Alma Mater Studiorum - Università di Bologna

Department of Computer Science and Engineering - DISI
Second cycle degree in Artificial Intelligence

Deep Learning approaches for multimodal analysis in car collision forensics

Master degree thesis

Presented by:
Alessio Bonacchi

Relator:
Prof. Eng.
Luigi di Stefano
Co-Relator:
Eng.
Davide Castellucci

Session IV
Academic year 2022-2023

I would like to thank my mum, my dad, my girlfriend, my relatives and all my friends for always supporting me during the difficult moments and celebrating the best ones with me throughout this journey.
Additionally, I would like to thank Professor Luigi Di Stefano and tutor Davide Castellucci, as well as the entire Datavision team, for giving me the opportunity to carry out the internship and making it a highly formative experience, both technically and personally.

Abstract

Machine Learning and Deep Learning have gained significant traction, playing a pivotal role in addressing a myriad of challenges across various domains. In particular, we decide to exploit them in order to solve numerous tasks as car pose classification, regression of EES (Equivalent Energy Speed), car damage segmentation and 3D reconstruction from images and videos. This thesis will present the methods used to solve these problems, along with possible future improvements.

Keywords: Deep Learning, Machine Learning, Computer Vision, Classification, Cars, Regression, 3D Rendering, Instance Segmentation, Neural Radiance Fields, Gaussian Splatting

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main goal of this work, developed within the company DataVision s.r.o. during an internship, is to research and provide several tools for the diagnosis and forensics of car accidents. This was accomplished by exploiting Deep Learning and Computer Vision technologies in order to train models able to solve task like the classification of the pose of the car from an image (car pose classification) and the estimation of the Energy Equivalent Speed (EES), namely the energy equivalent to crashing into a wall at a certain speed expressed in km/h. Furthermore, models for car damage detection and segmentation, and 3d reconstruction have been developed using the latest technologies available in order to provide a complete overview of the accident.

To achieve these goals, several models and strategies have been employed: classification and regression models, cross validation and many others that will be explained deeply in the next chapters.

The project started as a proof of concept for an external company, which provided us with his data, and then moved to an internal project with the development of the segmentation and 3d reconstruction tools.
The aim of the project is to build an all-encompassing system of tools able to diagnose and analyse the context of a car collision, in order to gather all the possible information from the scene and provide the user with a full overview over the accident. Then, once the data are processed, the collision can be well documented in

order to ensure a correct and righteous possible trial and insurance resolution.
In this way, a model able to estimate the energy dissipated in the crash is needed, and in order to make the predictions more accurate we also need a model able to classify which area of the car have been damaged, since the consistency of the damage of course depends on the different rigidity of the part involved.
In addition to that, to have a proof of the correct disposition of the vehicles at the moment of the accident an easy to use 3d reconstruction model is fundamental. It is useful as a proof having a model that from a video builds a 3d render of the scene, enabling the user to better analyse the dynamic of the collision.
In addition a model that is able to spot the damage on the cars and segment the precise areas that are involved is fundamental to estimate and quantify the amount of damages, both to predict the entity and type of the damage and which parts of the car need to be replaced.

1.1 Proposed solutions

This section is meant to give a brief overview over the solutions adopted to solve the problems assigned during the internship. The choice of technologies able to solve the tasks defined have been guided everytime by the research of the actual state of the art in order to offer the best possible product and results.
For the car pose classification task, discussed in Chapter 3, a pre-trained Resnet50 [10] served as a backbone for the classifier model, attached to a finetuned classification head. The choice of the Resnet was obvious since it achieves the state of the art in many classification task. Furthermore, it strikes a favorable balance between performance and training speed. Its ease of extensibility stems from the straightforward training of deeper ResNet models facilitated by residual connections. Given the ongoing expansion of the dataset, the potential need for employing a deeper network in the future is foreseeable.
Even for the EES regression a Resnet [10] was used in order to extract features from car crash images, which have been used per se or combined with the numerical features such as rigidity, brand, etc. to estimate the target. Here the focus was also on best practices to preprocess and label both image and numerical data, since we

were dealing with a very small dataset, initially composed of barely 400 instances. As a consequence, cross validation was employed in order to check if the model was able to generalize well and not overfit on the train data.
For the detection and segmentation of the damages on cars bodywork, described in Chapter 5, 2 models have been selected:

- Mask-RCNN [7] : this has been the state of the art in segmentation and detection area for a long time, it represent a go-to, easy to train and configure base model. It was used as a first approach to this task.

- YoloV8[2] : the new frontier for vision AI, since its release have gained the state of the art title in many vision tasks. In the solution implemented it represents the technology that is used to solve the task.

Then, for the 3d rendering task, described in Chapter 6, we decided to employ the latest models from the Neural Radiance Fields [12] area. Thanks to this technology we can synthetize a 3D render directly from a video or even from few images, as it will be explained in the next chapters, by exploiting neural networks that can perform a differentiable learning in order to reconstruct the 3D scene topology.
In particular this was an opportunity to work with Gaussian Splatting [11], a very recent 3D rendering model that came out during the timeframe of the internship. Gaussian Splatting is peculiar for its extremely brilliant results, comparable and even better to those produced by MipNerf360[3], the State of the Art in this context. But the main characteristic is that it reduces the training time of a single scene from days to hours, allowing for an almost real time rendering.
For the last part of the work we also wanted to try to render the 3D scene from a single image. We initially employed PixelNerf [17] but the results were not too good. However after Gaussian Splatting was released, DreamGaussian [16] was designed to solve the single image rendering task and we tested out on our images. Although this results are not brilliant as for videos, the ability to predict unseen views are promising, opening the path for the future of vision AI.

Chapter 2

Infrastructure

2.1 Data annotation platform

The platform that we used to store, annotate and store the data is called BeeYard. BeeYard, an in-progress platform owned by the company, assists data scientists in every step of the Machine Learning Operations process, encompassing data gathering and structuring, annotation and categorization, and testing and advancement, to deploy machine learning algorithms reliably, stably, and at scale.

BeeYard aids data scientists at each stage of the machine learning project. For instance, it facilitates the creation of well-organized datasets accessible to all stakeholders through secure and managed channels.

The user can make use of the hybrid cloud capability to move data between edge and cloud infrastructures. When dealing with extensive machine learning tasks and experimenting with different hypotheses, one has the option to utilize either BeeYard's cloud infrastructure or their own private cloud. This hybrid cloud setup combines on-premises or private cloud infrastructure with a public cloud, enabling the transfer of data and applications between the two environments.

Additionally, it offers numerous functionalities to ensure the security and reliability of the dataset. Users have the ability to arrange the data into workspaces with predetermined limitations. Then it offers increased flexibility, various deployment options, enhanced security, compliance, and the ability to maximize existing infrastructure.

Pre-established, organizational annotation labels along with additional annotation resources can be implemented to maintain uniform data and decrease inaccuracies caused by human annotators.

BeeYard offers a robust set of tools for annotation, significantly reducing the time needed to annotate a dataset. This allows for the distribution of annotation tasks to annotators located worldwide.

It also enables the training of complex neural networks or running predictable algorithms on extensive datasets. In the process, it effortlessly gathers performance metrics for the user.

Because of its open architecture, BeeYard offers the possibility of constructing composite AI models by combining non-deterministic algorithms with deterministic algorithms. These models have the advantage of enhanced performance and improved ability to provide explanations.

Then each model trained or executed on the platform can be saved and stored, kept open for future versioning or manteinance.

One of the main features that make BeeYard crucially useful is its power to maximize the potential of cloud computing for algorithm development. By making it straightforward to transfer workloads onto more potent infrastructures, it helps to shorten iterative stages. Implementing this technique is far more expedient than utilizing less powerful in-house resources when developing machine learning models. Essentially, BeeYard enables faster development of critical machine learning models in the cloud instead of relying on local computers with fewer resources.

For the backend, a custom API was developed to facilitate the utilization of a Python SDK. This allowed for seamless integration and utilization of the SDK's capabilities. On the frontend, the focus was on image visualization, annotation, and annotation correction. The images were stored in "cells," which served as groups for images associated with the same cars or collision scenes. These cells were labeled with tags aimed at expressing and enabling searchability of shared properties among cells and evey image was also labeled with other tags. Also every cell expose a properties field where a set of data at our disposal can be stored in order to better describe the cell itself.

In addition, the cells provided the option to incorporate additional files to enhance

the explanation within the cell or insert "annotations" on top of the images without altering them.

Image 2.1 show how a cell look like in BeeYard.

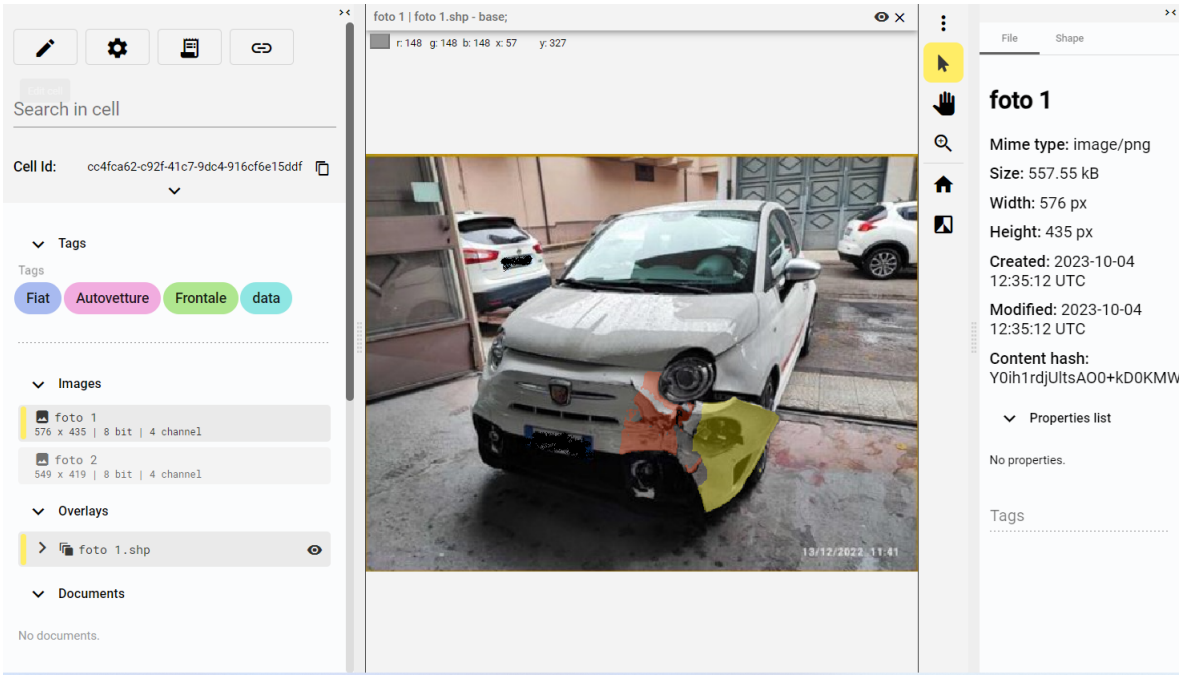

Figure 2.1: An example BeeYard cell

2.2 Hardware and software used

The experiments have been run on two devices:

- *Trainer01*: GPU NVIDIA GeForce GTX 1080 with 12 GB of RAM running on Ubuntu server 20.04. Server with 31 GB RAM, 125 GB SSD, 503 GB HDD, and CPU: 4x Intel(R) Core(TM) i7-10700KF CPU @ 3.80GHz.
- *Trainer02*: GPU NVIDIA GeForce RTX A600 with 51.5 GB of RAM running on Ubuntu server 20.04. Server with 49 GB RAM, 125 GB SSD, 503 GB HDD, and CPU: 4x Intel(R) Core(TM) i7-10700KF CPU @ 3.80GHz.

The communication with the GPU was established through an SSH connection on a dedicated machine that operated constantly in a server room. Consequently, it was unnecessary to maintain a continuous connection with the GPU, enabling extended

periods of training.
A significant benefit was the ability of caching images on the local machine, enabling the proxy to check for their existence in the cache whenever the script downloads them. Additionally, the proxy would verify if the cached image was still up to date. In cases where it was, the image would be served directly from the cache, eliminating the need to download it from the cloud. On the other hand, if the image was not present or outdated in the cache, it would be fetched from the cloud and then stored in the cache. As a result, this feature accelerated the training process by avoiding repetitive downloads of images from the cloud, as they were readily available in the cache.

To begin with, we used the SDK to gather all images classified by certain tags by creating a query that specified the required tags and attributes in MongoDB. This allowed us to extract a list of cell entities encompassing all necessary details and attachments in a JSON format from the response to this query. This processed data then proved to be usable on the backend side with Python to construct our dataset.
The SDK also encompassed various additional features such as the option to append tags to the images, apply overlays for visually representing the tags on the image, incorporate files containing the training statistics, and several other functions. These functionalities aimed to enable and streamline the utilization of all the capabilities provided by BeeYard on the backend.

Chapter 3

Car Pose Classifier

In this chapter the methodologies and strategies used to build the neural network for the car pose classification task are explained. The task consist of analyzing an input image of a car and then classify its pose into 3 categories: Front, Back and Lateral.

3.1 Dataset

To facilitate the model training process, two distinct datasets were employed. The initial dataset, of a proprietary nature, was hosted on BeeYard but was not annotated, while the subsequent dataset was sourced from online repositories [1].

3.1.1 BeeYard dataset

Initially, the purpose was to annotate the dataset on BeeYard with a model that was iteratively fed with an enlarged dataset, composed by images that were classified by the model itself and manually checked for errors everytime.
In fact, this dataset was not labeled but it only contained car images grouped in cells, where among each cells were contained images of the same car. Each cell have a tag "Brand" that specifies the brand of the car and every image has a tag that specifies if the car in the picture is a render or a real car image.

This information was utilized for the meticulous labeling and curation of the dataset. Our objective was to establish a well-balanced set, encompassing a diverse array of cars from various brands, while maintaining an equitable distribution between rendered and real images.

Furthermore, since these images are collected from reseller and automotive websites, it can happen that some of them do not depict a car but advertisement, some details on the bodywork, the internal or the engine. We have been labeling images like these with tags Ads, Internal and Close, since we want our model, for this purpose, to be able to distinguish and well categorize car images and separate them from uninformative ones.

Figure 3.1: An example of an Internal car image, not useful for the task

The pipeline followed to annotate this dataset consisted in the following steps:

1. Manually annotate 100 cells, roughly 400 images.
2. Repeat:
   (a) Train the model on the annotated data
   (b) Select a new set of cells and run the inference on them
   (c) Save the inferred results as tags
   (d) Manually check and correct the inferred tags

The dataset was then split in train, validation and test with a proportion of 70-20-10%.

3.1.2 Roboflow dataset

The dataset collected from RoboFlow[13] consists of 7690 car images divided in classes Front, Rear, Lateral and Rejected. Since pictures of class Rejected are not informative for the task, by representing collided and closeups on vehicles, they are removed from the final dataset.

Figure 3.2: Example of Front image from RoboFlow dataset

Differently from the BeeYard dataset, here we had not to perform a rebalancing on the dataset, since the collection of images appear to be heterogeneous.

The dataset comes already split in two folders Train and Test. Each folder contains the respective images and a csv file "classes", which exposes for every filename a one hot encoding for the possible pose classes.
The proportion on the train, validation and test split is the same as the one used for the BeeYard dataset (70-20-10%).

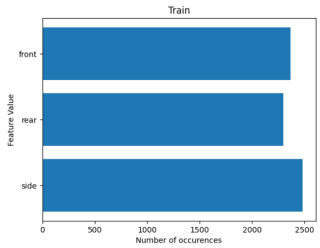

Figure 3.3: Train balance

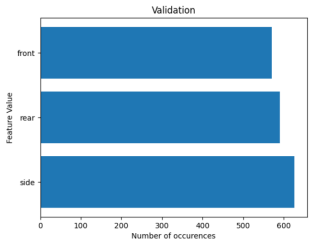

Figure 3.4: Validation balance

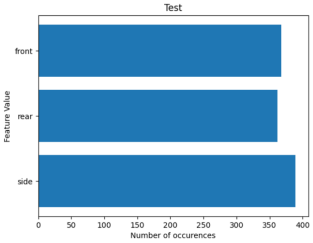

Figure 3.5: Test balance

Figure 3.6: Class balance analysis over the dataset

3.2 Data processing and augmentation

For both dataset target features have been represented with OneHot encoding, so having a column for every class value, with a one or a zero if the instance is or not of that class. This choice was made firstly because each class is mutually exclusive, so it is not possible for an instance to assume two different values and secondly because an order relationship between values does not exist in this case, so using a label encoding makes no sense.

Regarding the images, for both dataset we decided to resize every image to a standard dimension of 224x224, then converting them to tensors and normalize them with the mean and standard deviation of the ResNet used.

Then, for each of the two dataset a different processing was applied, since each one of them exposes different characteristics.

For BeeYard one, we decided to augment the image by using grayscale, horizontal flip and color jitter transforms on train and validation splits. This choice was guided by the fact that since the images, especially in the first stages, were very few then we needed the model to see many different images as possible, preserving the semantic meaning, in order to bring better results. For BeeYard's dataset augmentation, our approach was rooted in the recognition that 'diversity in training data is the cornerstone of robust model development.' This philosophy aligns with the wisdom shared by Geoffrey Hinton, a trailblazer in neural networks, who emphasizes that 'a model trained on a diverse set of augmented data is more likely to capture the underlying features of the target domain'.

On the other hand, regarding RoboFlow dataset, the number of images is not a problem since we have enough of them. The approach employed in this context was to introduce images with inherent noise to the model, thereby enhancing its robustness and promoting broader generalization capabilities. Primarily grayscale was applied with a certain percentage to augment the dataset, then we moved to the appliance of Gaussian blur and Salt and Pepper noise. In particular we employed the first one with mean equal to 5 and standard deviation equal to 1, while the Salt and Pepper percentage was set to 0.01. An example can be seen in image 3.7.

Figure 3.7: Example of augmentation on RoboFlow dataset.

These deliberate choices in augmentation techniques align with the fundamental principle that noise introduces a layer of complexity essential for model resilience. By progressively incorporating variations such as Gaussian blur and Salt and Pepper noise, we sought to expose the model to diverse environmental conditions it might encounter during deployment. Through this meticulous approach to dataset augmentation, encompassing both quantity and quality considerations, our aim was to equip the model with the adaptability and robustness necessary for real-world applications, where variability and unpredictability are inherent challenges.

Another noteworthy aspect is the continuous expansion of the BeeYard dataset. Consequently, we opted to create a snapshot of the dataset in the form of a CSV file, comprising the cell ID, image name, and corresponding label. This approach allowed us to maintain consistency in the images used for testing the deep learning model.

The process involved offline handling during cell collection, where queries were em-

ployed to gather pertinent data. To ensure a streamlined structure, we refrained from storing the actual images. Instead, we opted for a more lightweight snapshot, containing only essential information—cell ID, image name, and label. This decision aimed at expediting the retrieval of images from BeeYard without the need to re-extract label information from the dataset.

3.3 ResNet50

The model used as a backbone for our neural network is a ResNet50 [10].
ResNet50, short for Residual Network with 50 layers, is a variant of the ResNet architecture. Introduced the groundbreaking paper "Deep Residual Learning for Image Recognition,"[10] ResNet50 is celebrated for its ability to tackle the challenges associated with training very deep neural networks.
Primarily, ResNet was chosen for the fact that it has consistently demonstrated to be able to reach state of the art performance in many machine vision taks, making it a robust choice, since we want our models to be as more accurate and reliable as possible.
Another ResNet50's distinguishing feature lies in its deep structure, comprising 50 layers. This depth facilitates the extraction of intricate features, a crucial aspect in the nuanced analysis required for car collision forensics.

Furthermore, ResNet introduces a new way to mitigate the vanishing gradient problem by introducing residual connections. This mechanism allows the network to learn residual functions, making the training smoother and convergence faster.
More in detail, residual connections consist of connections that allows to skip of certain layers, enabling the flow of information across these layers without hindrance. The residual block is structured to allow the model to learn the residual function, which consists of the difference between the input and output of a layer. Mathematically, if x is the input of our layer and $F(x)$ is the corresponding transformation

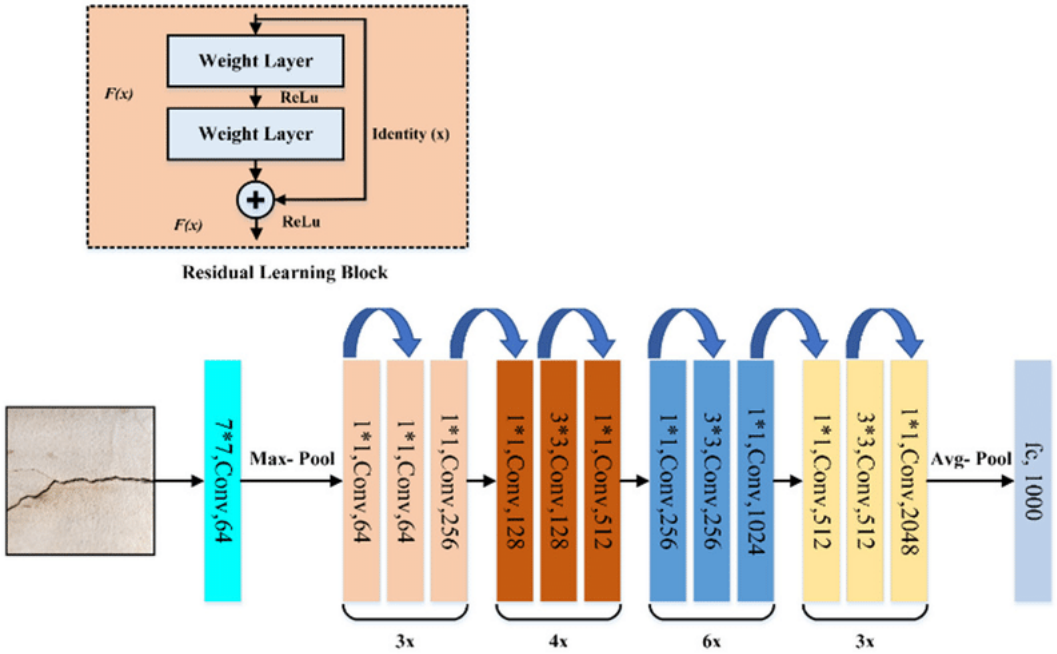

Figure 3.8: ResNet50 architecture and highlight on residual connection.

applied by the layer, then the output is:

$$y = F(x) + x$$

This allow the gradient to flow directly trough the skip connection, securing the information from the input to be preserved.

It is also very important the fact that it is possible to use a pretrained version on ImageNet dataset, which is extremely useful when dealing with very small dataset as in our case. Since it is a very deep model training the whole architecture on such small set of data could lead to bad overfitting problems. On the other hand it is still possible to finetune its weights to exploit the whole architecture capabilities to extract deep features.

In the end, its versatility stands out, allowing to develop solutions for the analysis of images and numerical data, as we will see in the next chapter.

3.4 Architecture

Talking about the architecture employed to solve the task, a similar structure was employed when dealing both with BeeYard and RoboFlow dataset.

The image is processed initially by a ResNet50 producing a feature tensor with 1000 weights. Then this tensor is given as input to a deep classification head in order to produce a label, categorizing the image into a pose. A diagram showing the general architecture of the model can be seen in 3.9.

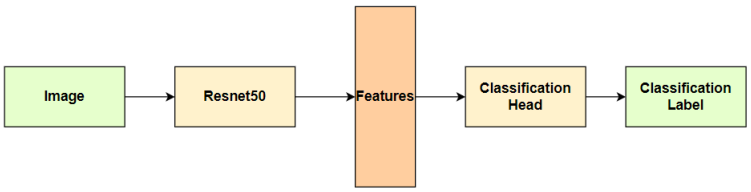

Figure 3.9: General architecture for car pose classification.

The main difference in the architecture used for each of the dataset available consisted in the employment of the backbone as it is, freezing its layer in a transfer learning fashion or to finetune also its weights.
Since in the case of the BeeYard dataset, especially in the first stages we had to deal with a very little amount of data a transfer learning approach was chosen in order not to risk for an overfitting. In this way, we prevent the model to adapt too much on the training data, but at the same time perform badly on the test. In addition, this allowed to train the model in a reduced amount of time, which was useful to perform multiple test to tune the hyperparameters.
On the other hand, we were dealing with more images, then while using transfer learning to check the initial performances, we decided to run a finetuning also on the backbone, by unfreezing its layers. In this case, we let the model to adapt to our domain since here the risk of overfitting is lower. Also in the future perspective that more new data are added, it could be possible that by combining these new dataset, we want our model to be domain specific and focus only on the task of pose classification
Another difference was the classification heads. For the RoboFlow dataset model we decided to employ a sequential block composed by a linear layer with input dimension equal to the output dimension of the feature extractor and output dimension

equal to 256, a Rectified Linear Unit layer, or ReLU, a linear layer with input dimension = 256 and output dimension 64, another ReLu and the last linear layer with input size 64 and output size equal to the number of classees. For the BeeYard dataset we decided to use an even simpler model by removing the intermediate linear layer and insert a Dropout layer after the first ReLU.
Then a SoftMax activation function was employed, since we are dealing with a multiclass classification problem, to map the outputs to a probability distribution over the classes. To compare the outputs of the neural network to the true labels CrossEntropy loss was computed at train time, while during test the maximum value of this vector is computed, identifying the predicted label.

3.5 Parameters and hyperparameters

As stated before, the loss function that we wanted to minimize to make our model learn is a Categorical Cross Entropy, since we are dealing with a multiclass classification task. This loss is defined by 3.1.

$$\text{Categorical Cross-Entropy Loss} = -\sum_{i=1}^{N} y_i \log(p_i) \tag{3.1}$$

where y_i and p_i are the are the groundtruth and the model score for each class i in C.
For the optimizer, Adam with initial learning rate equal to 0.001 was used. Adaptive Moment Estimation, shortly Adam, is an extension of the common SGD optimizer which incorporate adaptive learning rate for every parameter. Usually it is preferred over SGD for that reason, in fact it enables faster convergence, mitigating the need for manual tuning. Additionally, the incorporation of momentum and RMSprop features in Adam contributes to efficient optimization, resulting in improved convergence performance.
Talking about the batch size, we made different experiments, but we tested out that the one that scored the best was 16, representing a good trade off among frequency of updates and dimension of the dataset.
Since the momentum is completely handled by the optimizer, we decided to focus

on the learning rate by employing a scheduler. In particular the choice fell on ReduceLronPlateu, since we wanted a mechanism able to adapt the learning rate to the validation loss in order to mitigate the possible overfitting and lead to faster but astute convergence. In particular we decided to be a little more aggressive setting the patience to 3 and the multiplying factor to 0.1. We also pinned the verbose option in order to check the updates in real time during the train.
For experiments, an early stopping mechanism was inserted in the training pipeline, set with latency of 5 and a patience of 5, which means that the minimum number of epochs performed are 10. This was done in order to make the train process faster and be able to perform multiple runs to finetune the parameters.
Then we set up the number of epochs to 20 and we decided to store the model weights every time that the validation loss went below its minimum. In this way at the end of the training we are left with the checkpoint of the best performing epoch.

3.6 Experiments and results

Regarding the BeeYard dataset, the evaluation on the inference of the model was done manually by looking train after train the performances directly on the labels assigned. In this case we run at least 10 different training, every time including the previously tested out and corrected images in the train set, reaching nearly 2000 images available in total to train the model.
Since the main purpose is to annotate the dataset and we do not have original annotated data, the evaluation was done by manually checking the images and the assigned label in order to see if that was correct. By doing so we was also counting out how many predictions were correct out of the total number of images. At the end of the experiments, with our final model the labels were assigned the correct way in most of the cases, sometimes confusing Close, Ads and Unknown images.
This could be due to the fact that sometimes in Close images some details with text (mainly the model or brand of the car) are shown, as it is in the other two labels for obvious reasons (for example, it is common for Unknown images to show irrelevant details of cars or machine manuals for example) and then it is possible that the model makes confusion for this reason, as we can see in 3.10.

Figure 3.10: Example of challenging distinctions between Close details (left) and Unknown (right).

Furthermore, this error did not affect our work in a bad way since our main focus was to estimate car's poses and we preferred to classify them in the right way them, instead of Ads or Unnkown.

On the other hand, we had an annotated test set at our disposal. For this reason, we could proceed to evaluate the performances of the model with an appropriate metric. In particular, F1 score was selected since it combines precision and recall into a single metric, providing a balance between the two and a more robust metric to evaluate our data.

The formula for the F1 score is defined as the harmonic mean of precision and recall. Precision (P) is the ratio of true positives to the sum of true positives and false positives, and recall (R) is the ratio of true positives to the sum of true positives and false negatives. The F1 score mathematical formulation is defined by 3.2.

$$F1 = \frac{2 \cdot (P \cdot R)}{(P + R)} \tag{3.2}$$

In particular, since we were dealing with a multiclass classification problem, we did

	Front	Lateral	Rear
F1 Score	0.99321574	0.99171271	0.998713

Table 3.1: Car pose classification F1 class scores.

not calculate an overall F-1 score. Instead, we computed the F-1 score per class in

a one-vs-rest manner. The results in 3.1 show the extremely good performances of our model. Boasting an impressive average F1 score, among the three classes, of 0.99454715, this model demonstrates production-worthy capabilities and long-term maintainability.

Chapter 4

EES Regression

In this chapter, the strategies and methodologies employed to tackle the EES regression problem are explained and commented. This task was part of a proof of concept requested from an external company and it consists in the estimation of the EES, namely Equivalent Energy Speed, which is the vehicle speed equivalent to the energy consumed to cause the vehicle deformation. Technically the task consists in analyzing an input image, representing a car accident, and possibly some numerical input values and then estimate the EES value for that car collision.

4.1 Dataset

Initially we had at our disposal a single image dataset, stored into BeeYard. Here every cell represented a distinct car crash, described by a set of images showing the collision and a set of attributes:

- **ees**.
- **brand**.
- **model**.
- **damage position**.
- **type of vehicle**.

Since this dataset contained different type of vehicles, including bicycle, we filtered our dataset to only contain automobiles, reducing the number of cells used from 481

to 362.

Furthermore, the images were varying in size and exhibited sometimes low quality as they have been gathered from numerous individuals using different devices.

In addition to that, the rigidity dataset was provided. The rigidity of a vehicle part refers to its ability to resist deformation or distortion under applied forces. Differently from the other, it contained columns brand and model, the position in the vehicle and the corresponding rigidity. Moreover it contained other 10 column features, including weights, height, length, etc. .

Figure 4.1: An example of an image from EES dataset.

This set of data was used as a support for the first one since it did not contain images per se. It was combined with the first one using columns 'model' and 'brand' as key to perform a join operation. Then, since rigidity changes based on the position of the damage on the vehicle, we had to match the corresponding rigidity.

The primary challenge when combining the two datasets was the absence of all car models and brands in the rigidity dataset, causing unmatched cells in BeeYard and a reduction in available images. Additionally, variations in car models restyling led to diverse bodyworks and, consequently, rigidity. To address this, an average rigidity value was calculated for each specific brand, model, and position to mitigate discrepancies arising from undefined restyling in the BeeYard dataset.

4.2 Data exploration and feature selection

Initially, we aimed to examine the distribution of the 'rigidity' feature to ensure it was not unbalanced, which could adversely affect training behavior and outcomes. We decided to plot a frequency histogram as we can see in 4.2. The distribution appears to be not well balanced, with very distant values and not centered peak. This could be a sign of outliers presence.

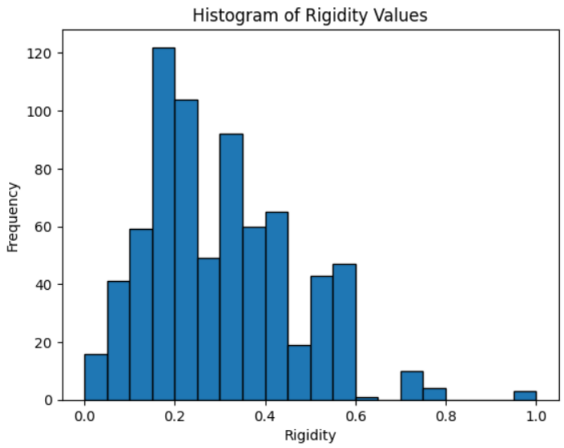

Figure 4.2: Rigidity distribution before IQR.

In order to deal with this problem IQR, namely InterQuantile Range, was employed. This is a measure of the spread of the dataset and it is often used to identify and eliminate outliers.

The IQR is calculated as the difference between the third quartile (Q3) and the first quartile (Q1) of a dataset. The formula is as in 4.1.

To identify potential outliers, a common rule of thumb is to consider values that fall below $Q11.5IQR$ or above $Q3 + 1.5IQR$. Data points outside this range are often treated as outliers and are subject to removal.

$$IQR = Q3 - Q1 \tag{4.1}$$

We can see the result of this operation in 4.3. After this operation, the distribution was well balanced and the dataset spread over the rigidity value was much more

contained.

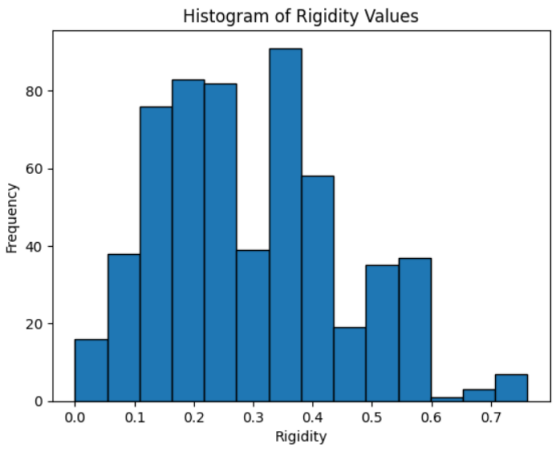

Figure 4.3: Rigidity distribution after IQR.

Talking about feature selection, we noticed that features different from 'brand', 'model', 'damage position' and 'rigidity' did not improve the model results. This is counter intuitive, since the more data the model is fed with, the more accurate the results should be. As a consequence, we decided to investigate this strange phenomena by using target correlation. The logic behind using correlation for feature selection is that good variables correlate highly with the target, while bad variables correlate poorly.

From what we have analyzed, we discovered that only the features cited above were highly correlated with the target, while the others showed near zero correlation.

An idea that came to our mind trying to explain this phenomena was that perhaps features like width, length, weight, ... are measure that were already took into account in order to estimate rigidity and that feature engraves all of them into one.

4.3 Data Processing and Augmentation

This section is meant to show and explain the techniques used for the processing and augmentation of both numerical and image data. Since we were dealing with a very small amount of data we needed to carefully and accurately decide how to transform them in order to get the best possible results. The whole dataset was then randomly split into train (70%), validation (10%) and test (20%) sets.

4.3.1 Image Data

Since images have different sizes, we decided to resize them to 512x512 pixels. This choice was driven by the scarcity of data , additionally we wanted to preserve images as similar to the originals as possible. Moreover we proceeded to convert them to tensors and then apply normalization to prepare them for the ResNet processing. Regarding augmentation, we decided to transform images to grayscale and perform some horizontal flip in order to provide more variety in the dataset. The objective was to inject diversity into the dataset due to its limited size. This approach aimed to prevent overfitting and enhance overall performance, ultimately seeking improved results.

4.3.2 Categorical and Numerical features

The dataset contain both these types of features:

- rigidity: numerical features, takes floating point values that express the rigidity of the car.
- model: categorical feature, string that express the model of the car.
- brand: categorical feature, string that express the brand of the car.

Talking about 'rigidity', since it is a numerical feature, normalization is a standard preprocessing step that contributes to better machine learning model behavior. In this case we employed a Min Max scaling technique to lie the values in the range of 0 and 1. This is important because few values might be very big and few might be in small ranges, which is a behaviour that is not recommended in machine learning scenarios.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{4.2}$$

On the other hand, for the remaining categorical variables we had to decide which type of encoding to employ. Here below a brief explanation of the different types of encoding techniques considered.

1. **Label Encoding**: the simplest one, yet the worse in this context since it assign every possible value of each feature with an increasing number. In our case the order between values is not needed since we are dealing with car brands and models. Moreover, another possible problem is that since the number of possible values is high, even considering that the dataset is small, this kind of encoding can introduce a noise in the learning that makes a value more "relevant" against one other. We tested it in our context for the very first trial, in order to have very fast initial deployment but we immediately moved to other encoding techniques since they are more appropriate.

2. **OneHot encoding**: this is the recommended option when the categorical feature do not need ordering in the encoding. In this case, we are representing the variable as a binary vector where the binary vector is all zeros except for the bit corresponding to the value, which is set to 1. The main drawback of this approach is that when a feature assume a great number of different values, the size of this vector grows incrementally, since we have to reserve a bit or columns for every new value. Despite this, it is a completely fair choice to encode categorical variables.

3. **Target encoding**: this is an encoding technique consisting in replacing a categorical value with the mean of the target variable. Instead of representing feature values with a binary vector, this encoding assigns each category a numerical value corresponding to the average target feature value for that category. This encoding is useful since it leverages information about the target variable, potentially capturing relationships between predicting features and the target.

In the end, we chose the OneHot encoding among the three approaches tested. It represents a balanced trade off between simplicity and efficiency, as we just needed to work with predictive features, differently from target encoding where we are dealing also with the target.

4.4 Cross Validation

In machine learning and deep learning, the goal is to create a model that can make accurate predictions on new and unseen data. To achieve this, we need to train it on a dataset that is representative of the population we are trying to fit. However, simply training the model on a single fold can lead to overfitting, where the model becomes too complex and specific to the training data, leading to poor performance on new data.

Moreover, since we were provided with a very small amount of data, we needed a good practice that could demonstrate the model stability and consistency. What we did not want to have, was a model that performed well when trained with a 'lucky' portion of small data, fitting it extremely well, while scoring poorly with other folds of the dataset.

For this reason, Cross-validation proved to be a valuable tool in addressing this issue, serving as a crucial instrument to tackle the problem effectively.

In particular, Cross Validation was used to assess model architecture stability and consistency before the "real" training: this was done in order to check if the random splits are not lucky ones, to test if the model could mantain its performances on every fold.

Cross Validation is a powerful technique that is a standard practice for evaluating and validating machine and deep learning model. It helps to detect overfitting and provides a way to test if the model is consistent or not.

It involves dividing the dataset into multiple subsets and then using them iteratively to train and validate the model.

In particular, K Cross Validation was employed. It involves dividing the data into k equal-sized subsets or folds. The model is then trained on k-1 folds and validated on the remaining fold. This process is repeated k times, with each fold being used for validation exactly once. The process is shown in 4.4.

For our experiments, we performed two distinct tests on each developed model, setting the k parameter to 5 and 7. As stated before, we wanted to be sure that our model performances were not influenced by the scarcity of data or the specific train

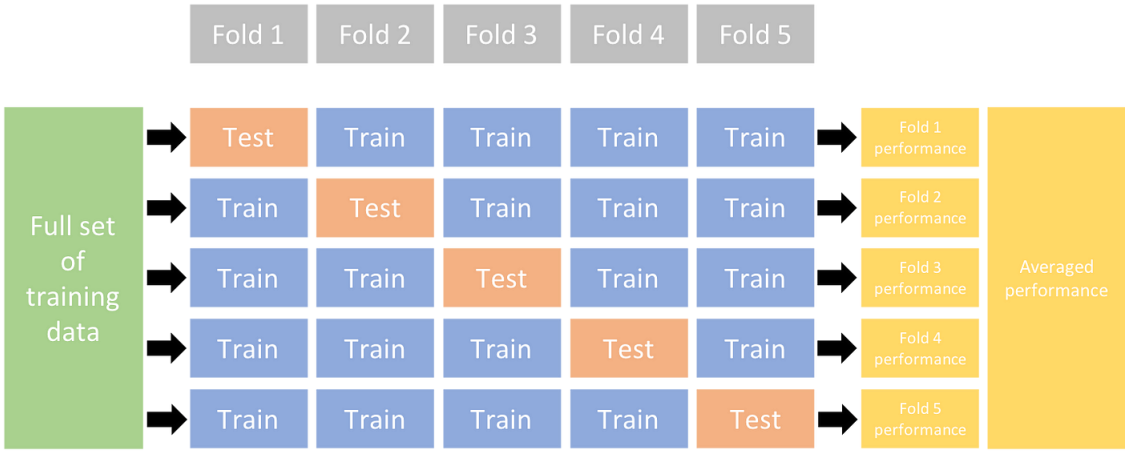

Figure 4.4: K Fold Cross Validation.

set that we randomly choose out of the whole dataset. The purpose was to test the consistency of the the model in order to provide the client with trustable and stable performances.

4.5 Architectures

For this task in particular, the focus was not only on parameters and hyperparameters tuning, but also into finding the best deep learning architeture that could fit the problem. We moved from very basic regressor solutions to more sophisticated ones, by combining the visual and numerical data that we had at our disposal, trying to understand the deepest relationship and patterns to estimate the Energy Equivalent Speed.

The image feature extractor used for this experiments is a ResNet50[10], an innovative 50 layered deep learning architecture that established different state of the art records in many visual tasks.

4.5.1 SVR

One of the first model that we have tried, along with the single image one, was an SVR, namely Support Vector Regressor, attached to a ResNet50 feature extractor. The purpose was to extract image features with the ResNet50 and then use the SVR as a regression head to predict the EES value.

Figure 4.5: SVR model.

The goal of SVR is to find a function that approximates the relationship between the input features and the continuous target that we want to predict, by minimizing the regression error.

Unlike Support Vector Machines (SVM), which are used for classification tasks, SVR seeks to find a hyperplane that best fits the data features in a continuous space. This is achieved by mapping the input variables to a high-dimensional feature space and finding the hyperplane that maximizes the margin (distance) between the hyperplane and the closest data points, while also minimizing the regression error.

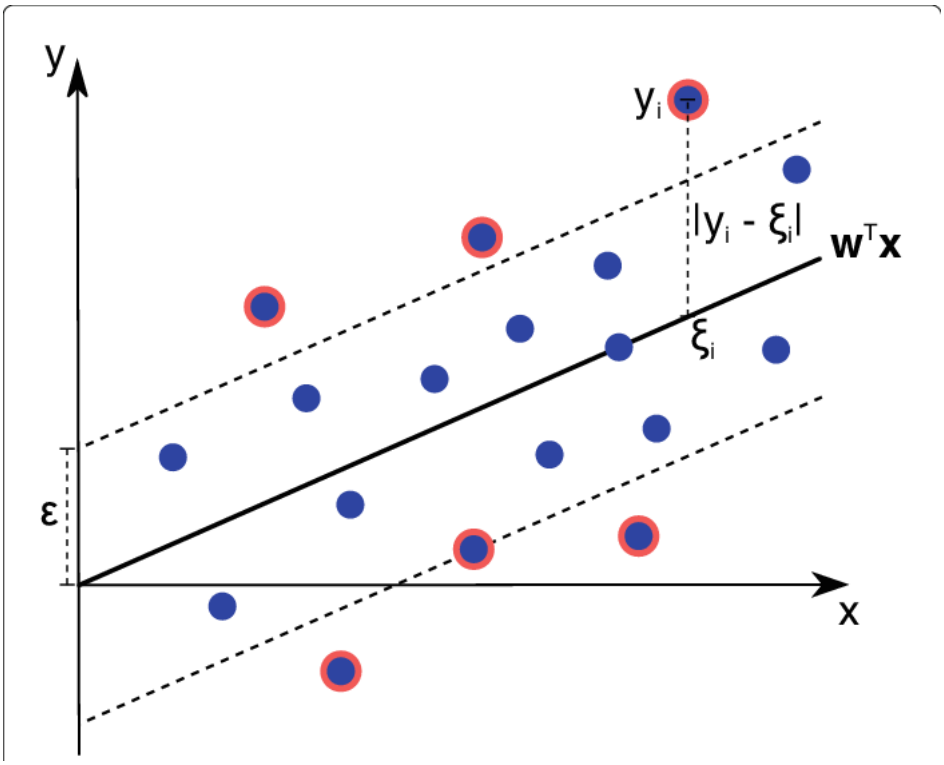

Figure 4.6: SVR principles of functioning.

In particular, for the experiments that we carried out, we employed a linear kernel for the SVR and we set the model parameters C to 1 and ϵ to 0.2. These parameters

are used to control the regularization and the margin of tolerance for errors.
In particular, C controls the tradeoff between achieving a low training error and an high testing error. This is due to the fact that this parameters controls the penalty for misclassification: a small value of C makes the decision boundary smooth, and a larger one aims to classify all training points correctly, potentially leading to a more complex decision boundary.
On the other hand, ϵ is the margin of tolerance for errors in SVR. SVR allows some errors within a distance of ϵ from the true regression function. This is particularly useful when dealing with noisy data or when a perfect fit is not possible or desired.
Back to our model, we subsequently fed our input image, after the aforementioned preprocessing, to the ResNet feature extractor, which gave us a tensor of size (1,1000). Furthermore we fit our SVR with all the features extracted from the whole training images in order to predict the EES value.
The initial choice on the SVR was also made in accordance with Occam's razor principle, by starting experimenting, especially when we are dealing with few data, from easier solution first and then moving to more complex ones. This is due to the fact that a more complex model could lead to overfit, while a more basic one is at the same time more convenient and precise.

4.5.2 Single Image model

As mentioned in the previous section, the first goals of this task included to develop the easiest as possible solution. At that time, once we started working on this task, we were not provided with rigidity data but only with the collisions image dataset. For this reason we decided to employ a model that used a ResNet50 as feature extractor and then a deep regression head to predict the EES value. The general architecture of this model is shown in

The regression head was composed by a linear layer receiving features from the ResNet feature extractor with an input dimension of 1000 and an output dimension of 512, followed by a ReLU and a Dropout layer. Then we had the same connection as before, where a ReLU and a Dropout are preceded by a linear layer receiving

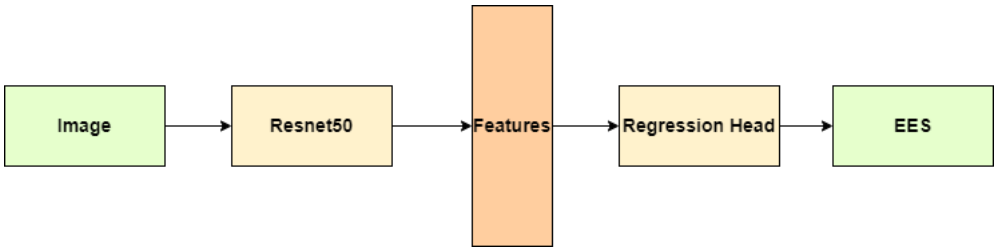

Figure 4.7: Single image model for EES regression.

input of size 512 and processing an output of size 256. All of that was finished by a single linear layer that gives as output the regressed EES value from a 256 sized tensor.

The overall performances with this model were surprisingly good, since it outperformed even the more complicated model developed, confirming that sometimes less is better.

4.5.3 Multi Image model

This model came out from the idea that in our BeeYard dataset car accidents, each of them described by a distinct cells, are described by multiple images. Additionally the 'human' idea that analyzing more data of the same context can make the prediction more accurate was transferred to the development of this model.

The concept was to extract features from each one of the inter cell collision images, then concatenate them and pass this feature vector to a regression head able to predict the target value.

The main problem here was that every cell has a different number of images attached. To solve this issue we analyzed the dataset discovering that for each cell, the dataset contained maximum five images.

Therefore the solution was to produce features from the image among the same cell, then if in the cell there were less than five images we passed to the feature extractor the remaining number of images filled with zero values to reach the limit. In this case, we were just attaching a vector of zeros to the concatenation of the other image features in order to standardize the length of extracted features.

Consequently, these features are concatenated and passed trough a deep regressor.

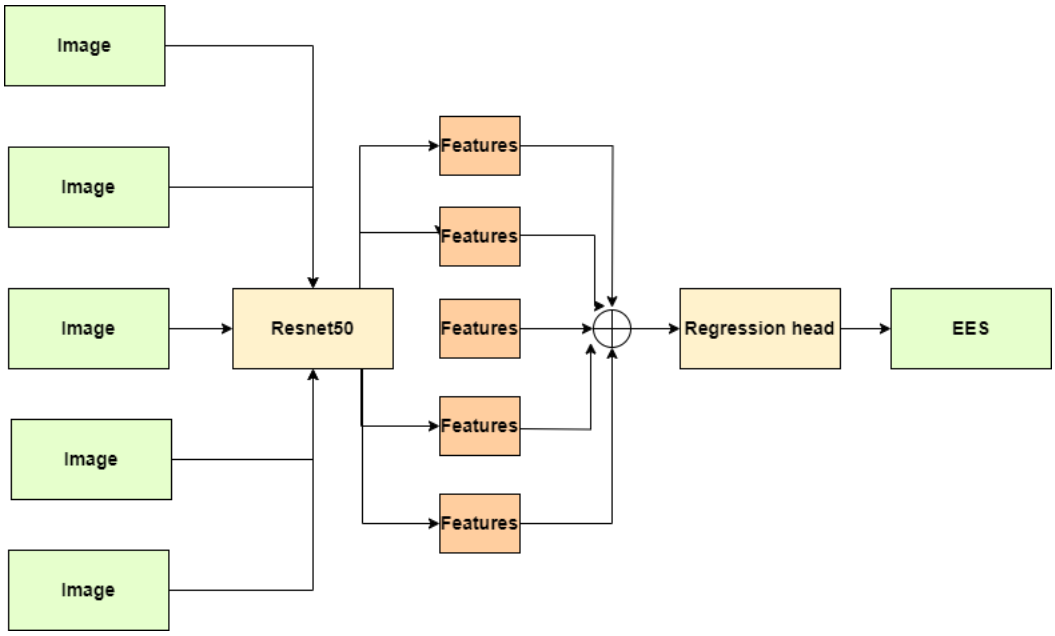

Figure 4.8: Multi Image model for EES regression.

This was composed by a linear layer accepting input features of size 5000 and output size equal to 1024, followed by a ReLu and a Dropout. In addition to that, another linear layer with input size 1024 and output size 512, concatenated to a ReLU and a Dropout layer was added. The same was done with another linear layer accepting input of size 512 and output size of 256, concluding with a final linear layer converting input tensor from size 256 to 1, which was the regressed output value.
Even if the idea was intuitive, results were the opposite since it performed worse than the single image model. The reason could be due to the fact that there are many cells with just a single image and in that case, four vectors of zeros introduced too much noise and this affected the performances of the model.

4.5.4 Single Image Mixed model

In this case, along with the image processing, the model takes into account also the numerical features. Every collision image, analyzed independently, was connected with its own model, brand and rigidity information, which were processed by a deep feature extractor composed by linear layers each connected to a ReLU, that extracts 512 sized feature vectors.
In particular, it consisted of a sequential stack of fully connected layers with in-

creasing dimensions (3 to 512), each followed by a Rectified Linear Unit (ReLU) activation function.

Then the features extracted from the image and the ones extracted from the numerical features were concatenated and passed to an another network, which produced the regression value. This was composed by a linear layer which took in input the concatenated feature vector of size 1512 and output a tensor of size 512, and ReLU is applied. Then it was processed by another linear layer which gave as output a tensor of shape 256, which is then processed by a ReLU and a Dropout layers. Finally the last linear layer accepted this tensor and returned the regressed value.

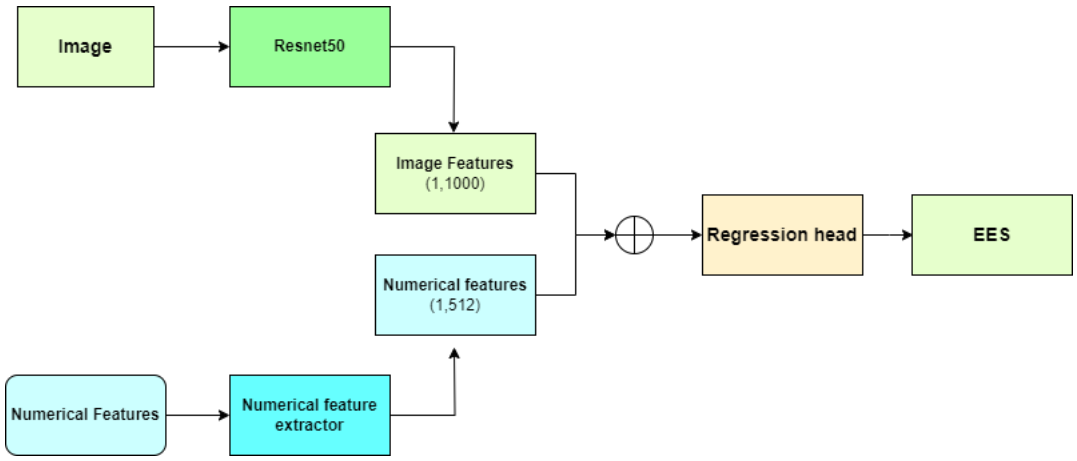

Figure 4.9: Single Image Mixed model architecture for EES regression.

This demonstrated to be the best trade off between complexity and efficiency, giving the best results obtained.

4.5.5 Multi Image Mixed model

For this model we decided to apply the previous successful idea of combining images and numerical features, with the intuitive but not as successful idea of combining images of the same car accident. We did not expect great results from this experiment since the previous simpler trial, but we decided to try it for the sake of completeness.

Then we processed the images with the same architecture used to extract concate-

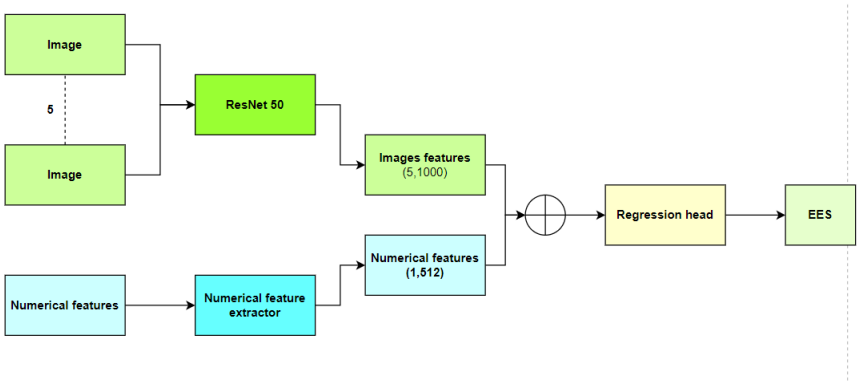

Figure 4.10: Multi Image Mixed model architecture for EES regression.

nated features from the 5 input images provided, and the same model also to extract features from the numerical data, as in 4.5.3 and 4.5.4. Then we combined these features as a concatenation, resulting into a tensor of size 5512.

These are now processed by a linear layer which returns a feature vector of size 1024, and a ReLU is applied. Then another linear layer transforms the 1024 sized tensor into one of size 512, with the appliance of a ReLU and a Dropout layer. Finally this output was processed by another linear layer which gave out tensor of size 256, a ReLU, a Dropout and the final layer which returned the regressed value.

As expected results were worse than the one produced from the multi image model. If previously the zero images introduced a noise component that disturbed the training, in this case these and the fact that we double processed the images and the numerical features, led the model to confuse itself and not be able to reproduce results similar to the others. Nevertheless, results were not extremely bad but it resulted in the poorer performer.

4.6 Parameters and Hyperparameters

In this section the parameters and hyperparameters exploited to train and test our model are presented. In particular, many different combinations of them were tried for each model, but we found out that the parameters that gave the best results for each model are slightly similar. Here will be presented the ones used to train the most performant model.

For the optimizer we chose Adam with initial learning rate equal to 0.01. Then we

used a learning rate scheduler (ReduceLROnPlateu) attached to the validation loss, by setting the patience to 5 epochs and the division factor to 10. We tried also other scheduler, but this proved to perform the better, maybe for its dynamical behavior. For the batch size we selected 16, which represented a good trade off between the frequency of weights updates and training time. In particular, bigger batch size showed worse results, probably for the fact that, since data are scarce, the updates were not enough.

Then we employed an early stopping mechanism with patience set to 10, meaning that if the validation loss did not change substantially in 10 epochs, then the train is stopped. This is essential to prevent overfitting and reduce training time.

We set the number of epochs to 50, since one epoch, including train and validation, took 2 minutes approximately. We also saved the checkpoint of the model resulting in the lowest validation loss, to store the best performing state of the model.

4.7 Loss Functions

For this task, we decided to experiment also with the selection of the loss functions, given the nature of the dataset we had to work with. The choice of the loss function is crucial as it directly influences the learning process of a model, impacting its ability to accurately optimize and generalize from the training data to unseen examples. Selecting an appropriate loss function aligns the model's objectives with the task at hand, influencing the convergence behavior and overall performance.

Since we were dealing with a very complicate dataset, given the scarcity of images, the focus of our analysis was on examining the potential influence of outliers on the training process. For this reason we decided to experiment a various number of alternatives in order to deal and prevent this unwanted behavior.

In the end, the MSE was used for the tests and inference results obtained using this loss are displayed in the next section.

In order to select the optimal loss, we trained every model architecture using every loss among the ones presented in this section. Consequently, we compared, for each model architecture separately, the scores obtained by using such losses. We observed

that the one that gave the more stable and better scores were MSE, representing the best trade-off among simplicity and effectiveness. In particular, among every losses MSE was the one that, for every model architecture, provided us with the best scores. Even if it could be counterintuitive, since we experimented with very complicated loss functions as Huber loss, then we can, again, address to the Occam's Razor principle for which sometimes simpler options work better.
In the following section, all the loss functions employed in our experiments will be presented.

4.7.1 MSE Loss

This was the very first loss function used for this task. It is a standard choice in regression task. It is the simplest and most common function, nonetheless one of the most complete.

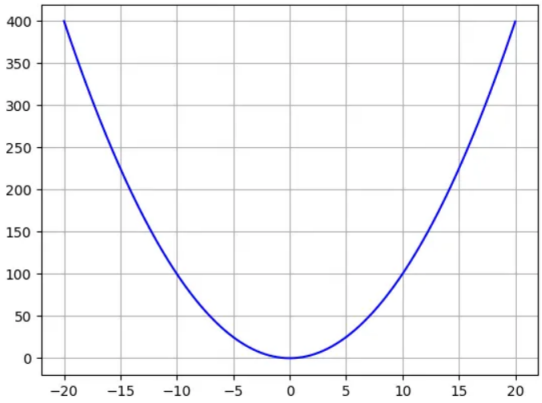

Figure 4.11: MSE loss function.

To calculate the MSE, it is necessary to take the difference between the model's predictions and the ground truth, square it, and average it out across the whole dataset. The resulting loss will never be negative, since we are squaring the errors. The MSE loss is formally defined by 4.3.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2 \tag{4.3}$$

The main advantage of MSE is that it is great for ensuring that the trained model has no outlier predictions with huge errors, since the MSE puts larger weight on these errors due to the squaring part of the function.
On the other hand, if the model makes a single very bad prediction, the squaring part of the function magnifies the error. In any case, we usually do not take this into account since we are aiming for a well-rounded model that performs good enough on the majority of the samples.

4.7.2 MAE Loss

The Mean Absolute Error is a little different from MSE, but this little difference introduces different behavior in the training of the models.

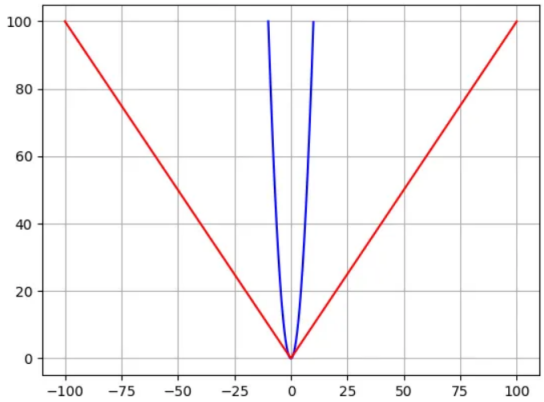

Figure 4.12: Comparison between MAE (red) and MSE (blue).

In fact, to calculate the MAE, one takes the difference between the model's predictions and the ground truth, apply the absolute value to that difference, and then average it out across the whole dataset. Hence , on the contrary of MSE, the square of the error is not performed. MAE, as MSE, will never be negative since it is a sum of absolute values of errors.
The MAE loss is formally defined by 4.4.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|x_i - y_i| \tag{4.4}$$

The main advantage is that, by considering the absolute values, every error contributes uniformly on a linear scale. This ensures that, unlike MSE, outliers do not overly influence the outcome. Consequently, MAE loss function offers a balanced and comprehensive assessment of the model's performance.
On the other hand, the main disadvantage is that the large errors coming from the outliers end up being weighted the exact same as lower errors. This could lead to the model to perform exceptionally well in general, yet occasionally producing a few notably inaccurate predictions.

4.7.3 Huber Loss

The Huber Loss offers a good balance between MSE and MAE. In particular, it was selected since it is pretty strong against outliers, being less sensitive to them than the MSE as it treats error as square only inside an interval.
To easily explain 4.5, the loss depends on the error itself. In particular, for error values lower than δ MSE is used, otherwise MAE is selected.

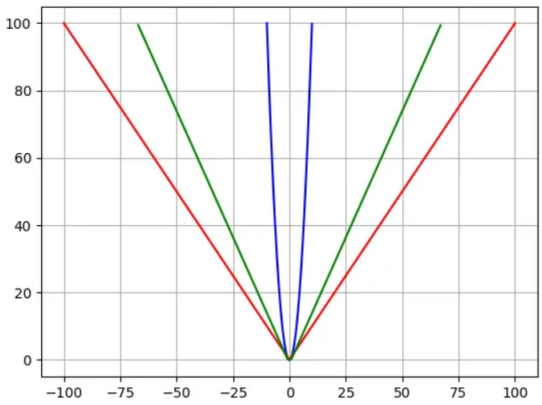

Figure 4.13: Comparison between MAE (red), MSE (blue) and Huber loss (green).

Utilizing MAE for larger loss values reduces the impact of outliers, ensuring a balanced model. Simultaneously, employing MSE for smaller loss values maintains a quadratic function near the center.

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & if\ |(y - \hat{y})| < \delta \\ \delta((y - \hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases} \tag{4.5}$$

As we can see from 4.13, Huber loss stands in the middle between MAE and MSE, giving an intermediate importance to outliers. This was used to prevent outliers impacting heavily on the training, also given the scarcity of data in the dataset.

4.7.4 Log-Cosh Loss

The Log-Cosh Loss function is a smooth approximation of the Mean Squared Error (MSE) loss and is particularly useful in scenarios where one wants a robust loss function that is less sensitive to outliers. The loss is formally described by 4.6

$$Log - coshloss = \frac{1}{N}\sum_{i=1}^{N}(\cosh(x_i - y_i)) \tag{4.6}$$

This loss was tried along with Huber loss to smooth the influence of outliers on the training phase. In particular, it is smoother than MSE because the hyperbolic cosine function has a quadratic growth rate for large values, similar to the squared term in the MSE loss. However, it doesn't explode as rapidly as the squared term, making it less prone to being heavily influenced by extreme values.

4.8 Analysis and comparison of the results

The metrics used to evaluate the results on the test set of the model developed were the mean and the standard deviation of the distance from the correct prediction, as requested from the client.

Results are displayed in table 4.1. We can see that the best performant model is the one that combines the single accident image along with the numerical features: this is probably due to the fact that its processing represents a good balance among image and feature information extraction, exploiting all informative details from this data.

SVM, one of the architecture firstly tested to solve the task, proved to be a solid baseline, solving the problem with ease by not using any complicate deep learning architecture unless ResNet as feature extractor.

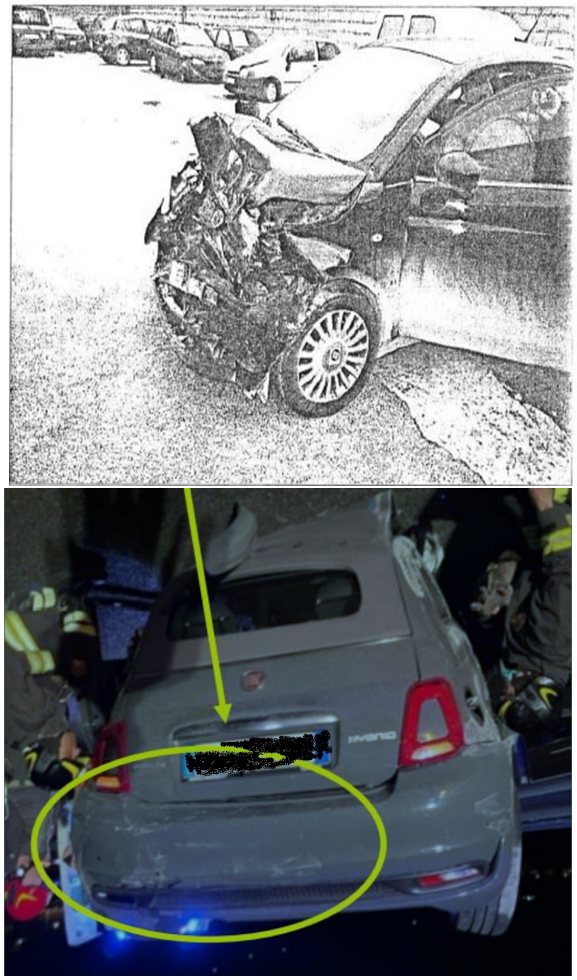

Figure 4.14: Example of low quality images in EES dataset.

Similarly, adhering to the principle of simplicity, the standalone image model exhibited excellent performance. Assuming that all the relevant accident information is inherently embedded in the images, it is intuitive that a model structured in this manner would effectively extract and leverage the essential data.
As already explained, the multi image models results were worse than the one obtained by simpler models. This could be attributed to the limited availability of data, suggesting that previous model complexity might be suitable, as a more intricate model could potentially struggle to effectively accommodate the sparse dataset at hand.

One of the problems, in addition to the scarcity of data, was the fact that images in the dataset were not always qualitatively perfect, as we can see in 4.14. In this cases

Model	Mean	STD
SVM	8.17	4.56
Single Image	5.73	3.96
Multi Image	9.68	4.19
Mixed Single Image	4.75	3.23
Mixed Multi Image	13.35	4.71

Table 4.1: Results of EES regression.

it is intuitive that even a deep learning architecture can not do anything, in front of extremely low resolution and noisy images. Removal was not an option, firstly because they represented a good slice of the dataset, and then because we would be left with too few images after that.

In particular we checked the test images that scored poorly (distance from the original target above 10) and we found out that these were always low quality images. For experimental sake, since we were told to not remove any image from the dataset, we tried to exclude these bad scorers from the test and we got a mean of 3.46 from the single image mixed model.

Despite this, to conclude this discussion is essential to compare the required application performances and the actual scores provided by the best model. In particular, the desired result was to classify every image in a range of ± 5 from the real value, with a standard deviation around 3, regarding EES regression. We can check from the results that we almost satisfied the desired performances, with a little margin on the standard deviation, that anyway was not an hard concern for the client. in addition to the desired metrics, we also provided the customer with a measure of the Mean Percentage Error(MPE), to provide a relative metric that highlight the proportion between the error and each observation. MPE is mathematically described by Equation 4.7, where $\hat{y}_i$ and y_i are respectively the predicted and the actual value, while N is the total number of instances.

$$\text{MPE}(y, \hat{y}) = \frac{100\%}{N} \sum_{i=0}^{N-1} \frac{y_i - \hat{y}_i}{y_i}. \tag{4.7}$$

In particular, out best performant model, exposed a MPE of 14%, which is a good value considered the amount of data available.

In the end, it was surprising that our models could regress the EES values so accurately even in presence of such images, setting an extremely good baseline for when more data will be added to the infrastructure.

Chapter 5

Car Damage Segmentation

Damage detection and segmentation is a new deep learning challenge for insurance companies that want to have a quick estimation of the damages on the cars after a collision, directly from the photos or videos acquired on the spot.
In particular, we wanted to have a tool that could apply a segmentation mask directly on the car image, especially on the part on which the damage is detected. In the future, the goal would be to experiment this segmentation procedure by applying it directly on 3D models of cars, reconstructed as explained in the next chapters.
This could be addressed as an instance segmentation task: it involves identifying and delineating individual objects within an image. Differently from semantic segmentation, which groups pixels into common categories (e.g., road, person, sky), instance segmentation aims to differentiate between each distinct instance of an object. In other words, it not only labels the pixels belonging to a certain class, but also it distinguishes between separate instances.

5.1 Dataset

The dataset [6] contains 10000 labeled images and it was divided into train (7000 images), validation (2000 images) and test (1000 images) sets. An example of image contained in the dataset is shown in Image 5.1.

Each image was labeled with a segmentation mask which consents the identification of damages on the bodywork. The labels covered a wide range of possible damages

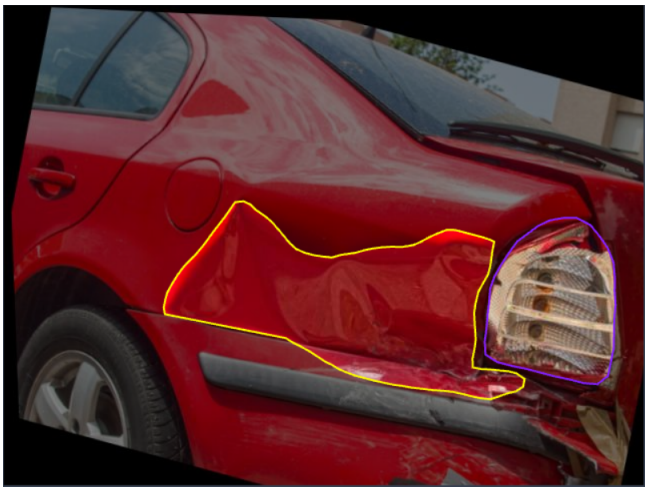

Figure 5.1: An example of an image from damage segmentation dataset.

that can be encountered on a car. In particular, classes of damages are:

- **crack**: indicates the presence of a crack in the bodywork.
- **crash**: indicates the presence of a massive crash on the car.
- **dent**: indicates the presence of a dent on the bodywork
- **dislocated part**: indicates the dislocation of a bodywork's part.
- **glass shatter**: indicates whether the glass is broken.
- **lamp broken**: indicates the presence of broken lamp.
- **no part**: indicates the presence of a missing part in the bodywork.
- **rub**: indicates the presence of a rub.
- **scratch**: indicates the presence of a scratch
- **tire flat**: indicates the presence of a tire flat.

Images did not have a standardized size, so a processing phase was needed.

5.1.1 COCO Format

The car damage segmentation dataset was provided in COCO format. In particular, COCO (Common Objects in Context) format is a standard format for storing and

sharing annotations for images and videos. It was developed for the COCO image and video recognition challenge, which is a large-scale benchmark for object detection and image segmentation.
In COCO format, image annotations are stored into a JSON file containing information about the images and their respective annotations. In particular, in our case the JSON file contained a list of *images*, each of them described by its image id, file path, size and other metadata, and a list of *categories*, each one linked to a damage segmentation class trough an ID.
In addition, it presents a list of *annotations* containing a set of annotated objects. Each of them is described by an ID, an image ID and a category ID, which specify the respective image and damage category, as well as the bounding box, the area and the segmentation mask information. In particular, the bounding box is described by the points identifying its corners, while the segmentation information are stored as a set of points defining the segmentation polygon.
Then, once obtained the dataset in this format, we were able to host it in BeeYard infrastructure, trough the usage of custom scripts that made use of the Python SDK of BeeYard in order to handle the upload of data in this format.

5.2 Data Processing

Since we were dealing with visually annotated images, a normal image processing could not work here. In fact operating solely on the image would completely make useless the bounding box and mask information.
For this reason we needed a dedicated library, which is able to handle image processing and augmentation in the context of object detection and segmentation. This library was Albumentations[5], a Python library for fast and flexible image augmentations which can handle bounding box and mask data into its transformations.
In particular, since images had all different sizes we wanted to standardize their dimensions with a geometrical resize to 456x456 pixels. Before that we applied padding, in case the images were too small. Since there was no geometric connection within the damages on the car, we also decided to apply random horizontal

flips with a low probability to additionally augment the data. In the end, the image was converted to a PyTorch tensor.
This whole transformation processing handled automatically the consequent bounding box and mask adaptation, which needs to fit to the new transformed image.

5.3 Architectures

The architectures employed to solve this task were Mask R-CNN[7] and Yolov8[2]. The first one represented the first approach to the task, since it established as state of the art in this context for several years in the past. It seemed the perfect point where to start our experiments to obtain the first results.
Then we moved to Yolov8, one of the most recent and innovative technologies built since this time to address this task. We decided to use the most recent version for the fact that we are constantly seeking into applying the newest technologies available.

5.3.1 Mask R-CNN

In the realm of computer vision, the quest for more accurate and granular image understanding has led to the development of advanced frameworks that transcend traditional object detection. Mask R-CNN, an evolution of the Faster R-CNN architecture, stands at the forefront of these advancements, offering a robust solution to the challenges of instance segmentation.
Mask R-CNN has emerged as a pivotal tool for tasks requiring not only precise object localization but also intricate pixel-wise segmentation.
This model is conceptually very simple, since given an input image will produce three distinct outputs for each object instance : a class label, a bounding box offset and a segmentation mask.
In particular, as we can observe in Image 5.2, the model consists of a two-staged architecture. The first one is the Region Proposal Network. Here, image features are directly extracted from the picture using a deep learning model, for example a ResNet50[10], and then are processed in order to predict candidate object detection

boxes, the so called Region of Interest. This part was also done in Faster R-CNN. In the second stage, the RoIs are processed by ROI Align: this extracts features from each of the candidate boxes, that will be used for the classification, detection and segmentation tasks.

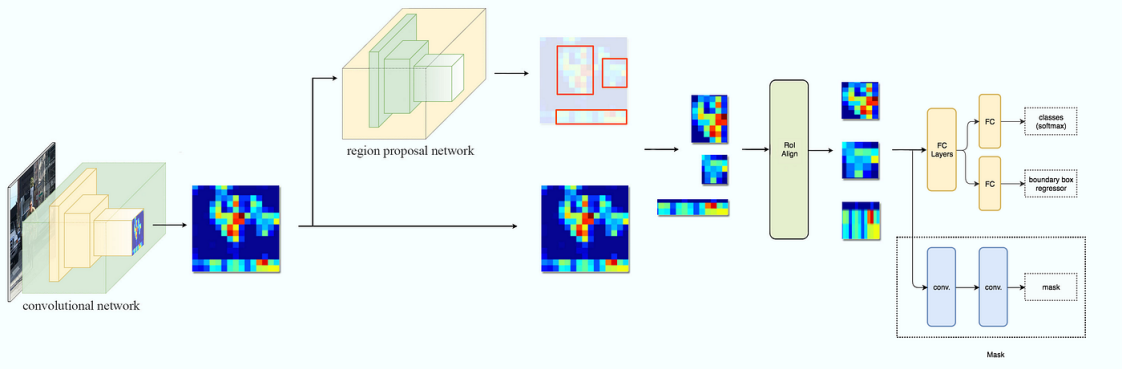

Figure 5.2: Mask-RCNN architecture diagram.

Mask R-CNN introduced a revolution in terms of instance segmentation. In fact, previously with Faster R-CNN, ROI Pooling was used to extract features from the Region of Interest. Furthermore, this mechanism operates trough successive quantizations in order to produce small feature maps. This process causes a misalignment between the ROI and the extracted features, which have a large negative impact on the prediction of pixel-wise masks.

ROI Aling, instead, works by properly aligning the input and the extracted features by avoiding quantizations and replacing them with bilinear interpolations, in order to compute the precise values of the input features at four evenly sampled locations within each Region of Interest (RoI) bin, and then aggregating using max pooling.

In fact, differently from Faster R-CNN, Mask R-CNN is able to handle instance segmentation task, and then output a segmentation mask for each detected object, by performing it in parallel within the classification and bounding box detection.

In particular, the backbone used to compute the features from the entire image was the ResNet 50 FPN, which proved to be the most efficient architecture to integrate, giving excellent gains in both accuracy and speed.

Since we were dealing with 3 tasks at the same time, namely classification, object detection and segmentation, we needed a loss for the model that entangled all the

errors and improvements for each class. In particular the loss employed in Mask R-CNN was the sum between the loss of the classification, the loss of the bounding box regression and the loss of the segmentation.
Furthermore, the loss of the bounding box regression was the L1 loss between the original bounding box and the predicted one, while the loss of the classification corresponded to the cross entropy loss between the predicted and ground truth class. In addition, the loss for the segmentation was defined by the cross entropy between the predicted mask and the ground truth one.
Formally the loss function is described by Equation 5.1.

$$Loss = Loss_{cls} + Loss_{Bbox} + Loss_{mask} \tag{5.1}$$

With:

$$L_{cls} = \sum_{i=1}^{N} \sum_{j=1}^{C} y_{ij} \log \hat{y}_{ij} \tag{5.2}$$

Where N is the number of bounding boxes, C is the number of classes and, yij and $\hat{y}_{ij}$ represents the ground truth and predicted class, respectively.

$$L_{Bbox} = \sum_{i=1}^{N} \sum_{b \in x,y,w,h} L1(b_i - \hat{b}_i) \tag{5.3}$$

Where N is the number of bounding boxes and b_i and $\hat{b}_i$ represents the ground truth and the predicted bounding box.

$$L_{mask} = -\sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{M} y_{ijk} \log \hat{y}_{ijk} \tag{5.4}$$

Where M is the height and width of the mask, j is the index pointing to the actual mask, while $\hat{y}_{ijk}$ and y_{ijk} represents the predicted and the ground truth mask respectively.

Parameters and hyperparameters

Regarding the training of this model, in order to make the loss function defined in the previous section decrease, we employed an SGD optimizer, set with an initial learning rate of 0.01 and a momentum of 0.9. We also used weight decay set to 0.0005 in order to prevent overfitting during the training process.
A linear learning rate scheduler was used in order to help the optimizer in reaching the convergence.
Another way to avoid overfitting from this process was to employ an early stopping mechanism, which stopped the training if the validation loss did not improve for 10 consecutive epochs.
In order to evaluate the model during training, we made use of a custom python class COCOEvaluator, which served as an intermediary in order to deal with the real time display of evaluation loss during training, by receiving the output of the model at each epoch, and then computing the final score by aggregating classification, bounding box regression and masking score.
The model had been trained for a total of 20 epochs, showing a great learning capability by reducing the train loss and validation loss to 0.0547 and 0.0624 respectively.

5.3.2 Yolov8

Yolov8 [2] is a new State of the Art model that can be used for classification, object detection and instance segmentation. It was created by Ultralytics[8], who gave the birth also to the influential and industry-defining Yolov5[**?**] model. YOLOv8 includes numerous architectural and developer experience changes and improvements over YOLOv5.
In particular, the Yolo (You Only Look Once) family of models has become very famous in the computer vision world in recent years. This is given by the fact that all this models can provide very high accuracy, while at the same time mantaining a small size. The result is that these models can be trained on a single GPU, which makes them easily accessible.

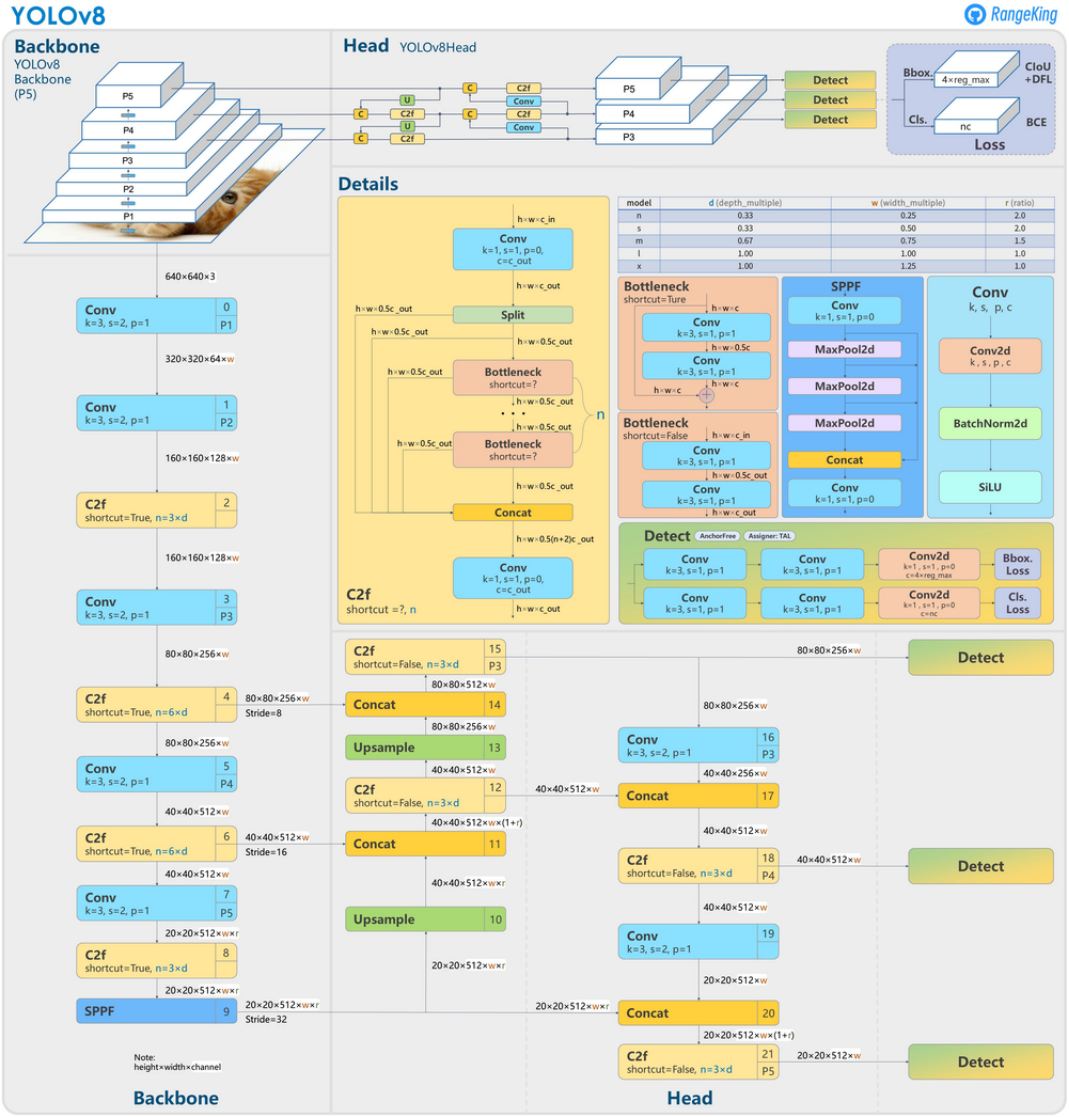

Figure 5.3: Yolov8 architecture diagram.

Regarding the improvements made by Yolov8, the first noticeable feature is that it is an anchor-free model. Yolov5 was anchor-based, since a set of predefined bounding boxes of a certain height and width were specified. These boxes are defined to capture the scale and aspect ratio of specific object classes one wants to detect and are typically chosen based on object sizes in the training datasets. The main problem here is that these anchor boxes represent the distribution of the target benchmark dataset, but not the distribution of a possible custom set.

Yolov8 predicts directly the center of an object, instead of the offset from a known anchor box. In particular, this feature speed up Non Maxima Suppression phase by reducing the number of box predictions.

Another important feature that has been added to Yolov8 is the Mosaic Augmentation: this is an online augmentation that is done automatically during training that involves stitching four images together, encouraging the model to acquire knowledge of objects in novel positions, amidst partial occlusion, and in the presence of varying surrounding pixels. This feature is turned off after 10 epochs, since results show that going onwards it degrade the results.
We decided to employ a pretrained version of the model by loading the weights available from the official library. Furthermore, we decided to finetune it on our data: this time we used a different dataset, which was available on BeeYard. Nonetheless, it was encoded in the same COCO format, and the processing done on the images was the same as the one that we used for the previously exposed set.

Parameters and hyperparameters

We set up a training phase composed by 2000 iterations: this was possible since the model is incredibly fast in processing images, which allowed to set a great number of iterations.
In addition, we adjusted patience to 500 to regulate the effect of overfitting during the training.

5.4 Analysis of the results

Proceeding with a qualitative analysis on the results brought by both the employed models, we can state that we managed to reach very high standards in damage segmentation. We visually tested both on images present in the test set, and on real car images acquired by us and, as a consequence, the models showed up great adaptability, being able to agilely detect and segment images coming from both contexts.

As we can see from 5.4, the results of car damage segmentation on images respected the previous preamble. The models clearly detected the area where the damage was located and precisely highlight it with a mask. If the car presented more than one

Figure 5.4: Examples of car damage segmentation.

damage then each of them was described by a different color.

Additionally results of the inference on real images, acquired by us on the streets, were astounding, as 5.5 show. This was a real world challenge since images found in an online dataset are usually better acquired. Most importantly the models, which were fitted with images coming from the same dataset could expose better results with images coming from the same set, due to hidden patterns or acquisition modes that differed from real world acquired data.

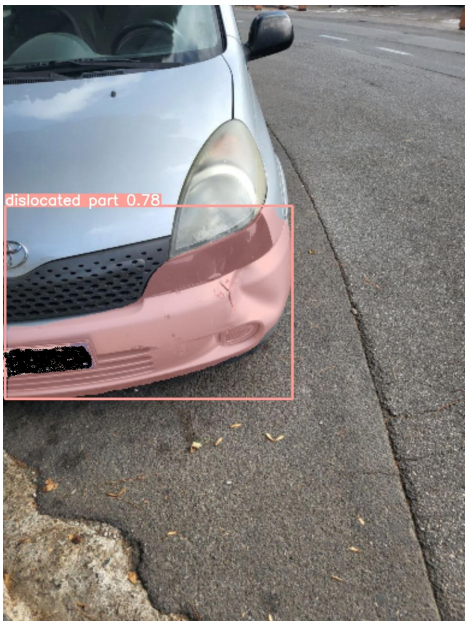

Figure 5.5: Example of car damage segmentation on privately acquired image.

This is very important since we were researching to put in production a model that should embrace real world problems and be able to act as a consistent tool for a person operating on the field. Thus it was very important having a model which was

not only performant on benchmarks and test sets, but a model which was robust to every image that is submitted.

By analysing the performances of each one of the model alone, we can see that as expected Yolov8 accomplished the task better than Mask R-CNN. This was pretty predictable and it is intuitive since it has obtained state of the art results on several other tasks, and introduced a great innovation in this context, outperforming the others. We could appreciate this by noticing that in certain images Yolo could spot even small details that Mask R-CNN could not get.

Nevertheless, the former model's performances are not so significantly inferior: even if it is an older model and is not so innovative as the other, it was still able to produce very good results, at least on the test set. The difference with Yolo can be felt by analyzing the real world images.

Yolov8, in fact, did an extremely good job on them, being able to spot almost every damage on the car. On the other hand, even if Mask R-CNN managed to detect the most evident collisions, sometimes we noticed that it could not spot smaller details.

Figure 5.6: Effect of reflections in the segmentation: Mask R-CNN(left) vs Yolov8(right).

In particular, we noticed that sometimes it got misled by light reflections. Especially

	mAP BBox	mAP mask
Mask R-CNN	0.682	0.669
Yolov8	0.853	0.836

Table 5.1: Mask R-CNN and Yolov8 mAP scores.

in the cases of dents, for which a cavity is created in the bodywork of the car by a collision, light played a crucial role in the hardness of spotting the damage.
As we can see from 5.6, Mask R-CNN could not detect the presence of the dent in the image, but on the other hand the result is supported by Yolov8, which managed to segment the damage, proving to be resistant both to reflection and shadows.
Apart from this qualitative results, which constituted our main source of model evaluation, we additionally generated Mean Average Precision (mAP) scores for both object detection and segmentation task.
In particular, this metric is based on the computation of Precision and Recall, where a True Positive is determined by the fact that Intersection over Union of the mask or the bounding box with respect to the ground truth is over a certain threshold. In our case we set this threshold to 0.5. Consequently, Average Precision is computed from the Area Under the Curve (AUC) of the Precision-Recall curve and then, the mean is taken across all the classes to obtain the overall mAP.

Results in Table 5.1 provide a solid proof that confirms the qualitative analysis done by analysing the inferred masks and bounding boxes produced by the models.

Chapter 6

3D Rendering

This task was accomplished in order to provide insurance companies with a tool that let them quickly reconstruct the circumstances of a collision from a video acquired directly on the scene.
This could be a very important instrument to acquire every context detail into a 3D render, in order to have a deeper understanding of the scene and allowing to a more appropriate forensics analysis.
Technically, the task consisted of submitting a video to our model, which then will be processed in a 3D reconstructed scene. The crucial detail here was that, since a video can not possibly acquire every 360 degree angle within its acquisition, the model would have to infer novel views that can not be visualized in the video itself. To solve this issue we had to employ the most recent technologies for 3D Rendering, namely Neural Radiance Fields[12] and Gaussian Splatting[11].

6.1 COLMAP

A video is a collection of pictures, each one acquired from a sequential but different place in the space, with different position but also direction. Since a simple video does not encode information about the camera positions that contributed to its formation, we needed a tool that reconstructs the 3D structure of the scene from it. This process is called Structure from Motion and we used COLMAP[15] in order to accomplish this stage.
Theoretically, Structure-from-Motion (SfM) is the process of reconstructing 3D

structure from its projections into a series of images. The input consists of a set of overlapping images of the same object, taken from different viewpoints.

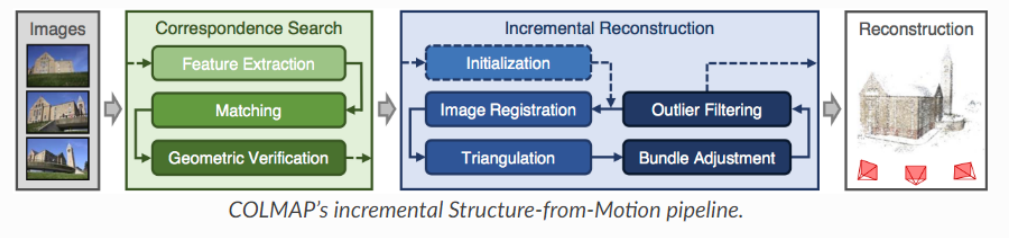

Figure 6.1: COLMAP pipeline.

The images that compose the video will always overlap since the camera is gradually translating for each acquisition. Instead we will not have a smooth video, but just a disconnected sequence of frames.
The output was a 3D render of the object, and the estimated intrinsic and extrinsic camera parameters of all images. Since the 3D reconstruction would have been managed by NeRF or Gaussian Splatting, then we were only interested in the intrinsic and extrinsic parameters inferred.
COLMAP divides this process into three stages:

1. Feature detection and extraction
2. Feature matching and geometric verification
3. 3D reconstruction

Thanks to its modularity, COLMAP allows user to just extract intrinsic and extrinsic parameters, avoiding the 3D formation and reconstruction. This was crucial for our application, otherwise it would have consumed useless time.
COLMAP is crucial since the models presented above do not work directly with the video from scratch, but they need Structure from Motion information, since the video does not provide any data about camera positions and parameters.

6.2 Neural Radiance Fields

Neural Radiance Fields (NeRF) [12] represents a cutting-edge approach in computer graphics and computer vision, offering a powerful paradigm for capturing and rendering 3D scenes.

NeRF leverages deep neural networks to model the volumetric scene function directly. This novel technique enables the generation of high-quality, detailed 3D reconstructions and realistic renderings from sparse and unstructured 2D images.

Diverging from traditional approaches that depend on explicit geometric representations or surface meshes, NeRF directs its attention to implicitly learning the inherent properties of a scene. Through training a neural network to predict radiance values at any specific point in space, NeRF attains notable adaptability, excelling in capturing intricate scene details such as fine textures, reflections, and lighting effects. This implicit representation contributes to a more precise and efficient synthesis of 3D scenes, particularly in environments characterized by complex geometries and varied lighting conditions.

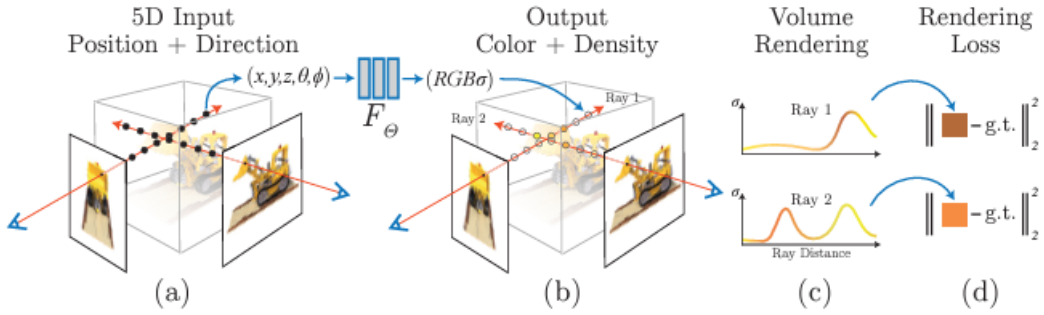

Figure 6.2: Neural Radiance Field structure.

Directly, this technology offers a neural network that can reconstruct complex three-dimensional scenes from a partial set of two-dimensional images.

In Neural Radiance Fields theory, the scene is represented by a 5D function, whose input are the 3D location of the pixel that we want to render (x, y, z) and (θ, ϕ), which represents spherical coordinates specifying the direction. Since the camera is always pointed at the object, we only need two rotation parameters to fully describe

the pose. In fact, ϕ represents the inclination, while θ represents the azimuth angle. The output of this function is an emitted color (r, g, b) and a volume density σ.
In order to train this architecture, we were firstly needed to generate a sampled set of 3D points by marching camera rays trough the scene.
Every ray is described by two vectors:

- v_0: it corresponds to the camera position.
- v_d: it corresponds to the direction.

Then the parametric equation, as in Equation 6.1, describes any point in the ray.

$$P = v_0 + t * v_d \tag{6.1}$$

Then, in order to perform the so called "ray marching", we set the parameter t larger and larger until the ray reached some interesting location in the 3D space. Then we needed to sample a set of this points, possibly informative (in the sense that they are not empty points of space, but they belong to the object). There are many approaches for sampling, a remarkable one is Hierarchical Volume Sampling for which we allocate samples proportionally to their expected effect on the final rendering.
The last preprocessing stage that was done on this sample points is Positional Encoding. This is done in order to map these points to an higher dimensional space using a set of high-frequency functions. The main reason was that that fine-grained elements like vibrant colors, intricate geometry, and texture contribute to the perceptual sharpness and vividness of images to the human eye. Then if the model could not represent these kind of features, it would produce images that might look bleak or over-smoothed.
Subsequently, these 3D points along with the viewing direction were fed to an MLP to produce a color and a volume density. Consequently, volume rendering techniques were employed to accumulate densities and colors into a 2D rendered image, which was comparable with the original image trough a loss function and finally the back-propagation process started. The whole process is well represented in 6.2

Since the full procedure is differentiable the MLP was trained by using gradient descent to minimize errors between each observed image and all corresponding views, rendered from the representation.
In particular for our task, we employed an already internally implemented version of Mip-NeRF 360[3]. This recent version of NeRF extended the original one with the objective of reducing blurring effects and visual artifacts.
NeRF uses a single ray per pixel, which often caused blurring or aliasing at different resolutions. Mip-NeRF uses a geometrical shape known as a conical frustum to render each pixel, instead of a ray, which reduces aliasing, makes it possible to show fine details in an image, and reduces error rates by between 17-60%. The model is also 7% faster than NeRF. In particular, Mip-NeRF 360's name is given by the fact that it is optimized to render 360 degree scenes around an object, which is the principal scenario when dealing with our car collision 3D rendering.
In addition, this NeRF version is considered by the community as the actual State of the Art at this time, since it produces the highest results on most of 3D reconstruction benchmarks.

6.3 Gaussian Splatting

This technology was released during the period of the internship and we immediately decided to test it out for the purpose of our 3D car collision reconstruction.
In detail, 3D Gaussian Splatting is a rasterization technique that allows real-time rendering of photorealistic scenes learned from small samples of images.
In computer graphics, rasterization is the task of taking an image described in a vector graphics format and converting it into a raster image.
This involves:

1. Having data describing the scene.
2. Draw the data on the screen.

In this case we are trying to rasterize gaussians, as in Image 6.3.
These 3D gaussians are described by:

- **Position**: where it is located in xyz space.

- **Covariance**: how it is stretched, defined by a 3x3 matrix.
- **Color**: which color it is, in RGB.
- **Alpha**: how transparent it is.

Then, regarding the methods that let millions of gaussians to be combined in order to reproduce a real scenario acquired by a video or a set of images, we have to investigate how this technology works.

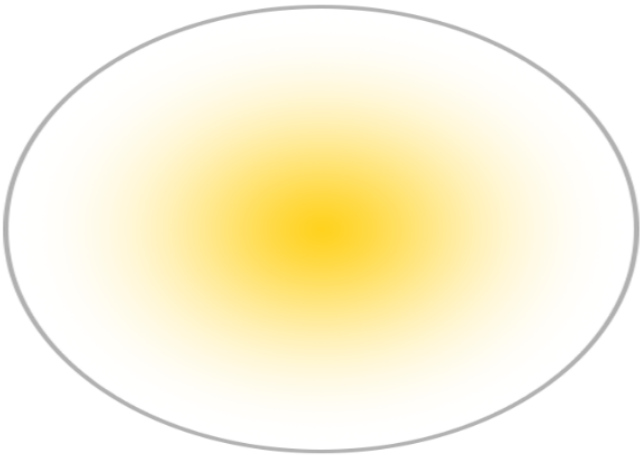

Figure 6.3: Rasterization of a single Gaussian.

The whole process starts by Structure from Motion done using COLMAP, as explained in 6.1. In particular, it starts with the same input as in NeRF methodology, but it also takes into account the sparse point cloud which is produced during the SfM process for free.
Consequently, from this point cloud, every point is converted to a Gaussian, which is described only by position and color, two features that can be inferred from Structure from Motion data. Although the rasterization could be possible, in order to obtain high quality results we need to train the architecture in order to find the perfect combination of Gaussians.
In particular the training steps are:

1. Pick a random camera.

2. Rasterize the Gaussians, using Differentiable Gaussian Rasterization, in order to produce a 2D image.

3. Compute the loss between the ground truth and rasterized image.

4. Adjust the Gaussian parameters according to the loss.

5. Apply densification and pruning.

Furthermore, this algorithm is repeated for several times, usually 7000 or 30000 iterations, in order to produce optimal results. The diagram in Image 6.4 shows the optimization process.

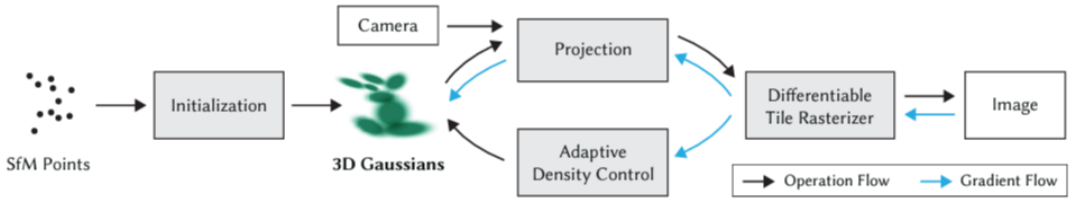

Figure 6.4: Structure of Gaussian Splatting optimization.

To get more in detail, the technique used for the rasterization is crucially needed to be both fast and differentiable. This because the process needed to be executed in almost real-time and also we wanted it differentiable in order to backpropagate the loss and adapt the Gaussian parameters to optimize them.
To give a brief explanation of the Differentiable Gaussian Rasterization method, it starts by splitting the screen into 16x16 tiles and filtering out 3D Gaussians based on both the view frustum and each individual tile. Then, each Gaussian is instantiated according to the number of tiles that overlap, assigned with a key which combines view space depth and tile ID and then sorted according to this key. In this way a list for every tile is created, loaded with the information of the Gaussians overlapping it, ordered by depth. Then for each pixel of a tile we accumulate the value of color and α of the closest Gaussians, until a target saturation for α is reached, meaning that we have found enough Gaussians that describe the surface of the object.
Once the image is rasterized, the loss within the ground truth is computed. In particular, the loss function is an L1 combined with a D-SSIM term, as shown in

Equation 6.2 .

$$L = (1 - \lambda)L_1 + \lambda L_{D-SSIM}, \lambda = 0,2 \tag{6.2}$$

Then the optimization is carried out by Stochastic Gradient Descent and the Gaussian parameters, since the rasterization process is differentiable, are updated. The last thing that is left to do is Adaptive Gaussian Control, which is responsible of the handling of poorly performant Gaussians.
This process is greatly helpful since it helps to move from a random sparse set of gaussians to a denser set that best represents the scene
In particular if the α of a Gaussian gets lower with respect to a threshold, meaning that it is almost transparent, then it is removed. On the other case, if the gradient is too high for a Gaussian, meaning that it is wrong, then if the Gaussian is large it will be divided, while in the case it is small, the Gaussian is cloned and moved in the direction of the positional gradient. This, in particular manages regions that are not well reconstructed yet, and by moving and splitting Gaussians we are trying to reconstruct deeply the missing scene and its details.
Then once the optimization process is done, the refined scene representation is available as a point cloud (.ply) and ready to be viewed. We used the available Gaussian Splatting interactive viewer in order to display the results. This is developed using SIBR[4] framework, a specialized collection of libraries and toolkits for quickly implementing Image-Based Rendering (IBR) algorithms. This allowed us, not only to view the resulted model, but also to edit it by engraving the desired object into a cube of variable dimension. The purpose for it was putting the focus on the object and eliminating the noise and background.

6.3.1 Masking

Since the background of the 3D reconstructed object is not always informative, we tried to find a technique that was able to mask it out automatically during the optimization process.
The most intuitive, but yet not effective, idea of masking the images given in input

can not work. This because COLMAP feature extraction process relies mostly on background feature points in order to match the images, and then provide the SfM structure. Consequently this masking technique can not be applied.
The other attempt we tested was feeding COLMAP with the original images to produce Structure from Motion. Then, we masked the images to compare them with the rendered ones. In fact, every camera is associated with its own ground truth image, which will be assessed within the rendered one to produce the loss. The idea was to influence the model by feeding the masked image instead of the original one to reconstruct only the object and not the background within the computation of the loss.

6.4 Analysis of the results

In the presentation of our Gaussian Splatting 3D Rendering results, we delve into the efficacy of our approach in transforming sparse point cloud data into a visually compelling and detailed 3D representation.
As we can see from Image 6.5, the results brought by the Gaussian Splatting optimization procedure were astounding. Every detail on the car, including the license plate, that have been concealed for privacy reasons, were precisely rendered with a high level of quality and definition.

Figure 6.5: Gaussian Splatting car collision reconstruction.

Especially by zooming in within the interactive viewer we could see that even the car interior had been rendered, including small details like the steering wheel and

the pattern of the front seats, which were hard to reproduce since they are behind the front glass, making it subject to reflection. In addition, we had to remember that these views are completely generated, since there is no image in the dataset that zoom so close on the car interior.

Another detail that came immediately to our eyes is the bumper damage located on the side of the vehicle, precisely on the back door. Here the optimization process had been extremely precise in rendering , with astounding high detail, a very complex damage. Through 3D reconstruction, we were able to analyze the extent of the damage more thoroughly, as we can estimate its depth and extent. The fact that it was able to reproduce high level details was a crucial matter since we had to apply it in the real world, and the customer needed to be provided with the highest quality reproduction in order to be able to estimate the damage entity.

Another feature that made Gaussian Splatting efficient is the time employed to produce the result, which in the case of the example shown in Image 6.5 was under 40 minutes, included the SfM process. This is critical when one wants the customer to be able to process a real scene in quasi real time, making the 3D model available faster than other methods.

In fact, by comparing with Mip-NeRF 360 the same processing would have employed more than one day to achieve almost the same quality of the one made by Gaussian Splatting, which is a huge disadvantage when the insurance company wants to have the 3D model available as soon as possible. The computation with Gaussian Splatting, as it does not need a neural network to operate, is extremely fast and better than NeRF options in this field.

On the other hand, a little disadvantage when using Gaussian Splatting was that as we gradually moved away from the central object the background started to fill with noisy Gaussians, producing a sort of "cloudy" effect. This behavior could be limited trough the usage of the interactive viewer, by limiting the rendering area to a bounding box that encapsulated the object. In this way, only the interested area was rendered, leaving all the noise away from the visualization.

In conclusion, Gaussian Splatting proved to be an efficient solution to deal with the task of 3D reconstructing a car collision scene. Its highly detailed results and the fact that the whole process could be done in almost real time made it an appealing

solution to our task.

6.5 Damage segmentation on 3D reconstructed cars

Since the Gaussian Splatting method should have been engraved into a framework for the analysis of car accidents and collision scenes, it should have been possible to integrate it within the car damage segmentation process.
This would enable the client to just record one video from the scene, reconstructing it with Gaussian Splatting and then mask the rendered images.

Figure 6.6: Damage segmentation applied to a Gaussian Splatting render.

As we can see from Image 6.6, the mask seems to fit perfectly with the damage present on the vehicle, even if this image was a render produced by Gaussian Splatting. The two methods, segmentation and reconstruction, appeared to work very well together, making their integration possible.

Chapter 7

Few Shot 3D Rendering

Although the goal of reconstructing a damaged car from a series of photos or a video had been achieved, we decided to push further and test technologies capable of modeling a 3D scene from a single image. This was done for the fact that a video could not be always available, for this reason we needed a technology that could perform the task just from few images.

This new objective was evidently more complex in terms of accomplishment, as capturing the dynamics and spatial forms of an object from just one photo was decidedly challenging even for the human mind. This entails predicting novel views of the object, views that the model had never seen before and must therefore reconstruct from scratch.

This was challenging since the model needed to render a 360-degree panoramic of the object with only a single viewpoint available, and not hundreds as in the case of Gaussian Splatting[11].

In this chapter, we will explore DreamGaussian[16], a recent technology based on Gaussian Splatting, that is capable of reconstructing, even if with significantly lower accuracy compared to the latter, a 3D model from a single image.

7.1 DreamGaussian

DreamGaussian is an extremely recent technology for Image-to-3D that combines 3D Gaussian Splatting and a generative approach to reduce reconstruction time and prevent the need of more data. The whole process is shown in Image 7.1

To go more in detail of its functioning, a sphere filled with 3D Gaussians is randomly initialized. Gaussians are encoded in the same fashion as in Gaussian Splatting work. Then a pretrained StableDiffusion[14] is adopted as the 2D diffusion prior and it is the main responsible of the optimization process. In fact, since we were dealing with only one image, it generates novel views to reproduce the behavior seen in Gaussian Splatting and then compares the rendered Gaussian image with the generated one. Then the Gaussians are also optimized by using the densification and pruning mechanism seen in Chapter 6.

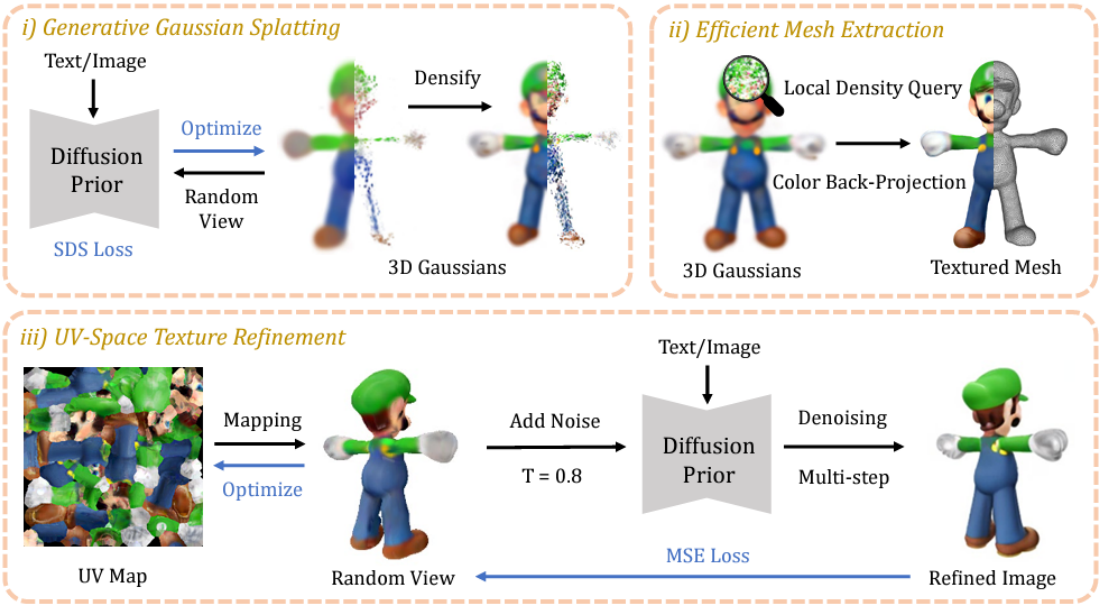

Figure 7.1: DreamGaussian pipeline.

Then, after the 3D Gaussian optimization, the mesh refinement process starts, consisting of two stages: Local Density Query and Color back-projection. In the first one, Marching Cubes algorithm is applied. This is a computer vision mechanism used to extract the polygonal mesh from a three-dimensional scalar field. In order to apply this algorithm, the 3D space must be densified into a density grid, and this is done by dividing the space into 16^3 blocks and eliminating the Gaussians which center is located outside each local block. Consequently, each block is queried into a 8^3 dense grid, which lead to a final 128^3 dense grid, and the weighted opacity of the remaining 3D Gaussians are summed up.

After obtaining the mesh geometry, the rendered RGB image is projected onto its

surface to create a texture. This is done by unwrapping the mesh's UV coordinates and setting up an empty texture image. Using different viewpoints, we generated RGB images for the texture. Each pixel from these images is then mapped to the corresponding location on the UV coordinates of the mesh.
The final process involves starting with an initialized texture, then rendering a blurry image from any camera view. Next, random noise is added to the image and a multi-step denoising process is run using a 2D diffusion prior, resulting in a refined image. This refined image is then used to optimize the texture through a pixel-wise Mean Squared Error (MSE) loss.

7.1.1 Analysis of the results

As we expected, the results could not overcome the one produced by Gaussian Splatting algorithm. Although, combining the fact that the whole training took less than 10 minutes and that the model had only one image at its disposal, the 3D renderings that came out were surprising.

Figure 7.2: DreamGaussian reconstruction of a car, the real image is in top left, while the others are renders of the 3D model from different views.

As it can be observed from Image 7.2, the renderings are not well refined in the details, even if the process managed to reconstruct the yellow brake inside the wheel and a little bit of the interior. In particular what got to our eyes was the noisy mesh that covers the 3D model, making the bodywork of the car not so smooth.
Apart from that, we have to remind that this 3D render was produced by feeding only one image in the process, the one in top left position in Image 7.2, and the

fact that it managed to resemble the whole geometry and shapes of the vehicle is remarkable.

In particular, one detail that we were surprised to see was that the 3D model inferred the presence of the back light, even if in the original image they were not captured.

Chapter 8

Future works and improvements

In the last days of this internship we started working on the development of a camera optimization algorithm for Gaussian Splatting made 3D rendering, which due to the lack of time we could not terminate.
More in detail, this consisted in regressing the camera parameters, namely the rotation and translation with respect to the reference center, of a camera that rendered a precise view of the splat that match an image of the car taking from that position.
To summarize, we needed to find from where the car picture was taken and then render the view from the Gaussian splatting with the regressed camera parameters.
Afterwards, we started to design a deep learning process which, given a starting initialization of the camera translation and rotation, begins with the rendering of the splat from that angle, and then compare the render within the original trough the use of a loss. Initially, we used MSE as the loss, since we had to compare a real car image within the rendered one.
This whole process looks intuitively fine, since we had configured the camera translation and rotation vectors as the model parameters, and then the rendering from the splat was done as in Gaussian Splatting training, so in a differentiable way.
Probably with more time at our disposal we can figure out where the process encountered a non differentiable operation and then solve the task.

More in detail, we wanted not only to compare the render within a real image, but also with a sketch, as in Image 8.1, provided by an insurance company, which is usually used on car collision reports to determine the positions of the damages.

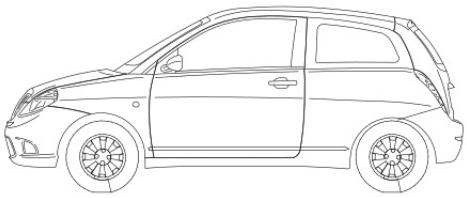

Figure 8.1: Sketch image of a car.

Subsequently, we wanted to regress the camera parameters in order to render the view of the car that matched the sketch, and finally run the damage segmentation inference on that render in order to have a proper mask describing the collision effects that could also be applied to the sketch itself.

The regression process remained unchanged as before, although here the main problem is that now comparing the rendered image and the sketch is no more trivial. In fact we tried out different strategies but none of them worked out.

Firstly we started to compare the masks of the sketch and of the rendered car image with the usage of the segmentation model. There were two main problems here: the first one was that, while we can mask out the sketch just one time and then reuse it, we had to mask the render in real time, iteration after iteration, and this made the whole process extremely slow. In addition, we tested out two ways of comparing the masked images but none of them produced feasible results. The first one was to basically perform Intersection over Union (see Equation 8.1) between the masked

sketch and the masked render, while in the second case we applied a proprietary function of OpenCV named *matchshapes*, which enables to compare two shapes and returns a metric showing the similarity. This function is computed based on the hu-moment values and it provides a value that the lower it is, the better the match.

$$IOU = \frac{areaofoverlap}{areaofunion} \tag{8.1}$$

Unfortunately none of these methods worked, and the explanation is very simple. As we can see from image 8.2, the masks are not so informative since they can create a lot of confusion in the encoding of different views. In particular, the masks of the back and front of the car are extremely similar, not permitting the establishment of a feasible metric.

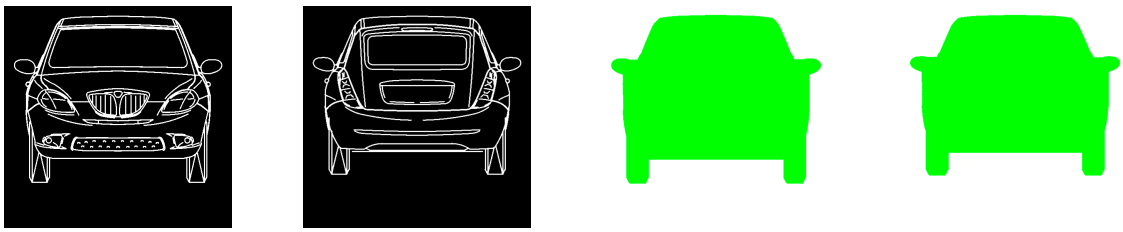

Figure 8.2: Front and back views of sketched car and their masked equivalents.

Then we tried to experiment the comparison between the sketch and the edged rendered image, in the sense that we applied Canny edge detection on it, trying to make it similar to the sketched one. In particular, we used *Canny* algorithm from OpenCV and then we finetuned the two thresholds to fit the gaussian rendered image.

Then in order to compare them we used two metrics: Hausdorff Distance and Cosine Similarity.

Regarding the first one, it is used to quantify the dissimilarity between two shapes or contours represented by pixel sets in images. Given two binary images A and B (where pixels are either foreground or background), the Hausdorff Distance measures the maximum distance from any point in one shape to the closest point in the other shape. The metric is mathematically described by Equation 8.2:

$$H(A, B) = \max\left(\sup_{a \in A} \inf_{b \in B} d(a, b), \sup_{b \in B} \inf_{a \in A} d(a, b)\right) \tag{8.2}$$

where A and B are the sets of points representing the two images and $d(a, b)$ is the Euclidean distance between pixels a and b.

On the other hand, Cosine Similarity can be applied to vectors obtained by flattening the pixel values of the images. Each image is treated as a vector in a high-dimensional space, where the dimensions correspond to individual pixels.

The cosine similarity between two image vectors A and B is given by Equation 8.3:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|} \tag{8.3}$$

where $A \cdot B$ is the dot product of the flattened image vectors, and $\|A\| \cdot \|B\|$ are the magnitudes of the vectors.

To compare the two metrics, Hausdorff Distance is more sensitive to the spatial arrangement of pixels, while Cosine Similarity focuses on overall pixel intensity patterns and is less concerned with the pixels spatial disposition.

The reason they do not have worked can be found in the fact that the application of Canny on rendered images is that these ones can be very noisy since they are extracted from the Gaussian Splatting itself. This results in obtaining some external edges that can not be removed even by moving the algorithm thresholds. Another reason is that these two metrics work very well when dealing with very simple contours, while in our case the edges were extremely dense and this probably confuse the computation.

Chapter 9

Conclusion

Thanks to this work, we were able to provide a 360 degree framework supporting car collision forensics and insurance analysis.

These tools can be integrated in BeeYard, in order to provide the client with a comprehensive platform to perform his analysis, being able to describe extensively the whole details of the accidents.

The whole research was done pursuing the achievement of State of the Art results, and for that reason by employing the most recent technologies available.

To sum up we developed a model for the classification of car's pose, reaching very high accuracy and integrating it to estimate the Energy Equivalent Speed involved in the accident.

We developed a model for the estimation of Equivalent Energy Speed, which was trained in presence of very few data, but nonetheless scored very well. This was possible also for the technique used to clean and augment the data, but also for the accurate setting of the architecture and training procedure.

We explored two solutions for the damage segmentation task, namely Mask-RCNN [7] and Yolov8[2], which then we deployed in order to provide an accurate tool for the detection of damages on cars.

Then we deployed Gaussian Splatting[11] model in order to reconstruct the collision scene by providing a video of it. The results were extremely satisfactory, being able to capture even the smallest details on the bodywork.

In the end, we deployed DreamGaussian[16] model to perform the same task done by Gaussian Splatting, but only with a single image. The results are poorer with

respect to this last one, but the fact that could infer details that are not present in the original picture makes it a good baseline to be investigated and further developed for the cases where the video is not available.

Bibliography

[1] Cardd dataset, jan 2023.

[2] Yolov8 by ultralytics, January 2023.

[3] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR*, 2022.

[4] Sebastien Bonopera, Peter Hedman, Jerome Esnault, Siddhant Prakash, Simon Rodriguez, Theo Thonat, Mehdi Benadel, Gaurav Chaurasia, Julien Philip, and George Drettakis. sibr: A system for image based rendering, 2020.

[5] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020.

[6] CarDD. Detection dataset. https://universe.roboflow.com/cardd-diezp/detection-m16cd, jun 2023. visited on 2024-01-03.

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[8] Ultralytics Inc. Ultralytics website.

[9] Glenn Jocher. YOLOv5 by Ultralytics, May 2020.

[10] He Kaiming, Zhang Xiangyu, Ren Shaoqing, and Sun Jian. Deep residual learning for image recognition. Dec 2015.

[11] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023.

[12] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. 2020.

[13] Inc. Roboflow. Roboflow website.

[14] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and BjÃ¶rn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[15] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited, 2016.

[16] Jiaxiang Tang, Jiawei Ren, Hang Zhou, Ziwei Liu, and Gang Zeng. Dreamgaussian: Generative gaussian splatting for efficient 3d content creation, 2023.

[17] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. pixelnerf: Neural radiance fields from one or few images, 2021.