Università degli studi di Salerno
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Machine Learning

Project Work

A.A. 2024-2025

Group 30 members:

Senatore Annachiara	xxxxxxxxxx	a.senatore150@studenti.unisa.it
Infante Angelo	xxxxxxxxxx	a.infante32@studenti.unisa.it

Sommario

1. Introduction

This project is part of the field of computer vision for autonomous vehicles, with the aim of creating a semantic segmentation system for images acquired on board vehicles in rural environments. The images, captured by a video camera mounted on the moving vehicle, are used to classify each pixel into one of the following eight classes: sky, rough path, smooth path, passable grass, tall vegetation, low vegetation not passable, puddle, obstacle. Additionally, a background class is considered as default.

The developed architecture is based on an autoencoder neural network, designed to perform image segmentation with stringent computational constraints, imposed by the need to perform onboard inference on devices with limited resources. The proposed solution aims to balance accuracy, computational efficiency and GPU memory usage, making it suitable for use in real-world scenarios.

The activity was carried out on a predefined dataset, consisting of 931 images, acquired by the same video camera in different daylight conditions. The dataset was divided into training and validation sets according to independently defined criteria, respecting the constraint of not extending the dataset with new external images. It was possible, however, to apply data augmentation techniques to improve the robustness of the model.

2. System requirements

This section outlines the specific scope, foundational design choices, and operational parameters of the semantic segmentation system developed for autonomous navigation within dynamic rural environments.
To ensure the successful development and subsequent deployability of this semantic segmentation system within its intended resource-constrained operational environment, a precise set of functional and non-functional requirements, alongside explicit project constraints, has been meticulously defined. These foundational criteria guide every stage of the project, from initial design and implementation to rigorous testing and final evaluation, ensuring alignment with the overarching objectives and operational limitations.
The system's core function is to meticulously classify each pixel within images captured by an onboard camera, thereby constructing a detailed understanding of the vehicle's immediate surroundings. The output of this system is a dense pixel-wise mask, where each pixel is assigned to one of nine distinct semantic categories. These categories meticulously delineate common rural scene elements such as various path types (rough and smooth), diverse forms of vegetation (including passable grass, tall vegetation, and low non-passable vegetation), environmental features like puddles, and general obstacles, along with a crucial background class to encompass unclassified regions.
The entire development and validation effort was conducted utilizing a custom dataset comprising of images. These images, captured by the same video camera under a diverse array of daylight and environmental conditions, offer a rich and varied representation of typical rural scenes, essential for training a robust model. The dataset's fixed nature implied a strict adherence to the constraint against augmenting it with external images. However, to significantly bolster the model's robustness and enhance its generalization capabilities across varying visual conditions, a range of data

augmentation techniques was applied during the training phase, effectively expanding the dataset's perceived diversity and exposing the model to a wider spectrum of visual scenarios.

2.1. Functional requirements

The system's core operational purpose dictates that it must accurately perform semantic segmentation on input RGB images, meaning every pixel within an incoming image must be correctly classified into its corresponding semantic category. The direct output of this process is required to be a pixel-wise mask, where each individual pixel carries a specific integer value ranging from 0 to 8. This numerical representation uniquely and unambiguously represents one of the nine predefined semantically relevant classes, a format specifically chosen for its direct interpretability by subsequent control and path planning modules, as well as for efficient data storage and transmission within the autonomous system.

2.2. Non-functional requirements

Operational viability is governed by several critical non-functional requirements. A key operational mandate is that the entire system, encompassing both the computationally intensive training phase and the real-time inference phase, must be fully operable within the Google Colab environment. This ensures broad accessibility, simplifies collaborative development efforts, and leverages cloud-based GPU resources without necessitating extensive local hardware setups. Regarding computational resources, strict limits are imposed on GPU memory usage, a direct reflection of the project's focus on deployability on constrained hardware.
Specifically, GPU memory consumption must not exceed 4 GB during inference to ensure compatibility with target embedded devices and must remain below 5 GB during the training phase, optimizing resource utilization within the shared Colab environment. Furthermore, the model's overall performance must embody an optimal trade-off across three critical dimensions that are interdependent in autonomous systems: the achieved segmentation accuracy, quantified by mean Intersection-over-Union (mIoU) per class, the processing time per image, ensuring real-time capabilities measured in frame rate (frames per second) and overall memory consumption and computational complexity, reflecting its efficiency and feasibility for practical deployment on limited-power, edge hardware.

2.3. Constraints

Several specific constraints have been established to guide the project's scope and methodological approach. Foremost, a fundamental limitation is that no new data from external sources are permitted to be added to the predefined dataset; all development, training, and validation must rely solely on the provided image collection.
Additionally, a fundamental requirement is that the model's architecture, the methodology employed for training, and its resulting performance must be fully transparent, explainable, and rigorously reproducible. This is paramount for scientific validation, for building trust in autonomous systems, and for facilitating future development. Finally, for ease of access, collaboration, and demonstration, all developed code, encompassing training scripts, testing procedures, and saved model weights, is mandated to be delivered as a comprehensive and self-contained Google Colab notebook, ensuring that the entire project can be run and verified with minimal setup and dependencies.

3. Model architecture

The semantic segmentation model used in this project is based on DeepLabV3 with MobileNetV3 Large as the feature extraction backbone. This choice was driven by the need to balance accuracy and computational efficiency, particularly due to the resource constraints imposed by running the model in real time on embedded systems or limited environments such as Google Colab with restricted GPU memory.
In the following sections, the architecture is described in detail, including its components, the design rationale behind the selected model, and the way it was adapted to the specific requirements of this project.

3.1. Overview of the Architecture

Semantic segmentation requires assigning a class label to each pixel of an image. To achieve this, the architecture must be capable of understanding both local details (edges, textures, fine structures) and global context (shapes, object location, spatial relationships).
DeepLabV3 is a well-established architecture that addresses these needs by combining dilated convolutions for context capture and a dedicated decoder module for accurate spatial reconstruction.
The architecture is broadly divided into two main components:

- **Encoder:** A series of convolutional blocks that progressively reduce the spatial dimension of the image, extracting compact latent representations.
  Each block includes:
  - 2D Convolutions
  - Batch Normalization
  - ReLU Activation Function
  - MaxPooling for Downsampling
- **Decoder:** reconstructs the segmentation map from the latent representation, using:
  - Transposed Convolution (deconvolution)
  - Skip connections (optional) to improve spatial reconstruction

Unlike traditional autoencoder architectures where downsampling and upsampling are symmetrical, DeepLabV3 uses an asymmetric design, where the encoder performs most of the semantic abstraction, and the decoder refines the boundaries and restores resolution. The decoder outputs a dense map of class scores (logits), one per pixel, without applying a softmax activation. The final prediction mask is computed by taking the argmax over the class dimension, assigning to each pixel the label of the class with the highest score.

3.2. MobileNetV3 Large Backbone

MobileNetV3 Large is used as the feature extractor within the encoder. It is specifically designed for high efficiency and low computational cost, making it ideal for deployment in environments with limited memory or power consumption.

The architecture of MobileNetV3 Large introduces two key innovations:

- **Depthwise Separable Convolutions**: Instead of performing a full convolution over all input channels, depthwise separable convolutions divide the operation into two simpler steps: depthwise convolution and pointwise convolution. This reduces the number of parameters and operations drastically, without significantly compromising representational power.

- **Inverted Residuals with Linear Bottlenecks**: Each block in MobileNetV3 Large includes a narrow input and output with an expanded intermediate representation. The structure helps to preserve information flow and allows the network to train efficiently even with a small number of parameters.

In this project, the MobileNetV3 Large backbone is initialized with weights pretrained on ImageNet. These weights provide a strong starting point for the training process, allowing faster convergence and better generalization on the relatively small custom dataset used.

3.3. Atrous Spatial Pyramid Pooling (ASPP)

A core component of the encoder in DeepLabV3 is the ASPP module. The purpose of ASPP is to extract information from the feature maps at multiple receptive field sizes, without reducing their spatial resolution. This is achieved through atrous convolutions with different dilation rates, which allow the network to see a larger context around each pixel without increasing the number of parameters or reducing feature map resolution.
The ASPP module within the DeepLabV3 framework typically comprises:

- Multiple Parallel Atrous Convolution Layers: These layers apply convolutions with increasing dilation rates (commonly r=6, 12, 18 for a stride=16 output stride, adjusted based on the specific backbone and output stride). This captures different ranges of contextual information.
- A 1x1 Convolution: Applied at the native scale of the feature map to capture local context.
- An Image-Level Pooling Branch: This branch performs global average pooling on the feature map, followed by a 1x1 convolution. It captures broad, global contextual information, which is then bilinearly upsampled to the original feature map size and concatenated with other features.
- A Final Concatenation and Fusion Step: The outputs from all parallel branches (atrous convolutions, 1x1 convolution, and image-level pooling) are concatenated along the channel dimension. This concatenated feature map is then typically processed by another 1x1 convolution to fuse the multi-scale features, forming a comprehensive representation that is then passed to the decoder for further refinement.

This design is particularly well suited for complex natural scenes, such as the environments in this project, where the size and appearance of classes can vary widely (e.g., tall vegetation vs. puddles).

3.4. Decoder and Segmentation Map Reconstruction

After the encoder has extracted high-level semantic features, the decoder is responsible for restoring the spatial resolution and refining the boundaries of the predicted segmentation. This is crucial to produce accurate and visually coherent masks, especially at the borders between different classes.
The decoder architecture consists of the following stages:

- Upsampling: High-level feature maps are bilinearly upsampled to increase their spatial dimensions.
- Low-level Feature Fusion: The upsampled features are concatenated with lower-level features from the encoder that retain more spatial details. This fusion helps the network recover fine-grained structures lost during downsampling.
- Refinement Convolutions: After fusion, a series of convolutional layers process the combined features to refine the segmentation output.

- Final 1×1 Convolution: A pointwise convolution is applied to project the features into a tensor with a number of channels equal to the number of classes (in this case, 9). The result is a dense map of raw class scores (logits), where each pixel has a score for each class.

At the end of the model, during inference, the predicted class for each pixel is determined by selecting the class with the highest score using an argmax operation over the class dimension. The final output of the model is a semantic mask where each pixel is assigned to one of the following classes:

- Background (0)
- Smooth Trail (1)
- Traversable Grass (2)
- Rought Trail (3)
- Puddle (4)
- Obstacle (5)
- Non Traversable Low Vegetation (6)
- High Vegetation (7)
- Sky (8)

Below we will show the mapping between the classes of our model with the respective colors they assume inside the masks.

Class Name	Class Number	RGB Values	Color
Background	0	255, 255, 255	
Smooth Trail	1	178, 176, 153	
Traversable Grass	2	128, 255, 0	
Rought Trail	3	156, 76, 30	
Puddle	4	255, 0, 128	
Obstacle	5	255, 0, 0	
Non Traversable Low Vegetation	6	0, 160, 0	
High Vegetation	7	40, 80, 0	
Sky	8	1, 88, 255	

Table 1. Class mapping.

The model outputs a tensor of shape [B, 9, H, W], where B is the batch size and H and W are the spatial dimensions. The predicted class for each pixel is obtained by taking the argmax along the channel dimension.

3.5. Structure of the chosen neural network

The structure of the neural network DeepLabV3 with MobileNetV3 Large as the feature extraction backbone network, chosen for training will be shown below.

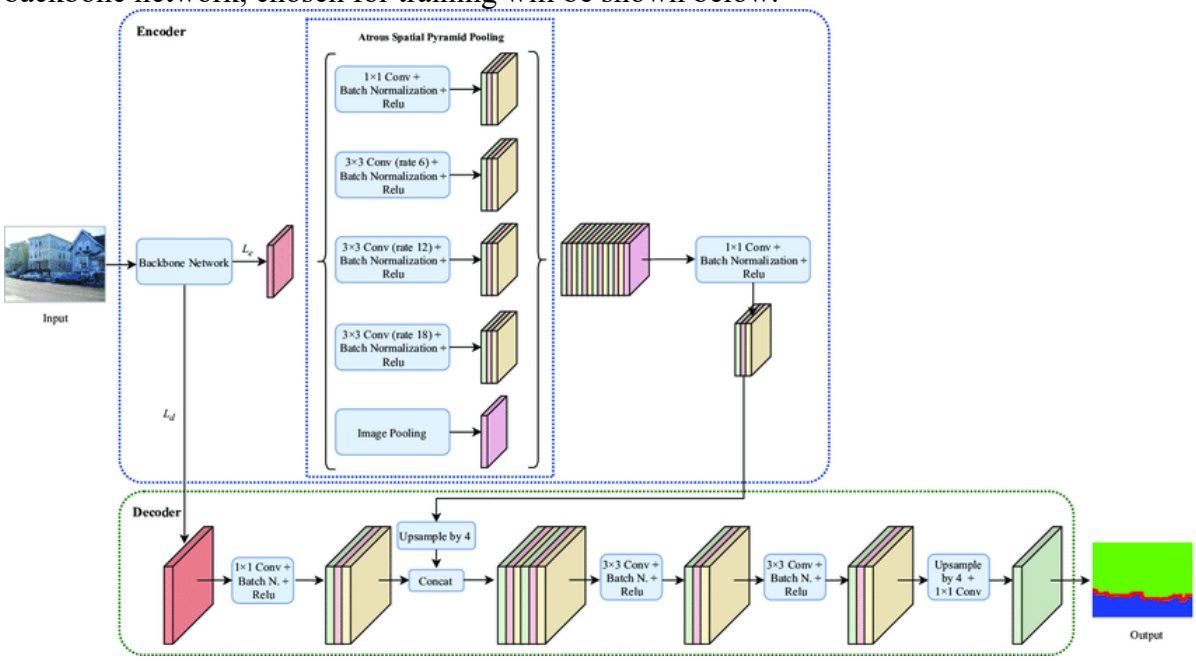

Figure 1. Neural Network Structure

3.6. Justification for Model Selection

The decision to utilize **DeepLabV3** with a **MobileNetV3 Large** backbone was made after a careful evaluation of the intricate trade-offs among performance, segmentation accuracy, and resource consumption. This architecture presents several distinct advantages when compared to other models, such as U-Net, particularly for the specific requirements of this project. Notably, it delivers high accuracy while maintaining a remarkably low memory footprint, a critical factor given the project's stringent requirement to keep GPU memory usage under 4–5 GB.

Furthermore, the MobileNetV3 Large backbone facilitates fast inference speeds, making the system highly scalable and adaptable for various target hardware platforms. Its modular design, which distinctly separates the encoder and decoder components, offers significant flexibility for architectural tuning and experimental iterations. Finally, the availability of pre-trained weights for MobileNetV3 Large, specifically those trained on ImageNet, substantially reduces the overall training time required and markedly improves the model's performance when dealing with smaller, custom datasets such as the one used in this project.

Consequently, this model is well-suited for real-world embedded applications, particularly for autonomous vehicles navigating rural environments, where real-time semantic segmentation must be achieved with minimal computational overhead.

3.7. Other architectures tested

During the initial phase of the project, we explored various architectural solutions with the goal of identifying a model that offered a good trade-off between accuracy and resource consumption, particularly in light of the constraints imposed by the limited GPU memory available during training.

The first solution we experimented with was the development of a custom architecture built from scratch. However, despite the model’s low computational footprint, the segmentation performance

was inadequate for the project's requirements.
We then moved on to more established architectures. We tested a **U-Net** network with a **ResNet-34** backbone, which showed moderately improved performance, but still failed to provide a satisfactory balance between segmentation quality and computational cost. Switching to a deeper backbone, such as **ResNet-50**, resulted in only marginal performance improvements, while significantly increasing memory usage and training time, making it an unsustainable choice under our hardware constraints.
Similarly, we evaluated different configurations of the **DeepLabV3** network, combining it with both **ResNet-50** and **ResNet-34** backbones. While these configurations offered better segmentation accuracy, their high GPU memory consumption made them impractical for our use case.

In light of these considerations, we selected the **DeepLabV3** architecture with a **MobileNetV3 Large** backbone. This solution proved particularly effective in balancing performance and resource usage. MobileNetV3 was specifically designed for scenarios where computational efficiency is a priority, and it enabled us to meet hardware constraints without significantly compromising segmentation quality.
This architectural choice thus represented the best possible compromise between accuracy, efficiency, and compatibility with the available resources.

4. Preprocessing Pipeline

In any computer vision pipeline, especially in semantic segmentation, the preprocessing phase plays a fundamental role in ensuring that the data are standardized, clean, and suitable for training robust models. In this project, the preprocessing pipeline has been carefully designed to ensure both uniformity of input data and the generalization capability of the model. This is particularly important when working with relatively small datasets or when the model is expected to operate under varying environmental conditions, as in the case of rural outdoor scenes.

The preprocessing operations include a sequence of transformations applied to both input images and their corresponding segmentation masks. These transformations can be broadly grouped into resizing, normalization, and data augmentation, with the latter applied only during the training phase.

4.1. Input Normalization and Resizing

The first critical step in the preprocessing pipeline is resizing all input images and their corresponding masks to a fixed target resolution. This is a necessary requirement for most deep learning models, including DeepLabV3, as they demand uniform input dimensions to ensure consistent tensor shapes throughout the network layers.

In this project, all images and masks are uniformly resized to a standardized resolution of 512x512 pixels. This specific resolution was chosen to balance the preservation of image detail with the computational and GPU memory constraints defined for the project. The resizing process ensures that the entire dataset can be efficiently batched and processed during both training and inference.

Following resizing, input images undergo normalization. This process scales the pixel values to align with the statistical distribution of the data on which the backbone network was originally pre-trained, typically ImageNet. Specifically, each RGB channel is normalized using the well-established mean and standard deviation of the ImageNet dataset:

- Mean: [0.485, 0.456, 0.406]
- Standard Deviation: [0.229, 0.224, 0.225]

This normalization step is crucial as it helps the pretrained MobileNetV3 Large backbone to effectively transfer its learned features to the new domain, significantly improving convergence speed and overall training stability.

4.2. Data Analysis

Before defining the augmentation strategies, a preliminary analysis of the dataset was performed to understand its characteristics and identify potential challenges. The dataset comprises **931** labeled images, all collected from diverse rural environments under varying lighting and weather conditions.
Due to a small dataset, containing few samples, data augmentation techniques were applied to improve the model's learning and generalization performance.
We also point out the possibility of increasing performance by **removing some samples** from the available dataset, which are considered incorrectly classified and, consequently, harmful to the model's training. This obviously comes at the expense of an even smaller dataset.
The images exhibit high variability across several key aspects:

- **Vegetation Density and Type**: Ranging from dense tall grasses and bushes to sparser low vegetation.
- **Terrain Appearance**: Including dry, muddy, rocky, and uneven path surfaces.
- **Lighting Conditions**: Encompassing scenes under direct sunlight, deep shadows, and overcast skies.
- **Presence or Absence of Water**: Specifically, occurrences of puddles, which represent a distinct and often challenging class to segment.
- **Obstructions and Obstacles**: Various elements such as rocks, trees, fences, and other impediments commonly found in rural settings.

A significant finding from this analysis was the **unbalanced class distribution** within the dataset. Some classes, such as 'sky' and 'background', appear much more frequently and occupy larger areas in the images compared to minority classes like 'puddles' and 'obstacles'. This inherent class imbalance strongly motivated the implementation of robust data augmentation techniques to synthetically increase the diversity and representation of the underrepresented classes, thereby improving the model's generalization capabilities and preventing biases towards dominant classes. Furthermore, various considerations have been made regarding the type of loss function to use to improve the model's performance despite this issue. These will be presented later.

Overviews on class imbalance

Class	Number of pixels in the Dataset
Background	6858153
Smooth Trail	36688048
Traversable Grass	32586167
Rought Trail	37058441

Puddle	449138
Obstacle	2111474
Non Traversable Low Vegetation	15221916
High Vegetation	88094620
Sky	24988107

Table 2. Pixel distribution between classes.

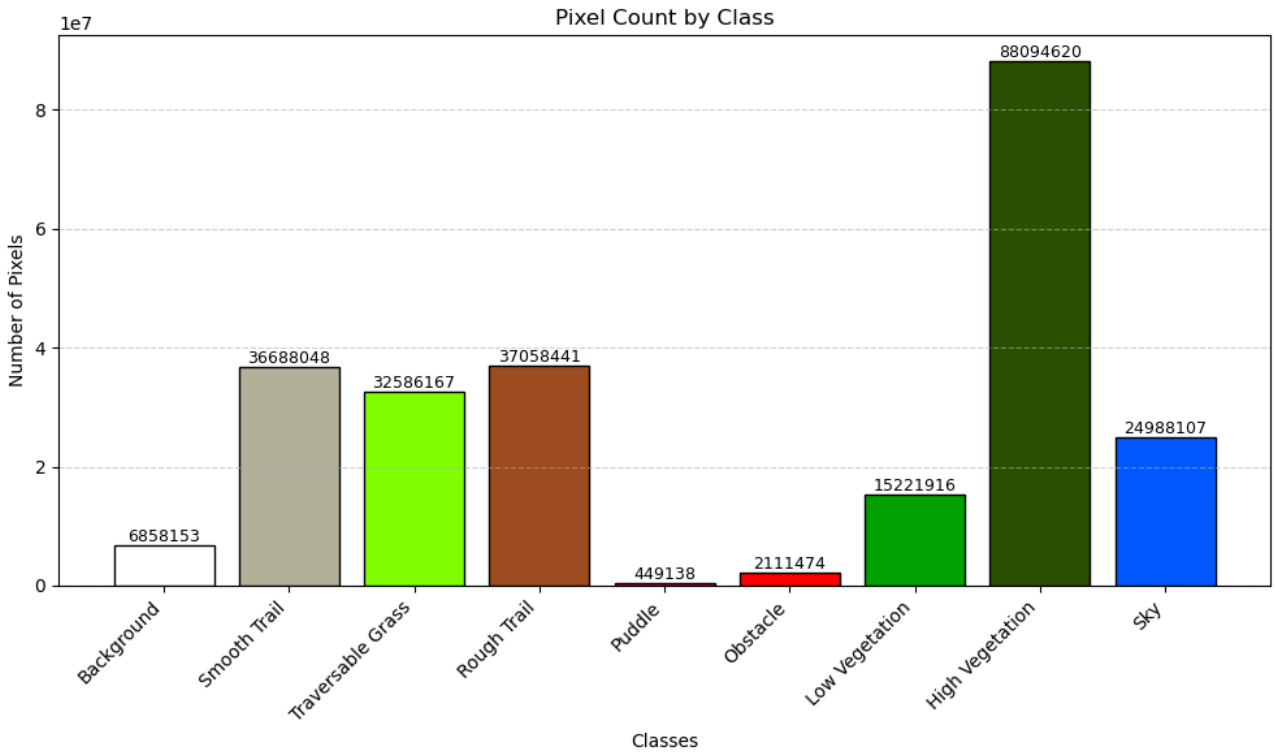

Figure 2. Pixel distribution between classes.

4.3. Data Augmentation

To further enhance the model's robustness, improve its generalization capability, and mitigate the effects of class imbalance and limited dataset size, an extensive data augmentation pipeline was implemented strictly during the training phase. Data augmentation artificially increases the variability of the training data by applying a series of random geometric and photometric transformations that simulate different real-world conditions and viewpoints. This strategy is particularly effective for augmenting smaller datasets and addressing class imbalance issues.
For this project, each original training image was augmented **one times**, effectively duplicating the training set's size. Increasing the number of augmented samples was also tried but this led to a decrease in the model's performance.
The augmentation pipeline was implemented using the **Albumentations library**, A library that offers much more complete and efficient support for simultaneously managing images and masks than the **torchvision.transforms library**. A random transformation from those selected will be applied to each sample. The specific augmentation strategies employed include:

- **Random Resized Crop**: Randomly crops a portion of the image and resizes it to the target dimensions (512x512), with varying scales and aspect ratios. This helps the model generalize to objects appearing at different sizes and locations within the frame.
- **Horizontal Flip**: Flips the image horizontally, increasing spatial diversity and aiding the model in generalizing to symmetric scenes.
- **Vertical Flip**: Flips the image vertically, further enhancing spatial diversity.
- **Rotate**: Applies small-angle rotations (up to 20 degrees) to the image. This simulates slight camera misalignments and improves rotational invariance.
- **Affine Transformations**: Applies random combinations of translation, scaling, rotate and shear transformations. This helps the model become invariant to minor positional or structural variations of objects.
- **Perspective Transformation**: Simulates a change in viewpoint by applying a perspective distortion to the image. This allows the model to better handle objects seen from slightly different angles or perspectives.
- **Random Brightness and Contrast**: Randomly adjusts the brightness and contrast of the image. This improves robustness to varying lighting conditions, such as shadows, highlights, or different times of day.
- **Gaussian Blur**: Applies a soft blur to the image using a Gaussian kernel with random intensity. This helps the model remain effective even when input images are slightly out of focus or contain motion blur.

One of the key advantages of using the Albumentations library lies in its automatic handling of transformations applied to both images and segmentation masks. During data augmentation, it is crucial that any transformation applied to an image is consistently mirrored on the corresponding mask to preserve the spatial alignment between the two. Albumentations simplifies this process by internally distinguishing between geometric transformations (such as rotations, translations, flips, resizing, elastic deformations, etc.) and visual-only transformations (such as brightness, contrast, or color adjustments). Geometric transformations are applied equally to both the image and the mask, while visual transformations are applied only to the image, leaving the mask unchanged—which is the correct behavior in semantic segmentation tasks. This separation is handled automatically and transparently for the user: by simply passing both the image and the mask as part of the input dictionary to the transformation's __call__() method, Albumentations ensures that each operation is applied appropriately to its corresponding data type. As a result, this reduces the risk of manual errors in preprocessing, ensures consistency in augmented datasets, and significantly simplifies the augmentation pipeline, making it more modular and easier to maintain.

Oversampling techniques were also tried for the rarer classes, enforcing the data augmentation for samples containing those classes. This did not lead to significant performance improvements and, consequently, the applied techniques were discarded, since they only led to higher resource consumption. The problem of rare classes was treated by using a specific loss function, which will be discussed later.

Crucially, all augmentations are applied synchronously to both the input image and its corresponding segmentation mask. This meticulous synchronization is vital for semantic segmentation tasks, as it preserves the precise pixel-wise correspondence between the transformed input image and its ground truth label, ensuring the integrity of the training data.

5. Implementation

This chapter details the technical implementation of the semantic segmentation model, encompassing environment setup, custom dataset handling, model architecture definition, utility functions for data splitting and metric computation, and the complete training and inference pipelines. All components are designed to ensure reproducibility and efficient operation within the specified computational constraints.

5.1. Environment Setup and Operational Parameters

The development environment has been configured by importing essential libraries such as PyTorch for deep learning operations, albumentations for data augmentation, PIL (Pillow) for image manipulation, and scikit-learn for managing dataset splits.
Key operational parameters have been defined as follows:

- Image Dimensions: All images are resized to 512x512 pixels (IMG_HEIGHT, IMG_WIDTH) to ensure uniformity and compatibility with the model's input.
- Batch Size: A BATCH_SIZE of 8 was selected to balance computational efficiency and the stability of the training process.
- Data Paths: Paths (DATA_DIR, TEST_DIR) for accessing the training and test datasets have been specified.
- Model Saving: The trained model is saved to a predefined path (MODEL_SAVE_PATH) as best_segmentation_model.pth.
- Logging: A directory (LOG_DIR) has been designated for TensorBoard logs, facilitating training monitoring.

A crucial aspect of this implementation is the mapping between the RGB color values of the segmentation masks and the numerical class IDs. The COLOR_TO_ID variable defines 9 distinct classes:

- Background
- Smooth Trail
- Traversable grass
- Rough Trail
- Puddle
- Obstacle
- Non Traversable Low Vegetation
- High Vegetation
- Sky This mapping is fundamental for converting ground truth masks into the numerical format required for model training.

5.2. Dataset Management: The SegmentationDataset Class

The SegmentationDataset class, extending torch.utils.data.Dataset, is responsible for loading, preprocessing, and applying transformations to the data.
During initialization, the class configures a base transformation that includes image resizing, pixel value normalization (using typical means and standard deviations from ImageNet pre-trained models), and conversion to PyTorch tensors. A NumPy mapper is created to efficiently convert segmentation masks from an RGB format to a class ID format.
The class scans data directories to identify RGB image and label mask pairs. If data augmentation is enabled, multiple augmented versions of each original sample are generated and added to the dataset. The __len__ and __getitem__ methods allow access to individual samples, applying the appropriate transformations (augmentation or base) and returning the image and mask as tensors.

5.3. Model Architecture: The SegmentationModel Class

The heart of the implementation lies in the SegmentationModel class, which encapsulates the DeepLabV3 architecture.
During initialization, a deeplabv3_mobilenet_v3_large model pre-trained on COCO_WITH_VOC_LABELS_V1 weights is loaded from torchvision.models.segmentation. This pre-training provides the model with a solid knowledge base of general visual features, accelerating the convergence process. The model's final classifier layer is then replaced with a new convolutional layer. This new layer is specifically configured to produce an output channel count equal to the number of our segmentation classes (9), adapting the model to our specific task.
The forward method defines the data flow through the model, returning the segmentation predictions.

5.4. Dataset Splitting and Data Augmentation Strategies

The create_split function is crucial for preparing the DataLoaders used during training and validation.
A key element is the definition of the augmentation_transformation, a robust transformation pipeline implemented with *albumentations.Compose*. This pipeline randomly applies one of the following transformations: RandomResizedCrop, HorizontalFlip, VerticalFlip, Rotate, Affine, Perspective, RandomBrightnessContrast, and GaussianBlur. Such augmentations are fundamental for improving model robustness and its ability to generalize to unseen data, reducing overfitting. Resizing, normalization, and tensor conversion transformations are also applied here, ensuring data format consistency.

The **create_split** function creates two dataset instances: one with data augmentation enabled (augmented_dataset) and one without (dataset). The original dataset is split into training and validation sets using random_split, with a default ratio of 80% for training. For the training set, indices are extended to include all augmented versions of the original images, maximizing the effectiveness of data augmentation. Finally, DataLoader instances are generated for both sets, configuring batch size, number of workers, and shuffle/pin_memory options. Information about sample distribution, including the number of original and augmented images, is also provided. This function is used to perform a a split of the dataset into training set and validation set with a percentage of 80/20.

Instead, he **create_split_k_fold** function is designed to generate data loaders for K-fold cross-validation, a robust technique for model evaluation that ensures all data points are used for both training and validation, and that each data point is validated exactly once.

- Initialization:
    - It takes data_dir (defaulting to './resources/train') as the root directory for the dataset and num_fold (defaulting to 3) to specify the number of folds for cross-validation.
    - num_augmentation is set to 1, meaning one augmented version is created for each original image. Consequently, num_versions_per_image becomes 2 (original + 1 augmented).
    - Two SegmentationDataset instances are created:
        - augmented_dataset: This dataset includes both original and augmented images. It's configured to enable augmentation (augmentation=True) and generate one augmentation per image (num_augmentations=num_augmentation).
        - dataset: This dataset contains only the original images, with augmentation disabled (augmentation=False). This is used for the validation sets to ensure evaluation on original, unaugmented data.

    - KFold from sklearn.model_selection is initialized with n_splits=num_fold, shuffle=True to randomize samples before splitting, and a fixed random_state for reproducibility.
- Fold Generation:
    - The function iterates through each fold generated by kf.split(range(len(dataset))). For each fold, KFold provides train_indices and val_indices corresponding to the *original* dataset.
    - For the training set, all_train_indices are meticulously constructed. For each original index in train_indices, it includes both the original image's index from augmented_dataset and its augmented version's index. This ensures that the training set for each fold benefits from data augmentation.
    - train_dataset is created as a Subset of the augmented_dataset using all_train_indices.
    - val_dataset is created as a Subset of the *original* dataset using val_indices, ensuring that validation is performed on unaugmented data.
    - train_loader and val_loader are instantiated as DataLoader objects for the respective datasets, with BATCH_SIZE, num_workers=0 (for simplicity in this example, but can be higher for performance), shuffle=True for training, and pin_memory=True for potentially faster data transfer to GPU.
    - Each pair of (train_loader, val_loader) constitutes a fold and is appended to the folds list.
- Output:
    - A message indicating the number of folds created is printed to the console.
    - The function returns folds, which is a list where each element is a tuple containing the train_loader and val_loader for a specific fold.

This function thus enables a structured K-fold cross-validation strategy, crucial for obtaining a more reliable estimate of the model's performance and for mitigating potential biases from a single train-validation split.

5.5. Evaluation Metrics: Intersection over Union (IoU)

To quantify the accuracy of the model's segmentation predictions, the key metric used is the **Intersection over Union (IoU)**, often also referred to as the Jaccard Index. This metric is widely adopted in semantic segmentation problems and is an excellent indicator of how well the regions segmented by the model overlap with the ground truth regions.

- IoU Calculation:
    - For each segmentation class, the IoU is calculated as the ratio between the area of intersection of the predictions and the ground truth, and the area of their union. In simple terms, it measures how well the region predicted by the model "covers" the correct region in the ground truth mask, relative to the total area they occupy together.
    - The Mean IoU (mIoU) is the average of the IoUs calculated for all classes. This provides a single summary metric indicating the model's overall performance across all categories. An IoU value of 1 indicates perfect segmentation, while 0 indicates no overlap.

The metric will ignore class 0 (background)

5.6. Training Loop and the train_model Function

The core of the model's learning process is encapsulated within the train_model function, which can truly be considered an orchestrator directing the training across epochs and cross-validation folds.

- Training Preparation:
  - The function takes crucial inputs such as the number of epochs (num_epochs), the device for training (CPU or GPU), the optimizer (optimizer), a learning rate scheduler (scheduler), loss functions (criterion), and a variable to track the best validation IoU (Intersection over Union) found so far (best_val_iou).
  - A SummaryWriter from TensorBoard is initialized; this serves as our digital "logbook" to record metrics and visualize the training progress.
  - For each fold of the cross-validation, the model is moved to the chosen device (GPU for speed, CPU for compatibility).
- The Training Cycle per Epoch:
  - For each epoch, the function alternates between a training phase and a validation phase.
  - Training Phase:
    - The model is set to train() mode.
    - Each batch of images and masks is loaded from the train_loader.
    - Gradients are zeroed in the optimizer before each backward pass to prevent unwanted accumulation.
    - The *forward pass* is executed: images pass through the model to generate predictions.
    - *Losses* are calculated by comparing predictions with ground truth masks. Typically, different losses are combined, such as DiceLoss (useful for balancing imbalanced classes) and FocalLoss (which helps the model focus on difficult errors).
    - The *backward pass* is performed: gradients are calculated with respect to the losses.
    - *Gradient clipping* is applied, a technique to prevent excessively large gradients that could destabilize training.
    - The optimizer updates the model's weights.
    - The scheduler is activated to adjust the learning rate, usually reducing it as training progresses, allowing the model to "fine-tune" its weights better.
    - Training metrics (like average loss) are updated and logged to TensorBoard.
  - Validation Phase:
    - The model is set to eval() mode to disable features like dropout and batch normalization, which are specific to training.
    - Gradient computation is disabled (torch.no_grad()) to save memory and speed up validation, as we are not updating weights.
    - Each batch of images and masks is loaded from the val_loader.
    - Predictions and losses are calculated, just like in training.
    - The *Mean IoU* (Mean Intersection over Union) for validation is calculated. This is a key indicator of segmentation quality: the closer it is to 1, the better the model is segmenting.
    - Validation metrics (loss and IoU) are updated and logged.
- Best Model Management:
  - At the end of each epoch, if the current validation IoU surpasses the best_val_iou recorded so far, the current model is considered the best, and its weights are saved. This ensures that at the end of training, we will have the model that performed best on the validation data.

- Output and Cleanup:
    - At the end of each epoch, training and validation metrics are printed to the console.
    - GPU memory is managed, emptying the cache if cuda is used, to prevent memory issues.

5.7. Model Saving

Once the training process is complete (or during training, as specified in the training loop), the ability to save the model's state is fundamental for reusing it in the future without having to retrain it from scratch.

- torch.save(best_model.state_dict(), MODEL_SAVE_PATH): This line of code is responsible for saving only the *weights* (or state) of the model that achieved the best validation IoU during training. The file is saved to the path specified by MODEL_SAVE_PATH, which in our case is best_segmentation_model.pth. Saving only the state_dict() is a common and recommended practice, as it allows the weights to be loaded onto a previously defined model instance, making the file smaller and more flexible.
- A confirmation message is printed, indicating where the model was saved and what its best validation IoU was, and on which fold it was achieved.

5.8. The Model Loading Function: load_model

The load_model function is a utility designed to load a previously trained DeepLabV3 segmentation model from a saved file, preparing it for immediate use.

- Function Signature: load_model(path, device='cuda')
    - path: The file path where the model's weights (.pth file) are stored.
    - device: The computing device ('cuda' for GPU or 'cpu') where the model should be loaded.
- Loading Process:
    1. It first instantiates a new SegmentationModel to create the correct network architecture.
    2. This new model instance is then moved to the specified device.
    3. Finally, torch.load(path, weights_only=True) reads the saved weights from the file, and model.load_state_dict(...) applies these weights to the model's architecture, restoring its trained state.

5.9. The Single Image Prediction Function: predict

The predict function is the final and operational component of our pipeline, designed to take a single image as input and generate its segmentation mask using the trained model. This is where all the knowledge acquired by the model is applied to "see" and classify the pixels of a new image. Before analyzing it, let's consider the pil_to_tensor function, that converts a PIL image to a PyTorch tensor by first transforming it into a NumPy array and then applying the img_transform pipeline. The transformation pipeline, img_transform, includes the following steps:

- Resize: Resizes the input image to fixed dimensions defined by IMG_HEIGHT and IMG_WIDTH.
- Normalize: Applies channel-wise normalization using mean and standard deviation values precomputed from ImageNet, aligning with the expectations of most pretrained models.
- ToTensorV2: Converts the image from a NumPy array to a PyTorch tensor in the proper format (C x H x W).

The predict function performs model inference:
This function takes three main arguments:

1. model: Our pre-trained and loaded DeepLabV3 model.
2. image: The image we want to segment. Note that it expects an image as a NumPy array (like those obtained by reading an image with OpenCV or PIL converted to an array).
3. device: The device on which to perform the prediction ('cuda' for GPU or 'cpu').
4. target_shape: An optional parameter to rescale the output mask to the original input image dimensions, if necessary.

The Step-by-Step Prediction Process:

1. Evaluation Mode (model.eval()): First, the model is set to evaluation mode. This is vital because it disables training-specific behaviors, such as dropout (which introduces randomness) and Batch Normalization statistic updates. This ensures that predictions are deterministic and consistent.
2. Gradient Disabling (with torch.no_grad():): To optimize performance and memory usage, gradient calculation is temporarily disabled. During prediction, we don't need to calculate how the model's weights should change, so avoiding these calculations makes the process faster and more efficient.
3. Image Pre-processing:
    - The input image (which is a NumPy array) is converted into a PIL Image object and then processed by pil_to_tensor. This transforms it into the normalized and resized tensor expected by the model.
    - unsqueeze(0): An extra dimension is added to the beginning of the tensor. This is because PyTorch models expect inputs in "batch" format (even if we are processing only one image, it is considered a batch of size 1).
    - .to(device): The image tensor is moved to the specified computing device (GPU or CPU).
4. Model Inference (output = model(image_tensor)): The pre-processed image is finally passed to the model. The model performs its "forward pass," processing the image through all its layers and producing an output. This output is a map of "logits" or raw scores for each class, for each pixel.
5. Conversion to Probabilities and Classes (predictions = torch.nn.functional.softmax(output, dim=1)):
    - The softmax function is applied to the output. This transforms the logits into probabilities, ensuring that the scores for each pixel sum to 1.
    - pred_labels = torch.argmax(predictions, dim=1): For each pixel, the class with the highest probability is selected. The result is the final segmentation mask, where each pixel contains the numerical ID of the predicted class.
6. Post-processing for Final Output:
    - squeeze(0): The batch dimension (which we added with unsqueeze(0)) is removed.
    - .cpu().numpy().astype(np.uint8): The tensor is moved from the GPU to the CPU (if it was on GPU), converted into a NumPy array, and then into an 8-bit integer format, which is ideal for representing mask class IDs.
    - Optional Resizing: If a target_shape (e.g., the original image dimensions before resizing) was provided, the prediction mask is rescaled to those dimensions. This is useful for overlaying the mask onto the original image without distortion. Image.NEAREST interpolation is used to maintain discrete class IDs.

5.10. Model Testing

After training and saving the best model, the next step is to evaluate its performance on a completely independent test dataset, which the model has never seen during training or validation. The steps for testing will now be listed.

- Model Loading and Preparation:

    - The previously saved trained model is loaded from MODEL_SAVE_PATH.
    - Loading samples to be used for testing from the provided directory.
- Inference and Evaluation:
    - For each sample, the function to perform the prediction is called and the classes present in the ground truth and in the predicted mask and the metrics relating to the IoU for the various classes are calculated.
- Result Visualization:
    - At the end of the various predictions, the results of the various iterations can be viewed, if desired, showing the sample, the true mask and the predicted mask, and, finally, the final score obtained, calculated by averaging all the IoU values of the various predictions.

5.11. TensorBoard

TensorBoard integration in the Jupyter Notebook environment is crucial for visualizing and analyzing the model's training process.
The key commands are:

- **%load_ext tensorboard**: This IPython magic command loads the TensorBoard extension into the Jupyter session, enabling the use of TensorBoard-specific commands.
- **%tensorboard --logdir ./runs --port 6007**: This command launches the TensorBoard web interface.
    - --logdir ./runs: Specifies the directory (./runs) where the training logs (event files generated by SummaryWriter) are stored. TensorBoard reads these files to display metrics, model graphs, and other relevant data.
    - --port 6007: Sets the network port for TensorBoard. While the default is 6006, specifying a different port (like 6007) helps avoid conflicts or allows multiple TensorBoard instances.

5.12. Other Informations

In addition to the implementation structure already explained, the source files contain the choices related to the main hyperparameters and the calls to the functions analyzed, to perform the training of the model correctly, the saving/loading of the various models and their testing. Furthermore, there is an additional function to map the pixels contained in a mask in their respective RGB values (support function for printing the masks during the test).
The id_to_rgb_mask function converts a single-channel, numerical ID mask (where each pixel contains a class ID) back into a three-channel RGB color mask.

- Inputs:
    - id_mask_np: A 2D NumPy array representing the segmentation mask with class IDs as pixel values.
    - id_to_color_map: A dictionary mapping class IDs to their corresponding RGB color tuples (e.g., {0: (255, 255, 255)}).
- Process:
    1. Initializes an empty 3-channel RGB NumPy array of the same height and width as the input id_mask_np.
    2. Iterates through each class_id and its color_rgb from the id_to_color_map.
    3. For each class_id, it finds all pixels in id_mask_np that match that class_id and assigns the corresponding color_rgb to those pixels in the rgb_mask.
- Output:
    - A 3D NumPy array ((H, W, 3)) representing the colored segmentation mask, ready for visualization.

6. Training Strategy

The semantic segmentation model is trained using a structured and iterative process based on a train–validate–analyze–refine cycle. The objective is not only to maximize the segmentation performance in terms of mean Intersection-over-Union (mIoU), but also to ensure that the system remains lightweight, reproducible, and efficient under constrained hardware conditions.
Throughout development, the training code is executed both in Google Colab and locally using a Conda-managed Python environment, as well as on dedicated GPU servers provided the University. This allows for flexibility in experimentation and reduces dependency on runtime limitations typical of browser-based platforms.

6.1. Loss Function

The loss function used during training is a combination of Dice Loss and Focal Loss, designed to tackle common challenges in segmentation tasks, particularly class imbalance and boundary localization.

Before reaching this conclusion, various alternative techniques were tried:

- **Categorical Cross Entropy**. The loss function typically used for multiclass problems. To address the class imbalance issue, a vector containing the various class weights calculated based on the pixel frequencies within the masks was also provided as input. This loss function, even in combination with Focal Loss and/or Dice Loss, did not yield any improvements.
- **Only one function.** Using a single loss function instead of a combined one was also attempted. However, this did not yield any improvement in the attempt to optimize the average IoU on validation and improve the recognition of rare classes.

Now we will discuss the chosen solution.
Dice Loss maximizes the overlap between predicted and ground truth masks by focusing on the region-level agreement, making it especially useful when dealing with small or sparse classes. Focal Loss complements this by penalizing incorrect predictions more heavily for underrepresented or difficult samples, shifting the learning focus toward harder examples.
The final loss function is a weighted sum of the two components:

$$\boldsymbol{Loss = \alpha \cdot DiceLoss + \beta \cdot FocalLoss}$$

where α and β were empirically tuned to achieve the best convergence behavior.

6.1.1. DiceLoss

Dice Loss is a commonly used loss function in semantic segmentation tasks, particularly in scenarios with significant class imbalance, such as small foreground objects on large background areas. Dice Loss is derived from the **Dice Similarity Coefficient (DSC)**, also known as the **F1 Score**, which measures the overlap between two sets. In segmentation tasks, these sets represent the predicted mask and the ground truth mask.
The Dice Coefficient is defined as:

$$\boldsymbol{DSC = \frac{2 * |X \cap Y|}{|X| + |Y|}}$$

where:

- X is the set of pixels predicted as positive (predicted mask)
- Y is the set of actual positive pixels (ground truth),
- $|X \cap Y|$ represents the intersection between prediction and ground truth.

In the continuous case (with probabilistic outputs between 0 and 1), it is expressed as:

$$\boldsymbol{DSC} = \frac{\boldsymbol{2\sum_i p_i g_i}}{\boldsymbol{\sum_i p_i + \sum_i g_i}}$$

where p_i is the predicted value for pixel i, and g_i is the ground truth (0 or 1).
The **Dice Loss** is then defined as:

$$Dice\ Loss = 1 - DSC$$

A smoothing term (epsilon) is often added to avoid numerical instability when the denominator is close to zero:

$$\boldsymbol{Dice\ Loss = 1 - \frac{2\sum_i p_i g_i + \epsilon}{\sum_i p_i + \sum_i g_i + \epsilon}}$$

In our project, the use of Dice Loss allowed us to handle the segmentation of small or sparsely represented objects more effectively. It contributed to improving segmentation quality in critical areas where standard metrics such as Cross Entropy tended to underperform.

6.1.2. FocalLoss

Focal Loss is a loss function designed to address the problem of **extreme class imbalance**, which is common in tasks such as object detection and semantic segmentation, especially when one or more classes are significantly underrepresented compared to the background or dominant classes. This loss function was introduced by Lin et al. in the context of the RetinaNet framework, with the goal of improving the training of classifiers on highly imbalanced datasets.
Traditional loss functions like Binary Cross Entropy (BCE) tend to be dominated by easy examples (i.e., correctly classified samples with high confidence), while underemphasizing hard examples, such as minority classes or boundary regions in segmentation masks. As a result, the model may learn in a suboptimal way. Focal Loss tackles this issue by down-weighting the contribution of easy examples and focusing the training process on hard-to-classify samples.

Binary Case

In the binary setting, Focal Loss modifies the Binary Cross Entropy (BCE) by introducing a modulation term that reduces the loss contribution from well-classified examples:

$$\boldsymbol{FL(p_t) = -\alpha_t(1 - p_t)^{\gamma} \log(p_t)}$$

where:

- p_t is the predicted probability for the true class,
- $\gamma \geq 0$ is the focusing parameter controlling how much the loss down-weights easy examples (higher γ\gammaγ means more focus on hard examples),
- α_t is a class balancing weight useful for imbalanced datasets.

When an example is classified with high confidence (i.e., p_t close to 1), the term $(1 - p_t)^\gamma$ approaches zero, thus reducing the loss for that example. Conversely, hard examples with low p_t get higher weight.

Multi-Class Case

For multi-class segmentation or classification, Focal Loss extends from Categorical Cross Entropy, applying the modulation and class balancing weights across all classes:

$$\boldsymbol{FL = -\sum_{c=1}^{C} \alpha_c (1 - p_{t,c})^\gamma y_c \log(p_{t,c})}$$

where:

- C is the total number of classes,
- $y_c \in \{0,1\}$ is the one-hot ground truth label for class ccc,
- $p_{t,c}$ is the predicted probability for class ccc,
- α_c is the class balancing weight for class ccc,
- γ is the focusing parameter.

In practice, after applying the softmax function to obtain predicted probabilities $p_{t,c}$, the loss is computed to emphasize hard or underrepresented classes.

In our project, Focal Loss was considered as a complement to Dice Loss, particularly for handling the strong imbalance between foreground and background pixels. Its ability to focus on difficult-to-classify pixels adds value to tasks involving fine-grained segmentation or detection of small objects.

6.2. Hyperparameter Analysis

The various hyperparameters used will be analyzed below.

- **Batch size:** choosing a BATCH_SIZE of 8 is a compromise mainly dictated by GPU VRAM memory constraints. Segmenting 512x512 pixel images with a complex model like DeepLabV3 requires significant memory consumption to store intermediate activations and gradients during training. A larger batch size could easily cause "Out Of Memory" (OOM) errors on GPUs with limited VRAM and overcoming constraints on the use of computational resources. A batch of 8 offers a balance between computational efficiency (taking advantage of GPU parallelism) and gradient stability (being more stable than very small batches) without exceeding the hardware capabilities.
- **Learning rate:** the learning rate of 1e-4 (0.0001) was chosen as an optimal compromise between convergence speed and training stability.
    - Empirical Tests: Lower learning rate values proved to be too conservative, excessively slowing down the model's convergence. Conversely, higher values led to performance degradation, resulting in instability and loss oscillations.
    - Fine-tuning Pre-trained Models: This moderate value is particularly well-suited for fine-tuning pre-trained models, allowing for a "fine-tuning" of weights without compromising previously acquired knowledge.

- **Coefficients for the loss function:** The combined loss function (0.7 * DiceLoss + 0.3 * FocalLoss) was chosen to leverage the strengths of both losses in semantic segmentation, particularly after fine-tuning.
    - Dice Loss (0.7 weight): Predominant because it directly optimizes regional overlap and effectively handles class imbalance. Since the primary evaluation metric is IoU, making dice loss more influential is a natural choice. In fine-tuning, it ensures the model continues to refine object shapes and boundaries.
    - Focal Loss (0.3 weight): Contributes to improving performance on hard examples or ambiguous pixels (often at object borders), preventing easy examples from dominating the loss. The lower weight ensures the loss doesn't become too sensitive to noise or destabilize training.
- **γ value for FocalLoss: t**he γ parameter in Focal Loss is crucial for modulating its behavior, reducing the weight of easy examples and increasing that of hard examples.
    - Standard and Proven Value: The choice of γ=2.0 is widely adopted and supported by empirical evidence (proposed in the original Focal Loss paper). This value has been shown to offer an optimal balance.
    - Effective Focusing: With γ=2.0, well-classified (easy) examples see their contribution to the loss drastically reduced, while the weight of difficult examples remains significant. This pushes the model to focus more on problematic cases (e.g., borders, ambiguous pixels).
    - Training Stability: A γ=2.0 value ensures robust focusing without introducing instability into the training process, which might occur with excessively high γ values.
- **Number of folds:** the choice to perform cross-validation with 5 folds was made because, in this way, the original dataset (without augmentation) is always split with a percentage of approximately 80/20 between training set and validation set, thus maintaining the percentages chosen also for the split without cross-validation.
- **Number of epochs:** the number of epochs varied based on the tests performed. The final choice fell on 20 training epochs, as it was observed that the model's learning did not improve significantly after this number of epochs, with the selected parameters.
- **Early stopping value:** the early stopping value was chosen to avoid continuing to train a model that isn't improving and tends to overfit. The threshold of 10 was chosen, which is high for 20 epochs but appropriate when training has been performed with a larger number of epochs.

6.3. Training Description

Training is carried out using the Adam optimizer, with an initial learning rate set to 1e-4. A data loader is defined to handle batch-wise image and mask loading, applying transformations such as resizing (to 512x512 pixels), normalization, and data augmentation through the Albumentations library. To comply with memory constraints, the batch size is adjusted to ensure that GPU memory usage during training remains below 5 GB. In most cases, a batch size of 8 with image resolution of 512×512 is used. When running locally or on MIVIA servers, larger batch sizes are occasionally used to speed up convergence, taking advantage of greater GPU memory availability. An early stopping mechanism is implemented, monitoring the validation loss and halting training if no improvement is observed for a defined number of epochs. Model weights are saved at each epoch with improved performance. An early stopping strategy is also implemented to avoid prolonged training on models that do not improve their learning and tend to overfit.

Initially, a training strategy was used based on splitting the dataset into a training set and a validation set with a ratio of 80/20. This allowed for initial performance and constraint compliance

tests, also testing training and validation with different networks, a greater number of epochs, and different hyperparameter values.
After making the first choices regarding the network and parameters, the training strategy moved on to cross-validation. This happened later and not initially because this technique requires a longer period to be used, reducing the number of attempts that can be made. For this reason, it was chosen to carry it out only after having been certain about the choice of the main aspects.

Furthermore, it was chosen to perform data augmentation **on-the-fly** during training rather than generating and storing augmented samples **offline**. This decision was driven by several considerations aimed at optimizing both resource usage and training efficiency:

- **Reduced Storage Requirements**:
  Offline augmentation requires pre-generating and saving all augmented images, which can significantly increase storage usage, especially when dealing with large datasets or multiple augmentation combinations. On-the-fly augmentation eliminates this overhead by applying transformations dynamically at runtime.
- **Greater Variability Across Epochs**:
  With on-the-fly augmentation, the model is exposed to different transformed versions of the same image in each epoch. This leads to a more diverse training experience and improves generalization. In contrast, offline augmentation produces a fixed set of transformed samples, limiting variability.
- **Faster Dataset Iteration**:
  Loading precomputed augmented data can increase I/O load, especially when the dataset is large. With on-the-fly augmentation, the raw dataset remains lightweight, and transformations are applied in-memory, typically using efficient CPU-based operations. This results in better utilization of system resources without I/O bottlenecks.
- **Efficient Resource Management**:
  Performing augmentation during training avoids the need to store multiple copies of the same image in memory or on disk. This is especially important when operating in resource-constrained environments (e.g., limited disk or RAM availability).
- **Flexibility and Maintainability**:
  On-the-fly augmentation allows easy modification or tuning of the transformation pipeline without needing to regenerate the entire augmented dataset. This improves experimentation speed and simplifies workflow iteration.

6.4. Inference Workflow

After the training phase, the trained model is employed to perform inference on arbitrary, previously unseen images. This process follows the same preprocessing and prediction pipeline used during validation, ensuring consistency and promoting code reusability. Specifically, images are resized to the input dimensions expected by the model, normalized using the same statistics applied during training, and passed through the network for prediction.
The output segmentation mask is post-processed, typically by mapping class indices to RGB colors, to produce a human-readable result. This inference phase is not intended for formal evaluation metrics but rather serves to qualitatively assess the model's generalization capabilities on new, real-world examples.
The entire inference procedure is optimized to be lightweight and efficient, enabling its execution in diverse environments, such as Google Colab or local machines. When deployed on the MIVIA

servers, the availability of high-performance GPUs significantly accelerates inference, particularly in scenarios involving the segmentation of large batches of high-resolution images. This makes the system suitable for practical applications and real-time or near real-time performance testing.

7. Evaluation Analysis

The performance evaluation of the trained model, in addition to calculating the IoU, also takes into account other parameters, already explained in the previous analyses. These will be presented again below for further analysis.

- **Intersection over Union:** this metric, as previously mentioned, is the main metric for analyzing the model's performance. It is calculated both for the individual classes, with the exception of class 0 (background), and as an average of the latter, to give an idea of the general performance achieved.
- **Resource constraints:** another evaluation metric to keep in mind is the resource usage constraint. In fact, the model, during training, must not exceed 5 GB of GPU RAM, while during testing, it must not exceed 4 GB of GPU RAM. This means we need to keep an eye on the weight of our model and neural network.
- **Frame rate of processing:** this metric, used in particular during model testing, allows you to establish how quickly the trained model reacts to inputs using a pre-established amount of resources.

These values were constantly monitored during development to choose the best compromise between accuracy and computational efficiency.

8. Results

The results obtained from an example will be presented, using cross-validation with the various choices documented so far, using 5 folds and 20 training epochs.

8.1. Training and Validation losses

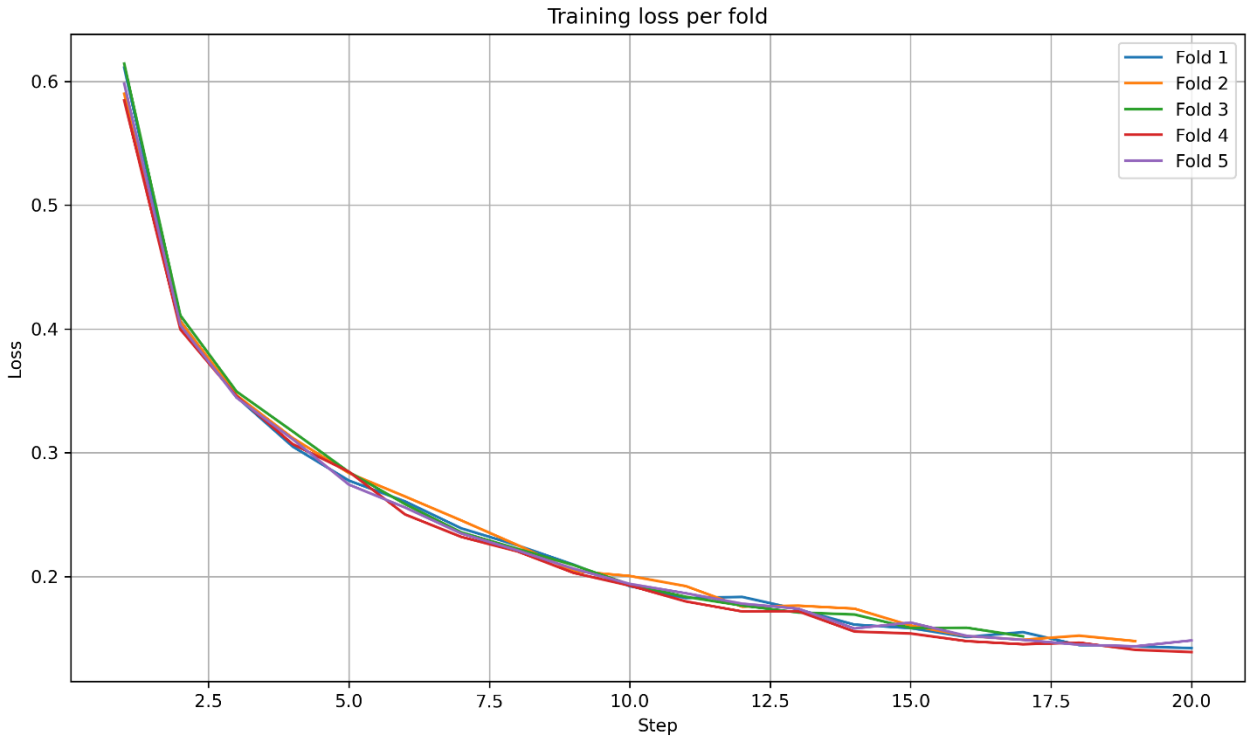

Figure 3. Training Loss per fold.

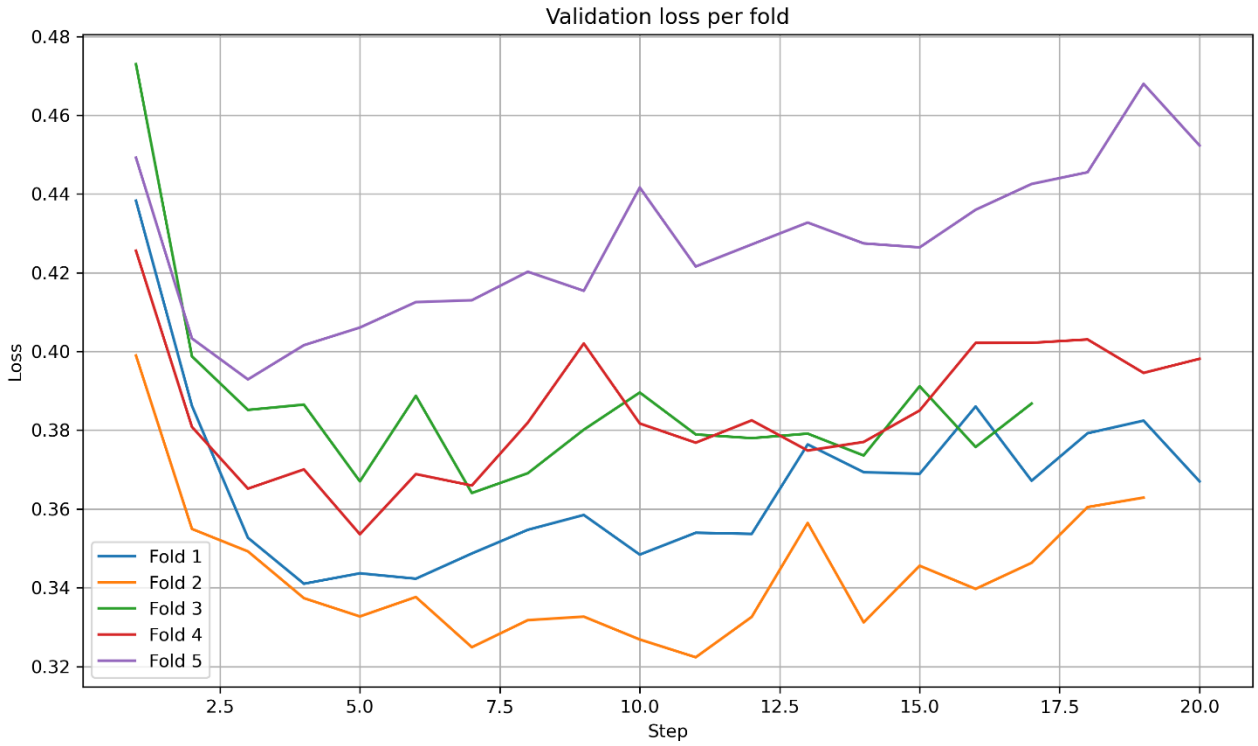

Figure 4. Validation Loss per fold.

8.2. Pixel-Wise Accuracies and Validation Mean IoUs

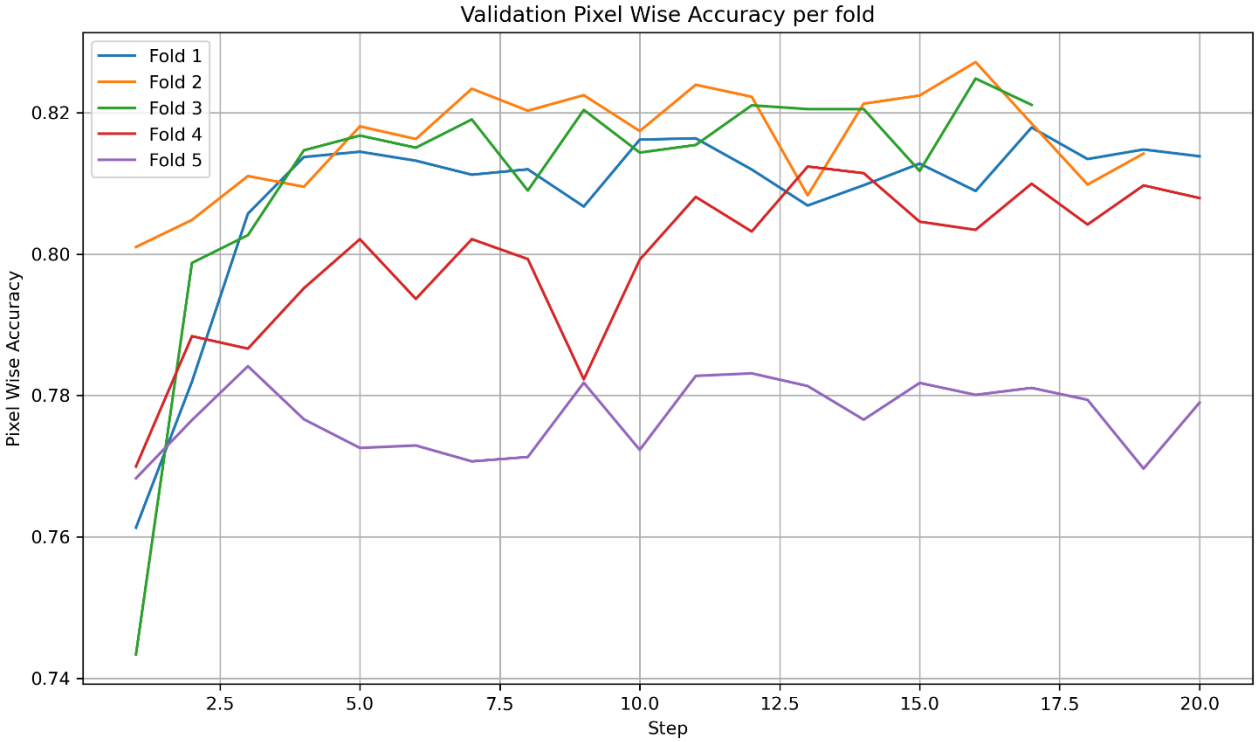

Figure 5. Validation Pixel Accuracy per fold.

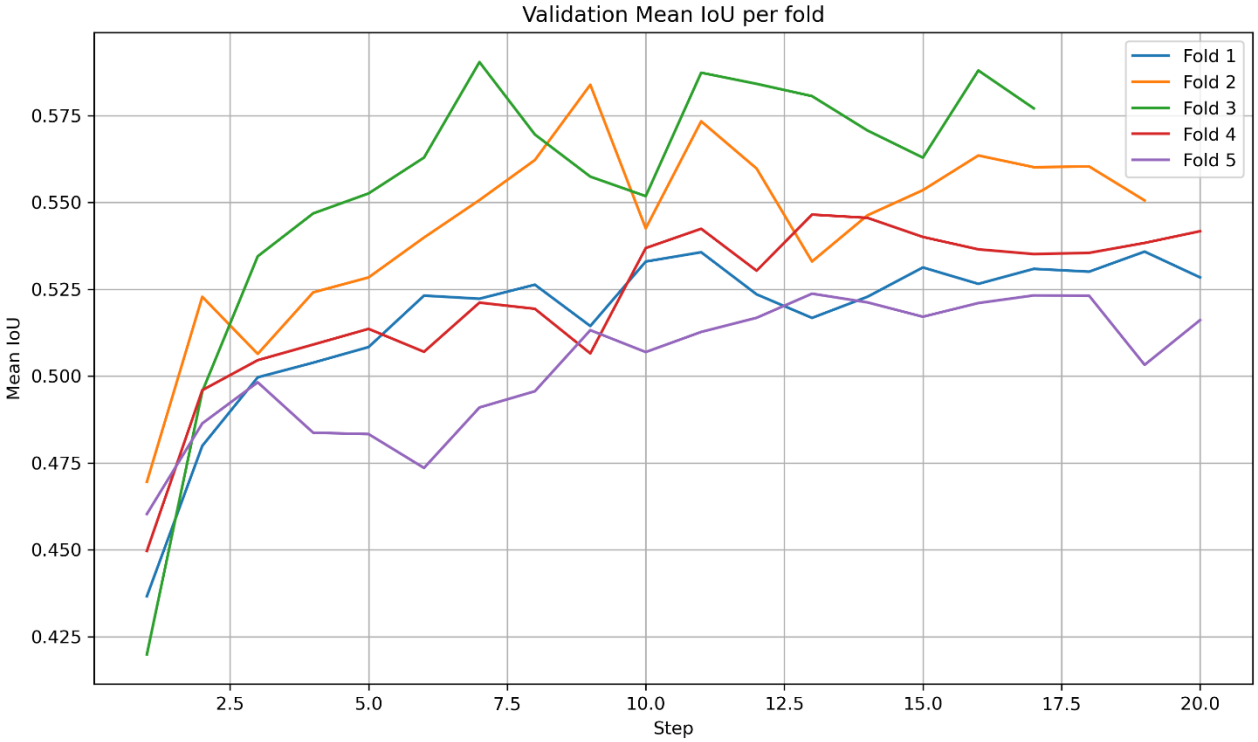

Figure 6. Validation Mean Intersection over Union per fold.

8.3. IoUs per Classes

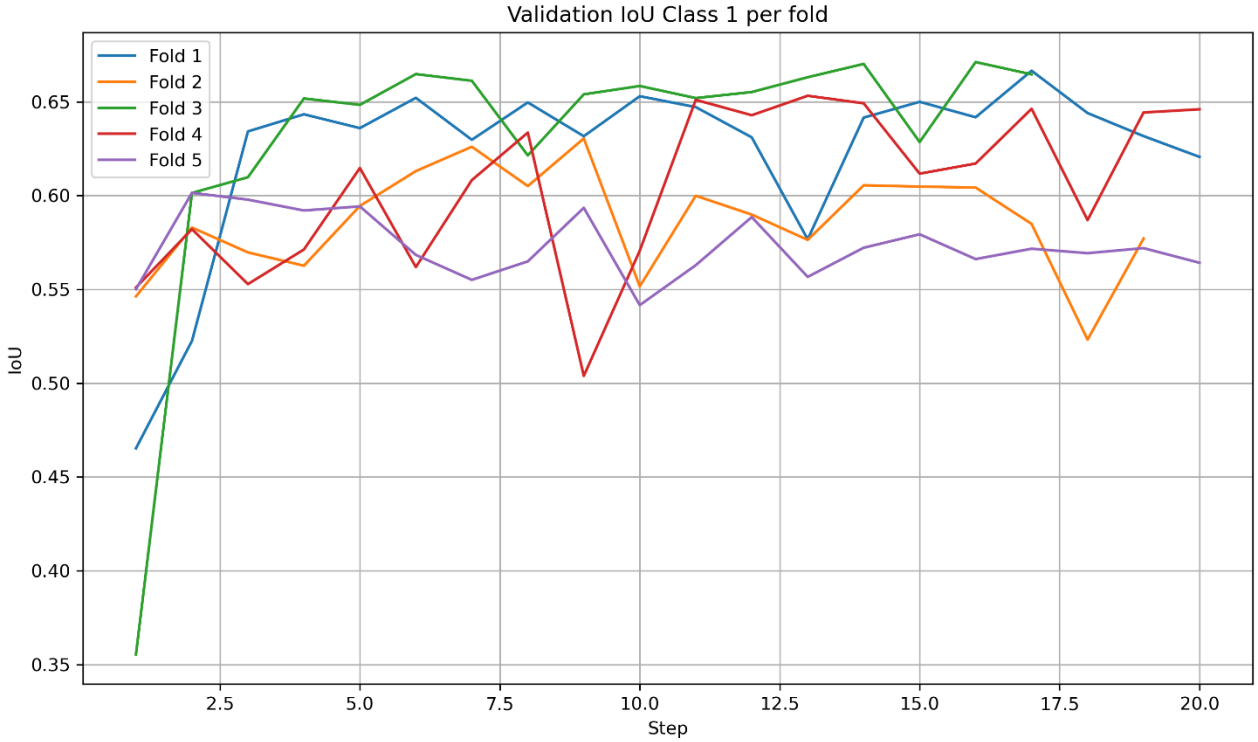

Figure 7. Intersection over Union Class 1 per fold.

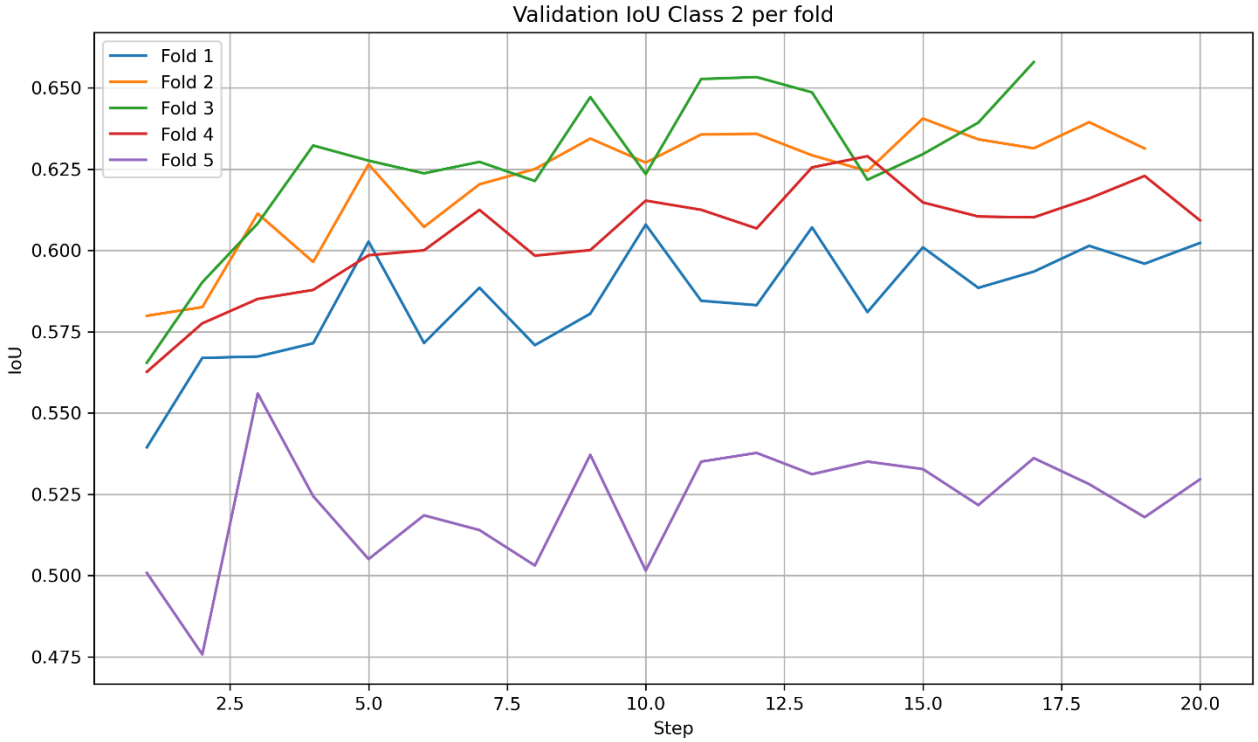

Figure 8. Intersection over Union Class 2 per fold.

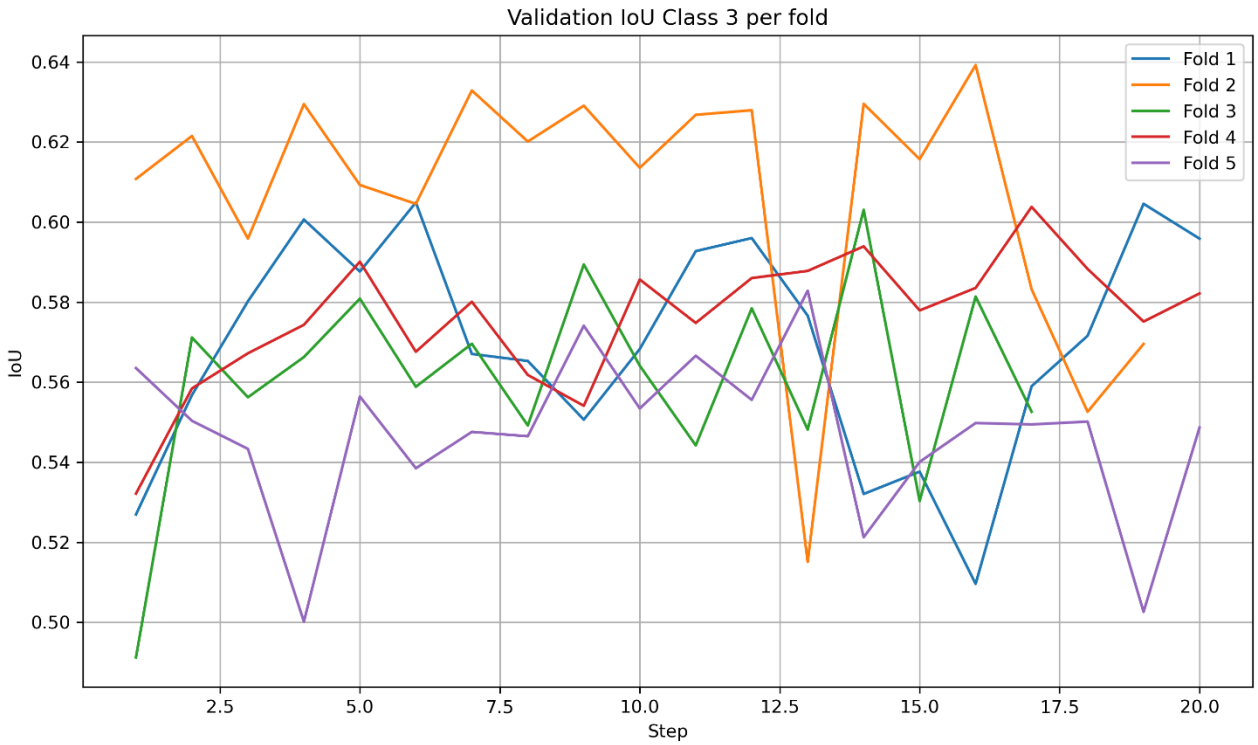

Figure 9. Intersection over Union Class 3 per fold.

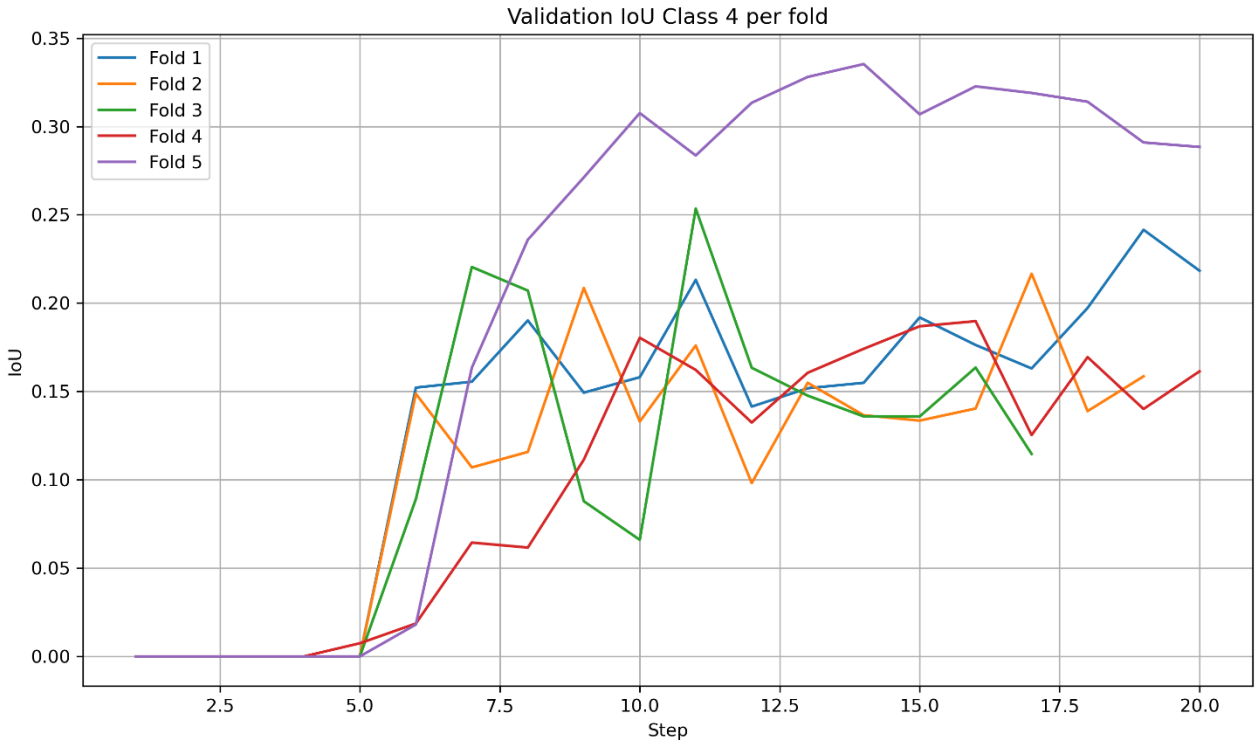

Figure 10. Intersection over Union Class 4 per fold.

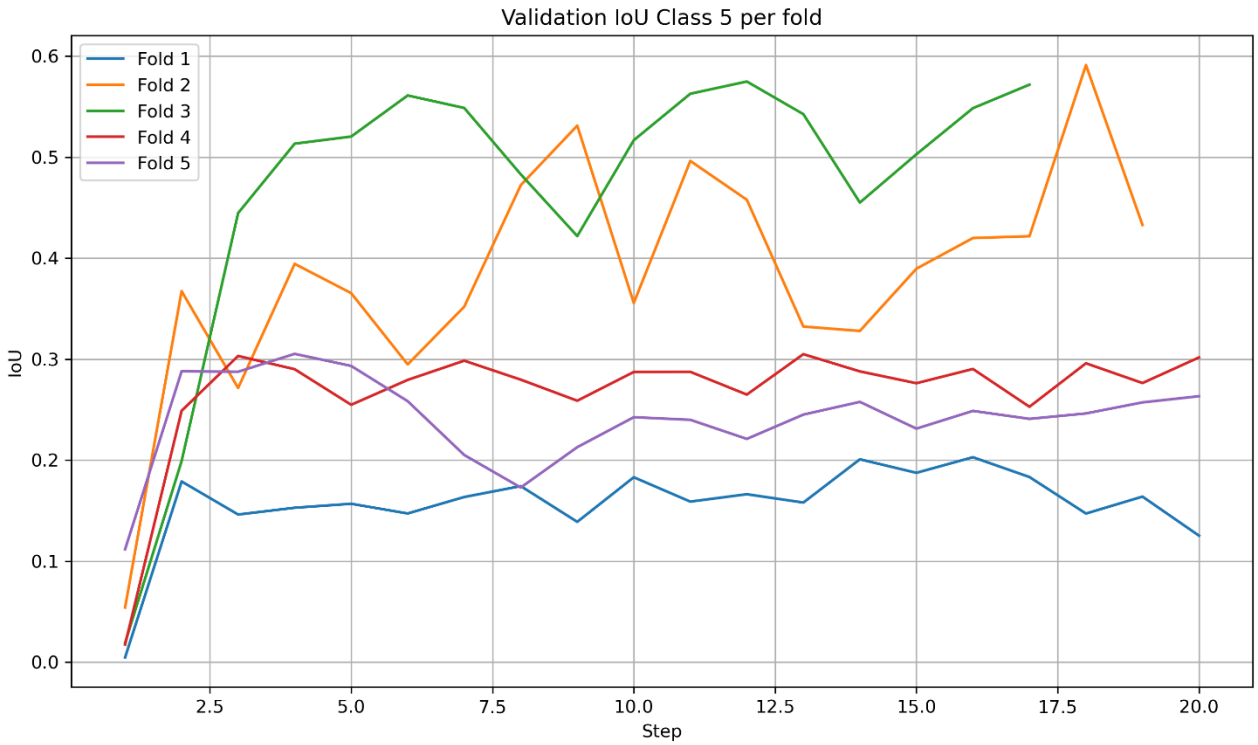

Figure 11. Intersection over Union Class 5 per fold.

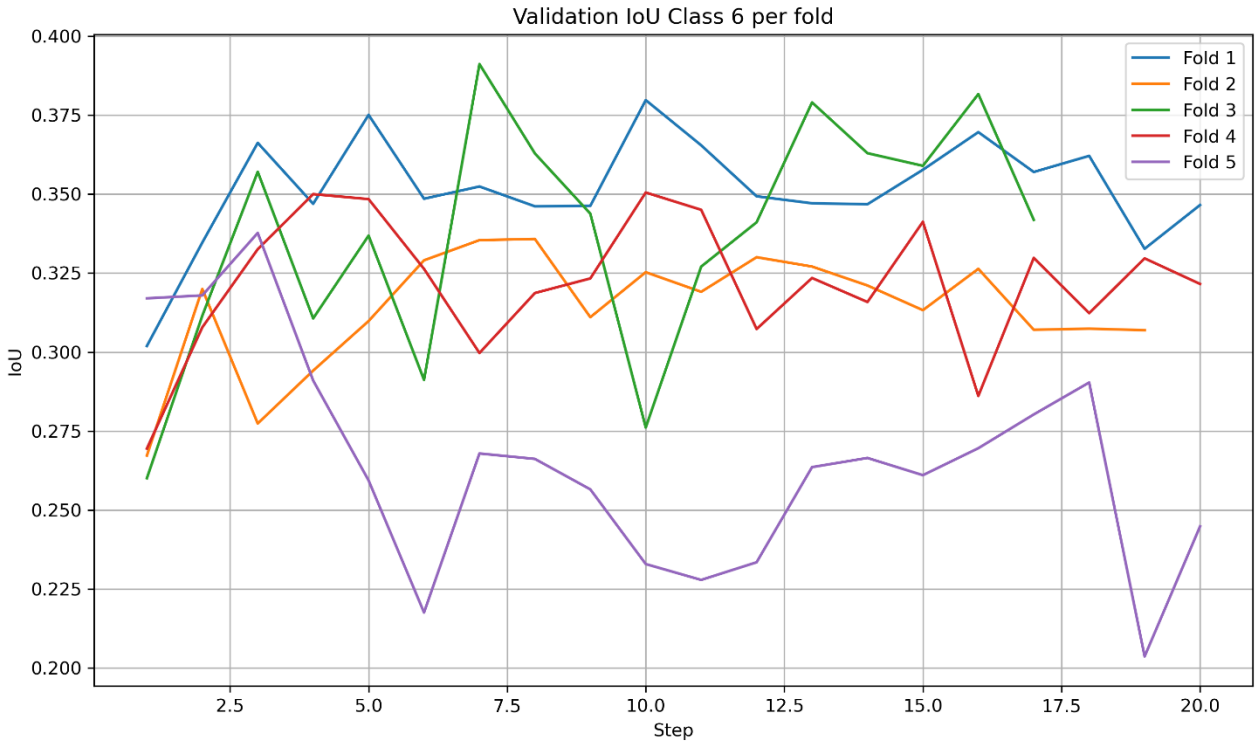

Figure 12. Intersection over Union Class 6 per fold.

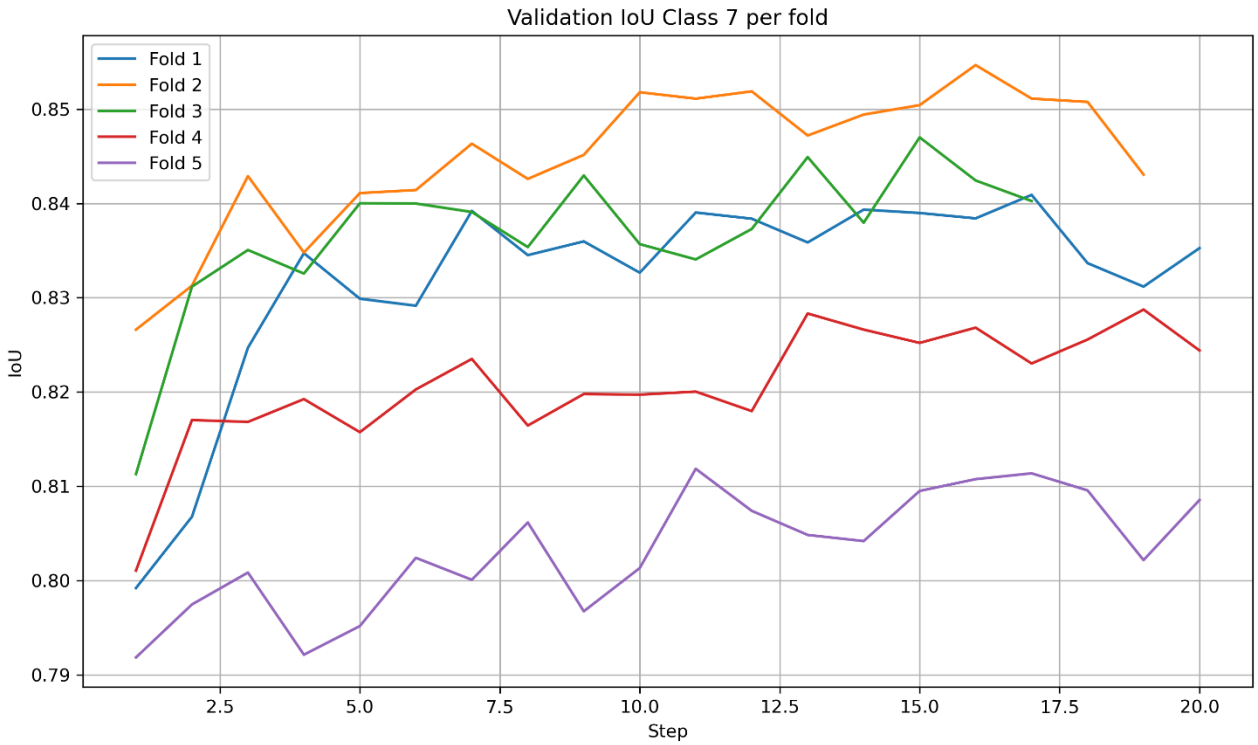

Figure 13. Intersection over Union Class 7 per fold.

8.4. Considerations

The validation loss curve exhibits significant oscillations, indicating some instability in the model's generalization process. This behavior can be partly attributed to a relatively high learning rate, which makes the optimization more sensitive to local gradient variations, and to the use of a small batch size, which introduces higher variance in the gradient estimates. Both factors may contribute to a less stable training process, with visible effects on the validation metrics.

The performance metrics obtained across the different folds show similar behavior, indicating consistency and reproducibility in the training process. The average Intersection over Union (IoU) remains relatively stable, fluctuating around a value of 0.58. The oscillations observed in the metric curves can be attributed to the specific combined loss function used, which incorporates both Focal Loss and Dice Loss. This design choice aims to penalize the more frequent classes more heavily, promoting the recognition of rarer and imbalanced ones. However, such an approach may introduce greater variability during optimization, resulting in less regular learning curves.

All IoU metrics do not take class 0 (background) into account.

9. Conclusions

The objective of this project was to develop an efficient semantic segmentation system for processing images acquired onboard autonomous vehicles operating in rural environments. The entire work was carried out under strict computational constraints, with a particular focus on compatibility with embedded devices featuring limited memory resources.

By adopting the DeepLabV3 architecture with a MobileNetV3 Large backbone, it was possible to achieve a solid trade-off between accuracy, computational efficiency, and GPU memory usage. The training strategy included data augmentation techniques and the use of a combined loss function (Dice Loss + Focal Loss), designed to enhance the model's ability to recognize underrepresented classes, at the cost of introducing greater variability in the validation curves.

The use of 5-fold cross-validation allowed for a more robust evaluation of the model's generalization ability, showing good consistency across the different folds. The average Intersection over Union (IoU), which remained stable around 0.55, confirms the validity of the approach in terms of reproducibility and reliability.

It should be noted that, after having carried out some experiments, reducing the number of samples in the dataset, eliminating those considered to be incorrectly classified, the performance improved. In particular, by eliminating 63 bad samples and then training the model with 868 starting samples, the performance relative to the average IoU on validation settled around 0.6, showing a slight improvement.

In conclusion, the developed model represents a practical and effective solution for the intended application, demonstrating strong adaptability even in real-world operating conditions. Nevertheless, further improvements are possible, for instance by optimizing the loss function weights, exploring even lighter architectures, or integrating post-processing techniques to further refine the segmentation masks.