RUHR UNIVERSITÄT BOCHUM

RUB

RUB Faculty of Computer Science

Analysis and Evaluation of Experimentation Management Tools applied to Multi-Model Machine Learning in Autonomous Driving Systems

Henriette Knopp

Schriftliche Prüfungsarbeit für die Bachelor-Prüfung des Studiengangs Angewandte Informatik an der Ruhr-Universität Bochum

Abgabedatum – 17. April 2023.
Software Engineering Faculty.

Erstprüfer: Prof. Dr. Thorsten Berger
Zweitprüfer: Dr. Sven Peldszus

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für eine andere Prüfung eingereichten Arbeit.
Ich erkläre weiterhin, dass ich die Arbeit nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Datum	autor

Contents

Abstract

Autonomous Driving Systems (ADS) are usually complicated large scale systems and contain several nontrivial interconnected ML models. They are part of the bigger category of Multi Machine Learning Model Systems, which are software systems that rely on more than one ML model. Because the systems share their environment with humans, the application of ADS is very volatile. The problem is that an environment under the influence of humans is not easy to predict. Resulting safety concerns are cause to fulfill certain standards and expectations toward quality that also affect the development phase of the system [2, 1]. To fulfill all the safety aspects, every part of the development process of ADS needs to be evaluated. While there is already research on the topic of testing ADS [3], we want to take a closer look at experimentation management in ADS with multiple machine learning models. We especially want to investigate whether current methods and tools are feasible for such systems and discover if there is potential for future research.
This thesis is an exploratory case study on a small scale subject system that was previously developed as a prototype for a Formula Student race car, the KoopaCar. In this thesis, we want to explore how experimentation management tools can support the development of Multi Machine Learning Model Systems, with a focus on ADS. We want to especially discover how the requirements for experimentation management can differ, and if the requirements of the management of multiple machine learning models are supported by experimentation management tools.

1 Introduction

1.1 Context

This thesis revolves around Multi Machine Learning Model Systems, which are systems that are used to implement an intelligent or autonomous behavior. While such systems are used in a variety of fields and domains, this thesis originates in the domain of ADS. The expectations for those systems is that they rely on multiple machine learning models. An example for such a system in the domain of ADS is Apollo [4].
The other important part of this thesis are experimentation management tools. Experimentation management tools are software applications that were developed to help practitioners develop and optimize ML models. The problem is that to develop an optimal model extensive experimentation is necessary which results in equally extensive accumulations of data and results. To organize experiments and compare ML experiments, experimentation management tools are used.
As subject system for the thesis, a prototype that was previously developed by students during a university course will be used. This system is called KoopaCar. It originates from the system that was developed as a prototype for the Formula Student Germany [5], which is a design competition for students. The system needs to be further developed to fulfill certain requirements. More details on this follow later.

1.2 Problem

This thesis is motivated by previous research on ADS and ML management tools. In [3], Peng et al. published a case study on Apollo, which focused on the usage of ML models in the system. It is apparent that there are a lot of ML models used in the system. While Peng et al. focused on typical software engineering practices and especially testing, this thesis will evaluate ML experiments to optimize the ML models. For a system to run without errors, it is crucial that the calculations in the system run on accurate data. Since the data is processed using ML models, it

is worth to take a look at how the process of achieving accurate predictions can be improved. The problem is that there is little to no research into ML experiments in ADS or in general Multi Machine Learning Model Systems. This thesis will provide a first insight into the problem.
This thesis will then go on to evaluate how experimentation management tools can be used to improve ML experiments in Multi Machine Learning Model Systems. This follows the work of Idowu et al. in [6] and [7]. Idowu et al. state that the usage of asset and experimentation management tools helps to avoid problems commonly associated with the development of ML models from a software engineering point of view [6]. This means that when taking a look at practices associated with ML experiments, it is necessary to take a look at experimentation management tools, since those tools are known to improve the development of ML models.

1.3 Research Questions

The goal of this thesis is to evaluate and analyze experimentation management tools in Multi Machine Learning Model Systems. The problems outlined in 1.2 indicate challenges that will be worked on in this thesis. First, it is necessary to take a look at what ML experiments look like in Multi Machine Learning Model Systems. Then, by implementing and running matching experiments, experimentation management tools can be evaluated. This leads to the following research questions:

RQ1: How do the ML experiments during the development of ADS and the management of such experiments and corresponding assets differ from the workflow related to other single ML model development and maintenance that is common in other fields?

RQ2: How can experimentation management tools improve the development of intelligent systems that integrate several ML models?

RQ3: How can existing experimentation management tools be further developed to better support the development of intelligent systems implementing integrating multiple ML models?

1.4 Approach

To answer the research questions, the following approach will be used. First, the subject system, the KoopaCar, needs to be further developed to match the characteristics of Multi Machine Learning Model Systems. To this end, we will first take a look at the current state of the system before diving into, how the system was further developed.
Following this, we will begin to take a look at ML experiments in Multi Machine Learning Model Systems. Here, we will investigate different strategies that could be used to optimize the ML models used in the KoopaCar. Those experiments will then be run and evaluated. The quality and evaluation of the experiments does not match actual ML experiments. Despite that, the experiments run over the course of this thesis remain representative.
The experimentation management tool that will be evaluated is selected in a documented selection process. Experiences that were made while running the ML experiments using the experimentation management tool will then be presented.
The code for the subject system can be found in [8]. The code for the simulation used to run the subject system can be found in [9].

2 Background

Before diving into beginning to answer the research questions regarding Multi Machine Learning Model Systems, it is useful to take a look at the background for this thesis. This chapter gives the necessary background on topics related to the before mentioned research questions. At the core of the thesis are Multi Machine Learning Model Systems. We will use ADS as a possible application of such systems. ADS are very popular in recent years and rely heavily on ML and Artifical Intelligence (AI) to achieve autonomous behavior. With that in mind, this chapter will first introduce Robotics Operation System 2 (ROS 2), which was used to develop the subject system. Later subsections contain introductions to Multi Machine Learning Model Systems in general, ML experiments, and ML experiment or management tools.

2.1 Robotics Operating System (ROS 2)

The following contains a short introduction to ROS 2 [10]. It is useful to understand some key concept before diving into the subject system used in this thesis. The following information is from the ROS 2 documentation in [11].
ROS 2 is a collection of software libraries that allow a more consistent development of robotics system. Despite the name, ROS 2 is not actually an operating system. Systems developed with ROS 2 have a certain architecture that revolves around nodes and the data exchange between them. Nodes usually contain Python or C++ code and are executed to perform tasks in the robotic systems. ROS 2 organizes nodes in packages that can be deployed.
The features topics, services, and actions all implement data exchange and communication between nodes. Topics implement a publisher-subscriber scheme, where nodes can exchange data by either sending messages with information to a topic or by listening to such a topic and reacting to any new messages. Any node can subscribe to any topic and use the information contained in the messages. Services are an alternative concept to the publisher-subscriber scheme that the subject system for this thesis, the KoopaCar, uses. Similarly, services are used to exchange data, but in contrast to the publisher-subscriber concept this exchange does not happen

continuously, but only, when a client calls the service. Actions are a more complex variation of services that are meant for long-running tasks like teleoperation of robots. Lastly, parameters are used for node settings. Parameters can be defined in the node and hold any datatype. Parameters can also be changed from outside the node and thus used for settings. An application for parameters could be to change the frequency in which a node executes a task.

2.2 Machine Learning Models and Machine Learning Experiments

This sections will provide the basic background on ML models that are used within the thesis. Since this thesis evaluates and analyzes ML management tools, it is important to have a basic understanding of ML models and common practices. First, the development cycle of ML models is introduced. Later, background on ML model types, training, and evaluation is provided. The last section will then present common practices used in ML experiments. The sections only gives a brief overview of the most important concepts.
The information on Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), ML model training and hyperparameters in general is mainly from Goodfellow et al. in [12].

2.2.1 Development Cycle of Machine Learning Models Systems

Before moving onto the details of ML models, this section summarizes the bigger picture of ML mode development. The development cycle of ML experiments can be seen in Figure 2.1. The development starts very similar to any software project. In the first development phase, requirements are being documented and analyzed. In the next development phase, a data set is being created. To this end, data is being collected and cleaned, next features are being extracted. These steps are repeated until the data set is sufficient. After all requirements are collected and a sufficient dataset was built, the development of the actual ML model starts. The first step in the next phase is to design a model. This means that depending on the requirements, a matching model architecture is chosen and implemented. Then, the model is trained on the data set. After the training is finished, the model is being evaluated and then optimized. Optimization refers to tuning the hyperparameters, which are essentially training parameters, retraining the model, and comparing the results to find parameters that yield the best predictions on

unknown data. The process of optimizing a model is what the term ML experiments refers to. The details of how the training works, what hyperparameters there are, and how a model is evaluated will be summarized in the following. As a last step in a model's lifecycle, the model is being deployed. Deployment means that it is used in a software system. To this end, the model needs to be monitored as long as it is in use.

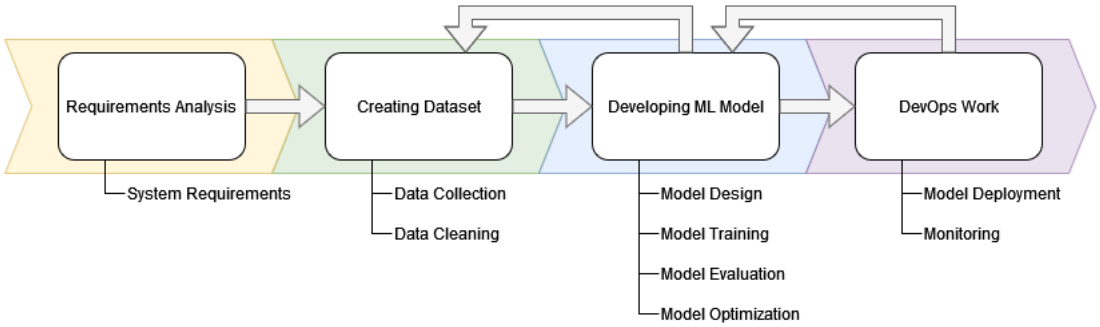


Figure 2.1: A graphical overview over the lifecycle of a ML Model or ML Model System. The figure is based on Figure 1 from [6]. The lifecycle is split into four phases Requirements Analysis, Creating Dataset, Developing ML Model, and DevOps Work. The big gray errors indicate the transitions between the phases. Below the phases important work steps are listed.

2.2.2 Multilayer Perceptron

The MLP is one of the basic types of ML models. There are different types of ML models, which are used for different tasks. The MLP is used to perform a classification task, meaning that it assigns labels to outputs. It is essentially a mapping between the inputs x and the outputs y. [12]
A MLP takes a vector of data as input and outputs another vector as a result. The sizes of both vectors are chosen when defining the models and can not be changed later. The MLP consists of several perceptrons organized in layers. An example of what a MLP can look like can be seen in Figure 2.2.

First, we will take a look at how perceptrons work. A single perceptron takes an input x and runs it through an activation function $\phi(x)$. The result is the perceptrons output y. There are several activation functions, which are best suited for different use-cases. Going into depth on this matter does not fit the scope

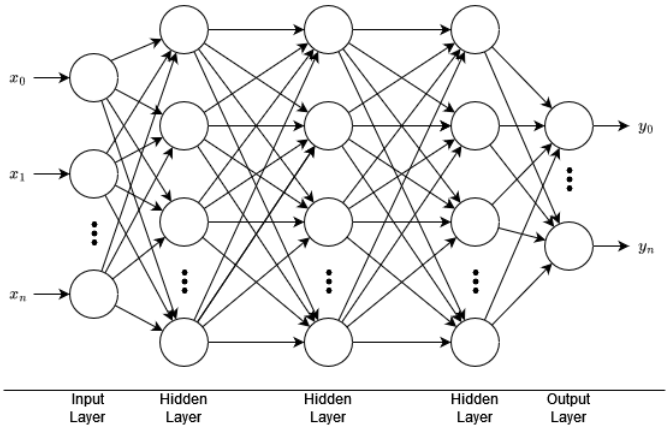


Figure 2.2: Example for a Multilayer Perceptron (MLP). The MLP has three hidden layers. It takes an input x of size n and outputs a vector y of size m. The first layer is called input layer and the last output layer

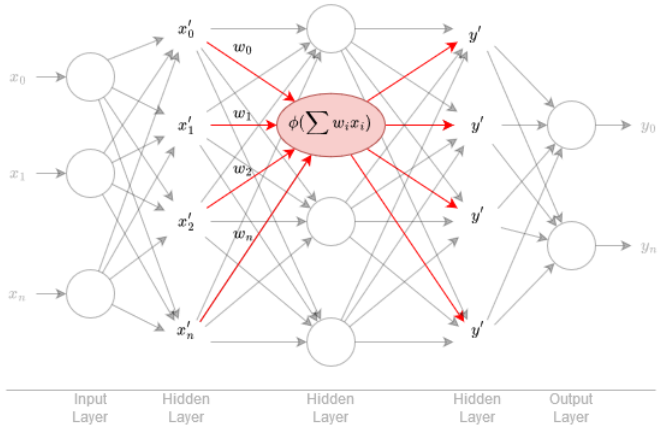


Figure 2.3: The same multilayer perceptron as in Figure 2.2 with a single perceptron highlighted in the second hidden layer. The figure shows the input and output of the perceptron, together with the computations that happen in between. Note that the biases are missing in this description.

and context of this thesis. Regardless, in the following there are some examples for activation functions and a brief description. An overview of the activation functions can be seen in Figure 2.4. The first activation function in the figure is a simple linear function $f(z) = z$. The linear function can be used as activation function, but is not the best choice in most cases. Next, there is the Sigmoid function $sigmoid(z) = \frac{1}{1+\exp^{-z}}$. The sigmoid function is an activation function that

is often used in the output layer. It is very simple, and has the property that it normalizes the output. Another activation function that is often used in the output layer is the Softmax function $y_i = \frac{\exp^+ z_i}{\sum \exp^{z_i}}$. The Softmax function has the advantage that it can be used to produce probabilities for a set of mutual exclusive classes. Lastly, the Rectified Linear Unit (ReLU) generates a linear mapping for values larger than 0, $ReLU(z) = max(0, z)$. ReLU has risen in popularity in the last few years since it makes the training converge faster. [12]

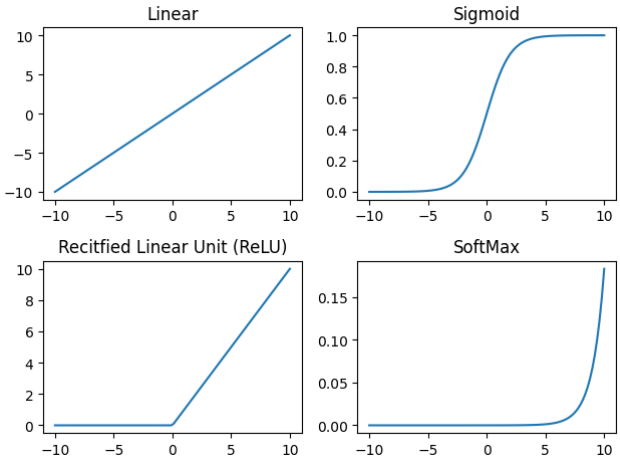


Figure 2.4: Overview over a selection of activation functions. The figure shows the plot for the linear, sigmoid, ReLU and Softmax activation functions. The activation functions are all used in different use-cases. Activation functions are used in MLPs, CNNs and other ML models.

The MLP consists of a set of perceptrons, which are interconnected. There is usually an input layer, followed by a set of hidden layers and then an output layer. For each layer in the MLP, each perceptron takes the input from every perceptron in the previous layer. The input is weighted and summed up $\sum w_i x_i$. The weights are updated when the model is trained to change the model's behavior. The result is then run through the activation function ϕ to generate the output y. This process can be seen in Figure 2.3.

2.2.3 Convolutional Neural Networks

CNN another type of neural networks. A CNN usually consists of multiple convolutional layers. Figure 2.5 shows an example for what such an architecture can look like. The name of the convolutional layers comes from the mathematical

operation convolution, which they implement. CNNs are used to extract features from images or sequences of data in vector form. [12]
Every convolutional layer receives an input of a predefined shape. It is possible to use matrices or vectors. The input is then processed using a kernel, which is essentially the heart of the convolutional layer. The kernel is smaller than the input and is slid over it. The kernel is multiplied with the input for every position. In doing so, not every element of the input has a direct influence on every element in the output. This is called sparse connectivity. [12]
Pooling layers are used to decrease the size of the input data of a layer. By taking averages or maxima from quadrants in the input matrix, the size of the input for the next layer becomes smaller. [12] By reducing the size of the input data of a layer, the calculations will be sped up, increasing the model's efficiency. The process is called down sampling [13]. Figure 2.6 visualizes how pooling layers work. The example that is shown is a max pooling layer.

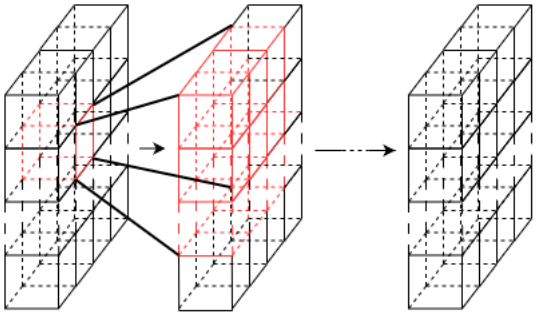


Figure 2.5: Example of a convolutional neural network. The blocks symbolize the input. The red markings indicate how one element of the input affects the elements of the output, this is what is known as sparse connectivity.

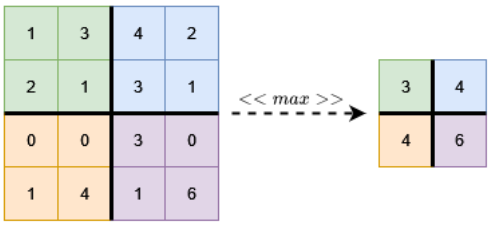


Figure 2.6: Example of a pooling layer. The example shows the application of max pooling to a 4x4 matrix. In max pooling, the input is split into sections and the maximum value from each section is selected. The other values are discarded. The result is a 2x2 matrix.

The internal parameters that are updated in training using optimization algorithms are the kernels.
In some cases, a combination of CNNs and MLPs is used to make a predictions based on image data. In this case, the CNN is used to extract information and reduce the data complexity, while the MLP is used to classify the image based on the extracted features.

2.2.4 Machine Learning Model Training

The previous sections gave some background on ML models and started to introduce the term ML experiments. Section 2.2.1 stated that repeated training is part of the development cycle for ML models. In the following, we will take a look at what is meant when talking about ML model training. The general concept should be familiar, nevertheless it will be introduced again in the following to lay a basis for the introduction of hyperparameters.
The goal is to optimize a model. Optimizing means that the prediction the model produces are as close to the ground truth as possible. The ground truth refers to the correct predictions and are only known for labeled data sets, i.e., the training set. How close the prediction is to the ground truth is measured using a loss function. There are different loss functions for different types of ML models or models' outputs. One very common loss function is the Mean Squared Error (MSE), which is shown in Equation 2.1. N is the number of samples that are being computed, y_i is the true label for the i-th sample in the set, and $\hat{y_i}$ the corresponding prediction. The MSE is equivalent to the l_2 norm. It is easy to see that the loss calculated with the MSE becomes 0 in the case that the prediction and the expected output are the same. [12]

$$MSE = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y_i})^2 \tag{2.1}$$

An alternative for MSE is Cross-Entropy, which can be seen in Equation 2.2. The Cross-Entropy loss function uses the probabilities for the possible classes to calculate the loss. n refers to the number of classes, y_i the true probability of the i-th class for a given sample, and $\hat{y_i}$ the predicted probability.

$$CE = -\sum_{i=0}^{n} y_i log(\hat{y_i}) \tag{2.2}$$

In addition, training the model should also be efficient to reduce training times. Efficiency in training refers to memory usage and run time. Training large networks can result in a long training times and high memory usage. Therefore, an

efficient use of limited computational power is one crucial aspect in model training.

To get a better understanding of how a ML model is trained and optimized, it is useful to think of the model as an approximation of a non-trivial function. The ML model approximates the relation that exists between the data (input) and desired prediction (output). The model is used as a general mapping that is fitted to approximate this relationship during training. It extracts features and learns which feature matches which possible prediction.

To find the model's internal parameters that yield the best result, several training iterations are run on a dataset. One such iteration is called epoch. In the first epoch, the weights are initialized. Note that going into depth on weight initialization exceeds the scope of this thesis. During training, every epoch consists of the following steps. First, a data samples are passed through the network. This is called forward pass or forward-propagation. Then, by comparing the result to the ground truth, the loss is being calculated using a loss function. After every sample was run through the network, the average loss is being calculated. Using back-propagation, another algorithm such as stochastic gradient descent, and the loss from before, the weights or other internal parameters of the network are then being updated. This is called backward pass. This basic concept behind the training can be generalized for almost any type of model or model architecture. More on back-propagation can be found in [12].

The learning progress can be influenced using different parameters. Some of those parameters will be explained in the section 2.2.6. But before diving into the hyperparameters in depth, the terms over and under fitting are introduced, because they influence how hyperparameters are chosen.

2.2.5 Overfitting and Underfitting

Before introducing hyperparameters and discussing model evaluation, it is useful to introduce the concepts of over and underfitting. Both occur, if parameters are chosen in a way that lead the model to not fit the data at all or to fit the data too good.
Under fitting means that a model was not trained good enough to yield good predictions. This can be seen in a poor accuracy on the training set, as well as previously unseen.
Overfitting is a more interesting concept. There is no easy explanation to why

overfitting occurs. The basic explanation behind the term is that the model was trained to fit the training data too good. But by doing so, it also learned features that only occur in the training data. As a result, the accuracy on data that is not included in the training set, and thus previously unseen, is very low.
Because of over and underfitting it is important to evaluate ML models and tune hyperparameters accordingly. Which values for each hyperparameter yield the best results is different for every model and for every data set. As a result, it is necessary to experiment with possible parameters and compare the results from the experiments to the performance on a validation set.

2.2.6 Hyperparameters

The parameters that influence the training are commonly called hyperparameters. It is important to differentiate between hyperparameters and parameters in general. Hyperparameters are parameters that can be influenced manually to change the model's behavior. Parameters on the other hand usually refer to internal model parameters like weights. They define a ML model's behavior and are updated during training, but not influenced manually. In the following, some hyperparameters that are used in ML experiments in this thesis are introduced. Table 2.7 gives an overview over parameters that will be discussed in the following.

There are a lot of hyperparameters that can be used to influence a ML model. Only a few of them are relevant in this thesis. The few that are used in experiments in a later chapter will be explained in the following. More information on all the hyperparameters can be found in [12].
The first hyperparameter is the training and validation split. It is a common practice to not use the whole data set for training [14]. Instead, the set is split into two parts. One, the larger, is used for training and the other for validation. By using two sets and calculating the error for both sets, despite only using one for training, it is possible to detect overfitting during training [14]. A good indicator for overfitting is, when the loss on the validation set increases, while the loss on the training set continuous to decrease. In addition to splitting the data set into a training and validation set, the data set is also commonly shuffled. Since the order and also the composure of the training and validation set has an influence on the training results, randomizing the set in advance can be advisable.

The next hyperparameter in the table is the number of epochs. An epoch refers to one step in the model training. In each step of the training, the complete data set is passed through the network and the loss propagated back. The number

Hyperparameters	Effect on the models behaivour or training in general
number of hidden layers	It increases what the model is capable of learning from the relation between data and expected outputs.
training validation split	It changes how much data is used for training and validation. It can impact the ML model's performance such that fewer data for training means that the model is not learning as good. Fewer data for validation means that overfitting and underfitting are not detected as reliable.
data shuffling	Randomizes the order of the data samples used and can change the entire set for training in the case the set is shuffled before it is being split into training and validation set
number of epochs	It changes the length of training, too few epochs cause underfitting and too many possibly overfitting.
batch size	It changes the sizes of the mini-batches. Smaller batch-sizes take up less memory, but result in more variance in the training results.
learning rate	Learning rate changes the size of an update steps. Too small learning rates mean that training progresses slower, too high learning rates can result in training to diverge.
convolution kernel size	Changes the number of learnable parameters in a model. A wider kernel reduces the model's capacity, and a narrower reduces memory cost.

Table 2.7: List of Hyperparameters and the effect on the ML model's behavior. The table is based on Table 11.1 in [12]. For more information on the different hyperparameters and their effect, read [12].

of epochs needs to be high enough to make progress in training and avoid under fitting. If training would go on forever, at some point the loss and other metrics converge or the model begins to overfit. Training should be stopped either if the ML model is not learning anymore or before overfitting occurs. [12]

The last hyperparameter in the list is the batch-size. The term batch-size is related to training with mini-batches. Mini-batches means that for every training epoch not the whole data set is used at once, but instead several iterations are run with smaller subsets of the training set. These sub sets are called mini-batches and

their size is regulated with the hyperparameter batch-size. Choosing a small batch size as benefits for the memory usage during training, since less information needs to accessible at once. It also makes it feasible to use large data sets in training that would otherwise not fit in the computer's memory. If the batch-size is chosen too small, the learning becomes unstable. This means that the training diverges from iteration to iteration. If the batch-size is chosen bigger, the training is more stable in later epochs. The downside is that the training slows down and uses more memory. [12]

In the next section, metrics are introduced that can be used to evaluate ML models and their performance.

2.2.7 Machine Learning Model Evaluation

In the previous sections, ML model training was introduced together with several hyperparameters. It was mentioned that the goal in designing a ML model and in ML model experiments is to optimize a model. Section 2.2.4 and 2.2.6 stated that hyperparameters can be manipulated to influence the trainings results. In this section, we will now take a look at how a ML model can be evaluated. To this end, we will introduce metrics that will be used in ML experiments in later parts of the thesis. For every metric, the mathematic background and usage are outlined. For example, it is explained which values or patterns indicate that the model was over or underfitted.

The only metric that is used in this thesis is the loss. The loss contains the result of the loss function that was used in training. During training, the loss is used to update the weights. For every epoch, the loss is calculated by averaging the deviation of the prediction from the ground truth for every data sample. The two loss functions MSE and Cross-Entropy were explained in section 2.2.4. Minimizing the loss means that the ML model makes fewer mistakes. The loss being high indicates underfitting. If the loss deviates and does not converge towards 0, it might be an indicator that the model is not learning how the data and the predictions are related. Overfitting can also be detected using the loss. To do so, both a training and a validation set is necessary. In the case that the loss on the training set converges towards 0 and the loss on the validation set diverges, the model is being over fitted.

Usually, the loss is not the only metric used to evaluate ML models. Other metrics are used instead or in addition to the loss. Examples of other more complicated

metrics are for example Precision and Recall or the F1 score. More information on other metrics can be found in [12].
Since the goal of this thesis is to evaluate experimentation management tools and practices for ML experiments, only the loss function will be used to evaluate the ML models.

2.2.8 Machine Learning Experiments

To proceed further, a general understanding of the key concepts behind ML experiments that were already mentioned in the previous sections is required. In addition, this section introduces how ML experiments are run and common practices.
First, the term ML experiment refers to the repeated training of ML models with different hyperparameters. The goal to tune the hyperparameters to optimize the model's performance. The ML model's performance can be evaluated by looking at metrics in section 2.2.7. Section 2.2.1 states that experimentation is part of the ML model's development cycle. Section 2.2.6 introduced different hyperparameters and how they can affect the model's performance.
The sections above focused on the background, concepts, and mathematics behind ML experiments. This section will dive into practices and will focus on the software side of ML experiments. First, we will take a look at how ML models are implemented and later on how experiments are managed. This section focuses only on management practices without ML management tools. ML management tools will be introduced in section 2.3.

ML models are usually implemented using common ML learning frameworks, such as PyTorch [15] or TensorFlow [16]. Most commonly, Python is used as a programming language. The purpose of ML frameworks is to provide implementations of functions that are often used in ML. For example, the frameworks provide implementation for common loss functions like MSE or Cross-Entropy. Keras [17] is a high-level API that is amongst others implemented as API of TensorFlow, known as TensorFlow Keras. It allows the user to build and train ML models without having to go into depth on implementing the training algorithm.
In most cases, the frameworks used for implementing and training the ML model provide objects that can be used to evaluate the chosen metrics. In the case of TensorFlow Keras, this function training is `tensorflow.keras.fit()`. The function returns an object, called `history` that contains the metrics specified in the call of `tensorflow.keras.fit()`. The training script is run like any Python script. Common practice is to define parameters that are passed to the training function,

when it is being called.
After every experiment run, the model is evaluated. To compare runs, practitioners often use spreadsheets, where the results for every run are entered manually. The usage of spreadsheets is error-prone and time-consuming. Sometimes practitioners visualize the data.
The code as well as data sets and trained models are versioned. To this end, practitioners rely on Git. In the case of the datasets, Git is not always usable. To achieve good training results, data sets with a sufficient amount of data are necessary. Therefore, a lot of storage space is needed. Git can not always handle the amount of data necessary to train a ML model. Depending on the size of the module, practitioners encounter the same problem for versioning trained models.

The problems associated with ML experiments that were described before, underline the necessity for ML management tools. With the background on ML models and ML experiment that this section provided, the next section will take a look at what ML management tools are and which features they can provide.

2.3 ML Experimentation Management Tools

The previous sections outlined, how ML experiments are run. It also highlighted that there are a lot of different parameters that influence the experiments results, which in turn result in a lot of experiments. All of these experiments need to be evaluated and compared. It is essential that the runs are organized properly to ensure that any run can be reproduced to verify the results.
Despite management tools being on the rise, some practitioners still organize and compare ML model experiments manually. Management tools are designed for common experiments, with one model being optimized. The goal of this thesis is to evaluate how such tools can support the development of Multi Machine Learning Model Systems. But first, it is important to have an understanding of ML management tools and the features they provide. Therefore, the following will give an introduction to ML management tools, their different categories and features.
The three main categories for ML management tools are asset management, experimentation management, and model management and MLOps [18]. In the following, there will be a short introduction for every category and a summary of the features each tool is known for.

The first category that was named are asset management tools. In [6] Idowu et al. did a survey on assent management tools, which will serve as basis for this introduction. Asset management tools are used to version and store any assets that are related to a ML model system or an ML experiment. Asset management is important at any point of the ML model's lifecycle.
To understand what asset management tools do, it is essential to first understand what the term asset refers to in the context of machine learning. Idowu et al. divided assets into the subcategories resources, software, metadata, and execution data. Resources refer to objects and information that is necessary for an ML experiment. This includes for example the data set, but also the model itself and the run environment. Software refers to source code or notebooks that were created by practitioners and are related to the experiment. Next, the metadata is data that exists around the ML experiment. Metadata could reference git commits, dependencies, or information on the state of the ML model system. Lastly, execution data is the term for any data that is created while the system or experiment is being executed. These are for example logs or experiment results.
All the features of asset management tools are related to storing and versioning the above-mentioned data. It is possible that asset management tools implement operations to store and version data on external databases.

The next category are experimentation management tools. Experimentation management tools are more focused on ML experiments. Thus, they are useful during the development and optimization stage of a ML model system. They help practitioners to organize and evaluate different experiments [18].
Features include the collection of meta information that is related to an experiment. In this case, meta information means, amongst other, references to code or git versions or the environment in which the experiment is being run. Experimentation management tools should support logging. Logging means that the user can choose metrics, parameters or other artifacts which will be collected and versioned by the tool. Experimentation management tools usually also have visualization features. They visualize the metrics which are used to evaluate and compare ML experiments. In most cases, it is possible to export an overview over the experiments together with their metrics.
In [7] Idowu et al. proposed a metamodel for experimentation management tools. In their paper on this metamodel, they also introduced core features for experimentation management and summarized challenges.

Next in the list is run orchestration. Run orchestration refers to tools that allow the user to organize their different experiments. This allows for more efficient use of time and computation power available.
Tools that support run orchestration usually implement other features like asset

or experimentation management as well.

The last category on the list is MLOps. MLOps refers to all actions that manage the complete model's lifecycle [18]. This includes amongst others monitoring. MLOps is not the focus of this thesis. Therefore, we will not go into depth on features available for such tools.

2.4 Multi Machine Learning Model Systems

In this section, we will take a look at multi machine learning model systems. The term refers to systems that consist of more than one ML model. It is important to note that in our understanding, the ML models are not independent of each other. This means that those models have a common influence on a result or output. Most commonly, multi machine learning model systems are related to intelligent systems or autonomous behavior.

Over the course of the thesis, we will differentiate between two cases or structures for Multi Machine Learning Model Systems. The first one being that two models run sequential. In this case, the output of the first model serves as input for the second model.
In the second case, two or more models are ordered parallel. This means that, the input of all parallel ML models has a common impact on the next system component. This component can be an algorithm or a ML model. In the case that it is a ML model, the models that are running parallel and the model that fuses their output are also structured sequentially.
All two cases, sequential structure, parallel structure, are visualized below. Figure 2.8 shows the sequential structure that was described. Figure 2.9 visualizes the parallel structure. The last component in Figure 2.9 could be both a ML model or other software component.

Both structures are not mutually exclusive. They can be combined to create more complicated systems. It is important to note that if a system has a more complicated structure, it can always be broke down to smaller components that match the structures that were described. A section from a system that contains multiple ML models in a structure as described before will be referred to as ML model pipeline or in short pipeline in the next chapters. The term Multi Machine Learning Model System refers to the complete system, if such a system contains more

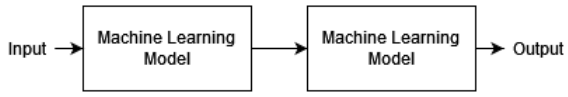


Figure 2.8: Sequential Structure of Multi Machine Learning Model System. Input is passed to the first ML model, processed and then the output of the first ML model is passed to the next ML model to produce the output

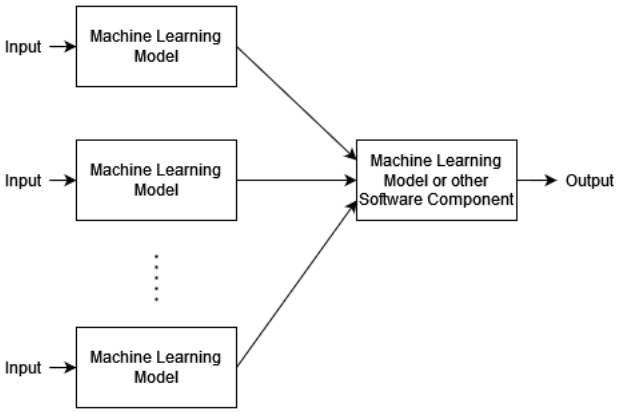


Figure 2.9: Parallel Structure of Multi Machine Learning Model System. n ML models receive input that is independent of each other. The output of the ML models is passed to the next component to produce the final output.

than one ML model, as described before.

The goal of this thesis is to evaluate what challenges there are for running ML experiments and optimizing a ML model pipeline. We will evaluate how the performance and the loss of such a Multi Machine Learning Model System can be evaluated, and how the different structures that were described here affect the performance and loss.

2.4.1 Example for Multi Machine Learning Model Systems in Autonomous Driving

An example of Multi Machine Learning Model Systemsm in the context of autonomous driving is Apollo. [4] Apollo is an ADS system that was developed by

Baidu. The following contains a short introduction to Apollo and outlines the system architecture. By taking a look at Apollo, we will be able to draw comparisons to the subject system used in this thesis and generalize the experiences made in the context of ADS and ML experiments to any Multi Machine Learning Model System.

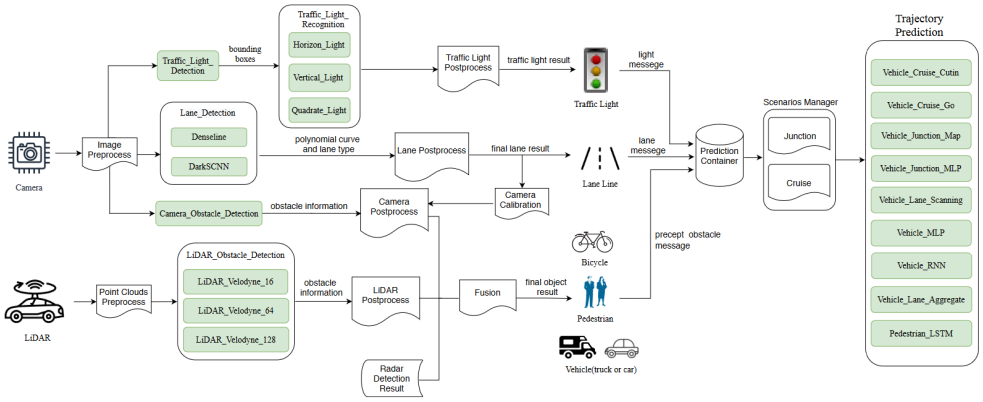


Figure 2.10: Overview of the relevant components in Apollo. The image is Figure 1 from [3]. It shows the data flow through the system and highlights the relevant components. The green boxes are ML models. The figure shows that information is processed by a camera and a LiDAR and process using ML models and other software components to detect objects like traffic lights, lane markings, bicycles, pedestrians, and other vehicles. The information about all of these objects is used for the trajectory prediction, which is necessary to foresee the behavior of other traffic participants.

An overview of the relevant parts involving ML models in Apollos system architecture can be seen in Figure 2.10. There we can see that Apollo relies on information from both LiDAR and Camera to make decisions. The information from both sensors is being processed individually and later fused. The data from the camera is used to detect traffic lights and lanes. Other objects are detected with both the data from the camera and from the LiDAR after the signals were fused.
The structure that is visible fits the structural types for Multi Machine Learning Model Systems that were described in section 2.4. In Apollo, some models run sequential while others are running parallel to each, other. This shows that the structures identified before are useful to achieve a generalization of experiences made with ADS. The results of this thesis can be applied to any system that implements ML in a parallel or sequential structure.

Peng et al. analyzed the usage of ML models and stated that there are a total of 28 ML models in the system [3]. Not all of these models are used simultaneously. From Figure 2.10 we can see that 18 of the 28 ML models are used simultaneously. The models are imported and run from several ML frameworks. This is what motivates the necessity to take a look at the development and optimization practices related to ML models in large software applications.

Chapter 3 will show that the software architecture of the subject system is very similar. Thus, the subject system can be used to gather representative results, while remaining manageable without creating too much of an overhead. The number of ML models in Apollo that need to be developed and optimized shows the necessity to evaluate development strategies and the usage of experimentation management tools. Chapter 4 and 5 will evaluate ML experiments and management tools in systems like Apollo.

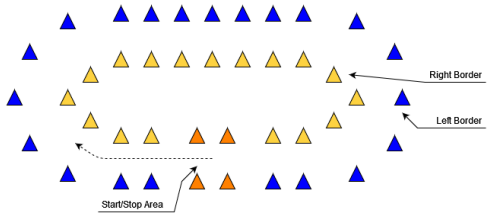
3 Subject System – KoopaCar

As mentioned before, this thesis is a case study collecting experiences with ML management tools in Multi Machine Learning Model Systems. Therefore, it is necessary to introduce a subject system. Since the focus of this thesis are Multi Machine Learning Model Systems, the subject system needs to contain more than one ML model. Its structure should also be similar to real applications of Multi Machine Learning Model Systems. The system should also be easy to handle and run to reduce the overhead for running and managing the subject system. Consequently, the KoopaCar will be used as subject system for this thesis.

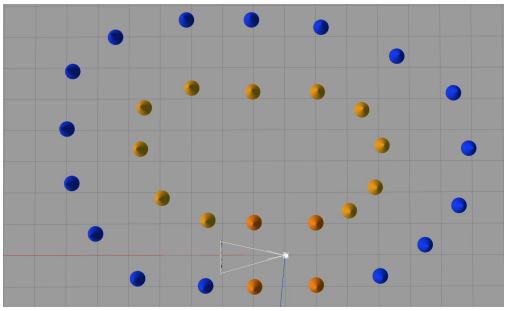
3.1 Application of the KoopaCar

The KoopaCar was first developed as a prototype for a race car for the Formula Student Germany [5]. Formula Student Germany is a design competition where groups of students design and develop racecars. There are different events where teams compete with their self build racecars against each other. One part of the events are static events, where teams present their designs and business plans related to the race car. The other type of events are dynamic events. Another part of the events are dynamic events, which can best be described as races and are scored based on the time it takes to complete the course. Recently, the Formula Student Germany began to host a driverless competition. In this competition, teams compete with self-driving racecars. A detailed description of all events and rules can be found in the Formula Student rule book. [19]
Inspired by the RUB Motorsports [20] team, groups of student developed prototypes for the Formula Student driverless competition using a Turtlebot3 [21]. The RUB motorsports team is a team of students from the Ruhr Universität Bochum that participates in the Formula Student Germany. In the Formula Student competition, racetracks are always marked with different colored cones. The left side of the racetrack is marked with blue cones and the right side with yellow cones. The start and stop area is surrounded with orange cones [19]. The goal for the prototype is to detect the racetrack and drive through it without any user input.

In the best case scenario, the Trutlebot would drive two laps. In the first lap, it would create a map of the racetrack. In the second lab, the Turtlebot can the follow the optimal trajectory.
An example of the KoopaCar operating environment can be seen in Figure 3.1. Figure 3.1a is a drawing of what a racetrack could look like. In this example, the racetrack is almost circular. The start and stop area is framed with four orange cones. The outer circle is the left track border and is marked with blue cones. The inner circle is the right track border and is marked with yellow cones. This makes the driving direction clockwise. Figure 3.1b shows a top-down view of the racetrack inside the simulation used for development.



(a) Drawing of a racetrack.



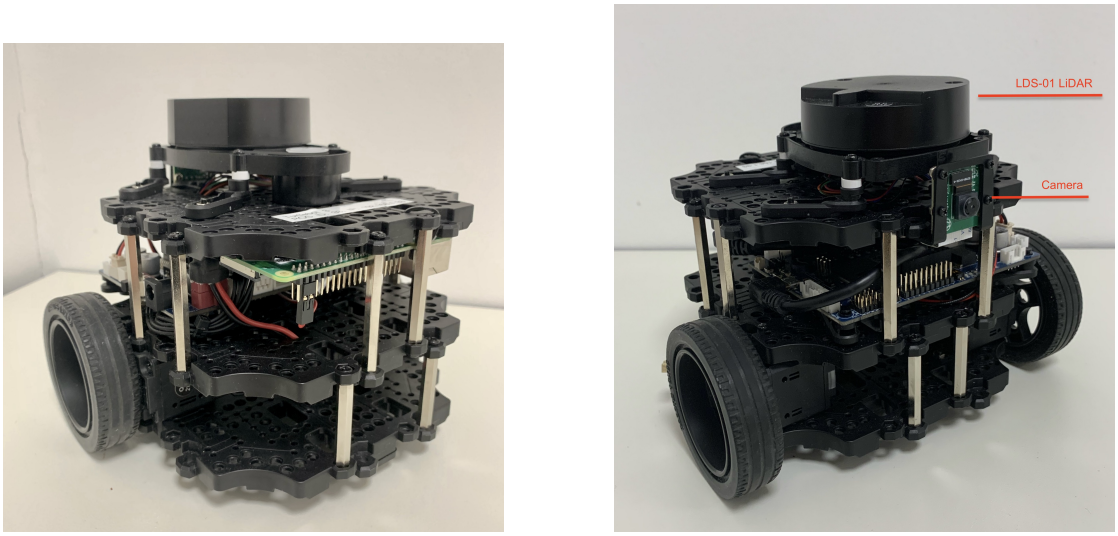
(b) Image of the racetrack taken in a simulation.

Figure 3.1: Examples for a racetrack following the Formula Student rule book. The driving direction is counterclockwise. The starting area is at the bottom of the figures, marked with four orange cones. The yellow cones mark the right side of the racetrack and the blue cones the left side. Figure 3.1b is a screenshot from a simulation. The simulation software that was used is Gazebo [22]. The simulation environment that was used to create the screenshot can be found in [9].

The next sections describe the KoopaCar's hardware and software architecture. Despite the KoopaCar being developed as a prototype previous to thesis, more development was necessary to fulfill the additional requirements for the subject system. As mentioned before, the system needs to rely on more than one ML model and still be fairly simple to reduce any overhead to evaluate ML management tools in Multi Machine Learning Model Systems.

3.2 Hardware Architecture

The hardware used for the KoopaCar is a Turtlebot3. The Figure 3.2 gives an overview of the hardware. The Turtlebot is a small drivable robot with multiple layers. At the core is a microcontroller that is connected to multiple sensors and a motor unit. In general, the Turtlebot can be upgraded by adding additional sensors [23]. The Turtlebot that is used for the KoopaCar has a two-wheel drive, a Raspberry Pi camera [24], and a LDS-01 LiDAR sensor [25]. The usage of cameras and LiDAR sensors in ADS is very common since both sensors complement each other well [26], but some systems use additional sensors to obtain more information to make more reliable assumptions and predictions on their surrounding. The KoopaCar only relies on images taken from the camera and scans returned from the LiDAR. Both sensors are described more detailed in the following.



(a) Back of the KoopaCar (b) Front of the KoopaCar

Figure 3.2: The figures show images of the KoopaCar. The KoopaCar consists of multiple layers. On top of it is the LDS-01 LiDAR. The layer below holds the computation unit. At its core is a Raspberry Pi [27]. On the bottom layer, the motor and wheels are mounted. The red markings in the images indicate the position of the most important components of the KoopaCar.

The Raspberry Pi camera is a simple camera module. The small module is mounted on the top front of the Turtlebot. It can be used to take still images or transmit a video stream. The camera can be accessed via APIs using Python.

A LiDAR sensor is used to collect distance measurements of the sensors surrounding using light. It sends out light beams in every direction. The sensor detects the scattered light that returns to the sensor. Using the time it takes for the light to return, the sensor then calculates the distance the beam traveled. The results are commonly known as scan or LiDAR scan. The LDS-01 is a 2d LiDAR sensor. This means that the sensor returns a 2-dimensional point cloud. With the knowledge of the distance for every beam, one can calculate coordinates in the plane the LiDAR beams travel in.

3.3 Software Architecture

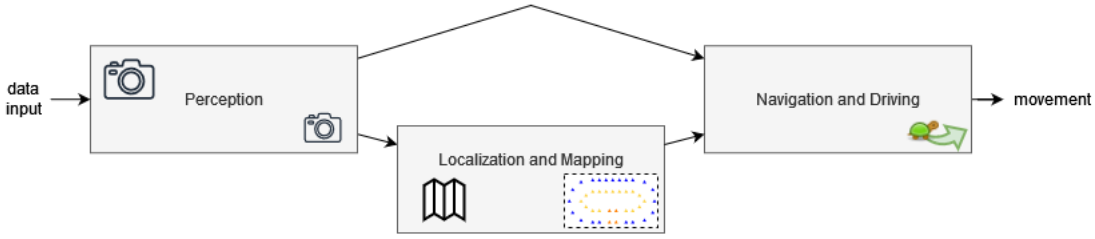


Figure 3.3: High level description of the KoopaCar's software architecture. The architecture consists of three modules, the Perception module, the Localization and Mapping module, and the Navigation and Driving module, which are represented by the three boxes in the figure. The black arrows indicate the flow of data and information between the three modules. The data input for the Perception module originates from the sensors that are used. The figure states that the output of the system is movement. What is meant with this is that what can be perceived as a result of the internal computation is the movement of the KoopaCar, and thus it is shown as so-called output.

This section describes the software architecture. The first part of this section will be a high level description. The later parts contain a more in depth description of the individual modules.
On a higher level of abstraction, the system architecture can be split into three part. There is a Perception module, a Localization and Mapping module, and a Navigation and Driving module, as can be seen in Figure 3.3. The perception module is used to detect objects in the immediate surrounding of the KoopaCar. Using the information gathered in the perception module, the Localization and Mapping module creates a map and localizes the KoopaCar in it. Lastly, the

Navigation and Driving module uses both the map and the information on the immediate surrounding to navigate the KoopaCar through the racetrack. It interacts with the motor to control speed and orientation. This thesis focuses on ML models and their interaction. Since no other module but the perception module uses ML models, it has the most relevance for this thesis. The implementation of the KoopaCar can be found in [8].

3.3.1 Perception Module

The perception module works with ROS 2 nodes and topics. The concept behind both being modularizing code and exchanging a continuous flow of data. A longer description of the concepts can be found in chapter 2. A graph of the nodes and topics as well as the data flow can be seen in Figure 3.4.
The perception module takes data from the camera and the LiDAR as input. The data is processed independently for each input using ML models. As can be seen in Figure 3.4 the ML models run parallel to each other. The following contains a step by step description that follows the data flow through the graph of ROS2 nodes.
The camera sends images to the corresponding topics. To this end, the *camera node* uses a Python API to take images. Images are sent to the *image processing node* using the topic */image_raw*. The *image processing node* can be used to edit or process the image. One possibility would be to resize the images. The *image processing node* then publishes the images to the topic */proc_img*.

3.3.1.1 Yolov5 in the KoopaCar

The *camera object detection node* subscribes to this topic and performs object detection on the images. To this end, the node uses the ML model Yolov5 [28]. Yolov5 is the fifth complete version of the model called Yolo [29]. Yolov5 treats the object detection as a single regression problem. This means that a single model is used, which takes images as input and outputs bounding boxes and labels for detected objects. With this approach, the model is faster than comparable models and achieves high accuracy [28]. More information on Yolo in general and Yolov5 can be found in [29] and [28].
The KoopaCar uses an implementation of Yolov5 that is published and maintained by Ultralytics. [30] Ultralytics implemented Yolov5 in PyTorch and provides all necessary scripts to train the model on custom data, evaluate the model performance and export the model to other formats like TensorFlow lite. Yolov5 takes

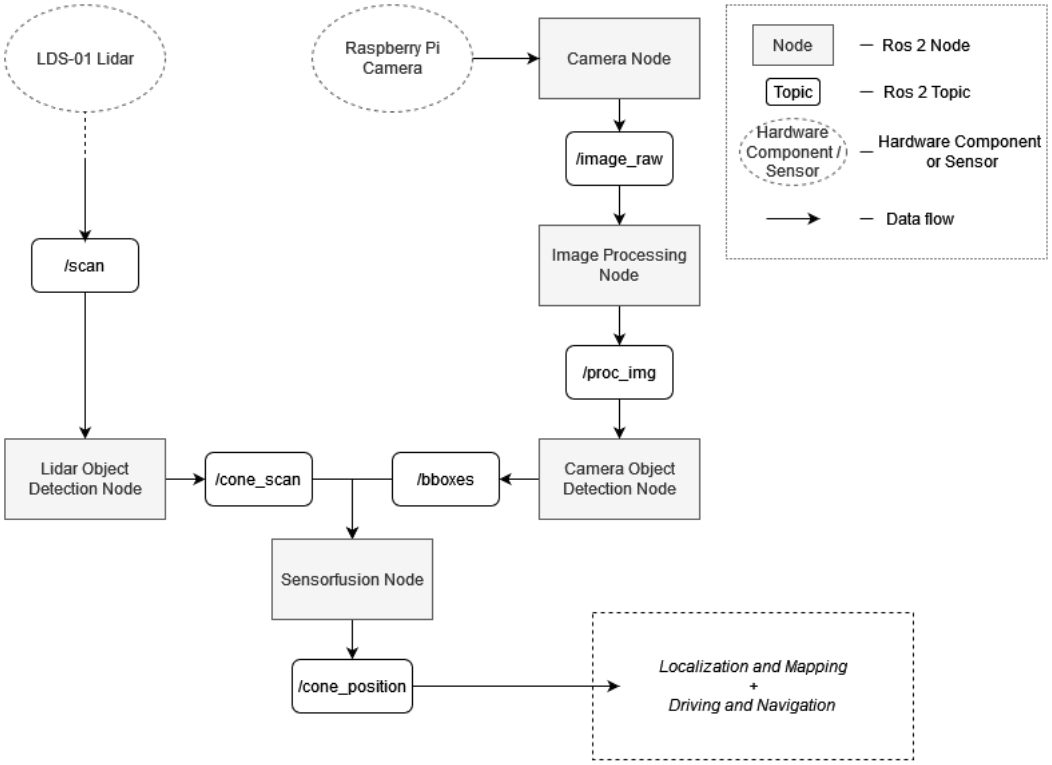


Figure 3.4: Low-level description of the KoopaCar's Perception module. The figure shows the low level system architecture of the perception module. The gray boxes are ROS 2 nodes, and the white boxes with the rounded corners are ROS 2 topics. The back arrows indicate the data flow through the system. The elements with dotted lines put the system into the context of the complete system. The data flow stems from the sensors, which provide the input. The results of the Perception module are passed to the other software modules used in the system, the Localization and Driving module and the Driving and Navigation module.

images and inputs and outputs bounding boxes and labels for the objects it detects. The bounding boxes are published to the topic */bboxes*.

3.3.1.2 LiDAR-CNN

Parallel to this, LiDAR scans are being published to the topic */scan*. Lidar scans are sent as ROS 2 sensor messages that contain a vector of length 360 together with

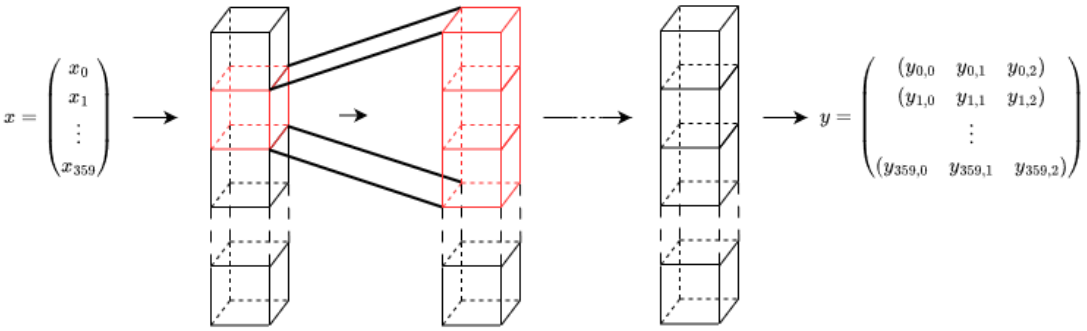


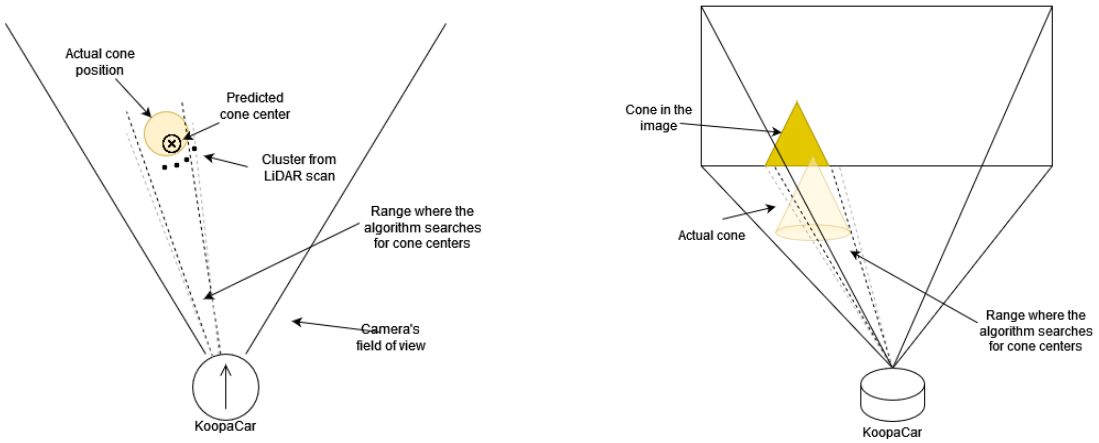
Figure 3.5: Architecture of the CNN with 1d convolutions used to classify LiDAR scans, called LiDAR-CNN. The input for the ML model is a vector with length 360. A series of 1d convolutional layers is used to extract features and perform a semantic segmentation. The 1d convolution is visualized in red. The ML model consists of one input layer, 20 convolutional layers, and one output layer. All layers use *ReLU* as activation function, except from output layer, which uses *Softmax*. The output of the model is a 360x3 matrix. It contains the probability that a point belongs to the three classes, for every element that was input into the model. There are no pooling layers used.

additional information about the LiDAR. Each entry in the vector corresponds to the distance a LiDAR beam traveled. This is done automatically, when using the default Turtlebot3 setup.
The *lidar object detection node* tries to detect cones in the vicinity of the KoopaCar using data provided by the LiDAR. To this end, the node performs a semantic segmentation. This means that an algorithm or ML model is used to assign labels to every point in the input vector. In the case of the KoopaCar the labels are *cone*, *no-cone*, and *outlier*.
The semantic segmentation in the KoopaCar is performed using a CNN. Deciding on a ML model to use in the KoopaCar was a tedious process. There are several complete implementations for ML models that implement object detection or segmentation on LiDAR data, like Yolov5. The problem is that all of these models use LiDAR data with more layers, than the KoopaCar can access. This means that to be able to use one of these models, modifications in the code would have been necessary. The alternative solution was to implement our own CNN. The advantage of this solution is that the model can be fitted to the requirements of the KoopaCar. It is also a contrast to Yolov5 the other ML model in the KoopaCar, which can prove interesting, when we will begin with ML experiments and exper-

imentation management in the later chapters.
The LiDAR-CNN that is used in the KoopaCar is inspired by the 2dLaserNet proposed in [31]. The 2dLaserNet uses 1d convolutions to process LiDAR scans. Following this, the architecture of the LiDAR-CNN was derived. The architecture can be seen in Figure 3.5. The vector of length 360 taken from the LiDAR scan message is used as input. The CNN then performs the semantic segmentation and outputs a 360×3 matrix containing probabilities for the three class labels for every entry in the input vector. The points that have a high probability (above a set threshold) that they belong to the *cone* class are run through a clustering algorithm to identify sets of points that belong to the same cone. The cone center and with that the cone position is being approximated by extending the line between the point from each cone cluster that is closest to the KoopaCar and the KoopaCar's position. The cone positions that were derived are then being published to the topic */cone_scan*.

3.3.1.3 Sensor Fusion

After the *camera object detection node* and the *lidar object detection node* published the predictions on cones in the KoopaCar's vicinity, next both predictions are being fused. To this end, the *Sensor fusion Node* subscribes to the topics */cone_scan* and */bboxes* and performs a late sensor fusion. Late sensor fusion means that the data from the sensors that are being fused was processed prior to the fusion. In the case of the KoopaCar it means that the estimated position of cones is known from processing LiDAR scans and the label of cones is known for cones that are in the camera's field of view. The goal now is to determine the position of the cones in the KoopaCar's field of view by supplementing the data from the */cone_scan*. To this end, an algorithm iterates through the bounding boxes, starting with the one that matches the cone closest to the KoopaCar. How close the cone is to the KoopaCar is derived from the height of the corresponding bounding box. For every bounding box, the algorithm then estimates what angle to the KoopaCar the cone is located. Using the estimated angle, searches for a cone center from the */cone_scan* that lies within a range around it. In the case that more than cone cluster matches the range, the one closest to the KoopaCar is chosen. After matches were found, the cones' positions, which are gathered from cone clusters, and their corresponding labels, which are gathered from the bounding boxes, are published to the topic */cone_position*. The sensor fusion is visualized in Figure 3.6.



(a) Top-down view

(b) 3d view

Figure 3.6: The figures are a visualization of the sensor fusion algorithm used in the KoopaCar Perception Module. The algorithm relies on cone predictions in images and LiDAR scans. It uses the position of a cone in the image to find an approximated angle range, where it searches for a cone prediction from the LiDAR. The range is indicated by the dotted line in both figures. The 3d view shows what the sensor fusion looks like from the point of view of the camera. The dark yellow cone in the back is the cone in the image. The lighter yellow cone is the actual cone that was perceived. The actual cone is visualized by the yellow circle in the top-down view. The top-down view shows what the sensor fusion looks like from the point of view of the LiDAR data. The black dots indicate the LiDAR scan. The circle with the x marks the predicted cone position.

3.4 Comparison to Autonomous Driving Systems (ADS)

This thesis investigates ML management tools and their impact on the development of ML models in Multi Machine Learning Model Systems. The main inspiration for these systems comes from ADS. Therefore, the subject systems used to collect experiences has to be comparable to real Multi Machine Learning Model Systems or more specific ADS. This section outlines differences between the KoopaCar and ADS and also shows that despite those differences the systems are still comparable. The similarities and differences are shown in Table 3.7 and described in depth in the following.

Similarities	Differences
Uses LiDAR and camera to gather information on the surrounding.	The surrounding of the KoopaCar is limited to a racetrack and thus a simplified version of the real world.
Similar system architecture. Processing sensor data, using ML models, and high level sensor fusion.	The KoopaCar can only drive slowly and has limited hardware capabilities for computation.
Fulfills the same or similar tasks, like object detection and sensor fusion.	KoopaCar does not have to follow traffic rules.

Table 3.7: Table outlining the similarities between the KoopaCar and real ADS systems.

3.4.1 Similarities

The first similarity is the sensors that are used in the systems. The KoopaCar relies on LiDAR data to gather distance information of its surrounding and supplements this data using images from a camera. The same sensors are used in real world ADS like Apollo [3]. While it is possible that other sensors are used to supplement the data in addition to the LiDAR and camera, the usage of multiple sensors is common, and they usually always include LiDAR and camera. This is, because both sensors provide different types of information.
Next, the system architecture is very similar as well. This is partially because the usage of different sensors induces a certain structure. The data from the sensors needs to be processed and fused to make useful assumptions on the systems surrounding. The late fusion that is performed in the KoopaCar can be found in Apollo as well [3].
The last similarity is that the KoopaCar fulfills similar tasks as real world ADS do. What is meant with this is that after the environment was perceived, similar decisions have to be made to plan a route and trajectory to follow.

3.4.2 Differences

The main difference between the KoopaCar and other ADS is the application environment of the system. The environment in which the KoopaCar operate is limited to a racetrack. Additionally, it can be assumed that there are no other contestants or obstacles on the track. Thus, the KoopaCar only needs to perceive the boundaries of the track. ADS like Apollo, operate in the real world and have to manage the normal traffic. Other traffic participants are bicycles, pedestrians,

and other vehicles.
Another difference is that the KoopaCar can only drive at low velocities. As a result, there is no way in which people in the KoopaCar's surrounding can be injured. This means that there are fewer concerns for safety.
While the KoopaCar fulfills similar tasks to real ADS, the tasks are not the same. The main difference is that the KoopaCar does not have to follow traffic rules. This makes the decisions that have to be made simpler.

It can be seen that the KoopaCar is in many ways similar to real ADS. The difference is that the KoopaCar is simplified in many ways. This means that the implementation and maintenance of the system is easier. Despite the system being less complicated, it is similar enough to real ADS to ensure that the results of this thesis can be transferred.

4 Machine Learning Experiments in Multi Machine Learning Model Systems

This section investigates how ML experiments work in Multi Machine Learning Model Systems. The goal is to outline the differences to normal ML experiments to gather expectations for features of ML management tools. Section 2.4 outlined the structure of Multi Machine Learning Model Systems. It stated that there are three basic structures into which any Multi Machine Learning Model System can be broken down. The first structure are ML models in a sequential order, and the second and third structure are parallel models that produce the input for another ML model or different software component.
Currently, it is common practice to optimize each model in a Multi Machine Learning Model System individually. The underlying assumption is, that if the loss of each component in a system is minimized, then the overall loss will be minimized as well. This section investigates, whether this approach is optimal based on the structures of the Multi Machine Learning Model System. The first sections will introduce experiment or training patterns. A pattern describes ways the ML model pipeline with parallel structured models can be optimized. The later part of this chapter will focus, on the implementation of such patterns or experiments in the context of the subject system.

The KoopaCar, which is the subject system used in this thesis, is a simplified version of an ADS. It only uses two ML models that are structured parallel. As a result, we will only be able to investigate experimentation and training patterns for parallel ordered ML model pipelines. More precisely, we will only be able to investigate patterns for two parallel models whose result is fused using a software component that is not a ML model to produce a combined output.
The idea behind a training pattern is, that it describes the flow of data through the pipeline as well as the loss that is used to optimize ML models during training. Figure 4.1 shows the type of ML model pipeline that will be used to describe and investigate different patterns. It also highlights the data flow and loss at several

points in the system. The data flow is shown by the yellow lines in the figure. The loss can be calculated after every component, ML model or otherwise. To this end, the output of the component will be compared to an expected result or ground truth.
The different patterns utilize the data flow and loss differently to optimize the models. The following sections will introduce the patterns that were considered as possible approaches for the optimization of a ML model pipeline. Each section will outline the pattern and give suggestions for the implementation.

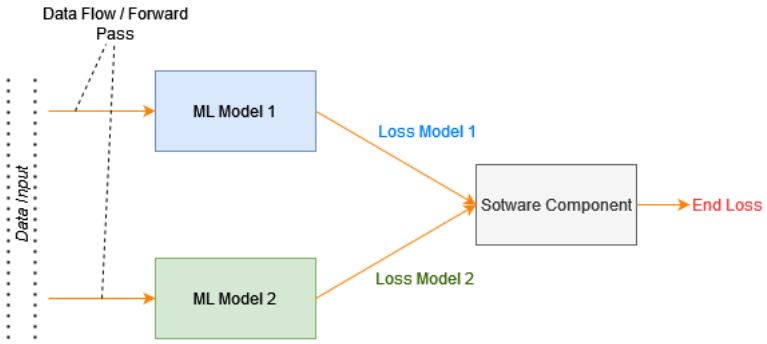


Figure 4.1: Data flow through the ML pipeline on the forward pass. The yellow arrows indicate the flow of data. The boxes are ML models or other software components. The figure shows the loss that is available at different points in the pipeline.

4.1 Experiment Pattern 1 – Individual Model Training

When using the first pattern, each ML model is trained and optimized separately. The idea is that by minimizing the error, each model produces, the error in the pipeline's output is minimized as well. Individual Model Training is the approach for training that is most commonly used. Each model is trained by first passing the data through the network on the forward pass, and then updating the parameters that define the models by calculating the loss at the end of the forward pass and propagating this loss through the model on the backward pass. This is visualized in Figure 4.2.
For this training pattern to work, a data set for every model with the inputs and expected outputs in necessary. Using this data, it is possible to calculate the loss for every ML model individually and update the corresponding model.

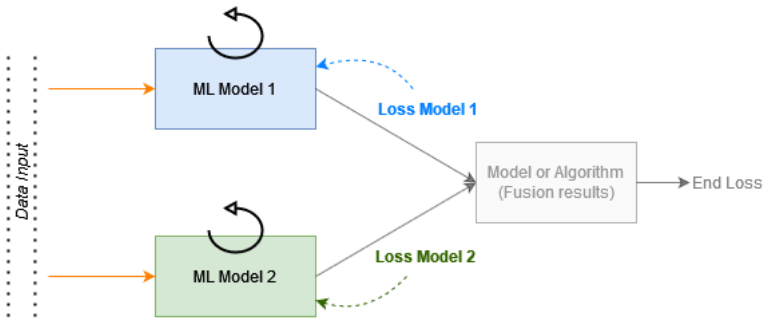


Figure 4.2: Training and optimizing a pipeline using Individual Model Training. The data is passed through every ML model. The loss for every model is calculated and used to update the corresponding model. The rest of the pipeline is not necessary for training.

On a software scale, the first training pattern is implemented by creating a training script for every ML model that is being used. Since this is common practice, models like Yolov5 are deployed with a corresponding training script.
A possible disadvantage of this training pattern is, that it optimizes the individual ML models further than it would have been necessary. In the case of the KoopaCar, the Yolov5 model optimizes the positions of the bounding boxes as well as the labels. Since the labels, are the most important information for the result of the pipeline, the accuracy of the bounding boxes has less impact on the end result and does not have to be full optimized to achieve the same end result.

4.2 Experiment Pattern 2 – Partial End Loss Training

The second pattern changes that not all ML models are trained individually, as in the first pattern. Instead, a part of the ML models is updated using the end loss. The rest of the pipeline will be trained and optimized independently with its individual loss. This means for the corresponding model, normal experiments will be run, as in Individual Model Training. Figure 4.3 visualizes the pattern.
In this case, the necessary data set becomes more complicated. First, all the inputs are needed. Next, the expected outputs of the ML model that is trained independently are needed. Lastly, the expected output of the ML model pipeline is needed as well. Using all this information, it is possible to calculate the losses for the individual models and the end loss.

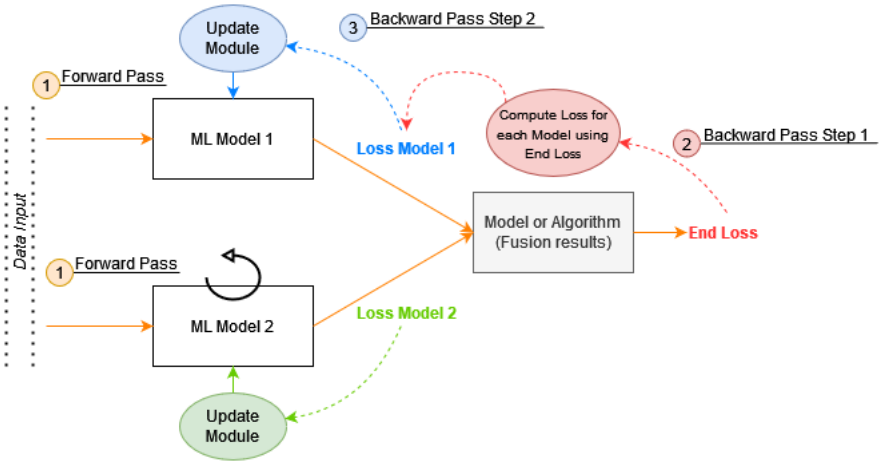


Figure 4.3: Training and optimizing a pipeline using training Partial End Loss Training. One of the ML models is updated using the end loss. To this end, the forward pass is extended and the end loss calculated. Next, the loss needs to be traced back to the ML model. In this step, the loss for model 1 is calculated. This is indicated in red and blue. The other model is trained individually, this is indicated in green.

To be able to implement Partial End Loss Training the training algorithms used to train the ML models that are now being trained using the end loss need to be split into their forward and backward passes. More on this will follow in 4.3. The rest of the ML models can be trained using an independent training script.
The advantage of this training pattern, compared to the first pattern is, that the ML models that are updated using the end loss are not optimized further than necessary. Another reason for using this approach could be, that the end loss does not include the information needed to update the models. This means, that it might be possible that the end prediction does not include the complete output of the ML model and thus the loss is not representative of the model's performance.

4.3 Experiment Pattern 3 – Simultaneous End Loss Training

In the third pattern, the whole pipeline is being trained as one. This means that the loss at the end of the pipeline is being used to optimize the models, no matter their position in the pipeline. Figure 4.4 visualizes the pattern. The data is passed

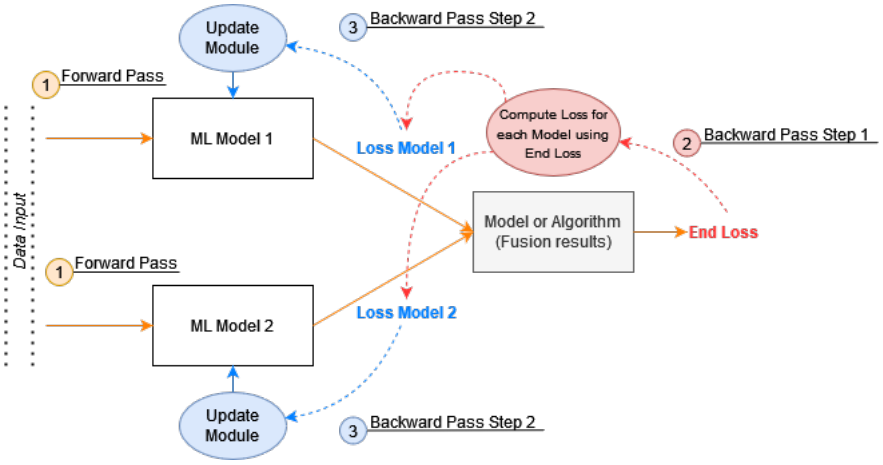


Figure 4.4: Visualization of the training process for an ML model pipeline using Simultaneous End Loss Training. In this figure both ML models are being updated using the end loss. The data is passed through the whole pipeline, indicated in yellow. After the end loss is calculated and processed, which is indicated in red, the ML models are updated. This is indicated in blue.

through the complete pipeline and the loss calculated at the end. This loss is then being used to update the ML models. In the case of the KoopaCar, the loss needs to be split into a loss indicating the error in the position of a prediction and a loss indicating the label of the prediction. This process is necessary to use the correct loss to update the ML models.
The information that is necessary for this approach to work is the input data and the expected output of the ML model pipeline. Using this, the deviation between the predicted and expected output can be calculated.
To implement this pattern, the training algorithms of the ML models need to be split into the respective forward and backward passes. In each epoch during training, first the batch of data samples would be passed through the ML model pipeline and with this through the ML models. This will be called forward pass on the scale of the pipeline. Next, the loss at the end of the pipeline is calculated. This loss is then being propagated back through the system. This means that the end loss is being traced back to each model in a way, that the loss that was induced by the corresponding model becomes apparent. Next, the models are being updated by running the backward pass with the corresponding loss, that was measured at the end of the pipeline.
The advantage of this training pattern is that, with most certainty, the end result of the pipeline is being optimized, and the ML models are not optimized further than necessary. The disadvantage is that it is uncommon to train ML models this

way. Consequently, there is no easy solution to implement Simultaneous End Loss Training. Another problem is, that during the fusion of the ML models predictions information can be lost. This information can have no further impact on the training of the ML model. It is unclear, if and how the end loss is representative to update the ML models.
It is left to determined, how using the end loss affects the training progress of the ML models.

4.4 Experiment Pattern 4 – Alternating End Loss Training

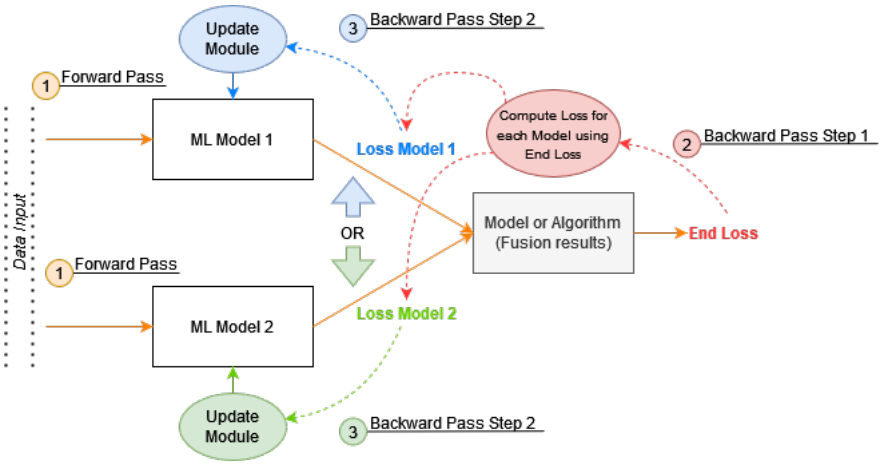


Figure 4.5: Visualization of the training process for an ML model pipeline using Alternating End Loss Training. Here the data is again passed through the whole pipeline, indicated in yellow. Then the end loss is being calculated and processed, as can be seen in the red markings. After this, one of the two ML models is updated.

The fourth pattern is very similar to the third pattern. But instead of updating both models in every step, only one model will be updated. Figure 4.5 visualizes the process involved in training. The process in running such experiments is the same as for the second. The main difference is that after the end loss was calculated, code logic decides which model will be optimized. The motivation behind this pattern is, that it is possible that one model performs better than the other. To optimize the end loss it might not be necessary to train both models in every iteration but only the one, that has a lower accuracy. As a result, fewer calculations might be performed.

There could also be variations in which both models are updated until the loss for a model lies under a predefined threshold. Once the model's loss is under the threshold, it will not be updated in the next training iterations. This approach could be implemented similar to early stopping approaches in ML model training. The training continues until all models are under the threshold.
The data that is necessary for this pattern to work, is the same as for the second training pattern. The general implementation is the same as for the second pattern and comes with the same challenges as well. The only addition is the code logic that determines which model is being updated.
An advantage of the fourth training pattern could be that each iteration of the training algorithm becomes more efficient. This is because fewer operations are executed in every iteration. On the other hand, it is not clear, if this results in more iterations being necessary to achieve the same results as for the second training pattern. This means, that the total efficiency stays the same.

4.5 ML experiments on the subject system

To investigate how the patterns presented in the previous sections perform, experiments involving the possible implementations for the different patterns were run. The structural example, used to motivate and visualize the patterns, was based on the ML model pipeline in the KoopaCar. The system architecture for the KoopaCar was described in depth in section 3.3, a short summary follows. Images taken by the cameras are processed using Yolov5 while LiDAR scans are being processed by a Convolutional Neural Network, that was implemented using TensorFlow Keras. The results of both models are fused in the sensor fusion. The result of the pipeline are positions of cones relative to the KoopaCar and the cone classes for every cone. The ML model pipeline as it was just described can be seen in Figure 4.6. This pipeline is part of the system of nodes that was shown in Figure 3.4. The first experiment will be to optimize the model used for processing LiDAR scans. The second experiment was to optimize Yolov5, which was trained on a custom data set. Both of these experiments can be matched to the first pattern, described in section 4.1.
The third experiment is to train the ML model pipeline as one system, following the third pattern. While the first two experiments will be fairly trivial, the implementation and results of the third one are uncertain.
Therefore, the focus in the first two experiments will be on running and evaluating the experiments, while the focus for the third is the implementation.
The ML experiments can be found in [8] on the branch `tools-mlflow`.

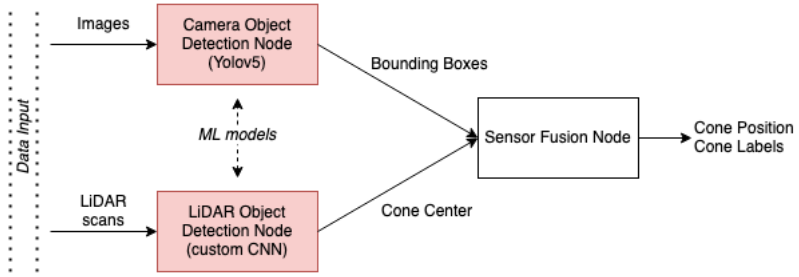


Figure 4.6: ML model pipeline in the subject system (KoopaCar)

4.5.1 Building and Optimizing ML Models using Individual Model Training

The first two experiments will focus on optimizing the ML models in the KoopaCar using the first experimentation or training pattern, which is Individual Model Training. This means, that the ML model pipeline is not used. Instead, we will use a data set for each model and run experiments separately. The datasets contain the input and the expected outputs of the ML models. Using the outputs, the deviation between expectations and predictions is used to calculate the loss.

4.5.1.1 Optimizing the LiDAR-CNN

The model used to process the LiDAR scans is a convolutional neural network (CNN). The model performs a semantic segmentation on the LiDAR scans, which are sets of vectors consisting of 360 scalars. One vector corresponds to one scan. The model's output is a vector for every vector input, consisting of probabilities for each class. The result of this segmentation is used to identify clusters of points that belong to the same cone. The cone's position is being approximated by extending the line between the cluster and the KoopaCar. A more detailed description and figures visualizing it can be found in Section 3.3.
The experiments that were performed, tried to optimize the accuracy of the segmentation by tuning the hyperparameters. The hyperparameters that were changed over the course of the experiments are the number of epochs, the batch-size and the training validation split.
The data set that was used consists of 392 scans. The scans were collected in a simulation built in Gazebo [9]. A scan includes the inputs from camera and LiDAR, the expected results for both models, and the correct positions of the cones relative to the KoopaCar. The scans were all taken by placing the KoopaCar in

50 different positions over 8 different simulations. Some scans had to be discarded due to mislabeling, thus resulting in the 392 scans used for training and validation.

In the experiment, the goal was to find a combination of batch-size, epochs and training-validation split that results in short training times and minimizes the loss. The loss function that was used is Cross-Entropy. Cross-Entropy is often used as loss function for classification problem with multiple classes. We also used Mean Squared Error (MSE) for a series of experiments. MSE did not yield any different results, but Cross-Entropy made spikes and overfitting more visible. The following experiments were all run with Cross-Entropy as loss function.

First, we will take a look at how the training validation split affects the training results. We ran a training iteration with a validation-training split of 20/80, meaning 80% of the dataset are used as a training set and 20% are used for validation. In this run, we used a batch-size of 16, and we ran training for 64 epochs. This run will be compared to a training run with a training-validation split of 10/90, batch-size of 16, and 64 epochs. Figure 4.7 shows the training loss and the validation loss over the course of training for the first run, meaning training validation split of 20/80. The x-axis shows the epochs. Figure 4.8 shows the training and validation loss as well but for the second run, meaning a training validation split of 10/90 was used. Using a larger share for the validation set, means that the size of the training set is reduced. In our case, the dataset is already fairly small. Due to this reason, it is advisable to use a training-validation split of 10/90. By comparing the plot, we can see, that the validation loss becomes larger for a larger validation set. When the validation set is too small it becomes more difficult to detect overfitting, but on the other hand is enough data in the training set crucial to make sure the ML model learns the necessary features. In Figure 4.8 the validation loss has less and smaller spikes, than in Figure 4.8. This means, that there is less variance. Both options are equal. The training loss has less variance for a bigger training set, but the validation loss becomes smaller due to the size of the validation set. Going forward, we will use a training validation split of 10/90.

Next, we will try to find a batch-size, that optimizes the training further. To this end, we will run the previous run with a training-validation split of 10/90 and 64 epochs again. Once we will use a batch-size of 8 and once a batch size of 32. Figure 4.9 shows the training and validation loss per epoch for the run with batch-size 32. Figure 4.10 shows the plot of the training and validation loss for batch-size 8. The run with batch-size 32 showed fewer spikes and irregularities. The curve, especially at the end of training, is smoother for a batch-size of 8. Since the data

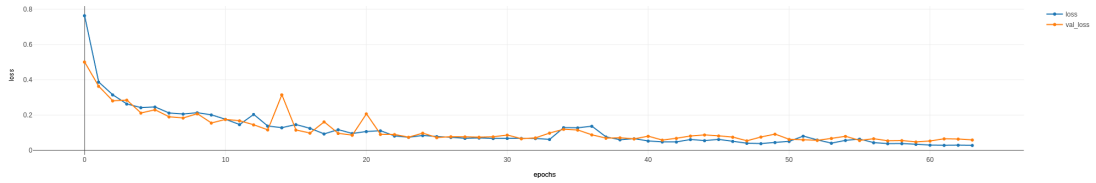


Figure 4.7: LiDAR-CNN training results. The training was run with a training validation set of 20/80, a batch-size of 16 and was run for 64 epochs. The validation loss is about as high as the training loss over the course of the 64 epochs. There are some small spikes in the validation loss. The training loss shows minor irregularities.

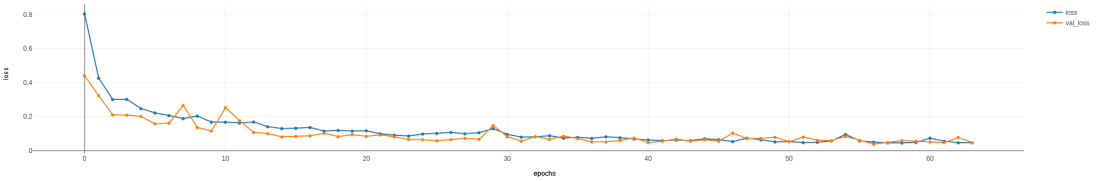


Figure 4.8: LiDAR-CNN training results. The training was run with a training validation set of 10/90, a batch-size of 16 and was run for 64 epochs. The validation loss is lower than the training loss in the beginning, but is about the same towards the end of training. There are a few small spikes in the validation loss. The training loss does not show any irregularities.

only consists of vectors of size 360 there should be no problem with the memory usage of the machine used for training. This means for the last experiments we will be using a batch-size of 32.

Lastly, we want to see at what point overfitting occurs, to be able to achieve the best training results possible. To this end, we ran the previous experiment again with an increased number of epochs. We used a batch-size of 32 and a training-validation split of 10/90. Training ran for 512 epochs. A plot of the training and validation loss can be seen in Figure 4.11. The curve shows, that with increasing number of epochs the number of spikes rises significantly. Additionally, after around 100 to 150 epochs, the validation loss becomes higher than the training loss and starts to diverge. The end result of this series of experiments is, that due to the limited amount of data, a training-validation split of 10/90 yields better

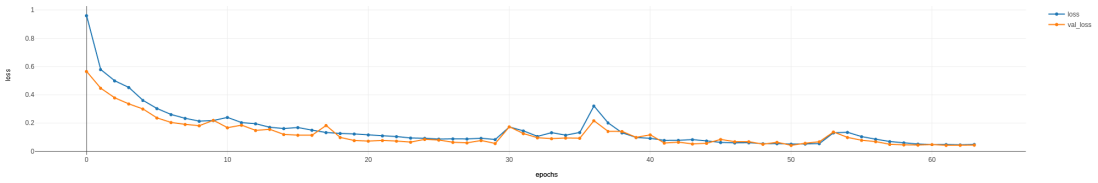


Figure 4.9: Lidar-CNN training results. The training was run with a training validation set of 10/90, a batch-size of 32 and was run for 64 epochs. The validation loss has a few notable spikes. The training loss has only one notable spike.

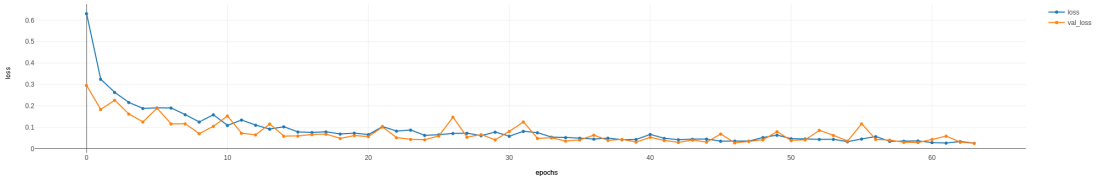


Figure 4.10: Lidar-CNN training results. The training was run with a training validation set of 10/90, a batch-size of 8 and was run for 64 epochs. The training loss does not show any irregularities. The validation loss has a high variance, meaning there are several small spikes.

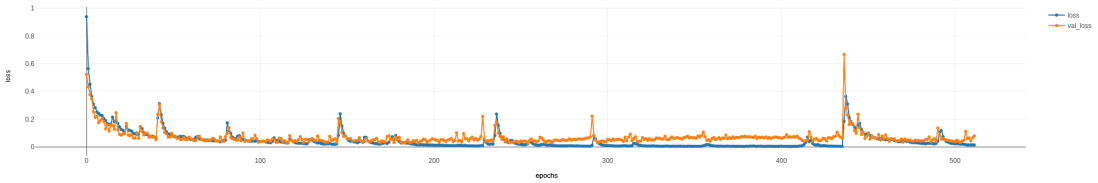


Figure 4.11: LiDAR-CNN training results. The training was run with a training validation set of 10/90, a batch-size of 32 and was run for 512 epochs. The validation and training loss converges for the 100 epochs. After this, it is notable, that the validation loss begins to diverge. There are several spikes on both the training and the validation set. The spikes become worse the longer the training lasts.

training results. It is feasible to use a batch-size of 32. This does not impact the efficiency of the training negatively, but rather makes the training loss spike

less and converge better. Training should not run any longer than 128 epochs. After that, overfitting occurs, which means, that the ML model performs worse on previously unseen data.

4.5.1.2 Optimizing Yolov5 on Custom Dataset

For training and optimizing Yolov5, we will only take a look at the impact of the batch-size and number of epochs. Since Yolov5 is trained on image data and is in general a significantly more complex ML model compared to the LiDAR-CNN from the previous experiment, this time we will have to focus more on finding a set of hyperparameters that make the training process efficient while optimizing the ML model's performance.
While evaluating the model's performance, we will take an especially close look at the classification error. Yolov5 provides for both training and validation set three different error metrics, the *box_loss*, the *cls_loss*, and the *obj_loss*. The *box_loss* is the loss that describes the error regarding the position of bounding boxes. It is calculated using bonding box regression. The *obj_loss* describes the error on whether an object exists at a position. Last, the *cls_loss* is the loss describing the error in classification. This means, the deviation in the assigned labels. Since Yolov5 is mainly used to determine a cones class, this is the loss that will be mainly used in the following experiments.

The first series of experiments will evaluate the effect of the batch-size on training. We created two runs. The first run used a batch-size of 8 and the second one of 32. The number of epochs is 128 for both runs. The rest of the hyperparameters are the default Yolov5 parameters.
Since the training and validation *cls_loss* did not reveal any distinctive features, the other losses were also used to evaluate the influence of the batch-size on training Yolov5. Figure 4.12 shows the different losses over the course of training for a batch-size of 8. Figure 4.13 shows the same metrics for a batch-size of 32. It seems, that a smaller batch-size yields better results, since there are fewer spikes. There is only one large spike in the validation *obj_loss*. The loss for a batch-size of 8 shows fewer irregularities and thus a smoother curve. The problem is, that the ML model trains faster with a batch-size of 32. Training the model with a batch-size of 32 takes 5.2 minutes, and training with a batch-size of 8 6.7 minutes for 128 epochs. Therefore, we will use a batch-size of 16 as a compromise.

With the results from the last experiment, we will now evaluate at what point, the model begins to overfit, so that the number of epochs that yields the lowest

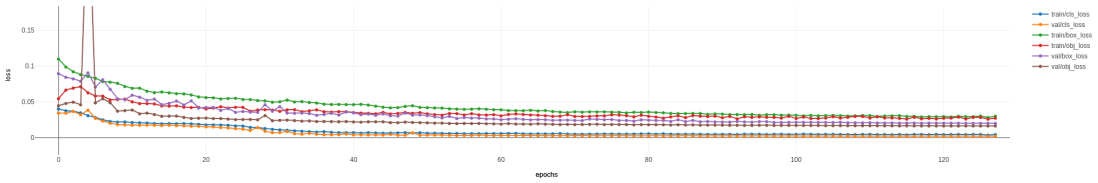


Figure 4.12: Yolov5 training results. Training was run using a batch-size of 8 and lasted for 128 epochs. The plot shows the training and validation loss for the losses Yolov5 provides. Apart from an immense spike in the validation obj_loss around epoch 5 the losses all converge smoothly.

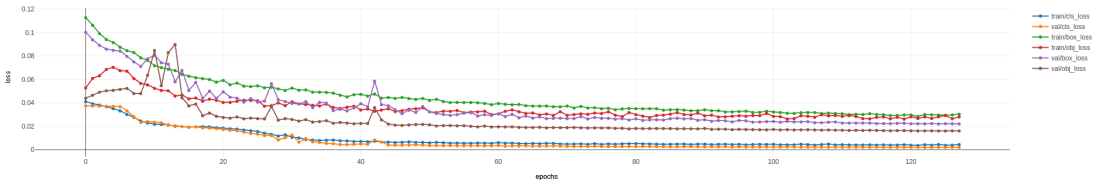


Figure 4.13: Yolov5 training results. Training was run using a batch-size of 32 and lasted for 128 epochs. There are several spikes in the training and validation obj_loss and the validation box_loss. The rest of the losses converge smoothly.

loss without overfitting can be chosen. We are using a batch-size of 16 and run the training for 512 epochs. There were no indicators for overfitting. Thus, we repeated the training with the same hyperparameters and doubled the number of epochs. Figure 4.14 shows the training results for an experiment with batch-size 16 and 512 epochs. There are still no indicators for overfitting. Despite that, it seems that the training *cls_loss* converges towards 0.03. After around 200 epochs the loss is slightly higher than 0.03 and after 512 epochs the loss is still close to 0.03. The same is true for the other metrics that can be seen in the plot in Figure 4.14.
The spikes in the beginning of the training indicate, that it might be advisable to use a batch-size of 8 instead of 16.

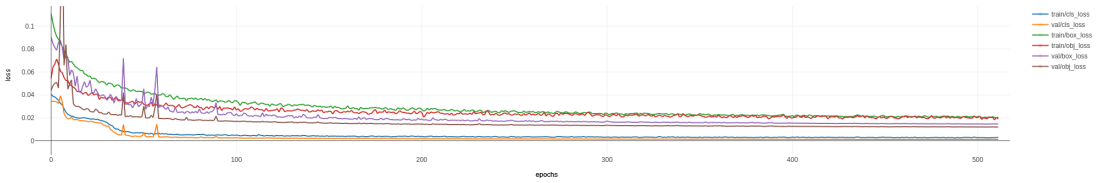


Figure 4.14: Yolov5 training results. Training was run using a batch-size of 16 and lasted for 512 epochs. There are several spikes in all the losses in the beginning of the training until about epoch 50 to 60. From there on, the training converges. After around 200 epochs, there is no progress in training.

4.5.1.3 Result of Individual Model Training

Following the optimization of the individual models, it is left to determine the effect on the end result of the pipeline. To this end, it is necessary to implement loss functions. A suggestion would be to use an approach similar to bounding box regression to determine the deviation of the position in the prediction as *pos_loss*. The deviation in the labels or classification could be measured using the Cross-Entropy loss function.
As can be seen in Section 4.5.2 the implementation of these loss functions is unfinished. The problem is that the training and optimization of the ML models using Individual Model Training, was done using the individual training scripts for each ML model. To evaluate the end loss over the epochs of training, a script combining the different training algorithms would have been necessary. The implementation for this was not finished due to problems in the implementation problems. The problems are highlighted in Section 4.5.2.
Another problem is the data set. The set contains the positions of every cone that is in the simulation. To be able to calculate the loss, the expected output of the pipeline needs to be used. The problem is that the KoopaCar does not perceive every cone in the environment. The data set should be refined further to contain the expected output of the ML model pipeline.

4.5.2 Training the Complete Pipeline using Simultaneous End Loss Training

Training the complete pipeline is a training pattern proposed in section 4.3. This section will cover efforts that were made to implement the training pattern and

run experiments.
To start off, there were two general approaches to implement pattern 43 The first one, that was also proposed in section 4.3, is to reorganize all the forward and backward passes in a single loop. An alternative implementation would be to use the training loop of one of the models and integrate the other models into this loop. To this end, the training loop of the first model is interrupted between the forward and backward pass and additional code to train the other models added.

Implementation Approach 1. The advantage of the first implementation is that the solutions scales for an increasing number of ML models. It is not trivial, which model's algorithm should be chosen to include the pipeline training, when implementing the second approach. Additionally, the training algorithm could include a lot of unnecessary dependencies that create an overhead in training. Restructuring the training in an independent loop, treats all models, that are part of the pipeline, equally. The problem with the first approach is that most ML model training algorithms are not meant to be taken apart and split into their respective passes. Especially, when using a ML framework and not implementing the ML model from scratch, it might not be possible to run the respective passes separately with code logic influencing the backward pass in between the passes.
While implementing the pipeline training using the first approach, several problems arose. The main problem was, that the Yolov5 model, that was used in the KoopaCar has a lot of dependencies. Due to the lack of knowledge of the different Yolov5 model classes and the overwhelming amount of errors, the approach was eventually abandoned. The main issue was to create a Yolov5 model object that is intact. It became apparent that the object was corrupt, when we were not able to predict bounding boxes using the object. The output, the model returned, was unexpected. It is possible that one or multiple layers were missing, or other steps had to be added to create an intact model.

Implementation Approach 2. The second approach has the advantage, that at least one model's training algorithm can be kept intact. This means, that it was possible to avoid taking apart the Yolov5 training and instantiate a new Yolov5 model. Thus, it was possible to find a workaround for the main problems with the first implementation approach.
In the second approach, the Yolov5 training algorithm was used as it was. In between the Yolov5 forward and backward pass, new code was integrated that ran the forward and backward pass for the LiDAR-CNN.
This solution solved problems that lead us to abandon the first approach, but in doing so showed different problems, that already persisted before. The problem is

that it does not seem to be possible to train a TensorFlow Keras model and use a loss to update the weights, that was not calculated using the model internal y and y_{pred}. What this means is that to train the model and update the weight in each iteration, the loss function is being called. It is possible to implement a custom loss function, but the problem is that the loss function needs to be differentiable to provide gradients to update the model's weights. It does not seem to be possible to inject a loss, that was calculated outside the Keras training loop, and use this loss to update the weights.
A solution would be to implement the LiDAR-CNN from scratch with the help of a MLframework. There might be other solutions, but no feasible solution was found over the course of this thesis.

Apart from the problems in the implementation process mentioned above, it would have been necessary to calculate an end loss and use this loss to update the ML models in the pipeline.
It is uncertain how this could have been done in the context of the subject system. It was mentioned in a previous section that the relationship between the ML models and the output of the ML model pipeline is not always trivial. Section 4.3 also stated, that it is unclear what the best way is to update the ML models using the end loss. Using the KoopaCar as an example, this problem can be phrased more precisely.
In the case of the KoopaCar the output of the ML model pipeline provides the cones' positions and the cones' labels. The positions of the prediction can be compared against the one of the ground truth using an approach similar to bounding box regression. In this approach, the difference between the prediction and the truth is being calculated, by comparing the respective area and the overlap of prediction and truth. The difference is then being normalized to form the loss. The labels are being processed by applying Cross-Entropy. This means, that there are two losses at the end of the pipeline. The first is the *end/pos_loss* and the second is the *end/cls_loss*. One problem is that the data set that was used does not contain the expected output. Instead, it contains the true position for every cone in the simulation.
Another problem is that during the sensor fusion, a lot of information on bounding boxes and the LiDAR scans is lost. The only information that is still there at the end of the pipeline is on cones and Yolov5 predictions that were matched in the sensor fusion. It is uncertain how the training is affected, if the limited information on LiDAR scans and bounding box prediction is used to update the ML models. In addition, it is not clear, how the information on the deviation of the cone's position relates to the results of the semantic segmentation that was performed by the LiDAR-CNN.

In general, it became apparent, that the proposed way to approach training ML models in Multi Machine Learning Model Systems, comes with a lot of challenges during implementation. The problem is that it is not common to split the training the way it would be necessary to implement training patterns proposed in this thesis. ML model frameworks like TensorFlow Keras make it easy to implement simple models. If practitioners or researchers want to use training algorithms, they will have to rely on other methods to implement the ML model. It is not clear if this problem persists with other ML model frameworks.
More time for research and development into the training patterns is necessary to make meaningful statements on their benefits for Multi Machine Learning Model Systems.

5 Machine Learning Management Tools and Multi Machine Learning Model Systems

The goal of this thesis is to evaluate experimentation management tools in Multi Machine Learning Model Systems. Research question 2 investigates how existing experiment management tools can improve the development of Multi Machine Learning Model Systems. Research question 3 then investigates how the tools could be further developed to improve the development. To this end, experiment management tools were applied in ML experiments on the subject system, the KoopaCar described in Chapter 3, to collect experiences and begin to answer the two research questions.
Therefore, we will first take a look at available tools and select a tool to evaluate in this chapter. Then, the tool will be introduced, and its main features summarized to get an overview in advance. Later there will be a report on experiences that were made using the tool in the context of the subject system. Note, as a basis for the evaluation, we will use the experiments proposed and described in chapter 4.

5.1 Tool Selection

To begin, we will select a tool to be evaluated. To this end, we will formulate a search query based on important keywords. To limit the results from the initial search, inclusion and exclusion criteria will be applied to the search results. From the remaining results, an experimentation management tool is chosen that holds a wide popularity. The motivation for this kind of prioritization is that due to the limited time for this thesis, it is not possible to analyze a wide array of experimentation management tools. By prioritizing a popular experimentation management tool, the results of this evaluation will have a wider influence on practitioners.

5.1.1 Sources and Search Query

Google will be used as the primary source for ML management tools. Important keywords that will be used in the search queries are listed in Table 5.1.

Machine Learning	Experimentation Management Tools	Versioning
Multiple Model Machine Leaning System	Asset Management Tool	Reusable
Multi Machine Learning Model System	Tracking Tool	Reproducibility

Table 5.1: Keywords for ML management tool selection

From the keywords, the following search query was derived:

Q: *((Machine Learning) OR (Multiple Model Machine Learning) AND (((Experimentation OR Asset Management) OR Tracking) Tools) AND (Versioning OR Reusable OR Reproducibility*

Note that the terms *Multiple Model Machine learning* and *Multi Machine Learning Model Systems* did not result in search results for tools that support Multi Machine Learning Model Systems. This aligns with the first literature research into the topic. There are no tools that specialize for the usage in the context of Multi Machine Learning Model Systems. In addition, the keyword *Multi Machine Learning Model Systems* limited the search result, which is why it was not used in the query.
Among the Google search results was a blog post by Neptune AI.[32] The blog post gave a good overview over ML management tools and their features, and was used in addition to the rest of the search results.

5.1.2 Selection Criteria

The following selection criteria were used to select ML model tools for evaluation. The criteria are split into inclusion and exclusion criteria. If a tool meets the inclusion criteria, it is eligible for selection. If a tool conflicts with any of the exclusion criteria, it is being discarded from the pool of tools to be evaluated.
The selection criteria used for the tool selection in this thesis are inspired by the selection criteria in [6].

5.1.2.1 Inclusion Criteria

The inclusion criteria are:

IC_1 Primary purpose is experimentation management or whole lifecycle

IC_2 Usage with Git, direct integration or project structure that works with Git

IC_3 Flexible project structure

The idea behind IC_1 is that the primary focus of this thesis is on experimentation management. Thus, we will need to evaluate tools that support common features for experimentation management like logging and visualization. To this end, we will focus on experimentation management tools or tools that support the whole life cycle of a ML model or Machine Learning Model System. The latter usually implement experimentation management features in addition to the DevOps features that are common for such tools.
The idea behind IC_2 is that Git is essential in modern software development. We want to be able to use Git during the development of the subject system. The motivation for this criterion is that, there are asset management tools that do not rely solely on Git. The problem with git is that it is not powerful enough to support every asset and the amount of data used for ML experiments.
A flexible project structure, as it was named in IC_3, is essential for the usage with the subject system and most other Multi Machine Learning Model Systems. In the case of the subject system, the system has a project structure that is induced by ROS. During the selection process of the ML management tool, it is not clear how ML experiments fit into this structure. Thus, to be able to explore all possibilities and options, it is important that the experimentation management tool allows a user defined project structure.

5.1.2.2 Exclusion Criteria

The exclusion criteria are:

EC_1 lack of documentation (in English)

EC_2 inaccessible without a license

EC_3 framework proposed in paper but not actually implemented

The exclusion criteria focus mainly on the accessibility of the tool and the documentation.
It is important that the tool is documented in English. Otherwise, the tool cannot

be used by most practitioners. Next, the tool should be available without purchasing a license. ML management tools should be available without barriers.
Next, frameworks that are only proposed in papers and not implemented are of no benefit for the thesis. The idea is to evaluate the benefits of ML management tools, by applying them to the subject system.

5.1.3 Tools

The initial Google search supplemented with the Neptune AI blog post mentioned earlier yielded a total of 18 different machine learning management tools. All tools are listed in the following.

- MLflow
- Kubeflow
- Tensor Board
- Pachyderm
- Clear.ml
- Sacred
- Polyaxon
- Guild.ai
- Neptune AI
- Comet
- Determined.ai
- Datum
- Feature Forge
- DVC by iterative.ai
- Weights and Bias
- Verta
- AWS Sagemaker Studios

After applying the inclusion and exclusion criteria, the last four tools in the list, the ones in the right column, were eliminated. The reasons for this are that DVC is a tool that only implements asset management. The other tools are not available without paid licenses.
While taking a closer look at the tools in question in became apparent that most of the tools implement the same or very similar features. From the first glance, there is no reason to assume that any of the tools might be an especially good or bad fit for Multi Machine Learning Model Systems.

Therefore, the tool that will be evaluated is MLflow. It implements features that most of the other tools implement as well. The features that are presented on the tool's website could be beneficial for ML experiments in Multi Machine Learning Model Systems. Note that one can assume that the experiences and results made with MLFlow are with most certainty transferable to other ML management tools. As mentioned before, due to the limited time available for this thesis, only one tool will be evaluated. In addition to the similarity of features, MLFlow is also a tool that appears in many blog posts from websites in the machine learning community. Combined with the ratings on the tool's GitHub repository, let us assume that the tool is popular.
With this in mind, the next section will outline the approach to evaluating the experimentation management tool that was selected. The last section in this chapter will describe MLFlow in depth and presents the tool's evaluation.

5.2 Tool evaluation

The evaluation of tools in this thesis is only based on experiences. To this end, the tool will first be introduced, which includes a summary of the core features and tool usage. Next, the experiences made while running the experiments that were introduced in 4.5 will be presented. There are some aspects that will be focused on that might be of interest for practitioners or other researchers.
The first aspect is effort. Effort corresponds to the time and the amount of work necessary to start using the tool. It also refers to the amount of additional code or workload necessary to use the tool for an ML experiment.
The next aspect is flexibility. Flexibility refers to the project structure. The structure is often induced by practices related to the field of application. In the case of the subject system, the project structure is induced by the usage of ROS. Flexibility means that the project structure can be arbitrary for the tool to work.
Lastly, there will be a focus on the general features available for the tool that is being evaluated. This will extend the previous summary of the available features with a report on how useful they were when running the experiments on the subject system. A focus will be on whether a feature was especially useful in the context of Multi Machine Learning Model Systems.

5.3 MLFlow

MLFlow is an open source ML management tool designed to improve the development of ML models and Machine Learning Model System. The tool supports asset management, experimentation management, run orchestration, support for deploying, and maintaining models. In the following, we will take a look at key concepts in MLflow and some important features. Later, we will take a look at experiences made with MLflow while working with the subject system.
The experiments and thus the logging can be found in [8] on branch `tools-mlflow`.

5.3.1 Main Concepts of MLflow

There are four main concepts in MLflow. The first is MLflow Tracking, then MLflow Projects, MLflow Models, and last MLflow Registry. Each of those concepts implements different ML management features. The features reach from simple asset management to DevOps features.

The first concept is MLflow tracking. This is also the simplest of the MLflow concepts. With MLflow tracking, practitioners are able to log parameters, code versions, metrics, and output. Logs are created when running machine learning code.
MLflow Tracking is a combination of a Python API and a UI. The API can be used to log different types of artifacts at runtime from the machine learning code. The UI can then be used to organize the logs further and compare experiment runs. It is possible to visualize metrics as a graph.
The UI is hosted to a free port, either on the local machine or by a server or database. There are several possible setups that are described in the MLflow documentation.

The next concept is MLflow Projects. MLflow Projects allow users to package their machine learning code in a project to make it reproducible. To create a project, any directory or git repository can be used. A YAML formatted file is then used to specify the project. This file is called MLproject file. An example of such a file can be seen in Listing 5.2.
In the MLproject file, the user defines the project's name. In the example, the project's name is *koopacar*. Next, the Python environment that is used to execute the project is specified. It is possible to use a virtual environment or a Conda environment. It is also possible to not specify any environment and use

the local environment instead. In the example, a virtual environment was used, which is specified in the file *requirements.yaml*. The *requirements.yaml* file holds a list of packages and package versions that exist in the environment. Following the project name and environment are the entry points. Entry points can be viewed as commands that can be executed in the MLproject. For every entry point, the practitioner needs to specify a command and can define parameters. In the example the entry points are *lidar*, *yolo*, and *main*. *main* is the default entry point that is executed if no entry point is specified. In the example, there are a number of parameters with default values for each of the entry points.

```
name: koopacar

python_env: requirements.yaml

entry_points:
    lidar:
        parameters:
            data_path: { type: string, default: "/home/ubuntu/koopacar-system/data/perception/
                training_data/lidar_03" }
            val_split: { type: float, default: 0.2 }
            epochs: int
            batch_size: int
        command: "python3 src/perception/models/lidar/train.py -d {data_path} -V {val_split} -B {
            batch_size} -e {epochs}"

    yolo:
        parameters:
            model-cfg: { type: string, default: "yolov5n.yaml" }
            epochs: { type: int, default: 64 }
            batch-size: { type: int, default: 16 }
            data: { type: string, default: 'complete_04.yaml' }
        command: "python3 src/perception/models/camera/train.py --data {data} --epochs {epochs} --
            weights '' --cfg {model-cfg} --batch-size {batch-size} --cache"

    main:
        parameters:
            model-cfg: { type: string, default: "yolov5n.yaml" }
            epochs: { type: int, default: 64 }
            batch-size: { type: int, default: 16 }
            data: { type: string, default: 'custom.yaml' }

        command: "python3 src/perception/models/camera/train.py --data {data} --epochs {epochs} --
            weights '' --cfg {model-cfg} --batch-size {batch-size} --cache"
```

Listing 5.2: Example for an MLProject File, `MLproject` from [8] on branch `tools-mlflow`. The file is YAML formatted. It defines the project name, Python enviornment and entry points for the project. The entry points are *lidar*, *yolo* and *main*. For each entry points a series of parameters were defined and a command that is executed. The entry point *main* and *lidar* are the exact same in this example.

An MLproject can be executed either using the MLflow command line tool or the

Python API. Using the Python API makes it possible to organize projects runs and different entry points into a multistep workflow. This feature is what was previously introduced as run orchestration.
The multistep workflow organizes the execution of experiment runs using the entry points from the MLproject file. Using the Python API, the entry points can be started and the status of the processes can be checked. Consequently, the multistep workflow implements run orchestration.

MLflow Models is a format to package ML models. A model can be saved in different flavors. Flavors refer to the API or software that was used to develop the model. An example for such a flavor is PyTorch. MLflow Models can be served and used by downstream applications.

The last concept that was listed is MLflow Registry. MLflow Registry is used to support the full lifecycle of a ML model. An MLflow Model can be registered to use features for the complete ML model's lifecycle. It allows model lineage and versioning. In addition, it also allows sorting models into development stages and transition between them.

Since the focus of the thesis is on ML experiments, the concepts MLflow Tracking and MLflow Projects will be the focus of the thesis. The concepts MLflow Models and MLflow Registry will be left out in the evaluation. In the following, features from MLflow Tracking were used to log metrics in different ML experiments on the subject system. We will also evaluate the benefits of organizing the subject system as MLflow Project.
The following sections are a collection of experiences made while working with MLflow in the context of an ADS system prototype. The prototype or subject system was used as motivation to run a set of experiments. The experiments that were run, and their evaluation, can be found in section 4.5.

5.3.2 Tracking in Single ML Model Experiments

The first experiments were with single ML models that were to be optimized. The MLflow Python API was used to track metrics used by a TensorFlow Keras model. Listing 5.4 contains an extract from the script used for training the Keras model. The extract shows the usage of the MLflow Python API.

```
def train(data_path):
    mlflow.tensorflow.autolog()
```

Listing 5.3: Example of MLflow Tracking an a ML experiment, `lidar_train.py` from [8] on branch `tools-mlflow`. The code shows the first lines of the function `train()`. There the MLflow Python API is used to log metrics using the call `mlflow.tensorflow.autolog()`. This automizes the logging and works since the ML is implemented in TensorFlow.

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-d', '--data', type=str, help="Reference a path to the data set as
        absolute path. -d / --data",
                        required=True)

    args = parser.parse_args()

    mlflow.set_experiment("lidar-cnn")

    with mlflow.start_run():
        train(args.data)


if __name__ == '__main__':
    main()
```

Listing 5.4: Example of MLflow Tracking an a ML experiment, `lidar_train.py` from [8] on branch `tools-mlflow`. The code extract shows the function `main()`. After the arguments that are passed to main are prased, the MLflow Python API is used to set the experiment and start an experiment run. In the experiment the function `train()` is called.

The usage of the Python API is simple and well documented. When using a popular ML framework like TensorFlow or PyTorch for training, it is not necessary to manually add metrics that need to be tracked. The Python API from MLflow automatically tracks the metrics used in the model. Therefore, it is not necessary to add a lot of code manually to implement tracking. Since it is also possible to track metrics and parameters manually, MLflow tracking can also be useful in Multi Machine Learning Model Systems and ML models that were implemented without relying on a ML framework for training. Runs are organized into experiments. It is possible to maintain multiple experiments simultaneously and compare different runs of the same experiment. Consequently, it is possible to create experiments for every model in a system or other experiment type. For more information about experiment types, read Chapter 4. On the other hand, apart from logging metrics

and organizing and versioning runs, MLflow Tracking has no more interesting features.
To store and maintain artifacts belonging to an ML experiment, MLflow creates a new directory, called *mlruns*, in the directory the experiment was run from. This directory contains a folder for every experiment. The folders are named using the experiment ID that MLflow assigns either randomly or can be alternatively set by the user. The folder for an experiment contains directories for every experiment run. Instead of logging to a local directory, it is also possible to store artifacts, parameters, and metrics in a remote database. In our case, we only used tracking to a local directory. When running a lot of experiments, a lot of data is accumulated. Thus, is might be reasonable to use tracking on a remote server or database instead.
When running experiments using MLflow Tracking for every model in the system, it also means that for every model there is a different *mlruns* folder, assuming the different ML models are not located in the same directory. At least, this was the case for the KoopaCar. With an increasing number of ML models, it might become difficult to navigate between different experiments.

Next, we used MLflow for a different ML experiment. In this case, the ML model that was used for experiments was yolov5. Yolov5 is a ML model deployed by Ultralytics [33]. It comes with several Python scripts, for example, scripts that allow users to train the model on custom data. For training the model with MLflow Tracking, the fork from the Ultralytics repository by ElefHead was used. Since the training script for yolov5 is more complicated than the one for the previous model. Yolov5 allows the implementation of logging by using callbacks.
The MLflow Python API was used to implement callbacks, which are called at several stages of the training algorithm. Whenever one of the callbacks is called, the following functions from the `MLflowLoggers` class are called to manage MLflow Tracking.

The only problem with the code was that the whole repository was stored to log the model. Doing so filled up the disk quickly. To work around this, the functions manually passed.

5.3.3 MLflow Project for the KoopaCar

Both experiments were also run as MLflow Project in addition to the normal MLflow Tracking. Creating an MLproject file and adding default parameters for the different entry points made it easier to run the same experiment repeatedly. Part of the MLproject file can be seen in Listing 5.2. Other benefits of creating

an MLproject are MLflow stores and displays more meta information, as well as providing the command that can be used to reproduce the experiment.
In later stages, a virtual environment was used to run the experiments. By doing so and adding the corresponding YAML file to the MLproject file, running the project and the experiments on another system became easy. Creating an MLproject is simple and does not take up a lot of time. The improved consistency and easy execution of experiments is a nice feature.
Another benefit of running experiments as MLflow Project is that the *mlruns* directory is located in the same directory as the *MLproject* file. In the case of the subject system, this means that the *mlruns* directory is located in the root of the repository, which is convenient.

5.3.4 MLflow UI for Comparing Runs and Plotting

The MLflow UI was used to visualize and compare different runs. The UI can be started from the directory that contains the *mlruns* directory and hosted to the localhost or a remote server. The UI can be viewed from any browser by connecting to the server or machine hosting the UI on the corresponding port.
When first starting the UI, the user sees an overview over experiments and runs of those experiments. The overview can be seen in Figures 5.5 and 5.6. In the top left corner is a list of the experiments in the directory. The runs for the selected experiment are shown in the middle of the UI. The overview can be shown as a table, as it is in Figure 5.5. Alternatively, the user can configure a chart view. In the chart view, chosen metrics of the runs are plotted together. This can be seen in Figure 5.6. In the example, the left plot is a bar plot showing the losses for the experiments. The plot next to it shows again the loss, but this time plotted as curves against the number of epochs.

In addition to the overview over experiments and runs, a run can be selected to provide more information on meta information and parameters, code, artifacts, or metrics that were logged. The screenshot in Figure 5.7 shows what this looks like. Some of the information that is shown in the screenshot is only stored and shown, if configuring the directory as MLflow Project, marked in green. Meta information, description, parameters, metrics, tags, and artifacts are listed below each other in the overview. Each category can be opened up or closed down to reveal or hide the information. Meta information includes, the experiment name, run name, corresponding IDs, duration, username, date, Git commit, status, and lifecycle stage. If the experiment is run as MLproject it also includes the entry point that was used and the source meaning the project. The run command is only available, if the experiment was run as MLproject. The command can be used

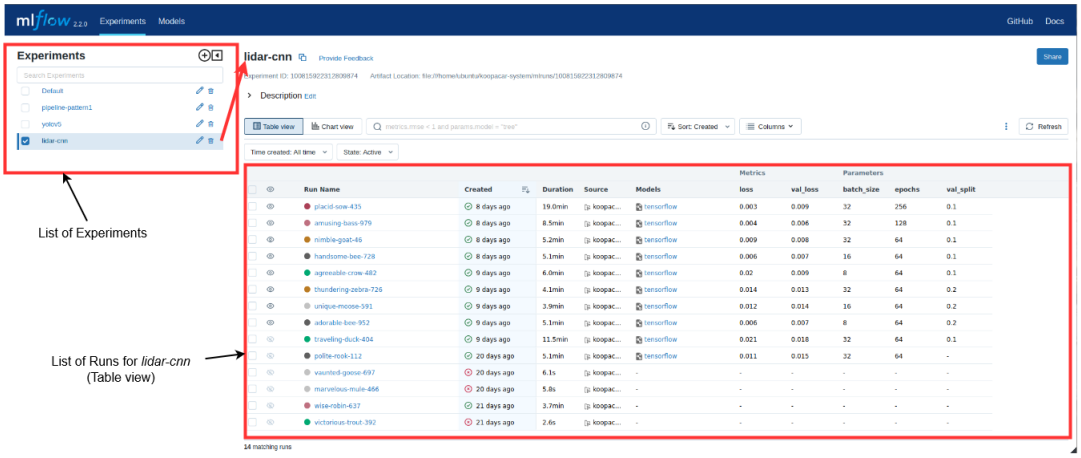


Figure 5.5: Overview of the MLflow UI. The screenshot shows the table view of the *lidar-cnn* experiments. The red box on the top left highlights the different experiments that can be selected. The names for the different runs were created automatically. It is possible to rename the runs or to assign a user defined name when recording an experiment run. The big red box in the middle of the screenshots highlights the table view of runs that were logged under the experiment name *lidar-cnn*.

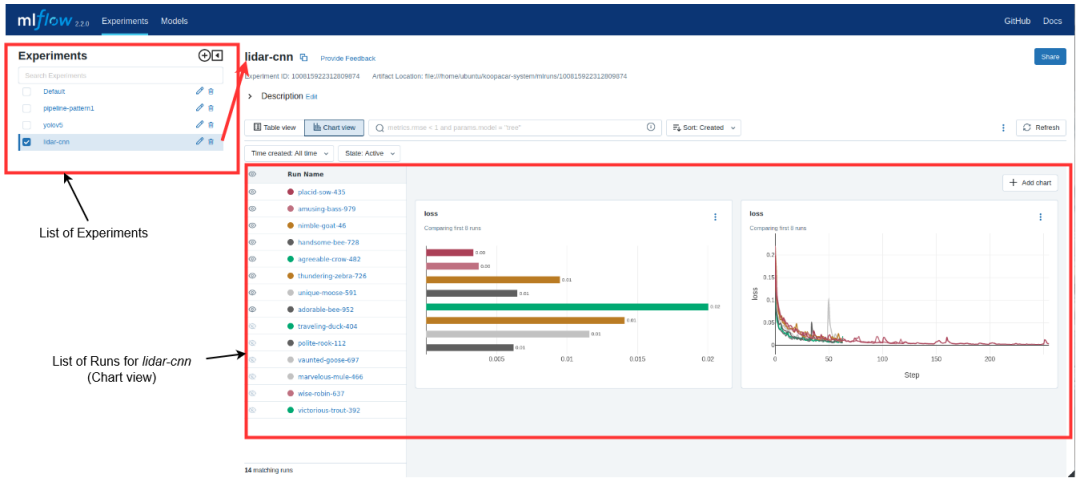


Figure 5.6: Overview of the MLflow UI. The overview is the same as in Figure 5.5. Instead of the table view, the red box in the middle highlights the chart view. In this view, charts can be used to compare different experiment runs directly.

to reproduce the experiment. The description can be used to add a user defined description. The tabs for parameters and metrics contain lists of the parameters

and metrics that were used in training. Lastly, the artifacts contain any artifacts logged during the experiment run. This usually includes the trained model.
Figure 5.8 shows how metrics for a run are plotted. On the left-hand side is a toolbar from which, amongst others, additional metrics can be selected. In the middle, the user can see the plot, which can be adjusted. The user can for example zoom in or manipulate the axes. Below the plot is a list with important values for the metrics shown in the plot. The minimum, maximum and last value are printed. The plot can be downloaded as spreadsheet (.csv) or image in different data formats.

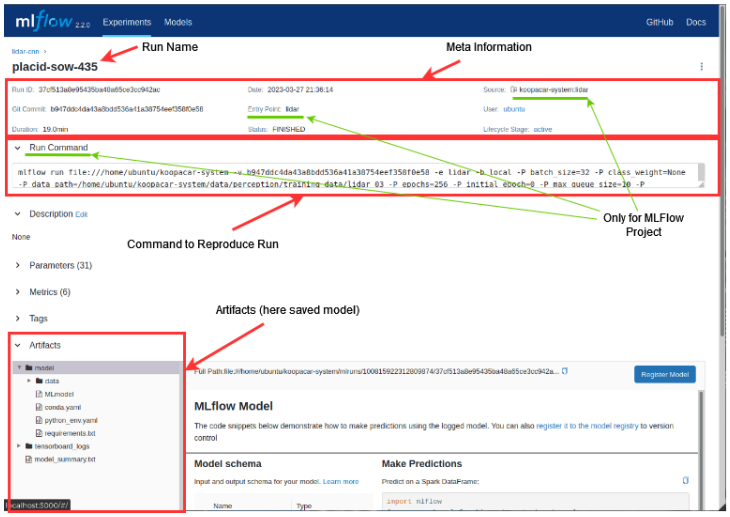


Figure 5.7: Overview of experiment run. The Screenshot shows the overview of an experiment run. On the top is a collection of meta information. Below is the run command that can be used to reproduce the run. The description was left empty. Following this are menu points for parameters, metrics, tags, and artifacts. They can be opened up to reveal more information on the run.

5.3.5 Experiments with Multiple Machine Learning Models

In this section, we will focus on how MLflow can be used to run ML models in Multi Machine Learning Model Systems. In Chapter 4 four training patterns were proposed that can be used to train ML model pipelines. A pipeline in this context is a segment of the Multi Machine Learning Model System that includes two or more ML models.

Figure 5.8: Plotting metrics for experiment run. The screenshot shows what the menu to plot metrics for an experiment run looks like. There are options on the left-hand side that allow the user to format the plot and select different or additional metrics to plot. The plot is shown in the middle of the screenshot. Below is a list with interesting or important values for every metric in the plot.

The four patterns are Individual Model Training, Partial End Loss Training, Simultaneous End Loss Training, and Alternating End Loss Training. The biggest difference is between the first and the last three training patterns. The first pattern trains the ML models in the pipeline independent of each other. The other three patterns rely on the end loss of the pipeline to train the models. The difference between the three being, when and which ML model is updated. For more information on this, read Chapter 4.

Experiments for the Individual Model Training do not differ from common ML experiments. The training for each ML model can be run from their individual training scripts. It could be possible to use a multistep workflow to evaluate the end loss in between the model updates.
Experiments for the last three training patterns differs a lot from common ML experiments. The only support for End Loss Training Patterns would be multistep workflows. The training algorithms for each model have to split into their respective forward and backward passes. The passes are then called using entry points in the multistep workflow.

The problem with the multistep workflow is that it is not well suited for the type of experiments that were proposed in this thesis. It is a feature meant for run orchestration and thus only allows the user to start processes in form of entry points. The status of the processes can be used to implement further code logic, but neither is suited for experiments with the End Loss Training Patterns.
Since the entry points are only mappings of Python scripts and functions to MLflow commands, it is advisable to implement the training algorithms without a multistep workflow. The necessary training algorithms can be implemented by calling the corresponding scripts and functions without using the MLflow entry points. These only create more abstraction in the code, which leads to errors and makes the development process more complicated.

In general, there are no conflicts in using MLflow with a Multi Machine Learning Model System. Most features are usable regardless of the complexity of the project or number of ML models used. MLflow is very flexible in how it can be used. The Python API offers the user the possibility to use MLflow in a way, such that tracking and versioning works for multiple ML models in one project.
Despite the flexibility, there is no special support for Multi Machine Learning Model System, as was expected. MLflow tracking worked well and in regard to this there are no missing features. Logging is flexible in a way that in addition to loss and other common metrics, it would also be possible to log the execution time of ML models.
Despite this, the implementation of End Loss training patterns was not finished. The problem for this lies not with experimentation management tools, but rather ML practices and ML frameworks. It is possible that more advanced experimentation management tools can support the development and training in such patterns further. Because of more urgent issues on the matter, there are no specific suggestions for features of such tools to be made at this point.

6 Result and Conclusion

This chapter will be a collection of the results of this thesis. To this end, we will take a look at the learnings and put them into the bigger context. We will also take a look at problems and challenges, and then take a look at possible next steps. The goal of this thesis was to analyze and evaluate experimentation management tools in multi machine learning model systems. To this end, we derived three research questions.

RQ1: How do the ML experiments during the development of ADS and the management of such experiments and corresponding assets differ from the workflow related to other single ML model development and maintenance that is common in other fields?

RQ2: How can experimentation management tools improve the development of intelligent systems that integrate several ML models?

RQ3: How can existing experimentation management tools be further developed to better support the development of intelligent systems implementing integrating multiple ML models?

To be able to provide answers for the research questions, we took a look at ML model experiments in Multi Machine Learning Model Systems. The findings are summarized in the following.

6.1 Conclusion

By taking a look at Apollo and the subject system of this thesis, the KoopaCar, we noticed that ML models in Multi Machine Learning Model Systems follow a certain structure. They are either running sequentially, meaning one model follows the other, or parallel, meaning two or more ML models provide input for an additional ML model or another software component. This thesis took an in depth look at the parallel structures of Multi Machine Learning Model Systems. From this, four training patterns were derived. The patterns are Individual Model Training,

Partial End Loss Training, Simultaneous End Loss Training, and Alternating End Loss Training.
The first training pattern is to train every ML model independently. For the other patterns, the forward and backward passes for each model are split, and the end loss is used to update the ML models. The second training pattern updates all models in every training iteration. The third training pattern does not update both models in every iteration. Instead, only the ML model creating the biggest error is updated. The last and fourth training pattern updates a part of the ML models with the end loss, and the rest of the ML model is trained independently. An effort was made to run experiments to evaluate the different training patterns. We ran and evaluated experiments focusing on the first training pattern. While implementing experiments for the other training patterns, we encountered many hurdles. Therefore, based on the work of this thesis, there are no definite results to be presented on the training patterns.

The experiments that were previously mentioned were used to evaluate and analyze how experimentation management tools can be used to help with the development of multi ML model systems. To this end, a methodical search was performed to survey possible experimentation management tools to be analyzed. From the pool of possible tools, MLFlow was selected.
With MLflow, different features were used to support the ML experiments that were previously mentioned. We used tracking and the MLFlow project structure to organize and log metrics during experiments for single ML models. The features were also usable for other experiments that were proposed. The MLFlow features were flexible to be used in almost any context.

6.2 Results

With this in mind, we are able to answer the research questions. The answer to research question 1 is that ML experiments can differ from normal ML experiments. This can be seen in the different training patterns. The End Loss training patterns all rely on splitting the respective passes for each ML model, which is an uncommon method. While experiments for a single ML only focus on optimizing the one model, ML models in Multi Machine Learning Model Systems can be put into a bigger context. The dependency on the result of other models makes it more difficult to optimize the end result of a Multi Machine Learning Model System. These dependencies have to be taken into account when running ML experiments.

The answer to research question 2 is that there are ML management tools and especially experimentation management tools that support the development of Multi Machine Learning Model Systems. The features of the most common tools are usually very basic and versatile. Neither the project nor the code is forced into patterns that conflict with other practices. Our findings are that using experimentation management tools with Multi Machine Learning Model Systems improves the development process the same way as it does for single ML models.
The findings for the first two research questions revealed on the one hand that experimentation management tools can offer some support to the development of Multi Machine Learning Model Systems. On the other hand, there are a lot of open questions around the types of experiments that can be run in Multi Machine Learning Model Systems. Therefore, the answer to research question 3 is that at this point in time, there are no specific features missing. Note that experimentation management tools offer no dedicated support to the development process.

Problems. That this thesis did not identify a reason for further research and development of experimentation management tools, does not mean that no challenges or problems came up while evaluating experimentation management tools in Multi Machine Learning Model Systems.
One finding of this thesis is that there is little to no research into ML experiments in Multi Machine Learning Model Systems. Consequently, the four training or experimentation patterns were proposed. The problems occurred when implementing experiments to compare the different patterns. The main problem is that the TensorFlow Keras is a high-level API for TensorFlow and thus did not allow individualizing the training algorithm as would have been necessary without implementing the whole training algorithm from scratch. It can be assumed that the same is true for other high-level APIs. Training the ML models this way is not common practice, therefore, practitioners often implement ML models from scratch using low-level APIs, which allows more room for individualization.

6.3 Outlook

In outlining and evaluating current problems and the state of the research, this thesis serves as a motivation for future work.
There are two main points that deserve additional research and effort to find conclusive results.
The first point are the four training patterns that were proposed in Chapter 4. It

is left to compare the different patterns in a series of experiments comparing the time and effort it takes to optimize the performance of a Multi Machine Learning Model System, and how the performance of the Multi Machine Learning Model System is affected by the different training patterns.
The second point goes hand in hand with the first one. This thesis identified that the implementation of ML models needs to be done using low-level APIs to be able to individualize the training algorithm. Despite the existence of such APIs, it is difficult to develop new ML models or even modify existing models like Yolov5 to fit into for example the four training patterns. The problem is that research or software engineers that work on robotics, ADS, or other intelligent or Multi Machine Learning Model Systems are no experts in the field of ML or AI. There has been research into the practices regarding the development of ML models. To name an example, Amershi et al. published a case study on software engineering practices in ML model development. [34] The problem is that the research does not put enough focus on the role of ML frameworks. Therefore, it could prove beneficial to take a deeper look at ML model frameworks and other practices regarding the implementation from a software engineering point of view.

The conclusion of this thesis that experimentation management tools are useful for developing Multi Machine Learning Model Systems. The proposed patterns and the subject system are two valuable contributions that lay a foundation for further research.
Software that was produced over the course of this thesis can be found in [8] and [9].

List of Figures

List of Tables

Listings

Bibliography

[1] Sina Shafaei, Stefan Kugele, Mohd Hafeez Osman, and Alois Knoll. "Uncertainty in Machine Learning: A Safety Perspective on Autonomous Driving". In: *Computer Safety, Reliability, and Security*. Ed. by Barbara Gallina, Amund Skavhaug, Erwin Schoitsch, and Friedemann Bitsch. Cham: Springer International Publishing, 2018, pp. 458–464. ISBN: 978-3-319-99229-7 (cit. on p. 1).

[2] Markus Winkler, Håkan Erander, Jerome Buvat, Amrita Sengupta, Rainer Mehl, Sandhya Sule, Subrahmanyam KVJ, and Yashwardhan Khemka. *The autonomous car A consumer perspective*. Tech. rep. Capgemini Research Institute, 2019. URL: https://www.capgemini.com/wp-content/uploads/2019/05/30min-%E2%80%93-Report.pdf (cit. on p. 1).

[3] Z. Peng, J. Yang, T. -. P. Chen, and L. Ma. "A first look at the integration of machine learning models in complex autonomous driving systems: A case study on Apollo". In: *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1240–1250. DOI: 10.1145/3368089.3417063. URL: https://jinqiuyang.github.io/papers/fse20.pdf (cit. on pp. 1, 2, 21, 22, 32).

[4] Baidu. *Apollo*. Accessed Apr. 10, 2023. URL: https://developer.apollo.auto/ (cit. on pp. 2, 20).

[5] Formula Student Germany. Accessed Apr. 11, 2023. URL: https://www.formulastudent.de/fsg/ (cit. on pp. 2, 23).

[6] Samuel Idowu, Daniel Strüber, and Thorsten Berger. "Asset Management in Machine Learning: A Survey". In: *ICSE'21: ACM/IEEE International Conference on Software Engineering, Software Engineering in Practice Track (SEIP)*. 2021, pp. 51–60 (cit. on pp. 3, 7, 18, 52).

[7] Samuel Idowu, Daniel Strüber, and Thorsten Berger. "EMMM: A Unified Meta-Model for Tracking Machine Learning Experiments". In: *SEAA'22: Euromicro Conference on Software Engineering and Advanced Applications*. IEEE. 2022 (cit. on pp. 3, 18).

[8] Henriette Knopp and Tim Nyul. *koopacar-system*. GitHub repository. 2023. URL: https://github.com/JetteJarl/koopacar-system (cit. on pp. 4, 27, 40, 56, 57, 59, 69, 83).

[9] Henriette Knopp and Tim Nyul. *koopacar-simulation-assets*. GitHub repository. 2023. URL: https://github.com/JetteJarl/koopacar-simulation-assets (cit. on pp. 4, 24, 41, 69, 83).

[10] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074 (cit. on p. 5).

[11] *ROS 2 Documentation: Foxy*. Accessed Apr. 14, 2023. URL: https://docs.ros.org/en/foxy/index.html (cit. on p. 5).

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MITP, 2018 (cit. on pp. 6, 7, 9–16).

[13] Ding-Xuan Zhou. "Theory of deep convolutional neural networks: Downsampling". In: *Neural Networks* 124 (2020), pp. 319–327. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2020.01.018. URL: https://www.sciencedirect.com/science/article/pii/S0893608020300204 (cit. on p. 10).

[14] Y. Xu and R. Goodacre. "On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating the Generalization Performance of Supervised Learning". English. In: *Journal of Analysis and Testing* 2.3 (2018). Cited By :277, pp. 249–262. URL: www.scopus.com (cit. on p. 13).

[15] PyTorch. *PyTorch*. Accessed Apr. 10, 2023. URL: https://pytorch.org/ (cit. on p. 16).

[16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 16).

[17] Keras. Accessed Apr. 13, 2023. URL: https://keras.io (cit. on p. 16).

[18] Jakub Czakon. *ML Experiment Tracking: What It Is, Why It Matters, and How to Implement It*. Accessed Apr. 13, 2023. URL: https://neptune.ai/blog/ml-experiment-tracking (cit. on pp. 17–19).

[19] Formula Student Germany. *Formula Student Rules 2022*. 2022. URL: https://www.formulastudent.de/fileadmin/user_upload/all/2022/rules/FS-Rules_2022_v1.0.pdf (cit. on p. 23).

[20] RUB Motorsports. Accessed Apr. 11, 2023. URL: https://www.rubmotorsport.de/ (cit. on p. 23).

[21] *Turtlebot3*. Accessed Apr. 13, 2023. URL: https://www.turtlebot.com/turtlebot3/ (cit. on p. 23).

[22] Gazebo Simulation. Accessed Apr. 11, 2023. URL: https://staging.gazebosim.org/home (cit. on p. 24).

[23] *Turtlebot3 e-Manual Overview*. Accessed Apr. 13, 2023. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/#overview (cit. on p. 25).

[24] *Camera*. Accessed Apr. 13, 2023. URL: https://www.raspberrypi.com/documentation/accessories/camera.html (cit. on p. 25).

[25] *Turtlebot3 e-Manual Appendix LDS-01*. Accessed Apr. 13, 2023. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/ (cit. on p. 25).

[26] Feihu Zhang, Daniel Clarke, and Alois Knoll. "Vehicle detection based on LiDAR and camera fusion". In: *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. 2014, pp. 1620–1625. DOI: 10.1109/ITSC.2014.6957925 (cit. on p. 25).

[27] *Raspberry Pi hardware*. Accessed Apr. 13, 2023. URL: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html (cit. on p. 25).

[28] Glenn Jocher. *YOLOv5*. May 18, 2020. URL: https://docs.ultralytics.com/ (cit. on p. 27).

[29] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. DOI: 10.48550/ARXIV.1506.02640. URL: https://arxiv.org/abs/1506.02640 (cit. on p. 27).

[30] Glenn Jocher. *YOLOv5 by Ultralytics*. Version 7.0. May 2020. DOI: 10.5281/zenodo.3908559. URL: https://github.com/ultralytics/yolov5 (cit. on p. 27).

[31] Burak Kaleci, Kaya Turgut, and Helin Dutagaci. "2DLaserNet: A deep learning architecture on 2D laser scans for semantic classification of mobile robot locations". In: *Engineering Science and Technology, an International Journal* 28 (2022), p. 101027. ISSN: 2215-0986. DOI: https://doi.org/10.1016/j.jestch.2021.06.007. URL: https://www.sciencedirect.com/science/article/pii/S2215098621001397 (cit. on p. 30).

[32] Patrycja Jenkner. *15 Best Tools for ML Experiment Tracking and Management*. Accessed Apr. 13, 2023. URL: https://neptune.ai/blog/best-ml-experiment-tracking-tools (cit. on p. 52).

[33] Glenn Jocher, Ayush Chaurasia, Alex Stoken, Jirka Borovec, NanoCode012, Yonghye Kwon, Kalen Michael, TaoXie, Jiacong Fang, imyhxy, Lorna, Zeng Yifu, Colin Wong, Abhiram V, Diego Montes, Zhiqiang Wang, Cristi Fati, Jebastin Nadar, Laughing, UnglvKitDe, Victor Sonck, tkianai, yxNONG, Piotr Skalski, Adam Hogan, Dhruv Nair, Max Strobel, and Mrinal Jain. *ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation*. Version v7.0. Nov. 2022. DOI: 10.5281/zenodo.7347926. URL: https://doi.org/10.5281/zenodo.7347926 (cit. on p. 60).

[34] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. "Software Engineering for Machine Learning: A Case Study". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2019, pp. 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042 (cit. on p. 69).

[35] Ganesh Jagadeesan. *Yolov5/dev/mlflow-resume*. https://github.com/ElefHead/yolov5/tree/dev/mlflow-resume. 2022 (cit. on p. 83).

[36] Gartner. *Gartner Forecasts More Than 740,000 Autonomous-Ready Vehicles to Be Added to Global Market in 2023*. Nov. 2019. URL: https://www.gartner.com/en/newsroom/press-releases/2019-11-14-gartner-forecasts-more-than-740000-autonomous-ready-vehicles-to-be-added-to-global-market-in-2023.

[37] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

A Appendix

```
    def log_artifacts(self, artifact: Path, relpath: str = None) -> None:
        """Member function to log artifacts (either directory or single item).
        Args:
            artifact (Path): File or folder to be logged
            relpath (str): Name (or path) relative to experiment for logging artifact in mlflow
        """
        if not isinstance(artifact, Path):
            artifact = Path(artifact)
        if artifact.is_dir():
            self.mlflow.log_artifacts(f"{artifact.resolve()}/", artifact_path=str(artifact.stem))
        else:
            self.mlflow.log_artifact(str(artifact.resolve()), artifact_path=relpath)

    def log_model(self, model_path: Path, model_name: str = None) -> None:
        """Member function to log model as an Mlflow model.
        Args:
            model_path: Path to the model .pt being logged
            model_name: Name (or path) relative to experiment for logging model in mlflow
        """
        self.mlflow.pyfunc.log_model(artifact_path=self.model_name if model_name is None else
             model_name,
                                     code_path=[str(ROOT.resolve())],
                                     artifacts={"model_path": str(model_path.resolve())},
                                     python_model=self.mlflow.pyfunc.PythonModel())

    def log_params(self, params: Dict[str, Any]) -> None:
        """Member funtion to log parameters.
        Mlflow doesn't have mutable parameters and so this function is used
        only to log initial training parameters.
        Args:
            params (Dict[str, Any]): Parameters as dict
        """
        try:
            flattened_params = MlflowLogger._flatten_params(params_dict=params)
            run = self.client.get_run(run_id=self.run_id)
            logged_params = run.data.params
            [
                self.mlflow.log_param(key=k, value=v) for k, v in flattened_params.items()
                if k not in logged_params and v is not None and str(v).strip() != ""]
        except Exception as err:
            LOGGER.warning(f"Mlflow: failed to log all params because - {err}")

    def log_metrics(self, metrics: Dict[str, float], epoch: int = None, is_param: bool = False) ->
          None:
        """Member function to log metrics.
        Mlflow requires metrics to be floats.
        Args:
            metrics (Dict[str, float]): Dictionary with metric names and values
```

```
        epoch (int, optional): Training epoch. Defaults to None.
        is_param (bool, optional): Set it to True to log keys with a prefix "params/". Defaults
            to False.
    """
    prefix = "param/" if is_param else ""
    metrics_dict = {
        f"{prefix}{k.replace(':','-')}": float(v)
        for k, v in metrics.items() if (isinstance(v, float) or isinstance(v, int))}
    self.mlflow.log_metrics(metrics=metrics_dict, step=epoch)
```

Listing A.1: Extract from mlflow_utils.py from [35]. The code extract shows the implementation of logging functions used to run MLflow Tracking in Yolov5. The functions are part of a class called *MLflowLogger*. The methods delegate the logging to the MLflow Python API.

Other software that was produced over the course of this thesis can be found in [8] and [9].