Car Damage and Repair Cost Estimation Using Image Segmentation

Alexandre Baptista, nr fc64506 & Carolina Rodrigues, nr fc62910
M.Sc. Data Science
Deep Learning class 2024/2025

Abstract

Accurate vehicle damage assessment is fundamental for automotive industries to manage insurance claims and repair pricing, yet this process still often depends on manual, on-site inspections that are time-consuming, costly, and prone to human subjectivity.

Some early digital solutions are being developed, showing that deep learning can detect damage in images and provide repair estimate, however these tools remain largely unused by insurers, only cover a few and frequent classes (scratch, dent, crack) often miss and omit rare damage patterns, and it ignores the size and severity of each instance therefore it does not translate detection into a reliable repair cost estimates. This limitation leaves a clear gap for a finer-grained, per-damage system that scales to more categories and adjusts cost to the actual extent of each defect.

To address this gap, we aim to build a YOLOv8-based segmentation pipeline that can turn a customer-supplied photo into a structured damage report. The system detects and masks multiple exterior damages, classifies their type and severity, and provides an appropriate repair-cost range. Using the Car Damage Severity Detection (CarDD) dataset extracted in YOLOv8 format, that allows multi-label polygon masks per image, three main experiments were conducted: (1) hyper-parameter search on the balanced YOLOv8m-seg baseline, (2) impact of augmentation, and (3) model-size comparison (s, m, l) across 10–50 epochs.

We found that the best performer, YOLOv8-l-seg at 50 epochs, reached mAP50 = 0.45 for bounding boxes and 0.44 for masks, delivering accurate and consistent detection. These results confirm the feasibility of rapid, photo-based damage triage that reduces on-site inspection costs and improves claim cost estimations. Remaining challenges include class imbalance and translating mask geometry into precise costs. Future work will add a regression head that leverages mask size, location and learned severity cues to output a per-damage repair-cost range.

1. Introduction

Accurate assessment of vehicle damage underpins insurance claims and repair pricing, however manual, on-site inspections remain the norm for vehicle-damage assessment, making claims handling slow, costly and subject to human bias. Recent AI pilots (such as Solera's *Guided Image Capture* [1]) show that photo-based triage is feasible, but uptake is limited. These tools cover only the most frequent classes and provide coarse, panel-level cost estimates, neglecting rare defects and severity differences.

To cover this gap, we intend to apply **instance-segmentation task** and **damage detection** using YOLOv8 segmentation models.

The YOLO ("You Only Look Once") family of models offers real-time object detection by processing an entire image in a single pass, achieving impressive speed without sacrificing accuracy [2]. The latest release, **YOLOv8**, adds improved backbones and training tricks, while **YOLOv8-SEG** extends the framework with instance-level masks combining YOLO's detection efficiency with pixel-accurate segmentation [3, 4]. This makes the architecture well-suited for fine-grained vehicle-damage localization.

The goal is to build a YOLOv8-based segmentation pipeline that, given one or more images of the same damaged vehicle as input, detects and classifies multiple vehicle damage as well as estimating the corresponding repair costs. The system will:

- Detect and localize exterior damage precisely on the vehicle;
- Categorize types of damage (scratches, dents, cracks, etc.) and severity;
- Estimate a repair-cost range tailored to each defect's size and seriousness.

The project will be available in github: https://github.com/Princesacorderosa/CarDamage_DL

This is particularly interesting for insurance and automotive industries, where this process is often done with manual inspections. With this project we intend to improve the accuracy and efficiency of damage detection and cost prediction. This makes the inspection process faster, more reliable and cost-effective, cutting the on-site physical inspections expenses.

1.1. Research and background

We will review previous academic papers on detection of car damage (such as R.E. van Ruitenbeek, S. Bhulai, *Convolutional Neural Networks for vehicle damage*

detection [5]. This study develops a model to locate vehicle damages and classify them into twelve categories using multiple deep learning algorithms.

And car industry case study on how AI and image recognition technology can improve insurance claims estimation processes. [6]

Also, current investigations and practical applications within the industry have revealed the capabilities of artificial intelligence and deep learning in the identification and categorization of vehicle damage. Research conducted by R.E. van Ruitenbeek and S. Bhulai employs convolutional neural networks (CNNs) to identify and classify car damages into various categories, yielding encouraging results in terms of accuracy and classification. In a similar vein, organizations such as Audatex utilize image recognition technologies to streamline the processing of insurance claims by evaluating vehicle images and assessing the severity of damage.

Nevertheless, these methodologies encounter numerous challenges. A significant number of models struggle to adapt to real-world variations, including lighting conditions, image quality, and occlusions. Furthermore, they may be unable to identify rare or subtle forms of damage due to a lack of adequate training data. Moreover, most of these systems exhibit a deficiency in interpretability, complicating the justification of decisions—an aspect that is particularly crucial in regulated sectors such as insurance. Concerns regarding computational efficiency and the ability to adapt to new types of damage also pose obstacles for large-scale implementation.

2. Dataset

For the initial phase of this project, we employed the **"Damaged Vehicle Images"** dataset available on Roboflow [7]. This collection contains 5 000 + annotated photographs, each exported in COCO JSON format with bounding boxes and class labels for common damage types such as *scratches*, *dents*, and *broken windshields*.

This original dataset was distributed as follows: Train set: 94% (4,777 images), Validation set: 4% (193 images) and Test set: 2% (102 images). But, to ensure a more balanced and standardized evaluation, we created a new project on Roboflow, uploaded the dataset, and used the platform's built-in tools to apply a new **80/10/10 split** for training, validation, and testing.

However, this generated a folder called **Damaged-Vehicle-Images-1/** containing train/, valid/, and test/, folders where each image was organized by damage category. An example of some images in the train folder in Figure 1.

Figure 1 - sample images from the training set

However, when running a 15-epoch plain CNN it reached 75% training accuracy but only 5% validation accuracy. This represents a clear case of overfitting or potential data leakage, which made us believe that that the original dataset might have contributed to the poor generalization performance.

Therefore, we decided to first revise our dataset before altering the modeling strategy, replacing it with a new and more structured dataset: "***Car Damage Severity Detection – CarDD***" [6]. With this substitution the goal was to rerun the model under better-controlled conditions and observe whether these issues could be mitigated.

2.1. Dataset structure

The new dataset was extracted in **YOLOv8** format, which includes:

- **Images** and corresponding **TXT annotation files** for object detection;
- A **YAML configuration file** that defines label names and IDs;

Unlike the previous dataset, this one not only provides **class labels** but also specifies the **location** and allows a count of each damage type per image.

The YOLOv8 formatted dataset was a better choice than the original because its multi-label polygon annotations and standardized, leakage-free split provide a richer and cleaner training signal than the single-class, folder-based COCO export. Which is particularly useful for both object detection and tasks such as cost estimation.

The dataset is split in a 70/20/10 ratio, with: **Train:** 1395 images and labels; **Validation:** 401 images and labels; **Test:** 204 images and labels. Having a total of **11 damage categories**, including:

- 0 - car-part-crack, 1 - detachment, 2 - flat-tire, 3 - glass-crack, 4 - lamp-crack, 5 - minor-deformation, 6 - moderate-deformation, 7 - paint-chips, 8 - scratches, 9 - severe-deformation, 10 - side-mirror-crack.

An example of some images with the corresponding labels is shown in Figure 2 below.

Figure 2 - Sample of images and labels of the new dataset

2.2. Preprocessing and Sanity Checks

To ensure dataset integrity and avoid potential sources of error, such as data leakage, image corruption, or class imbalance, we conducted some **sanity checks** before training our models.

a) Data Leakage Detection

As a first step, we verified whether there were any duplicate or near-duplicate images shared between the train and valid splits.

To do this, the process scanned all .jpg files in both folders, then for each image, it computed a SHA-1 hash to detect exact byte-level duplicates and a perceptual hash (pHash) to detect similar images, allowing for minor differences of Hamming distance ≤ 5 bits

Then compared the hashes across both splits and it would flag and list any image pairs that were exact copies (with identical SHA-1 hashes) or near-duplicates (pHash Hamming distance ≤ 5).

Result:

Since no exact duplicates or near-duplicates were found between the train and valid sets, we can confirm that there is **no data leakage**

b) Image corruption

To ensure file integrity, we scanned all images in each split (train, valid, and test), using *PIL.Image.open()* function, and calling *img.verify()* on each file.

This allows us to validate and check whether each image file can be opened, without fully loading it into memory. If any exception is raised, due to file corruption or unreadability, the corresponding file is flagged and listed.

Result:

No corrupt or unreadable images were detected. Indicating that all images across the sets are intact and readable.

c) Class and Label distribution

We analyzed the label distribution by counting the total number of bounding boxes per class in each data split (train, validation, and test). This allows us to detect potential class imbalance and verify if each class has at least 10 samples, particularly in the validation set.

To do this, a dataframe was generated, with one row per split (train, val, test), one column per class and each cell containing the number of bounding boxes for that class in the corresponding split.

Result:

From the resulting dataframe, we observe that most classes have sufficient representation across splits. However, some classes such as *"detachment"* and *"side-mirror-crack"* have counts just above the minimum threshold of 10 samples in the validation set. This imbalance is visualized in Figure 3.

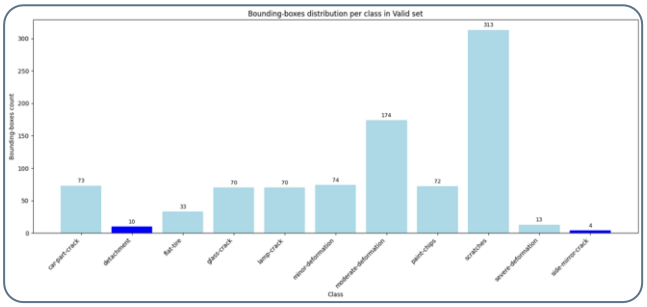

Figure 3 – Histogram of class distribution. "detachment" and "side-mirror-crack" classes are highlighted in dark blue

This suggests a **class imbalance**, which we plan to address during training using one or more of the following strategies:

1. **Redo the dataset split** using stratified sampling by class and shuffling (so 80/10/10 split instead of relying on the default 70/20/10 split);
2. Apply **targeted data augmentation** to underrepresented classes using transformations;
3. **Merge rare classes** into similar ones to increase class support and simplify the prediction.

Once these strategies are applied, we will re-run the training to confirm if the validation accuracy, which would confirm effective mitigation of the imbalance. These steps are essential to ensure fair evaluation and to reduce bias toward the most frequent damage categories.

3. Methodology

The methodology integrates YOLOv8 segmentation models, and the workflow is represented in Figure 4.

The experiments conducted are as follows:

1. **Baseline + Hyper-parameter search**: Tune learning rate and optimizer for YOLOv8m baseline.

2. **Augmentation test:** Baseline Vs. Albumentations enabled.
3. **Model-size comparison**: Best YOLOv8m against YOLOv8s and YOLOv8l, each trained for 10, 25 and 50 epochs.

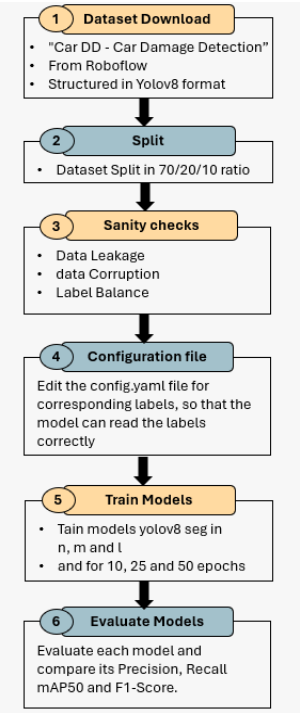

Figure 4 – Methodology workflow

4. Experiments and Results

Re-evaluation of Baseline Model on New Dataset

Before implementing the strategies mentioned above, we re-ran the same model used with the first dataset, on the new CarDD dataset to evaluate whether the overfitting issue would persist. And compared model performance across 15, 50 and 100 epochs.

Table 1 - Training Results (Keras CNN, no augmentation)

Epochs	*Training Accuracy*	*Training Loss*	*Validation Accuracy*	*Validation Loss*
15	0.4359	1.4874	0.3317	**1.9439**
50	**0.9961**	0.0180	**0.3940**	**5.4632**
100	1.0000	**0.0028**	0.3541	**8.0316**

As shown in Table 1, we can see that:

- After 50 epochs: training accuracy approached 99%, while validation accuracy plateaued around 39%, and validation loss increased dramatically from ~2 to 5.5.
- After 100 epochs: the training loss dropped near zero, indicating the model was memorizing the training data, and validation loss exceeded 8.0, confirming overfitting.

Even when using the improved CarDD dataset, the baseline model exhibited clear signs of overfitting when trained without data augmentation and without addressing class imbalance.

- **Accuracy curves:** show that training accuracy climbs toward 1.0 starting around epoch 30, while validation accuracy flattens between 0.35–0.40, likely limited by class imbalance or lack of regularization techniques.
- **Loss curves:** the training loss decreases steadily until around epoch 30, where it reaches near zero. This coincides with the training accuracy curve climbing toward 1.0, confirming overfitting. Validation loss remains close to training loss during the first ~10 epochs, however, after approximately epoch 15, validation loss begins to rise steadily, forming a clear upward slope, a classic sign of high variance and overfitting.

The results obtained strongly support the need to introduce additional **regularization strategies** to improve model generalization and address class imbalance.

But, before implementing and training more advanced models, we will first apply these strategies incrementally to the baseline setup starting with class balancing techniques and re-run training to evaluate their impact.

Subsequently, we will incorporate the full regularization pipeline and train the more complex architectures, allowing us to **compare performance across all configurations** and select the most effective approach based on empirical results.

4.1. Baseline model and Hyper-parameter

There are multiple pre-trained YOLO model we could choose (*n*, *s*, *m*, *l*, or *x* versions) those versions are based on size, where smaller ones (n and s) are more adequate for edge and mobile deployment as the larger ones (l and x) versions are good with accurate detection.

The summary for each version is:

- ***YOLOv8m-seg**: **191** layers, **27,285,968** parameters*
- ***YOLOv8s-seg**: **151** layers, **11,821,056** parameters*
- ***YOLOv8l-seg**: **231** layers, **45,997,728** parameters*

The YOLOv8**m**-seg model was chosen as the baseline because of its well-balanced architecture, which strikes a compromise between computational efficiency and performance accuracy.

In comparison to smaller models such as YOLOv8**s**-seg and larger variants like YOLOv8l-seg, the **YOLOv8m-seg** offers an appropriate number of layers and parameters. This configuration renders it effective for both object detection and segmentation tasks without imposing excessive resource demands.

Consequently, it is particularly suitable for applications that require high precision while also ensuring a reasonable

inference speed.

With the baseline, we will fine tune it until we find the best hyper parameters to use, the model yolov8m-seg was run in 10 epochs and for each run we changed the optimizer and learning rate, results were noted in the Table 2 below.

Table 2 - Training Results Highlighted in blue → baseline model; in green → best model parameters (underlined and in bold, are the first- and second-best performing metrics, respectively; (B) = bounding box detection, (M) = segmentation masks

Model	Learning rate:	Optimizer:	time(s):	precision(B)	recall(B)	F1(B)	mAP50(B)	precision(M)	recall(M)	F1(M)	mAP50(M)
YOLOv8m -seg	0.01	* not specified	584,54	0,415	**0,412**	0,413	**0,391**	0,426	0,396	0,411	**0,384**
	0.01	**Adam**	565,15	0,462	0,345	0,395	0,323	0,470	0,332	0,389	0,304
	0.05	**Adam**	541,25	0,554	0,253	0,347	0,223	0,562	0,244	0,340	0,211
	0.005	**Adam**	525,50	0,342	0,374	0,357	0,348	0,338	0,364	0,350	0,340
	0.01	SGD	**519,65**	0,425	0,461	0,442	0,412	0,423	0,451	0,437	0,403
	0.01	**Radam**	508,08	**0,510**	0,376	**0,433**	0,380	**0,511**	**0,369**	**0,428**	0,363

The chosen baseline model → YOLOv8m: Balance between speed and accuracy.

Hyper-parameter sweep → (lr ∈ {0.005, 0.01, 0.05}, Adam vs RAdam).

What we concluded was that the best hyperparameters were: **learning rate: 0.01** with **optimizer: RAdam.**

4.2. Augmentation test

Since sanity checks revealed moderate class imbalance, we decided to test YOLOv8s built-in **Albumentations** (by activating it with augment=True) and compared results to the non-augmented baseline. A sample of the images with the augmentations mentioned are as shown in the Figure 5.

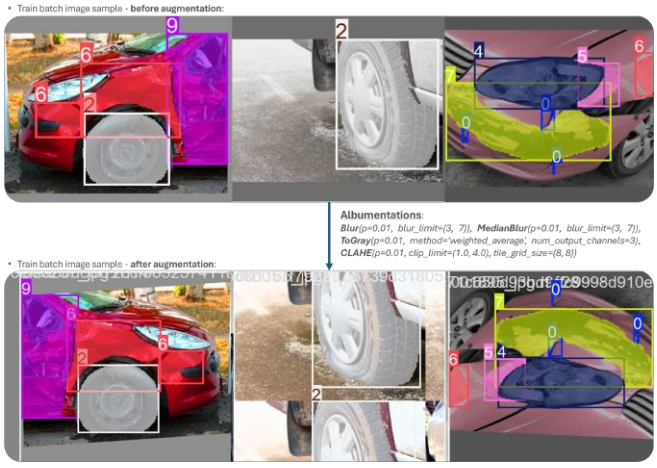

Figure 5 – Image sample with the augmentations applied

From the results in Table 3 we observe that with 25 epochs, Albumentations slightly improved precision and mAP but decreased recall. In 50 epochs, improved in precision and mAP for object detection (B) but decreased for the masks (M).

Table 3 - Results of Baseline with vs without augmentation

Yolov8m

Yolov8m baseline	precision(B)	recall(B)	F1(B)	mAP50(B)	precision(M)	recall(M)	F1(M)	mAP50(M)
10ep - No augmentation	0,51017	0,37605	0,4330	0,38042	0,51122	0,36858	0,4283	0,36333
10ep - With augmentation	0,47033	0,39487	0,4293	0,38585	0,46334	0,38714	0,4218	0,37576
25ep -No Augmentation	0,51905	0,42047	0,4646	0,40831	0,52865	0,40696	0,4599	0,40015
25ep -With Augmentation	**0,57255**	0,39713	0,4690	**0,40846**	**0,58204**	0,39231	**0,4687**	**0,40623**
50ep -With Augmentation	**0,57475**	0,39379	0,4674	0,41227	0,55029	0,38597	0,4537	0,39725

This happens because, with more training time the detector learns object centers and box extents well, so **bbox precision/mAP keep improving**. Mask heads, however, must predict exact contours, so after many epochs they over-fit to augmented artefacts such as jagged edges or color shifts. The result is tighter boxes but **messier, less accurate masks**, hence lower mAP for the "M" columns.

In order to better understand these specifications, we made some predictions for the model yolov8m-seg for 25 and 50 epochs, as shown in Figure 6 and 7, respectively.

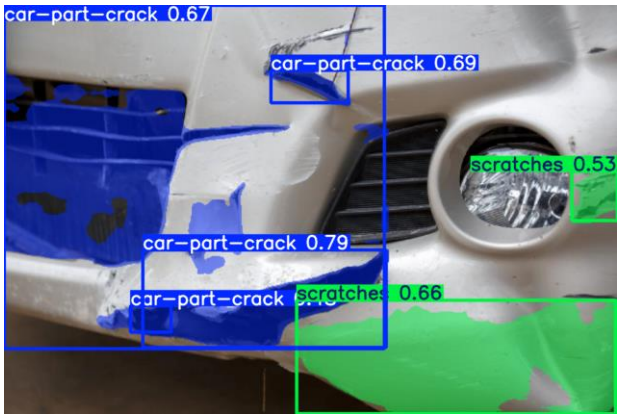

Figure 6 – Image prediction for V8m in 50 epochs

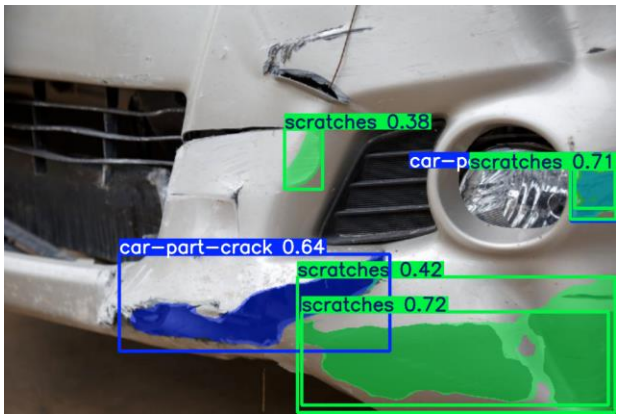

Figure 7 – Image prediction for V8m in 50 epochs

As we can see in Figure 6, for 25 epochs there are fewer boxes with some low-confidence detections (*scratches 0.38*). Masks are fairly smooth but occasionally miss thin cracks, confirming the lower precision but higher recall.

And in Figure 7, for 50 epochs, there are more boxes with higher scores (*car-part-crack 0.79*), showing the precision boost. Masks cover larger areas but spill over panel edges and miss fine detail, illustrating the drop in mask mAP.

Result:

Visually, the 50-epoch model is "sure" about where damage is but less exact about **how much** of the panel is affected—exactly what the metric trends indicate.

4.3. YOLOv8m Vs. YOLO (s and l) versions

Finally, once discovered the best hyper parameters, and with proven improvement with built in augmentations, with this experiment we intended to explore if under the same conditions as the best performing baseline, using the other yolo versions such as yolov8s-seg and yolov8l-seg for 10, 25 and 50 epochs.

Table 4 - Results of s and l versions with augmentation

Yolov8s

Yolov8s	precision(B)	recall(B)	F1(B)	mAP50(B)	precision(M)	recall(M)	F1(M)	mAP50(M)
10ep -With Augmentation	0,31202	0,41965	0,3579	0,36638	0,57452	0,34099	0,4280	0,35275
25ep -With Augmentation	0,40805	0,49807	0,4486	0,40804	0,39865	0,47358	0,4329	0,39366
50ep - With Augmentations	0,4838	0,4082	0,4428	0,3888	0,4987	0,4051	0,4471	0,3832

Yolov8l

Yolov8l	precision(B)	recall(B)	F1(B)	mAP50(B)	precision(M)	recall(M)	F1(M)	mAP50(M)
10ep -With Augmentation	0,41829	0,39295	0,4052	0,3635	0,37203	0,34402	0,3575	0,25432
25ep -With Augmentation	0,40631	0,51403	0,4539	0,4207	0,39572	0,49504	0,4398	0,40881
50ep - With Augmentations	**0,50113**	0,45456	**0,4767**	**0,44807**	**0,50591**	**0,45789**	**0,4807**	**0,43559**

The values for the validation loss and training loss for each of these models were ploted so that we could see its behavior trough the epochs, separetad into Segmentation loss curves Figure 8 and Box loss curves Figure 9.

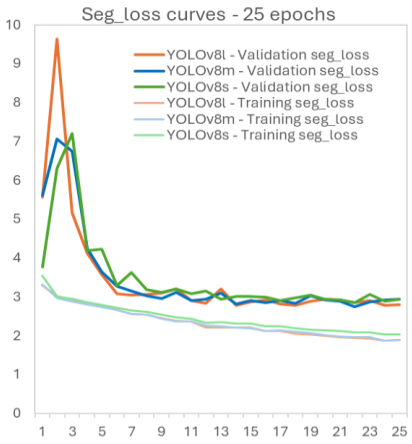

Figure 8 - Results of train and valid Segmentation loss curves

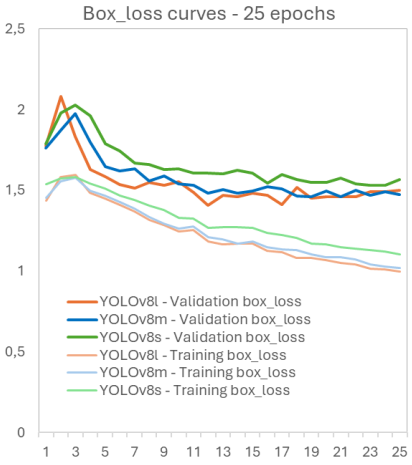

Figure 9 - Results of train and valid Box loss curves

Result

Amongst all models tested, **YOLOv8-l trained for 50 epochs** delivered the most consistent and best overall performance. Therefore, is the one chosen for final predictions**.**

5. Final Predictions

For the final prediction we used YOLOv8l trained for 50 epochs. Resulting in the following images:

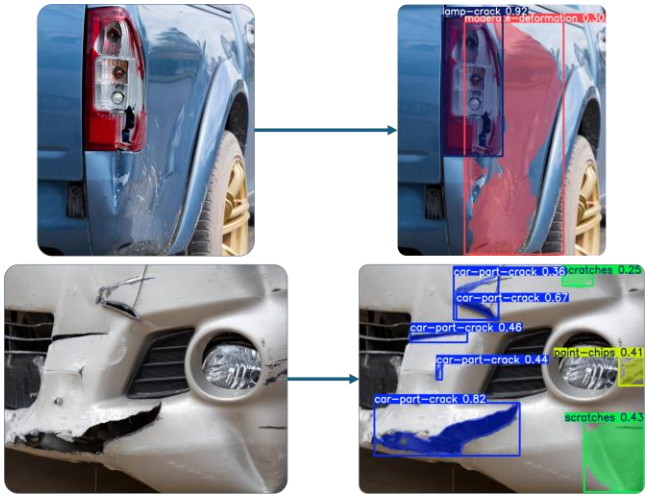

Figure 10 - Results of train and valid Segmentation loss curves

6. Conclusions

Although we expected the larger models (YOLOv8l) to outperform the smaller ones, results were that: only after 50 epochs it improved mAP only marginally over the medium model (YOLOv8m) yet required far more training time and GPU memory.

Future work: Extend the model for precise cost-estimation → using mask size to predict damage-specific repair-cost estimates based on the size, location and severity of the affected area.

7. References

[1] Solera. *Vehicle claims – Guided Image Capture*, from https://www.solera.com/solutions/vehicle-claims/.

[2] Ultralytics. (2025). *Ultralytics documentation (YOLO).*, from https://docs.ultralytics.com/pt/.

[3] Ultralytics. (2025). YOLOv8 models: Performance metrics, from https://docs.ultralytics.com/pt/models/yolov8/ .

[4] Aytaç, M. (2023, April 4). How to segment with YOLOv8.Medium. https://medium.com/@Mert.A/how-to-segment-with-yolov8-f33b1c63b6c6

[5] R. E. van Ruitenbeek and S. Bhulai, "Convolutional Neural Networks for Vehicle Damage Detection," Machine Learning with Applications, vol. 7, 2022. [Online]. Available: https://doi.org/10.1016/j.mlwa.2022.100332

[6] Celebal Technologies, "Leading Insurance Provider Leverages AI-Powered Vehicle Damage Detection," Case Study. [Online]. Available: https://celebaltech.com/case-studies/leadinginsurance-provider-leverages-ai-powered-vehicle-damage-detection

[7] Roboflow dataset, "Car Damage Severity Detection – CarDD" [Online]. Available: https://universe.roboflow.com/car-damage-detection-cardd/car-damage-severity-detection-cardd

[8] YOLOv8.org. "How to Fine-Tune YOLOv8." [Online]. Available: https://yolov8.org/how-to-use-fine-tune-yolov8/

[9] Ultralytics, "YOLOv8 Models — Documentation." [Online]. Available: https://docs.ultralytics.com/models/yolov8.