Comperative Analysis of Vision Transformer Architectures: SegFormer and Mask2Former for Semantic Segmentation in Unstructured Road Environments using the Indian Driving Dataset (IDD-Lite)

"Master of Science in Artificial Intelligence & Machine Learning"

by

Steven Gerard Mascarenhas

24044407

Supervised by Emil.I.Vassev

Department of Computer Science and Information Systems

Faculty of Science and Engineering

University of Limerick

27/8/2025

Abstract

Semantic segmentation is a critical perception task for autonomous driving, enabling a vehicle to understand its surroundings at a pixel level. While Vision Transformer (ViT) architectures have demonstrated superior performance on structured benchmarks, their efficacy remains largely untested in the chaotic and unstructured road environments prevalent in many parts of the world. This research conducts a systematic comparative analysis of two leading ViT architectures, SegFormer [5] and Mask2Former[6], on the uniquely challenging Indian Driving Dataset subset that is IDD-Lite [7] . The study evaluates the models' performance using key semantic segmentation metrics, including mean Intersection over Union (mIoU) and Pixel Accuracy, to quantify their ability to accurately classify complex scenes.In addition to evaluate multiple model variants under different training strategies, including fine-tuning from Cityscapes pre-trained weights and training from scratch. The analysis aims to identify the architectural strengths and weaknesses of each model, providing crucial insights into the suitability of transformer-based approaches for enabling safe and reliable autonomous navigation in real-world, unpredictable settings.

The results reveal a definitive conclusion: transfer learning is essential for this task, as models trained from scratch failed catastrophically, with some scoring an IoU of 0.0 on the critical 'living things' class. The comparative analysis of pre-trained models identifies a clear trade-off between accuracy and efficiency. Mask2Former with a Swin-Small backbone with Cityscapes pretrained weights fine tuned on IDD-lite emerged as the most accurate model, achieving a mean Intersection over Union (mIoU) of 0.761012, showcasing superior performance on difficult minority classes. Conversely, SegFormer with a B2 backbone with Cityscapes pretrained weights fine tuned on IDD-lite proved to be the most efficient, delivering a strong mIoU of 0.723008 at a much faster inference speed of 10.696 FPS.When comparing the SegFormer-B2 models, the version with Focal Loss did achieve a slightly higher mIoU of 0.724108 compared to the 0.723008 from the segformer with Cityscapes pretrained weights fine tuned on IDD-lite model with standard loss. However, the analysis concludes that this impact was "marginal". For the critical 'living things' class, the SegFormer-B2 with Cityscapes pretrained weights fine tuned on IDD-lite with standard loss scored an IoU of 0.554328, which was slightly higher than the 0.550969 achieved by the Focal Loss version. Therefore, your results indicate that Mask2Former (Swin-Small) with Cityscapes pretrained weights fine tuned on IDD-lite is the top model for accuracy.

These findings in the experiments, results and analysis provide crucial insights into the architectural strengths and weaknesses of each model, highlighting that the optimal choice is application-dependent, balancing the need for high accuracy with the practical constraints of real-time deployment in unpredictable global environments

Declaration

I hereby declare that I finished this dissertation without the prohibited assistance of third parties or the use of aids other than those specified; suggestions taken directly or indirectly from other sources have been identified as such. This dissertation has never before been submitted to another Irish or foreign examination board in the same or a similar format. The dissertation work was completed at the University of Limerick between May and August 2024 under the supervision of Dr. Emil Vassev.

Steven Gerard Mascarenhas

Limerick, 2024

ACKNOWLEDGEMENTS

I have received a great deal of support and guidance throughout my Master's degree at the University of Limerick and would like to take the opportunity to thank those who helped me and motivated to always improve throughout this year. First of all, I would like to express my heartfelt gratitude to Dr. Emil Vassev for his timely guidance and support throughout the duration of this dissertation. I would also like to thank him for his valuable suggestions, feedback, and generous advice, which empowered me to push my limits and successfully complete the work. It was an honor to work under his supervision.

I am also grateful to Dr Gauri Vaidya for her research methodology course, which helped me with my dissertation

Finally, I would like to thank all my family and friends, especially my mother, father, sisters and brothers for their love, encouragement and constant support to pursue my dreams. Above all, I am grateful to the Almighty God for his blessings throughout this journey

Data Source and Generative AI

Dataset Source

For this research, I used the IDD-Lite subset of the Indian Driving Dataset (IDD**)**, which contains finely annotated images of unstructured Indian road scenes. The dataset was obtained from the official website [34] . The study used the provided train and validation splits only; the **test** split (images only) was not used for training or evaluation. The dataset's original creators made the dataset mostly for academic research [7].

In the course of this dissertation, I employed a Generative AI tool, ChatGPT and gemini, for various aspects of the research and writing process. Specifically, ChatGPT was utilized in the following ways:

Analysis and Interpretation: ChatGPT was employed to support the analysis of model performance metrics, clarify complex concepts, and help articulate the implications of the results.

Writing Support: ChatGPT was used to draft and refine sections of the dissertation, including the Abstract, Introduction, background, and Conclusion. The AI helped in structuring sentences, enhancing readability, and ensuring coherence across different sections.

Technical Explanations: The tool was also used to generate explanations for certain technical processes, such as the Segformer and Mask2former Architectures .This helped in articulating complex technical details in a more understandable manner.

Creating Combined lines Graphs plots and Bar plots: I already had the implementation code of the line plots such as pixel accuracy mean Iou and bar plots of Iou per class for the models,but I used gemini to make all the line plots as a combined plot for that experiments and similarly the bar plots.

The use of ChatGPT and Gemini was supplementary to my own analysis, writing, and critical thinking, and all the content generated was carefully reviewed and edited to ensure accuracy and alignment with the research objectives.

Table of Contents

List of Figures 10
List of Tables 11
Chatper 1: Introduction 12
1.1 Motivation 12
1.2 Aims and Objectives 13
1.3 Challenges in Unstructured Road Environments 13
Chatper 2: Background 14
2.1 Overview of the Indian Driving Dataset 15
2.2 Data Collection and Frame Selection 16
2.3 Label Hierarchy and Statistics of the Indian Driving Dataset 16
2.4 Class imbalance and rare categories 17
2.5 Ambient and environmental variability 17
2.6 IDD-lite 18
2.7 SegFormer Architecture 18
2.7.1 The Hierarchical Mix Transformer (MiT) Encoder 19
2.7.2 The Lightweight All-MLP Decoder 20
2.8 Mask2Former Architecture 20
2.8.1 Backbone Feature Extractor 21
2.8.2 Pixel Decoder 21
2.8.3 Transformer Decoder 22
2.8.4 Training and Memory Efficiency 23
Chatper 3: Evaluation Metrics in Semantic Segmentation 23
Chatper 4: Commonly used Loss Functions in Semantic Segmentation 24
Chatper 5: Related Work 25

5.1 CNN-Based Architectures....25
5.2 Transformer-Based Architectures....25
5.3 Comparative Performance on the Structured Cityscapes Dataset....26
5.4 Semantic Segmentation in Unstructured Road Environments....26
5.4.1 Modified Deeplab V3+....26
5.4.2 Eff-Unet....27
5.5 Loss Functions in Semantic Segmentation....27
5.6 Research Gaps and Future Scope....28
Chatper 6: Rationale and EDA of IDD-Lite....30
6.1 Rationale for selecting Segformer and Mask2former....30
6.2 Rationale for Dataset Selection....30
6.3 Exploratory Data Analysis for IDD-Lite....30
6.3.1 Dataset Composition and Structure....30
6.3.2 Dataset File Counts....31
6.3.3 Image and Mask Dimensions....31
6.3.4 Image Resolution Analysis....32
6.3.5 Unique Class Labels and Label Verification....32
6.3.6 Pixel-wise Class Distribution....33
6.3.7 Visual Inspection of Images and Masks....34
Chatper 7: Methodology and Implementation....35
7.1 Data Preparation and Preprocessing....35
7.2 Model Initialization and Configuration....36
7.3 Implementation Details and References....36
7.4 SegFormer Experiments....36
7.5 Mask2Former Experiments....37
7.6 Training Configuration and Environment....37
7.7 Loss Functions....38
7.8 Evaluation Procedure and types of visualization....39
7.8.1 Evaluation metrics used:....39
7.8.2 Results and Visualization....40
7.8.3 Qualitative Visualizations for ground truth vs predicted semantic masks....40
7.9 Implementation of Segformer Training Pipeline....41
7.9.1 .Importing libraries for Segformer....41

7.9.2 .Data Collection: Image and Mask Paths 41
7.9.3 Data Preprocessing and Data Augmentation 42
7.9.4 Model Initialization For for Idd-lite 43
7.9.5 Computing Metrics for training and evaluation for Segformer : Mean IoU, Pixel Accuracy, Per-Class IoU 44
7.9.6 focal loss implementation for Segformer cityscapes pretrained model experiments 45
7.9.7 Training configuration 45
7.9.8 Training the model 46
7.10 Implementation of Mask2former Pipeline 46
7.10.1 Importing Libraries for Mask2former 46
7.10.2 Data Collection: Image and Mask Paths 46
7.10.3 Data Preprocessing and Data Augmentation 47
7.10.4 Model Intialization 49
7.10.5 Computing Metrics for training and evaluation for Mask2former : Mean IoU, Pixel Accuracy, Per-Class IoU 50
7.10.6 Adapting the Trainer for Mask2Former 50
7.10.7 .Training Configuration 51
7.10.8 Training the model 52
Chatper 8: Results and analysis 53
8.1 Experiments 1 : SegFormer (with backbones B0/B1/B2) : Cityscapes-Pretrained, Fine-Tuned on IDD-Lite models 53
8.1.1 Run 1: SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model 53
8.1.2 Run 2: SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model 55
8.1.3 Run 3: SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model 57
8.1.4 Combined Comparative Analysis with Qualitative Visual Analysis of Experiment 1 59
8.2 Experiment 2: SegFormer (with backbones B0/B1/B2) trained from scratch on IDD-Lite models 64
8.2.1 Run 1: SegFormer-B0 trained from scratch on IDD-Lite model 64
8.2.2 Run 2: SegFormer-B1 trained from scratch on IDD-Lite model 66
8.2.3 Run 3: SegFormer-B2 trained from scratch on IDD-Lite model 68
8.2.4 Combined Comparative Analysis with Qualitative Visual Analysis of Experiment 2 70

8.3 Experiment 3: SegFormer- (with backbones B0/B1/B2) Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function 74
8.3.1 Comparative Analysis with Qualitative Visual Analysis of Experiment 3 74
8.4 Experiment 4: mask2former (with backbones swin-tiny/swin-small) : Cityscapes-Pretrained, Fine-Tuned on IDD-Lite models 81
8.4.1 Run 1: Mask2former swin-tiny Cityscapes-pretrained, fine-Tuned on IDD-Lite model..... 81
8.4.2 Run 2: Mask2former swin-small Cityscapes-pretrained, fine-Tuned on IDD-Lite model..... 83
8.4.3 Combined Comparative Analysis with Qualitative Visual Analysis of Experiment 4 85
8.5 Experiment 5: Mask2former- (with backbones swin-tiny/swin-small) trained from scratch on IDD-Lite models. 88
8.5.1 Combined Comparative Analysis and Qualitative Visual Analysis of Experiment 5 89
Chatper 9: Conclusion..... 93
Chatper 10: References 96
Chatper 11: Appendices 100

List of Figures

Figure: 1:Label Hierarchy of Indian Driving Dataset....17
Figure: 2:Segformer Architecture....18
Figure: 3:Mask2former Architecture....21
Figure: 4: IDD-Lite dataset structure....31
Figure: 5:Image and Mask Dimension Of IDD-Lite....32
Figure: 6:Image Resolution of IDD-Lite....32
Figure: 7:Unique Class Labels and Label Verification of IDD-Lite....32
Figure: 8:Pixel wise Distribution of IDD-lite train set....33
Figure: 9:Pixel wise Distribution of IDD lite Validation set....34
Figure: 10:Example OF IDD-Lite Image and Its corresponding ground-truth semantic masks with predefined color-coded class mappings....35
Figure: 11: Training vs Validation Loss of Segformer-B0 Cityscapes Pretrained,fine-tuned on IDD-lite model....54
Figure: 12: Training vs Validation Loss of Segformer-B1 Cityscapes Pretrained,fine-tuned on IDD-lite model....56
Figure: 13:Training vs Validation Loss of Segformer-B2 Cityscapes Pretrained,fine-tuned on IDD-lite model....58
Figure: 14:Mean IOU and Pixel Accuracy per Epoch of the segformer (b0/b1/b2),Cityscapes Pretrained ,finetuned on IDD-lite Models....61
Figure: 15:Per Class IoU of Segformer (b0/b1/b2),Cityscapes Pretrained ,fine-tuned on IDD-lite models....62
Figure: 16:Visualization of Ground Truth vs Predicted Semantic maps of Segformer (b0,b1,b2) Cityscapes Pretrained fine tuned on IDD-Lite Models....63
Figure: 17:Training vs Validation of Segformer-b0 trained from scratch on IDD-lite....65
Figure: 18: Training vs Validation of Segformer-b1 trained from scratch on IDD-lite....67
Figure: 19:Training vs Validation of Segformer-b2 trained from scratch on IDD-lite....69
Figure: 20:Mean IoU and Pixel Accuracy of Segformer(b0/b1/b2) trained from scratch on IDD-lite....71
Figure: 21:Per Class IoU of Segformer (b0/b1/b2),trained from scratch on IDD-lite models72
Figure: 22:Visualization of Ground Truth vs Predicted Semantic maps of Segformer (b0,b1,b2) trained from scratch on IDD-Lite Models....73
Figure: 23: Mean IoU and Pixel Accuracy of segformer (b0,b1,b2), Cityscapes Pretrained,fine tuned on IDD-lite models with focal loss....76
Figure: 24:Per Class IoU of Segformer (b0/b1/b2),Cityscapes ptrained,fine-tuned on IDD-lite models with focal loss....77
Figure: 25: Training and Validation of Segformer (b0/b1/b2),Cityscapes ptrained,fine-tuned on IDD-lite models with focal loss....78
Figure: 26: Visualization of Ground Truth vs Predicted Semantic maps of Segformer (b0,b1,b2) Cityscpaes pretrained ,finetuned on IDD-Lite Models with focal loss....79
Figure: 27:training vs validation loss of Mask2former (swin-tiny) ,Cityscapes pretrained ,fine-tuned on IDD-lite model....82

Figure: 28: training vs validation loss of Mask2former (swin-small) ,Cityscapes pretrained ,fine-tuned on IDD-lite model 84
Figure: 29:Mean IoU and Pixel Accuracy of Mask2former (swin-tiny/swin-small) ,Cityscapes pretrained ,fine tuned on IDD-lite....86
Figure: 30:Per-Class IoU of Mask2former (swin-tiny/swin-small) Cityscapes pretrained, ,fine tuned on IDD-Lite 87
Figure: 31 Visualization of Ground Truth vs Predicted Semantic maps of Mask2former (swin-tiny/swin-small) , Cityscpaes pretrained ,finetuned on IDD-Lite Models....88
Figure: 32: Mean iou and pixel accuracy trained from scratch per epoch on IDD-lite 90
Figure: 33:Per-Class IoU of Mask2former(swin-tiny/swin-small) trained from scratch on IDD-Lite....91
Figure: 34:training vs validation of mask2former (swin-tiny/swin-small) ,trained from scratch on IDD-lite 92
Figure: 35:Visualization of Ground Truth vs Predicted Semantic maps of Mask2former (swin-tiny/swin-small) ,trained from scratch on IDD-Lite Models 93

List of Tables

Table 1:Image and Mask counts per IDD-lite dataset split....31
Table 2:Segformer B0 Cityscapes-pretrained ,fine tuned on IDD-lite Final metrics 53
Table 3:Segformer B0 Cityscapes-pretrained ,fine tuned on IDD-lite Final metrics of IoU Per Class....54
Table 4:SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics 55
Table 5:SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics of IoU Per Class 56
Table 6:SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics 57
Table 7:SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics of IoU Per Class 57
Table 8:Combined Comparative analysis of SegFormer(b0,b1,b2), Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics....60
Table 9:SegFormer-B0 trained from scratch on IDD-Lite model Final Metrics 64

Table 10: :SegFormer-B0 trained from scratch on IDD-Lite model Final Metrics of IoU per class....65
Table 11::SegFormer-B1 trained from scratch on IDD-Lite model Final Metrics 66
Table 12::SegFormer-B1 trained from scratch on IDD-Lite model Final metrics of IoU per class....67
Table 13::SegFormer-B2 trained from scratch on IDD-Lite model Final Metrics 68
Table 14::SegFormer-B2 trained from scratch on IDD-Lite model Final metrics of IoU per Class....69
Table 15:Combined Comparative analysis of SegFormer(b0,b1,b2), trained from scratch on IDD-Lite model Final Metrics 71
Table 16::Combined Comparative analysis of SegFormer(b0,b1,b2), Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss Final Metrics 75
Table 17:Mask2former Swin-tiny Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics 81
Table 18:: Mask2former Swin-tiny Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics IoU Per Class....81
Table 19:Mask2former Swin-Small Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics 83
Table 20:Mask2former Swin-tiny Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics IoU per class 84
Table 21:Combined Comparative analysis of Mask2Former(swin-tiny/swin-small), Cityscapes Pretrained,,fine tuned on IDD-Lite model Final Metrics 86
Table 22:Combined Comparative analysis of Mask2Former(swin-tiny/swin-small), trained from scratch d on IDD-Lite model Final Metrics 89

Chatper 1: Introduction

1.1 Motivation

Semantic segmentation plays a pivotal role in autonomous-driving perception by assigning a class label to every pixel, yielding dense scene understanding of the vehicle's surroundings [1],[2]. Historically, the task was dominated by convolutional neural networks as documented across comprehensive segmentation surveys [4]. More recently, Vision Transformers have driven a paradigm shift, with SegFormer and Mask2Former establishing strong state-of-the-art results in semantic segmentation benchmarks [5], [6].

Despite these advances, a validation gap persists. Prominent autonomous-driving benchmarks, most notably Cityscapes are concentrate on highly structured urban environments with well-defined infrastructure and relatively constrained scene variability and

label sets [3]. In contrast, many world regions exhibit diverse, unstructured traffic scenes that are less orderly and more ambiguous, straining models that are tuned to structured data.

The Indian Driving Dataset (IDD) was created to address precisely this gap, capturing unstructured road scenes from Indian cities with higher class diversity, different label distributions, and more ambiguity than structured environment datasets [7]. In this work, we evaluate advanced ViT architectures under these more challenging and realistic conditions to assess their robustness beyond standard structured benchmarks.

1.2 Aims and Objectives

The primary aim of this research is to conduct a systematic comparative analysis of the Vision Transformer architectures **SegFormer** and **Mask2Former** on the Indian Driving Dataset (IDD-Lite) in order to evaluate their effectiveness for semantic segmentation in unstructured road environments relevant to autonomous driving.

The Specific Objective Aims are:

- To investigate the impact, implement and fine-tune SegFormer models (B0, B1, and B2 backbones) on the IDD-Lite dataset using domain-pretrained Cityscapes weights.

- To investigate the impact, implement and train SegFormer models (B0, B1, and B2 backbones) from scratch on the IDD-Lite.

- To investigate the impact of Focal Loss for performance by fine-tuning the SegFormer models that were pre-trained on Cityscapes and fine-tuned on the IDD-Lite dataset

- To investigate ,implement and fine-tune Mask2Former models (Swin-Tiny and Swin-Small backbones) on the IDD-Lite dataset using domain-pretrained Cityscapes weights.

- To investigate , implement and train Mask2Former models (Swin-Tiny and Swin-Small backbones) from scratch on the IDD-Lite dataset .

- To quantitatively compare the performance of all models using key metrics, including Mean Intersection over Union (mIoU) and Per-Class IoU for accuracy, and Inference Speed (FPS) for computational efficiency.

- To conduct a qualitative Visual analysis of the ground truth vs predicted semantic maps of the models on the validation set, to identify their strengths and weaknesses in segmenting specific objects and complex scenes.

1.3 Challenges in Unstructured Road Environments

Deploying autonomous vehicles in unstructured environments, such as those depicted in the IDD, presents a unique set of challenges not commonly found in well organised traffic

systems. These challenges directly test the limits of semantic segmentation models. Key issues include:

- **Ambiguous Road Boundaries**: Unlike the clearly marked lanes in structured datasets, roads in the IDD often have poorly defined boundaries. The roadsides may consist of muddy terrain that is still partially drivable, and the road surface itself can be covered by dirt, making the edges very ambiguous [7].

- **High Diversity of Traffic Participants**: The variety of vehicles is much larger and includes novel classes like 'auto-rickshaws' and 'animals'. Standard categories like 'cars' also exhibit greater visual diversity due to variations in manufacturing years and significant wear and tear. Furthermore, there is a high density of motorcycles, often with multiple riders, and pedestrians who cross roads at arbitrary locations instead of designated crosswalks [7].

- **Diverse and Difficult Ambient Conditions**: The dataset features high variation in lighting, with images captured at different times of day, including dawn, dusk, and harsh mid-day sun. This results in heavy shadows, which are common in the images. The visual quality is also affected by greater variation in particulate matter due to fog, dust, or smog, leading to significant appearance changes [7].

- **Lack of Adherence to Traffic Rules**: A defining feature is that traffic participants, especially bikes and autorickshaws, are less likely to follow traffic discipline. This results in less correlation between vehicle behavior and road infrastructure like lanes or traffic lights, making scenes inherently less predictable for an algorithm to interpret[7]

Chatper 2: Background

Introduction to Image Segmentation

Image segmentation is a fundamental task in computer vision that involves partitioning an image into coherent regions, where each region corresponds to a meaningful object or area of interest. Unlike traditional image classification, which assigns a single label to an entire

image, segmentation provides pixel-level understanding of the scene, enabling higher-level reasoning about object boundaries, spatial relationships, and scene composition[2].

Generic image segmentation methods aim to partition an image into meaningful regions .They are typically grouped into three main categories:

- **Semantic Segmentation**
  Semantic segmentation assigns a class label to every pixel in the image, ensuring that each pixel is classified according to a predefined set of categories for example road, vehicle, pedestrian, sky[2] . This technique is widely used in applications such as autonomous driving, medical imaging, and scene understanding, where precise pixel-level classification is critical.

- **Instance Segmentation**
  Unlike semantic segmentation, which treats all objects of the same class as one region, instance segmentation distinguishes between different instances of the same class[2]. For example, two cars in the same image will each be segmented as separate entities, even though they belong to the same "vehicle" category. This provides finer granularity and is crucial for tasks such as object counting, tracking, and robotic manipulation .

- **Panoptic Segmentation**
  Panoptic segmentation unifies the objectives of semantic and instance segmentation. It assigns both a class label and an instance ID to every pixel, offering a comprehensive scene-level understanding[2]. By combining the strengths of the previous two approaches, panoptic segmentation enables richer contextual analysis of both "stuff"' classes for example sky, road and "thing" classes such as cars, people within the same framework .

Together, these techniques form the foundation of modern image segmentation research, supporting downstream applications in domains such as autonomous navigation, video surveillance, and image editing[2].

2.1 Overview of the Indian Driving Dataset

Autonomous driving systems rely on accurate scene understanding.. Publicly available benchmarks such as Cityscapes and KITTI have enabled rapid progress, but they primarily reflect structured driving environments**,** These include roads with clear lane markings, orderly traffic, a few categories of participants, and fairly uniform backgrounds. This is quite different from the situations faced in many areas of Asia, Africa, and South America.

Indian roads in particular feature ambiguous lane boundaries, muddy roads, mixed vehicle types, animals, billboards and frequent jaywalkers. To address this gap, Varma et al.

introduced the Indian Driving Dataset (IDD), collecting 10,004 finely annotated images from 182 Indian driving sequences with a rich label set of 34 classes [7]. A large-scale dataset for road scene understanding in unstructured environments [7] . It supports research on semantic segmentation, instance segmentation, and object detection [7].This expanded label set includes new road scenes for example, drivable off-road areas and reflects much greater within-class variability than structured-road datasets [7]. Varma et al [7] also reports that state-of-the-art segmentation models trained on structured benchmarks like the Cityscapes dataset suffer a marked drop in accuracy when applied to IDD, underscoring the severity of the domain shift introduced by unstructured conditions [7].

2.2 **Data Collection and Frame Selection**

The data was collected using a forward-facing stereo camera mounted on a vehicle, driven through the cities of Hyderabad and Bangalore and their outskirts in India. These locations were specifically chosen to include a rich mixture of urban, semi-urban, and rural settings, featuring highways, single-lane roads, and dense city streets.[7]

A total of 182 distinct drive sequences were recorded to ensure a diverse range of environmental and traffic conditions.

As noted by the dataset's authors [7], **Varma et al,** these environments are highly unstructured due to several factors:

1. Rapid urban development leading to frequent construction zones,
2. Poorly defined road boundaries,
3. A high density of pedestrians and two-wheelers,
4. A wide variety of vehicle models with significant variations in appearance .

2.3 Label Hierarchy and Statistics of the Indian Driving Dataset

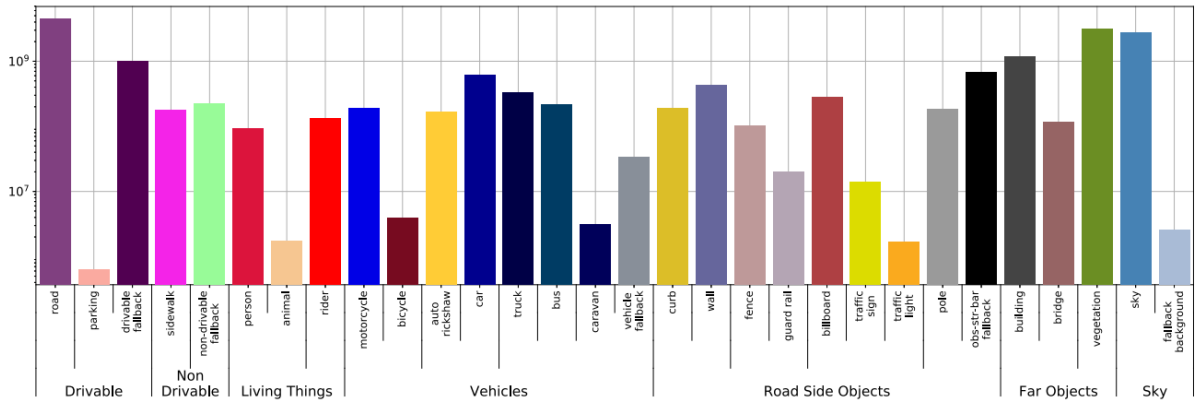


Figure: 1:Label Hierarchy of Indian Driving Dataset

The Indian Driving Dataset (IDD) employs a four-level label hierarchy The dataset introduces several unique labels which are not present from European benchmark datasets which includes autorickshaw, animal, billboard, curb/median etc, and three classes indicating safe drivable, unsafe drivable and non-drivable flat surfaces [7]. They designed a four-level hierarchy:

- **Level 1**: 7 coarse classes that merge closely related categories for example all drivable surfaces, all vehicles. Level 1 classes correspond to the categories Drivable, Non-drivable, Living-things, Vehicles, Roadside objects, Far objects, and Sky as shown in Figure: 1.
- **Level 2**: 16 classes, including separate categories for specific vehicle types or background structures.
- **Level 3**: 26 classes.
- **Level 4**: 30 classes corresponds to finer annotations

The hierarchy is designed by grouping ambiguous labels such that lower levels such as level 1 have less ambiguity and higher levels such as level 4 capture finer distinctions [7].

2.4 Class imbalance and rare categories

Although IDD increases the number of pixels for many classes, it still suffers from severe class imbalance at the fine level. Labels like animal, traffic light, parking, trailer and rail track are extremely rare [7]. These labels are grouped into fallback classes at lower hierarchy levels to reduce ambiguity.

2.5 Ambient and environmental variability

IDD captures wider ranges of weather and lighting than structured datasets Scenes include midday, dawn and dusk[7]. Road conditions have muddy surfaces. The dataset therefore serves as a test-bed **for** robust segmentation methods and domain generalisation**.**

Furthermore, state-of-the-art models trained on Cityscapes exhibit substantially lower accuracies when evaluated on IDD [7].

2.6 IDD-lite

To facilitate research on this challenging setting, a condensed IDD-Lite variant was created for quick experimentation on resource-constrained setups. IDD-Lite captures the essence of unstructured traffic scene images that feature unclear or faded road boundaries. It includes a mix of vehicle types, such as cars, two-wheelers, and auto-rickshaws. The scenes often show frequent occlusions and unpredictable traffic,but maps the fine-grained IDD labels into seven broad categories that is driving area, non-driving area, living beings, cars, roadside items, distant objects, and sky at level 1 in Figure: 1 [7].

2.7 SegFormer Architecture

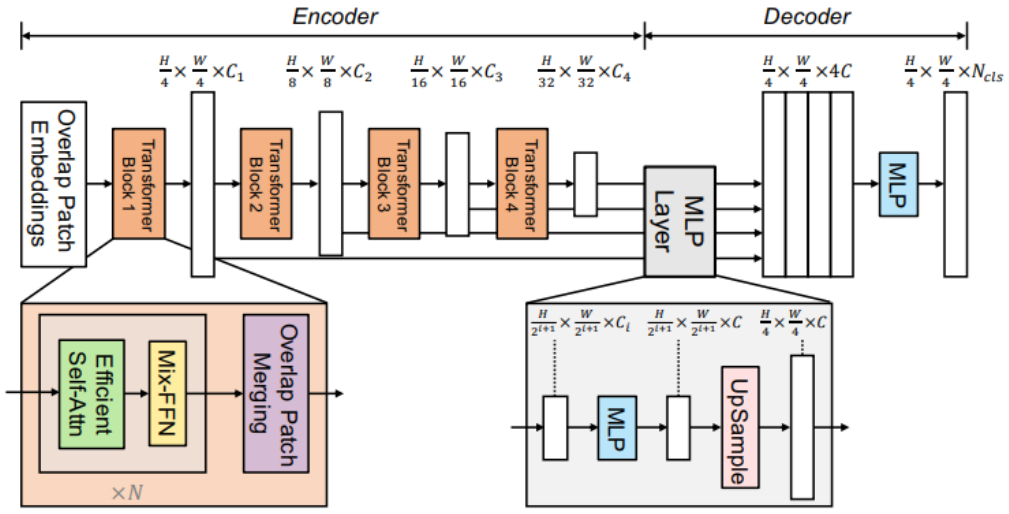


Figure: 2:Segformer Architecture

SegFormer is a semantic segmentation framework designed with simplicity, efficiency, and power as its core principles. It unifies a hierarchical Vision Transformer encoder with a remarkably lightweight Multilayer Perceptron decoder. A series of models, from MiT-B0 which is the lightweight encoder to MiT-B5 which is the largest largest encoder, were designed to provide a trade-off between performance and computational cost. This design intentionally avoids the complex, hand-crafted modules common in many CNN-based segmentation models, demonstrating that a sufficiently powerful encoder can enable a highly simplified decoder, leading to a more efficient overall architecture[5].

2.7.1 The Hierarchical Mix Transformer (MiT) Encoder

- **Hierarchical Feature Representation:** Unlike the standard ViT, which produces a single, low-resolution feature map, the MiT encoder is designed to generate CNN-like multi-scale features. It processes an image to produce feature maps at 1/4, 1/8, 1/16, and 1/32 of the original resolution ,providing both high resolution, fine grained features from early stages and low-resolution, coarse-grained features from later stages, which is crucial for effective semantic segmentation [5].

- **Overlapped Patch Merging:** To create the feature hierarchy, SegFormer employs a patch merging process. However, instead of the non-overlapping patches used in the original ViT, which can disrupt local continuity, SegFormer uses an overlapping patch merging strategy. This helps to preserve the local information and relationships between adjacent patches, leading to more coherent feature extraction [5].

- **Efficient Self-Attention:** A major bottleneck in Transformers is the quadratic computational complexity ($O(N^2)$) of the self-attention mechanism with respect to the number of patches (N). To mitigate this, SegFormer employs a sequence reduction technique. Before the attention operation, the key and value sequences are spatially reduced by a reduction ratio R. This lowers the complexity from $O(N^2)$ to ($O(\frac{N^2}{R})$), making the self-attention mechanism significantly more efficient and scalable to high-resolution images[5].

- **Positional-Encoding-Free Design:** A critical innovation in SegFormer is the elimination of explicit positional encodings. Standard ViTs rely on fixed-size positional encodings, which must be interpolated when the test image resolution differs from the training resolution, often leading to a drop in accuracy.

  SegFormer replaces this with a Mix-FFN layer, which incorporates a 3x3 depth-wise convolution within the feed-forward network of the Transformer block.

  This simple addition allows the network to learn implicit positional information from the data itself, making the encoder robust and adaptable to arbitrary test resolutions without performance degradation[5].

2.7.2 The Lightweight All-MLP Decoder

The SegFormer decoder consists solely of simple MLP layers, avoiding the complex modules like Atrous Spatial Pyramid Pooling that are common in CNN-based decoders.

The decoding process involves four steps [5]:

1. The multi-level features from the four stages of the MiT encoder are passed through an MLP layer to unify their channel dimensions.
2. All feature maps are up-sampled to 1/4 of the original image resolution.
3. The up-sampled feature maps are concatenated.
4. An MLP layer fuses the concatenated features
5. Finally , another MLP layer takes the fused feature to predict the segmentation mask M with a $\frac{H}{4} \times \frac{W}{4}$ × Ncls resolution, where Ncls is the number of categories.

The effectiveness of this simple design is explained by an analysis of the encoder's Effective Receptive Field . The lower stages of the MiT encoder produce local attention patterns similar to convolutions, capturing fine details. In contrast, the highest stage produces highly non-local attention, effectively capturing global context. The simple MLP decoder is sufficient because its task is merely to aggregate and fuse these already powerful, multi-level features. This elegant design shifts the model's complexity almost entirely to the encoder, resulting in a highly efficient and effective segmentation framework [5]

2.8 Mask2Former Architecture

Mask2Former (Masked-attention Mask Transformer) is a universal image segmentation model. Instead of using task-specific designs for panoptic, instance or semantic segmentation, Mask2Former uses a single architecture that generalises across all segmentation tasks. The model improves on earlier universal architectures like MaskFormer by combining a deformable-attention pixel decoder with a Transformer decoder that uses a masked attention

mechanism [6]. The key components of the architecture and their roles are described below.

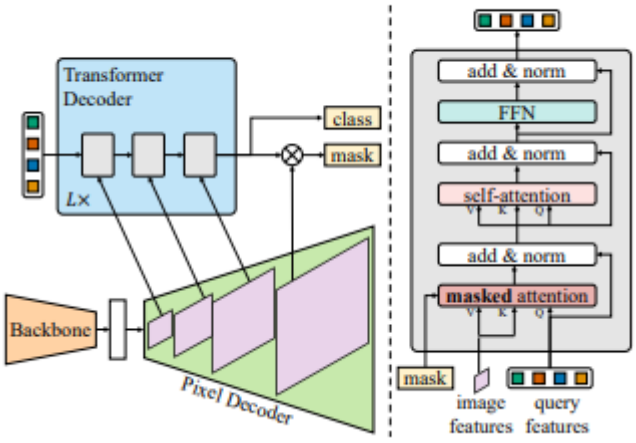


Figure: 3:Mask2former Architecture

2.8.1 Backbone Feature Extractor

1. **Purpose**: The backbone extracts a hierarchy of features from the input image at multiple spatial resolutions. A CNN such as ResNet or a Vision Transformer such as Swin Transformer can be used as the backbone.

   Given an input image, the backbone produces a pyramid of feature maps at progressively lower resolutions. These multi-scale features capture both fine and coarse image structure [6].

2. **Flexibility**: Mask2Former is designed so that different backbones can be used without modifying the rest of the architecture. Using a transformer-based backbone for example Swin, improves performance on high-resolution tasks.

2.8.2 Pixel Decoder

The pixel decoder upsamples the coarse features from the backbone into a high-resolution representation suitable for mask prediction. Mask2Former adopts an advanced multi-scale deformable attention decoder instead of the Feature Pyramid Network used in MaskFormer:

1. **Multi-Scale Deformable Attention** Layers**:** Six layers of multi-scale deformable attention operate on feature maps at different scales at 1/32, 1/16 and 1/8 of the input size. These layers effectively model long-range dependencies and allow the decoder to focus on salient regions [6]

2. **Lateral Upsampling**: A lateral upsampling layer with a skip connection produces a final high-resolution feature map from the coarse multi-scale deformable attention features [6]. This map serves as the per-pixel embedding used to decode masks.

3. **High-resolution Strategy**: Instead of feeding all scales to every Transformer decoder layer, Mask2Former uses an efficient multi-scale strategy: successive layers of the Transformer decoder receive different scales from the pixel decoder in a round-robin fashion , allowing high-resolution features to be used without excessive computation [6].

2.8.3 Transformer Decoder

The Transformer decoder processes **object queries** and the per-pixel features to produce mask and class predictions:

1. **Object Queries**: A fixed number of learnable query embeddings (for example 100) are initialised. Each query is intended to represent a potential segment in the image. These queries are updated across layers of the decoder through self-attention and cross-attention [6]

2. **Masked Attention Mechanism**:

    - Standard cross-attention lets each query attend to the entire feature map, which is computationally expensive and may distract the model with irrelevant background pixels.
    - Therefore, Mask2Former uses masked attention, cross-attention is restricted to the foreground region of the predicted mask from the previous layer. This focuses computation on localised regions and improves both efficiency and performance [6].
    - The attention mask is derived by thresholding the mask predicted by the previous decoder layer and resizing it to the current feature map resolution [6].

3. **Optimisation Improvements**:

    - The order of self-attention and cross-attention is swapped: cross-attention (masked attention) is applied first, followed by self-attention, which improves information flow and convergence [6].
    - The initial query features themselves are learnable and are directly supervised to function like region proposal [6].
    - Dropout is removed from the decoder because it was found to decrease performance [6].

4. **Outputs**: The decoder produces unnormalised *logits* for both the class and the mask associated with each query. The class-queries logits have shape (batch_size, num_queries, num_labels + 1) and, after applying a softmax, represent probabilities over the possible classes including a null class. The mask-queries logits have shape (batch_size, num_queries, height, width) and, after applying a sigmoid, give per-pixel probabilities for the segmenthuggingface.co. These logits are post-processed to generate the final panoptic, instance or semantic segmentation maps [6].

2.8.4 Training and Memory Efficiency

To make training feasible on high-resolution images, the authors introduce an efficient mask-loss strategy. Instead of computing the mask loss on every pixel, they uniformly sample a fixed number of points (112×112) for each mask and compute the loss on those points only. This reduces memory consumption by roughly a factor of three [6]. Despite the sampling, the model still achieves state-of-the-art accuracy on multiple datasets.

Overall,Mask2Former's architecture unifies panoptic, instance and semantic segmentation. It comprises a flexible backbone, a powerful pixel decoder based on deformable attention and lateral upsampling, and a Transformer decoder with a masked attention mechanism and learnable queries. By focusing attention within predicted mask regions, using multi-scale features efficiently, and optimising the decoder design, Mask2Former achieves high accuracy and fast convergence while remaining memory-efficient. These innovations enable a single model to outperform specialised architectures across different segmentation tasks

Chatper 3: **Evaluation Metrics in Semantic Segmentation**

- **Pixel Accuracy (PA)**: This is a very simple metric that calculates the ratio between the amount of properly classified pixels and the total number of pixels[14].

  mean Pixel Accuracy (mPA), which computes this ratio on a per-class basis and then averages the results[14].

- **Intersection over Union (IoU)**: Also known as the Jaccard Index, IoU is a statistic used to compare the similarity between the predicted segmentation and the ground truth. It is calculated as the ratio of the intersection of the pixel results with the ground truth to their union [14].

- **Mean Intersection over Union (mIoU)**: This is a straightforward and widely used extension of IoU. It is the class-averaged Intersection over Union, meaning the IoU is calculated for each class and then averaged[14].

- **Frequency-Weighted IoU (FwIoU)**: This is an improved version of mIoU that weighs the importance of each class based on how often it appears in the ground truth. The weight for each class is determined by the total number of pixels labeled as that class[14].

Chatper 4: Commonly used Loss Functions in Semantic Segmentation

- **Cross-Entropy (CE) Loss:** This is the standard classification loss used in semantic segmentation to quantify the divergence between the predicted class probability distribution and the ground truth for each pixel. It is widely adopted because it encourages the network to assign high probability to the correct class label at every pixel [8]

- **Focal Loss:** Focal loss modifies the traditional cross-entropy objective to address severe class imbalance by down-weighting well-classified examples. By reducing the contribution of easy, correctly classified pixels, it enables the network to concentrate on learning from the harder, misclassified cases[8].

- **Dice Loss:** Originating from the Dice similarity coefficient, Dice loss directly optimizes the overlap between predicted and ground-truth segmentation masks. It is particularly advantageous when segmenting objects that occupy a small portion of the image or when class imbalance is significant, as it prioritizes matching the shape and size of the target regions [8]

- **IoU (Jaccard) Loss:** IoU loss is derived from the Intersection over Union metric. Its purpose is to maximize the ratio of the intersecting area of the predicted and ground-truth masks to the union of these areas, thereby improving overall region agreement in segmentation tasks [8].

- **Tversky Loss:** The Tversky loss generalizes Dice and IoU losses by introducing two weighting parameters that independently adjust the penalties for false positives and false

negatives. This flexibility allows researchers to tailor the loss function to emphasise the more critical type of classification error in a given application [8].

- **Combo Loss:** A combo loss combines Dice loss with a weighted cross-entropy term. By doing so, it leverages the smooth gradient properties of cross-entropy while simultaneously optimising overlap through the Dice component, which is helpful in scenarios with class imbalance [8]

Chatper 5: Related Work

5.1 CNN-Based Architectures

Convolutional neural networks (CNNs) formed the foundation of early semantic segmentation, using fully convolutional networks to extract hierarchical features from local receptive fields[9]. The Pyramid Scene Parsing Network (PSPNet) introduced a pyramid pooling module that pools features at several spatial scales and fuses them to provide a global context prior[10]. PSPNet uses a pre-trained ResNet-101 backbone with dilated convolutions and, when trained on both the fine and coarse annotations of Cityscapes, achieves about 80.2 % mean IoU on the test set[10].

The DeepLab family expands the receptive field by replacing standard convolutions with atrous (dilated) convolutions and combining several dilation rates in an Atrous Spatial Pyramid Pooling (ASPP) module [11].DeepLabv3+ extends DeepLabv3 by adding a simple decoder to recover object boundaries[11] and uses a modified Xception backbon**e** that is explicitly pre-trained on the ImageNet-1K dataset[11]. When trained and evaluated with coarse annotations, DeepLabv3+ attains 82.1 % mean IoU on Cityscapes[11].

Despite these advances, CNN-based segmentation still relies on local convolutions and extra modules for global context. This design means that convolutional networks struggle to capture long-range dependencies without additional context mechanisms [10]**,** motivating research into architectures that better model global relationships.

5.2 Transformer-Based Architectures

Vision Transformers (ViTs) have emerged as a powerful alternative by introducing global self-attention, enabling direct modeling of long-range dependencies without dedicated context modules. SegFormer [5] employs a hierarchical encoders such as Mit-b4 and Mit-b5 with lightweight MLP decoder, using ImageNet-1K-pretrained weights,where Mit-b4 achieves 83.9% mIoU at multiscale inference and 82.3% on singlescale,while Mit-b5 achieves 84.0 % mIoU at multiscale inference and 82.4% on singlescale on Cityscapes.

Mask2Former [6] reframes segmentation as a mask classification problem, using masked cross-attention within a Transformer decoder and backbones such as Swin-base and Swin-large pretrained on ImageNet-22K with Swin-base achieving 83.3% mIoU on singlescale while with multi-scale inference, it attains 84.5 % mIoU, Swin-Large yields 83.3% mIoU as well on singlescale inference and 84.3% mIoU at multiscale on Cityscapes structured dataset.

5.3 Comparative Performance on the Structured Cityscapes Dataset

Across the Cityscapes benchmark, Vision Transformer–based models show a consistent 2 to 4 percentage point improvement over the strongest CNN baselines. This performance gain is largely attributed to their ability to capture both global and local context natively, leverage hierarchical multi-scale features, and simplify decoder design, leading to improved segmentation accuracy in complex urban scenes.

5.4 Semantic Segmentation in Unstructured Road Environments

5.4.1 Modified Deeplab V3+

The modified DeepLabV3+ architecture was specifically developed for semantic segmentation in unstructured driving environments, with a primary focus on dense traffic scenarios prevalent in the India Driving Dataset

Key modifications include:

- A dilated Xception backbone as the encoder, which uses dilated convolutional layers to maintain receptive field while reducing parameters compared to other DeepLabV3+ implementations. It also incorporates depthwise separable convolutions[12].

- A modified Atrous Spatial Pyramid Pooling module employing lower atrous rates of 4, 8, and 12 (instead of 6, 12, 18) to better capture multi-scale context of dense and varied objects. This module also uses depthwise separable convolutions[12].

• The decoder upsamples the Atrous Spatial Pyramid Pooling output and combines it with low-level encoder features to improve spatial detail[12].

This configuration achieved 68.41% mIoU for Level 3 labels on the IDD test set, outperforming the original DeepLabV3+ with dilated Xception 65.7% mIoU and processing 2.1 frames per second, demonstrating its suitability for real-time applications[12].

5.4.2 Eff-Unet

Eff-UNet is a novel architecture designed for semantic segmentation in unstructured environments, specifically on the India Driving Lite Dataset (IDD-Lite). It combines:

- An EfficientNet encoder for feature extraction, chosen for its compound scaling method that uniformly scales network depth, width, and resolution for improved performance. Its basic building block is the mobile inverted bottleneck convolution with squeeze-and-excitation optimization[13].

- A UNet-style decoder that reconstructs fine-grained segmentation maps through upconvolutions and concatenation with corresponding feature maps from the contracting path. These are widely known as skip connections, which combine high-level contextual features with low-level spatial information for precise segmentation [13].

The EfficientNet-B7 encoder was initialised with ImageNet-pretrained weights, providing a strong starting point for feature representation. These weights were fine-tuned end-to-end on IDD-Lite, enabling domain adaptation to the unstructured road scenes characteristic of the dataset[13].

Eff-UNet achieved 62.76% mean Intersection over Union on the IDD-Lite test dataset, securing first prize in the 2019 IDD-Lite segmentation challenge[13]. This was accomplished while maintaining high efficiency, as EfficientNet is noted for being significantly smaller and faster than other Convolutional Neural Networks

5.5 Loss Functions in Semantic Segmentation

The selection of loss functions is pivotal in the development and performance of deep learning models for semantic image segmentation. A comprehensive survey by Azad et al [8]. evaluated 25 loss functions, categorising them and conducting unbiased experiments to guide optimal selection. This study specifically assessed the impact of six distinct loss functions Dice Loss, Focal Loss, Tversky Loss, Focal Tversky Loss, Jaccard (IoU) Loss, and Lovász-Softmax Loss on two prominent architectures: UNet which is CNN-based and TransUNet which is Vision Transformer-based [8].

All models were trained using a compound loss:

$L_{total} = 0.5 * L_{var} + 0.5 * L_{CE}$.,

where Lvar is the varying segmentation loss and Lce is Cross-Entropy Loss [8].

Key Findings:

Medical Image Segmentation on the Synapse Dataset

- For the TransUNet model, the Jaccard (IoU) Loss proved most effective, achieving 80.85% Dice Score Coefficient. This notably outperformed the worst-performing Focal Loss by 7.06% Dice Score Coefficient points on this dataset[8].

- Overlap-aware losses, specifically Jaccard and Tversky Losses, demonstrated significant advantages by producing sharper segmentation boundaries and better capturing small organs such as the pancreas and gallbladder, where Dice Loss and Focal Loss showed reduced performance. This suggests their effectiveness for handling fine details and potentially addressing class imbalances in medical imaging scenarios[8].

- The experiments concluded that for the Synapse dataset, the choice of loss function was more data-dependent than model-dependent, as no noticeable difference in optimal loss function was observed between the UNet and TransUNet architectures in this domain [8].

Urban Scene Segmentation on the Cityscapes Dataset

- The conventional UNet model displayed low sensitivity to loss function choice, with only approximately 1.00% mIoU (mean Intersection over Union) variation across the different loss functions evaluated. For UNet, Tversky Loss 63.52% mIoU and Dice Loss 63.28% mIoU were the best performers [8] .

- Conversely, the TransUNet model exhibited higher sensitivity, showing a 2.55% mIoU discrepancy between its best performer, with Jaccard Loss 58.97% mIoU, and its worst performer, Dice Loss 56.42% mIoU . Jaccard Loss consistently emerged as the best choice for TransUNet on this dataset [8].

These results show that the choice of loss function matters depending on the model architecture which means that the best loss function for a CNN like UNet might not work well for a Vision Transformer model like TransUNet. The need for thorough testing with different loss functions for difficult segmentation tasks, especially when using Vision Transformer models is important.

5.6 Research Gaps and Future Scope

This study is positioned to address critical research gaps and outline future directions at the intersection of advanced computer vision architectures and the challenges of autonomous driving in unstructured environments.

1. **Performance Benchmark in Unstructured Domains:** While the superiority of Vision Transformers like SegFormer and Mask2Former is well-established on structured datasets for example Cityscapes, there is a significant lack of systematic, comparative benchmarking on chaotic, real-world datasets such as the Indian Driving dataset IDD-Lite.

2. **Lack of Domain-Specific Pre-training:** Most approaches fine-tune models with general-purpose ImageNet weights directly on IDD-Lite. A key gap exists in understanding the impact of intermediate, domain-specific pre-training. It is an open question whether fine-tuning on a large-scale structured driving dataset like Cityscapes before fine-tuning on IDD-Lite would improve generalization and performance on unstructured scenes.

3. **Performance of Training from Scratch:** Since most Vision Transformers and CNNs leverage pre-trained weights from large-scale datasets, a fundamental gap exists in understanding how a model trained entirely from scratch performs on a smaller, specialized dataset like IDD-Lite. This investigation will quantify the impact and necessity of pre-training in this specific domain.

4. **Comparative Analysis of Encoders on IDD-Lite:** A gap exists in understanding the specific performance contributions of the different underlying encoders ,SegFormer's MiT encoders that is B0 to B5 versus. Mask2Former's Swin Transformer encoders that is swin tiny to swin large in the context of unstructured domains. It is unknown how the architectural differences between these backbones affect performance on the IDD-Lite dataset.

5. **Effective Loss Function for Vision Transformers on IDD-Lite:** The literature shows that the optimal loss function for Vision Transformers is highly data-dependent. While certain losses are effective on structured datasets, a significant research gap exists in identifying the most effective loss function for example Dice, Focal or Tversky loss for training models like SegFormer and Mask2Former on the unique class imbalances and visual complexities of the IDD-Lite dataset.

6. **Scaling to the Full IDD Dataset:** This study uses the IDD-Lite dataset for efficient benchmarking. A direct future step is to scale the experiments to the full Indian Driving Dataset (IDD), which features a much larger set of images and a more complex, fine-grained label hierarchy having 34 classes. This would test the scalability and performance of the superior architecture on a significantly more challenging task.

Chatper 6: Rationale and EDA of IDD-Lite

6.1 Rationale for selecting Segformer and Mask2former

The detailed architectural principles of SegFormer and Mask2Former have been outlined in the Background chapter. In the context of methodology, these models were selected because they represent two state-of-the-art Vision Transformer approaches that align with the research objectives. SegFormer was chosen for its lightweight, efficiency-oriented design optimised for real-time inference[5], while Mask2Former was selected for its universal segmentation framework and strong benchmark performance across multiple segmentation tasks[6]. Evaluating both on IDD-Lite enables a systematic comparison between efficiency and segmentation accuracy between vision transformer architectures in unstructured road environments.

6.2 Rationale for Dataset Selection

To robustly evaluate model performance in challenging and unpredictable conditions, the Indian Driving Dataset IDD-Lite was selected. Standard autonomous driving datasets often feature well-structured road environments, which do not fully represent the complexity of real-world scenarios. The IDD-Lite dataset is uniquely suited for this study as it contains scenes with inconsistent lane markings, and diverse vehicle classes characteristic of the unstructured road environments[7] central to this research

6.3 Exploratory Data Analysis for IDD-Lite

Before training the semantic segmentation models, an exploratory data analysis of the IDD-Lite dataset was conducted to verify dataset integrity, confirm image and semantic correspondence and to highlight potential challenges such as class imbalance. This analysis ensured that the dataset was structurally consistent and suitable for semantic segmentation experiments

6.3.1 Dataset Composition and Structure

The IDD-Lite dataset is organised into three splits train,validation, and test. The training and validation splits contained paired RGB images and their corresponding semantic masks, stored in separate directories such as (leftImg8bit/[train, val, test]) for images and gtFine/[train , val] for masks as shown in Figure: 4

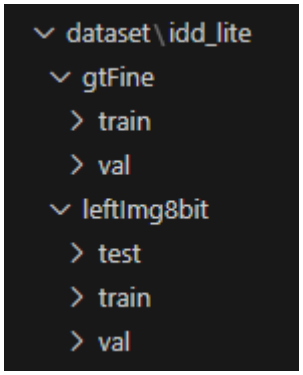


Figure: 4: IDD-Lite dataset structure

In contrast, the test split only contains RGB images without corresponding semantic masks. Since semantic masks are essential for training and evaluating semantic segmentation models, the test split was excluded from the experimental workflow.

Accordingly, all subsequent analysis and model development were performed using the training split and the validation split .

6.3.2 Dataset File Counts

To ensure dataset integrity, the number of images and semantic masks in each split was verified. The counts are summarised in Table 1.

Table 4.1: Image and mask counts per dataset split

Split	Images	Masks
Train	1,403	1,403
Validation	203	203
Test	406	0

Table 1:Image and Mask counts per IDD-lite dataset split

6.3.3 Image and Mask Dimensions

To ensure structural consistency, the spatial dimensions of images and their corresponding semantic masks were checked in both the training and validation splits. The analysis showed

that all images had a resolution of 320 × 227 pixels with three channels (RGB). The corresponding masks also had the same resolution but with a single channel (grayscale).

The results, summarized below, confirm that there were no mismatches between images and masks.This means the dataset is properly set up for pixel-wise semantic segmentation.

```
TRAIN: Unique image shapes: {(320, 227, 3)}, Unique mask shapes: {(320, 227, 1)}
VAL: Unique image shapes: {(320, 227, 3)}, Unique mask shapes: {(320, 227, 1)}
```

Figure: 5:Image and Mask Dimension Of IDD-Lite

This verification step confirmed that all data samples were structurally valid and ready for training. There were no inconsistencies that could disrupt the segmentation process.

6.3.4 Image Resolution Analysis

Image resolutions were analysed to check for inconsistencies. Across the training and validation splits, all 1,607 images were found to have uniform dimensions of 320 × 227 pixels, with no outliers detected. This consistency simplifies preprocessing since resizing operations can be applied uniformly across the dataset, thereby facilitating efficient batching during training.

```
Unique image resolutions (width x height) and their counts:
320x227 : 1607 images
```

Figure: 6:Image Resolution of IDD-Lite

6.3.5 Unique Class Labels and Label Verification

To make sure the semantic masks in the dataset were valid and consistent, we scanned all masks in the train and validation splits to extract the unique pixel values. This check confirms that only the expected label IDs are present and that there are no issues or corrupted values.

```
TRAIN unique IDs: [0, 1, 2, 3, 4, 5, 6, 255]
VAL unique IDs: [0, 1, 2, 3, 4, 5, 6, 255]
```

Figure: 7:Unique Class Labels and Label Verification of IDD-Lite

These results align with the official IDD-Lite specification, which defines seven semantic classes from 0 to 6 and uses the 255 as the ignore label. The ignore label is not taken in training or evaluation, ensuring the model only learns from valid semantic categories.

This step confirms that the dataset is complete, clean, and consistent with its published label hierarchy, and provides a reliable foundation for preprocessing and model training.

6.3.6 Pixel-wise Class Distribution

To identify potential class imbalance, pixel counts for each class were computed for both the train and validation splits. Results indicated a heavy dominance of Drivable, Sky, and Far Objects, which collectively accounted for the majority of pixels. In contrast, Living Things and Non-drivable regions were underrepresented as shown in figure. Such imbalance may bias the model towards majority classes, highlighting the importance of loss function or data augmentation strategies during training.

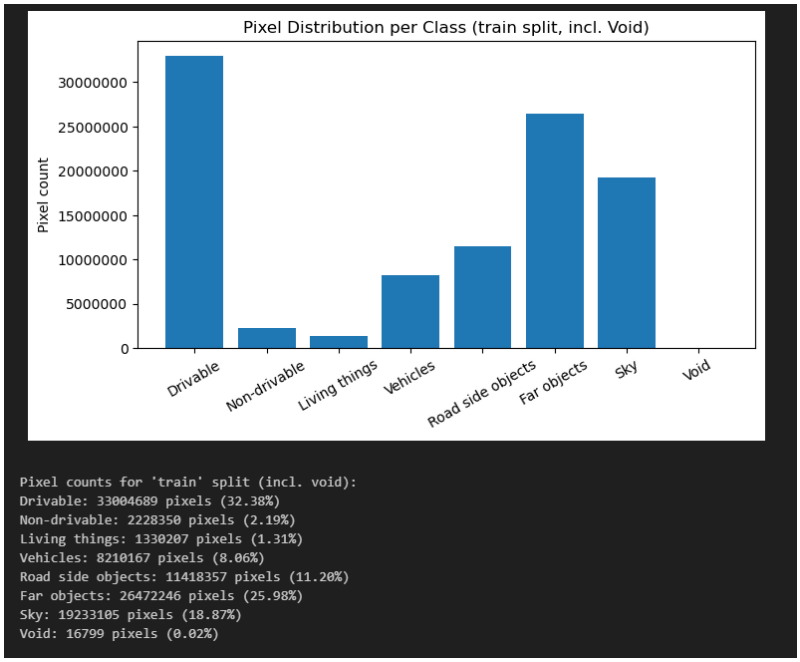


Figure: 8:Pixel wise Distribution of IDD-lite train set

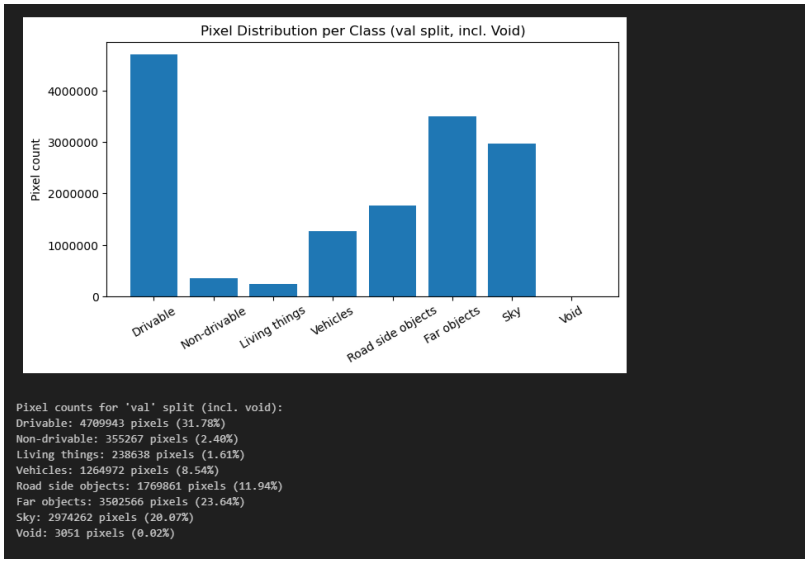


Figure: 9:Pixel wise Distribution of IDD lite Validation set

6.3.7 Visual Inspection of Images and Masks

To validate the semantic masks, we visualised example images alongside their ground-truth segmentation maps. A predefined color palette was applied for clarity, where each semantic class was assigned a distinct RGB value.

Specifically:

- Drivable = purple
- Non-drivable = pink
- Living things = cyan
- Vehicles = orange
- Roadside objects = light brown
- Far objects = teal
- Sky = sky blue
- Void (ignored class) = black

An example is shown in Figure: 10, where the road surface is correctly identified as Drivable as purple , the cars and rickshaw are detected as Vehicles as orange, pedestrians are mapped under Living things as cyan, and background categories such as Sky and Far objects are correctly represented.

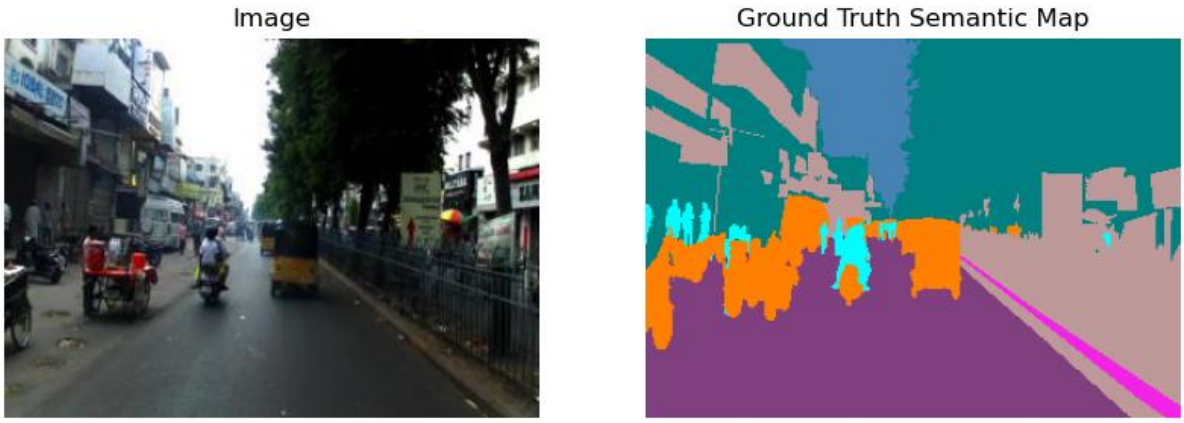


Figure: 10:Example OF IDD-Lite Image and Its corresponding ground-truth semantic masks with predefined color-coded class mappings

This visualization confirmed that the class labels were properly encoded and consistent with the dataset specifications, thereby ensuring reliable pixel-level annotations for subsequent model training.

Chatper 7: Methodology and Implementation

This section provides a detailed account of the complete implementation and training pipeline for the comparative analysis of SegFormer and Mask2Former architectures. The methodology is designed for clarity, reproducibility, and consistency across all experiments. It outlines the dataset preparation,preprocessing , model intialization configurations, loss functions,training configuration,environment and evaluation metrics employed.

7.1 Data Preparation and Preprocessing

The training pipeline for all experiments began with the consistent preparation of the Indian Driving Dataset (IDD-Lite). All data processing was performed using a custom PyTorch Dataset class to handle the unique directory structure and apply consistent transformations.

- **Image and Mask Loading:** The get_paths function was developed to systematically locate and pair RGB images (_image.jpg) with their corresponding ground-truth segmentation masks (_label.png) from the IDD-Lite dataset. This function ensures deterministic ordering and strict adherence to the dataset's naming conventions for both the training and validation splits.

- **Semantic Label Mapping:** A label mapping was established to align the seven semantic classes of IDD-Lite with the models' outputs. Dictionaries for bidirectional mapping (ID2LABEL and LABEL2ID) were created, and a dedicated ignore_index of 255 was set to disregard void or unlabeled pixels during both training and evaluation.

- **Data Augmentation:** To prevent overfitting and enhance the models' ability to generalize, the Albumentations [25] library was used to apply a set of stochastic augmentations to the training data. The pipelines included RandomResizedCrop to a size of 512x512 pixels,Horizontal Flip ,Affine transformations, Color Jitter and GaussianBlur .The input image and mask size was consistently set to 512x512 across all experiments, a value adapted from the original SegFormer paper's methodology [5] to maintain a fair comparison.

- **Final Data Transformation using the Image Processors :** The Hugging Face ImageProcessor for each model Segformer Image Processor [15] and Mask2former Image Processor [16] handled the final model-specific transformations. It was configured with do_rescale as True and do_normalize as True and crucially, with do_reduce_labels as False so the IDD-Lite labels are already correctly configured.

7.2 Model Initialization and Configuration

The experiments were conducted under three distinct conditions for SegFormer and two for Mask2Former to provide a comprehensive comparative analysis. All models were implemented in PyTorch using the Hugging Face transformers library, leveraging publicly available pre-trained checkpoints from the Hugging Face Hub.

7.3 Implementation Details and References

For this study, the models were not built from scratch. Instead, we used the official implementations of SegFormer and Mask2Former from the Hugging Face transformers library[15][16] . These are classes like SegformerForSemanticSegmentation , SegformerConfig [15] and Mask2FormerForUniversalSegmentation,Mask2FormerConfig[16] and to fine-tune on IDD for segformer and mask2former I referred to this implementations as well in [17] and [18] respectively. This was a good way to start because these implementations are well-tested and are based on the original research papers. So, this is reliable foundation .

7.4 SegFormer Experiments

- **Experiment 1: SegFormer (with B0/B1/B2 backbones) and Cityscapes-Pretrained Weights, Fine-tuned on IDD-Lite** This experiment used three SegFormer models with B0 [19] , B1 [20] , or B2 [21] backbones that used Cityscapes-pretrained weights from the pre-trained Cityscapes checkpoints and fine tuned on IDD-lite . The SegformerForSemanticSegmenration.from_pretrained ( ) method was used with num_labels as 7 and semantic_loss_ignore_index as 255 . The ignore_mismatched_sizes [15] as True parameter was essential for replacing the original classification head with a new one for the IDD-Lite classes.

- **Experiment 2: SegFormer (with B0/B1/B2 backbones) Trained from Scratch on IDD-Lite** This experiment used to train SegFormer models with B0 [19], B1 [20], and

B2 [21] backbones with randomly initialized weights, ensuring no transfer learning benefits. This was accomplished by first loading the model's architectural configuration via SegformerConfig.from_pretrained () [15], with the pretrained cityscapes checkpoints and then instantiating the SegformerForSemanticSegmentation class[directly from this configuration object, as demonstrated in [22].

- **Experiment 3: SegFormer (with B0/B1/B2 backbones) and Cityscapes-Pretrained Weights with Focal Loss, Fine-tuned on IDD-Lite** This experiment mirrored Experiment 1,[19],[20],[21] but used a custom FocalLossTrainer to explore the impact of Focal Loss on class imbalance, with the loss objective configured with a gamma of 2.0 .

7.5 Mask2Former Experiments

- **Experiment 4: Mask2Former (with Swin-Tiny/Swin-Small backbones) and Cityscapes-Pretrained Weights, Fine-Tuned on IDD-Lite** This experiment used two Mask2Former models with Swin-Tiny [29] or Swin-Small [30] backbones that used Cityscapes-pretrained weights from the pre-trained Cityscapes checkpoints and fine tuned on IDD-lite.The Mask2FormerForUniversalSegmentation.from_pretrained ( ) method was used, with num_labels as 7, ignore_value as 255 and ignore_mismatched_sizes [16] as True configured to adapt the pre-trained classification head to the 7-class task while preserving the encoder weights.

- **Experiment 5: Mask2Former (with Swin-Tiny/Swin-Small backbones) Trained from Scratch on IDD-Lite** This experiment used to train Mask2Former, Swin-Tiny [29] or Swin-Small [30] backbones , with randomly initialized weights to measure performance without transfer learning. This was accomplished by first loading the model's architectural configuration via Mask2FormerConfig.from_pretrained () [16] with the pretrained cityscapes checkpoints and then instantiating the Mask2FormerForUniversalSegmentationclass directly from this configuration object, as demonstrated in [22]

7.6 Training Configuration and Environment

The training procedures were adapted to each model's unique architecture while maintaining consistency in core hyperparameters for a fair comparison.

- **Custom Implementations:** For Mask2Former, a custom Mask2FormerTrainer was implemented to handle its unique list-based label format and multi-task loss. For SegFormer with Focal Loss, a custom FocalLossTrainer was used.

- **Optimizer:** The adamw_torch optimizer was chosen for all experiments, with a learning rate of 6e-5 and a polynomial learning rate scheduler [32]. The learning rate,

along with the use of the adamw_torch optimizer [33], was adopted from the original SegFormer paper [5].

- **Hyperparameters:** Key hyperparameters, defined in the TrainngArguments [26] class, were kept consistent across all experiments: a batch size of 4 and a maximum of 10 epochs which were adapted from the Eff-UNet paper [13] . The training parameters were configured for logging, evaluation, and checkpointing, with a robust strategy to save the best-performing model based on validation metrics. A warmup ratio of 0.1 and a weight decay of 0.01 [32] were applied to improve convergence and reduce overfitting. To further enhance training stability, mixed-precision training fp16 as True and a fixed random seed as 42 were employed for efficiency and reproducibility. In addition, an EarlyStoppingCallback [27] with a patience of 3 epochs was implemented as a regularization measure to prevent overfitting and improve efficiency.

- **Hardware and Software Environment:** All experiments were executed using using separate **Kaggle Notebooks**, with each notebook dedicated to a single experimental run. This design choice ensured complete isolation between experiments, allowing every run to be fully reproducible with its own code, configuration, and results contained within a version-controlled environment. The experiments were conducted using a NVIDIA A100 Tensor GPU, The implementation relied primarily on the PyTorch deep learning framework, the Hugging Face transformers library for model configuration [15],[16] and training, and standard Python libraries for data handling, preprocessing, and visualization.

7.7 Loss Functions

The choice of a loss function is crucial for guiding the model's learning process. For these experiments, different loss functions were employed to suit each model's architecture and address specific challenges like class imbalance.

- **SegFormer Loss Functions:**
    - By default, the SegformerForSemanticSegmentation class [15] in the Hugging Face library uses a standard Cross-Entropy Loss. This is a common choice for per-pixel classification tasks, as it measures the difference between the predicted and true probability distributions for each pixel.
    - In Experiment 3, a **Focal Loss** was used to address class imbalance. As proposed in the paper [23], the loss function dynamically down-weights the loss from easy, well-classified examples, thereby forcing the model to focus

on a sparse set of hard, misclassified examples. The formula for Focal Loss is given by:

$$\mathrm{FL}(pt) = -(1-pt)\gamma \log(pt)$$

- Here, pt is the predicted probability for the ground-truth class, and γ is a tunable focusing parameter. We used a value of $\gamma = 2.0$ from the segmentation-models-pytorch (SMP) library [31] to maximize the effect of down-weighting.

- **Mask2Former Loss Function:**

  - By default, the Mask2FormerForUniversalSegmentation class [16] uses a powerful, built-in multi-task loss function. This loss is a combination of a classification loss and a mask loss. The mask loss is itself a combination of Binary Cross-Entropy Loss and Dice Loss.

  - This auxillary loss function automatically handles class imbalance and focuses on both pixel-level classification and mask-level overlap, making it well suited for Mask2Former's unified segmentation approach. All loss components were configured to ignore the 255 label.

7.8 Evaluation Procedure and types of visualization

Model performance was evaluated on the validation set at the end of each epoch using a custom compute_metrics function and the Hugging Face evaluate library [24].

7.8.1 Evaluation metrics used:

The following metrics were used for quantitative evaluation:

- **Mean Intersection over Union (mIoU):** This is the primary metric for semantic segmentation. It measures the overlap between the predicted segmentation mask and the ground-truth mask. The IoU is calculated for each class individually and then averaged to give a single overall score. A higher mIoU indicates better overall segmentation quality across all classes.

- **Pixel Accuracy:** This metric calculates the percentage of pixels that were correctly classified across the entire image. It provides a simple, high-level overview of a model's performance .

- **Per-Class IoU:** This metric provides a detailed breakdown of the IoU for each of the seven individual classes. Unlike mIoU, which is an average, the per-class IoU offers granular insight into a model's strengths and weaknesses on specific object categories, such as vehicles or living things . This is particularly useful for analyzing performance on rare or difficult to segment classes.

7.8.2 Results and Visualization

The final training metrics and logs were automatically saved by the Hugging Face Trainer [26] into the trainer.state.log.history attribute and a history.json file in the output directory. The checkpoint of the best-performing model, based on the validation mean IoU, was also saved.

For visualization and analysis, the results were accessed by loading the checkpoint of the best model and the trainer.state.log.history file. The following plots were generated to visualize the models' performance:

- **Line Plots** were used to visualize the training and validation loss and Mean IoU, Pixel Accuracy over each epoch. These plots were critical for assessing model convergence, for diagnosing potential overfitting and to get the overall performance of the model

- **Bar Plots** were used to compare the performance of each model and backbone. This included Per-Class Performance: A plot showing the IoU score for each of the seven semantic classes, which allowed for a detailed analysis of each model's strengths and weaknesses.

The detailed code implementations is in Chatper 11:appendices

7.8.3 Qualitative Visualizations for ground truth vs predicted semantic masks

For a deeper qualitative analysis, a side-by-side comparison was generated to display the original image, the ground-truth mask, and the predicted semantic maps. The implementation of this visualization differed slightly between SegFormer and Mask2Former to accommodate their distinct output formats.

The following functions were used for both models to standardize the visualization process for implementation:

1. colorize_mask: A function that takes a segmentation mask and converts it into a color-coded image using a predefined color palette. The palette includes a distinct color for each of the 7 classes as well as for the 255 ignore label.

2. unnormalize_img: A function that reverses the normalization applied during preprocessing, converting the image tensor back to a standard RGB format for display.

2. SegFormer Visualization: For SegFormer, the model outputs a low-resolution tensor of logits . The visualization code first sets the model to evaluation mode model.eval () and then applies an argmax operation along the class dimension to get the predicted class ID for each pixel. This resulting prediction map is then passed to the function to create the final visualization. These images were generated using a random selection of samples from the validation set.

3. Mask2Former Visualization: For Mask2Former, the visualization process is more complex due to its query-based architecture. The model does not output a standard logit map. Instead, the raw output (class logit queries and mask logit queries) is passed to the processor.post_process_semantic_segmentation function [16]. This function handles the complex logic of combining the query predictions and resizing them to the original image dimensions. The final, resized prediction map is then passed to the colorize_mask function. These images were also generated using a random selection of samples from the validation set, with the model set to evaluation mode model.eval prior to inference.

The detailed code implementations is in Chatper 11:appendices

7.9 Implementation of Segformer Training Pipeline

7.9.1 .Importing libraries for Segformer

All necessary libraries for the experiment. For PyTorch torch is the core deep learning framework. os, numpy, and PIL are used for file system interaction, numerical operations, and image handling, respectively. Albumentations is a specialized library for efficient computer vision data augmentation. The key components from the Hugging Face transformers libraries **SegformerForSemanticSegmentation** and **SegformerImageProcessor [15]** provides the pre-trained model and its specific image processor respectively for segformer, and the high-level **Trainer API** [26] and **TrainingArguments [26]** for orchestrating the training loop. Finally, the evaluate library is imported for metric calculation [15].

7.9.2 .Data Collection: Image and Mask Paths

For the procedure of locating and pairing input images with their corresponding ground truth segmentation masks. The IDD-Lite dataset's directory structure has RGB images under

leftImg8bit/<split>/<city>/*_image.jpg path and matching gtFine/<split>/<city>/*_label.png for train and val.

The get_image_mask_paths function traverses, under each split, replaces the _image.jpg suffix with _label.png, and returns deterministically ordered lists of image and mask pairs for

train and val. It operates on the strict naming convention of the dataset, where an image named _image.jpg has a corresponding mask named _label.png. The function is called for both the training and validation splits to generate exhaustive lists of file paths, which are the foundational inputs for the data loading pipeline.

7.9.3 Data Preprocessing and Data Augmentation

7.9.3.1 Class Names and Label Mapping

I defined the **seven** semantic classes used in IDD-Lite and assigned num_labels as 7. and constructed dictionaries to map between class indices and their string names such as **bidirectional mappings** ID2LABEL (index to name) and LABEL2ID (name to index) As they are required for model configuration and evaluation.

7.9.3.2 Data Augmentation

For Data Augmentation, two Albumentations pipelines are defined and applied to both the image and its corresponding mask. For training, a composed transform is used that first performs a RandomResizedCrop to five hundred twelve by five hundred twelve pixels, with a scale range of zero point seven to one point zero and an aspect ratio between zero point seven five and one point three three, applied with a probability of one hundred percent.

Subsequently, a HorizontalFlip is applied with a fifty percent probability, followed by a light Affine jitter with a translation of up to five percent, a scale range of zero point nine to one point one, and rotation between negative fifteen and fifteen degrees, also with a fifty percent probability. Finally, mild photometric changes are introduced through ColorJitter, with brightness, contrast, and saturation factors of up to zero point two and a hue factor of up to zero point zero five, applied with a fifty percent probability. A GaussianBlur is also applied with a twenty percent probability.

For validation, a deterministic Resize to 512 x 512 is used, with no stochastic augmentation. Normalization is intentionally not performed in Albumentations[25] because it is handled later by the Hugging Face SegformerImageProcessor[15]

7.9.3.3 Setting up the Segformer Preprocessor

To handle the model-specific formatting of the input data, I initialized the SegformerImageProcessor from the Hugging Face transformers library[15]. I configured this processor to align with the requirements of both my dataset and the model.

First, I set do_rescale as True to rescale the image pixel values from a [0, 255] range to [0.0, 1.0]. Subsequently, I set do_normalize as True to apply the standard ImageNet mean and standard deviation normalization, which is a mandatory step to match the data distribution the model expects.

To tailor it to my IDD-Lite datatset, I configured by setting num_labels as7 for the seven class label categories. Crucially, I specified an ignore_index of 255, which instructs the loss function to disregard any pixels with this value in the ground truth masks during training. I then integrated this processor object into my custom Dataset class, where it performs the final transformation on the augmented data before it is passed to the model.

7.9.3.4 Custom Dataset Class

I implemented a custom PyTorch Dataset class named IDDSegDataset to efficiently load and prepare image-mask pairs for the model.

For each data sample, the class performs a four-step pipeline:

First, I load the RGB image and its mask from disk. Second, I clean the apply Albumentations [25], to both the image and mask using stochastic augmentations for training like random resized crop, horizontal flip, light affine, color jitter, and optional blur and a deterministic resize to 512×512 for validation. Third, I pass the augmented arrays to the Hugging Face processor, which rescales and normalizes the image and returns tensors for both pixel_values and labels . Finally then drop the singleton batch dimension and pass pixel values and labels. I conclude by constructing two dataset instances for train_ds with the training augmentations and val_ds with the validation resize only.

7.9.4 Model Initialization For for Idd-lite

(A). For Domain Specific Pretraining experiments for Segformer model and fine-tuning IDD-lite.

I initialized SegFormer using the Hugging Face Transformers library [15],

The model was instantiated by loading a public Cityscapes checkpoint and adapting its classification head to the seven classes of the IDD-Lite dataset.

This was achieved by calling the SegformerForSemanticSegmentation.from_pretrained () method with a specific checkpoint identifier (for example : nvidia/segformer-bX-finetuned-cityscapes-1024-1024) where the 'bX' corresponds to the B0 [19] , B1 [20] , or B2 [21] encoder variant depending on the experiment. This method loads all encoder and decoder weights from the pre-trained model. To configure the model for my task, I passed several key arguments, I set num_labels as 7 and provided the id2label and label2id mappings to ensure the new classification head was correctly sized and aligned with my class structure. The crucial ignore_mismatched_sizes as True parameter was set to discard of the original

Cityscapes-trained head with this new, randomly initialized seven-class head. Finally, I set semantic_loss_ignore_index as 255 ,[15], to ensure void pixels were ignored during the loss calculation, aligning the model with the data preprocessing and evaluation metrics. This initialization way helps to retain the powerful, pre-trained encoder weights, and helps to fine-tune on the IDD-Lite Dataset

I repeated this process for the B0, B1, and B2 encoder backbones to study the effect of model capacity under these training conditions.

(B).Train from scratch experiments for segformer model on IDD-lite

For the experiments designed to train the model from scratch, I used an alternative initialization path to remove any benefit from pre-training and establish a performance baseline with random weights.

This involved a two-step process. First, I created a SegformerConfig object [15], which serves as an architectural blueprint for the model, by calling its .from_pretrained() method. I used the same checkpoint identifier (for example, nvidia/segformer-bX-finetuned-cityscapes-1024-1024, where 'bX' corresponds to the B0 [19], B1 [20], or B2 [21] encoder variant) not to load the model's weights, but to borrow its architectural blueprint. I then modified this configuration object with the parameters for my target dataset, setting num_labels to seven and providing the id2label and label2id mappings.

Second, I instantiated the SegformerForSemanticSegmentation model by passing this modified config object directly to its constructor. Because I provided a configuration object rather than using the model's own .from_pretrained() method, the resulting model was created with the correct architecture but with all of its weights randomly initialized. This procedure guarantees that no knowledge from the pre-trained checkpoint was carried over. I repeated this process for the B0, B1, and B2 encoder backbones to study the effect on how performance scales with model size

7.9.5 Computing Metrics for training and evaluation for Segformer : Mean IoU, Pixel Accuracy, Per-Class IoU

We use Hugging Face's evaluate library[24] to compute mean IoU, pixel accuracy, and per-class IoU, while ignoring 255 pixels in both training and evaluation.

I provided a custom function, compute_metrics, to the Trainer to perform a comprehensive evaluation at the end of each epoch. I designed this function to handle the specific output format of the SegFormer architecture and to calculate a rich set of metrics.

The first and most critical step I implemented within this function was the upsampling of the model's output logits. Because the SegFormer architecture produces low-resolution predictions, I used the torch.nn.functional.interpolate method with a bilinear mode to resize the logits to match the exact spatial dimensions of the full-resolution ground truth labels.

After upsampling, I determined the final class prediction for each pixel by applying an argmax operation across the class dimension of the logit tensor. I then passed the resulting prediction maps and the ground truth labels to the Hugging Face evaluate library to calculate the

primary metric, Mean Intersection over Union (mIoU**)**, along with pixel accuracy and per-class IoU. I configured the metric to ignore any pixels with the label 255 to ensure consistency with my data preprocessing.By default, the SegformerForSemanticSegmentation implementation in the Hugging Face transformers library [15] uses a pixel-wise,cross-entropy loss function which ignores label 255 to compute the loss.

Finally, I formatted all calculated metrics such as into a single dictionary and returned it to the Trainer. The Trainer used this dictionary for logging my model's performance at each epoch and for identifying the best-performing model checkpoint based on the mean_iou score.

7.9.6 focal loss implementation for Segformer cityscapes pretrained model experiments

For the experiments involving Focal Loss, the standard Hugging Face Trainer was extended to use a custom loss function. This was achieved by creating a new class, FocalLossTrainer, that inherits from the base Trainer class.

The key modification in this custom trainer was to override the compute_loss method. This allowed for the replacement of the default cross-entropy loss with a Focal Loss objective from the segmentation-models-pytorch library[24].

Within this overridden method, the model's output logits are first upsampled using bilinear interpolation to match the spatial dimensions of the ground truth labels, a necessary step for the SegFormer architecture. The Focal Loss function is then instantiated and configured for the multi-class task with a gamma parameter of 2.0, which controls the focusing effect on hard to classify examples. The ignore_index was set to 255 to ensure consistency with the data preprocessing.

Finally, this loss function is applied to the upsampled logits and labels to compute the final loss value, which is then returned to the training loop. This custom FocalLossTrainer was used in place of the standard Trainer.

7.9.7 Training configuration

To configure the entire training process for my experiments with Mask2Former, I used the TrainingArguments [26] class from the Hugging Face transformers library. This allowed me to define all hyperparameters and settings in a single, reproducible object.

I set the output_dir to a unique path for each experimental run to store all model checkpoints and results. I also specified a logging_dir and set the report_to parameter to ["tensorboard"] to enable real-time monitoring of my training metrics.

For the core training parameters, I set the number of training epochs to ten and the batch size to four. For optimization, I chose the adamw_torch optimizer [33] with an initial learning_rate of 0.00006 and a polynomial learning rate scheduler[32], following the original Segformer paper [5].

I implemented a robust strategy for evaluation and checkpointing. By setting eval_strategy and save_strategy to “epoch”, I instructed the Trainer [26] to perform a full evaluation and save a model at the end of every epoch. To ensure I only retained the best-performing version, I set load_best_model_at_end to True, which uses the metric_for_best_model which I set to "mean_iou" to identify and restore the best checkpoint after training is complete.

Finally, to accelerate training and ensure reproducibility, I enabled mixed-precision training by setting fp16 to True and used a global seed of 42.

7.9.8 Training the model

I ran the entire process using the Hugging Face Trainer or FocalLossTrainer for the focal-loss experiments , which takes care of the training loop, evaluation, and checkpointing. I set it up with my SegFormer model, either the Cityscapes-pretrained version or the from-scratch version. I also included the training_args hyperparameter block, along with my train_ds and val_ds datasets, and the custom compute_metrics function that tracks mean IoU, pixel accuracy, and per-class IoUs. To prevent overfitting and reduce unnecessary computation, I added an EarlyStoppingCallback [27] with a patience of three. This tells the Trainer to stop if the main metric, mean_iou, does not improve for three consecutive epochs.

To run the model I used trainer.train() [26] , which executes the full loop defined in my TrainingArguments [26] up to 10 epochs, evaluation and checkpointing each epoch, and early stopping with a patience of 3. The Trainer also restores the best checkpoint by validation mean IoU, writes metrics to TensorBoard, and records per-epoch logs in trainer.state.log_history.

7.10 Implementation of Mask2former Pipeline

7.10.1 Importing Libraries for Mask2former

I used standard libraries such as os, numpy, and PIL for file handling, numerical operations, and image processing. The core deep learning framework I employed was PyTorch, from which I used the Dataset class to build my custom data loader. For data augmentation, I utilized the albumentations [25] library.

From the Hugging Face transformers library, I imported several key components specific to this experiment. I selected Mask2FormerForUniversalSegmentation [16] as my model architecture and its corresponding Mask2FormerImageProcessor to handle the required data preprocessing. To manage the overall workflow, I used the same high-level APIs as in my previous experiments: the Trainer class to orchestrate the training loop and the TrainingArguments [26] class to configure all of my hyperparameters.

Finally, I imported the evaluate library to calculate the Mean Intersection over Union (mIoU) metric for performance assessment.

7.10.2 Data Collection: Image and Mask Paths

To implement the data collection pipeline, I first defined a Python function, `get_paths`, to locate and pair the input images with their corresponding ground truth segmentation masks.

I designed this function to traverse the IDD-Lite dataset's directory structure, which is organized by split (train/val) and city. The function's logic relies on the strict naming convention of the dataset; for each image file ending in _image.jpg, it programmatically generates the path to the corresponding mask by replacing that suffix with _label.png.

I then called this function for both the training and validation splits to generate exhaustive and deterministically ordered lists of file paths. These lists served as the foundational inputs for my custom PyTorch Dataset class.

7.10.3 Data Preprocessing and Data Augmentation

7.10.3.1 Class Names and Label Mapping

Similarly for mask2former ,I defined the seven semantic classes used in IDD-Lite and assigned num_labels as 7. and constructed dictionaries to map between class indices and their string names such as bidirectional mappings ID2LABEL (index to name) and LABEL2ID (name to index) As they are required for model configuration and evaluation.

7.10.3.2 Data Augmentation

For Data Augmentation like we did for segformer , we do the same implementation for mask2former as well , two Albumentations pipelines are defined and applied to both the image and its corresponding mask. For training, a composed transform is used that first performs a RandomResizedCrop to five hundred twelve by five hundred twelve pixels, with a scale range of zero point seven to one point zero and an aspect ratio between zero point seven five and one point three three, applied with a probability of one hundred percent.

Subsequently, a HorizontalFlip is applied with a fifty percent probability, followed by a light Affine jitter with a translation of up to five percent, a scale range of zero point nine to one point one, and rotation between negative fifteen and fifteen degrees, also with a fifty percent probability. Finally, mild photometric changes are introduced through ColorJitter, with brightness, contrast, and saturation factors of up to zero point two and a hue factor of up to zero point zero five, applied with a fifty percent probability. A GaussianBlur is also applied with a twenty percent probability.

For validation, a deterministic Resize to 512 x 512 is used, with no stochastic augmentation. Normalization is intentionally not performed in Albumentations because it is handled later by the Hugging Face Mask2formerImageProcessor[16]

7.10.3.3 Setting up the Mask2former Preprocessor

To handle the model-specific formatting of the input data, I initialized the Mask2formerImageProcessor [16] from the Hugging Face transformers library. I configured this processor to align with the requirements of both my dataset and the model.

First, I set do_rescale as True to rescale the image pixel values from a [0, 255] range to [0.0, 1.0]. Subsequently, I set do_normalize as True to apply the standard ImageNet mean and standard deviation normalization, which is a mandatory step to match the data distribution the model expects.

To tailor it to my IDD-Lite datatset, I configured by setting num_labels as 7 for the seven class label categories. Crucially, I specified an ignore_index of 255, which instructs the loss function to disregard any pixels with this value in the ground truth masks during training. I then integrated this processor object into my custom Dataset class, where it performs the final transformation on the augmented data before it is passed to the model.

7.10.3.4 Custom Dataset Class

I implemented a custom PyTorch Dataset class named IDDSegDataset to efficiently load and prepare image-mask pairs for the Mask2Former model. For each data sample, the class performs a multi-step pipeline:

First, I load the RGB image and its corresponding mask from disk. Second, I clean the mask by replacing any invalid labels with an ignore index of 255. Third, I apply Albumentations to both the image and the sanitized mask, using stochastic augmentations for training—like random resized cropping, horizontal flipping, and color jittering and a deterministic resize to 512×512 for validation.

Finally, I pass the augmented arrays to the Hugging Face Mask2FormerImageProcessor. This processor converts the data into the specific format required by the model, returning tensors for pixel_values, mask_labels, and class_labels. I also retain the final augmented semantic mask as labels for use in my evaluation function.

I conclude by constructing two dataset instances, train_ds with the training augmentations and val_ds with only the validation resize.

7.10.3.5 Data Collate Function

To correctly batch the specialized dictionaries produced by my IDDSegDataset, I implemented a custom **collate_fn** and passed it to the Trainer. This function is responsible for taking a list of individual data samples and merging them into a single batch that can be fed directly to the model.

My implementation handles the different data types within each sample dictionary with specific logic. For the pixel_values and the raw labels tensors, I used torch.stack. This operation takes the sequence of individual tensors from the batch and concatenates them along a new dimension, effectively creating a single batch tensor (for example a list of N image tensors of shape [Channel, Height, Width] becomes a single tensor of shape [N, C, H, W]).

In contrast, the **mask_labels** and **class_labels** are gathered into a Python list. This was a deliberate and critical step, as the Mask2FormerForUniversalSegmentation model's forward pass is specifically designed to accept these particular label structures as a list of tensors, where each element corresponds to a sample in the batch. My custom collate function thereby ensures that every batch of data is structured in the precise format required by both the model for its forward pass and the evaluation function for metric computation.

7.10.4 Model Intialization

(A). For Domain Specific Pretraining experiments for Mask2former model and fine-tuning IDD-lite

I initialized Mask2former **using the Hugging Face Transformers library**,

The model was instantiated by loading a public Cityscapes checkpoint and adapting its classification head to the seven classes of the IDD-Lite dataset.

This was achieved by calling the Mask2FormerConfig.from_pretrained.from_pretrained () method with a specific checkpoint identifier (for example : "facebook/mask2former-swin-XX-cityscapes-semantic") where the 'XX' corresponds to the tiny [29] or small [30] encoder variant depending on the experiment. This method loads all encoder and decoder weights from the pre-trained model. To configure the model for my task, I passed several key arguments, I set num_labels as 7 and provided the id2label and label2id mappings to ensure the new classification head was correctly sized and aligned with my class structure. The crucial ignore_mismatched_sizes as True parameter was set to discard of the original Cityscapes-trained head with this new, randomly initialized seven-class head. Finally, I set ignore_value as 255 as per the hugging face mask2former documentation [16] to ensure void pixels were ignored during the loss calculation, aligning the model with the data preprocessing and evaluation metrics. This initialization way helps to retain the powerful, pre-trained encoder weights, and helps to fine-tune on the IDD-Lite Dataset

I repeated this process for the swin-tiny and swim-small encoder backbones to study the effect on how performance scales with model size

(B).Train from scratch experiments for mask2former model on IDD-lite

For the experiments designed to train the Mask2Former model from scratch, I used an alternative initialization path to remove any benefit from pre-training and establish a performance baseline with random weights.

This involved a two-step process. First, I created a Mask2FormerConfig object[16], which serves as an architectural blueprint for the model, by calling its .from_pretrained () method. I used a public checkpoint identifier (for example, "facebook/mask2former-swin-XX-cityscapes-semantic", where 'XX' corresponds to the tiny[29] or small Swin [30] Transformer backbone) not to load the model's weights, but to borrow its architectural blueprint. I then modified this configuration object with the parameters for my target dataset, setting num_labels to seven and providing the id2label and label2id mappings.

Second, I instantiated the Mask2FormerForUniversalSegmentation [16] model by passing this modified config object directly to its constructor. Because I provided a configuration object rather than using the model's own .from_pretrained () method, the resulting model was created with the correct architecture but with all of its weights randomly initialized. This procedure guarantees that no knowledge from the pre-trained checkpoint was carried over.Similarly I repeated this process for the swin-tiny and swin-small backbones to study the effect of model size under these training conditions.

7.10.5 Computing Metrics for training and evaluation for Mask2former : Mean IoU, Pixel Accuracy, Per-Class IoU

In the compute_metrics function, evaluation is delegated to Hugging Face's Evaluate library[24] by loading the "mean_iou" metric module with miou = evaluate.load("mean_iou"). This provides a standardized method for calculating segmentation metrics without custom implementations. The function first unpacks the eval_pred tuple, which contains the model's post-processed predictions (semantic class maps already resized to match ground-truth shapes via prediction_step) and the corresponding ground-truth labels, converting both to CPU-resident NumPy arrays for compatibility. These arrays are then passed to miou.compute (predictions as np_preds, references as np_labels, num_labels as 7, ignore_index as 255), configured to align with IDD-Lite's seven classes and exclude void pixels, ensuring consistency with data preprocessing and the model's loss handling. This call yields a dictionary with mean IoU which is the primary metric , overall pixel accuracy, and a per-category IoU array, the per-class values are mapped to the dataset's class names and formatted into a results dictionary (for example "mean_iou", "pixel_accuracy", and named keys like "drivable_iou"). The Trainer [26] uses this dictionary to log performance per epoch, visualize in TensorBoard, and select the best checkpoint based on mean_iou, as specified in TrainingArguments such as metric_for_best_model as "mean_iou", load_best_model_at_end as True .

7.10.6 Adapting the Trainer for Mask2Former

To accommodate the unique data requirements and output structure of the Mask2Former architecture, the standard Hugging Face Trainer[26] was extended by creating a custom Mask2FormerTrainer class. This class inherits from the base Trainer and overrides three key methods to ensure compatibility.

- to_device Method

This method was overridden to correctly handle the specific data structure of Mask2Former's labels. The Mask2FormerImageProcessor generates mask_labels and class_labels as lists of tensors. The standard Trainer does not natively handle this format, so this custom method iterates through these lists and moves each tensor individually to the target computational device in my case it's the GPU.

- compute_loss Method

The compute_loss method was overridden to align with the Mask2FormerForUniversalSegmentation [16] model's forward pass. The implementation first removes the labels key from the inputs, as this contains the full semantic mask used only for evaluation. It then passes the required inputs, the pixel_values, mask_labels, and class_labels to the model. By Default the hugging face Mask2Former model [16] uses an Auxillary loss function which computes its internal loss as a weighted combination of cross-entropy on class predictions, binary cross-entropy on mask predictions, and Dice loss for overlap, automatically incorporating the ignore_value as 255 for void pixels.

this method then extracts the final loss value from the model's output object and returns it to the training loop.

- prediction_step Method

Finally, the prediction_step method, invoked during evaluation, was overridden to manage Mask2Former's complex raw outputs. After removing the "labels" key and performing a no-gradient forward pass with pixel_values, mask_labels, and class_labels, the method computes the loss. The raw outputs (class_queries_logits and masks_queries_logits)[16] which the model outputs are then post-processed using the processor.post_process_semantic_segmentation [16] function, which combines the query predictions into final semantic class maps via argmax and resizes them to the target_sizes derived from the ground truth labels' shapes. This step is essential for pixel-to-pixel comparison in compute_metrics. The ground truth labels are detached and moved to CPU for efficiency. The method returns the loss, processed predictions, and label list in the format expected by the Trainer's evaluation loop, enabling accurate metric computation like mIoU.

These adaptations allow the Trainer to fully support Mask2Former's unified segmentation framework, facilitating efficient fine-tuning and evaluation on the IDD-Lite dataset without modifications to the underlying Hugging Face library.

7.10.7 .Training Configuration

To configure the entire training process for my experiments with Mask2Former, I used the TrainingArguments class [26] from the Hugging Face transformers library. This allowed me to define all hyperparameters and settings in a single, reproducible object.

I set the output_dir to a unique path for each experimental run to store all model checkpoints and results. I also specified a logging_dir and set the report_to parameter to ["tensorboard"] to enable real-time monitoring of my training metrics.

For the core training parameters, I set the number of training epochs to ten and the batch size to four. For optimization, I chose the adamw_torch optimizer [33] with an initial learning_rate of 0.00006 and a polynomial learning rate scheduler [32], following the original Segformer paper [5].

I implemented a robust strategy for evaluation and checkpointing. By setting eval_strategy and save_strategy to “epoch”, I instructed the Mask2formerTrainer to perform a full evaluation and save a model at the end of every epoch. To ensure I only retained the best-performing version, I set load_best_model_at_end to True, which uses the metric_for_best_model which I set to "mean_iou" to identify and restore the best checkpoint after training is complete.

Finally, to accelerate training and ensure reproducibility, I enabled mixed-precision training by setting fp16 to True and used a global seed of 42.

7.10.8 Training the model

To train the model , I ran the entire process using the Hugging Face Mask2former Trainer [26] or, which takes care of the training loop, evaluation, and checkpointing. I set it up with my Mask2former model, either the Cityscapes-pretrained or the train from scratch version. I also included the training_args hyperparameter block, along with my train_ds and val_ds datasets, and the custom compute_metrics function that tracks mean IoU, pixel accuracy, and per-class IoUs. To prevent overfitting and reduce unnecessary computation, I added an EarlyStoppingCallback[27] with a patience of three. This tells the Trainer to stop if the main metric, mean_iou, does not improve for three consecutive epochs.

To run the model I used trainer.train()[26], which executes the full loop defined in my TrainingArguments [26] up to 10 epochs, evaluation and checkpointing each epoch, and early stopping with a patience of 3. The Trainer [26] also restores the best checkpoint by validation mean IoU, writes metrics to TensorBoard, and records per-epoch logs in trainer.state.log_history.

Chatper 8: Results and analysis

This chapter presents the comprehensive results and analysis from the series of experiments and the models in Chatper 7: designed to evaluate the Vision Transformer architectures SegFormer and Mask2Former for semantic segmentation on the IDD-Lite dataset. The findings are systematically presented to address the core research objectives

8.1 Experiments 1 : SegFormer (with backbones B0/B1/B2) : Cityscapes-Pretrained, Fine-Tuned on IDD-Lite models

8.1.1 Run 1: SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model

This section presents the results from our SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model . The goal of this first experiment was to set a baseline for performance and speed to compare against the other models. We used a model that was pre-trained on the Cityscapes dataset and then fine-tuned it on our IDD-Lite dataset

- **Overall Final Metrics**

	Mean IoU	Pixel Accuracy	Inference speed(fps) / eval_samples_per_second	parameters	Epochs
Final Metrics	**0.8849**	**0.675163**	**8.163**	**3,715,943**	**10**

Table 2:Segformer B0 Cityscapes-pretrained ,fine tuned on IDD-lite Final metrics

Table 2 shows the main results for the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model. It achieved a mean IoU (mIoU) of 0.675168 and an overall pixel accuracy of 0.8849. The eval_samples_per_second is also called as inference speed, it was able to process 8.16 frames per second (FPS) during the evaluation. These numbers give us a good overall picture of the model's performance, showing that it is capable but still has room for improvement on the challenging IDD-Lite dataset.

- **Final Metrics of IoU Per Class**

	Drivable IoU	Non Drivable IoU	Living things IoU	Vehicles IoU	Road Side Object IoU	Far Objects IoU	Sky IoU

Final Metrics IoU Per Class	**0.930926**	**0.386539**	**0.452286**	**0.753445**	**0.48649**	**0.767897**	**0.948561**

Table 3:Segformer B0 Cityscapes-pretrained ,fine tuned on IDD-lite Final metrics of IoU Per Class

To get a better understanding of these results, we looked at the IoU scores for each specific class, as shown in Table 3. The model was very good at segmenting large and simple areas. For example, the 'sky' class scored 0.948561 IoU, and the 'drivable' class scored 0.938926 IoU This is likely because these parts of the image are large and visually consistent. The model also performed reasonably well with the 'vehicles' class at 0.758445 IoU which is an important class for any self-driving system.

However, the model struggled with more difficult classes. This is likely due to significant class imbalance in the training data, where classes that occupy fewer pixels such as 'living things' at 0.452286 IoU and 'non-drivable' surfaces at 0.386539 IoU are seen less frequently by the model than dominant classes like 'drivable' and 'sky'. This imbalance can prevent the model from learning the features of these minority classes effectively.

- **Training vs Validation Loss of Segformer-B0 Cityscapes Pretrained,fine-tuned on IDD-lite**

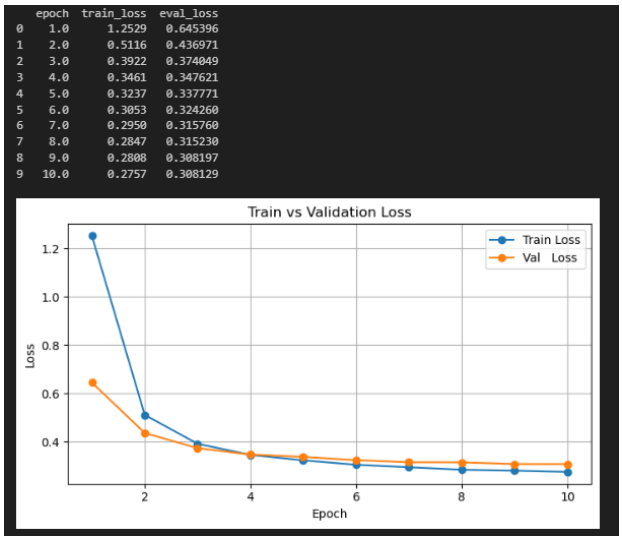


Figure: 11: Training vs Validation Loss of Segformer-B0 Cityscapes Pretrained,fine-tuned on IDD-lite model

The training and validation loss curves for the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model are presented in Figure: 11. The model demonstrates a stable

training process, with both loss values decreasing sharply in the initial epochs before converging. After 10 epochs, the training loss stabilized at a final value of 0.2757, while the validation loss converged to 0.308129.

The small and stable gap between these final values indicates that the model generalizes well to unseen data and did not suffer from significant overfitting.

Finally, in terms of its size, the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model has 3,715,943 trainable parameters. This fits its role as a fairly small and an efficient model.

8.1.2 Run 2: SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model

This section presents the results from the second model in our experiment, SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model .This model, which is larger than the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model, was also pre-trained on the Cityscapes dataset and fine-tuned on IDD-Lite to assess the impact of increased model complexity on performance.

- **Overall Final Metrics**

	Mean IoU	**Pixel Accuracy**	**Inference speed(fps) / eval_samples_per_second**	**parameters**	**Epochs**
Final Metrics	**0.694255**	**0.892985**	**8.022**	**13,679,047**	**10**

Table 4:SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics

Table 4 shows the main results for the SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model. It achieved a mean IoU of 0.694255 and an overall pixel accuracy of 0.892985. Its inference speed was 8.022 frames per second (FPS). Compared to the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model, these results represent a modest improvement in overall pixel accuracy. More significantly, the primary metric for this task, mean IoU, increased by 1.91 percentage points. This gain is accompanied by a slight decrease in inference speed, highlighting the trade-off between model size and performance

- **Final Metrics of IOU Per Class**

	Drivable IoU	**Non Drivable IoU**	**Living things IoU**	**Vehicles IoU**	**Road Side Object IoU**	**Far Objects IoU**	**Sky IoU**
Final Metrics	**0.93930**	**0.43762**	**0.471967**	**0.766017**	**0.514095**	**0.778901**	**0.9518841**

IoU Per Class							

Table 5:SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics of IoU Per Class

A detailed breakdown of the per-class IoU scores is presented in Table 5 The performance pattern is consistent with the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model, showing high proficiency on large, visually distinct classes like 'sky' as 0.951884 IoU and 'drivable' as 0.939302 IoU.

While the SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model shows slight improvements across most categories compared to the baseline, it continues to struggle with the same difficult classes. This is likely due to the significant class imbalance in the training data, where minority classes like 'living things' as 0.471967 IoU and 'non-drivable' as 0.43762 IoU are under-represented.

In terms of complexity, SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model has 13,679,047 trainable parameters, making it substantially larger than the B0 baseline.

- **Training vs Validation for Segforme-B1,Cityscapes-Pretrained,fine-tuned on IDD-lite**

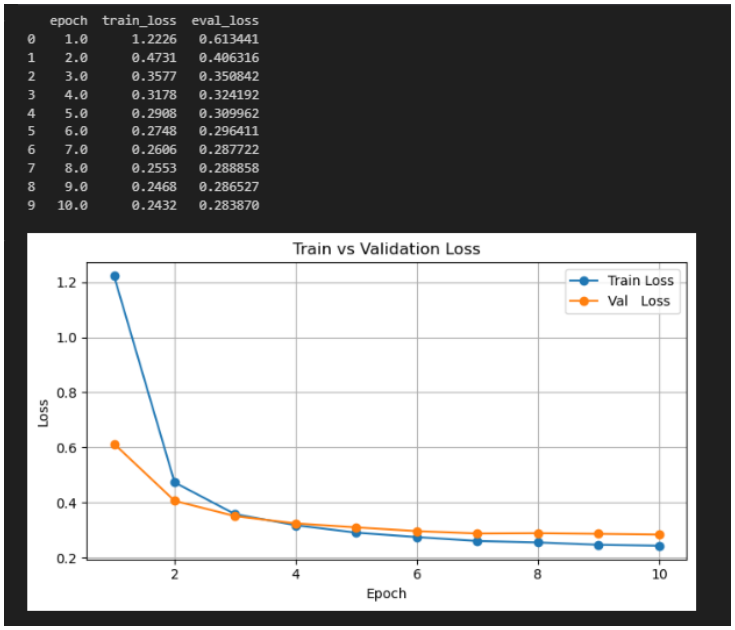


Figure: 12: Training vs Validation Loss of Segformer-B1 Cityscapes Pretrained,fine-tuned on IDD-lite model

The training and validation loss curves for the SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model are presented in Figure: 12 . The model demonstrates a stable training process, with both loss values decreasing sharply in the initial epochs before converging. After 10 epochs, the training loss stabilized at a final value of 0.2432, while the validation loss converged to 0.28387.

The small and stable gap between these final values indicates that the model generalizes well to unseen data and did not suffer from significant overfitting.

8.1.3 Run 3: SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model

This section presents the results from the third and largest model in our experiment, SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model. This model was also pre-trained on the Cityscapes dataset and fine-tuned on IDD-Lite to evaluate if a significant increase in model size and complexity would overcome the challenges observed with the smaller models.

- **Overall Final Metrics**

	Mean IoU	Pixel Accuracy	Inference speed(fps) / eval_samples_per_second	parameters	Epochs
Final Metrics	**0.723008**	**0.902447**	**10.696**	**27,352,007**	**10**

Table 6:SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics

Table 6 shows the main results for the SegFormer-B2 model. It achieved a mean IoU of 0.723008 and an overall pixel accuracy of 0.902447. Its inference speed was 10.696 frames per second (FPS). These results show the highest mIoU and pixel accuracy so far, but also the highest inference speed, which is a counter-intuitive finding that will be explored in the comparative analysis.

- **Final Metrics of IOU Per Class**

	Drivable IoU	Non Drivable IoU	Living things IoU	Vehicles IoU	Road Side Object IoU	Far Objects IoU	Sky IoU
Final Metrics IoU Per Class	**0.94412**	**0.465841**	**0.554328**	**0.798926**	**0.55333**	**0.788861**	**0.955648**

Table 7:SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics of IoU Per Class

A detailed breakdown of the per-class IoU scores is presented in Table 7. The performance pattern is consistent with the previous models, showing high proficiency on large, visually distinct classes like 'sky' at 0.955648 IoU and 'drivable' at 0.94412 IoU .

While the B2 model shows the best performance yet across the difficult minority classes, such as 'living things' at 0.554328 IoU and 'non-drivable' at 0.465041 IoU , these scores are still significantly lower than those for the dominant classes. This reinforces the conclusion that class imbalance is a primary limiting factor for this architecture on this dataset.

- **Training vs Validation Loss for Segformer-B2 Cityscapes-Pretrained,fine-tuned on IDD-Lite model**

	epoch	train_loss	eval_loss
0	1.0	0.9627	0.461005
1	2.0	0.3633	0.325950
2	3.0	0.2933	0.304154
3	4.0	0.2607	0.279522
4	5.0	0.2431	0.276330
5	6.0	0.2305	0.272775
6	7.0	0.2226	0.268195
7	8.0	0.2141	0.261970
8	9.0	0.2097	0.263305
9	10.0	0.2064	0.264042

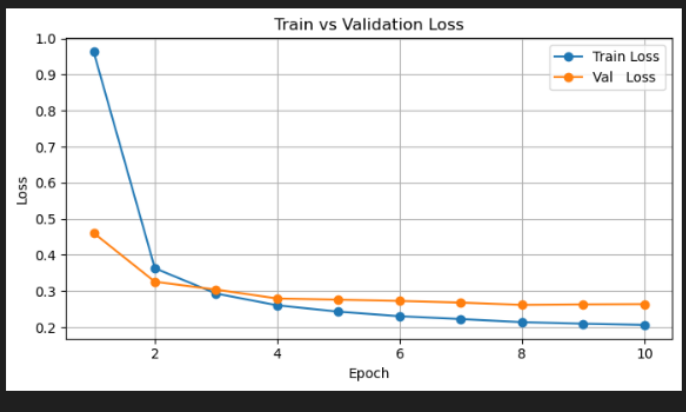


Figure: 13:Training vs Validation Loss of Segformer-B2 Cityscapes Pretrained,fine-tuned on IDD-lite model

The training and validation loss curves for the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model are shown in Figure: 13 . The model demonstrates a stable training process, with both loss values decreasing sharply in the initial epochs before converging. After 10 epochs, the training loss stabilized at a final value of 0.2064, while the validation loss converged to 0.264042.

The validation loss curve closely follows the training loss curve, indicating that the model generalizes well and did not suffer from significant overfitting.

In terms of complexity, the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model has 27,352,007 trainable parameters, making it the largest and most complex model in this experiment.

8.1.4 Combined Comparative Analysis with Qualitative Visual Analysis of Experiment 1

This section synthesizes the findings from the three SegFormer variants to draw overall conclusions about the effect of backbone/encoder on performance for the IDD-Lite dataset. The analysis will show a clear relationship between the number of model parameters and segmentation accuracy.

- **Overall Performance Analysis**

The results from this experiment demonstrate a clear and direct relationship between model size and segmentation performance. As shown in Table 8, the largest model, the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model , consistently outperformed the smaller variants across all primary accuracy metrics. This indicates that the model's increased parametric capacity was a key factor in achieving superior results on the complex IDD-Lite dataset.

The most critical metric, mean IoU, increased with each step up in model size, with the most significant gain of 2.88 percentage points occurring between the B1 and B2 models. This suggests that the 27.3 million parameters of the the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model allowed it to learn more robust and detailed features compared to the smaller B0 and B1 architectures.

This trend of superior performance is not just evident in the final scores, it was consistent throughout the entire training process. The progression charts for both Mean IoU and Pixel Accuracy in Figure: 14 provide powerful evidence for this. They clearly show a performance hierarchy where the the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model achieved a higher score at every single stage of training. This indicates that its larger parametric capacity led to more effective and consistent learning, not just a better final result.

Furthermore, a significant finding was observed in the inference speed. Counter-intuitively, the largest model, the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model, was also the fastest at 10.696 FPS. This anomaly suggests that its architecture is highly optimized for the parallel processing capabilities of modern GPUs, challenging the typical assumption that increased model size must come at the cost of speed.

Model Variant	Mean IoU	Pixel Accuracy	Inference Speed (FPS)	Total Parameters

SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model	0.675168	0.8849	8.163	3,715,943
SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model	0.694255	0.892985	8.022	13,679,047
SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model	0.723008	0.902447	10.696	27,352,007

Table 8:Combined Comparative analysis of SegFormer(b0,b1,b2), Cityscapes-pretrained, fine-tuned on IDD-Lite model Final Metrics

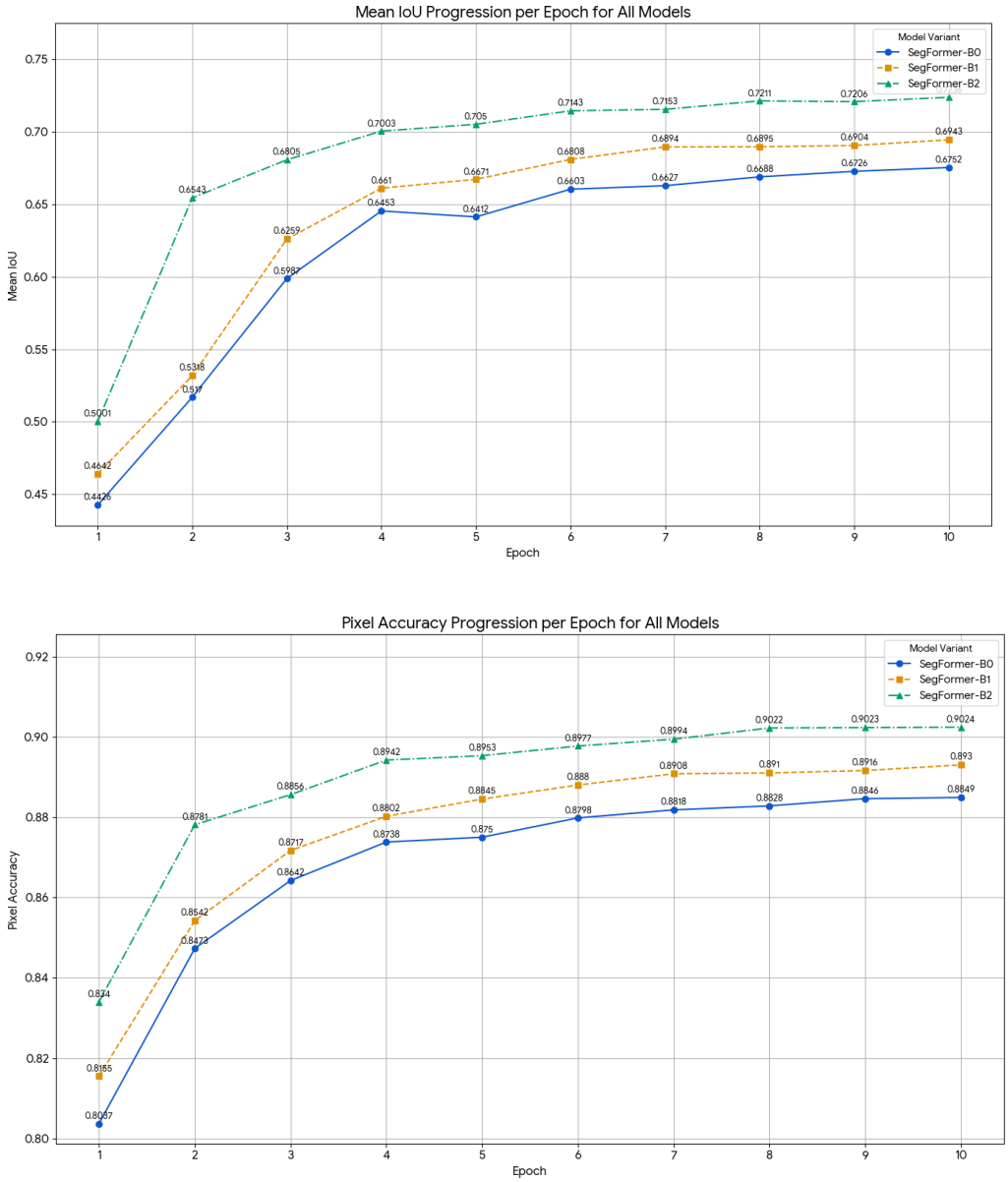


Figure: 14:Mean IOU and Pixel Accuracy per Epoch of the segformer (b0/b1/b2),Cityscapes Pretrained ,finetuned on IDD-lite Models

- **Per-Class Performance Analysis**

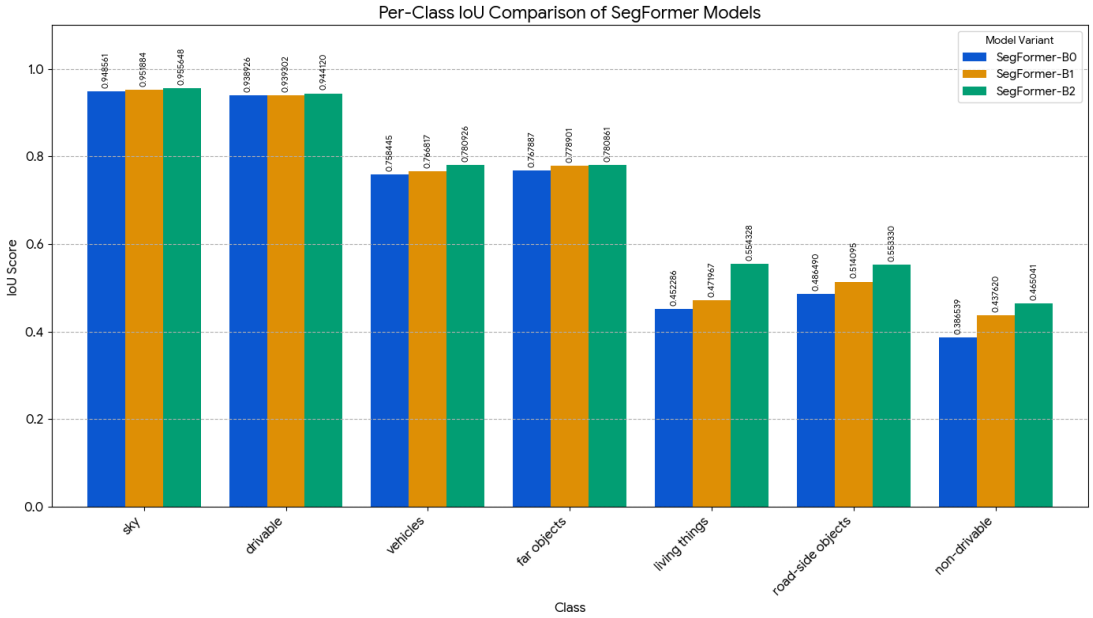


Figure: 15:Per Class IoU of Segformer (b0/b1/b2),Cityscapes Pretrained ,fine-tuned on IDD-lite models

The benefits of increased model size become even more apparent when analyzing the per-class performance, as visualized in the grouped bar chart in Figure: 15.

For the high-performing, visually dominant classes like 'sky' and 'drivable', all models achieved high scores, though the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model, still secured the top performance, showing even these classes benefit from a larger model. A similar trend of steady, modest improvement is seen in the well-defined classes of 'vehicles' and 'far objects', where the larger models provided greater precision.

However, the most significant gains occurred in the most challenging classes. The IoU score for 'living things' jumped from 0.452 for the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model to 0.554 for the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model, and 'road-side objects' improved from 0.486 to 0.553. Even the most difficult class, 'non-drivable' surfaces, saw consistent improvement with model size. This demonstrates that the greater parametric capacity of the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model was crucial for learning the subtle and varied features of these difficult but essential objects, a task where the smaller models were less successful.

- **Qualitative Visual Analysis of ground truth vs predicted semantic map of experiment 1**

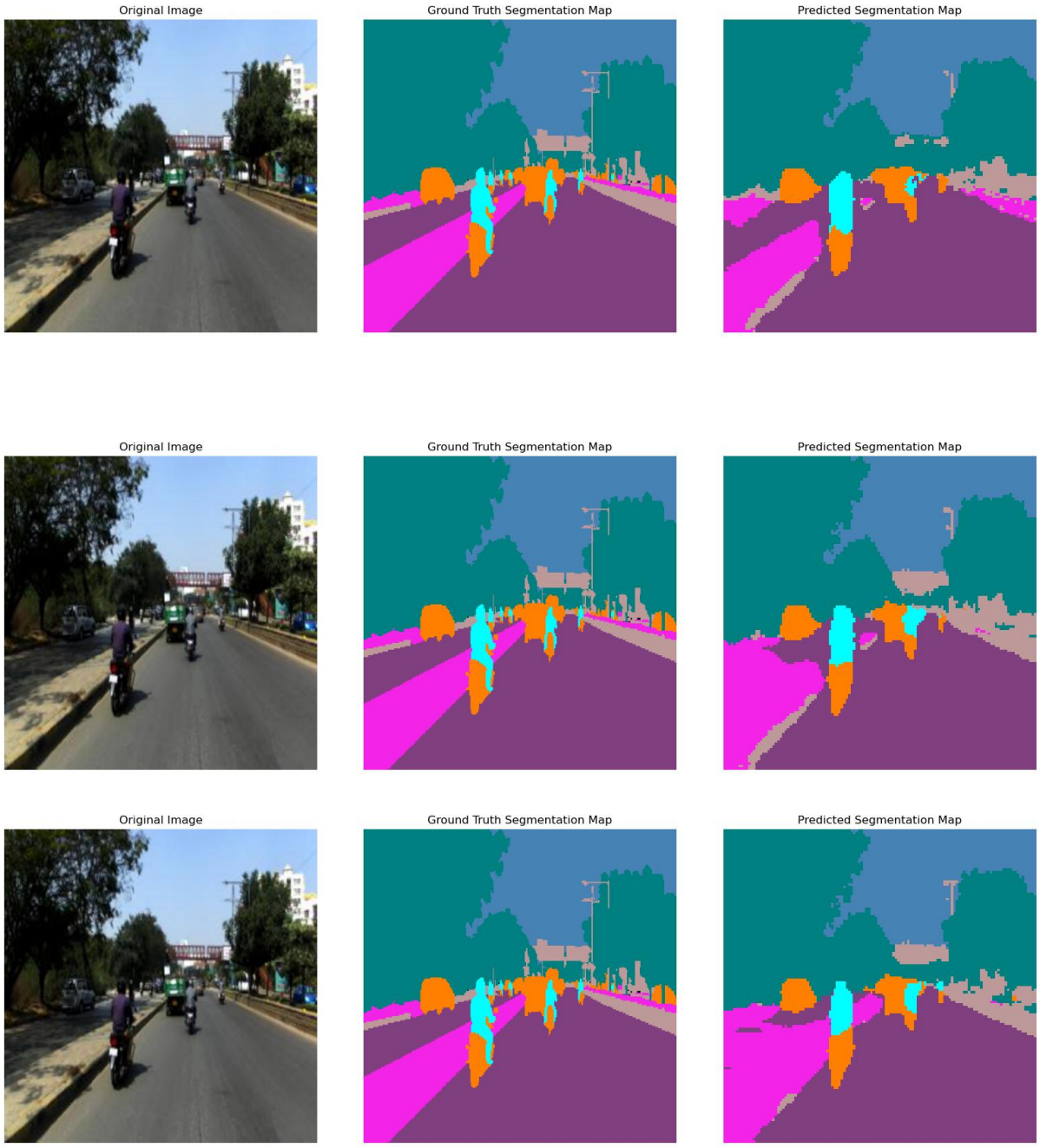


Figure: 16:Visualization of Ground Truth vs Predicted Semantic maps of Segformer (b0,b1,b2) Cityscapes Pretrained fine tuned on IDD-Lite Models

The quantitative improvements are strongly **c**onfirmed by the qualitative results, as shown in the side-by-side comparison in Figure: 16. The visualization confirms that while all models accurately segment the large, visually dominant classes like 'sky' and 'drivable' areas, the key differences emerge in the more challenging categories.

In Figure: 16

- The SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model which has the fewest parameters, is the least refined. It fails on a critical task by merging the

rider, belonging to the 'living things' class, with the motorcycle from the 'vehicles' class. Furthermore, it renders distant 'vehicles' and 'road-side objects' as indistinct shapes, lacking clear definition

- The SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model: The medium-sized model shows a noticeable improvement. The segmentation of the rider and other vehicles is cleaner and more defined than the SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model However, it still fails to capture the precise boundaries and smaller details, such as the space between the rider and the motorcycle.

- The SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model: The largest model provides the best result, which most closely resembles the ground truth. The primary reason for this is its 27.3 million parameters, which provide a greater capacity to learn the distinct features of these difficult, under-represented classes.

This allows the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model to not only correctly distinguish between 'living things' and 'vehicles' but also to produce sharper boundaries for 'road-side objects' and 'far objects', while more accurately classifying the 'non-drivable' surfaces. This visual evidence confirms that the increase in model size directly translated to a more refined and accurate understanding of the entire seven-class scene.

8.2 Experiment 2: SegFormer (with backbones B0/B1/B2) trained from scratch on IDD-Lite models

8.2.1 Run 1: SegFormer-B0 trained from scratch on IDD-Lite model

This section presents the results for the SegFormer-B0 trained from scratch on IDD-Lite model which was trained from scratch on the IDD-Lite dataset, without using any pre-trained weights from Cityscapes. The primary goal is to establish performance for a non-pre-trained model.

- **Overall Final Metrics**

	Mean IoU	Pixel Accuracy	Inference speed(fps) / eval_samples_per_second	parameters	Epochs
Final Metrics	**0.392878**	**0.757093**	**9.144**	**27,352,007**	**10**

Table 9:SegFormer-B0 trained from scratch on IDD-Lite model Final Metrics

The results from this experiment show a significant performance decrease when training from scratch. As shown in Table 9 , the model achieved a mean IoU of only 0.392078 and a pixel accuracy of 0.757093. This is a substantial drop compared to the segformer-b0 cityscapes-

pretrained, fine-tuned on IDD-Lite model and indicates that the model struggled to learn meaningful representations from the limited IDD-Lite dataset alone.

- **Final Metrics of IOU Per Class**

	Drivable IoU	**Non Drivable IoU**	**Living things IoU**	**Vehicles IoU**	**Road Side Object IoU**	**Far Objects IoU**	**Sky IoU**
Final Metrics IoU Per Class	**0.821836**	**0.000346**	**0.0**	**0.298823**	**0.125258**	**0.604977**	**0.898907**

Table 10: :SegFormer-B0 trained from scratch on IDD-Lite model Final Metrics of IoU per class

The severe impact of training from scratch is most evident in the per-class performance, detailed in Table 10. The model completely failed to learn the 'living things' class, resulting in an IoU of 0.0, and performed exceptionally poorly on 'non-drivable' surfaces at 0.000346 IoU. From the perspective of developing a safe autonomous driving system, this constitutes a critical failure. This result serves as powerful evidence that without the foundational knowledge provided by pre-training, the model lacked the capacity to overcome the severe **class imbalance** in the dataset and learn the features of these crucial minority classes.

- **Training vs validation loss of** SegFormer-B0 trained from scratch on IDD-Lite model

	epoch	train_loss	eval_loss
0	1.0	1.4289	1.067051
1	2.0	1.0197	0.954413
2	3.0	0.8772	0.819673
3	4.0	0.7954	0.774794
4	5.0	0.7409	0.744298
5	6.0	0.7036	0.716569
6	7.0	0.6764	0.689320
7	8.0	0.6565	0.672221
8	9.0	0.6392	0.666908
9	10.0	0.6297	0.659029

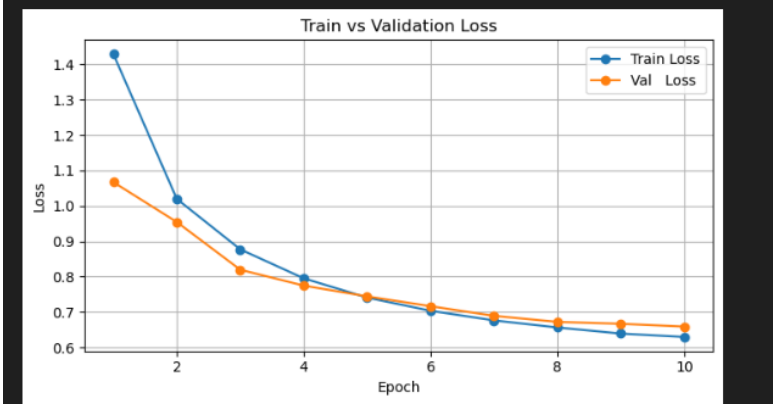


Figure: 17:Training vs Validation of Segformer-b0 trained from scratch on IDD-lite

Figure: 17 shows the training and validation loss curves for the model trained from scratch, revealing a significantly less effective learning process compared to the pre-trained models. The training converged at a much higher error value, with a final training loss of 0.6297 and a validation loss of 0.659029. The notable and persistent gap between the validation and training loss indicates that the model struggled to generalize its learning to unseen data. This difficulty in generalization is a clear consequence of training on a complex, imbalanced dataset without the foundational knowledge provided by pre-trained weights.

The model's architecture is identical to its segformer-b0 cityscapes-pretrained, fine-tuned on IDD-Lite model, with a total of 3,715,943 trainable parameters. This confirms that the dramatic difference in performance is due to the training strategy in the absence of pre-trained weights and not a difference in the model's size.

8.2.2 Run 2: SegFormer-B1 trained from scratch on IDD-Lite model

This section presents the results for the SegFormer-B1 model trained from scratch on the IDD-Lite dataset. The goal is to evaluate the performance of the intermediate-sized model without the benefit of pre-trained weights.

- **Overall Final Metrics**

	Mean IoU	**Pixel Accuracy**	**Inference speed(fps) / eval_samples_per_second**	**parameters**	**Epochs**
Final Metrics	**0.414711**	**0.771531**	**8.148**	**27,352,007**	**10**

Table 11::SegFormer-B1 trained from scratch on IDD-Lite model Final Metrics

Training the SegFormer-B1 trained from scratch on IDD-Lite model resulted in poor overall performance, though it was a slight improvement over the B0 model trained under the same conditions. As shown in Table 11, the model achieved a mean IoU of 0.414711 and a pixel accuracy of 0.771531.

	Drivable IoU	**Non Drivable IoU**	**Living things IoU**	**Vehicles IoU**	**Road Side**	**Far Objects IoU**	**Sky IoU**

					Object IoU		
Final Metrics IoU Per Class	**0.838495**	**0.000346**	**0.0**	**0.298823**	**0.125258**	**0.604977**	**0.898907**

- **Final Metrics of IOU Per Class**

Table 12::SegFormer-B1 trained from scratch on IDD-Lite model Final metrics of IoU per class

The per-class IoU scores, detailed in Table 12, confirm the severe impact of training from scratch. Similar to SegFormer-B0 trained from scratch on IDD-Lite model , the B1 variant completely failed to learn the 'living things' class, achieving an IoU of 0.0. This is a critical failure that renders the model unsuitable for a real-world autonomous driving application. The poor performance across all minority classes serves as powerful evidence that without the foundational knowledge from pre-training, even a larger model cannot overcome the severe class imbalance in the dataset.

- **Training vs Validation Loss of SegFormer-B1 trained from scratch on IDD-Lite model**

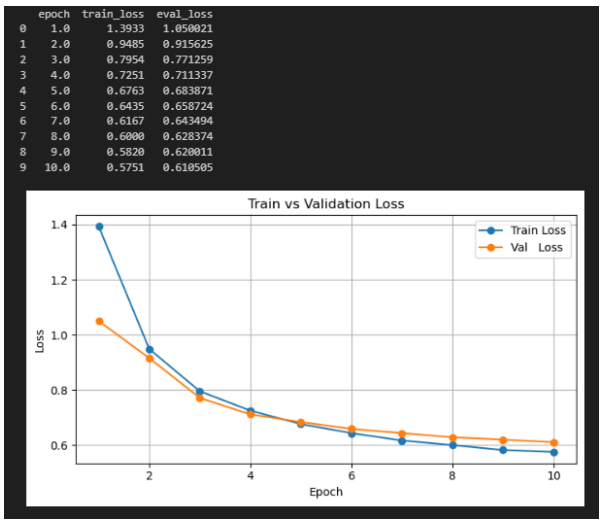


Figure: 18: Training vs Validation of Segformer-b1 trained from scratch on IDD-lite

The training and validation loss curves in Figure: 18 show a similar pattern to the B0 trained from scratch model, converging at a high loss value and indicating a struggle to generalize effectively. The final training loss was 0.5751, while the final validation loss was significantly higher at 0.610505. The persistent gap between the two curves confirms that the model had difficulty generalizing its learning from the training data to the unseen validation data.

The model's architecture is identical to segformer-b1 cityscapes-pretrained, fine-tuned on IDD-Lite model, with a total of 13,679,047 trainable parameters. This confirms that the poor

performance is a direct result of the training strategy in the absence of pre-trained weights, and not a difference in the model's capacity. A qualitative analysis would visually confirm the quantitative results, showing a complete failure to identify any instances of the 'living things' class.

8.2.3 Run 3: SegFormer-B2 trained from scratch on IDD-Lite model

This section presents the results for the largest model, SegFormer-B2, when trained from scratch on the IDD-Lite dataset. The goal is to see if the model's significantly larger capacity can overcome the lack of pre-trained knowledge.

- **Overall Final Metrics**

	Mean IoU	**Pixel Accuracy**	**Inference speed(fps) / eval_samples_per_second**	**parameters**	**Epochs**
Final Metrics	**0.447861**	**0.787059**	**8.065**	**27,352,007**	**10**

Table 13::SegFormer-B2 trained from scratch on IDD-Lite model Final Metrics

Even with its large size, the B2 model trained from scratch performed poorly. As shown in Table 13, it achieved a mean IoU of 0.447861 and a pixel accuracy of 0.787059. While this is the best result of the three models trained from scratch, it is still dramatically worse than even the SegFormer-B0 trained from scratch on IDD-Lite model.

	Drivable IoU	**Non Drivable IoU**	**Living things IoU**	**Vehicles IoU**	**Road Side Object IoU**	**Far Objects IoU**	**Sky IoU**
Final Metrics	**0.854895**	**0.105355**	**0.014667**	**0.393024**	**0.199819**	**0.65018**	**0.917086**

IoU Per Class							

- **Final Metrics of IOU Per Class**

Table 14::SegFormer-B2 trained from scratch on IDD-Lite model Final metrics of IoU per Class

The per-class IoU scores, detailed in Table 14, show that even the largest model could not learn the most challenging classes without pre-training. Performance on 'living things' at 0.014667 IoU and 'non-drivable' at 0.105355 IoU was exceptionally poor. This provides the strongest evidence yet that model size alone cannot compensate for the lack of foundational knowledge from pre-training when dealing with a complex and imbalanced dataset.

- **Training vs Validation Loss of SegFormer-B2 trained from scratch on IDD-Lite model**

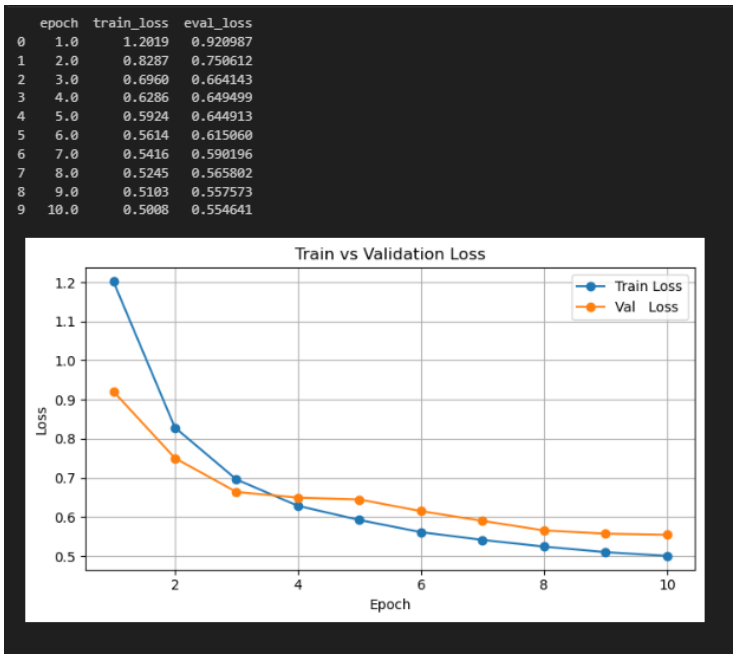


Figure: 19:Training vs Validation of Segformer-b2 trained from scratch on IDD-lite

The training and validation loss curves in Figure: 19 show that the model converged at a high loss value, with a final validation loss of 0.554641. The persistent and significant gap between the higher validation loss and the lower training loss suggests the model was beginning to overfit the training data, failing to generalize well to unseen data.

The model's architecture has 27,352,007 trainable parameters. The fact that this very large model performed so poorly is a powerful argument for the importance of transfer learning.

8.2.4 Combined Comparative Analysis with Qualitative Visual Analysis of Experiment 2

This section synthesizes the findings from the three SegFormer variants B0, B1, and B2 that were trained from scratch on IDD-Lite. The goal is to evaluate the effect of model scale on performance when no pre-trained knowledge is used. The results provide a powerful argument for the necessity of transfer learning.

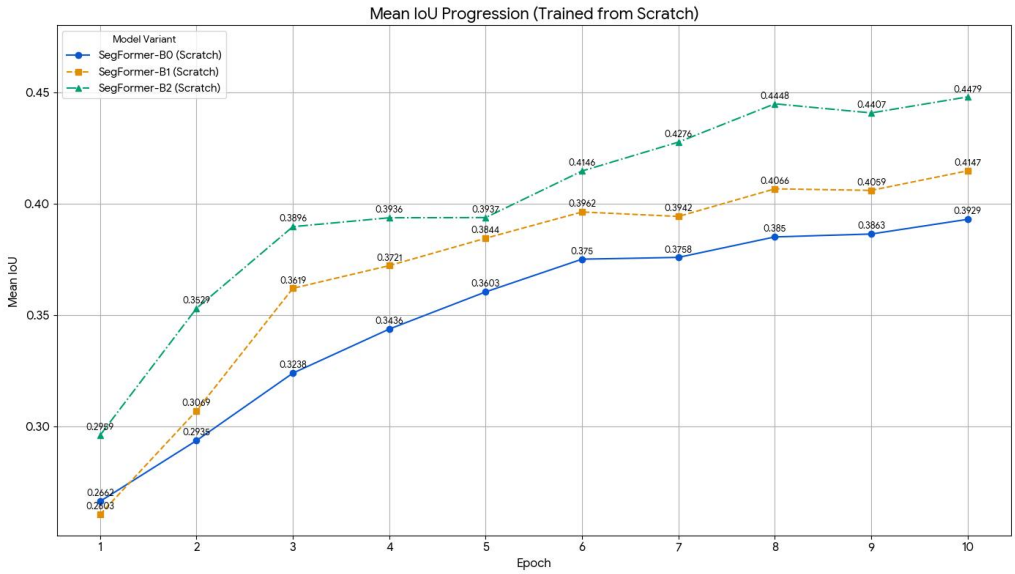
- **Overall Performance Analysis**

The comparative results for the models trained from scratch, shown in Table 15, reveal a consistent but marginal increase in performance with model size. The largest model, SegFormer-B2 trained from scratch on IDD-Lite model, achieved the highest mean IoU of 0.447861, but this represents only a modest improvement over the smaller variants. This trend of a slight but consistent performance hierarchy is also evident throughout the entire training process, as shown in the Mean IoU and Pixel Accuracy in Figure: 20 progression charts.

Crucially, this small gain in accuracy came at a direct cost to efficiency. The models here followed the expected speed-accuracy trade-off, with the inference speed decreasing as the number of parameters increased. The overall low scores across all metrics reinforce the central conclusion that pre-training is the most critical factor for success on this dataset.

Model Variant	Mean IoU	Pixel Accuracy	Inference Speed (FPS)	Total Parameters
SegFormer-B0 trained from scratch on IDD-Lite model	0.392078	0.757093	9.144	3,715,943
SegFormer-B1 trained from scratch on IDD-Lite model	0.414711	0.771531	8.148	13,679,047
SegFormer-B2 trained from scratch on IDD-Lite model	0.447861	0.787059	8.065	27,352,007

Table 15:Combined Comparative analysis of SegFormer(b0,b1,b2), trained from scratch on IDD-Lite model Final Metrics



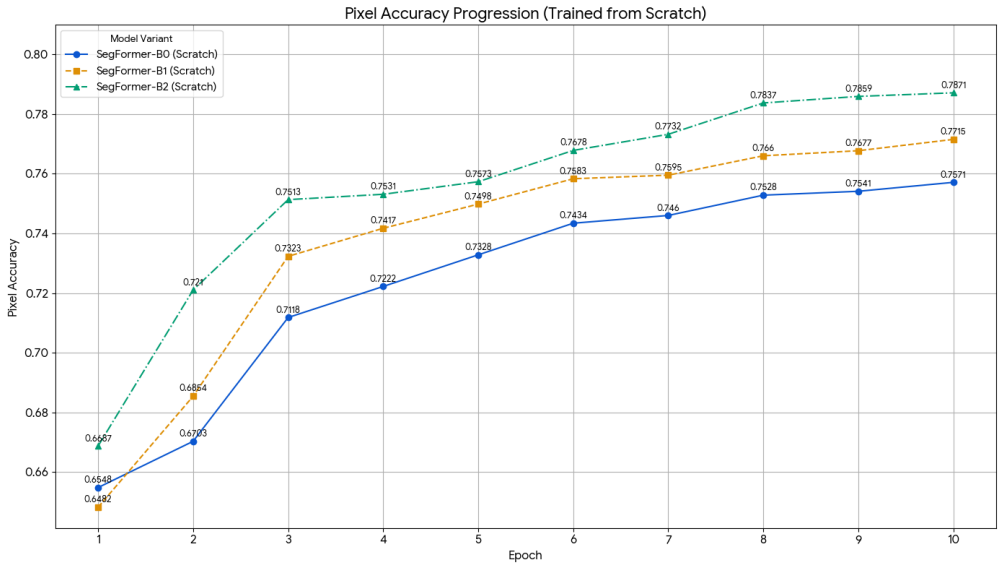


Figure: 20:Mean IoU and Pixel Accuracy of Segformer(b0/b1/b2) trained from scratch on IDD-lite

- **Per-Class Performance Analysis**

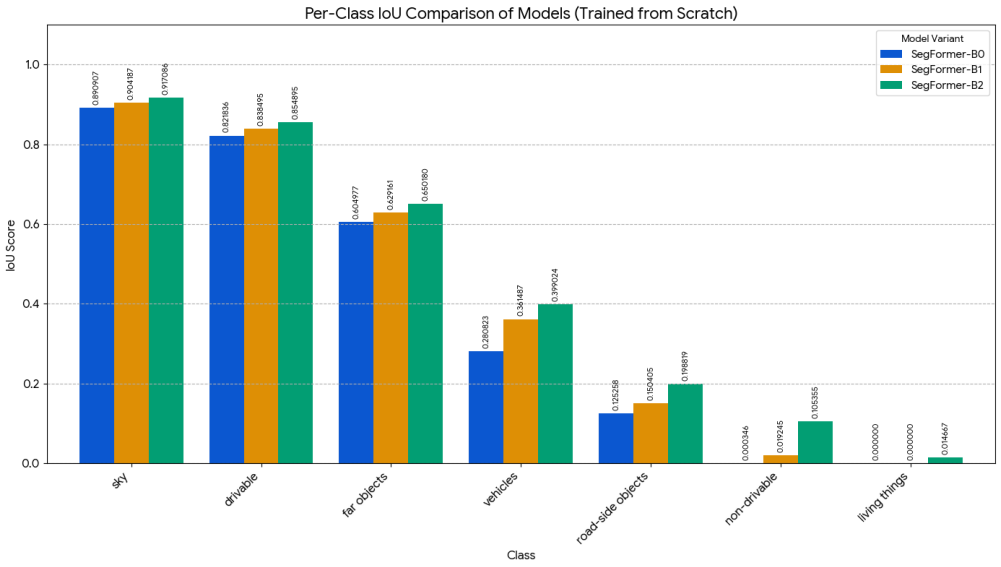
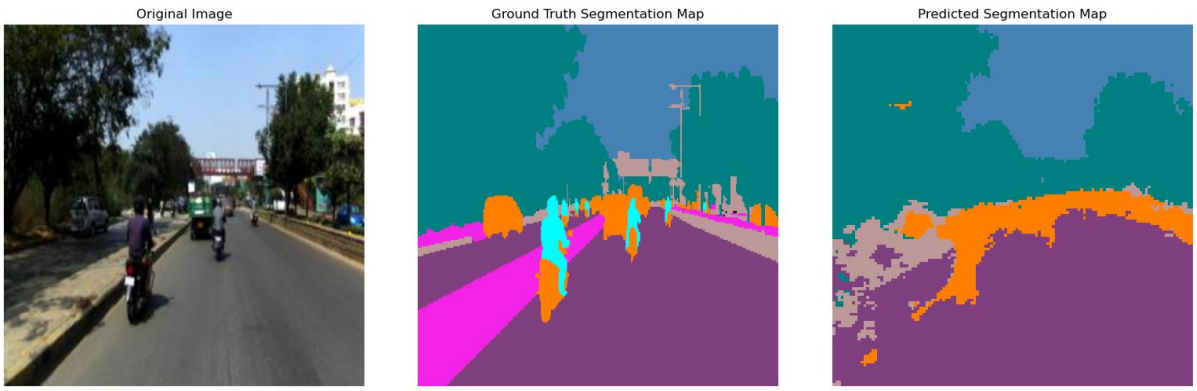


Figure: 21:Per Class IoU of Segformer (b0/b1/b2),trained from scratch on IDD-lite models

The grouped bar chart in Figure: 21 provides visualization of the challenges of training from scratch. The most dramatic finding is the failure on the 'living things' class, which contains safety-critical objects like pedestrians and riders. Both the SegFormer-B0 and SegFormer-B1 models completely failed to learn this class, scoring an IoU of 0.0, while the B2 model managed only a negligible 0.014667 IoU. This critical failure confirms that without a pre-trained foundation, none of the models could learn to reliably identify these vulnerable road users.

- **Qualitative Visual Analysis of ground truth vs predicted semantic map of experiment 2**



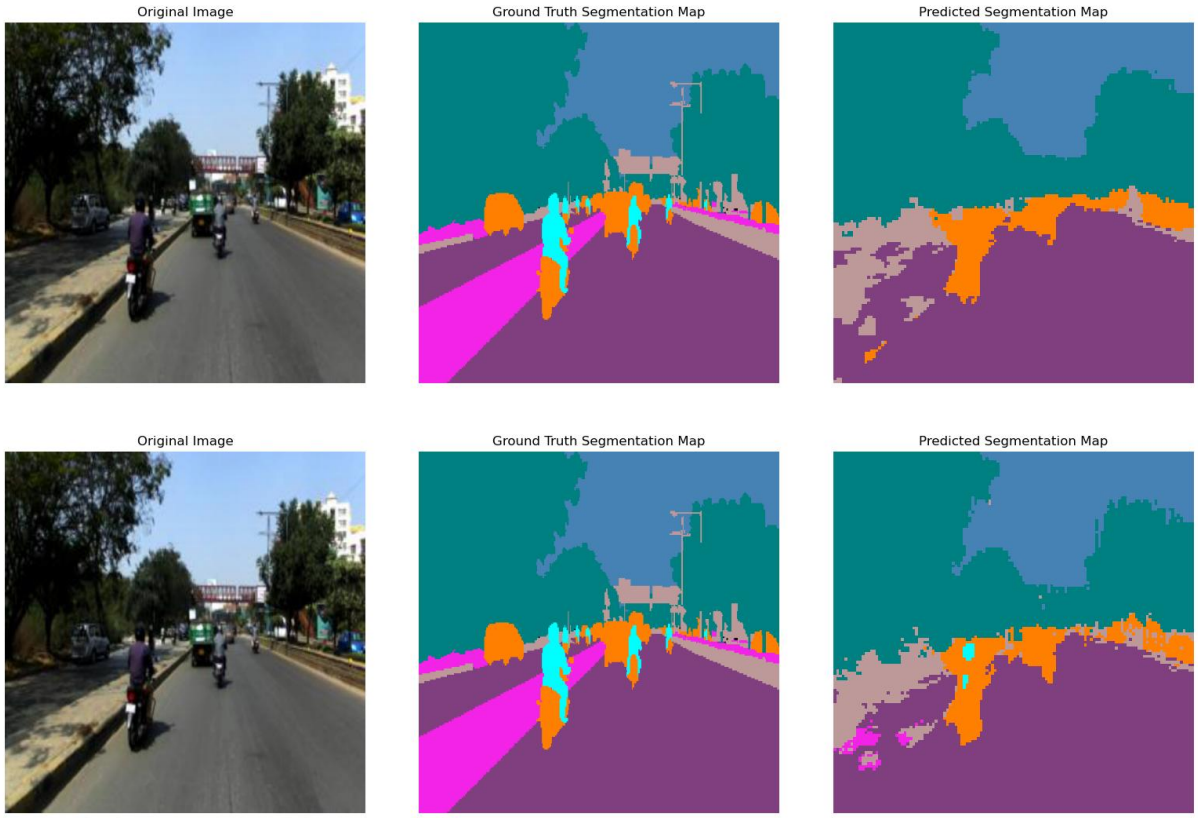


Figure: 22:Visualization of Ground Truth vs Predicted Semantic maps of Segformer (b0,b1,b2) trained from scratch on IDD-Lite Models

The qualitative comparison in Figure: 22 provides a stark visualization of the results of training from scratch segformer b0 b1 and b2 on IDD-Lite, confirming the poor quantitative scores.

- The SegFormer-B0 trained from scratch on IDD-Lite model, which has the fewest parameters, produces a chaotic and inaccurate segmentation. It completely fails to identify the rider as 'living things' class and renders 'vehicles' as fragmented, unrecognizable shapes.

- The SegFormer-B1 trained from scratch on IDD-Lite model shows no meaningful improvement. Despite its larger size, it also completely fails to detect the 'living things' class and produces an equally fragmented and noisy segmentation map for other objects.

- The SegFormer-B2 trained from scratch on IDD-Lite model, despite being the largest, also fails on the critical task of identifying the rider. Its output remains chaotic and does not resemble the ground truth in any meaningful way for the complex classes.

This visual evidence provides a powerful and definitive confirmation of the quantitative data. Unlike in experiment 1, an increase in model size provided no significant practical benefit. The universal failure across all three models proves that using pre-trained weights is an essential requirement to achieve competent segmentation on this dataset, as model size alone is insufficient.

8.3 Experiment 3: SegFormer- (with backbones B0/B1/B2) Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function

8.3.1 Comparative Analysis with Qualitative Visual Analysis of Experiment 3

This section synthesizes the findings from the three SegFormer variants B0, B1, and B2 cityscapes pre-trained checkpoints from hugging face and then fine-tuned using Focal Loss on IDD-Lite. The goal is to evaluate if this loss function, designed to combat class imbalance, improves performance on challenging minority classes.

- **Overall Performance Analysis**

The summary of results in Table 16 shows a familiar trend: performance improves with model size. The SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function achieved the highest mean IoU of 0.724108, confirming it as the best-performing model in this experiment. The progression charts for both Mean IoU and Pixel Accuracy in Figure: 23 further illustrate this, providing powerful evidence that the B2 model consistently outperformed the other two at every epoch of the training process. This confirms that even when using an advanced loss function, a larger model capacity is key to achieving the best results.

Model Variant	Mean IoU	Pixel Accuracy	Inference Speed (FPS)	Total Parameters
SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function	0.675081	0.884184	8.32	3,715,943
SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite	0.700411	0.894858	8.368	13,679,047

model with focal loss function				
SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function	0.724108	0.902748	8.515	27,352,007

Table 16::Combined Comparative analysis of SegFormer(b0,b1,b2), Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss Final Metrics

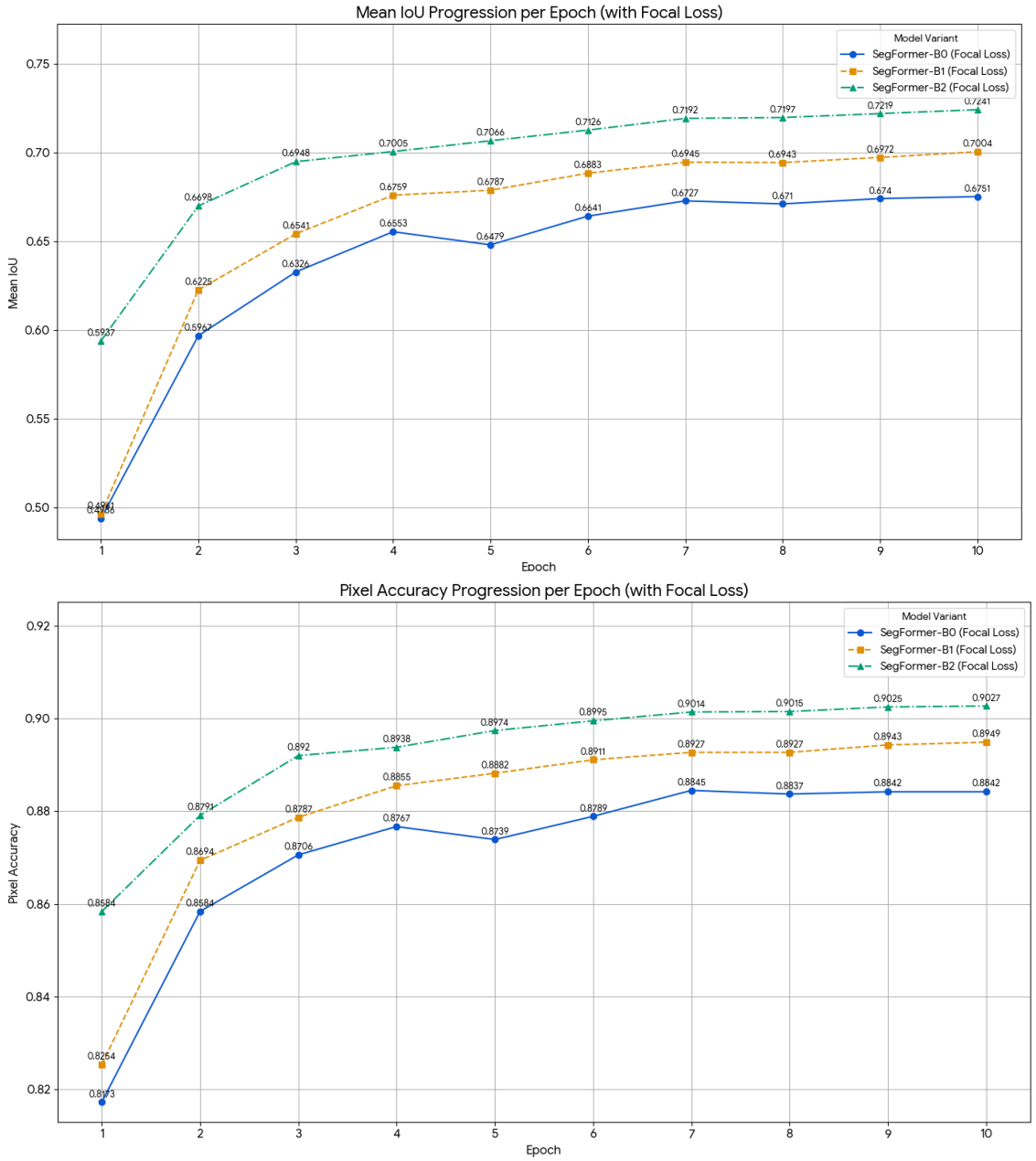


Figure: 23: Mean IoU and Pixel Accuracy of segformer (b0,b1,b2), Cityscapes Pretrained,fine tuned on IDD-lite models with focal loss

- **Analysis of Per-Class Performance with Focal Loss**

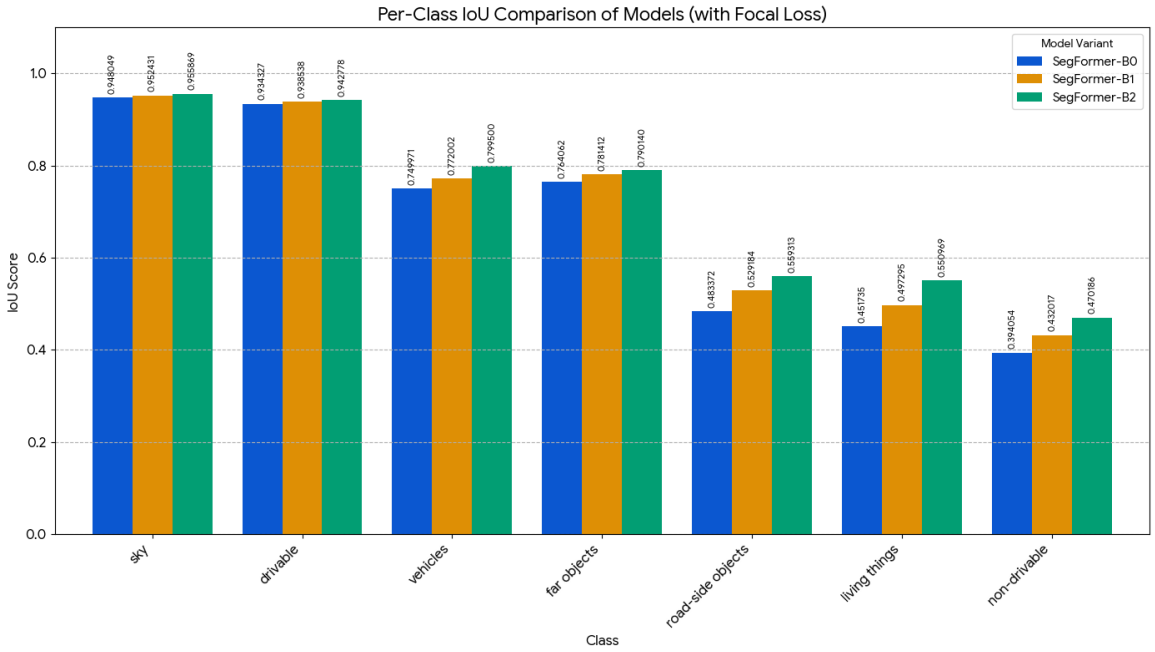


Figure: 24:Per Class IoU of Segformer (b0/b1/b2),Cityscapes ptrained,fine-tuned on IDD-lite models with focal loss

This chart in Figure: 24 visualizes the performance of the models when trained with a loss function specifically designed to address class imbalance.

- **Performance Hierarchy**: The chart shows the same familiar trend where performance scales with model size. The SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function consistently outperforms the B1 and B0 models across all seven classes.

- **Impact on Minority Classes**: The primary goal of using Focal Loss was to improve performance on challenging, under-represented classes. With Focal Loss, the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function achieved an IoU of 0.550969 on the 'living things' class. This is a respectable score but does not represent a significant improvement over the standard loss function from Experiment 1 which scored 0.554328.

Conclusion for this indicates that while Focal Loss is a valid strategy for class imbalance, its positive impact in this experiment was marginal. The results suggest that the architectural capacity of the SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function itself was the most critical factor in handling these difficult classes, more so than the specific loss function used.

- **Comparative Analysis of Training and validation loss of SegFormer- (with backbones B0/B1/B2) Cityscapes-pretrained, fine-tuned on IDD-Lite model with Focal Loss**

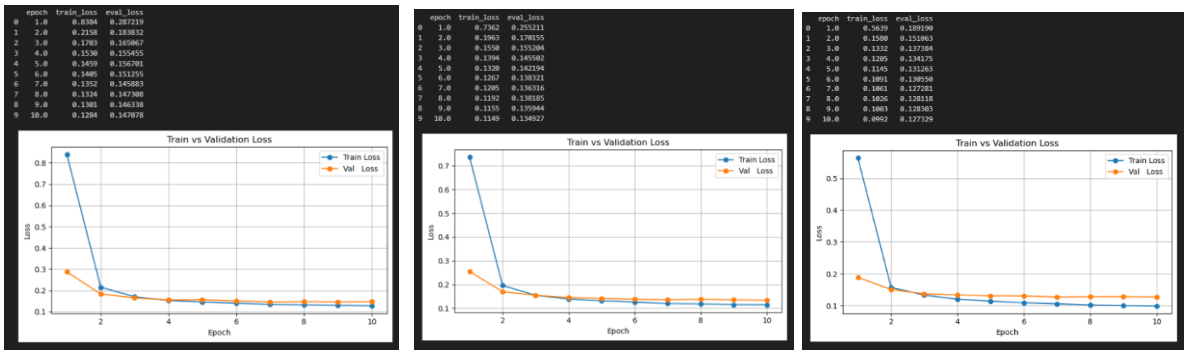


Figure: 25: Training and Validation of Segformer (b0/b1/b2),Cityscapes ptrained,fine-tuned on IDD-lite models with focal loss

A side-by-side comparison of the training and validation loss curves for all three models B0, B1, and B2 in Figure: 25 reveals a universally effective and stable training process when using focal loss. All three models demonstrate rapid convergence in the early epochs before stabilizing with a very small gap between their training and validation loss curves, indicating that none suffered from significant overfitting.

However, a clear hierarchy is visible, with the largest model, The SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function, consistently achieving the lowest validation loss at every epoch and converging at the lowest value of validation loss of 0.127329. This provides powerful evidence that even with an advanced loss function, the greater capacity of the larger model allows it to find a more optimal and effective solution.

- **Qualitative Visual Analysis of ground truth vs predicted semantic map of experiment 3**

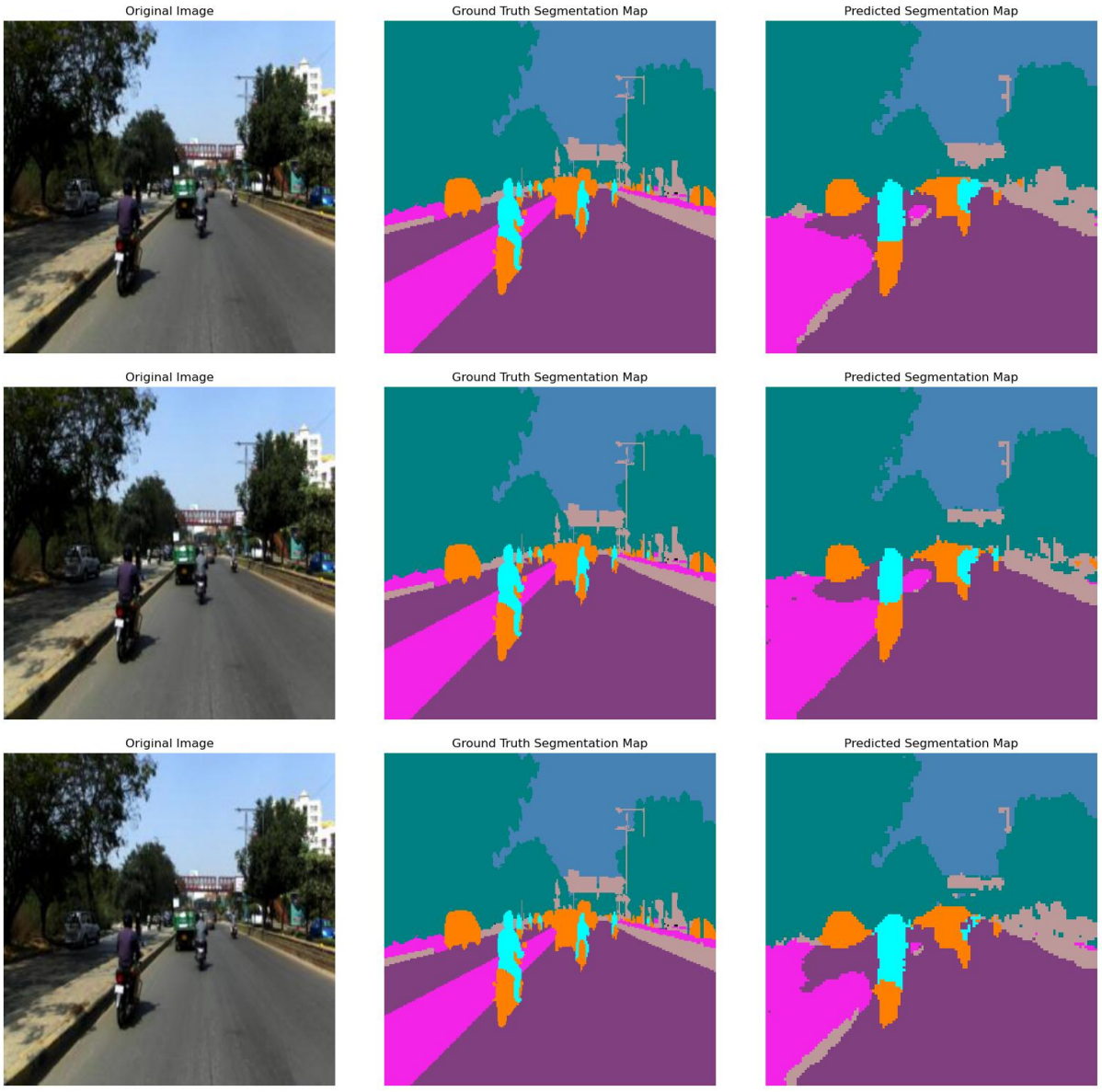


Figure: 26: Visualization of Ground Truth vs Predicted Semantic maps of Segformer (b0,b1,b2) Cityscpaes pretrained ,finetuned on IDD-Lite Models with focal loss

The qualitative comparison in Figure: 26 provides a detailed confirmation of the quantitative results, revealing a clear hierarchy in segmentation quality that directly correlates with model size.

- The SegFormer-B0 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function output is the least refined, it not only fails to cleanly separate the rider in 'living things' class from the motorcycle in 'vehicles' class, but also produces a noisy and fragmented segmentation for distant 'vehicles' and misclassifies parts of the 'non-drivable' class.

- The intermediate SegFormer-B1 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function offers a noticeable improvement with cleaner boundaries for the primary 'vehicles', but it still shows considerable noise on the 'non-drivable' surfaces and lacks the precision to fully resolve the rider.

In contrast, the The SegFormer-B2 Cityscapes-pretrained, fine-tuned on IDD-Lite model with focal loss function produces a segmentation map that is visibly superior and most closely resembles the ground truth . Its larger parametric capacity allows it to achieve a much cleaner segmentation of the 'non-drivable' areas, provide sharper boundaries for 'road-side objects', and, most critically, it successfully distinguishes between the 'living things' and 'vehicles' classes. This detailed visual evidence confirms that while Focal Loss contributes to a stable training process, the architectural size of the SegFormer-B2 model is the primary reason for its ability to produce a more accurate and refined understanding of the entire scene.

8.4 Experiment 4: mask2former (with backbones swin-tiny/swin-small) : Cityscapes-Pretrained, Fine-Tuned on IDD-Lite models

8.4.1 Run 1: Mask2former swin-tiny Cityscapes-pretrained, fine-Tuned on IDD-Lite model

This section presents the results for the Mask2Former model with a Swin-Tiny backbone. This model uses pre-trained Cityscapes checkpoint from hugging face and fine-tuned on IDD-Lite. The goal is to establish the performance of this different architectural approach.

- **Overall final metrics**

	Mean IoU	Pixel Accuracy	Inference speed(fps) / eval_samples_per_second	parameters	Epochs
Final Metrics	**0.755368**	**0.91235**	**3.1**	**27,352,007**	**10**

Table 17:Mask2former Swin-tiny Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics

The Mask2Former Swin-Tiny model demonstrated very strong performance. As shown in Table 17, it achieved a mean IoU of 0.755368 and a high pixel accuracy of 0.91235. Its inference speed was 3.1 frames per second (FPS**)**. These results, particularly the mean IoU, are significantly higher than the SegFormer with backbones b0,b1 and b2 Cityscapes-pretrained, fine-tuned on IDD-Lite models, suggesting a superior architectural design for this task, but this comes at a considerable cost to inference speed.

- **Overall final metrics of IoU Per Class**

	Drivable IoU	Non Drivable IoU	Living things IoU	Vehicles IoU	Road Side Object IoU	Far Objects IoU	Sky IoU
Final Metrics IoU Per Class	**0.953305**	**0.488607**	**0.653312**	**0.843985**	**0.584484**	**0.802236**	**0.96165**

Table 18:: Mask2former Swin-tiny Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics IoU Per Class

The per-class IoU scores, detailed in Table 18, show a remarkable improvement over the SegFormer models, especially in the most challenging minority classes. This is where the architectural differences of Mask2Former become most apparent.

The IoU for 'living things' reached 0.653312, a score nearly 10 percentage points higher than the best SegFormer model. This is a critical finding, as it demonstrates that the Mask2Former's mask-based classification approach is significantly more effective at overcoming the class imbalance problem and identifying crucial, under-represented objects like pedestrians and riders.

Furthermore, strong performance is seen across other difficult categories. The model achieves a high IoU on 'vehicles' at 0.843985 and 'road-side objects' at 0.584484, showing a more robust ability to define object boundaries compared to the Segformer variants. While it still scores lowest on the visually ambiguous 'non-drivable' class at 0.488607, the score is substantially higher than any of the previous models.

- **Training vs Validation Loss of Mask2former swin-tiny Cityscapes-pretrained, fine-Tuned on IDD-Lite model**

	epoch	train_loss	eval_loss
0	1.0	49.3663	32.385559
1	2.0	31.1953	28.793482
2	3.0	27.5512	26.178999
3	4.0	25.4500	25.501549
4	5.0	24.0476	24.450104
5	6.0	23.2554	23.874029
6	7.0	22.2326	23.910851
7	8.0	21.4743	23.565046
8	9.0	20.9852	23.494198
9	10.0	20.5324	23.379778

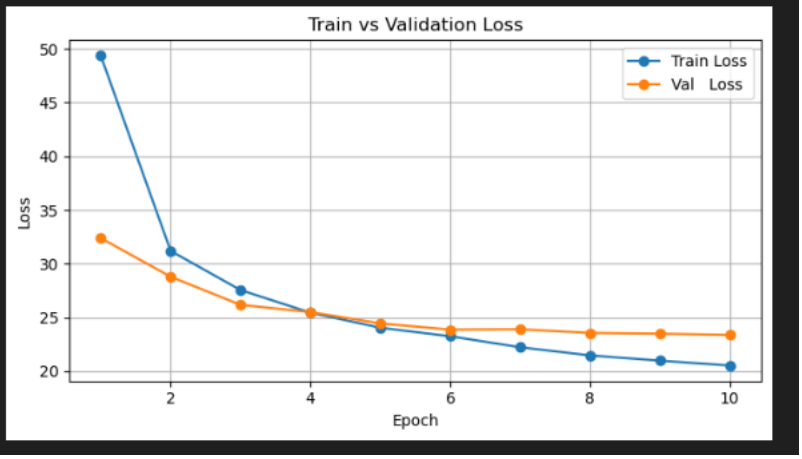


Figure: 27:training vs validation loss of Mask2former (swin-tiny) ,Cityscapes pretrained ,fine-tuned on IDD-lite model

The training and validation loss curves in Figure: 27 show a stable and effective learning process. The model converges to a final validation loss of 23.379778. The validation loss consistently tracks the training loss with a small, stable gap, which indicates that the model did not suffer from significant overfitting and generalizes well to unseen data.

The Mask2Former Swin-Tiny model has a total of 47,402,882 trainable parameters. This is substantially larger than even the largest SegFormer model, which helps explain its superior accuracy but also contributes to its much lower inference speed. A qualitative analysis would likely show a predicted semantic map with much sharper boundaries and a more accurate classification of small objects like pedestrians compared to the SegFormer models experimented so far.

8.4.2 Run 2: Mask2former swin-small Cityscapes-pretrained, fine-Tuned on IDD-Lite model

This section presents the results for the Mask2Former model with a Swin-Small backbone, which is larger than the Swin-Tiny variant. This model uses a pre-trained Cityscapes checkpoint and fine-tuned on IDD-Lite to evaluate the impact of a larger backbone on this architecture.

- **Overall final metrics**

	Mean IoU	Pixel Accuracy	Inference speed(fps) / eval_samples_per_second	parameters	Epochs
Final Metrics	**0.761012**	**0.916358**	**3.637**	**27,352,007**	**10**

Table 19:Mask2former Swin-Small Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics

The Mask2Former Swin-Small model achieved the highest accuracy scores of any model tested. As shown in Table 19, it reached a mean IoU of 0.761012 and a pixel accuracy of 0.916358. However, this increase in accuracy came at a significant cost to performance, with the inference speed dropping to 3.637 frames per second (FPS), the slowest of all pre-trained models.

- **Overall final metrics of IoU Per Class**

	Drivable IoU	Non Drivable IoU	Living things IoU	Vehicles IoU	Road Side Object IoU	Far Objects IoU	Sky IoU
Final Metrics IoU Per Class	**0.952834**	**0.472189**	**0.665156**	**0.850069**	**0.610715**	**0.813791**	**0.962328**

Table 20:Mask2former Swin-tiny Cityscapes-pretrained,fine-tuned on IDD-Lite model Final Metrics IoU per class

The per-class IoU scores, detailed in Table 20, demonstrate the superior performance of this larger model, particularly on the most difficult minority classes. The IoU for 'living things' reached an impressive 0.665156, and 'road-side objects' scored 0.610715, the highest scores achieved for these classes across all experiments. This provides strong evidence that the combination of the Mask2Former architecture with a larger backbone is highly effective at overcoming the class imbalance problem.

- **Training vs Validation Loss of Mask2former swin-small Cityscapes-pretrained, fine-Tuned on IDD-Lite model**

	epoch	train_loss	eval_loss
0	1.0	46.9514	30.509886
1	2.0	29.2344	26.832285
2	3.0	25.9630	25.539101
3	4.0	24.3544	24.565720
4	5.0	23.0151	24.337751
5	6.0	21.9577	23.805161
6	7.0	21.0188	23.311628
7	8.0	20.3047	23.067421
8	9.0	19.7783	23.104546
9	10.0	19.3143	22.871021

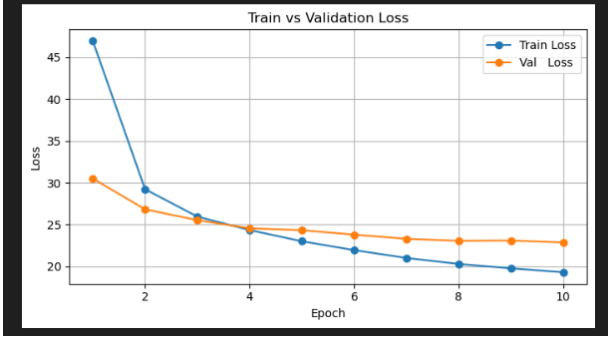


Figure: 28: training vs validation loss of Mask2former (swin-small) ,Cityscapes pretrained ,fine-tuned on IDD-lite model

The training and validation loss curves in Figure: 28 show a stable and effective learning process. The model converges to a final validation loss of 22.871021. The small and consistent gap between the training and validation loss indicates that the model did not suffer from significant overfitting and generalized well.

The Mask2Former Swin-Small Cityscapes pretrained model ,finetuned on IDD lite has a total of 68,720,786 trainable parameters, making it the largest and most complex model evaluated in this research. This large size is the primary reason for its high accuracy and its low inference speed.

8.4.3 Combined Comparative Analysis with Qualitative Visual Analysis of Experiment 4

This section synthesizes the findings from the two Mask2Former variants to draw conclusions about the effect of backbone size on this architecture's performance.

- **Overall Performance Analysis**

The comparison between the two Mask2Former variants in Table 21 reveals a clear case of diminishing returns regarding model size. the Mask2former swin-small Cityscapes-pretrained, fine-Tuned on IDD-Lite increases the model's complexity by 21.3 million parameters compared to the Swin-Tiny variant. However, this massive increase in complexity yielded a negligible performance gain; the mean IoU improved by only 0.005644, and the pixel accuracy saw a similarly small increase of 0.004008. This trend is confirmed by the learning progression charts in Figure: 29, which show the performance of both the Swin-Tiny and Swin-Small models tracking each other very closely throughout all 10 epochs.This minimal improvement across both metrics does not justify the significant increase in model size, indicating that the performance gains from further scaling the backbone**.**

Model Variant	Mean IoU	Pixel Accuracy	Inference Speed (FPS)	Total Parameters
Mask2former swin-tiny Cityscapes-pretrained, fine-Tuned on IDD-Lite model	0.755368	0.91235	3.1	47,402,882
Mask2former swin-small Cityscapes-pretrained, fine-Tuned on IDD-Lite model	0.761012	0.916358	3.637	68,720,786

Table 21:Combined Comparative analysis of Mask2Former(swin-tiny/swin-small), Cityscapes Pretrained,,fine tuned on IDD-Lite model Final Metrics

have become marginal.

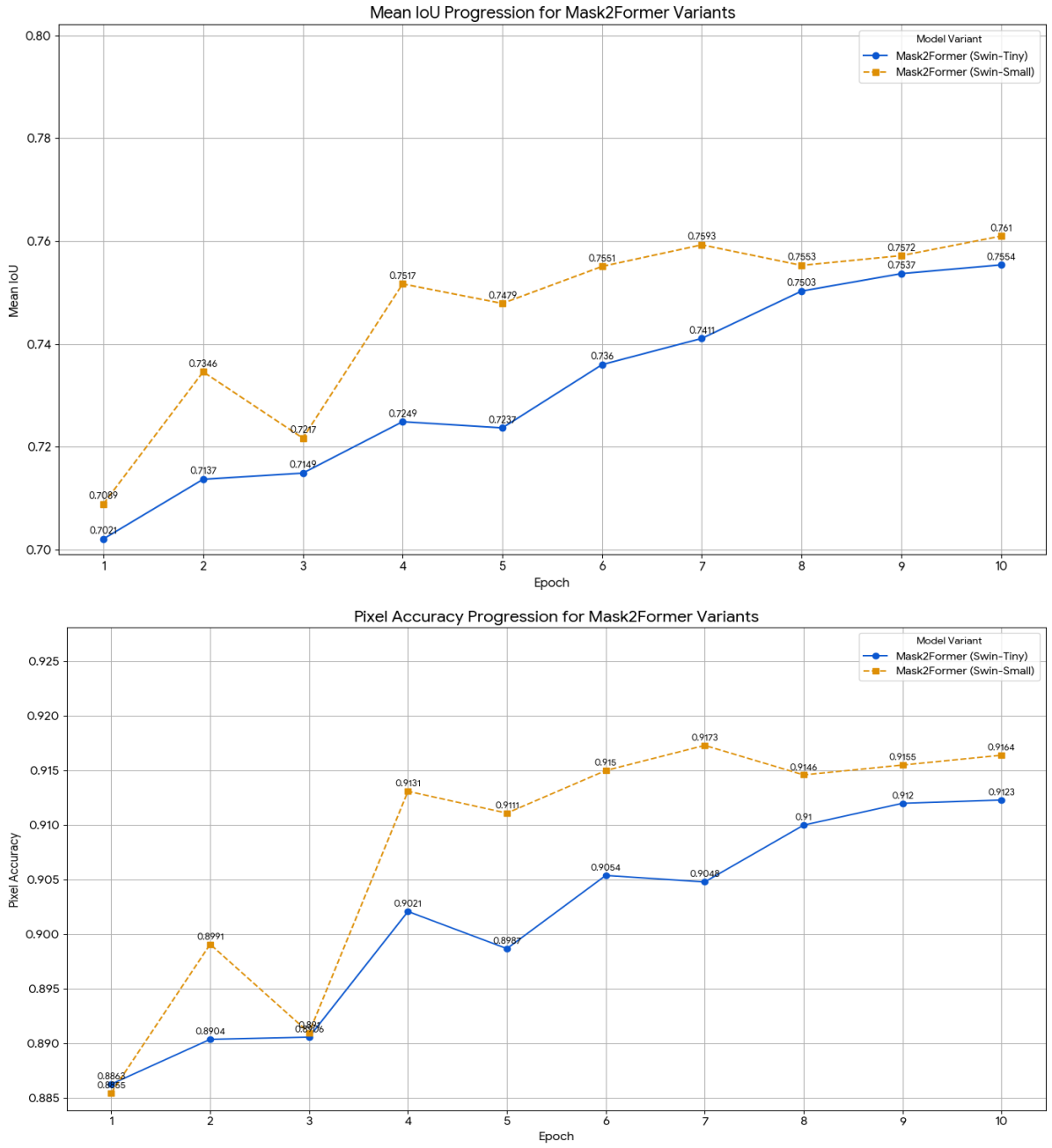


Figure: 29:Mean IoU and Pixel Accuracy of Mask2former (swin-tiny/swin-small) ,Cityscapes pretrained ,fine tuned on IDD-lite

- **Analysis of Per-Class Performance**

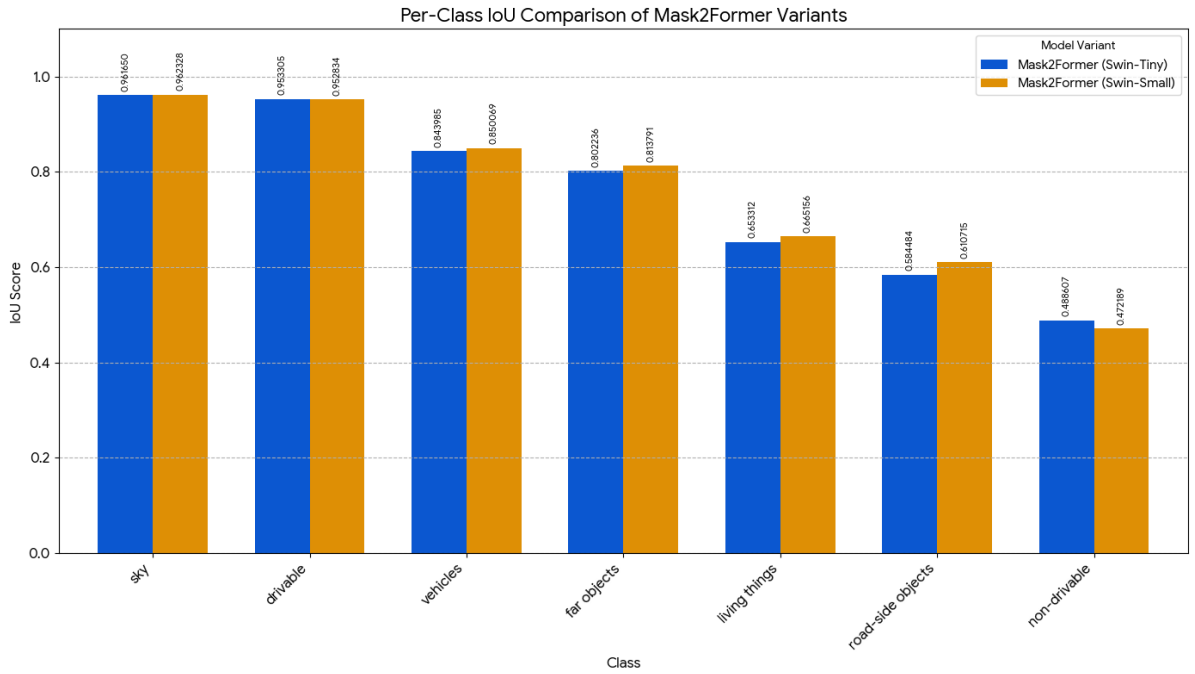


Figure: 30:Per-Class IoU of Mask2former (swin-tiny/swin-small) Cityscapes pretrained, ,fine tuned on IDD-Lite

Both the Swin-Tiny and Swin-Small variants that Cityscapes-pretrained, fine-tuned on IDD-Lite models in Figure: 30 demonstrate a high level of proficiency across all seven classes, confirming the superior capability of the Mask2Former architecture. For the dominant, high-performing classes like 'sky' and 'drivable', both models achieve excellent scores, with the Swin-Small backbone providing a negligible lead. Performance on well-defined classes like 'vehicles' and 'far objects' is also strong for both, with the Swin-Small again showing a minor improvement.

The key difference is seen in the most challenging categories. The larger Swin-Small model's greater parametric size allows it to achieve slightly higher scores on the safety-critical 'living things' class and on 'road-side objects'.

- **Qualitative Visual Analysis of ground truth vs predicted semantic map of experiment 4**

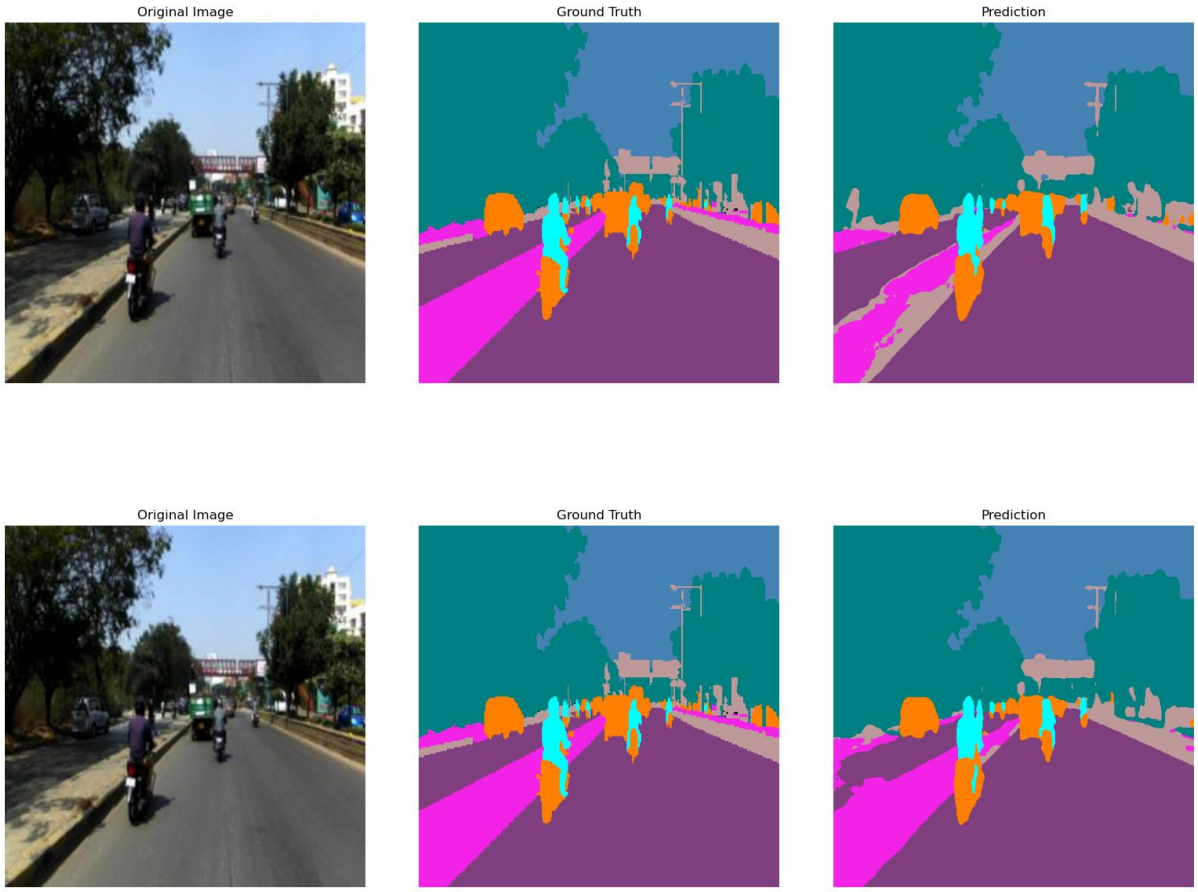


Figure: 31 Visualization of Ground Truth vs Predicted Semantic maps of Mask2former (swin-tiny/swin-small) , Cityscpaes pretrained ,finetuned on IDD-Lite Models

In Figure: 31

- The Swin-Tiny model already produces a very accurate and clean segmentation map. It correctly separates the rider in 'living things' class from the motorcycle in 'vehicles' class and provides sharp boundaries for most objects in the scene.

- The larger Swin-Small model offers subtle refinements over the Swin-Tiny. The boundaries around the distant 'vehicles' and 'road-side objects' are slightly sharper, and the classification of the 'non-drivable' shoulder is marginally cleaner.

the visual difference in quality between the Swin-Tiny and the Swin-Small model is very slight. This confirms that the massive increase in parameters for the Swin-Small backbone yielded a negligible improvement in visual quality, supporting the choice that the Swin-Tiny model is the more efficient and practical choice.

8.5 Experiment 5: Mask2former- (with backbones swin-tiny/swin-small) trained from scratch on IDD-Lite models.

8.5.1 Combined Comparative Analysis and Qualitative Visual Analysis of Experiment 5

This section synthesizes the findings from the two Mask2Former variants , Swin-Tiny and Swin-Small that were trained from scratch. The goal is to evaluate the effect of backbone size on this architecture's performance when no pre-trained knowledge is used.

- **Overall Performance analysis**

When trained from scratch, the Mask2Former models performed poorly, and increasing the model size provided a slight advantage. As shown in Table 22, the larger Swin-Small model achieved a higher mean IoU of 0.490991 and pixel accuracy of 0.797438. The learning progression charts Figure: 32 show unstable training for both models, with performance fluctuating significantly, though the Swin-Small variant consistently maintained a slight edge. The key takeaway is that without pre-training, both models struggled to learn effectively, and the larger model's complexity offered only a minor benefit.

Model Variant	Mean IoU	Pixel Accuracy	Inference Speed (FPS)	Total Parameters
Mask2former swin-tiny Cityscapes-pretrained, fine-Tuned on IDD-Lite model	0.475113	0.787097	4.18	47,402,882
Mask2former swin-small Cityscapes-pretrained, fine-Tuned on IDD-Lite model	0.490991	0.797438	3.414	68,720,786

Table 22:Combined Comparative analysis of Mask2Former(swin-tiny/swin-small), trained from scratch d on IDD-Lite model Final Metrics

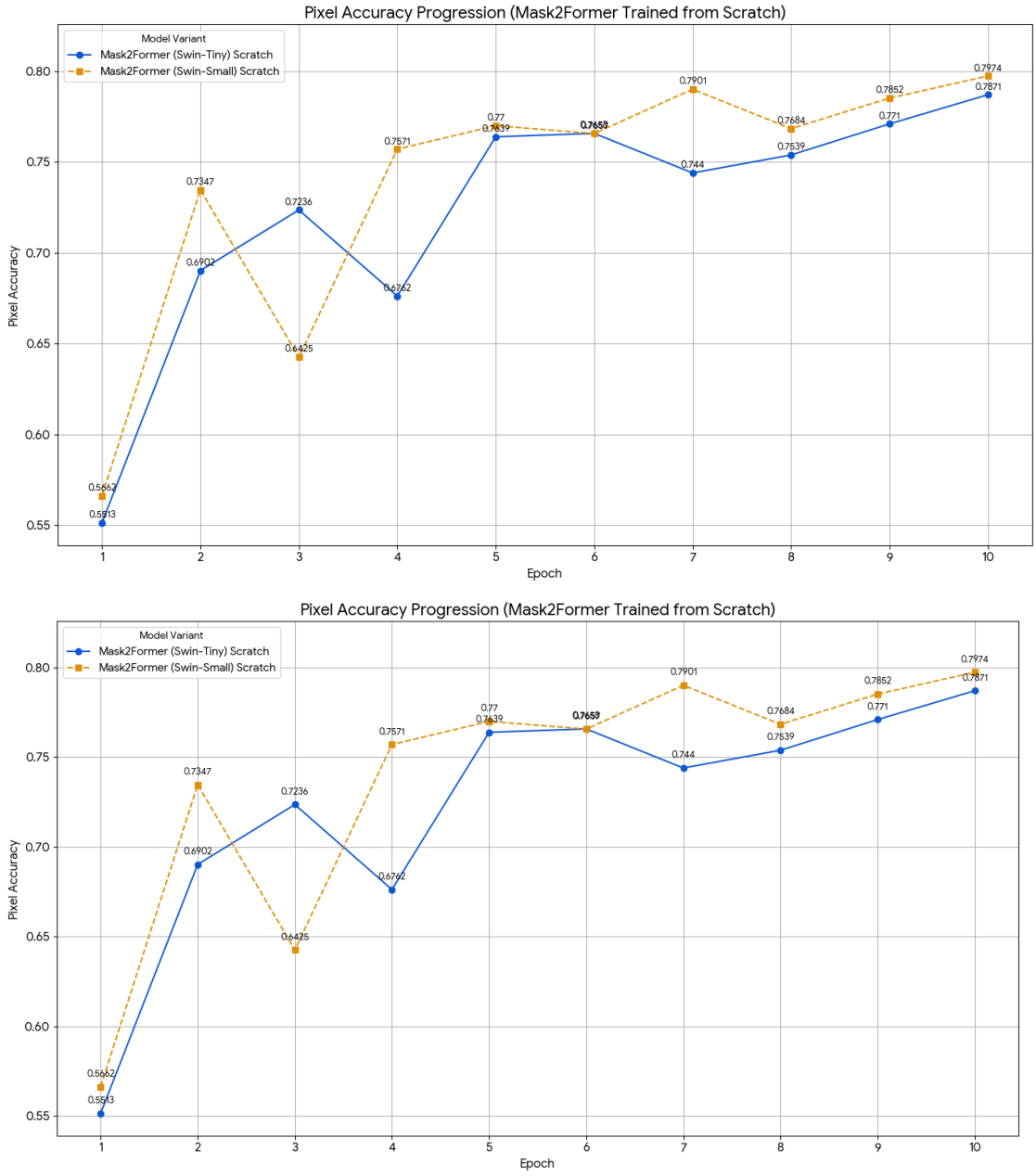


Figure: 32: Mean iou and pixel accuracy trained from scratch per epoch on IDD-lite

- **Analysis of Per-Class Performance**

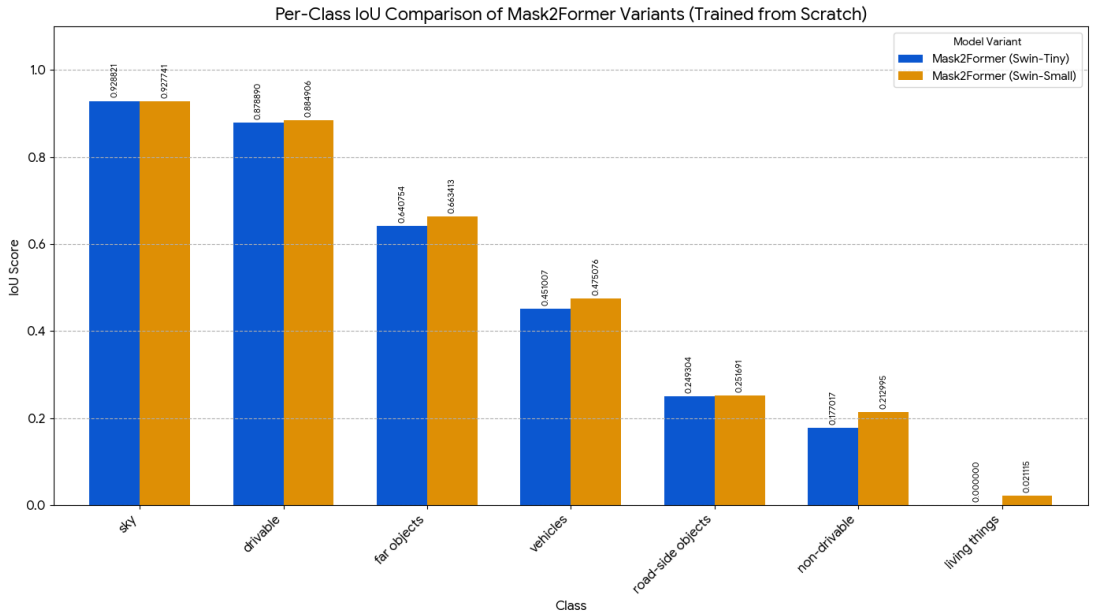


Figure: 33:Per-Class IoU of Mask2former(swin-tiny/swin-small) trained from scratch on IDD-Lite

A detailed comparison of the per-class IoU scores, visualized in the grouped bar chart in Figure: 33, reveals the severe impact of training from scratch on both Mask2Former variants. The most significant finding is the critical failure on the 'living things' class, where the Swin-Tiny model scored 0.0 and the larger Swin-Small model scored a negligible 0.021115. This confirms that neither architecture could learn to identify these safety-critical road users without a pre-trained foundation. Furthermore, unlike the pre-trained experiments, increasing the model size offered no consistent or meaningful advantage, with both models performing exceptionally poorly on other challenging classes like 'non-drivable' and 'road-side objects'. This detailed breakdown provides strong evidence that the architectural advantages of Mask2Former are rendered ineffective without the foundational knowledge provided by pre-training.

- **Training vs Validation loss for Mask2former (with backbones swin-tiny/swin-large) trained from scratch**

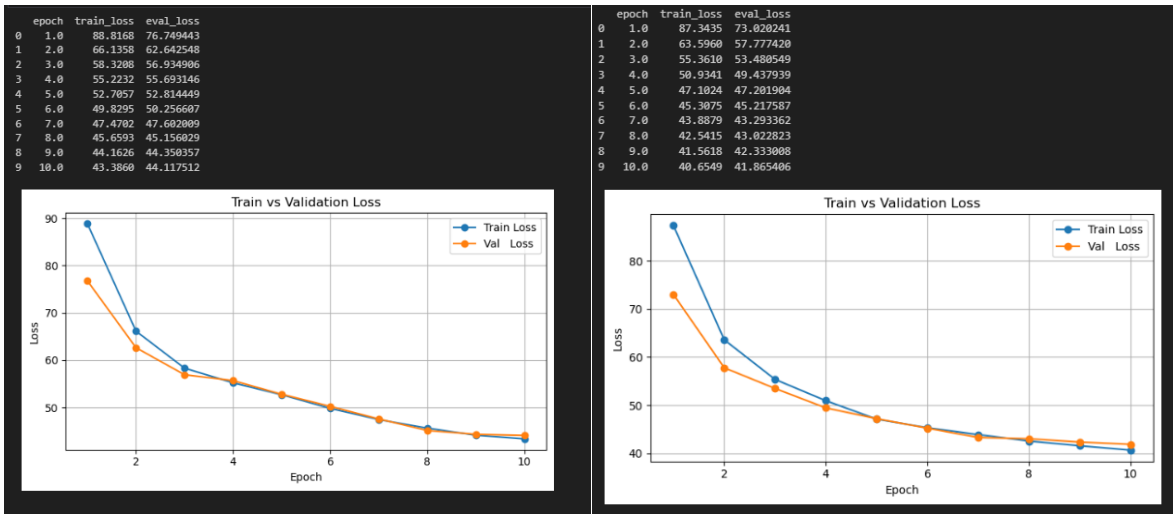
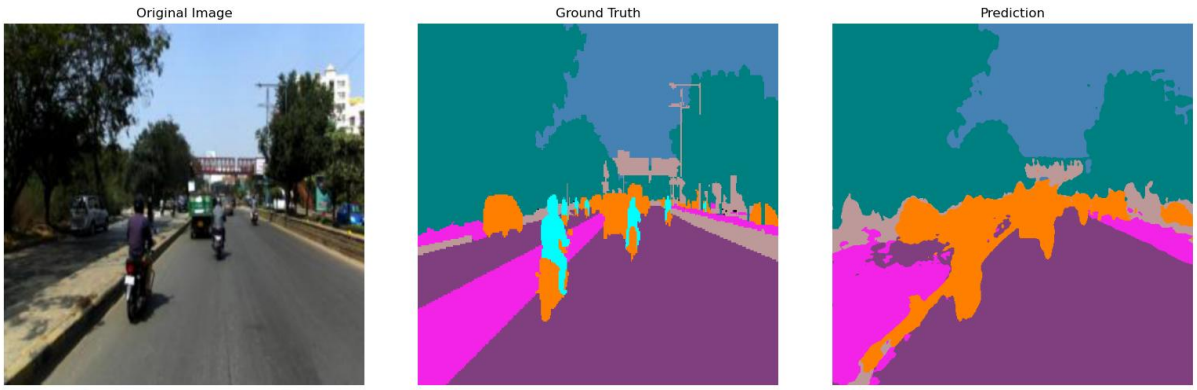


Figure: 34:training vs validation of mask2former (swin-tiny/swin-small) ,trained from scratch on IDD-lite

The training and validation loss curves, shown in Figure: 34, reveal a highly ineffective and unstable learning process for both Mask2Former variants when trained from scratch. Both the Swin-Tiny and Swin-Small models converged at an extremely high final error value, with validation losses of 44.117512 and 41.865406 respectively. The significant and persistent gap between the higher validation loss and the lower training loss for both models is a clear indicator of poor generalization and a tendency towards overfitting. This demonstrates that without the foundational knowledge provided by pre-training, both architectures struggled to learn meaningful features from the training data and apply them to unseen validation data, confirming that transfer learning is essential for this complex task

- **Qualitative Visual Analysis of ground truth vs predicted semantic map of experiment 5**



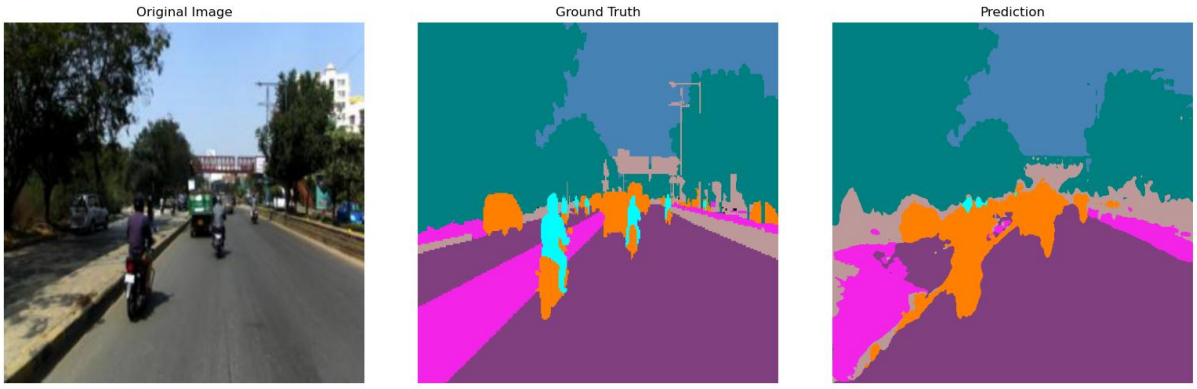


Figure: 35:Visualization of Ground Truth vs Predicted Semantic maps of Mask2former (swin-tiny/swin-small) ,trained from scratch on IDD-Lite Models

In Figure: 35 both the Swin-Tiny trained from scratch on IDD lite and Swin-Small Swin-Tiny trained from scratch on IDD lite models in completely fail to identify the rider, which belongs to the 'living things' class. The predictions for this safety critical objects has no meaning , with the models aggressively misclassifying the area as 'vehicles' and 'non-drivable' surfaces. This visual failure directly corresponds to the near zero IoU scores for the 'living things' class. The overall segmentation maps are chaotic and bear little resemblance to the ground truth, confirming that without pre-training, neither architecture was able to learn the complex features of the dataset.

Chatper 9: Conclusion

This dissertation conducted a systematic comparative analysis of two leading Vision Transformer architectures, SegFormer and Mask2Former, to evaluate their effectiveness for semantic segmentation in the challenging, unstructured road environments depicted in the Indian Driving Dataset (IDD-Lite). Through a series of five distinct experiments, the research successfully addressed all its core objectives, producing a clear set of findings regarding the capabilities, trade-offs, and optimal implementation strategies for these state-of-the-art models in a real-world context.

Fulfillment of Research Objectives and Summary of Findings

The study's primary objectives were fulfilled, yielding critical insights into the performance of these advanced models:

- **Objectives 1 & 4 (using domain pretrained weights):** The objectives to implement and fine-tune pre-trained SegFormer and Mask2Former models were successfully met. The results established the top-tier performance for each model family, with the SegFormer-B2 achieving a Mean Intersection over Union (mIoU) of 0.723008 and the Mask2Former Swin-Small reaching 0.761012 mIoU. This confirmed their viability for this complex task.

- **Objectives 2 & 5 (Training Models from Scratch):** The objectives to train both architectures from scratch yielded a definitive conclusion. Both SegFormer and Mask2Former models failed catastrophically when trained without pre-trained weights. SegFormer-B0 and B1 models scored an IoU of 0.0 on the critical 'living things' class , while the Mask2Former Swin-Tiny also scored 0.0. This established a crucial understanding that using domain pretrained weights is essential for achieving meaningful performance on the IDD-Lite dataset.

- **Objective 3 (Investigating Focal Loss with domain pretrained weights):** The objective to investigate the impact of Focal Loss was addressed by comparing its performance against the standard Cross-Entropy Loss. The findings indicated that Focal Loss resulted in only a marginal performance improvement, with the SegFormer-B2 model's mIoU increasing slightly to 0.724108. The analysis concluded that the model's architectural capacity was a more critical factor than the specific loss function used.

- **Objective 6 (Quantitative Comparison):** This objective was met through a comprehensive quantitative analysis that revealed a clear accuracy-efficiency trade-off. Mask2Former was proven superior for accuracy, especially on difficult minority classes like 'living things'. Conversely, SegFormer was the winner for computational efficiency, with the SegFormer-B2 delivering an inference speed of 10.696 FPS, over three times faster than Mask2Former's Swin-Small at 3.637 FPS.

- **Objective 7 (Qualitative Analysis):** The final objective was met by visually comparing the predicted segmentation maps against the ground truth. This qualitative analysis confirmed the quantitative findings, starkly illustrating the chaotic and inaccurate predictions of the from-scratch models. For domain pre-trained Cityscapes weight models, the analysis revealed Mask2Former's superiority in producing cleaner boundaries and more accurately separating objects like a rider from a motorcycle, a task where SegFormer models struggled.

Overall, the Mask2Former model with Cityscapes pretrained weights fine tuned on IDD-lite achieved the highest overall accuracy, while the impact of Focal Loss on the SegFormer with Cityscapes pretrained weights fine tuned on IDD-lite model was minimal. The Mask2Former (Swin-Small) model proved to be the best-performing model for accuracy, achieving the highest mean Intersection over Union (mIoU) of 0.761012 across all experiments. When comparing the SegFormer-B2 models, the version with Focal Loss did achieve a slightly higher mIoU of 0.724108 compared to the 0.723008 from the segformer with Cityscapes pretrained weights fine tuned on IDD-lite model with standard loss. However, the analysis concludes that this impact was "marginal". For the critical 'living things' class, the

SegFormer-B2 with Cityscapes pretrained weights fine tuned on IDD-lite with standard loss scored an IoU of 0.554328, which was slightly higher than the 0.550969 achieved by the Focal Loss version. Therefore, your results indicate that

Mask2Former (Swin-Small) with Cityscapes pretrained weights fine tuned on IDD-lite is the top model for accuracy.

Recommendations for Future Work

Based on the findings and limitations of this study, several avenues for future work are recommended:

- **Scaling to the Full IDD Dataset:** A direct next step is to scale the experiments to the full Indian Driving Dataset (IDD), which features a more complex, fine-grained label hierarchy of 34 classes.

- **On-Device Performance Benchmarking:** A critical step towards real-world application would be to deploy the best-performing models on embedded systems to evaluate true real-world inference speed, power consumption, and memory usage.

- **Investigating Other Loss Functions:** A broader investigation into other advanced loss functions, such as Dice or Tversky loss, could still yield benefits for training Vision Transformers on the unique class imbalances of the IDD-Lite dataset.

- **Exploring Larger Backbones:** Future work could also extend the evaluation to larger encoder variants. For SegFormer, this includes MiT-B3, MiT-B4, and MiT-B5, which offer progressively higher capacity and accuracy at the cost of increased complexity. For Mask2Former, experiments with Swin-Base and Swin-Large backbones would provide insights into how scaling the backbone affects performance trade-offs in unstructured road environments.

Chatper 10: References

[1] J. Janai, F. Güney, A. Behl, and A. Geiger, “Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art,” *arXiv:1704.05519 [cs]*, Dec. 2019, Available: https://arxiv.org/abs/1704.05519

[2] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos, “Image Segmentation Using Deep Learning: A Survey,” *arXiv:2001.05566 [cs]*, Nov. 2020, Available: https://arxiv.org/abs/2001.05566

[3] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 3213-3223, doi: 10.1109/CVPR.2016.350.

[4] F. Lateef and Y. Ruichek, "Survey on semantic segmentation using deep learning techniques," Neurocomputing, vol. 338, pp. 321-348, Apr. 2019, doi: 10.1016/j.neucom.2019.02.003.

[5] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, “SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers,” *arXiv:2105.15203 [cs]*, Oct. 2021, Available: https://arxiv.org/abs/2105.15203

[6] B. Cheng, I. Misra, A. G. Schwing, A. Kirillov, and R. Girdhar, “Masked-attention Mask Transformer for Universal Image Segmentation,” *arXiv:2112.01527 [cs]*, Jun. 2022, Available: https://arxiv.org/abs/2112.01527

[7] G. Varma, A. Subramanian, A. Namboodiri, M. Chandraker, and C. Jawahar, “IDD: A Dataset for Exploring Problems of Autonomous Navigation in Unconstrained Environments.” Available: https://arxiv.org/pdf/1811.10200

[8] R. Azad *et al.*, "Loss Functions in the Era of Semantic Segmentation: A Survey and Outlook," *arXiv.org*, Dec. 08, 2023. https://arxiv.org/abs/2312.05391

[9] Abid, N. Mehta, Z. Wu, and R. Timofte, "ContextFormer: Redefining Efficiency in Semantic Segmentation," *arXiv.org*, 2025. https://arxiv.org/abs/2501.19255.

[10] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid Scene Parsing Network," *arXiv.org*, Apr. 27, 2017. https://arxiv.org/abs/1612.01105

[11] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation," *arXiv.org*, 2018. https://arxiv.org/abs/1802.02611

[12] B. Baheti, S. Innani, S. Gajre, and S. Talbar, "Semantic scene segmentation in unstructured environment with modified DeepLabV3+," *Pattern Recognition Letters*, vol. 138, pp. 223–229, Oct. 2020, doi: https://doi.org/10.1016/j.patrec.2020.07.029.

[13] B. Baheti, S. Innani, S. Gajre, and S. Talbar, "Eff-UNet: A Novel Architecture for Semantic Segmentation in Unstructured Environment," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Jun. 2020, doi: https://doi.org/10.1109/cvprw50498.2020.00187.

[14] I. Ulku and E. Akagündüz, "A Survey on Deep Learning-based Architectures for Semantic Segmentation on 2D Images," *Applied Artificial Intelligence*, pp. 1–45, Feb. 2022, doi: https://doi.org/10.1080/08839514.2022.2032924.

[15] "SegFormer," *Huggingface.co*, 2019. https://huggingface.co/docs/transformers/en/model_doc/segformer#segformer

[16] "Mask2Former," *Huggingface.co*, 2019. https://huggingface.co/docs/transformers/en/model_doc/mask2former#mask2former .

[17] "Fine-Tune a Semantic Segmentation Model with a Custom Dataset," *huggingface.co*. https://huggingface.co/blog/fine-tune-segformer

[18] Alara Dirik, “Train a MaskFormer Segmentation Model with Hugging Face Transformers - PyImageSearch,” *PyImageSearch*, Mar. 13, 2023. https://pyimagesearch.com/2023/03/13/train-a-maskformer-segmentation-model-with-hugging-face-transformers/.

[19] “nvidia/segformer-b0-finetuned-cityscapes-1024-1024 · Hugging Face,” Huggingface.co, 2017. https://huggingface.co/nvidia/segformer-b0-finetuned-cityscapes-1024-1024

[20] “nvidia/segformer-b1-finetuned-cityscapes-1024-1024 · Hugging Face,” *Huggingface.co*, 2017. https://huggingface.co/nvidia/segformer-b1-finetuned-cityscapes-1024-1024

[21] “nvidia/segformer-b2-finetuned-cityscapes-1024-1024 · Hugging Face,” *Huggingface.co*, 2017. https://huggingface.co/nvidia/segformer-b2-finetuned-cityscapes-1024-1024.

[22] TensorTeach, “Training GPT2 From Scratch In Hugging Face | Generative AI with Hugging Face | TensorTeach,” *YouTube*, Sep. 21, 2023. https://www.youtube.com/watch?v=98jROp7Ij9A.

[23] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal Loss for Dense Object Detection,” *arXiv:1708.02002 [cs]*, Feb. 2018, Available: https://arxiv.org/abs/1708.02002

[24] “ Evaluate,” *Huggingface.co*, 2024. https://huggingface.co/docs/evaluate/en/index

[25] “albumentations,” *PyPI*, May 27, 2025. https://pypi.org/project/albumentations/

[26] “Trainer,” *Huggingface.co*, 2019. https://huggingface.co/docs/transformers/en/main_classes/trainer#trainer

[27] “Callbacks,” *Huggingface.co*, 2019. https://huggingface.co/docs/transformers/en/main_classes/callback.

[29] “facebook/mask2former-swin-tiny-ade-semantic · Hugging Face,” *Huggingface.co*, 2017. https://huggingface.co/facebook/mask2former-swin-tiny-ade-semantic.

[30] “facebook/mask2former-swin-small-ade-semantic · Hugging Face,” *Huggingface.co*, 2017. https://huggingface.co/facebook/mask2former-swin-small-ade-semantic.

[31] “Losses — Segmentation Models documentation,” *Readthedocs.io*, 2025. https://smp.readthedocs.io/en/latest/losses.html

[32] “Optimization,” *huggingface.co*. https://huggingface.co/docs/transformers/en/main_classes/optimizer_schedules

[33] “Adam,” *Huggingface.co*, 2025. https://huggingface.co/docs/bitsandbytes/main/en/reference/optim/adam

[34] “IDD,” *idd.insaan.iiit.ac.in*. https://idd.insaan.iiit.ac.in/

Chatper 11: Appendices

1.Exploratory Data Analysis

a. Count Images and Masks per Split in Train and Validation

```
IMG_DIR = r"dataset/idd20k_lite/leftImg8bit"
MASK_DIR = r"dataset/idd20k_lite/gtFine"
SPLITS = ["train", "val"]
```

```
def count_files(root, split, suffix):
    split_dir = os.path.join(root, split)
    count = 0
    if not os.path.exists(split_dir):
        return 0
    for city in os.listdir(split_dir):
        city_dir = os.path.join(split_dir, city)
        for f in os.listdir(city_dir):
            if f.endswith(suffix) and "inst" not in f:
                count += 1
    return count

for split in SPLITS:
    n_imgs = count_files(IMG_DIR, split, ".jpg")
    n_masks = count_files(MASK_DIR, split, "_label.png")
    print(f"{split.upper()}: Images = {n_imgs}, Semantic Masks = {n_masks}")
```

```
[9]
...  TRAIN: Images = 1403, Semantic Masks = 1403
     VAL: Images = 204, Semantic Masks = 204
```

3. Check Image and Mask Shapes

Verification of all images and their ground truth masks on train and validation that they may have expected and matching shapes. This step ensures that the data is consistent and suitable for training a segmentation model.

image is RGB hence 3 channels semantic masks is greyscale hence 1 channel

```python
def check_shapes(img_dir, mask_dir, split):
    img_shapes, mask_shapes = set(), set()
    split_img = os.path.join(img_dir, split)
    split_mask = os.path.join(mask_dir, split)
    for city in os.listdir(split_img):
        city_img = os.path.join(split_img, city)
        city_mask = os.path.join(split_mask, city)
        for f in os.listdir(city_img):
            if f.endswith('.jpg'):
                img = Image.open(os.path.join(city_img, f))
                img_shape = img.size + (len(img.getbands()),)  # (width, height, channels)
                img_shapes.add(img_shape)
                mask_name = f.replace("_image.jpg", "_label.png").replace(".jpg", "_label.png")
                mask_path = os.path.join(city_mask, mask_name)
                if os.path.exists(mask_path):
                    mask = Image.open(mask_path)
                    mask_shape = mask.size + (len(mask.getbands()),)  # (width, height, channels)
                    mask_shapes.add(mask_shape)
    print(f"{split.upper()}: Unique image shapes: {img_shapes}, Unique mask shapes: {mask_shapes}")

for split in ["train", "val"]:
    check_shapes(IMG_DIR, MASK_DIR, split)
```

```
TRAIN: Unique image shapes: {(320, 227, 3)}, Unique mask shapes: {(320, 227, 1)}
VAL: Unique image shapes: {(320, 227, 3)}, Unique mask shapes: {(320, 227, 1)}
```

4. List Unique Class Labels

The unique class IDs present in the semantic masks. This allows us to confirm the number and identity of the semantic classes in the dataset.

```python
def scan_unique_ids(mask_dir, split):
    unique_ids = set()
    split_mask = os.path.join(mask_dir, split)
    for city in os.listdir(split_mask):
        city_mask = os.path.join(split_mask, city)
        for f in os.listdir(city_mask):
            if f.endswith('_label.png') and 'inst' not in f:
                arr = np.array(Image.open(os.path.join(city_mask, f)))
                unique_ids.update(np.unique(arr))
    # print(f"{split.upper()} unique IDs:", sorted(unique_ids))
    print(f"{split.upper()} unique IDs:", sorted([int(i) for i in unique_ids]))

for split in ["train", "val"]:
    scan_unique_ids(MASK_DIR, split)
```

8. Check Image Resolution Distribution

```python
import os
from PIL import Image
from collections import Counter

#
#

IMG_DIR = r"dataset/idd_lite/leftImg8bit"

SPLITS = ["train", "val"]

res_counter = Counter()

for split in SPLITS:
    split_dir = os.path.join(IMG_DIR, split)
    if not os.path.exists(split_dir):
        continue
    for city in os.listdir(split_dir):
        city_dir = os.path.join(split_dir, city)
        for fname in os.listdir(city_dir):
            if fname.endswith('.jpg'):
                path = os.path.join(city_dir, fname)
                try:
                    img = Image.open(path)
                    w, h = img.size
                    res_counter[(w, h)] += 1
                except Exception as e:
                    print(f"Error reading {path}: {e}")

print("Unique image resolutions (width x height) and their counts:")
for (w, h), cnt in res_counter.items():
    print(f"{w}x{h} : {cnt} images")
```

```
Unique image resolutions (width x height) and their counts:
320x227 : 1607 images
```

a. Pixel-wise Class Distribution for Train and Validation Splits

To understand the data distribution across classes, we calculate the total number of pixels assigned to each semantic class in both the training and validation splits. This helps identify class imbalance issues, which can be important when training semantic segmentation models.

According to the official IDD-Lite paper, there are 7 classes: we ignore 255 because its not for training

0 = Drivable
1 = Non-drivable
2 = Living things
3 = Vehicles
4 = Road side objects
5 = Far objects
6 = Sky

255=ignored

```python
CLASS_NAMES_1 = [
    "Drivable",
    "Non-drivable",
    "Living things",
    "Vehicles",
    "Road side objects",
    "Far objects",
    "Sky",
    "Void"   #not for training
]
```

```python
def plot_class_distribution(mask_dir, split, class_names):
    pixel_counts = np.zeros(len(class_names), dtype=np.int64)
    split_mask = os.path.join(mask_dir, split)
    for city in os.listdir(split_mask):
        city_mask = os.path.join(split_mask, city)
        for f in os.listdir(city_mask):
            if f.endswith('_label.png') and 'inst' not in f:
                arr = np.array(Image.open(os.path.join(city_mask, f)))
                for i in range(len(class_names)-1):  # class indices 0-6
                    pixel_counts[i] += np.sum(arr == i)
                pixel_counts[-1] += np.sum(arr == 255)  # void class
    plt.figure(figsize=(8,4))
    plt.bar(class_names, pixel_counts)
    plt.title(f"Pixel Distribution per Class ({split} split, incl. Void)")
    plt.ylabel("Pixel count")
    plt.xticks(rotation=30)
    plt.ticklabel_format(style='plain', axis='y')
    plt.show()
    print(f"Pixel counts for '{split}' split (incl. void):")
    for name in class_names:
        i = class_names.index(name)
        print(f"{name}: {pixel_counts[i]} pixels ({pixel_counts[i]/pixel_counts.sum()*100:.2f}%)")

plot_class_distribution(MASK_DIR, "train", CLASS_NAMES_1)
plot_class_distribution(MASK_DIR, "val", CLASS_NAMES_1)
```

b.To Visualize Example Images and ground truth Semantic Masks

We randomly select a few images and display them alongside their ground-truth semantic segmentation masks from the train folder.

```python
CLASS_COLORS = [
    (128,  64,128),   # Drivable (purple)
    (244,  35,232),   # Non-drivable (pink)
    (0, 255,255),     # Living things (cyan)
    (255,128,  0),    # Vehicles (orange)
    (190,153,153),    # Road side objects (light brown)
    (0, 128,128),     # Far objects (teal)
    (70,130,180),     # Sky (sky blue)
    (0, 0, 0) ,       # Void (black)
]
CLASS_COLORS_NORM = [tuple(c/255 for c in rgb) for rgb in CLASS_COLORS]
def mask_to_rgb(mask):
    rgb = np.zeros(mask.shape + (3,), dtype=np.float32)
    for idx, color in enumerate(CLASS_COLORS_NORM):
        rgb[mask == idx] = color
    return rgb
def show_examples(img_dir, mask_dir, split, n=4):
    imgs = []
    masks = []
    split_img = os.path.join(img_dir, split)
    split_mask = os.path.join(mask_dir, split)
    for city in os.listdir(split_img):
        city_img = os.path.join(split_img, city)
        city_mask = os.path.join(split_mask, city)
        for f in os.listdir(city_img):
            if f.endswith('.jpg'):
                imgs.append(os.path.join(city_img, f))
                mask_name = f.replace("_image.jpg", "_label.png").replace(".jpg", "_label.png")
                masks.append(os.path.join(city_mask, mask_name))
    idxs = random.sample(range(len(imgs)), min(n, len(imgs)))
    for idx in idxs:
        img = np.array(Image.open(imgs[idx]))
        mask = np.array(Image.open(masks[idx]))
        mask_rgb = mask_to_rgb(mask)

        plt.figure(figsize=(10, 4))
        plt.subplot(1, 2, 1)
        plt.imshow(img)
        plt.title('Image')
        plt.axis('off')
        plt.subplot(1, 2, 2)
        plt.imshow(mask_rgb)
        plt.title('Ground Truth Semantic Map')
        plt.axis('off')
        plt.show()
show_examples(IMG_DIR, MASK_DIR, "train", n=4)
```

A. Segformer Pipeline

1.Importing Libraries

```python
# Cell 1: Imports & Constants
import os
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import albumentations as A
from transformers import (
    SegformerForSemanticSegmentation,
    SegformerImageProcessor,
    TrainingArguments,
    Trainer
)
import evaluate

import albumentations as A
```

2.Collection of Image & Mask Paths

Traverse the folder structure to list `.jpg` and matching `_label.png` files.

```python
def get_image_mask_paths(root_img, root_mask, split="train"):
    image_paths, mask_paths = [], []
    split_dir = os.path.join(root_img, split)
    for city in os.listdir(split_dir):
        img_folder  = os.path.join(root_img, split, city)
        mask_folder = os.path.join(root_mask, split, city)
        for fname in os.listdir(img_folder):
            if fname.endswith("_image.jpg"):
                image_paths.append(os.path.join(img_folder, fname))
                mask_name = fname.replace("_image.jpg", "_label.png")
                mask_paths.append(os.path.join(mask_folder, mask_name))
    return image_paths, mask_paths


IMG_ROOT  = "dataset/idd_lite/leftImg8bit"
MASK_ROOT = "dataset/idd_lite/gtFine"

train_img, train_mask = get_image_mask_paths(IMG_ROOT, MASK_ROOT, "train")
val_img,   val_mask   = get_image_mask_paths(IMG_ROOT, MASK_ROOT, "val")
```

3.Preprocessing and Data Augmentation

3.1.Preprocessing step

```
# Define class names (as previously established)
CLASS_NAMES = [
    "Drivable", "Non-drivable", "Living things",
    "Vehicles", "Road side objects", "Far objects", "Sky"
]
NUM_CLASSES = 7

ID2LABEL = {i: name for i, name in enumerate(CLASS_NAMES)}
LABEL2ID = {name: i for i, name in enumerate(CLASS_NAMES)}
```

3.2.Data Augmentation

```
# Cell 4: Transforms & Processor
import albumentations as A


train_transforms = A.Compose([
    A.RandomResizedCrop(size=(512, 512), scale=(0.7, 1.0), ratio=(0.75, 1.33), p=1.0),
    A.HorizontalFlip(p=0.5),
    A.Affine(translate_percent=0.05, scale=(0.9, 1.1), rotate=(-15, 15), p=0.5),
    A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.05, p=0.5),
    A.GaussianBlur(p=0.2),


])

val_transforms = A.Compose([
    A.Resize(512, 512),

])
```

3.3.Preprocessing step to Setup the Segformer Image Processor and do rescale,normalization and to make sure ,it doesn't include the include the 255 ignore label ID when training

```
processor = SegformerImageProcessor(
    # "nvidia/segformer-b0-finetuned-cityscapes-1024-1024",
    # do_reduce_labels=False,
    # do_rescale=True,
    # do_normalize=True
    num_labels=7,
    do_reduce_labels=False,
    do_rescale=True,
    do_normalize=True,
    ignore_index=255
)
```

1.4 Data preprocessing step we create Custom PyTorch Dataset for IDD- Lite,we define a PyTorch Dataset that loads images and masks, applies preprocessing, and ensures all invalid mask values are mapped to 255 (ignore index).

```python
import numpy as np
import torch
from torch.utils.data import Dataset
from PIL import Image

IGNORE_INDEX = 255
NUM_CLASSES = 7  # 0-6

class IDDSegDataset(Dataset):
    def __init__(self, image_paths, mask_paths, transform=None, processor=None):
        self.image_paths = image_paths
        self.mask_paths = mask_paths
        self.transforms = transform
        self.processor = processor

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

    def __getitem__(self, idx):
        # 1) Load
        img = np.array(Image.open(self.image_paths[idx]).convert("RGB"))
        mask = np.array(Image.open(self.mask_paths[idx]), dtype=np.int64)

        # 2) Clamp invalid → IGNORE_INDEX
        mask = np.where((mask < 0) | (mask >= NUM_CLASSES), IGNORE_INDEX, mask)

        # 3) Albumentations (resize+augs only)
        if self.transforms is not None:
            out = self.transforms(image=img, mask=mask)
            img, mask = out["image"], out["mask"]

        # 4) Processor: just call, don't rename keys
        if self.processor is not None:
            proc = self.processor(
                images=img,
                segmentation_maps=mask,      # pass as segmentation_maps
                return_tensors="pt",
                # do_resize=False,             # already resized in Alb
                # do_rescale=True,
                # do_normalize=True
            )
            # # Just use the processor's output as is!
            # labels_tensor = proc["labels"].squeeze(0)
            # if idx < 5:
            #     print(f"[DEBUG][{idx}] final labels uvals:", labels_tensor.unique().tolist())
            # return {k: v.squeeze(0) for k, v in proc.items()}


            labels_tensor = proc["labels"].squeeze(0)

            # if idx < 5:
            #     u = labels_tensor.unique().tolist()
            #     print(f"[DEBUG][{idx}] final labels uvals:", u)
            #     print("Num ignore pixels:", (labels_tensor == 255).sum().item())
            #     if IGNORE_INDEX in u:
            #         print(f"[ERROR][{idx}] IGNORE_INDEX still present!")
            # drop batch dim and return
            return {k: v.squeeze(0) for k, v in proc.items()}

        raise RuntimeError("Processor is required for this dataset")
```

```
# Cell 6: Build datasets
train_ds = IDDSegDataset(
    train_img, train_mask,
    transform=train_transforms,
    processor=processor
)
val_ds = IDDSegDataset(
    val_img, val_mask,
    transform=val_transforms,
    processor=processor
)
```

4. Model Initialization

4.1.For Domain Specific Pretraining and Fine tuning on IDD-lite

Initialization SegFormer pretrained cityscapes model and , and configure the segmentation head for 7 classes, ignoring 255 for loss/metrics and also the encoder changes with respect to b0 to b2.

```
model = SegformerForSemanticSegmentation.from_pretrained(
    "nvidia/segformer-b0-finetuned-cityscapes-1024-1024",
    num_labels=7,
    id2label=ID2LABEL,
    label2id=LABEL2ID,
    ignore_mismatched_sizes=True,
    use_safetensors= True,
    semantic_loss_ignore_index=255
)
```

4.2.For training From Scratch on IDD-lite

Initialization SegFormer pretrained cityscapes model but initialize the SegformeConfig to train from scratch and , and configure the segmentation head for 7 classes, ignoring 255 for loss/metrics and also the encoder changes with respect to b0 to b2.

```
from transformers import SegformerForSemanticSegmentation, SegformerConfig

config = SegformerConfig.from_pretrained(
    "nvidia/segformer-b0-finetuned-cityscapes-1024-1024",
    num_labels=7,
    id2label=ID2LABEL,
    label2id=LABEL2ID,
    ignore_mismatched_sizes=True,
    use_safetensors= True,
    semantic_loss_ignore_index=255
)

model = SegformerForSemanticSegmentation(config)
```

5.Computing Metrics: Mean IoU, Pixel Accuracy, Per-Class IoU

We use Hugging Face's evaluate library to compute mean IoU, pixel accuracy, and per-class IoU, while ignoring 255 pixels in both training and evaluation.

```python
import torch
import torch.nn.functional as F
import evaluate
import numpy as np
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

CLASS_NAMES = [
    "drivable",
    "non_drivable",
    "living_things",
    "vehicles",
    "road_side_objects",
    "far_objects",
    "sky",
]

miou_metric = evaluate.load("mean_iou")

def compute_metrics(eval_pred):
    logits, labels = eval_pred

    # — 1. Bring logits → torch.Tensor (unpack tuples if needed)
    if isinstance(logits, np.ndarray):
        logits_t = torch.from_numpy(logits)
    else:
        logits_t = logits
    if isinstance(logits_t, tuple):
        logits_t = logits_t[0]

    # — 2. Resize to match label H×W
    if logits_t.shape[2:] != labels.shape[1:]:
        logits_t = F.interpolate(
            logits_t,
            size=labels.shape[1:],
            mode="bilinear",
            align_corners=False
        )

    # — 3. Argmax → (B, H, W) → numpy
    preds = torch.argmax(logits_t, dim=1).cpu().numpy()
    if isinstance(labels, torch.Tensor):
        lbls = labels.cpu().numpy()
    else:
        lbls = labels

    # — 4. Compute HF-evaluate raw MIoU + per-category IoU
    raw = miou_metric.compute(
        predictions=preds,
        references=lbls,
        num_labels=7,
        ignore_index=255,
        reduce_labels=False,
    )

    # — 5. Build & plot confusion matrix
    # flatten everything and mask out IGNORE
    p_flat = preds.reshape(-1)
    l_flat = lbls.reshape(-1)
    mask = l_flat != 255
    p_flat = p_flat[mask]
    l_flat = l_flat[mask]

    cm = confusion_matrix(l_flat, p_flat, labels=list(range(len(CLASS_NAMES))))
    cm_norm = cm.astype(float) / (cm.sum(axis=1, keepdims=True) + 1e-12)

    plt.figure(figsize=(6,5))
    plt.imshow(cm_norm, vmin=0, vmax=1, interpolation="nearest")
    plt.title("Confusion Matrix")
    plt.colorbar(label="Fraction of true-class pixels predicted as each class")
    ticks = np.arange(len(CLASS_NAMES))
    plt.xticks(ticks, CLASS_NAMES, rotation=45, ha="right")
    plt.yticks(ticks, CLASS_NAMES)
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.tight_layout()
    plt.show()

    # — 6. Extract your original metrics
    pixel_acc      = raw["overall_accuracy"]
    mean_iou       = raw["mean_iou"]
    # mean_acc       = raw.get("mean_accuracy", None)
    per_cat_iou    = raw["per_category_iou"]
    per_cat_acc    = raw.get("per_category_accuracy", None)

    per_class_iou  = {f"{CLASS_NAMES[i]}_iou": float(per_cat_iou[i])
                      for i in range(len(CLASS_NAMES))}
    # per_class_acc  = ({f"{CLASS_NAMES[i]}_accuracy": float(per_cat_acc[i])
    #                    for i in range(len(CLASS_NAMES))}
    #                   if per_cat_acc is not None else {})

    # — 7. Return everything
    return {
        "pixel_accuracy": pixel_acc,
        # "mean_accuracy" : mean_acc,
        "mean_iou"      : mean_iou,
        **per_class_iou,
        # **per_class_acc,
    }
```

6.Training Configuration

```python
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./outputs_Segformer/Segformer_exp4/segformer_b0_train_from_scratch_with_data_augmentation",           # Where checkpoints & outputs are saved
    logging_dir="./outputs_Segformer/Segformer_exp4/segformer_b0_train_from_scratch_with_data_augmentation/logs",       # TensorBoard logs in output dir
    report_to=["tensorboard"],                          # Enable TensorBoard logging

    num_train_epochs=10,                                # Number of training epochs
    per_device_train_batch_size=4,                      # Training batch size
    per_device_eval_batch_size=4,                       # Eval batch size
    optim="adamw_torch",
    learning_rate=0.00006,                                 # Learning rate (from Mask2Former paper)
    weight_decay=0.01,
    warmup_ratio=0.1,                                                           # Weight decay for regularization
    lr_scheduler_type="polynomial",                    # Poly LR scheduler


    # logging_strategy="steps",      # log training metrics every logging_steps
    # logging_steps=50,              # ← change to how often you want train-loss logs
    # eval_strategy="steps",         # run validation every eval_steps
    # eval_steps=200,                # ← change to how often you want val-loss/metrics
    # save_strategy="steps",         # optionally checkpoint every save_steps
    # save_steps=200,

    # eval_strategy="epoch",                       # Evaluate after each epoch
    # save_strategy="epoch",
    logging_strategy="epoch",    # log training metrics once at end of each epoch
    eval_strategy="epoch",       # run validation once at end of each epoch
    save_strategy="epoch",       # checkpoint once at end of each epoch
    save_total_limit=1,                                # Save after each epoch
    # save_total_limit=10,                                 # Only keep the best checkpoint
    load_best_model_at_end=True,                        # Restore best checkpoint after training
    metric_for_best_model="mean_iou",                   # Metric to pick best checkpoint
    greater_is_better=True,                             # Higher mean IoU is better

    # logging_strategy="epoch",                           # Log metrics after each epoch
    # logging_steps=50,                                  # (optional: only if logging_strategy="steps")

    fp16=True,                                          # Mixed precision for faster training (if supported)
    seed=42,                                            # Reproducibility
                                    # Required for custom dataset/collate
)
```

7.Trainer configuration: instantiate the HF Trainer with model, TrainingArguments, train and val datasets, metric function, and an EarlyStoppingCallback (patience = 3)

```
from transformers import Trainer, EarlyStoppingCallback

trainer_segformer_b0_train_from_scratch_with_data_augmentation = Trainer(
    model           = model,
    args            = training_args,
    train_dataset   = train_ds,
    eval_dataset    = val_ds,
    compute_metrics = compute_metrics,
    callbacks       = [EarlyStoppingCallback(early_stopping_patience=3)],
)
```

8.For Computing of Focal loss instead of pixel wise cross entropy

```python
import torch.nn as nn
import segmentation_models_pytorch as smp
class FocalLossTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False,**kwargs):
        # pull out labels and run forward
        labels = inputs.pop("labels")           # (B, H, W)
        outputs = model(**inputs)
        logits  = outputs.logits                # (B, C, H, W)

        # resize logits to match labels if needed
        if logits.shape[-2:] != labels.shape[-2:]:
            logits = nn.functional.interpolate(
                logits,
                size=labels.shape[-2:],
                mode="bilinear",
                align_corners=False,
            )

        # instantiate FocalLoss
        focal_fn = smp.losses.FocalLoss(
            mode='multiclass',
            alpha=None,      # or e.g. [0.25,0.75,…] per class
            gamma=2.0,       # focusing parameter γ
            ignore_index=IGNORE_INDEX
        )

        # compute focal loss
        loss = focal_fn(logits, labels)

        # return as HF expects
        return (loss, outputs) if return_outputs else loss
```

9.Trainer configuration: instantiate the Focal Loss HF Trainer with model, TrainingArguments, train and val datasets, metric function, and an EarlyStoppingCallback (patience = 3)

```python
from transformers import Trainer, EarlyStoppingCallback

trainer_segformer_b0_finetuned_cityscapes_with_data_augmentation_focal_loss = FocalLossTrainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    compute_metrics=compute_metrics,
    callbacks=[EarlyStoppingCallback(early_stopping_patience=3)],
)
```

10.Segformer Model training

```python
_ = trainer_segformer_b0_train_from_scratch_with_data_augmentation.train()
```

11.Code for the the evaluation metrics displayed after training

```python
import pandas as pd
from IPython.display import display

# 1) Pull out the "best by mean_iou" log entry
eval_logs = [log for log in trainer_segformer_b0_train_from_scratch_with_data_augmentation.state.log_history if "eval_mean_iou" in log]
best_log  = max(eval_logs, key=lambda x: x["eval_mean_iou"])

# 2) Define the exact 22 columns in order
cols = [
    "eval_loss",
    "eval_pixel_accuracy",
    # "eval_mean_accuracy",
    "eval_mean_iou",
    "eval_drivable_iou",
    "eval_non_drivable_iou",
    "eval_living_things_iou",
    "eval_vehicles_iou",
    "eval_road_side_objects_iou",
    "eval_far_objects_iou",
    "eval_sky_iou",
    # "eval_drivable_accuracy",
    # "eval_non_drivable_accuracy",
    # "eval_living_things_accuracy",
    # "eval_vehicles_accuracy",
    # "eval_road_side_objects_accuracy",
    # "eval_far_objects_accuracy",
    # "eval_sky_accuracy",
    "eval_runtime",
    "eval_samples_per_second",
    "eval_steps_per_second",
    "epoch",
]

# 3) Build the full one-row DataFrame
data = {k: best_log.get(k) for k in cols}
df = pd.DataFrame([data], index=["Final Metrics"])

# 4) Split into three DataFrames

# 4a) Summary: first 4 + last 4 columns
summary_cols = cols[:4] + cols[-4:]
df_summary = df[summary_cols].copy()
df_summary.index = ["Final Metrics"]

# 4b) IoU per class: columns 4–10
iou_cols = cols[4:11]
df_iou = df[iou_cols].copy()
df_iou.index = ["Final Metrics: IoU Per Class"]

# # 4c) Accuracy per class: columns 11–17
# acc_cols = cols[11:18]
# df_acc = df[acc_cols].copy()
# df_acc.index = ["Final Metrics: Accuracy Per Class"]

# 5) Ensure all columns show
pd.set_option("display.max_columns", None)
```

Overall Final Metrics

```python
# 6) Display the three tables
display(df_summary)
```

Final Metrics of IOU Per Class

```python
display(df_iou)
```

12.Plotting Training and Validation Loss For segformer

```
import pandas as pd
import matplotlib.pyplot as plt

# 1) turn the Trainer log history into a DataFrame
logs = pd.DataFrame(trainer_segformer_b0_train_from_scratch_with_data_augmentation.state.log_history)

# 2) isolate one row per epoch where the *training* loss was logged
#    (the Trainer calls this column "loss", not "train_loss")
train_logs = (
    logs[logs["loss"].notna()]
        .drop_duplicates("epoch", keep="first")
        .loc[:, ["epoch", "loss"]]
        .rename(columns={"loss": "train_loss"})
)

# 3) isolate one row per epoch where the *validation* loss was logged
val_logs = (
    logs[logs["eval_loss"].notna()]
        .drop_duplicates("epoch", keep="first")
        .loc[:, ["epoch", "eval_loss"]]
)

# 4) merge them so we only keep epochs where *both* exist
df = pd.merge(train_logs, val_logs, on="epoch")

print(df)  # just to sanity-check you've got both columns

# 5) plot
plt.figure(figsize=(7,4))
plt.plot(df["epoch"], df["train_loss"], marker="o", label="Train Loss")
plt.plot(df["epoch"], df["eval_loss"],  marker="o", label="Val   Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Train vs Validation Loss")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

13.Total Parameter Count of the Model for Segformer

```
def count_parameters(model):
    total = sum(p.numel() for p in model.parameters())
    trainable = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f"Total parameters: {total:,}")
    print(f"Trainable parameters: {trainable:,}")


count_parameters(model)
```

14.Plot for Mean iou per epoch and pixel accuracy per epoch

```python
import matplotlib.pyplot as plt

# Pixel Accuracy plot
pixel_acc = metrics_per_epoch["Pixel Accuracy"]
plt.figure(figsize=(7, 4))
plt.plot(metrics_per_epoch["epoch"], pixel_acc, marker="o", label="Pixel Accuracy")
for x, y in zip(metrics_per_epoch["epoch"], pixel_acc):
    plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")  # 4 decimals; adjust as needed
plt.xlabel("Epoch")
plt.ylabel("Pixel Accuracy")
plt.title("Pixel Accuracy per Epoch")
plt.legend()
plt.ylim(pixel_acc.min() - 0.01, pixel_acc.max() + 0.01)
plt.grid(True)
plt.tight_layout()
plt.show()

# # Mean Accuracy plot
# mean_acc = metrics_per_epoch["Mean Accuracy"]
# plt.figure(figsize=(7, 4))
# plt.plot(metrics_per_epoch["epoch"], mean_acc, marker="s", color="orange", label="Mean Accuracy")
# for x, y in zip(metrics_per_epoch["epoch"], mean_acc):
#     plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")
# plt.xlabel("Epoch")
# plt.ylabel("Mean Accuracy")
# plt.title("Mean Accuracy per Epoch")
# plt.legend()
# plt.ylim(mean_acc.min() - 0.01, mean_acc.max() + 0.01)
# plt.grid(True)
# plt.tight_layout()
# plt.show()

# Mean IoU plot
mean_iou = metrics_per_epoch["Mean IoU"]
plt.figure(figsize=(7, 4))
plt.plot(metrics_per_epoch["epoch"], mean_iou, marker="^", color="green", label="Mean IoU")
for x, y in zip(metrics_per_epoch["epoch"], mean_iou):
    plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")
plt.xlabel("Epoch")
plt.ylabel("Mean IoU")
plt.title("Mean IoU per Epoch")
plt.legend()
plt.ylim(mean_iou.min() - 0.01, mean_iou.max() + 0.01)
plt.grid(True)
plt.tight_layout()
plt.show()
```

15.Bar Plot for IoU per Class for the Segformer Model

```python
import matplotlib.pyplot as plt

# 1) Extract the best-by-mean_iou log entry
eval_logs = [l for l in trainer_segformer_b0_train_from_scratch_with_data_augmentation.state.log_history if "eval_mean_iou" in l]
best_log  = max(eval_logs, key=lambda x: x["eval_mean_iou"])

# 2) Define class names and pull IoU values from best_log
class_names = [
    "Drivable", "Non-drivable", "Living things",
    "Vehicles", "Road-side objects", "Far objects", "Sky"
]
iou_values = [
    best_log["eval_drivable_iou"],
    best_log["eval_non_drivable_iou"],
    best_log["eval_living_things_iou"],
    best_log["eval_vehicles_iou"],
    best_log["eval_road_side_objects_iou"],
    best_log["eval_far_objects_iou"],
    best_log["eval_sky_iou"],
]

# 3) Plot
plt.figure(figsize=(8, 5))
bars = plt.bar(class_names, iou_values)
plt.xlabel("Class")
plt.ylabel("IoU")
plt.title(" Final evaluation metrics for IoU per Class")
plt.ylim(0, 1.0)
plt.xticks(rotation=45, ha="right")

# 4) Annotate each bar with its IoU value
for bar in bars:
    height = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        height + 0.01,
        f"{height:.6f}",
        ha="center",
        va="bottom"
    )

plt.tight_layout()
plt.show()
```

16.Visualizing Ground Truth vs Predicted Semantic Map

```python
import torch
import numpy as np
import matplotlib.pyplot as plt

# Your color palette for 7 classes + void (same as your EDA)
CLASS_COLORS = [
    (128,  64,128),   # Drivable (purple)
    (244,  35,232),   # Non-drivable (pink)
    (0, 255,255),     # Living things (cyan)
    (255,128,  0),    # Vehicles (orange)
    (190,153,153),    # Road side objects (light brown)
    (0, 128,128),     # Far objects (teal)
    (70,130,180),     # Sky (sky blue)
    (0, 0, 0) ,       # Void (black)
]

def colorize_mask(mask):
    # mask: (H, W) np array of int
    mask_color = np.zeros((*mask.shape, 3), dtype=np.uint8)
    for class_id, color in enumerate(CLASS_COLORS[:-1]):
        mask_color[mask == class_id] = color
    mask_color[mask == 255] = CLASS_COLORS[-1]  # void/ignore
    return mask_color

def unnormalize_img(img):
    # img: (3, H, W), normalized
    mean = np.array([0.485, 0.456, 0.406])
    std  = np.array([0.229, 0.224, 0.225])
    img = img.permute(1,2,0).cpu().numpy()  # (H, W, 3)
    img = img * std + mean
    return np.clip(img, 0, 1)

def visualize_sample(model, dataset, idx, device='cuda'):
    model.eval()
    sample = dataset[idx]
    img = sample['pixel_values'].unsqueeze(0).to(device)  # (1, 3, H, W)
    with torch.no_grad():
        logits = model(pixel_values=img).logits  # (1, num_classes, H, W)
        pred = torch.argmax(logits, dim=1).squeeze().cpu().numpy()  # (H, W)
    # Prepare for display
    gt = sample['labels'].cpu().numpy() if torch.is_tensor(sample['labels']) else sample['labels']
    orig = unnormalize_img(sample['pixel_values'])
    gt_color = colorize_mask(gt)
    pred_color = colorize_mask(pred)
    # Plot
    plt.figure(figsize=(16,5))
    plt.subplot(1,3,1); plt.imshow(orig); plt.title("Original Image"); plt.axis('off')
    plt.subplot(1,3,2); plt.imshow(gt_color); plt.title("Ground Truth Segmentation Map"); plt.axis('off')
    plt.subplot(1,3,3); plt.imshow(pred_color); plt.title("Predicted Segmentation Map"); plt.axis('off')
    plt.tight_layout()
    plt.show()

# Example: visualize 3 random samples from validation set
import random
random_indices = random.sample(range(len(val_ds)), 3)
for idx in random_indices:
    visualize_sample(model, val_ds, idx)
```

B.Mask2former Pipeline

1.Importing Libraries

```
import os
import random
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset
from PIL import Image
import albumentations as A
from transformers import (
    Mask2FormerImageProcessor,
    Mask2FormerForUniversalSegmentation,
    TrainingArguments,
    Trainer,
)
import evaluate
```

2.Collection of Image & Mask Paths

Traverse the folder structure to list `.jpg` and matching `_label.png` files.

```
def get_image_mask_paths(root_img, root_mask, split="train"):
    image_paths, mask_paths = [], []
    split_dir = os.path.join(root_img, split)
    for city in os.listdir(split_dir):
        img_folder  = os.path.join(root_img, split, city)
        mask_folder = os.path.join(root_mask, split, city)
        for fname in os.listdir(img_folder):
            if fname.endswith("_image.jpg"):
                image_paths.append(os.path.join(img_folder, fname))
                mask_name = fname.replace("_image.jpg", "_label.png")
                mask_paths.append(os.path.join(mask_folder, mask_name))
    return image_paths, mask_paths


IMG_ROOT  = "dataset/idd_lite/leftImg8bit"
MASK_ROOT = "dataset/idd_lite/gtFine"

train_img, train_mask = get_image_mask_paths(IMG_ROOT, MASK_ROOT, "train")
val_img,   val_mask   = get_image_mask_paths(IMG_ROOT, MASK_ROOT, "val")
```

```
def get_paths(root_img, root_mask, split):
    imgs, msks = [], []
    base_i = os.path.join(root_img, split)
    base_m = os.path.join(root_mask, split)
    for city in os.listdir(base_i):
        for fn in os.listdir(os.path.join(base_i, city)):
            if fn.endswith(".jpg"):
                imgs.append(os.path.join(base_i, city, fn))
                msks.append(
                    os.path.join(
                        base_m,
                        city,
                        fn.replace("_image.jpg", "_label.png")
                    )
                )
    return imgs, msks

train_imgs, train_msks = get_paths(IMG_ROOT, MSK_ROOT, "train")
val_imgs,   val_msks   = get_paths(IMG_ROOT, MSK_ROOT, "val")
```

3.Data preprocessing and Data Augmentation

1. Data Preprocessing step

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

# 2) your dataset roots
IMG_ROOT = "dataset/idd20k_lite/leftImg8bit"
MSK_ROOT = "dataset/idd20k_lite/gtFine"

# 3) classes + mappings
CLASS_NAMES = [
    "Drivable", "Non-drivable", "Living things",
    "Vehicles", "Road side objects", "Far objects", "Sky",
]
NUM_CLASSES = len(CLASS_NAMES)
IGNORE_INDEX = 255
ID2LABEL = {i: n for i, n in enumerate(CLASS_NAMES)}
LABEL2ID = {n: i for i, n in enumerate(CLASS_NAMES)}
```

3.2 Data augmentation

```
train_tfms = A.Compose([
    A.RandomResizedCrop(size=(512, 512), scale=(0.7, 1.0), ratio=(0.75, 1.33), p=1.0),
    A.HorizontalFlip(p=0.5),
    A.Affine(translate_percent=0.05, scale=(0.9, 1.1), rotate=(-15, 15), p=0.5),
    A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.05, p=0.5),
    A.GaussianBlur(p=0.2),
    # A.CoarseDropout(num_holes=8, max_h_size=8, max_w_size=8, p=0.5),  # Optional
], additional_targets={"mask": "mask"})

# Validation transforms: only resize for fair evaluation
val_tfms = A.Compose([
    A.Resize(512, 512),
], additional_targets={"mask": "mask"})
```

3.3 Preprocessing step to Setup the Mask2former Image Processor and do rescale,normalization and to make sure ,it doesn't include the include the 255 ignore label ID when training

```
from transformers import Mask2FormerImageProcessor

processor = Mask2FormerImageProcessor(
    num_labels=7,
    do_reduce_labels=False,
    do_rescale=True,
    do_normalize=True,
    ignore_index=255
)
```

3.4 Data preprocessing step where we create Custom PyTorch Dataset for IDD- Lite,we define a PyTorch Dataset that loads images and masks, applies preprocessing, and ensures all invalid mask values are mapped to 255 (ignore index).

```
class IDDSegDataset(Dataset):
    def __init__(self, img_paths, mask_paths, tfms, proc):
        self.img_paths  = img_paths
        self.mask_paths = mask_paths
        self.tfms       = tfms
        self.proc       = proc

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

    def __getitem__(self, i):
        img  = np.array(Image.open(self.img_paths[i]).convert("RGB"))
        mask = np.array(Image.open(self.mask_paths[i]), dtype=np.int64)
        mask = np.where((mask < 0)|(mask >= NUM_CLASSES), IGNORE_INDEX, mask)

        if self.tfms:
            out = self.tfms(image=img, mask=mask)
            img, mask = out["image"], out["mask"]

        bf = self.proc(
            images=[img],
            segmentation_maps=[mask],
            return_tensors="pt"
        )
        return {
            "pixel_values": bf["pixel_values"].squeeze(0),
            "mask_labels":  bf["mask_labels"][0],
            "class_labels": bf["class_labels"][0],
            "labels":       torch.from_numpy(mask).long(),
        }

train_ds = IDDSegDataset(train_imgs, train_msks, train_tfms, processor)
val_ds   = IDDSegDataset(val_imgs,   val_msks,   val_tfms,   processor)
```

3.5 Data Collate Function

```
def collate_fn(batch):
    return {
        "pixel_values": torch.stack([b["pixel_values"] for b in batch]),
        "mask_labels":  [b["mask_labels"]  for b in batch],
        "class_labels": [b["class_labels"] for b in batch],
        "labels":       torch.stack([b["labels"]  for b in batch]),
    }
```

4.Model Intialization

4.1. For Domain Specific Pretraining and Fine tuning on IDD-lite

Initialization Mask2former pretrained cityscapes model and , and configure the segmentation head for 7 classes, ignoring 255 for loss/metrics and also the encoder changes with respect to swin-tiny to swin-small.

```
model = Mask2FormerForUniversalSegmentation.from_pretrained(
    "facebook/mask2former-swin-tiny-cityscapes-semantic",
    num_labels=7,
    id2label=ID2LABEL,
    label2id=LABEL2ID,
    ignore_mismatched_sizes=True,
    ignore_value=255,
    use_safetensors=True,

)
```

4.2. For training From Scratch on IDD-lite

Initialization Mask2former pretrained cityscapes model but initialize the Mask2formerConfig to train from scratch and , and configure the segmentation head for 7 classes, ignoring 255 for loss/metrics and also the encoder changes with respect to swin-tiny to swin-small.

```
from transformers import Mask2FormerConfig, Mask2FormerForUniversalSegmentation

config = Mask2FormerConfig.from_pretrained(
    "facebook/mask2former-swin-tiny-cityscapes-semantic",
    num_labels=7,
    id2label=ID2LABEL,
    label2id=LABEL2ID,
    ignore_value=255
)

# 2 Instantiate the model **from config**, giving you random init
model = Mask2FormerForUniversalSegmentation(config)
```

5.Computing Metrics: Mean IoU, Pixel Accuracy, Per-Class IoU

We use Hugging Face's evaluate library to compute mean IoU, pixel accuracy, and per-class IoU, while ignoring 255 pixels in both training and evaluation.

```python
import torch
import numpy as np
import evaluate
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# Your class names
CLASS_NAMES = [
    "drivable",
    "non_drivable",
    "living_things",
    "vehicles",
    "road_side_objects",
    "far_objects",
    "sky",
]

miou = evaluate.load("mean_iou")

def compute_metrics(eval_pred):
    preds, labels = eval_pred

    # 1. Convert tensors to numpy arrays (your style)
    np_preds = []
    for p in preds:
        if isinstance(p, torch.Tensor):
            np_preds.append(p.cpu().numpy())
        else:
            np_preds.append(p)
    np_labels = []
    for l in labels:
        if isinstance(l, torch.Tensor):
            np_labels.append(l.cpu().numpy())
        else:
            np_labels.append(l)

    # 2. Compute HF metrics
    raw = miou.compute(
        predictions=np_preds,
        references=np_labels,
        num_labels=7,
        ignore_index=255,
    )

    pixel_acc   = raw["overall_accuracy"]
    mean_iou    = raw["mean_iou"]
    # mean_acc    = raw.get("mean_accuracy", None)
    per_cat_iou = raw["per_category_iou"]
    # per_cat_acc = raw.get("per_category_accuracy", None)

    # 3. Confusion Matrix (mask out ignore pixels, flatten)
    p_flat = np.concatenate([m.flatten() for m in np_preds])
    l_flat = np.concatenate([m.flatten() for m in np_labels])
    mask   = (l_flat != 255)
    p_flat = p_flat[mask]
    l_flat = l_flat[mask]
    cm = confusion_matrix(l_flat, p_flat, labels=list(range(len(CLASS_NAMES))))
    cm_norm = cm.astype(float) / (cm.sum(axis=1, keepdims=True) + 1e-12)

    plt.figure(figsize=(6,5))
    plt.imshow(cm_norm, vmin=0, vmax=1, interpolation="nearest")
    plt.title("Confusion Matrix")
    plt.colorbar(label="Frac of true-class → predicted-class")
    ticks = np.arange(len(CLASS_NAMES))
    plt.xticks(ticks, CLASS_NAMES, rotation=45, ha="right")
    plt.yticks(ticks, CLASS_NAMES)
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.tight_layout()
    plt.show()

    # 4. Per-class results as dicts
    per_class_iou = {
        f"{CLASS_NAMES[i]}_iou": float(per_cat_iou[i])
        for i in range(len(CLASS_NAMES))
    }
    # per_class_acc = {}
    # if per_cat_acc is not None:
    #     per_class_acc = {
    #         f"{CLASS_NAMES[i]}_accuracy": float(per_cat_acc[i])
    #         for i in range(len(CLASS_NAMES))
    #     }

    # 5. Return everything
    return {
        "pixel_accuracy": pixel_acc,
        # "mean_accuracy": mean_acc,
        "mean_iou": mean_iou,
        **per_class_iou,
        # **per_class_acc,
    }
```

6. Custom Trainer Subclass

```python
class Mask2FormerTrainer(Trainer):
    def to_device(self, data, device):
        data = super().to_device(data, device)
        data["mask_labels"]  = [m.to(device) for m in data["mask_labels"]]
        data["class_labels"] = [c.to(device) for c in data["class_labels"]]
        return data

    def compute_loss(self, model, inputs, return_outputs=False,**kwargs):
        inputs_labels = inputs.pop("labels")
        outputs = model(
            pixel_values=inputs["pixel_values"],
            mask_labels=inputs["mask_labels"],
            class_labels=inputs["class_labels"],
        )
        loss = outputs.loss
        return (loss, outputs) if return_outputs else loss

    def prediction_step(self, model, inputs, prediction_loss_only, ignore_keys=None):
        labels = inputs.pop("labels")
        with torch.no_grad():
            outputs = model(
                pixel_values=inputs["pixel_values"],
                mask_labels=inputs["mask_labels"],
                class_labels=inputs["class_labels"],
            )
        loss = outputs.loss
        preds = processor.post_process_semantic_segmentation(
            outputs,
            target_sizes=[lab.shape for lab in labels],
        )
        label_list = [lab.detach().cpu() for lab in labels]
        if prediction_loss_only:
            return (loss, None, None)
        return (loss, preds, label_list)
```

7.Training Configuration

```python
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./outputs_Mask2former/Mask2former_exp3/Mask2former_swin_tiny_trained_from_scratch_with_data_augmentation",            # Where checkpoints & outputs are saved
    logging_dir="./outputs_Mask2former/Mask2former_exp3/Mask2former_swin_tiny_trained_from_scratch_with_data_augmentation/logs",      # TensorBoard logs in output dir
    report_to=["tensorboard"],                          # Enable TensorBoard logging

    num_train_epochs=10,                                # Number of training epochs
    per_device_train_batch_size=4,                      # Training batch size
    per_device_eval_batch_size=4,                       # Eval batch size
    optim="adamw_torch",
    learning_rate=0.00006,                                 # Learning rate (from Mask2Former paper)
    weight_decay=0.01,
    warmup_ratio=0.1,                                                       # Weight decay for regularization
    lr_scheduler_type="polynomial",                     # Poly LR scheduler


    # logging_strategy="steps",      # log training metrics every logging_steps
    # logging_steps=50,              # ← change to how often you want train-loss logs
    # eval_strategy="steps",         # run validation every eval_steps
    # eval_steps=200,                # ← change to how often you want val-loss/metrics
    # save_strategy="steps",         # optionally checkpoint every save_steps
    # save_steps=200,

    # eval_strategy="epoch",                       # Evaluate after each epoch
    # save_strategy="epoch",
    logging_strategy="epoch",    # log training metrics once at end of each epoch
    eval_strategy="epoch",       # run validation once at end of each epoch
    save_strategy="epoch",       # checkpoint once at end of each epoch
    save_total_limit=1,                                 # Save after each epoch
    # save_total_limit=10,                                  # Only keep the best checkpoint
    load_best_model_at_end=True,                        # Restore best checkpoint after training
    metric_for_best_model="mean_iou",                   # Metric to pick best checkpoint
    greater_is_better=True,                             # Higher mean IoU is better

    # logging_strategy="epoch",                           # Log metrics after each epoch
    # logging_steps=50,                                   # (optional: only if logging_strategy="steps")

    fp16=True,                                          # Mixed precision for faster training (if supported)
    seed=42,                                            # Reproducibility
    remove_unused_columns=False,                        # Required for custom dataset/collate
)
```

8.Trainer configuration: instantiate the HF Trainer with model, TrainingArguments, train and val datasets, metric function, and an EarlyStoppingCallback (patience = 3)

```python
from transformers import Trainer, EarlyStoppingCallback
trainer_mask2former_swin_tiny_trained_from_scratch_with_data_augmentation = Mask2FormerTrainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
    callbacks= [EarlyStoppingCallback(early_stopping_patience=3)],
)
```

9.Model Training

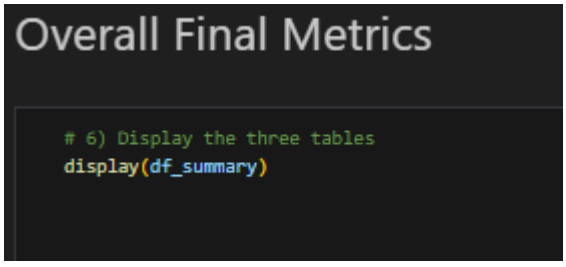
```python
_ = trainer_mask2former_swin_tiny_trained_from_scratch_with_data_augmentation.train()
```

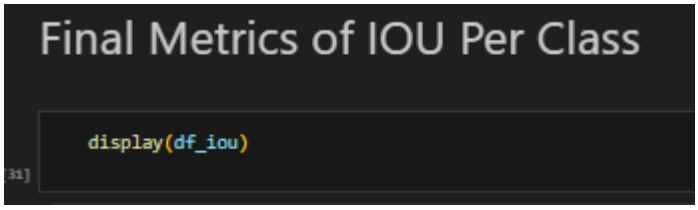
10.Plotting Training and Validation loss for Mask2former

```python
import pandas as pd
import matplotlib.pyplot as plt

# 1) turn the Trainer log history into a DataFrame
logs = pd.DataFrame(trainer_mask2former_swin_tiny_trained_from_scratch_with_data_augmentation.state.log_history)

# 2) isolate one row per epoch where the *training* loss was logged
#    (the Trainer calls this column "loss", not "train_loss")
train_logs = (
    logs[logs["loss"].notna()]
        .drop_duplicates("epoch", keep="first")
        .loc[:, ["epoch", "loss"]]
        .rename(columns={"loss": "train_loss"})
)

# 3) isolate one row per epoch where the *validation* loss was logged
val_logs = (
    logs[logs["eval_loss"].notna()]
        .drop_duplicates("epoch", keep="first")
        .loc[:, ["epoch", "eval_loss"]]
)

# 4) merge them so we only keep epochs where *both* exist
df = pd.merge(train_logs, val_logs, on="epoch")

print(df)  # just to sanity-check you've got both columns

# 5) plot
plt.figure(figsize=(7,4))
plt.plot(df["epoch"], df["train_loss"], marker="o", label="Train Loss")
plt.plot(df["epoch"], df["eval_loss"],  marker="o", label="Val   Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Train vs Validation Loss")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

11.Code for the the evaluation metrics displayed after training

```python
import pandas as pd
from IPython.display import display

# 1) Pull out the "best by mean_iou" log entry
eval_logs = [log for log in trainer_mask2former_swin_tiny_trained_from_scratch_with_data_augmentation.state.log_history if "eval_mean_iou" in log]
best_log  = max(eval_logs, key=lambda x: x["eval_mean_iou"])

# 2) Define the exact 22 columns in order
cols = [
    "eval_loss",
    "eval_pixel_accuracy",
    # "eval_mean_accuracy",
    "eval_mean_iou",
    "eval_drivable_iou",
    "eval_non_drivable_iou",
    "eval_living_things_iou",
    "eval_vehicles_iou",
    "eval_road_side_objects_iou",
    "eval_far_objects_iou",
    "eval_sky_iou",
    # "eval_drivable_accuracy",
    # "eval_non_drivable_accuracy",
    # "eval_living_things_accuracy",
    # "eval_vehicles_accuracy",
    # "eval_road_side_objects_accuracy",
    # "eval_far_objects_accuracy",
    # "eval_sky_accuracy",
    "eval_runtime",
    "eval_samples_per_second",
    "eval_steps_per_second",
    "epoch",
]

# 3) Build the full one-row DataFrame
data = {k: best_log.get(k) for k in cols}
df = pd.DataFrame([data], index=["Final Metrics"])

# 4) Split into three DataFrames

# 4a) Summary: first 4 + last 4 columns
summary_cols = cols[:4] + cols[-4:]
df_summary = df[summary_cols].copy()
df_summary.index = ["Final Metrics"]

# 4b) IoU per class: columns 4–10
iou_cols = cols[4:11]
df_iou = df[iou_cols].copy()
df_iou.index = ["Final Metrics: IoU Per Class"]

# # 4c) Accuracy per class: columns 11–17
# acc_cols = cols[11:18]
# df_acc = df[acc_cols].copy()
# df_acc.index = ["Final Metrics: Accuracy Per Class"]

# 5) Ensure all columns show
pd.set_option("display.max_columns", None)
```





12.Total Parameter Count of mask2former model

```
def count_parameters(model):
    total = sum(p.numel() for p in model.parameters())
    trainable = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f"Total parameters: {total:,}")
    print(f"Trainable parameters: {trainable:,}")


count_parameters(model)
```

13.Plot for Mean iou per epoch and pixel accuracy per epoch for mask2former

```python
import matplotlib.pyplot as plt

# Pixel Accuracy plot
pixel_acc = metrics_per_epoch["Pixel Accuracy"]
plt.figure(figsize=(7, 4))
plt.plot(metrics_per_epoch["epoch"], pixel_acc, marker="o", label="Pixel Accuracy")
for x, y in zip(metrics_per_epoch["epoch"], pixel_acc):
    plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")  # 4 decimals; adjust as needed
plt.xlabel("Epoch")
plt.ylabel("Pixel Accuracy")
plt.title("Pixel Accuracy per Epoch")
plt.legend()
plt.ylim(pixel_acc.min() - 0.01, pixel_acc.max() + 0.01)
plt.grid(True)
plt.tight_layout()
plt.show()

# Mean Accuracy plot
# mean_acc = metrics_per_epoch["Mean Accuracy"]
# plt.figure(figsize=(7, 4))
# plt.plot(metrics_per_epoch["epoch"], mean_acc, marker="s", color="orange", label="Mean Accuracy")
# for x, y in zip(metrics_per_epoch["epoch"], mean_acc):
#     plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")
# plt.xlabel("Epoch")
# plt.ylabel("Mean Accuracy")
# plt.title("Mean Accuracy per Epoch")
# plt.legend()
# plt.ylim(mean_acc.min() - 0.01, mean_acc.max() + 0.01)
# plt.grid(True)
# plt.tight_layout()
# plt.show()

# Mean IoU plot
mean_iou = metrics_per_epoch["Mean IoU"]
plt.figure(figsize=(7, 4))
plt.plot(metrics_per_epoch["epoch"], mean_iou, marker="p", color="green", label="Mean IoU")
for x, y in zip(metrics_per_epoch["epoch"], mean_iou):
    plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")
plt.xlabel("Epoch")
plt.ylabel("Mean IoU")
plt.title("Mean IoU per Epoch")
plt.legend()
plt.ylim(mean_iou.min() - 0.01, mean_iou.max() + 0.01)
plt.grid(True)
plt.tight_layout()
plt.show()
```

14.bar plot of final iou per class for mask2former

```python
import matplotlib.pyplot as plt

# 1) Extract the best-by-mean_iou log entry
eval_logs = [l for l in trainer_mask2former_swin_tiny_trained_from_scratch_with_data_augmentation.state.log_history if "eval_mean_iou" in l]
best_log  = max(eval_logs, key=lambda x: x["eval_mean_iou"])

# 2) Define class names and pull IoU values from best_log
class_names = [
    "Drivable", "Non-drivable", "Living things",
    "Vehicles", "Road-side objects", "Far objects", "Sky"
]
iou_values = [
    best_log["eval_drivable_iou"],
    best_log["eval_non_drivable_iou"],
    best_log["eval_living_things_iou"],
    best_log["eval_vehicles_iou"],
    best_log["eval_road_side_objects_iou"],
    best_log["eval_far_objects_iou"],
    best_log["eval_sky_iou"],
]

# 3) Plot
plt.figure(figsize=(8, 5))
bars = plt.bar(class_names, iou_values)
plt.xlabel("Class")
plt.ylabel("IoU")
plt.title(" Final evaluation metrics for IoU per Class")
plt.ylim(0, 1.0)
plt.xticks(rotation=45, ha="right")

# 4) Annotate each bar with its IoU value
for bar in bars:
    height = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        height + 0.01,
        f"{height:.6f}",
        ha="center",
        va="bottom"
    )

plt.tight_layout()
plt.show()
```

15.Visualizing Ground Truth vs Predicted Semantic Map for mask2former

```python
import matplotlib.pyplot as plt

# Pixel Accuracy plot
pixel_acc = metrics_per_epoch["Pixel Accuracy"]
plt.figure(figsize=(7, 4))
plt.plot(metrics_per_epoch["epoch"], pixel_acc, marker="o", label="Pixel Accuracy")
for x, y in zip(metrics_per_epoch["epoch"], pixel_acc):
    plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")  # 4 decimals; adjust as needed
plt.xlabel("Epoch")
plt.ylabel("Pixel Accuracy")
plt.title("Pixel Accuracy per Epoch")
plt.legend()
plt.ylim(pixel_acc.min() - 0.01, pixel_acc.max() + 0.01)
plt.grid(True)
plt.tight_layout()
plt.show()

# Mean Accuracy plot
# mean_acc = metrics_per_epoch["Mean Accuracy"]
# plt.figure(figsize=(7, 4))
# plt.plot(metrics_per_epoch["epoch"], mean_acc, marker="s", color="orange", label="Mean Accuracy")
# for x, y in zip(metrics_per_epoch["epoch"], mean_acc):
#     plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")
# plt.xlabel("Epoch")
# plt.ylabel("Mean Accuracy")
# plt.title("Mean Accuracy per Epoch")
# plt.legend()
# plt.ylim(mean_acc.min() - 0.01, mean_acc.max() + 0.01)
# plt.grid(True)
# plt.tight_layout()
# plt.show()

# Mean IoU plot
mean_iou = metrics_per_epoch["Mean IoU"]
plt.figure(figsize=(7, 4))
plt.plot(metrics_per_epoch["epoch"], mean_iou, marker="p", color="green", label="Mean IoU")
for x, y in zip(metrics_per_epoch["epoch"], mean_iou):
    plt.text(x, y + 0.002, f"{y:.4f}", ha="center", va="bottom")
plt.xlabel("Epoch")
plt.ylabel("Mean IoU")
plt.title("Mean IoU per Epoch")
plt.legend()
plt.ylim(mean_iou.min() - 0.01, mean_iou.max() + 0.01)
plt.grid(True)
plt.tight_layout()
plt.show()
```