Neural Networks and Deep Learning

Assignment 3: Implementation and Analysis of Fast-SCNN

University of Tehran

Faculty of Electrical and Computer Engineering

Submitted By:

Mohammad Taha Majlesi
Student ID: 810101504

September 26, 2025

Contents

1 Question 1: Urban Scene Segmentation **5**
1.1 Description of the Proposed Model (Fast-SCNN) . . . 5
1.2 Dataset Preparation and Analysis . . . 7
1.2.1 Dataset Analysis: CamVid . . . 7
1.3 Optimizers, Metrics, and Loss Function . . . 8
1.3.1 Evaluation Metrics . . . 8
1.3.2 Loss Function: Categorical Cross-Entropy . . . 9
1.3.3 Optimizer: Adam with a Polynomial Decay Schedule . . . 10
1.4 Model Implementation Details . . . 10
1.4.1 Depthwise Separable Convolution . . . 10
1.4.2 Inverted Residual Block . . . 13
1.4.3 Pyramid Pooling Module (PPM) . . . 16
1.4.4 Feature Fusion Module (FFM) . . . 17
1.4.5 Final Model Assembly . . . 18
1.5 Training and Evaluation Experiments . . . 19
1.5.1 Baseline Training with Cross-Entropy Loss . . . 19
1.5.2 Experiment 2: Training with IoU Loss . . . 20
1.5.3 Experiment 3: Training with Dice Loss . . . 22
1.5.4 Experiment 4: Data Augmentation . . . 24
1.6 Final Evaluation on the Test Set . . . 28
1.7 Conclusion . . . 29

List of Figures

1 The high-level architecture of the Fast-SCNN model, illustrating the flow from input image through the four main stages: Learning to Downsample, Global Feature Extractor, Feature Fusion, and the final Classifier. 6
2 Distribution of pixels per class across the entire training set. The y-axis is on a logarithmic scale, highlighting the extreme imbalance between common classes (like 'Road') and rare classes (like 'Bicyclist'). 7
3 Four sample frames from the CamVid dataset. The top row shows the original input images, and the bottom row shows the corresponding ground truth segmentation masks, where each color represents a unique object class. 8
4 Python implementation of the Dice Coefficient metric using TensorFlow. The function calculates the Dice score for each class and then returns the mean score across all classes. 9
5 Python implementation of the IoU Score metric using TensorFlow. Similar to the Dice implementation, it computes the per-class IoU and then averages them. 9
6 Illustration of a standard convolution operation. A single filter moves across the input volume, processing all channels simultaneously. 11
7 Illustration of the first step: Depthwise Convolution. A separate filter is applied to each input channel. 12
8 Illustration of the full Depthwise Separable Convolution, combining the depthwise and pointwise steps. 13
9 Diagram of a standard residual block. 14
10 Diagram of the Inverted Residual Block used in Fast-SCNN. Note the narrow-wide-narrow structure. 14
11 Code snippet showing the implementation of the Inverted Residual Block (termed 'Bottleneck' in the code). 15
12 Conceptual diagram of the Pyramid Pooling Module (PPM). 16
13 Code snippet for the implementation of the Pyramid Pooling Module. . . 17
15 Model Accuracy (left) and Dice Coefficient (right) during training. 19
16 Model IoU Score (left) and Loss (right) during training. 19
17 Sample prediction results from the baseline model. Top: original image, Middle: ground truth mask, Bottom: predicted mask. The model successfully identifies major classes like road, car, building, and sky. 20
18 Accuracy and Dice curves when training with IoU Loss. 21
19 IoU and Loss curves when training with IoU Loss. 21
20 Qualitative results from the model trained with IoU loss. The segmentation quality is slightly degraded, with more noisy predictions around object edges. 22

21 Accuracy and Dice curves when training with Dice Loss. 23
22 IoU and Loss curves when training with Dice Loss. 23
23 Qualitative results from the model trained with Dice loss. The segmentation is noticeably worse, with less coherent object shapes. 24
24 Examples of the data augmentation applied. 25
25 Accuracy and Dice curves when training with augmented data. 26
26 IoU and Loss curves when training with augmented data. 26
27 Qualitative results from the model trained with augmentation. Notably, smaller objects like people are not detected, likely due to the class imbalance being exacerbated by the simple augmentations. 27
28 Final model training curves on the combined dataset: Accuracy (left) and Dice (right). 28
29 Final model training curves on the combined dataset: IoU (left) and Loss (right). 28
30 Sample predictions on the test set. The model continues to perform well on large structural classes but struggles with smaller, more detailed objects, which is consistent with its performance on the validation set. 29

List of Tables

1 Structure of the Inverted Residual Block, as described in the reference paper. 15
2 Structure of the Feature Fusion Module. 18
3 Detailed architecture of our implemented Fast-SCNN model, matching the specification from the paper. 18
4 Summary of parameters per module in our implementation. 18
5 Final comparison of all models on the validation data. The baseline cross-entropy model without augmentation performed the best. 27
6 Final model performance on the Test set. 29

1 Question 1: Urban Scene Segmentation

1.1 Description of the Proposed Model (Fast-SCNN)

The academic paper introduces a model, **Fast-SCNN**, designed to achieve state-of-the-art performance in the domain of semantic segmentation of images, particularly for real-time applications. Semantic segmentation is the task of classifying every pixel in an image into a specific category (e.g., road, car, person, sky). This is a fundamental task for autonomous systems like self-driving cars and robotics, where understanding the surrounding environment is critical.

The primary challenge addressed by Fast-SCNN is the inherent trade-off between speed and accuracy. While large, complex models can achieve high accuracy, their computational cost makes them unsuitable for real-time deployment on resource-constrained hardware (like the embedded systems in a car). Fast-SCNN is architected to be both fast and accurate, making it viable for these applications without requiring powerful, energy-intensive GPUs.

The core innovation of Fast-SCNN is its **two-branch architecture**:

1. **A Deep, Low-Resolution Branch:** This branch processes a downsampled version of the input image. Its purpose is to capture **global context** and high-level semantic information. By working on a smaller image, it can use deeper layers with more complex feature extractors (like inverted residual blocks) without incurring a massive computational penalty. This branch answers the question, "What is the overall scene composition?"

2. **A Shallow, High-Resolution Branch:** This branch processes the original, high-resolution image using only a few, lightweight layers. Its goal is to capture fine-grained **spatial details**, like sharp object boundaries and small objects. This branch answers the question, "Where are the precise edges of objects?"

The final, high-accuracy segmentation mask is produced by fusing the outputs of these two branches. This design is highly efficient because, unlike traditional two-branch models that process the full image in both paths, Fast-SCNN shares the initial layers. This shared "Learning to Downsample" module extracts initial features that are then fed into both the deep and shallow branches, significantly reducing redundant computations.

Architectural Components The model is composed of four main stages, as illustrated in Figure 1.

- **Learning to Downsample:** The first stage consists of three convolutional layers (one standard, two depthwise separable) that rapidly reduce the spatial resolution

of the feature map while extracting basic low-level features. This is the shared backbone for both branches.

- **Global Feature Extractor:** This is the deep branch. It takes the low-resolution feature map from the previous stage and processes it through a series of efficient *inverted residual blocks* (bottlenecks), followed by a *Pyramid Pooling Module (PPM)*. This module captures context at multiple scales, which is crucial for robust scene understanding.
- **Feature Fusion:** This module intelligently merges the high-level contextual features from the deep branch with the low-level spatial details from the shallow branch. The outputs are simply added together element-wise.
- **Classifier:** The final stage consists of two depthwise separable convolution layers and a final standard convolution layer that upsamples the merged features and produces the final pixel-wise classification map.

Comparison to U-Net and FCN Fast-SCNN shares conceptual similarities with encoder-decoder models like FCN and U-Net, but with key differences for efficiency.

- **vs. U-Net:** U-Net is famous for its extensive skip connections, which merge features from the encoder path to the decoder path at multiple resolution levels. This is excellent for accuracy but computationally expensive. Fast-SCNN uses only a **single, lightweight skip connection** (the shallow branch) that fuses with the final output of the deep branch, making it much faster.
- **vs. FCN (Fully Convolutional Network):** FCN pioneered end-to-end segmentation. However, its upsampling path is often simplistic. Fast-SCNN's two-branch fusion is a more sophisticated way to recover spatial detail than simple transposed convolutions.

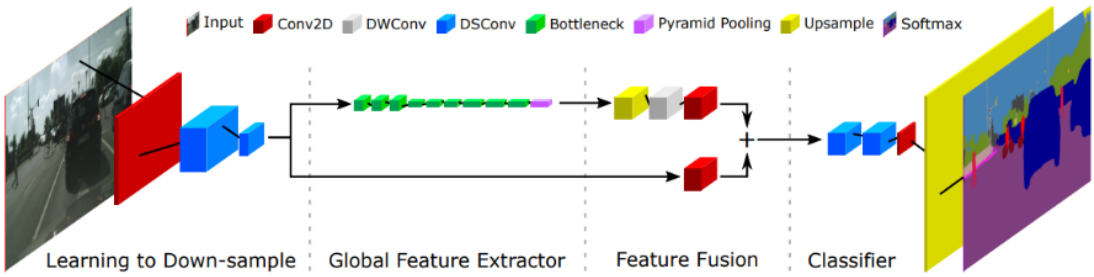

Figure 1: The high-level architecture of the Fast-SCNN model, illustrating the flow from input image through the four main stages: Learning to Downsample, Global Feature Extractor, Feature Fusion, and the final Classifier.

1.2 Dataset Preparation and Analysis

1.2.1 Dataset Analysis: CamVid

The dataset used for this project is **CamVid (Cambridge-driving Labeled Video Database)**, one of the pioneering datasets for semantic segmentation of street scenes. It consists of high-resolution video frames (960x720 pixels) captured from a car's perspective, with per-pixel annotations for 32 distinct semantic classes. For practical use, these are often merged into a smaller set of 12 classes, including one 'void' or 'unlabeled' class. The dataset is pre-split into 367 training, 101 validation, and 233 test images.

A critical characteristic of CamVid, and most real-world segmentation datasets, is severe **class imbalance**. As shown in Figure 2, classes like 'Road', 'Sky', and 'Building' occupy a massive number of pixels, while classes like 'Bicyclist', 'SignSymbol', and 'Fence' are extremely rare. This imbalance poses a major challenge for model training; a naive model could achieve high pixel-wise accuracy by simply predicting the majority class everywhere. This motivates our choice of more robust evaluation metrics and loss functions later on.

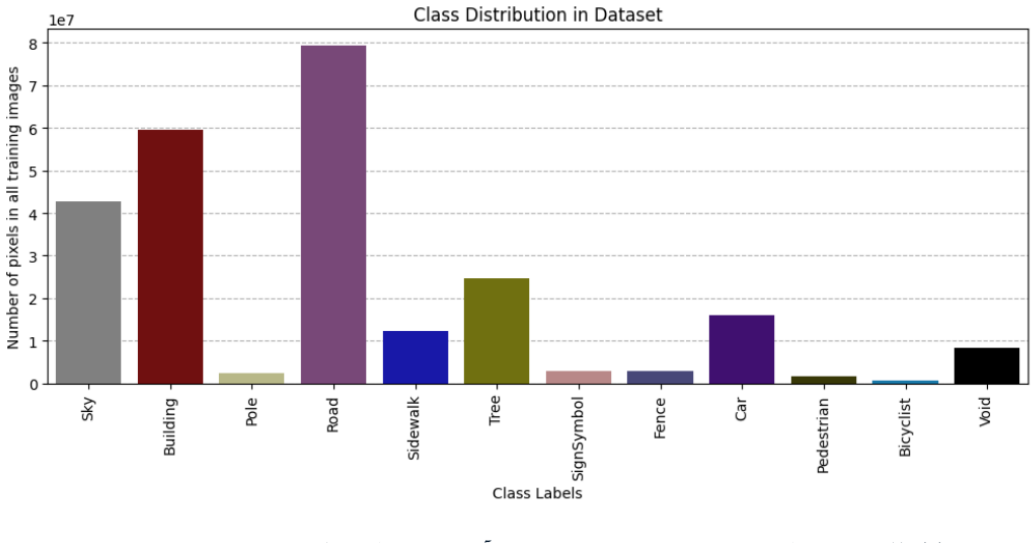

Figure 2: Distribution of pixels per class across the entire training set. The y-axis is on a logarithmic scale, highlighting the extreme imbalance between common classes (like 'Road') and rare classes (like 'Bicyclist').

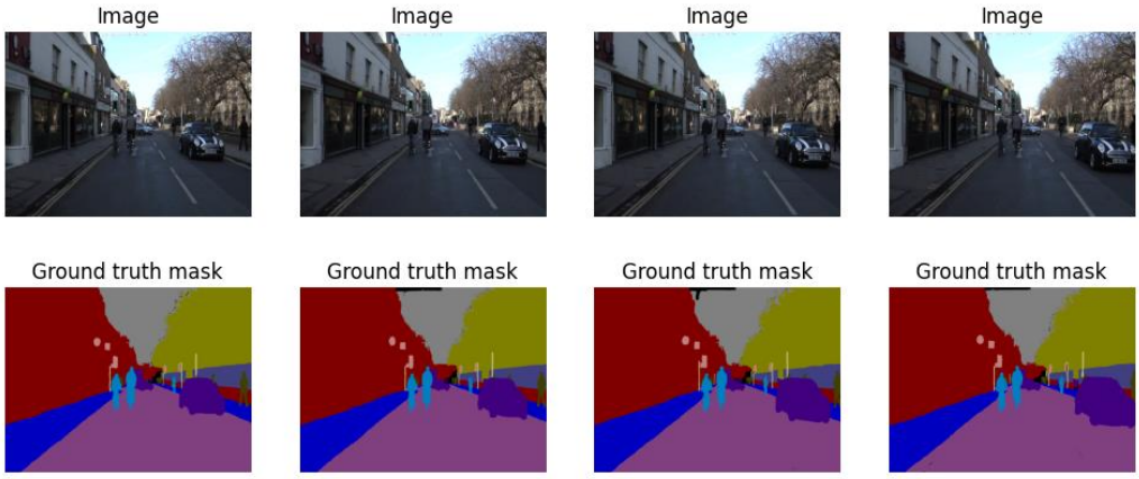

Figure 3: Four sample frames from the CamVid dataset. The top row shows the original input images, and the bottom row shows the corresponding ground truth segmentation masks, where each color represents a unique object class.

Given the high resolution of the images and the hardware constraints (limited GPU memory), we initially proceed without data augmentation. We will revisit augmentation later if the model's performance is not satisfactory.

1.3 Optimizers, Metrics, and Loss Function

1.3.1 Evaluation Metrics

To evaluate the performance of our segmentation model, especially in the context of class imbalance, standard accuracy is insufficient. We therefore implement and use two widely adopted metrics in semantic segmentation: the Dice Coefficient and Intersection over Union (IoU).

Dice Coefficient (F1 Score) The Dice Coefficient is a statistic used to gauge the similarity of two samples. For two sets A (ground truth pixels) and B (predicted pixels), it is defined as the size of the overlap doubled, divided by the sum of the sizes of the two sets.

$$\text{Dice Coefficient} = \frac{2 \times |A \cap B|}{|A| + |B|}$$

It is conceptually identical to the F1 score. A value of 1 indicates a perfect match, while 0 indicates no overlap. It is less sensitive to true negatives than accuracy and is therefore more robust to class imbalance.

```python
def dice_coefficient(y_true, y_pred, smooth=1e-6):
    num_classes = y_pred.shape[-1]
    y_pred = tf.argmax(y_pred,axis=-1)
    y_pred =  tf.one_hot(y_pred, depth=num_classes, axis=-1)
    y_true_one_hot = tf.one_hot(y_true, depth=num_classes, axis=-1)
    intersection = tf.reduce_sum(tf.multiply(y_true_one_hot,y_pred),axis=[1,2])
    sum_true = tf.reduce_sum(y_true_one_hot,axis=[1,2])
    sum_pred = tf.reduce_sum(y_pred,axis=[1,2])
    return tf.reduce_mean((2.0 * intersection + smooth) / (sum_true + sum_pred +
    smooth))
```

Figure 4: Python implementation of the Dice Coefficient metric using TensorFlow. The function calculates the Dice score for each class and then returns the mean score across all classes.

Intersection over Union (IoU / Jaccard Index) IoU is another crucial metric that measures the overlap between the predicted segmentation and the ground truth. It is defined as the size of the intersection divided by the size of the union of the two sets.

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|}$$

Like the Dice coefficient, IoU ranges from 0 to 1. It is a very strict metric; to get a high IoU score, the predicted segmentation must align almost perfectly with the ground truth. We calculate the mean IoU (mIoU) by averaging the IoU scores across all semantic classes.

```python
def iou_score(y_true, y_pred, smooth=1e-6):
    num_classes = y_pred.shape[-1]
    y_pred = tf.argmax(y_pred,axis=-1)
    y_pred =  tf.one_hot(y_pred, depth=num_classes, axis=-1)
    y_true_one_hot = tf.one_hot(y_true, depth=num_classes, axis=-1)
    intersection = tf.reduce_sum(tf.multiply(y_true_one_hot,y_pred),axis=[1,2])
    sum_true = tf.reduce_sum(y_true_one_hot,axis=[1,2])
    sum_pred = tf.reduce_sum(y_pred,axis=[1,2])
    union = sum_true + sum_pred - intersection
    return tf.reduce_mean((intersection + smooth) / (union + smooth))
```

Figure 5: Python implementation of the IoU Score metric using TensorFlow. Similar to the Dice implementation, it computes the per-class IoU and then averages them.

1.3.2 Loss Function: Categorical Cross-Entropy

The most common loss function for multi-class classification problems like semantic segmentation is **Categorical Cross-Entropy**. For each pixel, the model outputs a probability distribution over all possible classes. Cross-entropy measures the dissimilarity between this predicted distribution and the true distribution (which is a one-hot vector).

The formula is:

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(p_{ij})$$

where N is the number of pixels, C is the number of classes, y_{ij} is 1 if pixel i truly belongs to class j (and 0 otherwise), and p_{ij} is the model's predicted probability that pixel i belongs to class j. This loss function effectively pushes the model to assign a high probability to the correct class for every pixel.

1.3.3 Optimizer: Adam with a Polynomial Decay Schedule

The paper details a specific training regimen. We chose the **Adam** optimizer, which is an adaptive learning rate optimization algorithm that is well-suited for a wide range of problems. It converges faster than standard Stochastic Gradient Descent (SGD). Following the paper, we initially used a batch size of 16 (limited by GPU memory) and an L2 weight regularizer to prevent overfitting.

For the learning rate, we implemented a **polynomial decay schedule**. This means the learning rate starts at a higher value (e.g., 0.045) and gradually decreases over the course of training according to a polynomial function. This strategy is highly effective: a larger learning rate at the beginning allows the model to make rapid progress, while a smaller learning rate later on allows it to fine-tune its weights and settle into a good minimum.

1.4 Model Implementation Details

The Fast-SCNN model is constructed from several key, highly efficient building blocks. We implemented each of these modules from scratch.

1.4.1 Depthwise Separable Convolution

This is the cornerstone of modern efficient CNNs like MobileNet and, by extension, Fast-SCNN. A standard convolution applies a full 3D filter across all input channels at once, mixing spatial and cross-channel information. A depthwise separable convolution decouples this into two simpler, more efficient steps:

1. **Depthwise Convolution:** A single 2D spatial filter is applied to *each input channel independently*. This captures spatial patterns within each channel but does not combine information across channels.

2. **Pointwise Convolution:** A simple 1x1 convolution is used to linearly combine the outputs of the depthwise layer. This creates new features by mixing information across channels.

This factorization dramatically reduces the number of parameters and computations. For example, a single 3x3 convolution on an input with 81 channels requires 81 parameters, whereas a depthwise separable convolution requires only 36 parameters to achieve a similar receptive field.

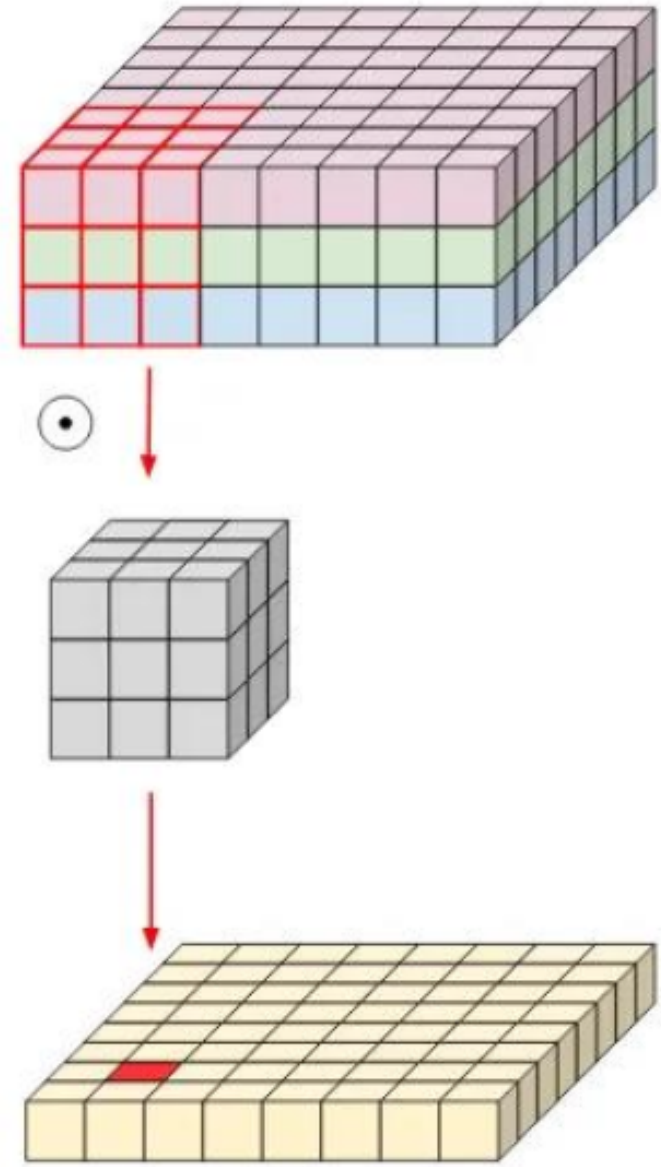

Figure 6: Illustration of a standard convolution operation. A single filter moves across the input volume, processing all channels simultaneously.

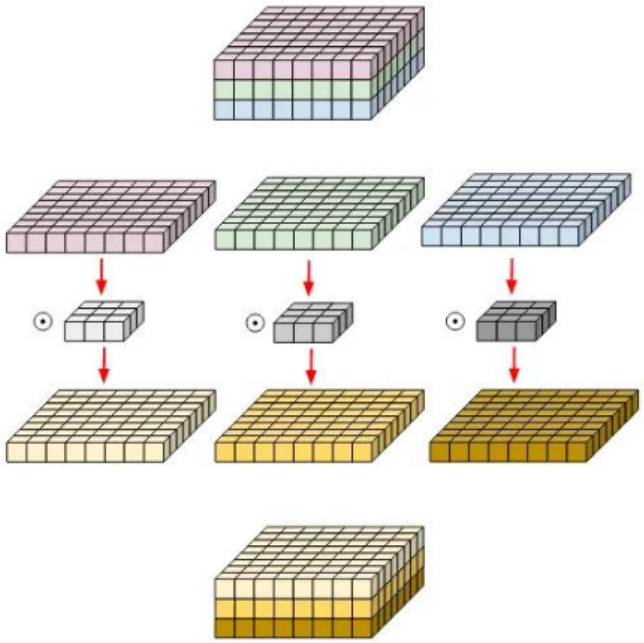

Figure 7: Illustration of the first step: Depthwise Convolution. A separate filter is applied to each input channel.

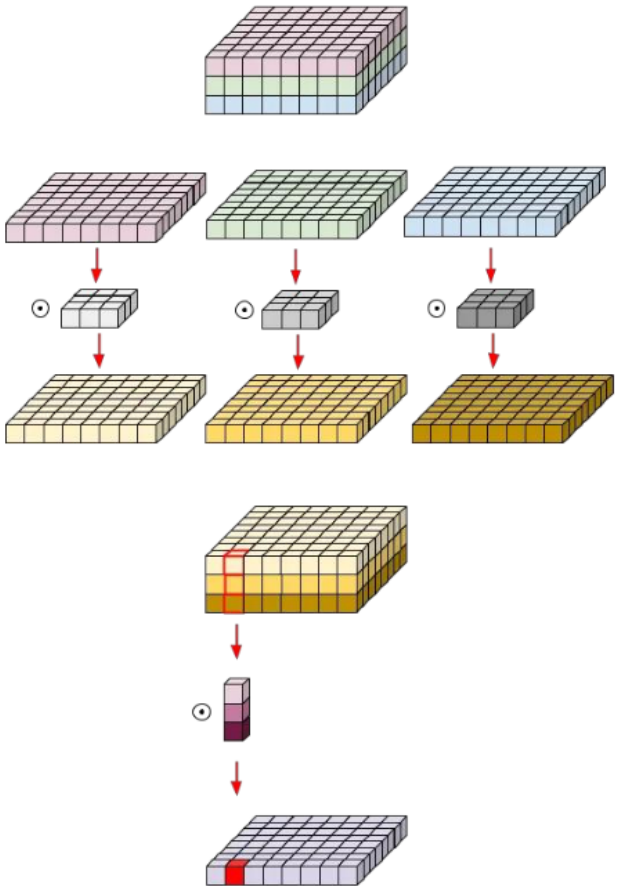

Figure 8: Illustration of the full Depthwise Separable Convolution, combining the depthwise and pointwise steps.

1.4.2 Inverted Residual Block

Residual blocks (from ResNet) use skip connections to allow gradients to flow more easily through deep networks, preventing the vanishing gradient problem. The **inverted** residual block, introduced in MobileNetV2, is a more memory-efficient variant. While a standard residual block has a "wide -¿ narrow -¿ wide" structure, the inverted block has a **"narrow -¿ wide -¿ narrow"** structure. The input features are first expanded to a higher-dimensional space using a 1x1 convolution, processed with a 3x3 depthwise convolution, and then projected back down to a low-dimensional representation with another 1x1 convolution. The skip connection adds the narrow input to the narrow output.

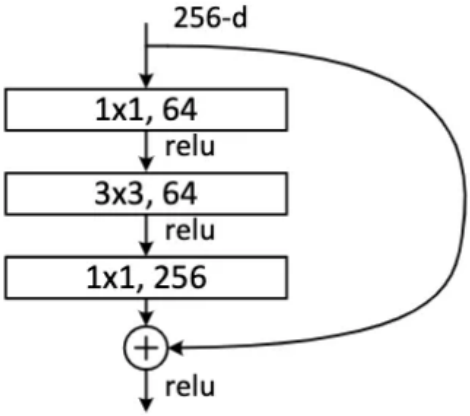

Figure 9: Diagram of a standard residual block.

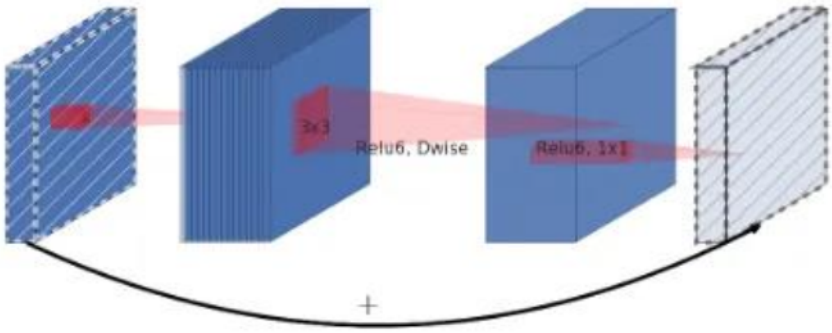

Figure 10: Diagram of the Inverted Residual Block used in Fast-SCNN. Note the narrow-wide-narrow structure.

```python
class Bottleneck(layers.Layer):
  def __init__(self, expansion_factor, in_channels, out_channels, strides,
  repeat):
    super(Bottleneck,self).__init__()
    self.blocks = []

    for i in range(repeat):
      if i == 0:
        self.blocks.append(self._make_block(expansion_factor, in_channels,
        out_channels, strides))
      else:
        self.blocks.append(self._make_block(expansion_factor, out_channels,
        out_channels, 1))

  def _make_block(self, expansion_factor, in_channels, out_channels, strides):
    model = models.Sequential([
      layers.Conv2D(expansion_factor * in_channels, kernel_size=1, strides=1,
                    padding='same', use_bias=False),
      layers.BatchNormalization(),
      layers.ReLU(),
      layers.DepthwiseConv2D(kernel_size=3, strides=strides, padding='same',
      use_bias=False),
      layers.BatchNormalization(),
      layers.ReLU(),
      layers.Conv2D(out_channels, kernel_size=1, strides=1, padding='same',
      use_bias=False),
      layers.BatchNormalization(),
    ])
    return model

  def call(self, x, training = False):
    out = x
    for block in self.blocks:
      residue = out
      out = block(out, training = training)
      if residue.shape == out.shape:
        out = layers.Add()([out,residue])
    return out
```

Figure 11: Code snippet showing the implementation of the Inverted Residual Block (termed 'Bottleneck' in the code).

Table 1: Structure of the Inverted Residual Block, as described in the reference paper.

Input	**Operator**	**Output**
$h \times w \times c$	Conv2D 1x1, f	$h \times w \times tc$
$h \times w \times tc$	DWConv 3x3, /s	$\frac{h}{s} \times \frac{w}{s} \times tc$
$\frac{h}{s} \times \frac{w}{s} \times tc$	Conv2D 1x1, -	$\frac{h}{s} \times \frac{w}{s} \times c'$

1.4.3 Pyramid Pooling Module (PPM)

To help the model understand objects and context at different scales, the Global Feature Extractor terminates with a Pyramid Pooling Module. This module performs average pooling on the feature map at several different scales (e.g., pooling the entire map into 1x1, 2x2, 3x3, and 6x6 grids). Each pooled feature map is then processed by a 1x1 convolution, upsampled back to the original feature map size using bilinear interpolation, and finally concatenated with the original input feature map. This provides the subsequent layers with a rich, multi-scale representation of the scene.

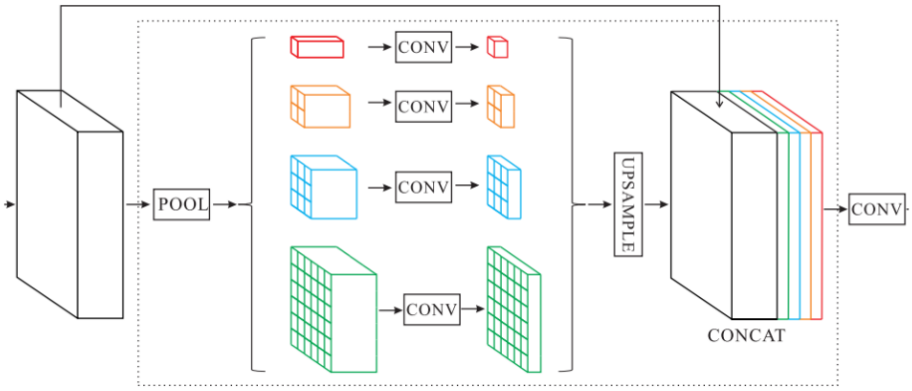

Figure 12: Conceptual diagram of the Pyramid Pooling Module (PPM).

```python
class PPM(layers.Layer):
  def __init__(self, input_shape, out_channels, pool_sizes=[1, 2, 3, 6]):
    super(PPM,self).__init__()
    self.pool_modules = []
    out_channels_per_pool = out_channels // len(pool_sizes)
    h, w = input_shape[:2]

    for pool_size in pool_sizes:
      self.pool_modules.append(
        models.Sequential([
              layers.AveragePooling2D(pool_size=(h // pool_size, w //
              pool_size),
              strides=(h // pool_size, w // pool_size),
              padding='valid'),
            layers.Conv2D(out_channels_per_pool, kernel_size=1, use_bias=False),
            layers.BatchNormalization(),
            layers.ReLU(),
        ])
      )
    self.conv = layers.Conv2D(out_channels, kernel_size=1, use_bias=False)
    self.bn = layers.BatchNormalization()
    self.relu = layers.ReLU()

  def call(self, x, training = False):
    h, w = x.shape[1:3]
    pools = [x]
    for pool_module in self.pool_modules:
      pooled = pool_module(x, training=training)
      features = tf.image.resize(pooled, (h, w), method='bilinear')
      pools.append(features)
    out = tf.concat(pools, axis=-1)
    out = self.conv(out)
    out = self.bn(out, training=training)
    out = self.relu(out)
    return out
```

Figure 13: Code snippet for the implementation of the Pyramid Pooling Module.

1.4.4 Feature Fusion Module (FFM)

This module is where the two branches of the network finally meet. It takes the high-resolution features from the shallow path and the upsampled low-resolution features from the deep path and merges them. The process involves convolutions on both streams to align their channel dimensions, followed by a simple element-wise addition to combine their information.

Table 2: Structure of the Feature Fusion Module.

Resolution Level	**Operation Sequence**
Higher Resolution	Conv2D 1x1, -
Lower Resolution (X times smaller)	Upsample ×X, DWConv (dilation X), Conv2D 1x1, -
Fusion	Element-wise Addition

1.4.5 Final Model Assembly

By combining these modules according to the architecture specified in the paper (Table 3), we construct the complete Fast-SCNN model. We verified our implementation by checking the output shapes at each layer against the paper's specifications. Our final model has approximately 1.15 million trainable parameters, which is very close to the 1.11 million reported in the paper, confirming the correctness of our implementation.

Table 3: Detailed architecture of our implemented Fast-SCNN model, matching the specification from the paper.

Input Shape	**Block**	**t**	**c**	**n**	**s**
1024 × 2048 × 3	Conv2D	-	32	1	2
512 × 1024 × 32	DSConv	-	48	1	2
256 × 512 × 48	DSConv	-	64	1	2
128 × 256 × 64	Bottleneck	6	64	3	2
64 × 128 × 64	Bottleneck	6	96	3	2
32 × 64 × 96	Bottleneck	6	128	3	1
32 × 64 × 128	PPM	-	128	-	-
32 × 64 × 128	FFM	-	128	-	-
128 × 256 × 128	DSConv	-	128	2	1
128 × 256 × 128	Conv2D	-	19	1	1

Table 4: Summary of parameters per module in our implementation.

Stage	**Module 1**	**Module 2**	**Module 3**	**Module 4**
Learning to Down-sample	Conv2D (864)	BN (128)	DSConv1 (2016)	DSConv2 (3760)
Global Feature Extractor	Block1 (167k)	Block2 (310k)	Block3 (560k)	PPM (50k)
Feature Fusion	FFM (27k)			
Classifier	DSConv1 (18k)	DSConv2 (18k)	DSConv3 (1.4k)	

1.5 Training and Evaluation Experiments

1.5.1 Baseline Training with Cross-Entropy Loss

We first trained the model for 100 epochs using the cross-entropy loss function. The training and validation curves are shown below. The model learns effectively, with the validation metrics steadily improving and converging after approximately 50 epochs. The final validation accuracy, Dice score, and IoU reached 0.9135, 0.6748, and 0.5883, respectively. These results are close to those reported in the paper, indicating a successful implementation and training process.

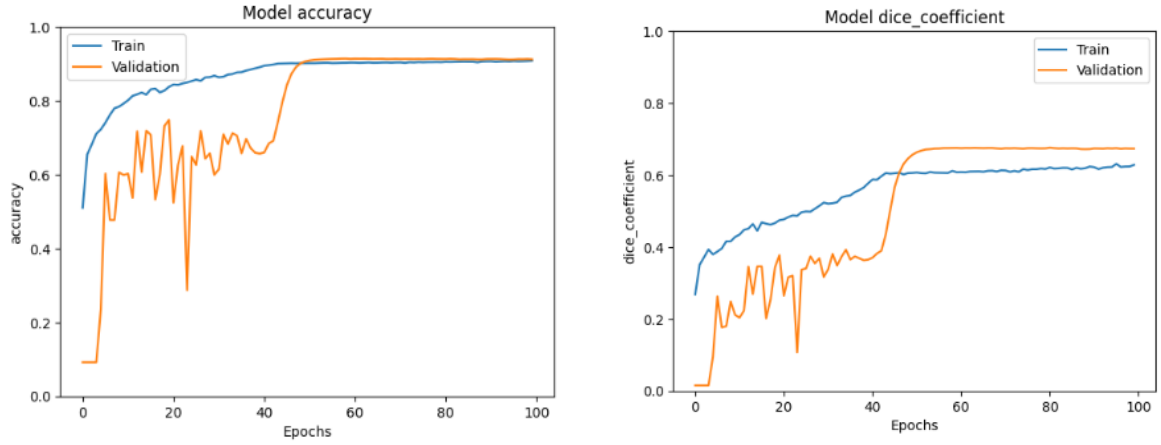

Figure 15: Model Accuracy (left) and Dice Coefficient (right) during training.

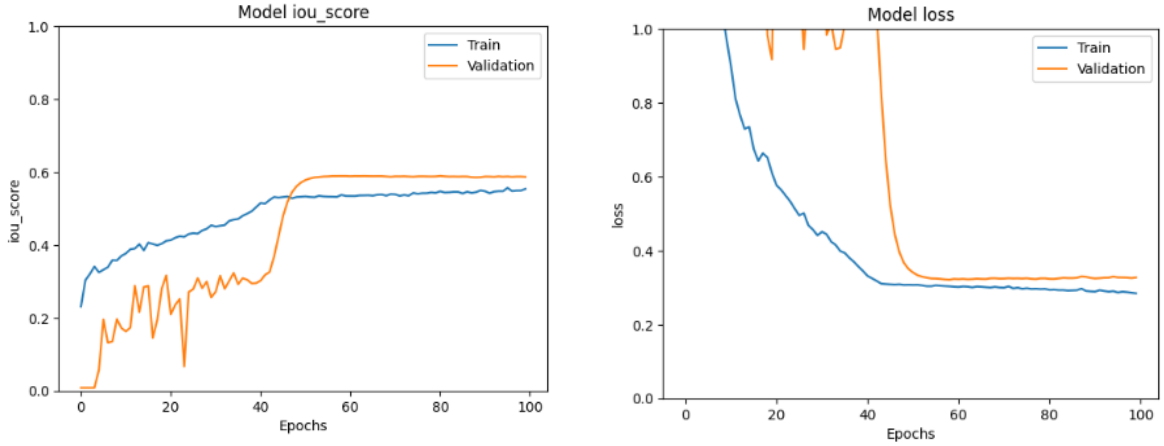

Figure 16: Model IoU Score (left) and Loss (right) during training.

Qualitative results (Figure 17) show that the model correctly segments large regions and captures the overall scene structure well. While there are minor inaccuracies at object boundaries, the performance is visually impressive, especially given the model's efficiency.

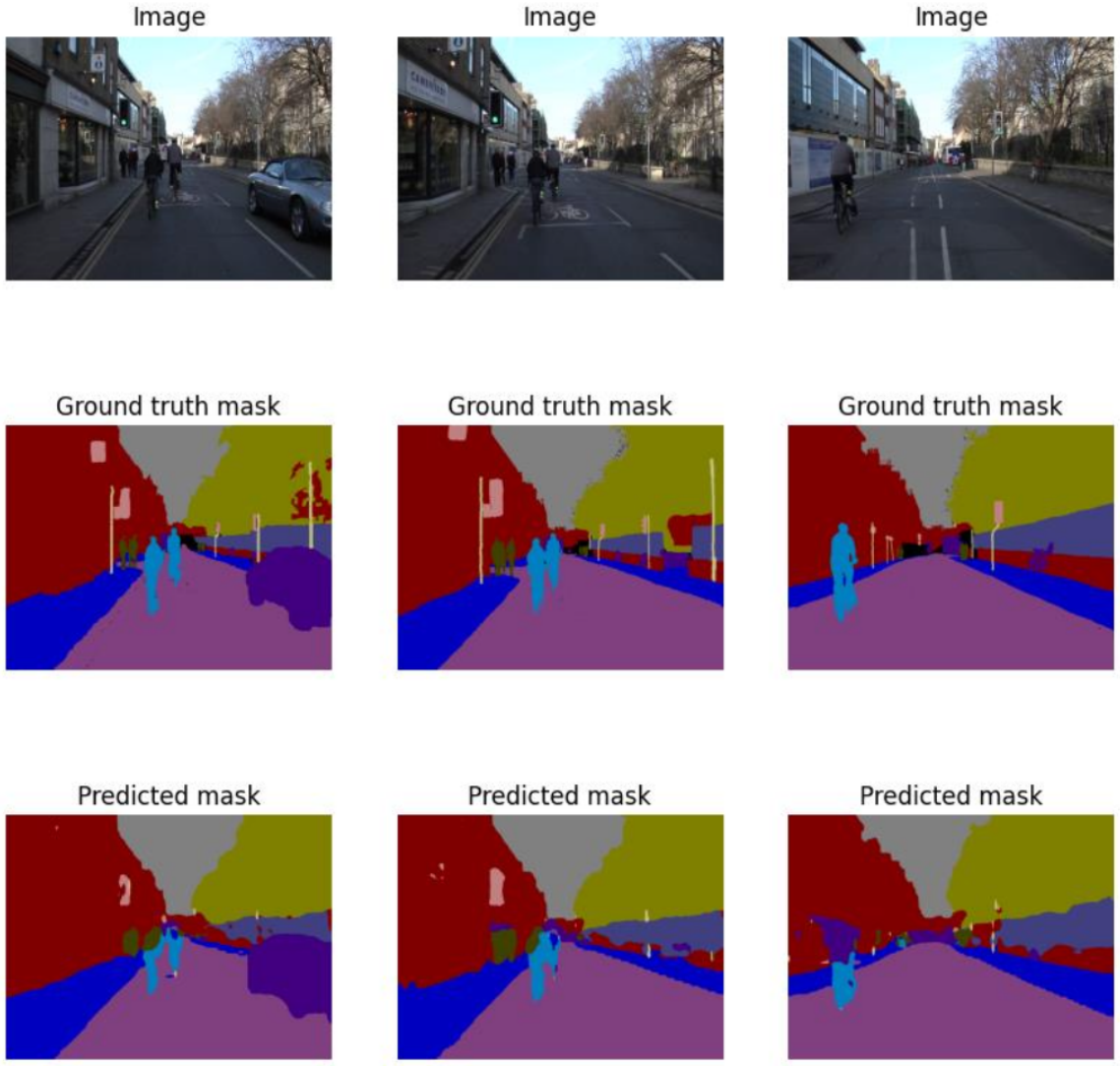

Figure 17: Sample prediction results from the baseline model. Top: original image, Middle: ground truth mask, Bottom: predicted mask. The model successfully identifies major classes like road, car, building, and sky.

1.5.2 Experiment 2: Training with IoU Loss

To see if we could improve performance, we experimented with different loss functions. Our first alternative was an IoU-based loss, defined as '1 - mIoU'. The rationale is to directly optimize the metric we care about most. However, the results show that while the model still converges, its performance is slightly worse than with cross-entropy. The final validation mIoU was 0.5725. This is a common finding: while metrics like IoU are excellent for evaluation, they can be noisy and non-smooth as loss functions, making optimization more difficult than the pixel-wise stability of cross-entropy.

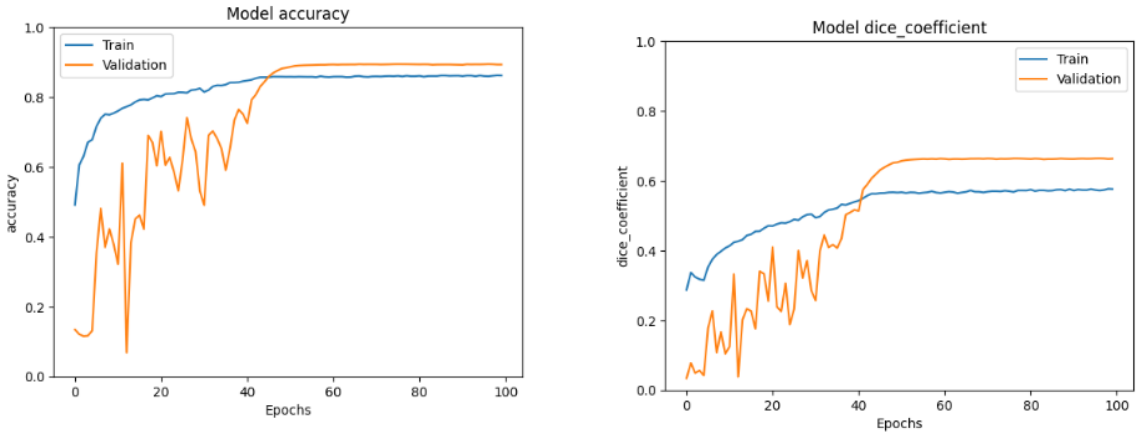

Figure 18: Accuracy and Dice curves when training with IoU Loss.

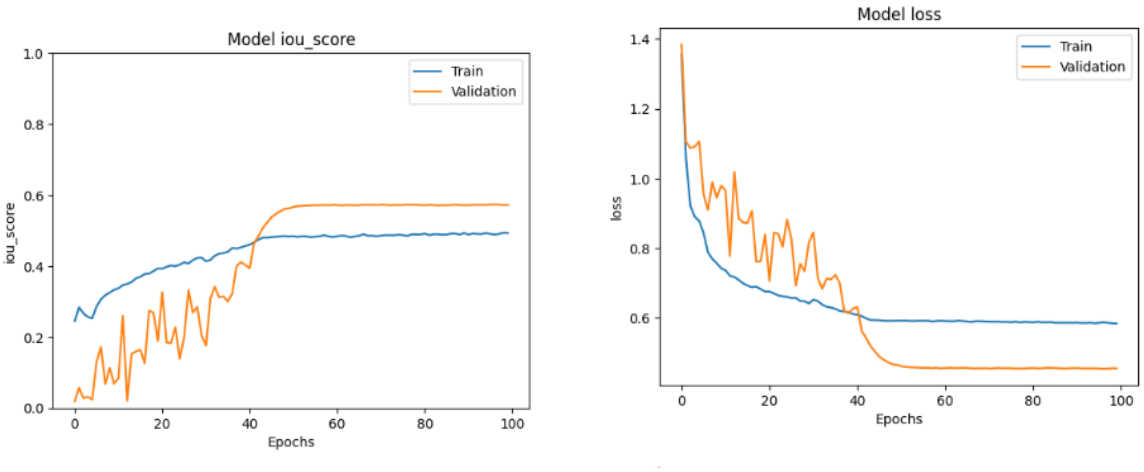

Figure 19: IoU and Loss curves when training with IoU Loss.

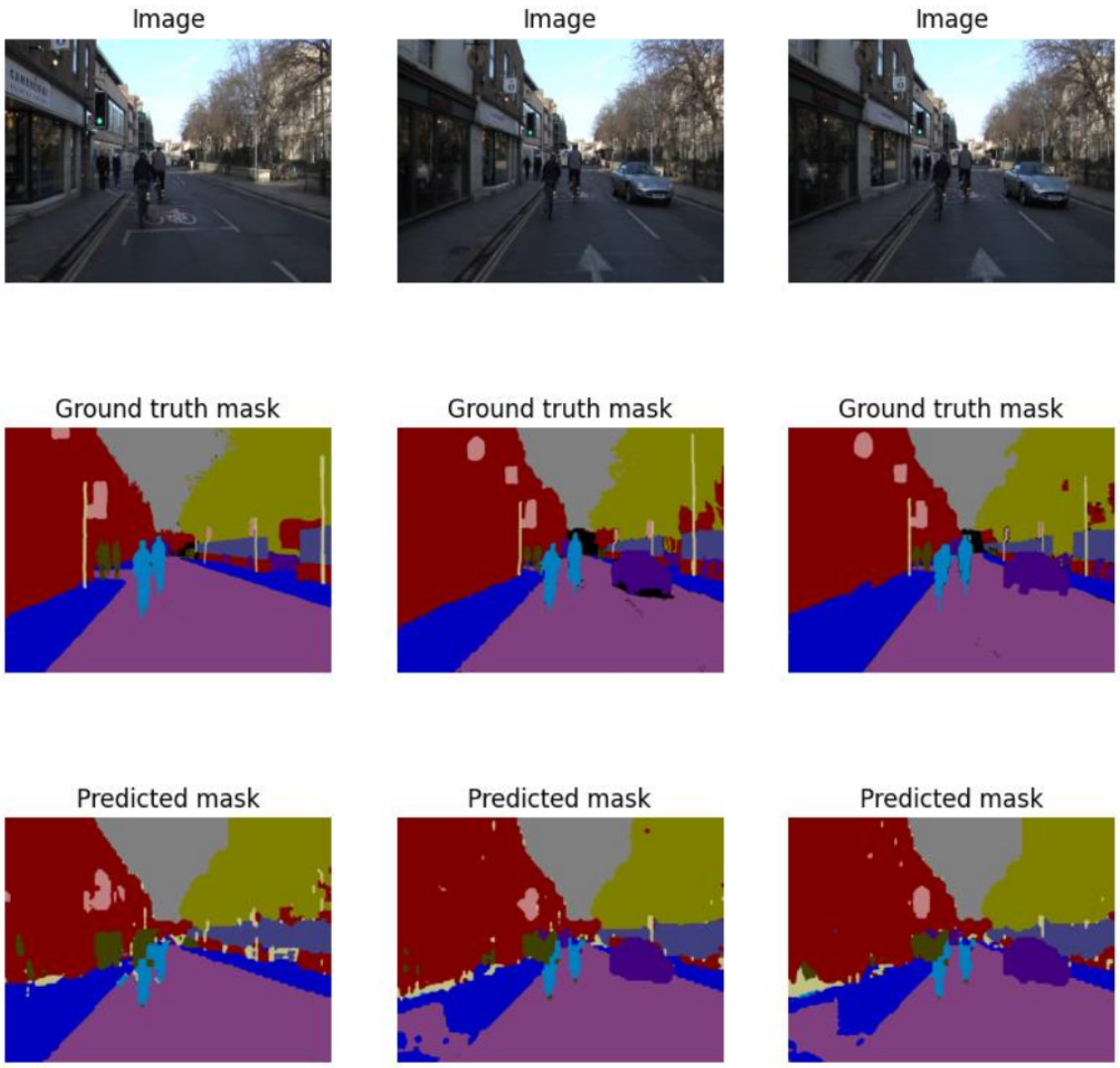

Figure 20: Qualitative results from the model trained with IoU loss. The segmentation quality is slightly degraded, with more noisy predictions around object edges.

1.5.3 Experiment 3: Training with Dice Loss

Next, we used a Dice-based loss, defined as '1 - DiceCoefficient'. Since the Dice metric is well-suited for imbalanced datasets, we hypothesized this might improve performance on rare classes. The results, however, were the weakest of the three. The validation Dice score peaked at 0.6384, and the mIoU at 0.5446. The training curves were also less stable. This further supports the conclusion that cross-entropy provides the most stable and effective optimization landscape for this task.

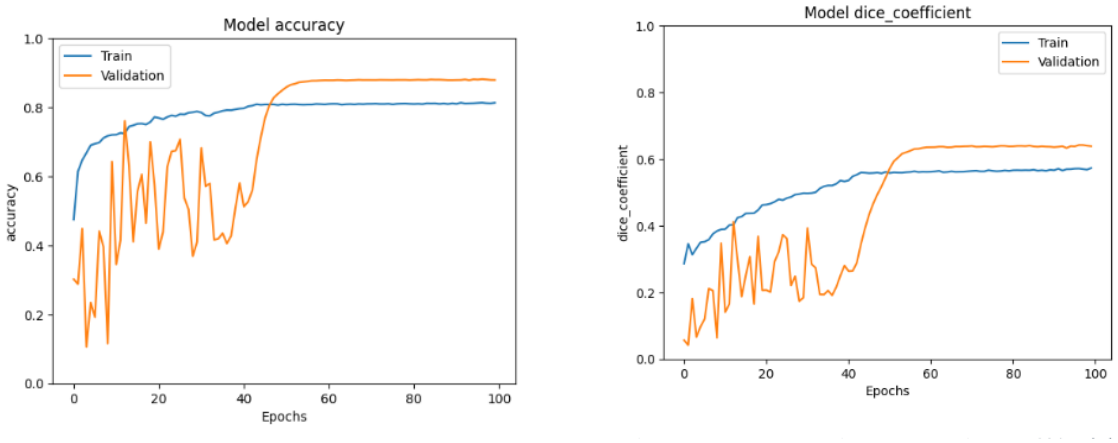

Figure 21: Accuracy and Dice curves when training with Dice Loss.

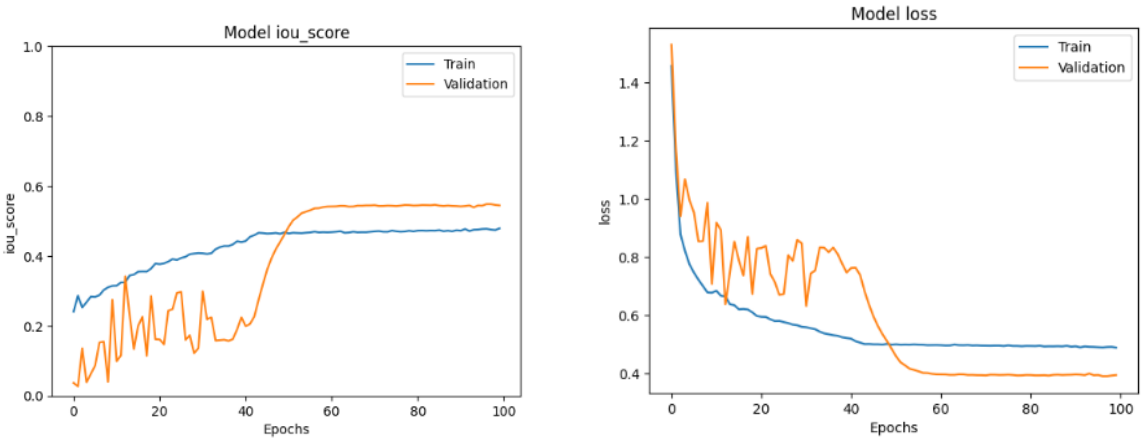

Figure 22: IoU and Loss curves when training with Dice Loss.

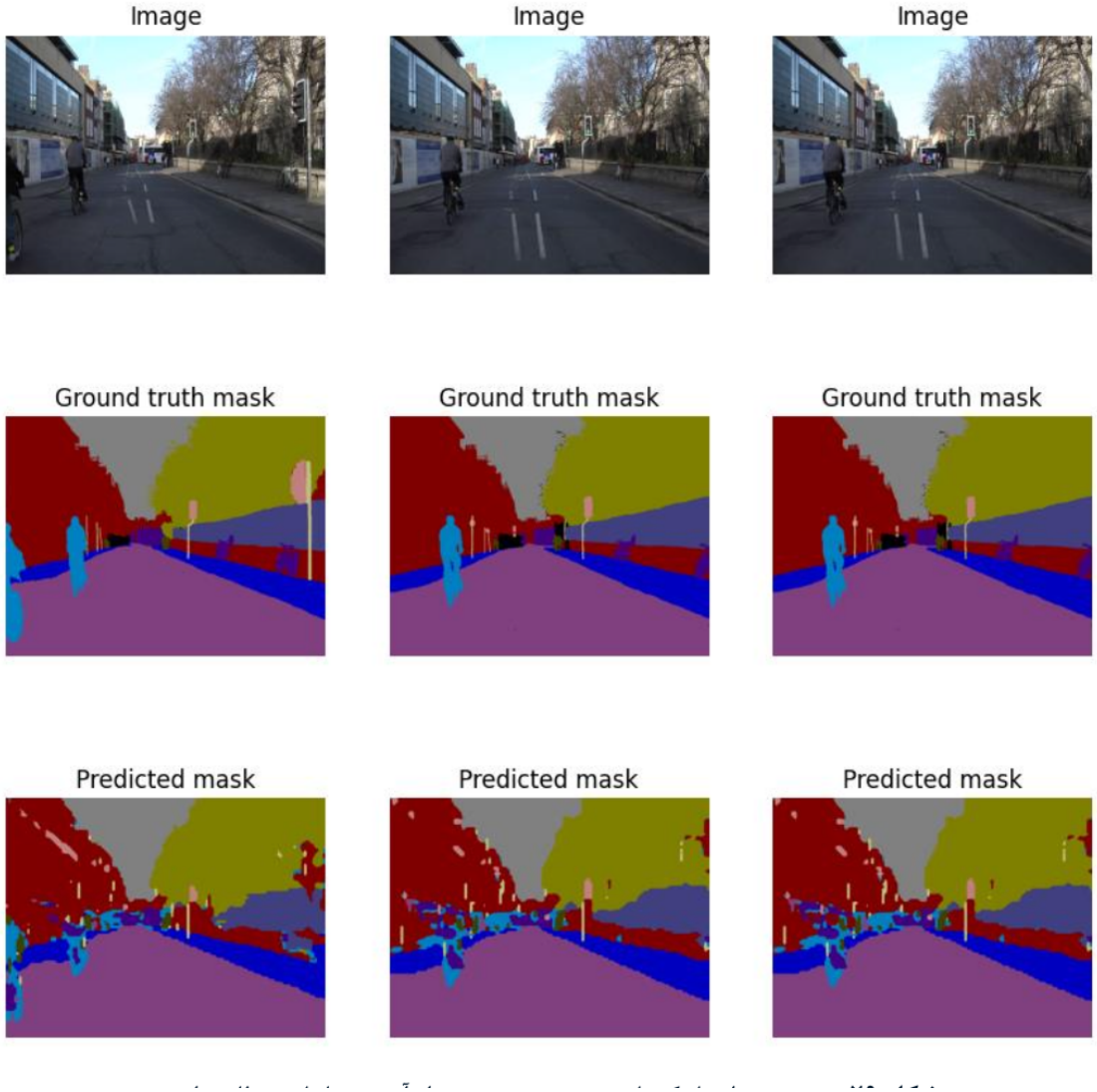

Figure 23: Qualitative results from the model trained with Dice loss. The segmentation is noticeably worse, with less coherent object shapes.

1.5.4 Experiment 4: Data Augmentation

Data augmentation is a standard technique to improve model generalization and robustness. We applied simple augmentations—random horizontal flipping, brightness adjustments, and adding a small amount of noise—to a subset of the training data. Due to GPU memory limitations, we could only augment 50 images, creating a new training set of 417 images. Surprisingly, the model trained on this augmented data performed worse than the baseline. This is likely because the augmentations were not diverse enough and the newly added images were too similar to the originals to provide significant new information, essentially acting as noisy duplicates.

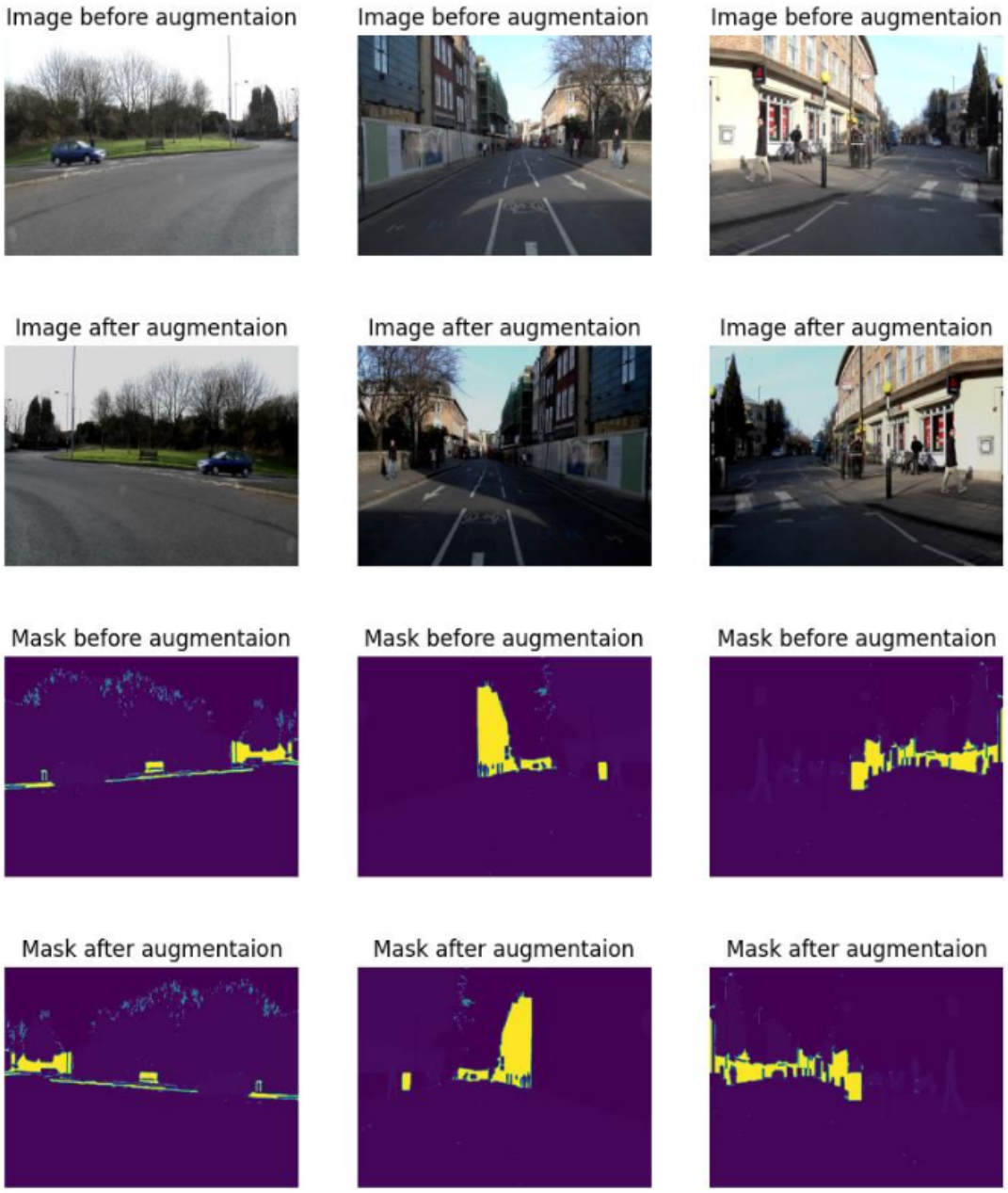

Figure 24: Examples of the data augmentation applied.

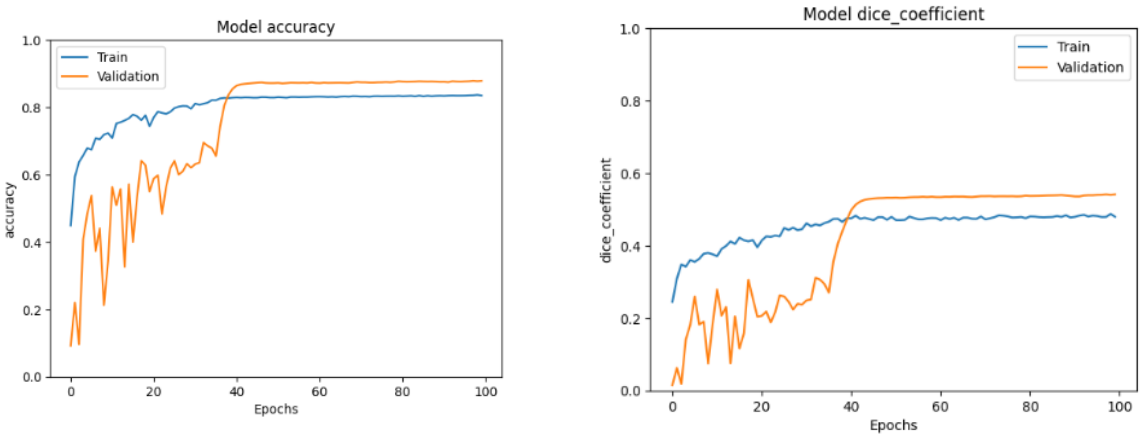

Figure 25: Accuracy and Dice curves when training with augmented data.

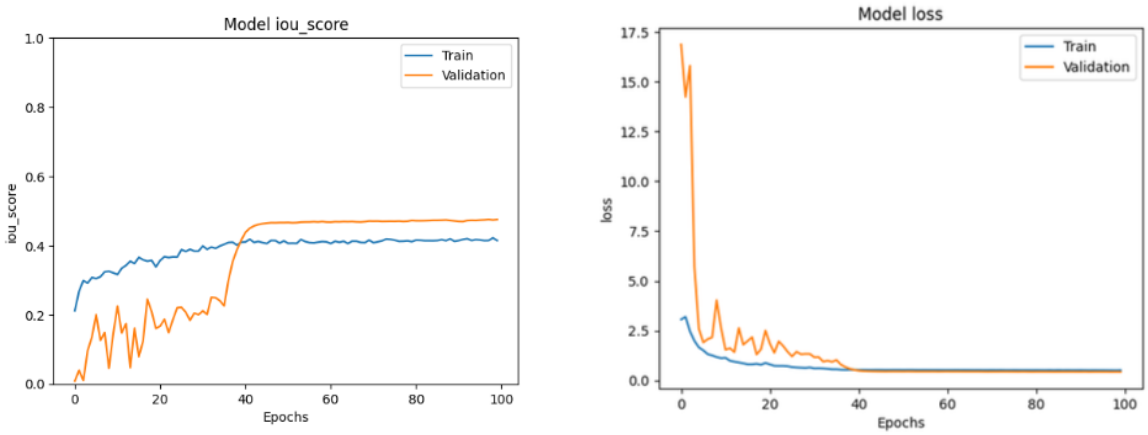

Figure 26: IoU and Loss curves when training with augmented data.

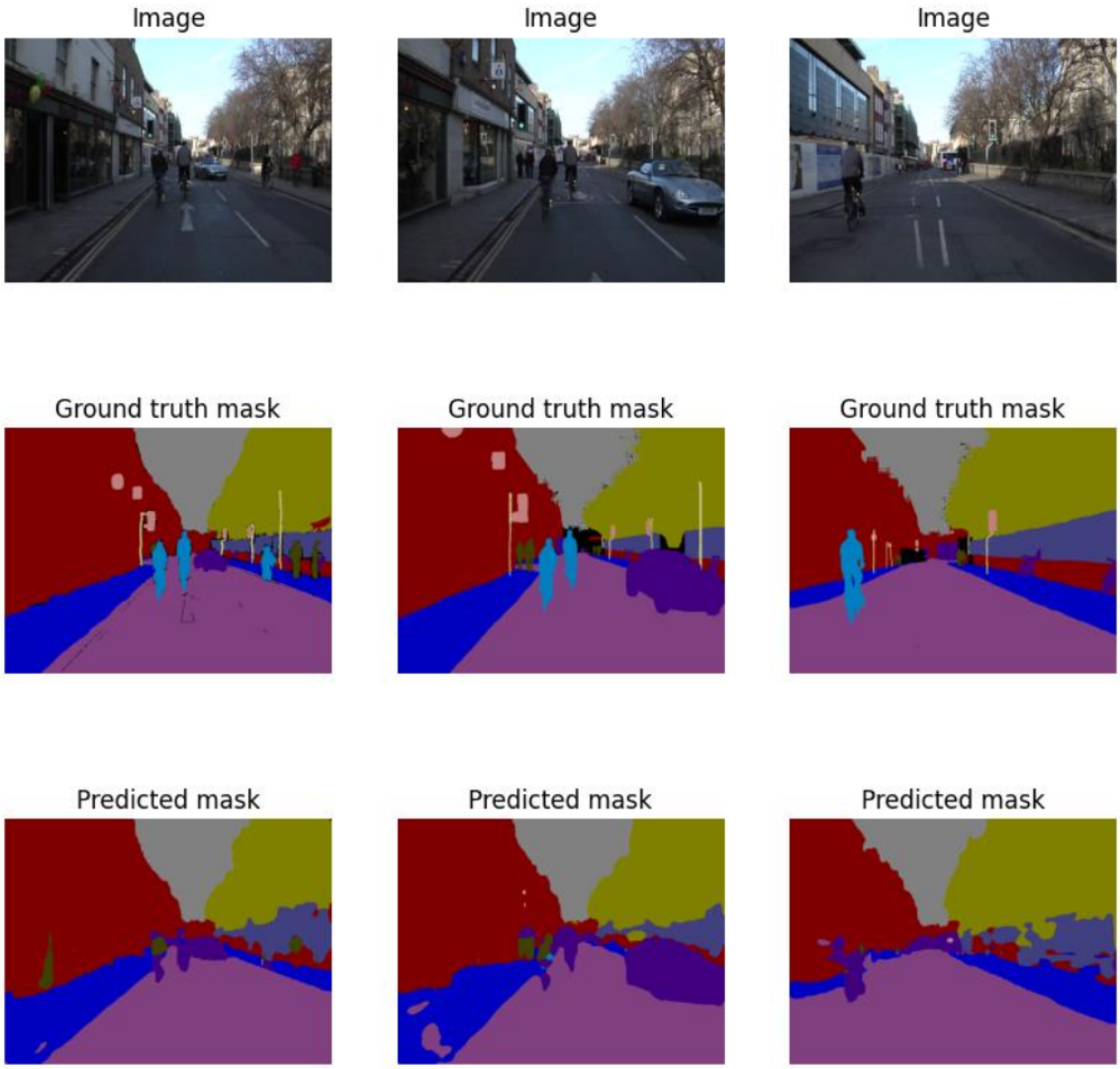

Figure 27: Qualitative results from the model trained with augmentation. Notably, smaller objects like people are not detected, likely due to the class imbalance being exacerbated by the simple augmentations.

Table 5: Final comparison of all models on the validation data. The baseline cross-entropy model without augmentation performed the best.

	Train				**Validation**			
Model Config	Acc	Dice	IoU	Loss	Acc	Dice	IoU	Loss
Cross-Entropy	**0.908**	**0.628**	**0.554**	**0.284**	**0.912**	**0.674**	**0.587**	**0.327**
IOU Loss	0.862	0.576	0.493	0.584	0.893	0.663	0.573	0.455
Dice Loss	0.813	0.573	0.479	0.489	0.879	0.638	0.545	0.394
Augmentation	0.835	0.480	0.415	0.511	0.879	0.541	0.475	0.430

1.6 Final Evaluation on the Test Set

Based on the results, we selected our best model—the one trained with cross-entropy loss and no augmentation—for final evaluation on the held-out test set. We retrained this model on the combined training and validation sets to leverage all available data.

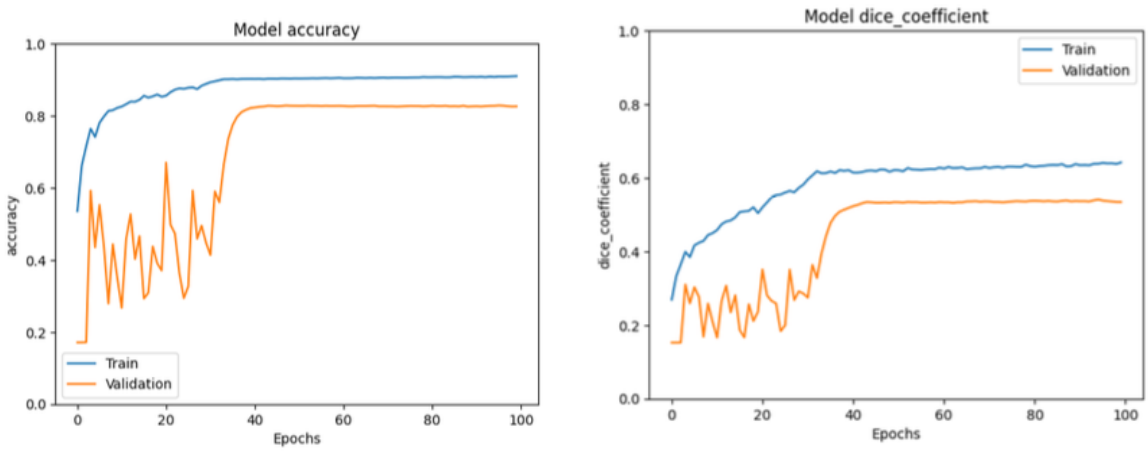

Figure 28: Final model training curves on the combined dataset: Accuracy (left) and Dice (right).

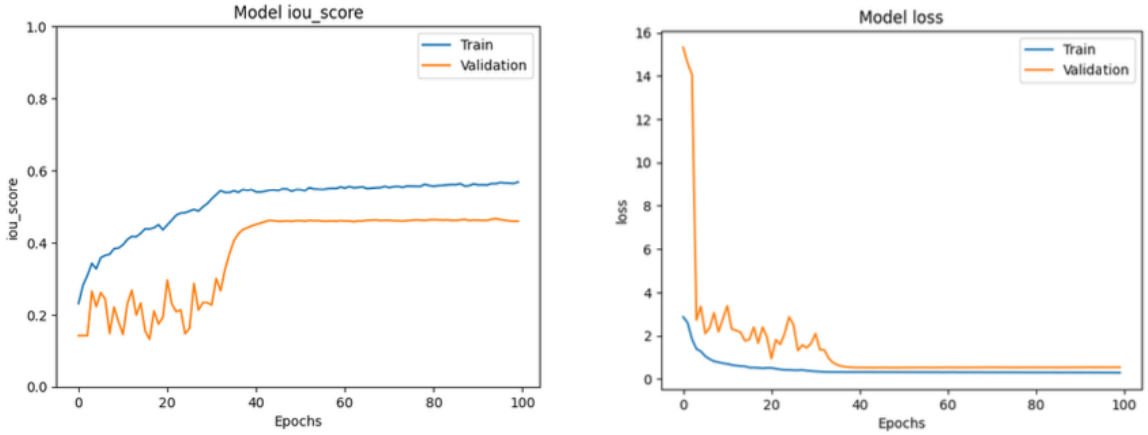

Figure 29: Final model training curves on the combined dataset: IoU (left) and Loss (right).

The final performance on the test set was slightly lower than on the validation set. This is a common occurrence and may suggest that the validation set is slightly "easier" or not perfectly representative of the full data distribution found in the test set.

Table 6: Final model performance on the Test set.

Dataset Split	Accuracy	Dice	IoU	Loss
Train (Combined)	0.910	0.642	0.568	0.285
Test	0.826	0.535	0.459	0.537

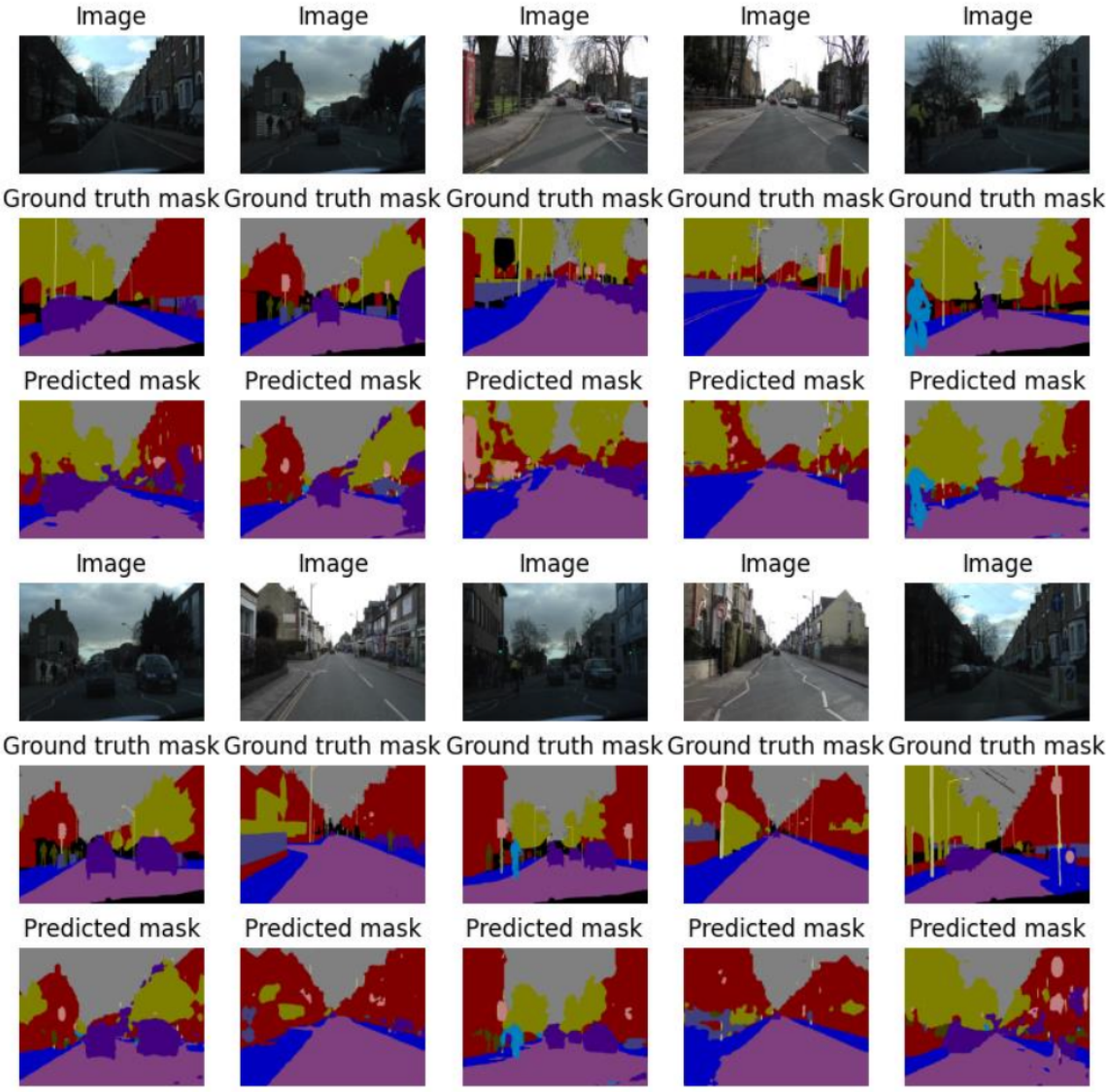

Figure 30: Sample predictions on the test set. The model continues to perform well on large structural classes but struggles with smaller, more detailed objects, which is consistent with its performance on the validation set.

1.7 Conclusion

In this project, we successfully implemented the Fast-SCNN architecture for real-time urban scene segmentation. Through this process, we gained a deep, practical understanding

of efficient CNN design principles, including depthwise separable convolutions, inverted residual blocks, and multi-scale feature fusion.

Our key takeaways are:

1. We successfully replicated the Fast-SCNN model, achieving performance metrics on the CamVid dataset that are comparable to those reported in the original paper.

2. Through experimentation, we confirmed that categorical cross-entropy is a more stable and effective loss function for this task compared to directly optimizing evaluation metrics like IoU or Dice.

3. We explored data augmentation, learning that its effectiveness is highly dependent on the diversity and quantity of the transformations, especially when dealing with high-resolution images and memory constraints.

4. The biggest challenge was managing the class imbalance inherent in the dataset. Metrics like Dice and IoU proved essential for meaningful evaluation, and the model's primary weakness remains the accurate segmentation of rare classes.

Overall, this project provided an end-to-end experience in tackling a complex computer vision problem, from understanding the theory and implementing the architecture from scratch to rigorously training, evaluating, and troubleshooting the model.