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 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.

I build my own version of the deeplabv3plus model base on the original implementation, which is also described in the MODEL.md file in the repository or see it in 3. For detailed evaluation, see the EVALUATION.md file in the repository. A brief summary is provided in 4.

Some tricks I used in the training process include:

- Implemented a 5-fold cross-validation strategy to improve model robustness and reduce overfitting.
- Employed a combination of label smoothing cross-entropy loss and Dice loss to enhance segmentation performance. See 5 for details.
- Utilized data augmentation techniques to increase the diversity of training samples and improve generalization. See 7 for brief details and the EVALUATION.md file for more details.
- Implemented a cosine learning rate scheduler for better convergence during training.
- Experimented with different model aggregation strategies to optimize final performance. See 6 for details.

Finally, I use the DeepLabV3+ model to achieve a mean Intersection over Union (mIoU) of 61.9% (Figure 12, 13) on the official validation set. All the figures see in 9.

2 Experiments and Results

2.1 Initial Training without Cross-Validation

I initially conducted experiments without cross-validation to establish a baseline. However, I observed that the model quickly began to overfit, with the loss no longer decreasing after a certain point. Despite this limitation, the model achieved a mean Intersection over Union (mIoU) of approximately 51%. The training curve is shown in Figure 1.

To address the overfitting issue, I implemented a 5-fold cross-validation approach by **splitting the official training set into five subsets**. The combined loss with weighting between label smoothed cross-entropy and Dice loss (0.6:0.4) proved most effective during this phase. This significantly improved performance, with the mIoU reaching approximately 54% by the end of training. Figure 4 compares mIoU across all folds on the official validation set.

2.2 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 but seemingly no improvement. The model aggregation show in Figure 5 highlights the performance differences between the three aggregation strategies. The best performing model from this phase was then used as the pre-trained model for the next training iteration.

2.3 Training Results for Epochs 50-70

Encouragingly, after I apply data augmentation techniques (The level I choose ”standard” first), 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 6, showing the detailed performance metrics across different folds.

2.4 Training Results for Epochs 70-150

For the later training phase, I choose ”advanced” data augmentation, which includes more aggressive transformations. And start to use the cosine learning rate strategy.

2.5 Final Optimization and Results

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 13, 14, and 15, with detailed visualizations available in the directory `outputs/deeplabv3plus_test_results`. Figure 13 shows the per-class IoU scores, Figure 14 presents the confusion matrix for segmentation results, and Figure 15 illustrates the relationship between class weights and IoU scores. Figure **??** compares the performance of the three model aggregation approaches.

3 Models

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.

3.1 Model Architecture Details

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

3.1.1 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.

3.1.2 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.

3.1.3 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

3.1.4 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).

3.1.5 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.

3.2 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.

4 Finial Evaluation Resolution Strategy

For the final evaluation on the validation set, I implemented two distinct approaches to assess the model's performance under different resolution conditions:

4.1 Fixed Working Resolution Evaluation

In the first approach (implemented in `test_deeplabv3plus.py`), both input images and evaluation metrics were processed at the consistent working resolution of 512×1024 pixels. This approach maintains perfect alignment with the training conditions, allowing for a direct assessment of the model's performance without introducing resolution-related variables.

4.2 Original Resolution Evaluation

The second approach (implemented in **`test_deeplabv3plus_origin_resolution.py`**) was designed to evaluate the model under real-world deployment conditions where output is needed at the original high resolution. In this pipeline:

- Input images are downsampled from 1024×2048 to 512×1024 for model inference
- The resulting predictions are then upsampled back to the original 1024×2048 resolution
- Evaluation metrics (including mIoU) are calculated at the full 1024×2048 resolution

This approach provides insights into how the model's predictions translate to full-resolution applications. While this method typically yields slightly lower mIoU scores compared to the fixed resolution evaluation (approximately 0.06% decrease), it provides a more realistic assessment of the model's practical performance on high-resolution imagery. The difference in performance highlights the challenges of resolution changes in semantic segmentation tasks.

5 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:

5.1 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: $\text{target} = 1 - \text{smoothing}$
- For other classes: $\text{target} = \text{smoothing}/\text{num\_classes}$

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.

5.2 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.

5.3 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.

6 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 Figure 4.

7 Data augmentation

To improve model generalization, I implemented a three-level data augmentation framework for the Cityscapes dataset:

1. **None**: No augmentation, only image resizing to target dimensions
2. **Standard**: A set of basic augmentations including horizontal flipping (50% probability), Gaussian blur (30%), gamma adjustment (25%), random upscaling (50%), random cropping (40%), and color jittering (50%)
3. **Advanced**: All standard augmentations plus more aggressive transformations like color channel swapping (10%), grayscale conversion (10%), sharpening (10%), color balance adjustment (10%), enhanced color jittering (15%), noise addition (10%), modified elastic transformation (10%), and color-based image mixup (5%)

The augmentation level could be specified in the configuration file or overridden via command-line arguments. For cross-validation, the data module ensured that:

- Training folds received the specified augmentation level
- Validation folds always used 'none' augmentation to maintain evaluation consistency
- Each fold received independent dataset copies with appropriate transform settings

All transformations were carefully designed to preserve semantic integrity by avoiding black border artifacts, using reflection padding instead of constant black values, and preventing confusing scenes through selective application of transforms.

8 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.

9 Figures

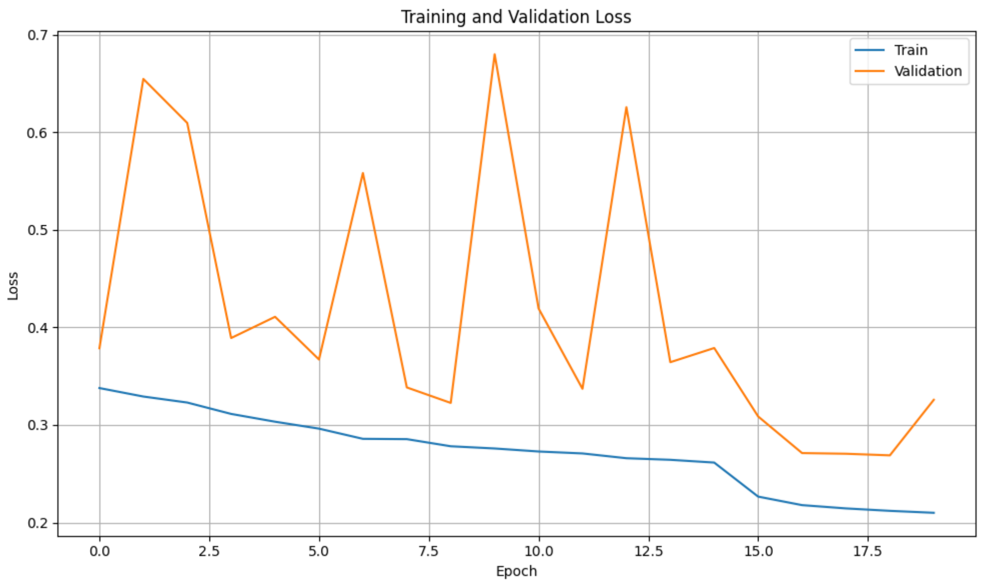

Figure 1: Training curve for epochs 10 to 30 without cross-validation, showing signs of overfitting

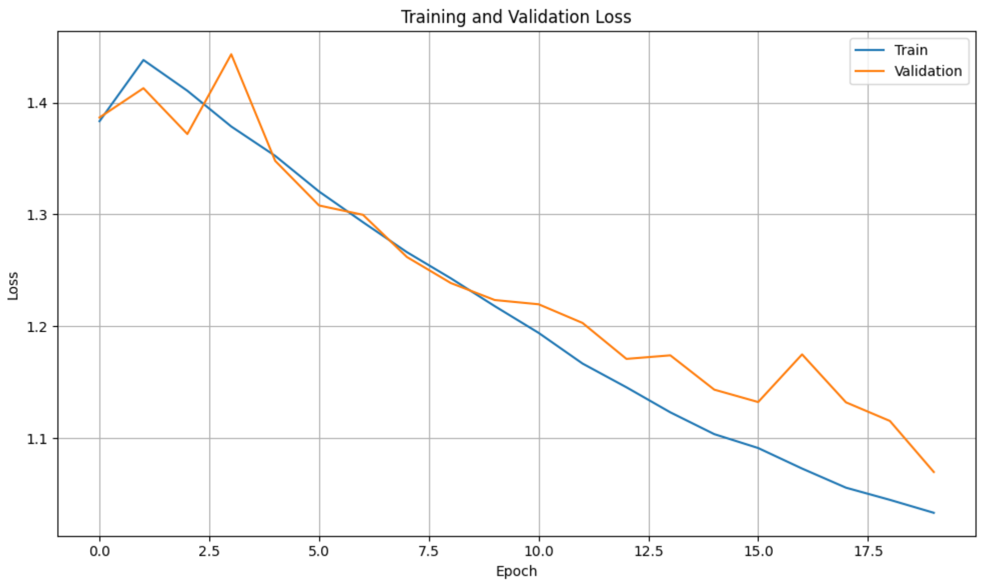

Figure 2: Training curve for epochs 10 to 30 with cross-validation, showing improved generalization

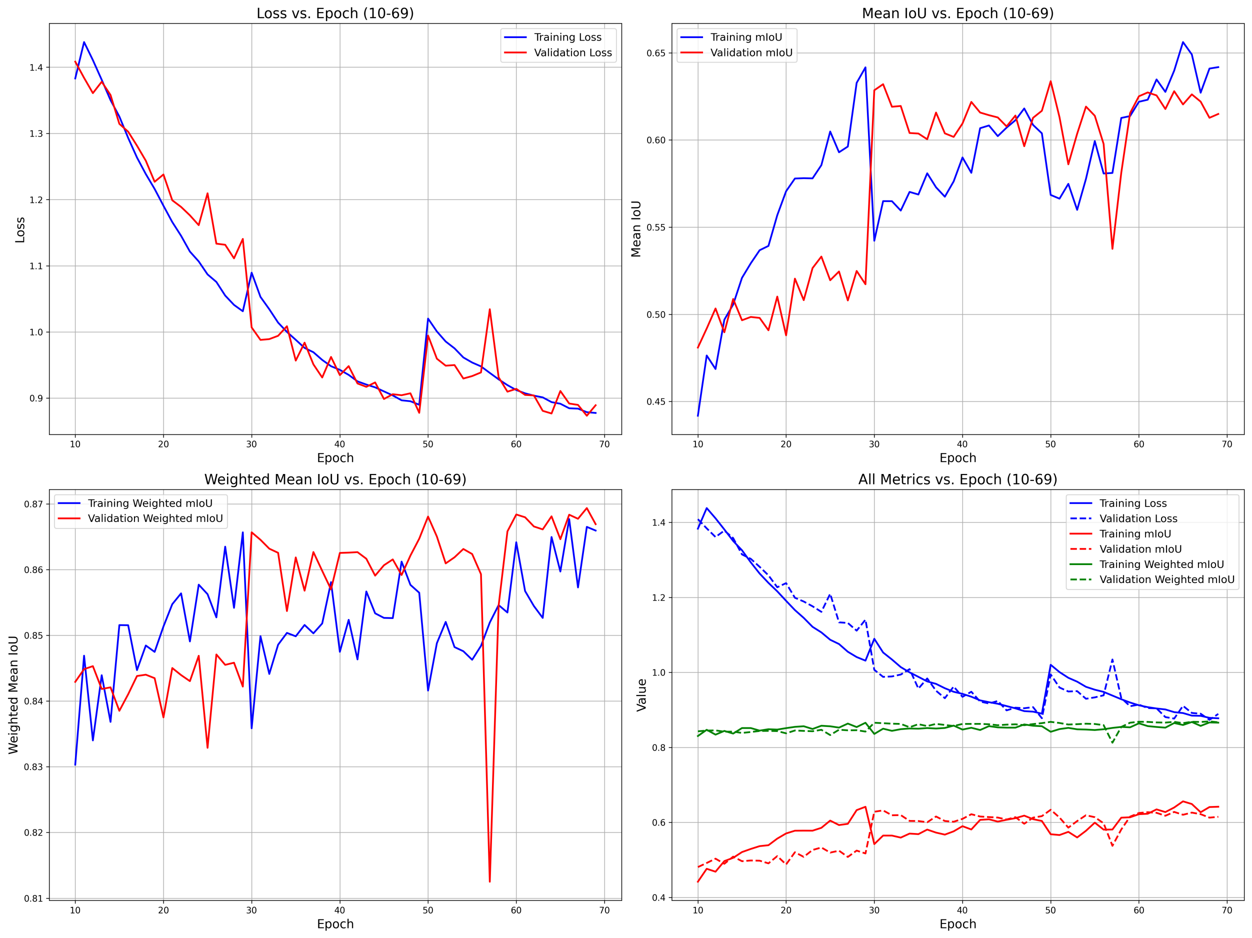

Figure 3: Training metrics for epochs 10-69 for fold 1, showing performance patterns throughout consecutive training intervals with cosine learning rate scheduling and model aggregation applied after each 20-epoch segment

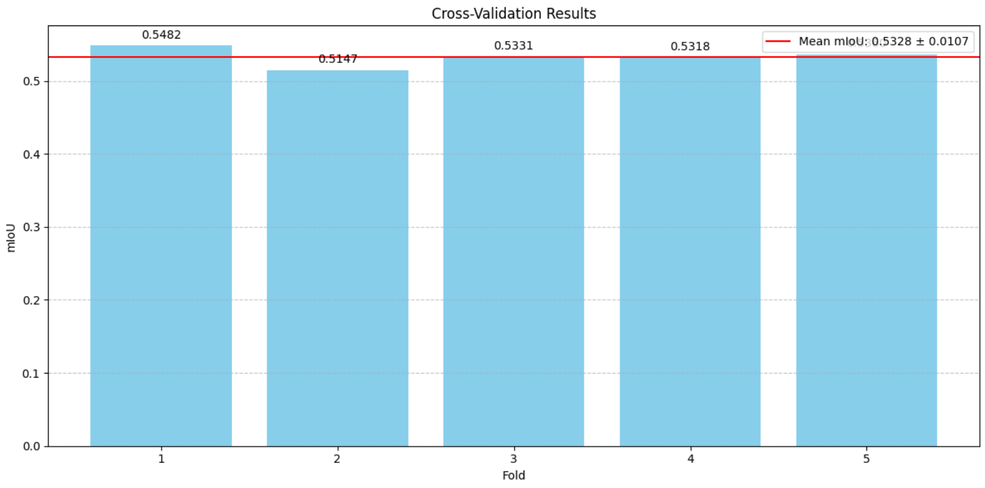

Figure 4: mIoU comparison across all folds on the official validation set, epochs 10-30

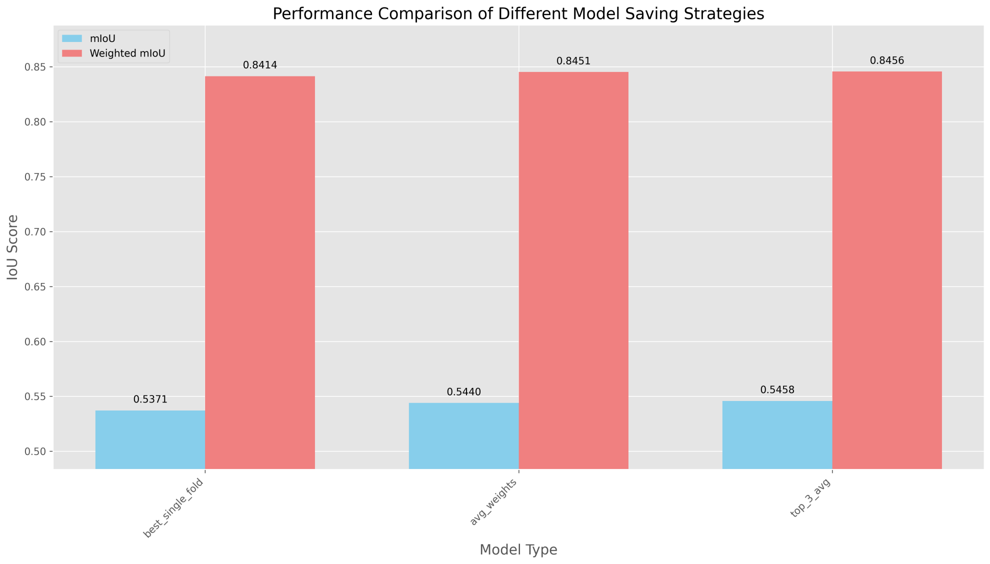

Figure 5: Comparison of the three model aggregation approaches for epochs 30-50, highlighting their performance differences

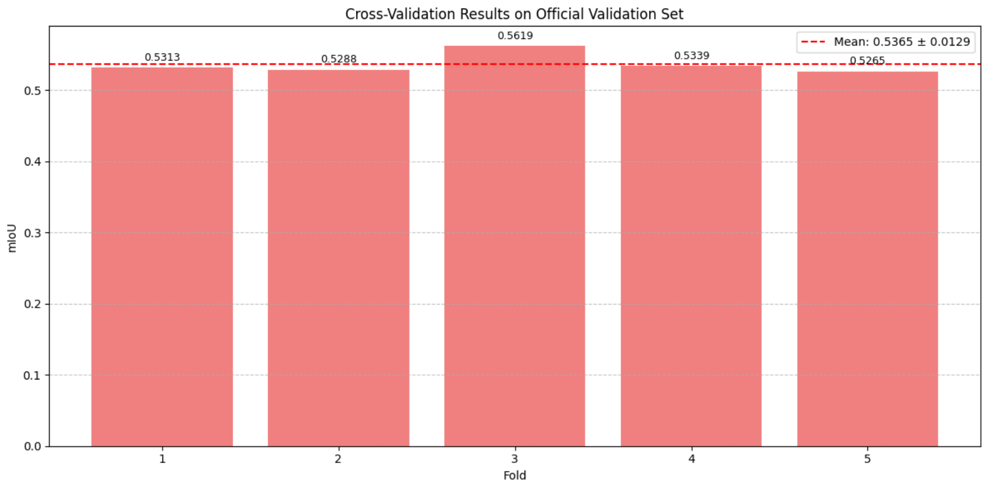

Figure 6: mIoU comparison across all folds on the official validation set, epochs 50-70

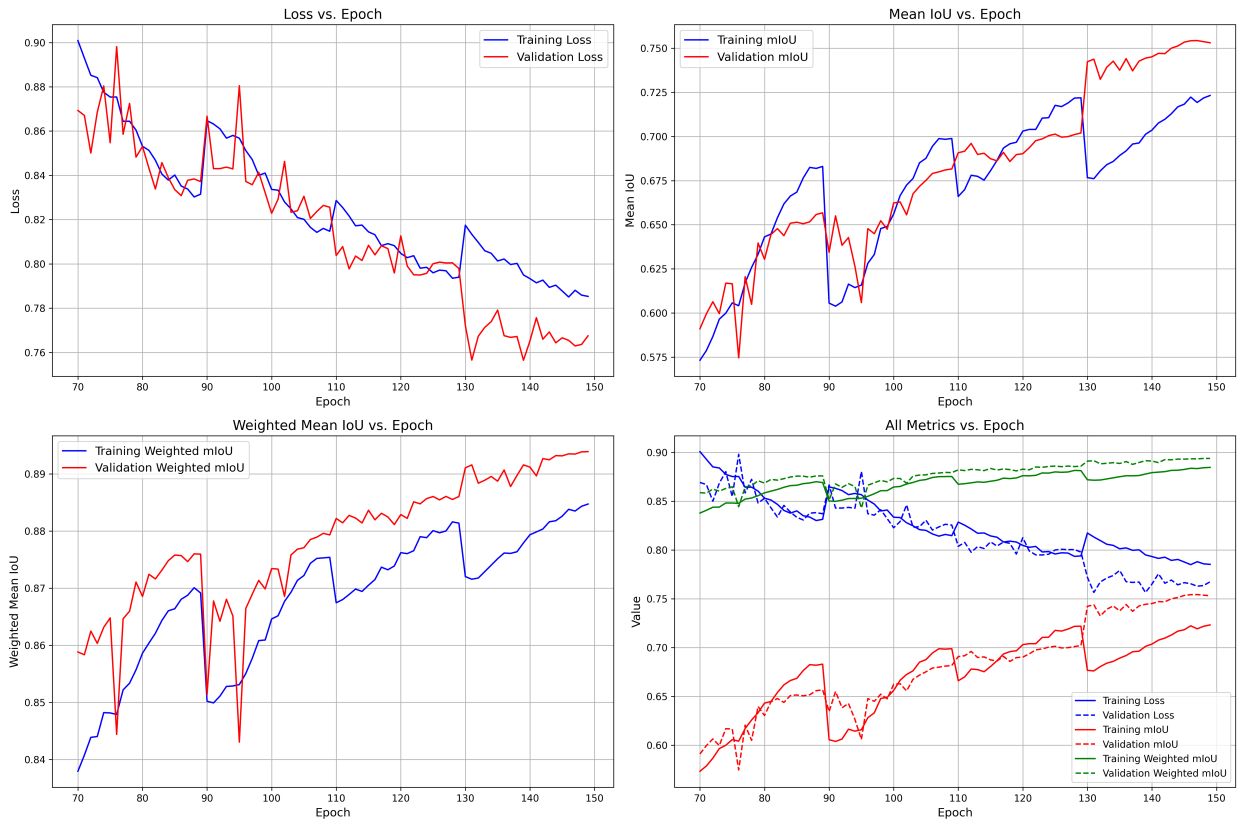

Figure 7: Training metrics for epochs 70-150 for fold 1, showing performance patterns throughout consecutive training intervals with cosine learning rate scheduling and model aggregation applied after each 20-epoch segment

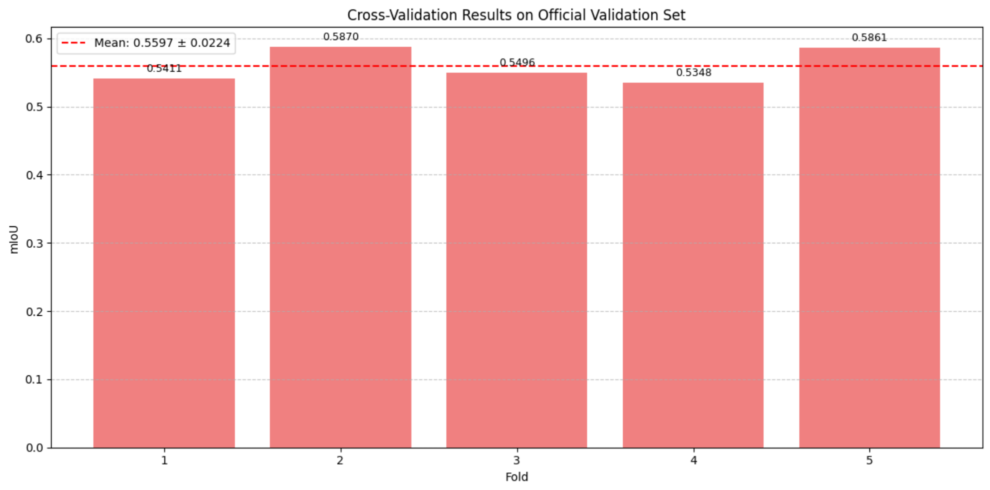

Figure 8: mIoU comparison across all folds on the official validation set, epochs 70-90

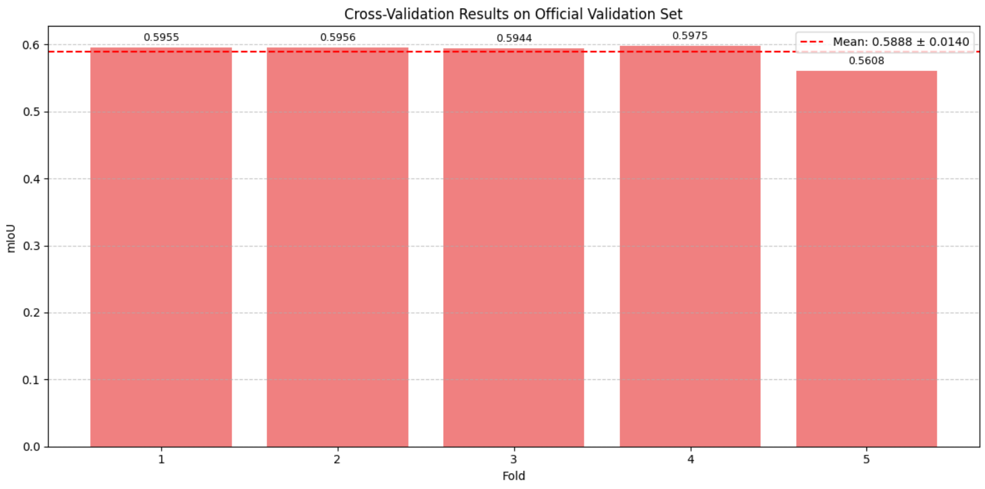

Figure 9: mIoU comparison across all folds on the official validation set, epochs 90-110

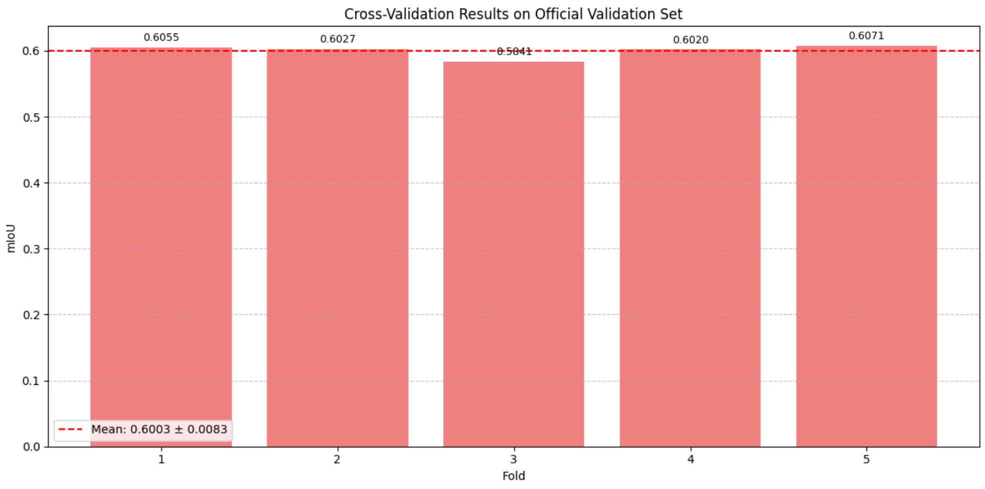

Figure 10: mIoU comparison across all folds on the official validation set, epochs 110-130

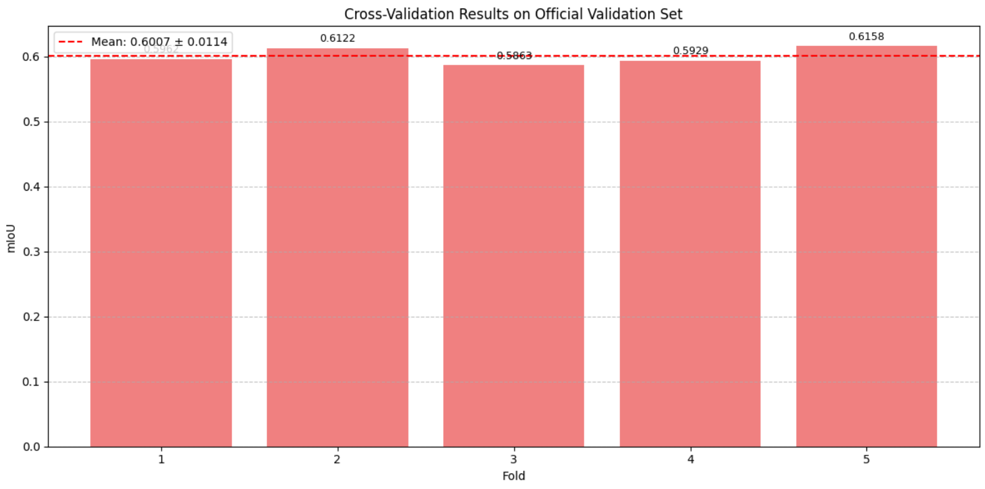

Figure 11: mIoU comparison across all folds on the official validation set, epochs 130-150

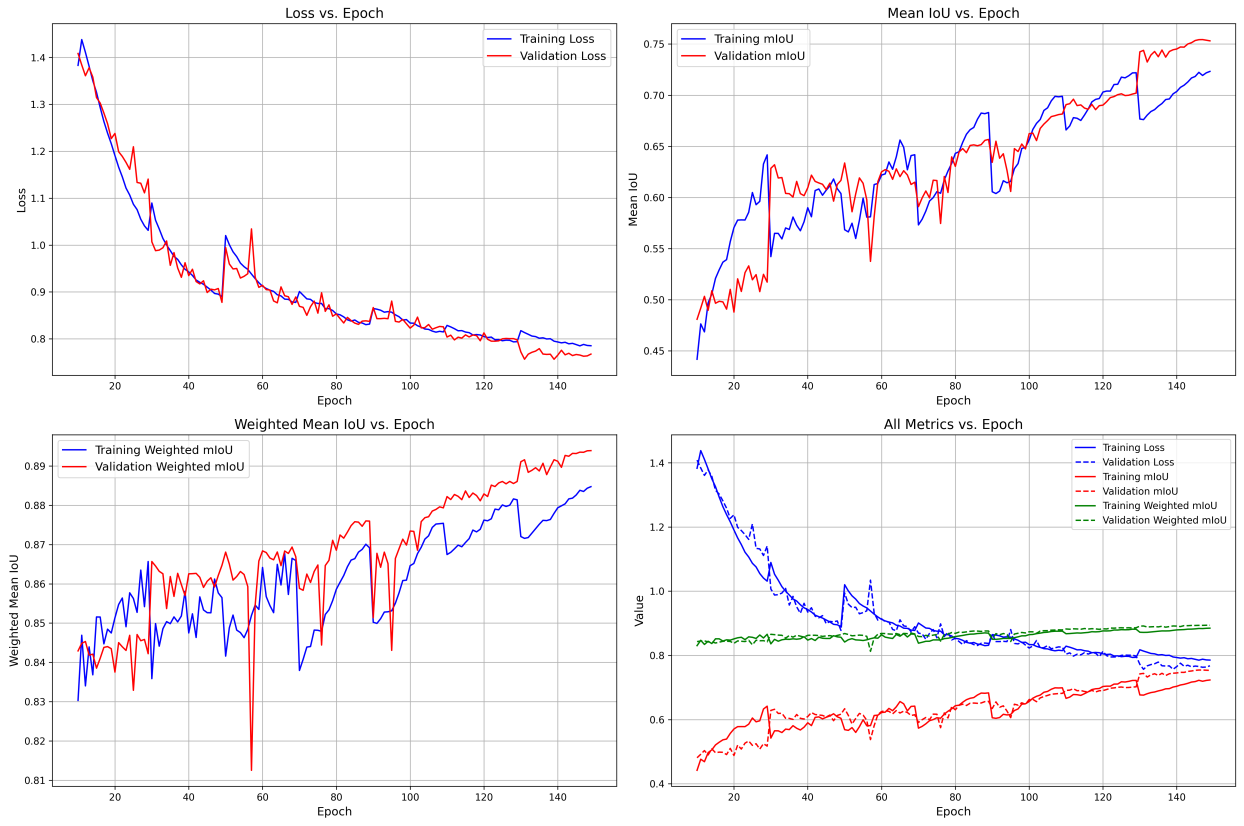

Figure 12: Training metrics for epochs 10-149 for fold 1, showing performance patterns throughout consecutive training intervals with cosine learning rate scheduling and model aggregation applied after each 20-epoch segment

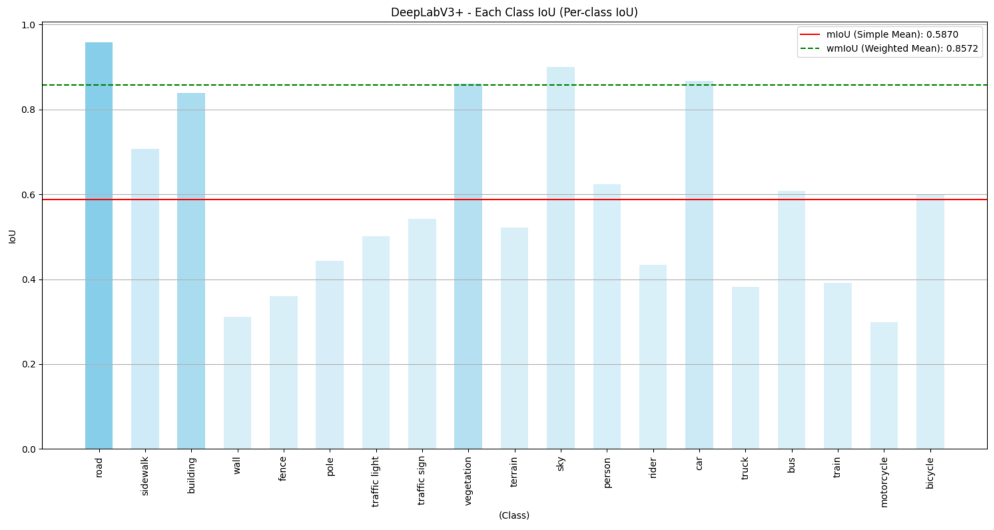

Figure 13: Per-class IoU scores for the final model, showing the model's performance across different semantic categories

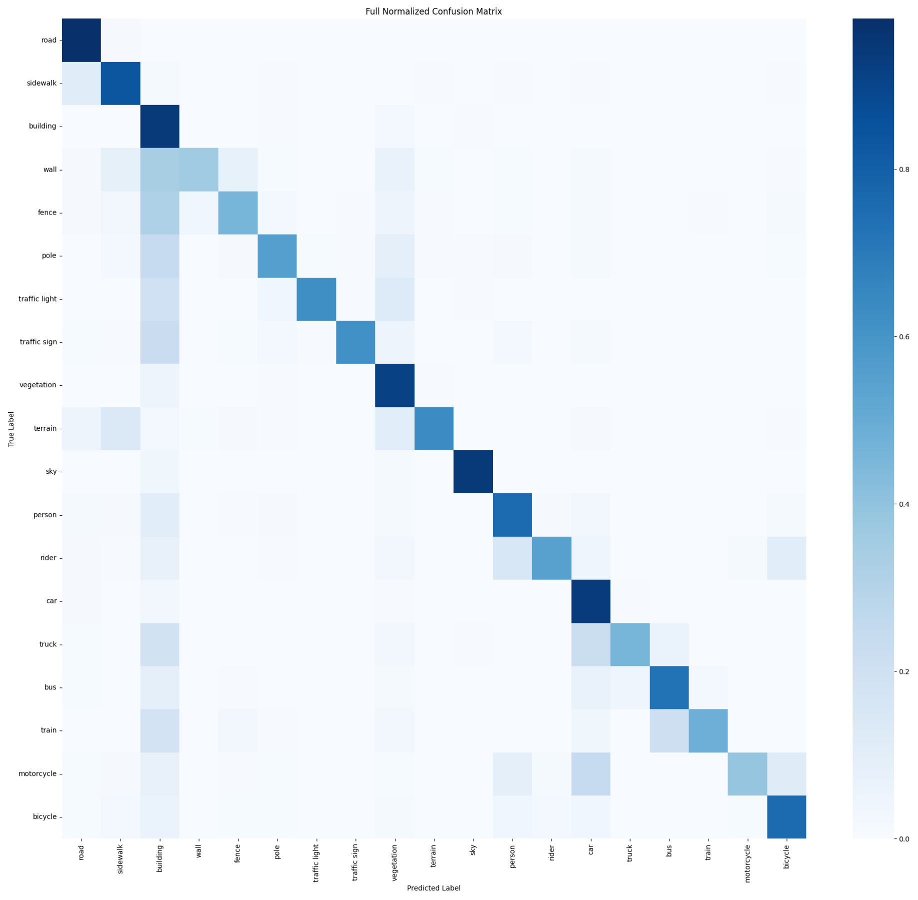

Figure 14: Confusion matrix for the segmentation results, illustrating the distribution of predicted classes versus ground truth

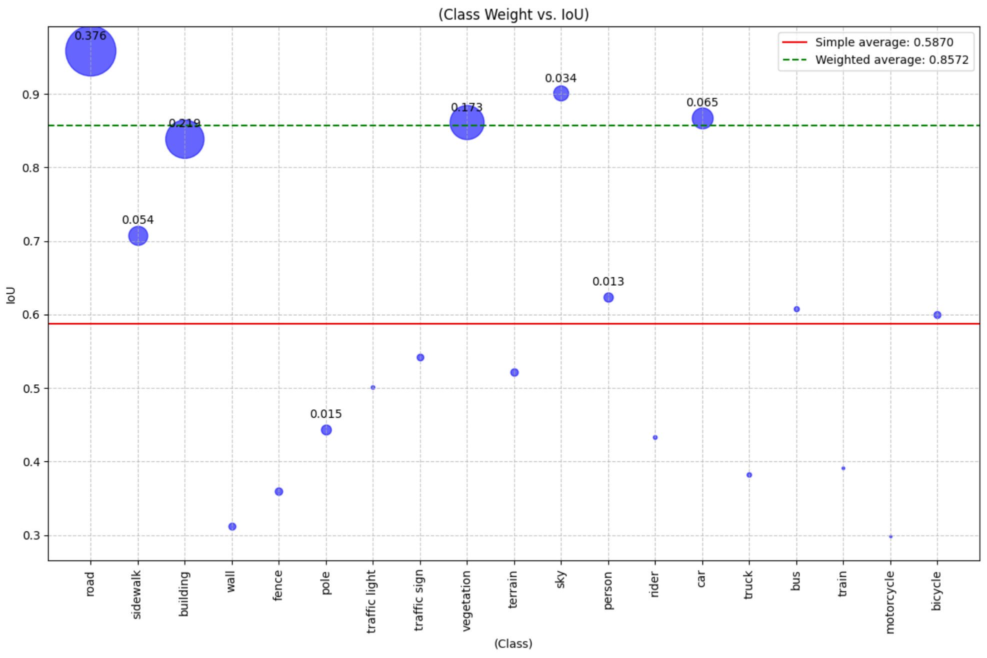

Figure 15: Correlation between class weights and IoU scores, showing the relationship between class frequency and segmentation performance