


JOURNEY FROM ACADEMIC PAPER TO INDUSTRY USAGE

WHAT IMPLEMENTING CONDITIONAL IMITATION LEARNING TAUGHT US ABOUT REPRODUCING A PAPER

by Markus Hinsche

MOTIVATION: REPRODUCIBILITY OF PUBLICATIONS

If you are a researcher in computer science, you know the following situation: You find a paper on arXiv, you read it, and you like it a lot . In fact, you want to try and run the algorithm yourself, so you look if the authors also happen to publish the data, the code and so on. Reproducibility is the first step to improve upon the results of an existing paper. Throughout academia as well as industry, there are several reasons why it is necessary to reproduce results:

- Reproduce the paper as a baseline for comparison.
- Verify your understanding of the paper.
- Use the research method as a submodule in your own stack (e.g. an OCR model in a document analyzer, or a lane detection model in an autonomous driving system).
- Check how the algorithm extends to different datasets (often a proprietary dataset you have access to).
- Extend the model for improved results (e.g. trying a different loss function, or tweaking the network architecture).

At Merantix, we push state of the art of research into industrial application. Therefore, we are highly interested in the latest results from research. From publications it is hard to tell how well approaches generalize to other datasets or different applications, and thus re-implementation is required. After reproducing the published results, the presented methods can be adapted to new applications. Although our engineers are experts in the field of deep learning, we often cannot fully reproduce the results of a paper. This can be due to various reasons, e.g., because of missing data. Other reasons will be mentioned in the remainder of this article. Despite difficulties when reproducing papers, we come up with solutions.

No matter why you need to re-implement a paper, due to their complexity there are often small implementation details that are missing a mention in the paper: this may either be some of the code, the data, a hyperparameter, etc. These details might be missing for several reasons: The authors usually have only limited space in the conference or journal version for their paper. Furthermore, authors might not know what is obvious and what should be stated. In some occasions, some researchers omit details that would make the paper less strong or hide details so they can publish them in follow-up publications.

Researchers have varying tolerance for reproducibility issues. Some enjoy reverse-engineering missing implementation details, either because of the detective work involved or because they seek to increase their understanding of the method. Others go as far as accusing the field of a full-fledged “**reproducibility crisis**”. Regardless of your disposition, we’d like to share some tips for reproducing results more easily and quickly.

In the following, we will present our journey from an academic paper to its industry usage. You will get to know at a simple approach for a fully autonomous driving system: The paper “End-to-End Conditional Imitation Learning” (abbreviated CIL) [1] provides a solid example for a simple autonomous driving model. The paper proposes a “rich” network architecture which involves a CNN with a multi-task learning setup.

By reading this blog article you will hopefully learn some tricks to get better at reproducing other people’s research.

Autonomous driving robustness

At **Merantix**, we work on improving the robustness of autonomous driving systems. In order to do so, we have built a scenario-based testing platform: Given a set of carefully curated, reproducible scenarios, the performance of different autonomous vehicle systems is analyzed. As CIL provides one of the most straight-forward models for autonomous vehicle navigation through end-to-end behavior cloning, it serves as a good baseline to develop our testing platform against.

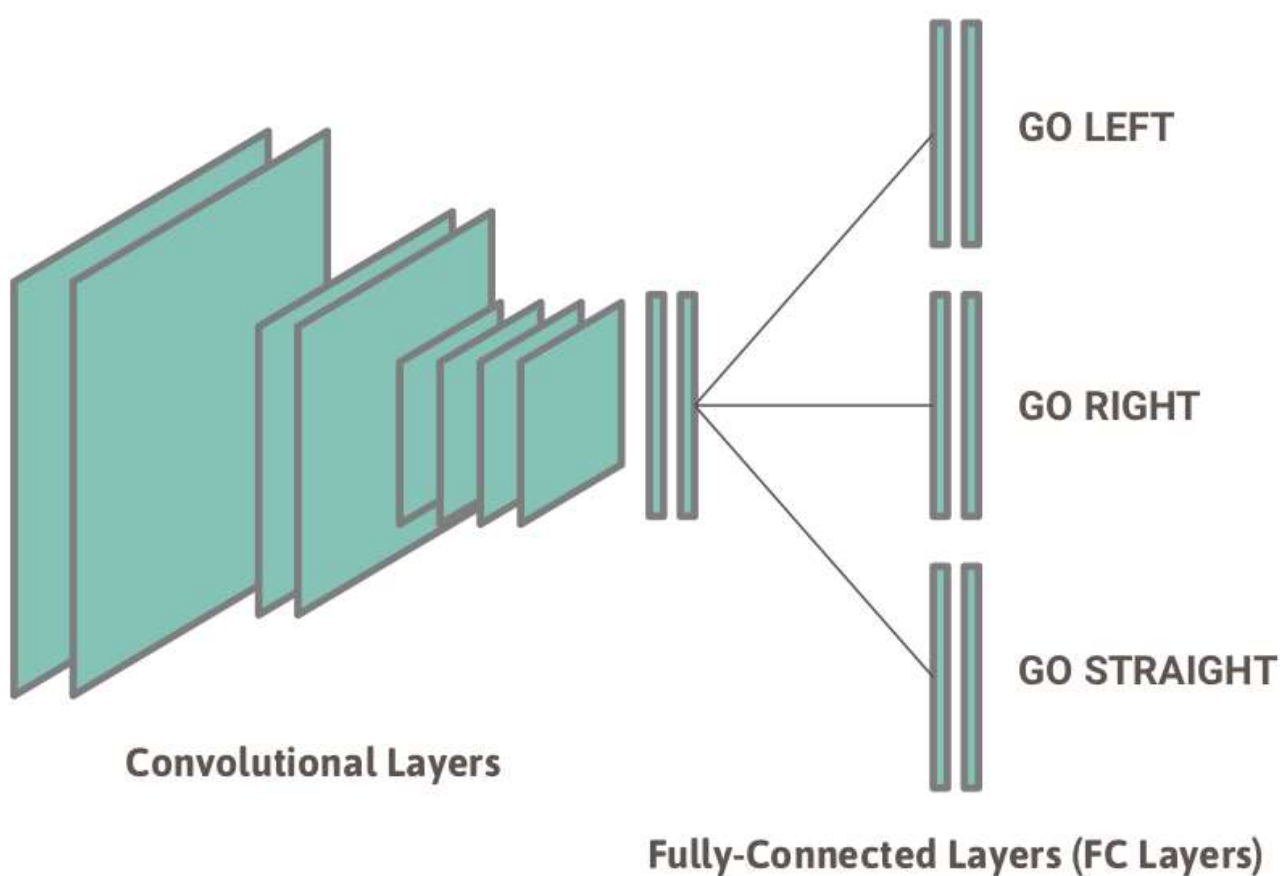
End-to-end Conditional imitation learning

How is an autonomous driving system designed? Many autonomous driving stacks are modular. In fact, these designs work best as of now. Explicit decomposition of autonomous driving into modules has one disadvantage though: each module such as lane marking detection, path planning, and control has to be designed separately. This is quite labour-intensive. In order to generate a simpler model, end-to-end learning approaches have been proposed [1] [2], which optimize all processing steps simultaneously. One of them is *End-to-end Conditional imitation learning (CIL)*.

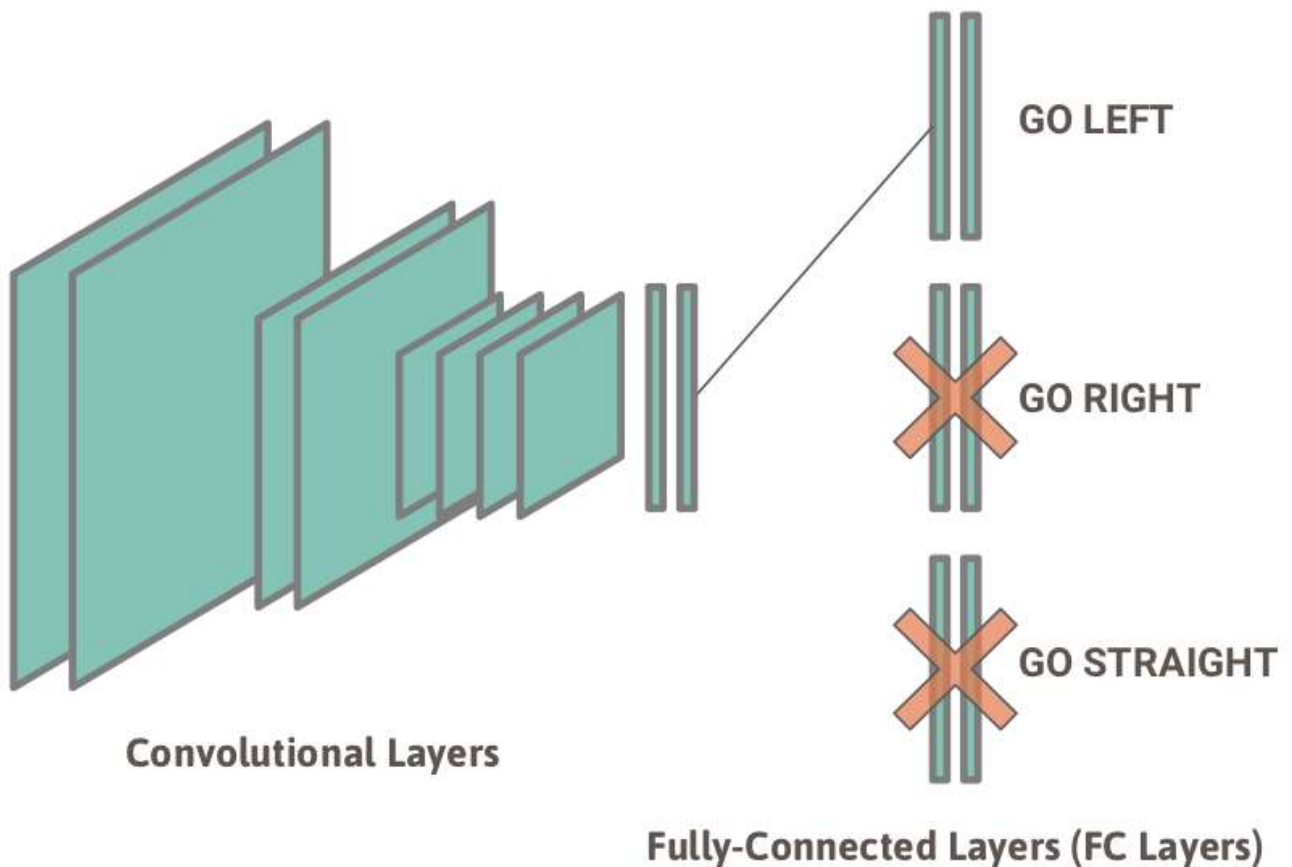
Imitation learning tries to mimic the behavior of an expert (in this case a human driver). It aims to directly generalize the expert strategy observed in the demonstrations to unvisited states, which is achieved by regression, e.g., steering angle regression. The autonomous driving system can’t know everything from demonstration only. There are a few problems that need to be tackled in imitation learning.

Firstly, imitation learning takes only expert data to train. There are some problems associated with this approach. If for example the trained agent slides off the street for some reason, it will have difficulties to get back onto the street because the agent has seen only very few situations in the expert training data where the expert covered such behavior. Due to this fact, the distributions of the training data and the target domain are distinct. This phenomenon is referred to as covariate shift, and it is especially troublesome for autonomous driving.

Furthermore, in contrast to simple teach-and-repeat approaches, the model should learn an underlying navigation policy. Instead of solely repeating what it saw, navigation commands (e.g., where to turn left or right) should be provided as context that the model takes into account. Assume, your car is driving you to the local cinema. This involves going straight for 1km, then turning right, and then left. The model needs to take these navigational commands (straight for 1km, right, left) into account to make the correct turns. The conditional imitation learning paper [1] achieves exactly that. In addition to having an end-to-end architecture, it uses conditional branches (sometimes also called network heads) selected according to high-level navigational commands.



End-to-end Conditional imitation learning is a simple way to learn an autonomous driving model, which features a rich network architecture: The main part of the network is mostly convolutional. It is connected to one of multiple different branches.



The network has multiple branches, but only one is active: The image shows an example where the high-level command indicates to go left, therefore only the GO LEFT branch is activated.

Let's go a bit more into the details of how this works: CIL learns multiple tasks at once—steering, throttle, and brake. Due to the continuous nature of these outputs, it solves multiple regression tasks jointly.

In our implementation we used the original dataset that the authors provided with the paper. It contains driving sequences recorded by human drivers in CARLA, an open source driving simulator. The dataset consists of multiple entries, each of which contains an observation, a high-level command, and an action:

- The observations are the perceptual input (RGB images) from a windshield camera.
- The high-level commands are one of the following: "follow the lane," "drive straight at the next intersection," "turn left at the next intersection," and "turn right at the next intersection."
- The actions are the regression targets (steer, throttle, brake).

Given an observation and control commands, the dataset is used to train a deep network to predict the expert's action.

In our case, we are dealing with code tailored to do inference, but we want to repurpose it to do both training and inference. In the course of training the CIL stack, we made several discoveries debugging in this particular setting. While some of these discoveries might be tailored toward the CIL stack specifically, other insights can be applied to many other papers.

ENABLE BATCH NORMALIZATION

The first thing to do when adapting inference code to training code is to add a loss term and an optimizer to optimize the loss. The paper describes a mean squared error loss for the regression tasks and the Adam optimizer to minimize the loss. So far so good! We trained our model and the loss curve shows the loss decreasing as expected. Could it be this easy? Will it work in the first attempt? Excited by the loss curve we deployed our model to simulation. Disappointingly, we realized that the car barely moved.

Remember that the authors only released the inference code and we want to reuse the paper's code for training. The network definition has to account for that in a couple of places. Let's take dropout as one example: dropout usually gets enabled during training (as a regularization method to increase network robustness), but disabled during inference.

The first thing we looked at was the code for the network to see if anything was not right. And there it was, staring us in the face: `batch_norm(input, is_training=False)`. Just like with dropout, we have to enable batch normalization for a network designed to be trained with batch normalization during training. This is mentioned in the paper, but since the paper's code was only doing inference it was—for obvious reasons—disabled during inference.

After this change the trained model started moving the car. Hooray! 🙌

SPEED-UP THE TRAINING CYCLE

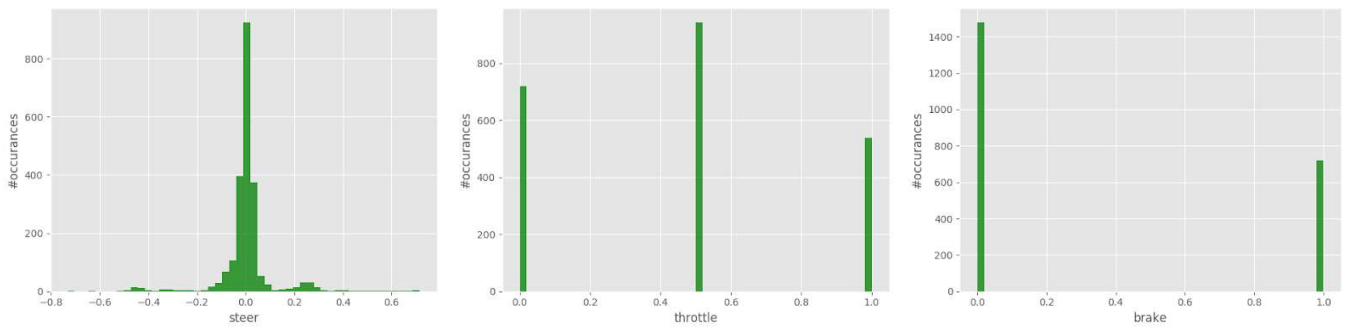
Now that we enabled batch normalization, the vehicle moves. But does it move into the right direction? Will it pay attention to road signs? Will it behave like a human driver at all? Let's step back for a moment and look at the evaluation process:

1. We will make an adjustment to our training procedure, e.g. enable dropout.
2. To see how well this adjustment works, we fully train the model and then observe how it behaves in simulation. The entire training takes 27 hours on a Tesla P100 GPU.
3. We will run the trained model in simulation and see how it deals with different situations.

Due to the long training, we can only try out one single adjustment per day. There are many more adjustments we will need to make. After each adjustment, we will have to evaluate if this improves our model.

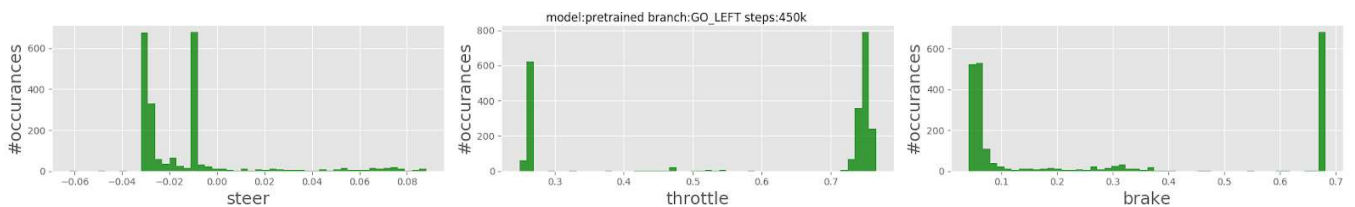
In order to debug the training faster, we looked for ways to know if training works early on. One way is to look at the predictions that our network makes for steer, throttle, and brake; and compare it to the values in the input data.

First, let's look at the distribution of the input data:



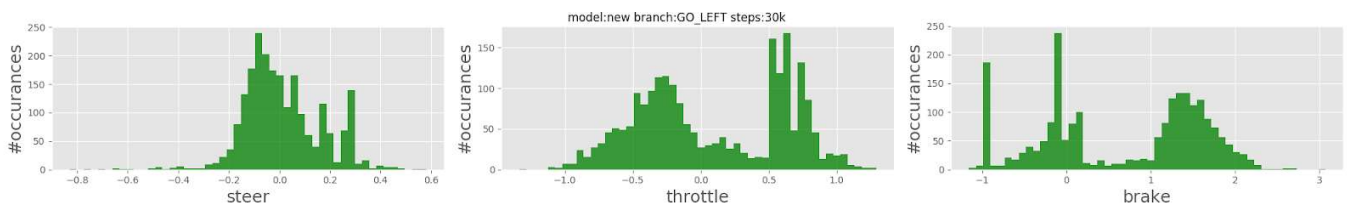
The paper is accompanied by this input data. The distribution of this training data shown as a Histogram for each of the regression targets. Left: steer, Middle: throttle, Right: brake.

We then looked at the output distribution of the pretrained network that was submitted with the paper: This distribution is relatively close (except for the missing throttle peak at 0.5) to the distribution of the input data. This is what you would expect, since the network should learn to output values following a similar distribution as the training data.



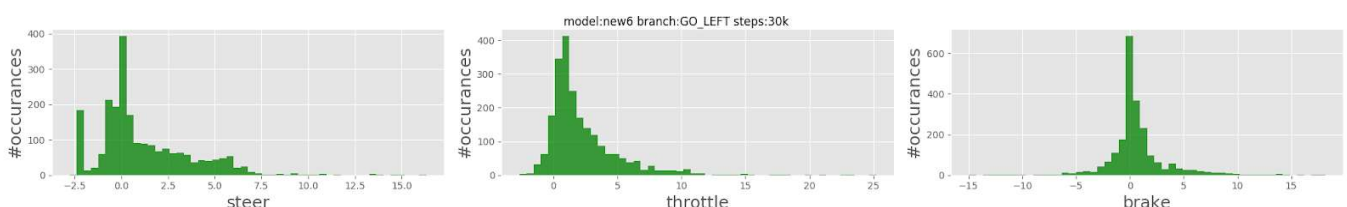
A trained baseline model was published together with the paper. We ran this model and looked at its output distribution. Left: steer, Middle: throttle, Right: brake.

We found that looking at the output distributions of the model after only a fraction of the total number of training steps needed for a fully trained model gave a good indication of how it would behave eventually. For example, we observed the following distribution during an early checkpoint:



We successfully trained our model for 30,000 training steps. This shows the histogram of targets: Left: steer, Middle: throttle, Right: brake.

However, when there were bugs, e.g. in choosing wrong hyperparameters (as is the case in the following image), we noticed that the absolute values for some outputs got quite big which is an indication for a failed training.



We unsuccessfully trained our model for 30,000 training steps. This shows the Histogram of regression targets: Left: steer, Middle: throttle, Right: brake. Training is going to fail because of we are already seeing large absolute steer values. (The steer values should mostly fall between -1 and 1 as seen in the input data distribution.)

In general, it is helpful to find a good experimental setup where one round of training the entire network (with a rough indication of expected performance) can be done as fast as possible. If the network starts to exhibit strange behavior early in the training that is an indication for a bug. For our training, it is therefore possible to gain insights by looking at the network's outputs after training for only 2–3 hours on a Tesla P100 (instead of 27 hours for the whole training). This accelerates the development when debugging training.

Further evaluation of a trained model:

Until now, we had only looked at numbers, more specifically the loss and the output distribution. After the entire training finished, we also had a look at qualitative results (not just quantitative numbers): the car's driving behavior. However, to recognize small improvements we need a more structured way of extensively evaluating the vehicle's behavior. Luckily, CARLA provides a **benchmark** to achieve exactly that. The benchmark puts the car into different types of situations (drive straight, driving with turns, driving with other agents) in different environments. This way, you can quantify the model's performance in different environments such as the town used for training and an unseen town as well as various weather conditions.

Evaluating the behavior of autonomous driving software in a highly structured way is one of the most important things to get right in order to deploy safely and iterate quickly. This is why at Merantix, we are working with leading manufacturers to develop a cutting edge scenario-based testing platform, which delivers these structured insights for developers.

When testing complex autonomous driving software, it is increasingly difficult to measure safety relevant progress, reliably compare models to track improvements, keep the testing system operationally viable, and allocate resources effectively. We implement end-to-end offline testing of self-driving stacks on a catalogue of thousands of very short driving scenarios which are based on simulation or recorded as sensors logs by autonomous test vehicles. Using this methodology we maximize the value of autonomous driving log data and dramatically accelerate the speed and efficiency of development, resulting in significant competitive advantages.

TUNING HYPERPARAMETERS

We are working in a multi-task learning setting which in this case means that we have three different outputs: throttle, steer, and brake. Under these conditions it is common to have a weight for each output. The combined loss function is given by:

$$l_{total} = \lambda_{throttle} l_{throttle} + \lambda_{steer} l_{steer} + \lambda_{brake} l_{brake}$$

The choice of weight hyperparameters were not mentioned in the paper. However, the author was responsive and via GitHub issues and email, he informed us what hyperparameter values were chosen.

Unfortunately, with these hyperparameters the vehicle slipped off the street. This is probably due to a detail that is different in the implementation. It seems the car slides off the street because in our implementation, it did not learn to steer well. We suspect that steer is set too low.



Prior to the tuning of hyperparameters, we did not adjust the weight for steering, therefore the vehicle is drifting off the street.

To overcome the issue, we had to tune the weight parameters ourselves. We had to account for the different scale of parameters regarding absolute values. For example, while the average speed value is 17 (km/h), the throttle values in the given data lie between 0.0 to 1.0.

There is a lot of literature on how to tune hyperparameters in multi-task settings [3][4].

Several valid approaches exist, especially when your loss is a good proxy for the task you are trying to perform. For CIL, the regression loss is somewhat removed from our goal—driving safely and comfortably. On the other hand, we have the advantage of being able to inspect our results qualitatively: we can observe *how* the vehicle drives.

Example: Initially the vehicle slipped off the street. To overcome this limitation, we increased the weight of steering *steer* in order to see if the car would stay on the road.

REACH OUT TO THE AUTHORS

Most researchers are excited to see their papers cited and extended, and their open source code adopted and improved upon. If you're really stuck on an implementation detail, don't rule out contacting the author via email, GitHub, twitter, etc. In our case, the main author was already active on GitHub, and very responsive and helpful via email.

After we had tuned throttle, steer, and brake, we noticed the vehicle stops on the street from time to time. Once the vehicle stopped, it does not move anymore. We expected the throttle output to have picked this up.

That is when we emailed the authors of the paper! Et voilà, we learned that they had similar findings. They offered a solution that works well: An additional output is trained to give the vehicle a little push (throttle at low velocities). In order to not push into other people or vehicles, the amount of push depends on a new output—predicted speed. This is a workaround since the throttle output should also learn that the vehicle can accelerate at low velocities.

In retrospect, there were some indications of this solution in the inference code. When seeing the speed branch our first reaction was: Let's get rid of it. It makes the network's architecture conceptually simpler and it removes the additional speed hyperparameter. However, when deployed in simulation, the car stops from time to time as only frame-by-frame information is available. Despite the speed branch being a workaround, it works well since it makes the car overcome the stopping problem. Sometimes the car still slows down a little, but thanks to the workaround, it recovers.

KNOW YOUR DATA

We noticed another strange issue: our trained model ran over red traffic lights. Apparently for L3, L4 and L5 autonomous driving, running over red lights is not ok 😬. We started brainstorming about all kinds of potential issues: the detection might not work well because it is an end-to-end stack, the resolution of image crop might be too small, maybe the camera is not high enough so it does not see the traffic lights. Having thought about the problem for some time, we realized the solution is way more simple: After looking at the actual training data we were then relieved to find that in the training data the vehicle was ignoring traffic lights as well! So the training had no chance to learn the behavior of stopping at red lights because it wasn't present in the data. Lesson learned: Know your data!

SENSITIVITY TO INITIALIZATION

One circumstance that made debugging this setup hard was the following: when training multiple times with minor modifications, the resulting driving abilities of our stack varied a lot. We found that the network has a huge sensitivity to initialization. With this in mind, it is helpful to look at the output distributions of an intermediate model checkpoint same as we did to speed-up the training cycle. If the model performed poorly early on, we would terminate training and then reinitialize with the same hyperparameters.

CONCLUSION

"End-to-end Driving via Conditional Imitation Learning" is a great paper with an interesting solution to a difficult problem. When implementing difficult and complicated methods, there are bound to be blind spots that one has to work through.

Going back to the initial issue: It is challenging to reproduce results of a complicated deep learning pipeline. According to our experience, there are some general measures you can take to mitigate this challenge:

- Know your data: You can do this by plotting the distributions of the training dataset. You also want to look at some samples (in the case of CIL, training sequences).
- Speed-up the training cycle: In order to iterate quickly, figure out how you can reduce the training time. Are there any conclusions you can draw before the training has completely finished? Is your training setup easy to work with? For example, how easy is it to trigger a new training? Ideally, triggering a training takes one command line call from your development machine.
- Pay attention to the details in the paper. If you can't find something in the paper or the related work cited, try to contact the authors (e.g., via GitHub issues, or email).
- You might need workarounds to adjust the paper to your own use case. The paper is the result of an academic setup, so many things are not designed for a plug-and-play use in industry ([see our related post](#)).
- We discussed many measures very specific to the CIL paper: We enable batch normalization, we tune hyperparameters, and the sensitivity to initialization. For your distinct paper some challenges will most likely be of different nature.

Taking these measures enabled us to get to a working implementation that we can now use in our product to test autonomous vehicles.



For this article, we purposefully selected details of the paper we had to look into to make the system work. Most things in the paper were described soundly and worked very well. We implemented everything exactly as reported in the paper, and by doing so, we even surpassed the benchmark results of the paper (which was the baseline that we tried to match).

Lastly, making the source code of a paper open source helps. This is why we are very open to suggestions on how to improve our implementation as well. If you would like to contribute an improvement or anything else, even better! Head over to our [GitHub repository](#) for more. We released the imitation-learning code under the MIT license to make it as open as possible for anyone to use.

Acknowledgements

Thanks to:

- [John](#) for writing the initial prototype for the training code
- [Robert](#), [Filippo](#), [Clemens](#), [Mark](#), and [Rasmus](#) for the valuable feedback to the code and this article
- the [CARLA](#) team for providing a practical environment to prototype autonomous driving

References

1. Codevilla, F., Müller, M., López, A., Koltun, V. and Dosovitskiy, A., 2018, May. End-to-end driving via conditional imitation learning. In *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE.
2. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J. and Zhang, X., 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.
3. Cabi, S., Colmenarejo, S.G., Hoffman, M.W., Denil, M., Wang, Z. and De Freitas, N., 2017. The intentional unintentional agent: Learning to solve many continuous control tasks simultaneously. *arXiv preprint arXiv:1707.03300*.
4. Cheng, M.Y., Gupta, A., Ong, Y.S. and Ni, Z.W., 2017. Coevolutionary multitasking for concurrent global optimization: With case studies in complex engineering design. *Engineering Applications of Artificial Intelligence*, 64, pp.13–24.

