Computer Vision

Vehicles Damage Detection

Carlos Serrano Gutiérrez
ALGEBRA UNIVERSITY COLLEGE, JUNE, 2024

Content

1. Introduction

In this computer vision project, we explore semantic segmentation to address a specific problem within this field. The project is structured as follows:

1. Select the problem.
2. Define and prepare the dataset.
3. Set the metrics for evaluating the models.
4. Train, test, and evaluate several models.
5. Develop and evaluate several custom models based on insights from previous models and fine-tune the hyperparameters.
6. Enrich dataset by creating and labeling new images

By following this plan, I aim to tackle a challenging problem in semantic segmentation.

2. Problem

The initial problem was to determine the specific part of a car that has sustained damage. This posed a complex challenge, as it required the model to simultaneously identify the part of the car and detect the damage on it. Due to the inherent complexity and the dual-task nature of this problem, it became evident that a more focused approach was needed.

Given my interest in automotive-related projects and the potential to integrate this work with other ongoing projects, I decided to continue with this theme. This led to two potential directions:

1. Developing a model to differentiate between various parts of the car.
2. Developing a model to classify the type of damage on a car.

While both directions were intriguing, practical considerations guided the choice. The dataset available for the first problem was more suited to object detection rather than semantic segmentation. In contrast, I found a dataset that was ideal for the second problem, which focused on determining whether the car has a severe damage and where.

The goal of the current project is to develop a model that can identify the presence of damage on a car. This problem has numerous real-world applications, such as:

- **Insurance**: Automating the detection and classification of car damages can streamline claims processing and assessment.
- **Second-Hand Car Marketplace**: Platforms for buying and selling used cars can utilize this model to automatically detect and classify damages in car photographs, enhancing transparency and trust.
- **Rental Car Inspections**: Car rental companies can use the model for pre- and post-rental inspections to document vehicle conditions accurately.

These applications demonstrate the practical significance and potential impact of solving this problem effectively.

3. Dataset

The dataset utilized in this project originates from Roboflow and is specifically from the following URL: Car Damages - v5 2023-02-28 11:48am (roboflow.com). In the time of writing, the dataset version in use is the latest, denoted as "2023-02-28 11:48am.".

Although the dataset said it has 1974 images for training, the reality was that it hasn't. After downloading the dataset was divided as follows:

	TRAIN	TEST	VALID
PERCENTAGE	85%	5%	10%
NUMBER	502	68	28

To check that this disparity between what Roboflow said and what actually was downloaded, I downloaded via zip and with the following code:

```
from roboflow import Roboflow

rf = Roboflow(api_key="Q2rKRs5mjIazFZw5vELJ")
project = rf.workspace("project-p5nyc").project("car-damages-v3gyz")
version = project.version(5)
dataset = version.download("coco-segmentation")
```

In both cases I had the same result.

The dataset has three classes:

- Severe Damages
- No Damage
- Null (no class)

Figure 1 How dataset looks like

The dataset was downloaded in COCO JSON format, as model cannot directly interpret that format, it was necessary to transform all annotations into labeled masks. For this purpose, a GitHub repository containing valuable scripts was identified: https://github.com/bnsreenu/python_for_microscopists/tree/master/335.

Among these scripts, the most pertinent one was the script responsible for converting COCO JSON annotations into masks. Minor modifications were applied to generate a dataset suitable for training. The script used is included in the codebase and is named `coco-to-mask.py`.

The training of some of the models with the hole dataset could take more than 2 hours for a 10 epochs training. Because of that a script for splitting the model in an specific percentage was develop (`dataset-divider.py`). The model resulted from this script was used in the first stages of the model's selection, so it was a faster task. The percentage finally selected was 40% which reduced considerably the training time.

4. Metrics

In the realm of computer vision, metrics play a crucial role in the evaluation process, providing a quantitative view of how well the model is. In this projects there are 4 metrics used:

4.1. Loss Function

Measures the difference between the predicted output of the model and the actual ground truth. It quantifies how well the model's predictions match the expected results. The loss function is essential during training as it guides the optimization process, helping the model learn to minimize errors.

4.2. Dice Coefficient (F1-Score)

It is one of the most widespread scores for performance measurement in computer vision. Dice coefficient is calculated from the precision and recall of a prediction. Then, it scores the overlap between predicted segmentation and ground truth. It also penalizes false positives.

For better understanding:

- True Positives (TP): represent the number of pixels that has been properly classified as a class.
- False Positives (FP): represent the number of background (null) pixels that has been classified as a class.
- False Negatives (FN) represent the number of classes pixels that has been misclassified as background.
- True Negatives (TN): represent the number of pixels that has been properly classified as background.

So, the precision is the TP divided by all positives:

$$precision = \frac{TP}{TP + FP}$$

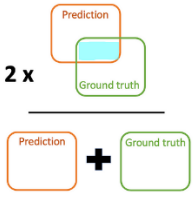
And the recall, also known as *sensitivity*, is the number of TP divided by all samples that should been identified as positive:

$$recall = \frac{TP}{TP + FN}$$

So once knowing this, can be said that Dice Coefficient is a harmonic mean of precision and recall. In other words, it is calculated by 2*intersection divided by the total number of pixels in both images.

$$Dice = \frac{2TP}{2TP + FP + FN}$$

$$\text{Dice} = \frac{2\ X\ Area\ of\ overlap}{Total\ area} =$$

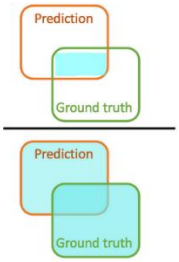


4.3. IoU (Intersection over Union)

Intersection over Union, also known as Jaccard index, is the area of the intersection over union of the predicted segmentation and the ground truth:

$$IoU = \frac{TP}{TP + FP + FN}$$

$$\text{IoU} = \frac{Area\ of\ overlap}{Area\ of\ union} = \frac{}{}$$



IoU provides a measure of how well the predicted mask aligns with the ground truth mask. It is similar to the Dice Coefficient, but it penalize more the FP and the FN.

4.4. Pixel Accuracy

It is the ratio of correctly predicted pixels to the total number of pixels. It measures the proportion of true results (both TP and TN) among the total number of cases examined. Accuracy is a straightforward and widely used metric that provides a quick overview of the model's overall performance.

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP}$$

5. Training Process

The process begins with training the first model, followed by validation. Metrics are recorded at each stage, and upon completion of each epoch, these metrics are automatically saved to a file. Additionally, the most recent model update is saved, ensuring that the best-performing model can be easily retrieved later, as the epoch with the highest performance metrics may not always be the final one. Once models are trained, they are analyzed and compared with other models.

6. Models

In this section will try to find which hyperparameters and state-of-the-art models work good for the present problem. For that reason, there has been prepared two sets of hyperparameters that will be used to train and evaluated two state-of-the-art models. After the analysis, will be trained the best model with the best hyperparameters for a longer period of time, in order to get a good prediction model. With this data a custom model will be try to create, with the focus on making a better solution.

6.1. State-of-the-art Models

The models used are the FCN-ResNet50 and the DeepLabV3-ResNet50. Both uses the ResNet backbone for feature extraction but differ in in their classification approach: FCN-ResNet50 utilizes a fully connected layer, whereas DeepLabV3-ResNet50 employs atrous convolution and Spatial Pyramid Pooling (SPP) to effectively capture multi-scale contextual information. This is how this models looks in code:

```
model1 = models.segmentation.fcn_resnet50(pretrained=False,
num_classes=num_classes)

model2 = models.segmentation.deeplabv3_resnet50(pretrained=False,
num_classes=num_classes)
```

The two sets of hyperparameters are:

1. First Hyperparameters
    a. Batch Size: 16
    b. Learning Rate: 0.001
    c. Optimizer: Adam
    d. Criterion: CrossEntropyLoss
2. Second Hyperparameters
    a. Batch Size: 32
    b. Learning Rate: 0.01
    c. Optimizer: SGD
    d. Criterion: CrossEntropyLoss

6.1.1. Five Epochs

This section will analyze the result of training both models with both set of hyperparameters for 5 epochs and with the dataset reduced in 40%. This section is not for getting a good model but for analyzing which one will be the best.

6.1.1.1. FCN-ResNet50

This is the comparative result of training the FCN-ResNet50 model with the hyperparameters. Yellow is with first hyperparameters and blue with second hyperparameters.

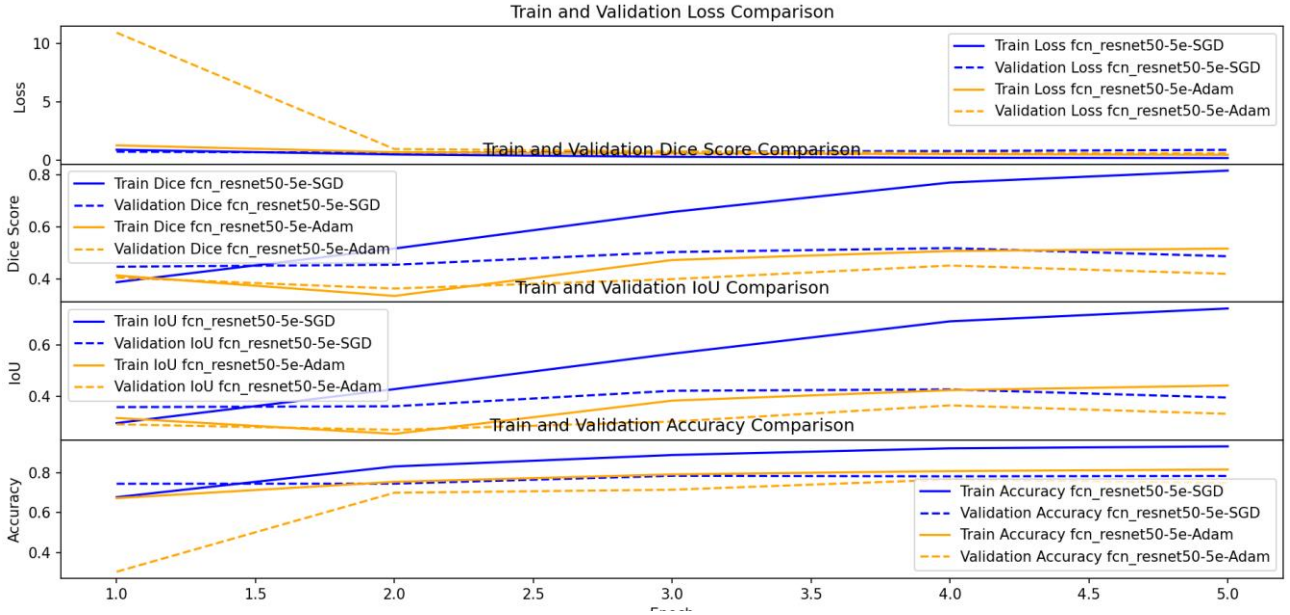


Figure 2 Comparation of the FCN-ResNet50 model with different hyperparameters

EPOCH	TRAIN LOSS	VAL LOSS	TRAIN DICE	VAL DICE	TRAIN IOU	VAL IOU	TRAIN ACCURACY	VAL ACCURACY
1	1.2763	10.9275	0.4139	0.4057	0.3159	0.2913	0.6724	0.3033
2	0.6896	0.9619	0.3354	0.3638	0.2544	0.2700	0.7542	0.6995
3	0.5851	0.7682	0.4732	0.3998	0.3833	0.3019	0.7915	0.7147
4	0.5203	0.6107	0.5076	0.4518	0.4239	0.3648	0.8081	0.7659
5	0.4683	0.6050	0.5168	0.4201	0.4420	0.3321	0.8160	0.7491

Table values of first hyperparameters

EPOCH	TRAIN LOSS	VAL LOSS	TRAIN DICE	VAL DICE	TRAIN IOU	VAL IOU	TRAIN ACCURACY	VAL ACCURACY
1	0.9018	0.7338	0.3882	0.4474	0.2962	0.3582	0.6772	0.7445

2	0.4969	0.6397	0.5172	0.4550	0.4278	0.3615	0.8314	0.7452
3	0.2956	0.7303	0.6574	0.5037	0.5653	0.4215	0.8887	0.7846
4	0.2148	0.7942	0.7701	0.5187	0.6910	0.4267	0.9223	0.7818
5	0.1897	0.8935	0.8159	0.4875	0.7408	0.3953	0.9320	0.7838

Table values of second hyperparameters

The model behaves similar with both hyperparameters in validation, in the contrary of training where the second hyperparameters got better result. This is not something good at all, the difference between validation and train with the second hyperparameters, suggest that the model is overfitting. This means that the model is learning that good the training images that he is not able to generalize. The first hyperparameters set don't overfit at all the models.

In both situation, model is getting a good pixel accuracy, especially compared with the Dice Coefficient and the IoU. This suggest that the model is good determining the background, but not determining the actual classes.

The graphics, in both cases, of IoU and Dice Coefficient has the same shape but in different values.

After seeing the metrics, we can say that second hyperparameters in a short term, but they overfit extremely fast. Also, they make the model train to fast in comparation to the validation which continuous stable in almost all the moment of the training, even making worse metrics at the end than in the epoch 3. It is being generalizing in all metrics, because it can be generalized. On the other hand, the model with the first hyperparameters is growing (slowly) all time which make them a better option in the future.

6.1.1.2. DeepLabV3-ResNet

This is the comparative result of training the DeepLabV3-ResNet50 model with the hyperparameters. Yellow is with first hyperparameters and blue with second hyperparameters.

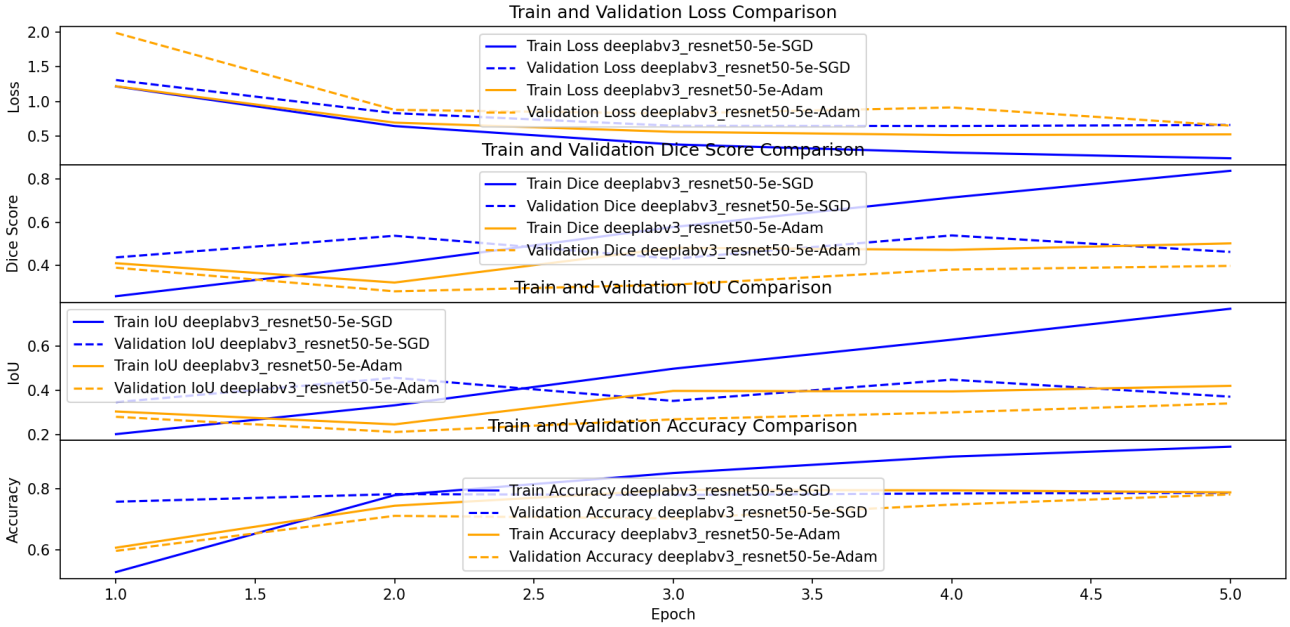


Figure 3 Comparation of the DeepLabV3-ResNet50 model with different hyperparameters

EPOCH	TRAIN LOSS	VAL LOSS	TRAIN DICE	VAL DICE	TRAIN IOU	VAL IOU	TRAIN ACCURACY	VAL ACCURACY
1	1.2185	1.9844	0.4109	0.3902	0.3040	0.2801	0.6070	0.5966
2	0.6964	0.8796	0.3218	0.2813	0.2459	0.2116	0.7441	0.7111
3	0.5642	0.8271	0.4819	0.3119	0.3968	0.2686	0.7960	0.7027
4	0.5175	0.9143	0.4724	0.3817	0.3945	0.2999	0.7944	0.7478
5	0.5279	0.6532	0.5025	0.3987	0.4196	0.3405	0.7879	0.7811

Table values of first hyperparameters

EPOCH	TRAIN LOSS	VAL LOSS	TRAIN DICE	VAL DICE	TRAIN IOU	VAL IOU	TRAIN ACCURACY	VAL ACCURACY
1	1.2161	1.3077	0.2588	0.4377	0.2020	0.3460	0.5273	0.7573
2	0.6470	0.8324	0.4085	0.5374	0.3319	0.4561	0.7786	0.7821
3	0.3835	0.6512	0.5776	0.4314	0.4967	0.3517	0.8511	0.7793
4	0.2665	0.6476	0.7138	0.5390	0.6275	0.4474	0.9047	0.7844
5	0.1846	0.6631	0.8369	0.4630	0.7670	0.3707	0.9372	0.7865

Table values of second hyperparameters

Very same situation as before. The model with second hyperparameters is very good training but in other to generalize is even worse than with first hyperparameters, even though it performs a lot worse in training. In the other hand, the first hyperparameter don't overfit at all the model with only 5 epochs.

There is a huge difference between accuracy, and dice and IoU. In both hyperparameter set the model is doing very well selecting the background but not that well in selecting the classes. The model is losing to find the TP.

In this occasion the shape of functions Dice and IoU is not completely the same as it was before. Although shape is similar.

The model is not improving any of its metrics in both cases, but in training that with second hyperparameters is doing so. We can conclude that for this model no matter the set of hyperparameters that it will behave more less the same in validation.

6.1.1.3. Conclusion

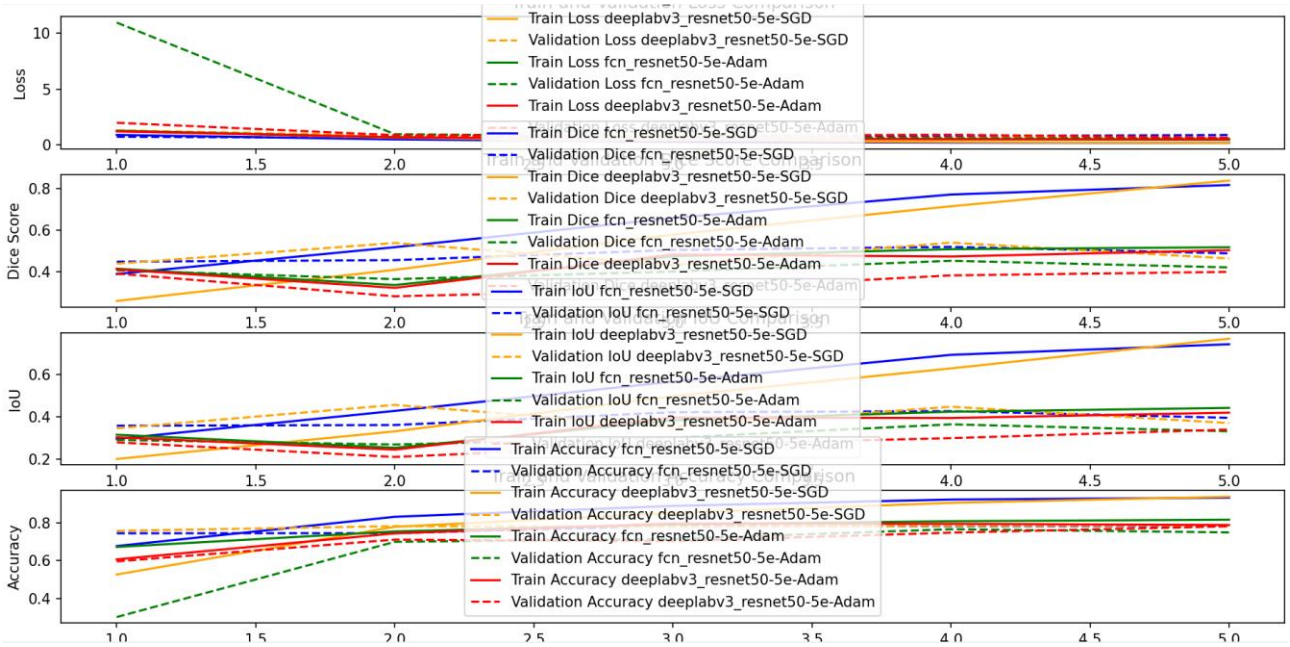


Figure 4 Comparation of training DeepLabV3-ResNet50 and FCN-ResNet50 models whith different hyperparameters

The graphic shown before is the comparation of all the models trained till the moment. It is hard to read, but focusing on the blue line (FCN-ResNet50 with second hyperparameters) outstand in all metrics but in loss that is lightly bigger in the final epoch. This is better seen in the testing results:

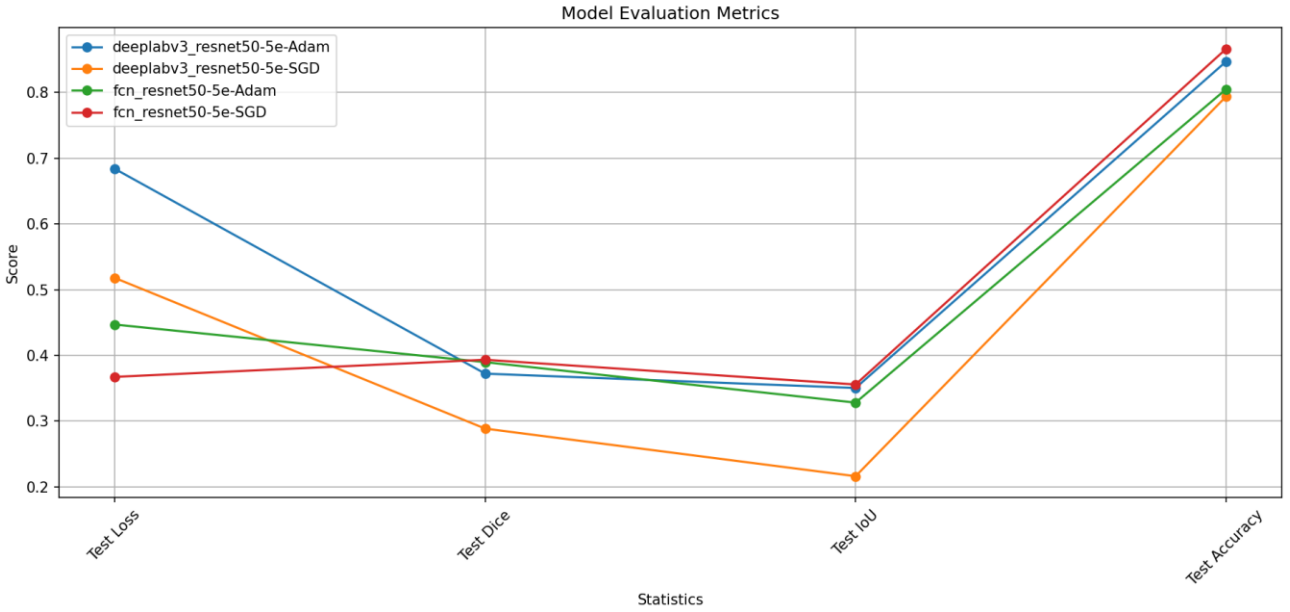


Figure 5 Comparation of testing DeepLabV3-ResNet50 and FCN-ResNet50 models with different hyperparameters

The one that was blue before now is red. We can see that the best performer in this moment of the project is the FCN-ResNet with the second parameters. But as I said in the previous analysis, the hyperparameters tend to overfit quickly. Nevertheless, I wanted to see if this was true at all. So, what I did is, I load the FCN-ResNet model and trained again with the second hyperparameters and the hole dataset. The results after 4 epochs (9 in total) were how was expected:

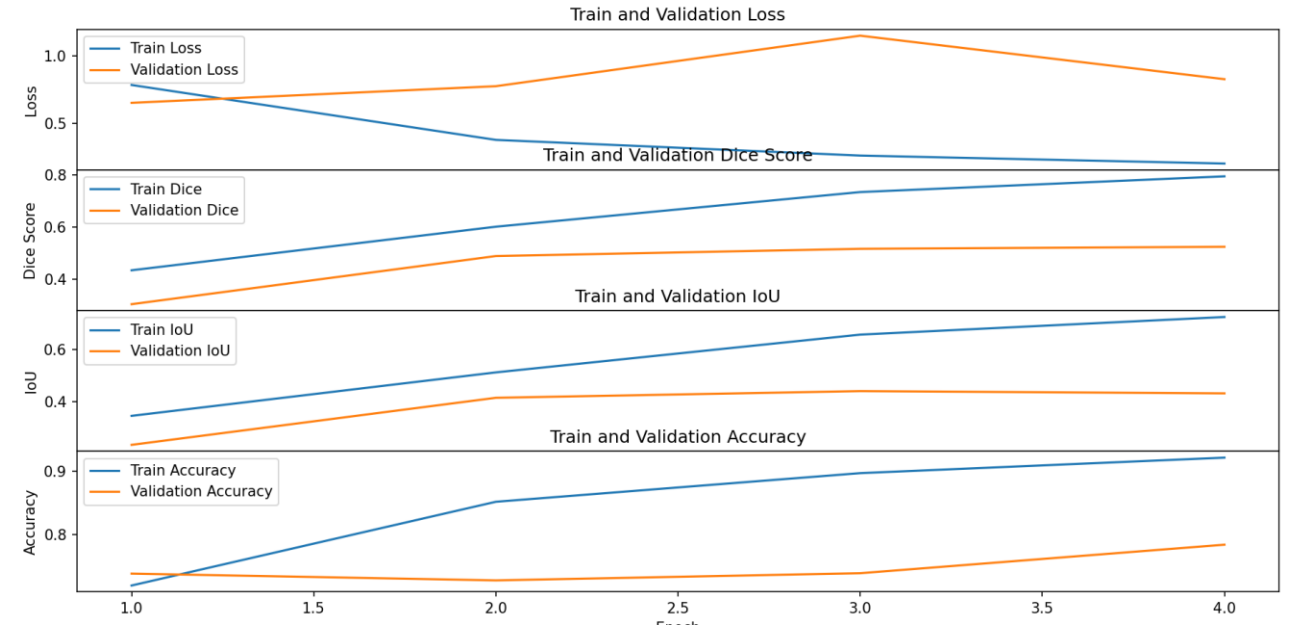


Figure 6 FCN-Resnet50 trained with second hyperparameter sets

IoU and Dice validation stabilized while training was growing. Even though pixel accuracy got a slightly better result at the end, the previous result where always the same. The big problem and the one that made me stop the training was the huge difference between training loss and validation loss.

The other metrics could suggest that the dataset is not the best one, and that the difference between classes is not clear for the model. But the difference between losses is a symptom of overfitting. With this it can be concluded that the second hyperparameters overfits the models, but not the second hypothesis (dataset is not good at all).

For validate the second hypothesis, in the next section I have trained the models for 10 epochs.

6.1.2. Conclusion

The first model that was trained due to its better performance with both model is FCN-ResNet50 although both models where trained. This training was with the first hyperparameter set, with the hole dataset and a 10-epochs-duration.

The result of training FCN-ResNet50 where disappointing:

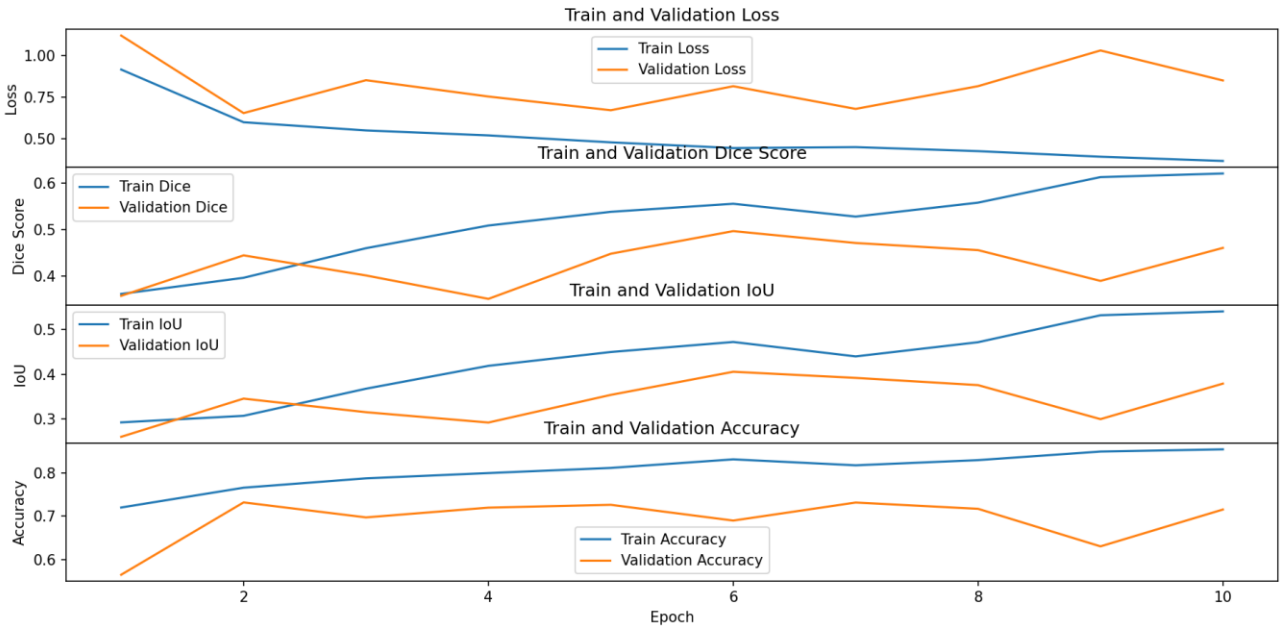


Figure 7 Results of training the FCN-ResNet50 during 10 epochs

Although after 6 epoch metrics where promising since before that they had a bit of correction, could be think that the model learned some patters and could perform great. In epoch 6 IoU and Dice got their ATH (all time high), and even though loss and accuracy went down could be think as the model learned the actual classes but forgot the background. Nevertheless, after that the model stabilized and made the same shape for all metrics which is a very strange situation.

After this bad result, there were not too much to expect for the DeepLavV3-ResNet. Two images say more than hundred words:

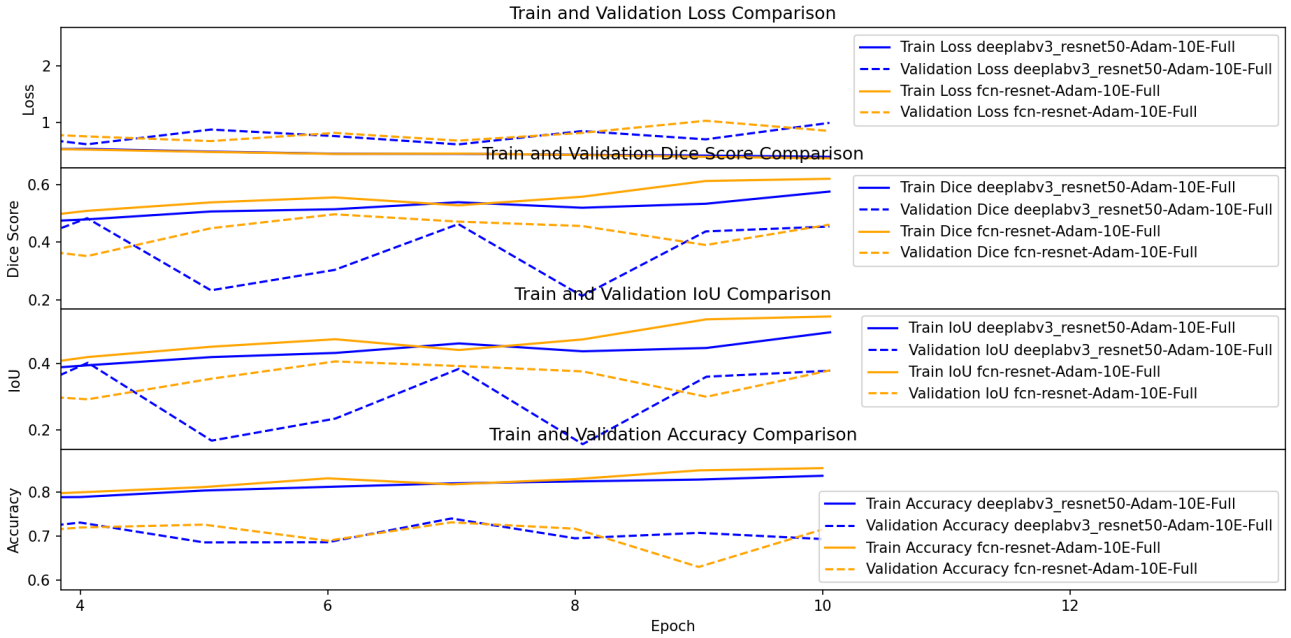


Figure 8 Comparation between training DeepLabV3-ResNet50 and FCN-ResNet50 with the same hyperparameters, full dataset and 10 epochs

Graphics are move to the left to see the final results

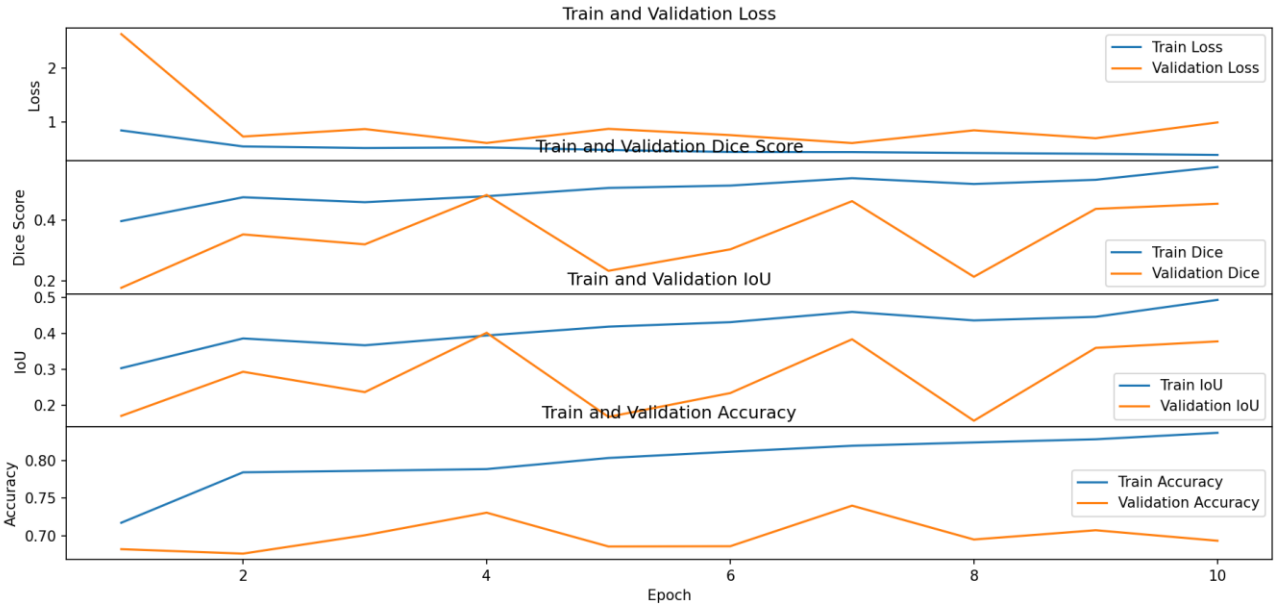


Figure 9 Result of training DeepLabV3-ResNet50 with the first hyperparameters set, full dataset and 10 epochs

The model behaves irregularly In all metrics. Don't make the loss function lie you, it is "stable" in the value of 1. Dice and IoU are equally terrible, and accuracy is as always has been.

In comparation with the other model, although it is not stable, has almost the same values for every metric at the end and in almost every epoch (but the ones it goes down). Also, the ATH in validation for every metric is in 4th epoch and is higher than the other.

This is the result of testing each of the DeepLabV3-ResNet metrics:

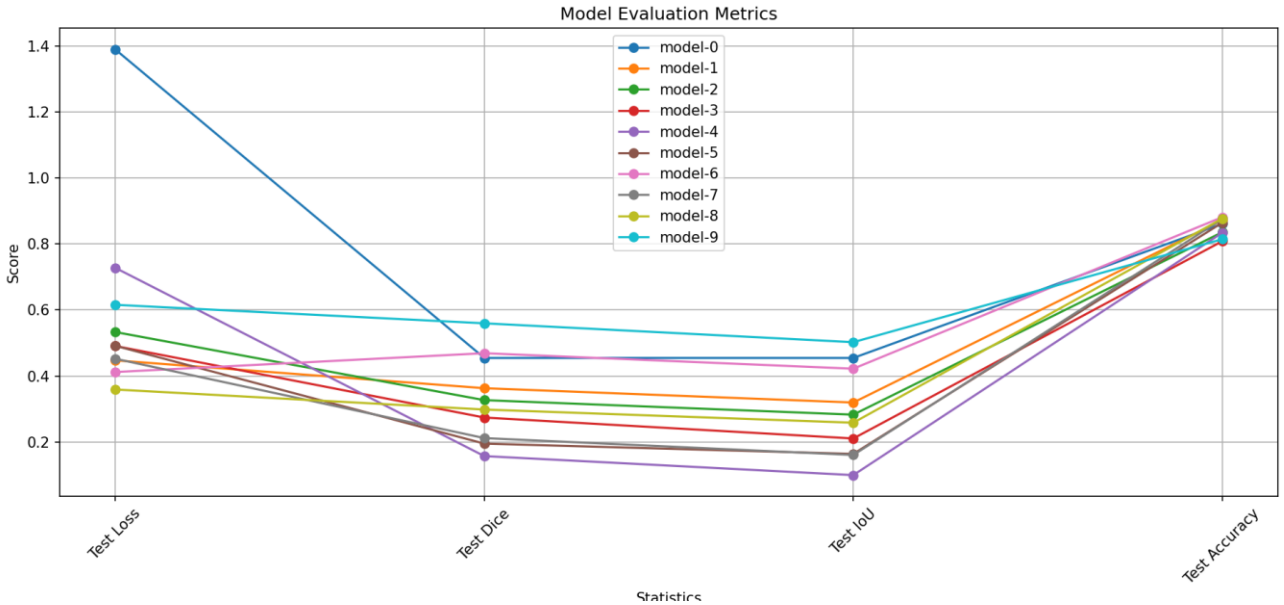


Figure 10 Result of testing each epoch of previous DeepLabV3-ResNet50

Although all its values are the best in epoch 4, the corresponding model (model 3) is not. Actually, the best in average can be said is the model-6 (epoch 7), which is logic since the model has very good metrics in there.

This are the result of testing each of the epochs of the FCN-ResNet-50 model:

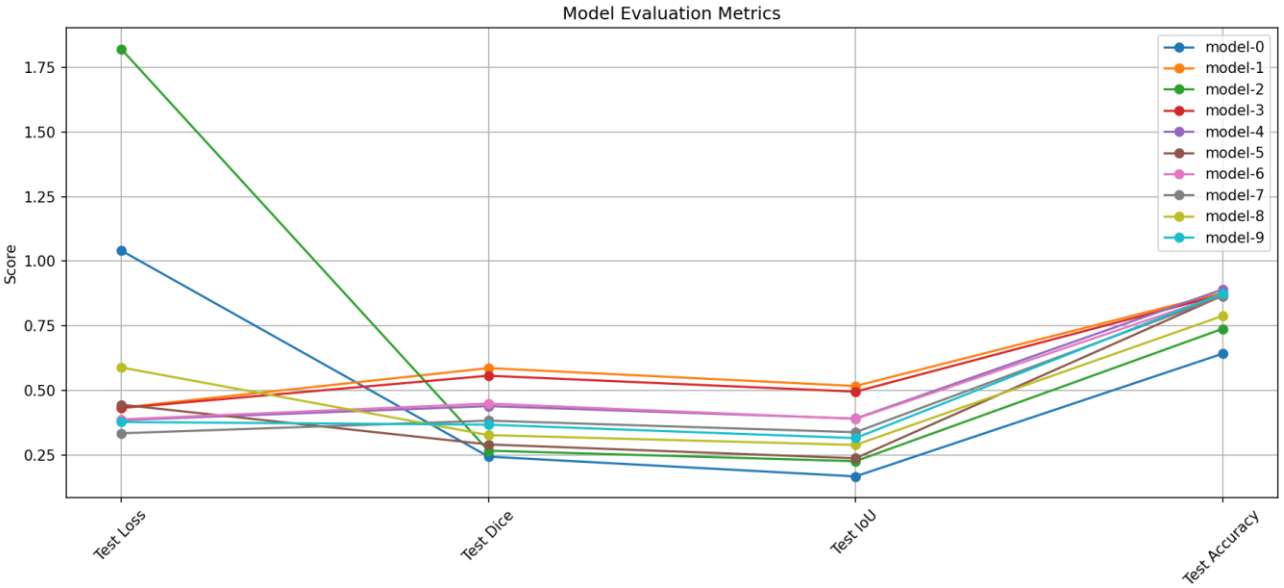


Figure 11 Result of testing each epoch of previous FCN-ResNet50

It is observed that the model-1 and model-3 corresponding to 2nd and 4th epochs are the ones that performs better than any other epochs in most of the metrics but in loss, even though these epochs are the worse when training. In loss, is the model-7 (8th epoch) the one that performs best, but this model performs very average in the rest of the metrics.

For concluding I have put together the model-6 of DeepLabV3-ResNet50 and FCN-ResNet50's model 1 and 3. And this is the result:

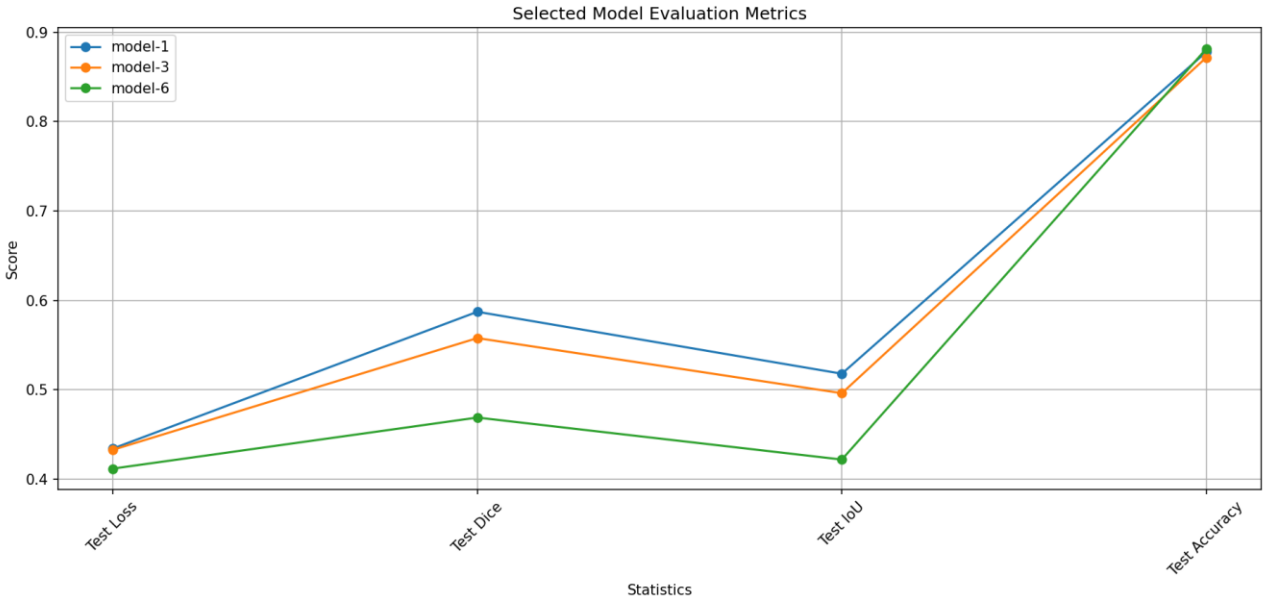


Figure 12 Comparation between the best epochs of previous models

It is observed that model-1 performs best in terms of Dice and IoU scores and matches model-6 in accuracy. However, model-1 has a higher loss value compared to model-6. Given that the difference in loss between model-6 and model-1 is relatively small compared to the significant differences in Dice Coefficient and IoU scores between model-1 and the other models, model-1 of FCN-ResNet50 can be considered the overall 'best' model.

6.2. Custom Models

As the ResNet as backbone as feature extraction wasn't performing good at all in the previous models. Also the heads such as DeepLabV3 or the FCN, when creating a model the objective was to find inspiration in other state-of-the-art models. This other model where SegNet and U-Net.

6.2.1. First

The first model was a simplified version of the SegNet model architecture. This first version has 6 layers. As a summary the first three layers captures context and features by downsampling the image, and upsampling reconstructs the detailed segmentation map. The model is in the `custommodels.py` file, and it is the `CustomSegNet3`.

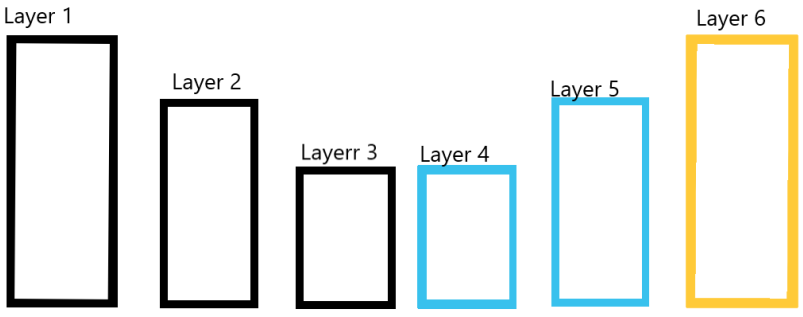


Figure 13 Architecture of the first custom model

At first, I created the `CustomSegNet` but it took almost 3 minutes just for making one step out of thirty two (for each epoch) in the training session. So, I needed to simplify it. The first reduced version (`CustomSegNet2`), it first three layers were composed of one layer of convolution and max pooling at the end. The last version has two convolutions before the max pooling. This is how the layer 2 of the last version looks like:

```
        # Layer 2
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

The first convolutional layer takes 64 input feature maps and produces 128 output feature maps. Normalizes the 128 feature maps applies the ReLU activation function that introduces non-linearity. After it there is a similar convolutional layer. Each convolutional layer has a filter of 3x3. Then the max pooling reduces spatial the dimensions by half retaining important features while discarding less important details. All layers from 1 to 3 reduces in half the dimensions of the image and apply only one convolution to the image in the simplest version and two convolutions in the `CustomSegNet3`.

The last three layers are the same for both reduced versions. The layers 4 and 5 apply an upsample to the image and then similar convolution of the previous layers. Layer four is as follows:

```
        # Layer 4
        self.layer4 = nn.Sequential(
            nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True),
            nn.Conv2d(256, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )
```

The upsample increases the dimension of the image by a factor of 2 using bilinear interiorization. And a similar convolution to the previous layers is applied

The last layer is similar to layer 4 and 5, but instead of applying normalization and ReLU, it applies SoftMax as activation function to produce the class probabilities for each pixel.

The result of training both reduced versions for 8 epochs was:

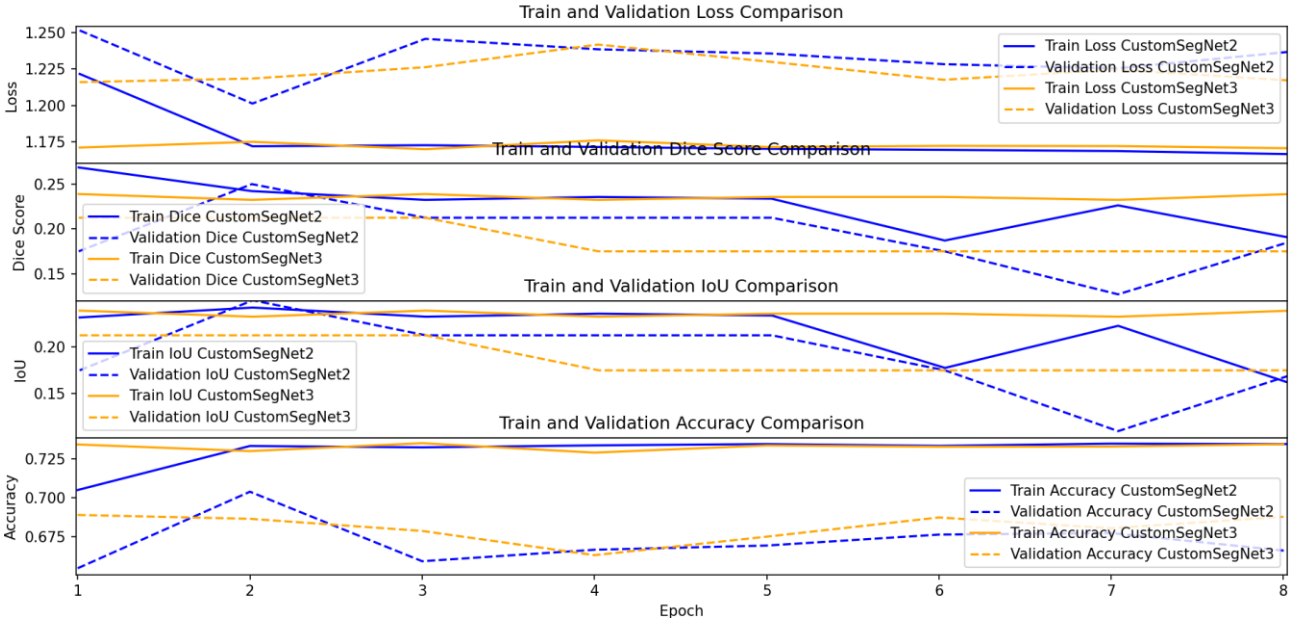


Figure 14 Comparation of training a first version model of the first custom model and a more complex version

The results in loss are bad and although the difference seems too big in the graphic, it is only of 0.05 in average, which the only thing that tells is that the models are not complex enough for the current problem. Both has a good accuracy

respect from other metrics. This can be because the models are not bad at all to determine some pixels, but the poor results in Dice Score and IoU, tell us that few times are when the model determines well a specific class that is not background. In general, both models perform similar although the `CustomSegNet2` is more instable. I would choose this last one since the other model is too stabilized and I don't think it will perform better in the future, since it is in the same values for each of the metrics for 4 epochs. This is the result of testing the `CustomSegNet2`:

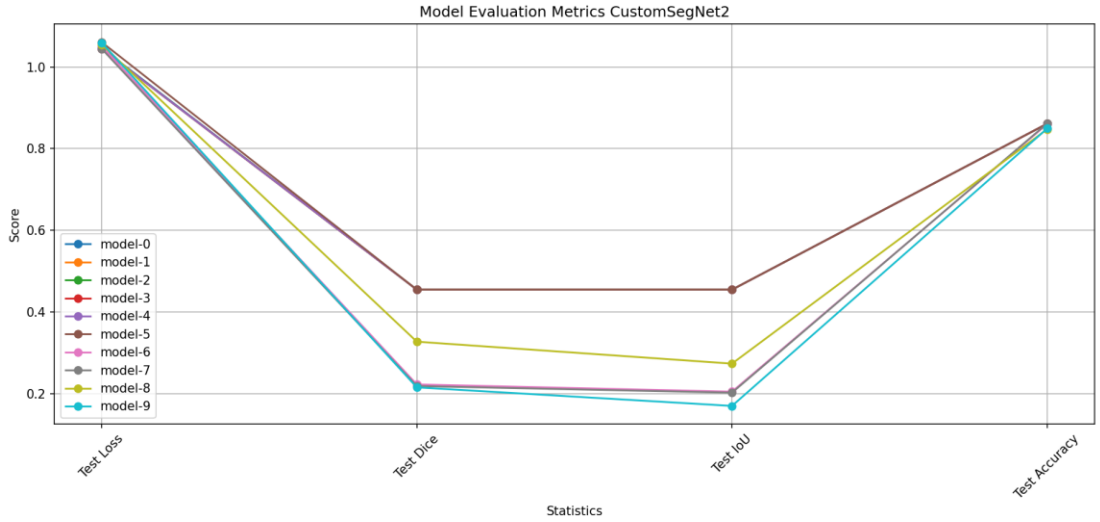


Figure 15 Evaluation of each epoch of the first version of the first custom model

All epochs perform equally in loss and accuracy which is not surprising a lot, since although during training was an instable performance, the metrics values where very similar all time. It is obvious that model 5 and 6 (epoch 6 and 7) performs similar in Dice and IoU but outstand from the rest. I would say that model 6 is better since is an epoch older.

6.1.2. Second

Due to the bad performance of the SegNet inspired models, this model is inspired in the U-Net model architecture. It is in the `customunetmodels.py` file, and it is the `CustomUNet2`.

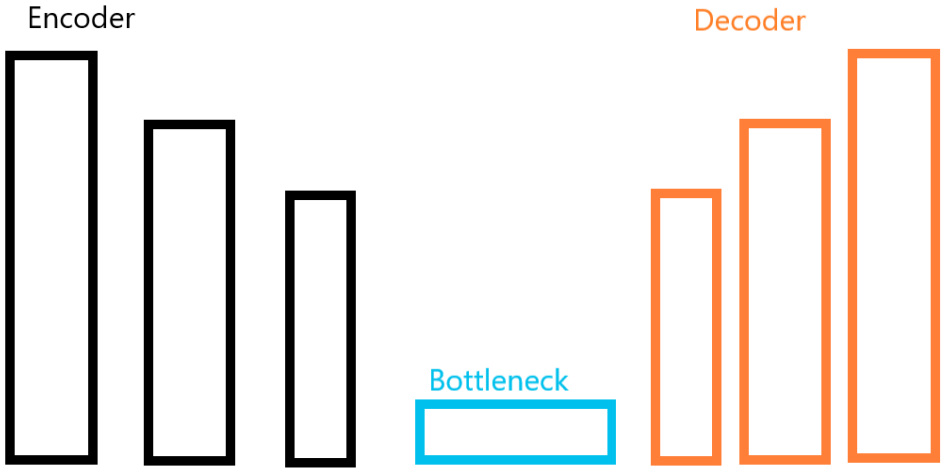


Figure 16 Architecture of the second custom model

The encoder is responsible for capturing the context and extracting features from the input image. It progressively downsamples the image while learning increasingly complex features at each layer. This layer has 3 layers in total that has this structure

```
        # encoder 2
        self.encoder2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

Two sets of convolutions, batch normalization and ReLU that has the same size and same filter size. At the end it applies an special reduction in half. Similar as in the SegNet.

The bottleneck is the deepest part of the network, positioned between encoder and decoder it main purpose is to capture high abstract and high-level features of the input image. By this stage, the spatial dimensions are significantly reduced, focusing on the most important features. It is like encoder but at the end it does not apply the max pooling. This is the bottleneck of the present model:

```
        # bottleneck
        self.bottleneck = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True)
        )
```

The decoder's primary function is to reconstruct the image from the abstract features captured by the bottleneck. It progressively upsamples the image back to its original dimensions. This is how the decoder looks like:

```
        # decoder 2
        self.decoder2 = nn.Sequential(
            nn.Conv2d(256 + 128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
        )
```

In this implementation there are connections between layers as it s in the U-Net architecture. Nevertheless there is another implementation that has the very same architecture but it does not have the connections between layers. That model is in the same `customunetmodels.py` file, and it is the `CustomUNet`. This is how connections are implemented:

```
    def forward(self, x):
        # Encoder
        enc1 = self.encoder1(x)
        enc2 = self.encoder2(enc1)
        enc3 = self.encoder3(enc2)

        # Bottleneck
        bottleneck = self.bottleneck(enc3)

        # Decoder
        dec3 = self.decoder3(torch.cat([bottleneck, enc3], dim=1))
        dec2 = self.decoder2(torch.cat([dec3, enc2], dim=1))
        dec1 = self.decoder1(torch.cat([dec2, enc1], dim=1))
```

```
        # Final convolution
        output = self.final_conv(dec1)

        return output
```

Basically a connection is when the input of the decoder is a combination of the output of the corresponding encoder and the output of the previous layer. The input when there is not connection is just the output of the previous layer. This is the result of training both models:

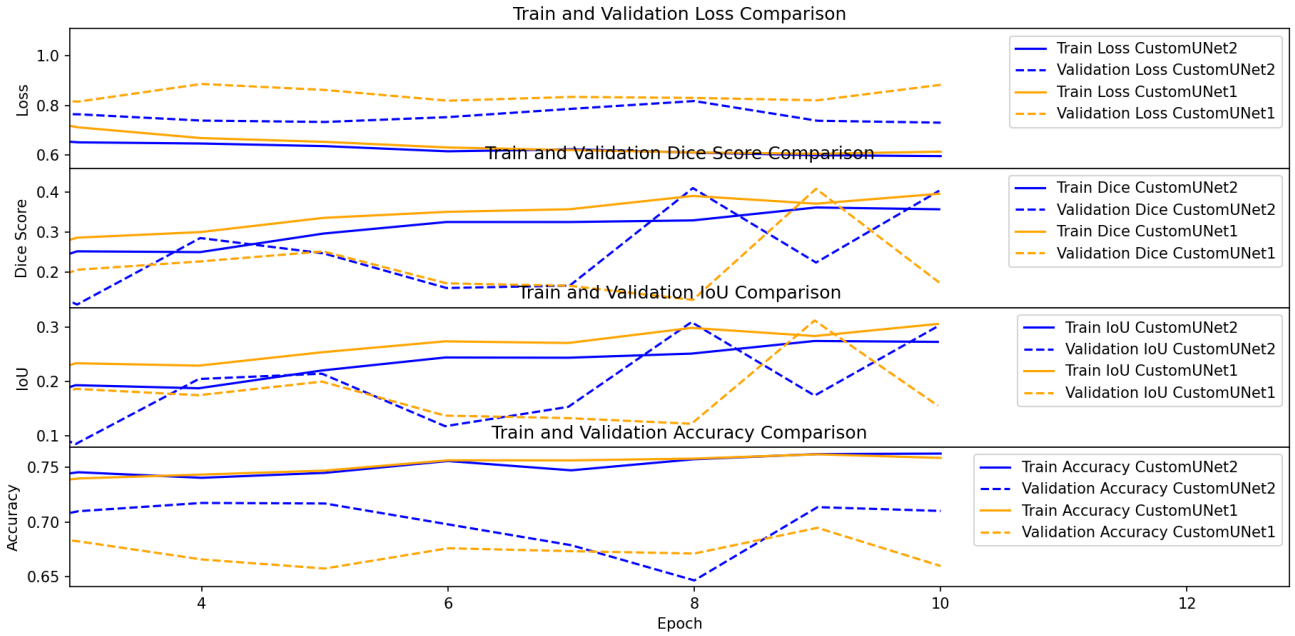


Figure 17 Comparation of training a first version model of the second custom model and a more complex version

Both models perform similarly, but the connections seem to work a bit better in validations. The performance in loss is quite similar for both in training, but the connections are slightly better, nevertheless the value is still high (even in training). The Dice and IoU respect from the pixel accuracy does not differ a lot not only because specially the dice is not low but above all because the accuracy is not high at all. This can suggest that the model is saying that lot of the pixels are classes, but they are not. This means that the model is failing specially in determining the background. Since the `CustomUNet2` performance is better, especially in validation where has a very stable but with the perspective of decreasing, I though could be the best model, so I tested and here are the results:

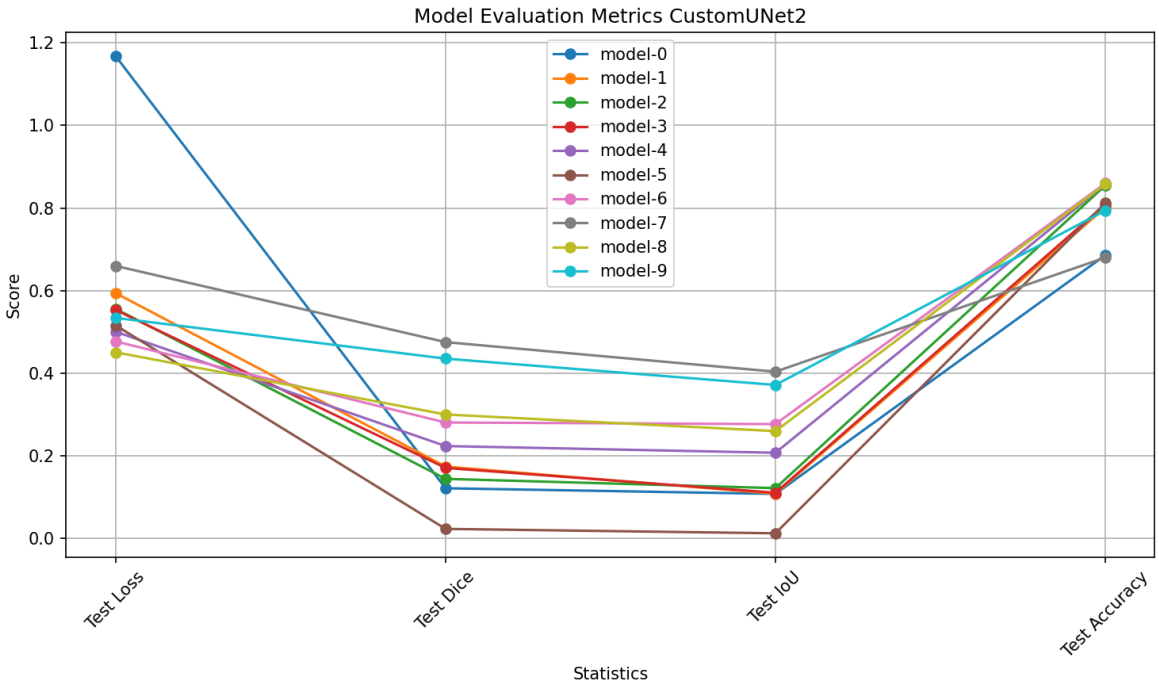


Figure 18 Evaluation of each epoch of the second version of the second custom model

Although the better performance in Dice and IoU is in model-7 (epoch 8) and model-9, but they perform poorly in Accuracy and specially in loss. Nevertheless, the best performance in loss and in accuracy is of the model-8 (epoch 9). This model performs like the one of epoch 7, but slightly better than that one in every metric. The difference between model-8 and model-9 is more less the same in loss, where model-8 is better, and in Dice and IoU where is the model-9 the best of both. So can be said that the model-8 and model 9 are the best model of `CustomUNet2`. This reduced version of the U-Net, although has been a reduced version suggest that with a more complex model and some changes in the architecture, this problem could be resolved better.

6.3. Conclusion

After comparing the best models of state-of-the-art models and the two best custom models, can be said which is the best model. The comparation of the best model of CustomUNet2 (model-8) with the best model of CustomSegNet2 (model-6) and the best model of FCN-ResNet50 (model-1) is as follows:

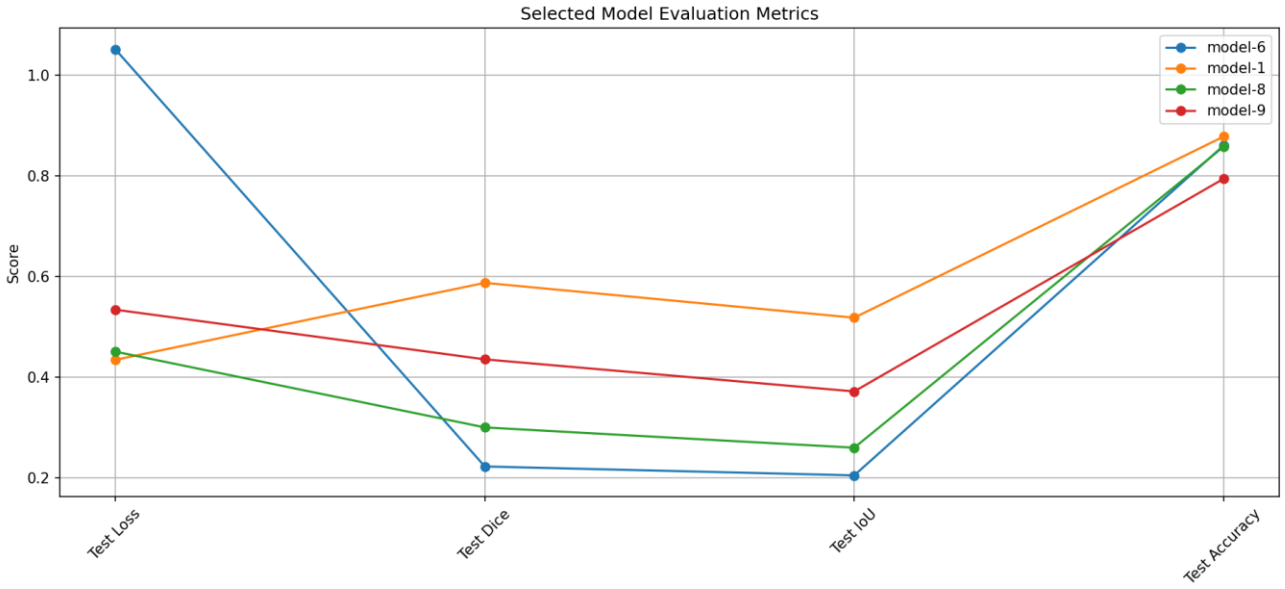


Figure 19 Comparation of the best models (custom and state-of-the-art)

CustomUNet2 (model-8) is the best model I have done, since has a very good testing lost and perform slightly better than the CustomSegNet2 (model-6) in the other metrics. The three models have same accuracy which means they got right many pixels, but this does not mean that got right the pixels they had to, that are the ones that correspond to the damages of the car. This can be known thanks to the IoU and the Dice Coefficient. And since this metrics are low for every of the models, especially the custom, we can assert that the models are still not understanding which pixels correspond to the classes.

With so can be say that the U-Net architecture has potential in the present problem, since a reduced and differently implemented version has perform "like" a complete version of two different ResNet implementations. Nevertheless, the best model I have trained is the FCN-Resnet50.

7. Custom Dataset

I have prepared a custom dataset with six images from internet and three generated by AI, since taking photos of cars that have severe damages it is quite a challenge. This images after being labeled has been passed to the custom model of CustomUNet2 (model-9) to see it's predictions. This choice was made because, while FCN-Resnet50 demonstrates superior performance, CustomUNet2 performs enough good and is customizable, allowing for potential future enhancements.

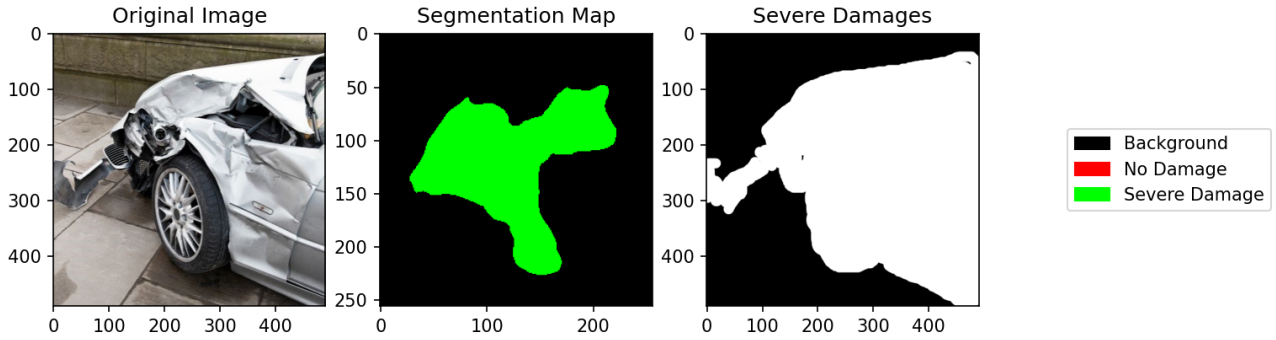


Figure 20 Best model first prediction

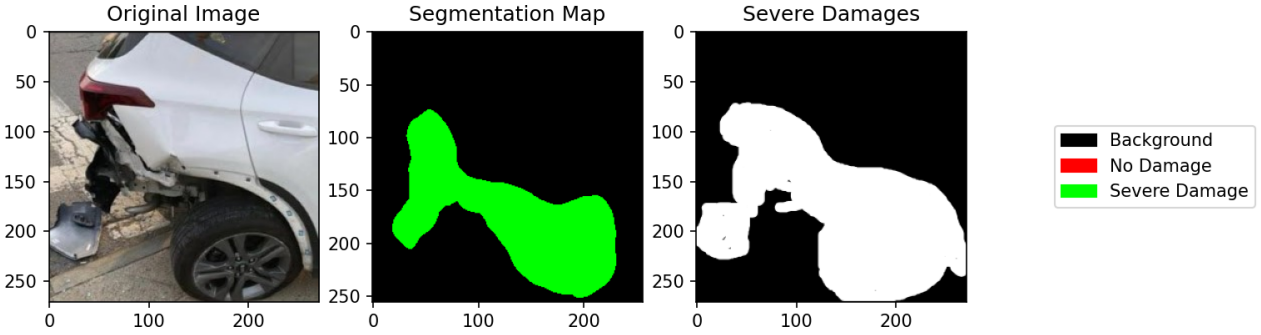


Figure 21 Best model second predictions.

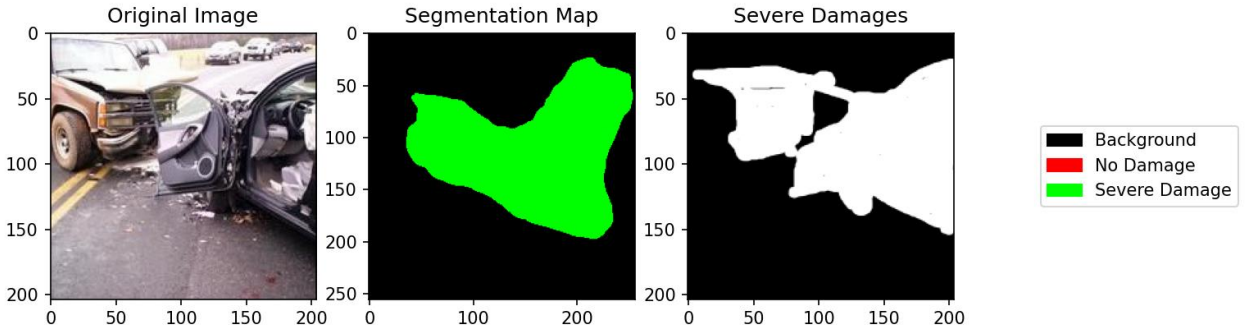


Figure 22 Best model third predictions.

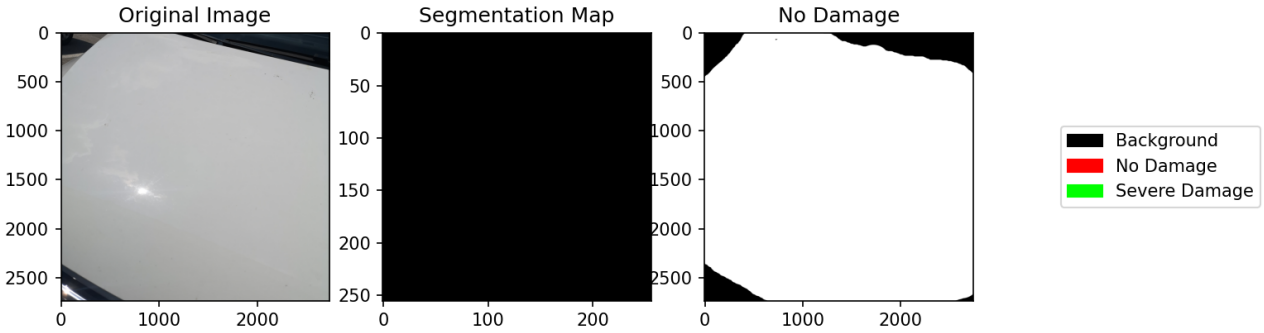


Figure 23 Best model fourth predictions.

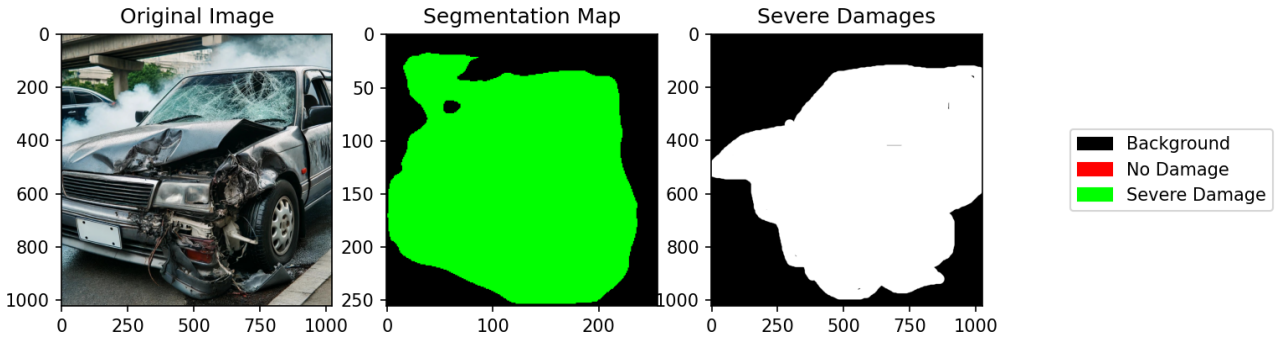


Figure 24 Best model fifth predictions (AI Image).

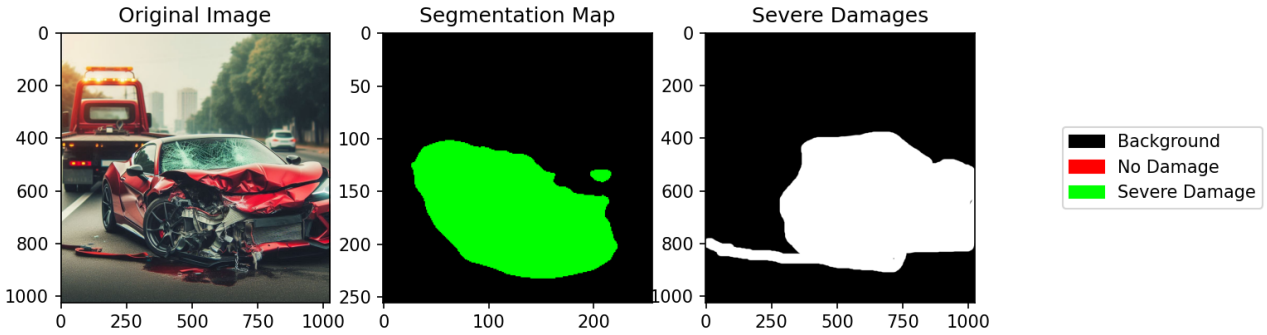


Figure 25 Best model sixth prediction (AI image).

In making some observations, the model seems to not know what exactly “no damage” means. When there is a separation between cars and pieces of the car it surprisingly predicts it well. This is probably because are same color pixels separated and model has already learned that.

8. Repository

In this section will be explained what the [MrCharlesSG/ComputerVisionProject (github.com)]() repository has and how and why each part of the repository is used for. In the repository can be find all the dataset, metrics and models presented in this document, all together with the scripts used.

- Custom-dataset, labeled images presented in the Custom Dataset section
- Project.pdf, this document in pdf and Project.docx this document in Word Document format

- In code folder can be find every script and python code used in the project
    - scripts
        - coco-to-mask.py, transform a COCO JSON annotated dataset into a mask format.
        - dataset-divider.py, select the custom percentage of a dataset.
        - model-predicts-custom-images.py, a model predicts with the custom dataset. This script has been used in the seventh section (Custom Dataset) to evaluate and plot the model predictions, all together with the grown truth and the correct label.
        - All the files starting in plot, show the graphics presented in the document.
        - All files starting in testing, test a model or all of the models of the epochs.
        - visualize-COCO-into-mask.py, show the image and its annotations from the COCO JSON format.
    - custom_seg_net_models.py, store the models that uses the SegNet architecture.
    - custom_unet_models.py, store the models that uses the U-Net architecture
    - datasetseg.py, store the class that is used to transform images and mask, so models can read them.
    - hyperparams.py, store the hyperparameters sets.
    - main.py, script that starts a training
    - metrics_functions.py, store the metrics implementation.
    - train_custom_models.py and train_torch_models.py, are used but the main.py to train custom or state-of-the-art models.
    - utils.py, store some utils functions.