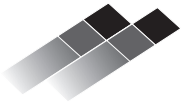

Engineering Practice Report

Deep Motion Planning for Autonomous Driving

August 16, 2021 - November 4, 2021

Student	Pei-Yi Lin
student ID number	03728797
	peiyi.lin@tum.de
Supervisor	Stefan Matthes
	fortiss GmbH
	Guerickestraße 25,
	80805 München

Abstract

In the 3-month internship, I had the chance to take a detailed look how deep learning algorithms can be applied to autonomous driving, especially in the part of motion planning. I started with literature review, and later on reimplemented a repository from a previous research, [4] Learning by Cheating.

I have put my code on GitHub. `https://github.com/b07901064/public_1`

1 Motivation and Goal of the Project

I am interested in the field of machine learning. I've already taken courses in the university. However, I didn't have many chances to put the theory into practice. Therefore, I would like to do an internship that would somehow relate to machine learning. Luckily, I got the chance to work on the [5] Carla autonomous driving project in fortiss.

Machine learning algorithms has many applications, and autonomous driving is one of them. It is also a widely discussed topic in the recent years. I am really found of the topic, and I was excited that I might have the opportunity to learn how deep learning algorithms can be applied to autonomous driving. Originally, my goal was to first read about some of the state-of-the-art approaches, and then implement an algorithm for motion planning.

2 Technical Background and Starting Point for the Work

I am studying electrical and computer engineering in the 5th bachelor semester. Before the internship, I had only two courses concerning machine learning in the university, which were "Machinelle Intelligenz und Gesellschaft" and "Introduction to Machine Learning". I've learnt some theories and traditional machine learning algorithms, e.g. regression, SVM, decision tree, clustering, and I had a little bit of programming exercises. To be honest, I never had a course that focuses on teaching just Python, the programming language. Thus, even though I knew Python, I didn't really have a solid foundation.

In addition, I didn't know much about deep learning. It was only briefly introduced in the university lecture. Neither did I have the experiences with PyTorch nor Tensorflow.

Hence, I started with literature review on state-of-the-art deep learning-based approaches (see References). Most of the paper I read used either imitation learning or reinforcement learning. I was not familiar with neither the two machine learning areas, but I've learnt a lot during the process. After reading, I felt that I'd personally prefer the imitation learning approach better. There was one paper I liked particularly, which is [4] Learning by Cheat-

ing. Therefore, I decided to implement an algorithm based on their approach.

After two weeks of literature review, I spent another week at home looking into the syntax of the framework, PyTorch, before I started to set up the working environment in the office. My college introduced the cloud computing concept to me, i.e. how I can connect to their server and work in my docker container. I also learnt to use conda environment to manage dependencies and isolate projects. At the same time, since I was working on an Ubuntu machine, I got more familiar with the terminal as well.

I tried out two repositories during the internship. The first repository requires user to collect data with the Carla server. Since installing a Carla simulator on the company server is not an easy task, the plan was originally first collect data on the local machine with the installed Carla server, and then upload the collected data to the company server and train the model there. I've install multiple versions of Carla simulator. For each simulator, I had to set up a virtual environment, and specified the correct `$PYTHONPATH` and sometimes custom .egg files. The authors loaded the data in a complicated format which we couldn't really use. Therefore, with the help of my supervisor, I wrote a data loader file to load data in binary format with the module pickle. It is an easy and efficient way to load data. I was able to load the data and see channels of images. However, when I went back after the weekend, it seemed that I could not establish connection between the client and Carla server, even though I didn't change anything. At the end, we gave up on that repository because we had seen too many problems with it. For instance, the provided environment was really out dated, that was why we had to try different versions of Carla simulator and spend quite some time just to set up the environment.

The second repository is from the same author. It's from his another paper about their reinforcement learning approach to the Carla autonomous driving challenge, but he also included their LBC code in the repository. The environment provided in the second repository is more up-to-date and it doesn't contain many other unnecessary packages. They also provide a dataset online, so that I didn't really have to collect data with the Carla server myself. I downloaded the dataset and basically start my real work from there.

3 Solution Concept and Realization of the Project

3.1 Data Loader

My task then was to implement a baseline. At first I had no idea where to start, I stared at their code for days and there were a lot of thing I couldn't understand. I asked a college for some instructions, he suggested that I should start from loading the data.

The provided dataset is in another format than the one the author used. The authors had

the data in lmdb format, whereas the one I downloaded contains many subfolders, and each subfolder includes one folder with the images and a .json file which contains the numerical values and images file path. The whole dataset is hundreds GB large. I had to write a data loader to load the data efficiently, in terms of both memory and speed.

I first had to define what information one item contains. One item could be the whole scenario, i.e. the whole subfolder, or it could also just be the information from one time-frame. We looked into their code, and we guessed they used just the information from one timeframe plus some future ego-vehicle's locations for each item.

I had some futile attempts at first. I was copying the data from one list to another, which is very inefficient. And then I figured out a way to access the data correctly. I then loaded the data from each subfolder as a SingleDataset and then I put these SingleDatasets in a list, and later concatenate all the SingleDataset into one big ConcatDataset. I can adjust how many data I'd want to load, and also how many future ego-vehicle's locations I'd like to include for each item, however I like. Most of the time I only wanted to debug and see if the code works, so I could just load a small dataset, and it won't take too long for me to test. If I wanted to train the model seriously, I could also load more data.

After I finished loading the data. I could finally start building the network.

3.2 BEV Model

The birds-eye-view model (BEV model), is the privileged model. It has the BEV map as input, and it outputs the predicted locations for some future time frames (it can also be specified). The authors used ResNet-18 as backbone, added a speed encoder, and fed the output from the ResNet and speed encoder to a 3-layer convolutional network, and then they applied a Softmax layer to produce a spatial probability distribution, so basically a heat map for predicted locations.

The BEV model is relatively easy to train. I could use batch size of 256 on one GPU. And the model learns fast. The loss decreases drastically in the first episode, and continues to decrease in the later episodes. The output trajectory speaks to the ground truth trajectory after one episode, and it should also be a drivable trajectory.

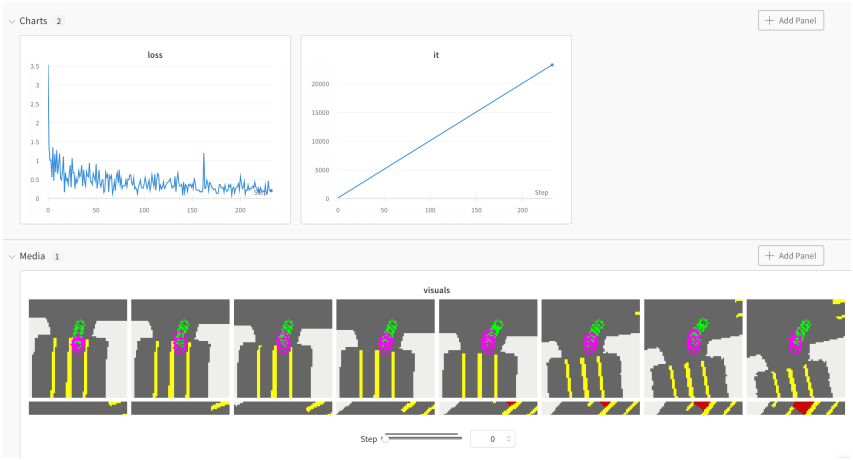

Figure 1: (green: ground truth, pink: predicted trajectory.)The model did not know how to drive at first and stayed where it was.

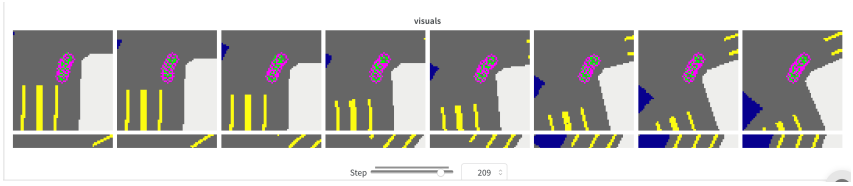

Figure 2: (green: ground truth, pink: predicted trajectory.) After some training, the predicted trajectory is very close to the ground truth trajectory.

3.3 RGB Model

The camera-view model (RGB model). Takes extra two inputs, i.e. the RGB images from the frontal camera plus the segmented images. This model uses the output trajectory from the BEV model as the ground truth (tgt_bev_locs).

The rgb_model network than takes only the RGB images and speeds (rgbs, spds) as input, and outputs the predicted locations on the RGB images (pred_rgb_locs), and also the predicted segmentations (pred_sems). The predicted locations on the RGB images is then mapped to the birds-eye-view image (pred_bev_locs). We can calculate the loss between the predicted location and the ground truth; this loss is called act_loss. In this case, the L1-loss is used. And act_loss has 6 columns, where each column corresponds to a command. The location loss (loc_loss) is calculated from the act_loss, dependent on if the command is to follow a lane or to turn. Meanwhile, the model also tries to learn the segmentation. Namely, how to perceive and understand the RGB images from the frontal cameras.

The segmentation did not work at first. The model should label the RGB images. However, I did not provide the right ground truth segmentation for it to learn. Since it is all about labelling the images, the ground truth segmentation should be just labels, and has the size of (height, width, num_labels). I didn't know about that at first. From what I saw in the dataset, the segmented images are only .png files. I was not aware that my dataset is different from the one the authors used. If I read it as normal images, it would then have the wrong dimension in the 3rd order, so I just read it as gray-scale images. This way the ground truth values are just zeros. As a result, when I visualize the training process, the ground truth segmentation was all black, and the model learnt the segmentation to be all black, too. Nonetheless, even without the segmentation, the RGB model can still learn pretty well.

The RGB model is more difficult to train. It takes more inputs, uses a bigger model, and it also requires the previously learnt BEV model. Thus, it also takes more GPU memory. I could only use batch size of 56 in this case. Ideally, I'd like to use a batch size of 128 or above. In the beginning, I didn't pass the learnt BEV model, and the output did not make much sense. Both ground truth locations and predicted locations looked like random points and the trajectories were definitely not drivable.

After I passed the BEV model, the trajectoroes started to make some sense. I could see the reasonable ground truth trajectory from the very beginning. The predicted trajectory, on the other hand, took a while to make sense. It first learnt how fast it should drive, and stopped when it needed to stop. For instance, it should stop at red light. However, I could not see the model turning before I went to bed. To my surprise, when I woke up the next morning and check again, I was thrilled to see the model slowly learning how to turn left and right.

There is one scenario that were captured by the logger more than once. From the two figures, we can observe that the model didn't know how to turn at first, and in the later episode, it learnt to turn better, and produced a predicted trajectory closer to the ground truth.

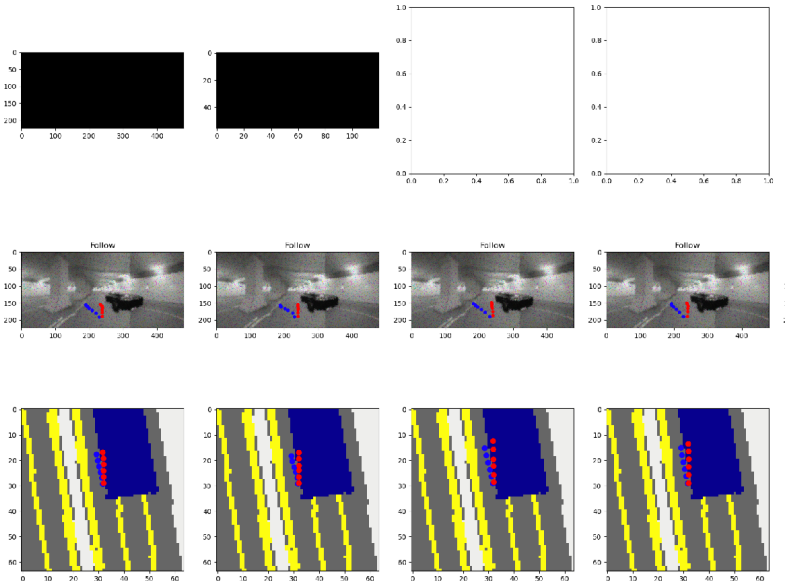

Figure 3: (blue: ground truth, red: predicted trajectory.) The model only knew to drive straight.

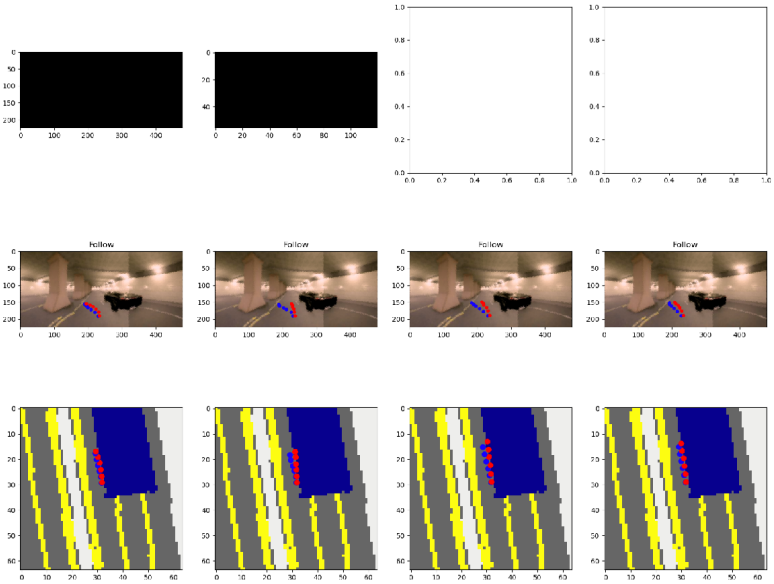

Figure 4: (blue: ground truth, red: predicted trajectory.) Later on, the model has learnt how to turn better.

Later on, I wanted to get my model to learn the segmentation as well. I fixed my dataloader. This way, I can first map the RGB values to labels and have the right segmentation as the ground truth. The cross entropy loss will then be computed between the ground truth and the predicted segmentation. The cross entropy loss function provided in PyTorch (torch.nn.functional.cross_entropy) only takes the data of type “long” as input. That was a point we wanted to optimize. Since we were tight on GPU memory, we tried to write a custom function to calculate the cross entropy. After all, the cross entropy formula is not that complicated, and the segmentations are just labels that could be stored in a byte. However, we had to call torch.gather(), and it only take the data of type “long” as input. Unfortunately, it turned out that it was not something we could optimize on.

Up to now, I have got the model to learn how to recognize the road, but it has not yet learnt how to recognize other objects.

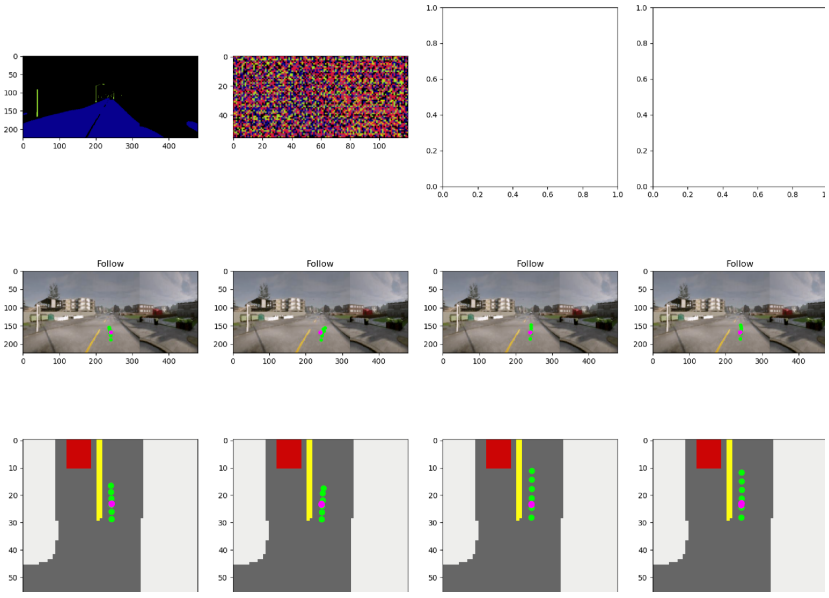

Figure 5: (green: ground truth, pink: predicted trajectory.) The model has not yet learn how to drive, nor how to perceive the images.

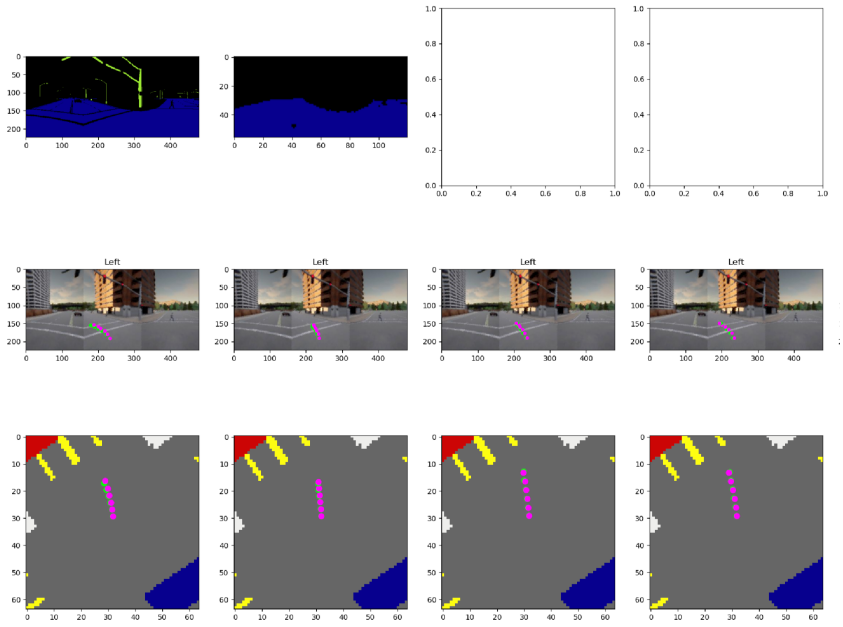

Figure 6: (green: ground truth, pink: predicted trajectory.) Later on, the model has learnt how to do the road segmentation, and can drive well.

4 Results Evaluation and Comparison with Goals

The author also provided their output for our reference. I think I got almost everything right. The model I trained outputs a reasonable trajectory, and it also learnt to how to understand perceive the frontal camera images. Maybe it is a small pity that it has not learnt how to recognize the road lane, compared with the results the authors provided.

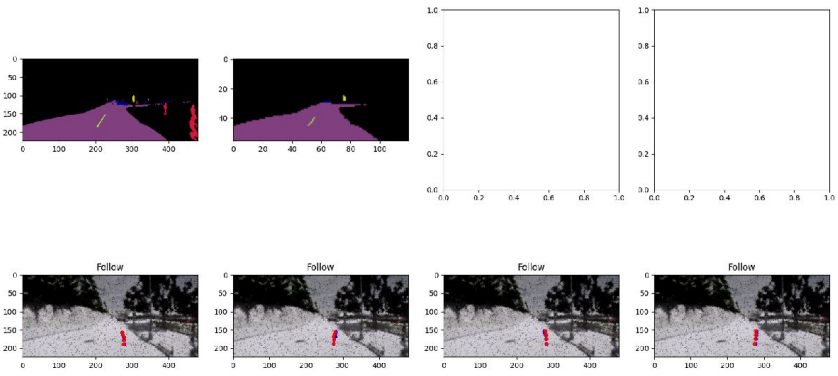

Figure 7: reference output from the authors

I did not really follow my initial plan in this internship, which was to implement a new model for motion planning. At the beginning, we were talking about only using the birds-eye-view map (not the frontal camera images) as the input, and implement a new idea. That was why I chose to use the privileged model from the LBC paper as the baseline. However, I was not really familiar with Python programming and deep learning practices. I had a lot to learn by just implementing a simple baseline. Even though I spent a lot of time discussing the theoretical part of the new idea with my supervisor, I still don't know where to start if I'd like to implement it in a program. Hence, I chose to implement the sensorimotor and see how it performs.

Now I have successfully trained both the privileged agent and the sensorimotor agent. The next step could be letting the agent run on the Carla simulator and see how it performs. However, at the moment I am more interested in how to optimize the training speed. I have observed a fluctuation in the time that my training episodes need. The first episode could take almost 12 hours, and I was using only one fourth of the whole given dataset. On the contrary, the later episodes sometimes only need 4 hours. Even if we exclude the first episode, since it is typical that the first one needs longer, I still see this fluctuation in the training time. Some episodes needed 4 to 5 hours, whereas the other ones needed 9-10 hours.

5 Summary

I have learnt a lot during this internship. Although I couldn't make it to my supervisor's expectation, I have already exceeded my own expectation. In the beginning, I didn't think I could get anything working. But in the end, I got two models working.

During this journey, I learnt a lot of the essential skills an engineer needs, which could not be taught in an university. I started to read papers just a few months ago because of this internship. I learnt how to work in a docker container on server. I got more familiar with terminal commands. I started to use git because of this internship. I learnt how to use the debugger and how to manage virtual environments. I can program in python better and understand the code from other people faster now, compared with myself just two months ago. I gained real practices with deep learning and the framework PyTorch. I had a look into the research field, which was something I was curious about.

Huge thanks to my supervisor, Mr Matthes. No matter how many trivial questiones I asked, he was always there and replied patiently.

References

[1] Tanmay Agarwal, Hitesh Arora, and Jeff G. Schneider. Affordance-based reinforcement learning for urban driving. *ArXiv*, abs/2101.05970, 2021.

[2] Mayank Bansal, Alex Krizhevsky, and Abhijit S. Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *ArXiv*, abs/1812.03079, 2019.

[3] Di Chen, Vladlen Koltun, and Philipp Krähenbühl. Learning to drive from a world on rails. *ArXiv*, abs/2105.00636, 2021.

[4] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. In *Conference on Robot Learning (CoRL)*, 2019.

[5] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. Carla: An open urban driving simulator. *ArXiv*, abs/1711.03938, 2017.

[6] Eshed Ohn-Bar, Aditya Prakash, Aseem Behl, Kashyap Chitta, and Andreas Geiger. Learning situational driving. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11293–11302, 2020.

[7] Aditya Prakash, Aseem Behl, Eshed Ohn-Bar, Kashyap Chitta, and Andreas Geiger. Exploring data aggregation in policy learning for vision-based urban autonomous driving. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11760–11770, 2020.

[8] Nicholas Rhinehart, Rowan McAllister, and Sergey Levine. Deep imitative models for flexible inference, planning, and control. *ArXiv*, abs/1810.06544, 2020.

[9] Marin Toromanoff, Émilie Wirbel, and Fabien Moutarde. End-to-end model-free reinforcement learning for urban driving using implicit affordances. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7151–7160, 2020.

[10] Wenyuan Zeng, Wenjie Luo, Simon Suo, Abbas Sadat, Binh Yang, Sergio Casas, and Raquel Urtasun. End-to-end interpretable neural motion planner. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8652–8661, 2019.