Vehicle Test and Validation, Vehicle Operation

3D Vehicle Model (point cloud) processing in Python environment

Igor Racca

K45DZH

Budapest University of Technology and Economics
Autonomous Vehicles Control Engineering MSc

Budapest University of Technology and Economics
Autonomous Vehicles Control Engineering MSc

INTRODUCTION

PROJECT

This project subject is 3D vehicle model processing in a Python environment. The goal of this work is to process a 3D vehicle model (using point cloud) of damaged vehicles, using a neural network in order to identify how damaged a vehicle is.

TASKS

For this task, a neural network model was provided. It was previously implemented by another student. The improvements made in this work are the following: optimizing hyperparameters, adding classification for damage, adding accuracy for both training, validation, and testing, saving and loading the neural network weights, and adding a prediction. Additionally, this document will serve as documentation for the next students that will work on this project.

NEURAL NETWORK

ENVIRONMENT

We have utilized the desktop graphical user interface (GUI) Anaconda Navigator [1] to launch and manage conda packages. Through the navigator, we utilize the application Jupyter Notebook [2], where we can see the live code and its visualizations. The neural network was implemented in Python 3, using the NumPy library. Additionally, an open-source machine learning library based on the Torch library was used, which is PyTorch [3].

DATA

The input is composed of tensors, which are stored in the Data folder. Each vehicle model is composed of two .mat files, which are called _EES and _KUL. The ESS and KUL data are respectively, the labeled data (a percentage of how the vehicle is damaged) and the point-cloud of the vehicle. The vehicle 3D model (point-cloud) is illustrated in the figure below:

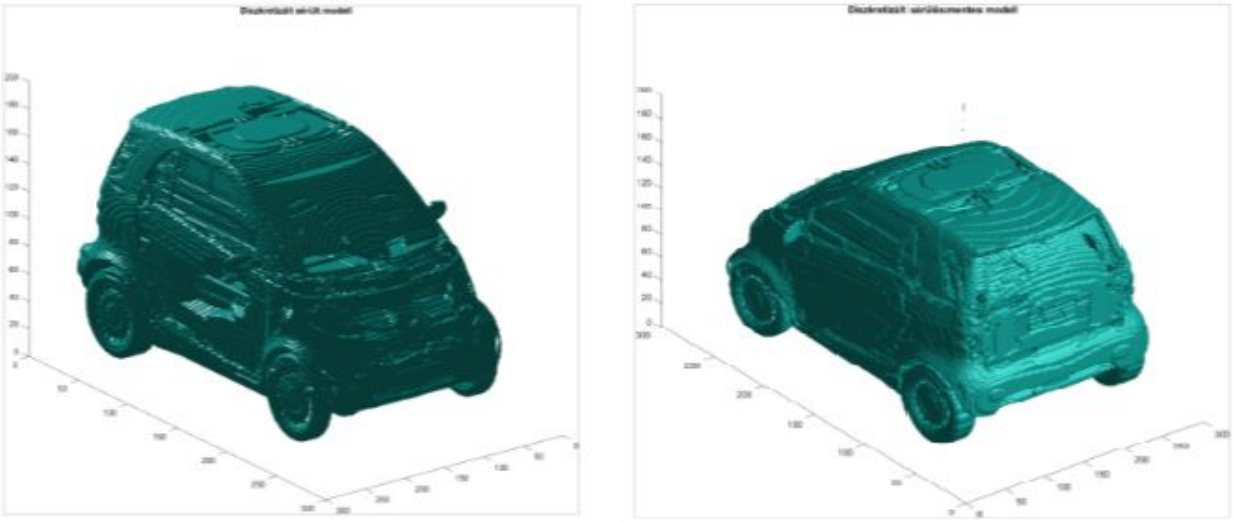

Figure 1.
Vehicle 3D Model

These tensors contain a point cloud of the vehicle model in a 3D space. Each point has its own x, y, and z coordinates, as well as the value regarding the condition of the vehicle at this particular point, which is shown in the image (Figure 2.) and described as in the following table:

-1	Lack of volume
1	Extra volume
0	Normal volume

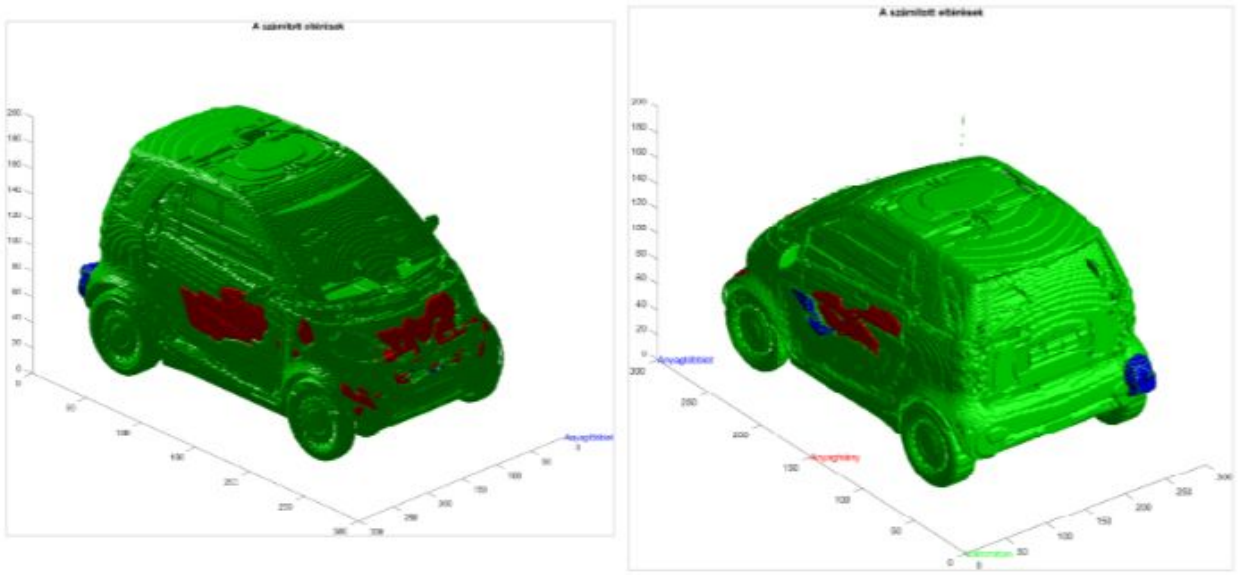

Figure 2.
Vehicle Point Cloud values

SOURCE CODE

For the sake of simplicity and better understanding, the source code file will be explained as it is structured in the IPYNB file, and not the entire code will be illustrated here. The IPYNB extension is a notebook document which can be used in Jupyter Notebook, it offers an interactive computational environment. The source code is explained into the cells, where which section will be described in detail.

In the 1st cell, the required libraries are imported (Figure 3.). Within the 2nd cell, the Cuda device is specified. (Figure 3.)

```
In [1]: import datetime
        import os
        import numpy as np
        import scipy.io as sio
        import torch
        from torch import nn, optim
        from tqdm import tqdm
        from tqdm import trange
        import matplotlib.pyplot as plt

In [2]: device = torch.device("cuda:0")
```

Figure 3.
1st and 2nd cell

The 3rd cell defines the function *read_data*, which reads the input data and split the datasets into training, validation, and testing. (Figure 4.) The first parameter is the base path, where the files are located. The second parameter regards the proportion of the splitting of the dataset (*split_size=[0.6, 0.2, 0.2]*), in this case, 60% of the input data was used for training, 20% for validation, and 20% for testing. In addition, the data is shuffled before the splitting. Regarding the batch size, this Neural Network is in batch mode, which means the batch size is equal to the total dataset.

```
In [3]: #  the split_size gives how much of the data goes to the train/test.
        def read_data(base_path='./Data',split_size=[0.6, 0.2, 0.2]):
            """
            Reads all of the .mat files in the given base_path, and returns a dict with the data found there.
            :param split_size:
            :param base_path: The directory that should be read in.
            :return: a dict, containing the EES and difference tensors.
            """
```

Figure 4.
3rd cell

Within the 4th cell, only the previous function is called. So it can be run independently of the neural network:

```
In [4]: # Load the data and get the splitted dataset
        dataset = read_data()

        100%|██████████| 100/100 [00:13<00:00,  7.59it/s]

        Adatbetöltés kész
```

Figure 5.
4th cell

The classification made in this work is a very simple classification, which might need improvement in the future. Since the neural network output is a float number, that represents the percentage of damage to the vehicle. The function simply divides the total percentage in three and gives a class for each interval (except for the Not Damaged class, which is equal to zero). The classes together with the labels can be seen in the figure. (Figure 6.). This class also contains a method that compares whether two labels are in the same class.

```
In [5]: class Classificator:

            #-------------------------
            # Label   Class
            #-------------------------
            # 0:      Not damaged
            # 1:      Slightly damaged
            # 2:      Damaged
            # 3:      Very damaged
            #-------------------------
```

Figure 5.
5th cell

The model *CarBadnessGuesser* subclasses the nn.Module [3], which is a base class for neural network modules. As mentioned before, the model was previously developed and provided, therefore no changes were made within the model. (Figure 6). The second parameter of the constructor of the neural network is the hyperparameter learning rate, for this work, the value was set to 0.01. The validation frequency is also set in the constructor, as well as the loss function, which is the MSE loss (Mean Square Error).

In [6]:
```
class CarBadnessGuesser(nn.Module):
    def __init__(self, lr=0.01):
        super(CarBadnessGuesser, self).__init__()

        self.valid_freq = 10

        self.model = nn.Sequential(
            nn.Conv3d(in_channels=1, out_channels=3, kernel_size=(10, 5, 5), stride=(10, 5, 5)),
            nn.BatchNorm3d(3),
            nn.Conv3d(in_channels=3, out_channels=2, kernel_size=5),
            nn.BatchNorm3d(2),
            nn.Conv3d(in_channels=2, out_channels=1, kernel_size=3),
            nn.BatchNorm3d(1),
            nn.AdaptiveMaxPool3d((1, 1, 10)),
        )

        self.linear = nn.Sequential(
            nn.Linear(in_features=10, out_features=5),
            nn.ReLU(),
            nn.Linear(in_features=5, out_features=1),
            nn.ReLU()
        )
        if torch.cuda.is_available():
            self.linear.cuda()
            self.model.cuda()

        self.loss_fn = nn.MSELoss()
        self.optimizer = optim.Adam(list(self.model.parameters()) + list(self.linear.parameters()), lr=lr)

        self.classificator = Classificator()
```

Figure 6.
6th cell

Yet regarding the neural network class within the 6th cell, it contains the forward, train, validation, and test functions. The second parameter of the training function is the number of epochs, which were set to 50 for this work. The headers of the functions are illustrated in Figure 7.

Additionally, it also contains the functions: *save_weights* and *load_weights*, which save and load the neural network weights regarding the last training. The load function always loads the last saved weights within the current folder, while the save function overwrites the last saved weights in the current folder, but also saves a copy of the previously saved weights within the training folder, in case past training weights are desired. The training, test, and validation functions work similarly, all of them calculate the loss and the accuracy of the steps. The validation and test functions contain the command *with torch.no_grad():* since it should not compute the gradients.

The input data and it respective label (*data['KUL'].cuda()* and *data['EES'].cuda()*) are read from the respective dataset, the output is returned by the *prediction = self(input_data)*, which calls the neural network forward function. The loss is calculated

(*loss = self.loss_fn(prediction, label)*), and the total loss and correct predictions are updated at every step. In order to check if the prediction is right, the classification is called. In the case of the training, the *loss.backward()* is called in order to backpropagate the gradient along with the neural network. In addition, within the train function, the validation and batch losses are plotted. (Figure 9)

```
class CarBadnessGuesser(nn.Module):
    def __init__(self, lr=0.01):

    def forward(self, x):

    def train(self, epochs=50):

    def validation(self):

    def test(self):

    def save_weights(self, save_dir="./training"):

    def load_weights(self):
```

Figure 7.
Neural Network class

The last cell contains the Main function. As Figure 8. illustrates, it is possible to set the program for either or both training and/or testing. The neural network is instantiated and whether TRAIN is set to *true*, the neural network will be trained and validated (Figure 9.), and after that, the weights will be saved.

```
In [7]: if __name__ == "__main__":

            # set mode
            TRAIN = True
            TEST = True

            # instanciate the NN
            net = CarBadnessGuesser()
            torch.backends.cudnn.enabled = False

            # Train, Validate and Save the model
            if(TRAIN):
                net.train()
                net.save_weights()

            # Load the model and Test
            if(TEST):
                net.load_weights()
                net.test()
```

Figure 8.
Main function

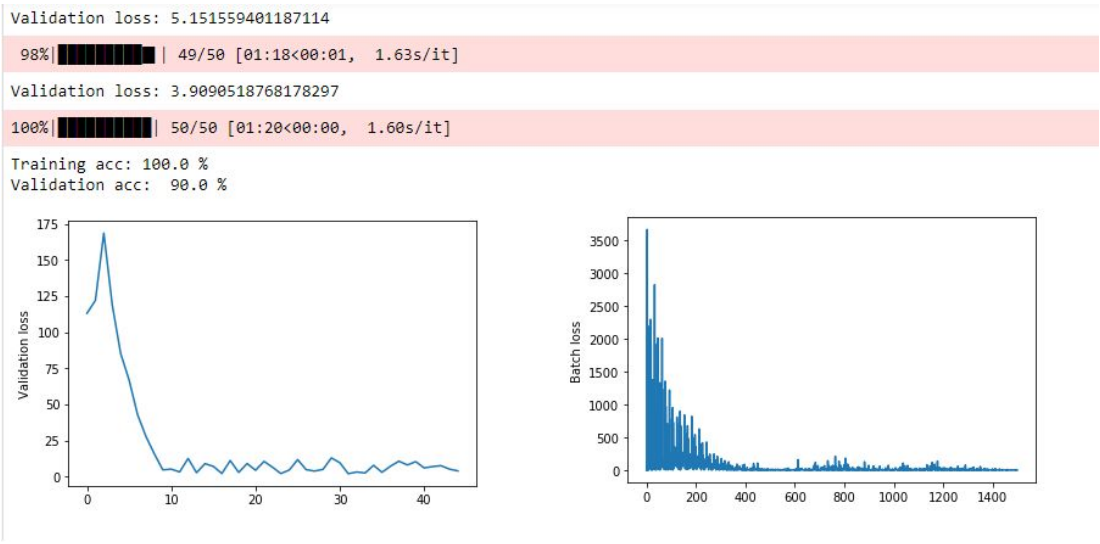

Figure 9.
Training and Validation

When TEST is set to *true*, the neural network will load the weights and after that, start testing (Figure 10.).

```
---------------------------------

Prediction:  1.1521445512771606
Excpected:   2.9321401119232178

Class:           Slightly damaged
Expected Class:  Slightly damaged

loss:  3.168384313583374

---------------------------------

Prediction:  35.56421661376953
Excpected:   41.423095703125

Class:           Damaged
Expected Class:  Damaged

loss:  34.32646560668945

---------------------------------

Prediction:  0.0697123110294342
Excpected:   0.16387200355529785

Class:           Not Damaged
Expected Class:  Not Damaged

loss:  0.008866047486662865


---------------------------------
Test Accuracy:  90.0  %
Average Loss:   34.760701724185814
---------------------------------

Test is completed
```

Figure 10.
Testing

PREDICTION

Another script called *CarBadness_Predict.ipynb* was implemented in order to predict new data. The script is pretty similar. It basically:

- Read the input data from a different folder (*Data_Predict*) (Figure 11.)
- Instantiate the Neural Network (the unused functions, as Training, Validation and Saving are not used)
- Load the saved weights (within the folder *Weights*)
- Give the predictions of the new data

```
In [3]: def read_data(base_path='./Data_Predict'):
            """
            Reads all of the .mat files in the given base_path, and returns a dict with the data found there.
            :param BASE_PATH: the path where the INPUT DATA will be read.
            :return: a dict, containing the EES and difference tensors.
            """
```

Figure 11.
Read_data

Additionally, the data splitting and shuffling were removed since there should be only one dataset in this program.

Regarding the correctness of the Neural Network, 10 new input data were tested and the Accuracy for these data was 40%. (Figure 12.)

```
loss:  683.4257202148438

---------------------------------

Predicted:  0.7324368357658386
Expected:   34.7273063659668

Predicted Class:  Not Damaged
Expected Class :  Damaged

loss:  1155.651123046875


---------------------------------
Prediction Total Accuracy:  40.0  %
Total Average Loss:   961.4565315180355
---------------------------------
```

Figure 12.
Prediction

CONCLUSION

We can conclude that the neural network has a good performance and accuracy using the 50 vehicle models as input data, which are stored in the *Data* folder. Since more data is available (4000 vehicle models within the *Data1* folder), it would probably be enough to confirm how the neural network would perform. More predictions should be made with different data in order to have a better understanding of the neural network.

Unfortunately, using all the data for training is not possible, since the point cloud is a massive representation of the whole vehicle, processing all the points in the Neural Network would not be a viable approach in terms of computation. For this reason, the approach required in the future might be to eliminate all the inner points within the point cloud, thus leaving only the points which represent the “shell” of the vehicle model.

In order to achieve better results. possible improvements or experimentations (some when training with more data is available) that might be tried in this project are the following:

- Check predictions of the NN with more and different data (there are 4000 data available)
- Change the batch size (currently: batch mode)
- Shuffling training set (it currently performs badly)
- Trying a different number of epochs
- Trying a different Learning rate
- Trying mini-batches
- Trying Dropout
- Check whether there is no Overfitting/ Underfitting
- Check more predictions of not damaged vehicles

Since understanding source-code that was written by someone else is not an easy task, the intention of this work is also to provide information and make it easier for the next students that will work on this project.

REFERENCES

1. Anaconda Navigator Documentation [Web]

https://docs.anaconda.com/anaconda/navigator/

2. Jupyter Notebook Documentation [Web]

https://jupyter.readthedocs.io/en/latest/

3. PyTorch Documentation [Web]

https://pytorch.org/docs/stable/index.html