Computer Vision homework 4

Pan Changxun 2024011323

May 2025

1 Introduction

This report documents the implementation of the DeepLabV3+ model for semantic segmentation on the Cityscapes dataset. The goal is to achieve high accuracy in segmenting urban scenes into 19 different classes, such as road, car, pedestrian, etc. The report covers the dataset preparation, model architecture, training strategy, and evaluation metrics.

The codes for this project are available in the GitHub repository: `https://github.com/CXP-2024/CV-hw/tree/main/CV-hw4`. A detailed description can be obtained from the README file in the repository. Also, I achieved a 3-level augmentation for the dataset, which is also described in the AUGMENTATION.md file in the repository.

2 Models and Experiments

I first tried to use the UNet model, but it did not perform well for this segmentation task. I then implemented the DeepLabV3 model and further improved it to DeepLabV3+. The DeepLabV3+ architecture consists of the following key components:

- A backbone network using ResNet with atrous convolutions
- Atrous Spatial Pyramid Pooling (ASPP) to capture multi-scale context
- A decoder module that refines the segmentation results
- Skip connections that combine low-level features with high-level features

This architecture is particularly effective for semantic segmentation tasks as it captures both detailed spatial information and broad contextual information.

2.1 Model Architecture Details

The DeepLabV3+ model I implemented consists of several carefully designed components:

2.1.1 ResNet Backbone

I implemented a custom ResNet backbone with atrous (dilated) convolutions:

- Initial layer: 7×7 convolution with stride 2, followed by batch normalization, ReLU, and max pooling
- Layer 1: 3 ResNet blocks with 64 channels
- Layer 2: 4 ResNet blocks with 128 channels, stride 2
- Layer 3: 6 ResNet blocks with 256 channels, dilation rate 2
- Layer 4: 3 ResNet blocks with 512 channels, dilation rate 4

Each ResNet block consists of two 3×3 convolutional layers with batch normalization and ReLU, along with a residual connection. The increasing dilation rates in deeper layers ensure a larger receptive field without decreasing spatial resolution.

2.1.2 ASPP Module

The Atrous Spatial Pyramid Pooling module consists of:

- One 1×1 convolution
- Three 3×3 atrous convolutions with dilation rates of 12, 24, and 36
- A global average pooling branch followed by a 1×1 convolution

These five branches are concatenated and fed through a 1×1 convolution with 256 output channels, followed by batch normalization, ReLU, and a dropout layer (rate=0.5) to obtain the final ASPP features.

2.1.3 Decoder Module

The decoder integrates the semantically rich features from the ASPP module with spatially detailed low-level features from earlier layers:

- Low-level features from the first ResNet layer are processed by a 1×1 convolution to reduce channels to 48
- ASPP features are upsampled to match the spatial dimensions of the low-level features
- Both feature maps are concatenated and processed by two 3×3 convolutions
- The result is then upsampled to input resolution and fed to a final classifier

2.1.4 Final Classification Layer

A simple 1×1 convolution transforms the decoder output into class logits with 19 channels (one for each class in the Cityscapes dataset).

2.1.5 Weight Initialization

All convolutional layers use Kaiming initialization to ensure proper gradient flow during training, while batch normalization layers are initialized with weight=1 and bias=0.

2.2 Model Implementation Details

2.2.1 Forward Pass Process

The forward pass through the model is implemented as follows:

1. Input image is processed by the ResNet backbone
2. Encoder produces high-level features (output of layer 4) and low-level features (output of layer 1)
3. High-level features are passed through the ASPP module for multi-scale context extraction
4. ASPP features and low-level features are combined in the decoder module
5. Final classifier converts the feature map to logits
6. Output is upsampled to input resolution using bilinear interpolation

2.2.2 Key Implementation Components

Atrous (Dilated) Convolutions I used dilated convolutions to increase the receptive field without increasing the number of parameters or reducing the spatial resolution. In standard convolutions, the filter elements are applied to adjacent input elements. In dilated convolutions, gaps ("holes") are introduced between filter elements, effectively increasing the receptive field while maintaining the same number of parameters.

For example, with a dilation rate of 2, each filter element applies to inputs that are 2 pixels apart, effectively doubling the receptive field without increasing the filter size. This is crucial for semantic segmentation where both global context and fine details need to be preserved.

ASPP Design Considerations The ASPP module captures multi-scale information through parallel atrous convolutions with different dilation rates (12, 24, and 36). These different rates allow the network to capture context at various scales:

- Smaller dilation rates: capture fine details and local context
- Larger dilation rates: capture broader context and object relationships
- Global average pooling branch: captures image-level context

Skip Connection Implementation The skip connection between the encoder and decoder is crucial for recovering spatial details lost during downsampling. By connecting the low-level features from the first ResNet block to the decoder, the model can combine semantic information (from deep layers) with spatial information (from shallow layers), resulting in more accurate boundary delineation.

The low-level features undergo a 1×1 convolution to reduce the channel dimension from 64 to 48, making the fusion more balanced and computationally efficient.

2.3 Model Input and Evaluation Resolution

To balance performance and computational efficiency, all training and evaluation were conducted at a consistent resolution of 512×1024 pixels. The original Cityscapes images (1024×2048) were resized to this working resolution during both training and testing phases.

This resolution choice was explicitly defined in the `config.yaml` file:

```
data:
  image_size: [512, 1024]  # height, width
```

Both input images and ground truth labels were resized to this resolution. During inference, the model predictions remain at 512×1024 resolution rather than being upscaled back to the original 1024×2048 resolution. This approach ensures consistency between training and evaluation conditions, while significantly reducing memory requirements and computational load.

The model architecture was designed to handle this specific resolution, with the encoder progressively reducing spatial dimensions and the decoder carefully upsampling features back to the input dimensions. The final interpolation layer in the DeepLabV3+ model upsamples the features to match the input resolution of 512×1024:

```
# Upscale to input resolution
x = F.interpolate(x, size=input_shape, mode='bilinear', align_corners=False)
```

This consistent handling of resolution across both training and evaluation phases ensures fair comparisons and reliable performance metrics.

2.4 Loss Function Design

For loss calculation, I implemented a comprehensive loss function strategy to address the challenges of semantic segmentation. Initially, I used standard cross-entropy loss during the first training phase. For later training stages, I developed a more sophisticated combined loss approach found in `utils/losses.py`.

My loss function implementation consists of three main components:

2.4.1 Label Smoothing Cross Entropy Loss

To reduce overfitting and improve model generalization, I implemented label smoothing for the cross-entropy loss. Rather than using hard targets (1 for correct class, 0 for others), label smoothing assigns a value slightly less than 1 to the correct class and small non-zero values to other classes:

- For the correct class: target $= 1 - \text{smoothing}$
- For other classes: target = smoothing/num_classes

This technique helps prevent the model from becoming overconfident and improves its ability to generalize. I used a smoothing factor of 0.1, which provided a good balance between confidence and generalization.

2.4.2 Dice Loss

To directly optimize for the intersection over union metric, I implemented Dice loss, which is particularly effective for segmentation tasks with class imbalance. The Dice coefficient measures the overlap between predicted and ground truth segmentations:

$$\text{Dice} = \frac{2 \times \text{intersection} + \text{smooth}}{\text{union} + \text{smooth}} \tag{1}$$

Where:

- intersection = sum of element-wise multiplication of predictions and targets
- union = sum of predictions + sum of targets
- smooth = a small constant (1.0) to prevent division by zero

The Dice loss is calculated as $1 - \text{Dice}$, so minimizing this loss maximizes the Dice coefficient.

2.4.3 Combined Loss Function

To leverage the benefits of both approaches, I created a combined loss function that is a weighted sum of label smoothing cross-entropy and Dice loss:

$$\text{Loss} = \alpha \times \text{CrossEntropyLoss} + (1 - \alpha) \times \text{DiceLoss} \tag{2}$$

Where $\alpha = 0.6$ gives equal weight to both loss components.
This combined approach addresses multiple challenges:

- Cross-entropy provides stable gradients and good convergence properties
- Label smoothing prevents overconfidence and improves generalization
- Dice loss directly optimizes for overlap metrics (similar to IoU)
- The combination helps balance between pixel-wise accuracy and region-based segmentation quality

Additionally, all loss functions properly handle ignored pixels (labeled with 255) and support class weights to address class imbalance issues.

2.5 Model Aggregation

After training all folds, I evaluated three model aggregation approaches:

1. The best mIoU model from all folds
2. The average of parameters from the best models across all folds
3. The average of parameters from the top 3 best mIoU models across all folds

The best performing model from these three approaches was then used as the pre-trained model for the next 20-epoch training iteration. Since the initial training was for 10 epochs, subsequent training stages began at epochs 10, 30, 50, etc.

The results for the first cross-validation stage (epochs 10-30) are shown in Figures 2 and 3. Figure 2 shows the training curves and IoU progression for fold 1, while Figure 3 compares mIoU across all folds on the official validation set.

2.6 Training Results for Epochs 30-50

During the second training phase (epochs 30-50), I observed a slight decrease in overall performance. However, the averaged model still maintained a respectable mIoU of 54%, indicating the robustness of the model averaging approach. The training performance for this stage is illustrated in Figures 4 and 6. Figure 4 presents the training curves and IoU progression for fold 1, while Figure 6 shows the mIoU comparison across all folds.

2.7 Training Results for Epochs 50-70

Encouragingly, after I apply data augmentation techniques, performance improved during the third training phase (epochs 50-70), with the best model achieving an mIoU of 56%. This improvement suggests that the learning rate adjustment strategy and continued training were effective in refining the model parameters. The results of this training phase are presented in Figures 7 and 8, showing the detailed performance metrics across different folds.

2.8 Final Optimization and Results

For final optimization (70-90 epochs), I implemented a cosine learning rate scheduler and fine-tuned the combined loss function with adjusted weights. The combined loss with equal weighting between label smoothed cross-entropy and Dice loss (0.5:0.5) proved most effective during this phase. With this approach, the best model achieved an mIoU of approximately 58%, although there was considerable variance across folds. Given the performance plateau and the significant variance, I decided to conclude the training at this point. (I also tried to use the warmup learning rate strategy, but it did not yield better results. I find when the learning rate is reset after the period, the optimization process is not as effective as the cosine learning rate strategy.(not stable))

The final test results are presented in Figures 11, 12, and 13, with detailed visualizations available in the directory `outputs/deeplabv3plus_test_results`. Figure 11 shows the per-class IoU scores, Figure 12 presents the confusion matrix for segmentation results, and Figure 13 illustrates the relationship between class weights and IoU scores. Figure 10 compares the performance of the three model aggregation approaches.

3 Acknowledgements

I would like to acknowledge the authors of the DeepLabV3+ architecture, whose work provided the foundation for this implementation. I also utilized GitHub Copilot as a programming assistant to help streamline code development and debugging.

4 Figures

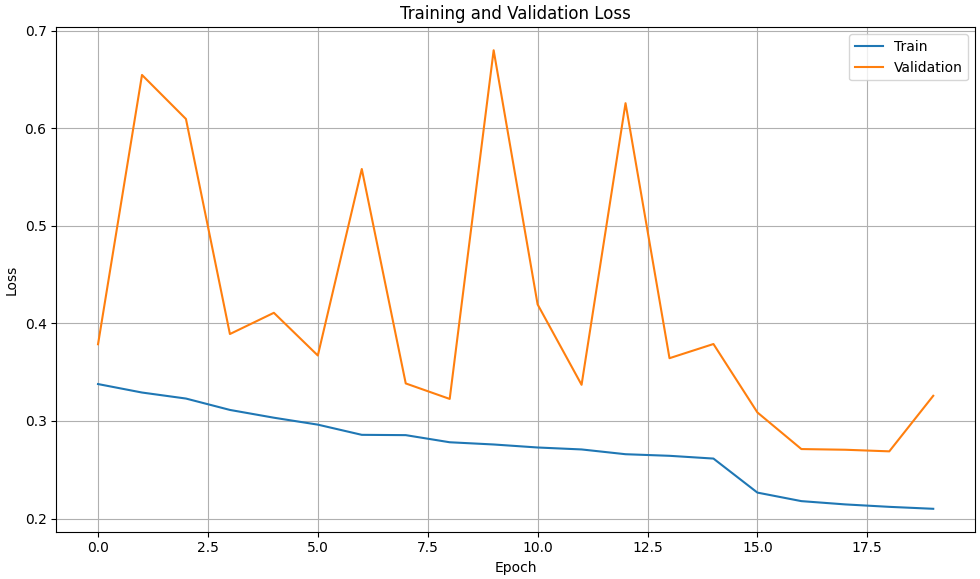
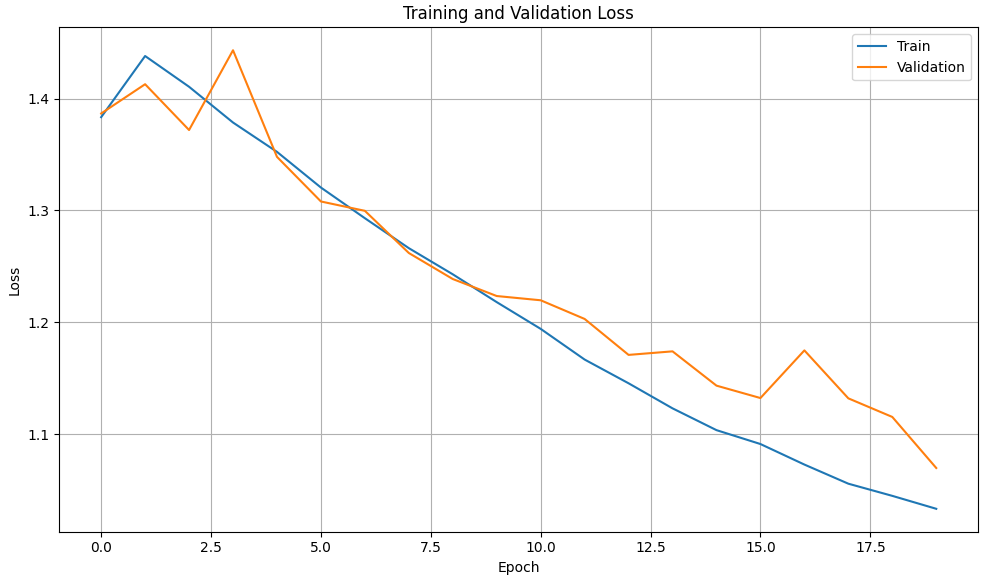
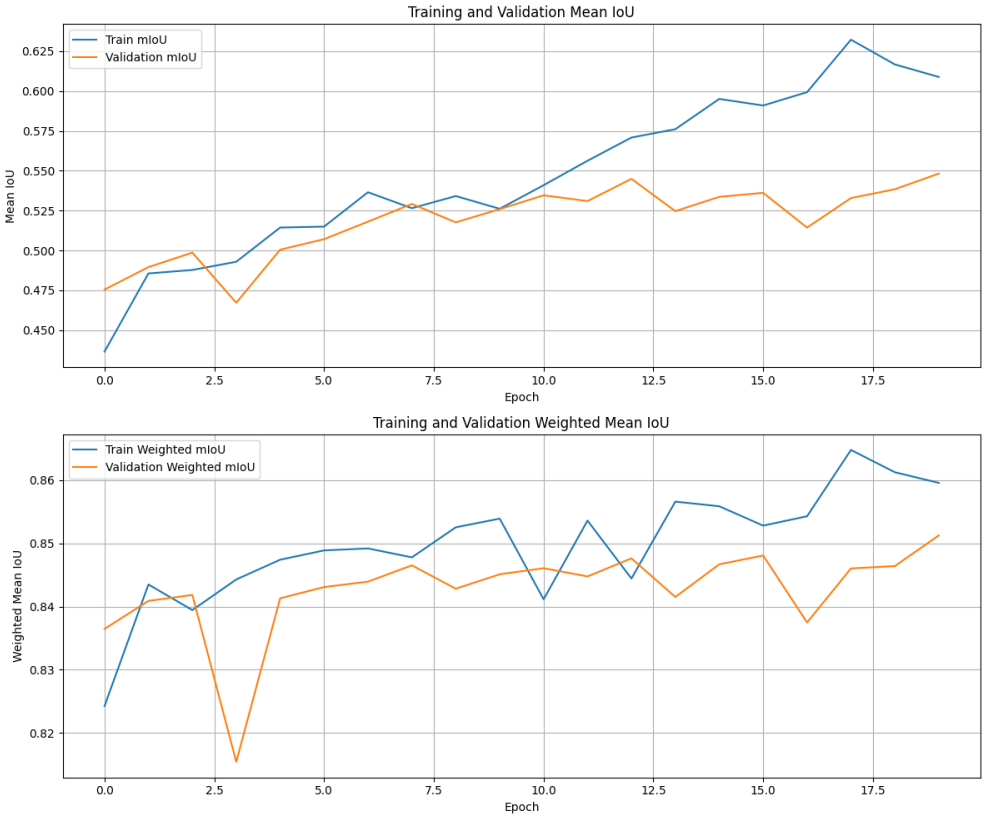


Figure 1: Training curve for epochs 10 to 30 without cross-validation, showing signs of overfitting



(a) Training curves for fold 1, epochs 10-30



(b) IoU progression history for fold 1, epochs 10-30

Figure 2: Training performance for the first cross-validation stage (epochs 10-30)

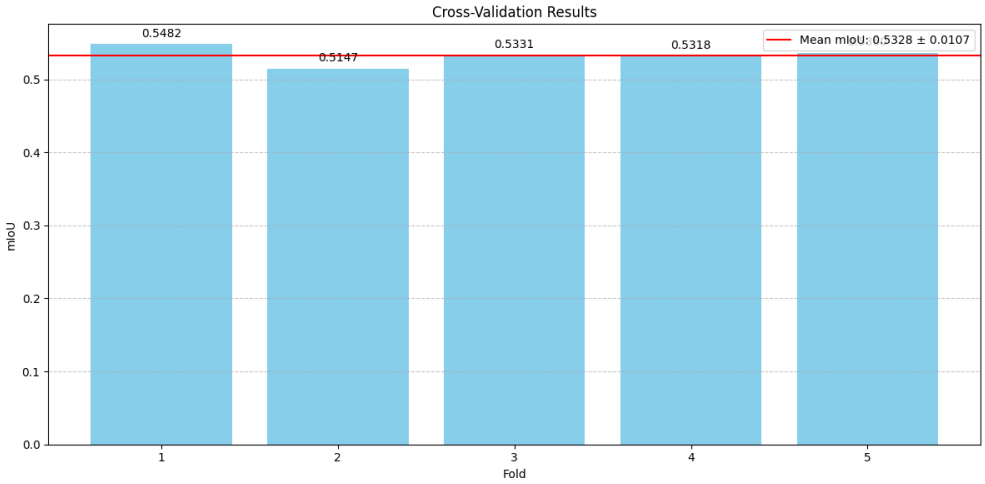
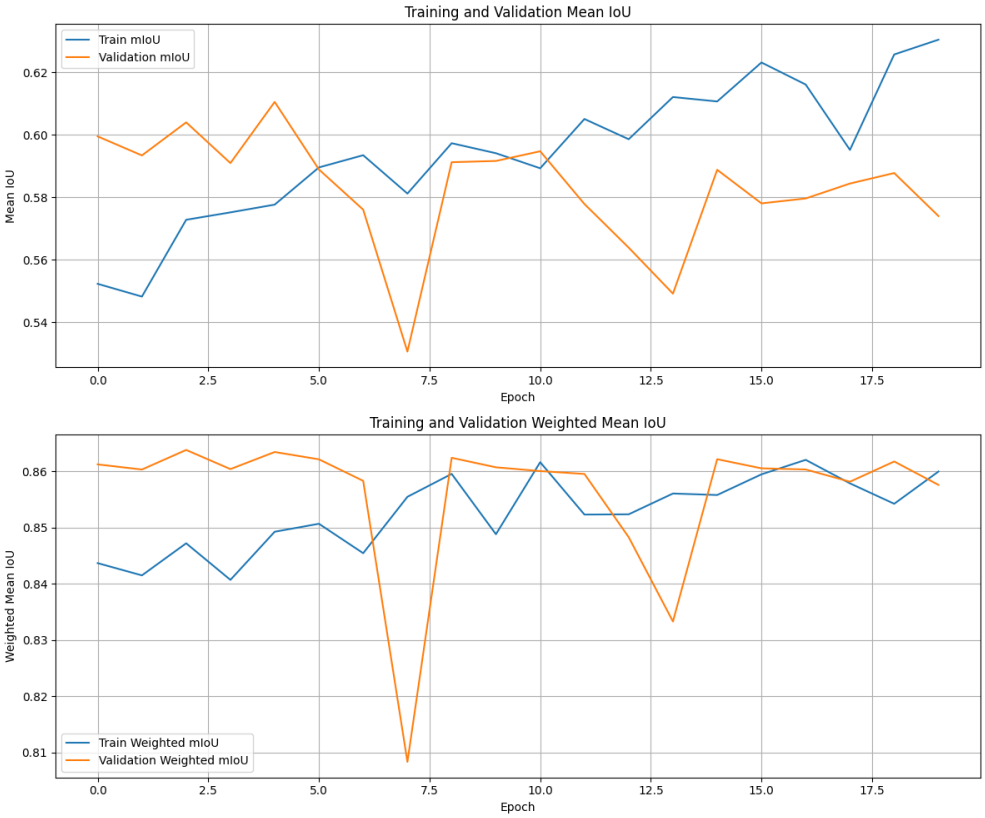


Figure 3: mIoU comparison across all folds on the official validation set, epochs 10-30

(a) Training curves for fold 1, epochs 30-50



(b) IoU progression history for fold 1, epochs 30-50

Figure 4: Training performance for the second cross-validation stage (epochs 30-50)

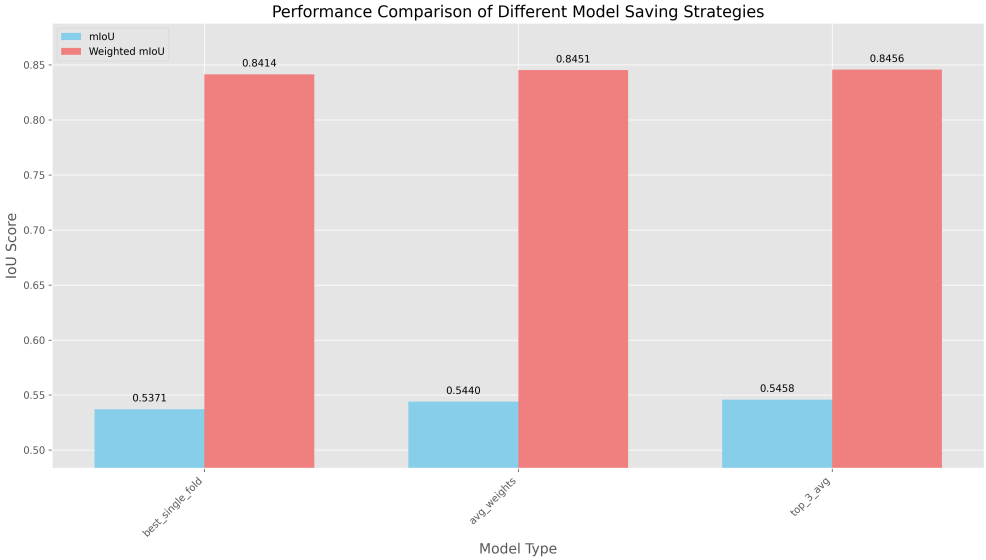


Figure 5: Comparison of the three model aggregation approaches for epochs 30-50, highlighting their performance differences

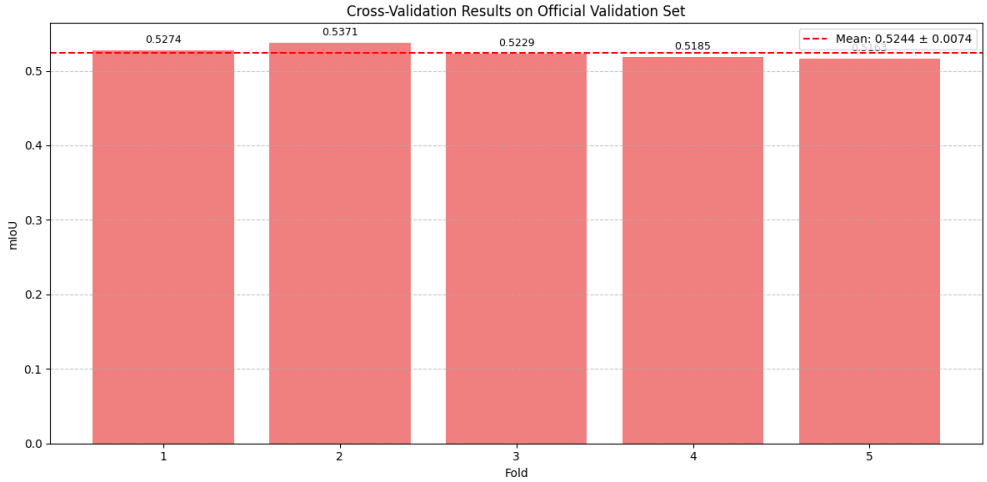
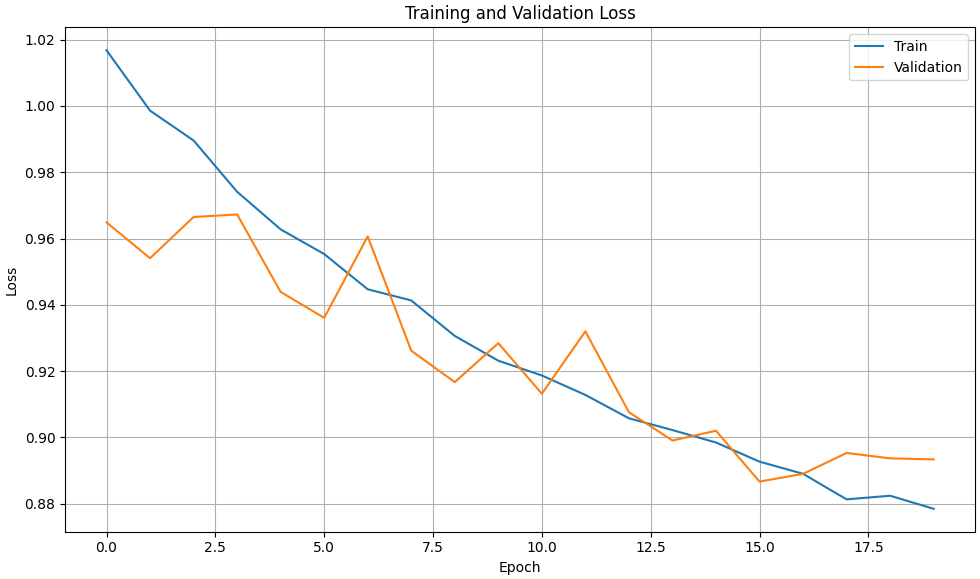
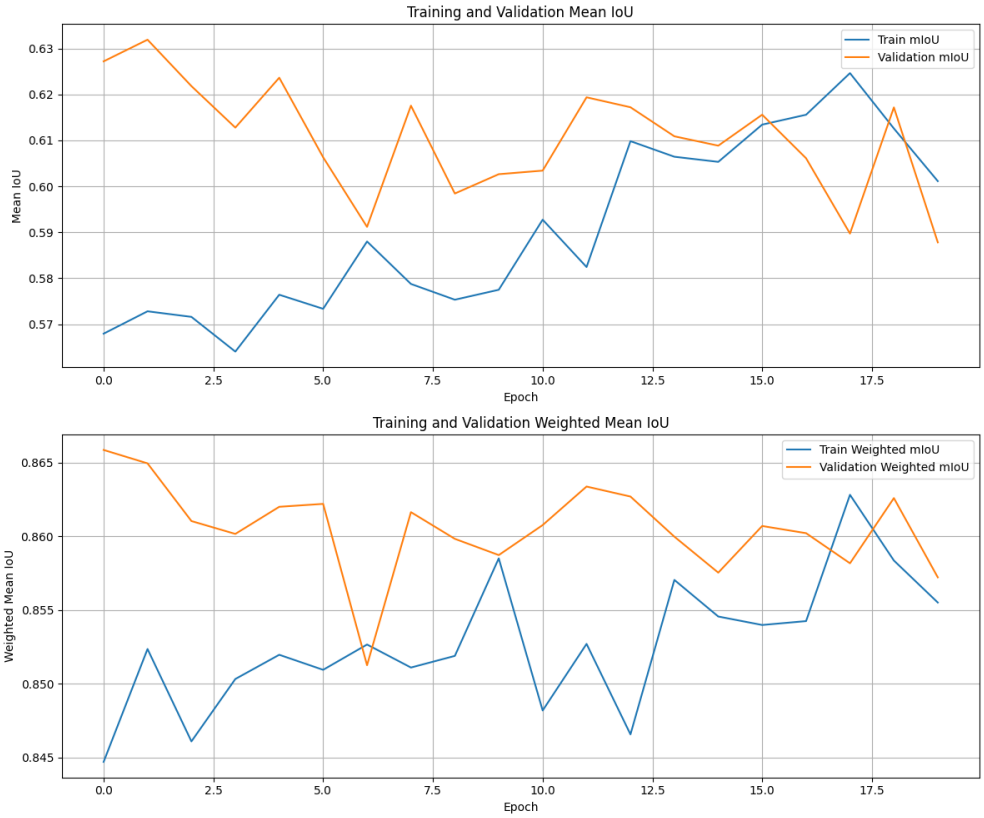


Figure 6: mIoU comparison across all folds on the official validation set, epochs 30-50



(a) Training curves for fold 1, epochs 50-70



(b) IoU progression history for fold 1, epochs 50-70

Figure 7: Training performance for the third cross-validation stage (epochs 50-70)

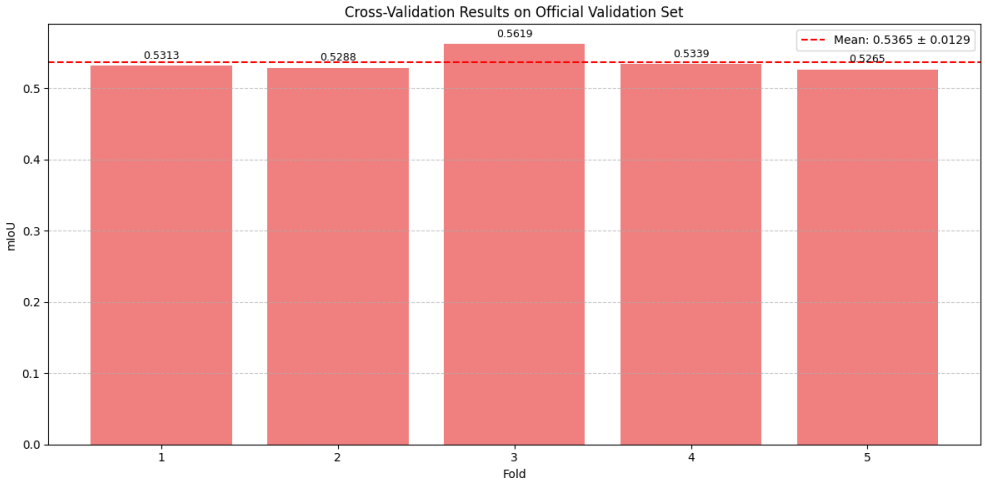


Figure 8: mIoU comparison across all folds on the official validation set, epochs 50-70

Figure 9: mIoU comparison across all folds on the official validation set, epochs 70-90

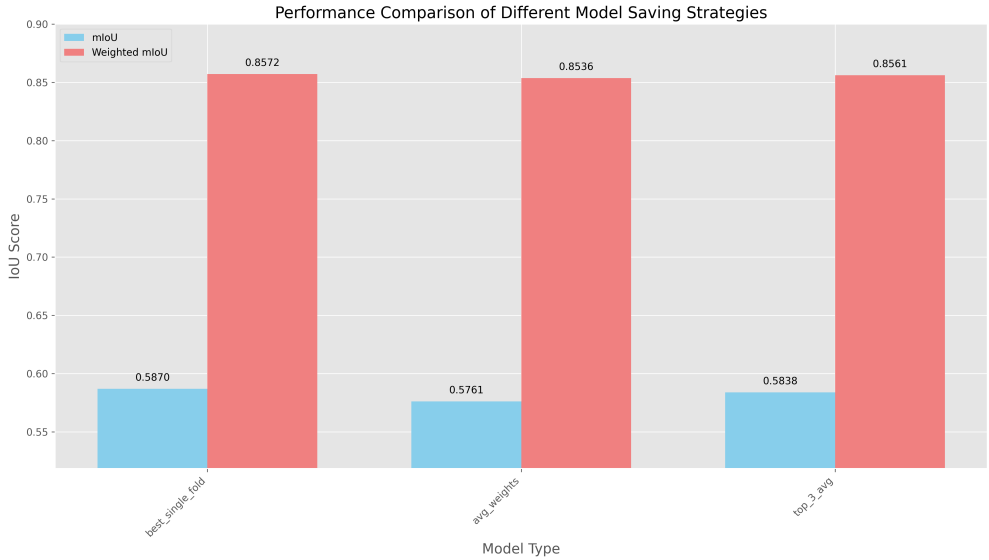


Figure 10: Comparison of the three model aggregation approaches for epochs 70-90, highlighting their performance differences

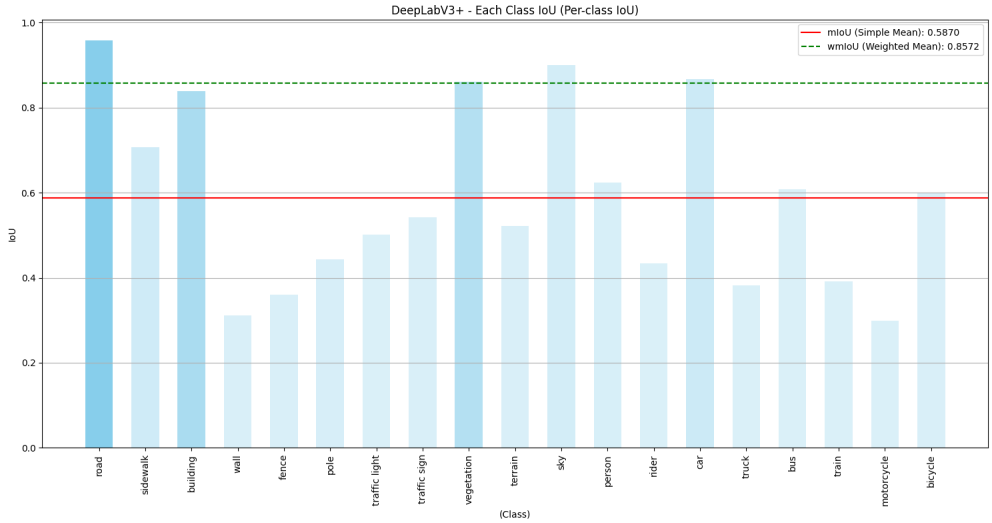


Figure 11: Per-class IoU scores for the final model, showing the model's performance across different semantic categories

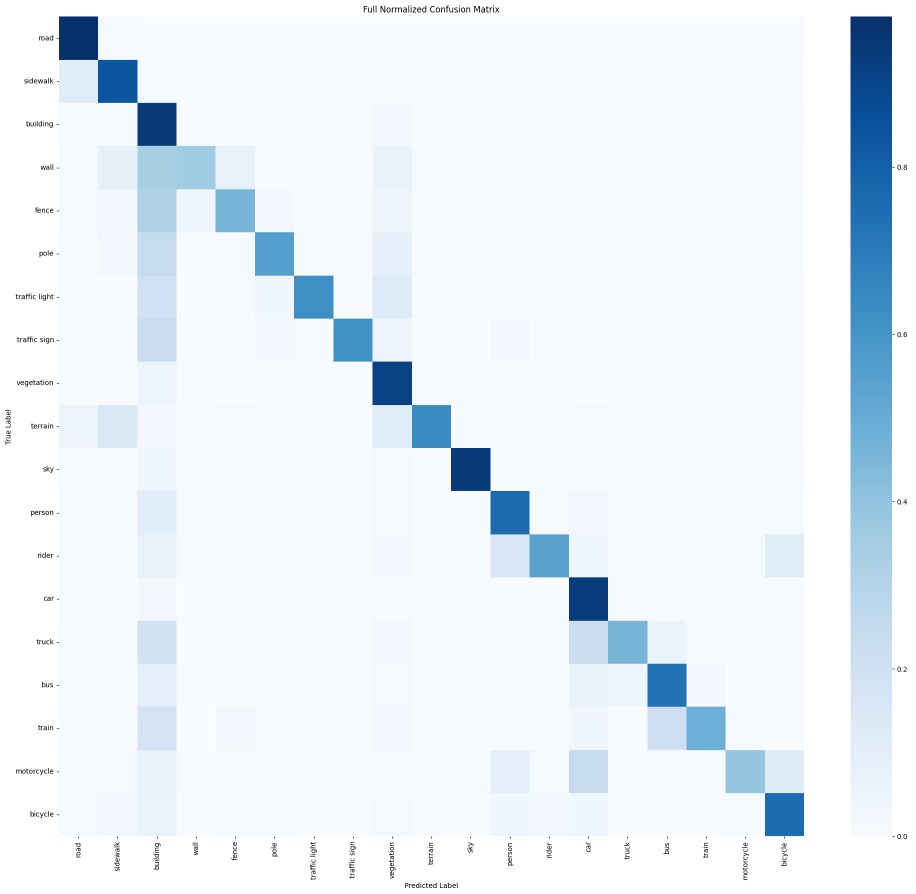


Figure 12: Confusion matrix for the segmentation results, illustrating the distribution of predicted classes versus ground truth

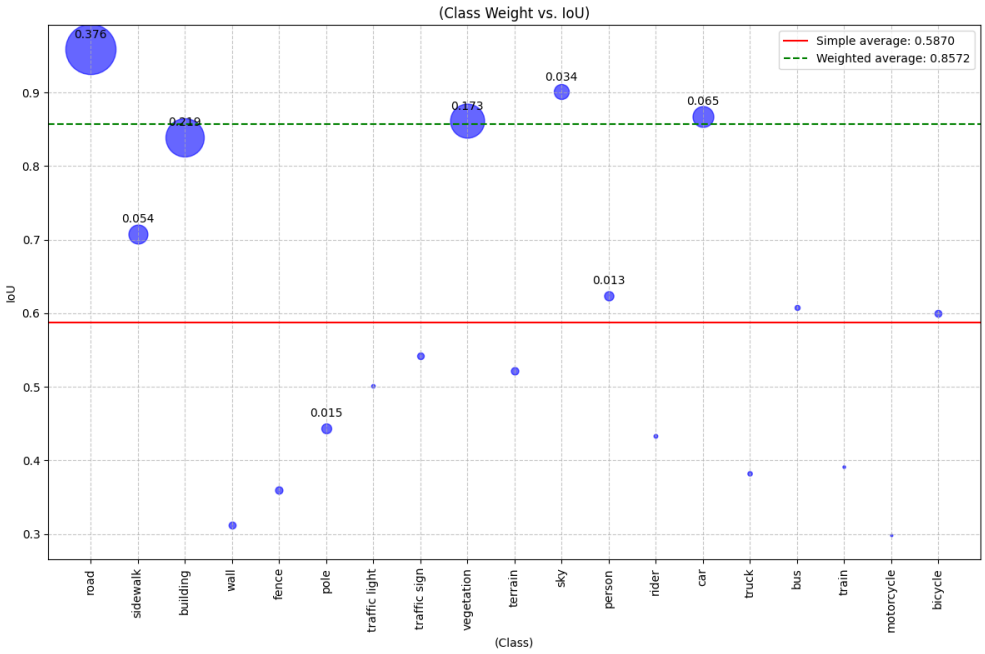


Figure 13: Correlation between class weights and IoU scores, showing the relationship between class frequency and segmentation performance