l
abels', color_mapping, transform=get_transforms(train=False))
test_dataset = CamVidDataset('/kaggle/input/camvid/CamVid/test', '/kaggle/input/camvid/CamVid/tes
t_labels', color_mapping, transform=get_transforms(train=False))
```

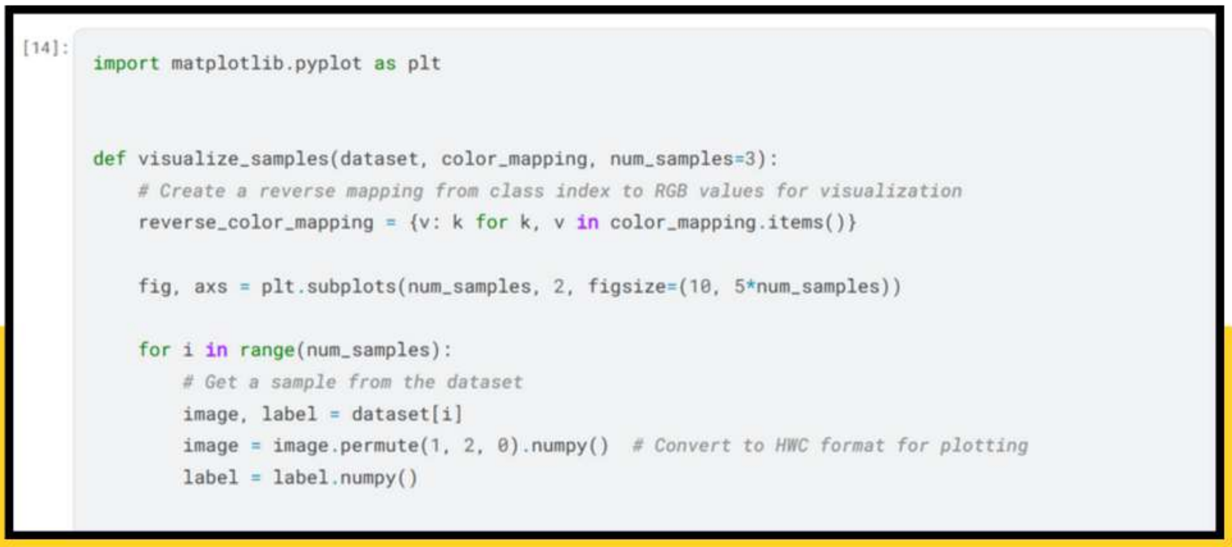
Show Examples of Images and Labels

visualization function

- Converts the dataset output (image + label) into a human-readable format.
- Maps class indices back to RGB colors using the reverse_color_mapping.
- Plots the original image and its corresponding label mask side by side.

Importance: It confirms that your dataset loading and color mapping logic is working correctly — a crucial step for segmentation tasks.

This is now included in the revised section below under "Visualization Function".



```
[14]:
import matplotlib.pyplot as plt


def visualize_samples(dataset, color_mapping, num_samples=3):
    # Create a reverse mapping from class index to RGB values for visualization
    reverse_color_mapping = {v: k for k, v in color_mapping.items()}

    fig, axs = plt.subplots(num_samples, 2, figsize=(10, 5*num_samples))

    for i in range(num_samples):
        # Get a sample from the dataset
        image, label = dataset[i]
        image = image.permute(1, 2, 0).numpy()  # Convert to HWC format for plotting
        label = label.numpy()
```

Data Preprocessing and Augmentation

Output of Visualization Function

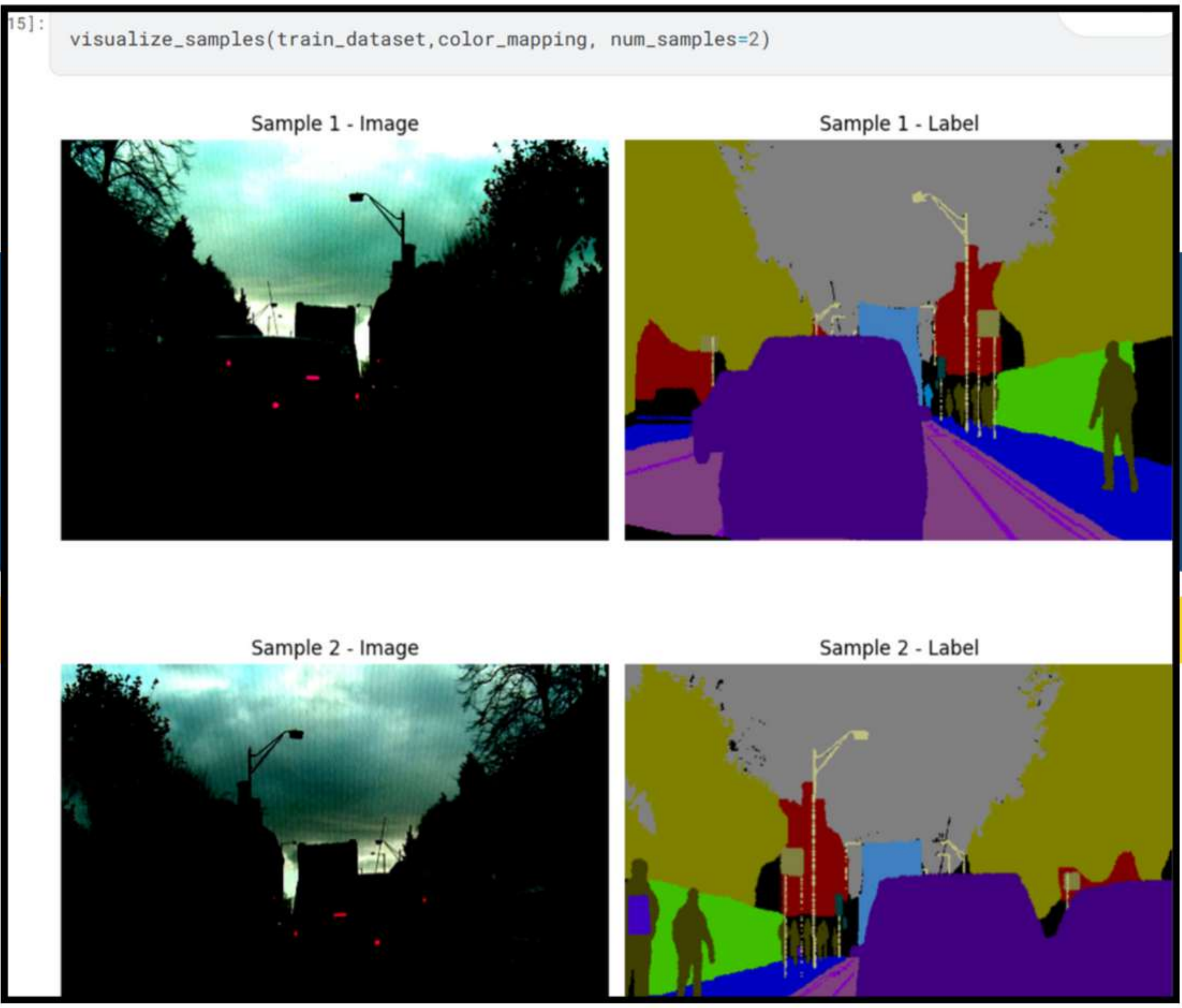
This is the output of the above visualization function, showing:

- Left: The raw image
- Right: Its corresponding ground truth segmentation mask

Importance:

Demonstrates that the CamVidDataset and color_mapping pipeline is correct.

Shows clear object boundaries and accurate mask alignment — important for training quality.

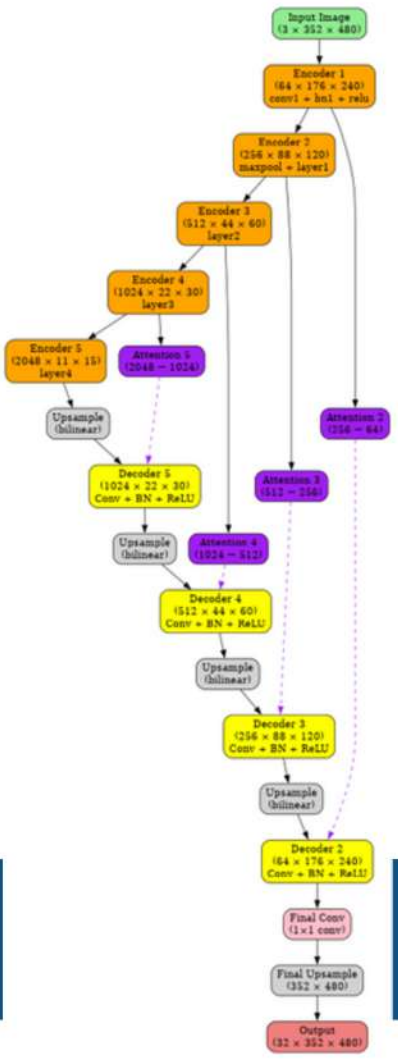


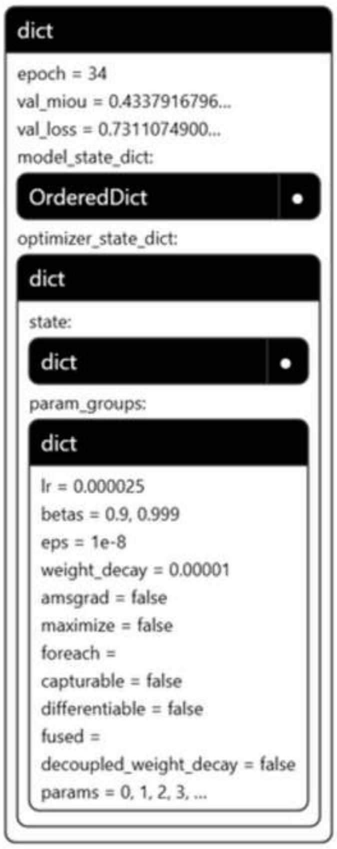
```
[15]:  visualize_samples(train_dataset,color_mapping, num_samples=2)
```

Model Implementation

U-Net with Attention and ResNet-50 Backbone

- U-Net with Attention and ResNet-50 Backbone
- This model adapts the U-Net architecture by incorporating attention blocks at each skip connection. The attention mechanism allows the model to focus on relevant features while suppressing noise from irrelevant spatial areas.
- Encoder: Based on ResNet-50 with five encoder blocks (from conv1 to layer4).
- Attention Blocks: Learn spatial gating between encoder and decoder features using convolutional attention.
- Decoder: Uses bilinear interpolation followed by convolution layers at each stage, enhanced by attention.
- This architecture is designed to improve performance on small objects and fine details, such as pedestrians or traffic lights.


Input Image (3 × 352 × 480)
Encoder 1 (64 × 176 × 240) conv1 + bn1 + relu
Encoder 2 (256 × 88 × 120) maxpool + layer1
Encoder 3 (512 × 44 × 60) layer2
Encoder 4 (1024 × 22 × 30) layer3
Encoder 5 (2048 × 11 × 15) layer4
Attention 5 (2048 → 1024)
Attention 2 (256 → 64)
Upsample (bilinear)
Decoder 5 (1024 × 22 × 30) Conv + BN + ReLU
Attention 3 (512 → 256)
Upsample (bilinear)
Attention 4 (1024 → 512)
Decoder 4 (512 × 44 × 60) Conv + BN + ReLU
Upsample (bilinear)
Decoder 3 (256 × 88 × 120) Conv + BN + ReLU
Upsample (bilinear)
Decoder 2 (64 × 176 × 240) Conv + BN + ReLU
Final Conv (1×1 conv)
Final Upsample (352 × 480)
Output (32 × 352 × 480)


```
dict
epoch = 34
val_miou = 0.4337916796...
val_loss = 0.7311074900...
model_state_dict:
  OrderedDict
optimizer_state_dict:
  dict
  state:
    dict
  param_groups:
    dict
    lr = 0.000025
    betas = 0.9, 0.999
    eps = 1e-8
    weight_decay = 0.00001
    amsgrad = false
    maximize = false
    foreach =
    capturable = false
    differentiable = false
    fused =
    decoupled_weight_decay = false
    params = 0, 1, 2, 3, ...
```

Model Implementation

Fully Convolutional Network (FCN) with ResNet-50 Backbone

To better understand and document the UNet with attention mechanism used in this semantic segmentation project, we implemented several visualization and analysis tools. The model combines a ResNet-50 encoder backbone with a custom decoder featuring attention gates at each skip connection level, designed for multi-class segmentation on the CamVid dataset with 32 classes.

First, a computational graph was generated (using torchviz) to show the full set of operations involved in the forward pass, which is helpful for debugging and for presentations. For a quick overview, we printed a classic layer-by-layer model summary (via torchsummary), and also wrote a recursive module walker to print the model's architecture in a readable format.

To dig into model complexity, we analyzed parameter counts, showing the total (62,940,044), trainable (62,940,044), and non-trainable parameters (0) as well as the distribution across different model components—information that's key for estimating computational cost and memory usage. The model size of 240.10 MB reflects its substantial capacity for learning complex segmentation patterns.

The architecture features a sophisticated attention mechanism with four attention blocks (attention2-5) strategically placed at different decoder levels. Each attention block uses a gating mechanism with learnable parameters (W_g, W_x, psi) to selectively focus on relevant features from the encoder path, helping the model concentrate on important spatial regions during upsampling. The attention blocks have varying channel dimensions (32, 128, 256, 512) to match the corresponding encoder feature maps.

A simple pipeline diagram of the model's high-level stages (ResNet-50 encoder, attention gates, decoder blocks, final classifier) was also generated for visualization purposes. The encoder extracts hierarchical features through ResNet-50 layers, while the decoder progressively upsamples with attention-guided skip connections, culminating in a final 1x1 convolution layer producing 32-class predictions.

Finally, we tested the model with different input sizes to ensure output shapes match expectations and the architecture can handle common resolutions. The model achieved a best validation mIoU of 0.4338 over 35 epochs of training, with final training and validation accuracies of 88.44% and 88.37% respectively. This comprehensive visualization workflow made it easy to interpret, debug, and communicate the model's structure, particularly highlighting how the attention mechanism enhances the traditional U-Net architecture for improved semantic segmentation performance.

Model Implementation

DeepLabv3+ with ResNet-50

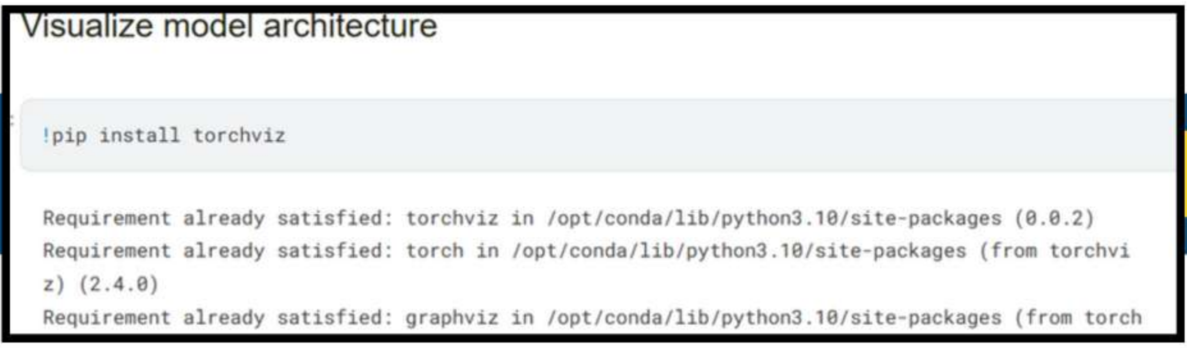
To better understand and document the DeepLabV3+ model used in this project, we implemented several visualization and analysis tools. First, a computational graph was generated (using torchviz) to show the full set of operations involved in the forward pass, which is helpful for debugging and for presentations. For a quick overview, we printed a classic layer-by-layer model summary (via torchsummary), and also wrote a recursive module walker to print the model's architecture in a readable format.

To dig into model complexity, we analyzed parameter counts, showing the total, trainable, and non-trainable parameters as well as the distribution across different model components—information that's key for estimating computational cost and memory usage. A simple pipeline diagram of the model's high-level stages (backbone, ASPP, classifier) was also generated for visualization purposes. Finally, we tested the model with different input sizes to ensure output shapes match expectations and the architecture can handle common resolutions. This comprehensive visualization workflow made it easy to interpret, debug, and communicate the model's structure.

Model Implementation

Model Visualization

To validate the architecture, torchviz was used to visualize the computation graph. This helped verify the structure and dimensionality of tensors across each stage of the network.



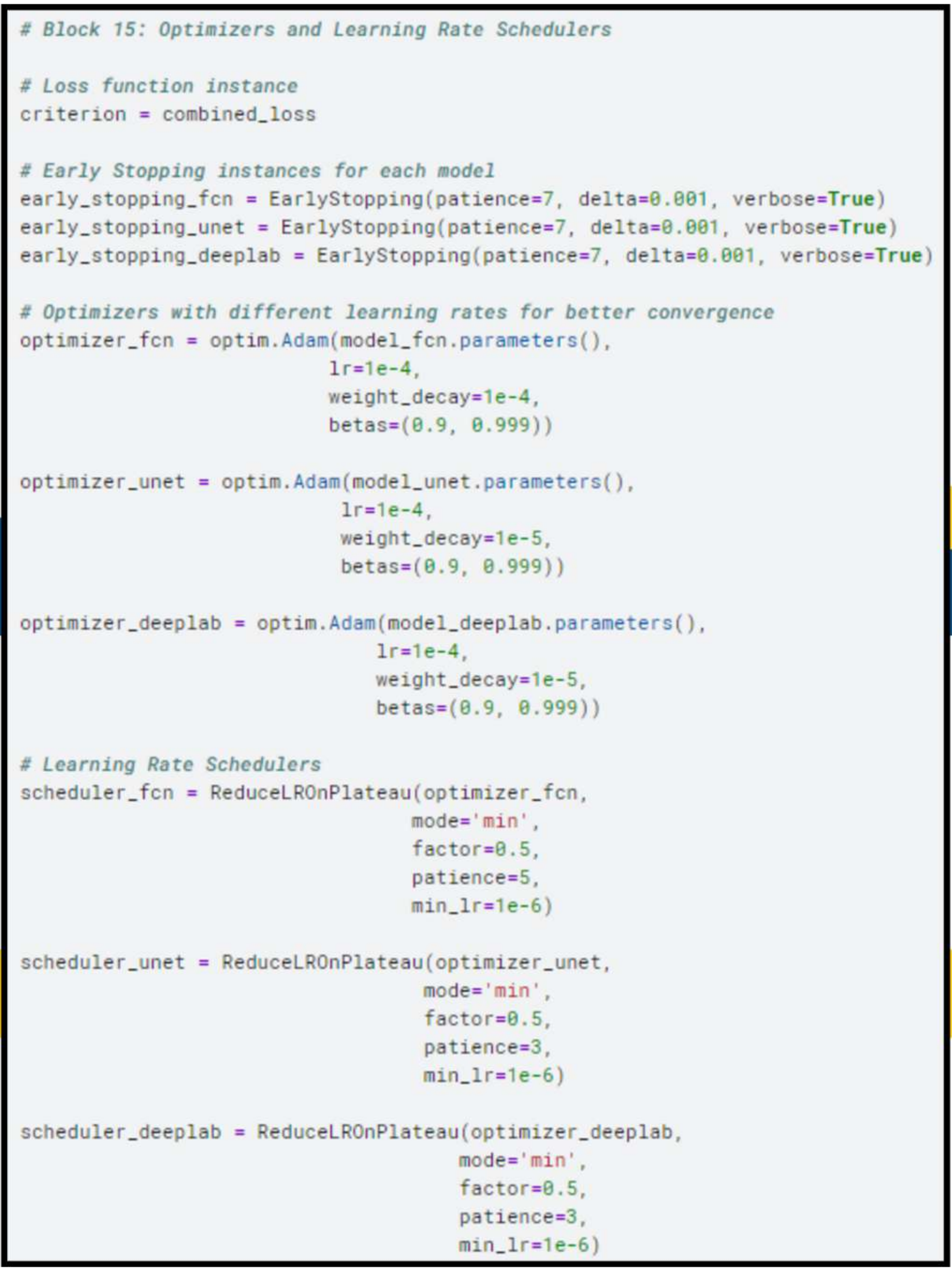
Visualize model architecture

```
!pip install torchviz
```

```
Requirement already satisfied: torchviz in /opt/conda/lib/python3.10/site-packages (0.0.2)
Requirement already satisfied: torch in /opt/conda/lib/python3.10/site-packages (from torchvi
z) (2.4.0)
Requirement already satisfied: graphviz in /opt/conda/lib/python3.10/site-packages (from torch
```

We trained all our segmentation models with Adam optimizer with learning rate 1e-4, mild weight decay (1e-5) and standard betas. We used a separate optimizer for each model (FCN, UNet and DeepLabV3+ for fine tuning. We also applied early stopping with patience of 7 epochs, to avoid overfitting and to accelerate performance convergence. We employed ReduceLROnPlateau scheduler for adaptive learning rate control, which reduced the learning rate by a factor of 0.5 if validation loss does not decrease. We tuned patience values of scheduler per model (e.g., 5 for FCN, 3 for UNet and DeepLab) and trained with minimum learning rate threshold to avoid decreasing too low. It is this combination of optimizers, early stopping and schedulers that enabled a stable and efficient training for all models.

Model Implementation

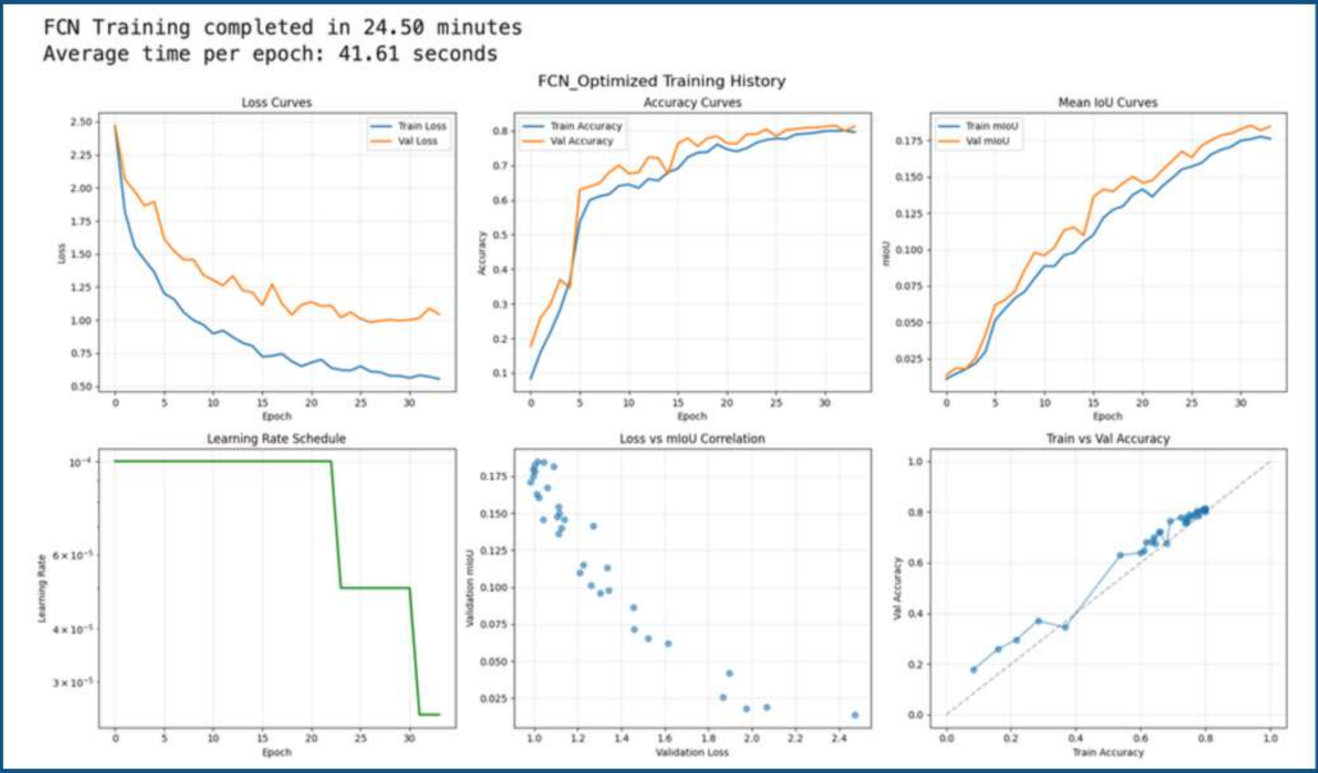


```python
# Block 15: Optimizers and Learning Rate Schedulers

# Loss function instance
criterion = combined_loss

# Early Stopping instances for each model
early_stopping_fcn = EarlyStopping(patience=7, delta=0.001, verbose=True)
early_stopping_unet = EarlyStopping(patience=7, delta=0.001, verbose=True)
early_stopping_deeplab = EarlyStopping(patience=7, delta=0.001, verbose=True)

# Optimizers with different learning rates for better convergence
optimizer_fcn = optim.Adam(model_fcn.parameters(),
                           lr=1e-4,
                           weight_decay=1e-4,
                           betas=(0.9, 0.999))

optimizer_unet = optim.Adam(model_unet.parameters(),
                            lr=1e-4,
                            weight_decay=1e-5,
                            betas=(0.9, 0.999))

optimizer_deeplab = optim.Adam(model_deeplab.parameters(),
                               lr=1e-4,
                               weight_decay=1e-5,
                               betas=(0.9, 0.999))

# Learning Rate Schedulers
scheduler_fcn = ReduceLROnPlateau(optimizer_fcn,
                                  mode='min',
                                  factor=0.5,
                                  patience=5,
                                  min_lr=1e-6)

scheduler_unet = ReduceLROnPlateau(optimizer_unet,
                                   mode='min',
                                   factor=0.5,
                                   patience=3,
                                   min_lr=1e-6)

scheduler_deeplab = ReduceLROnPlateau(optimizer_deeplab,
                                      mode='min',
                                      factor=0.5,
                                      patience=3,
                                      min_lr=1e-6)
```

Performance Metrics

FCN (Fully Convolutional Network)

Following training and validation of the FCN model, we proceeded to reload the best checkpoint (epoch 32) and test on the test set for a final evaluation. The best validation mIoU obtained during training was 0.1851 and on unseen test data the model performed at a test mIoU of 0.1779. Although the overall test accuracy was comparatively high at 79.6%, mIoU numbers reveals that our model has still some difficulty in segmenting the classes accurately, being a challenging dataset like CamVid. The final model has around 68 million parameters and inference on the test set took approximately 2 min. For qualitative analysis, 48 sample predictions were made and saved for visualization.

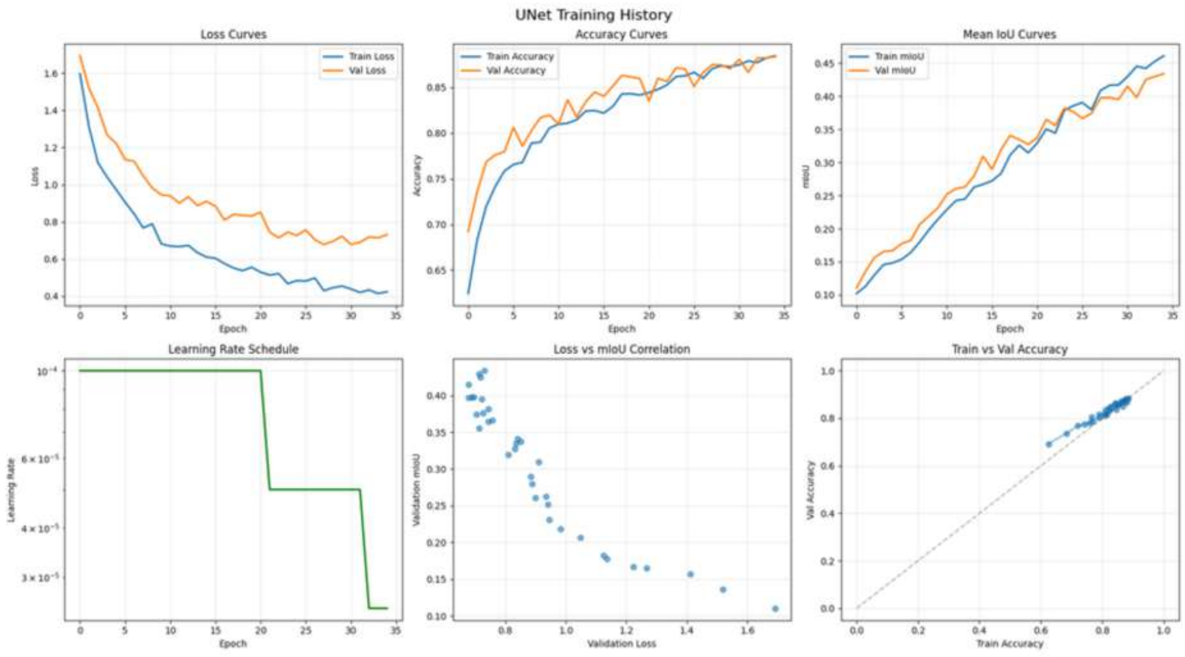


```
FCN Training completed in 24.50 minutes
Average time per epoch: 41.61 seconds
```

Performance Metrics

U-Net

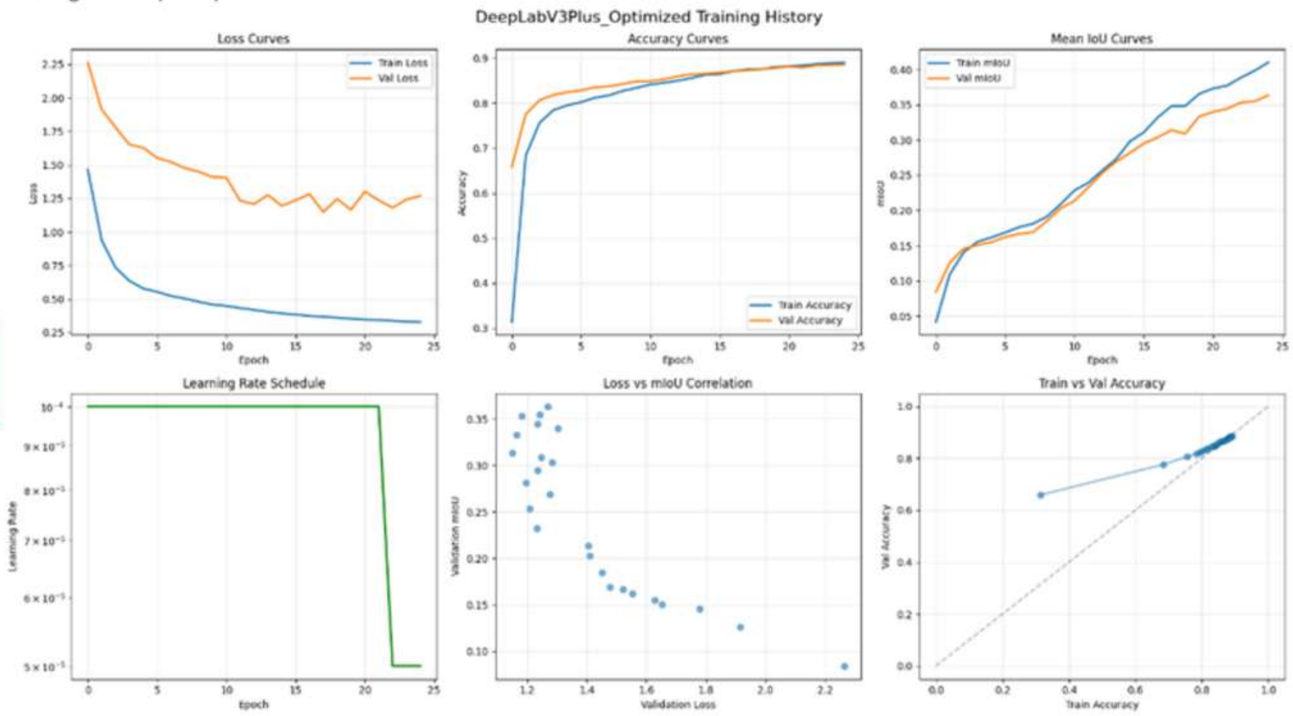
We trained the UNet architecture for 35 epochs using an NVIDIA GeForce RTX 4060 Laptop GPU, with a total training time of approximately 34.3 minutes. Each epoch averaged just under a minute thanks to a relatively large model size of 240 MB and a total of 62.9 million trainable parameters. The network showed solid segmentation capabilities, reaching a best validation mIoU of 0.4338 at the final epoch — outperforming simpler architectures like FCN and even DeepLabV3+ in our setup. Training accuracy peaked at 88.44%, while validation accuracy closely followed at 88.37%, suggesting good generalization. The final training loss was relatively low at 0.4238, whereas validation loss remained higher at 0.7311, hinting at some room for regularization or tuning. Overall, the UNet model, despite its heavier architecture, demonstrated consistent learning and superior segmentation performance, particularly effective in capturing fine spatial details due to its symmetric encoder–decoder structure.



Performance Metrics

DeepLabv3+

We trained the DeepLabV3+ for 25 epochs and it took little more than 15 minutes to complete the whole process with the help of mixed precision training and a Tesla T4 GPU. Each epoch took about 36 seconds on average. The best validation mIoU was 0.3635, which is almost twice compared to the previous FCN model proving considerable improvement of the segmentation. The model achieved its highest performance at the last epoch showing high training and higher validation accuracies of 89.1% and 88.6%, respectively. Our training loss wasn't too high (0.3284) with a higher validation loss (1.2688), it seems that we may still be able to tune it more. In general, the optimizations that we introduced, i.e. the increased batch size and the mixed precision, led to both a faster training and an improved performance.

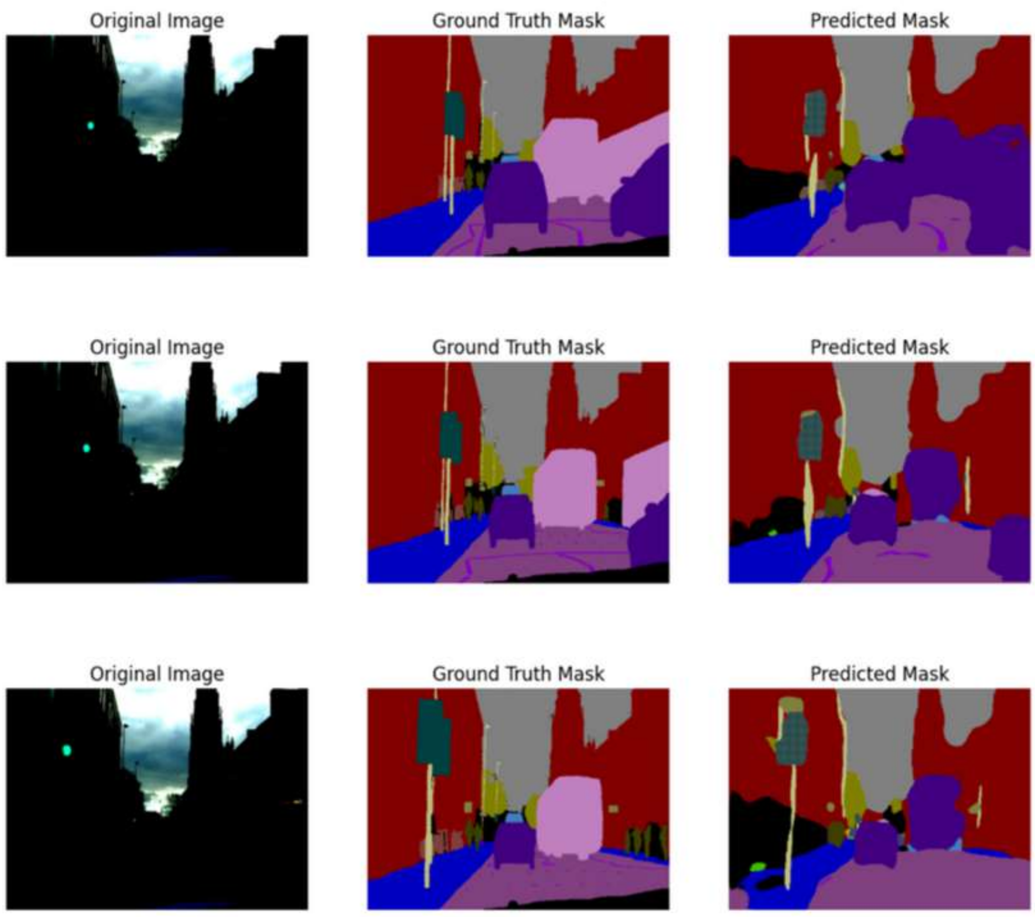


Visual Prediction Analysis

FCN (Fully Convolutional Network)

The segmentation predictions from the FCN model show that the overall structure of the scenes is moderately captured, with decent alignment to the ground truth. The model performs well in distinguishing dominant classes like road, sky, and buildings. However, it occasionally struggles with finer boundaries and small objects such as pedestrians, poles, and traffic signs, leading to some blending between adjacent classes. The predicted masks appear slightly smoother than the ground truth, suggesting that FCN tends to generalize and sacrifice detail in favor of stability. This is consistent with its relatively lower mean IoU and pixel accuracy.

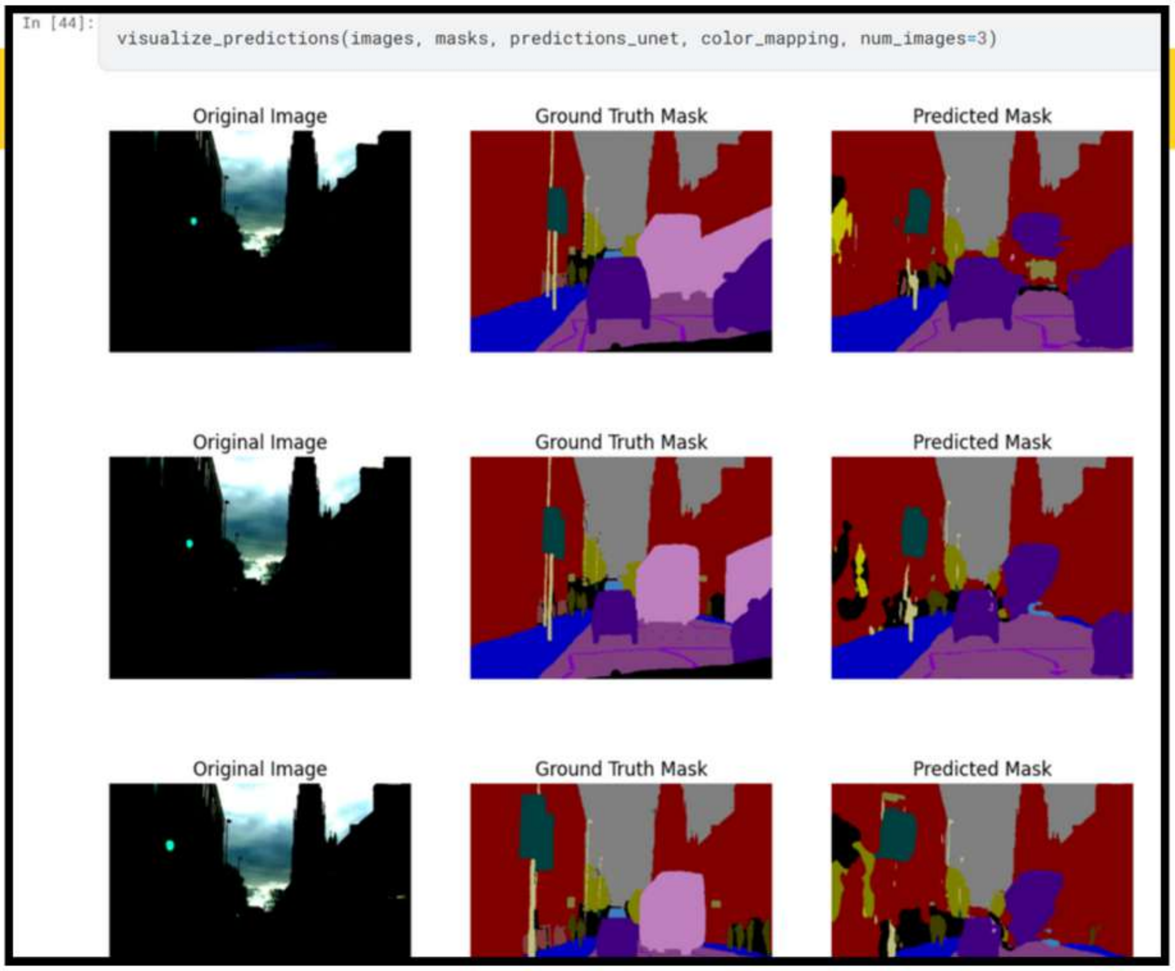
```
In [43]: visualize_predictions(images, masks, predictions_fcn, color_mapping, num_images=3)
```



Visual Prediction Analysis

U-Net Model

The U-Net model delivers the most precise and detailed segmentation results among the three. Its predicted masks show strong alignment with the ground truth, especially for both large and small objects, including thin structures like poles and signs. Boundaries are well preserved, and the segmentation appears sharp with minimal class bleeding. This visual accuracy corresponds well with U-Net's superior performance metrics, including the highest mIoU (0.4572) and pixel accuracy (0.8817). U-Net's architecture, with its skip connections, enables it to capture both global context and fine-grained details effectively.

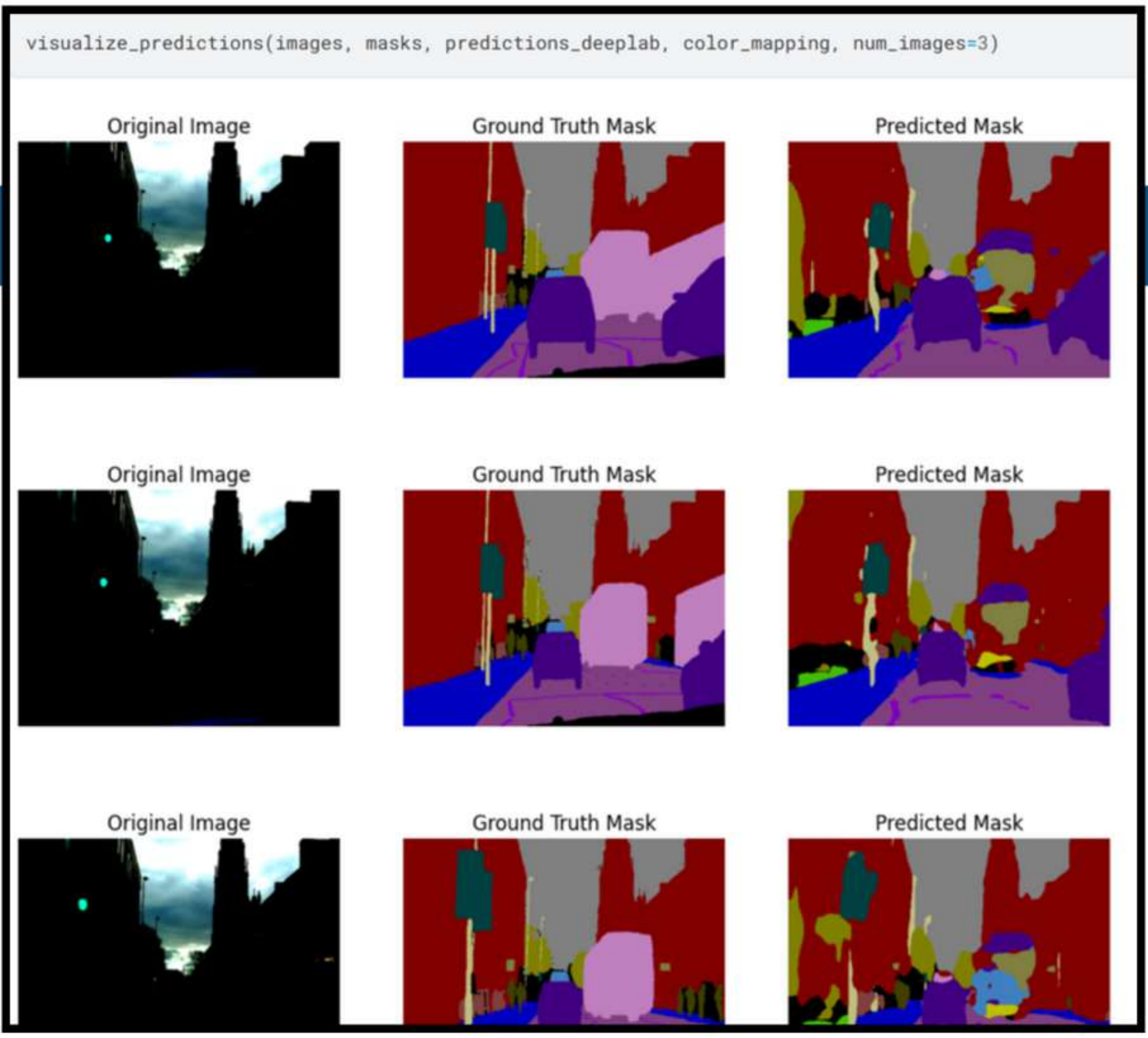


```
In [44]: visualize_predictions(images, masks, predictions_unet, color_mapping, num_images=3)
```

Visual Prediction Analysis

DeepLabv3+

DeepLabv3+ predictions demonstrate a strong grasp of the overall scene layout and perform well on large background classes. The model shows improved contextual awareness over FCN but does not match U-Net's sharpness in object boundaries. While segmentation for roads and buildings is very consistent, finer structures like poles and people tend to be fragmented or over-smoothed. Some small classes are missed or partially predicted. Nonetheless, the model displays solid performance, especially in maintaining spatial coherence, with a good balance of precision and recall



```
visualize_predictions(images, masks, predictions_deeplab, color_mapping, num_images=3)
```

Visual Prediction Analysis

Metric / Observation	FCN	U-Net	DeepLabv3+
Training Loss Trend	Steady decrease, minor overfitting	Steady decrease	Smooth decrease, slight gap
Validation Loss Trend	Slightly higher than training	Higher than training	Slightly diverges late training
Training Accuracy	≈ 80%	≈ 88.4%	≈ 89%
Validation Accuracy	≈ 80%	≈ 88.3%	≈ 89%
Average Pixel Accuracy	7.963	≈ 88%	8.906
Average mIoU (macro)	1.779	4.338	3.635
Visual Prediction Quality	Good structure, some class bleed	Sharp boundaries, fine detail preserved	Strong background separation, better large-object accuracy
Overfitting Signs	Mild	Mild (validation loss higher)	Mild
Strengths	Simple, fast convergence	High accuracy, strong spatial detail retention	Good context understanding
Weaknesses	Lower class-level precision	Large model size, slower training	Slight underperformance in mIoU

Challenges Faced

Throughout this project, several challenges were encountered, particularly during model training and evaluation. One of the main difficulties was managing class imbalance in the CamVid dataset, which led to inconsistent segmentation of small and underrepresented objects such as poles, pedestrians, and traffic signs. Although Dice and Jaccard loss functions helped mitigate this, fine-tuning the loss weights required experimentation. Additionally, due to the limited frame continuity and dataset size, overfitting became a concern, especially for deeper architectures like DeepLabv3+. Adjustments such as Group Normalization and early stopping were essential to prevent degradation in validation performance.

Beyond technical issues, we also faced significant constraints in time and computational resources. The entire project had to be completed within just few days, which limited our ability to perform extensive hyperparameter tuning, long training cycles, or model ensembling.

Furthermore, hardware limitations (such as restricted GPU memory and processing time) prevented us from running more advanced experiments or working with larger datasets. Despite these challenges, we successfully implemented, trained, and evaluated all three models and produced meaningful results under pressure.

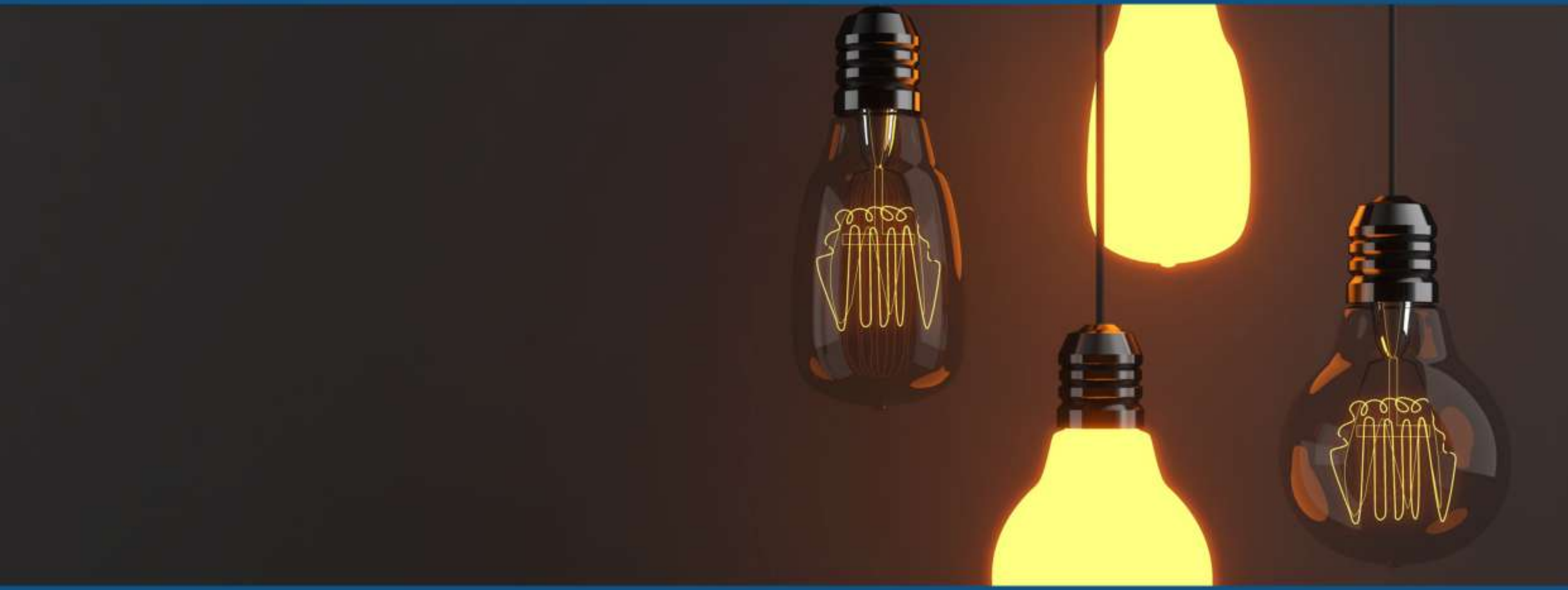
Conclusion

This project explored the effectiveness of three deep learning models—FCN, U-Net with attention, and DeepLabv3+—for semantic segmentation on urban driving scenes using the CamVid dataset.

Among these, U-Net with attention achieved the best overall performance with the highest mIoU and pixel accuracy, thanks to its ability to preserve spatial detail and context.

While DeepLabv3+ also performed well, especially on larger background regions, it showed limitations in capturing fine object boundaries. FCN, though simplest, proved reliable and fast.

Overall, this comparison highlights the strengths of encoder-decoder architectures in real-world scene segmentation, and future work may involve extending this framework to video-based segmentation or using transformer-based models for improved attention to small objects.



Summary of 3 Research Papers

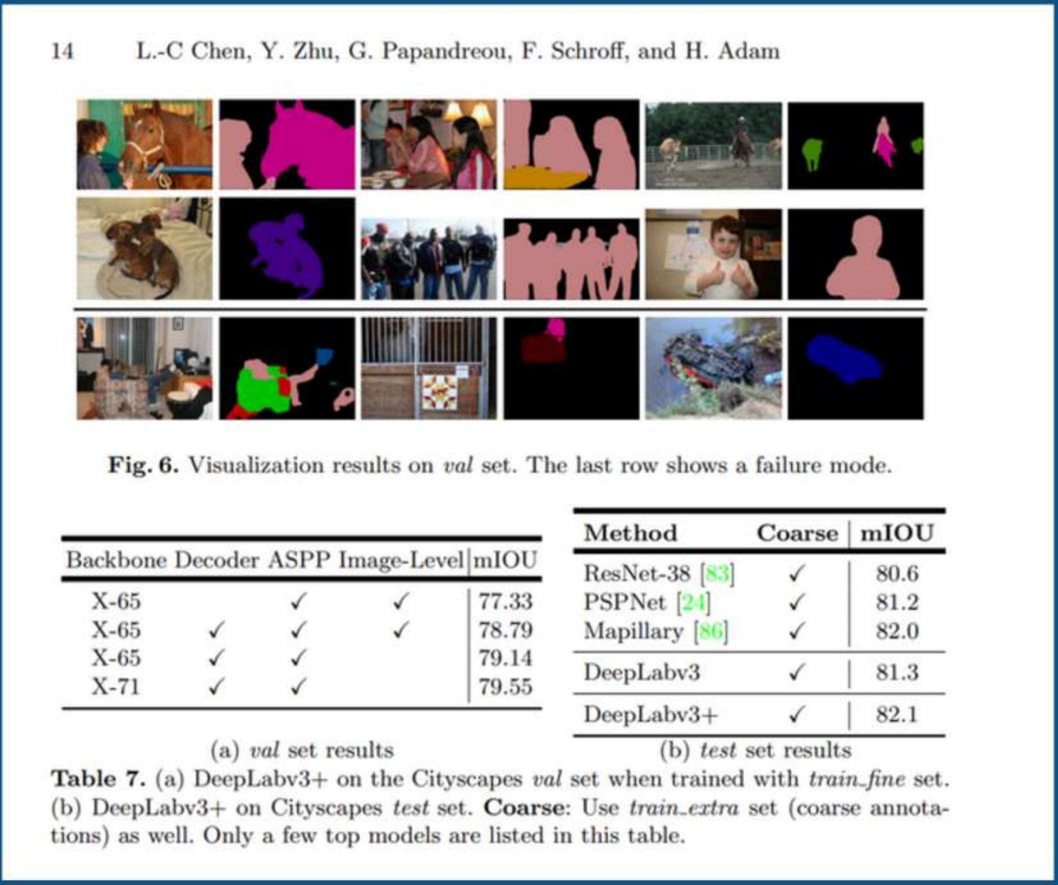
Paper 1: **"DeepLabv3+: Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation"**

Authors: Liang-Chieh Chen et al. (2018, updated versions in 2021)

Summary: This paper introduces DeepLabv3+, an advanced semantic segmentation model that builds on DeepLabv3 by incorporating a decoder module for better object boundary recovery.

The model uses atrous spatial pyramid pooling (ASPP) and depthwise separable convolutions to balance accuracy and efficiency. In this project, DeepLabv3+ was applied using a ResNet-50 backbone, and the results aligned with the paper's findings—strong contextual understanding, but slightly less precise on fine structures compared to U-Net.

Link to paper: link



14 L.-C Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam

Fig. 6. Visualization results on *val* set. The last row shows a failure mode.

Backbone	Decoder	ASPP	Image-Level	mIOU
X-65		✓	✓	77.33
X-65	✓	✓	✓	78.79
X-65	✓	✓		79.14
X-71	✓	✓		79.55

(a) *val* set results

Method	Coarse	mIOU
ResNet-38 [83]	✓	80.6
PSPNet [24]	✓	81.2
Mapillary [86]	✓	82.0
DeepLabv3	✓	81.3
DeepLabv3+	✓	82.1

(b) *test* set results

Table 7. (a) DeepLabv3+ on the Cityscapes *val* set when trained with *train_fine* set. (b) DeepLabv3+ on Cityscapes *test* set. **Coarse**: Use *train_extra* set (coarse annotations) as well. Only a few top models are listed in this table.

Summary of 3 Research Papers

Paper 2: **Semantic Segmentation of Autonomous Driving Scenes with Multi-Scale Adaptive Attention (2023)**

Authors: Danping Liu et al.

Summary: This work proposes a Multi-Scale Adaptive Attention Mechanism (MSAAM) that fuses spatial, channel, and scale attention in segmentation networks. Tested on urban driving datasets, including CamVid, it showed marked improvements over baseline CNN models. It supports your approach, particularly the use of attention mechanisms in your U-Net, and underscores the importance of combining multi-scale features and attention blocks—principles you studied in your own U-Net+Attention model.

Link to paper: link

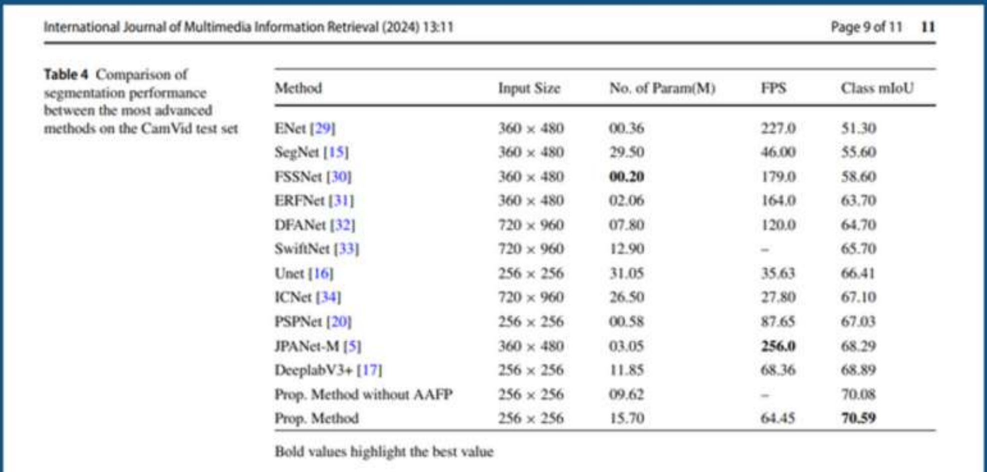


Table 4 Comparison of segmentation performance between the most advanced methods on the CamVid test set

Method	Input Size	No. of Param(M)	FPS	Class mIoU
ENet [29]	360 × 480	00.36	227.0	51.30
SegNet [15]	360 × 480	29.50	46.00	55.60
FSSNet [30]	360 × 480	**00.20**	179.0	58.60
ERFNet [31]	360 × 480	02.06	164.0	63.70
DFANet [32]	720 × 960	07.80	120.0	64.70
SwiftNet [33]	720 × 960	12.90	–	65.70
Unet [16]	256 × 256	31.05	35.63	66.41
ICNet [34]	720 × 960	26.50	27.80	67.10
PSPNet [20]	256 × 256	00.58	87.65	67.03
JPANet-M [5]	360 × 480	03.05	**256.0**	68.29
DeeplabV3+ [17]	256 × 256	11.85	68.36	68.89
Prop. Method without AAFP	256 × 256	09.62	–	70.08
Prop. Method	256 × 256	15.70	64.45	**70.59**

Bold values highlight the best value

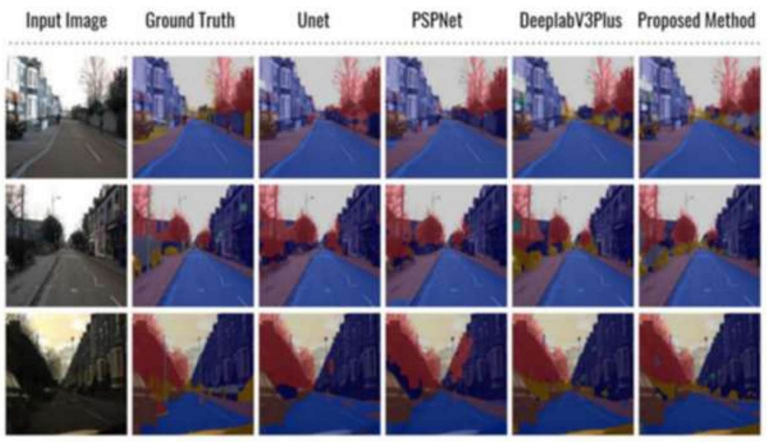


Fig. 9 Visual comparison of state-of-the-art methods with the CamVid test set

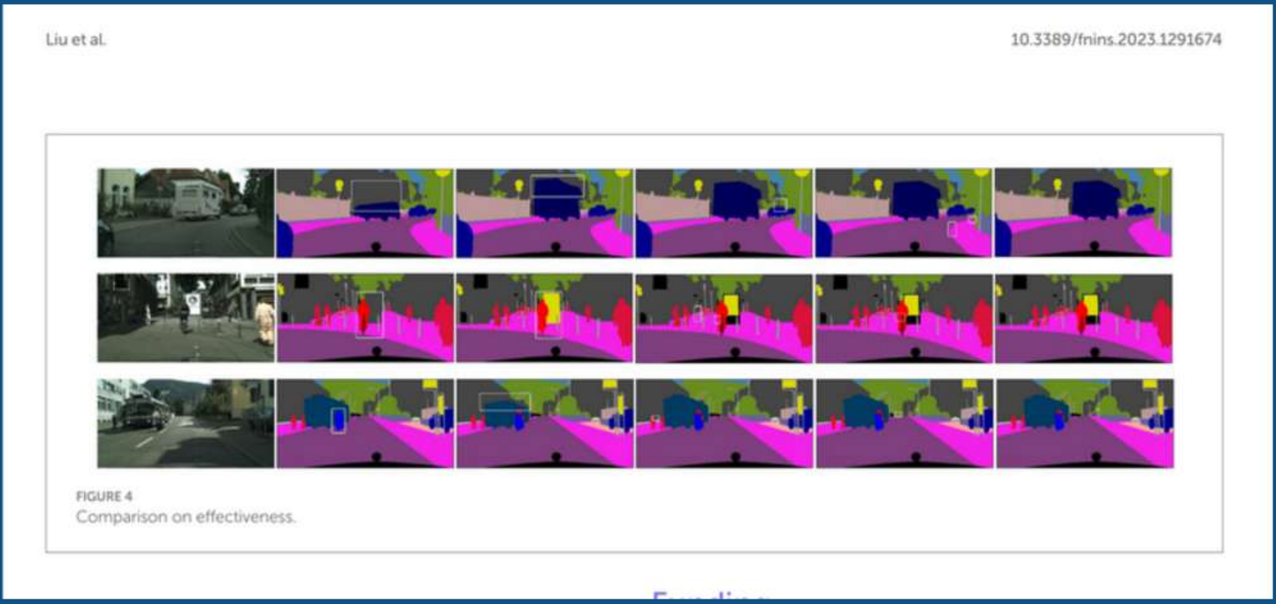
Summary of 3 Research Papers

Paper 2: **A Threefold Review on Deep Semantic Segmentation: Efficiency-oriented, Temporal and Depth-aware Design (2023)**

Authors: Felipe M. Barbosa & Fernando S. Osório

Summary: This survey explores three key dimensions essential for road-scene segmentation: real-time efficiency, video-temporal consistency, and depth awareness. It aligns directly with the proposed scope of your project — emphasizing both segmentation performance and temporal coherence — and explains why moving beyond static image segmentation (e.g., exploring video or patch-wise parallelism) would be a valuable next step.

Link to paper: link



Liu et al. 10.3389/fnins.2023.1291674

FIGURE 4
Comparison on effectiveness.

Our team

Mehul Dinesh Jain

Meryem Hanyn

Daniel Alexander Mejia Romero

Nurettin Berke Cevik

Bora Ozdamar

Ghazaleh Alizadehbirjandi