Application of Improved Light-weight Feedback Attention Networks for Multi-Class Segmentation in Urban Scenes

Jiaxi Yang
ECE-Department (MEng)
Concordia University
Montreal, Canada
jiaxi.yang@mail.concordia.ca

Yuhang Chen
ECE-Department (MEng)
Concordia University
Montreal, Canada
yuhang.chen@mail.concordia.ca

Qian Zhang
ECE-Department (MEng)
Concordia University
Montreal, Canada
qian.zhang@mail.concordia.ca

Abstract

Image segmentation is one of the most important techniques for computer vision. It is not only the foundation for biomedical imaging, but also essential in urban planning and autonomous driving. In this project, we extend the Feedback Attention Network (FANet), originally designed for precise biomedical segmentation, to address the complex demands of urban scenes through the Cityscapes dataset. The original FANet was designed for binary segmentation and applying it to multi-class segmentation is a non-trivial task. To address this challenge, we propose a new competitive layer for inference. This enhancement, coupled with model light-weighting and the incorporation of spatial and channel squeeze and excitation (scSE) modules, not only reduces the computational complexity but also maintains or even surpasses the original model performance. Through rigorous testing and evaluation, the modified FANet demonstrates robust adaptability and strong performance across diverse urban environments. The results highlight the model's versatility, confirming its extended applicability beyond its original medical domain and its potential as a powerful tool for a broad range of image segmentation tasks.

Keywords

Feedback Attention Network, FANet, multi-class segmentation, urban scenes, model light-weighting

I. Introduction

In the rapidly evolving field of computer vision, image segmentation is essential for interpreting and processing visual data across various applications. From detailed medical diagnostics to complex urban scene analysis, the ability to precisely delineate objects against diverse backgrounds is critical. Among the array of technologies developed for image segmentation, deep learning frameworks, particularly the Feedback Attention Network (FANet), have achieved notable advancements. Initially designed for binary segmentation of medical images, FANet effectively utilized feedback mechanisms across training epochs to set new benchmarks for segmentation precision.

However, FANet's potential is not confined to medical imaging alone. As image capture methods become increasingly complex, leading to an explosion of data complexity, there is a pressing need to adapt and optimize such deep learning frameworks for new domains. This paper discusses the expansion of FANet's capabilities from its traditional role in medical image segmentation to address multi-class segmentation challenges in urban environments. To facilitate this transition, we have introduced several methodological enhancements aimed at decoding the intricate structure of urban scenes, thereby bridging the gap between the precise needs of biomedical imaging and the varied components of natural scenes such as vehicles, pedestrians, and roadways.

The adaptation includes integrating a competitive layer during the prediction phase to effectively distinguish between multiple categories such as vehicles, pedestrians, and roadways, enhancing the network's ability to manage complex urban scenes. Additionally, to address the challenges of model light-weighting which is crucial for processing efficiency. Meanwhile, we replaced the original SE (Squeeze-and-Excitation) modules with scSE (concurrent spatial and channel squeeze and excitation) modules. This enhancement enhances the accuracy of the model amidst reductions in its complexity.

These improvements ensure that FANet's effectiveness extends beyond the confines of medical imaging to meet the multi-faceted challenges presented by our selected urban dataset. Through rigorous experimental simulation and analysis, we validate the effectiveness of our improved FANet approach and its implications for urban scene applications. This exploration underscores FANet's adaptability and marks a significant step towards its evolution into a versatile framework capable of tackling the next generation of image segmentation challenges.

II. Problem Statement

The pursuit of advanced image segmentation technologies has predominantly focused on enhancing precision and accuracy within well-defined, controlled imaging environments. In the field of biomedical imaging, success largely hinges on the stability and predictability of medical images. However, moving beyond the structured realm of biomedical imaging to the more complex and unpredictable urban scene imaging introduces a series of unique and uncharted challenges.

In the processing of urban scenes, variations in environmental conditions such as lighting and weather, along with the presence of a higher density of overlapping objects—such as vehicles, pedestrians, and diverse types of roadways—introduce complexities not typically encountered in medical image segmentation. Consequently, the problem at hand is twofold. First, there is a need for a segmentation model that is not only sensitive to the subtleties of urban imagery, particularly to overlapping areas, but also robust against its inherent variabilities. Second, and more critically, this model must adaptively refine its learning process, utilizing iterative feedback to enhance its precision with each new data exposure.

To address these challenges, we have expanded FANet's capabilities by integrating a competitive layer during the prediction phase. This addition facilitates multi-class segmentation by enabling the network to effectively distinguish between categories such as vehicles, pedestrians, and roadways, thus enhancing the model's ability to manage the complexities of urban scenes. Moreover, to maintain high performance amidst the increased demand on computational resources, the network has undergone light-weighting modifications. We have strategically reduced the complexity of certain network modules without compromising the overall segmentation quality.

Additionally, to address the potential loss of accuracy due to the model's light-weighting, the original SE (Squeeze-and-Excitation) modules were replaced with more advanced scSE (spatial and channel squeeze and excitation) modules. These modules are specifically designed to enhance feature recalibration capabilities both spatially and across channels, ensuring that segmentation accuracy is maintained or even improved despite the model's reduced scale.

This necessitates a reevaluation of the existing feedback mechanisms and attention modules, originally designed for the homogeneity of medical datasets, repurposing them to accommodate the heterogeneity of urban scenes. As FANet transitions to this new application, assessing its performance and iteratively refining its architecture to ensure robust segmentation across multiple object classes in complex urban environments is crucial.

This paper will elaborate on the enhancements required for FANet to tackle these emerging challenges. Subsequent sections will detail the specific issues within the selected dataset, describe the adaptations to the network, and present an analysis of the outcomes to demonstrate how the revised FANet architecture rises to the complex task of multi-domain image segmentation.

III. Literature Review

The field of image segmentation has witnessed significant advancements through the integration of deep learning techniques, specifically in adapting to complex imaging environments beyond traditional medical settings. Here, we highlight image segmentation for urban scenes, along with key developments and methodological innovations that have influenced current practice.

A. Image Segmentation Techniques for Urban Environments

The adaptation of segmentation models to urban environments has necessitated handling a higher degree of variability and unpredictability in scene composition. One seminal work in this area is by Cordts et al. (2016)[1], who demonstrated the use of Convolutional Neural Networks (CNNs) for semantic understanding of urban scenes. Their research highlighted the ability of CNNs to accurately recognize and categorize critical urban elements like roads, vehicles, and pedestrians, even in densely populated settings.

B. Advancements in Deep Learning for Urban Scene Analysis

Deep learning frameworks have continuously evolved to address the dynamic aspects of urban environments. Neuhold et al. (2017)[2] extended the capabilities of deep learning in urban scene segmentation by introducing a method that leverages large-scale video data to improve the temporal consistency and accuracy of segmentation models. This approach is particularly beneficial for applications such as autonomous driving, where understanding the temporal dynamics of urban scenes is crucial.

C. Utilization of Attention Mechanisms in Urban Scene Segmentation

The application of attention mechanisms in image segmentation has provided significant improvements in the discriminative capabilities of models focused on urban scenes. A noteworthy contribution was made by Zhao et al. (2017)[3], who incorporated pyramid scene parsing networks that apply attention mechanisms to aggregate context from different scales, thus enhancing the segmentation of complex urban landscapes. This methodology has improved the accuracy and efficiency of segmenting detailed urban components.

D. Context-Aware Segmentation Approaches

Recent advancements have also focused on context-aware segmentation techniques that adapt to the specific challenges presented by urban environments. For instance, Zhang et al. (2018)[4] developed a context encoding module that effectively captures contextual relationships between different urban elements, significantly boosting the performance of segmentation tasks in heterogeneous urban settings.

IV. Problem formulation

Modify the original FANet approach to suit the complexities of the new datasets, including changes in input data characteristics and expected segmentation outcomes. This section involves multiple components and will outline the theoretical modifications proposed for adapting FANet to handle diverse data types effectively:

A. Dataset Replacement

To adapt the FANet model, originally utilized in the domain of medical imaging for binary classification tasks, to the new and complex urban scenarios presented by the Cityscapes dataset, it was essential to modify the network structure. This adaptation involved introducing a competitive layer during the prediction phase of the model, transitioning from binary to multi-class segmentation. This adjustment enables effective categorization of various urban elements such as vehicles, pedestrians, and roadways. Such an adaptation necessitates a comprehensive reevaluation of the network architecture to cope with the increasing complexity and diversity of urban scenes. This step ensures that the model can efficiently manage and accurately process the details of city landscapes.

B. Introduction of a Competitive Layer

To facilitate the multi-class segmentation, a competitive layer has been integrated during the prediction phase. This layer functions to discern and classify multiple overlapping urban elements, enhancing the network's ability to differentiate between closely situated and often overlapping objects in urban environments.

C. Model Light-weighting

Given the computational intensity associated with processing high-resolution urban imagery, reducing the model's size and complexity was imperative. This light-weighting was achieved by scaling down certain network modules, thus decreasing the computational demand and improving the model's responsiveness and usability in resource-constrained environments.

D. Accuracy Preservation Amidst Model Reduction

A critical aspect of the model adaptation was to ensure that the reduction in complexity did not compromise the segmentation accuracy. To address the potential loss of accuracy due to model light-weighting, the original SE (Squeeze-and-Excitation) modules were replaced with scSE (concurrent spatial and channel squeeze and excitation) modules. These modules are designed to enhance feature recalibration capabilities both spatially and per channel, thereby maintaining, and sometimes even improving, segmentation accuracy despite the reduced model size.

E. Optimization Goal

The overarching goal is to develop an optimized version of FANet that not only addresses the increased complexity and class variability found in the Cityscapes dataset but also operates efficiently within the constraints of reduced computational resources. The objective is to achieve this without sacrificing the accuracy required for reliable urban scene analysis.

V. Analysis

A. Original FANet Model: Principles and Application

The original Feedback Attention Network (FANet) is designed for high-precision biomedical image segmentation, leveraging the principles of deep learning to enhance segmentation accuracy. The core mechanism of FANet centers on feedback loops that utilize errors from previous predictions to refine future outputs. This innovative approach integrates attention mechanisms to focus computational resources on critical areas of the image that are prone to segmentation errors. The specific network architecture is as follows:

•**SE-Residual Block:** The SE-Residual Block(Fig. 1), central to FANet's architecture, consists of two 3x3 convolutions, each followed by batch normalization (BN) and ReLU activation. An identity mapping links the input and output of these convolutions, helping mitigate the effects of vanishing or exploding gradients common in deep networks.

Adopting techniques from Hu et al.[4], the SE layer within the block dynamically recalibrates channel-wise feature responses using a global average pooling that compresses feature maps into a channel descriptor. This descriptor is then processed through a neural network that scales feature channels, enhancing relevant features and suppressing less significant ones.

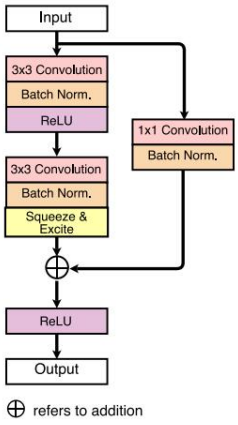


Fig. 1: SE-Residual Block

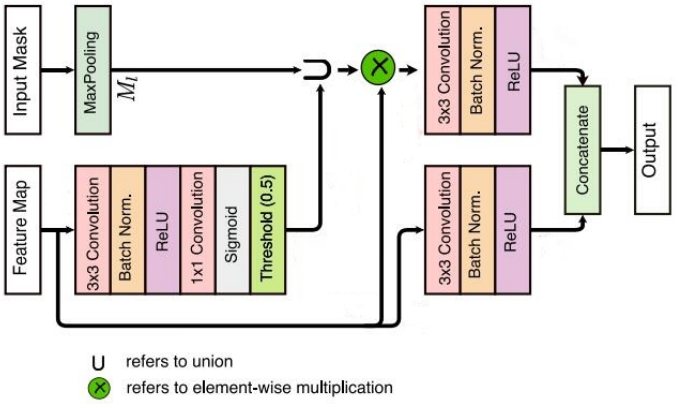


Fig. 2: MixPool Block

•MixPool Block: The MixPool block(Fig. 2) in FANet introduces hard attention via binary masks that selectively process features, focusing computational efforts on crucial attributes while ignoring the irrelevant. This selective enhancement and suppression optimize computational efficiency and enhance the model's interpretability.

•FANet Encoder-Decoder Structure: FANet employs an encoder-decoder framework(Fig. 3), augmented with SE-residual and MixPool blocks, to capture deep semantic information and reconstruct accurate segmentation maps. The architecture features recurrent learning mechanisms that allow continuous refinement of predictions, leveraging feedback from prior outputs to improve future segmentation tasks. This streamlined architecture highlights FANet's ability to dynamically learn and adapt to complex segmentation challenges, offering a marked improvement over traditional methods by incorporating iterative refinement and targeted attention mechanisms.

FANet employs a convolutional neural network (CNN) architecture enhanced with attention modules, which help to emphasize salient features while suppressing less relevant information. The feedback mechanism is a distinctive feature that differentiates FANet from other segmentation models, allowing for continuous improvement of the segmentation results as the network learns from its previous predictions.

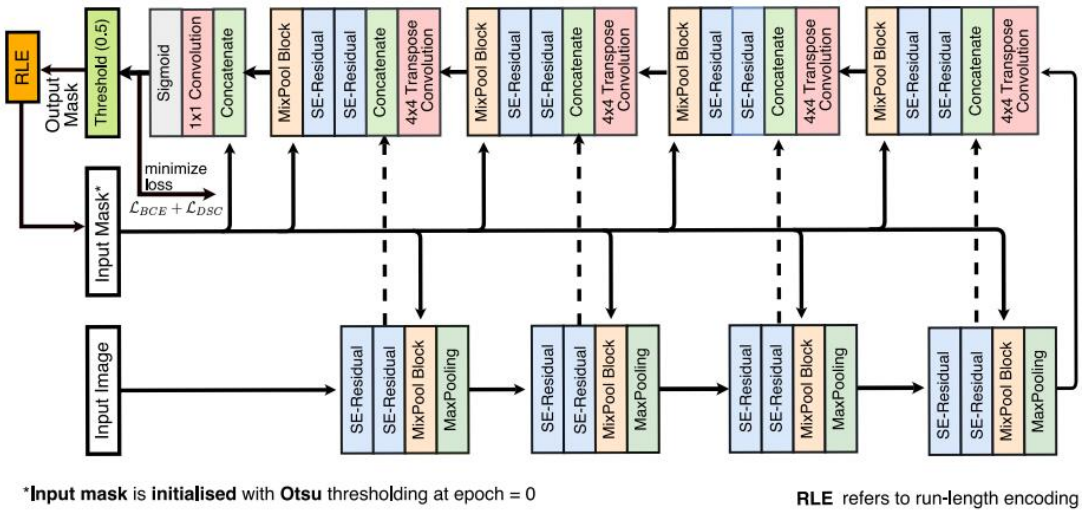


Fig. 3: FANet Framework

B. Improved FANet Model with New Dataset "Cityscapes"

1) Dataset Description: In this study, we use a publicly available dataset named Cityscapes applied in the field of autonomous driving. Cityscapes is a semantic understanding image dataset about urban street scenes. It mainly contains street scenes from 50 different cities and has 3475 high-quality pixel-level annotated images of driving scenes in urban environments. The resolution size of the dataset images is 2048x1024. We have divided the dataset into 3127 and 348 images. Among them, 3127 images

are used as the training set, and the rest are used as the verification set and the test set. The samples of the Cityscapes Dataset are shown in Figure 4 and 5. In order to adapt the dataset to the network structure of FANet, we have segmented the dataset with a total of four categories labeled as vehicles, pedestrians, roads, and other backgrounds, which are shown in Figure 6.

Fig. 4: The samples of Cityscapes Dataset

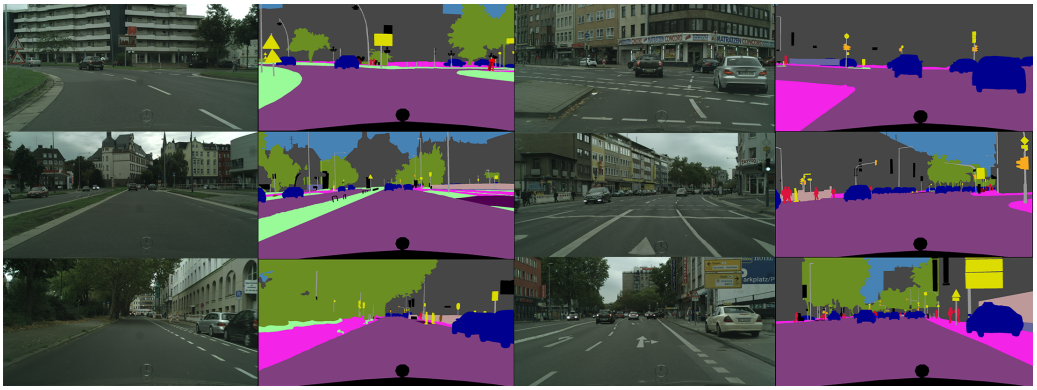


Fig. 5: The samples of labels

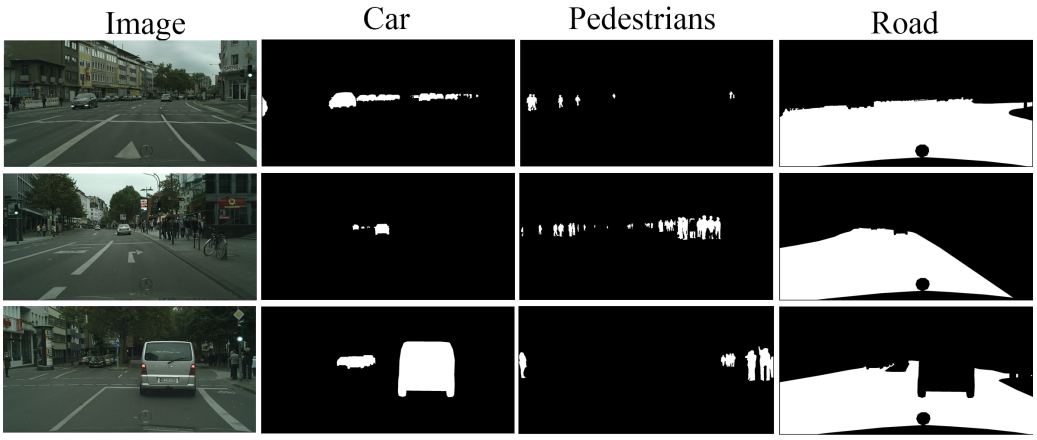


Fig. 6: The segment of labels

2) Adaptation to Cityscapes Dataset by Introducing a Competitive Layer: The transition to the Cityscapes dataset, which captures complex urban scenes, required significant modifications to the original FANet model. The primary challenge was the adaptation from a binary classification system, which segments images into foreground and background, to a multi-class system capable of identifying multiple distinct categories such as vehicles, pedestrians, and various types of road infrastructure.

To address this challenge, a competitive layer was introduced in the prediction phase of the network. This layer operates by evaluating the activations corresponding to different class predictions and enforcing competition among them. This mechanism ensures that the most relevant features for each class are enhanced while others are suppressed, making it particularly effective for scenes with overlapping objects where clear delineation is crucial. In the experiments of this paper, we use different labels to train the network separately, which can create different weights. Then during testing, we use the different models generated during training to extract semantic information from a single input image. In this step, we get several different feature maps. Finally, we add this data to the competitive layer, which finds the most probable category in each pixel and finally outputs a multi-categorized semantic segmented image

3) Model Light-weighting: Adapting FANet for the complex Cityscapes dataset required not only an expansion of capabilities to handle multi-class segmentation but also necessitated significant modifications to reduce the network's size and complexity. This process, known as model light-weighting, is crucial for enhancing the network's operational efficiency, particularly in resource-constrained environments.

The lightweight of the FANet architecture involved the selective reduction of network modules and layer dimensions, which effectively reduced the computational load while maintaining essential performance capabilities. This approach ensures quicker response times and a more user-friendly experience, as the streamlined model demands fewer computational resources. Here we reduce the dimensions of the two corresponding groups of network modules and layers, as shown in Fig 9

4) Optimization with Attention Mechanism Modules: Inspired by the Spatial and Channel Squeeze & Excitation Attention Module[6], we propose a novel residual block named scSE-Residual. After we lightweight the network structure, the result of the accuracy of image segmentation will be decreased to some extent. Therefore we used Spatial and Channel Squeeze & Excitation Attention Block (scSE) to combine with the residual block of the network. This attention block not only possesses less number of parameters, but also enhances the feature extraction ability of the network.The structure of the scSE attention module is shown in Fig.7.

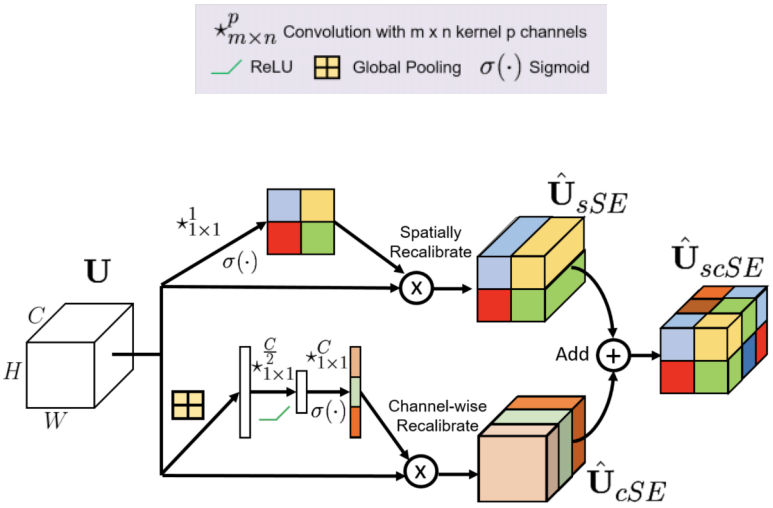


Fig. 7: Spatial and Channel Squeeze Excitation(scSE) Block

As can be seen from the figure, scSE attention is a combination of two SE attention mechanisms. The upper branch of the module is a Channel Squeeze and Spatial Excitation Block (sSE). The lower branch is a Spatial Squeeze and Channel Excitation Block (cSE).In the sSE module, the incoming feature layer U (H×W×C) will be input, where H represents the height of the feature map, H represents the width of the feature map, and C represents the number of channels in the feature map. The input feature map is immediately processed using a 1×1×1 convolutional layer to output a H×W×1 spatial attention map. Finally, activated by a sigmoid function, the weights of each spatial location are generated and the feature layer $\hat{U}_{sSE}$ is output.This processing is shown in expression 1, where f represents the convolution operation, U represents the input feature map, and $kernel$ represents the size of the convolution kernel. The sSE attention module is a spatial attention mechanism that works by focusing on spatial locations in the feature map. By emphasizing important local regions in the image, the spatial

attention mechanism enables the model to better capture critical spatial information, such as features of key parts or specific shape features.

$$\hat{U}_{sSE} = \sigma(f(U, kernel)) \tag{1}$$

The bottom branch of the scSE attention module is a cSE attention module. cSE is a channel attention module whose main role is to re-assign weights to the information content of each channel of the input feature map. By learning the importance of different channels, the model can select the more important feature channels while suppressing those that are not important. This helps the model to focus on those features that are more critical to the task at hand. The feature map undergoes two processes through the cSE attention module, the Squeeze operation and the Excitation operation. First the feature map U goes through the Squeeze operation first, which is a global average pooling of the feature map U to obtain a $1\times1\times$C vector, where the average of each channel represents the global response of that channel.The Squeeze operation is shown by expression 2. where $x_{i,j,c}$ are the values of the c^{th} channel of the feature map at position i,j. Next, the Excitation operation is performed on Z_c obtained after the Squeeze operation to learn the weights of each channel from the global information of that channel using two fully connected layers. In the first fully connected layer, the number of channels is dimensionalized down to $\frac{1}{2}$ of the original number of channels, and then restored to the original number of channels C in the second fully connected layer. This "bottleneck" structure allows the network to extract the most useful information from the features obtained by global average pooling, and then enhance the importance of this critical information by dimensionalizing back to a higher dimensional space. Eventually, the weights of each channel are fixed to 0-1 by a sigmoid activation function, and the Excitation operation is shown by expression 3, where W1 represents the weights of the first full connection and W2 represents the weights of the second full connection.

$$Z_c = \frac{1}{H \times W} \sum_{H}^{i=1} \sum_{W}^{j=1} x_{i,j,c} \tag{2}$$

$$\hat{U}_{cSE} = \sigma(W1, W2) \tag{3}$$

Followed by, the combination of the sSE module with the cSE module through element-wise addition forms the scSE attention mechanism module. scSE attention module not only possesses the advantage of the channel attention mechanism (cSE) for the selection of the important features of each channel, but also possesses the ability of the spatial attention mechanism (sSE) to capture the key spatial information. sSE with the cSE fusion method is shown by expression 4.

$$\hat{U}_{scSE} = \hat{U}_{sSE} + \hat{U}_{cSE} \tag{4}$$

Eventually, the scSE attention mechanism module is combined with the residual block to form our proposed scSE-Residual Block.The scSE-Residual Block's structure is shown in Fig. 8. The network structure diagram of scSE-Residual Block introduced after lightweight improvement of FANet is shown in Fig. 9.

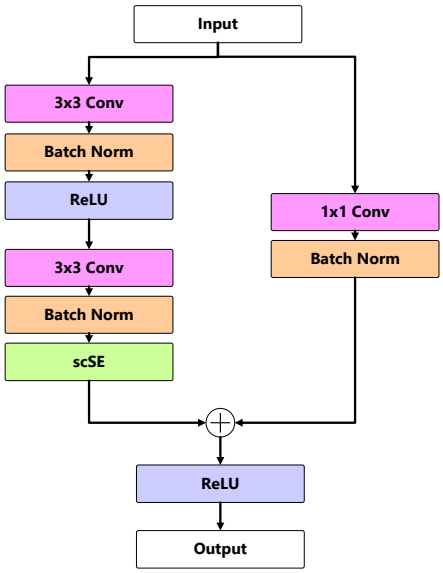


Fig. 8: Structure of scSE-Residual

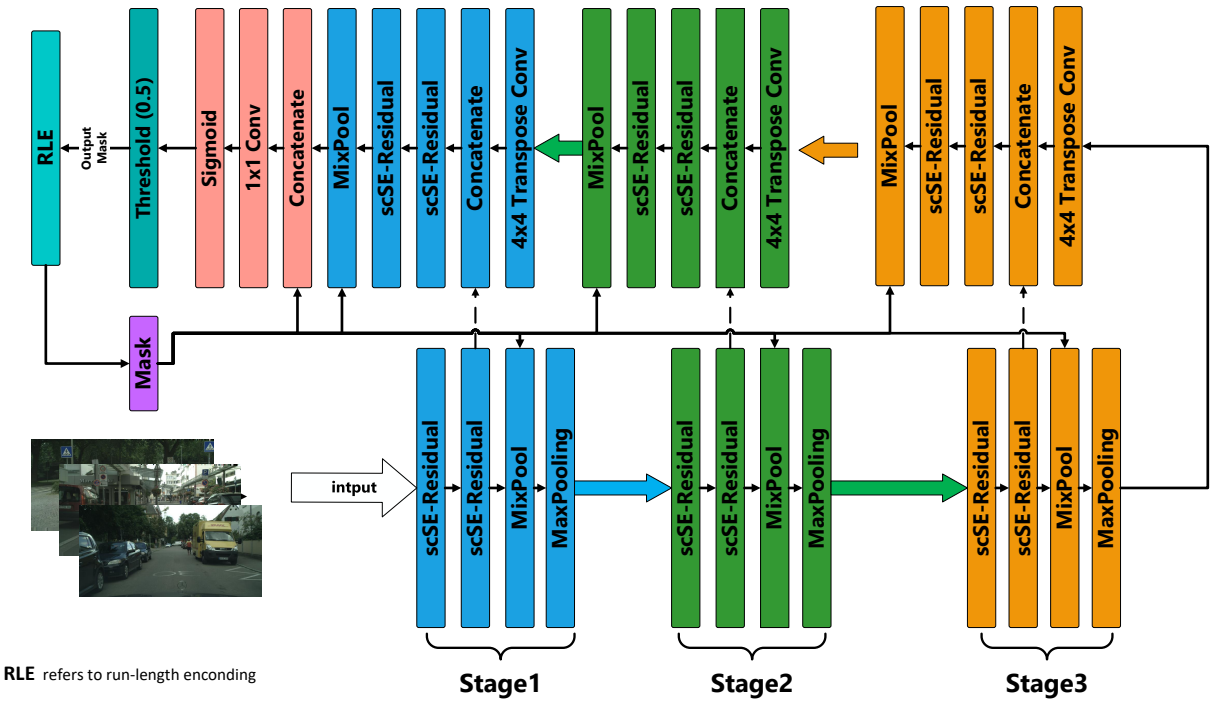


Fig. 9: Structure of Lightweight FANet With Attention Mechenism

As shown in the Fig. 9, we replaced the SE-Residual Block with scSE-Residual Block, and we reduced upsampling and downsampling block to three block respectively.

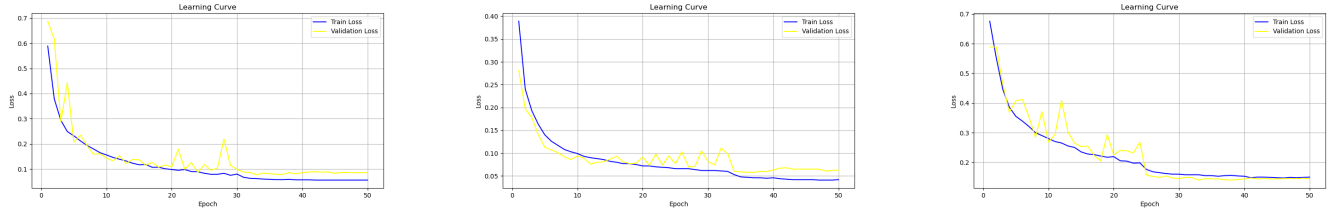
VI. Experimental Results

A. Implementation Details

All the training is performed on a T4 GPU using the PyTorch 2.2.1 framework. For test inference, we have used an NVIDIA RTX 4060 GPU for our method. Our model is trained for 50 epochs (empirically set) using an Adam optimizer with a learning rate of $1 * \exp^{-3}$ for all the experiments, batch size was set as 12.

B. Results and Analysis

As shown in Figs. 10, the three categories of vehicles, people, and roads were trained using 50 epochs to achieve the fitted state. As shown in Table 1, the model size of our proposed method is reduced to 25% of the size of the original method.



(a) Car (b) Road (c) Pedestrian

Fig. 10: Learning curves for vehicles (a), roads (b) and people (c)

Although we have lightened the network, which can lead to a decrease in accuracy, we have added a residual module with a scSE attention mechanism to compensate for this significant decrease in accuracy. The accuracy results of our approach in

semantic segmentation of vehicles and roads are almost close. The results in pedestrian category segmentation are better than the original method, Jaccard value rises from 0.4528 to 0.5413, F1 value rises from 0.5533 to 0.6495, Recall rises from 0.572 to 0.6709, Precision value rises by 11.32% to 77.6%. f2 value rises from 0.549 to 0.645. mIoU value went up by 0.0452 to 0.7292.

Methods	Category	Jaccard	F1	Recall	Specificity	Accuracy	F2	mIoU	Size(MB)
Original	car	0.0263	0.0322	**0.1682**	0.9046	0.915596	0.037733	**0.4614**	30
Light-weighting	car	0.0246	0.029	0.1641	0.905	0.905954	0.034053	0.4606	7.6
Light-weighting+scSE	car	**0.0267**	**0.0329**	0.1669	**0.9047**	**0.915748**	**0.038513**	0.4612	7.8
Original	people	0.4528	0.5533	0.572	0.9176	0.995089	0.549075	0.684	30
Light-weighting	people	0.4939	0.6054	**0.6823**	0.9169	0.995032	0.62651	0.7048	7.6
Light-weighting+scSE	people	**0.5413**	**0.6495**	0.6709	**0.9184**	**0.996241**	**0.645008**	**0.7292**	7.8
Original	path	**0.0006**	**0.0012**	**0.1094**	0.6625	0.655846	**0.002224**	0.3286	30
Light-weighting	path	0.0005	0.001	0.1082	**0.6719**	0.644969	0.001973	**0.3326**	7.6
Light-weighting+scSE	path	0.0004	0.0008	0.1088	0.6638	**0.656963**	0.001557	0.3282	7.8

TABLE I: Result

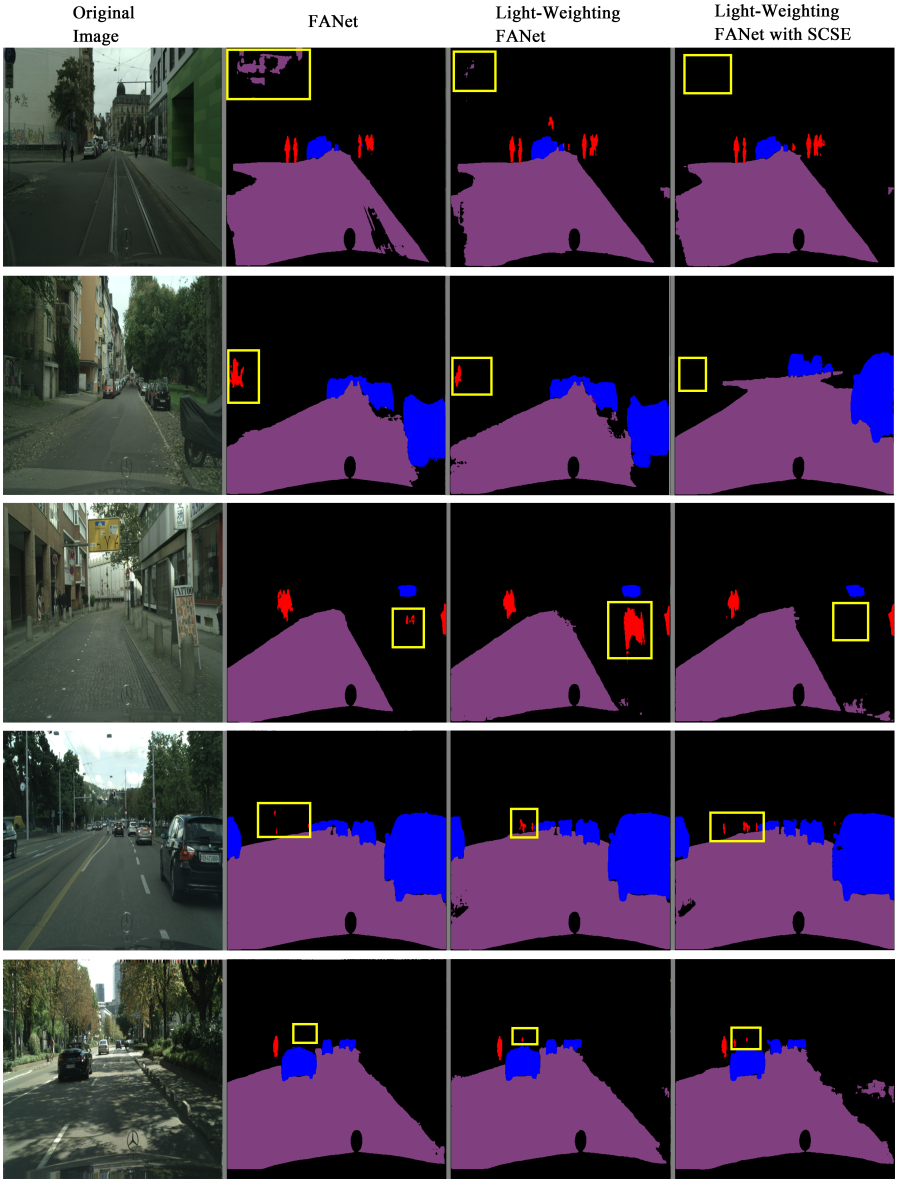


Fig. 11: Visulization Results

It can also be seen from the visualization results in Fig.11 that although our method still has some misclassifications and missed detections. In comparison to the original method, which in the first set of results recognizes buildings as roads as well, the method we have adopted avoids misclassification to a certain extent. In the second set of results, the original method recognizes walls as pedestrians, and our method effectively avoids this misclassification. And in this set of results, the segmentation accuracy of the original method for roads is not as high as that of our proposed method. In the fourth set of experimental results, it is shown that our method can effectively mitigate the high rate of missed detection for pedestrians. In the original method, only one pedestrian can be segmented and the segmentation accuracy is not particularly high, while our method can segment four pedestrians and the accuracy is significantly higher than the original method.

VII. Discussion

These results highlight the potential of applying advanced segmentation techniques, originally developed for medical imaging, to broader applications such as urban scene analysis. The successful adaptation of FANet to urban landscapes underscores the flexibility and scalability of the original architecture.

The integration of scSE modules has proven crucial not only in maintaining segmentation accuracy but often in enhancing it, despite the model's reduced complexity after light-weighting. However, challenges remain in handling edge cases and extremely crowded urban scenarios. Future work will focus on further enhancing the model's adaptability and exploring additional techniques for feature recalibration.

VIII. Conclusion

The adaptation of the Feedback Attention Network (FANet) from its original application in biomedical imaging to the complex dynamics of urban scene segmentation with the Cityscapes dataset marks a substantial advancement in the field of image segmentation. This adaptation not only extended FANet's capabilities to tackle multi-class segmentation in densely populated urban settings but also demonstrated the model's scalability and flexibility. Strategic enhancements, including the integration of a competitive layer and scSE modules, have not only upheld but in many instances improved the segmentation accuracy. Additionally, efforts to lighten the network have significantly increased its suitability for deployment in resource-constrained environments, enhancing its applicability for real-time critical applications such as urban planning and autonomous driving.

While the adaptation of FANet to urban scene segmentation has shown promising results, there are several avenues for further development to enhance its effectiveness and applicability in real-world scenarios. Moving forward, we aim to refine the accuracy and robustness of our segmentation algorithm by integrating more advanced deep learning models, which will better handle the intricate and variable aspects of urban environments. To address the fluctuating conditions encountered in autonomous driving, such as diverse weather and lighting scenarios, we plan to expand our dataset to include these complexities. This expansion will allow for more comprehensive training and testing, ensuring the model's robust performance under varied conditions.

Moreover, we recognize the critical need for real-time processing capabilities in autonomous driving systems. Our ongoing efforts will focus on optimizing the algorithm's speed and efficiency to deliver quick responses essential for the safety and reliability of self-driving vehicles. Ultimately, we will conduct extensive on-road testing to validate the performance of our updated models, ensuring they meet the high safety standards required for deployment in autonomous driving applications.

REFERENCES

[1] M. Cordts et al., "The Cityscapes Dataset for Semantic Urban Scene Understanding," in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 3213–3223. doi: 10.1109/CVPR.2016.350.

[2] G. Neuhold, T. Ollmann, S. R. Bulo, and P. Kontschieder, "The Mapillary Vistas Dataset for Semantic Understanding of Street Scenes," in 2017 IEEE International Conference on Computer Vision (ICCV), Venice: IEEE, Oct. 2017, pp. 5000–5009. doi: 10.1109/ICCV.2017.534.

[3] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid Scene Parsing Network," in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI: IEEE, Jul. 2017, pp. 6230–6239. doi: 10.1109/CVPR.2017.660.

[4] H. Zhang et al., "Context Encoding for Semantic Segmentation," in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA: IEEE, Jun. 2018, pp. 7151–7160. doi: 10.1109/CVPR.2018.00747.

[5] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, "Squeeze-and-Excitation Networks," IEEE Trans. Pattern Anal. Mach. Intell., vol. 42, no. 8, pp. 2011–2023, Aug. 2020, doi: 10.1109/TPAMI.2019.2913372.

[6] A. G. Roy, N. Navab, and C. Wachinger, "Recalibrating Fully Convolutional Networks With Spatial and Channel 'Squeeze and Excitation' Blocks," IEEE Trans. Med. Imaging, vol. 38, no. 2, pp. 540–549, Feb. 2019, doi: 10.1109/TMI.2018.2867261.

[7] N. K. Tomar et al., "FANet: A Feedback Attention Network for Improved Biomedical Image Segmentation," IEEE Trans. Neural Netw. Learning Syst., vol. 34, no. 11, pp. 9375–9388, Nov. 2023, doi: 10.1109/TNNLS.2022.3159394

[8] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA: IEEE, Jun. 2015, pp. 3431–3440. doi: 10.1109/CVPR.2015.7298965.

[9] G. Han et al., "Improved U-Net based insulator image segmentation method based on attention mechanism," Energy Reports, vol. 7, pp. 210–217, Nov. 2021, doi: 10.1016/j.egyr.2021.10.037.

IX. APPENDIX

A. Train:

```
class DATASET(Dataset):
   def __init__(self, images_path, masks_path, size, transform=None):
      super().__init__()

      self.images_path = images_path
      self.masks_path = masks_path
      self.transform = transform
      self.n_samples = len(images_path)

   def __getitem__(self, index):
      """ Image """
      image = cv2.imread(self.images_path[index], cv2.IMREAD_COLOR)
      mask = cv2.imread(self.masks_path[index], cv2.IMREAD_GRAYSCALE)

      if self.transform is not None:
         augmentations = self.transform(image=image, mask=mask)
         image = augmentations["image"]
         mask = augmentations["mask"]

      image = cv2.resize(image, size)
      image = np.transpose(image, (2, 0, 1))
      image = image/255.0
      image = image.astype(np.float32)


      mask = cv2.resize(mask, size)
      mask = np.expand_dims(mask, axis=0)
      mask = mask/255.0
      mask = mask.astype(np.float32)

      return image, mask

   def __len__(self):
      return self.n_samples

def train(model, loader, mask, optimizer, loss_fn, device):
   epoch_loss = 0
   return_mask = []

   model.train()
   for i, (x, y) in enumerate(loader):
      x = x.to(device, dtype=torch.float32)
      y = y.to(device, dtype=torch.float32)

      b, c, h, w = y.shape
      m = []
      for edata in mask[i*b : i*b+b]:
         edata = " ".join(str(d) for d in edata)
         edata = str(edata)
         edata = rle_decode(edata, size)
         edata = np.expand_dims(edata, axis=0)
         m.append(edata)
```

```
      m = np.array(m, dtype=np.int32)
      m = np.transpose(m, (0, 1, 3, 2))
      m = torch.from_numpy(m)
      m = m.to(device, dtype=torch.float32)

      optimizer.zero_grad()
      y_pred = model([x, m])
      loss = loss_fn(y_pred, y)
      loss.backward()
      optimizer.step()

      with torch.no_grad():
         y_pred = torch.sigmoid(y_pred)
         y_pred = y_pred.cpu().numpy()

         for py in y_pred:
            py = np.squeeze(py, axis=0)
            py = py > 0.5
            py = np.array(py, dtype=np.uint8)
            py = rle_encode(py)
            return_mask.append(py)

      epoch_loss += loss.item()

   epoch_loss = epoch_loss/len(loader)
   return epoch_loss, return_mask

def evaluate(model, loader, mask, loss_fn, device):
   epoch_loss = 0
   return_mask = []

   model.eval()
   with torch.no_grad():
      for i, (x, y) in enumerate(loader):
         x = x.to(device)
         y = y.to(device)

         b, c, h, w = y.shape
         m = []
         for edata in mask[i*b : i*b+b]:
            edata = " ".join(str(d) for d in edata)
            edata = str(edata)
            edata = rle_decode(edata, size)
            edata = np.expand_dims(edata, axis=0)
            m.append(edata)

         m = np.array(m)
         m = np.transpose(m, (0, 1, 3, 2))
         m = torch.from_numpy(m)
         m = m.to(device, dtype=torch.float32)


         y_pred = model([x, m])
         loss = loss_fn(y_pred, y)
         epoch_loss += loss.item()
```

```
            y_pred = torch.sigmoid(y_pred)
            y_pred = y_pred.cpu().numpy()

            for py in y_pred:
                py = np.squeeze(py, axis=0)
                py = py > 0.5
                py = np.array(py, dtype=np.uint8)
                py = rle_encode(py)
                return_mask.append(py)

    epoch_loss = epoch_loss/len(loader)
    return epoch_loss, return_mask

if __name__ == "__main__":
    """ Seeding """
    seeding(42)

    """ Directories """
    create_dir("files")

    """ Training logfile """
    train_log_path = "files/train_log.txt"
    if os.path.exists(train_log_path):
        print("Log file exists")
    else:
        train_log = open("files/train_log.txt", "w")
        train_log.write("\n")
        train_log.close()

    """ Record Date & Time """
    datetime_object = str(datetime.datetime.now())
    print_and_save(train_log_path, datetime_object)

    """ Hyperparameters """
    size = (512, 512)
    batch_size = 12
    num_epochs = 50
    lr = 1e-3
    checkpoint_path = "files/checkpoint.pth"

    """ Dataset """
    path = "./VOCdevkit"
    (train_x, train_y), (valid_x, valid_y) = load_data(path)
    train_x, train_y = shuffling(train_x, train_y)

    data_str = f"Dataset Size:\nTrain: {len(train_x)} - Valid: {len(valid_x)}\n"
    print_and_save(train_log_path, data_str)

    """ Data augmentation: Transforms """
    transform = A.Compose([
        A.Rotate(limit=35, p=0.3),
        A.HorizontalFlip(p=0.3),
        A.VerticalFlip(p=0.3),
        A.CoarseDropout(p=0.3, max_holes=10, max_height=32, max_width=32)
    ])
```

```
""" Dataset and loader """
train_dataset = DATASET(train_x, train_y, size, transform=transform)
valid_dataset = DATASET(valid_x, valid_y, size, transform=None)

data_str = f"Dataset Size:\nTrain: {len(train_x)} - Valid: {len(valid_x)}\n"
print_and_save(train_log_path, data_str)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=2
)

valid_loader = DataLoader(
    dataset=valid_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=2
)

""" Model """
device = torch.device('cuda')
model = FANet()
# model.load_state_dict(torch.load(checkpoint_path, map_location=device))
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=lr)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience
    ↪ =5, verbose=True)
loss_fn = DiceBCELoss()
loss_name = "BCE Dice Loss"

data_str = f"Hyperparameters:\nImage Size: {size}\nBatch Size: {batch_size}\nLR: {
    ↪ lr}\nEpochs: {num_epochs}\n"
data_str += f"Optimizer: Adam\nLoss: {loss_name}\n"
print_and_save(train_log_path, data_str)

""" Training the model. """
best_valid_loss = float('inf')
train_mask = init_mask(train_x, size)
valid_mask = init_mask(valid_x, size)

for epoch in range(num_epochs):
    start_time = time.time()

    train_loss, return_train_mask = train(model, train_loader, train_mask, optimizer
        ↪ , loss_fn, device)
    valid_loss, return_valid_mask = evaluate(model, valid_loader, valid_mask,
        ↪ loss_fn, device)
    scheduler.step(valid_loss)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        data_str = f"Saving checkpoint: {checkpoint_path}"
```

```
            print_and_save(train_log_path, data_str)
            torch.save(model.state_dict(), checkpoint_path)

            train_mask = return_train_mask
            valid_mask = return_valid_mask

        end_time = time.time()
        epoch_mins, epoch_secs = epoch_time(start_time, end_time)

        data_str = f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s\n'
        data_str += f'\tTrain Loss: {train_loss:.3f}\n'
        data_str += f'\t Val. Loss: {valid_loss:.3f}\n'
        print_and_save(train_log_path, data_str)
```

B. Block:

```
import torch
import torch.nn as nn

""" Squeeze and Excitation block """
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)
class sSE(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.Conv1x1 = nn.Conv2d(in_channels, 1, kernel_size=1, bias=False)
        self.norm = nn.Sigmoid()
    def forward(self, U):
        q = self.Conv1x1(U)
        q = self.norm(q)
        return U *q
class cSE(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.Conv_Squeeze = nn.Conv2d(in_channels,
                                      in_channels // 2,
                                      kernel_size=1,
                                      bias=False)
        self.Conv_Excitation = nn.Conv2d(in_channels // 2,
                                         in_channels,
                                         kernel_size=1,
```

```
                                bias=False)
        self.norm = nn.Sigmoid()

    def forward(self, U):
        z = self.avgpool(U) # shape: [bs, c, h, w] to [bs, c, 1, 1]
        z = self.Conv_Squeeze(z) # shape: [bs, c/2, 1, 1]
        z = self.Conv_Excitation(z) # shape: [bs, c, 1, 1]
        z = self.norm(z)
        return U * z.expand_as(U)
class scSE(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.cSE = cSE(in_channels)
        self.sSE = sSE(in_channels)
    def forward(self, U):
        U_sse = self.sSE(U)
        U_cse = self.cSE(U)
        return U_cse+U_sse
class new_ResidualBlock(nn.Module):
    def __init__(self, in_c, out_c):
        super(new_ResidualBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_c, out_c, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_c)

        self.conv2 = nn.Conv2d(out_c, out_c, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_c)

        self.conv3 = nn.Conv2d(in_c, out_c, kernel_size=1, padding=0)
        self.bn3 = nn.BatchNorm2d(out_c)

        self.se = scSE(out_c)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x1 = self.conv1(x)
        x1 = self.bn1(x1)
        x1 = self.relu(x1)

        x2 = self.conv2(x1)
        x2 = self.bn2(x2)

        x3 = self.conv3(x)
        x3 = self.bn3(x3)
        x3 = self.se(x3)

        x4 = x2 + x3
        x4 = self.relu(x4)

        return x4
""" 3x3->3x3 Residual block """
class ResidualBlock(nn.Module):
    def __init__(self, in_c, out_c):
        super(ResidualBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_c, out_c, kernel_size=3, padding=1)
```

```
        self.bn1 = nn.BatchNorm2d(out_c)

        self.conv2 = nn.Conv2d(out_c, out_c, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_c)

        self.conv3 = nn.Conv2d(in_c, out_c, kernel_size=1, padding=0)
        self.bn3 = nn.BatchNorm2d(out_c)

        self.se = SELayer(out_c, out_c)
        # self.se = scSE(out_c)#new attention
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x1 = self.conv1(x)
        x1 = self.bn1(x1)
        x1 = self.relu(x1)

        x2 = self.conv2(x1)
        x2 = self.bn2(x2)

        x3 = self.conv3(x)
        x3 = self.bn3(x3)
        x3 = self.se(x3)

        x4 = x2 + x3
        x4 = self.relu(x4)

        return x4

""" Mixpool block: Merging the image features and the mask """
class MixPool(nn.Module):
    def __init__(self, in_c, out_c):
        super(MixPool, self).__init__()

        self.fmask = nn.Sequential(
            nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_c, 1, kernel_size=1, padding=0),
            nn.Sigmoid()
        )

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_c, out_c//2, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_c//2),
            nn.ReLU(inplace=True)
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(in_c, out_c//2, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_c//2),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, m):
        fmask = (self.fmask(x) > 0.5).type(torch.cuda.FloatTensor)
```

```
        m = nn.MaxPool2d((m.shape[2]//x.shape[2], m.shape[3]//x.shape[3]))(m)
        x1 = x * torch.logical_or(fmask, m).type(torch.cuda.FloatTensor)
        x1 = self.conv1(x1)
        x2 = self.conv2(x)
        x = torch.cat([x1, x2], axis=1)
        return x
```

C. Loss:

```
class DiceLoss(nn.Module):
    def __init__(self, weight=None, size_average=True):
        super(DiceLoss, self).__init__()

    def forward(self, inputs, targets, smooth=1):

        #comment out if your model contains a sigmoid or equivalent activation layer
        inputs = torch.sigmoid(inputs)

        #flatten label and prediction tensors
        inputs = inputs.view(-1)
        targets = targets.view(-1)

        intersection = (inputs * targets).sum()
        dice = (2.*intersection + smooth)/(inputs.sum() + targets.sum() + smooth)

        return 1 - dice

class DiceBCELoss(nn.Module):
    def __init__(self, weight=None, size_average=True):
        super(DiceBCELoss, self).__init__()

    def forward(self, inputs, targets, smooth=1):

        #comment out if your model contains a sigmoid or equivalent activation layer
        inputs = torch.sigmoid(inputs)

        #flatten label and prediction tensors
        inputs = inputs.view(-1)
        targets = targets.view(-1)

        intersection = (inputs * targets).sum()
        dice_loss = 1 - (2.*intersection + smooth)/(inputs.sum() + targets.sum() +
            ↪ smooth)
        BCE = F.binary_cross_entropy(inputs, targets, reduction='mean')
        Dice_BCE = (0.5 * BCE) + (0.5 * dice_loss)
        return Dice_BCE

model:
class EncoderBlock(nn.Module):
    def __init__(self, in_c, out_c, name=None):
        super(EncoderBlock, self).__init__()

        self.name = name
        self.r1 = ResidualBlock(in_c, out_c)
        self.r2 = ResidualBlock(out_c, out_c)
        self.p1 = MixPool(out_c, out_c)
```

```
        self.pool = nn.MaxPool2d((2, 2))

    def forward(self, inputs, masks):
        x = self.r1(inputs)
        x = self.r2(x)
        p = self.p1(x, masks)
        o = self.pool(p)
        return o, x

class DecoderBlock(nn.Module):
    def __init__(self, in_c, out_c, name=None):
        super(DecoderBlock, self).__init__()

        self.upsample = nn.ConvTranspose2d(in_c, in_c, kernel_size=4, stride=2, padding
            ↪ =1)
        self.r1 = ResidualBlock(in_c+in_c, out_c)
        self.r2 = ResidualBlock(out_c, out_c)
        self.p1 = MixPool(out_c, out_c)

    def forward(self, inputs, skip, masks):
        x = self.upsample(inputs)
        x = torch.cat([x, skip], axis=1)
        x = self.r1(x)
        x = self.r2(x)
        p = self.p1(x, masks)
        return p

class FANet(nn.Module):
    def __init__(self):
        super(FANet, self).__init__()

        self.e1 = EncoderBlock(3, 32)
        self.e2 = EncoderBlock(32, 64)
        self.e3 = EncoderBlock(64, 128)
        self.e4 = EncoderBlock(128, 256)

        self.d1 = DecoderBlock(256, 128)
        self.d2 = DecoderBlock(128, 64)
        self.d3 = DecoderBlock(64, 32)
        self.d4 = DecoderBlock(32, 16)

        self.output = nn.Conv2d(16+1, 1, kernel_size=1, padding=0)

    def forward(self, x):
        inputs, masks = x[0], x[1]

        p1, s1 = self.e1(inputs, masks)
        p2, s2 = self.e2(p1, masks)
        p3, s3 = self.e3(p2, masks)
        p4, s4 = self.e4(p3, masks)

        d1 = self.d1(p4, s4, masks)
        d2 = self.d2(d1, s3, masks)
        d3 = self.d3(d2, s2, masks)
        d4 = self.d4(d3, s1, masks)
```

```
        d5 = torch.cat([d4, masks], axis=1)
        output = self.output(d5)

        return output

if __name__ == "__main__":
    x = torch.randn((2, 3, 256, 256)).cuda()
    m = torch.randn((2, 1, 256, 256)).cuda()
    model = FANet().cuda()
    y = model([x, m])
    print(y.shape)
```

D. Model:

```
import torch
import torch.nn as nn
from blocks import ResidualBlock, MixPool,new_ResidualBlock
'''new attention'''
class EncoderBlock(nn.Module):
    def __init__(self, in_c, out_c, name=None):
        super(EncoderBlock, self).__init__()

        self.name = name
        self.r1 = new_ResidualBlock(in_c, out_c)
        self.r2 = new_ResidualBlock(out_c, out_c)
        self.p1 = MixPool(out_c, out_c)
        self.pool = nn.MaxPool2d((2, 2))

    def forward(self, inputs, masks):
        x = self.r1(inputs)
        x = self.r2(x)
        p = self.p1(x, masks)
        o = self.pool(p)
        return o, x

class DecoderBlock(nn.Module):
    def __init__(self, in_c, out_c, name=None):
        super(DecoderBlock, self).__init__()

        self.upsample = nn.ConvTranspose2d(in_c, in_c, kernel_size=4, stride=2, padding
            ↪ =1)
        self.r1 = new_ResidualBlock(in_c+in_c, out_c)
        self.r2 = new_ResidualBlock(out_c, out_c)
        self.p1 = MixPool(out_c, out_c)

    def forward(self, inputs, skip, masks):
        x = self.upsample(inputs)
        x = torch.cat([x, skip], axis=1)
        x = self.r1(x)
        x = self.r2(x)
        p = self.p1(x, masks)
        return p
'''network'''
class FANet(nn.Module):
    def __init__(self):
        super(FANet, self).__init__()
```

```
        self.e1 = EncoderBlock(3, 32)
        self.e2 = EncoderBlock(32, 64)
        self.e3 = EncoderBlock(64, 128)
        self.e4 = EncoderBlock(128, 256)

        self.d1 = DecoderBlock(256, 128)
        self.d2 = DecoderBlock(128, 64)
        self.d3 = DecoderBlock(64, 32)
        self.d4 = DecoderBlock(32, 16)

        self.output = nn.Conv2d(16+1, 1, kernel_size=1, padding=0)

    def forward(self, x):
        inputs, masks = x[0], x[1]

        p1, s1 = self.e1(inputs, masks)
        p2, s2 = self.e2(p1, masks)
        p3, s3 = self.e3(p2, masks)
        p4, s4 = self.e4(p3, masks)

        d1 = self.d1(p4, s4, masks)
        d2 = self.d2(d1, s3, masks)
        d3 = self.d3(d2, s2, masks)
        d4 = self.d4(d3, s1, masks)

        d5 = torch.cat([d4, masks], axis=1)
        output = self.output(d5)

        return output
class FANet_3layer(nn.Module):
    def __init__(self):
        super(FANet_3layer, self).__init__()

        self.e1 = EncoderBlock(3, 32)
        self.e2 = EncoderBlock(32, 64)
        self.e3 = EncoderBlock(64, 128)
        # self.e4 = EncoderBlock(128, 256)

        # self.d1 = DecoderBlock(256, 128)
        self.d2 = DecoderBlock(128, 64)
        self.d3 = DecoderBlock(64, 32)
        self.d4 = DecoderBlock(32, 16)

        self.output = nn.Conv2d(16+1, 1, kernel_size=1, padding=0)

    def forward(self, x):
        inputs, masks = x[0], x[1]

        p1, s1 = self.e1(inputs, masks)
        p2, s2 = self.e2(p1, masks)
        p3, s3 = self.e3(p2, masks)
        # p4, s4 = self.e4(p3, masks)

        # d1 = self.d1(p4, s4, masks)
        d2 = self.d2(p3, s3, masks)
```

```
        d3 = self.d3(d2, s2, masks)
        d4 = self.d4(d3, s1, masks)

        d5 = torch.cat([d4, masks], axis=1)
        output = self.output(d5)

        return output
class l3_FANet(nn.Module):
    def __init__(self):
        super(l3_FANet, self).__init__()

        self.e1 = EncoderBlock(3, 32)
        self.e2 = EncoderBlock(32, 64)
        self.e3 = EncoderBlock(64, 128)
        # self.e4 = EncoderBlock(128, 256)

        # self.d1 = DecoderBlock(256, 128)
        self.d2 = DecoderBlock(128, 64)
        self.d3 = DecoderBlock(64, 32)
        self.d4 = DecoderBlock(32, 16)

        self.output = nn.Conv2d(16+1, 1, kernel_size=1, padding=0)

    def forward(self, x):
        inputs, masks = x[0], x[1]

        p1, s1 = self.e1(inputs, masks)
        p2, s2 = self.e2(p1, masks)
        p3, s3 = self.e3(p2, masks)
        # p4, s4 = self.e4(p3, masks)

        # d1 = self.d1(p4, s4, masks)
        d2 = self.d2(p3, s3, masks)
        d3 = self.d3(d2, s2, masks)
        d4 = self.d4(d3, s1, masks)

        d5 = torch.cat([d4, masks], axis=1)
        output = self.output(d5)

        return output
class NewFANet(nn.Module):
    def __init__(self):
        super(NewFANet, self).__init__()

        self.e1 = EncoderBlock(3, 32)
        self.e2 = EncoderBlock(32, 64)
        self.e3 = EncoderBlock(64, 128)
        self.e4 = EncoderBlock(128, 256)
        self.e5 = EncoderBlock(256, 512)

        self.d0 = DecoderBlock(512, 256)
        self.d1 = DecoderBlock(256, 128)
        self.d2 = DecoderBlock(128, 64)
        self.d3 = DecoderBlock(64, 32)
        self.d4 = DecoderBlock(32, 16)
```

```
        self.output = nn.Conv2d(16+1, 1, kernel_size=1, padding=0)

    def forward(self, x):
        inputs, masks = x[0], x[1]

        p1, s1 = self.e1(inputs, masks)
        p2, s2 = self.e2(p1, masks)
        p3, s3 = self.e3(p2, masks)
        p4, s4 = self.e4(p3, masks)
        p5, s5 = self.e5(p4, masks)

        d0 = self.d0(p5, s5, masks)
        d1 = self.d1(d0, s4, masks)
        d2 = self.d2(d1, s3, masks)
        d3 = self.d3(d2, s2, masks)
        d4 = self.d4(d3, s1, masks)

        d5 = torch.cat([d4, masks], axis=1)
        output = self.output(d5)

        return output
if __name__ == "__main__":
    x = torch.randn((2, 3, 256, 256)).cuda()
    m = torch.randn((2, 1, 256, 256)).cuda()
    model = l3_FANet().cuda()
    y = model([x, m])
    print(y.shape)
```

E. Test:

```
import os, time
from operator import add
import json
import numpy as np
import cv2
from glob import glob
from tqdm import tqdm
import torch
from sklearn.metrics import confusion_matrix, accuracy_score
import torch.nn as nn
from model import FANet,NewFANet,l3_FANet
from utils import create_dir, seeding, init_mask, rle_encode, rle_decode, load_data
from collections import defaultdict
from Miou import fast_hist,per_class_iu,per_class_PA

def comput_miou(y_true, y_pred):
    hist = fast_hist(y_true, y_pred, 2)
    # print(hist)
    mIoUs=per_class_iu(hist)
    miou=round(np.nanmean(mIoUs) , 2)
    # print(miou)
    return miou

def precision_score(y_true, y_pred):
    intersection = (y_true * y_pred).sum()
```

```
    return (intersection + 1e-15) / (y_pred.sum() + 1e-15)

def recall_score(y_true, y_pred):
    intersection = (y_true * y_pred).sum()
    return (intersection + 1e-15) / (y_true.sum() + 1e-15)

def F2_score(y_true, y_pred, beta=2):
    p = precision_score(y_true,y_pred)
    r = recall_score(y_true, y_pred)
    return (1+beta**2.) *(p*r) / float(beta**2*p + r + 1e-15)

def dice_score(y_true, y_pred):
    return (2 * (y_true * y_pred).sum() + 1e-15) / (y_true.sum() + y_pred.sum() + 1e
        ↪ -15)

def jac_score(y_true, y_pred):
    intersection = (y_true * y_pred).sum()
    union = y_true.sum() + y_pred.sum() - intersection
    return (intersection + 1e-15) / (union + 1e-15)

def calculate_metrics(y_true, y_pred, img):
    y_true = y_true.cpu().numpy()
    y_pred = y_pred.cpu().numpy()

    y_pred = y_pred > 0.5
    y_pred = y_pred.reshape(-1)
    y_pred = y_pred.astype(np.uint8)

    y_true = y_true > 0.5
    y_true = y_true.reshape(-1)
    y_true = y_true.astype(np.uint8)

    ## Score
    score_jaccard = jac_score(y_true, y_pred)
    score_f1 = dice_score(y_true, y_pred)
    score_recall = recall_score(y_true, y_pred)
    score_precision = precision_score(y_true, y_pred)
    score_fbeta = F2_score(y_true, y_pred)
    score_acc = accuracy_score(y_true, y_pred)
    score_miou=comput_miou(y_true, y_pred)
    # print(score_miou)
    confusion = confusion_matrix(y_true, y_pred)
    if confusion.shape[1] > 1:
      if float(confusion[0,0] + confusion[0,1]) != 0:
          score_specificity = float(confusion[0,0]) / float(confusion[0,0] + confusion
              ↪ [0,1])
      else:
          score_specificity = 0.0
    else:
      score_specificity = 0.0
    return [score_jaccard, score_f1, score_recall, score_precision, score_specificity,
        ↪ score_acc, score_fbeta , score_miou]

def mask_parse(mask):
    mask = np.squeeze(mask)
    mask = [mask, mask, mask]
```

```
    mask = np.transpose(mask, (1, 2, 0))
    return mask

class CustomDataParallel(torch.nn.DataParallel):
    """ A Custom Data Parallel class that properly gathers lists of dictionaries. """
    def gather(self, outputs, output_device):
        # Note that I don't actually want to convert everything to the output_device
        return sum(outputs, [])


def convert_to_list(obj):
    if isinstance(obj, np.ndarray):
        return obj.tolist()
    return obj
def find_max(dd):
    matrices = [dd[key] for key in dd.keys()]
    stacked_matrices = np.stack(matrices)
    max_values = np.max(stacked_matrices, axis=0)
    max_index=np.argmax(stacked_matrices,axis=0)
    # np.savetxt('max_index.csv', max_index, delimiter=',')
    keys = list(dd.keys())
    result_matrix=[]
    for xx,row in enumerate(max_values):
        result_matrix.append([])
        for yy, max_val in enumerate(row):
            result_matrix[xx].append((keys[max_index[xx][yy]],max_val))
    return result_matrix
if __name__ == "__main__":
    color_dict={'car':(255,0,0),
                'people':(0,0,255),
                'path':(128,64,128)
                }
    """ Seeding """
    seeding(42)

    """ Load dataset """
    path = "VOCdevkit"
    (train_x, train_y), (test_x, test_y) = load_data(path)

    test_num=348
    test_x,test_y=test_x[:test_num],test_y[:test_num]
    """ Hyperparameters """
    size = (512, 512)
    num_iter = 15
    # checkpoint_dict={
    # 'car':r'files\checkpoint_car_original.pth',
    # 'people':r'files\people_original.pth',
    # 'path':r'files\path_original.pth',
    # }

    # checkpoint_dict={
    # 'car':r'files\3layer_car.pth',
    # 'people':r'files\3layer_people.pth',
    # 'path':r'files\3layer_road.pth', # }
    checkpoint_dict={
        'car':r'files\checkpoint_att3layercar.pth',
```

```
    'people':r'files\checkpoint_att3ped.pth',
    'path':r'files\checkpoint_a3road.pth',
}
""" Directories """
dir_name='3layer_attention'
create_dir(dir_name)

""" Load the checkpoint """
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


# model = model.to(device)

model_dict={}
for key,value in checkpoint_dict.items():
    # model = FANet()
    model=l3_FANet()
    # model=NewFANet()
    temp_model=nn.DataParallel(model)
    temp_model.load_state_dict(torch.load(value, map_location=device),strict=False)
    model_dict[key]=CustomDataParallel(temp_model).to(device)
    model_dict[key].eval()

""" Testing """
mask_dict=defaultdict(list)
for key,model in model_dict.items():
    prev_masks = init_mask(test_x, size)
    save_data = []
    file = open(f"{dir_name}/test_results.csv", "a+")
    file_path=f"{dir_name}/test_results.csv"
    is_empty = os.stat(file_path).st_size == 0
    if is_empty:
        file.write("Category,Iteration,Jaccard,F1,Recall,Precision,Specificity,
            ↪ Accuracy,F2,Mean Time,Mean FPS,mIoU\n")

    for iter in range(num_iter):

        metrics_score = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        tmp_masks = []
        time_taken = []

        for i, (x, y) in tqdm(enumerate(zip(test_x, test_y)), total=len(test_x)):
            ## Image
            image = cv2.imread(x, cv2.IMREAD_COLOR)
            image = cv2.resize(image, size)
            img_x = image
            image = np.transpose(image, (2, 0, 1))
            image = image/255.0
            image = np.expand_dims(image, axis=0)
            image = image.astype(np.float32)
            image = torch.from_numpy(image)
            image = image.to(device)

            ## Mask
            mask = cv2.imread(y, cv2.IMREAD_GRAYSCALE)
            mask = cv2.resize(mask, size)
```

```
            mask = np.expand_dims(mask, axis=0)
            mask = mask/255.0
            mask = np.expand_dims(mask, axis=0)
            mask = mask.astype(np.float32)
            mask = torch.from_numpy(mask)
            mask = mask.to(device)

            ## Prev mask
            pmask = prev_masks[i]
            pmask = " ".join(str(d) for d in pmask)
            pmask = str(pmask)
            pmask = rle_decode(pmask, size)
            pmask = np.expand_dims(pmask, axis=0)
            pmask = np.expand_dims(pmask, axis=0)
            pmask = pmask.astype(np.float32)
            if iter == 0:
                pmask = np.transpose(pmask, (0, 1, 3, 2))
            pmask = torch.from_numpy(pmask)
            pmask = pmask.to(device)

            with torch.no_grad():
                """ FPS Calculation """
                start_time = time.time()
                network_out=model([image, pmask])
                pred_y = torch.sigmoid(network_out)
                end_time = time.time() - start_time
                time_taken.append(end_time)

                score = calculate_metrics(mask, pred_y, img_x)
                metrics_score = list(map(add, metrics_score, score))
                pred_y = pred_y[0][0].cpu().numpy()
                if iter==num_iter-1:
                    filtered_array = np.where(pred_y > 0.5, pred_y, 0)
                    mask_dict[key].append(filtered_array)

                pred_y = pred_y > 0.5
                # print(pred_y)
                pred_y = np.transpose(pred_y, (1, 0))
                pred_y = np.array(pred_y, dtype=np.uint8)
                pred_y = rle_encode(pred_y)
                prev_masks[i] = pred_y
                tmp_masks.append(pred_y)

        """ Mean Metrics Score """
        print(metrics_score)
        jaccard = metrics_score[0]/len(test_x)
        f1 = metrics_score[1]/len(test_x)
        recall = metrics_score[2]/len(test_x)
        precision = metrics_score[3]/len(test_x)
        specificity = metrics_score[4]/len(test_x)
        acc = metrics_score[5]/len(test_x)
        f2 = metrics_score[6]/len(test_x)
        miou=metrics_score[7]/len(test_x)

        """ Mean Time Calculation """
        mean_time_taken = np.mean(time_taken)
```

```
        print("Mean Time Taken: ", mean_time_taken)
        mean_fps = 1/mean_time_taken

        print(f"Category: {key} ,Jaccard: {jaccard:1.4f} - F1: {f1:1.4f} - Recall: {
            ↪ recall:1.4f} - Precision: {precision:1.4f} - Specificity: {specificity
            ↪ :1.4f} - Acc: {acc:1.4f} - F2: {f2:1.4f} - Mean Time: {mean_time_taken
            ↪ :1.7f} - Mean FPS: {mean_fps:1.7f} - mIoU: {miou:1.7f}")

        save_str = f"{key},{iter+1},{jaccard:1.4f},{f1:1.4f},{recall:1.4f},{precision
            ↪ :1.4f},{specificity:1.4f},{acc:1.7f},{f2:1.7f},{mean_time_taken:1.7f},{
            ↪ mean_fps:1.7f},{miou:1.7f}\n"
        file.write(save_str)

        save_data.append(tmp_masks)
    save_data = np.array(save_data, dtype=object)
""" Saving the masks. """
category_list=list(mask_dict.keys())
print(category_list)
for i, (x, y) in tqdm(enumerate(zip(test_x, test_y)), total=len(test_x)):
    image = cv2.imread(x, cv2.IMREAD_COLOR)
    image = cv2.resize(image, size)


    temp_mask_dict={}
    for key in category_list:
        temp_mask_dict[key]=mask_dict[key][i]
    mix_matrix=find_max(temp_mask_dict)

    name = y.split("\\")[-1].split(".")[0]
    sep_line = np.ones((size[0], 10, 3)) * 128
    tmp = [image, sep_line]
    mask=np.zeros((size[0],size[1], 3), dtype=np.uint8)


    for ii in range(len(mix_matrix)):
        for jj in range(len(mix_matrix[ii])):
            category,value=mix_matrix[ii][jj]
            if value>0.5:
                mask[ii][jj]=color_dict[category]

    # cv2.imwrite(f"multi_results/{name}_mask.png", mask)
    # for key in category_list:
    # for ii in range(len(mask_dict[key][i])):
    # for jj in range(len(mask_dict[key][i][ii])):
    # value=mask_dict[key][i][ii][jj]
    # if value>0.5:
    # mask[ii][jj]=color_dict[key]
    tmp.append(mask)
    cat_images = np.concatenate(tmp, axis=1)
    cv2.imwrite(f"{dir_name}/{name}.png", cat_images)
```

F. Miou:

```
import numpy as np
from PIL import Image
def fast_hist(a, b, n):
```

```
    k = (a >= 0) & (a < n)
    return np.bincount(n * a[k].astype(int) + b[k], minlength=n ** 2).reshape(n, n)
def per_class_iu(hist):
    return np.diag(hist) / np.maximum((hist.sum(1) + hist.sum(0) - np.diag(hist)), 1)
```

G. Utils:

```
import os
import time
import random
import numpy as np
import cv2
from tqdm import tqdm
import torch
from sklearn.utils import shuffle

""" Seeding the randomness. """
def seeding(seed):
    random.seed(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True

""" Create a directory. """
def create_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

""" Load the Kvasir-SEG dataset """
def load_data(path):
    def load_names(path, file_path):
        f = open(file_path, "r")
        data = f.read().split("\n")[:-1]
        path='VOCdevkit\VOCdevkit'
        images = [os.path.join(path,"JPEGImages", name) + ".png" for name in data]
        masks = [os.path.join(path,"pedestrians", name.replace('_leftImg8bit','
            ↪ _gtFine_color')) + ".png" for name in data]
        return images, masks

    train_names_path = f"{path}/train.txt"
    valid_names_path = f"{path}/val.txt"

    train_x, train_y = load_names(path, train_names_path)
    valid_x, valid_y = load_names(path, valid_names_path)

    return (train_x, train_y), (valid_x, valid_y)

""" Shuffle the dataset. """
def shuffling(x, y):
    x, y = shuffle(x, y, random_state=42)
    return x, y

def epoch_time(start_time, end_time):
```

```
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

def print_and_save(file_path, data_str):
    print(data_str)
    with open(file_path, "a") as file:
        file.write(data_str)
        file.write("\n")

def rle_encode(x):
    '''
    x: numpy array of shape (height, width), 1 - mask, 0 - background
    Returns run length as list
    '''
    dots = np.where(x.T.flatten()==1)[0] # .T sets Fortran order down-then-right
    run_lengths = []
    prev = -2
    for b in dots:
        if (b>prev+1): run_lengths.extend((b+1, 0))
        run_lengths[-1] += 1
        prev = b
    return run_lengths

def rle_decode(mask_rle, shape):
    '''
    mask_rle: run-length as string formated (start length)
    shape: (height,width) of array to return
    Returns numpy array, 1 - mask, 0 - background

    '''
    s = mask_rle.split()
    starts, lengths = [np.asarray(x, dtype=int) for x in (s[0:][::2], s[1:][::2])]
    starts -= 1
    ends = starts + lengths
    img = np.zeros(shape[0]*shape[1], dtype=np.uint8)
    for lo, hi in zip(starts, ends):
        img[lo:hi] = 1
    return img.reshape(shape)

""" Initial mask build using Otsu thresholding. """
def init_mask(images, size):
    def otsu_mask(image, size):
        img = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
        img = cv2.resize(img, size)
        blur = cv2.GaussianBlur(img,(5,5),0)
        ret, th = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
        th = th.astype(np.int32)
        th = th/255.0
        th = th > 0.5
        th = th.astype(np.int32)
        return img, th

    mask = []
    for image in tqdm(images, total=len(images)):
```

```
        name = image.split("/")[-1]
        i, m = otsu_mask(image, size)
        # cv2.imwrite(f"mask/{name}", np.concatenate([i, m*255], axis=1))
        m = rle_encode(m)
        mask.append(m)

    return mask
```

H. TrainingSetSegmentation:

```
import os
import random

segfilepath=r'/content/VOCdevkit/images'
saveBasePath=r"/content/VOCdevkit"

trainval_percent=1
# 9:1
train_percent=0.9

temp_seg = os.listdir(segfilepath)
total_seg = []
for seg in temp_seg:
    if seg.endswith(".png"):
        total_seg.append(seg)

num=len(total_seg)
list=range(num)
tv=int(num*trainval_percent)
tr=int(tv*train_percent)
trainval= random.sample(list,tv)
train=random.sample(trainval,tr)

print("train and val size",tv)
print("traub suze",tr)
ftrainval = open(os.path.join(saveBasePath,'trainval.txt'), 'w')
ftest = open(os.path.join(saveBasePath,'test.txt'), 'w')
ftrain = open(os.path.join(saveBasePath,'train.txt'), 'w')
fval = open(os.path.join(saveBasePath,'val.txt'), 'w')

for i in list:
    name=total_seg[i][:-4]+'\n'
    if i in trainval:
        ftrainval.write(name)
        if i in train:
            ftrain.write(name)
        else:
            fval.write(name)
    else:
        ftest.write(name)

ftrainval.close()
ftrain.close()
fval.close()
ftest.close()
```

I. LabelsSegmentationn:

```
import os
from PIL import Image
import threading
from threading import Thread

def filter_custom_rgb(image_path, output_folder, target_rgb, threshold=0,threadmax=
    ↪ None):
    image = Image.open(image_path)
    width, height = image.size
    threshold_value = threshold
    new_image = Image.new("RGB", (width, height), "black")
    new_pixels = new_image.load()
    for y in range(height):
        for x in range(width):
            r, g, b = image.getpixel((x, y))

            delta_r = abs(r - target_rgb[0])
            delta_g = abs(g - target_rgb[1])
            delta_b = abs(b - target_rgb[2])

            if delta_r <= threshold_value and delta_g <= threshold_value and delta_b <=
                ↪ threshold_value:
                new_pixels[x, y] = (r, g, b)
            else:
                new_pixels[x, y] = (0, 0, 0)
    filename = os.path.basename(image_path)
    new_image_path = os.path.join(output_folder, filename)
    new_image.save(new_image_path)
    threadmax.release()
def process_images_in_folder(input_folder, output_folder, target_rgb, threshold):
    os.makedirs(output_folder, exist_ok=True)
    image_files = [os.path.join(input_folder, filename) for filename in os.listdir(
        ↪ input_folder) if filename.endswith((".jpg", ".jpeg", ".png"))]
    threads = []
    for image_file in image_files:
        threadmax.acquire()
        thread = Thread(target=filter_custom_rgb, args=(image_file, output_folder,
            ↪ target_rgb, threshold, threadmax))
        thread.start()
        threads.append(thread)


input_folder = "VOCdevkit/VOC2007/masks"
output_folder = "VOCdevkit/VOC2007/people"
max_threads = 30
threadmax = threading.BoundedSemaphore(max_threads)
target_rgb = (222,16,70) #
threshold = 0

process_images_in_folder(input_folder, output_folder, target_rgb, threshold)
```