Université de Montréal

Programming tools for intelligent systems

with a case study in autonomous robotics

par

Breandan Considine

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Discipline

juillet 2019

© Breandan Considine, 2019

Summary

In which we describe tools for programming intelligent systems. (TODO)

Acknowledgements

Thank you! (TODO)

Mom, Gimmey, Mark, Dad, Hanneli Tavante for teaching me type theory and the beauty of functional programming, Siyan Wang for his friendship and adventures, Xiaoyan Liu for planting in me the seed of mathematics, Uncle Andy for watering the seed, Aunt Shannon, and Adam Devoe, Jacquie Kirrane for encouraging me to pursue grad school, Arthur Nunes-Harwitt for teaching me how to think like a computer scientist, Renee Miller for sparking my interest in neural science, Ian Clarke for teaching me how to write clean code, a clever new language called Kotlin and multi-armed bandits, Hadi Hariri for putting more faith than in me than I deserved, Eugene Petrenko for showing me the magic of DSLs and giving me advice about grad school, Rusi Hristov for teaching me Docker and bash. Isabella Albuquerque and João Monteiro for teaching me the meaning of hard work and how to write a good paper. Manfred Diaz and Maxime Chevalier Boisvert for the inspiration, conversations and feedback. Florian Golemo for setting the bar of excellence in research. Ryan Turner, Saikrishna Gottipati and Vincent Mai for the interesting conversations and adventures. Pascal Lamblin, Olivier Breleux and Bart van Merriënboer for blazing the trail, teaching me about Myia and lighting the path between ML and PL, Dmitry Serdyuk for introducing me to Montréal, Christian Perone for introducing me to Pytorch, Kiran Gopinathan for providing useful comments and feedback, Miltos Allamanis for showing me there is a place for SE in ML, Celine Begin at the Université de Montréal for giving a stranger a chance on a cold winter's eve, Stefan Monnier for thoughtfully and thoroughly replying to my long-winded emails, Andrea Censi for his encouragement, (last but not least) my wonderful advisors Liam Paull for taking a chance on an out-of-distribution sample, providing strong gradients and assigning far more credit to me than was deserved, and Michalis Famelis for teaching me the value of intuistionistic logic, formal methods, and self-discipline.

Table des matières

Summary iii

Acknowledgements v

Liste des tableaux xi

Liste des figures xiii

Chapitre 1. Introduction 3

1.1. Stages in the software development lifecycle 5

1.2. Designing intelligent systems 6

1.3. Implementation: Languages and compilers 8

1.4. Testing and validation 10

1.5. Software reproducibility and maintenance 11

1.5.1. Case Study 14

1.5.2. Iconography 14

Chapitre 2. Design: Programming tools for robotics 17

2.1. Introduction to the Robot Operating System 18

2.2. Prerequisities 21

2.3. Plugin development 21

2.3.1. Refactoring 22

2.3.2. The Parser 22

2.3.3. Running and debugging 23

2.3.4. User interface 24
2.4. Future work 27
Chapitre 3. Implementation: languages and compilers 29
3.1. Automatic differentiation 30
3.2. Differentiable programming 32
3.3. Static and dynamic languages 35
3.4. Imperative and functional languages 35
3.5. Kotlin 36
3.6. Kotlin∇ 37
3.7. Usage 39
3.8. Type systems 40
3.9. Shape safety 41
3.10. Testing 45
3.11. Operator overloading 47
3.12. First class functions 48
3.13. Numeric Tower 48
3.14. Algebraic data types 49
3.15. Multiple Dispatch 50
3.16. Extension Functions 51
3.17. Automatic, Symbolic Differentiation 52
3.18. Coroutines 52
3.19. Comparison 53

3.20. Future work 53

Chapitre 4. Testing and validation 55

4.0.1. Unit Testing 56

4.0.2. Integration Testing 56

4.0.3. Fuzz Testing 57

4.0.4. Property-based Testing 57

4.0.5. Metamorphic testing 58

4.0.6. Adversarial Testing 60

Chapitre 5. Software Maintenance and Reproducibility 65

5.1. Dependency management 65

5.2. Operating systems and virtualization 66

5.3. Containerization 67

5.4. Docker Introduction 69

5.4.1. Creating an image snapshot 70

5.4.2. Writing an image recipe 72

5.4.3. Layer Caching 74

5.4.4. Volume Sharing 78

5.4.5. Multi-stage builds 79

5.5. ROS and Docker 80

5.6. Duckiebot Development using Docker 81

5.6.1. Flashing a bootable disk 82

5.6.2. Web interface 82

5.6.3. Testing ROS 83

5.6.4. Build and deployment 83

5.6.5. Multi-architecture support 84

5.6.6. Running a simple HTTP file server 84

5.6.7. Testing the camera 85
5.6.8. Graphical User Interface Tools 85
5.6.9. Remote control 86
5.6.10. Camera Calibration 86
5.6.11. Wheel Calibration 86
5.6.12. Lane Following Demo 87
5.7. Retrospective 87
Chapitre 6. Case study: application for autonomous robotics 91
6.1. Design 91
6.2. Implementation 91
6.3. Testing and validation 91
6.4. Containerization 91
Chapitre 7. Conclusion 93
7.1. Future work 93
7.1.1. Requirements Engineering 93
7.1.2. Continuous Delivery and Continual Learning 94
Bibliography 95
Appendix A. Implementation: languages and compilers ..... A-i
A.1. Linear Regression from an AD Perspective ..... A-i
A.1.1. Finite Difference Method ..... A-i
A.1.2. Partial Differentiation ..... A-ii
A.1.3. Matrix Solution ..... A-iii

Liste des tableaux

3.1 Kotlin∇'s shape system specifies the output shape for matrix arithmetic. 42

3.2 Comparison of AD libraries. The ✍ symbol indicates work in progress. 54

Liste des figures

1.1 Royce's original Waterfall model, originally intended to describe the software development process, but the same sequence can be found in most engineering disciplines. We use it to help guide our discussion and frame our work inside of this process model. 5

2.1 Unique downloads of Hatchery between the time of its release and June 2019. `https://plugins.jetbrains.com/plugin/10290-hatchery`. 18

2.2 A typical ROS application contains a large graph of dependencies. 19

2.3 ROS run configuration. Accessible via: Run » Edit Configurations » + » ROS Launch . . . 24

2.4 The evolution of code. On the left are languages that force the user to adapt to the machine. To the right are increasingly flexible representations of source code. 25

2.5 Projectional editors such as MPS [Voelter and Solomatov, 2010, Pech et al., 2013] (shown above) are able to render plaintext source code in alternate formats for reading and manipulation. 26

2.6 Hatchery UI supports syntax highlighting, validation and project navigation. 26

2.7 Detection of local ROS packages. Accessible via: File » Settings » ROS config 27

3.1 *Differentiable programming* includes neural networks, but more broadly, arbitrary differentiable programs which use automatic differentiation and gradient-based optimization to approximate a loss function. *Probabilistic programming* [Carpenter et al., 2017, Gorinova et al.] is an emerging generalization of probabilistic graphical models, and uses various forms of Markov chain Monte Carlo (MCMC) and differentiable inference to approximate a probability density function. 33

3.2 Two equivalent programs, both implementing the function $f(l_1, l_2) = l_1 \cdot l_2$. 36

3.3 Adapted from van Merriënboer et al. [2018]. Kotlin∇ models are data structures, constructed by an embedded DSL, eagerly optimized, and lazily evaluated at runtime. 38

3.4 Implicit DAG constructed by the original expression, seen above. 40

5.1 Virtualization is a very resource expensive proposition. Containerization is cheaper, as it shares a kernel with the host OS. Emulation allows us to emulate hardware as software. Any of these methods can be used in conjunction with any other method. 67

5.2 Containers live in user space. By default they are sandboxed from the host OS and sibling containers, but unlike VMs, share a common kernel with each other and the host OS. All system calls are passed through host kernel. 68

5.3 Container infrastructure. **Left**: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we build ARM artifacts, which can be emulated on x86 via qemu [Bellard, 2005]. **Right**: Reinforcement learning stack. Build artifacts are trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their architecture, may be run on an ARM device using an accelerator. 81

5.4 Browser interface for individual Duckiebots. It is provided by Portainer, a RESTful web dashboard, which wraps the Docker CLI and offers support for container management, configuration, networking and terminal emulation (seen above), provided by xterm.js which is accessible at: `http://DUCKIEBOT_NAME:9000/#/container/container_name` `"Console"` 83

5.5 Early prototype of the Docker image hierarchy. Chaining unversioned autobuilds without disciplined unit testing tends to produce an undesirable domino effect, where breaking changes are allowed to propagate downstream, resulting in a cascade of silent failures. 88

5.6 The AI Driving Olympics, a primary use case for the system described above. . . . 89

Chapitre 1

Introduction

> There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought.
>
> –Leslie Lamport

Computational complexity is of such concern in computer science that a great deal of the field is dedicated to understanding it through the lens of function analysis and information theory. In software engineering, researchers are primarily interested in the complexity of building software – the digital manifestation of algorithms on physical hardware. One kind of software complexity is the cognitive effort required to understand a program.[1] While today's software is becoming rapidly more intelligent, it shows few signs of becoming more intelligible. Better tools are needed for managing the complexity in building software systems.

The objective of this thesis is to develop methods that reduce the cognitive effort required to build intelligent systems, using developer tools, programming language abstractions, automated testing, and virtualization technology.

Broadly speaking, intelligent systems differ from ordinary software systems in that they enable machines to detect patterns, perform tasks, and solve problems which they are not explicitly programmed to solve and which human experts were previously incapable of solving by hard-coding explicit rules. Typically, these systems are able to:

[1] This can be approximated by various metrics like cyclomatic or Halstead complexity.

(1) learn generalizable rules by processing large amounts of data

(2) tune a large number of free parameters (thousands to billions)

(3) outperform well-trained humans in domain-specific tasks

While the idea of intelligent systems has existed for decades, three critical developments made modern intelligent systems ultimately successful. First, computer processing power has become faster, cheaper, and much more readily available. Similarly, the digitalization of new datasets has made vast amounts of information available, and data storage costs have plummeted dramatically. (A $5 thumb drive today has 200 times more storage capacity than a 2,000 pound, 5 MB, IBM hard drive that leased for $3,000 per month in 1956.) Most importantly, has been the development of more efficient learning algorithms.

In recent years, computer science and software engineering has made significant strides in building and deploying intelligent systems. Nearly every mobile computer in the world is able to detect objects in images, perform speech-to-text and language translation. These breakthroughs were the direct result of fundamental progress in neural networks and representation learning. Also key to the success of modern intelligent systems was the adoption of collaborative open source practices, pioneered by the software engineering community. Software engineers developed automatic differentiation libraries like Theano [Al-Rfou et al., 2016], Torch [Collobert et al., 2002] and Caffe [Jia et al., 2014], and built many popular simulators for reinforcement learning, the ease of use and availability of which was crucial for popularizing deep learning techniques.

Intelligent systems are widely deployed in virtual settings like data science and cloud services. But even with the tremendous success of machine learning algorithms in fully-observable domains like image recognition and speech processing, intelligent systems have yet to be widely adopted in robotics (at the time of writing this thesis). This dilemma can be partly attributed to various theoretical problems such as domain adaption and transfer learning. Yet with the proven capabilities of modern learning algorithms, exponential increase in processing power, and decades-long effort in building physically-embodied intelligent agents, we should have more progress to show. Why has this goal evaded researchers for so long? One reason, we conjecture, is a lack of programming tools and abstractions for designing, developing, deploying and evaluating intelligent systems. In practice, these activities consume a large amount of cognitive effort without the right set of tools and abstractions.

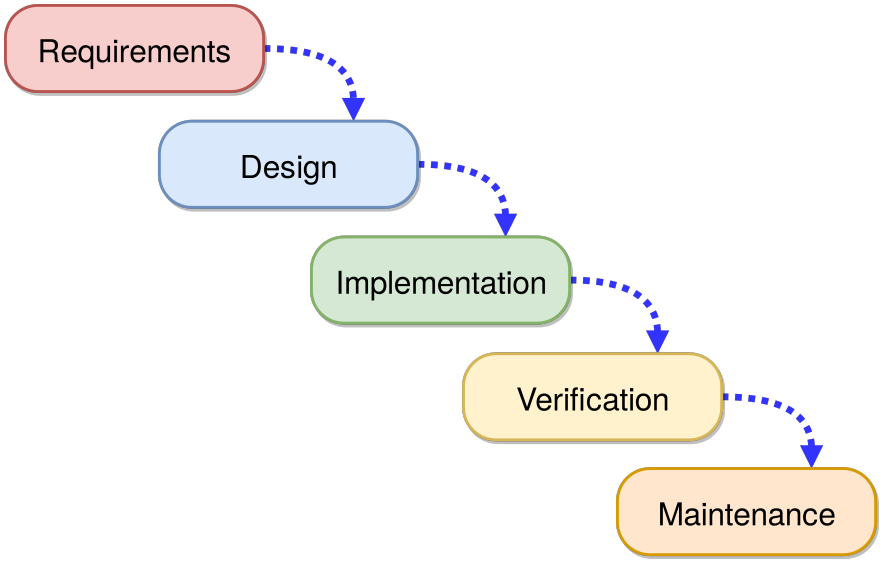


Fig. 1.1. Royce's original Waterfall model, originally intended to describe the software development process, but the same sequence can be found in most engineering disciplines. We use it to help guide our discussion and frame our work inside of this process model.

In this thesis, we explore several tools that facilitate the process of building intelligent systems, and which reduce the cognitive effort required to understand an intelligent program. First, we demonstrate an integrated development environment that assists users writing robotics software (Chapter 2). Next, we show a type-safe domain-specific language for differentiable programming, an emerging paradigm in deep learning (Chapter 3). To test this application, we use a set of techniques borrowed from property-based testing [Fink and Bishop, 1997] and adversarial learning [Lowd and Meek, 2005] (Chapter 4). Docker containers [Merkel, 2014] are used to automate the building, testing and deployment of reproducible robotics applications on heterogeneous hardware platforms (Chapter 5). Finally, as a proof of concept for these ideas, we build an intelligent system comprised of a mobile autonomous vehicle and an Android mobile application, using the tools previously described (Chapter 6).

1.1. Stages in the software development lifecycle

In traditional software engineering, the Waterfall Model (Figure 1.1) is a classical software process model comprised of five stages. While the Waterfall Model was an early model in software engineering, the same activities it describes can be found in most engineering process models. We propose contributions to four such areas: design in Chapter 2, implementation in Chapter 3, verification in Chapter 4 and maintenance in Chapter 5.

1.2. Designing intelligent systems

Today's software systems are deeply complex entities. Gone are the days where a solitary programmer, even a very skilled one, could maintain a large software system alone. To effectively scale modern software systems, programmers must pool their mental capacity to form a knowledge graph. Software projects which rely on a small set of maintainers tend to perish due to the so-called *bus factor* [Cosentino et al., 2015] – large portions of the knowledge graph are locked inside someone's head. Successful software projects learn how to distribute this graph and form connections to the outside world. The knowledge graph which accumulates around a software project contains facts, but it also contains workflows for programming, debugging, and delivery – common paths through the labyrinth of software development. Components of this graph can be committed to writing, but documentation is time-consuming and grows stale over time. What is needed is a system that offers the benefits of documentation without the burdens of maintenance.

The development of software systems has a second component, the social graph. The social graph of a successful software project contains product designers, managers and software engineers who work in concert to build software that is well-designed, cohesive, and highly performant. Sometimes this means revising the specification to accommodate engineering challenges, or rewriting source code to remove technical debt. Software design is a multi-objective optimization process and requires contributors with a broad set of skills and common set of goals. To produce software that approximates the criteria of its stakeholders, developers are asked to provide rapid prototypes, and continuously integrate user feedback. Yet today's software systems are larger and more unwieldy than ever. So finding ways to collaborate more efficiently is critical to building and maintaining intelligent systems.

First, let us consider the mechanical process of writing software with a keyboard.

Integrated development environments (IDEs) can assist developers building complex software applications by automating certain repetitive programming tasks. For example, IDEs perform static analyses and inspections for catching bugs quickly. They provide completion, refactoring and source code navigation, and they automate the process of building, running and debugging programs. While these tasks may seem trivial, their automation promises increased developer productivity by delivering earlier feedback, detecting clerical errors, and

allows developers to focus on fundamental design tasks. Rather than being forced to concentrate on the structure and organization of text, if developers are able to manipulate code at a semantic level, they will be much happier and more productive. Furthermore, by automating mechanical tasks in software development, these tools allow developers to direct their attention to the fundamental activity of writing and understanding programs.

But what are IDEs really doing? They are guiding developers through the knowledge graph of a software project. Consider what a new developer must learn to get up to speed: in addition to learning the language, developers must learn to use libraries and frameworks (arguably languages in their own right). They must become familiar with command line tools for software development, from build tools to version control and continuous integration. They must become familiar with the software ecosystem, programming styles, conventions and development workflows. And they must learn how to collaborate on a distributed team of developers. By automating common tasks in an interactive programming environment and making the graph connectivity explicit through document markup [Goldfarb, 1981] and projectional editing [Voelter et al., 2014], a well-designed IDE is a tool for graph traversal. It should come as little surprise IDEs are really graph databases.

In some aspects, the development of intelligent systems is no different than classical software engineering. The same principles and best-practices which guide software engineering are also applicable to intelligent systems. And the same activities, from design to maintenance will continue to play an important role in building intelligent systems. But in other respects, the generic programming tools used to develop traditional software will require domain-specific adaptations for learning systems to become truly first class citizens in the next generation of intelligent software, particularly in the case of robotics development.

Towards that goal, we developed an IDE for the Robot Operating System (ROS) called Hatchery. It supports a number of common workflows for ROS development, such as creating ROS nodes, Gazebo simulator integration, support for remote debugging, static analysis, autocompletion and refactoring. In Chapter 2 we discuss the implementation of these features and some of the challenges of building language support, programming tools and integrating with the ROS middleware. We argue that such tools reduce the cognitive complexity of building ROS applications by adopting explicit coding conventions, annotating unstructured text and automating repetitive development tasks.

1.3. Implementation: Languages and compilers

In the early days of machine learning, it was widely believed that human-level intelligence would emerge from a sufficiently descriptive first-order logic. By accumulating a database of facts and their relations, researchers believed they could use symbolic reasoning to bypass learning altogether. This rule-based approach dominated a large portion of early research in artificial intelligence and considerable effort was poured into the creation of domain-specific ontologies to capture human knowledge. Despite the best efforts of roboticists, signal processing engineers and natural language researchers, *expert systems* were unable to scale to real-world applications, causing a great disillusionment in artificial intelligence research for a number of decades. While computer scientists underestimated the difficulty of learning, expert systems excelled in areas where current machine learning systems struggle such as classical reasoning and interpretability, and there is growing evidence to suggest many of these ideas were simply ahead of their time. In our work, we take inspiration from some early work in symbolic reasoning, type systems and functional programming.

What was finally shown to scale, is the idea of connectionist learning. By nesting random function approximators, called perceptrons, and updating the free parameters using back-propagation [Werbos et al., 1990, Rumelhart et al., 1988], the resulting system is capable of learning a surprising amount of intelligent behavior. The approach, termed artificial neural networks (ANNs), can be traced back to the mid-20th century [Ivakhnenko and Lapa, 1965, Rosenblatt, 1958], but was not fully-realized in silico until after the widespread availability of cheap computing and large datasets [LeCun et al., 2015]. In theory, a single layer of nesting is able to approximate any continuous differentiable function [Hornik et al., 1989], but in practice, learning requires composing many such approximators in a deeply nested fashion, hence the term, *deep neural networks* (DNNs). The importance of depth was suspected for many years, but the original backpropagation algorithm had difficulty training DNNs due to the vanishing gradient problem [Bengio et al., 1994]. Solving this problem required a number of adaptations and many years to fully debug. It was not until 2013 when deep learning was competitive with humans in a number of domains.

While it took fundamental research in deep learning to realize the connectionist blueprint, the success of modern deep learning can be partly attributed to software tools for calculating mathematical derivatives, a key step in the backpropagation algorithm. Although it has

yet not been established if or how derivatives might be calculated in biological circuits, derivatives are essential for ANN training. For many years, the symbolic form of these derivatives were analytically derived when prototyping a new neural network architecture, a tedious and error-prone process. There is a well-known algorithm in the scientific computing community dating back to the 1970s, called *automatic differentiation* (AD) [Linnainmaa, 1970, Griewank et al., 1989], which is able to calculate derivatives for arbitrary differentiable functions. But surprisingly, it was not until much later, after the development of Theano [Al-Rfou et al., 2016] when AD became widely adopted in the machine learning community. This library alone greatly accelerated the pace of deep learning research and spurred the development of others like TensorFlow [Abadi et al., 2016] and PyTorch [Paszke et al., 2017].

Intelligent systems engineers must think carefully about languages and abstractions. If developers are required to implement backpropogation by hand, they will have scarce time to think about the high-level characteristics of these systems. Similarly, if programming abstractions are too specific, small variations will require costly reimplementation. This is no different from traditional software engineering – as engineers, we need to choose the right abstractions for the task at hand. Too low-level and the design is lost in the details, too abstract and the details are lost completely. With deep learning, the necessity of choosing good abstractions is even more important, as the relationship of between source code and runtime behavior is already difficult to debug, due to the complexity of neural networks and array programming. One component of that complexity can be found in the type system.

The majority of existing AD frameworks for machine learning are implemented in dynamically typed languages like Python, Lua and JavaScript, with some early implementations including projects like Theano, Torch and Caffe. Similar ideas have arisen in statically-typed, functional languages, such as Haskell (Stalin∇ Pearlmutter and Siskind [2008b]), Scala (Nexus Chen [2017]), F# (DiffSharp Baydin et al. [2015b]), Swift [Lattner and Wei, 2018], et al., but most of these are unable to check the shape of multidimensional arrays in their type system, and those which do are typically implemented in experimental languages with sophisticated type-level programming features. In our work, we demonstrate the viability of shape-checked differentiable programming in a widely-used language with parametric types. This ensures that programs on matrices, if they compile, are the correct shape and can be numerically evaluated at runtime.

Kotlin∇ is an embedded domain-specific language (eDSL) for differentiable programming in a language called Kotlin, a statically-typed programming language with support for asynchronous programming and multi-platform compilation. In Kotlin∇ (Chapter 3), we describe an algebraically-grounded implementation of forward-mode automatic differentiation with shape-safe tensor operations. Our approach differs from existing AD frameworks in that Kotlin∇ is the first shape-safe AD library fully compatible with the Java type system, requiring no metaprogramming, reflection or compiler intervention to use.

1.4. Testing and validation

Most naturally arising phenomena, particularly those related to vision, planning and locomotion are high dimensional creatures. Richard Bellman famously coined this problem as the "curse of dimensionality". Our physical universe is populated by problems which are simple to pose, but impossible to solve inside of it. Claude Shannon, a contemporary of Bellman, calculated the number of unique chess games to exceed 10^{120}, more than the number of atoms in the universe by approximately 40 orders of magnitude [Shannon, 1950]. At the time, it was believed that such problems would be insurmountable without fundamental breakthroughs in algorithms and computing machinery. Indeed, while Bellman or Shannon did not live to see the day, it took only half a century of progress in computer science before solutions to problems with the same order of complexity, first solved in the Cambrian explosion 541 million years ago, were approximated to a competitive margin in modern computers [Pratt, 2015].

While computer science has made enormous strides in solving the common cases, Bellman's curse of dimensionality still haunts the long tail of machine learning, particularly for distributions that are highly dispersed. Because the dimensionality of many real world functions that we would like to approximate is intractably large, it is difficult to verify the behavior of a candidate solution in all regimes, especially in settings where failure is rare but catastrophic. According to some studies, humans drivers average 1.09 fatalities per hundred million miles [Kalra and Paddock, 2016]. A new software build for an autonomous vehicle would need to accumulate 8.8 billion miles of driving in order to approximate the fatality rate of a human operator to within 20% with a 95% confidence interval. Deploying such a scheme in the real world would be logistically, not to mention ethically, problematic.

Realistically speaking, intelligent systems need better ways to practice their skills and probe the effectiveness of a candidate solution within a limited computational budget, without harming humans in the process. The goal of this testing is to highlight errors, but ultimately to provide feedback to the system. In software engineering, the real system under test are the ecosystem of humans and machines which provide each other's means of subsistence. The success of this arrangement depends on an external testing mechanism to enforce a minimum bar of rigor, typically some form of hardware- or human-in-the-loop testing. If the testing mechanism is not somehow opposed to the system under test, an intelligent system can deceive itself, which is neither in the system's nor its users' best interests.

In this Chapter 4 we present preliminary work in adversarial fuzz testing, building on prior literature in adversarial learning, metamorphic and property-based testing. A similar technique is used for testing the numerical correctness of Kotlin∇'s implementation. We present a simple algorithm for property-based shrinking using projected gradient descent and suggest several future directions for improvement.

1.5. Software reproducibility and maintenance

One of the challenges of building intelligent systems and programming in general, is the problem of reproducibility. Software reproducibility has several challenging aspects, including hardware compatibility, operating systems, file systems, build systems, and runtime determinism. While writing programs and feeding them directly into a computer may have once been common practice, today's source code is far too removed from its mechanical realization to be meaningfully executed in isolation. Today's handwritten programs are like schematics for a traffic light – built inside a factory, and which require a city's-worth of infrastructure, cars, and traffic laws to serve their intended purpose. Like traffic lights, source code does not exist in a vacuum – built by compilers, interpreted by virtual machines, executed inside an operating system, and which following a specific communication protocol – programs are essentially meaningless abstractions outside this context.

As necessary in any good schematic, much of the information required to build a program is divided into layers of abstraction. Most low-level instructions carried out by a computer during the execution of a program were not written nor intended to be read by the programmer and have since been automated and long forgotten. In a modern programming

language like Java, C# or Python, the total information required to run a simple program often numbers in the trillions of bits. A portion of that data pertains to the software for building and running programs, including the build system, software dependencies, and development tools. Part of the data pertains to the operating system, firmware, drivers, and embedded software. For most programs, such as those found in a typical GitHub repository[2], a vanishingly small fraction correspond to the source code itself.

Applied machine learning shares many of the same practical challenges as traditional software development, with source code, release and dependency management. The current process of training a deep learning model can be seen as particularly long compilation step, but it differs significantly in that the source code is a high-level language which does not directly describe the computation being performed, but is a kind of meta-meta-program. The first meta-program describes the connectivity of a large directed graph (i.e. a computation graph or probabilistic graphical model), parameterized by weights and biases. The tuning of those parameters is another meta-program, describing the sequence of operations required to approximate a program which we do not have access, save for some input-output examples. Emerging techniques in meta-learning and hyper-parameter optimization (e.g. differentiable architecture search [Liu et al., 2018]) add even further meta-programming layers to this stack, by searching over the space of directed graphs themselves.

Hardware manufacturers have developed a variety of custom silicon to train and run these programs rapidly. But unlike most programming, deep learning is a much simpler model of computation – so long as a computer can add and multiply, it has the ability to run a deep neural network. Yet due to the variety of hardware platforms which exist and the software churn associated with them, reproducing deep learning models can be painstakingly difficult on new hardware, even with the same source code and dependencies. Many graph formats, or *intermediate representations* (IRs) in compiler parlance, promise hardware portability but if developers are not careful, their models may not converge during training, or may produce different results on different hardware. Complicating the problem, IRs are produced by competing vendors, with competing chips and incompatible standards (e.g. MLIR [Lattner and Pienaar, 2019], ONNX [Bai et al., 2019], nGraph [Cyphers et al., 2018], Glow [Rotem et al., 2018], TVM [Chen et al., 2018] et al.) While some have tried to leverage existing

[2]cf. `https://help.github.com/en/articles/what-is-my-disk-quota`

compilers such as GHC [Elliott, 2018] or DLVM/LLVM [Wei et al., 2017], there are few signs of interoperability at the time of writing this thesis.

At the end of the day, researchers need to reproduce the work of other researchers, but the mental effort of re-implementing their abstractions can be tedious and detrimental towards scientific progress. Since it is necessary to reuse programs written by others, it would be convenient to have tools for reproducibility and incremental development. Fortunately, this is the same problem software developers have been attempting to solve for many years, through open source software. But source control management (SCM) alone is insufficient, since SCM tools are primarily intended for text. While text-based representations may be temporarily stable, as dependencies are periodically updated and rebuilt, important details about the original development environment can be misplaced. To reproduce a program in its entirety, we need a snapshot of all digital information available to the computer at the time of its execution, and ideally, the physical computer itself. Short of that, the minimal set of dependencies and a close replica is essential.

In order to mitigate the effects of software variability and assist the development of intelligent systems on heterogeneous platforms, we use a developer tool called Docker, part of a loosely-affiliated set of tools for build automation and developer operations which we shall refer to as *container infrastructure*. Docker allows developers to freeze a software system and its host environment, allowing developers (e.g. using a different environment) to quickly reproduce programs on another computer. Docker itself is a technical solution, but also encompasses a set of best-practices which are more procedural in nature. While Docker itself does not address the incompatibility of vendor standards and hardware drivers, it makes these variables explicit, and reduces the difficulty of reproducing software artifacts.

There is a second component to software reproducibility of intelligent systems, at the boundary of software and hardware: simulators. Today's simulators have become increasingly realistic, but most roboticists agree that simulation alone will never be enough to capture the full distribution of real world data. In this view, while simulation can be a useful tool for detecting errors, it cannot fully reproduce all the intricacies of the real world, and must never be used as a surrogate for training on real-world data. Others have suggested a middle road [Bousmalis et al., 2018], where judicious use of simulator training, alongside

domain adaption is a sufficiently rigorous environment for training intelligent systems. Regardless of which view prevails, our goal is to provide rapid feedback to developers, and to make the entire process from testing to deployment as reproducible as possible.

1.5.1. Case Study

All great software has a secret recipe: software gets better when its authors use the product. In the best case, the authors are the core users, ideally by choice, if not by necessity. When software engineers are using their own software on a regular basis – bumping into sharp corners and encountering edge cases firsthand – the product gets better. When there is an obviously missing feature, it gets implemented. When there is a bug, it gets fixed. It may not be easy to find users are so inclined, or to build software which is so useful, but there must be some overlap in order for good software to become great. Termed "dogfooding" [Harrison, 2006], this practice is an effective mechanism for building self-improving cybernetic systems and an important principle for open source software and safety-critical systems. Putting this principle into practice, we, as authors and primary users of these tools, validate their effectiveness by developing a robotics application within an IDE (Chapter 2), containing Kotlin∇ code (Chapter 3), tested using adversarial fuzz testing (Chapter 4), and which is built and maintained using the Docker stack (Chapter 5).

1.5.2. Iconography

Throughout this thesis, the following iconography is used to denote:


Shell commands intended for a laptop, or output derived thereof.

GrammarKit's `.bnf` parsing expression grammar (PEG) [3]


Either `Dockerfile` [4] or Docker Compose [5] syntax.

[3] GrammarKit usage notes: `https://github.com/JetBrains/Grammar-Kit/blob/master/HOWTO.md`

[4] Dockerfile reference: `https://docs.docker.com/engine/reference/builder/`

[5] Compose file reference: `https://docs.docker.com/compose/compose-file/`

Shell commands which should be run on a Raspberry Pi [6].

Duckietown Shell (`dts`) commands [7].

roslaunch `.launch` files [8].

Python source code [9].

Kotlin source code [10].

[6]Raspberry Pi: `https://www.raspberrypi.org/`

[7]Duckietown Shell: `https://github.com/duckietown/duckietown-shell-commands`

[8]ROS Launch XML: `http://wiki.ros.org/roslaunch/XML`

[9]Python documentation: `https://www.python.org/doc/`

[10]Kotlin documentation: `https://kotlinlang.org/docs/reference/`

Chapitre 2

Design: Programming tools for robotics

> "The hope is that, in not too many years, human brains and computing machines will be coupled together very tightly, and that the resulting partnership will think as no human brain has ever thought and process data in a way not approached by the information-handling machines we know today."
>
> –J.C.R. Licklider [1960], *Man-Computer Symbiosis*

In this chapter we will discuss the design and implementation of an integrated development environment (IDE) for building intelligent robotic software. Modern robots are increasingly driven by systems which learn and improve over time. Most researchers would agree that modern robotic systems have not yet achieved biologically competitive sensorimotor capabilities and most intelligent systems are not physically embodied. However, it is our view that any closed-loop control system that is not explicitly programmed to perform a specific task, but which learns it from experience is an *intelligent system*. Furthermore, any closed-loop system with physical motors is a *robotic system*. While research has demonstrated successful applications in both areas separately, it is widely believed the integration of intelligent systems in robotics will be tremendously fruitful when fully realized.

Hatchery is a tool designed to assist programmers writing robotics applications using the ROS middleware. At the time of its release, Hatchery was the first ROS plugin for the IntelliJ Platform [1], and today, is the most widely used with over 10,000 unique downloads. While the idea is simple, its prior absence and subsequent adoption suggests there is unmet demand for such tools in the development of intelligent software systems, particularly in domain-specific applications like robotics.

[1] An IDE platform for C/C++, Python and Android development

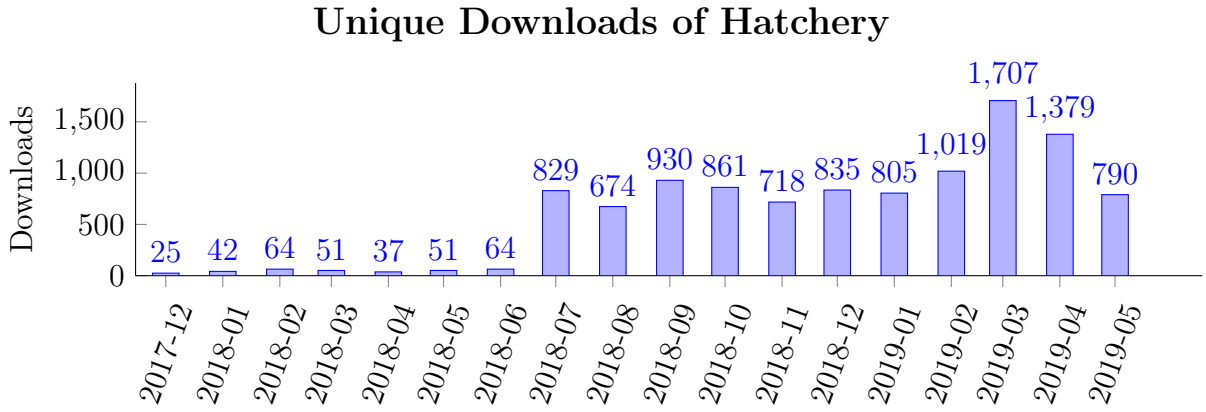


Fig. 2.1. Unique downloads of Hatchery between the time of its release and June 2019. `https://plugins.jetbrains.com/plugin/10290-hatchery`.

2.1. Introduction to the Robot Operating System

The Robot Operating System (ROS) [Quigley et al., 2009] is a popular middleware for robotics applications. At its core, ROS provides software infrastructure for distributed messaging, but also includes a set of community-developed libraries and graphical tools for building robotics applications. ROS is not an operating system (OS) in the traditional sense, but it does offer OS-like functionality such as shared memory and inter-process communication. Unlike pure message-oriented systems like DDS [Pardo-Castellote, 2003] and ZMQ [Hintjens, 2013], in addition to the communication infrastructure, ROS provides specific APIs for building decentralized robotic systems, particularly those which are capable of mobility. This includes standard libraries for serializing and deserializing geometric data, coordinate frames, maps, sensor messages, and imagery.

The ROS middleware provides several language front-ends for polyglot programming. According to one community census taken in 2018, 55% of all ROS applications on GitHub are written in C/C++, followed by Python with a 25% [Guenther, 2018] share. Source code for a typical ROS application contains a mixture of C/C++ and Python code, corresponding to the language preferences in the robotics and machine learning communities. Hatchery is capable of working with most common ROS client libraries, including rosjava for Java, rospy for Python, roscpp for C/C++, and several other language front ends.

A typical ROS project has several components, including the source code, configuration files, build infrastructure, compiled artifacts and the deployment environment. To build a

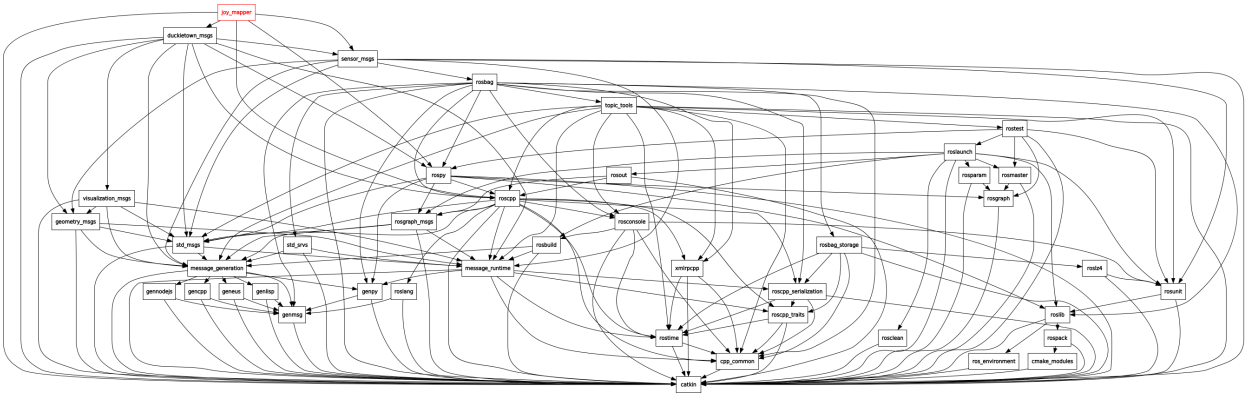


Fig. 2.2. A typical ROS application contains a large graph of dependencies.

simple ROS application, a number of steps are necessary. First, one must install the ROS system, which is only officially supported on Debian-based Linux distributions.[2] Assuming ROS has been installed to the default location, it can be sourced like so:

```
~$ source /opt/ros/<ROS DISTRO>/setup.[ba]sh
```

A minimal ROS application contains at least one *publisher* and *subscriber*, which pass messages over a shared communication channel. The publisher might be defined as follows:

./catkin_ws/src/pubsub/publisher.py

```
import rospy
from std_msgs.msg import String

i = 0
pub = rospy.Publisher("channel", String, queue_size=10)
rospy.init_node("publisher", anonymous=True)
rate = rospy.Rate(10)
while not rospy.is_shutdown():
    pub.publish("sent: %s" % i)
    rate.sleep()
    i += 1
```

As the publisher writes messages to `channel`, a subscriber can read them off like so:

[2]Detailed installation instructions may be found here: `http://wiki.ros.org/ROS/Installation`

./catkin_ws/src/pubsub/subscriber.py

```
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "received data %s", data.data)

rospy.init_node("subscriber", anonymous=True)
rospy.Subscriber("channel", String, callback)
rospy.spin()
```

All ROS packages have launch file, which contain a manifest of available nodes:

./catkin_ws/src/pubsub/pubsub.launch

```
<launch>
<node name="publisher" pkg="pubsub" type="publisher.py" output="screen"/>
<node name="subscriber" pkg="pubsub" type="subscriber.py" output="screen"/>
</launch>
```

To build and run the application, the following series of commands are required:

```
~$ cd catkin_ws && catkin_make
```

```
~$ roslaunch pubsub pubsub.launch
```

Rather than interacting with the command line, it would be convenient if there were a graphical tool which could perform all these tasks. Additionally, it would be helpful to detect if there were a navigable reference or typographical error in the launch file:

./catkin_ws/src/pubsub/pubsub.launch

```
<launch>
<node name="publisher" pkg="pubsub" type="pubsher.py" output="screen"/>
<node name="subscriber" pkg="pubsub" type="subscriber.py" output="screen"/>
</launch>
```

Notice how the typographical error is printed in red and the file reference is underlined in blue, indicating it can be selected. These are the kinds of UI features that IDEs provide.

2.2. Prerequisities

To simply run the tool, users should have the following software dependencies:

(1) MacOS or Debian Linux

(2) Robot Operating System (Electric Emys or later)

(3) Java SE (JRE 8+)

ROS users can use the following command to open a ROS project:

```
~$ git clone https://github.com/duckietown/hatchery && cd hatchery && \
   ./gradlew runIde [-Project="<ABSOLUTE_PATH_TO_ROS_PROJECT>"]
```

Duckietown users can simply use `dts`, the Duckietown Shell:

```
dt> hatchery
```

Hatchery can also be installed directly from inside the CLion or PyCharm IDEs, via:
File » Settings » Plugins » Marketplace » "Hatchery"

2.3. Plugin development

To build the IDE, some additional tools are helpful. First, is an IDE, and its source code. Assume that IDE_0 exists. In order to build a new IDE, IDE_1, we can load the source code from IDE_0 into IDE_0 and use IDE_0, to modify, compile and re-run the code, which becomes IDE_1, in which the process is repeated. However this approach has some disadvantages. First, most IDEs are already quite cumbersome to compile and run. As most auxiliary features are small by comparison, modern IDEs have adopted a modular design, which allows them to load specific packages (i.e. *plugins*) as needed. So most developers can skip the first step, and load their plugin inside IDE_0 directly. It is still convenient to have the platform source code for reference purposes, but in most cases this code is read-only.

For the purposes of this plugin, we use the IntelliJ Platform, a popular platform for cross-language development. By targeting an IDE platform with support for polyglot programming, we are able to focus on programming language-agnostic features in the ROS ecosystem, such as parsing and editing ROS-specific configuration files, build and run configuration and other common development tasks.

2.3.1. Refactoring

Refactoring is an essential feature in any IDE, and the essence of refactoring is renaming. Consider what must occur when a user wishes to rename a token in her program, such as the parameter named `data` on line #1 below:

```
def callback(data):                                                    1
    rospy.loginfo(rospy.get_caller_id() + "received data: %s", data.data)   2
```

If she were using the `vim` text editor, one solution would be to replace all textual occurrences of the string `data` within the file using `:%s/data/msg/g`, producing the following result:

```
def callback(msg):                                                     1
    rospy.loginfo(rospy.get_caller_id() + "received msg: %s", msg.msg)  2
```

There were four occurrences of the string `data`, only two of which were correctly renamed. Instead, the operation should only affect those identifiers which refer to the parameter:

```
def callback( data ):                                                  1
    rospy.loginfo(rospy.get_caller_id() + "received data: %s",  data .data)   2
```

Generally, we would like the ability to rename identifiers across files and languages. To do this, we need a richer understanding of code that transcends text. We need a parser.

2.3.2. The Parser

One of the most important and unappreciated components of an IDE is the parser. Unlike compilers, most IDEs do not use recursive descent or shift-reduce parsers, as they are not well-suited for real-time editing of source code. Edits are typically short, localized changes inside a large file, and are frequently invalid. As the IDE is expected to recover from local errors and provide responsive feedback to users while editing, re-parsing the entire document between edits would be expensive and unnecessary. In order to validate source code undergoing simultaneous modification, special consideration must be taken to ensure robust and responsive parsing.

Incremental methods such as Wagner [1997], Wagner and Graham [1997] incorporate caching and differential parsing to accelerate the analysis of programs under simultaneous modification. Fuzzy parsing techniques like those described in Koppler [1997] aim to increase the flexibility and robustness of parsing in the presence of local errors. Both of these techniques have played an important role in the development of language-aware programming tools, which must be able to provide rapid and specific feedback whilst the user is typing.

Modern parsers are seldom handwritten unless the language being parsed is very simple or raw performance is desired. Even for source code editing, where incremental parsing, type-checking and error-recovery is highly important, modern metacompilation toolkits such as ANTLR [Parr and Quong, 1995], or Xtext [Eysholdt and Behrens, 2010] cover a surprising number of common use-cases. Hatchery uses Grammar-Kit, a toolkit designed to assist users developing custom language plugins for the IntelliJ Platform. It uses a DFA-based lexer, JFlex [Klein et al., 2001], and a custom parser-generator loosely based on the parsing expression grammar (PEG) [Ford, 2004]. This grammar is translated to a program structure interface (PSI), the IntelliJ Platform's internal abstract syntax tree (AST). Here is an excerpt of a grammar for ROS `.msg` files:

BNF

```
rosInterfaceFile ::= (property|COMMENT)*
property ::= (TYPE KEY SEPARATOR VALUE)|(TYPE KEY) {
    pin=3
    recoverWhile="recover_property"
    mixin="edu.umontreal.hatchery.psi.impl.RosMsgNamedElementImpl"
    implements="edu.umontreal.hatchery.psi.RosMsgNamedElement"
    methods=[getType getKey getValue getName setName getNameIdentifier]
}
private recover_property ::= !(TYPE|KEY|SEPARATOR|COMMENT)
```

Grammar-Kit consumes this file, and generates source code for parsing ROS `.msg` files. Hatchery is also capable of parsing URDF, package and launch XML, and `.msg`/`.srv` files.

2.3.3. Running and debugging

ROS programs take a number of steps to compile and run. The follow steps are typical:

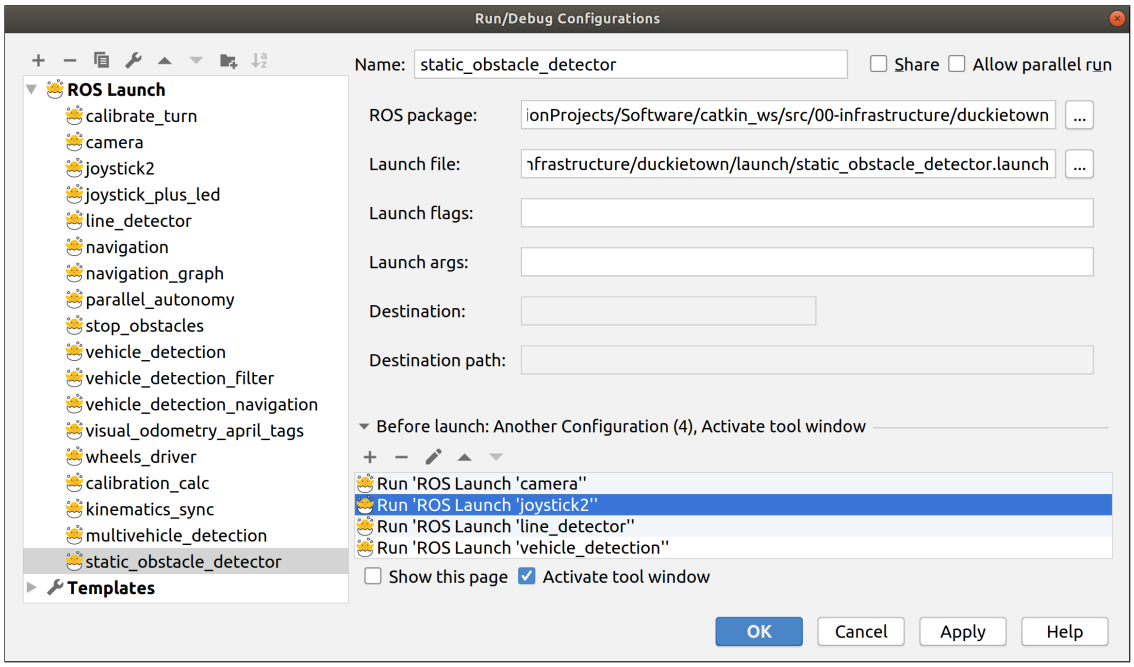


Fig. 2.3. ROS run configuration. Accessible via: Run » Edit Configurations » + » ROS Launch

```
~$ . /opt/ros/<DISTRO>/setup.[ba]sh &&
   cd <PROJECT>/catkin_ws &&
   catkin_make &&
   . devel/setup.sh &&
   [export ROS_MASTER_URI=<URI> &&]
   roslaunch [OPTIONS] src/.../<LAUNCH FILE> [ARGUMENTS]"
```

Hatchery provides assistance for these steps by providing a GUI for configuring, building and running ROS applications. Effectively it is a wrapper for the ROS command line interface.

2.3.4. User interface

An often overlooked, but important aspect of development tools is the user interface, being the primary mechanism through which developers edit source code. In the early days of modern computing, the only way of getting information into or out of a computer required punching holes in paper. Later, computers were equipped with technology to emit the same binary pattern as pixels, which could be used to display a small alphabet called ASCII. Later, through increased density and frequency, computers were able to render more sophisticated

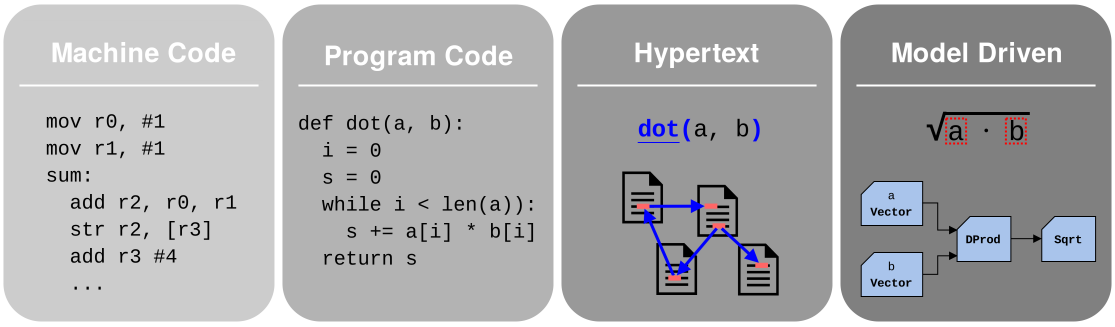


Fig. 2.4. The evolution of code. On the left are languages that force the user to adapt to the machine. To the right are increasingly flexible representations of source code.

shapes and animations. This evolution is the direct result of advances in graphics technology, but can also be seen as progress in program representation, where the explicit source code was simply a medium for communicating developer intent and machine interpretation.

ASCII is still the dominant medium for modern programming, although machines still use various forms of low level assembly code for execution. A great deal of software infrastructure is dedicated to translating between such representations via programming languages and compilers. While many developer tools provide a minimal command line interface (CLI), in addition to more sophisticated text editors for manipulating programs, these tools are still fairly restrictive. In the same way that early computer scientists probably did not fashion a new algorithm with a sequence of holes in mind, ASCII is also an indirect representation, albeit one slightly less contrived. Modern languages allow users to express their ideas in a notation is compact, familiar to use and easy to reason about its execution.

Modern programming tools are capable of representing code as a mixture of hypertext and graphical user interfaces. This provides a somewhat richer representation than plaintext and helps to capture programs' graph-based structure, but is rendered as ASCII with sparse visual cues. Some tools support larger character sets and font-based typographic ligatures, although the visual representation of source code remains highly linear.

More experimental UIs, such as model-driven engineering and *projectional editors* offer more flexible visual layouts, through macro-based text replacement. This uncoupling between the composition and representation of text raises many intriguing questions. With the proliferation of new abstractions and programming shorthands, what is the appropriate

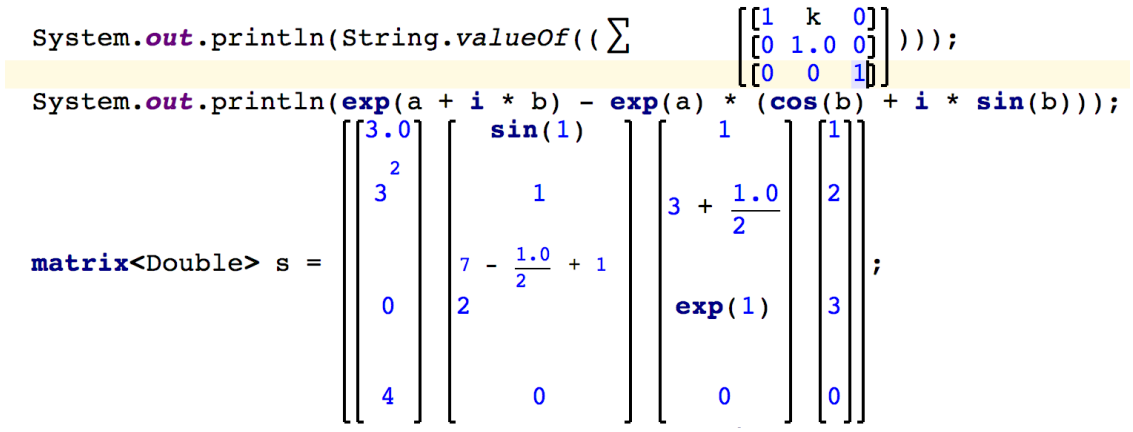


Fig. 2.5. Projectional editors such as MPS [Voelter and Solomatov, 2010, Pech et al., 2013] (shown above) are able to render plaintext source code in alternate formats for reading and manipulation.

level of notation required for a particular task? And who, or what, is the intended reader? These are important questions to ask when designing a developer tool.

In the case of Hatchery, we use a lightweight graphical user interface (GUI), most of which is provided by the underlying IDE platform. The plugin's primary job is to integrate smoothly with the IDE, which requires language- and framework-specific customizations.

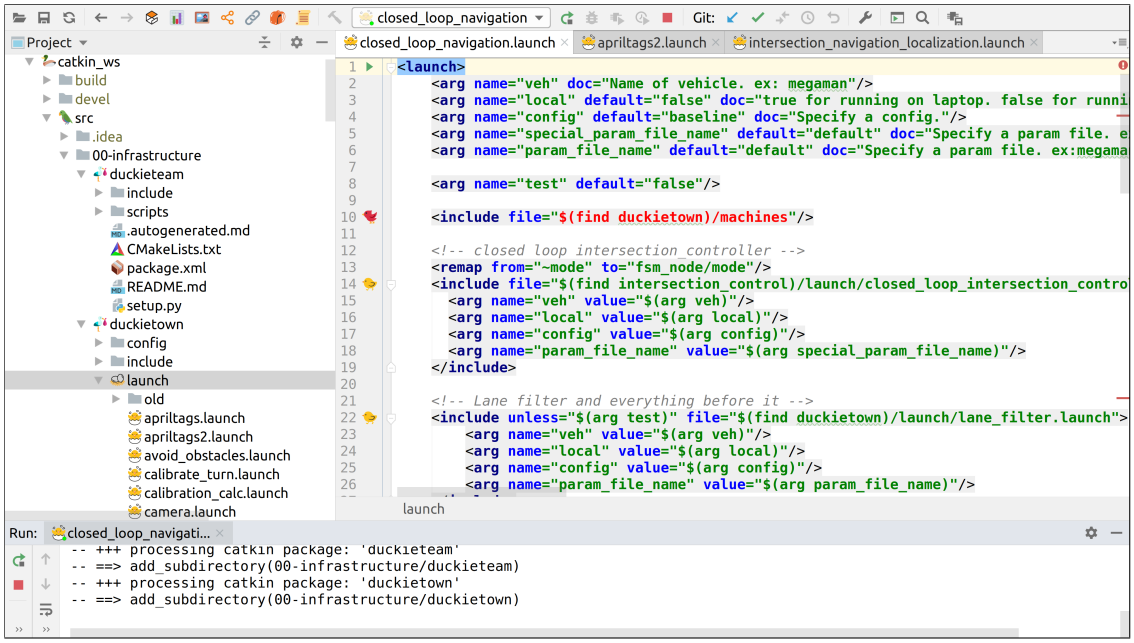


Fig. 2.6. Hatchery UI supports syntax highlighting, validation and project navigation.

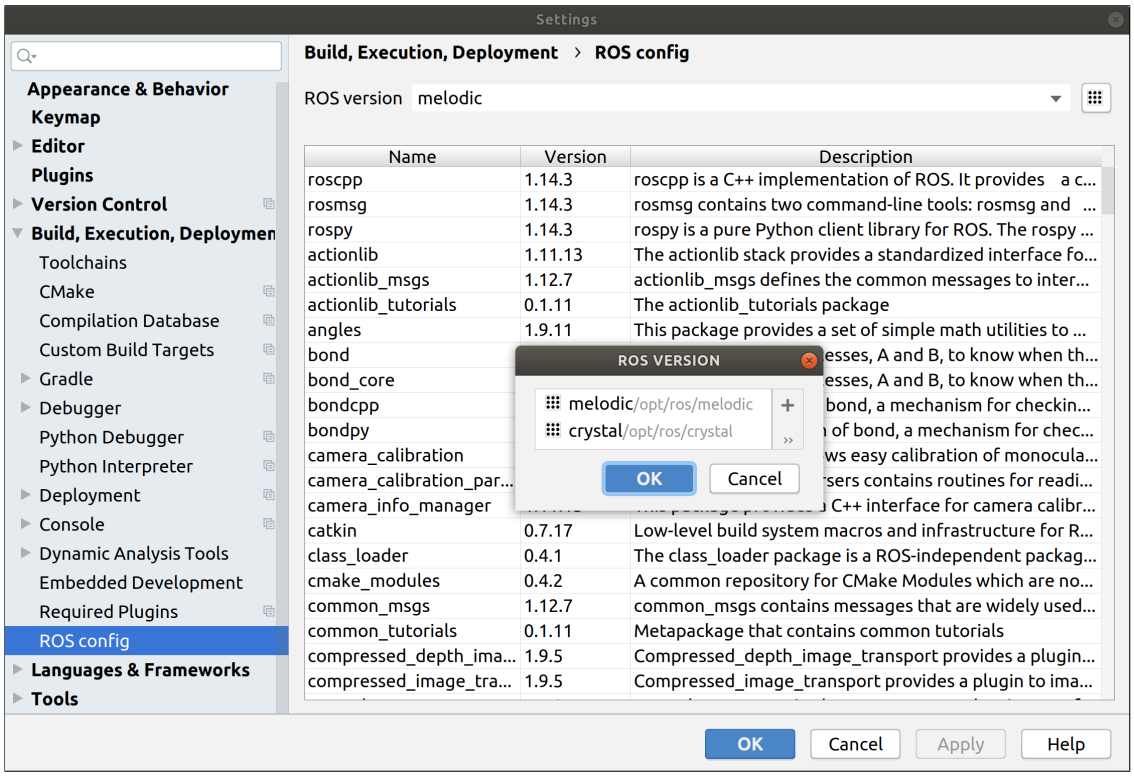


Fig. 2.7. Detection of local ROS packages. Accessible via: File › Settings › ROS config

2.4. Future work

Detecting and managing ROS installations.

Chapitre 3

Implementation: languages and compilers

> "The derivative, as this notion appears in the elementary differential calculus, is a familiar mathematical example of a function for which both [the domain and the range] consist of functions."
>
> –Alonzo Church [1941], *The Calculi of Lambda Conversion*

In this chapter, we will discuss the theory and implementation of a type safe domain-specific language for automatic differentiation (AD), which has produced a variety of applications in numerical optimization and machine learning. The key idea behind AD is fairly simple. A small set of primitive mathematical operations form the basis for all modern computers, and by composing these operations over the real numbers in an orderly fashion, one can compute any computable function. In machine learning, we are often given a computable function in the form of a program which does not work properly. We would like an algorithm for determining how to change the input slightly, so as to produce a more suitable output.

In 1964, such an algorithm was first conceived in Wengert [1964], whose method is known today as forward-mode AD. Not long after, a certain Richard Bellman reproduced Wengert's algorithm to numerically estimate the orbital dynamics of a two body system, recognizing its potential for, "the treatment of large systems of differential equations which might not otherwise be undertaken" [Bellman et al., 1965]. Around the same time, key details of the backpropogation algorithm first emerged [Dreyfus, 1990]. It was in Linnainmaa [1970] where the idea of calculating derivatives over computation graphs was first recorded. Linnaimaa's algorithm was particularly important for neural networks, and is today known as reverse-mode AD. But it was not until 2010 when standard software tools [Bergstra et al., 2010] for AD became widely available in machine learning. It is here where our journey begins.

3.1. Automatic differentiation

Given some input to a function, AD tells us how to change the input by a minimal amount, in order to maximally change the outputs. Suppose we are handed a function $P_k : \mathbb{R} \to \mathbb{R}$, composed of a series of nested functions, each with the same type:

$$P_k(x) = \begin{cases} p_0(x) = x & \text{if } k = 0 \\ (p_k \circ P_{k-1})(x) & \text{if } k > 0 \end{cases} \tag{3.1.1}$$

From the chain rule of calculus, we recall that:

$$\frac{dP}{dp_0} = \frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \cdots \frac{dp_1}{dp_0} = \prod_{i=1}^{k} \frac{dp_i}{dp_{i-1}} \tag{3.1.2}$$

Likewise, for a scalar function $Q(q_0, q_1, \ldots, q_n) : \mathbb{R}^m \to \mathbb{R}$, the gradient ∇Q tells us:

$$\nabla Q = \left(\frac{\partial Q}{\partial q_1}, \ldots, \frac{\partial Q}{\partial q_m} \right) \tag{3.1.3}$$

Occasionally, we may wish to compute the second-order partials for Q, i.e. the Hessian, $\mathbf{H}$:

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 Q}{\partial x_1^2} & \dfrac{\partial^2 Q}{\partial x_1 \, \partial x_2} & \cdots & \dfrac{\partial^2 Q}{\partial x_1 \, \partial x_m} \\ \dfrac{\partial^2 Q}{\partial x_2 \, \partial x_1} & \dfrac{\partial^2 Q}{\partial x_2^2} & \cdots & \dfrac{\partial^2 Q}{\partial x_2 \, \partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 Q}{\partial x_m \, \partial x_1} & \dfrac{\partial^2 Q}{\partial x_m \, \partial x_2} & \cdots & \dfrac{\partial^2 Q}{\partial x_m^2} \end{bmatrix} \tag{3.1.4}$$

More generally, for a vector function $\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^n$, the Jacobian $\mathbf{J}$ is defined as:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_n}{\partial x_1} & \cdots & \dfrac{\partial f_n}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_m \end{bmatrix} \tag{3.1.5}$$

For a vector function $\mathbf{P}_k(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^n$, the chain rule still applies:

$$\mathbf{J}_{\mathbf{P_k}} = \prod_{i=1}^{k} \mathbf{J}_{p_i} = \underbrace{\Big(\big((\mathbf{J}_{p_k} \mathbf{J}_{p_{k-1}}) \ldots \mathbf{J}_{p_2} \big) \mathbf{J}_{p_1} \Big)}_{\textit{“Reverse accumulation”}} = \underbrace{\Big(\mathbf{J}_{p_k} \big(\mathbf{J}_{p_{k-1}} \ldots (\mathbf{J}_{p_2} \mathbf{J}_{p_1}) \big) \Big)}_{\textit{“Forward accumulation”}} \tag{3.1.6}$$

For completeness, but rarely used in practice, is the second-order partials for vector functions:

$$\mathbf{H}(\mathbf{f}) = [\mathbf{H}(f_1), \mathbf{H}(f_2), \ldots, \mathbf{H}(f_n)] \tag{3.1.7}$$

We can use these tools to compute the direction to adjust the inputs of a computable function, in order to maximally change that function's output, i.e. the direction of steepest descent. Sometimes a function has the property that given an input a, no matter how a is changed, the output remains the same. We say that such functions have zero gradient for that input.

$$(\nabla F)(a) \approx \mathbf{0} \tag{3.1.8}$$

The cost of calculating the Hessian, $\mathbf{H}$ is approximately quadratic [Griewank, 1993] with respect to the number of independent variables under differentiation. If $\mathbf{H}(a)$ is tractable to compute and invertible, we could use the second-partial derivative test to determine that:

(1) If all eigenvalues of $\mathbf{H}(a)$ are positive, a is a local minimum
(2) If all eigenvalues of $\mathbf{H}(a)$ are negative, a is a local maximum
(3) If $\mathbf{H}$ contains a mixture of positive and negative eigenvalues, a is a *saddle point*

For some classes of computable functions, small changes to the input will produce a sudden large change in the output. We say that such functions are non-differentiable.

$$||\nabla F|| \approx \pm\infty \tag{3.1.9}$$

It is an open question whether non-differentiable functions exist in the real world [Buniy et al., 2005]. At the current physical (10nm) and temporal (10ns) scale of modern computing, there exist no such functions, but most modern computers are incapable of reporting the true value of their binary-valued functions. For all intents and purposes, programs implemented by most physical computers are discrete relations. Nevertheless, discrete programs are capable of approximating bounded functions on $\mathbb{R}^m$ to arbitrary precision given enough time and space. For most applications, a low precision (32-64 bit) approximation is sufficient.

There exists at the heart of machine learning a theorem that states a simple family of functions, which compute a weighted sum of a non-linear function $\varphi : \mathbb{R} \to \mathbb{R}$ composed with a linear function $\theta^\intercal x + b$, can approximate any bounded function on $\mathbb{R}^m$ to arbitrary precision. More precisely, the universal approximation theorem [Hornik et al., 1989] states that for all real-valued continuous functions $\mathbf{f} : C(\mathbb{I}_m)$, where $\mathbb{I}_m = [0,1]^m \to [0,1]$, there exists a function $\hat{\mathbf{f}} : \mathbb{R}^m \times \mathbb{R}^{n \times m}$, parameterized by $\Theta \in \mathbb{R}^{n \times m}$, taking an input $\mathbf{x} \in [0,1]^m$

and constants $n \in \mathbb{N}, \beta \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^n, \epsilon \in \mathbb{R}^+$ such that following statement holds:

$$\begin{aligned} \hat{\mathbf{f}}(\mathbf{x}; \Theta) &= \beta^\intercal \varphi_\odot \left(\Theta^\intercal \mathbf{x} + \mathbf{b}\right) \\ \forall \mathbf{x} \in \mathbb{I}_m,\ &|\hat{\mathbf{f}}(\mathbf{x}) - \mathbf{f}(\mathbf{x})| < \epsilon \end{aligned} \tag{3.1.10}$$

Where $\varphi_\odot$ indicates the nonlinear function φ applied elementwise to a vector. This theorem only tells us that Θ exists, but does not tell us how to find it nor does it place an upper bound on the constant n, somewhat limiting its practical applicability. But for reasons not yet fully understood, empirical results suggest it is possible to approximate many naturally-arising functions in a relatively short number of steps by composing several *layers* of $\Theta^\intercal \mathbf{x} + \mathbf{b}$ and φ in an alternating fashion, and updating each Θ using a procedure based on gradient descent. The resulting model might be expressed as follows[1],

$$\hat{\mathbf{P}}_k(\mathbf{x}; \boldsymbol{\Theta}) = \begin{cases} \hat{\mathbf{p}}_0(\Theta_0)(\mathbf{x}) & \text{if } k = 0 \\ \left(\hat{\mathbf{p}}_k(\Theta_k) \circ \hat{\mathbf{P}}_{k-1}(\boldsymbol{\Theta}_{[0,k-1]})\right)(\mathbf{x}) & \text{if } k > 0 \end{cases} \tag{3.1.11}$$

where $\boldsymbol{\Theta} = \{\Theta_0, \ldots, \Theta_k\}$ are free parameters and $\mathbf{x} \in \mathbb{R}^m$ is a single input. To approximate $\mathbf{P}(\mathbf{x})$, one must obtain $\mathbf{X} = \{\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(z)}\}, \mathbf{Y} = \{\mathbf{y}^{(0)} = \mathbf{P}(\mathbf{x}^{(0)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$ in as great and varied a quantity as possible and repeat the following procedure until $\boldsymbol{\Theta}$ converges:

$$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{1}{z} \nabla_{\boldsymbol{\Theta}} \sum_{i=0}^{z} \mathcal{L}\left(\hat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta}), \mathbf{y}^{(i)}\right) \tag{3.1.12}$$

In the general case, we can solve for the gradient using Equation 3.1.6. For most common $\mathcal{L}$, the complexity of this procedure is linear with z. As z can be quite large in practice, and since obtaining the exact gradient is not important, we use a stochastic variant by resampling a *minibatch* $\mathbf{X}', \mathbf{Y}'$ consisting of pairs $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$ for $i \sim \{0, ..., z\}$ without replacement on each update step. This is slightly noisier, but runs considerably more quickly.

3.2. Differentiable programming

The renaissance of modern deep learning is widely attributed to progress in three research areas: algorithms, data and hardware. Among algorithms, most research has focused on deep learning architectures and representation learning. Equally important, arguably, is the role that automatic differentiation (AD) has played in facilitating the implementation of these

[1]The notation below assumes some familiarity with currying and partial function application, in which $\hat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}^n \equiv \underbrace{\mathbb{R} \to \ldots \to \mathbb{R}}_{m} \to \mathbb{R}^n$. For further details, see Schönfinkel [1924], Curry [1967] et al.

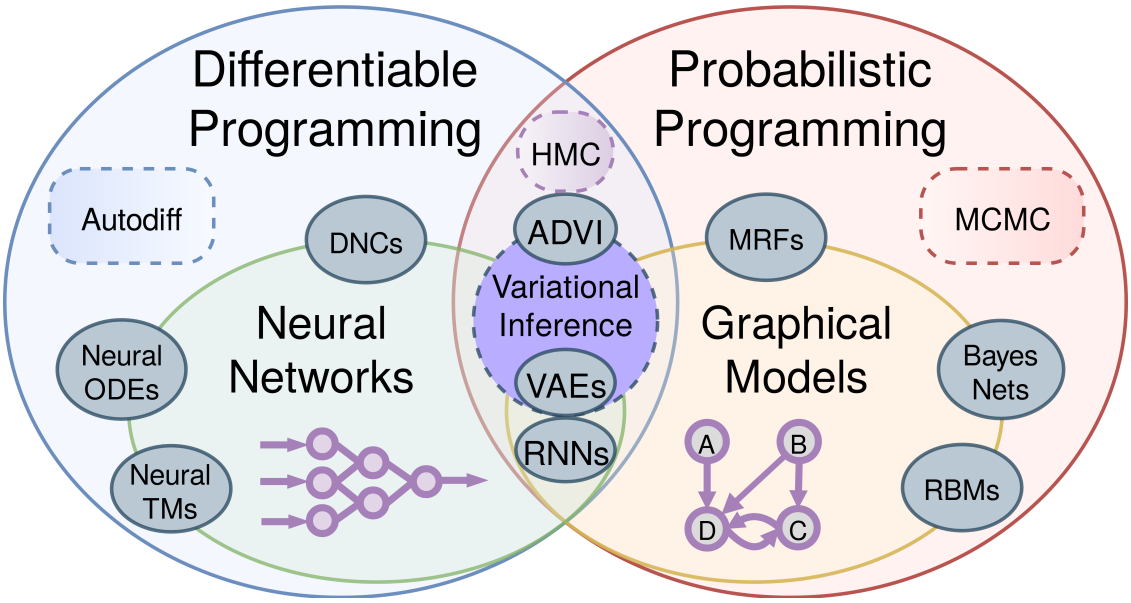


Fig. 3.1. *Differentiable programming* includes neural networks, but more broadly, arbitrary differentiable programs which use automatic differentiation and gradient-based optimization to approximate a loss function. *Probabilistic programming* [Carpenter et al., 2017, Gorinova et al.] is an emerging generalization of probabilistic graphical models, and uses various forms of Markov chain Monte Carlo (MCMC) and differentiable inference to approximate a probability density function.

ideas. Prior to the adoption of general-purpose AD libraries such as Theano, PyTorch and TensorFlow, gradients had to be derived manually. The widespread adoption of AD software simplified and accelerated the pace of gradient-based machine learning, allowing researchers to build deeper network architectures and new learning representations. Some of these ideas in turn, formed the basis for new methods in AD, which continues to be an active area of research in the programming language community.

A key aspect of the connectionist paradigm is gradient descent on a statistical loss function with respect to the free parameters of a neural network. For gradient descent to work, the representation must be differentiable almost everywhere. However many representations are non-differentiable in their natural domain. For example, the structure of language in its written form is not easily differentiable, as small changes to a word's symbolic representation can cause sudden changes to its semantic meaning. A key insight from representation learning is that many discrete data types can be mapped to a smoother latent space. For

example, if we represent words as a vector of real numbers, then it is possible to learn a mapping from the textual domain to the vector representation so that the semantic relations between words (as measured by their statistical co-occurrence in large language corpora) are geometrically preserved in vector space [Pennington et al., 2014] – words with similar meanings map to similar vectors. It so happens that many classes of discrete problems can be relaxed to continuous surrogates by learning such representations, or *embeddings* in an unsupervised, or semi-supervised manner.

Around the same time, the deep learning community realized that perhaps strict differentiability was not so important all along. It was shown in practice, that computers using low-precision arithmetic such as 8-bit floating point [Wang et al., 2018c] and integer [Jacob et al., 2018] quantization are able to train neural networks without sacrificing performance. Strong assumptions like Lipschitz-continuity and β-smoothness once thought to be indispensable for gradient-based learning could be relaxed, as long as the noise introduced by quantization was negligible compared to stochastic gradient methods. In hindsight, this should have been less surprising, since all digital computers use discrete representations anyway and were capable of training neural networks for nearly half a century. This suggests strict differentiability was not as important as having a good metric. As long as the loss surface permits metric learning, gradient descent is surprisingly resilient to quantization.

As deep learning solved problems across various domains, researchers observed that neural networks were part of a broader class of differentiable architectures that could be designed, implemented and analyzed in a manner not unlike computer programs. Hence the term *differentiable programming* (DP) was born. Today, DP has found a wide range of applications, from protein folding [AlQuraishi, 2018], to physics engines [de Avila Belbute-Peres et al., 2018, Degrave et al., 2016] and graphics rendering [Loper and Black, 2014] to meta-learning [Liu et al., 2018]. These domains have well-studied dynamics models, with parameters that can be tuned via gradient descent. Traditionally, handcrafted optimization algorithms were required to learn these parameters, but given a smooth metric, DP promises to do this for a broad class of models, more or less automatically. For discrete optimization however, DP is not sufficient. To automatically learn discrete relations without ad hoc embedding, additional tools, such as probabilistic programming, are likely needed. As seen in Figure 3.1, these two fields have developed many productive collaborations in recent years.

3.3. Static and dynamic languages

Most programs in machine learning and scientific computing are written in dynamic languages, such as Python. In contrast, most of the industry uses statically typed languages [Ray et al., 2017]. According to some studies, type related errors account for over 15% of software bugs [Gao et al., 2017]. While the causality on using static typing and fewer defects has not been conclusively established, dynamically typed languages are seldom used for building safety-critical systems, and the majority of robotics applications [Guenther, 2018] are written in statically typed languages.

Statically typed languages eliminate a broad class of runtime errors and allow users to reason more carefully about the behavior of programs without needing to execute them. In addition to stronger syntax validation for general-purpose programming, a well-designed library in a strongly typed language can eliminate specific errors related to API misuse that would otherwise require documentation and code samples to prevent, improving usability and reducing maintenance. Furthermore, strong type systems allow us to build more intelligent static analysis tools, which can provide relevant autocompletion, source code navigation, and earlier detection of runtime errors.

One frequent objection to using strongly typed languages is attributed to the additional burden of manual type annotation. While early type-safe languages like C/C++ and Java required programmers to exhaustively annotate function and variable declarations, with judicious use of type inference in modern languages like Kotlin, Scala, Rust et al., most type signatures may be safely omitted and easily recovered from the surrounding context. Type inference enables modern languages to offer the brevity of dynamically typed languages with the safety of static type checking.

3.4. Imperative and functional languages

Most programs written today are written in the imperative style, due the prevalence of the Turing Machine and von Neumann architecture [Backus, 2007]. λ-calculus provides an equivalent[2] language for computing, which we argue, is a more appropriate notation for expressing mathematical functions and computing their derivatives. In imperative programming the sole purpose of using a function is to pass it values, and there is no way to refer to

[2]In the sense that the Turing Machine and λ-calculus are both Turing Complete.

Imperative

```
fun dot(l1, l2) {
  if (len(l1) != len(l2))
    return error
  var sum = 0
  for(i in 0 to len(l1))
    sum += l1[i] * l2[i]
  return sum
}
```

Functional

```
fun dot(l1, l2) {
  return if (len(l1) != len(l2))
    error
  else if (len(l1) == 0) 0
  else
    head(l1) * head(l2) +
    dot(tail(l1), tail(l2))
}
```

Fig. 3.2. Two equivalent programs, both implementing the function $f(l_1, l_2) = l_1 \cdot l_2$.

a function directly. More troubling in the case of AD, is imperative programs have mutable state, which requires taking extra precautions when computing their derivatives.

The mathematical notion of function composition is a first-class citizen in functional programming. Just as we do in calculus, to take the derivative of a program composed with another program, simply apply the chain rule (cf. section 3.1). Since there is no mutable state in FP, no exotic data structures or compiler tricks are required.

For example, consider the vector function $f(l_1, l_2) = l_1 \cdot l_2$, seen in Figure 3.2. Imperative programs, by allowing mutation, are destroying intermediate information. In order to recover the computation graph for reverse mode AD, we either need to override the assignment operator, or use a tape to store the intermediate values. In pure functional programming, mutable variables do not exist, which makes our lives much easier.

Functional programming allows Kotlin∇ to use the same abstraction for representing mathematical functions and programming functions. All functions in Kotlin∇ are pure functions, composed of expressions which form a data-flow graph (DFG). An expression is simply a `Function`, which is only evaluated when invoked with numerical values, e.g. `z(0, 0)`. In this way, Kotlin∇ is similar to other compiled graph-based frameworks like TensorFlow and Theano.

3.5. Kotlin

When programming in a statically typed language, a common question one might ask the compiler is, "Given a value, `x`, can `x` be assigned to a variable of type `Y`?" (e.g. type checking

`x instanceof Y`) In Java, this question turns out to be unsound [Amin and Tate, 2016] and undecidable [Grigore, 2017] in the general case. It is possible to construct a Java program in which the answer is "yes" regardless of `Y`, or for which the answer cannot be determined in a finite amount of time. Undecidability is not necessarily a showstopper, but the unsoundness of Java's is more critical and unclear how to fix, even if it rarely occurs in practice.

Kotlin is a statically typed language that is well-suited for building cross-platform applications, with compiler support for JVM, JavaScript and native targets. Unlike most programming languages, the language was designed with IDE support from the outset, and owes much of its popularity to its programming ergonomics. Kotlin's type system [Tate, 2013] is strictly less expressive, but fully interoperable with Java's. It is unknown whether the the same issues which affect Java's type system are present in Kotlin's, but interoperability with Java has spurred its broad adoption and remains a key usability feature of the language.

In this work, we make use of several language features unique to Kotlin, such as first-class functions (section 3.12), extension functions (section 3.16), operator overloading (section 3.11), and algebraic data types (section 3.14). Furthermore, we make heavy use of Kotlin's DSL support for shape-safe array programming. Together, these language features provide a concise, flexible and type-safe platform for mathematical programming.

3.6. Kotlin∇

Prior work has shown it is possible to encode a deterministic context-free grammar as a *fluent interface* [Gil and Levy, 2016] in Java. This result was strengthened to prove Java's type system is Turing complete [Grigore, 2017]. As a practical consequence, we can use the same technique to perform shape-safe automatic differentiation (AD) in Java, using type-level programming. A similar technique is feasible in any language with generic types.

Differentiable programming has a rich history among dynamic languages like Python, Lua and JavaScript, with early implementations including projects like Theano [Al-Rfou et al., 2016], Torch [Collobert et al., 2002], and TensorFlow [Abadi et al., 2016]. Similar ideas have been implemented in statically typed, functional languages, such as Haskell's Stalin∇ [Pearlmutter and Siskind, 2008b], DiffSharp in F# [Baydin et al., 2015b] and recently Swift [Lattner and Wei, 2018]. However, the majority of existing automatic differentiation

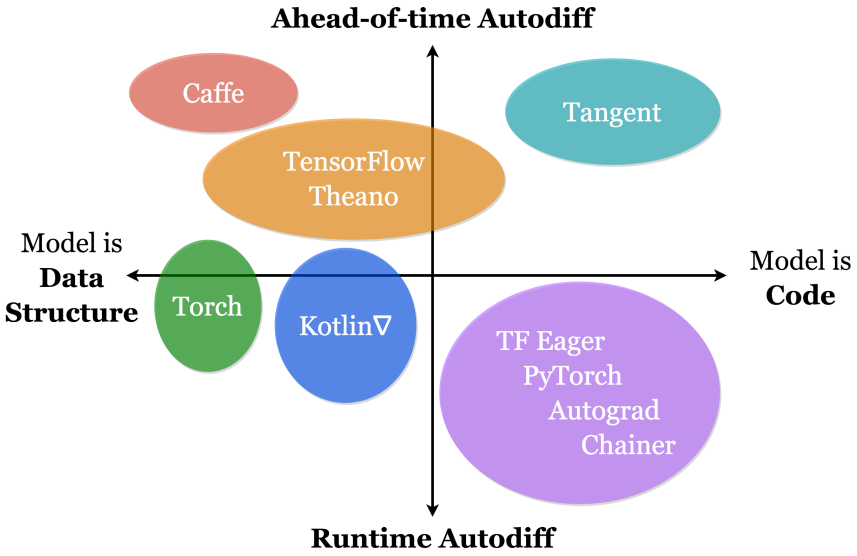


Fig. 3.3. Adapted from van Merriënboer et al. [2018]. Kotlin∇ models are data structures, constructed by an embedded DSL, eagerly optimized, and lazily evaluated at runtime.

(AD) libraries use a loosely-typed DSL, and few offer shape-safe tensor operations in a widely-used programming language.

Existing AD implementations for the JVM include Lantern [Wang et al., 2018b], Nexus [Chen, 2017] and DeepLearning.scala [Bo, 2018], however these are Scala-based and do not interoperate with other JVM languages. Kotlin∇ is fully interoperable with vanilla Java, enabling broader adoption in neighboring languages. To our knowledge, Kotlin has no prior AD implementation. However, the language has several useful features for implementing a native AD framework. Kotlin∇ primarily relies on the following language features:

- **Operator overloading and infix functions** allow a concise notation for defining arithmetic operations on tensor-algebraic structures, i.e. groups, rings and fields.
- **λ-functions** support functional programming, following Pearlmutter and Siskind [2008a,b], Siskind and Pearlmutter [2008], Elliott [2009, 2018], et al.
- **Extension functions** support extending classes with new fields and methods which can be exposed to external callers without requiring sub-classing or inheritance.

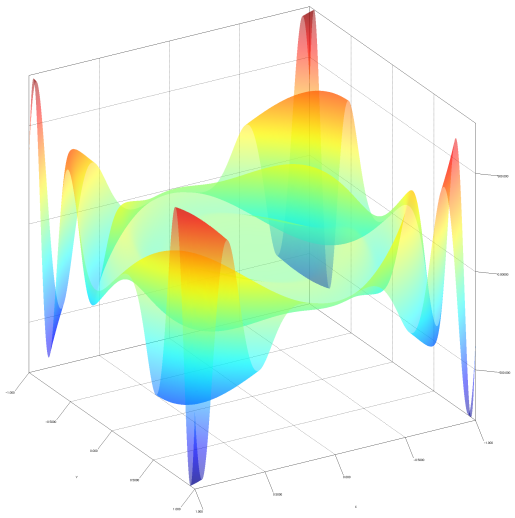
Kotlin∇ models are embedded domain-specific languages (eDSLs), which are essentially data structures masquerading as code. These structures may look and act like code, but are really functions for building an abstract syntax tree (AST). Often these ASTs represent

simple state machines, but are also used to embed a programming language. Popular examples include SQL/LINQ [Meijer et al., 2006], OptiML [Sujeeth et al., 2011] and other fluent interfaces [Fowler, 2005]. In a sufficiently expressive host language, one can implement any language as a library, without needing to write a lexer, parser, compiler or interpreter. And with proper typing, users will receive code completion and static analysis from their favorite developer tools. Functional languages are suitable host languages [Elliott et al., 2003, Rompf and Odersky, 2010], perhaps owing to the notion of code as data.

3.7. Usage

Kotlin∇ allows users to implement differentiable programs by composing expressions. Consider the following Kotlin∇ program with two inputs and one output:

```
with(DoublePrecision) { // Uses double precision numerics for evaluation
  val x = Var("x") // Declare immutable variables (these variables are
  val y = Var("y") // just symbolic constructs used for differentiation)
  val z = sin(10 * (x * x + pow(y, 2))) / 10 // Lazily evaluated
  val dz_dx = d(z) / d(x) // Supports Leibniz's notation
  val d2z_dxdy = d(dz_dx) / d(y) // Mixing higher order partials
  val d3z_d2xdy = grad(d2z_dxdy)[x] // Equivalent to d(f)/d(x)
  plot3D(d3z_d2xdy, -1.0, 1.0) // Plot in 3-space (-1 < x, y, z < 1)
}
```



```
val z = sin(10 * (x * x + pow(y, 2))) / 10 // Does not perform calculation
```

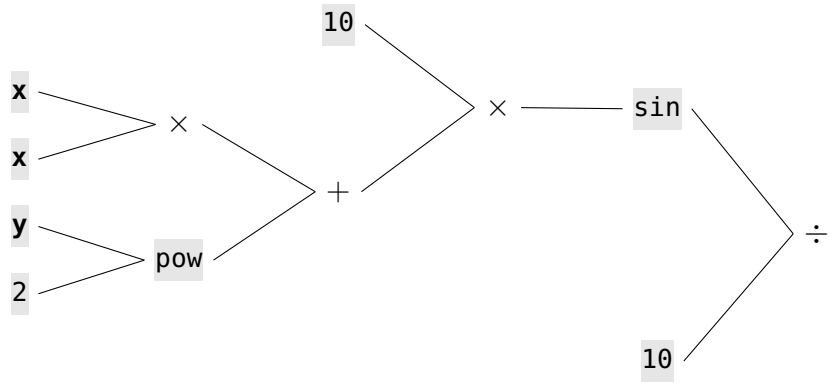


Fig. 3.4. Implicit DAG constructed by the original expression, seen above.

Above, we define a function with two variables and take a series of partial derivatives with respect to each variable. Expressions are lazily evaluated inside a numerical context, which may be imported on a per-file basis or lexically scoped for finer-grained control over the runtime behavior. The function is numerically evaluated on the interval $(-1, 1)$ in each dimension and rendered in 3-space.

An expression is simply a DSL for constructing `Functions`. No numerical calculations, save for constant folding and algebraic simplification, are performed at this stage. Functions are only evaluated once invoked with numerical values, e.g. `z(0, 0)`.

3.8. Type systems

Early work in type safe dimension analysis can be found in Kennedy [1994, 1996] showing how static types in array programming can be used to encode dimensionality and prevent common bugs related to dimension mismatch from arising. Kennedy is particularly interested in units of measurement, and was followed by Jay and Sekanina, Rittri, and Zenger [1997] who explore the application of dimension types for linear algebra. Later, Kiselyov [2005], Kiselyov et al. [2010] and Griffioen [2015], show how to encode numbers to infer array sizes in more complex ways. More recently, with the resurgence of interest in tensor algebra and array programming, Chen [2017] and Rink [2018] explore how shape safe tensor operations can be encoded in a sufficiently expressive type system.

The problem we would like to solve can be summarized as follows. Given two variables `x` and `y`, and operator `$`, how do we determine whether the expression `z = x $ y` is valid, and if so, what is the result type of `z`? For matrix multiplication, when $\mathsf{x} \in \mathbb{R}^{m \times n}$ and $\mathsf{y} \in \mathbb{R}^{n \times p}$, the expression is well-typed and we can infer $\mathsf{z} \in \mathbb{R}^{m \times p}$. More generally, we would like to infer the type of `z` for some operator $\otimes : (\mathbb{R}^{\mathbf{a}}, \mathbb{R}^{\mathbf{b}}) \to \mathbb{R}^{\mathbf{c}}$ where $\mathbf{a} \in \mathbb{Z}^q, \mathbf{b} \in \mathbb{Z}^r, \mathbf{c} \in \mathbb{Z}^s$ and $q, r, s \in \mathbb{Z}^{\geq}$. For most linear algebra operations such as matrix multiplication, $\mathcal{T}(\mathbf{a}, \mathbf{b}) \stackrel{?}{=} \mathbf{c}$ is computable in $\mathcal{O}(1)$ – we can simply check the inner dimensions for equivalence ($\mathbf{a}_2 \stackrel{?}{=} \mathbf{b}_1$) and use the outer dimensions for inference ($\mathbf{c} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{b}_2 \end{bmatrix}$).

Shape checking matrix operations is not always decidable. For arbitrary type functions $\mathcal{T}(\mathbf{a}, \mathbf{b})$, checking $\mathcal{T}(\mathbf{a}, \mathbf{b}) \stackrel{?}{=} \mathbf{c}$ requires a Turing machine, at which point it would be easier to implement at runtime rather than at the type level. If $\mathcal{T}$ is allowed to use the multiplication operator, as in the case of convolutional arithmetic [Dumoulin and Visin, 2016], shape inference becomes equivalent to Peano arithmetic, which is known to be undecidable [Gödel, 1931]. However shape checking convolutional arithmetic may be decidable.

First-class dependent types are useful for ensuring arbitrary shape safety (e.g. when concatenating and reshaping matrices [Xi and Pfenning, 1998]), but are unnecessary for simple equality checking, such as when multiplying two matrices.[3] Type checking ordinary matrix arithmetic is decidable in any type system loosely based on System $F_{<:}$. In practice, we can implement a shape-safe tensor algebra in any language with support for subtyping and parametric polymorphism, such as Java, Kotlin, C++, Rust or Typescript.

3.9. Shape safety

There are three broad strategies for handling shape errors in array programming:

(1) Conceal the error by implicitly reshaping or broadcasting arrays.
(2) Announce the error at runtime with a relevant message, e.g. `InvalidArgument`.
(3) Do not allow programs which can result in a shape error to compile.

[3]Less expressive type systems are still capable of performing arbitrary computation in the type checker. As specified, Java's type system is known to be Turing Complete [Grigore, 2017]. It may be possible to emulate a limited form of dependent types in Java by exploiting this property, although this may not be computationally tractable due to the practical limitations noted by Grigore.

Math†	Infix	Prefix	Postfix	Type
$A(B)$	`a(b)`			$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{\tau}) \to (\mathbb{R}^{\lambda} \to \mathbb{R}^{\pi})$
$A + B$	`a + b` `a.plus(b)`	`plus(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{\pi}) \to (\mathbb{R}^{?} \to \mathbb{R}^{\pi})$
$A - B$	`a - b` `a.minus(b)`	`minus(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{\pi}) \to (\mathbb{R}^{?} \to \mathbb{R}^{\pi})$
AB	`a * b` `a.times(b)`	`times(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times n}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{n \times p}) \to (\mathbb{R}^{?} \to \mathbb{R}^{m \times p})$
$\frac{A}{B}$ AB^{-1}	`a / b` `a.div(b)`	`div(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times n}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{p \times n}) \to (\mathbb{R}^{?} \to \mathbb{R}^{m \times p})$
$-A$ $+A$		`-a` `+a`	`a.unaryMinus()` `a.unaryPlus()`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{\pi}) \to (\mathbb{R}^{\tau} \to \mathbb{R}^{\pi})$
A+1 A-1	`a + 1` `a - 1`	`++a` `-a`	`a++, a.inc()` `a-, a.dec()`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}) \to (\mathbb{R}^{\tau} \to \mathbb{R}^{m \times m})$

sin(a)
cos(a)
tan(a)
		`sin(a)`
		`cos(a)`
		`tan(a)`
			`a.sin()`
			`a.cos()`
			`a.tan()`
				$(\mathtt{a} : \mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$

$\ln(A)$		`ln(a)` `log(a)`	`a.ln()` `a.log()`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}) \to (\mathbb{R}^{\tau} \to \mathbb{R}^{m \times m})$
$\log_b A$	`a.log(b)`	`log(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}^{m \times m}) \to (\mathbb{R}^{?} \to \mathbb{R})$
A^b	`a.pow(b)`	`pow(a, b)`		$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}, \mathtt{b} : \mathbb{R}^{\lambda} \to \mathbb{R}) \to (\mathbb{R}^{?} \to \mathbb{R}^{m \times m})$
$\sqrt{a}$ $\sqrt[3]{a}$	`a.pow(1.0/2)` `a.root(3)`	`a.pow(1.0/2)` `a.root(3)`	`a.sqrt()` `a.cbrt()`	$(\mathtt{a} : \mathbb{R}^{\tau} \to \mathbb{R}^{m \times m}) \to (\mathbb{R} \to \mathbb{R}^{m \times m})$
$\frac{da}{db}$ $a'(b)$	`a.diff(b)`	`grad(a)[b]`	`d(a) / d(b)`	$(\mathtt{a} : C(\mathbb{R}^m)^*, \mathtt{b} : \mathbb{R} \to \mathbb{R}) \to (\mathbb{R}^m \to \mathbb{R})$
∇a		`grad(a)`	`a.grad()`	$(\mathtt{a} : C(\mathbb{R}^m)) \to (\mathbb{R}^m \to \mathbb{R}^m)$

Tab. 3.1. Kotlin∇'s shape system specifies the output shape for matrix arithmetic.

Most array programming libraries such as NumPy [Van Der Walt et al., 2011] or TensorFlow [Abadi et al., 2016] use the first or second strategy. In Kotlin∇, we adopt the third, which allows an incremental type checker, such as those typically found in IDEs, to immediately detect when a matrix operation is invalid. Consider the following example:

```
val vecA = Vec(1.0, 2.0)      // Inferred type: Vec<Int, `2`>
val vecB = Vec(1.0, 2.0, 3.0) // Inferred type: Vec<Int, `3`>
val vecC = vecB + vecB
val vecD = vecA + vecB // Compile error: Expected Vec<2>, found Vec<3>
```

Attempting to sum two vectors whose shapes do not match will fail to compile.

```
val matA = Mat(`1`, `4`, 1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, `1`, `4`>
val matB = Mat(`4`, `1`, 1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, `4`, `1`>
val matC = matA * matB
val matD = matA * matC // Compile error: Expected Mat<4, *>, found Mat<1, 1>
```

Similarly, multiplying two matrices whose inner dimensions do not match will not compile.

```
val matA = Mat(`2`, `4`,
               1.0, 2.0, 3.0, 4.0,
               5.0, 6.0, 7.0, 8.0)
val matB = Mat(`4`, `2`,
               1.0, 2.0,
               3.0, 4.0,
               5.0, 6.0,
               7.0, 8.0)
val matC: Mat<Double, `2`, `2`> = a * b // Types are optional, but encouraged
val matD = Mat(`2`, `1`, 1.0, 2.0)
val matE = matC * matD
val matF = Mat(`3`, `1`, 1.0, 2.0, 3.0)
val matG = matE * matF // Compile error: Expected Mat<1, *>, found Mat<3, 1>
```

It is required to specify the parameter types in a method signature. Explicit return types are optional but encouraged for readability. If omitted, the type system can often infer them:

```
fun someMatFun(m: Mat<Double, `3`, `1`>): Mat<Double, `3`, `3`> = ...
fun someMatFun(m: Mat<Double, `2`, `2`>) = ...
```

Shape safety is currently supported up to rank-2 tensors, i.e. matrices. To check shapes in our type system, we enumerate a list of integer type literals as a chain of subtypes, so that $C <: C-1 <: C-2 <: \cdots <: 1 <: 0$, where C is the largest fixed-length dimension we wish to represent. Using this encoding, we are guaranteed linear growth in space and time for subtype checking. C can be specified by the user, who will need to regenerate the code.

```
interface Nat<T: `0`> { val i: Int }
// Integer literals have reified Int values should we need to compare them at runtime
sealed class `0`(open val i: Int = 0) { companion object: `0`(), Nat<`0`> }
sealed class `1`(override val i: Int = 1): `0`(i) { companion object: `1`(), Nat<`1`> }
```

```
sealed class `2`(override val i: Int = 2): `1`(i) { companion object: `2`(), Nat<`2`> }
sealed class `3`(override val i: Int = 3): `2`(i) { companion object: `3`(), Nat<`3`> }
//...Code for integer literals should be generated
sealed class `99`(override val i: Int = 99): `98`(i) { companion object: `99`(), Nat<`99`> }
```

Kotlin∇ supports shape-safe tensor operations by encoding tensor rank as a parameter of the operand type. Since integer literals are a chain of subtypes, we need only define tensor operations once using the highest literal, and can rely on Liskov substitution [Liskov, 1987] to preserve shape safety for all subtypes. Consider the rank-1 tensor (i.e. vector) case:

```
// <C: `1`> will accept 1 <= C <= 99 via Liskov substitution
infix operator fun <C: `1`, V: Vec<Float, C>> V.plus(v: V): Vec<Float, C> =
    Vec(length, contents.zip(v.contents).map { it.first + it.second })
```

The operator '+' can now be used like so. Incompatible operands will cause a type error:

```
// Type-checked vector addition with shape inference
val Y = Vec(`2`, listOf(1, 2)) + Vec(`2`, listOf(3, 4)) // Y: Vec<Float, `2`>
val X = Vec(‘2‘, listOf(1, 2)) + Vec(‘5‘, listOf(3, 4, 1, 2, 4)) // Y: Vec<???>
```

This technique can be easily extended to additional infix operators. We can also define a shape-safe vector initializer by overloading the invoke operator on a companion object:

```
open class Vec<E, Len: `1`> constructor(val len: Nat<Len>, val contents: List<E>) {
  companion object {
    operator fun <T> invoke(t: T): Vec<T, `1`> = Vec(`1`, listOf(t))
    operator fun <T> invoke(t0: T, t1: T): Vec<T, `2`> = Vec(`2`, listOf(t0, t1))
    operator fun <T> invoke(t0: T, t1: T, t2: T): Vec<T, `3`> = Vec(`3`, listOf(t0, t1, t2))
  }
}
```

Dynamic length construction is possible, although it may fail at runtime. For example:

```
val one = Vec(`3`, 1, 2, 3) + Vec(`3`, 1, 2, 3)   // Always runs safely
val add = Vec(`3`, 1, 2, 3) + Vec(`3`, listOf(t)) // May fail at runtime
val vec = Vec(1, 2, 3) // Inferred type: Vec<3>
val sum = Vec(‘2‘, 1, 2) + add // Compile error: Expected Vec<2>, found Vec<3>
```

Matrices and tensors have a similar syntax. For example, Kotlin∇ can infer the shape of matrix multiplication, and will not compile if the arguments' inner dimensions disagree:

```
val matL = Mat(`4`, `4`, // Inferred type: Mat<Int, `4`, `4`>
               1, 2, 3, 4,
               5, 6, 7, 8,
               9, 0, 0, 0,
               9, 0, 0, 0)
val matM = Mat(`4`, `3`, // Inferred type: Mat<Int, `4`, `3`>
               1, 1, 1,
               2, 2, 2,
               3, 3, 3,
               4, 4, 4)
val matN = matL * matM // Inferred type: Mat<Int, `4`, `3`>
val matO = matM * matM // Compile error: Expected Mat<3, *>, found Mat<4, 4>
```

A similar technique can be found in nalgebra [Crozet et al., 2019], a shape-checked linear algebra library for the Rust language which, like Kotlin∇, also uses synthetic type-level integers. This technique originates in Haskell, a language which supports more powerful forms of type-level computation, such as *type arithmetic* [Kiselyov, 2005]. Type arithmetic would make it easy to express array concatenation, convolutional arithmetic [Dumoulin and Visin, 2016] and other arithmetical operations, which are currently impossible to express in Kotlin∇, or highly impractical as each binary operator would require enumerating C^2 functions to accept every valid shape combination of vector operands.

3.10. Testing

Kotlin∇ claims to eliminate certain runtime errors, but how do we know the proposed implementation is not incorrect? One method for checking is called property-based testing (PBT) [Fink and Bishop, 1997] (cf. subsection 4.0.4), which is borrowed from the Haskell community and closely related to the notion of metamorphic testing [Chen et al., 1998] (cf. subsection 4.0.5). Notable implementations include QuickCheck [Claessen and Hughes, 2011], Hypothesis [MacIver, 2018] and KotlinTest [Samuel and Lopes, 2018], on which our test suite is based. PBT uses algebraic properties to verify the result of an operation by constructing semantically equivalent but syntactically distinct expressions, which should produce

the same answer. Kotlin∇ uses two such equivalences to validate its AD implementation:

(1) **Analytical differentiation**: manually differentiate selected functions and compare the numerical result of evaluating random chosen inputs from their domain with the numerical result obtained by evaluating AD on the same inputs.
(2) **Finite difference approximation**: sample the space of symbolic differentiable functions, comparing the numerical results suggested by the finite difference method and the equivalent AD result, up to a fixed-precision approximation.

For example, the following test checks whether the analytical derivative and the automatic derivative, when evaluated at random points, are equal to within numerical precision:

```
val x = Var("x")
val y = Var("y")
val z = y * (sin(x * y) - x)            // Function under test
val dz_dx = d(z) / d(x)                 // Automatic derivative
val manualDx = y * (cos(x * y) * y - 1) // Manual derivative

"dz/dx should be y * (cos(x * y) * y - 1)" {
    NumericalGenerator.assertAll { x0, y0 ->
        // Evaluate the results at a given seed
        val autoEval = dz_dx(x to x0, y to y0)
        val manualEval = manualDx(x to x0, y to y0)
        autoEval shouldBeApproximately manualEval // Fails iff eps < |adEval - manualEval|
    }
}
```

PBT will search the input space for two numerical values `x0` and `y0`, which violate the specification, then "shrink" them to discover pass-fail boundary values. We can construct a similar test using the finite difference method, e.g. $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$:

```
val dx = 1E-8
val x = Var("x")
val f = sin(x)
val df_dx = d(f) / d(x)
```

```
val fd_dx = (sin(x + dx) - sin(x)) / dx

"d(sin x)/dx should be equal to (sin(x + dx) - sin(x)) / dx" {
    NumericalGenerator.assertAll { x0 ->
        val autoEval = df_dx(x0)
        val fdEval = fd_dx(x0)
        autoEval shouldBeApproximately fdEval // Fails iff eps < |adEval - fdEval|
    }
}
```

There are many other ways to independently verify the numerical gradient, such as dual numbers or the complex step derivative. Another method to validate Kotlin∇'s implementation would be to compare the numerical output against the output of a well-known A.D. framework, such as TensorFlow's. In future work, we intend to conduct a more thorough comparison of numerical accuracy and performance.

3.11. Operator overloading

Operator overloading enables a concise notation for arithmetic on abstract types, where the types encode algebraic structures, e.g. `Group`, `Ring`, and `Field`. These abstractions are extensible to other mathematical structures, such as complex numbers and quaternions.

For example, we have an interface `Group` which overloads the operators + and ×:

```
interface Group<T: Group<T>> {
    operator fun plus(addend: T): T
    operator fun times(multiplicand: T): T
}
```

Here, we specify a recursive type bound using a method known as F-bounded polymorphism [Canning et al., 1989] to ensure that operations return the concrete value of the type variable `T`, rather than something more generic like `Group` (effectively, `T` is a `self` type). Imagine a class `Fun` which has implemented `Group`. It can be used as follows:

```
fun <T: Group<T>> cubed(t: T): T = t * t * t
fun <X: Fun<X>> twiceExprCubed(e: X): X = cubed(e) + cubed(e)
```

Like Python, Kotlin supports overloading a limited set of operators, which are evaluated using a fixed precedence. In the current version of Kotlin∇, operators do not perform any computation, they simply construct a directed acyclic graph representing the symbolic expression. Expressions are only evaluated when invoked as a function.

3.12. First class functions

With support for higher-order functions and lambdas, Kotlin treats functions as first class citizens. This allows us to represent mathematical functions and programming functions with the same underlying abstractions (i.e. typed FP). Following a number of recent papers in functional AD [Pearlmutter and Siskind, 2008a, Wang et al., 2018a], in Kotlin∇, all expressions are treated as functions. For example:

```
fun <T: Group<T>> makePoly(x: Var<T>, y: Var<T>) = x * y + y * y + x * x
val x: Var<DoubleReal> = Var()
val y: Var<DoubleReal> = Var()
val f = makePoly(x, y)
val z = f(1.0, 2.0) // Returns a value
println(z) // Prints: 7
```

Currently, it is possible to represent functions where all inputs and outputs share a single data type. It is possible to extend support for building functions with varying input/output types and enforce constraints on both, using covariant and contravariant type bounds. We leave this for further work.

3.13. Numeric Tower

Kotlin∇ uses a numeric tower [St-Amour et al., 2012]. First pioneered in the functional programming language, Scheme [Sperber et al., 2009], this strategy is also suited to object oriented programming [Niculescu, 2003, 2011] and widely used in other JVM libraries such as KMath [Nozik, 2019] and Apache Commons Math [Developers, 2012].

```
interface Group<X: Group<X>> {
    operator fun unaryMinus(): X
    operator fun plus(addend: X): X
    operator fun minus(subtrahend: X): X = this + -subtrahend
```

```
    operator fun times(multiplicand: X): X
}

interface Field<X: Field<X>>: Group<X> {
    val e: X
    val one: X
    val zero: X
    operator fun div(dividend: X): X = this * dividend.pow(-one)
    infix fun pow(exp: X): X
    fun ln(): X
}
```

The numeric tower also allows us to define common behavior such as subtraction and division on abstract algebraic structures, which are easily extended to concrete number systems. For example, to later define a field over the complex numbers or other algebras, [4] one must simply extend the numeric tower and override the default implementation. Most mathematical operations can be defined by composing a small set of primitive operations, which can later be differentiated in a generic fashion, rather than on an ad hoc basis.

3.14. Algebraic data types

Algebraic data types (ADTs) in the form of sealed classes (a.k.a. sum types) facilitate a limited form of pattern matching over a closed set of subclasses. When matching against subclasses of a sealed class, the compiler forces the author to provide an exhaustive control flow over all concrete subtypes of an abstract class. Consider the following classes:

```
class Const<T: Fun<T>>(val number: Number) : Fun<T>()
class Sum<T: Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>()
class Prod<T: Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>()
class Var<T: Fun<T>>: Fun<T>() { override val variables: Set<Var<X>> = setOf(this) }
class Zero<T: Fun<T>>: Const<T>(0.0)
class One<T: Fun<T>>: Const<T>(1.0)
```

When branching on the type of a sealed class, consumers must explicitly handle every case, since incomplete control flow will not compile rather than fail silently at runtime. Let us

[4] ex. In order to calculate derivatives in a quaternion neural network. [Isokawa et al., 2003]

now consider a simplified definition of `Fun`, a sealed class which defines the behavior of function invocation and differentiation, using a restricted form of pattern matching. It can be constructed with a set of `Var`s, and can be invoked with a numerical value:

```kotlin
sealed class Fun<X: Fun<X>>(open val variables: Set<Var<X>> = emptySet()): Group<Fun<X>> {
    constructor(vararg fns: Fun<X>): this(fns.flatMap { it.variables }.toSet())

    // Since the subclasses of Fun are a closed set, no `else -> ...` is required.
    operator fun invoke(map: Map<Var<X>, X>): Fun<X> = when (this) {
        is Const -> this
        is Var -> map.getOrElse(this) { this } // Partial application is permitted
        is Prod -> left(map) * right(map) // Smart casting implicitly casts after checking
        is Sum -> left(map) + right(map)
    }

    fun diff(variable: Var<X>): Fun<X> = when(this) {
       is Const -> Zero
       is Var -> if (variable == this) One else Zero
       // Product rule: d(u*v)/dx = du/dx * v + u * dv/dx
       is Prod -> left.diff(variable) * right + left * right.diff(variable)
       is Sum -> left.diff(variable) + right.diff(variable)
    }

    operator fun plus(addend: Fun<T>) = Sum(this, addend)

    operator fun times(multiplicand: Fun<T>) = Prod(this, multiplicand)
}
```

Kotlin's smart-casting implicitly downcasts the abstract type `Fun` as a subtype, such as `Sum` after performing an `is Sum` check. If `Fun` were not sealed, we would have needed to write `(this as Sum).left` instead to access its member, `left`. If the type cast was mistaken, a `ClassCastException` would need to be thrown, which smart casting also prevents.

3.15. Multiple Dispatch

In conjunction with ADTs, Kotlin∇ uses multiple dispatch to instantiate the most specific result type of an arithmetic operation based on the type of its component operands. Like

ADTs, multiple dispatch is not directly supported in the language, but it can be emulated using dynamic dispatch and overloading.

Building on section 3.14, imagine we would like to perform algebraic simplification, a useful trick for reducing expression swell [Laue, 2019] and improving numerical stability. We can use `when` to detect the type of a subexpression at runtime, then use *smart casting* to access its members, as though it were previously cast:

```
override fun times(multiplicand: Fun<X>): Fun<X> =
    when {
        this == zero -> this
        this == one -> multiplicand
        multiplicand == one -> this
        multiplicand == zero -> multiplicand
        this == multiplicand -> pow(two)
        // w/o smart cast: Const((this as Const).number * (multiplicand as Const).number)
        this is Const && multiplicand is Const -> Const(number * multiplicand.number)
        // Further simplification is possible using rules of replacement
        else -> Prod(this, multiplicand)
    }

val result = Const(2.0) * Sum(Var(2.0), Const(3.0))
//         = Sum(Prod(Const(2.0), Var(2.0)), Const(6.0))
```

Multiple dispatch allows us to put all related control flow on a single abstract class which is inherited by subclasses, simplifying readability, debugging and refactoring.

3.16. Extension Functions

Extension functions augment external classes with new fields and methods. By using context oriented programming [Hirschfeld et al., 2008], we can expose custom extensions (e.g. through `DoubleContext`) to consumers without requiring subclassing or inheritance.

```
object DoubleContext {
    operator fun Number.times(expr: Fun<Double>) = Const(toDouble()) * expr
}
```

Now, we can use the context to define another extension, `Fun.multiplyByTwo()`, which computes the product inside a `DoubleContext`, using the operator overload defined above:

```
fun Fun<Double>.multiplyByTwo() = with(DoubleContext) { 2 * this }
```

Extensions can also be defined in another file or context and imported on demand. This approach was borrowed from KMath [Nozik, 2019], another mathematical library for Kotlin.

3.17. Automatic, Symbolic Differentiation

It has long been claimed by the AD literature that automatic differentiation is not symbolic differentiation [Baydin et al., 2015a]. Many, including the author of this thesis, has suspected this claim to be misleading. Recently, the claim has been questioned [Wang et al., 2018b] and refuted [Laue, 2019]. While it may be true that certain implementations of automatic differentiation interleave numerical and symbolic differentiation during a program's execution, interleaving differentiation and numerical execution is certainly not a prerequisite for a differentiation library to be considered *automatic*, particularly when popular AD implementations, such as Theano [Al-Rfou et al., 2016], have chosen the symbolic route. It is our view that symbolic differentiation is one type of automatic differentiation, in particular, one which affords more flexibility to perform global optimizations and rewriting procedures. These optimizations would otherwise be impossible if attempted at runtime, with only partial information about the graph.

3.18. Coroutines

Coroutines are a generalization of subroutines for non-preemptive multitasking, typically implemented using continuations. Continuations are a mechanism that allow functions to access and modify subsequent computation. In continuation passing style (CPS), every function, in addition to its usual arguments, takes a second function representing the subsequent computation to be performed. Rather than returning to its caller, the function invokes its continuation immediately prior to completion, and the process is restarted. If the continuation is empty, the program halts.

One form of continuation, known as shift-reset a.k.a. delimited continuations, are sufficient for implementing reverse mode AD with operator overloading alone (without any additional data structures) as described by *Shift/Reset the Penultimate Backpropagator* [Wang et al., 2018b] and later in *Backpropagation with Continuation Callbacks* [Wang et al., 2018a]. Delimited continuations can be implemented with Kotlin's Coroutines support.

3.19. Comparison

Inspired by Stalin∇, Autograd, DiffSharp, Myia, Nexus, Lantern, Tangent et al., Kotlin∇ attempts to port recent developments in automatic differentiation (AD) to the Kotlin language. It introduces a number of experimental ideas, including compile-time shape-safety, algebraic simplification and numerical stability checking through property-based testing. Prior work, including Unlike most existing automatic differentiation libraries, Kotlin∇ is a purely symbolic, graph-based solution that does not require any compiler augmentation or runtime reflection. As we have seen, it achieves this primarily through operator overloading, polymorphism, and pattern matching. The practical advantage of this approach is that it can be implemented as a simple library or *embedded domain-specific language*, thereby employing the host language's type system. Our approach is particularly well-suited to functional programming, and makes use of a number of functional programming concepts, including lambda expressions, higher order functions, partial application, and algebraic data types.

3.20. Future work

Allowing users to specify a matrix's structure in its type signature, (e.g. Singular, Symmetric, Orthogonal, Unitary, Hermitian, Toeplitz) would allows us to specialize derivation over such matrices (cf. section 2.8 of Petersen et al. for a detailed review of specific techniques for calculating derivatives of structured matrices).

Integration with a dedicated linear algebra backend such as ND4J [Team, 2016], Apache Commons Math [Developers, 2012], EJML [Abeles, 2010] or JBlas [Braun et al., 2011]. Look into Makwana and Krishnaswami [2018].

Explore the meaning of derivatives in other calculi, and incremental computation following Ehrhard and Regnier [2003], Chen et al. [2012], Cai et al. [2014], Kelly et al. [2016].

Framework	Language	Symbolic Differentiation	Automatic Differentiation	Functional Programming	Type Safe	Shape Safe	Differentiable Programming	Multiplatform
Kotlin∇	Kotlin	✓	✓	✓	✓	✓	✍	✍
DiffSharp	F#	✗	✓	✓	✓	✗	✓	✗
TensorFlow.FSharp	F#	✗	✓	✓	✓	✓	✓	✗
Myia	Python	✓	✓	✓	✓	✓	✓	✗
Deeplearning.scala	Scala	✗	✓	✓	✓	✗	✓	✗
Nexus	Scala	✗	✓	✓	✓	✓	✓	✗
Lantern	Scala	✗	✓	✓	✓	✗	✓	✗
Grenade	Haskell	✗	✓	✓	✓	✓	✗	✗
Eclipse DL4J	Java	✓	✓	✗	✓	✗	✗	✗
Halide	C++	✗	✓	✗	✓	✗	✓	✗
Stalin∇	Scheme	✗	✓	✓	✗	✗	✗	✗

Tab. 3.2. Comparison of AD libraries. The ✍ symbol indicates work in progress.

Chapitre 4

Testing and validation

> "If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere... then we had better be quite sure the purpose put into the machine is the purpose which we really desire."
>
> –Norbert Wiener [1960], *Some moral and technical consequences of automation*

Today's deep neural networks are surprisingly effective when compared with handcrafted algorithms, but have known weaknesses. Training neural networks which can robustly transfer to new domains where the training and test distributions are dissimilar continues to pose a significant challenge. Furthermore, such models are susceptible to failure when presented with carefully crafted inputs. However the same gradient-based optimization techniques used for training neural networks can also be exploited to probe their failure modes.

In software engineering, techniques for software testing have become increasingly more automated and general-purpose. Developers love to write code, but loath writing tests. Yet tests are essential to validate an implementation's correctness. Tests prevent regressive behavior and are a form of specification in which the developer communicates the intended result of running a program. Techniques in coverage-guided fuzzing enable developers to write fewer tests with higher coverage. This is possible by automated testing.

In this chapter, we will explore the relationship between testing in machine learning and software engineering. We will see how the notion of adversarial testing shares a curious resemblance to fuzz testing in software engineering. In particular, we show how probabilistic sampling and constrained optimization can be seen as an extension of property-based testing (PBT) for adversarial training of differentiable programs, and propose a PBT algorithm which incorporates features of probabilistic programming and gradient-based optimization.

4.0.1. Unit Testing

In traditional unit testing, most tests are written in the following manner:

```
fun <C, D> unitTest(subroutine: (C) -> D) {
    val input = C(...)
    val expectedOutput = D(...)
    val actualOutput = subroutine(input)
    assertEquals(expectedOutput, actualOutput)
}
```

When carefully applied, it can be quite effective at detecting bugs and documenting the author's belief of preconditions and postconditions. The trouble is, someone needs to write a bunch of test cases for it to work properly. Additionally, it only tests subprograms, and must be updated when the program changes. This has the unintended side effect of decreasing agility, discouraging refactoring, or discarding prior work when tests become obsolete.

4.0.2. Integration Testing

In integration testing, we are more concerned about the overall behavior of a program, rather than the specific behavior of its component subroutines. Consider the following example:

```
fun <T, Y> integrationTest(program: (T) -> Y, inputs: Set<T>) =
    inputs.forEach { input ->
        try {
            if (checkPostCondition(program(input)))
                assert("Postcondition failed on $input")
        } catch(exception: SomeException) {
            assert(exception)
        }
    }
```

With this strategy, there are fewer tests to write down, since we only care about end-to-end behavior. Integration testing simply checks a program for terminating exceptions and simple post conditions. For this reason, it is often too coarse-grained.

For simplicity, in the following sections, we will only consider examples of programs which are pure functions, i.e. which have no external state and produce no side effects.

4.0.3. Fuzz Testing

Fuzz testing is an automated software testing methodology which generates random inputs to test a given program. For example, consider the following test:

```
@Repeat(1000)
fun <reified T, Y> simpleFuzzTest(program: (T) -> Y, oracle: (T) -> Y) {
        val input = generateRandomInput(T::class)
        assertEquals(program(i), oracle(i))
}
```

The trouble is, we need an oracle, an often unreasonable assumption. This is known as the *test oracle problem*. Furthermore, even if we had an oracle, if the space of inputs is large, it can take a long time to find an input where they disagree. Since a single call to `program(i)` can be be quite expensive in practice, this method can also be quite inefficient.

4.0.4. Property-based Testing

Property-based testing (PBT) is a specific type of fuzz testing which addresses the test oracle problem by using properties. PBT has two phases, seaching and shrinking. Users specify a property over all outputs and the test fails if it a counterexample can be found:

```
@Repeat(1000)
fun <reified T, Y> propertyTest(program: (T) -> Y, hasProperty: (Y) -> Boolean) {
    val input = generateRandomInput(T::class)
    if (!hasProperty(program(input))) {
        val shrunken = shrink(input, program, property)
        assert("Minimal counterexample of $property: $shrunken")
    }
}
```

Roughly speaking, `shrink` attempts to simplify the counterexample. For example, given a `program: (Float) -> Float`, a naïve shrinker might be implemented as follows:

```
tailrec fun <T: Float> shrink(input: T, program: (T) -> T, hasProperty: (T) -> Boolean): T =
    if (hasProperty(program(input - input / 2.0f)) return intput
    else shrink(input - input / 2.0f, program, hasProperty)
```

The trouble is, finding the right properties to test can be highly sensitive, and requires a lot of effort and domain-specific expertise. In addition, the user must specify a custom shrinker, which is unclear how to implement efficiently. What if there were a better way?

4.0.5. Metamorphic testing

It is often the case we would like to test the behavior of a program without providing an exhaustive specification. Many naturally-occurring generative processes exhibit a kind of local invariance – small changes to the representation do not drastically change the label. We can exploit this property to design general-purpose fuzzing methods given a small set of inputs and outputs. Metamorphic testing (MT) is a property testing methodology which addresses the test oracle problem and the challenge of cheaply discovering bugs in the low-data regime. It has been successfully applied in testing driverless cars [Zhou and Sun, 2019, Pei et al., 2017, Tian et al., 2018] and other stateful deep learning systems [Du et al., 2018].

First, let us consider the following concrete example, from Tian et al. [2018]: suppose we have implemented a program which takes an image from a vehicle while driving, and predicts the simultaneous steering angle of the vehicle. Given a single image and the corresponding ground-truth steering angle from an oracle (e.g. a human driver or simulator), our program should preserve invariance under various image transformations, such as limited illumination changes, linear transformations or additive noise below a certain threshold. Intuitively, the steering angle should remain approximately constant, regardless of any single transformation or sequence of transformations on the original image which satisfy our chosen criteria. If not, this is a strong indication our program is not sufficiently robust and may not respond well to the sort of variability it may encounter in an operational setting.

Metamorphic testing can be expressed as follows: Given an oracle $\mathbf{P} : \mathcal{I} \to \mathcal{O}$, and a set of inputs $\mathbf{X} = \{\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(z)}\}$ and outputs $\mathbf{Y} = \{\mathbf{y}^{(0)} = \mathbf{P}(\mathbf{x}^{(0)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$, a metamorphic relation (MR) is a relation $\mathcal{R} \subset \mathcal{I}^z \times \mathcal{O}^z$ where $z \geq 2$. In the simplest case, an MR is an equivalence relation, e.g.: $\langle \mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}' \rangle \in \mathcal{R} \Leftrightarrow \mathbf{x} \sim_{\mathcal{R}} \mathbf{x}' \Leftrightarrow \mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x}')$.

Suppose our MR is $\forall \varphi \in \mathcal{I} : ||\varphi|| \leq \varepsilon, \mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x}' = \mathbf{x} + \varphi) \approx \mathbf{y}$. Given a program $\hat{\mathbf{P}}$ and a set of inputs $\mathbf{X}$ and outputs $\mathbf{Y}$ from our oracle, the MR suggests an $\mathbf{X}', |\mathbf{X}| \ll |\mathbf{X}'|$ on which to test $\hat{\mathbf{P}}$, without requiring corresponding outputs from $\mathbf{P}$. If we can show that $\exists \mathbf{x}' \in \mathbf{X}' \mid \hat{\mathbf{P}}(\mathbf{x}') \not\approx \mathbf{P}(\mathbf{x})$, then we can infer at least one of the following:

(1) $\langle \mathbf{x}, \mathbf{P}(\mathbf{x}), \mathbf{x}', \mathbf{P}(\mathbf{x}') \rangle \notin \mathcal{R}$, i.e. our assumptions were invalid

(2) $\hat{\mathbf{P}}(\mathbf{x}') \not\approx \mathbf{P}(\mathbf{x}')$, i.e. the program under test is unsound

In either case, we have obtained useful information. If our assumptions were incorrect, we should adjust them to fix the problem. Otherwise, we have obtained a counterexample, and should update the program's implementation accordingly. Both of these are useful outcomes.

Consider the following example of an MT which uses an equivalence-based MR:

```
@Repeat(value = 1000)
fun <I, O> metamorphicTest(program: (I) -> O, mr: (I, O, I, O) -> Boolean) {
    val (input: I, output: O) = drawSampleFromData()
    val tx: (I) -> I = genLegalTx(program, mr, input, output)
    val txInput: I = tx(input)
    val txOutput: O = program(txIn)
    if (!mr(input, output, txInput, txOutput))
        assert("<$input, $output> not related to <$txInput, $txOutput> by $mr ($tx)")
}
```

The trouble is, generating valid transformations is a non-trivial exercise. We could try to generate random transformations until we find one which meets our criteria:

```
fun <I, O> genLegalTx(program: (I) -> O, mr: (I, O, I, O) -> Boolean, input: I, output: O) {
    while (true) {
        val tx: (I) -> O = genRandomTx() // Samples a random transformation
        val txInput: I = tx(input)
        val txOutput: O = program(txIn)
        if (mr(input, output, txInput, txOutput)) return tx
    }
}
```

But this would be very inefficient and depending on the type of input and output, is not guaranteed to terminate. We could handcraft a transformation, but this requires extensive domain knowledge. Instead, we should seek a more principled way to mutate a given input in our dataset so as to discover invalid outputs, which does not consume a lot of computation or domain expertise.

4.0.6. Adversarial Testing

This leads us to adversarial testing. In the general case, we are given an input-output pair from an oracle and a program approximating the oracle, but not necessarily the oracle itself. Our goal is to find a small change to the input value, which when fed to our program, produces the largest change to its output, relative to the original output.

Imagine a function $\hat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}$, where we seek to change each of the components $g_0, ..., g_{m-1}$ by a fixed amount, so as to produce the largest output value $\hat{\mathbf{P}}(g'_1, ..., g'_{m-1})$ directly. Suppose that for each input parameter $g_0, g_1, \ldots, g_{m-1}$, we have one of three choices to make: either we can increase the value by c, decrease the value by c, or leave it unchanged. We are given no further information about $\hat{\mathbf{P}}$. Consider the naïve solution, in which we try every combination of variable perturbations and select the inputs corresponding to the greatest output value:

Algorithm 1 Brute Force Adversary

procedure BfAdversary($\hat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}, g_0 : \mathbb{R}, g_1 : \mathbb{R}, \ldots, g_{m-1} : \mathbb{R}$)
  if $m = 1$ **then** ▷ Evaluate $\hat{\mathbf{P}}$ and return the best variable perturbation
    return arg max $\{\hat{\mathbf{P}}(g_0 + c), \hat{\mathbf{P}}(g_0 - c), \hat{\mathbf{P}}(g_0)\}$
  else ▷ Partially apply candidate perturbation and recurse
    return arg max $\{\hat{\mathbf{P}}(g_0 + c) \circ \text{BfAdversary}(\hat{\mathbf{P}}(g_0 + c), g_1, \ldots, g_{m-1}),$
      $\hat{\mathbf{P}}(g_0 - c) \circ \text{BfAdversary}(\hat{\mathbf{P}}(g_0 - c), g_1, \ldots, g_{m-1}),$
      $\hat{\mathbf{P}}(g_0) \circ \text{BfAdversary}(\hat{\mathbf{P}}(g_0), g_1, \ldots, g_{m-1})\}$
  end if
end procedure

As we can see, algorithm 1 is $\mathcal{O}(3^m)$ with respect to $\hat{\mathbf{P}}$ - not a very efficient search routine, especially if we want to consider a larger set of perturbances or need to repeat the procedure multiple times. Clearly, if we want to find the best direction in which to update $\mathbf{g}$, we need to be more careful about how we perform the search.

Even if we cannot compute a closed form derivative for $\hat{\mathbf{P}}$, if $\hat{\mathbf{P}}$ is differentiable almost everywhere, we can still use numerical differentiation to approximate pointwise values of its derivative. Consider algorithm 2, a refinement of algorithm 1 which uses the finite difference method to approximate the derivative with respect to each component of the input. This

tells us how to minimally change the input so as to produce the largest output in reach, without needing to exhaustively check every perturbation.

Algorithm 2 Finite Difference Adversary

procedure FDADVERSARY($\hat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}, g_0 : \mathbb{R}, g_1 : \mathbb{R}, \ldots, g_{m-1} : \mathbb{R}$)
  if $m = 1$ **then** ▷ Compute finite difference and perform gradient ascent (G.A.)
    return $g_0 + \frac{\hat{\mathbf{P}}(g_0) - \hat{\mathbf{P}}(g_0 + c)}{c}$
  else ▷ Apply single-step G.A. using componentwise finite difference and recurse
    return $g_0 + \frac{\hat{\mathbf{P}}(g_0, 0, \ldots) - \hat{\mathbf{P}}(g_0 + c, 0, \ldots)}{c}$, FDADVERSARY($\hat{\mathbf{P}}, g_1, \ldots, g_{m-1}$)
  end if
end procedure

We now have a procedure that is $\mathcal{O}(m)$ with respect to $\hat{\mathbf{P}}$, but must be recomputed for each input – we can still do better by assuming further structure on $\hat{\mathbf{P}}$. Furthermore, we have not yet incorporated any form of constraint on the input values. Perhaps we can combine the notion of metamorphic testing seen in subsection 4.0.5 with constrained optimization to accelerate the search for adversarial examples.

During backpropogation we perform gradient descent on a differentiable function with respect to its parameters for a specific set of inputs. In gradient-based adversarial testing, we perform gradient ascent on a loss function with respect to the inputs using a fixed parameter setting. Suppose we have a differentiable vector function $\hat{\mathbf{P}} : \mathbb{R}^m \to \mathbb{R}^n$, defined as follows:

$$\hat{\mathbf{P}}_k(\mathbf{x}; \boldsymbol{\Theta}) = \begin{cases} \hat{\mathbf{p}}_0(\Theta_0)(\mathbf{x}) & \text{if } k = 0 \\ \big(\hat{\mathbf{p}}_k(\Theta_k) \circ \hat{\mathbf{P}}_{k-1}(\boldsymbol{\Theta}_{[0,k-1]})\big)(\mathbf{x}) & \text{if } k > 0 \end{cases} \qquad \text{(Equation 3.1.11 revisited)}$$

In deep learning, given pairs $\mathbf{X} = \{\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(z)}\}, \mathbf{Y} = \{\mathbf{y}^{(0)} = \mathbf{P}(\mathbf{x}^{(0)}), \ldots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$ we want to find $\boldsymbol{\Theta}^* = \arg\min_{\boldsymbol{\Theta}} \mathcal{L}\big(\hat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta}), \mathbf{y}^{(i)}\big)$ which is typically achieved by performing stochastic gradient descent on the loss with respect to the model parameters:

$$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{1}{z} \nabla_{\boldsymbol{\Theta}} \sum_{i=0}^{z} \mathcal{L}\big(\hat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta}), \mathbf{y}^{(i)}\big) \qquad \text{(Equation 3.1.12 revisited)}$$

We can solve for the gradient by taking the product of the Jacobians (c.f. Equation 3.1.6), $\mathbf{J}_{\mathbf{p}_0} \ldots \mathbf{J}_{\mathbf{p}_k}$ with respect to $\boldsymbol{\Theta}$. In white box adversarial learning, we are given a fixed $\boldsymbol{\Theta}$[1] and

[1] In contrast with backpropogation, where the parameters $\boldsymbol{\Theta}$ are updated.

control the value of $\mathbf{x}$, so we can rewrite $\hat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \boldsymbol{\Theta})$ instead as $\hat{\mathbf{P}}(\mathbf{x})$, and take the gradient directly with respect to $\mathbf{x}$. Our objective is to find the "worst" $\mathbf{x}$ within a small distance of any $\mathbf{x}^{(i)}$, i.e. where $\mathbf{P}(\mathbf{x})$ least resembles $\hat{\mathbf{P}}(\mathbf{x})$. More concretely, this can be expressed as,

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \mathcal{L}\big(\hat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}\big) \text{ subject to } CS = \{\mathbf{x} \in \mathbb{R}^m \text{ s.t. } ||\mathbf{x}^{(i)} - \mathbf{x}|| < \epsilon\} \tag{4.0.1}$$

To do so, we initialize $\mathbf{x} \sim U[CS]$ and perform projected gradient ascent on the loss:

$$\mathbf{x} \leftarrow \boldsymbol{\Phi}_{CS}\Big(\mathbf{x} + \alpha \nabla_{\mathbf{x}} \mathcal{L}\big(\hat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}\big)\Big), \text{ where } \boldsymbol{\Phi}_{CS}(\phi') = \arg\min_{\phi \in CS} \frac{1}{2}||\phi - \phi'||_2^2 \tag{4.0.2}$$

Henceforth we shall refer to $\mathcal{L}\big(\hat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}\big)$ as $\mathcal{L}(\mathbf{x})$. Imagine a single test $\mathbf{T} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{B}$:

$$\mathbf{T}(\hat{\mathbf{P}}, \mathbf{x}) = \mathcal{L}(\mathbf{x}) < C \tag{4.0.3}$$

Where $C \in \mathbb{R}$. How can we find a set of inputs that break the test under a fixed computational budget (i.e. constant number of program evaluations)? In other words:

$$\{D_{\mathbf{T}} : \mathbf{x} \in CS \mid \hat{\mathbf{P}}(\mathbf{x}) \implies \neg\mathbf{T}\}, maximize|D_{\mathbf{T}}| \tag{4.0.4}$$

If the adversary has zero knowledge about the program's implementation or the data distribution, $D_{\hat{\mathbf{P}}}$ when searching for adversarial inputs, she can do no better than random search [Wolpert et al., 1997]. However, if she has information about the input distribution, she could re-parameterize that distribution to incorporate her knowledge. By assuming the program has previously been tested on common inputs, she might use the inverse distribution $x \sim \frac{1}{D_{\hat{\mathbf{P}}}}$ to select inputs that are less frequent in hopes of finding one not previously considered. In addition, if the tester knew some details of how $\hat{\mathbf{P}}$ were implemented, she could use classical fuzzing techniques to prioritize the search inputs for those more likely to break $\mathbf{T}$. We consider a extension of classical fuzzing techniques to differentiable functions on continuous random variables.

One strategy, independent of how candidate inputs are selected, is to use some form of gradient-based optimization in the search procedure. If the tester does not have access to the function directly, she can still use zeroth order optimization techniques to approximate the gradient. Given access to the program, she can compute the gradient of the program using automatic differentiation. The gradient of $\hat{\mathbf{P}}$'s loss with respect to $\mathbf{x}$ is $\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x})$.

Consider algorithm 3 for finding test failures. First, let us select a candidate input $\mathbf{x}_j$ according to sampling strategy $\mathcal{S}$ (e.g. uniform random, or a neural network which takes $\hat{\mathbf{P}}$

and $\mathbf{T}$ as input). If $\hat{\mathbf{P}}(\mathbf{x}^i)$ violates $\mathbf{T}$, we can append $\mathbf{x}^i$ to $D_{\mathbf{T}}$ and repeat. Otherwise, we will follow the gradient of $\mathcal{L}(\hat{\mathbf{P}}, \mathbf{x})$ with respect to $\mathbf{x}$ and repeat until test failure, gradient descent convergence, or a fixed number of steps C are reached before resampling $\mathbf{x}$ from the initial sampling strategy $\mathcal{S}_n$ to ensure each gradient descent trajectory will terminate before exhausting our budget.

We hypothesize that if $\hat{\mathbf{P}}$'s implementation were flawed and a counterexample to Equation 4.0.3 existed, as sample size increased, a subset of gradient descent trajectories would fail to converge, a portion would converge to local minima, and the remaining trajectories would discover inputs violating the program specification.

Algorithm 3 Probabilistic Adversary

procedure ProbAdversary($\mathcal{L} : \mathbb{R}^m \to \mathbb{R}$, $\mathcal{S}_n$, $\mathbf{T} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{B}$, budget: $\mathbb{Z}^+$)
    $D_{\mathbf{T}} \leftarrow \{\}, j \leftarrow 0$
    while $j \leq$ budget **do** ▷ Iterate until count exceeds our budget
        $\mathbf{x}_j \sim \mathcal{S}_m$ ▷ Sample from S
        if $\mathbf{T}\big(\mathbf{x}_j, \mathcal{L}(\mathbf{x}_j)\big)$ **then** ▷ Inside feasible set, perform gradient ascent
            $D_{\mathbf{T}} \leftarrow D_{\mathbf{T}}$ + DiffShrink$(-\mathcal{L}, \mathbf{x}_j, \mathbf{T})$
        else ▷ Outside feasible set, perform gradient descent
            $D_{\mathbf{T}} \leftarrow D_{\mathbf{T}}$ + DiffShrink$(\mathcal{L}, \mathbf{x}_j, \mathbf{T})$
        end if
        $j \leftarrow j + 1$
    end while
    return $D_{\mathbf{T}}$
end procedure

Algorithm 4 Differential Adversary

procedure DIFFADVERSARY($\mathcal{L} : \mathbb{R}^m \to \mathbb{R}$, $\mathbf{x}_0 : \mathbb{R}^m$, $\mathbf{T} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{B}$)
  $i \leftarrow 1$
  $t_0 \leftarrow \mathbf{T}\big(\mathbf{x}_0, \mathcal{L}(\mathbf{x}_0)\big)$ ▷ Store initial state to detect when test flips.
  while $i \leq I_{max}$ **and** $||\mathbf{x}_i - \mathbf{x}_{i-1}||_2^2 < \epsilon$ **do** ▷ While in budget and not converged.
    $\mathbf{x}_i \leftarrow \mathbf{\Phi}_{CS}\big(\mathbf{x}_{i-1} - \alpha \nabla_{\mathbf{x}_{i-1}} \mathcal{L}(\mathbf{x}_{i-1})\big)$ ▷ PGD step (cf. Equation 4.0.2)
    if $\mathbf{T}\big(\mathbf{x}_i, \mathcal{L}(\mathbf{x}_i)\big) \neq t_0$ **then** ▷ Boundary value was found.
      return $\{\mathbf{x}_{i-1}\}$ ▷ Return previous iterate.
    end if
    $i \leftarrow i + 1$
  end while
  if $\neg t_0$ **then** ▷ If initial test state was failure, return most recent iterate.
    return $\{\mathbf{x}_{i-1}\}$
  else ▷ Otherwise, there was no test failure and return the empty set.
    return $\varnothing$
  end if
end procedure

Another type of adversary is a generative adversary [Albuquerque et al., 2019].

Chapitre 5

Software Maintenance and Reproducibility

> "As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products."
>
> –Edsger W. Dijkstra [1972], *The humble programmer*

In this chapter, we will discuss the challenges of software reproducibility and how best practices in software engineering such as continuous integration and containers can help researchers mitigate the variability associated with building and running software. Our work focuses on computational determinism, and does not consider the variability of distributional shift or related statistical notions of variation.

In order to address the issue of software reproducibility, we constructed a set of tools and development workflows that draw on best practices in software engineering. These tools are primarily built around containerization, a widely adopted virtualization technology in the software industry. In order to lower the barrier of entry for developers to become productive contributors and minimize variability across hardware platforms, we provide a state-of-the-art container infrastructure based on Docker [Merkel, 2014], one popular container engine. Docker allows us to construct versioned deployment artifacts that represent the entire filesystem, and manages resource constraints via a sandboxed runtime environment.

5.1. Dependency management

One common source of variability in software development are software dependencies. For many years, developers struggled with dependency management before it was discovered

the dependency resolution problem was NP-complete [Abate et al., 2012]. If we assume no two versions of the same dependency can be installed simultaneously, then for a set of software packages which need to be installed, and dependencies required to install them, determining the latest version of the dependencies which satisfy all requirements is as hard as the hardest problems in NP. Informally, this problem is known as *dependency hell* and grows increasingly troublesome as software projects grow and introduce new dependencies.

Dependency hell does not just occur inside individual software projects, but across projects and development environments. Hundreds of package managers have been developed for various operating systems, programming languages, and development frameworks. Ubuntu has the Advanced Package Tool (`apt`), macOS has Homebrew (`brew`), Windows has Chocolatey (`choco`). Most programming language ecosystems have their own bespoke package managers; Conan for C/C++, Maven for Java, and Cabal for Haskell. Python has developed several overlapping solutions for package management, including pip, Anaconda, PyEnv, Virtualenv, and others. Some of these install system-wide packages, and others provide command line environments. Over the lifetime of a computer system, as packages are installed and removed it becomes difficult to keep track of changes and their effects.

The problem basically stems from the requirement that no two versions of the same dependency can be installed simultaneously. In addition, software installers tend to spray files across the file system, which can become corrupted and are difficult to completely remove. To address these issues, some notion of "checkpointing" is required, so that when new software is installed, any future changes can be traced and reverted. Backups would do the job, but are cumbersome to manage and are unsuitable for development purposes. Rather, it would be convenient if there were a tool which allowed applications to setup a private file system, install their dependencies, and avoid contaminating the host OS.

5.2. Operating systems and virtualization

With the growth of developer operations (devops) a number of solutions emerged for building and running generic software artifacts. Most universal are emulators, which effectively simulate a foreign processor architecture, and thereby any software which runs on it. Another solution was virtual machines (VMs), a form of isolated runtime environment which use a *hypervisor* to mediate access to hardware, but otherwise run software on physical

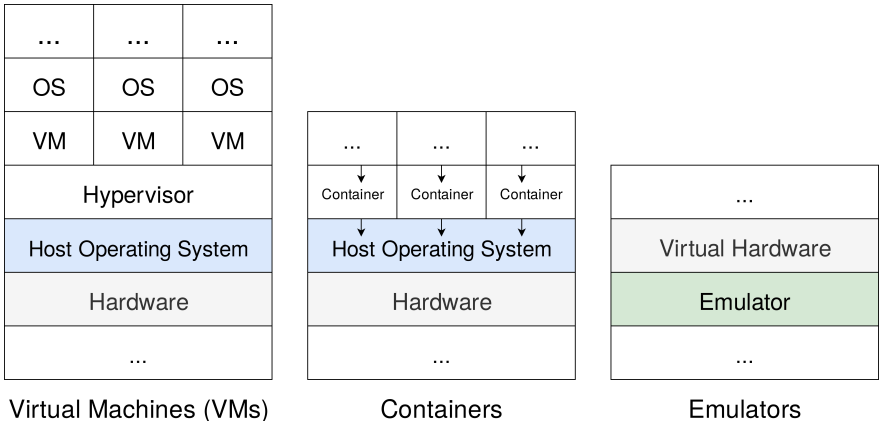


Fig. 5.1. Virtualization is a very resource expensive proposition. Containerization is cheaper, as it shares a kernel with the host OS. Emulation allows us to emulate hardware as software. Any of these methods can be used in conjunction with any other method.

hardware. The downside of both methods is their inefficiency. Virtual machines contain full-fledged operating systems and are cumbersome to run and debug, especially to build or run a small application on a foreign OS, as is often their use case. Emulators can run significantly more slowly than native machine code depending on the host and target architectures.

In 2006, Linux introduced a variety of new kernel features for controlling groups of processes, under the aegis of **cgroups** [Menage, 2007]. Collectively, these features support a form of lightweight virtualization, featuring many of the benefits of virtual machines (VMs) such as resource control and namespace isolation, without the computational overhead associated with full virtualization. These features paved the way for a set of tools that are today known as containers. Unlike VMs, containers share a common kernel, but remain isolated from their host OS and sibling containers. Where VMs often require server-class hardware to run smoothly, containers are suitable for a much broader class of mobile and embedded platforms due to their light resource footprint.

5.3. Containerization

One of the challenges of distributed software development across heterogeneous platforms is the problem of variability. With an increasing pace of software development comes the added burden of software maintenance. As hardware and software stacks evolve, so too must source code be updated to build and run correctly. Maintaining a stable and well-documented codebase can be a considerable challenge, especially in an academic setting

where contributors are frequently joining and leaving a project. Together, these challenges present significant obstacles to experimental reproducibility and scientific collaboration.

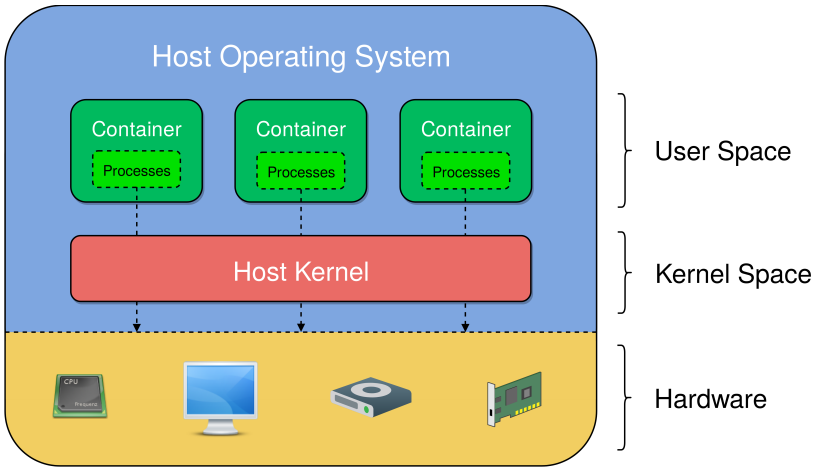


Fig. 5.2. Containers live in user space. By default they are sandboxed from the host OS and sibling containers, but unlike VMs, share a common kernel with each other and the host OS. All system calls are passed through host kernel.

Docker containers are sandboxed runtime environments that are portable, reproducible and version controlled. Each environment fully contains its dependencies, but remains isolated from the host OS and file system. Docker provides a mechanism to control the resources each container is permitted to access, and provisions a separate Linux namespace for each container, effectively isolating the network, users, and file system mounts from the host OS. Unlike virtual machines, container-based virtualization such as Docker only requires a lightweight host, which can support running many simultaneous containers with close to zero overhead compared to native Linux processes. A single Raspberry Pi is capable of simultaneously running hundreds of containers with no noticeable degradation in performance[1].

While containerization considerably simplifies the process of building and deploying applications, it also introduces some additional complexity to the software development lifecycle. Docker, like most container platforms, uses a layered filesystem. This enables Docker to take an existing "image" and change it by installing new dependencies or modifying its functionality. Images are typically based on a number of lower layers, which must periodically be

[1]cf. `https://blog.docker.com/2015/09/update-raspberry-pi-dockercon-challenge/`

updated. Care must be taken when designing the development pipeline to ensure that such updates do not silently break a subsequent layer, as we describe in section 5.7.

5.4. Docker Introduction

Suppose there is a program which is known to run on some computer. It would be nice to give another computer – any computer with an internet connection – a short string of ASCII characters, press ⏎, and return to see that program running. Never mind where the program was built or what software happened to be running at the time. This may seem trivial, but is a monumental software engineering problem. Various package managers have attempted to solve this, but even when they work as intended, only support natively compiled binaries on operating systems with the same package manager.

Docker[2] is a tool for portable, reproducible computing. With Docker, users can run any Linux program on almost any networked computing device on the planet, regardless of the underling operating system or hardware architecture. All of the environment preparation, installation and configuration steps can be automated from start to finish. Depending on how much network bandwidth is available, it might take some time, but users will never need to intervene in the installation process.

To install Docker itself, execute the following command on a POSIX-compliant shell of any Docker-supported platform:

```
~$ curl -sSL https://get.docker.com/ | sh
```

1

A Docker *image* is basically a filesystem snapshot – a single file that contains everything needed to run a certain Docker container. Docker images are hosted in *registries*, similar to Git repositories or VCS servers. The following command will fetch a Docker image, e.g. `daphne/duck` from the default Docker Hub repository:

```
~$ docker pull daphne/duck
```

1

Every Docker image has a image ID, a name and a tag:

[2]The following tutorial uses Docker, but the workflow described is similar to most container platforms.

```
~$ docker images
REPOSITORY      TAG         IMAGE ID          CREATED        SIZE
daphne/duck     latest      ea2f90g8de9e      1 day ago      869MB
```

To run a Docker container[3], use the following command:

```
~$ docker run daphne/duck
```

The following command will verify the container is indeed running:

```
~$ docker ps
CONTAINER ID     IMAGE           ...     NAMES
52994ef22481     daphne/duck     ...     happy_hamster
```

Note how Daphne's duck container has an alphanumeric container ID, a base image, and a memorable name, `happy_hamster`. This name is an alias for the container ID, which can be used interchangeably to refer to the container.

Docker images can be created two different ways. First, in subsection 5.4.1, we will see how to create a Docker image by taking a snapshot from a running container, then in subsection 5.4.2, how to create a new Docker container using a special kind of recipe, called a `Dockerfile`.

5.4.1. Creating an image snapshot

When a Docker container writes to its own filesystem, those changes are not persisted unless committed to a new image. For example, start a container with an interactive shell:

```
~$ docker run -it daphne/duck /bin/bash
root@295fd7879184:/#
```

Note the container ID: `295fd7879184`. If we write to disk and leave the container,

```
root@295fd7879184:/# touch new_file && ls -l
total 0
```

[3]When a Docker image is running, it is referred to as a *container*.

```
-rw-r--r-- 1 root root 0 May 21 20:52 new_file
root@295fd7879184:/# exit
```

`new_file` will not be persisted. If we re-run the same command again:

```
~$ docker run -it daphne/duck /bin/bash
root@18f13bb4571a:/# ls
root@18f13bb4571a:/# touch new_file1 && ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1
```

It seems like `new_file` has disappeared! Notice how the container ID (`18f13bb4571a`) is now different. This is because the command `docker run daphne/duck` created a new container from the base image `daphne/duck`, rather than restarting the previous container. To see all containers on a Docker host, run the following command:

```
~$ docker container ls -a
CONTAINER ID      IMAGE           STATUS            NAMES
295fd7879184      daphne/duck     Exited (130)      merry_manatee
18f13bb4571a      daphne/duck     Up 5 minutes      shady_giraffe
52994ef22481      daphne/duck     Up 10 minutes     happy_hamster
```

It appears `295fd7879184` a.k.a. `merry_manatee` survived, but it is no longer running. Whenever a container's main process (recall we ran `merry_manatee` with `/bin/bash`) finishes, the container will stop, but it will not cease to exist. In fact, we can resume the stopped container right where it left off:

```
~$ docker start -a merry_manatee
root@295fd7879184:/# ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 20:52 new_file
```

Nothing was lost! To verify this, we can check which other containers are running:

```
~$ docker ps
CONTAINER ID        IMAGE               ...       NAMES
295fd7879184        daphne/duck         ...       merry_manatee
18f13bb4571a        daphne/duck         ...       shady_giraffe
52994ef22481        daphne/duck         ...       happy_hamster
```

Now suppose we would like to share the container `shady_giraffe` with someone else. To do so, we must first snapshot the running container, or commit it to a new image with a name and a tag. This will create a checkpoint that we may later restore:

```
~$ docker commit -m "forking daphne/duck" shady_giraffe user/duck:v2
```

To refer to the container, we can either use `18f13bb4571a` or the designated name (i.e. `shady_giraffe`). This image repository will be called `user/duck`, and has an optional version identifier, `:v2`, which can be pushed to the Docker Hub registry:

```
~$ docker push user/duck:v2
~$ docker images
REPOSITORY    TAG         IMAGE ID         CREATED          SIZE
daphne/duck   latest      ea2f90g8de9e     1 day ago        869MB
user/duck     v2          d78be5cf073e     2 seconds ago    869MB
~$ docker pull user/duck:v2
~$ docker run user/duck ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1
```

This is a convenient way to share an image with colleagues and collaborators. Anyone with access to the repository can pull this image and continue where we left off, or create another image based on top.

5.4.2. Writing an image recipe

The second way to create a Docker image is to write a recipe, called a `Dockerfile`. A `Dockerfile` is a text file that specifies the commands required to create a Docker image,

typically by modifying an existing container image using a scripting interface. They also have special keywords (which are conventionally `CAPITALIZED`), like `FROM`, `RUN`, `ENTRYPOINT` and so on. For example, create a file called `Dockerfile` with the following content:

```
FROM daphne/duck      # Defines the base image
RUN touch new_file1   # new_file1 will be part of our snapshot
CMD ls -l             # Default command run when container is started
```

Now, to build the image, we can simply run:

```
~$ docker build -t user/duck:v3 .
```

The '`.`' indicates the same path as the `Dockerfile`. This command should produce something like the following output:

```
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM daphne/duck
--- ea2f90g8de9e
Step 2/3 : RUN touch new_file1
--- e3b75gt9zyc4
Step 3/3 : CMD ls -l
--- Running in 14f834yud59
Removing intermediate container 14f834yud59
--- 05a3bd381fc2
Successfully built 05a3bd381fc2
Successfully tagged user/duck:v3
```

The command, `docker images` should display an image called `user/duck:v3`:

```
~$ docker images
REPOSITORY    TAG        IMAGE ID         CREATED          SIZE
daphne/duck   latest     ea2f90g8de9e     1 day ago        869MB
user/duck     v2         d78be5cf073e     5 minutes ago    869MB
user/duck     v3         05a3bd381fc2     2 seconds ago    869MB
```

This procedure is identical to the snapshot technique performed in subsection 5.4.1, but the result is cleaner. Rather than maintaining a 869 MB image, we can just store the 4 KB text file instead. To run the resulting image, we can simply use the same command as before:

```
~$ docker run -it user/duck:v3
total 0
-rw-r--r-- 1 root root 0 May 21 21:35 new_file1
```

Notice that as soon as we run the container, Docker will execute the `ls -l` command as specified by the `Dockerfile`, revealing that `new_file1` was indeed stored in the image. However this default command can be overridden by supplying a custom command:

```
~$ docker run -it user/duck:v3 [custom_command]
```

5.4.3. Layer Caching

Layers are an important concept to understand when working with Docker. One way to think of a layer is like a Git commit – a set of changes to a previous image or layer, uniquely identified by a hash code. In a `Dockerfile`, layers begin with a keyword.

```
FROM daphne/duck
RUN touch new_file1                          # Defines a new layer
RUN mkdir config && mv new_file1 mkdir       # Layers can chain commands
RUN apt-get update && apt-get install -y wget  # Install a dependency
RUN wget https://get.your.app/install.sh     # Download a script
RUN chmod +x install.sh && ./install.sh      # Run the script
```

To build this image, we can run the following command:

```
~$ docker build -t user/duck:v4 .
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM daphne/duck
---> cd6d8154f1e1
...
```

```
Removing intermediate container 8fb56ef38bc8
---> 3358ca1b8649
Step 5/6 : RUN wget https://get.your.app/install.sh
---> Running in e8284ff4ec8b
...
2018-10-30 06:47:57 (89.9 MB/s) - 'install.sh' saved [13847/13847]
Removing intermediate container e8284ff4ec8b
---> 24a22dc2900a
Step 6/6 : RUN chmod +x install.sh && ./install.sh
---> Running in 9526651fa492
# Executing install script, commit: 36b78b2
...
Removing intermediate container 9526651fa492
---> a8be23fea573
Successfully built a8be23fea573
Successfully tagged user/duck:v4
```

Layers are conveniently cached by the Docker daemon. Should we need to run the same command twice, Docker will use the cache instead of rebuilding the entire image:

```
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM daphne/duck
---> cd6d8154f1e1
Step 2/6 : RUN touch new_file1
---> Using cache
---> 0473154b2004
...
Step 6/6 : RUN chmod +x index.html && ./index.html
---> Using cache
---> a8be23fea573
Successfully built a8be23fea573
Successfully tagged user/duck:v4
```

If we need to make a change to the Dockerfile, Docker will only rebuild the image starting from the first modified step. Suppose we add a new `RUN` command to the end of our `Dockerfile` and trigger a rebuild like so:

```
~$ echo 'RUN echo "Change here!"' >> Dockerfile
~$ docker build -t user/duck:v4 .
Sending build context to Docker daemon  2.048kB
...
Step 6/7 : RUN chmod +x index.html && ./index.html
---> Using cache
---> a8be23fea573
Step 7/7 : RUN echo "Change here!"
---> Running in 80fc5c402304
Change here!
Removing intermediate container 80fc5c402304
---> c1ec64cef9c6
Successfully built c1ec64cef9c6
Successfully tagged user/duck:v4
```

If Docker had to rerun the entire `Dockerfile` from top to bottom to every time it was rebuilt, this would be slow and inconvenient. Instead, Docker caches the unmodified steps by default, and only reruns the minimum set of steps necessary to rebuild. This can sometimes introduce unexpected results, especially when the cache is stale. To ignore the cache and force a clean rebuild, use the `-no-cache` flag when building a `Dockerfile`.

What does Docker consider when deciding whether to use the cache? First is the `Dockerfile` itself – when a step in a `Dockerfile` changes, both it and any subsequent steps will need to be rerun during a build. Docker also checks the build context for changes. When "`docker build -t TAG .`" is written, the '`.`' indicates the build context, or path where the build should occur. Often, this path contains build artifacts. For example:

```
FROM daphne/duck
COPY duck.txt .
```

```
RUN cat duck.txt
```

Now if we add some message in `duck.txt` and rebuild our image, the file will be copied into the Docker image, and its contents will be printed:

```
~$ echo "Make way!" > duck.txt && docker build -t user/duck:v5 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM daphne/duck
---> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
---> e0e03d9e1791
Step 3/3 : RUN cat duck.txt
---> Running in 590c5420ce29
Make way!
Removing intermediate container 590c5420ce29
---> 1633e3e10bef
Successfully built 1633e3e10bef
Successfully tagged user/duck:v5
```

As long as the first three lines of the `Dockerfile` and `duck.txt` are unmodified, these layers will be cached and Docker will not rebuild them. If the contents of the file `duck.txt` are subsequently modified, this will trigger a rebuild to occur. For example, if we append to the file and rebuild, the last two steps will be executed:

```
~$ echo "Thank you. Have a nice day!" >> duck.txt
~$ docker build -t user/duck:v5 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM ubuntu
---> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
---> f219efc150a5
Step 3/3 : RUN cat duck.txt
---> Running in 7c6f5f8b73e9
```

```
Make way!
Thank you. Have a nice day!
Removing intermediate container 7c6f5f8b73e9
---> e8a1db712aee
Successfully built e8a1db712aee
Successfully tagged user/duck:v5
```

A common mistake when writing `Dockerfile`s is to `COPY` more files than are strictly necessary to perform the following build step. For example, if `COPY . .` is written at the beginning of the `Dockerfile`, whenever a file in changed anywhere in the build context, this will trigger a rebuild of all subsequent build steps. In order to maximize cache reusability and minimize rebuild time, users should be conservative and only `COPY` the minimum set of files necessary to accomplish the following build step.

5.4.4. Volume Sharing

There is a second method of depositing data into a container, which does not require baking it into the parent image at compile time. This method is more appropriate for data which is required at runtime, but non-essential for the build. It takes the following form:

```
~$ docker run user/duck:v6 -v [HOST PATH]:[TARGET PATH]
```

Suppose we have a `Dockerfile` which provides a default `CMD` instruction:

```
FROM daphne/duck
CMD /bin/bash -c "/launch.sh"
```

If we built this image and tried to run it, the file `launch.sh` would be missing:

```
~$ docker build -t user/duck:v6 && docker run user/duck:v6
bash: /launch.sh: No such file or directory
```

Instead, when running the container, we need to share the file via the Docker CLI:

```
~$ echo -e '#!/bin/bash\necho Launching...' >> launch.sh && \
   chmod 775 launch.sh && \
   docker run user/duck:v6 -v launch.sh:/launch.sh
Launching...
```

This way, the local file `launch.sh` will be available to use from within the container at the designated path, `/launch.sh`.

5.4.5. Multi-stage builds

Docker's filesystem is additive, so each layer will only increase the size of the final image. For this reason, it is often necessary to tidy up unneeded files after installation. For example, when installing dependencies on Debian-based images, it is a common practice to run:

```
RUN apt-get update && apt-get install ... && rm -rf /var/lib/apt/lists/
```

This ensures the package list is not baked into the image (Docker will only checkpoint the layer after each step is complete). Builds can often take several steps, despite only producing a single artifact. Instead of chaining together several commands and cleaning up changes in a single step, multi-stage builds let us build a series of images inside a `Dockerfile`, and copy resources from one to another, discarding all intermediate build artifacts:

```
FROM user/duck:v3 as template1

FROM daphne/duck as template2
COPY --from=template1 new_file1 new_file2
FROM donald/duck as template3
COPY --from=template2 new_file2 new_file3
CMD ls -l
```

Now we can build and run this image as follows:

```
~$ docker build . -t user/duck:v4
Sending build context to Docker daemon  2.048kB
```

```
Step 1/6 : FROM user/duck:v3 as template1
--- e3b75ef8ecc4
Step 2/6 : FROM daphne/duck as template2
--- ea2f90g8de9e
Step 3/6 : COPY --from=template1 new_file1 new_file2
---> 72b96668378e
Step 4/6 : FROM donald/duck:v3 as template3
---> e3b75ef8ecc4
Step 5/6 : COPY --from=template2 new_file2 new_file3
---> cb1b84277228
Step 6/6 : CMD ls
---> Running in cb1b84277228
Removing intermediate container cb1b84277228
---> c7dc5dd63e77
Successfully built c7dc5dd63e77
Successfully tagged user/duck:v4
~$ docker run -it user/duck:v4
total 0
-rw-r--r-- 1 root root 0 Jul  8 15:06 new_file3
```

One application of multi-stage builds is compiling a project dependency from its source code. In addition to all the source code, the compilation process could introduce gigabytes of build artifacts and transitive dependencies, just to build a single binary. Multi-stage builds allow us to build the file, and copy it to a fresh layer, unburdened by intermediate files.

5.5. ROS and Docker

Prior work has explored the Dockerization of ROS containers [White and Christensen, 2017]. This work forms the basis for our own, which extends White and Christensen's work to the Duckietown platform [Paull et al., 2017], which is more hardware- and domain-specific.

The Duckietown platform supports two primary instruction set architectures: x86 and ARM. To ensure the runtime compatibility of Duckietown packages, we cross-build using hardware virtualization to ensure build artifacts can be run on either target architecture.

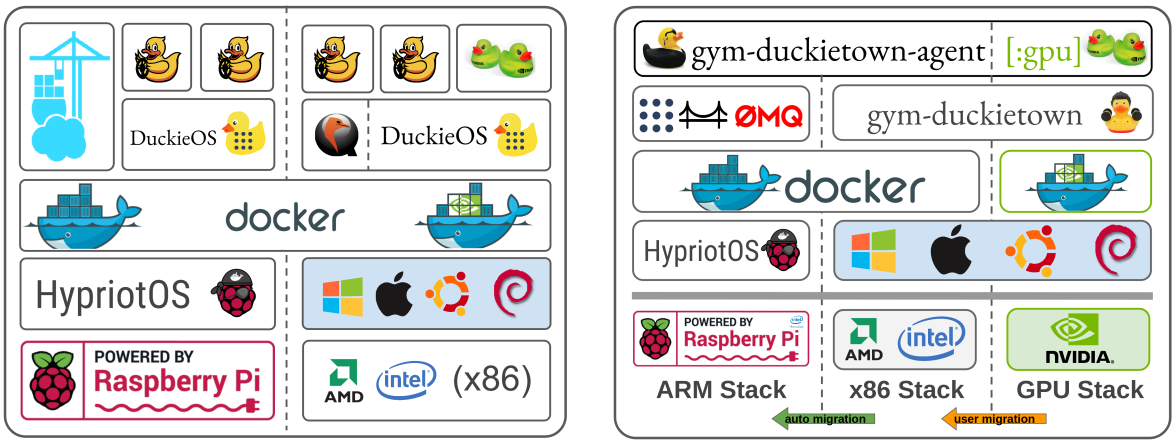


Fig. 5.3. Container infrastructure. **Left**: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we build ARM artifacts, which can be emulated on x86 via qemu [Bellard, 2005]. **Right**: Reinforcement learning stack. Build artifacts are trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their architecture, may be run on an ARM device using an accelerator.

Runtime emulation of foreign artifacts is also possible, using a similar technique.[4] For performance and simplicity, we only use emulation where necessary (e.g., on x86 devices). On ARM-native, the base operating system is HypriotOS, a lightweight Debian distribution for the Raspberry Pi and other ARM-based SBCs, with native support for Docker. For both x86 and ARM-native, Docker is the underlying container platform upon which all user applications are run, inside a container. Since both ROS and Docker have extensive command line interfaces, we provide a unified interface, the Duckietown Shell (`dts`), which wraps their functionality and runs common tasks.

5.6. Duckiebot Development using Docker

Software development for the Duckietown platform requires the following physical objects:

(1) Duckiebot (including custom hat, camera, wheels, and Raspberry Pi 3B+)[5]

(2) Micro SD card (16GB+ recommended)

(3) Personal computer

[4]For more information, this technique is described in further depth at the following URL: `https://www.balena.io/blog/building-arm-containers-on-any-x86-machine-even-dockerhub/`.

[5]Full materials list can be located at the following URL: `https://get.duckietown.org/`

(4) Internet-enabled router

(5) MicroSD card adapter

In addition, we assume the following software dependencies have been installed on (3):

(a) Docker CE

(b) POSIX-compliant shell

(c) `dts`, the Duckietown shell[6]

(d) Web browser (e.g. Chrome or Firefox)

(e) `wget`/`curl`

The following workflow has been tested extensively on Linux hosts running Ubuntu 16.04 (and to a lesser extent, Mac OS X and VMs). No other dependencies are assumed or required.

5.6.1. Flashing a bootable disk

Our largest single contribution to the Duckietown project involved writing a script for flashing bootable media with a DuckieOS image. Prior to its creation, installation was a manual and time-consuming process. This was automated in a bash script.

```
~$ bash -c "$(wget -O- h.ndan.co)"
```

1

Now, with the Duckietown Shell, the following command is all that is needed:

```
dt> init_sd_card [--hostname "DUCKIEBOT_NAME" --wifi "username:password"]
```

Users insert an SD card and follow the instructions provided. When complete, the card must be removed and inserted into the SD card slot on the Raspberry Pi. On first boot, care should be taken to ensure the device is powered continuously for a minimum of ten minutes.

5.6.2. Web interface

To access the DuckieOS web interface, users can visit the following URL in any JavaScript-enabled web browser: `http://DUCKIEBOT_NAME:9000/`. If the installation process successfully completed and the network is properly configured, the web application displayed in Figure 5.4 should be accessible. This application allows users who are uncomfortable using command line applications to manage software running on their Duckiebots.

[6]May be obtained at the following URL: `https://github.com/duckietown/duckietown-shell`

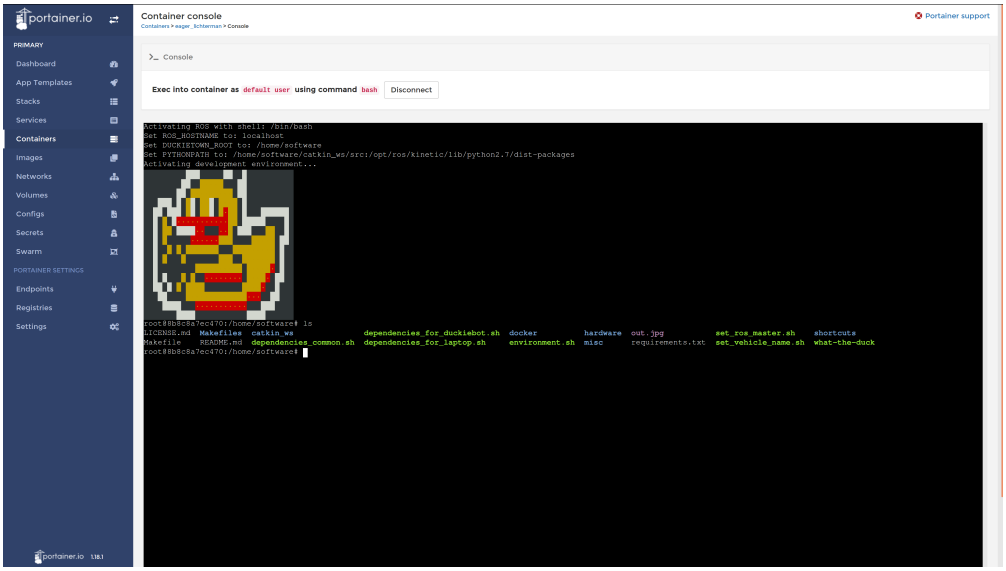


Fig. 5.4. Browser interface for individual Duckiebots. It is provided by Portainer, a RESTful web dashboard, which wraps the Docker CLI and offers support for container management, configuration, networking and terminal emulation (seen above), provided by xterm.js which is accessible at: `http://DUCKIEBOT_NAME:9000/#/container/container_name` » "Console"

5.6.3. Testing ROS

To verify Docker is working properly, launch a remote container, interactively, like so:

```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host \
   duckietown/rpi-ros-kinetic-base:master18
```

The `-H` flag indicates a remote Docker host on the local area network where the Docker command should be executed. For the `DUCKIEBOT_NAME` address it to work, mDNS must be configured properly in the network settings beforehand, otherwise an IP address is required.

5.6.4. Build and deployment

Docker images can be cross-compiled by enclosing the ARM-specific portion of the `Dockerfile` with the `RUN [ "cross-build-start" ]` and `RUN [ "cross-build-end" ]` instructions. The following command can be used for deployment:

```
~$ docker save TAG_NAME | ssh -C duckie@DUCKIEBOT_NAME docker load
```

Alternately, it is possible to build directly on ARM devices by creating a file named `Dockerfile.arm`, adding a base image and build instructions, then running the command:

```
~$ docker build --file=[FILE PATH]/Dockerfile.arm --tag [TAG NAME] .
```

5.6.5. Multi-architecture support

As of Docker version 18.09.6, ARM-specific `Dockerfile`s will not build on x86 machines[7], and attempting to build one will produce the following error when running `docker build`:

```
standard_init_linux.go:175: exec user process caused "exec format error"
```

1

In order to circumvent this restriction, ARM-specific `Dockerfile`s can ported to run on x86 by using the `RUN [ "cross-build-start" ]` and `RUN [ "cross-build-end" ]` directives, after the `FROM` and before the `CMD` instructions.

All Duckietown Docker images contain an emulator called QEMU - this allows us to run ARM images on x86 directly. To run a pure compute ROS node (i.e. one that does not require any camera or motor access) on an x86 platform, developers must supply a custom entrypoint to Docker when running the image using the entrypoint flag as follows:

```
~$ docker run ... --entrypoint=qemu3-arm-static IMAGE [RUN_COMMAND]
```

1

Here, `RUN_COMMAND` may be a shell such as `/bin/bash` or another command such as `/bin/bash -c "roscore"`. The entrypoint refers to the ARM emulator packaged within the base image, `duckietown/rpi-ros-kinetic-base`, which allows ARM binaries to be run on x86 hosts.

5.6.6. Running a simple HTTP file server

All persistent data is stored in `/data`. To serve this directory, a web server is provided:

[7] With the exception of the Mac OS Docker client, which offers multi-architecture support. Further details on multiarch support can be found here: `https://docs.docker.com/docker-for-mac/multi-arch/`. Later versions of Docker for Mac OS and Windows provide native ARM emulation, which was recently announced at the following URL: `https://engineering.docker.com/2019/04/multi-arch-images/`

```
~$ docker -H DUCKEBOT_NAME run -d -v /data:/data -p 8082:8082 \
   duckietown/rpi-simple-server:master18
```

To access this directory, visit the following URL: `http://DUCKIEBOT_NAME:8082/`

5.6.7. Testing the camera

The following command can be used to test the camera is working properly. By default, images will be hosted at: `http://DUCKIEBOT_NAME:8081/image.jpg`

```
~$ docker -H DUCKIEBOT_NAME run -d --privileged -v /data:/data -p 8081:8081
   duckietown/rpi-docker-python-picamera:master18
```

Like most commands, a Python-based shell is provided for the user's convenience:

```
dt> duckiebot demo --demo_name camera --duckiebot_name DUCKIEBOT_NAME
```

5.6.8. Graphical User Interface Tools

To use GUI tools, one must first allow incoming X connections from the host. On Linux hosts, this can be done by running `xhost +` outside Docker.[8] A container with common ROS GUI plugins can be started with following command:

```
~$ docker run -it --rm --net host \
   --env ROS_MASTER_URI=http://DUCKIEBOT_IP:11311 \
   --env ROS_IP=LAPTOP_IP \
   --env="DISPLAY" \
   --env="QT_X11_NO_MITSHM=1" \
   --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
   duckietown/rpi-gui-tools
```

Packaged within this image are common ROS plugins which can be run on graphical environments. A shell wrapper is also provided for convenience:

[8] See `http://wiki.ros.org/docker/Tutorials/GUI#The_safer_way` for a more secure alternative.

```
dt> start_gui_tools DUCKIEBOT_NAME rqt_image_view
```

The above command opens a ROS shell that will connect to the `DUCKIEBOT`'s ROS master node. To test the ROS connection works, run `roswtf`.

5.6.9. Remote control

The following container launches the joystick demo (USB joystick must be connected):

```
~$ docker -H DUCKIEBOT_NAME run --privileged --net host -v /data:/data \
   duckietown/rpi-duckiebot-joystick-demo:master18
```

```
dt> duckiebot demo --demo_name joystick --duckiebot_name DUCKIEBOT_NAME
```

```
dt> duckiebot keyboard_control DUCKIEBOT_NAME
```

5.6.10. Camera Calibration

The following container will launch the extrinsic calibration procedure:

```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host -v /data:/data
   duckietown/rpi-duckiebot-calibration:master18
```

Passing `-v /data:/data` is necessary so that all calibration settings will be preserved. When placed on the calibration pattern, the following commands will initiate an interactive calibration sequence for the camera.

```
dt> duckiebot calibrate_extrinsics DUCKIEBOT_NAME
```

```
dt> duckiebot calibrate_intrinsics DUCKIEBOT_NAME
```

5.6.11. Wheel Calibration

To calibrate the gain and trim of the wheel motors, the following commands are needed:

```
dt> duckiebot demo --demo_name base --duckiebot_name NAME
```

```
~$ rosservice call /DUCKIEBOT_NAME/inverse_kinematics_node/set_gain --GAIN    1
```

```
~$ rosservice call /DUCKIEBOT_NAME/inverse_kinematics_node/set_trim --TRIM    1
```

5.6.12. Lane Following Demo

Once calibrated, the lane following demo can be launched as follows:

```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host -v /data:/data    1
   duckietown/rpi-duckiebot-lanefollowing-demo:master18                       2
```

```
dt> duckiebot demo --demo_name lane_following --duckiebot_name DUCKIEBOT_NAME
```

5.7. Retrospective

One problem encountered during the development of Duckietown's Docker infrastructure was the matter of whether to package source code directly inside the container, or to store the sources externally (e.g. as described in subsection 5.4.4). If stored externally, a developer can share the sources in a shared volume and build the artifacts on container startup. Both approaches ensure reproducible build dependencies, but including the the source code in the image makes build artifacts are more interpretable and reduces startup time.

Initially, we made the explicit decision to ship user source code directly inside the image. As a consequence, any modifications to the source code would trigger a subsequent rebuild, tying the sources and Docker image together. While including sources enables easier troubleshooting and diagnostics, doing so adds some friction during development, which caused users to struggle with environment setup and Docker configuration issues.

The root cause of this friction was a product of imprecise versioning and over-automation. As version tags were initially omitted, all images were built and pulled from latest commit on the mainline development branch. The auto-build feature of the CI server caused upstream modifications to cascade to downstream images. The short-term solution was to disable

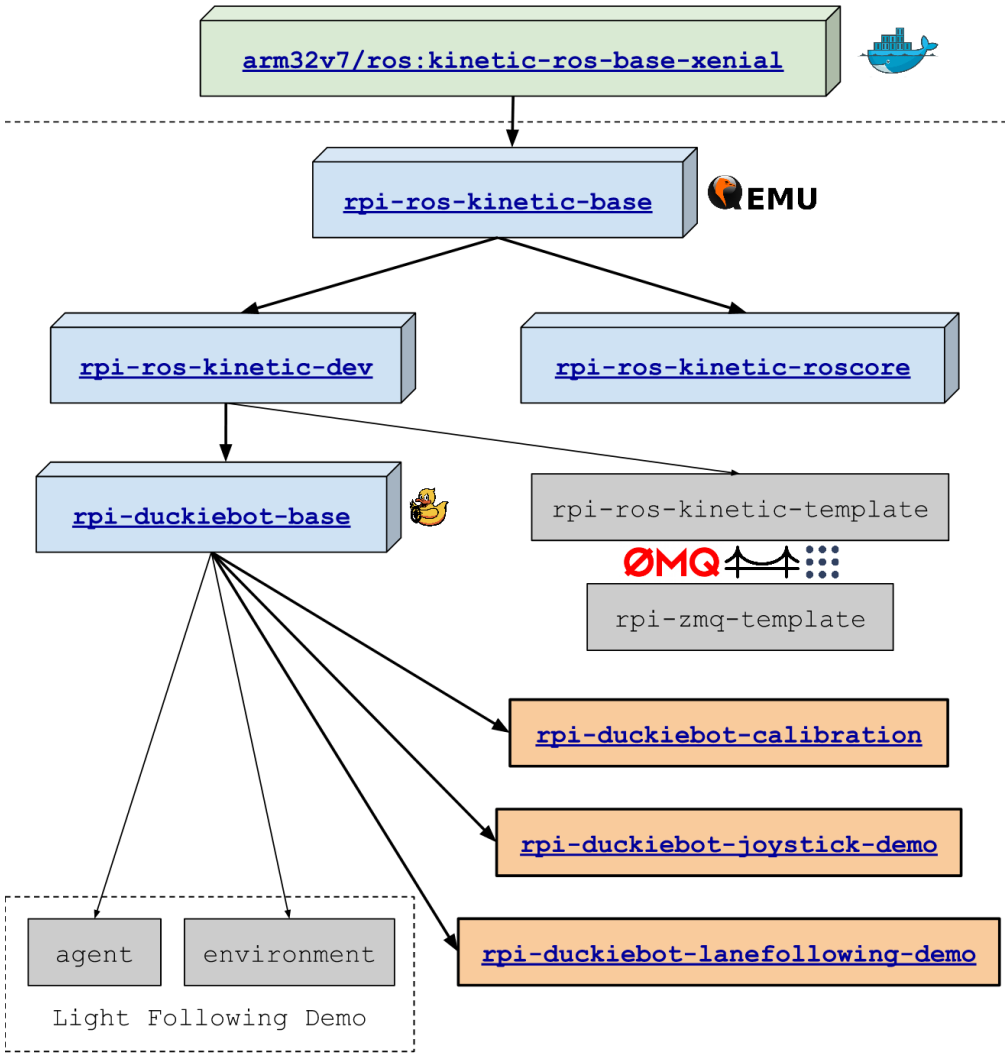


Fig. 5.5. Early prototype of the Docker image hierarchy. Chaining unversioned autobuilds without disciplined unit testing tends to produce an undesirable domino effect, where breaking changes are allowed to propagate downstream, resulting in a cascade of silent failures.

auto-building, and push local builds to the server manually, however fixing it required us to rethink the role of versioning and testing Docker builds in the CI toolchain.

One of the primary use cases for the container infrastructure is a biannual robotics competition called the AI Driving Olympics [Zilly et al., 2019]. We discovered a more stable solution is to store all sources on the local development environment and rebuild the image only when its upstream dependencies change. A Docker image, paired with a Git repository

Fig. 5.6. The AI Driving Olympics, a primary use case for the system described above.

and a commit message defines an AIDO submission. The image only contains its compiled upstream dependencies, and the source code is paired at runtime.

Chapitre 6

Case study: application for autonomous robotics

> "Thus, I came to the conclusion that the designer of a new system must not only be the implementor and the first large-scale user; the designer should also write the first user manual. The separation of any of these four components would have hurt TeX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important."
>
> – Donald E. Knuth [1989], *The errors of TeX*

As a case study, we have implemented a mobile application using ROS, Docker, and Android, using the proposed toolchain.

6.1. Design

Designed with Hatchery.

6.2. Implementation

Implementation includes Kotlin∇

6.3. Testing and validation

Verified using property-based testing.

6.4. Containerization

Deployed and CI-tested using Docker.

Chapitre 7

Conclusion

7.1. Future work

7.1.1. Requirements Engineering

Often it is not possible, or desirable to summarize the performance of a complex system using a single variable. In multi-objective optimization, we have the notion of pareto-efficiency...

Traditional software engineering has followed a rigorous process model and testing methodology. This model has guided the development of traditional software engineering, intelligent systems will require a re-imagining of these ideas to build systems that adapt to their environment during operation. Intelligent systems are designed with objective functions, which are typically one- or low-dimensional metrics for evaluating the performance of the system. Most often, these take the form of a single criteria, such as an *error* or *loss* which can represent descriptive phenomena such as latency, safety, energy efficiency or any number of objective measures.

For example, in the design of a web based advertisement recommendation system, we can optimize for various objectives such as click rate, engagement, sales conversion. So long as we can measure these parameters, with today's powerful function approximators, we can optimize for any single criterion or combination thereof. Much of the work involved in machine learning is to find representations which are amenable to learning, and preventing unintended consequences. For example, by optimizing for click rate, we create an artificial market for click bots. Similarly, in self driving cars, we often want to optimize for passenger

safety. However by doing so naïvely, we create a vehicle that never moves, or always yields to nearby vehicles.

When building an intelligent system developers must first ask, "What are the requirements of the system?" This question is often the most troublesome part, because the requirements must not be fuzzy specifications like traditional software engineering, but precise, programmable directives. "The system must be fast," is not sufficiently precise. These kinds of requirements must be translated into statistical loss functions, so intelligent systems engineers must be very precise when specifying requirements. If we simply say, "The system must produce a valid response as quickly as possible, in less than 100ms," is better, but leaves open the possibility of returning an empty response.

In traditional software engineering, it is reasonable to assume the people who are implementing a system have some implicit knowledge and are generally well-intentioned human beings working towards the same goal. When building an intelligent system, a more reasonable assumption is that the entity implementing our requirements is a naïve but powerful genie, and possibly an adversarial one. When given an optimization metric, it will take every available shortcut to meet that metric. If we are not careful about requirements engineering, this entity can produce a system that does not work, or has unintended consequences.

In the strictest sense, designing a good set of requirements is indistinguishable from implementing the system. With the right language abstractions (e.g. declarative programming), requirements and implementation can be the same thing. These ideas have been explored in recent decades with languages like SQL and Prolog. While these are toy systems, neural networks can express much larger classes of functions than traditional software engineering.

7.1.2. Continuous Delivery and Continual Learning

An ongoing trend in modern software and systems engineering is the transition away from long development cycles towards continuous integration and deployment. Development teams across the industry are encouraged to iterate in a series of short sprints between feature development and deployment. In some cases, software is shipped to users on a nightly basis, with automated testing and deployment. Similarly, intelligent systems have a need to continuously adapt to their environment, and will change their behavior on an even shorter basis.

Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pages 265–283, 2016.

Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. Journal of Systems and Software, 85(10):2228–2240, 2012.

Peter Abeles. Efficient java matrix library, 2010.

Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad

Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions. CoRR, abs/1605.02688, 2016. URL http://arxiv.org/abs/1605.02688.

Isabela Albuquerque, Joao Monteiro, Thang Doan, Breandan Considine, Tiago Falk, and Ioannis Mitliagkas. Multi-objective training of generative adversarial networks with multiple discriminators. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36th International Conference on Machine Learning, volume 97 of Proceedings of Machine Learning Research, pages 202–211, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/albuquerque19a.html.

Mohammed AlQuraishi. End-to-end differentiable learning of protein structure. Available at SSRN 3239970, 2018.

Nada Amin and Ross Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. Acm Sigplan Notices, 51(10):838–848, 2016.

John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. ACM, 2007.

Junjie Bai, Fang Lu, Ke Zhang, et al. ONNX: Open neural network exchange, 2019. URL https://github.com/onnx/onnx.

Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. Automatic differentiation in machine learning: a survey. CoRR, abs/1502.05767, 2015a. URL http://arxiv.org/abs/1502.05767.

Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. DiffSharp: Automatic differentiation library. CoRR, abs/1511.07727, 2015b. URL http://arxiv.org/abs/1511.07727.

Fabrice Bellard. Qemu, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track, volume 41, page 46, 2005.

Richard Ernest Bellman, Ho Kagiwada, and Robert E Kalaba. Wengert's numerical method for partial derivatives, orbit determination and quasilinearization. Communications of the ACM, 8(4):231–232, 1965.

Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2):157–166, 1994.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In Proceedings of the Python for scientific computing conference (SciPy), volume 4. Austin, TX, 2010.

Yang Bo. Deep Learning.scala: A simple library for creating complex neural networks. 2018. URL `https://github.com/ThoughtWorksInc/DeepLearning.scala`.

Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. In 2018 IEEE International Conference on Robotics and Automation (ICRA), pages 4243–4250. IEEE, 2018.

Mikio L Braun, Johannes Schaback, Matthias L Jugel, Nicolas Oury, et al. jBlas: Linear algebra for Java, 2011.

Roman V Buniy, Stephen DH Hsu, and Anthony Zee. Is Hilbert space discrete? Physics Letters B, 630(1-2):68–72, 2005.

Yufei Cai, Paolo G Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In ACM SIGPLAN Notices, volume 49, pages 145–155. ACM, 2014.

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In FPCA, volume 89, pages 273–280, 1989.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. Journal of statistical software, 76(1), 2017.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Tongfei Chen. Typesafe abstractions for tensor operations (short paper). pages 45–50, 2017. doi: 10.1145/3136000.3136001. URL http://doi.acm.org/10.1145/3136000.3136001.

Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong . . . , 1998.

Yan Chen, Joshua Dunfield, and Umut A Acar. Type-directed automatic incrementalization. ACM SIGPLAN Notices, 47(6):299–310, 2012.

Alonzo Church. The calculi of lambda-conversion. Princeton University Press, 1941.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. Acm sigplan notices, 46(4):53–64, 2011.

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Assessing the bus factor of Git repositories. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 499–503. IEEE, 2015.

Sébastien Crozet et al. nalgebra: a linear algebra library for Rust, 2019. URL https://nalgebra.org.

Haskell B Curry. Combinatory logic. 1967.

Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. arXiv preprint arXiv:1801.08058, 2018.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In Advances in Neural

Information Processing Systems, pages 7178–7189, 2018.

Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis Wyffels. A differentiable physics engine for deep learning in robotics. CoRR, abs/1611.01652, 2016. URL `http://arxiv.org/abs/1611.01652`.

Commons Math Developers. Apache commons math. Forest Hill, MD, USA: The Apache Software Foundation, 2012.

Edsger W Dijkstra. The humble programmer. Commun. ACM, 15(10):859–866, 1972.

Stuart E Dreyfus. Artificial neural networks, back propagation, and the Kelley-Bryson gradient procedure. Journal of guidance, control, and dynamics, 13(5):926–928, 1990.

Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Jianjun Zhao, and Yang Liu. Deepcruiser: Automated guided testing for stateful deep learning systems. arXiv preprint arXiv:1812.05339, 2018.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285, 2016.

Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. Theoretical Computer Science, 309(1-3):1–41, 2003.

Conal Elliott. The simple essence of automatic differentiation. Proceedings of the ACM on Programming Languages, 2(ICFP):70, 2018.

Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. Journal of functional programming, 13(3):455–481, 2003.

Conal M Elliott. Beautiful differentiation. In ACM Sigplan Notices, volume 44, pages 191–202. ACM, 2009.

Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pages 307–309. ACM, 2010.

George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes, 22(4):74–80, 1997.

Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In ACM SIGPLAN Notices, volume 39, pages 111–122. ACM, 2004.

M. Fowler. Fluent interface, 2005. URL http://martinfowler.com/bliki/FluentInterface.html.

Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in JavaScript. In Proceedings of the 39th International Conference on Software Engineering, pages 758–769. IEEE Press, 2017.

Yossi Gil and Tomer Levy. Formal language recognition with the Java type checker. 56, 2016.

Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. Monatshefte für mathematik und physik, 38(1):173–198, 1931.

Charles F Goldfarb. A generalized approach to document markup. In ACM Sigplan Notices, volume 16, pages 68–73. Citeseer, 1981.

Maria I Gorinova, Andrew D Gordon, and Charles Sutton. Slicstan: Improving probabilistic programming using information flow analysis.

Andreas Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. In Complexity in numerical optimization, pages 128–162. World Scientific, 1993.

Andreas Griewank et al. On automatic differentiation. Mathematical Programming: recent developments and applications, 6(6):83–107, 1989.

PR Griffioen. Type inference for array programming with dimensioned vector spaces. In Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, page 4. ACM, 2015.

Radu Grigore. Java generics are Turing Complete. pages 73–85, 2017. doi: 10.1145/3009837.3009871. URL http://doi.acm.org/10.1145/3009837.3009871.

Martin Guenther. Are serious things done with ROS in Python?, 2018. URL https://discourse.ros.org/t/are-serious-things-done-with-ros-in-python/4359/6. (Accessed on 04/12/2019).

Warren Harrison. Eating your own dog food. IEEE Software, 23(3):5–7, 2006.

Pieter Hintjens. ZeroMQ: messaging for many applications. "O'Reilly Media, Inc.", 2013.

Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. Context-oriented programming. Journal of Object technology, 7(3):125–151, 2008.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. Neural networks, 2(5):359–366, 1989.

Teijiro Isokawa, Tomoaki Kusakabe, Nobuyuki Matsui, and Ferdinand Peper. Quaternion neural network and its application. In International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, pages 318–324. Springer, 2003.

Alekseĭ Grigorevich Ivakhnenko and Valentin Grigorévich Lapa. Cybernetic predicting devices. CCM Information Corporation, 1965.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2704–2713, 2018.

C Barry Jay and Milan Sekanina. Shape checking of array programs. Technical report, Citeseer.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In Proceedings of the 22nd ACM international conference on Multimedia, pages 675–678. ACM, 2014.

Nidhi Kalra and Susan M Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? Transportation Research Part A: Policy and Practice, 94:182–193, 2016.

Robert Kelly, Barak A Pearlmutter, and Jeffrey Mark Siskind. Evolving the incremental λ calculus into a model of forward automatic differentiation. arXiv preprint arXiv:1611.03429, 2016.

Andrew Kennedy. Dimension types. In European Symposium on Programming, pages 348–362. Springer, 1994.

Andrew John Kennedy. Programming languages and dimensions. Technical report, University of Cambridge, Computer Laboratory, 1996.

Oleg Kiselyov. Number-parameterized types. The Monad. Reader, 5:73–118, 2005.

Oleg Kiselyov, S Peyton Jones, and Chung-chieh Shan. Fun with type functions (version 3). Tony Hoare's 75th birthday celebration, 2010.

Gerwin Klein, Steve Rowe, and Régis Décamps. JFlex-the fast scanner generator for Java. URL: http://www. jflex. de, 2001.

DE Knuth. The errors of TeX. software–practice and experience. Literate Programming; CSLI Lecture Notes, (27):607–681, 1989.

Rainer Koppler. A systematic approach to fuzzy parsing. Software: Practice and Experience, 27(6):637–649, 1997.

Chris Lattner and Jacques Pienaar. MLIR primer: A compiler infrastructure for the end of moore's law, 2019. URL `https://ai.google/research/pubs/pub48035`.

Chris Lattner and Richard Wei. Swift for TensorFlow. 2018. URL `https://github.com/tensorflow/swift`.

Söeren Laue. On the equivalence of forward mode automatic differentiation and symbolic differentiation. arXiv preprint arXiv:1904.02990, 2019.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. nature, 521(7553):436, 2015.

Joseph Carl Robnett Licklider. Man-computer symbiosis. IRE transactions on human factors in electronics, (1):4–11, 1960.

Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, pages 6–7, 1970.

Barbara Liskov. Keynote address - data abstraction and hierarchy. SIGPLAN Not., 23(5): 17–34, January 1987. ISSN 0362-1340. doi: 10.1145/62139.62141. URL `http://doi.acm.org/10.1145/62139.62141`.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. arXiv preprint arXiv:1806.09055, 2018.

Matthew M Loper and Michael J Black. OpenDR: An approximate differentiable renderer. In European Conference on Computer Vision, pages 154–169. Springer, 2014.

Daniel Lowd and Christopher Meek. Adversarial learning. In Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pages 641–647. ACM, 2005.

David R. MacIver. Hypothesis, 2018. URL `https://github.com/HypothesisWorks/hypothesis`.

Jan R Magnus and Heinz Neudecker. Matrix differential calculus with applications in statistics and econometrics. John Wiley & Sons, 2019.

Dhruv Makwana and Neelakantan Krishnaswami. NumLin: Linear Types for Linear Algebra. In 32th European Conference on Object-Oriented Programming (ECOOP 2018), 2018.

Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pages 706–706. ACM, 2006.

Paul B Menage. Adding generic process containers to the linux kernel. In Proceedings of the Linux symposium, volume 2, pages 45–57. Citeseer, 2007.

Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239):2, 2014.

Yuri Nesterov. Gradient methods for minimizing composite functions. Mathematical Programming, 140(1):125–161, 2013.

Virginia Niculescu. A design proposal for an object oriented algebraic library. Studia Universitatis" Babes-Bolyai", Informatica, 48(1):89–100, 2003.

Virginia Niculescu. On using generics for implementing algebraic structures. Studia Universitatis Babes-Bolyai, Informatica, 56(4), 2011.

Alexander Nozik. Kotlin - new language for scientific programming. In Proceedings of 19th International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Mar 2019. URL `https://indico.cern.ch/event/708041/contributions/3276141/`.

Gerardo Pardo-Castellote. OMG Data-Distribution Service: Architectural overview. In 23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings., pages 200–206. IEEE, 2003.

Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. Software: Practice and Experience, 25(7):789–810, 1995.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In 2017 IEEE International Conference on Robotics and Automation (ICRA), pages 1497–1504. IEEE,

2017.

Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM Transactions on Programming Languages and Systems (TOPLAS), 30(2):7, 2008a.

Barak A Pearlmutter and Jeffrey Mark Siskind. Using programming language theory to make automatic differentiation sound and efficient. pages 79–90, 2008b.

Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains mps as a tool for extending java. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, pages 165–168. ACM, 2013.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In proceedings of the 26th Symposium on Operating Systems Principles, pages 1–18. ACM, 2017.

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pages 1532–1543, 2014.

Kaare Brandt Petersen et al. The Matrix Cookbook.

Gill A Pratt. Is a cambrian explosion coming for robotics? Journal of Economic Perspectives, 29(3):51–60, 2015.

Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In ICRA workshop on open source software, volume 3, page 5. Kobe, Japan, 2009.

Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in GitHub. Commun. ACM, 60(10): 91–100, September 2017. ISSN 0001-0782. doi: 10.1145/3126905. URL `http://doi.acm.org/10.1145/3126905`.

Norman A Rink. Modeling of languages for tensor manipulation. arXiv preprint arXiv:1801.08771, 2018.

Mikael Rittri. Dimension inference under polymorphic recursion. Citeseer.

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In Acm Sigplan Notices, volume 46, pages

127–136. ACM, 2010.

Frank Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6):386, 1958.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. arXiv preprint arXiv:1805.00907, 2018.

David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. Cognitive modeling, 5(3):1, 1988.

Stephen Samuel and Leonardo Colman Lopes. KotlinTest, 2018. URL `https://github.com/kotlintest/kotlintest`.

Moses Schönfinkel. Über die bausteine der mathematischen logik. Mathematische annalen, 92(3):305–316, 1924.

Claude E Shannon. Programming a computer for playing chess. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 41(314):256–275, 1950.

Jeffrey Mark Siskind and Barak A Pearlmutter. Nesting forward-mode ad in a functional framework. Higher-Order and Symbolic Computation, 21(4):361–376, 2008.

Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised 6 report on the algorithmic language Scheme. Journal of Functional Programming, 19(S1):1–301, 2009.

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In International Symposium on Practical Aspects of Declarative Languages, pages 289–303. Springer, 2012.

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In Proceedings of the 28th International Conference on Machine Learning (ICML-11), pages 609–616, 2011.

Ross Tate. Mixed-site variance. FOOL, 2013.

Eclipse Deeplearning4j Development Team. ND4J: Fast, scientific and numerical computing for the JVM. Apache Software Foundation License 2.0, 2016. URL `https://github.com/eclipse/deeplearning4j`.

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In Proceedings of the 40th international conference on software engineering, pages 303–314. ACM, 2018.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2):22, 2011.

Bart van Merriënboer, Dan Moldovan, and Alexander Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In Advances in Neural Information Processing Systems, pages 6256–6265, 2018.

Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. Software Language Engineering, SLE, 16(3), 2010.

Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In International Conference on Software Language Engineering, pages 41–61. Springer, 2014.

Tim A Wagner. Practical algorithms for incremental software development environments. PhD thesis, Citeseer, 1997.

Tim A Wagner and Susan L Graham. Incremental analysis of real programming languages. In ACM SIGPLAN Notices, volume 32, pages 31–43. ACM, 1997.

Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In Advances in Neural Information Processing Systems, pages 10180–10191, 2018a.

Fei Wang, Xilun Wu, Gregory M. Essertel, James M. Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. CoRR, abs/1803.10228, 2018b. URL `http://arxiv.org/abs/1803.10228`.

Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In Advances in neural information processing systems, pages 7675–7684, 2018c.

Richard Wei, Lane Schwartz, and Vikram Adve. DLVM: A modern compiler infrastructure for deep learning systems. arXiv preprint arXiv:1711.03016, 2017.

Robert Edwin Wengert. A simple automatic derivative evaluation program. Communications of the ACM, 7(8):463–464, 1964.

Paul J Werbos et al. Backpropagation through time: what it does and how to do it. Proceedings of the IEEE, 78(10):1550–1560, 1990.

Ruffin White and Henrik Christensen. ROS and Docker. In Robot Operating System (ROS), pages 285–307. Springer, 2017.

Norbert Wiener. Some moral and technical consequences of automation. Science, 131(3410): 1355–1358, 1960.

Virginia Vassilevska Williams. Multiplying matrices in o (n2. 373) time. 2014.

David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. IEEE transactions on evolutionary computation, 1(1):67–82, 1997.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In ACM SIGPLAN Notices, volume 33, pages 249–257. ACM, 1998.

Christoph Zenger. Indexed types. Theoretical computer science, 187(1-2):147–165, 1997.

Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. Communications of the ACM, 62(3):61–67, 2019.

Julian Zilly, Jacopo Tani, Breandan Considine, Bhairav Mehta, Andrea F Daniele, Manfred Diaz, Gianmarco Bernasconi, Claudio Ruch, Jan Hakenberg, Florian Golemo, et al. The AI driving olympics at NeurIPS 2018. arXiv preprint arXiv:1903.02503, 2019.

Appendix A

Implementation: languages and compilers

A.1. Linear Regression from an AD Perspective

Recall the matrix equation for linear regression, where $\mathbf{X} : \mathbb{R}^{m \times n}$ and $\mathbf{\Theta} : \mathbb{R}^{n \times 1}$:

$$\hat{\mathbf{f}}(\mathbf{X}; \mathbf{\Theta}) = \mathbf{X}\mathbf{\Theta} \tag{A.1.1}$$

Imagine we are given the following dataset:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} = \begin{pmatrix} 1 & \dots & x_{0n} \\ \vdots & \ddots & \vdots \\ 1 & \dots & x_{mn} \end{pmatrix}, \mathbf{Y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{A.1.2}$$

Our goal in ordinary least squares (OLS) linear regression is to minimize the loss, or error between the data and the model's prediction:

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}; \mathbf{\Theta}) = ||\mathbf{Y} - \hat{\mathbf{f}}(\mathbf{X}; \mathbf{\Theta})||^2 \tag{A.1.3}$$

$$\mathbf{\Theta}^* = \underset{\mathbf{\Theta}}{\operatorname{argmin}} \mathcal{L}(\mathbf{X}, \mathbf{Y}; \mathbf{\Theta}) \tag{A.1.4}$$

A.1.1. Finite Difference Method

First, we consider the scalar case, where $\hat{\mathbf{f}}(\mathbf{X}; \mathbf{\Theta}) = \hat{f}(x; \theta_1, \theta_0) = \theta_1 x + \theta_0$. Since $\mathbf{X}, \mathbf{Y}$ are considered to be fixed, we can rewrite $\mathcal{L}(\mathbf{X}, \mathbf{Y}; \mathbf{\Theta})$ as simply:

$$\mathcal{L}(\mathbf{\Theta}) = \mathcal{L}(\theta_1, \theta_0) = \frac{1}{m} \sum_{i=0}^{m} (y_i - (\theta_1 x_i + \theta_0))^2 \tag{A.1.5}$$

To find the minimizer of $\mathcal{L}(\mathbf{\Theta})$, we need $\nabla_{\mathbf{\Theta}} \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_0}]$. There are various ways to compute this, of which we consider two: (1) the finite difference method (FDM), and (2) symbolic

differentiation. First, let's see FDM with centered differences:

$$\frac{\partial \mathcal{L}}{\partial \theta_0} = \lim_{h\to 0} \frac{\sum_{i=0}^{m} \left(y_i - (\theta_1 x_i + \theta_0 + h)\right)^2 - \sum_{i=0}^{m} \left(y_i - (\theta_1 x_i + \theta_0 - h)\right)^2}{2hm} \tag{A.1.6}$$

$$= \lim_{h\to 0} \frac{1}{2hm} \sum_{i=0}^{m} \left(y_i - (\theta_1 x_i + \theta_0 + h)\right)^2 - \left(y_i - (\theta_1 x_i + \theta_0 - h)\right)^2 \tag{A.1.7}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \lim_{h\to 0} \frac{\sum_{i=0}^{m} \left(y_i - ((\theta_1 + h) x_i + \theta_0)\right)^2 - \sum_{i=0}^{m} \left(y_i - ((\theta_1 - h)\, x_i + \theta_0)\right)^2}{2hm} \tag{A.1.8}$$

$$= \lim_{h\to 0} \frac{1}{2hm} \sum_{i=0}^{m} \left(y_i - ((\theta_1 + h)\, x_i + \theta_0)\right)^2 - \left(y_i - ((\theta_1 - h)\, x_i + \theta_0)\right)^2 \tag{A.1.9}$$

Using computer algebra, these equations can be simplified considerably:

$$\frac{\partial \mathcal{L}}{\partial \theta_0} = \lim_{h\to 0} \frac{1}{2hm} \sum_{i=0}^{m} \left(4h(\theta_0 + \theta_1 x_i - y_i)\right) \tag{A.1.10}$$

$$= \boxed{\frac{2}{m} \sum_{i=0}^{m} \left(\theta_0 + \theta_1 x_i - y_i\right)} \tag{A.1.11}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \lim_{h\to 0} \frac{1}{2hm} \sum_{i=0}^{m} \left(4hx_i(\theta_1 x_i + \theta_0 - y_i)\right) \tag{A.1.12}$$

$$= \boxed{\frac{2}{m} \sum_{i=0}^{m} (x_i)(\theta_1 x_i + \theta_0 - y_i)} \tag{A.1.13}$$

(1) https://www.wolframalpha.com/input/?i=(y_i-((%CE%B8_1%2Bh)x_i%2B%CE%B8_0))%5E2-(y_i-((%CE%B8_1-h)x_i%2B%CE%B8_0))%5E2

(2) https://www.wolframalpha.com/input/?i=(y_i-(%CE%B8_1*x_i%2B%CE%B8_0%2Bh))%5E2%E2%88%92(y_i-(%CE%B8_1*x_i%2B%CE%B8_0-h))%5E2

A.1.2. Partial Differentiation

Alternatively, we can calculate the partials analytically, by applying the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta_0} = \frac{\partial}{\partial \theta_0} \frac{1}{m} \sum_{i=0}^{m} (y_i - (\theta_1 x_i + \theta_0))^2 \tag{A.1.14}$$

$$= \frac{1}{m} \sum_{i=0}^{m} 2(y_i - (\theta_1 x_i + \theta_0)) \frac{\partial}{\partial \theta_0} (y_i - (\theta_1 x_i + \theta_0)) \tag{A.1.15}$$

$$= \frac{2}{m} \sum_{i=0}^{m} (y_i - (\theta_1 x_i + \theta_0))(-1) \tag{A.1.16}$$

$$= \boxed{\frac{2}{m} \sum_{i=0}^{m} (\theta_1 x_i + \theta_0 - y_i)} \tag{A.1.17}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial}{\partial \theta_1} \frac{1}{m} \sum_{i=0}^{m} (y_i - (\theta_1 x_i + \theta_0))^2 \tag{A.1.18}$$

$$= \frac{1}{m} \sum_{i=0}^{m} 2(y_i - (\theta_1 x_i + \theta_0)) \frac{\partial}{\partial \theta_1} (y_i - (\theta_1 x_i + \theta_0)) \tag{A.1.19}$$

$$= \frac{2}{m} \sum_{i=0}^{m} (y_i - (\theta_1 x_i + \theta_0))(-x_i) \tag{A.1.20}$$

$$= \boxed{\frac{2}{m} \sum_{i=0}^{m} (x_i)(\theta_1 x_i + \theta_0 - y_i)} \tag{A.1.21}$$

Notice how analytical differentiation gives us the same answer as the finite difference method (this is not by accident), with much less algebra. We can rewrite these two solutions in gradient form, i.e. as a column vector of partial derivatives:

$$\nabla_{\boldsymbol{\Theta}} \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_0} \\ \frac{\partial \mathcal{L}}{\partial \theta_1} \end{bmatrix} = \frac{2}{m} \begin{bmatrix} \sum_{i=0}^{m} (\theta_1 x_i + \theta_0 - y_i) \\ \sum_{i=0}^{m} (x_i)(\theta_1 x_i + \theta_0 - y_i) \end{bmatrix} \tag{A.1.22}$$

A.1.3. Matrix Solution

Having reviewed the scalar procedure for linear regression, let us now return to the general form of $\mathcal{L}(\boldsymbol{\Theta})$. Matrix notation allows us to simplify the loss considerably:

$$\mathcal{L}(\boldsymbol{\Theta}) = \frac{1}{m}(\mathbf{Y} - \mathbf{X}\boldsymbol{\Theta})^\intercal(\mathbf{Y} - \mathbf{X}\boldsymbol{\Theta}) \tag{A.1.23}$$

$$= \frac{1}{m}(\mathbf{Y}^\intercal\mathbf{Y} - \mathbf{Y}^\intercal\mathbf{X}\boldsymbol{\Theta} - \boldsymbol{\Theta}^\intercal\mathbf{X}^\intercal\mathbf{Y} + \boldsymbol{\Theta}^\intercal\mathbf{X}^\intercal\mathbf{X}\boldsymbol{\Theta}) \tag{A.1.24}$$

$$= \frac{1}{m}(\mathbf{Y}^\intercal\mathbf{Y} - 2\boldsymbol{\Theta}^\intercal\mathbf{X}^\intercal\mathbf{Y} + \boldsymbol{\Theta}^\intercal\mathbf{X}^\intercal\mathbf{X}\boldsymbol{\Theta}) \tag{A.1.25}$$

Matrix notation allows us to derive the gradient and requires far less algebra:

$$\nabla_{\boldsymbol{\Theta}}\mathcal{L}(\boldsymbol{\Theta}) = \frac{1}{m}(\nabla_{\boldsymbol{\Theta}}\mathbf{Y}^\intercal\mathbf{Y} - 2\nabla_{\boldsymbol{\Theta}}\boldsymbol{\Theta}^\intercal\mathbf{X}^\intercal\mathbf{Y} + \nabla_{\boldsymbol{\Theta}}\boldsymbol{\Theta}^\intercal\mathbf{X}^\intercal\mathbf{X}\boldsymbol{\Theta}) \tag{A.1.26}$$

$$= \frac{1}{m}(0 - 2\mathbf{X}^\intercal\mathbf{Y} + 2\mathbf{X}^\intercal\mathbf{X}\boldsymbol{\Theta}) \tag{A.1.27}$$

$$= \boxed{\frac{2}{m}(\mathbf{X}^\intercal\mathbf{X}\boldsymbol{\Theta} - \mathbf{X}^\intercal\mathbf{Y})} \tag{A.1.28}$$

For completeness, and to convince ourselves the matrix solution is indeed the same:

$$= \frac{2}{m}\left(\underbrace{\begin{bmatrix}1 & \dots & 1\\ x_0 & \dots & x_m\end{bmatrix}}_{\mathbf{X}^\intercal}\underbrace{\begin{bmatrix}1 & x_0\\ \vdots & \vdots\\ 1 & x_m\end{bmatrix}}_{\mathbf{X}}\underbrace{\begin{bmatrix}\theta_0\\ \theta_1\end{bmatrix}}_{\boldsymbol{\Theta}} - \underbrace{\begin{bmatrix}1 & \dots & 1\\ x_0 & \dots & x_m\end{bmatrix}}_{\mathbf{X}^\intercal}\underbrace{\begin{bmatrix}y_0\\ \vdots\\ y_m\end{bmatrix}}_{\mathbf{Y}}\right) \tag{A.1.29}$$

$$= \frac{2}{m}\left(\underbrace{\begin{bmatrix}m & \sum_{i=0}^m x_i\\ \sum_{i=0}^m x_i & \sum_{i=0}^m x_i^2\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{X}}\underbrace{\begin{bmatrix}\theta_0\\ \theta_1\end{bmatrix}}_{\boldsymbol{\Theta}} - \underbrace{\begin{bmatrix}\sum_{i=0}^m y_i\\ \sum_{i=0}^m x_i y_i\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{Y}}\right) \tag{A.1.30}$$

$$= \frac{2}{m}\left(\underbrace{\begin{bmatrix}m\theta_0 + \sum_{i=0}^m \theta_1 x_i\\ \sum_{i=0}^m \theta_0 x_i + \sum_{i=0}^m \theta_1 x_i^2\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{X}\boldsymbol{\Theta}} - \underbrace{\begin{bmatrix}\sum_{i=0}^m y_i\\ \sum_{i=0}^m x_i y_i\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{Y}}\right) \tag{A.1.31}$$

$$= \boxed{\frac{2}{m}\underbrace{\begin{bmatrix}\sum_{i=0}^m \theta_1 x_i + \theta_0 - y_i\\ \sum_{i=0}^m (x_i)(\theta_1 x_i + \theta_0 - y_i)\end{bmatrix}}_{\mathbf{X}^\intercal\mathbf{X}\boldsymbol{\Theta} - \mathbf{X}^\intercal\mathbf{Y}} = \begin{bmatrix}\frac{\partial\mathcal{L}}{\partial\theta_0}\\ \frac{\partial\mathcal{L}}{\partial\theta_1}\end{bmatrix} = \nabla_{\boldsymbol{\Theta}}\mathcal{L}(\boldsymbol{\Theta})} \tag{A.1.32}$$

Notice how we recover the same solution obtained from partial differentiation and finite difference approximation, albeit in a more compact form. For a good introduction to matrix calculus, the textbook by Magnus and Neudecker [2019] is an excellent guide, of which Petersen et al. offer a review of important identities.

OLS linear regression is a convex optimization problem. If $\mathbf{X}^\intercal\mathbf{X}$ is invertible, i.e. full-rank, this implies a unique solution $\mathbf{\Theta}^*$, which we can solve for directly by setting $\nabla_{\mathbf{\Theta}}\mathcal{L} = \mathbf{0}$:

$$0 = \mathbf{X}^\intercal\mathbf{X}\mathbf{\Theta} - \mathbf{X}^\intercal\mathbf{Y} \tag{A.1.33}$$

$$\mathbf{\Theta} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{Y} \tag{A.1.34}$$

Solving this requires computing $(\mathbf{X}^\intercal\mathbf{X})^{-1}$ which is at least $\mathcal{O}(n^{2.373})$[Williams, 2014] to the best of our knowledge, i.e. quadratic with respect to the number of input dimensions. Another way to find $\mathbf{\Theta}^*$ is by initializing $\mathbf{\Theta} \leftarrow \mathbf{0}$ and repeating the following procedure until convergence:

$$\mathbf{\Theta}' \leftarrow \mathbf{\Theta} - \alpha\nabla_{\mathbf{\Theta}}\mathcal{L}(\mathbf{\Theta}) \tag{A.1.35}$$

Typically, $\alpha \in [0.001, 0.1]$. Although hyperparameter tuning is required to find a suitable α (various improvements like Nesterov momentum [Nesterov, 2013] and quasi-Newton methods also help to accelerate convergence), this procedure is guaranteed to be computationally more efficient than matrix inversion for sufficiently large m and n. In practice, the normal equation is seldom used unless m is very small.