CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Debloating Machine Learning Systems

Master's thesis in Computer science and engineering

Mihkel Sildnik

Yan Wang

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Debloating Machine Learning Systems

Mihkel Sildnik
Yan Wang

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Debloating Machine Learning Systems
Mihkel Sildnik, Yan Wang

Supervisor: Ahmed Ali-Eldin Hassan, CSE
Co-Supervisor: Philipp Leitner, CSE
Examiner: Ivica Crnkovic, CSE

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2021

Debloating Machine Learning Systems
Mihkel Sildnik
Yan Wang,
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The size and complexity of software systems tend to grow over time. As a side-effect, this increase can potentially lead to the accumulation of unused code, also known as bloat. In this study, we assess the prevalence of bloat in Machine Learning (ML) systems, give an overview of a selection of existing debloating tools and study their applicability to workloads in this field. In order to assess the tools, we run a number of experiments on five different ML models, that are written using the PyTorch library. The debloating target is a Docker image containing the ML library and other dependancies required besides the model itself and the dataset. Cimplifier is the only tool we test that was able to generate working images. While the literature in the field of debloating suggests a possible reduction in metrics such as memory usage or power consumption, our testing only shows a reduction in storage size. Most of the removed files are parts of the Nvidia CUDA toolkit and the Intel Math Kernel Library. To summarize, Cimplifier gives promising results when it comes to storage reductions (around 50%) but is unable to impact other metrics such as GPU usage, power consumption or workload runtime.

Keywords: Computer, science, computer science, machine learning, bloat, debloating, project, thesis.

Acknowledgements

We would first like to thank our thesis supervisors, Asst. Prof. Ahmed Ali-Eldin Hassan and Asst. Prof. Philipp Leitner. Thanks for all the guidance and support during the entire thesis and for always being open to questions. Their quick and pinpoint feedbacks always directed us towards the right path.

We would also like to express our sincere gratitude to our examiner Prof. Ivica Crnkovic, for his interest in this project and for taking the time to examine this thesis.

Thanks to the original Cimplifier development team, including Mohannad Alhanahnah, for kindly sharing the tool with us, which is used extensively during this project.

Finally, we want to thank our families and friends for always being supportive during the whole process.

Mihkel Sildnik & Yan Wang, Gothenburg, June 2021

Contents

List of Figures

List of Tables

1
Introduction

The size and complexity of software systems tend to grow over time. Software developers have the understandable goal of trying to create a product that appeals to the highest number of people possible. This naturally leads to ever-increasing code-bases as each update adds in new features, customization options, and patches. As a side-effect, this increase in code-bases potentially leads to the accumulation of unused code, also known as bloatware or code-bloat. This thesis aims to study code-bloat in Machine Learning Systems, quantifying the bloat impact and how code-bloat can be removed.

1.1 Overview

Code bloat is becoming a significant issue in most large software projects, including Machine Learning (ML) code-bases. For example, TensorFlow, which has been an open-source project for five years, has close to 4 Million lines of code (LoC) (the growth can be seen in Figure 1.1), with each new release increasing the amount of bloat [1]. Even worse, the amount of bloat added is accelerating, and code is rarely removed. At the heart of our thesis is the hypothesis that a sizeable part of these code-bases is technical debt, inefficient code, or largely unnecessary features (code bloat). This has also been identified by Sculley et al. [1], who discussed the software's hidden technical debt and model inefficiencies in ML software and the algorithms they run. Code bloat is a problem since it negatively affects performance and leads to waste of resources and increased risks of security vulnerabilities. In addition, it has been shown recently that energy consumption can be reduced by up to 60% by just removing bloat[2].

While bloat by itself is bad, the problem is made worse by the rising popularity of ML workloads in all ICT industries. Cloud-based AI and ML workloads are notorious for their energy consumption [3], with recent reports showing that the training of a single average Deep Neural Network model can produce carbon emissions similar to the emissions produced by 2 to 10 cars operating for 15 to 20 years each [4].

Even moderate scaling inefficiencies on the order of 5-10% accumulate across many training steps and training runs, and further increase the enormous carbon footprint of deep-learning and its associated environmental impact [5].

[1] The figure was created by using git-of-theseus (https://github.com/erikbern/git-of-theseus) on the github repository of TensorFlow (https://github.com/tensorflow/tensorflow

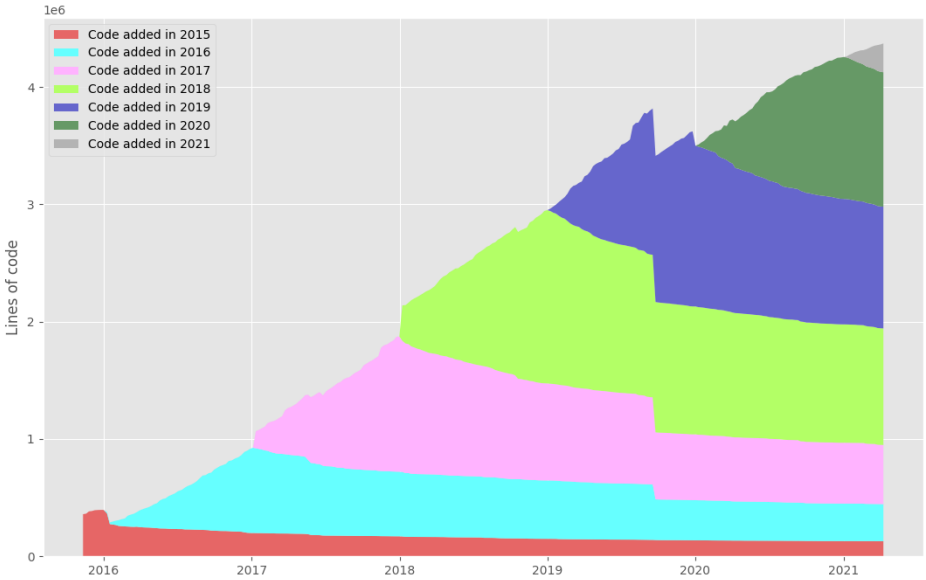


Figure 1.1: Lines of code in TensorFlow since creation

In this thesis, we start by evaluating the existing debloating technologies on various libraries and identify cimplifier as a tool for debloating the Machine Learning project.

1.2 Problem statement

Machine Learning(ML) systems have a special capacity for incurring technical debt, as shown by Sculley et al. [1]. Besides, all the maintenance problems of traditional code, the ML systems have their own unique challenges. This makes debt difficult to detect because the bloat exists at a system level rather than only on the code level. Traditional debloating techniques may end up corrupting or invalidating binaries by not considering the fact that ML system behavior is influenced by their data input. In this case, fixing code level debt is insufficient to handle and reduce system-level debts.

In our thesis we try to answer the following three Research Questions (RQs):

RQ1: How prevalent is code bloat in existing ML systems?

RQ2: Can existing debloating researches be applied to ML systems? If yes, what kind of changes are required, and how much bloat can get removed in practice?

RQ3: Can the Machine learning system still able to maintain reasonable service integrity after debloating, in terms of storage, CPU, memory and energy consumption?

One of the key contributors to code complexity and technical debt in classical software engineering settings is dependency debt, J. David et al. [6]. In ML systems, data dependencies cost more than code dependencies. The data dependency problem can be divided into different categories.

- Unstable data dependencies because ML systems often consume signals as input features produced by other systems, but the input signal could change over time.
- Underutilized data dependencies, which provides little incremental modeling benefit.

Another problem is how readily the currently existing debloating tools can be used on ML systems. Since we do not find any relevant paper on debloating ML systems, it is interesting to see how and if those tools can be applied to the tested ML systems. In this thesis, we first discuss how existing tools, including docker-slim, wholly!, and Cimplifier can work with ML systems and run experiments in order to answer our Research Questions.

The final problem we identify for this project is the removal of the bloated part while still keeping the Machine Learning system integrity intact. This issue is interesting to investigate because for traditional software we can introduce test cases to cover as many logic branches needed for business logic and start debloating based on that. However, this approach to the problem does not fit Machine Learning systems, considering that the library usage depends on the specific models or datasets used and that there are no exact results to test against for the results as there is inherent randomness in the training process.

1.3 Approach and results

A short overview of our approach is that we conduct a review of existing tools, conduct an overview of popular python machine learning libraries, construct some machine learning workloads and apply the found existing debloating tools on them. We also analyze the results as well as work on improving an existing tool.

1.3.1 Evaluation criteria

A way to check the integrity of a machine learning system after debloating is to compare the output of the system to the original. This check would have to take variance into account since there is randomness inherent to most training methods. Since different models use different parts of the library, it might mean that it is not possible to guarantee a new model's integrity with tests that use a different type of model. On the other hand it might be possible to create a large model that uses a very large part of the library, although this might prove to give a false sense of security as in this case the debloating is not as extensive as it would be in normal circumstances.

In order to gauge the applicability of different tools, we create and try to debloat a variety of models. We compare the output of both the original and debloated versions in order to determine if the process has caused adverse effects. Aside from breaking the model it is possible that the debloating process removes optimizations or otherwise slows down or hinders the model.

1.3.2 Challenges

In the course of our thesis we tackle a multitude of different problems. Some of the problems are:

- It is difficult to say if there is a difference in the bloat of different machine learning libraries or how to assess that before we have actually tried to debloat them.
- When it comes to workloads, we initially had no access to hardware, that can run large and modern models, since we originally had access to only weaker GPUs, later on we got access to a stronger workstation as well as ran instances on AWS.
- Existing tools might not work in our case. Reading literature on the topic of debloating, it was quite common for the tools or approaches to work on CC++ code or need the source code to be in a language that can be compiled into LLVM bitcode. This makes it incompatible for us since we are looking at libraries written infor python.
- Even if a tool would work for us we might not get access to the tool or we might get access too late in the process.
- According to our knowledge no paper exists that talk about how to debloat in ML systems so existing tools might work for the authors own goals but might not work for our specific case. It is hard to say if it would be possible for us to improve the tools capabilities or if we even get access to the source code of the tool.
- Many of the tools have been developed within academic context. Hence, many of the tool developers moved on, providing no or very little support. Many of the codebases are academic projects with little documentation, and are less mature.

1.3.3 Main contributions of the thesis

The first main contribution of our thesis is that it gives an overview of existing tools for debloating in general as well as their applicability to machine learning. This can help the reader quickly navigate the tools to better find something suitable for their own needs.

Another contribution is the analysis of existing tools on machine learning workloads, which gives an insight into if these tools work for this use case and how much improvement they give in different areas such as storage size, memory usage, GPU utilization and power consumption.

1.4 Outline

Chapter 2 gives an overview of the more popular Machine Learning systems, explains the concept of debloating, an introduction to the models and software used in the thesis and an overview of existing debloating tools. Chapter 3 presents the research methodologies used and goes over the evaluation environment. The results of the study are presented in Chapter 4. The study's threats to validity are discussed in Chapter 5. Finally, the paper ends with an conclusion of the thesis work and the suggusted future work in Chapter 6.

2

Background and previous research

The problem of code bloat has seen significant interest over the past years[7, 8]. It has long been known that bloated software results in much worse security [9], worse performance [10], and higher energy consumption [11]. While bloating in ML systems has not received the attention it should, in terms of techniques to remove the bloat in the system, some work has been done on big data systems at large[12, 7]. Extra research is required as Machine Learning software has its own unique set of problems when it comes to code-bloat[1]. Furthermore, a lot of the techniques used for debloating require a set of input/output pairs [13], comprehensive test suite[14] or some other way of specifying the requested feature set which is not realistic for a general-purpose Machine Learning toolbox.

Bloating can be due to a large number of causes within ML systems and the models they run. Sculley et. al. [1] discuss the following main categorised of bloat which they refer to as technical debts:
1. ML-system anti-patterns. Design anti-patterns exist in most ML-Systems, for example TensorFlow and PyTorch [15]. In order to use generic packages which are not a good fit for the problem, glue code is added; Mis-management of data introduces data pipeline jungles; Dead-end experimental code in the production code-base due to writing experimental code and rarely cleaning it afterwards; Lack of strong abstractions that can lead to poor design choices in the system [16]. Wrong data types, multiple languages used during system development creates more code issues, worsened when the main method of testing is based on prototyping.
2. Configuration bloat. In many machine learning systems developed today, the lines of code is eclipsed by the lines of configuration used.
3. Costs due to being always online. Cloud-based ML systems are built to continuously react to external input and stimulus, e.g. external input data streams. Unfortunately, they are also typically configured with fixed thresholds that are used to classify things into True or False.
4. Feedback loops. Hidden feedback loops can appear in ML systems where two systems affect the same entity, due to the poor software design of many of the subsystems, e.g. choosing to show two different parts of the same page, can end up affecting each other. In other cases, the ML systems and models are designed to influence the selection of its own future training data, again leading to bizarre failures and behaviours, e.g., the Microsoft Chatbot that became racist in one single day on Twitter.
5. Data dependencies bloat. The concept of data dependencies is new, and no tools exist today to analyze them. There are tools using static code analysis to identify

many code dependencies. However, in most ML systems, it is quite easy to build large data dependency chains that can be difficult to untangle.
One interesting approach used for debloating SW is to use ML algorithms to aid the debloating of SW. Two recent general software debloating systems that employ machine learning are CHISEL[13] and HECATE[17], those two tools will be discussed more detail in section 2.5.

2.1 Machine learning systems

Machine learning usually involves methods that analyze sample data, known as 'training data', and make predictions from that by define a model and train the model without explicitly programming to do so. Traditionally Machine learning can be divided into three categories based on the purpose of the system:

- Supervised learning: The system is fed with data with expected labels, or in some of the context was called tags, and the goal for the system is to learn a function out which could infer the parameters for maps the input data to desired labels.
- Unsupervised learning: The system is trained with unlabeled datasets without any supervision, this kind of system often used for find the underlying structure of the input datasets, group that data according to similarities and represent in a compressed format.
- Reinforcement learning: Unlike the previous two categories, reinforcement learning is more focus on to use the input datasets to training the system to make a sequence of decisions. Usually for such kind of system we need to carefully define the reward and penalties policy for the actions the system performs in order to maximize the total reward.

Nowadays, the machine learning system is written as a collection of all those categories to provide more features and satisfy general needs. There are several libraries are popular and commonly used, four of them are listed below:

- TensorFlow[1]: an end-to-end open-source platform for machine learning. It is distributed with a comprehensive, flexible ecosystem of tools, libraries, and community resources which lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications.
- PyTorch[2]: an open source machine learning library. While it has a C++ interface, the python interface is more polished and the main focus of development. PyTorch is mainly developed by Facebook's AI Research lab (FAIR). It provides two high-level features: tensor computing with strong acceleration and neural networks utilizing automatic differentation. PyTorch supports both CPU and GPU(CUDA and ROCm) acceleration.
- scikit-learn[3][18]: this library support vector machines, random forests, gra-

[1]https://www.tensorflow.org/
[2]https://pytorch.org/
[3]https://scikit-learn.org

dient boosting, k-means, and DBSCAN, featured with various classification, regression, and clustering algorithms.

- Keras[4]: a deep learning API running on top of the machine learning platform TensorFlow2.1, which is developed with a focus on enabling fast experimentation. Those APIs aim to let the researcher be able to go from idea to result as fast as possible.

Therefore the size of the software is more than needed. Furthermore, the machine learning approach is getting more and more widely used, aiming to replace the traditional program in more fields and be deployed to various networks. Even many copies existed in a single network, which causes much waste on both resources and the environment. So, in this case, debloating in such systems is needed more than ever.

2.2 Debloating

Software bloating impacts the security and performance of software systems negatively, [19]. Hence several debloating techniques have been proposed. These debloating techniques aim to remove code bloat, i.e., excess library code and features that are not needed by the end-user(or, in other words, bloat). For example, [20] shows that libc in average across the Ubuntu Desktop environment, only 5% is used and that the VLC media player make use of 18% of the original binary size.

Most of the debloating system based on the granularity could be categorized into two parts:

- Files optimization: These tools aim to remove unused files in the system, for example, unused binary/library files, to reduce the size of the software deliveries and lower down the risk of the attacker taking advantage of the command or tools.
- Binary optimization: The other part of the tools employs compiler or other disassembly tools to understand the program binary flow, prune the un-visited logic branch, and then rewrite the binary file to generate a new one that only contains the necessary instructions. In this way, the execute instructions do not contribute to user-needed feature logic but could potentially be used by hackers to break the system is removed.

2.3 Machine learning models

In this section, we give a short overview of the types of ML models or specific models that we use in this thesis.

[4]https://keras.io

2.3.1 Neural network

Artificial neural networks, usually simply called neural networks, are computer algorithms consisting of a collection of connected units called neurons or nodes. Each node computes an output based on the inputs it receives. The output values are modified by changing the multipliers in the output calculation called weights. These weights are changed during training in order to improve the results of the network based on a chosen metric. Neural networks have exploded in popularity with the growing availability of datasets and computational power as they show superior performance in solving many classification, identification, and AI problems. For more information on modern neural networks, please refer to [21].

2.3.2 LSTM

Long short-term memory (LSTM) is a Recurrent Neural Network (RNN) with feedback connections. A Recurrent Neural Network excels at being used to process images and videos as well as data presented as a data series. It was popularized by [22].

2.3.3 YOLOv3

YOLO[23] (You only look once) is a real-time object detection system. It is extremely fast and accurate. YOLOv3 is 4x faster than Focal Loss, and still able to achieve a similar accurate result. The system is easily turnable between speed and accuracy simply by changing the model's size with no retraining required.

2.3.4 Botorch

Botorch[24] is a library for Bayesian Optimization research created using PyTorch.

2.3.5 Diffusion

Improved diffusion[25] is a model based on Dense Discrete Phase Model (DDPM)s[26], it can achieve competitive log-likelihoods while maintaining high sample quality.

2.3.6 DLA

Deep Layer Aggregation(DLA)[27] aims to improve combination of semantic and spatial information for recognition and localization, and achieve a fewer parameters network with better accuracy.

2.4 Docker

Docker is an open platform for developing, shipping and running applications. The three main components of Docker are the Dockerfile, which is like a blueprint that specifies the contents, an image created from the Dockerfile and finally the container

created from the image. The image is an read-only template and the container is a virtualized run-time environment created based on the image. This containerization allows for easy reproducability and sharing of applications. It also allows for easily rerunning a workload as you just have to recreate a container and also makes the different runs independent of each other as the entire container is created anew.

In our thesis we create images that have the required ML systems and GPU libraries installed in order to run our experiments. The model files are made available to containers through the use of volumes. We also exclusively look at tools that work on docker images and containers as docker is very commonly used and offers good guarantees to reproducability and independence of test runs.

2.5 Existing debloating tools

There are quite a large number of tools for debloating that already exist. They can be divided into two groups according to their granularity. The tools have different goals with some of them placing emphasis on increasing security while others are meant to make the software more suitable for embedded systems. There are also large differences between the prerequisites needed to use the tools. Some require source code and extra configuration/test files while others require only the binaries to debloat.

2.5.1 File optimization tools

The granularity of removal for this subset of tools is on the file level. They remove unused files from the target.

2.5.1.1 Docker-slim

Docker-slim [28] is a tool that optimizes docker containers. It uses static analysis to inspect container data and metadata. It inspects the running application using dynamic analysis. Using those results from the analysis, a smaller image is generated. While the tool will work for any dockerized application, it automates app interactions for HTTP API applications. If the application lacks an HTTP API, the user has to interact with the container while it is being analyzed to allow the tool to observe the application behavior.

The tool has been successfully used on a number of different images. For example a Ubuntu:14.04 based Node.js image was reduced from 432MB to 14MB, a 30-fold reduction, a CentOS:7 based python image was reduced from 647MB to 23MB, a 28-fold reduction.

Our work tests this tool in a slightly different area, machine learning systems. This area differs in that no HTTP API is available at all, meaning that the we have to manually interact with the container to a larger extent than normal. Our testing and results are explained in more detail in section 4.2.1

2.5.1.2 Cimplifier

Cimplifier [29] is another tool that focuses on optimizing the docker containers. It dynamically analyzes how the application executables use resources inside a container, then partitions the container for removing the redundant part while still satisfying the executable needed environment. The dynamic analysis is done using *strace* to get a system call log, then analyze the requirement of libraries and resources, etc. The tool's output is a set of containers, with only the executable and the resources needed to run them.

This tool has shown a great potential of reducing the size of containers. For example, an Nginx image was reduced from 502MB to 6MB with a 95-fold reduction, and a Redis image was reduced from 153MB to 12MB, a 92-fold reduction.

Our thesis makes use of the work done in this paper. We use the tool created for this paper and apply it to a different context, machine learning systems.

2.5.2 Binary optimization tools

The granularity of removal for this subset of tools is on the binary level. They work on either source code which are compiled into bitcode or directly on binaries which are then analyzed. They remove entire files as well as cut unneeded parts of files or code. Unfortunately none of the tools mentioned in this section are applicable for us as they are not meant for code written in python.

2.5.2.1 CHISEL

CHISEL [13] enables programmers to customize and debloat programs. A simplified yet high-level specification of the desired functionality and the original program is given to the system as inputs. The program generates a minimized version of the input program while guaranteeing that it matches the feature specification. An example of the feature specification input given was of a program that compiles then runs the program and checks input-output behaviours. The system does not guarantee an optimal minimum but guarantees a so called 1-minimality, which means that removing any single element from the minimized program makes the program fail the feature property test.

The system uses an algorithm called Delta Debugging, which converts the input program into a list of elements with arbitrary granularity (lines, tokens or functions). The algorithm tries different granularity and tests different subsets and their complements for passing the property test. The system uses model-based reinforcement learning in order to reduce the number of trials in Delta Debugging.

The effectiveness of CHISEL was tested on 10 utilities from GNU CoreUtil. First they used SPARROW, a static analyzer for C programs, to remove all unreachable functions. That reduced the total number of statements from 172 304 to 55 848. After running CHISEL only a total 6,111 statements where left over all 10 utilities.

A statement is a syntactic unit that expresses some action to be carried out. It should not be confused with lines of code, as it is possible to have multiple LoC on a single line as well as take multiple lines for a single statement.

2.5.2.2 HECATE

HECATE [17] leverages several open-source tools and deep learning to identify program function boundaries and bodies from a binary executable in an unsupervised fashion. Evaluation using real-world applications shows that Hecate can efficiently customize large-scale software, accuracy up to 96.28% for feature/function identification and significantly reduce the attack surface, up to 67% reduction of program attack surface according to the paper.

The problem of debloating is considered a multi-class classification issue. Class labels are functions, class is function body instruction, and testing sample maps to execution path. Based on this, Recursive Neural Network (RNN) is used to obtain binary code vector, followed by a multi-class Convolutional Neural Network (CNN) classifier to identify the feature-constituent functions.

2.5.2.3 Wholly!

Wholly! [30] is a tool for building and packaging software with C/C++ code. It has been used to build entire operating systems, such as FreeBSD and macOS. It can also be used to build Alpine and docker containers. Wholly! uses Linux containers in order to insure integrity and reproducibility. The clang compiler is used to generate LLVM bitcode for all of the produced libraries and binaries, which is analysed and optimized.
Each package is described by a recipe that contains information for building it and a contents file that is used for releasing it. The recipe is a YAML-formatted file that contains a link to the source code, names of other Wholly! packages that are build dependencies, commands to run and paths to additional needed files. The contents file is in the same format and splits the package into sub-packages that are defined by the files that compose them and a checksum for integrity.

Each build follows the following pattern:

1. Wholly! generates a Dockerfile from the recipe and a container is created from the resulting image
2. Files from the dependency sub-packages are copied into the container
3. Source code is downloaded
4. The supplied commands (from the recipe) are executed
5. The resulting image is split into sub-packages according to the contents file

In the paper the authors compare three nginx images: one produced with Wholly! and two from Docker hub (nginx:mainline-alpine and nginx:official). The Wholly! image is around 25 times smaller than the ngingx:official image and almost 4 times smaller than the Alpine version. They also benchmarked the servers response time

for 10 000 requests to the server. The average response times were: Wholly! - 3.09ms, official - 3.20ms and Alpine - 3.57ms.

2.5.2.4 Nibbler

Nibbler[31] works on debloating the shared libraries used by a binary application without source code. For a given binary application, Nibbler disassemble the shared libraries used by it, then utilize the statically analyzer to reconstruct the FCG for each of the libraries, then identify all the functions that are not used and remove them, replace with trapping instructions, finally generate a thinner version library of each library.

With unused code removed, Nibbler could achieve 56% and 82% of functions and code reduction in real-world binaries and in SPEC CPU2006, respectively.

2.5.2.5 LibFilter

LibFilter[32] is another tool that focuses on shared libraries used by an application, which also aims to identify unused functions and erase them without access to source code, in order to reduce the number of gadets that could be utilizable by attacker.

Egalito[5] is utilized to perform static analysis on binaries and share libraries, can construct a function call graph. The unreachable functions will be replaced by one byte *hlt* instruction, then new version of shared libraries will be generated.

LibFilter reduces number of bytes in Coreutils by 37.2% and 29.7% in SQLite with all functionality test pass after debloating.

2.5.2.6 JShrink

JShrink[33] chooses the object oriented application as the debloating target because in the Object Oriented programming model: a simple application would use a large amount of libraries to do even simple work. This research presents an end to end Java bytecode debloating framework, and also a profiling agent JMTrace for capturing the usage of dynamic features in java code. This research achieves 46.8% bytecode size reduction.

2.5.2.7 WebJShrink

WebJShrink[34] is a follow up work built upon JShrink, it transform JShrink to a web service which could provide 11% on average size reduction with maintain 100% unit tests pass, also provide visualization on the bloat part to give a hint of the reduction.

[5] `https://egalito.org/`

3
Methods

This section goes over our approach and evaluation environment. A compact overview of the main steps and their connection to the RQs can be seen in figure 3.1.

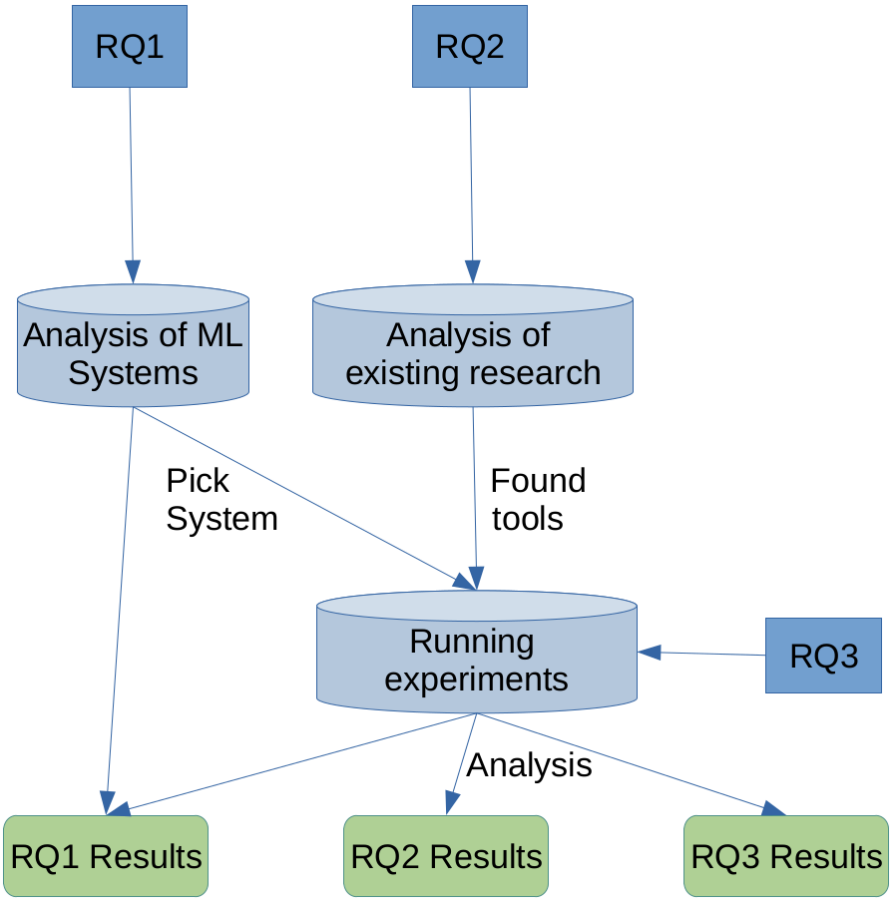


Figure 3.1: An overview of the approach and mappings to RQs

3.1 Approach

The first part of our approach is to assess the existence and prevalence of bloat in a handful of the most popular machine learning systems. The results of which answer RQ1 and help choose a system for further study for RQ2 and RQ3. The second part of our approach goes over the evaluation environment: the hardware, datasets,

images and models used in our experiments. The results of the experiments are analysed to answer RQ2 and RQ3.

3.1.1 Analysing ML systems

Our first aim is to assess the existence and prevalence of bloat in a handful of the most popular machine learning systems. An overview of the steps we take in this part can be seen in figure 3.2.

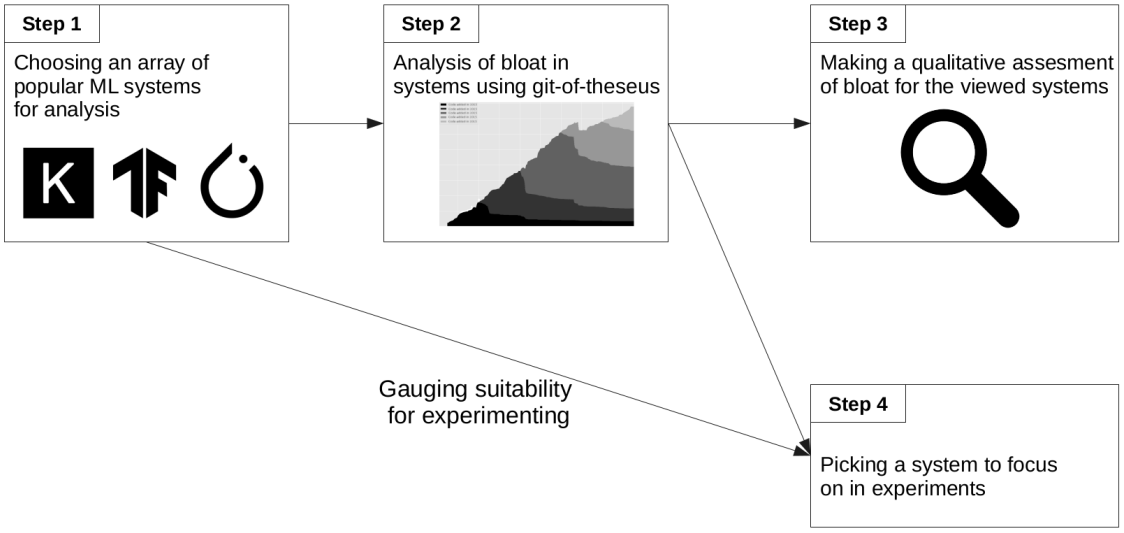


Figure 3.2: An overview of the approach to analyzing ML systems

Step 1: We look at a number of different ML systems and pick a smaller selection based mainly on their popularity in the ML field and that they are available in Python as the language is popular and we are well versed in it since we also want a system to work with in RQ2 and RQ3. We chose to analyse three of the more popular ML systems, namely: PyTorch, TensorFlow and Keras.

Step 2: The analysis of the systems chosen in Step 1 was done by using a tool called git of theseus[1], which allowed us to look at the systems codebase over time while keeping track of code grouped by the year of committing. The figures generated from the tool gave a helpful overview as it allows us to see how much code is added and how much older code is removed.

Step 3: In this step we give an assessment to the existence and prevalence of bloat (RQ1). A strong indicator of bloat is a clear growing trend in the increase of code as well as the reduction of old code being minimal over time. The assessment in this manner is not quantitative as the exact classification of a part of code as bloat is subjective i.e dependant on the specific work.

Step 4: The system to be used for running the models is picked based on the analysis done in the *Step 2*, as we want to pick a system that we presume is more bloated.

[1] `https://github.com/erikbern/git-of-theseus`

We also look into factors such as popularity in the field, existence of models we can use, release schedule and API stability.

3.1.2 Experimental setup

An overview of the steps in the experimental part can be seen in the figure 3.3.

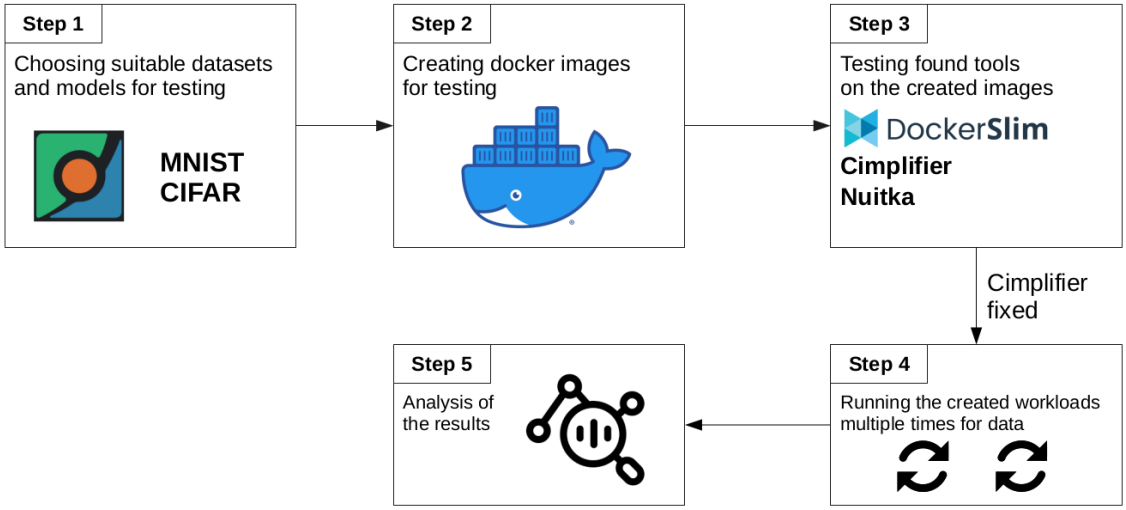


Figure 3.3: An overview of our experimental approach

Step 1: Firstly, we choose datasets and models to create workloads for further testing. We want datasets and models of different sizes which are also popular and publicly available so that our results are reproducible and that models are available that use these datasets. The availability of models is important since we are not interested in improving the models themselves, we simply need a working model to create workloads for testing.

Step 2: Since docker images are quite popular, we decided to use them in our thesis, since they make the workload easily rerunnable and shareable. We found a good starting image to work from that already contained the ML system and necessary prerequisites. The dataset and model files, which are required to train and verify the accuracy of the models we plan to test, are connected to the base image as a volume.

Step 3: During our research into SotA papers, we find a number of tools. In this step we try using existing tools on our previously created images and see if they are applicable for our purposes. We anticipate that some tools might require modifications to work. We expect to get a debloated version of the original image after applying the tool(s). RQ2 is partly answered in this step, as it can rule out tools that do not produce working images. RQ3 is mostly answered by this step as the accuracy of the models is checked.

Step 4: The debloated and original images are ran multiple (25) times in order to gather data. We log runtime and GPU metrics (usage, power consumption, memory

usage) with command
```
nvidia-smi -l 1
    -query-gpu=power.draw,utilization.gpu,memory.used
    -format=csv,nounits -f <FILENAME.csv>.
```

Step 5: In the final step we compare the results of the original and debloated images in order to see what if any improvements were gained by the application of the found tool(s). We also look at storage reductions in the compressed image size resulting from the debloating process. RQ2 and RQ3 are answered by these comparisons, as we can see if the models deteriorate in any way i.e if they are not harmed by the process which means the tools are suitable for this use.

3.2 Evaluation Environment

The evaluating environment consists of both the hardware platform we are running the training on and the datasets we are benchmarking.

3.2.1 Hardware

Since the type of hardware on which a machine learning system runs can influence which functions in the ML system is used, we have decided to test the tools on multiple different hardware configurations. We used three different hardware configurations which span a wide variety of powerfulness. The first configuration is a reasonably powerful workstation, the second configuration is a laptop with a weak GPU and the third is a high performance computation oriented instance provided by AWS.

3.2.1.1 Hardware configuration Workstation

The first configuration is based on a DELL T640 workstation. 4 x Nvidia 2080Ti Graphic cards are installed in this system. Each graphic card has 4352 CUDA cores, 11GB GDDR6 of memory, and a base clock speed of 1350MHz. 2 x Intel® Xeon® Gold 5218 Processor are installed in this workstation, each of them has 16 cores with hyperthreading enabled. Memory wise the machine contains 8 x 32GB DDR4 2933MHz memory. This workstation has a Seagate 15k RPM SAS-3 900G disk installed, with latest Ubuntu 18.04(Bionic) installed.

3.2.1.2 Hardware configuration Laptop

Hardware B is based on an Acer Nitro 5 an515-54 laptop, which has an Nvidia GTX 1050 Mobile GPU allowing for CUDA accelerated workloads. The GPU has 640 CUDA cores, 2 GB GDDR5 of memory and a clock speed of 1354MHz. The laptops CPU is Intel(R) Core(TM) i7-7700HQ CPU running at 2.80GHz with 16 GiB of DDR4 SODIMM ram running at 2400MHZ. For storage the laptop uses a NVME M.2 SSD. The tests were ran on the Ubuntu 20.04 (Focal Fossa) operating system.

3.2.1.3 Hardware configuration AWS

The instance type we ran using the EC2 service of AWS is "p3.2xlarge"[2]. The instance provides a single Nvidia Tesla V100 GPU. The GPU has 5120 CUDA cores, 32GB of HBM2 meory and a clock speed of 1380Mhz. The instance has 8 virtualized threads of custom Intel® Xeon® Scalable (Skylake) and 61 GB of memory. The storage is based on NVME SSD-s. Operating system vise Deep Learning AMI (Ubuntu 18.04) Version 44.0 is chosen (ami-078dbdfd9cab3abf9.

3.2.2 Datasets

The datasets used for training and testing our models are MNIST, EMNIST, CIFAR datasets and COCO datasets, which were picked since many different models and benchmarks already exist for these. They are also publicly available and do not contain any sensitive information.

3.2.2.1 MNIST

MNIST [35] is a dataset of handwritten digits. MNIST is a subset of the NIST dataset images that have been size-normalized and centered in a fixed-size image. The dataset consists of a training set with 60 000 images and a test set of 10 000 images. The images consist of 28x28 pixels of greyscale values.

3.2.2.2 EMNIST

The EMNIST [36] dataset is a dataset that has been created from the full NIST dataset following the same conversion paradigm used to create the MNIST dataset. Unlike MNIST, EMNIST also contains letters in addition to digits. The EMNIST dataset provides six different splits of characters.

- ByClass: 814,255 characters. 62 unbalanced classes. Contains all 10 digits and 26 letters in both uppercase and lowercases.
- ByMerge: 814,255 characters. 47 unbalanced classes. The same images as in ByClass but the uppercase and lowercase labels of the letters: c, i, j, k, l, m, o, p, s, u, v, w, x, y and z have been merged.
- Balanced: 131,600 characters. 47 balanced classes. Intended to be the most widely applicable split. It is a balanced subset of the ByMerge subset.
- Letters: 145,600 characters. 26 balanced classes. A balanced subset of letters where uppercase and lowercase classes are merged.
- Digits: 280,000 characters. 10 balanced classes. The digit equivalent of the Letters split.
- MNIST: 70,000 characters. 10 balanced classes. Matches the size and specifications of the original MNIST dataset, meant to be a drop-in replacement.

[2] `https://aws.amazon.com/ec2/instance-types/p3/`

3.2.2.3 CIFAR-10

CIFAR is a dataset are labeled subsets of the 80 million tiny images dataset. They were collected by Krizhevsky et al [37]. The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. Among the images 50,000 are training images and 10000 are test images. Those 60,000 images are divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1,000 randomly selected images from each class. The training batches contain the remaining images in random order, but the training batches are not selected evenly from each class, it may have more images from one class than another.

3.2.2.4 COCO / MSCOCO

COCO[38] is another large-scale dataset mainly used for object detection, segmentation, and captioning. COCO stands for common objects in context. This dataset is created by the Microsoft(R) research center, etc., which aims to support the studies of computer vision in an understanding of visual scenes. In this dataset, objects are labeled using per-instance segmentations to help in precise object localization.
Ninety-one object types are contained in the photos of this dataset, which are easily differentiated by a 4-years old kid. The 328k images in total contain 2.5 million labeled occurrences.

3.2.3 Models

In the thesis we test five different ML models, that use a total of three datasets.

3.2.3.1 LSTM/MNIST

The smallest model we test is a LSTM model meant for the MNIST dataset. The model is taken from GitHub[3] and small modifications are made to also log the time the workload takes.

3.2.3.2 DLA/CIFAR-10

The second model we test is a model meant for the CIFAR-10 dataset, code is grabbed from GitHub[4], and minor modifications are added for timing, more information about Deep Layer Aggregation(DLA) can be found at [27].

3.2.3.3 YOLOv3/COCO

YOLO model is a minimal PyTorch implementation from GitHub [5]. For the original paper, please refer to [23]. The training of the YOLOv3 model is conducted on the original COCO dataset.

[3] `https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/Basics/pytorch_rnn_gru_lstm.py`

[4] `https://github.com/kuangliu/pytorch-cifar`

[5] `https://github.com/eriklindernoren/PyTorch-YOLOv3`

3.2.3.4 Diffusion/COCO

Improved-Diffusion is another model run on the COCO dataset, and source code is published to GitHub [6]by OpenAI, the original paper could be found at [25]. We refer to this model later as just diffusion.

3.2.3.5 BoTorch

The final model that is evaluated is based on the BoTorch framework, and a toy example is created from the BoTorch tutorial[7]. It is a demonstration of implementing a simple multi-objective (MO) Bayesian Optimization (BO) closed-loop. We will refer to this model as botorch from now on.

3.2.4 Docker images

All of the Machine Learning workloads are executed inside a docker container. Two base images are used in this thesis, into which the datasets and models are added either as a volume.

3.2.4.1 anibali/pytorch:1.5.0-cuda10.2

One of the base images we pick is anibali/pytorch:1.5.0-cuda10.2 available from DockerHub. The image is picked since it has the PyTorch library as well as CUDA capability already available. The LSTM/MNIST and DLA/CIFAR-10 models use this base image.

3.2.4.2 wy0917/pytorch:1.8.1-cuda10.2

This image is created using a modified version of the Dockerfile from ANIBALI/PYTORCH:1.5.0-CUDA10.2, but with a newer version of PyTorch, since the YOLO, BoTorch and diffusion models need the libGL.so.1 library to run, the Dockerfile is uploaded to GitHub[8].

[6] https://github.com/openai/improved-diffusion
[7] https://botorch.org/tutorials/multi_objective_bo
[8] https://github.com/wy0917/pytorch-docker/blob/main/Dockerfile

4

Results

In this chapter we present the results of the ML system analysis as well as the results of the experiments we ran. We also explain in more detail the actual steps taken to generate these results.

4.1 An analysis of ML systems

We start with an analysis of the source code of three popular software systems used for ML, namely TensorFlow, Keras, and PyTorch. By looking at their size and historic growth rate shown in Figures 1.1, 4.1, and 4.2, we suspect that all three of them are bloated. The aforementioned graphs were created with the tool git-of-theseus. In the case of Keras, the figure contains analysis up to 24 March 2019. After that time, the project changed from being standalone to being a part of TensorFlow. In the figures, it is possible to keep track of the amount of code at a point in time that was added in a given year. As expected, the graphs similarly show strong growth through the years while the previous years' code slowly diminishes. The assumption that the systems are bloated is based on the fact that all three of them show a consistent and noticeable rise in the line of codes while at the same time minimally reducing code written in previous years. While this might all be the result of additional functionality or improvements, it is more likely that because the codebase is so large that some parts have become obsolete or share functionality with others. Even if all of it is new functionality, it is near impossible that anyone's application would require all of them.

We ruled out Keras because of its integration into TensorFlow in 2020. After evaluating the TensorFlow project, we decided that TensorFlow will not fit this thesis due to the shortness of the release cycles and the existing number of different branches. Currently, TensorFlow provides support for 1.x and 2.x release branches, and the release cycle is not fixed. For our thesis to have the most practical relevance, and as it is always best to always use the latest machine learning system version, it was important for us to pick a ML stack that is more stable with its release cycles. However, if we look at the PyTorch project, the release cycle is roughly 2-3 months, which should give us enough time to go over the code-base and study the bloat using the various tools. We also take into account the anecdotal evidence that TensorFlow's API was less stable through releases compared to PyTorch. In the end we chose PyTorch as our point of focus.

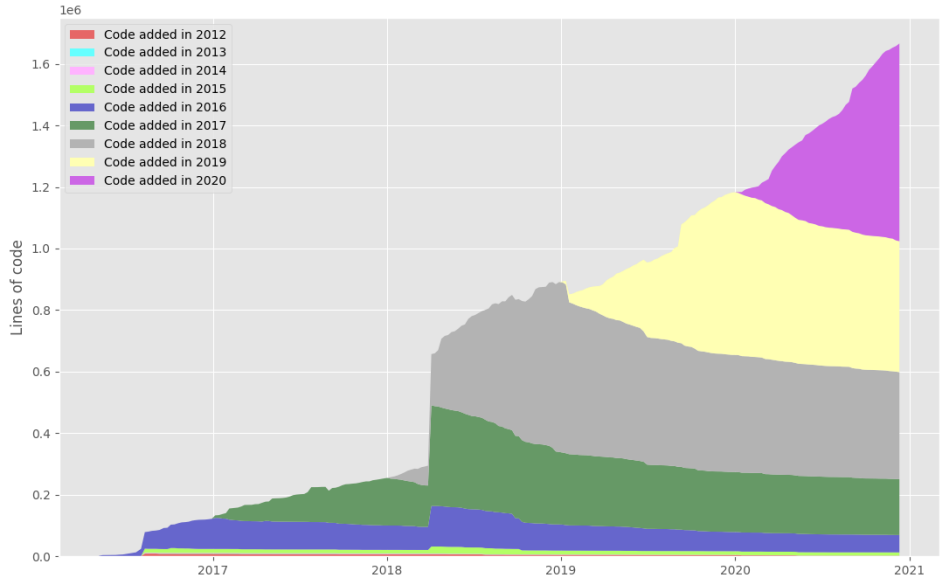


Figure 4.1: Lines of code in PyTorch since creation

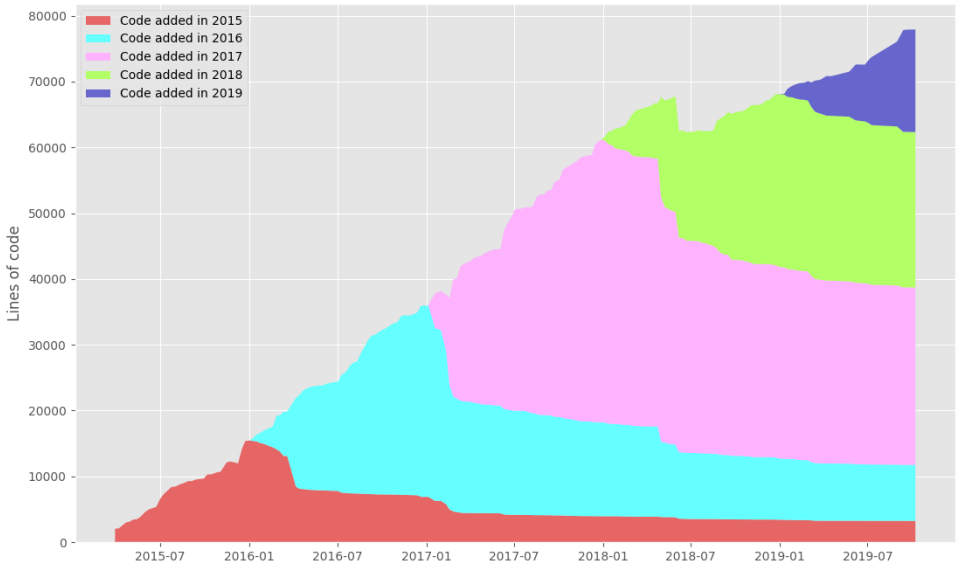


Figure 4.2: Lines of code in Keras since creation up to the merge with Tensorflow

4.2 Evaluation of debloating tools

In the experimental part of the thesis, we evaluate two debloating tools from the ones mentioned above, namely, docker-slim and cimplifier. To the best of our knowledge,

none of the tools we try have been used on machine-learning workloads. These tools are tested using the docker images mentioned before in section 3.2.3.

4.2.1 Docker-slim

The first tool we tested is docker-slim [28]. It has been shown to be effective with a wide array of different images with great success.

First, we try the tool on the image python:latest from Docker hub. This initial experiment succeeds in reducing the size of the image from 885MB to 33.7MB. However, when we experiment with docker-slim on the ML docker images, the tool does not work. The first problem we face is that docker-slim does not support all of the docker command line parameters, in our case when running the *docker run* command we have to supply either –gpus all or –runtime nvidia in order to make Nvidia GPU-s available inside the container. Neither of these flags were available in the latest release on github. We contacted the authors and were directed to the tools own website which had a newer version which had the –runtime flag available. This made the GPU visible inside the container during analysis. Unfortunately, all of the images created are unable to start at all or break when run with the flags –gpus all or –runtime nvidia. This means that none of the resulting images can use GPU acceleration, making them quite useless from a machine learning point of view. Also the tool barely reduces the size of the image, reducing it from 3.37GB to 3.36GB, a reduction far smaller than expected.

4.2.2 Cimplifier

Cimplifier [29] is the second tool we test [1]. We try the tool in two different environments in parallel. The first environment is a vagrant VM meant to replicate the original environment described in the paper and the second running on baremetal and using the latest versions of any libraries used.

The tool uses system call logs for resource identification i.e to determine which files are actually needed and in the case of splitting the image it also determines which files are needed in which compartment. The tool used to collect these logs in the paper was *strace* which we opted to use as well. The command uses the base process id - pid to know which process to look at. Using the collected logs and the original image, the tool generates a slimmed version of the original image.

The code required a few modifications in order to work. The first thing we changed was to set the partitioning policy to all-one-context which means that all of the executables stay in a single partition, i.e., just slimming the image. The tool also has the capability to partition the image into smaller images in order to compartmentalize different application components but we do not use this feature. After changing

[1] We would like to thank the authors of the original paper: Rastogi Vaibhav, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel as well as Mohannad Alhanahnah for supplying us with the source code.

the context type we have to update the function of the all-one-context mode as it seems that some of the logic changed after the function was written. Some smaller modifications are needed which seem to stem from the fact that the system calls that we collect differ from logs in the original paper. We assume this since, some of the calls collected by us cause the code to crash since the calls are not implemented. We also change some of the code's expectation for the logs, this is explained further in section 4.2.2.2.

4.2.2.1 Vagrant VM environment

The original code we got from the authors of cimplifier has a vagrant[2] environment configured. Vagrant is a tool for managing virtual environment, which can use Virtualbox[3] or VMware[4] as back-end to recreate the specified environment for development. The cimplifier code was written for the Fedora 12[5] environment, and used python pip for installing required libraries. However, since there is no version lock provided for each pip package, we had some problems in the beginning to make the code work. Trouble-shooting works was required to make the code run successfully again in the original environment. After the cimplifier project could be run successfully, we decided to share a package version lock file to gitlab for any future researchers to use[6]. Furthermore, some manual work was needed to run the code. First, the correct process need to be identified, in our case the main docker instance. The following is a log from the process listing command `ps -ae --forest -o pid,ppid,comm:25`:

```
 PID   PPID  CMD
 3884     1 sh
 3885  3884  \_ docker
 4165  3885  |   \_ sleep
 3886  3884  \_ forward-journal
```

So pid 3884 is the process that needs to be traced. After identifying the process then we start running the model or workload, so that we can get all the needed system calls logged. However, the strace log can not be used as is, everything before setting up the container namesapce systermcall pivot_root() line in the log needs to be deleted to remove the unneeded syscalls. This variant of using the tool is not used for any of the testing as it adds an unneeded layer of virtualization by using Virtualbox Hypervisor.

4.2.2.2 Running on bare-metal

We also try Cimplifier outside of a VM, running on the baremetal with containers. We start by installing every package needed using pip inside of a virtual environment. Luckily the code works with the latest versions of all the libraries used.

[2] https://www.vagrantup.com/
[3] https://www.virtualbox.org/
[4] https://www.vmware.com/
[5] https://docs.fedoraproject.org//en-US/Fedora/12/html/Release_Notes/
[6] https://gitlab.com/wyan/vagrant-pip-req/-/blob/master/requirements.txt

The logs are collected by using *strace* inside the container. Two bash terminals are opened inside the container, one that is used to run the python file (model) and the other to *strace* the first one. The pid is found by using *ps*. After comparing our logs with the code we see that our logs do not contain the pivot_root system call which was expected in the code. We modify the tool to not expect the pivot_root syscall at the start of the logs. Another issue we run into is that the code expects to find fork or clone system calls in order to look at different log files. Unfortunately our logs do not contain any of these system calls so we have to concatenate our log files into one. We also remove some code that was meant for a depth-first search of the executable tree, this is made redundant by our previous concatenation of the logs.

After these changes we are able to successfully slim down our initial testing LSTM machine learning image in a way that the resulting image retains the ability to use GPU-s which are needed for machine learning workloads. This way of collecting the logs is used in the debloating of the LSTM and DLA models.

4.2.2.3 Containerd automation

To be able to use the cimplifier tool, a complete *strace* log of the ML workload is needed. But due to the current architecture of docker system[7], starting *strace* on the `dockerd` process as previously mentioned in 4.2.2.1 will not catch the relevant logs, and calling *strace* after the container starts is too late to catch the entry command. The command flow can be seen in Figure 4.3, where we first see the docker command initiated by the user in a terminal, which uses REST to call dockerd which in turn calls containerd which finally starts the runc process, which handles the actual container itself. So a patch was created for runc to start the *strace* process right before the container changes namespace and starts to run the entry command, the patch will identify the runc process pid, current docker id and start strace on it. The patch use a `-e 'TRACE=true'` as an enable flag to start the strace command, and the pstree when running strace looks like following, process 199162 is the entry command process specified when starting the docker:

```
    PID    PPID COMMAND
 199142       1 containerd-shim
 199162  199142  \_ bash
 199196  199142  \_ strace
```

4.2.3 Image size reduction

The cimplifier tool is successfully used on six images. An initial test on the latest python image from Dockerhub and five ML images, the latter of which are explained further in section 3.2.3. All of the tested images are successfully reduced in size by the cimplifier tool, they also retain all of the functionality used while the containers were being *strace*-d for system calls.

[7] `https://containerd.io/`

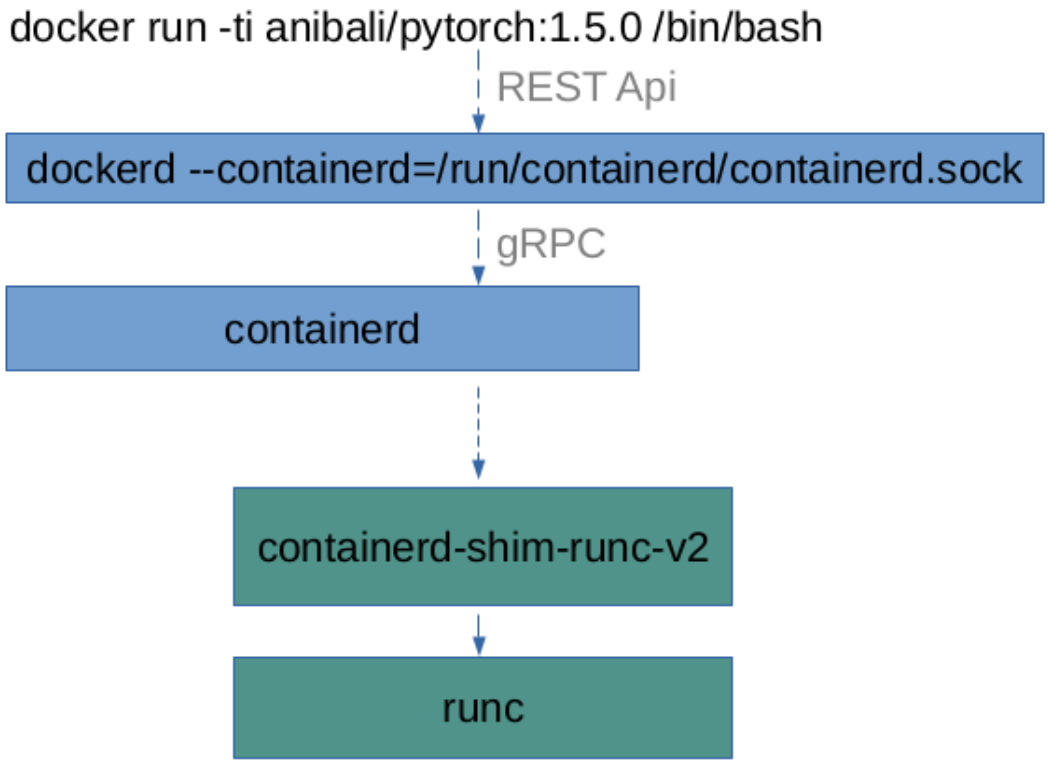


Figure 4.3: From `docker run` to container starts

The initial testing python-only image is reduced from 885MB to 26.7MB. In the case of ML models, one of two images is used as a base and the model-specific data is loaded in as a volume, which means the models themselves do not account for the storage at all. The original and debloated image sizes of the five ML images can be seen in Figure 4.4. The debloating process has very similar results for the LSTM and DLA custom image which are both reduced from the smaller base image of 3.37GB to 1.60GB and 1.63GB respectively. The larger three models are based on the larger image. The highest reductions are in the botorch image, which is reduced from 4.43GB to 2.37GB. The diffusion and YOLOv3 images have similar results with debloated image sizes of 2.54GB and 2.48GB respectively. These results tie into RQ1 and suggest that at the very least in the case of images containing standard installations of the ML system, there exists quite a lot of bloat storage-wise. We also confirm RQ2 as with modifications cimplifier is able to debloat a ML system and in practice removes a significant portion of the bloat.

4.2.3.1 Overview of removed files

The debloated and original images are compared using the tool container-diff[8]. This tool returns a list of files that were removed from the original when compared to the debloated image. This list is then further filtered to look at the files that are

[8] `https://github.com/GoogleContainerTools/container-diff`

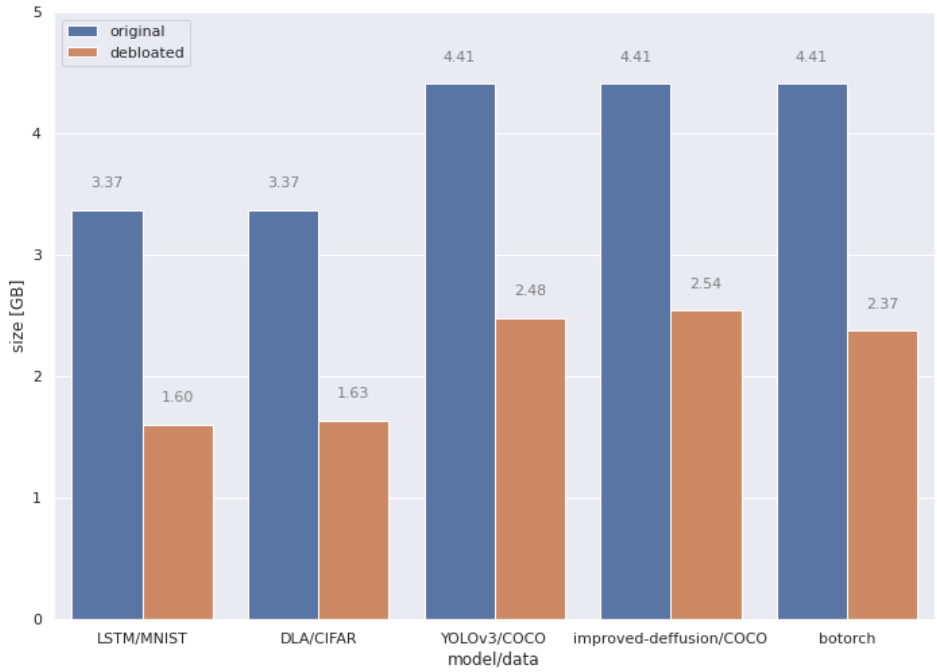


Figure 4.4: Size comparison

larger than 10MB, this is done to make manual inspection manageable. The inspection is hampered by the fact that the tool does not take shortcuts into account, for example the library file libcusolver.so links to libcusolver.so.10 which links to libcusolver.so.10.3.0.89, while showing the same size for all of them in the results. After further inspection we also notice that every file in the directory "'/home-/user/miniconda/pkgs/'" are actually links, while these files make up around half of the filtered results from container-diff. These factors together make the automation of inspection unviable for this thesis and causes a degree of uncertainty to the results.

In general, the inspection shows that the majority of the removed files are compute libraries in the form of .so files. To be more specific they are parts of the Nvidia CUDA toolkit and the Intel Math Kernel Library. For example in the case of the YOLOv3 model, the debloating process removes 1561MB of .so files, that were larger than 10MB individually. This means that of the 1.93GB removed, those .so files make up around 80%. This ratio increases even further in the cases of the LSTM and DLA models, reaching up to around 89%. When comparing the five different models between themselves, the largest difference comes from the .so files that were removed. There are also .so files that were debloated from all of the images, for example libublas.so, which is the CUDA library for BLAS (Basic Linear Algebra Subprograms[9]).

[9] `https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms`

4.2.4 Performance evaluation

The machine-learning image workload consist of training the model on the training dataset and afterward checking its accuracy on the test dataset. This workload of training plus testing was timed for all of the images with different amounts of epochs or in the case of larger models the amount of steps or iterations was changed. Each combination of epoch and image was run 25 times in order to average out any natural variance. The average running times for the YOLOv3 and LSTM models can be seen in Tables 4.1 and 4.2. As can be seen in the tables, no clear effect from debloating can be seen. While there are small differences between averages, there is no clear trend when comparing different workloads and the differences are at most a fraction of percentage points different but falling within the range of the results of the original image. The previously mentioned comparisons between the original and debloated images for the YOLOv3 model can be seen as boxplots in the Figure 4.5. The results for all models were very similar and lead to the same conclusions. The tables and figures on the results for the other models can be seen in Appendix A.

We also *strace* the LSTM ML image with a different number of epochs of training and find that the tool's effectiveness is not affected. This observation means that it should be possible to run a very small workload and thus relatively quickly debloat an image that could then be used for the actual training. This is beneficial since for larger models the act of collecting logs while training can become a bottle-neck as the number of calls being made grows to be very large.

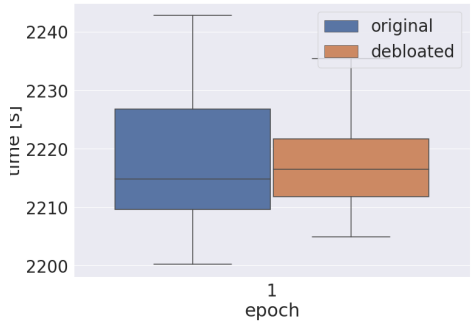
Epochs	Original	Debloated
5	56.88 ±2.66	54.83 ±0.94
15	146.08 ±1.24	147.29 ±1.19
25	238.03 ±1.87	239.94 ±2.29

Table 4.1: The average time (s) elapsed for training and evaluating the LSTM/MNIST model

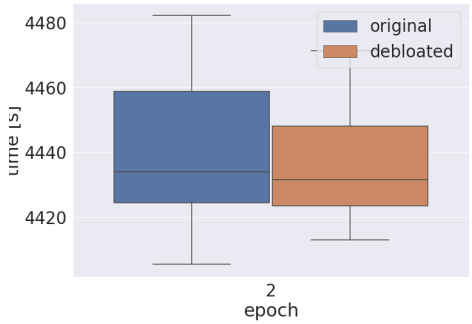
Epochs	Original	Debloated
1	2217.09 ±11.79	2217.81 ±8.46
2	4439.76 ±20.07	4436.69 ±17.25
4	8849.81 ±51.37	8862.60 ±38.06

Table 4.2: The average time elapsed for training and evaluating the YOLOv3/COCO model

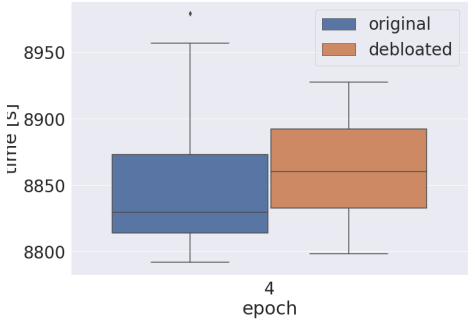
In addition, we collect the power consumption of both the original and debloated images. The power consumption metrics are collected once as the comparison between the original and debloated data shows no clear difference between them. These metrics were collected on the workstation for all of the models except for DLA/CIFAR. The LSTM/MNIST model while timed on the laptop hardware could not be used



(a) Runtime with 1 epochs of training



(b) Runtime with 2 epochs of training



(c) Runtime with 4 epochs of training

Figure 4.5: Comparison of runtimes between base and slimmed images at different amount of epochs for YOLOv3/COCO

for measuring power consumption as the GPU did not support that functionality. The collected power consumption metrics for the YOLOv3 model can be seen in Figure 4.6.

Along with the power consumption metrics we also collected GPU utilization and GPU memory usage. These metrics show almost identical results to the power consumption. Debloating does not affect them in any noticeable way. For example the memory usage for the original botorch model is 976MB for most of the training while the debloated one is 978MB. This is an extremely small difference and could easily be caused by an inherent inaccuracy of the sensors or logging software. The power consumption graphs for the other models are in Appendix A.

The aforementioned results suggest that the debloating done by the cimplifier tool does not affect the performance of the models. We also compare the accuracy of the original and debloated models and note that the integrity of the model is unaffected by debloating, answering RQ3.

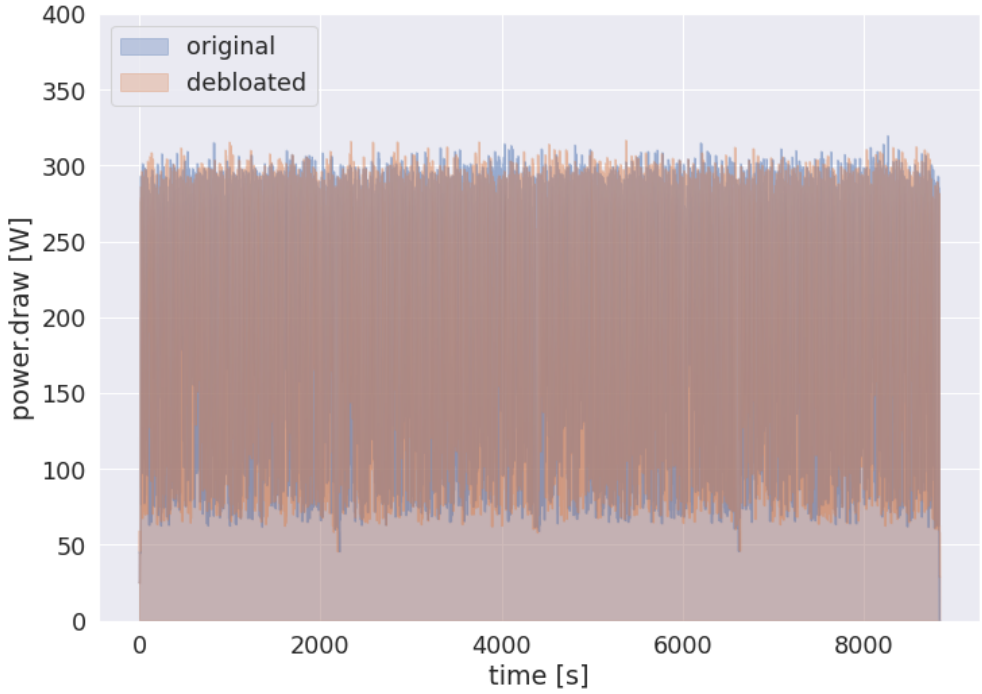


Figure 4.6: Yolo power consumption comparison for 4 epochs

5
Threats to Validity

This chapter goes over the potential threats to validity of this thesis.

5.1 Construct Validity

Construct validity is the extent to which a test measures what it claims, or purports, to be measuring. In the case of measuring storage reductions of the images, it can be presumed that it is quite accurate. Although, in the case of measuring the source and extent of bloat there is a definite degree of uncertainty stemming from the fact that the tools used reported shortcuts as having the size of the destination file. In the case of runtime the measurements should be quite precise as time measurement is a very popular and tested capability of Python. We also feel that we ran enough tests to overcome the fact that the runtimes are influenced by a number of uncontrollable factors such as caching and running background tasks. Overall we feel that the measurements of the GPU utilization, power consumption and memory usage are quite reliable as the measurement tools are provided by the hardware manufacturer. In the case of tests running on AWS the virtualization adds an unknown factor to the results, although the effect should be small and that the debloated and original model ran on the same instance allowing for apt comparisons.

5.2 External Validity

External validity means to what extent the results can be expected to apply to other cases and in general, other than the ones studied here. We feel that the results shown here generalize quite well, as we tested models of different sizes and with different hardware. We tested five models, and we feel confident that the model selection covers an array of popular types, but there is no way guarantee whether these results generalize to models that differ from the ones tested or use a different software stack.

The actual storage reductions gained are very dependant on the original image as having unneeded files, experimental models or other artifacts from the work process will obviously increase the amount of unused files in the end. It is also possible that the actual storage reductions are smaller if great care was put into creating the image with size in mind. Our base images were picked as a ready-to-use solution and as such might differ from images used in the field. When starting from a similar image we can presume a high generalizability for different models, since the main storage reductions came from removing unused Nvidia and Intel kernels

and basically all modern ML workloads are ran using Nvidia GPU and TPU-s and very rarely using CPU-s as the accelerator. We find it very unlikely that a single model would make use of all of the available kernels. Also a very large proportion of models popular today are based on neural networks and the differences between types of NN-s are in the layout and type of neurons used, meaning that the kernels used should have a large amount of overlap. It should be noted that AMD's ROCm platform was recently made available for PyTorch[1] and that the results might be very different for images using AMD accelerated workloads.

In the case of other measurements we think that since these were not affected by the debloating process, that this will hold for other models as well.

5.3 Internal Validity

Internal validity is the extent to which there could be other explanations to describe the observations i.e how valid our claims are. We can confidently say that the storage reductions are caused by the application of the Cimplifier tool as the debloated image was directly created from the original by the tool.

[1] https://pytorch.org/blog/pytorch-for-amd-rocm-platform-now-available-as-python-package/

6
Conclusion and Future Work

The goal of this thesis was to study the existence of bloat in machine learning systems as well as see how existing debloating tools can be applied in this field. The answers to the RQs of this thesis are summarised here.

RQ 1: *How prevalent is code bloat in existing ML systems?* Based on the graphs produced by git-of-theseus and the experimental results showing vast reductions in storage, we can confidently say that bloat exists in ML systems.

RQ 2: *Can existing debloating researches be applied to ML systems? If yes, whatkind of changes are required, and how much bloat can get removed in practice?* We were able to successfully use the cimplifier tool. The tool required a few modifications but these changes were not made necessary by the target being an ML system but from the desire to use the all-in-one mode and our differences in *strace* results. In the case of our experiments we saw on an average around two-fold reduction in image size.

RQ 3: *Can the Machine learning system still able to maintain reasonable service integrity after debloating, in terms of storage, CPU, memory and energy consumption?* Some of the tools we try such as docker-slim and LibFilter results in a program that does not run at all or is unable to use GPU acceleration. Using cimplifier does not in our experience suffer from these problems and our testing shows promising results when it comes to storage reductions (around 50%) but is unable to impact other metrics such as GPU usage, power consumption or workload runtime.

6.1 Future work

This section goes over two paths of work that we were unable to finish as well as name further avenues to improve upon the work presented in this thesis.

6.1.1 Nuitka + LibFilter

A more aggressive experiment is carried out to try to use Nuitka to compile the python code to an executable and then use libfilter to debloat the compiled binary. However, it turns out that this practice does not quite work out. Nuitka is a python compiler written in python. It translates Python modules into C level program then

uses libpython and static C files of its own to execute it in the same way as CPython does. The version of Nuitka we test is 0.6.15.

Nuitka provides three options for the generated binary application. In the default mode, the resulting binary still depends on CPython and used C extension modules being installed, and relies on it to import corresponding libraries. With `--plugin-enable=torch --plugin-enable=numpy --plugin-enable=pkg-resources` option enabled, the LSTM_MNIST script successfully compiles to a library, and it can run without any problems for the training steps. However, after using the libfilter tool to debloat the executable file, running it leads to a core-dump. This means that libfilter considered some statements to be unnecessary and replaced them with *hlt* which causes a core-dump when the statement is actually ran i.e it was not actually unnecessary.

The other two modes are *standalone* and *onefile* modes. Both modes are aimed to make the distribution of the created binary easier. However, *onefile* mode is only suggested to be used when *standalone* mode is successful. Since running in *onefile* mode removes some of the debuggin log output it makes the debugging more difficult. Unfortunately, we didn't make *standalone* mode work for PyTorch. When compiling with the PyTorch module for LSTM/MNIST, the produced binary complains about missing python files or torch lib share libraries.

6.1.2 Singularity

Originally cimplifier targets docker images for debloating. We try to extend to functionality of cimplifier by writing an equivalent implementation for singularity. Singularity is an open-source containerization software, that unlike docker does not require root privileges for containers. The decision to try making cimplifier work with singularity is mostly undertaken so cimplifier can be tested using a HPC cluster we have access to. Unfortunately, it turns out that the amount of work needed to implement this functionality is larger than expected and the decision was made to instead use AWS for testing larger workloads.

6.1.3 Additional avenues

The work can be extended by running more workloads that differ either by the ML system used, the type of model or the accelerator used. For example further testing should be done using the beta version of PyTorch that is based on ROCm instead of CUDA for GPU acceleration. As the CUDA libraries were a large part of the removed files, it is possible that using the ROCm backend could offer greatly different results. Also all of the models tested here use PyTorch or an extension of PyTorch in the form of BoTorch, so experiments using other libraries such as TensorFlow might offer greater insight. As mentioned before a lot of existing debloating tools target C/C++ or code that compiltes into LLVM bitcode, so a great direction would be to try those tools and see if they work on the code generated by Nuitka.

Bibliography

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 2503–2511.

[2] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta, "Does lean imply green? a study of the power performance implications of java runtime bloat," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 259–270, 2012.

[3] P. Henderson, J. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau, "Towards the systematic reporting of the energy and carbon footprints of machine learning," *arXiv preprint arXiv:2002.05651*, 2020.

[4] E. Strubell, A. Ganesh, and A. Mccallum, "Energy and policy considerations for modern deep learning research," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 13 693–13 696, 04 2020.

[5] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica *et al.*, "Exascale deep learning for climate analytics," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 649–660.

[6] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at google," in *Proceedings of the Third International Workshop on Managing Technical Debt*, 2012, pp. 1–6.

[7] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *Proceedings of the 2013 international symposium on memory management*, 2013, pp. 119–130.

[8] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 869–886.

[9] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1697–1714. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/azad

[10] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 268–278.

[11] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta, "The interplay of software bloat, hardware energy proportionality and system bottlenecks," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, ser. HotPower '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2039252.2039253

[12] K. Nguyen, K. Wang, Y. Bu, L. Fang, and G. Xu, "Understanding and combating memory bloat in managed data-intensive systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 4, pp. 1–41, 2018.

[13] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394.

[14] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1009–1022.

[15] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray, "Refactoring software, architectures, and projects in crisis," 1998.

[16] A. Zheng, "The challenges of building machine learning tools for the masses," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[17] H. Xue, Y. Chen, G. Venkataramani, and T. Lan, "Hecate: Automated customization of program and communication features to reduce attack surfaces," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2019, pp. 305–319.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] T. Kim and D. Wu, "Feast 2017: The second workshop on forming an ecosystem around software transformation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2633–2634. [Online]. Available: https://doi.org/10.1145/3133956.3137052

[20] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 869–886. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[22] F. Beaufays, "The neural networks behind google voice transcription," *Google Research blog*, 2015.

[23] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[24] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization," in *Advances in Neural Information Processing Systems 33*, 2020. [Online]. Available: http://arxiv.org/abs/1910.06403

[25] A. Nichol and P. Dhariwal, "Improved denoising diffusion probabilistic models," 2021.

[26] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," 2020.

[27] F. Yu, D. Wang, E. Shelhamer, and T. Darrell, "Deep layer aggregation," 2019.

[28] Docker-slim: Don't change anything in your docker container image and minify it by up to 30x. [Online]. Available: https://github.com/docker-slim/docker-slim

[29] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 476–486.

[30] L. Gelle, H. Saidi, and A. Gehani, "Wholly!: a build system for the modern software stack," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2018, pp. 242–257.

[31] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 70–83. [Online]. Available: https://doi.org/10.1145/3359789.3359823

[32] B. Shteinfeld, "Libfilter : Debloating dynamically-linked libraries through binary recompilation," 2019.

[33] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, *JShrink: In-Depth Investigation into Debloating Modern Java Applications*. New York, NY, USA: Association for Computing Machinery, 2020, p. 135–146. [Online]. Available: https://doi.org/10.1145/3368089.3409738

[34] K. Macias, M. Mathur, B. R. Bruce, T. Zhang, and M. Kim, *WebJShrink: A Web Service for Debloating Java Bytecode*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1665–1669. [Online]. Available: https://doi.org/10.1145/3368089.3417934

[35] The mnist database of handwritten digits. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[36] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2921–2926.

[37] V. U. Prabhu and A. Birhane, "Large image datasets: A pyrrhic win for computer vision?" 2020.

[38] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2015.

A

Graphs of results

This appendix shows the figures and tables that were not shown in the main Results chapter.

Epochs	Original	Debloated
5	246.19 ±0.85	253.06 ±2.72
15	738.33 ±7.06	740.83 ±6.58
25	1237.69 ±7.76	1239.37 ±3.69

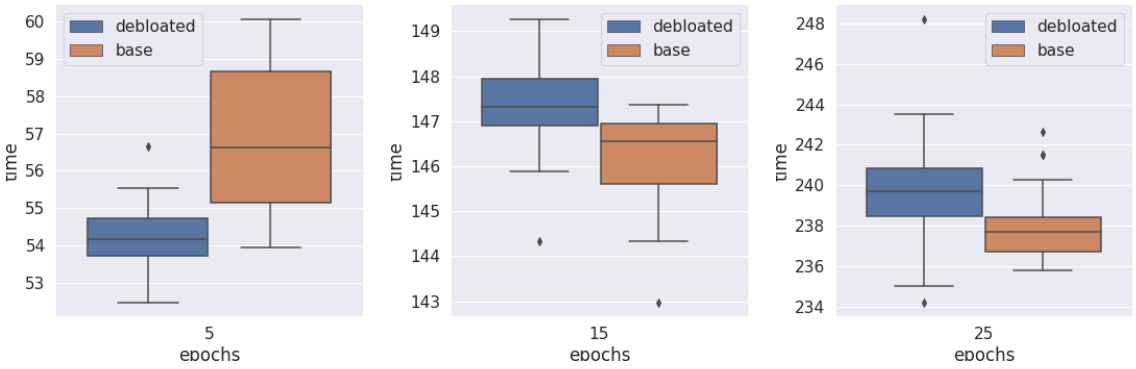
Table A.1: The average time elapsed for training and evaluating the DLA/CIFAR-10 model

Steps	Original	Debloated
1024	284.60 ±5.95	286.89 ±6.15
4096	1117.21 ±27.94	1113.92 ±27.37

Table A.2: The average time elapsed for training and evaluating the improved Diffusion/COCO model

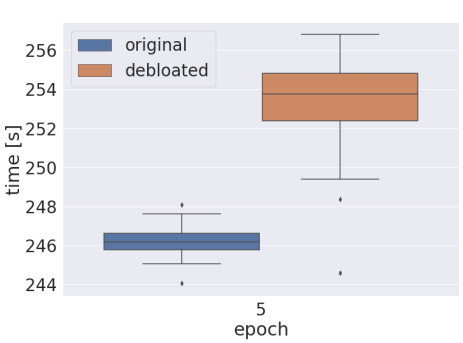
Iterations	Original	Debloated
3	1401.62 ±4.86	1406.39 ±7.41
6	2949.47 ±101.73	2948.45 ±99.22
9	4408.77 ±158.37	4381.38 ±170.92

Table A.3: The average time elapsed for training and evaluating the botorch model

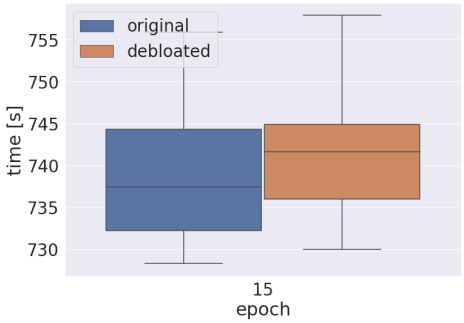


(a) Runtime with 5 epochs of training **(b)** Runtime with 15 epochs of training **(c)** Runtime with 25 epochs of training

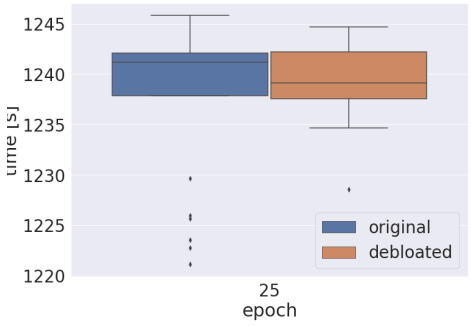
Figure A.1: Comparison of runtimes between base and slimmed images at different amount of epochs for LSTM/MNIST



(a) Runtime with 5 epochs of training

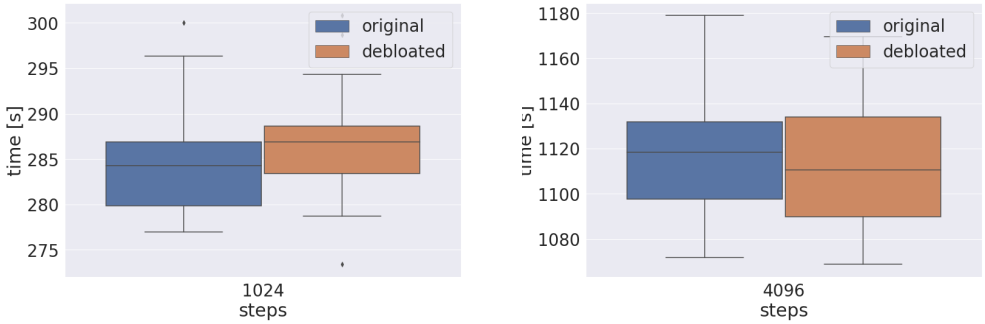


(b) Runtime with 15 epochs of training



(c) Runtime with 25 epochs of training

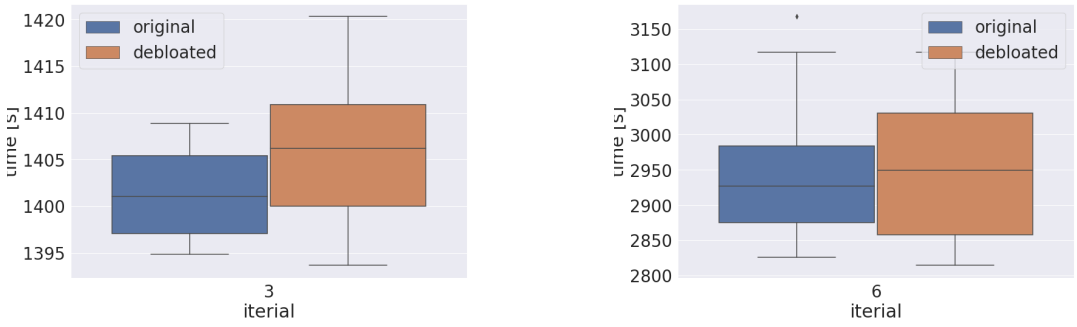
Figure A.2: Comparison of runtimes between base and slimmed images at different amount of epochs for DLA/CIFAR-10



(a) Runtime with 1024 steps of training

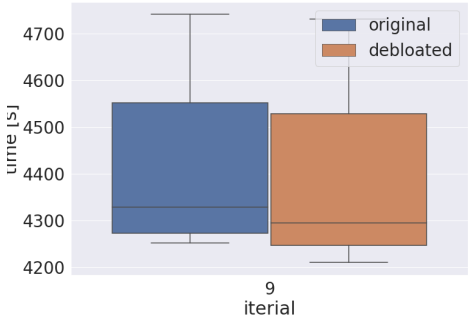
(b) Runtime with 4096 steps of training

Figure A.3: Comparison of runtimes between base and slimmed images at different amount of iterations for diffusion/COCO



(a) Runtime with 3 iterations of training

(b) Runtime with 6 iterations of training



(c) Runtime with 9 iterations of training

Figure A.4: Comparison of runtimes between base and slimmed images at different amount of iterations for Botorch/Bayesian Optimization

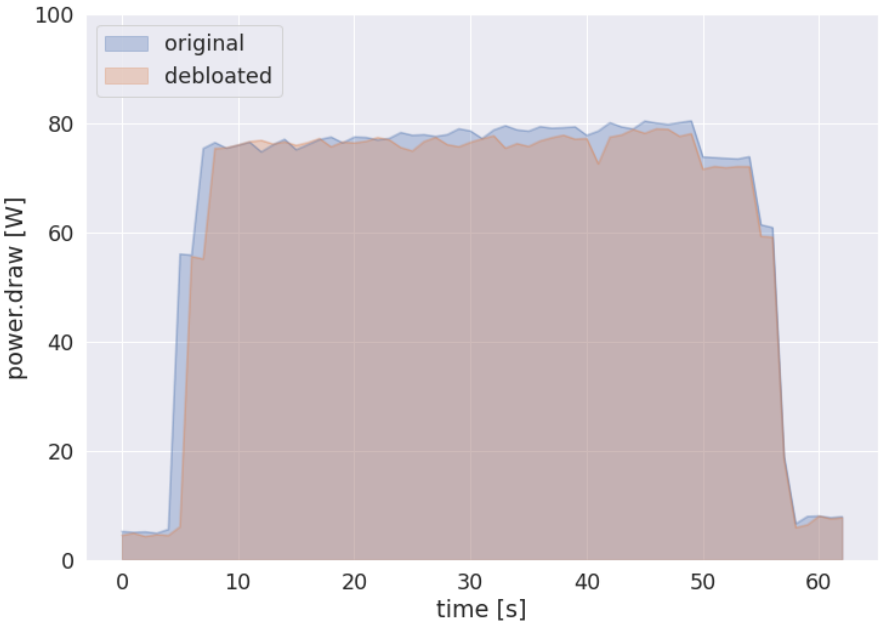


Figure A.5: LSTM/MNIST power consumption comparison for 5 epochs

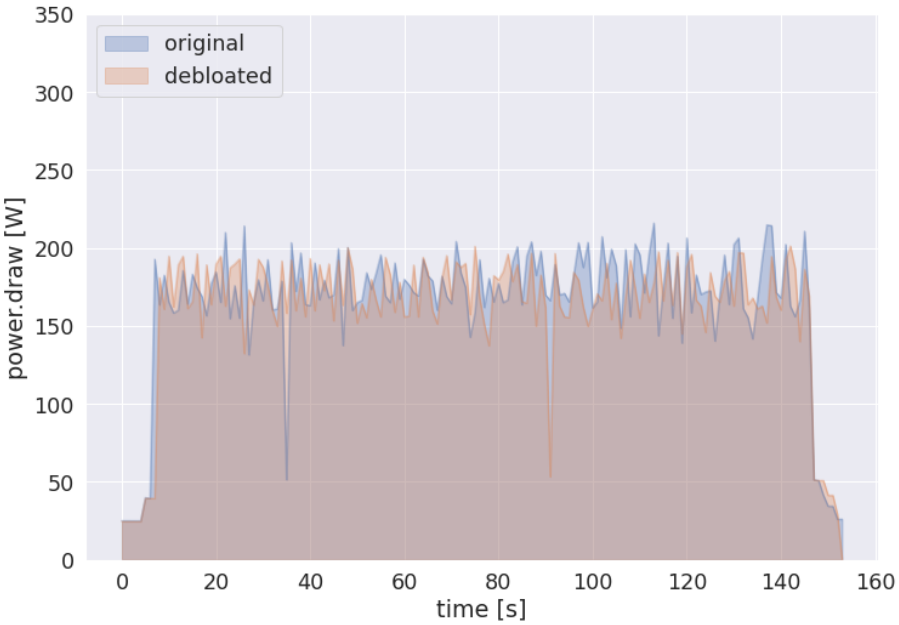


Figure A.6: DLA/CIFAR-10 power consumption comparison for 5 epochs of training

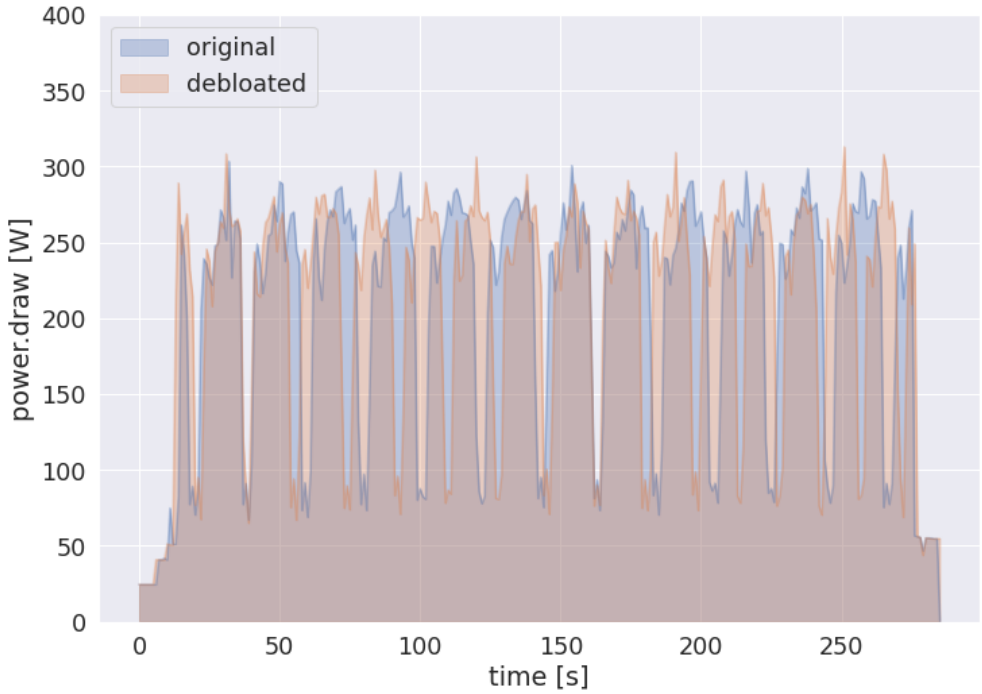


Figure A.7: Diffusion power consumption comparison for 1024 steps

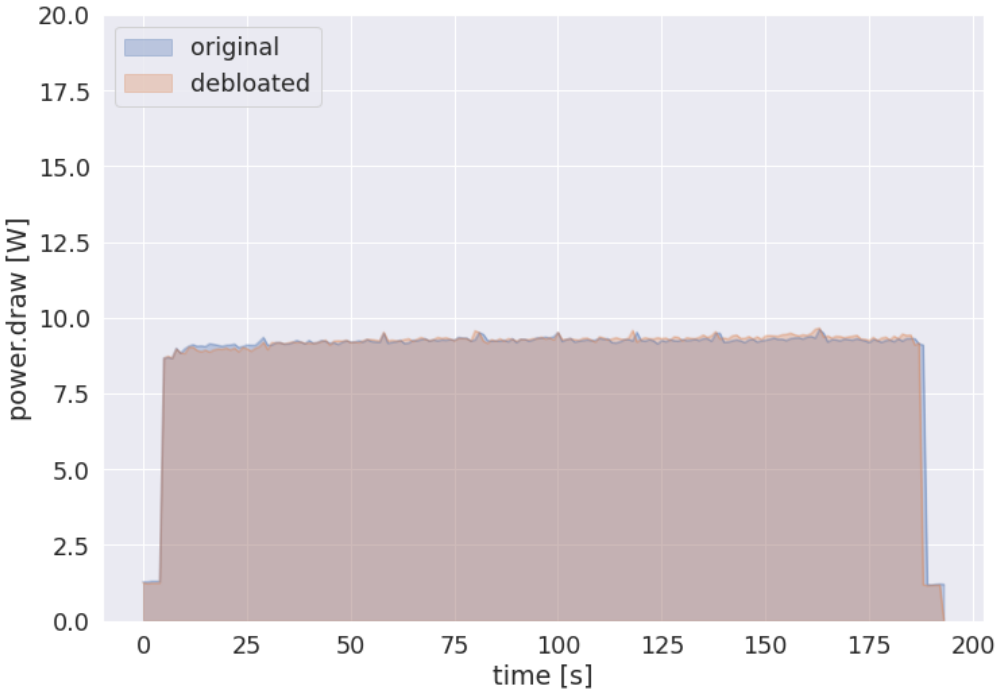


Figure A.8: Botorch power consumption comparison for