Continuous Integration of Neural Networks in Autonomous Systems*.

Bruno Steffen[1 [0000−0003−1769−1824]], Jonas Zohren[1 [0000−0001−9392−7582]], Utku Pazarci[1 [0000−0001−7595−2901]], Fiona Kullmann[1 [0000−0001−5858−0659]], and Hendrik Weißenfels[1 [0000−0001−7001−1037]]

Technische Universität Dortmund, 44149 Dortmund, Germany
http://www.tu-dortmund.de/

Abstract. The perception of the autonomous driving software of the FS223, a low-level sensor fusion of Lidar and Camera data requires the use of a neural network for image classification. To keep the neural network up to date with updates in the training data, we introduce a Continuous Integration (CI) pipeline to re-train the network. The network is then automatically validated and integrated into the code base of the autonomous system. The introduction of proper CI methods in these high-speed embedded software applications is an application of state-of-the-art MLOps techniques that aim to provide rapid generation of production-ready models. It further serves the purpose of professionalizing the otherwise script-based software production, which is re-done almost completely every year as the teams change from one year to the next.

Keywords: ML Ops · Continuous Integration · Neural Networks.

1 Motivation and Background

Since 1981 SAE international[2] hosts the Formula SAE, a student design competition where teams around the world design and manufacture formula-style racing cars. The Formula SAE requires that major design decisions and implementations must not be made by professionals, but rather by students. This paper is written by members of the German team of TU Dortmund University, GET racing[3], who work on the autonomous driving capabilities of their vehicle as the competitions have started featuring a driverless format.

* This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-49252-5_21. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

[2] SAE international is a standards developing organization for engineers, see: https://www.sae.org/

[3] GET racing participates annually in the events since 2005, see https://www.get-racing.de/

In this paper, we discuss a solution within the software stack that is developed to enable the latest vehicle manufactured by GET racing (the FS223) to autonomously participate in the racing events on Formula SAE-compliant racing tracks. The software is designed to be safe and reliable but is also complex and requires a diverse set of software components. These components range from the perception of the environment to the processing of the recorded data and finally to the transmission of commands to the underlying actuators in the vehicle for steering and acceleration.

The presented solution is part of the perception component, where the challenge lies in the recognition of track-specific features. These are colored cones marking the edge of the track. To solve this issue, camera images are classified by a neural network as portraying blue, yellow, or orange cones, or alternatively, no cones at all. However, while the actual task of image classification is not particularly unique, the circumstances under which this network has to be developed, trained, and maintained are unusual.

On the one hand, GET racing as a student team does not have a lot of capital to invest in computational power that is crucial for the training of complex neural networks. As a consequence, the decision was made to use a fairly simplistic and lightweight neural network, building a framework around it that constantly uses the available computing power. On the other hand, the dataset used for the training of the network is not static, instead, it changes on a weekly basis. The original dataset does not generalize well to real life as it does not cover a wide variety of scenarios, resulting in the misclassification of many cones when testing on the field. Consequently, the team adds new data to the dataset after every test run. The network's performance can hence increase on a weekly basis, if and only if it is retrained or improved with the updated dataset.

Overall, to enable the work with limited computational power as well as an ever-changing dataset, we present a solution to continuously train and validate the neural network when one of the base components for the training of the network changes. The presented solution aims at the rapid generation of machine learning models through automation using CI-pipelines.

In section 1.1 a discussion about related work takes place to put this paper in the context of the machine learning landscape. Section 2 introduces all concepts that are required for the understanding of the paper. In section 3.1, the approach and important implementation details are explained. The solution is evaluated in section 4 and future prospects are described in section 5.

1.1 Related Work

Traditionally ML-based approaches are based on a data-centric design containing the data acquisition, analysis, and preparation for the ML models, also known as the CRISP-DM model [15], depicted in fig. 1.

While the focus typically lies in the design and training of the models, a paradigm called MLOps coins the idea to improve and accelerate the process for providing production-ready software [5]. One recent and well-known application by Tesla is described by Andrej Karpathy's Ted Talk called "AI for Full-Self

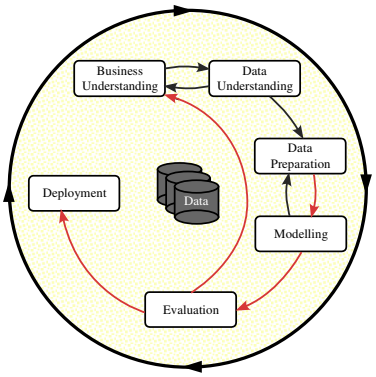


Fig. 1: Crisp-DM Process Model for Data Mining (edited from [15]).

Driving at Tesla". Their approach uses continuous integration to constantly feed critical sensor data of vehicles into their supercomputer Dojo [9] for NN refinement. This approach aims at the improvement of the network for edge cases. While the project of Tesla vastly exceeds the complexity proposed in this paper, we also present an MLOps approach to continuously improve a neural network through automation. We do this by applying DevOps techniques such as continuous integration to the machine learning process, a workflow that was also presented and evaluated by Karamitsos et al. [4].

2 Preliminaries

This section introduces basic concepts and the overall software architecture for easier comprehension of the ideas and implementations shown in section 3. In section 2.1, we discuss the DevOps tools that are used to build our development framework. Section 2.2 introduces the theory behind image classification using neural networks. Finally, section 2.3 presents the operational software pipeline that is used for autonomous driving.

2.1 DevOps Tools

To solve a difficult task such as autonomous driving, a certain level of software complexity is necessary. The complexity lies within individual tasks such as the perception of objects or control theory, but also results from the composition and connection of components.

DevOps tools and practices combat these difficulties. The methodology enforces development practices that ensure stable versioning and continuous delivery of software. This is typically accomplished using tools such as version control systems, but also Continuous Integration (CI), where staged builds (typically referred to as build pipelines) consisting of shell scripts are triggered once code is updated in order to test, build, and deploy software [1]. A prominent provider for

such services is GitLab[4]. Within GitLab, the build pipelines have multiple capabilities that exceed simple scripts. It is possible to trigger other pipelines and to use (software-) artifacts from external pipelines and feedback can be portrayed through metrics and info texts.

Overall, build pipelines are perfectly suited to improve the quality of code through automated testing which enables developers to "Commit Daily, Commit Often" to introduce a culture which can greatly improve debugging capabilities [8].

2.2 Image Classification

Image classification is a fundamental problem in computer vision that has been prevalent for decades, and it has been used throughout history as one of the key instruments to benchmark the various approaches to artificial intelligence. There are many techniques that can be used to classify images into meaningful categories, namely support vector machines, fuzzy sets, genetic algorithms, or random forests [11,7]. However, with increasing computational power and availability of recorded data, the current state of the art is deep learning using neural networks (NNs).

While the state-of-the-art architectures of NNs evolve year by year, the training process has remained mostly the same. First, the given dataset is split into training, validation, and test sets. The neural network is trained and improved using the training set, while the validation set is used for further refinement of the model and its hyperparameters. Finally, the test set is used to check the performance of the neural network on unseen data.

During the training process, complex networks are prone to overfitting [16] generating perfect accuracy in prediction performed on the training data. However, at the same time, these models lose accuracy on the unseen test data. Consequently, the goal is to improve the performance of the neural network on unseen test data, rather than training data.

2.3 JARVIC

In order to understand how and why image classification is used, core principles of JARVIC, a self-developed software used for autonomous driving, must be introduced. In essence, perceived sensor data is processed and then used to generate commands for actuators that control the steering and throttle of the vehicle. This is done using a software pipeline that is depicted in fig. 2 and is inspired by the pipeline of AMZ from 2019 [3]. Starting from the top, we have the Perception component, which takes in Lidar and camera data and processes these to allow for the detection of object location relative to the vehicle. This information and additional sensor data in the form of vehicle odometry (acceleration, wheel speed, etc.) is used by the Estimation component, to generate a map

[4] GitLab is a DevOps platform that aims to assist software developers with project management, versioning, etc. See `https://about.gitlab.com/company/`

and localize the car within. The map is processed by the Planning component calculating a suitable trajectory for the vehicle to follow. Finally, the Controls component uses the trajectory to derive control commands for the steering and throttle of the vehicle.

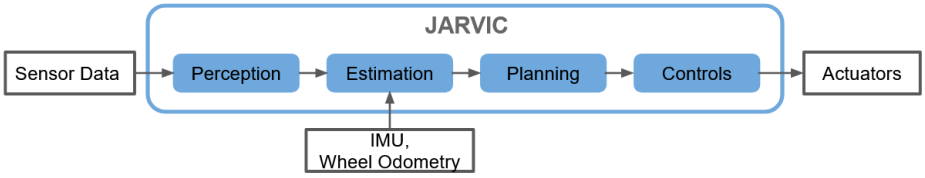


Fig. 2: Abstraction of JARVIC Software architecture.

The neural network described in this paper is found in the Perception component. As previously mentioned, the input for this component is Lidar point clouds and camera images. Both sources are used in a sensor fusion approach to combine the advantages of both sensors. Lidar is specialized in depth perception and camera images are great for the detection of objects and colors [14].

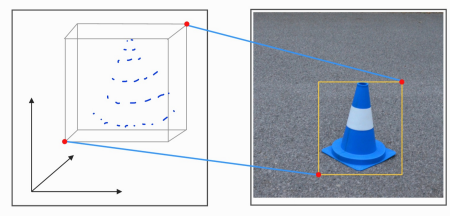


Fig. 3: Fusion of Lidar and Camera data[5].

The fusion workflow is depicted in fig. 3 and begins by matching the coordinate systems of both sensors. Once the Lidar detects a point cloud that resembles an object, the corresponding area in the video feed is cut out and then classified by color or as a no cone. This classification is done using a fairly simple neural network designed for image classification. Once the image is classified, the pose and color of the cones are forwarded to the Estimation component for further processing.

3 The Neural Network CI-Pipeline

The motivation section hinted at the unique characteristics of the use case regarding NN training. The NN of choice is the MobileNetV2 [10] which is fairly

[5] The figure is kindly provided by Leon Schwarzer[0000−0002−0882−3912]

small with its 3.4 million adjustable parameters. This neural network is designed for usage on mobile and embedded devices.
A lightweight neural network was picked for two reasons. First, the task of image classification on a small (56x60 pixels) image does not require complex approximations. Choosing a more complex architecture such as the DenseNet with 46 million parameters, can perform worse since it has a higher runtime than smaller NNs [10] and is prone to overfitting due to high variance [6]. Second, the usage of a GPU was avoided to cut costs and power consumption. Instead, the NN is executed on an Edge TPU co-processor that is integrated into the hardware using a Coral PCIe Accelerator [6]. While the Coral setup is extremely efficient, it is also limited in its capacity to run complex neural networks.
An additional unique characteristic of our scenario is the frequently changing dataset of cone images used for training. The first iteration of the dataset was an adapted version of the Formula Student Objects in Context (FSOCO) dataset [13] which emerged through the collaborative efforts of multiple Formula SAE teams. Hence, the cone images vary in quality and a multitude of cameras were used for capturing. This is not necessarily bad, as a diverse dataset for training can lead to great generalization of the NN. However, the raw images from FSOCO did not exactly portray the characteristics found in real data. In

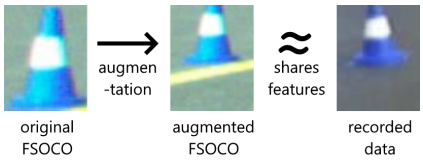


Fig. 4: Data augmentation of FSOCO dataset to resemble the real data.

an attempt to match the dataset closer to the real data, an augment of the FSOCO dataset is performed. A depiction of that step can be seen in fig. 4. The augmentation focuses on imitating the size variance and cone position in captured images, even adding cropped cones to the augmented dataset. Analyzing, the average image shows that a cone is further zoomed out and less centered, compared to the original dataset. Using a CI-pipeline helps explore numerous configurations quickly, including augmentation setting and the integration of recorded data (from manual testing sessions) into the training dataset.
To ensure the usage of the best-performing NN, the following four requirements are used:

1. Train a new NN once the augmentation algorithm is changed
2. Train a new NN once images are added to the training dataset
3. Train a new NN once the training algorithm or NN architecture change

[6] Coral offers hardware and software platforms for embedded systems. Our accelerator: `https://coral.ai/products/pcie-accelerator`

4. Validate the performance of any newly trained neural network

Section 3.1 will discuss the solution using CI-Pipelines to automate the training and validation process of the neural network.

3.1 Design and Implementation

Referring back to fig. 1, we want to improve this paradigm through the use of MLOps techniques such as CI-pipelines. The goal is to fully automate the modeling, evaluation, and deployment steps (signified by the red arrows) for the rapid generation of models. This means that any changes triggered by the Data Preparation step automatically result in a trained model, an evaluation, and the deployment of production-ready software. The developer can then analyze the automatic evaluation, improve their Business or Data Understanding, and start the cycle again if expectations are not met.

To best explain the approach hands-on the focus will be kept on the workflow of the scenario when the data augmentation is updated since the other workflows are part of this case. The design, consisting of a pipeline, is depicted in fig. 5. The process starts, once a developer changes the augmentation script *augment.py* inside the *prepare_dataset* project. The pipeline executes the script, preparing and augmenting the original FSOCO dataset. To accelerate feedback, a fraction of the FSOCO dataset is augmented, which serves as a sample.

The output is then combined with a set of self-recorded cone images called *self_train*, to train the NN with the corresponding *train.py* script. The resulting network is validated using the *validate.py* script with self-recorded data stored in *self_val*. If the validation shows an improvement to the previous version, the neural network is updated within JARVIC. As mentioned before, the cases of retraining of the NN when the dataset changed (meaning *self_train* increased) or when the training algorithm changed (*train.py*), are almost identical to the presented one. In both cases, the untouched pipeline steps can be skipped, resulting in a shortened overall pipeline.
While the design above describes a single pipeline, the reality is a little bit more nuanced. Separate CI-pipelines are used for the dataset preparation, NN training, and for validation. These pipelines are then triggered in succession, automatically passing on the respective artifacts to continue validating the model. An additional detail is that it does not make sense to endlessly increase the amount of self-shot images for training. While adding data is beneficial, especially data that is captured using the real sensors, there is a point of diminishing returns with regard to the resulting NN performance [2]. Hence, the pipeline is used to test different combinations of datasets with varying sizes and characteristics, to ultimately find a suitable combination.

For demonstration purposes, we provide an exemplary simplified implementation of the proposed solution [12].

4 Conclusion

As hinted at throughout the paper, the aim of the presented solution is to save time and resources through the automation of training and validation of machine learning models. Using our solution, the pipeline does indeed provide a production-ready model that can be integrated into the JARVIC software stack. The CI-pipeline allows the user to assert the viability of NN training setups with the push of a button. This means that the hurdle to test ideas is almost diminished, meaning that a broad spectrum of ideas can be put into practice.
Even though manually running the scripts required for training and validation might not always take a lot of time, it does require human interaction every time the prior step is finished. In our use case, this sometimes meant the difference between 48 hours and 8 hours to retrieve a model, even though computation took the same amount of time.
To further assess the viability of the proposed solution, we regard the performance of the cone classifier NN used within JARVIC. The constant testing of new setups was primarily used to change the pre- and post-processing steps. This resulted in a significant increase in the performance of the neural network, as depicted in section 4. These confusion matrices show the performance of the neural networks when classifying images of *blue*, *yellow*, *orange*, and *large orange* cones and *background* images. In the beginning, the neural network exhibited only an accuracy of 18.37% on the self-collected test dataset, with a significant number of misclassifications, as depicted in the confusion matrix of fig. 6a. However, as seen in fig. 6b, the classes are now predicted correctly with an accuracy of 91.83%. In

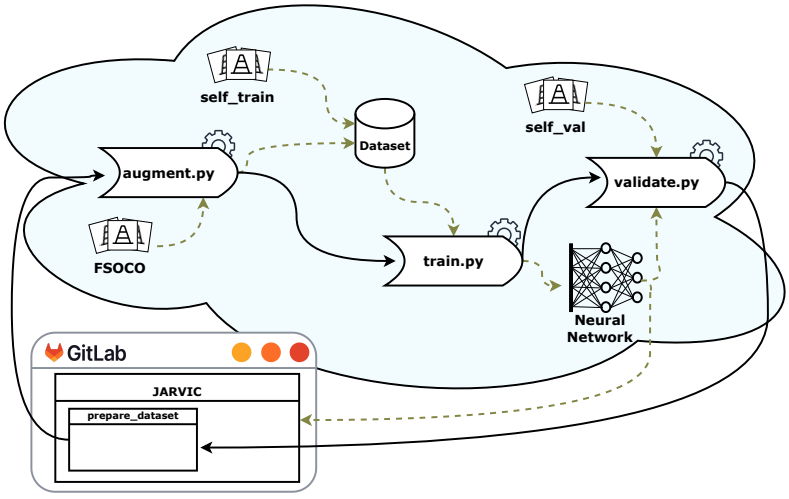
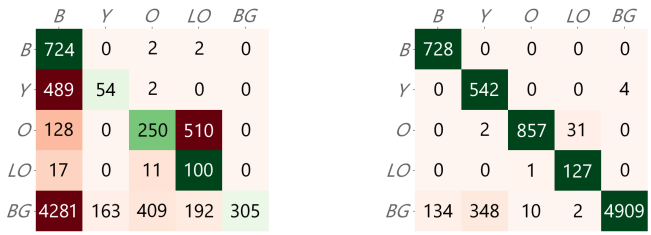


Fig. 5: Overall idea for the design of the continuous NN integration.



(a) Initial performance (b) Final performance

Fig. 6: Confusion matrices of classification NNs. Green cells signify correct and red cells incorrect classifications.

this case, achieving further improvement is exceedingly challenging, as the test dataset was manually labeled, and some cases are impossible to classify even by humans.
While such an improvement of neural networks is not exclusive to cases where automated training and validation are in place, this mechanism motivated the developers to try out dozens of varying setups.

5 Future Work

The current architecture of the CI-pipelines already provides validation of the trained neural network with data that is captured on the same hardware as installed on the final vehicle. The quality of the NN can therefore be assessed by

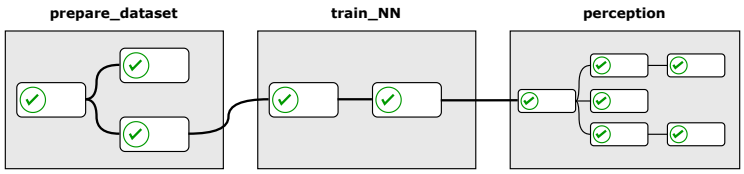


Fig. 7: Concept for extending the NN validation with the Perception CI-pipeline.

the number of correctly classified cones. However, it is not clear whether NNs with equally large test errors, perform equally well in an actual racing scenario. Depending on the algorithms used for the map generation or the boundary estimation of the track, different errors could lead to varying results. With a test error of 50%, one classification NN could be correct for one set of cones (blue or yellow) and completely wrong for the other, while a second NN could have the same error distributed over both kinds of cones equally. It is obvious that

depending on the evaluation of the NN, either one of these errors could be better or worse.

This is why validation of a newly trained NN in the already existing validation pipeline within JARVIC could result in a more accurate representation of the NN performance. The concept for this step is depicted in fig. 7. Some of the validation techniques used in the CI-pipeline for the *Perception* component also include the execution of the *Estimation* and *Planning* components. Hence, deeper implications of the NN performance could be analyzed, and a final conclusion of the performance of the NN could be drawn.

References

1. Fowler, M., Foemmel, M.: Continuous integration (2006)
2. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
3. Kabzan, J., Valls, M.I., Reijgwart, V.J., Hendrikx, H.F., Ehmke, C., Prajapat, M., Bühler, A., Gosala, N., Gupta, M., Sivanesan, R., et al.: Amz driverless: The full autonomous racing system. Journal of Field Robotics **37**(7), 1267–1294 (2020)
4. Karamitsos, I., Albarhami, S., Apostolopoulos, C.: Applying devops practices of continuous automation for machine learning. Information **11**(7), 363 (2020)
5. Kreuzberger, D., Kühl, N., Hirschl, S.: Machine learning operations (mlops): Overview, definition, and architecture. IEEE Access (2023)
6. Lever, J., Krzywinski, M., Altman, N.: N. model selection and overfitting. Nature Methods (2016). https://doi.org/https://doi.org/10.1038/nmeth.3968
7. Lu, D., Weng, Q.: A survey of image classification methods and techniques for improving classification performance. International Journal of Remote Sensing **28**(5), 823–870 (2007). https://doi.org/10.1080/01431160600746456
8. Meyer, M.: Continuous integration and its tools. IEEE Software **31**(3), 14–16 (2014). https://doi.org/10.1109/MS.2014.58
9. Salecker, J.: Whitepaper-tesla-floating-formats (11 2021)
10. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: Inverted residuals and linear bottlenecks (2019)
11. Stathakis, D., Vasilakos, A.: Comparison of computational intelligence based classification techniques for remotely sensed optical image classification. IEEE Transactions on Geoscience and Remote Sensing **44**(8), 2305–2318 (2006). https://doi.org/10.1109/TGRS.2006.872903
12. Steffen, B., Zohren, J., Pazarci, U., Kullmann, F., Weißenfels, H.: Gitlab Demo Project for Paper "Continuous Integration of Neural Networks in Autonomous Systems" (Sep 2023). https://doi.org/10.5281/zenodo.8370907, `https://doi.org/10.5281/zenodo.8370907`
13. Vödisch, N., Dodel, D., Schötz, M.: Fsoco: The formula student objects in context dataset. arXiv preprint arXiv:2012.07139 (2020)
14. Wei, P., Cagle, L., Reza, T., Ball, J., Gafford, J.: Lidar and camera detection fusion in a real-time industrial multi-sensor collision avoidance system. Electronics **7**(6), 84 (May 2018). https://doi.org/10.3390/electronics7060084, `http://dx.doi.org/10.3390/electronics7060084`
15. Wirth, R., Hipp, J.: Crisp-dm: Towards a standard process model for data mining. In: Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining. vol. 1, pp. 29–39. Manchester (2000)
16. Ying, X.: An overview of overfitting and its solutions. In: Journal of physics: Conference series. vol. 1168, p. 022022. IOP Publishing (2019)