CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Studying an Architectural Pattern for Deep Learning training code

Collecting and Addressing Current Software Quality Issues within Academia and the Automotive Industry for Deep Learning

Master's thesis in Computer science and engineering

Behroz Razaq
Sebastian Johansson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Studying an Architectural Pattern for Deep Learning training code

Collecting and Addressing Current Software Quality Issues within Academia and the Automotive Industry for Deep Learning

Behroz Razaq
Sebastian Johansson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Studying MODLR an Architectural Pattern for Deep Learning Code
Collecting and Addressing Current Software Quality Issues within Academia and the Automotive Industry for Deep Learning
BEHROZ RAZAQ & SEBASTIAN JOHANSSON

Supervisor: Hans-Martin Heyn, Department of Computer Science and Engineering
Advisor: Dhasarathy Parthasarathy, PhD
Examiner: Birgit Penzenstadler, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Studying MODLR an Architectural Pattern for Deep Learning Code
Collecting and Addressing Current Software Quality Issues within Academia and the Automotive Industry for Deep Learning
BEHROZ RAZAQ & SEBASTIAN JOHANSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Deep learning has become more popular throughout the years, consequently, an expansion of new developments within the field has occurred. As deep learning is mainly practiced by writing code, many established software engineering practices can be transferred to the field. While this has happened to some extent in some areas, like requirement engineering and MLOps, other subfields have lagged behind. Writing reusable and modular code is important for easy development, but there does not seem to exist a convention for how to write such code for deep learning training. Therefore, the architectural pattern MODLR was created and in this thesis, it was analyzed against found problems from practitioners of deep learning. One of the main goals of MODLR is to decouple loss code and to show the relevance of this focus, GitHub repositories were mined and automatically categorized projects based on their loss code, with the help of an LLM. The results show that MODLR is a good fit for an architectural pattern within the space of deep learning. As a bonus, it also shows one of the ways LLMs can be used to help research with automatic large-scale analysis of code.

Keywords: Deep Learning, Architectural Pattern, Software Pattern, Loss Code, Open-Source Mining, Software Quality

Acknowledgements

This thesis has been performed within the Department of Computer Science and Engineering, for the division of Software Engineering, at Chalmers University of Technology.
We want to thank all the people that took their time to be a part of the interview study, your knowledge and experience were all very helpful. We also want to thank our supervisor, Hans-Martin Heyn, for helping with structuring the thesis and guiding us through the harder moments. Another thanks to our examiner Birgit Penzenstadler, who have been very flexible and given good feedback. We are also thankful of our advisor Dhasarathy Parthasarathy, who originally came up with the idea for this thesis, for his continuous support and knowledge sharing through out the project.

I, Behroz, am grateful to my brother and especially my mom for their unwavering support during this thesis.

I, Sebastian, want to thank my family for their support during my studies.

Our final acknowledgements go to all the colleagues and friends that we made around ping pong tables across the office.

Behroz Razaq, Gothenburg, May 2024
Sebastian Johansson, Gothenburg, May 2024

Contents

List of Figures

List of Tables

1

Introduction

Deep Learning (DL) is a relatively old concept but a lot more accessible now that computational resources have progressed to handle both the data flow and the complexity that is inherent to training DL models [2]. This has the effect of making deep learning more popular, and the number of public repositories, as of 2024, is greater than two hundred thousand on GitHub[1]. DL code is usually written in the Python programming language[2], and the most common frameworks for training Deep Neural Networks (DNNs) are TensorFlow[3] and PyTorch[4]. There are, however, problems with DNN training code from a software engineering perspective, as this kind of code usually adheres to an ad-hoc architecture [46]. At the same time, software engineering research has not kept up with the rapid pace of innovation within the field of DL [14].

No attempt to formally introduce a standard architecture for training DL models has been found during the writing of this thesis. Although there is at least one in-house architecture that is actively being used and maintained, which will be the base of this thesis. The architecture in question is called (D)MoTR, which stands for Dataset, Model, Task, and Runner. See the description of dataset, model, and runner in List 1.

The (D)MoTR architecture is being developed in an automotive OEM company and started because they wanted to merge multiple different DL projects into a common DL framework. This was, in turn, done so that code could be more easily shared and reused within the different departments of the company.

The (D)MoTR architecture was designed to solve some reoccurring problems within the company. One problem was that for each project, a lot of the DL training code was written from scratch, which in turn meant that code was seldom reused. Another problem was that the models were trained on different modalities, which in this thesis are defined as different data types for model inputs and/or outputs, such as computer vision, natural language, and time series, e.g., CAN signal values stored in tables with timestamps. A third problem was that developing a model from scratch required a lot of software development knowledge, even if using one of the already existing DL training frameworks.

[1] `https://www.github.com/`
[2] `https://www.python.org/`
[3] `https://www.tensorflow.org/`
[4] `https://www.pytorch.org/`

Algorithm 1: Gradient decent implementation for training DNNs

Parameters : Number of iterations n, batch size b

for _ *in* n **do**
  $B \leftarrow \{(x_j, y_j)\}_{j=1}^{b},\ (x_j, y_j) \sim Unif(\mathcal{D})$
  $L = \sum_{\{(x_j,y_j)\}\in B} \mathcal{L}_F(\mathcal{M}(x_j), y_j)$
  $\mathcal{M}_P \leftarrow \mathcal{O}(\mathcal{M}_P, L)$
end

The (D)MoTR architecture is designed iteratively, with each iteration increasing the pattern generalizability. The last few iterations of (D)MoTR were further inspired by another DL framework called *PyTorch Lightning*[5]. This framework's main goal is to automate the training loop, letting the user write less boilerplate code and streamline the DL training process. The framework only expects a training task, which is all the steps from retrieving a training data batch to doing the optimizer step of updating the model parameters. There is, however, one part that differentiates (D)MoTR from PyTorch Lightning and regular PyTorch, which is that this architecture focuses on the loss calculation. The loss is defined as a separate component, used in the task, to evaluate the model's output (prediction) based on a pair of inputs and ground truths and returning some sort of distance value between these two. Using regular PyTorch, the loss code can be used wherever the developer decides, while in PyTorch Lightning, it is usually defined in the training task or as a separate component, i.e., there is no requirement that the loss calculation is a separate component.

From observations and discussion, however, it seems that (D)MoTR can be decoupled even further into more components to be more general to DL training. To show this, we have to observe the math of what actually happens when training a DL model. The process for training DL models is usually an implementation of the optimization algorithm gradient descent [32]; see Algorithm 1.

From Algorithm 1, four separate components can be identified: $\mathcal{D}, \mathcal{L}, \mathcal{M}$, and $\mathcal{O}$. A fifth element, the runner $\mathcal{R}$, can be derived by separating the loop as a higher-level component. The five components result in a new architecture called *MODLR*; see List 1 for component descriptions. The difference between (D)MoTR and MODLR, in simplified terms, is that the task from (D)MoTR is converted to the optimizer and the loss in MODLR, although some things from the task, like the backward propagation, are moved to the runner. Supporting a decoupled loss is one of the main goals of MODLR, following in the footsteps of the later iteration of (D)MoTR.

Modularity, code reuse, and the potential for parallel development are increased in software with decoupled components [30]. Decoupled software components are inherently modular and can, therefore, be replaced by new components with similar functionality if the need arises. Such components also enable the reuse of software without decreasing maintainability, since developers can use the components instead of copying the code. Furthermore, different teams, companies, or academic departments can research and develop different components concurrently, without conflicts.

[5] `https://lightning.ai/docs/pytorch/stable/`

These statements are backed by the growth of the microservice architecture [10].

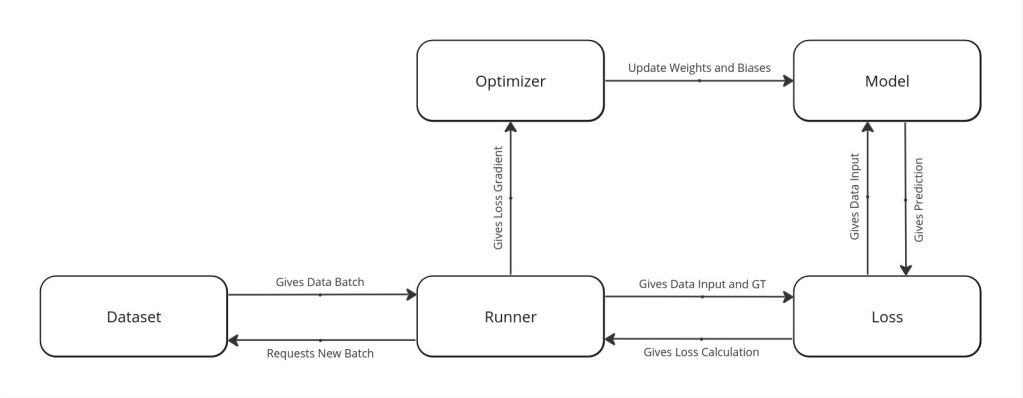

Figure 1.1: A communication diagram on MODLR. It follows an interactive systems architectural style [4], which explains how the different components interact with each other, similar to how the MVC architecture is presented [6].

These attributes of decoupled software components could also hold for DL training components. From observing how libraries are constructed and how people write DL code, we can see a trend that shows a lack of focus on the loss component, while more focus lies on the other components. Here are some developments for each of the other components:

There have been many breakthroughs in the domain of DL model architecture. For natural language processing, the standard model architecture used to be some

($\mathcal{M}$) **Model:** A DNN model with some defined network structure. Takes some input data, processes that through its neural network and gives an output. The DNN is configured with weights and biases, these are called model parameters (M_P).

($\mathcal{D}$) **Dataset:** Loads the training data. The data is loaded as batches (B), these are subsets of the dataset. Batches contain two fields, predictors (x) and target variables (y). It can contain functions to collect, process, and/or augment the data.

($\mathcal{L}$) **Loss calculation:** The loss is the output error of the model. It is calculated by comparing the model output on some input data with the true value of said data target variable. The comparison is done with a loss function (L_F). The loss calculation is part of the training step.

($\mathcal{O}$) **Optimizer:** Updates the model parameters to improve accuracy. Takes the parameters and the calculated loss as input, and alters the weights and biases in accordance with the loss through backpropagation. The parameter optimization is part of the training step.

($\mathcal{R}$) **Training loop (Runner):** The glue that connects the previous components and iteratively performs the training step for a set number of cycles.

List 1: List of components derived from Algorithm 1

form of recurrent neural network, but is now transformers that do not suffer from the same issues as the old architectures [43]. For image generation models, the generational adversarial network architecture has been replaced in many use cases with the diffusion architecture.

The common DL frameworks each have separate optimizer components, in which specific optimization algorithms can be specified. These are decoupled in the popular frameworks and enable code reuse, which has led to new algorithms being developed. Currently, the ADAM optimization algorithm is popular for many use cases [19].

The dataset's goal is to load data to the runner, but it may suffer from issues like the data not always being in the correct shape for the model or the data being biased in a way that could lead to overfitting. Several functions have been added to dataset components to mitigate these issues, including preprocessing and augmentation [40]. These are developments to the dataset that have occurred independent of the other components.

The training loop is just glue code for the other components, but there have also been developments separated from the other components here as well. One such development has been training DL models on multiple GPUs, which entails some concurrency issues. In the past, developers had to solve these issues for each training loop ad hoc. There are now frameworks in which the number of GPUs is just another parameter for the training loop component, in which concurrency issues are already handled [37].

While loss can be rather basic and use simple calculations like mean squared error (MSE), it can also be much more complex. If we look at how Generative Adversarial Networks (GANs) are created [16], where two models are used at the same time, the loss calculation for both of the models requires the other model's output, which simple loss code cannot directly support, and therefore the developers have to make their own code. This is where we, the authors, have seen something missing; there does not seem to be a good structure for how this is implemented in code. People seem to do it in their own way, which leads to less code reuse and even potentially less development in the field of finding new loss functions for other applications. It might, therefore, be important to find an architectural standard, even for loss, so that code reuse is more available. Therefore, the analysis will be on the MODLR architecture, which has a high support for loss and has not been written in literature as far as the authors know.

1.1 Statement of the problem

From the findings of the developers of (D)MoTR, the loss calculation is not usually decoupled from the other components in the DL training process. The PyTorch and TensorFlow frameworks do offer simple loss functions that compare a given model output with the ground truth of the data. These do not support more complex losses, which might need the output from multiple models and to be compared to multiple target variables with the use of multiple different simple loss functions. For more complex losses, the loss calculation is meant to be written by the user, but

there is no standard for how this is supposed to be written or how it is connected to the other components. Therefore, it would be a lot harder to share the plethora of model training code styles, which require a more complex loss, like GANs [16], whose loss requires multiple models, and object detection [47], which require different loss functions for the different steps, like classification, detection, segmentation, etc.

The architectural pattern MODLR is designed such that each component of the training process is decoupled from the others; see Figure 1.1. From observations, loss has been placed in three different components instead of being fully decoupled. It has been placed in the model, the runner, and in an object we call the training task (similar to the task in (D)MoTR and PyTorch Lightning). MODLR's approach is to be a standard that decouples even the loss in an attempt to solve several perceived issues with DL training code development. DL training code is often written in an ad hoc manner for each project, i.e., there does not seem to be a set standard for how DL code should be written, therefore leaving little room for code reuse between projects. Another reason MODLR was created was to more easily support multimodal projects. However, how well the architecture of MODLR does any of it is not well known.

1.2 Purpose of the study

The purpose of this study is to analyze a way to improve code reusability and potentially enable innovation within the domain of each DL training process component. This will be achieved by evaluating the design of the architectural pattern MODLR. In this thesis, we will try to show that loss is defined in an ad hoc manner, find issues with DL code, and see if MODLR can help address these issues.

By studying MODLR specifically and looking at other software engineering practices, like *Software Design* [4], it could enable developers to reuse and improve DL software components. This research will expand the body of knowledge about architectural design.

1.3 Research Questions

We have constructed four research questions, three for finding current problems with DL and one for analyzing MODLR.

Research question one will help with understanding what and where the problems lie when writing DL training code:
RQ1: *What are the current design problems when structuring, writing, and maintaining deep learning code bases?*

The second research question should answer what restrictions or problems the different parts of a DL training pipeline have and how they affect the other parts:
RQ2: *What aspects of the deep learning development cycle affect the code base structure?*

The third research question will look into the problems of working with different modalities and how one could lessen the impacts on these problems:
RQ3: *What are the impacts of having multiple modalities in the same deep learning code base?*

The fourth and final research question will look at how MODLR can mitigate the problems found in **RQ1-3**:
RQ4: *How can MODLR address these issues?*

1.4 Scope and Limitations

This study will only go in-depth on the two most popular DL frameworks, PyTorch and TensorFlow. We did not find a white paper study that shows this, but by doing some lookups on different sites, we can see the usage of different DL frameworks, which is shown in Appendix A. Here we can see that by pip installations and by count of repositories on GitHub, TensorFlow and PyTorch are much more popular than the other frameworks.

This paper will not take into consideration how well a DL model performs, neither in terms of speed of training nor accuracy. The focus will only be on the software architecture, and some implementation details might be ignored throughout the thesis.

2

Background and Related Work

This chapter is about the research background of the thesis.

2.1 Machine Learning, Code Smells, and Technical Debt

The interview study for this thesis will focus on maintenance and evolvability problems with DL training code. Related to this, there are code smells, which can reduce the maintainability and evolvability of programming code in general [11]. All the following studies are talking about coding in the Python[1] language, which could mean that some of the smells and debts are more related to the language. As will be shown in the following background work, however, there are problems with modularity in the context of DL code.

Gesi et al. compared how code smells differ between traditional software and DL code [13]. They found that DL code suffers from four additional problems when it comes to general development:

- The need to update or replace the ML libraries frequently, where function parameters may differ between versions.
- The need to change how the data is prepared frequently.
- The need to change the model architecture frequently.
- The need to remove redundant debugging code.

The percentage of maintenance-related modifications that are specific to DL code is 33%. Gesi et al. also looked at the most common types of code smells in DL code. They found that the most common code smells in DL code are that the ML libraries that are used are scattered over a lot of files, there is a lot of debugging code left in the production code, and there are many "Deep God Files", which essentially means large code files that do a lot of things at once. Deep God files and scattered ML libraries, according to the paper, are the most problematic code smells.

Another study also argues that ML systems have both traditional technical debt as well as additional ML-specific technical debt [39]. They say that ML-specific

[1] https://www.python.org/

technical debt is usually at a system level and, therefore, harder to detect. A lot of the debt is related to the problems of abstraction in an ML system. The study mainly talks about ML-enabled systems, which means not all the technical debt they talk about is relevant for training code. The most applicable technical debts for training code are the following:

- Configuration should be strict but easy to change, i.e., the code should not be run with bad configuration, and the user should be told about the errors. The configuration should be easy to change and have good default values for ease of use, and the user should be notified about unused configuration options to not clutter the configuration files.
- Dead Experimental Codepaths, i.e., it is so hard to make a proper change that developers fall back on doing a quick hack to test some new features.
- Abstraction Debt, i.e., there is no set standard for how to abstract the different components in ML code, which leads to people not abstracting anything at all.
- Prototype Smell, i.e., the developer chooses to develop a prototype rather than develop the new feature in the code directly, which generally means there is some problem with the evolvability or maintainability of the code base.

In another study that compares only code smells present in both DL code and traditional software, they found that the quantities of these types of code smells are not statistically different [17]. The difference they did find, however, is that when it comes to small code bases, DL code has more code smells, while for large code bases, traditional software has more code smells. The authors used tools for this kind of analysis and also did a manual analysis and made the observation that the "smelliest" part of DL code is the data pre-processing and model training. The authors also find that smelly code, for both traditional software and DL code, is more buggy. Their findings for what is the most common code smell in DL code differ from the first study and have the following three code smells instead:

- (LTCE) Long Ternary Conditional Expression, long, nested, and/or complex ternary conditions, e.g., `x = 2 if [complex logic] else 4`.
- (CCC) Complex Container Comprehension, complex list, map, set, etc. comprehensions, e.g. `l = [y(x.do_something()) for _, _, x in iter if (x / 2) & 1 == 0]`
- (LPL) Long Parameter List, functions or constructors that take in 10+ parameters.

2.2 Machine Learning and Software Design

Based on our current knowledge, there are not a lot of studies directed at the software design for machine learning. We found a Google research paper that talks specifically about the two main methods of creating ML models, notebook development and pipelines [27]. The main findings are that, generally, models are first

created in notebooks and feature engineered to fit the model's requirements, and then the model is reimplemented in the ML-pipeline context. Having to do the reimplementation step could be problematic, as many of the assumptions and nuances could be lost, leading to inconsistencies or issues in production. Being able to directly write the new code in the code base without having too much trouble training the model with different parameters would be the best.

There has been research done for machine learning and design patterns, as in [46]. They saw that ML developers generally do not use dedicated design patterns for ML systems but rather ad hoc patterns instead. Another thing this study showed was that basically all research within ML is for data management and not the software code itself.

2.3 Traditional Software Design

As the project is also about defining an architectural design pattern (a subcategory of software design), we look at SWEBOK [4] for what is important when describing software structure and architecture. MODLR is an architectural style in the subcategory interactive systems, which puts it in the same category as MVC architecture. There are three different types of quality attributes: runtime attributes, non-runtime attributes, and intrinsic attributes. For MODLR, the runtime attributes are not of concern, but non-runtime attributes, like modifiability, reusability, and testability, and some intrinsic attributes, like conceptual integrity, are more relevant. The book also talks about some important things to take into consideration when designing software, like *Concurrency*, *Data Persistence*, and *Error and Exception Handling and Fault Tolerance*, which are important factors to take into consideration when looking at MODLR. We will rely on SWEBOK for terminology and definitions around software design throughout the thesis.

Software architecture has been studied a lot. When looking at architecture as a concept, some architectures are prone to being brittle, which usually depends on things called *erosion* and *drift* [29]. A brittle architecture usually means it is difficult to maintain and evolve. Erosion occurs when developers violate the architecture, which Perry and Wolf [29] express can be as destructive as removing load-bearing walls. Drift, on the other hand, is a more gradual deviation from the architecture because the developers have a lack of awareness or appreciation of the architecture. Both of these things usually happen because the architecture is not mature enough, with either not well-defined constraints for the properties and relationships or because the architecture is not built for the problem.

For productivity sake, reuse is very important [29]. Being able to build upon others work will increase productivity a lot. Having a common architectural style helps with reuse. One problem that can happen when trying to make everything reusable is that too much is done. Having a hard time understanding which property to use when can mean that reusability is instead problematic for productivity.

As the main goals around MODLR are modularity, decoupling, and reusability, we also looked at how some conventional software designs achieve this. There are a lot

of papers talking about the architectural pattern Model-View-Controller (MVC) in particular (e.g., [6, 38]) and how it achieves the goals we want for MODLR. MVC can achieve modularity and reusability by decoupling the different modules and having a defined standard interface between them. With this, the view could be changed to other views without needing to change either the model or the controller. This is therefore similar to what MODLR is trying to achieve. The main difference is that the concerns are vastly different between the patterns.

There is also a lot of research on specifically reusability and modularity, which explains how to achieve this in general with code, like [35], which will help us give information for readers that potentially want to use MODLR themselves and help us in being able to define MODLR and what it is good at and also what it is bad at.

2.4 The Research Gap

From the literature review we conducted for the background of this thesis as well as in the planning of this thesis, we have not found research specific to software design and machine learning training. The focus seems to be on writing full pipelines, all the way from collecting data and training models to the deployment of said models (ml-ops). Various research has been done for code architecture and software engineering, but it is still missing in newer fields, like DL.

3

Research Methods

This chapter introduces the different methods performed during this thesis. The methods were of a mixed type, i.e., both qualitative and quantitative data were collected. This meant that planning would be different from only using one type.

3.1 Mixed methods

For a mixed-method design, the guidelines set by Creswell [9] were chosen. According to him, there are four points of extra importance when using mixed methods: timing, weighting, mixing, and theorizing or transforming perspectives.

For timing, i.e., what data collection should happen when, first repository mining (quantitative data collection) and then the interview study (qualitative data collection) were chosen. This way, the information from the data mining could be used to gain a deeper understanding and ask more informative questions of the interviewee during the interviews. The final step of the thesis timeline was an evaluation phase in which MODLR was evaluated based on the findings of the previous phases. Based on the knowledge gathered, the MODLR architecture could be judged more fairly.

The final result for each step was weighted as follows: The repository mining had a low weight, the interview study had a high weight, and the evaluation phase had a high weight. The role of the repository mining was supportive in nature. It was intended to show how the nature of DL training code was, in general, and could support the way the interview was conducted. The mining data could also strengthen the claims from the evaluation phase if MODLR is found to be a good architecture. The other two data collection steps had a high weight. This was because the main goal was to learn about DL code problems and see how MODLR compared against them.

For when the data should be mixed and how, the repository mining was mixed when the interview guide was created and the interviews were about to start. It was deliberate to use the knowledge from the mining to write better questions and confirm if the interviewees had seen the same thing. The findings were used to ask why the interviewee wrote code in such a way to better understand why some ways might be more popular than others. For the final mixing, both the interviews and the mining were used to check whether MODLR could be used and to see if other people use MODLR or if there are any other architectures. MODLR was also

compared against the problems found in the interviews.

As for the theorizing or transforming perspective, a software engineering perspective was used throughout this project. The focus was only on the maintainability and evolvability/expandability aspects of the software, not on the accuracy of the model. All the analysis was therefore focused on looking at the code specifically, not the use of the model afterward.

Even though the guidelines set by Creswell [9] were followed to some extent, he also gave a few predefined plans one could follow. The decision was to not follow any of them and rather use a new plan, following the structure of Creswell's plans, which can be seen in the figure 3.1. The figure follows the figures of Creswell and is almost copying the sequential explanatory/exploratory examples. The main difference is that the first step is the important step for Creswell's designs, instead of the second one. A final step was also added for evaluation, as this thesis will also include an evaluation step at the end, which was also deemed to be a more important step.

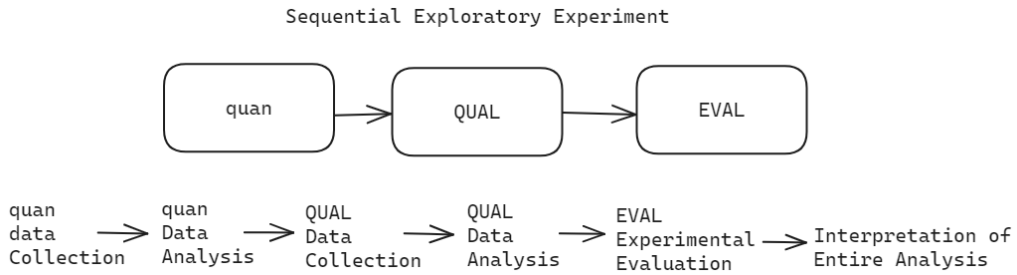

Figure 3.1: The mixed approach conducted in this thesis. Upper-case letters mean more important, while lower-case letters mean less important.

The reason Creswell's predefined plans were not followed was because the GitHub mining only gives an overview of what one of the potential underlying problems is. By focusing more on the interviews and confirming the findings of the mining with the interviewees, a broader view of the potential problems of writing DL code could be gained. This was expected to be where most of the problems with writing DL code were found.

3.2 GitHub Mining

The GitHub mining procedures goal was to check what the architectural layout of DL training code looks like, with a focus on the loss definition. To do this, the GitHub API was used to search for different DL taxonomies, e.g., *CNN*, *GAN*, etc., while also checking that the repository is based upon the PyTorch or TensorFlow framework. Then, using a large language model (LLM) [26], automatically classify all the files in that repository.

3.2.1 API search and filtering

The approach to collecting the data corpus for the mining operation was to collect similar amount of TensorFlow repositories and PyTorch repositories. A stratified sampling technique was incorporated for the corpus, recommended by [24], on 10 different DL model architectures. Seven of which were gotten from two sources talking about different DL architectures [1, 28]:

CNN - Convolutional Neural Network

RNN - Recurrent Neural Network

AE - Auto Encoder

RBM - Restricted Boltzmann Machine

GAN - Generative Adversarial Network

DBN - Deep Belief Network

RvNN - Recursive Neural Network

Three other architectures used with DL were also added:

- Transformers [44]
- Diffusion [15]
- GNN (Graph Neural Network) [31]

A total of 20 search queries were created for usage on repositories on GitHub[1], where each query used the framework name and the architecture together, e.g., `TensorFlow GAN` or `PyTorch CNN`. By searching for keywords, like PyTorch, the repositories gotten from the search all seem to have a connection to PyTorch. This is because the repository search looks through the repositories' metadata i.e. the repository name, description, and readme[2]. The only problem with this approach is that not all repositories that use the things from the query will be given. Although the missed repositories would have incomplete metadata, and an assumption was made that these repositories are not the most popular ones, as they would be harder to find by search.

The basic search API can "only" give the first 1000 repositories when performing a repository search. For this reason, random sampling, which is recommended in [24], does not work, as access to the whole population is not given. The approach was instead to collect the most popular and new repositories of these 1000 repositories per search query. This was done by ranking them on three different attributes: Stargazer, which is the most commonly used metric for popularity on GitHub and has been used by other researchers like [5]. Forks, which count how many times a project has been copied to a new repository that gets updated independently of the

[1] `https://github.com`

[2] `https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28#search-repositories`

original one. And last time since last push, the last time a code-related update has happened to the repo.

The forks and the last time since pushed are not as popular as stargazers when it comes to other papers. Forks were used because a focus was on code specifically, and forks can show how many other repos copy a style of code. A focus was also put on repositories that have updated their code more recently than others, as coding styles may have changed throughout the years, depending on things like which version of framework was used.

The calculation of the score can be seen in Equation 3.1. The R is the set of every repository, and every repository has the following three attributes: stargazer (s), forks (f), and time since last push (t). Q represents all repositories obtained from one of the 20 search queries. Q_* is a list of all the values for one of the three attributes from Q.

$$\begin{aligned} Q &\subset R \\ r &\in Q \\ X_* &\sim \mathcal{N}(E[Q_*], Var[Q_*]) \\ F_{X_*}(x) &= P(X_* \leq x) \end{aligned} \tag{3.1}$$

$$r_{score} = \begin{pmatrix} F_{X_s}(r_s) & F_{X_f}(r_f) & 1 - F_{X_t}(r_t) \end{pmatrix} \begin{pmatrix} 1.0 \\ 2.0 \\ 1.5 \end{pmatrix}$$

The scoring separates every query into its own subpopulation before calculating the final score for the repository. This is because the subpopulation does not have the same distribution as the original one, as the subpopulations are not sampled uniformly. The normal distribution therefore got calculated for every subpopulation, and with the distribution, the cumulative distribution probability for each attribute got calculated and multiplied with the weight for the attribute to get the total score. The use of the normal distribution is made under the assumption that every subpopulation is large enough to follow the central limit theorem [23], and therefore the distribution should be normal.

The weight size is a bit arbitrary, but the focus was to look into how people write DL code. Forks show which repositories the code is reused the most. This means that by looking at repositories that have a lot of forks, a lot of repositories looks like the one they forked. Focusing on time since last pushed, repositories that still gets updated will be selected for the corpus, instead of outdated repositories. It is less important than forks, but still relevant. Stargazer could show repositories that are popular, as this metric is meant to show popularity of a repository. This is less significant than time since last push, as old repos still can have a lot of stargazers, even if they are not maintained, and old repos are built more on old releases of TensorFlow and PyTorch, which has different way of training DL models.

After scoring, the repositories are ranked, and only the top repositories were picked and saved to the corpus. Even though the search queries were for different DL architectures, there are some repositories that get double-counted. To mitigate this, the repository was ignored in the second instance, and the next repository in the list was picked instead. Repositories that were forks of other repositories were excluded as well, as these repositories are assumed to be very similar to the original repository.

In total, 538 repositories was collected for the corpus. The total amount was less than $32 * 20 = 640$, as some types were not popular at all, for example, there were very few repositories that followed the RvNN model architecture.

3.2.2 Finding the loss calculation

With the corpus established, finding loss code definitions could start. From previous observations of loss definitions, the loss has been seen to either be defined in the *Runner*, the *Model*, or it could be fully decoupled within a function or a loss or task class. The assumption was therefore that most loss definitions would be found in either of these categories.

To verify the proportions of the decoupled loss component, and see how developers develop loss code definitions in DL, the loss must first be found within the different repositories. This will be done by extracting every loop defined on the module layer and function from each Python file within the repository. A pseudocode of the algorithm can be found in Algorithm 2.

As the pseudocode shows, there are three sub functions used to be able to categorize the code snippet. A Large Language Model (LLM) for logically reading the code and categorizing the code to some extent. Regex to check if the code snippet contains one of many pre-collected loss calculation functions by analyzing the imports and dependencies of the corpus. TreeSitter to look at the structure of the code and say whether the code is a part of a class or a loop, etc.

From these functions, a larger categorizing function was created. The function starts by using regex to check if a loss calculation function call, or constructor is present. These functions were found by creating another script for mining all the dependencies (imports) of the repositories in the corpus and manually checking if they are meant to be used for loss calculation. If this was false, the next step is to ask the LLM directly to see if the code is related to loss in any way. And if the LLM said that it was not relevant, the code snippet was assumed to not contain any loss code.

An important distinction for the thesis was to differentiate “loss usage” and “loss definition”. How loss is used, i.e., code snippets that contain a call to another function that calculates the whole loss, was not interesting. It is important to note that functions that uses other loss calculation functions, but only if some math or logic is present as well, are considered loss definitions, as they calculate a different thing. The LLM was used to categorize whether the code snippet contained a loss definition or not.

If all the previous steps passed, the code snippet was assumed to be a loss definition, so the next step was to check where it was located. To do this, the code snippet was checked to see if it was in a so-called standalone loop. A standalone loop in Python is where the loop is not defined within a function or a class, but rather a script file that can be run by itself, i.e., no class needs to be instantiated. It is also important to note that only two types of loops were looked at: for loops and while loops. There are other ways to iterate over a dataset in Python (like list comprehensions), which could include a loss calculation, but these techniques were deemed less practical and harder to find, and therefore skipped. If the code snippet was a standalone loop, the loss definition has to be in a runner. This is because it cannot be in a model, as this would require a model class, i.e., not a standalone loop. There might be other reasons for having a loss calculation in a standalone loop, but no instances were found to support this.

If the loss definition is defined in a function, but not a class, it can either be a loss in runner or a loss unknown category. It cannot be a loss in model, assuming that the model has to be defined within a class. It is possible to store all the weights and biases outside a class, and the function uses these weights in some way to actually be categorized as a loss in model, but this would be caught in the manual analysis explained later. If there was a training loop in the function, which we use an LLM to categorize, it would be seen as a loss in runner, for the same argument as the last paragraph. Otherwise, it would be categorized as loss unknown.

If the loss is in a function in a class, the type of class was required to categorize the loss position. This was checked by finding the characteristics of every other function within the class. The model characteristics are that it should either contain a forward function related to a model or define the layers of a DL model in some way. The characteristics for a runner is that it contains a training loop. The LLM was used to look for these characteristics. If none of these characteristics can be found, it gets categorized as loss unknown. If it has both of them, it is categorized as a loss in model and runner.

3.2.2.1 LLM and prompts for LLM

The used LLM was the **Mixtral-8x7B** [18], which is an open-source LLM model that was released shortly before the start of the thesis. Some preliminary comparison between Mixtral and another open-source model called **Phind-CodeLlama-34B-v2**[3], which is a bit older, was done, and the findings were that the Phind model performed worse than the Mixtral model.

To get good enough prompts, that is, making sure that the LLM understood the categorization it needed to do, a couple of techniques were used. The first technique was to take a set of tests for each prompt to test against. This was done in iterations, as during this step, it looked good enough for the test, and then a flaw was found that the LLM did multiple times. When this happened, the test set was updated to contain the problematic type, and the prompts were slightly modified to make sure it worked with that type as well.

[3]https://www.phind.com/blog/code-llama-beats-gpt4

Algorithm 2: Categorizing loss algorithm for TensorFlow and PyTorch Python files.

Data: *python_code*: Python function or standalone Python loop. Has access to the whole Python file as well.
Data: *loss_functions*: Pre-found loss functions used for calculating loss.
Result: Category of the function or standalone loop. Can be `no_loss`, `loss_unknown`, `loss_in_model`, `loss_in_runner`, `loss_in_model_runner`.

```
1  Function Regex(Find patterns based on input):
       // Use regex to find patterns in the input data.
2      return True if pattern found, false otherwise.
3  end
4  Function LLM(Categorize code logic):
       // Use LLM to categorize logic of code.
5      return True if it is said category, false otherwise.
6  end
7  Function TreeSitter(Categorize code structure):
       // Use TreeSitter to categorize the structure of code.
8      return True if it said category, false otherwise.
9  end
   /* Main code execution                                          */
10 if Regex(∄loss_functions in python_code) then
11     if LLM(python_code contains no loss code) then return no_loss ;
12 end
13 if LLM(python_code contains no loss definition) then return no_loss ;
14 if TreeSitter(python_code is a loop) then return loss_in_runner ;
   /* Loss is defined in a function                                */
15 if TreeSitter(python_code is not in a class) then
16     if LLM(python_code contains training loop) then return loss_in_runner
         ;
17     else return loss_unknown;
18 end
   /* Loss is defined in a class                                   */
19 foreach function in python_code's class do
20     if LLM(function contains training loop) then is_runner ← true ;
21     if LLM(function contains a model's forward function or model layers) then
         is_model ← true ;
22 end
23 if is_model then
24     if is_runner then return loss_in_model_runner;
25     else return loss_in_model;
26 end
27 else if is_runner then return loss_in_runner;
28 else return loss_unknown;
```

Inspiration was given from the zero-shot chain of thoughts [20] technique, which basically says that LLM gives more accurate answers if they are able to first "think" step-by-step instead of directly answering the question. This technique, however, uses two prompts: one for making the LLM write out each of the steps for coming to the answer, and one for extracting the actual answer. The answer wanted from the LLM was simple in this scenario (truth and false), so an instruction to give a truth and a false and using regex to extract the value worked well enough. This might not be the intended way of doing it; however, an increase in accuracy was seen with this implementation.

The final technique added was self-consistency [8]. This technique is important for consistency, as, in general, when an LLM is generating tokens, it is nondeterministic, if the settings are not set for it to be deterministic. The setting that usually changes this is called *temperature*, where a higher temperature means a more uniform distribution of the available generated tokens from the LLM. In other words, using a higher temperature on the same prompt leads to greater variance in the content of the response. This might seem to be a drawback, but the problem is that the best response from an LLM might not always be the deterministic response, so variance, to some degree, might give better answers.

The self-consistency technique was rather basic in this case, as the LLM used a non-zero temperature (i.e., the responses were nondeterministic). Then the LLM generated multiple responses, with the same prompt, and the final answer was the answer with the highest quantity.

3.2.2.2 Measuring predictive performance of the categorization

Establishing what category a code snippet belongs to was a multi-class classification task, since each can belong to one of five categories. To establish the predictive performance of the classification task established measures were chosen. The f_1-score, also known as f-measure, was picked for this purpose. It is the harmonic mean of precision (P) and recall (R) [36]. Furthermore, the f_1-score is commonly used as a measure for binary classification but can also be used for multi-class classification tasks [21]. A test data set was manually categorized, and the resulting categorizations were used to calculate predictive performance measures.

For the test data set, 30 repositories were sampled from the collected and scored corpus; see Section 3.2.1. These repositories were uniformly sampled and were only added to the test set if code defining some loss calculation could be found. For each repository, one code snippet defining some loss calculation code was manually categorized. The manual categorizations were then compared to the predictions of the automated categorization.

The number of repositories was chosen based on the observation that the interval was small enough. The first test was with 20 repositories, but the interval was rather large, and less certainty could be put on the f_1 score.

	run	mod	m&r	unk	not
run					
mod					
m&r					
unk					
not					

Table 3.1: Example of a multi-class confusion matrix. The vertical axis are the actual categories. The horizontal axis are the predicted categories. In this example the outcome will be focused on the *mod* category. The green cell is where both the actual and predicted categories were *mod*, i.e. tp_{mod}. The yellow cells are where both the actual and predicted categories were not *mod*, i.e. tn_{mod}. The dark gray cells are where the predicted category was *mod* and the actual was not, i.e. fp_{mod}. The light gray cells are where the actual category was *mod* and the predicted was not, i.e. fn_{mod}. The category names have been shortened for readability.

A multi-class confusion matrix was then constructed from the test results. From it, see Table 3.1, the true positive (tp), true negative (tn), false positive (fp), and false negative (fn) values for each category were derived [21]. Then the f_1-score's for each category were calculated.

$$P_{cat} := \frac{tp_{cat}}{tp_{cat} + fp_{cat}}$$

$$R_{cat} := \frac{tp_{cat}}{tp_{cat} + fn_{cat}}$$

$$F_1\text{-score}_{cat} := \frac{2 \times P_{cat} \times R_{cat}}{P_{cat} + R_{cat}}$$

The multi-class f_1-score was then calculated as the weighted mean of each categories individual f_1-scores [21]. The score is weighted based on the number of actual occurrences of each category in the test data set.

$$A = \{\textit{set containing each category}\}$$

$$n = \textit{test data sample size}$$

$$F_1\text{-score} := \frac{\sum_{cat \in A} F_1\text{-score}_{cat} \times (tp_{cat} + fn_{cat})}{n}$$

The f_1-score's confidence interval was then calculated using the Wilson direct method, becuase it is best method when the sample size < 100 [22]. The three parameters needed for this calculation were the size of the sample set (30), the confidence level (90%), and the f_1-score. Lastly the f_1-score was compared to the baseline of 0.2, which is the expected score that would be achieved through assigning categories uniformly to the test samples.

3.2.3 Manual analysis

The final step of the mining was to do a manual analysis of the "loss unknowns". The assumption is that these unknown losses are losses that are decoupled from the other components of MODLR. But there may be cases where this is not the case, so we had to look through these manually. There could also be other interesting instances of loss codes found throughout the manual mining.

With the manual analysis, the signature of the function was looked at for the loss code, what the function returns, and what it takes as parameters. This can show whether people have chosen a standard for how to write loss code, if they do use model as an input, or the output values of the model directly. Wrongly categorized predictions from the categorizer could be seen from the manual mining as well.

3.3 Interviews

Interviews can be represented in three parts: interviewee selection, the interview itself, and interview analysis. The following sections will explain how each of these three parts was executed.

3.3.1 Interview Selection

The people selected for the interviews were people who had DL knowledge, but preferably also other software engineering knowledge that could talk about problems with DL code, as people with software engineering experience might have a higher understanding of what is problematic code. This could be from either an academic or industrial background. The people with an industrial background came from the automotive OEM company, and the people with an academic background were PhD or postdoctoral students.

The selected people from the industry were mostly people who had worked with the (D)MoTR architecture. This was to further understand their thought process, what issues they faced, and why (D)MoTR was created. There were a total of four people who had been a part of the framework's development.

For the rest of the people, two of them also worked at the company but were not part of (D)MoTR. Six people from the academic side were also included in the interview study. Both academics and industrial people were selected to have a broader understanding of what writing DL code meant and to also see if the discipline differed in some way.

In total, there were 12 interviewees, six from the OEM company and six people from academia. The academic people had a range of experience with DL of one to five years. Most of the academics worked actively with DL during the time of the interviews as well. There were one academic who focused more on machine learning and requirement engineering, but they have worked with DL before. The people who worked in industry had a larger span from between two to ten years. Every industry interviewee had DL as a part of their area of work at the time of the interview.

3.3.2 Interview Guide

For the interviews, the approach was a general interview guide approach [42]. This meant that all the questions were written before the interview, but the interview could be flexible, and the participants could go more on a tangent than with other approaches. The thought about this was that the goal of the interview was to be exploratory; the participants were meant to show their thoughts, without too many interruptions. With this approach, a standardized open-ended interview technique could be used for later participants, which is much stricter, when more knowledge about the area is collected. The approach also gives the possibility to change the questions if the interviewees talks about another subject that was missed from the planned questions or if the questions did not give any results.

As for the questions themselves, they were focused on finding the answers to **RQ1-3**, while also understanding how a person who is used to writing DL code writes such code. The complete interview guide can be found in Appendix B. The guide starts with an introduction to the interviewee about the topic of the thesis and what research questions were meant to be answered throughout the interview. After that, the recording is supposed to start, and the first questions were about three definitions that were going to be used throughout the interview. This is to make sure that everyone has the same understanding of the words.

The next questions were about the background of the interviewee. This was meant to more easily categorize the interviewee based on DL experience and software engineering experience.

The next set of questions was about how their thought process would be if they were to develop a new training framework. Following their thought process and based on what they answered, extra questions were asked to add more constraints to the interviewee's framework to see if anything would change. The follow-up constraints were generally based on problems from the developers of (D)MoTR and what the company supervisor said were some problems they had encountered.

The questions after creating a DL framework were related to their experience with DL code. The questions related to whether they had seen structural or maintainability problems, challenges with specific components of DL training code, and asked them what type of code they had seen, based on the mining study conducted before.

The final questions were based on the MODLR architecture. These questions are used to see if the interviewee has seen this architecture before and to ask them what they think of this kind of architecture. It could also help in finding potential problems with the architecture.

3.3.3 Interview Analysis

The different interviews were all recorded and transcribed. With this transcription, coding was used to find a theme throughout the interviews [34]. The coding was done separately between the researchers and then compared to each other to minimize bias from the findings.

The interview analysis followed the coding manual of Sadaña [34]. The first step is to do "first cycle coding", where the chosen coding styles were attribute coding, structural coding, and process coding. Attribute coding was selected to be able to distinguish between people who are similar and different from each other. For the main content of the interview, structural coding was selected to give a theme for chunks of texts. Process coding was also selected, as one section of the interview is about the process of creating a DL training framework.

The next step is to find themes in the codes. The desired themes are related to code and what problems there might be when writing DL code. To find the themes, first a merging step was performed where very similar codes were merged into one. After that, every code that was not related to the thesis was removed. From here, every code was put on a canvas board and move around to similar other codes to try finding themes within them. With the different groupings, the themes were found. Some codes were used in multiple themes. The next step was to look at every found theme and connect it to one of the research questions **RQ1-3**. Some themes were found not to be useful for answering the questions.

The themes were also later used for the evaluation part of MODLR, to set a guideline for what would make an DL architectural pattern good or not; see Section 3.4.

An extra observation that was found was contradicting statements. The contradicting statements were written on a code level, rather than on a theme level. Using the attribute coding, the difference can be seen between how different types of people think DL code should be written.

3.4 Evaluation

The evaluation phase took the form of a qualitative analysis of MODLR, based on relevant themes acquired through studying the interviews. These themes were used as measures on MODLR and the pattern's effects on DL software through a comparative study. This was done by refactoring a set of open-source DL projects, to follow the MODLR architecture. A comparison was then performed on the software before and after the refactoring.

3.4.1 Selecting repositories to refactor

Three focus points were used when conducting the repository selection for the comparative study. First, the set of projects were chosen to be representative of different DL jobs, such as generation and detection, and different DL networks architectures, such as CNNs or Diffusion models. Second, for each type of project, one PyTorch and one TensorFlow implementation were added to the set. Third, only repositories that achieved a high score according to the previous scores and rankings were picked from, see Section 3.2.1. Based on these criteria, six projects were chosen to be refactored.

3.4.2 Refactoring repositories

The process of refactoring a project was done by rewriting its code in such a way that the different components had the same responsibilities and the same dependencies as specified by the MODLR architecture. The baseline was the communication diagram describe in the introduction, see Figure 1.1. Each of the projects was analyzed on a component basis. If components with coupling and dependencies that contradicted MODLR were found, these were rewritten. This was done for each project.

3.4.3 Comparing the refactored implementations

The analytical section of the comparative study was tackled in different ways. Each project was compared and analyzed from multiple angles. However, every angle of comparison was based on themes from the interview study. The intra project comparison was the first angle. Here, the implementations from before and after refactoring were contrasted for each project separately. Thereafter, the refactored implementations of similar project types implemented using different DL frameworks were compared. This was done to highlight any potential differences when implementing the MODLR design using different base frameworks. The last angle was the comparisons between different project types, independent of what framework they were implemented with. This could have led to a conclusion that MODLR might be better suited for some project types than others.

3.4.4 The purpose of the evaluation step

The comparative study, that has been described in this section, was designed to provide qualitative arguments for and against the design of MODLR. This was achieved by comparing different projects on different layers, see Section 3.4.3. It was also done in an attempt to measure how difficult the conversion process would be for refactoring different projects to follow the MODLR design.

4

Results

This chapter will go through the final result of the GitHub mining, the interview study and the evaluation of MODLR.

4.1 GitHub DL-Code Mining

The results of the mining can be seen in the following items: Figure 4.1, Figure 4.2, and Table 4.1.

The table, on the other hand, shows how many instances of each type of loss are found, as well as the quantity of repositories that contained that category. Here we can see that the repositories that contain loss calculations have more, on average, than just one loss calculation. For the unknown instances, there are more than six times of `loss_unknown` instances than there are repositories containing that category ($1314/217 \approx 6.06$). While the `loss_in_model` is around five times bigger ($1038/195 \approx 5.32$), the `loss_in_runner` and the `loss_in_model_runner` are three times bigger ($421/141 \approx 2.99$, $335/108 \approx 3.10$).

	`loss_unknown`	`loss_in_runner`	`loss_in_model`	`loss_in_model_runner`
Instances	1314	421	1038	335
Repo Instances	217	141	195	108

Table 4.1: A table showing how many loss instances found. It also shows how many repositories actually contains the selected category.

The Figure 4.1 shows what proportion of loss types found within each repository. In the x-axis, it shows the quantity of repos selected per search query, where the top x the highest scoring repos are selected. This shows that the proportions stabilizes for each category, and, seen in the figure, the `loss_unknown` stabilizes around 60%, which `loss_in_model` also does. The figure shows that the loss placement is more often in these two categories than `loss_in_runner` at around 40%, and `loss_in_model_runner` at around 30%. The `empty` category represent those repositories that did not have any loss, according to the mining algorithm that extracted the losses. Possible reasons for why there are such repositories, might be because they only contain loss calls (compared to loss definitions), they are wrongly categorized by the loss extractor model, or the repository does not contain any loss related code at all.

This result is also used to decide that there were enough repositories in the corpus. By looking at the graph and seeing that it stabilizes at around 16 repositories, the change to the final result would be small if adding any more repositories, while also taking a lot of time.

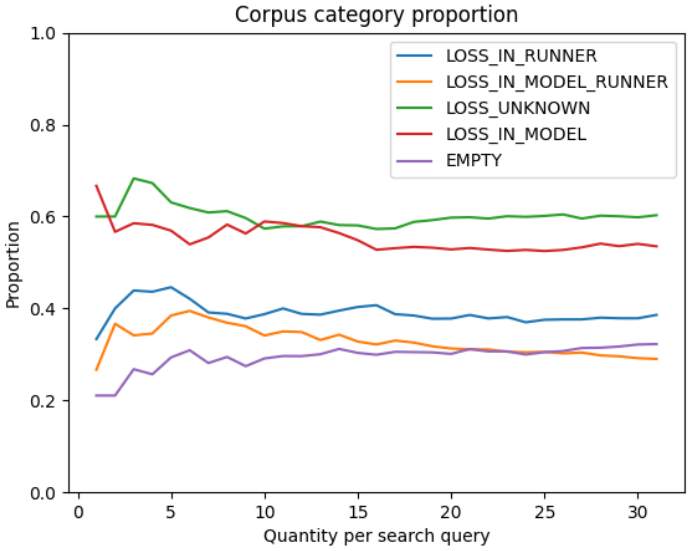

Figure 4.1: Proportion of how likely a repository uses one of the four categories (or none) based on how many repositories are taken from each search query, see Section 3.2.1.

The four way Venn diagram in Figure 4.2 shows to which extent the different repositories have an overlap of the different categories. It shows that the different repositories have some sort of overlap between the categories (167 repositories with non overlapping categories compared to 279 repositories with overlapping categories). Although, all the categories highest count is on their non overlapping counterparts. There are also some other interesting thing shown in the graph, like if a repository has the category `loss_in_model`, the likelihood that it also have a `loss_unknown` category is over 60% ($124/195 \approx 0.64$). There are also 20 repositories that have all the categories at once.

4.1.1 Categorization performance metrics

After sampling, labeling, and comparing with the predictions, the test data set yielded concrete metrics for the predictive performance of the automated loss categorization. With a 90% confidence level it has an f_1-score in the interval $[0.69, 0.91]$, see Appendix D for the raw data. That is a positive difference of $[0.49, 0.71]$ compared to the baseline f_1-score of 0.2.

4.1.2 Manual mining results

The manual mining results are shown in Table 4.2, Table 4.3, and Table 4.4. With the missed categorization and the found types of loss, an accuracy score is calculated

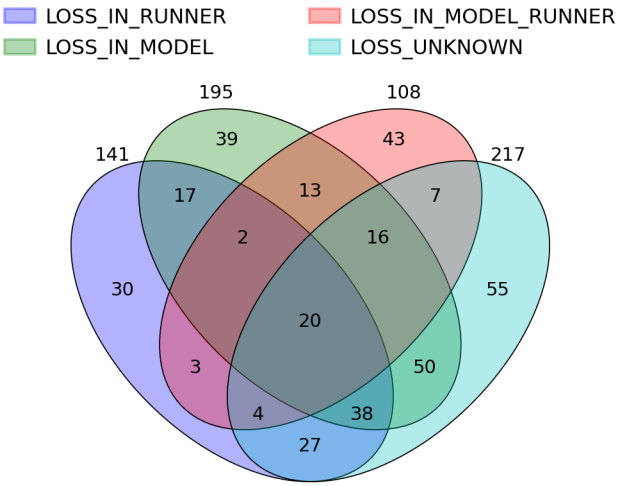

Figure 4.2: A four way Venn diagram that shows how the different loss categories overlap. We can see that the most popular way is to only use one category at a time, but there is more overlap than not.

to see the precision of the `loss_unknown` categorization: $\frac{1183}{1183+131} \approx 0.90$.

The biggest category of miss categorizations, seen in Table 4.2 are the loss call, loss in model, and no loss. The most common loss calls instances are when the loss call is put into more complex functions, which do more things than just calculating the loss. The most common loss in model instances are when the loss is put into a class that is a subclass, which hides the model's neural network in its superclass. The most common no loss instances are in functions with many lines of code that calculates something that is not the loss.

Type (count)	Description
Loss constructor (4)	Model wrongly predicted a class constructor representing loss
Loss in runner (5)	Model wrongly predicted a class or a function containing a runner and a loss
Loss call (65)	Model wrongly predicted functions or classes only containing a loss call
Loss in model (35)	Model wrongly predicted a class containing both loss and model
No loss (21)	Model wrongly predicted a loss was present
Loss in model runner (1)	Model wrongly predicted a class containing a model, runner, and loss
Total miss categorizations: 131	

Table 4.2: Found missed classifications from the manual mining of the `loss_unknown` category.

Most instances of the `loss_unknown` is an instance of loss standalone, which is a function, or a class, that is only meant to calculate loss, as seen in Table 4.3. There were 956 instances of this type, compared to the other categories that had 227 instances. Of these, the majority category is model output only, i.e. the functions takes model outputs as inputs, with 886 instances compared to 70 instances where the full model is the input. There were also many instances of the task-like structure, i.e. functions and classes doing the loss calculation, backward propagation, and the optimization step all at once, which had 68 instances. These instances usually either used the PyTorch lightning framework, or used the TensorFlow framework.

There are also a lot of testing instances using loss definition. Although, these instances usually come from the same repositories and there were not many different repositories that used testing found from the `loss_unknown` instances.

For the loss standalone instances, extra metrics collected can be seen in Table 4.4. Many of the standalone instances contained multiple losses at once, a total of 168 such instances. These instances usually have an input parameter specifying what type of operation should be done within the functions. As for the extra metric returned subcategory, 77 instances also returned other metrics, like some sub-step of the loss calculation, model output, and other metrics calculations.

Type (count)	**Description**
Multiple losses (168)	The loss function contained multiple different losses, which could be changed by changing the input to the function
Extra metrics returned (77)	As well as the loss returned, metrics of different kinds were also returned
Backward (6)	The calculated loss also got a backward call, as well as returning the loss
Regularization (11)	The returned loss was a kind of regularization loss
Gradient returned (5)	Loss and gradient returned
Optimizer value returned (1)	Optimized weights got returned and loss

Table 4.4: Special cases of the standalone losses extracted from `loss_unknown`.

4.2 Problems with DL code

This section presents the findings from the interview study. The section will first go through a simplified overview of the themes found from the interviews, and later go in-depth on the relevant themes for the research questions one to three. The section will also bring up some contradictions found between the interviewees.

4.2.1 Themes

The following themes were found throughout the interviews:

Type (count)	Description
Model dependency (70)	Standalone loss function or classes dependent on model(s)
Model output only (886)	Standalone loss function or classes that is not dependent on model(s) but only their outputs
Loss in test (58)	Code that uses loss definition in tests, or tests loss code directly
Loss in evaluation (27)	Code that uses loss definition to evaluate models
Loss in optimizer (1)	Loss definition is defined within an optimization class
Loss in tf.estimator (4)	Loss definition together with a TensorFlow estimator, a more extensive training task
Loss in a task structure (68)	Loss definition within a function or class that also does the backward propagation and optimizing step
Loss in a sampler (2)	Loss definition within a class that samples the losses that the model should be trained upon
Loss in hook (5)	Loss definition within a hook, or callback, usually used in a runner of some sort
Loss in dataset (1)	Loss definition is defined within a dataset class
Loss in quantizer (3)	Loss definition within a quantizer class
Gradient calculation (16)	Gradient and "gradpenalty" is calculated in place of loss
Loss in distribution (7)	Loss definition within a class calculating distributions (usually "kl loss")
Reinforcement score/loss (3)	Score calculations for reinforcement learning
Function returning loss functions (32)	Functions returning standalone loss functions (usually in GAN repositories)
Total standalone losses: 956	
Total other losses: 227	

Table 4.3: Found types of losses from the original `loss_unknown` instances.

Theme	Description
CODE IS FAULTY	There are faults (bugs) in DL code
TENSOR AND MODEL SHAPES ARE HARD TO KNOW	Especially with dynamically created computational graphs, like in PyTorch, knowing how each layer connects to each other is problematic
LACK OF UNIT TESTING	People seems to not test the logic of DL training code with unit tests
A LOT OF DEAD CODE IN DL	It appears that dead code artifacts are prevalent throughout DL code
NOT WELL DEFINED WHERE TO PLACE COMPONENTS	Everyone seems to have their own idea for what component should do what and no standard seems to be present
DATA IS PROBLEMATIC	The biggest hassle with DL seems to be that data has to be handled differently in different situations
CONFIGURATION IS NOT TRIVIAL	Using configurations to more easily log and compare runs is not easily
AMOUNT OF ABSTRACTION IS NON TRIVIAL	How much abstraction to use is a balancing game which does not seem to have an easy solution
DL IS TRIAL AND ERROR	The process of finding the best model seems to need a trial and error approach
EXECUTION ORDER NEEDS TO BE MODIFIABLE	Mostly for logging and adding extra functionality, but being able to change what gets executed when in the training loop seems to be important.
DL CODE IS HARD TO READ	Many people have said that understanding what is happening in DL code can be hard.
DL CODE IS HARD TO REUSE	A lot of training code is based on others' implementation, but it seems that reusing code requires a lot of effort
MODEL AND DATA ARE COUPLED	It seems that the model's neural network needs to take into consideration the shape of the data, or the other way around
DL CODE CAN BE HIGHLY COUPLED AND HARD TO REUSE	As some people write DL code in a way where everything is coupled, reuse is a struggle
MODALITY IS PROBLEMATIC BASED ON APPROACH	Some interviewees said they had minimum problem with modality while others had a lot
USING FRAMEWORKS MITIGATES FAULTY CODE	Using well tested and used frameworks removes the need to write a lot of the boilerplate code which can lead to fewer faults (bugs)
NON SOFTWARE ENGINEERS ARE DEVELOPING DL MODELS	DL is generally used by writing code, but there are a lot of people who do not have experience with writing code that still uses DL
LOSS IS SIMPLE ENOUGH TO NOT NEED CHANGE	Some people say that there is no need to focus on decoupling, or abstracting loss
TIME AND OTHER CONSTRAINTS MAKES THE CODE WORSE	With a tight time limit, not enough effort can be put into code quality
TASK LIKE STRUCTURE	There have been some instances where people use the training task instance, which encapsulate the loss and a couple of more components
MODLR SEEMS GOOD	For some of the interviewees, MODLR seems like a good architecture
NOT A STANDARD WAY FOR DEFINING NEW LOSSES	There lacks a standard for how to write loss code in both TensorFlow and PyTorch

Table 4.5: All found themes from the interview study, including a small description of what the theme is about.

The relevant themes will be further talked about in the sections talking about the research questions.

4.2.2 RQ1: What are the current design problems when structuring, writing, and maintaining deep learning code bases?

The related themes to RQ1 can be seen in Figure 4.3. The themes are mostly related to individual writing of DL code, but there are also some problems with current frameworks as well. After the figure, a more detailed view of what the interviewee said will be presented for each theme.

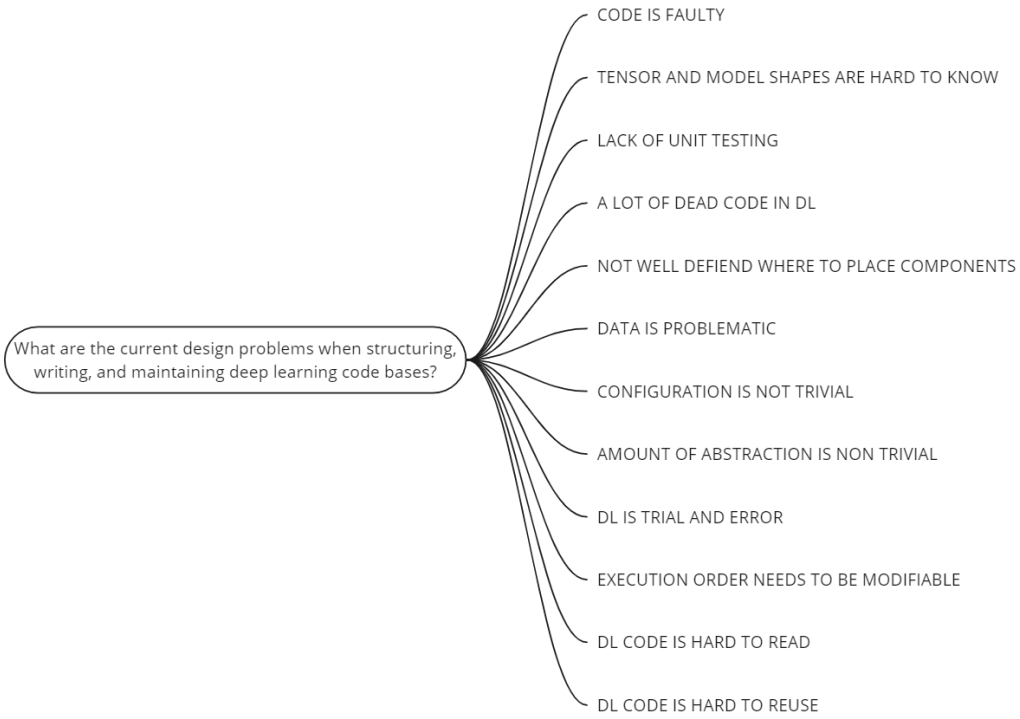

Figure 4.3: The found themes from the interview study that are related to RQ1.

CODE IS FAULTY

There were two main codes that contribute to this theme, `research code does not always work` and `can never be completely sure everything is correct when testing`. As for the first code, there were some interviewees that said some codebases, published by researchers, could not be run without encountering some sort of error. One of their assumptions is that the code gets changed after the fact of publishing the research and code, so following the original paper does not yield the same result. As for the other code, it means that by nature DL models are not deterministic, and neither is the training, so even though it seems that the model is learning, and no runtime error has occurred, there still might be some logical faults within the training program and restricts the model to be as accurate as possible.

This theme is a big issue if present in code. Although, it seems that not all interviewees has seen this, and the ones who did say that more "prestigious" publication have general less faults. Introducing more unit testing can help mitigate the faults

was also mentioned.

TENSORS AND MODEL SHAPES ARE HARD TO KNOW

For this theme, there are four main codes found: `forced to print shape`, `tracking tensor shapes is hard`, `not understanding how the model looks like`, and `static analysis of how the tensor changes would be preferable`. Understanding the shape of tensors and the model is important to reuse and understand the code. While TensorFlow's approach is to make a statically created computational graph, i.e., it gets easier to see how the shapes propagate throughout the program, the larger problem is with PyTorch approach to this, which uses a dynamically created computational graph. Simply said, it is much harder to see how the tensor shapes changes throughout the program.

One approach to this issue is the first code, printing out the shape of the tensor throughout the program. The difficulty with this is that the program has to be running for this to work, which might take up a lot of resources based on the model. The next three codes are related to not being able to look into the program statically to see how the network looks like, nor how the data changes throughout the forward pass.

LACK OF UNIT TESTING

The main codes for this theme are: `people do not test their code`, `non existing unit test in DL code`, `testing metrics not logic`, `need for more unit tests in DL`, `without unit tests there are no guaranties on the functionality`, and `training logic can be bug prone`. The interviewees that talked about the lack of unit tests said that a lot of logical problems could be solved by actually using more unit tests. Some mentioned things to have unit tests on would be that the tensor shapes are correct throughout the program.

Even though multiple interviewees said that DL code is faulty, they still claimed that a lot of people do not test their code, or, at least, not enough. The interviewees gave two explanation for this: The focus lies on testing the accuracy and other metrics of the model, rather than the code itself and their knowledge background do not come from software engineering, and they are not used to software engineering practices, such as unit testing.

A LOT OF DEAD CODE IN DL

Some interviewees said there were a lot of dead code in DL. The main code that corresponds to this is `dead code exists and hinders developers`. The main problem with dead code is that it makes the code harder to read, as knowing what is relevant and not cannot be easily seen. This, in turn, hinders other DL developers to reuse other people codes, as it is a lot harder copying the things needed. One of the reasons mentioned for why there is a lot of dead code is that the discipline of creating DL models requires a lot of trial and error and testing new things directly in the code.

NOT WELL DEFINED WHERE TO PLACE COMPONENTS

This theme is based on quite a few codes, but as a basis, it seems to not be a well-defined structure for how to develop DL code. For example, some interviewees said they would not put the data collection and preprocessing together with the training code at all, and instead have them as a different part. The interviewees also gave contradicting examples on how DL code should be structured and how components should be coupled or decoupled, which is more talked about in Section 4.2.5. Another problem that comes from not having a well-defined structure, one interviewee said, is that things get harder to find and the modules are scattered. This also leads to less readable code, which a couple of interviewees mentioned. One interviewee also mentioned that a lack of a standard structure leads to modularity issues, as not following a structure leads to spaghetti code, i.e., people write code where it feels it is needed, rather than thinking about the structure, which makes so that commonality between modules is much harder to establish. An interviewee also complained that there was no standard way for defining loss in PyTorch, requiring the developer themselves to decide how to implement it.

There were also interviewees that talked about the benefits of introducing a common pattern. This would mitigate the issue of having code scattered about and makes the codebase more understandable. One idea about having a standardized structure, one interviewee said, is that the data transfer between the different components would be more well-defined, making it easier to create diagnostic tools, or even manual diagnostic, for these components. Another interviewee talked about that it can be easy to miss some parts when developing DL training code and said that a common architecture that have stricter requirements opposed on its components could make it easier to see what is missing.

DATA IS PROBLEMATIC

The theme that most interviewee talked about was that in some way handling data has its problems. One issue is that data processing can also require a trial and error approach and have the same code issues as the model from this. Data can also be stored in different format in different places, which increases the complexity of the collection step. This further increases in complexity when you have to merge different types of data into one, like merging categorical variables to a time series, which one of the interviewee had difficulty with.

These things lead to a high complexity, which makes the code harder to read and understand, and therefore maintain. One interviewee also mentioned that when reading someone else's code, you cannot really understand how the data is stored for that person, and therefore the data code is complex. Another interviewee also mentioned that storing data and retrieving data from after it was collected is not easy either. They mentioned that because of a general DL framework they used, they had to compromise with implementing extra functionality to make it work, like checking the memory and removing items when it was full, just because the framework loaded data in batches that could be too big. They also mentioned that it is not clear what data transformation should be done within the preprocessing

during data collection, or during the loading of the data. This requires a compromise, as some data transformations takes a lot of time, but it is less flexible if not done during the loading phase. On top of that, reusing these data components between different modalities is also a problem, as the transformations are not general, another interviewee said.

CONFIGURATION IS NOT TRIVIAL

For this theme, there are two main codes, `managing different configuration is hard` and `reading and writing configuration is hard` and are related to how hard it is to write and use a configurable training framework. In the first code, one interviewee talked about it can be hard to just run different configurations. Although, this issue is mainly due to how the cloud infrastructure is done and tooling as a training configuration gets pushed to the cloud and how queuing works. They tried to run multiple different configuration before the training has started, so knowing which one actually got run was hard to figure out.

The second code is more about the actual configuration file, where another interviewee talks about the problem with reading and writing configuration. Their issue lies with having a lot of different parameters to be able to configure, many thousand lines, and keeping these in check took a lot of effort.

AMOUNT OF ABSTRACTION IS NON TRIVIAL

This theme is around the difficulty with choosing the amount of abstraction when developing DL training code. The main issue here is that there is no clear amount of abstraction needed when developing DL code. For instance, one interviewee talks about how the abstraction should be based on the use case. By this, they mean if the underlying model is a GAN, for example, different types of abstraction are needed, compared to how much a CNN needs. This interviewee and another also talks about the concerns with too much abstraction. Too much abstraction can make code less understandable, and especially if you want to reuse someone else's model, or other code, and have to go through layers upon layers of abstraction, they feel it is just unnecessary. It could also take a lot of time to abstract components, and building everything concrete is generally faster.

A related code, which is also a part of this theme, is `increasing generalizability can increase complexity`. The difference is that generalizability is more general than code abstraction, but two other interviewees came to the same conclusion as with the abstraction case talked about before. They both talked about finding an amount of generalizability that does not make everything too complex, and that should be the goal when generating frameworks.

DL IS TRIAL AND ERROR

This theme was mentioned as a problem in some previous themes. Because of the trial and error nature of developing DL models, it is important to support this. One interviewee says this requires that the code is evolvable, which means a

focus on being able to add new features and changes without too many difficulties. Two interviewees specifically says that the model definition, i.e., the neural network layers, requires trial and error to get a good enough model.

EXECUTION ORDER NEEDS TO BE MODIFIABLE

The runner, or training loop, is where the execution order is defined for what happens when. Four interviewees talked about the need to be able to change this execution order, or change what is happening in the training loop. One of them needed to do some other calculations based on physical properties of what the model is trying to predict. Another required to do post-processing on the output. A third interviewee needed to be able to add masking to the training. All these interviewees put an emphasis on changing the runner based on the task at hand.

DL CODE IS HARD TO READ

For this theme, there was quite a spread on what was hard to read in DL code. One interviewee, who worked with an intersection of DL and physics, said it was hard to distinguish between what was actually a calculation based on physics contra a part of the DL calculations required for training. They said it would be better if there was some way of "disentangle the science from the actual coding". One particular problem this interviewee had, was the need to reshape the data in some way and knowing if this was based on the physics or to fit the DL model.

Another interviewee had difficulty with the type system of Python and that not every developer specifies what type everything is. Python is dynamically typed, and the data types are not enforced and giving type hints, i.e., expected types for function calls and constructors arguments, is not enforced either. This then means that it can become a guessing game for what type some function actually requires, which this interviewee saw as an issue with the Python language.

A third interviewee told that academic DL code is seldom maintained and can be very hard to understand for others that are not a part of the original development group.

As the final point for this theme, two interviewees said it was hard to understand the data that is being used throughout the training code. One of them, related to the theme `tensor and model shapes are hard to know`, talked about knowing what are the shapes of the input data and how does this shape changes throughout the data preprocessing and augmentation steps. They said that the tensor operations in particular were quite hard to know how they transformed the data, and rather recommended the usage of a library called `einops` which is a library that allows for similar notations as something called Einstein Notations. The other interviewee talked about the data structure and how using the Python dictionary class to represent the data is problematic. A dictionary can dynamically add, change, and remove parts of the data and being a newcomer to a project and trying to understand what the data is, which usually is dynamically loaded from compressed data types like pickle, is not an easy task. The interviewee says that these dictionaries

can be nested as well, with multiple dictionaries within a dictionary, which makes this process even more confusing. They even argue that using tools for looking at the data does not help either, as what the data actually represent is not clear from only looking at the key names. Their recommendation is to rather write an extensive documentation for what all the keys represent.

DL CODE IS HARD TO REUSE

This theme is related to how DL developers often want to reuse other people's implementation in some way, and that the reuse is seldom as easy as copying and pasting the relevant code sections. One interviewee talked about the lack of a standard structure leads to spaghetti code and generally bad code structure. They argued that this directly leads to a harder time reusing DL code, and having a common structure would be beneficial for code reuse.

Another interviewee talked about having a hard time reusing predefined models, which means models where each layer are already defined, as these items are not customizable enough. It can be that it is necessary to log some things within the model, or other functionality that is crucial for the specific context. The interviewee says that sometimes it is easier to just implement it from scratch, rather than trying to reuse this predefined model.

Three different interviewees has also said that a lot of code they have gotten from other projects and people have a lot of issues, including modularity issues and structural issues. One of these interviewees continues, giving an example for why reusing code is problematic. They say that the data might be missing, the documentation is not understandable, you need a specific tool for actually running and training the model. They give one example of when an open-source repository required a closed source commercial tool for actually working, which made it practically impossible to use that repository without spending a lot of money.

These issues seem to generally come from repositories associated with research, and a few interviewees said themselves that they do not lay that much focus on maintainability attributes, but rather that there is a result within a given timeframe. Four different interviewees directly said that research DL code has some sort of modularity problem. They say that code is scattered around in different places, and it seems they either group everything together directly, or they write code based on convenience. Some interviewees have commented that this is related to people not coming from a software discipline, related to the `non software engineers are developing dl model` theme, and are not well accustomed to software engineering practices.

4.2.3 RQ2: What aspects of the deep learning development cycle affect the code base structure?

The related themes to RQ2 can be seen in Figure 4.4. The themes are talking about coupling and how they might affect other parts of DL code. After the figure, a more detailed view of what the interviewee said will be presented for each theme.

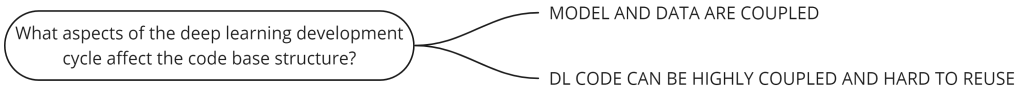

Figure 4.4: The found themes from the interview study that are related to RQ2.

MODEL AND DATA ARE COUPLED

Many of the interviewees said that the data is highly coupled to the model, which is what this theme is about. One interviewee talks about how a model cannot support general data, so what is needed to support such a scenario is having a modifiable middle layer between the data and the model which can transform the data to be the correct size. Otherwise, a general model for different modalities and different shapes of data would not be possible. Another interviewee said a similar thing, the only difference is that they suggest that the DataLoader class should be this middle layer and therefore be the class that actually is dependent on the model. A DataLoader is a class in PyTorch for loading data in memory and generating batches of data to be trained upon.

Three interviewees took the opposite approach and said that this should be the other way around, where the model, and its input layer, should be dependent on the data and how the data is processed. This has the problem of making the model less flexible instead. In both way, data and model are connected, and no interviewee gave a concrete solution for this.

DL CODE CAN BE HIGHLY COUPLED AND HARD TO REUSE

This theme talks about how other DL components are coupled together. One interviewee talked about how sometimes, based on the developer, every component is coupled. Some people write their code in so-called "God files", where every component is written within one file instead of being separated into smaller chunks. Another comment from another interviewee was that having general preprocessing and post-processing code is practically impossible, as these are always dependent on the underlying data and the output of the model.

4.2.4 RQ3: What are the impacts of having multiple modalities in the same deep learning code base?

There is only one related theme to RQ3, as seen in Figure 4.5. The theme, however, is rather big, where the interviewees take a lot of different approaches handling multiple modalities. After the figure, the different problems will be presented, as well as why some interviewee did not find it that hard to work with multiple modalities.

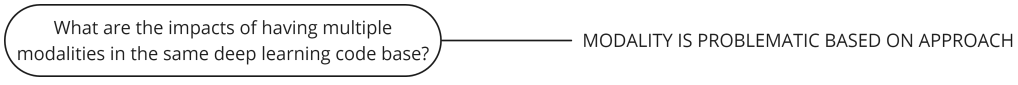

Figure 4.5: The found theme from the interview study that are related to RQ3.

MODALITY IS PROBLEMATIC BASED ON APPROACH

One interviewee said that the issue with supporting multiple modality is that the data that is being used requires different amount of resources. Their issue, in particular, was with how some data types (time series) require no memory management, while others (like computer vision) requires that you take care of that data and only cache some data at a time. Having a generic dataset type makes this harder as well, as then you cannot know what type of data is used, and making a generic memory resource manager then becomes problematic.

Another interviewee says that different modalities have different issues, which in turn makes it harder to do everything generalized. As an example, the interviewee says that the loss is dependent on modality and that all loss calculations cannot be used with the same modality. Similar to this, a third interviewee says that each and every modality needs its own encoding.

As for the data handling, three interviewees say that the data processing is dependent on modality. This means that reusing processing functionality is hard, if not impossible, between different modalities. Some examples the interviewee said were that a rotation transformation can be performed on images, but not on text. Even if one can do the same type of transformation, the transformation cannot always directly be transferred. One example, once again with images and texts, is that of slicing. Slicing a piece of text is too different from slicing images that these should also be seen as totally different operations.

Another problem, which is not only related to multiple modalities, but also to how files are stored and loaded, is that loading files that are stored differently require a lot of setup to make everything work according to one of the interviewee. Their concern was that different data was stored in different files with different types of storage. This required them to create their own data loading so that everything gets loaded in the correct way, which was a messy process.

While most interviewees said that multiple modalities require different types of encoding or decoding, and some also said it affected other components as well. One interviewee said that modality is purely a data concern and does not impact the training after the encoding part (and decoding part), and the other parts should not be impacted. What they claim is that all data gets converted to a vector in DL and from that point, everything is the same. The concept of modality is not as important in DL as it is a vector to vector process, but using non-DL machine learning instead, there are some models that are created specifically for some modality. They claim that multiple modalities are not that hard from a coding perspective in the context of DL training code.

This is not the only instance of interviewees saying that supporting multiple modalities is not a problem. Three other interviewees also say similar things, where only the data processing is affected by multiple modalities and this process is not that big of a concern. Two of those specifically said that the only component that needs to be changed to support multiple modalities is the dataset.

Figure 4.6: All the mentioned theme codes related to coupling. The left codes are related to having a high coupling between components, while the right side is having a low coupling between components. The number within the parenthesis is the quantity of interviewee that talked about the code.

4.2.5 Contradicting statements

During the interviews, a lot of talk happened concerning coupling. The interviewees talked about how they would develop a training framework, and sometimes also mentioned what component they would couple with other components. This can be seen in Figure 4.6. Some interviewees thought that different component were supposed to be more coupled than others, while others did not have the same approach. This figure shows that peoples approaches are not consistent when it comes to DL. Some examples are the code `unsure about loss separated from model`, which two interviewees claimed when talking about the MODLR architectural pattern, and `coupled loss and model is bad`, which two other interviewees claimed.

Another thing the figure shows is that there is no visible distinction between the academic side, nor the industrial side, they both have similar amount on both side of the scale of coupling vs. decoupling.

4.3 Evaluating MODLR

The repositories chosen for the comparative study are from one of three jobs or model architectures, see Table 4.6. This section displays and explains how the software architecture of each of these projects has changed when refactored to fit the MODLR design. The architecture of the projects are displayed using UML class diagrams. In these figures, names of framework implemented components will include the module which they are from, e.g., `optim.Adam` is a commonly used PyTorch implemented optimizer. The code before and after refactoring is openly available on GitHub [1]. The repositories are given ID numbers, which are used to refer to them. Each subsection includes descriptions of the projects as it is and explanations of the changes which have occurred through refactoring.

ID	Repository (user/name)	Start file path	Framework	Job/Model architecture
1	eriklindernoren/PyTorch-GAN	implementations/acgan/acgan.py	PyTorch	GAN
2	YunYang1994/TensorFlow2.0-Examples	6-Generative_Adversarial_Networks/Pix2Pix.py	TensorFlow	GAN
3	jwyang/faster-rcnn.pytorch	trainval_net.py	PyTorch	RPN
4	YunYang1994/TensorFlow2.0-Examples	4-Object_Detection/RPN/train.py	TensorFlow	RPN
5	adore801120/attention-is-all-you-need-pytorch	train.py	PyTorch	Transformer
6	Kyubyong/transformer	train.py	TensorFlow	Transformer

Table 4.6: Repositories chosen to be refactored for the comparative study. For each job or model architecture, one GitHub repository was picked from the collected corpus, see Section 3.2.1. The first two are implementations of generative adversarial networks producing images from labels. Second two are object detection models using region proposal networks to analyze images. The last two are implementations of the original transformer model architecture [44].

4.3.1 Applying the architecture to repository 1

This project, see Figure 4.7, was completely encapsulated in one file, i.e., all components used were either imported from the framework or defined in the start file. This has been changed now so that all components, see Figure 4.8, that are not imported from the framework are defined in separate files. The top of the start file defined and parsed arguments for specifying different configurations for the different training runs. This, along with the optimizer and data loader components, imported from PyTorch, are the only parts that have not been changed during the refactoring process.

The loss calculation, for this project, belongs to the category `loss_in_runner`, i.e. defined in the training loop. The `for-loop` component iterates over batches provided by the data loader. These are then used for generating outputs from the two models, and each model's output is used to calculate their respective loss. The discriminator model output is also used to calculate an accuracy metric. The losses are then backpropagated and the optimizers update their respective model's parameters.

After the project has been refactored, the `for-loop` is now decoupled to the Runner class, see Figure 4.8. This component is dependent on all the others, and is only

[1] https://github.com/BehrozRazaq/MODLR-comparative-study

responsible for integrating the other components and logging some metrics during the training process.

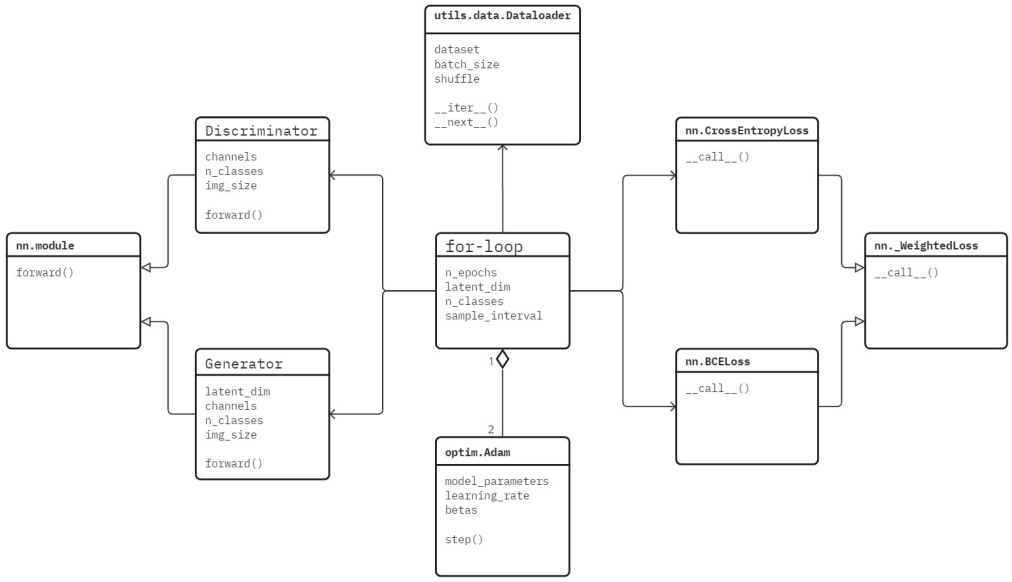

Figure 4.7: The class structure of repository 1 before being refactored. The *for-loop* component is a loop defined in the start file's module level.

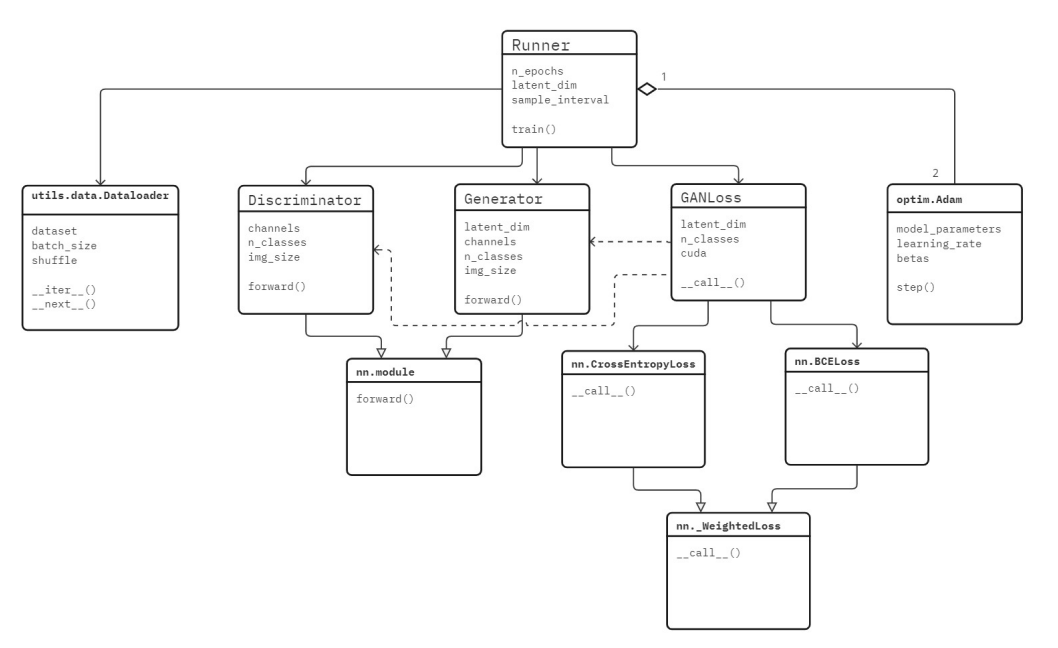

Figure 4.8: The class structure of repository 1 after being refactored to follow the *MODLR* design.

The loss calculation has been decoupled, from the `Runner`, as the `GANLoss` class. For each iteration of the training loop, the `Runner` calls on this component to cal-

culate the loss. As input, it sends the models and the training data. The `GANLoss` component then controls the flow of training data, model output generation, and calculation of the losses. Since the `Runner` still calculates the accuracy metric, the generated model outputs are cached after each call to the `GANLoss` component. This makes those outputs accessible to the training loop.

4.3.2 Applying the architecture to repository 2

This project, see Figure 4.9, is also fully contained within one file. Therefore, any component used is defined in this file or imported from TensorFlow. In the refactored version, see Figure 4.10, every component is defined in their own files and modules (where applicable). The optimizers are the only components that were not altered.

The start file has 16 function definitions without any clear delineations between them. Two dataset components are created, one for training and one for testing. However, some rows of code used to initialize these components are duplicated. Every function from `load()` to and including `load_image_test()` in the `for-loop` component, see Figure 4.9, are only used for the creation of these dataset components.

The `Generator()` and `Discriminator()` functions define the network architecture for each of the models and return the models as instances of `keras.Model`. These functions are also the only locations where `downsample()` and `upsample()` are called from. The `OUTPUT_CHANNELS` variable is only used by `Generator()`.

The loss calculation takes place in two separate functions, `generator_loss()` and `discriminator_loss()`. These take in their respective models' generated output as well as the target variables they compare said output against. They are also both dependent on the same loss object, which is an instance of the `keras.losses.BinaryCrossentropy` class. These loss functions are called from the `for-loop` component by passing on the models' generated outputs as arguments, i.e., it is a case of `loss_in_runner`. In the training loop, the losses' gradients are then calculated and used by the optimizers to update each model's parameters.

The refactored version of repository 2, see Figure 4.10, has decoupled components for the dataset, models, loss calculation, and runner. The new `Dataset` class has the functionality of the previous data processing functions and contains both the train and test datasets. For the models, a parent class, `_GANModel`, that inherits from `keras.Model` has been added. It also includes the common functions which both models are dependent on. Both models are defined as extensions of said parent class. The loss calculation was separated into the `GANLoss` class from the `for-loop`. For each call to calculate the loss, this component takes in both models and their respective input and target variables. It also contains an instance of the previously mentioned loss object. Lastly, the `for-loop` has been refactored to the `Runner` class, which handles the interactions between the other components and the logging of some progress metrics in the training loop.

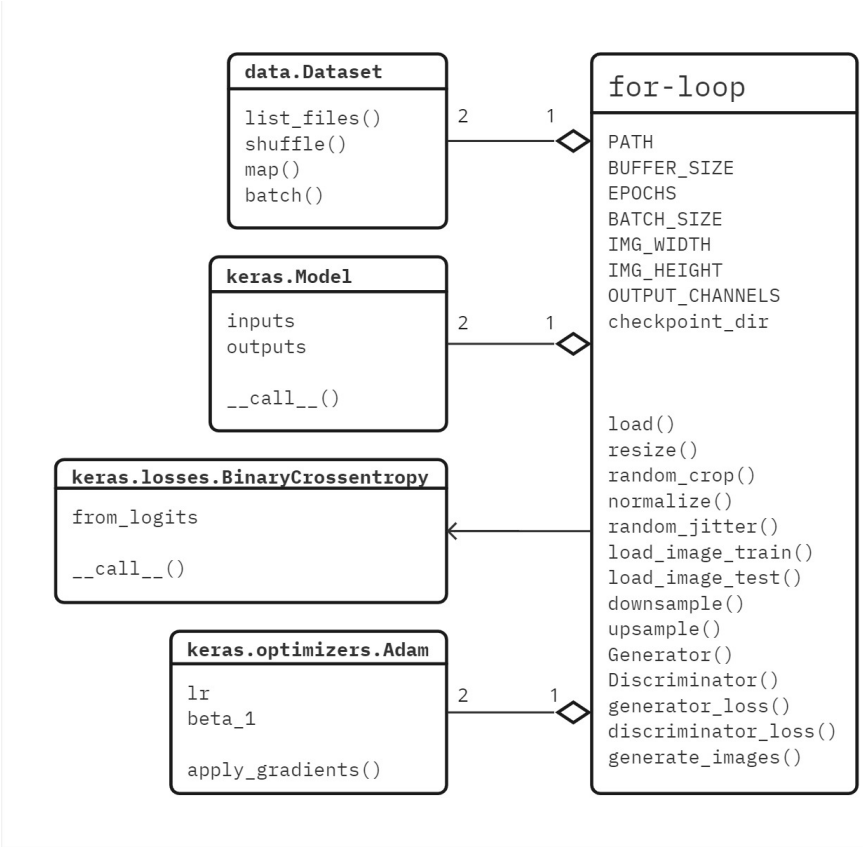

Figure 4.9: The class structure of repository 2. The `for-loop`, as in Figure 4.7, functions as a training loop and is defined at the module level in the start file. In the diagram it, to some extent, represents the start file itself.

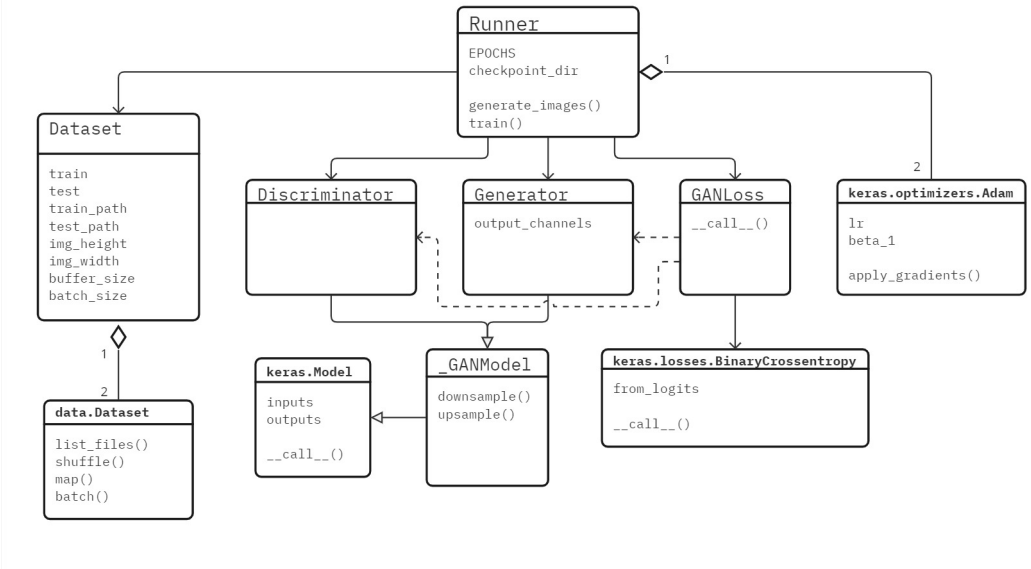

Figure 4.10: Class structure of repository 2 after being refactored.

4.3.3 Applying the architecture to repository 3

In repository 3, see Figure 4.11, some components are not defined in the start file or imported from the framework, unlike the previous projects. Mainly, the model components are defined in a separate module. The optimizer alone has not been altered during the refactoring process. Any new components derived during refactoring, see Figure 4.12, are defined in their own separate files.

To create the `Dataloader` multiple components are defined and initialized in the start file. This includes the `sampler` class and the `roibatchLoader` component, which are defined in and imported to this file, respectively. The instances of these components are only used to initialize the `Dataloader`.

This project has two model implementations that can be trained, `vgg16` and `resnet`. Both of extend the `_fasterRCNN` super class. This class in turn extends the frameworks `nn.module` class. Another model, `rpn`, is used as part of `_fasterRCNN`'s network architecture. When called to generate a prediction, `_fasterRCNN`, first uses its sub-model to generate its output and uses that to produce its own.

The loss calculation in the project is categorized as `loss_in_model`, i.e., the forward functions of the `_fasterRCNN` and `rpn` output their calculated model errors each time they are called while in training. The loss calculated by the larger model is dependent on the one returned by the smaller.

The training loop, see the `for-loop` in Figure 4.11, is dependent on all the previously described components. Moreover, it is also dependent on the `sampler` and `roibatchLoader`, but it only uses the `Dataloader`. It is also responsible for logging the progress and for saving the model parameters, during and after training.

The refactored version, see Figure 4.12, has four new components, the `Dataset`, the `fasterRCNNLoss`, and the `Runner`. The `Dataset` inherits from the `Dataloader` and contains all necessary dependencies needed for that component's initialization. The training loop is no longer dependent on the `sampler` or the `roibatchLoader`. The forward-pass function in the models no longer contain the loss calculation logic. The logic for both have been moved to the `fasterRCNNLoss` class. When the component is called on it first uses a model to generate a prediction on some output data, this is then used to calculate and return the loss. The `for-loop` has been extracted to its own file and is defined as the `Runner` class, with necessary attributes explicitly stated.

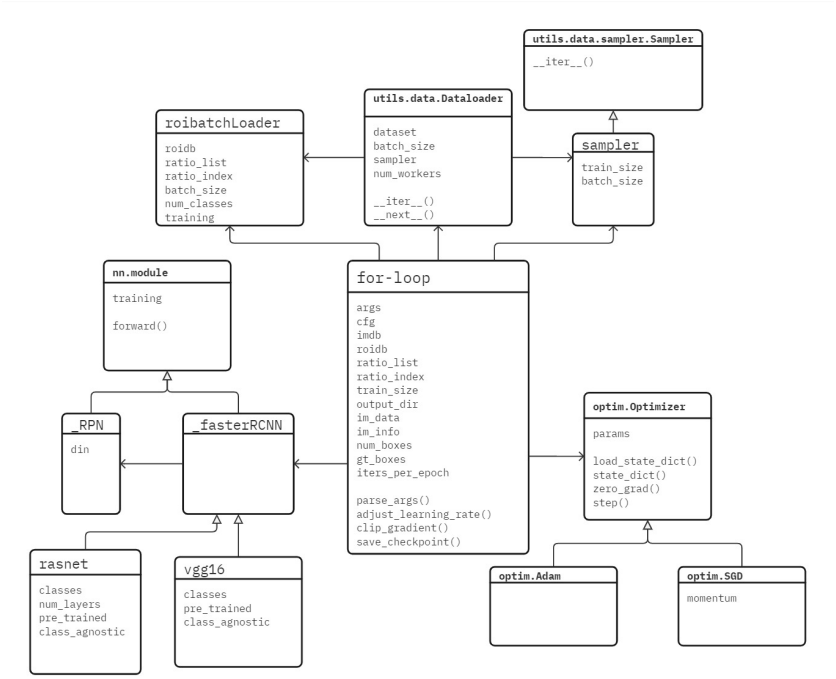

Figure 4.11: A diagram displaying the class structure of repository 3. The `for-loop` component is the model training loop and represents the module level of the start file. The `roibatchLoader` is defined in modules not included in the comparative study, and only its interface is of interest in this exercise.

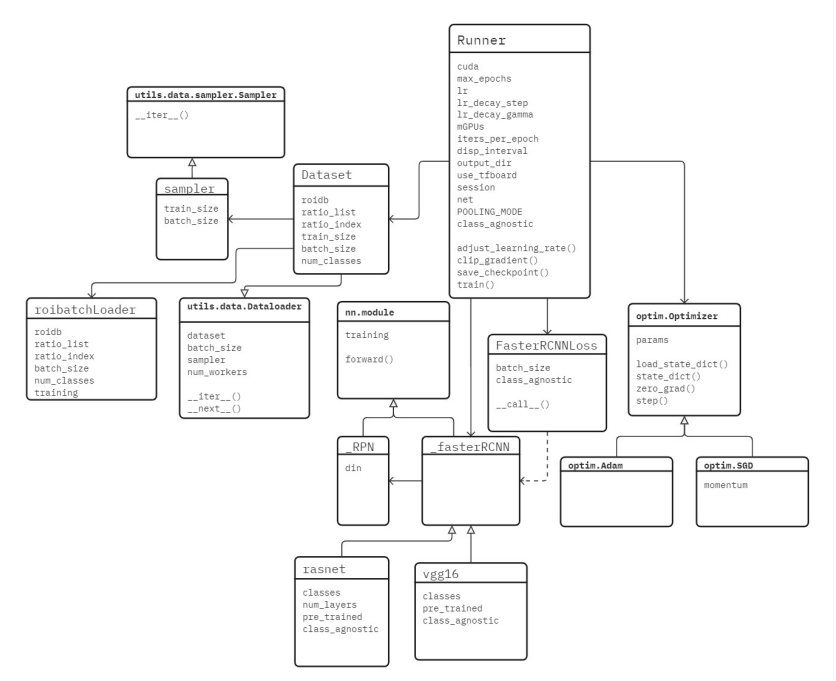

Figure 4.12: Class structure of repository 3 after being refactored.

4.3.4 Applying the architecture to repository 4

The components used in this project's start file, see the `for-loop` component in Figure 4.13, are mostly defined in said file or imported from the framework, except for the `rpn` model's class. New components, derived from the refactoring process, are defined in their own separate files. The optimizer and model are the only components not changed.

Every function, except `compute_loss()`, defined in the start file is only used for either loading, processing, or sampling data. Every iteration of the training loop, it calls `DataGenerator()` for a new data batch. That is the only interface between the training loop and these data related functions. `DataGenerator()`, see Listing 1, uses the `yield` keyword inside a `while`-loop to achieve similar functionality to an iterator.

```
def DataGenerator(synthetic_dataset_path, batch_size):
    """
    generate image and mask at the same time
    """
    image_label_path_generator = create_image_label_path_generator(
        synthetic_dataset_path
        )
    while True:
        images = np.zeros(
            shape=[batch_size, image_height, image_width, 3], dtype=np.float
        )
        target_scores = np.zeros(shape=[batch_size, 45, 60, 9, 2], dtype=np.float)
        target_bboxes = np.zeros(shape=[batch_size, 45, 60, 9, 4], dtype=np.float)
        target_masks = np.zeros(shape=[batch_size, 45, 60, 9], dtype=np.int)

        for i in range(batch_size):
            image_path, label_path = next(image_label_path_generator)
            image, target = process_image_label(image_path, label_path)
            images[i] = image
            target_scores[i] = target[0]
            target_bboxes[i] = target[1]
            target_masks[i] = target[2]
        yield images, target_scores, target_bboxes, target_masks
```

Listing 1: Function that on first call loads in the dataset and on subsequent calls samples and returns data batches.

The loss is calculated in the training loop by calling `compute_loss()`, putting the project in the `loss_in_runner` category. The loss' gradient is then computed and sent to the optimizer to update the model parameters, all within the loop.

After being refactored, the project has three new classes, see Figure 4.14. The `Dataset` class is an iterator with the functionality of the `DataGenerator()` and the other data related functions defined in the start file. The `RPNLoss` class is responsible for using the model to generate an output on some input data, calculating the error between that and a target variable, and then returning that error. The component is used by the `Runner`, which contains the functionality of the training loop. It is responsible for the interactions between the other components and for logging and saving the training progress.

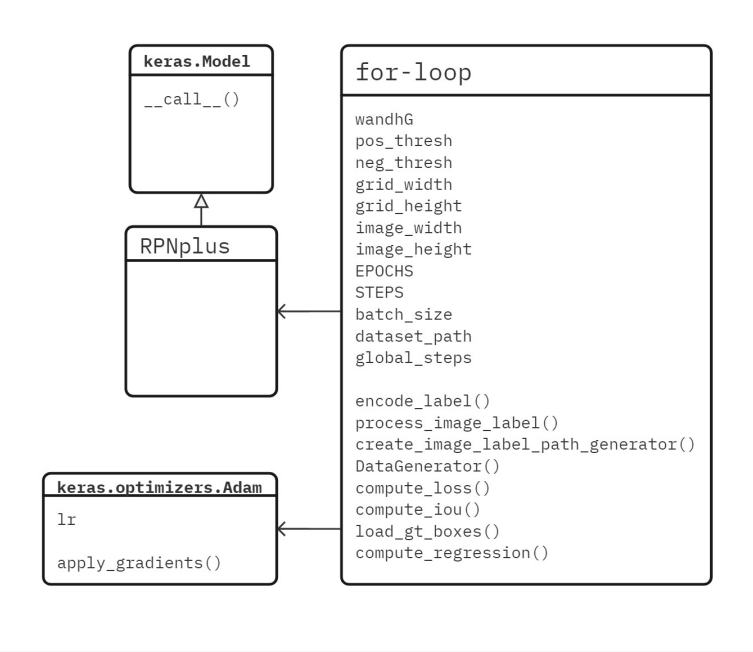

Figure 4.13: Class structure of repository 4 represented as a diagram. The `for-loop` component is, as before, the model training loop and represents the module level of the start file.

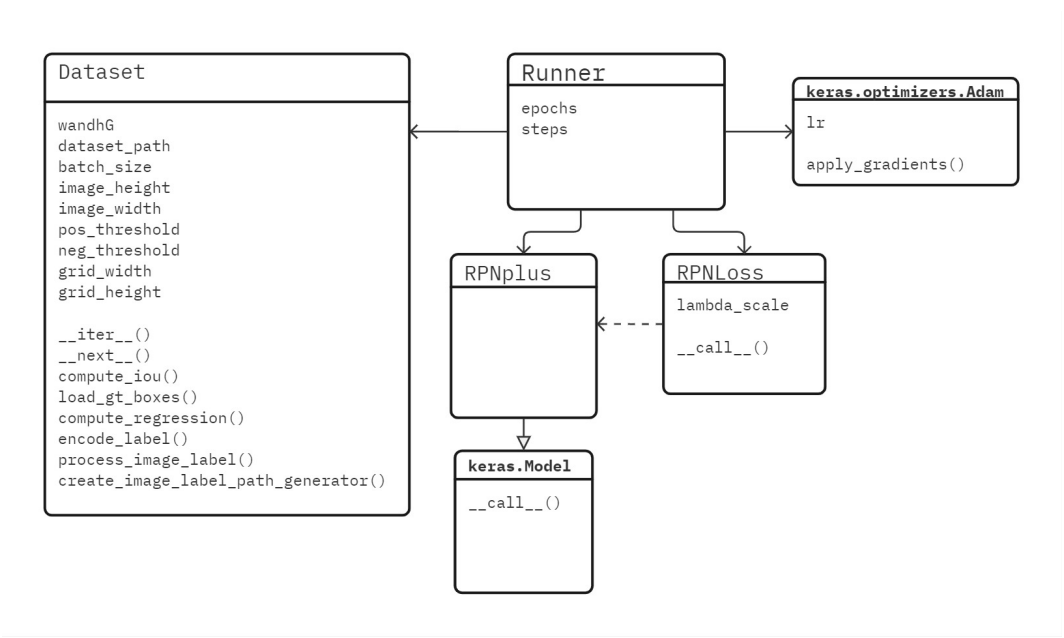

Figure 4.14: Class structure of repository 4 after being refactored.

4.3.5 Applying the architecture to repository 5

In the start file for this project, see Figure 4.15, components are imported from the framework, a `Models` module, and a `Optim` module. No changes have been made

to the model or optimizer components during the refactoring process. The new components, see Figure 4.16, are defined in their respective files and classes.

The task of loading, processing, and batching stored training data is handled by the `prepare_dataloaders_from_bpe()` function if the data is compressed and by `prepare_dataloaders()` if it is not. For this purpose, the functions use some components that are imported from the `torchtext` library. In the end, they both return an instance of `BucketIterator`, which has similar functionality to a data loader. They are both defined in the start file module, making it dependent on all component used by them.

The loss calculation takes place in the `cal_loss()` and the `cal_performance()` functions. These are used within the training loop and the project, therefore, belongs to the `loss_in_runner` category. After they return the loss, the training loop sends their gradients to the optimizer, which then updates the model parameters.

The project has a defined optimizer wrapper class, `ShceduleOptim`. It has an optimizer and alters said optimizers hyperparameters during training, specifically the learning rate parameter.

The `Transformer` class defines the DL model's network structure and its forward pass. Its network structure is an amalgamation of other models, such as the `Encoder` and the `Decoder`.

In `main()` the different components are initialized and then sent to `train()`. The training loop is defined within this function. It also logs the training progress and saves the final model once the process is over.

The refactored project, see Figure 4.16, includes some new component definitions. The `DataloaderFactory` class now handles the creation of `BucketIterator` components. The start file module is no longer dependent on data related components that it does not use, instead it makes a call to the factory which returns an iterable component. The initial plan was to write a wrapper class for the `BucketIterator`, however, it seems that it and the other `torchtext` components have been deprecated. Extending it would, therefore, have been a guessing game. Instead, the `DataloaderFactory` approach was taken. The loss calculation has been moved to the `TransformerLoss` class which, just as the other loss components, uses a model to generate and output on some data, then it calculates the loss based on some difference between the model prediction and some target variable. An instance of this loss class is used in the `Runner` class. Its functionality is that same as the train function from before, however, it has fewer dependencies and is only responsible for the flow and logging of data within the loop.

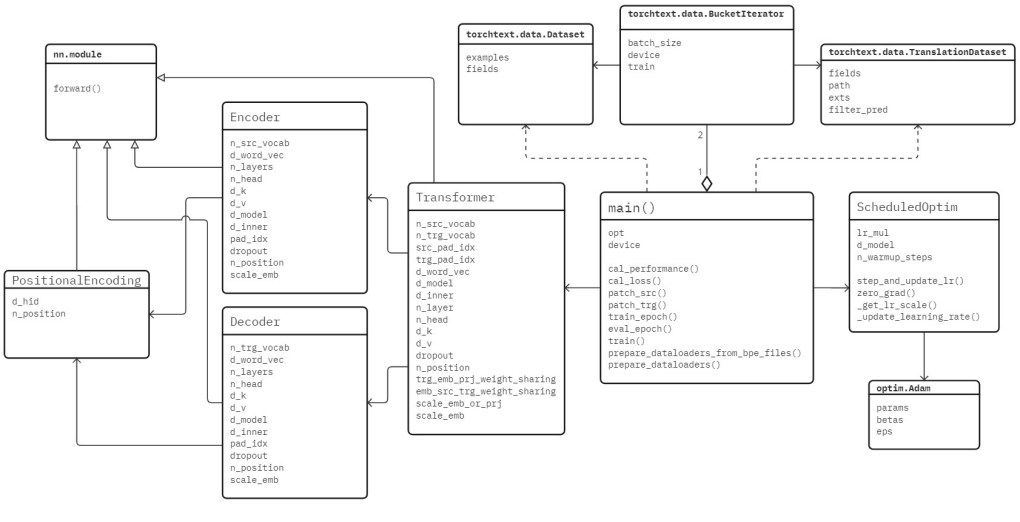

Figure 4.15: Class structure of repository 5 represented as a diagram. The `main()` component encapsulates the model training loop and represent the start file module. It imports some data related components from the `torchtext` library.

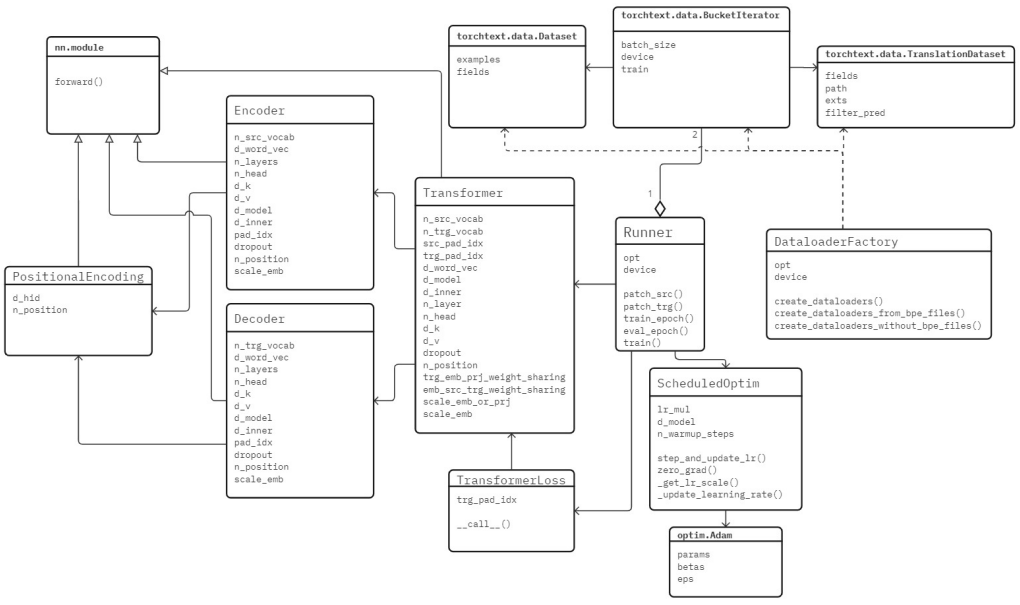

Figure 4.16: Class structure of repository 5 after being refactored.

4.3.6 Applying the architecture to repository 6

This project has separation of some functionality from the start file, mainly the Transformer class. All components in the refactored version, see Figure 4.18, have changes to their state in the original, see Figure 4.17. The new components are separated from the start file.

In this project, the loading, processing, and batching of data is handled by functions defined in the `start_file_module`. All functions in the component listed, including and above `get_batch()` are used for these purposes. The `get_batch()` function itself is the interface between the other functions and the training loop, returning an iterable collection of data batches. It also returns the number of input and number of target batches which it has produced, in effect, making the training loop dependent on these values.

The training loop does not have access to optimizer or loss components, instead it acquires references to these through the `Transformer`'s `train()` function. It is an initialization step for the training. It produces a `train_op` that stores the step-by-step process of generating some model outputs, calculating their loss, calculating the gradients, and finally updating the model parameters using an optimizer. This `train_op` is then used for each iteration of the training loop. This project's loss calculation belongs to the `loss_in_model` category, since it is defined in the `Transformer`.

The training loop is defined in the `start_file_module`, it initializes the model and from it receives the `train_op`. Using a `for`-loop, it iterates over the data, runs it through the `train_op`, logs the progress, and saves the final model parameters.

The refactored version of the project includes new components and changes to previous ones. The data related functions are no longer in the `start_file_module`, instead they have been encapsulated in the `Dataset` class. The training loop can instantiate multiple objects of this data class, whether it be for training or evaluation purposes. The `train()` function has been removed from the `Transformer`. Its functionality is now in the constructor of the `Runner`, i.e. the `train_op` is now defined in the beginning of the training loop component. The model is no longer dependent on the optimizer, since it is now connected to the `train_op` in the `Runner`. The loss calculation is now housed in the `TransformerLoss` component.

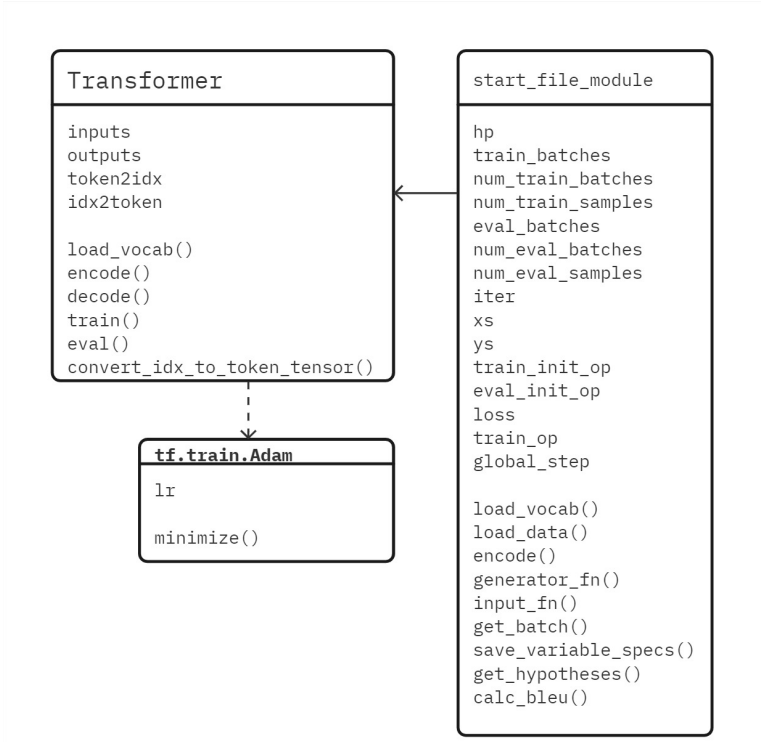

Figure 4.17: A diagram representing the class structure of repository 6. The `main()` function encapsulates the model training loop and represents the start file's module level, as in Figure 4.15.

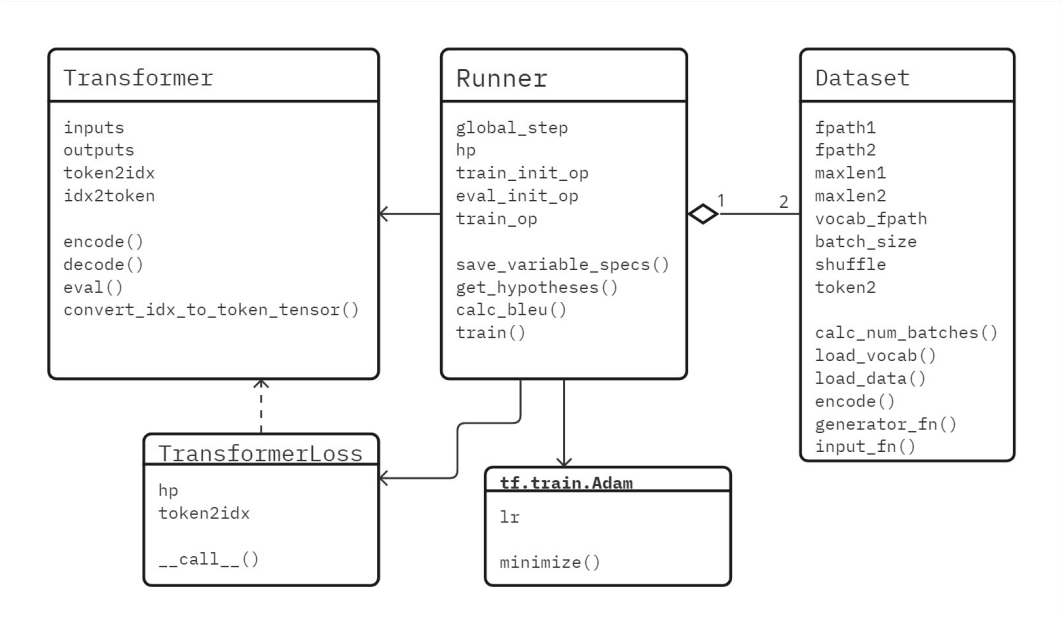

Figure 4.18: Class structure of repository 6 after being refactored.

5

Evaluation and Discussion

This chapter will go through the evaluation of the MODLR architecture and discuss which problems are relevant for the evaluation. It will also go through threats to validity and the value the findings brings to practitioners and researchers.

5.1 Choosing problems for MODLR comparison

All the found problems and themes can be found in Section 4.2. Not all the problems are, however, relevant for how well an architecture works. The architecture main goal is for DL training, which means instances of loading data and preprocessing data is not particularly relevant for the evaluation of MODLR. Therefore, in this section, a discussion of what is relevant and not will occur and what quality attribute each of the problems are related to. However, many of these attributes target similar things, so there will be overlap for what to check between them. The attributes are more described in Section 5.1.2.

The `data is problematic` theme is mainly related to data augmentation, i.e., doing some form of data transformation in the dataset, rather than in the preprocessing step. A problem arises here as the same data augmentation cannot be used for every type of data (modality) and using other dataset, or having enough configurability to solve this problem is therefore required. This means that the theme is related to two attributes: modularity and configurability.

For the `code is faulty` and `lack of unit testing` problems, they both require testability. The reasoning for this is, people have a lower chance of testing code that is not testable, as it is much more impractical. While there are other reasons not to test code as well, like too little time to implement a test suite, and not having experience with developing tests, these hurdle can lessen by testable code, as this would increase the speed and ease of testing [12]. One of the main reason for faulty code, i.e., the theme `code is faulty`, is lack of testing, or at least good testing, as it is meant to show that code is executed correctly for a set of inputs. Therefore, `code is faulty` is related to testability.

The problems `a lot of dead code in dl` and `dl is trial and error` are also related. The assumed reason for why there is dead code in DL, which was also explained by some interviewees, is that people want to test new features, and be able to test things with new setting, to get better models. Therefore, two attributes

are related to these two problems, evolvability, with an emphasis on the sub-attribute changeability, and configurability. The codes need to be evolvable to be able to add new features and swap out the features easily. Configurability is required to test new parameters for the training without the need to change the code.

Another problem related to configurability is the `configuration is not trivial` theme. This theme relates to how hard it can be to use configuration files, as well as how hard it is to set up configuration.

Then there is the problem of `not well defined where to place components`, which is not related to an attribute, but having a standard for how components should be placed should give a good answer to this.

The problem of `amount of abstraction is non trivial` are related to modularity and complexity. While higher abstraction can lead to higher modularity, as putting interfaces or abstract classes, different instances can be used that follows that interface for certain applications, it can also lead to higher complexity [45]. Therefore, the problem is related to both complexity and modularity.

`dl code is hard to reuse` is related to three different attributes: modularity, complexity, and evolvability. Higher modularity leads to more reusable code as being able to swap different modules at will, instead of needing to do a lot of extra setup, means the code is more reusable. Complexity comes in the form that if people do not understand the code, the reuse of the code becomes harder. Evolvability requires flexible code, which in turn, increases the ability to reuse code.

The final relevant problem is `dl code can be highly coupled and hard to reuse`. This is directly related to high coupling, which is connected to the modularity attribute.

5.1.1 Non relevant problems

The problems that are said not to be relevant to the evaluation of MODLR are: `tensor and model shapes are hard to know`, `execution order needs to be modifiable`, `dl code is hard to read`, `model and data are coupled`, and `modality is problematic based on approach`. The main reason for these to not be included are that they are more dependent on implementation details, i.e., things an architectural pattern cannot enforce, not related to training DL models, or problems not a DL developer can fix with code.

The `tensor and model shapes are hard to know` are directly related to the framework used. TensorFlow uses a static computational graph, while PyTorch uses a dynamic. This means that looking at how the tensor shape changes throughout a program is much easier in TensorFlow, as the training does not need to begin before looking at how the tensor shapes changes. In TensorFlow, they have something called *TensorBoard*, which can display this information. PyTorch needs to run through every step of the calculation chain to get this information from the tensors, because the graph is built dynamically. This means recompilations does not need to happen, although making it harder to track the tensor shapes.

The problem of `execution order needs to be modifiable` can be related to configurability, that the configuration chooses how the order should happen. It could also be a modularity issue, replacing the runner with another runner when another execution order is necessary. But the assumed best solution is to instead support hooks and callbacks, functions that get called based on some trigger which can be given to the runner. Under the assumption that this is the best solution, it is mostly based in how the runner is implemented, rather than the architecture. It is assumed to be the best solution, as this gives high amount of flexibility, without the need to write a lot of boilerplate code, in the instance of the modularity approach.

The `dl code is hard to read` can be contributed to a form of complexity, but generally not complexity of the architecture. A highly coupled architecture is probably harder to read, but the main issues from the interviewee were concerned with the code aspects instead. Two things that can make code more readable are to write less code, which does not always work, and to write good comments and documentation of the code [3].

`model and data are coupled` is related to how the output of the dataset needs to fit the size of the input layer of the model. This problem cannot be fixed with an architectural pattern. What can be done is to tell both the dataset and/or the model what shape they should output, or take as input. But this is domain specific (modality) for how well this would work and in some instances, the model is required to be built around the data shape.

The assumed approach for the problem `modality is problematic based on approach` is that modality is only problematic for data collection, data preprocessing, and data augmentation. This means that modality is only relevant for the data augmentation in the case of DL training code, and this step can be related to modularity, as it requires different datasets for the augmentation, but the problems specified from the interviewees did not talk about this, and therefore it is not relevant.

5.1.2 Evaluation attributes

The following attributes are relevant to the problems discussed above: *testability*, *evolvability*, *configurability*, *modularity*, and *complexity*.

Testability relates to how testable code is, i.e., how easy it is to write tests in the code [12]. To have testable code, two other attributes are needed: controllability and observability. Controllability means that tested functionality can be changed by giving different inputs, and observability means that the functionality performed can be observed.

Evolvability is dependent on many attributes, but the main takeaway is how easy is it to add new functionality and change current ones [25]. The main attribute from the evolvability perspective that seems relevant is changeability, how easy modifications can be done without affecting other components in an unexpected way.

Configurability is related to how easy it is to build configuration files and if different configurations causes problematic states [41]. To support configurability in DL, the

code must be controllable, similar to testability, as the different configuration needs to be sent to each of the functions throughout a DL training program.

Modularity is when there are no monolithic classes/files that do everything at once, i.e., separate components are used instead, and these components can be replaced for other components that follow the same interface [10].

There are many types of complexity when it comes to software. For this case, the complexity attribute is, among other things, about how coupled code is, i.e., how many other dependencies does a component have to other components, as well as how much the component does. This is therefore quite similar to modularity. It is also about different metrics which are used to measure how difficult a program is to understand for developers. The metric focused on in the next section is cognitive complexity [7].

5.2 MODLR evaluation

In this section, MODLR will be analyzed based on how it affected the stated attributes. For each attribute, an evaluation will be made on MODLR's effect on said attribute, i.e., is it a positive or negative effect. Examples, from Section 4.3, will be given on how MODLR impacts the attributes.

The testability of each repository was improved in two major and related ways. First, is the extraction and decoupling of an area of responsibility from one big component to multiple smaller ones. This was done by extracting the loss calculation from the model component in repositories 3 and 6; decoupling the loss from the training loop in repositories 1, 2, 4, and 5; separating the training loop as its own component in repositories 1, 2, and 4; and removing the models dependency on the optimizer in repository 6. These changes created decoupled and modular components which could be tested as their own units, independent of other functionality. The second way testability was improved was through the integration of multiple related functions into a single component. This was done by aggregating all the data related function into the dataset components in repositories 2, 3, 4, 5, and 6. It was also done by extracting the training loop related function into a separate component in repositories 5 and 6. These changes improved testability through modularity, i.e. by integrating the related functions into one component their relationships would not need to be redefined when doing integration tests.

The evolvability attribute was improved by increasing the projects' changeability. This was done by defining separated runner components from module level `for`-loops in repositories 1, 2, 3, and 4; decoupling data handling from the training loops in repositories 2, 4, 5, and 6; and isolating the loss from the training loop and model components in all repositories. All these changes had the same effect of increasing changeability, since each change enabled its respective component to be modified without affecting any other components.

Most of the repositories configurability attributes were also improved, but only by a single change. In repository 1, 2, 3, 4, and 6, the training loop was isolated to

a separate component with defined dependencies, i.e. the other components. As a consequence, it could be configured with any implementation of the other components, as long as they had similar interfaces. It could be argued that the refactoring of repository 5 did not have the same result, as its training loop was previously housed in a `train()` function with similar dependencies as the new `Runner` class. Its configurability was therefore not improved by the refactoring.

Modularity is the one attribute that was improved more than any other through applying the MODLR architecture on these repositories. Most steps involved in the refactoring process were about identifying, extracting, and defining components based on some interface set by the architecture. This was done in repository 2, 3, 4, 5, and 6 by decoupling the data handling to a separate component; in repository 1, 2, 4, and 5 by separating the loss from the training loop; and in repository 3 and 6 by isolating the loss from the model component. These decoupled components can now be replaced with other components as long as they follow the same interfaces, thereby increasing the repositories potential for code reuse.

The application of MODLR changed the complexity of the repositories in two ways. First, is the decreased cognitive complexity achieved through removing the loss calculations from the larger components, which was done in repositories 1, 2, 3, 4, and 6. These changes increased the understandability of the larger components, as they now had less code and functionality to be concerned with. The other way complexity was lowered is similar, however, it is more related to the separation of concern. This was done in the ways stated in the for the above attributes, each component has an area of responsibility, e.g., loss calculation or data handling, and by decoupling components based on these responsibilities, it is easier for developers to understand what each component does and does not do. This was done to some degree in all the repositories refactored.

Refactoring these repositories to fit the design of MODLR was a different experience for each repository. Repository 5 and 6, were the projects that needed the least amount of changes to fit the architecture. This might be because they are based on the transformer architecture, which is a relatively new and popular architecture in the DL community. This means that popular implementations of these probably have well-structured code. Repository 2 and 4 required most changes, as they were both written as monolithic structures, all contained within a single file. There was also little separation within those files as to what functionality was related to what aspect of the training process. A pattern could not be found between what framework, was easiest to refactor to fit the architecture. The refactoring of these projects was not a difficult process, and the changes that were needed to be made were easy to find and implement.

To conclude the evaluation of MODLR, the architecture seems to have positive effects on the relevant attributes and is also easy to apply to existing projects. Its applications are not dependent on the framework, DL job, or the model architecture. We believe it should be considered for anyone working on new DL training software.

5.3 Implications to researchers

We believe that the findings of this thesis further shows that the field of software engineering has to continue to focus on DL. This thesis also shows that there is a knowledge gap between DL practitioners and regular software engineers, as it seems that many problems can be fixed with more knowledge in that field.

This thesis also shows that the use of LLMs can be practical for some instances, especially for mining repositories. While LLMs are known for hallucinating, using different techniques, like self-consistency [8] and zero-shot reasoning [20], a pretty good consistency in the answer can be established. It is important to know that such techniques never will be fully correct, and it is always recommended to calculate some sort of expected performance when using such techniques.

There are problems with DL code which should be looked more into, like analyzing the tensors shapes, and debugging, which got mentioned in this thesis. Building tools for these things might be a solution. To be able to statically analyze the computational graph, even if they are constructed dynamically, would be helpful. Debugging tools that also can display data, and tensors, in a good way would also help a long way.

5.4 Implications to practictioners

The main usage of this thesis is to use the MODLR architecture and think about the concepts described in Section 4.2, where the found problems are talked about. Not starting to use a common architecture will likely lead to slower development, as code reuse is harder and ad hoc pattern will be more prevalent. This can also lead to maintainability issues, where coupling gets higher and changing one component might require changes in multiple other components. This does not mean that MODLR is the perfect solution, it is more of one example of an architecture that seems to be good enough for the application. To create other architectures might be better, although having an architecture that is, at least, compatible with what other people do is important, as code reuse will not be as easy otherwise.

As a practitioner, one should also try to start using more software engineering techniques when developing DL training code. Because of how long software engineers has been around, many of the techniques they use are well established and following their way of writing code is probably beneficial. One example is to write good documentation and comments in and around the code, as this will help with readability of the code, and therefore also maintainability. Another quite easy thing to implement is the concept of *Separation of Concern*, which basically says that software components should do one thing and one thing well. Having monolithic god files lowers how readable code is and is often less reusable.

5.5 Threats to validity

This section will go over the threats to validity found within this project. Runeson et al. has a set of four validity threats for case studies (i.e. the interview study), which are construct validity, internal validity, external validity, and reliability [33]. The same validity threats can be discussed for the analysis of MODLR. For the GitHub mining, reliability is not relevant.

5.5.1 Construct validity

For the interviews, there is a threat to construct validity, as we did not go back to the interviewees and ask them to confirm or deny the coding we designated to their interview transcript. We also did not give them the transcript we generated back to the interviewees to proofread what they said. This was purely because too little time was left. Therefore, we might have misinterpreted what the interviewees were saying and making the wrong conclusion about what they said. Some setup was done at the first part of the interview to minimize the interviewee to misinterpret the questions of the interview. The interview always started with explaining the research questions that the interview was meant to answer. The interview also started with trying to have the same definition for some terms that was going to be brought up within the interview.

The analysis of MODLR also have some construct validity threats, as for what should be measured for how good MODLR is. While the interview study gave reasoning to what people have had as problems with DL, actually measuring it is another problem, and deciding which problems are actually relevant to an architectural pattern is not an exact science. But by looking at the presented problems from the interviewees, and connecting them to other research that talks about similar things, a good enough way to rate MODLR could be done.

The mining part was meant to show how people do not have a standard way of writing loss code. While the automatic mining actually is only showing where the loss code is placed, it only shows partially how people write loss code. Another problem with the automatic mining is that one cannot be fully sure how well it represents the truth, as it might mislabel the categorizations. To make the automatic mining more believable, two things were also included with the mining, a manual mining and a test that checks how well the miner performed. The manual mining could show missed categorizations from the original hypothesis, and both the manual mining and the test shows how well the automatic mining was performed.

5.5.2 Internal validity

The interview study can have an internal validity threat from researcher bias, as well as the interview sampling. For the bias, the coding and themes were generated separately between the researchers and compared and discussed when differences occurred. For the sampling, where sampling the people in the same environment would more likely give similar answers, our sampling technique was meant to get a

good enough spread between the interviewees. About half of the interviewees were PhD students that have DL as part of their research topic, and another half were people working with DL within one of the large automotive OEM companies. This gave an even spread of academic DL users and industrial DL users, which should mitigate potential sampling biases.

Another potential problem was that some interviewees were done remote, which could create other biases based on interview environment, where the location could be noisy, the internet connection could be bad, and the face-to-face interaction could be affected. This was tried to be mitigated by always giving an opportunity for the interviewee to choose the location of their liking, so if they wanted to do it remote rather than face-to-face, it was mostly their choice, trying to always find the best opportunity for the interviewee was important. The interviewee were all less than an hour to also make the interviewee more comfortable.

The MODLR analysis required us to refactor already existing code. The problems removed of the code might have been removed based on the MODLR architecture, or it might have been removed just because of how the refactor happened. This can make a threat to the internal validity, as the refactor might include more refactoring than just following the MODLR architecture, which can therefore give biased result to how good MODLR is. There is no easy way to mitigate this more than discussing between the researchers about why some change makes something better, or worse, and being transparent with the reasoning for why the evaluation is the way it is.

For the mining, there are also some threats, mainly the LLM part. The LLM might give biased answer on the categorizations. To see how much bias, we did two extra steps, calculating the F_1 score by picking some random loss code from the dataset, and manual mining of one of the categories. These values show how well the LLM performed in its categorization. We also had a set of code snippets that were meant to be categorized in a certain way, which we incrementally increased based on first results, which we used as a requirement for our model to be able to find for it to be good enough for full data categorization. While there might still be biases left, the F_1 score should give a decent estimate for how good our categorization is.

5.5.3 External Validity

Case studies are generally not generalizable by themselves, as only a specific case is looked upon. Although we did include people from both academia, and industry, we only interviewed people from one specific industry, and only people from one university, which might make the results from the interview not generalizable. What we could see is that other papers have found similar things for some problems within DL, like dead code and modularity issues.

To further look into loss code, the interviewees were split on where loss usually is defined, which goes along with what the findings of the mining showed. Although our sampling for DL code is not general, our focus was on popular and updated repositories instead to look at how people write DL code nowadays. Therefore, the findings from the mining might not be generalizable in the sense of every DL code

ever written. Other than that, no known biases are with the sampling of the dataset.

The MODLR evaluation also have threats in the sense that the refactoring stage will be different between people and the refactored code might be interpreted differently between people. There are also some subjectiveness to what makes code good or bad, which might also give bias to the final conclusion. To mitigate this as most as possible, both the researchers discussed how MODLR solves, or did not solve, problems to have less individual bias. Being transparent is also important to show the readers what affected the conclusion so they can make their own.

5.5.4 Reliability

As mentioned before, for the interview study, there are individual biases for the coding decision of the transcript. This might lead to less reliability, i.e. another researchers might not have the same codes for the interviews. The same thing with the evaluation of MODLR, where personal evaluation was done to say how good MODLR did compare to the found problems. To mitigate these, a discussion happened between the researchers through each step and for the interview coding, each researcher made their own code first to later be compared and discussed if differences arose.

6

Conclusion

In this thesis, three different steps were taken to show that architectural pattern is needed in the context of DL training code and MODLR is a contestant for such a pattern. Firstly, we showed that the loss code is not placed the same way between different projects. The loss code does not have a standard way to be written either, and reusing the loss code, or evolving loss code, has taken a negative impact because of this. This was shown by mining GitHub repositories and using an LLM for classification of said repositories. Upon this, also a manual part, where one category type was manually analyzed, to further see how people write loss code. Secondly, we showed that there are many problems with DL training code. This was done by conducting an interview with people that are both from academia, and industry. The consensus was that there are issues with how DL code is written today, and a majority of the interviewees felt MODLR was a good enough architecture to follow. Third, MODLR was analyzed and evaluated based on the issues found with DL and by comparing them to how actual code is changed when refactored to follow the architectures design. We concluded, qualitatively, that MODLR does address many of the DL problems related to software quality and architecture.

As frameworks, like PyTorch, try to enable object-oriented design and other software engineering concepts, DL has become a software engineering discipline. Claiming that MODLR is the best architecture to use for structure is unknowable until other architectures for DL training code has been created. For future work, we suggest that others try to design different architectures and compare them to MODLR. There should be a focus on how current DL developers can become better software developers also, as sharing code increases productivity in actually developing DL code. Another thing that the interviewees felt was problematic was to statically analyze the code, instead of needing to write many print statements everywhere. Building tools that can also help to more easily show debug information, like what kind of data is loaded, how are tensors are shaped in this part, etc. There should also be some study about how people should write unit tests for DL training code and how to more easily support unit testing in DL, as now the focus only lies on writing test for how well the model predicts.

Bibliography

[1] Zahangir Alom et al. “A State-of-the-Art Survey on Deep Learning Theory and Architectures”. In: *electronics* 8.3 (2019), p. 292.

[2] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*. Vol. 1. MIT press Cambridge, MA, USA, 2017.

[3] Dustin Boswell and Trevor Foucher. *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. " O'Reilly Media, Inc.", 2011.

[4] Pierre Bourque and Richard E. Fairley. *SWEBOK V3.0*. IEEE Computer Society, 2014.

[5] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. “The open-closed principle of modern machine learning frameworks”. In: *proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 353–363.

[6] James Bucanek. “Model-view-controller pattern”. In: *Learn Objective-C for Java Developers* (2009), pp. 353–402.

[7] G. Ann Campbell. “Cognitive Complexity — An Overview and Evaluation”. In: *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2018, pp. 57–58.

[8] Xinyun Chen et al. *Universal Self-Consistency for Large Language Model Generation*. 2023. arXiv: 2311.17311 [cs.CL].

[9] J. W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. 3rd ed. Sage Publications Ltd., 2008.

[10] Lorenzo De Lauretis. “From Monolithic Architecture to Microservices Architecture”. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 93–96. DOI: 10.1109/ISSREW.2019.00050.

[11] Joao Paulo Fernandes, Marcelo Ribeiro, and Marco Tulio Valente. “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation”. In: *Empirical Software Engineering* 21.6 (2016), pp. 2291–2339.

[12] Vahid Garousi, Michael Felderer, and Feyza Nur Kılıçaslan. “A survey on software testability”. In: *Information and Software Technology* 108 (2019), pp. 35–64.

[13] Jiri Gesi et al. “Code smells in machine learning systems”. In: *arXiv preprint arXiv:2203.00803* (2022).

[14] Görkem Giray. “A software engineering perspective on engineering machine learning systems: State of the art and challenges”. In: *Journal of Systems and Software* 180 (2021), p. 111031.

[15] Vladimir Golkov et al. “Q-space deep learning: twelve-fold shorter and model-free diffusion MRI scans”. In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1344–1351.

[16] Ian Goodfellow et al. “Generative adversarial networks”. In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[17] Hadhemi Jebnoun et al. “The scent of deep learning code: An empirical study”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 420–430.

[18] Albert Q Jiang et al. “Mixtral of experts”. In: *arXiv preprint arXiv:2401.04088* (2024).

[19] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: (2017). arXiv: 1412.6980 [cs.LG].

[20] Takeshi Kojima et al. “Large language models are zero-shot reasoners”. In: *Advances in neural information processing systems* 35 (2022), pp. 22199–22213.

[21] Frank Krüger. “Activity, Context, and Plan Recognition with Computational Causal Behaviour Models”. PhD thesis. Dec. 2016.

[22] Kevin Fu Yuan Lam. “Confidence Intervals for the F1 Score: A Comparison of Four Methods”. In: *arXiv preprint arXiv:2309.14621* (2023).

[23] Lucien Le Cam. “The central limit theorem around 1935”. In: *Statistical science* (1986), pp. 78–91.

[24] Addi Malviya-Thakur and Audris Mockus. “The Role of Data Filtering in Open Source Software Ranking and Selection”. In: *arXiv preprint arXiv:2401.10136* (2024).

[25] Herwig Mannaert, Jan Verelst, and Kris Ven. “Towards evolvable software architectures based on systems theoretic stability”. In: *Software: Practice and Experience* 42.1 (2012), pp. 89–116.

[26] Humza Naveed et al. “A comprehensive overview of large language models”. In: *arXiv preprint arXiv:2307.06435* (2023).

[27] Katie O’Leary and Makoto Uchida. “Common problems with creating machine learning pipelines from existing code”. In: (2020).

[28] Josh Patterson and Adam Gibson. *Deep learning: A practitioner’s approach*. " O’Reilly Media, Inc.", 2017.

[29] Dewayne E Perry and Alexander L Wolf. “Foundations for the study of software architecture”. In: *ACM SIGSOFT Software engineering notes* 17.4 (1992), pp. 40–52.

[30] Danilo Pianini and Alessandro Neri. “Breaking down monoliths with Microservices and DevOps: an industrial experience report”. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021, pp. 505–514. DOI: 10.1109/ICSME52107.2021.00051.

[31] P Pradhyumna, G P Shreya, and Mohana. “Graph Neural Network (GNN) in Image and Video Understanding Using Deep Learning for Computer Vision Applications”. In: *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*. 2021, pp. 1183–1189. DOI: 10.1109/ICESC51422.2021.9532631.

[32] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors." In: *Nature* 323.6088 (1986), pp. 533–536.

[33] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14 (2009), pp. 131–164.

[34] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications Ltd, 2013. ISBN: 978-1-44624-736-5.

[35] Johannes Sametinger. *Software engineering with reusable components*. Springer Science & Business Media, 1997.

[36] Yutaka Sasaki. "The truth of the F-measure". In: *Teach Tutor Mater* (Jan. 2007).

[37] Kunal Sawarkar. 2022.

[38] Douglas C Schmidt et al. *Pattern-oriented software architecture, patterns for concurrent and networked objects*. Vol. 2. John Wiley & Sons, 2013.

[39] David Sculley et al. "Hidden technical debt in machine learning systems". In: *Advances in neural information processing systems* 28 (2015).

[40] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6.1 (July 2019), p. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0.

[41] Reinhard Tartler et al. "Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem". In: *Proceedings of the sixth conference on Computer systems*. 2011, pp. 47–60.

[42] Daniel W. Turner. "Qualitative Interview Design: A Practical Guide for Novice Investigators". In: *The Qualitative Report*. Vol. 15. 3. 2010, pp. 754–760. DOI: 10.46743/2160-3715/2010.1178.

[43] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[44] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[45] Stefan Wagner and Florian Deissenboeck. "Abstractness, specificity, and complexity in software design". In: *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering*. 2008, pp. 35–42.

[46] Hironori Washizaki et al. "Studying Software Engineering Patterns for Designing Machine Learning Systems". In: *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. Dec. 2019, pp. 49–495. DOI: 10.1109/IWESEP49350.2019.00017.

[47] Youzi Xiao et al. "A review of object detection based on deep learning". In: *Multimedia Tools and Applications* 79 (2020), pp. 23729–23791.

A

Framework Usage

Framework usage data, collected at 2024-02-20.

A.1 Stackoverflow survey

The 2023 Stackoverflow developer survey, found at `https://survey.stackoverflow.co/2023`, is a survey that all kind of developers can participate in each year. It can show trends in what programming languages are used, as well as other technologies relevant for development. They have a section called "Other frameworks and libraries" that talks about a plethora of frameworks used when developing code. Here we can see multiple DL frameworks that are used by different developers:

1. TensorFlow - 6405
2. *Scikit-Learn - 6339*
3. PyTorch - 5884
4. *Opencv - 5455*
5. Keras - 2839
6. *Hugging Face Transformers - 1852*
7. JAX - 641

The italic items are not *general* DL frameworks, but only DL related.

A.2 GitHub repo search

GitHub is one of the largest code distribution sites and can give an overview on how popular different things are. By using the repository searcher, one can give a good overview on how many repositories are using the different deep learning frameworks. One can get access to the search at `https://github.com/search` The only problem is that the repository does not disapear, so newer frameworks may have a disadvantage if only looking at these numbers. But looking at the quantities of repositories found with the search tool, we get:

1. TensorFlow - 150k

2. PyTorch - 126k
3. Keras - 64.8k
4. Chainer - 53.4k
5. torch - 28.4k
6. Caffe - 11.3k
7. FastAI - 6.7k
8. ONNX - 4.6k
9. MXNet - 2.7k
10. Theano - 2.3k
11. Deeplearning4j - 544, DL4J - 817
12. Microsoft Cognitive Toolkit - 20, CNTK - 557

We can see here that searching for TensorFlow and PyTorch, there are a lot more repositories than for the other frameworks.

A.3 Grey papers

There are also multiple grey papers that talk about popularity for the different frameworks. Looking at one article from Viso.ai (`https://viso.ai/deep-learning/deep-learning-frameworks/`), we get this list:

1. TensorFlow
2. Keras
3. PyTorch
4. MxNet
5. Chainer
6. Caffe
7. Theano
8. Deeplerning4j
9. CNTK
10. Torch

A.4 Pip packages

Python is one of the main programming languages for developing ML/DL models. Looking at how often pip (a python package manager) packages are installed

can indicate the popularity of those frameworks. This can be seen at `https://piptrends.com` Looking at the following table, we see that TensorFlow and PyTorch are the most popular packages:

Name	Stars	Forks	Updated	Weekly downloads
PyTorch	75.5k	20.9k	Today	5.2M
TensorFlow	180k	89.5k	Today	4.3M
Keras	60.3k	19.5k	Today	3.5M
FastAI	25.2k	7.5k	Today	110k
MXNet	20.7k	6.9k	Today	90k
Theano	9.8k	2.5k	1 month	87k
Chainer	5.8k	1.4k	2 years	5.4k
CNTK	17.4k	4.4k	2 years	425

B

Interview Guide

Who we are
Sebastian Johansson & Behroz Razaq
Studies Software Engineering Master at Chalmers
This interview is for our Master's Thesis about "Studying (D)MoTR: An architectural pattern for deep learning code in the automotive industry"

Purpose of Study
The purpose of this study is to find how useful D(MoTR) is, while also look at how to write training code for DL systems, where modularity and flexibility is the main goals.

Confidentiality
The contents of this interview should not be about things that are personal or private in nature.
However, we will not include your name in the final report anyway and the things you say will be under a label based on occupation and experience.

Structure of the interview
The interview will be that we first ask you some background questions to be able to more easily categorize and compare what you have said.
Then we will ask if you know, and have the same definitions as us, about some concepts.
Then we get to the interview questions, and in the final stage, you are able to ask us any questions you find relevant about the interview or its contents.
An interview takes around 45 minutes.

Final Questions
Do you have any final questions before we start the interview?
Is it okay that we record the interview?
Start Recording

Research Questions:
RQ1: What are the current design problems when structuring, writing, and main-

taining deep learning code bases?
RQ2: What aspects of the deep learning development cycle affect the code base structure?
RQ3: What are the impacts of having multiple modalities in the same deep learning code base?

Interview Questions

Definitions

1. Do you know what a modality, training task/loss, runner/training loop is in the context of ML?
    - If yes modality,
        (a) Please give a short definition
    - If yes training task,
        (a) Please give a short definition
    - If yes runner/training loop,
        (a) Please give a short definition

Background

2. What is your background with deep learning and/or machine learning?
    (a) What is your connection to the framework _____? #Interviewee from the company
3. What is your background with the field of software engineering beyond DL/ML?
4. Have you worked with a DL project that is supposed to be able to handle multiple modalities at once?
    - If yes,
        (a) What was the experience and where there any challenges supporting such paradigm?

Main Questions

From now on, when we talk about task, we will refer to a training task, when we talk about runner, we will mean the training loop.

DL development

We will ask you a couple of questions on how you would build a DL training framework and give you a couple of constraints along the way.

5. Starting with no constraints, how would you write a DL training pipeline, what would the structure look like?

- Would your approach change anything if it also needed to also support ____ ?
  (a) Multiple modalities
  (b) Expandability (evolvability)
  (c) People that do not want to program should be able to use predefined parts of the code, like different models, for more than just one project

Experience

6. From your experience in writing DL code, have you encountered any particular problems when examining or developing such code?
   (a) Structural problems, like modularity issues, spaghetti code, etc.
   (b) Maintainability/Evolvability problems, like shotgun surgery, dead code, hard to read, etc.
7. On a more component depth for training a DL model, are there any general challenges when it comes to ____ ? If so, can you explain why, or why not, these challenges affect the other components list out components from a code perspective?
   (a) Data collection
   (b) Data preprocessing
   (c) Model Definition
   (d) Training
   (e) Testing
8. We have mined ____ open source DL repositories from GitHub and found that if we only look at the model, task, and runner, the most used architecture is that of ____.
   (a) Is this what you have seen also?
      - If no, what is the architecture you have seen the most?
   (b) What is your thought about this architecture? What does this architecture do good and what does it do bad?

(D)MoTR
Background
Give paper on (D)MoTR definition
We will now give a definition of an architectural pattern called (D)MoTR which stands for *Dataset, Model, Task, Runner*, and then ask some questions around this.

- **Dataset** - The data preprocessing and loading step.

- **Model** - The model definition, i.e. the definition of the network layers.
- **Task** - The loss calculation/training task.
- **Runner** - The training loop

The main goal of (D)MoTR is to abstract the four components as their own thing. The main focus on the (D)MoTR architecture is that Task should also be abstracted and not a part of either the Model or the Runner, which it seems to usually be. Realizing that the task is not inherently dependent on Model, but could be used, even with different Models, this pattern follows the software pattern of Separation of Concern and could help with modularity problems when it comes to DL training.

9. Are there any unclarities around (D)MoTR?
10. What is your thought about (D)MoTR? Do the statement make sense?
11. Have you ever seen this kind of architecture before? In that case, could you give us some context about where it was used?
12. Do you see any problems or ambiguities with the (D)MoTR architecture?

Wrap up
Try summarizing some of the key points of the interview.
Is there anything else you would like to add?
Stop Recording

More Information
To get in touch with us, you can mail us via:
behrozr@chalmers.se
sebasjoh@chalmers.se

C

LLM Prompts

C.1 Prompt for loss related code

```
Asses the code snippet and determine if it contains the concept of
↪ loss from deep learning.
The code is written in python, and it uses either the PyTorch or
↪ TensorFlow frameworks.
Only consider the contents of the given code snippet and do not make
↪ any assumptions on how the code is structured outside of the code
↪ snippet.

## Descriptions of loss as a concept
Loss is a metric measuring a deep learning model's output error.
The loss calculation is a comparison, where a model's output, on some
↪ input data, is compared to a target variable.
In other words the calculation can be defined as some difference
↪ between a prediction and a ground truth.
The loss calculation can be done in place or retrieved by a loss
↪ function
When optimizing a deep learning models parameters this is done with
↪ the loss.
The loss can also be used as a metric for logging a models
↪ performance over time.
There are some synonyms to loss, which are equivalent if they follow
↪ this description.
These synonyms include, but are not limited to: Cost, Objective, and
↪ Error.

It can be used to update model
As long as a loss is retrieved either from a function call which was
↪ given a prediction and some target variable, that is a loss
↪ calculation.

## Code snippet
```python
{code_contents}
```
```

```
```

Response format
Provide your assessment based on the given criteria, and respond in
↪ the following format:
```vbnet
Explanation: [Brief justification concerning the presence or lack of
↪ the loss calculation within the analyzed code.]
Decision: [True, if the code snippet follows some or all of the
↪ description, or False, if it does not.]
```
Remember to strictly adhere to this response format without including
↪ any supplementary details.
```

## C.2 Prompt for loss definition in code

```
Question:
Determine if the code snippet contains ANY self defined losses.
The code is written in Python and is based on either the PyTorch or
↪ TensorFlow frameworks.
Do not think about the code outside of PyTorch or TensorFlow and the
↪ given code snippet.
Do not make assumption about the rest of the code base except the
↪ code snippet.

Descriptions of self defined loss:
Loss is what a DL system uses to optimize the model's weights and
↪ biases.
The loss is a mathematical function that calculates how wrong the
↪ model was from the actual answer.

To be categorized as a self defined loss, the loss that is being
↪ calculated has to be calculated in this specific code snippet.
If the snippet ONLY uses another function to calculate the loss, it
↪ is not a self defined loss, but rather a usage of loss.
This means if there are at least one mathematical operations between
↪ two or more values from a predefined function, it still count as
↪ a self-defined loss.
An example is shown below:
`MSELoss(x, y) # not a self-defined loss calculation`
`MSELoss(x, y) + L1Loss(x2, y2) # is a self-defined loss calculation`
If the snippet uses branching, i.e. if statements, for deciding the
↪ loss in some way, just assume it is a self defined loss, as even
↪ though they might only use pre defined functions, they still have
↪ added some of their own logic to it.
```
```

```
A self defined loss can also be calculated using none of the
 ↪  predefined loss functions.
These functions may be implementing existing (well defined) loss
 ↪  algorithm, but these should still be seen as self defined loss.
If it can be deduced that this is a loss calculation directly, or it
 ↪  is a well established way to calculate loss, assume it is
 ↪  self-defined loss.
Otherwise, look at the rest of the code and look if it is used as a
 ↪  loss in the code, i.e. it updates the model's weights and biases.
Example:
`x - y # is a self-defined loss calculation if the result is meant to
 ↪  be used for updating the model's weights and biases`

Code snippet:
```python
{code_contents}
```

Response:
Think step-by-step and analyse the code.
On the final line of your answer, your final decision should be
 ↪ generated following this template:
`Decision: [True, if the code snippet includes a loss definition,
 ↪ False otherwise]`
```

C.3 Prompt for training loop in function

```
Asses the function and determine if it contains a training loop for
 ↪  deep learning models.
The code is written in python, and it uses either the PyTorch or
 ↪  TensorFlow frameworks.
Only consider the contents of the given code snippet and do not make
 ↪  any assumptions on how the code is structured outside of the code
 ↪  snippet.

## Description of the model training loop
A model training loop iterates over data and updates the model's
 ↪  parameters based on the loss.
It usually have these four components:
 - Looping through mini-batches: Iterate through individual batches
  ↪  of inputs and targets, carrying out the subsequent processes.
 - Forward pass: Calculate the predicted outputs by feeding the
  ↪  current batch of inputs through the model. Compute the
  ↪  associated loss value by comparing these predictions against
  ↪  ground truth labels.
```

```
 - Backward pass: Propagate gradients throughout the entire model,
  ↪ starting at the topmost layer and moving backwards towards
  ↪ earlier layers, accumulating errors caused by incorrect
  ↪ predictions.
 - Weight update: Apply optimization algorithms, such as stochastic
  ↪ gradient descent (SGD), Adam, RMSProp, or Adagrad, adjusting the
  ↪ model's parameters according to calculated gradients.

We only care about a full training loop.
If the function is only a step which should be performed during a
 ↪ training loop, it should not be categorized as a training loop.

## Function code snippet
```python
{code_contents}
```

Response:
Think step-by-step and analyse the code.
On the final line of your answer, make a decision based on your
 ↪ analysis.
The final decision should be generated following this template:
`Decision: [True, if the function is some sort of loop, or iterator,
 ↪ over data that trains a model based on that data, False
 ↪ otherwise.]`
```

C.4 Prompt for model definition in function

```
Asses the function and determine if it defines deep learning model's
 ↪ structure or if it is a forward function.
Both of these components are described below.
It does not have to do both.
The code is written in python, and it uses either the PyTorch or
 ↪ TensorFlow frameworks.
Only consider the contents of the given function and do not make any
 ↪ assumptions on how the code is structured outside of the code
 ↪ snippet.

## Description of the model definition
Code that defines a machine learning model specifies the architecture
 ↪ of the neural network, including details about the number and
 ↪ types of layers used, their dimensions, connectivity patterns
 ↪ between layers, and specific hyperparameters such as kernel
 ↪ sizes, strides, padding modes, etc.
```

```
Essentially, defining the model involves creating a blueprint of the
 ↪  neural network structure, which determines the way information
 ↪  flows through the network during training and testing phases.

In popular deep learning frameworks such as PyTorch, TensorFlow, and
 ↪  JAX, models are often defined using classes, modules, or
 ↪  functions.

## Description of the forward function
The forward function takes in input data (usually represented by
 ↪  tensors) and passes them sequentially through each layer of the
 ↪  neural network model.
Each layer applies its own set of weights, biases, and activation
 ↪  functions before passing on the result to the next layer.
This process continues until the final layer produces output
 ↪  predictions.
These predictions are then returned by the forward function.
Note that this function computes only the prediction from the input
 ↪  data; it doesn't involve computing any losses nor updating
 ↪  parameters via backpropagation.

## Function code
```python
{code_contents}
```
Response:
Think step-by-step and analyse the code.
On the final line of your answer, make a decision based on your
 ↪ analysis.
The final decision should be generated following this template:
`Decision: [True, if the function follows either of the descriptions,
 ↪ either a forward function or model definition. Both are not
 ↪ needed for a True decision. False, only if neither component is
 ↪ found.]`
```

D

Calculating the f_1-score

D.1 Sampled repositories

Index in corpus	Loss location (path:row)	Actual	Prediction
33	labml_nn/gan/wasserstein/___init___.py:104	unk	unk
58	src/prototypical_loss.py:37	unk	unk
77	tasks/adding_task.py:107	unk	unk
111	train_MoEP_AE_macro_F1_scores.py:258	run	run
113	train_model.py:17	m&r	not
133	index.py:108	run	run
139	B-RBM.py:74	m&r	m&r
187	4-1.Seq2Seq/Seq2Seq.py:81	run	not
214	sockeye/loss.py:348	unk	mod
220	learned_gaussian_diffusion.py:122	mod	mod
243	NAS-Bench-201/losses.py:58	unk	unk
263	torch_geometric_signed_directed/utils/signed/prob_balanced.py:23	unk	unk
266	neuralmodels/sequence/grucopydecoder.py:144	mod	mod
268	toolbox/losses.py:20	unk	mod
292	tutorial-contents/406_GAN.py:51	unk	not
296	niftynet/layer/loss_gan.py:42	unk	unk
298	tfprob/gan/loss.py:20	unk	unk
324	networks/capsulenet/capsule_net.py:98	unk	unk
334	examples/lstm_time_series_regression/lst_regression.py: 50	mod	mod
336	main.py:57	unk	unk
363	lstm_architecture.py:233	run	run
364	nsm/graph_factory.py:1027	unk	unk
370	lab3_RNN/tf_utils.py:11	unk	not
395	util/loss_and_optim.py:14	unk	unk
423	rbm.py:152	mod	mod
439	rbm.py:735	m&r	not
449	model.py:131	mod	mod
459	metrics.py:33	unk	unk
531	motif-cnn/metrics.py:6	unk	unk
533	models/radio_resource_management.py:39	unk	unk

Table D.1: Sampled repositories along with their actual and predicted labels.

D.2 Multi-class confusion matrix

	run	mod	m&r	unk	not
run	3	0	0	0	1
mod	0	5	0	0	0
m&r	0	0	1	0	2
unk	0	2	0	14	2
not	0	0	0	0	0

Table D.2: Multi-class confusion matrix filled with validation results.

Category	tp	tn	fp	fn
run	3	26	0	1
mod	5	23	2	0
m&r	1	27	0	2
unk	14	12	0	4
not	0	25	5	0

Table D.3: Individual confusion matrix metrics for each category.

	f_1-score
run	0.86
mod	0.83
m&r	0.5
unk	0.88
not	0
Weighted mean	0.83

Table D.4: All calculated f_1-scores.

Sample size	Confidence level	f_1-score	Confidence interval
30	90%	0.83	0.8 ± 0.11

Table D.5: Confidence interval for f_1-score calculated using the Wilson direct method.