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

mai 2019

Summary

Table des matières

Liste des tableaux

Liste des figures

Chapitre 1

Introduction

> Intelligent system: *A computer system that uses techniques derived from artificial intelligence, particularly one in which such techniques are central to the operation of the system.*
>
> –Wikipedia

Computational complexity is of such concern in computer science that a great deal of the field is dedicated to understanding it using tools from 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 type of software complexity is the cognitive effort required to understand a program's source code, which can be approximated by metrics such as cyclomatic or Halstead complexity. While today's software is more intelligent than ever before, it shows few signs of becoming more intelligible, and better tools are needed for managing the complexity inherent in building it.

The objective of writing 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 container-based virtualization.

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 capable of:

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

(2) tuning a large number of internal parameters (thousands to billions)

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

While the idea of intelligent systems has been around for decades, three critical developments have made modern intelligent systems possible. 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/mo. 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 [**3**], Torch [**18**] and Caffe [**41**], and built many popular simulators for machine learning.

In this thesis, we explore various tools that facilitate the process of programming 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 [**27**] and adversarial learning [**51**] (Chapter 4). Docker containers [**54**] 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 developed (Chapter 6).

1.1. Stages in the software development lifecycle

In traditional software engineering, the Waterfall Model 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.

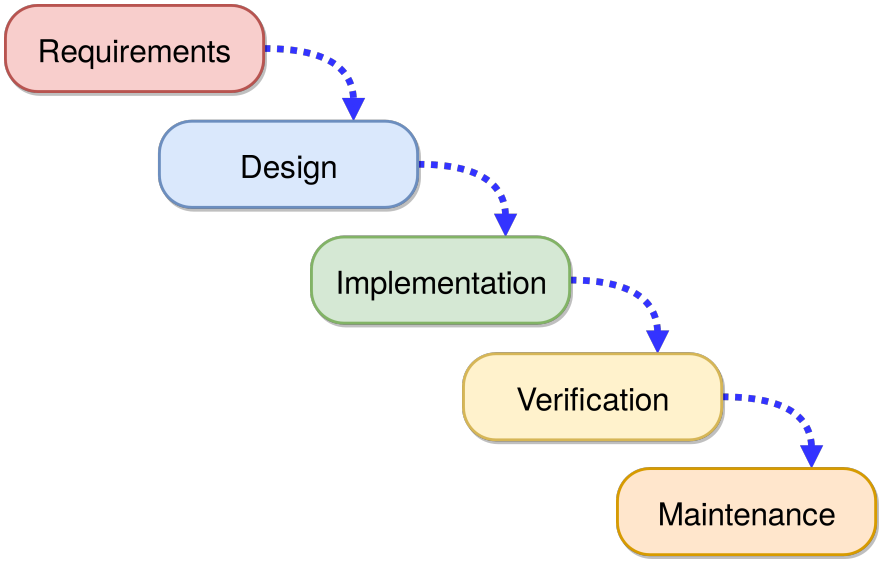

Fig. 1.1. Royce's original Waterfall model, originally intended to describe the software development process, but the same sequence can be found in most engineering disciplines. We use it to help guide our discussion and frame our work inside of this process model.

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, can maintain a large software system alone. To effectively scale 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* [**19**] - large portions of the knowledge graph are locked inside someone's head. Successful software projects learn how to distribute this graph and form new connections to the outside world. The knowledge graph which accumulates around a large software project contains facts, but it also contains workflows for programming, debugging, and delivery - all well-traveled paths through the labyrinth of software development. Components of this graph can be committed to writing in the form of documentation, but documentation is time-consuming and grows stale over time. What is needed is a system that preserves 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 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 work together 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, automating frequent tasks in software development, these mechanical tools enables them to focus on the fundamentals 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 **[31]** and projectional editing **[73]**, a well-designed IDE is a tool for graph traversal. It should come as no surprise IDEs are really graph databases.

In some aspects, the development of intelligent systems is no different than classical software engineering. While the former requires more human intervention, the same principles and best-practices which guide software engineering are also applicable to intelligent systems. And the same activities, from analysis, design, implementation, verification and maintenance will continue to play an important role in building intelligent systems. But in other aspects, the generic programming tools we use for developing traditional software will require domain-specific adaptations for machine learning to become a truly first class citizen in the next generation of software systems, particularly in the case of physically embodied agents. Towards that end, we propose Hatchery, an IDE for the Robot Operating System (ROS), a popular robotics middleware.

1.3. Implementation: Languages and compilers

In the early days of machine learning, it was widely believed the development of human-level intelligence simply required 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. 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 A.I. research for a number of decades. While they did not appreciate the difficulty of learning and were largely unsuccessful, expert systems excelled in areas where current machine learning systems struggle such as reasoning and interpretability, and there is reason to believe these ideas were ahead of their time.

What did finally work, is the idea of connectionist learning. By nesting random function approximators, called *artificial neural networks* (ANNs), and training the system using backpropagation [**80, 66**], the resulting system is capable of learning a surprising amount of intelligent behavior. The idea of neural network can be traced back to the mid-20th century [**39, 65**], but was not fully-realized in silico until after the widespread availability of cheap computing and large datasets [**46**]. In theory, a single layer of nesting is able to approximate any continuous differentiable function [**37**], 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 on DNNs due to the vanishing gradient problem [**9**]. 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 at least partly attributed to software tools for calculating derivatives, which are essential for the backpropagation. Although it has 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 had to be calculated manually when prototyping a new neural network architecture, a time-consuming and error-prone process. There is a well-known algorithm in the scientific computing community, called *automatic differentiation* (A.D.) [**48, 33**], which is able to calculate derivatives for arbitrary differentiable functions. But surprisingly, it was not until much later, after the development of *Theano* [**3**] 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 [**1**] and PyTorch [**58**].

Intelligent systems engineers must think carefully about languages and abstractions. If developers are required to implement backpropogation by hand, they will have little time to think about the high-level characteristics of these systems. Similarly, if the abstractions we use have too many assumptions baked in, small variations will require costly reimplementation. This is no different from traditional software engineering - as engineers, 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 connection of between source code and runtime behavior is already quite difficult to grasp, due to the nature of neural networks.

1.4. Testing: Verification 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 [**67**]. At the time, it was believed that such problems would be insurmountable without a fundamental change in algorithms or 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 approximated in the Cambrian explosion 541 million years ago, were solved to a competitive margin in modern computers.

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 disperse. Because the dimensionality of many real world problems we would like to solve is intractably large, it is difficult to be certain about the behavior of a candidate solution in all regimes. According to some studies, a human driver averages 1.09 fatalities per hundred million miles [**42**]. 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 driver 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, 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 is 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.

1.5. Software reproducibility and maitenance

One of the challenges of building intelligent systems and programming in general, is the problem of reproducibility. Software reproducibility has a number of 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 sufficient, the source code of modern program are usually far too removed from its mechanical implementation to be meaningfully executed in isolation. Today's handwritten programs are like schematics for a traffic light - built in a factory, and which require a city's-worth of infrastructure, cars, and traffic laws. 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 certain 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 long since been automated and 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. And for most programs, such as those found in a typical GitHub repository, a vanishingly small fraction correspond to the handwritten program 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 [**49**]) 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 produce different results on different hardware. Complicating the problem, IRs are produced by competing vendors, with competing chips and incompatible standards (e.g. MLIR, ONNX, NNEF, OpenVINO, et al.). While some have tried to leverage existing compilers such as GHC [**25**] or LLVM [**78**], there are few signs of convergence.

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 other researchers, it would be convenient if there were tools for reproducibility and incremental development. Fortunately, this is the same problem software developers have been attempting to solve for many years, via the open source community. But source control management (SCM) alone is insufficient, since SCM tools are primarily intended for text. While text-based representations may be stable for a time, 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. Short of that, the minimal set of dependencies for running a program is essential.

In order to mitigate the effects of software variability and assist the development of intelligent systems on heterogeneous platforms, we turn to 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 software on another computer. Docker itself is a technical solution, but it also encompasses a growing set of best-practices which are more organizational in nature.

While this does not address the incompatibility of vendor standards and hardware drivers, it makes these variables explicit, and reduces the associated difficulty of reproducing software artifacts.

There is a second component to software reproducibility of intelligent systems, at the boundary of software and hardware. While today's simulators have become increasingly realistic, 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 subtleties of the real world, and must never be used as a surrogate for training on real-world data. Others have suggested a middle road [**12**], where judicious use of simulator training, alongside domain adaption is a sufficiently rigorous setting for training intelligent systems. Regardless of which view prevails, our goal is to provide rapid feedback to developers, and make this process as reproducible as possible.

1.5.1. Case Study

All great software has a secret recipe: software gets better when engineers use the product. In the best case, software engineers 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 engineers who are so invested, or to build software which is so useful, but there must be some overlap in order for good software to become great. Termed "dogfooding" [**36**], this practice has been used for public relations, but is an effective mechanism of building for self-improving cybernetic systems. It is also an important principle for open source software and safety-critical systems.

Putting this principle into practice, as the authors and primary users of these tools, we validate their effectiveness by developing a robotics application within the IDE (Chapter 2), containing Kotlin∇ code (Chapter 3), and which is is built using the Docker stack (Chapter 5).

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, *Man-Computer Symbiosis* [**47**]

In this chapter we will discuss the design and implementation of an integrated development environment (IDE) for writing robotic applications. Modern robots are increasingly powered by systems which learn and improve over time. Today, most robotic systems are not high on the scale of general intelligence, and many artificially intelligent systems lack any physical control. However we take the somewhat liberal view that any closed-loop control system (e.g. a thermometer) is an intelligent system.

Our tool, called Hatchery, is designed to assist programmers writing robotics applications with the ROS middleware. At the time of its release, Hatchery was the first ROS plugin for the IntelliJ Platform, a popular IDE for C/C++, Python and Android, and a year later, is also the most widely used plugin with close to 10,000 downloads. While the idea is simple, its prior absence and subsequent adoption suggests these kinds of tools fill a much-needed gap in the development of programming tools for embedded systems, particularly for robotics.

2.1. Software architecture for robotics application

The Robot Operating System (ROS) is a popular middleware for robotics applications. At its core, ROS provides software infrastructure for distributed messaging, but also encompasses a set of community-developed libraries and graphical tools for building robotics applications. While ROS is not an operating system (OS) in the traditional sense, it does

implement specific OS-like features like shared memory and inter-process communication for robotics development. Unlike pure message-oriented middleware like DDS and ZMQ, in addition to the message broker, ROS provides specific features for building decentralized robotic systems, particularly those which are capable of mobility.

According to one community census in 2018, 55% of ROS applications on GitHub are written in C/C++, followed by Python at around 25% [**35**].

2.2. Foundations of a modern IDE

To build an IDE, one should have a few prerequisites. First, is an IDE, which we shall refer to as IDE_0, and its source code. Assume that IDE_0 exists. In order to build a new IDE, IDE_1, first load the source code from IDE_0 into IDE_0. Then, using IDE_0, modify and compile the source code. Finally, run the compiled artifact, which becomes IDE_1. Iterate as necessary.

While valid, this approach has some disadvantages. First, most IDEs are already quite large and take a long time for users to download, install, compile, and run. Since most new functionality is small by comparison, modern IDEs have adopted a modular design, which allows them to load certain packages (i.e. *plugins*), as needed. So most developers can skip the first step, and load such a plugin, using IDE_0 directly. It is still convenient to have the platform source code for reference purposes, but in most cases this source code is read-only.

2.2.1. The parser

An IDE consists of a few important components. First is a parser. This is typically not an ordinary parser, because most of the time users are modifying a program, their code is invalid. Even when the source code is invalid, the parser needs guide the user towards a functioning program. So this parser needs to be able to recover from syntactical errors.

We can parse URDF, package and launch XML, and srv files.

2.2.2. Refactoring

Refactoring support is implemented.

2.2.3. Running and debugging

Assistance for running ROS applications.

2.3. More ROS Tools

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* [**16**]

In this chapter, we will discuss the theory and implementation of a type safe domain specific language for automatic differentiation (AD), which has a variety of applications in numerical optimization and machine learning. The key idea behind AD is fairly simple. A small set of primitive 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, it is often the case we are given a computable function, in the form of a program[1], that 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, the seed of such an algorithm was first conceived by Robert Wengert [**79**]. This 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, and recognized its potential for, "the treatment of large systems of differential equations which might not otherwise be undertaken" [**8**]. Around the same time, key details of the backpropogation algorithm first emerged [**23**]. In 1970, Seppo Linnainmaa [**48**], conceived the idea of calculating derivatives over computation graphs. Linnaimaa's algorithm was particularly important for the development of neural networks, and is today known as reverse-mode AD. But it was not until 2010 when standard software tools [**10, 17**] for AD became widely available in machine learning. It is here where our journey begins.

[1]Not all programs are computable functions, but all computable functions are programs.

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 $p_n : \mathbb{R} \to \mathbb{R}$, composed of a series of nested functions:

$$p_n(p_0) = p_{n-1} \circ p_{n-2} \circ ... \circ p_1 \circ p_0 \tag{3.1.1}$$

From the chain rule of calculus, we recall that:

$$\frac{dp_n}{dp_0} = \prod_{i=1}^{n} \frac{dp_i}{dp_{i-1}} \tag{3.1.2}$$

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

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

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

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

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

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

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

$$\mathbf{H}(\mathbf{f}) = [\mathbf{H}(f_1), \mathbf{H}(f_2), \ldots, \mathbf{H}(f_m)] \tag{3.1.6}$$

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 a is changed, the output stays the same. We say that such functions have zero gradient for that input.

$$(\nabla F)(a) \approx \mathbf{0} \tag{3.1.7}$$

The cost of calculating the Hessian, $\mathbf{H}$ is approximately quadratic **[32]** 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.

$$||\nabla F|| \approx \pm\infty \tag{3.1.8}$$

It is an open question whether non-differentiable functions exist in the real world **[13]**. At the current physical (10nm) and temporal (10ns) scale of modern computing, there exist no such functions, but modern computers are not equipped with the capability to accurately report the true value of their binary-valued functions, so for all intents and purposes, programs implemented by most physical computers are discrete relations. Nevertheless, computers are capable of approximating bounded functions of R^n to arbitrary precision given enough time and space. For most applications, a fixed precision 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 $\varphi : R \to R$ 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 **[37]** states that for all real-valued continuous functions $f : C(\mathbb{I}_m)$, where $\mathbb{I}_m = [0, 1]^m$, there exists a function $\hat{f}$, parameterized by constants $n \in \mathbb{N}, \beta \in \mathbb{R}^n, b \in \mathbb{R}^n, \epsilon \in \mathbb{R}^+$ and $\theta \in \mathbb{R}^{m \times n}$:

$$\begin{aligned} \hat{f}(x) &= \beta^\intercal \varphi\left(\theta^\intercal x + b\right) \\ \forall x \in I_m,\ &|\hat{f}(x) - f(x)| < \varepsilon \end{aligned} \tag{3.1.9}$$

This theorem does not put an upper bound on the constant n, or tell us how to find θ, somewhat limiting its practical applicability. But for reasons not yet fully understood, empirical results suggest it is possible to obtain accurate approximations to many naturally-arising functions in a relatively short time by composing these linear and non-linear functions in an alternating fashion and iteratively updating θ in the direction suggested by $\nabla_\theta \mathcal{L}(\hat{f}(x;\theta), f(x))$. In this setting, we do not have access to f directly, but receive samples $Y = [f(x^{(1)}), \ldots, f(x^{(k)})]$, which we try to obtain in as large a quantity as possible and run the following procedure:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\hat{f}(X;\theta), Y) \tag{3.1.10}$$

For most common $\mathcal{L}$, the complexity of this procedure is linear with k. Since k tends to be large, a single iteration of this procedure can take a long time on a computer. Since obtaining the exact gradient is not important in practice, we use a stochastic variant by sampling a *minibatch* of rows X', Y', from X, Y, which 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 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.

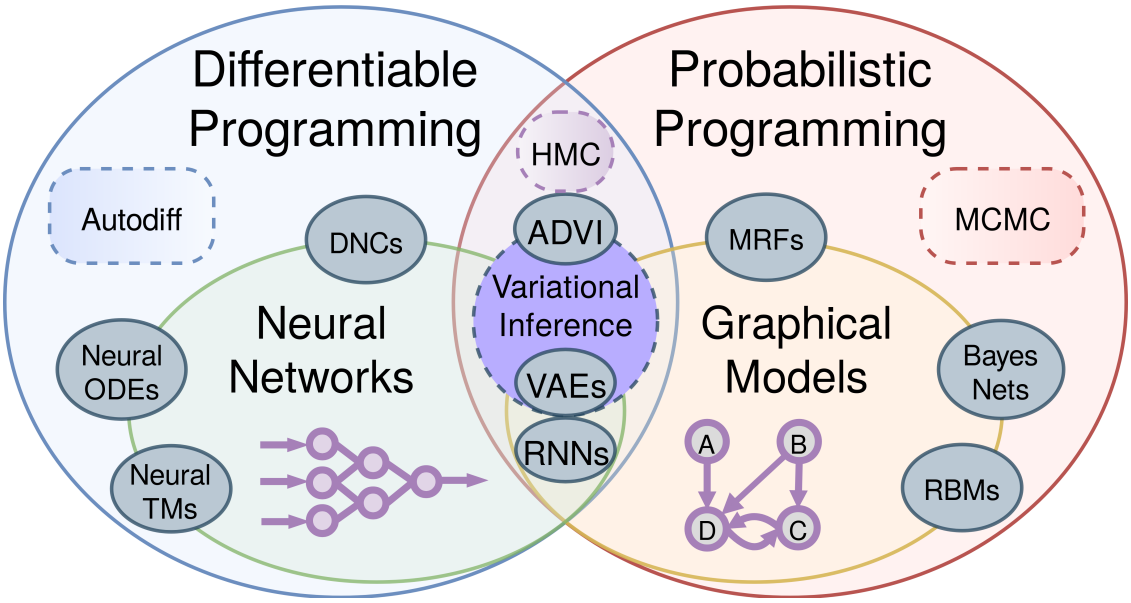

Fig. 3.1. *Differentiable programming* includes neural networks, but more broadly, arbitrary differentiable programs which use automatic differentiation and gradient-based optimization to approximate a loss function. *Probabilistic programming* is a generalization of probabilistic graphical models, and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

A key aspect of the connectionist paradigm is learning neural representations via gradient descent. According to theory, 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 deep 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 define a mapping from the textual domain to their vector-based 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 **[62]**. It so happens that many classes of discrete problems can be relaxed to continuous surrogates by using a good representation.

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 [**77**] and integer [**40**] quantization are able to train neural networks without sacrificing performance. Strong assumptions like Lipschitz-continuity and -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 not have come as a big surprise, 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, neural network training 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* was born. Today, differentiable programing has found a wide range of applications, from protein folding [**4**], to physics engines [**20, 21**] and graphics rendering [**50**] to meta-learning [**49**]. 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 suitable metric, differentiable programming promises to do this for a broad class of applications, more or less automatically.

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 [**63**].

Dynamically typed languages are commonly used for experimentation and prototyping. But are they scalable to production systems?

According to some studies, type errors account for over 15% of bugs [**29**]. While the causality between statically typed languages and fewer bugs is not widely established, types are necessary to build more powerful static analyses and tools for program understanding.

Strong, static types are important for reasoning about the behavior of complex programs. Statically typed languages offer a number of benefits to users, such as eliminating a broad class of runtime errors by virtue of the type system alone. Furthermore, a well-typed API can eliminate the potential for incorrect API usages by enforcing certain usage patterns.

Statically typed languages also provide several benefits for static code analysis, as tools can offer more relevant autocompletion suggestions, and provide early warnings for compile and probable runtime errors.

3.4. Imperative and functional languages

Most programs written today are written in the imperative style, due in part to the prevalence of the Turing Machine and von Neumann architecture. λ-calculus provides an equivalent[2] language for computing, which we argue, is a more natural way to express mathematical functions and calculate their derivatives. In pure imperative programming the sole purpose of a function is to pass it values, but we have no way to refer to the function itself, for example, to write a function which transforms another function. More troubling in the case of automatic differentiation, is that imperative programs have mutable state, which is not the case in mathematics, and which requires taking extra precautions when calculating mathematical derivatives.

In functional programming, function composition is a very natural pattern. To take the derivative of a function composed by another function, we simply apply the chain rule 3.1. Since there is no mutable state, we do not require any exotic data structures or compiler tricks to keep track of the state.

For example, consider the vector function $f(l_1, l_2) = l_1 \cdot l_2$, seen in 3.1. Imperative programs, by allowing mutation, are destroying information. In order to recover that computation graph for reverse mode AD, we need to either override the assignment operator, or use a tape to store the intermediate values, which is quite tedious. In pure functional programming mutable variables do not exist, which makes our lives much easier.

3.5. Kotlin

Kotlin is a strong, statically typed language, that is well-suited for building cross-platform applications, with implementations in native, JVM, and JavaScript.

When programming in a statically typed language, one common question is, "Given a value, `x`, can `x` be assigned to a variable of type `Y`?" (e.g. `x instanceof Y`) In Java, this question is both unsound [**5**] and undecidable [**34**] in the general case. It is possible to

[2] In the sense that λ-calculus is Turing Complete.

	Imperative	Functional
1	`fun dot(l1, l2) {`	`fun dot(l1, l2) {`
2	`  if (len(l1) != len(l2))`	`  return if (len(l1) != len(l2))`
3	`    return error`	`    error`
4	`  var sum = 0`	`  else if (len(l1) == 0) 0`
5	`  for(i in 0 to len(l1))`	`  else`
6	`    sum += l1[i] * l2[i]`	`    head(l1) * head(l2) +`
7	`  return sum`	`    dot(tail(l1) + tail(l2))`
8	`}`	`}`

Tab. 3.1. Two programs, implementing the function $f(l_1, l_2) = l_1 \cdot l_2$.

construct a Java program for which the answer is "yes" regardless of `Y`, or which the answer cannot be determined in a finite amount of time.

Kotlin's type system [**71**] is strictly less expressive, but fully compatible with Java.

3.6. Kotlin∇

Prior work has shown it is possible to encode a deterministic context-free grammar as a *fluent interface* [**30**] in Java. This result was strengthened to prove Java's type system is Turing complete [**34**]. 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 [**3**], Torch [**18**], and TensorFlow [**1**]. Similar ideas have been implemented in statically typed, functional languages, such as Haskell's Stalin∇ [**61**], DiffSharp in F# [**7**] and recently Swift [**44**]. However, the majority of existing automatic differentiation (AD) frameworks 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 [**76**], Nexus [**15**] and DeepLearning.scala [**11**], 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 contains a number of desirable features for implementing a native

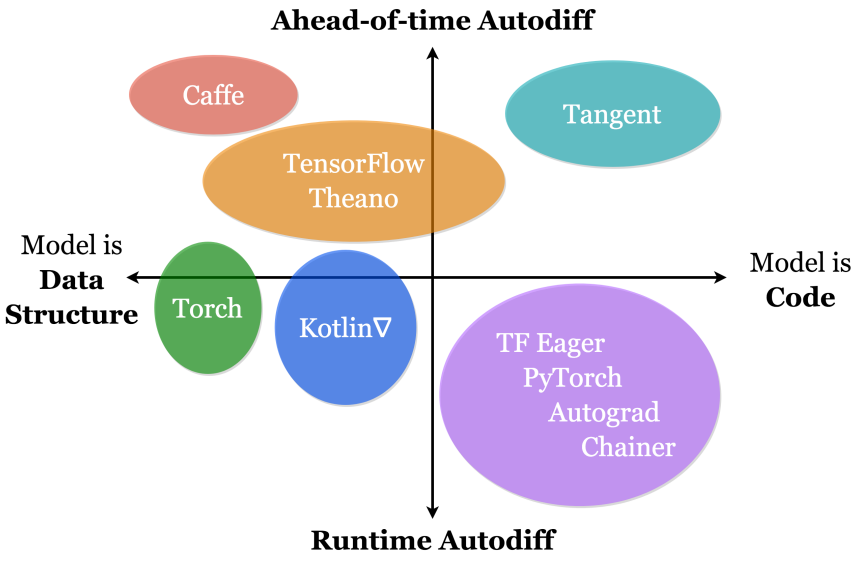

Fig. 3.2. Adapted from [**72**]. Kotlin∇ models are data structures, evaluated at runtime.

AD framework. In addition to type-safety and interoperability, 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 and coroutines** support functional differentiable programming with lambdas and shift-reset continuations, following [**60**] and [**76**].
- **Extension functions** support extending classes with new fields and methods and 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 just functions building an abstract syntax tree (AST), which can be evaluated eagerly or lazily depending on the developer's needs. Typically these ASTs represent simple state machines, but they can also be used to implement a full-fledged programming language. Popular examples include SQL/LINQ [**52**], OptiML [**70**] and other fluent interfaces [**28**]. 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 a carefully designed type system, users will automatically receive code completion and static analysis from their favorite developer tools. Many have observed that functional programming languages are

suitable host languages [**26, 64**], perhaps due to the concept of code as data and data as code[3].

3.7. Usage

Kotlin∇ allows users to implement differentiable programs by composing operations on numerical fields to form algebraic expressions. 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.

Listing 3.1. A basic Kotlin∇ program with two inputs and one output.

```
1  import edu.umontreal.kotlingrad.numerics.DoublePrecision
2
3  with(DoublePrecision) { // Use double-precision protocol
4      val x = variable("x") // Declare immutable vars (these
5      val y = variable("y") // are just symbolic constructs)
6      val z = sin(10 * (x * x + pow(y, 2))) / 10 // Lazy exp
7      val dz_dx = d(z) / d(x) // Leibniz derivative notation
8      val d2z_dxdy = d(dz_dx) / d(y) // Mixed higher partial
9      val d3z_d2xdy = grad(d2z_dxdy)[x] // Indexing gradient
10     plot3D(d3z_d2xdy, -1.0, 1.0) // Plot in -1 < x,y,z < 1
11 }
```

In Listing 3.1, we define a function with two variables and take a series of partial derivatives with respect to each variable. The function is numerically evaluated on the interval $(-1, 1)$ in each dimension and rendered in 3-space. We can also plot higher dimensional manifolds (e.g. the loss surface of a neural network), projected into four dimensions, and rendered in three, where one axis is represented by time.

Kotlin∇ treats mathematical functions and programming functions with the same underlying abstraction. Expressions are composed to form a data-flow graph (DFG). An expression is simply a `Function`, which is only evaluated once invoked with numerical values, e.g. `z(0, 0)`. In this way, Kotlin∇ is similar to other compiled graph based approaches like TensorFlow and Theano.

[3] i.e. homoiconicity, notably introduced by Lisp, one of the first functional programming languages

$$z = \sin\big(10(x * x + y^2)\big)/10,\ \texttt{plot}(\tfrac{\partial^3 z}{\partial x^2 \partial y})$$

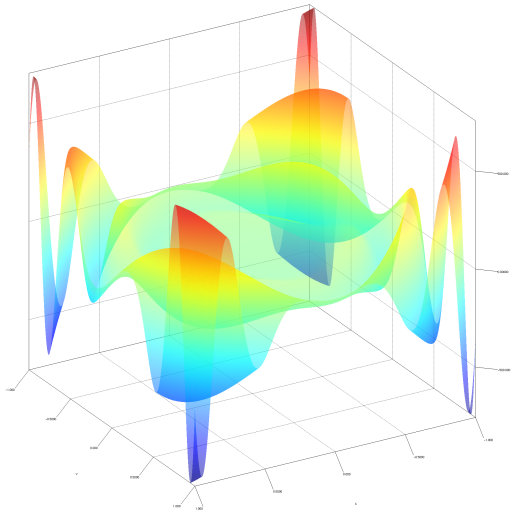

Fig. 3.3. Output generated by the program shown in Figure ??.

Listing 3.2. The equation below is an EDSL, and does not perform any computation.

```
1 val z = sin(10 * (x * x + pow(y, 2))) / 10
```

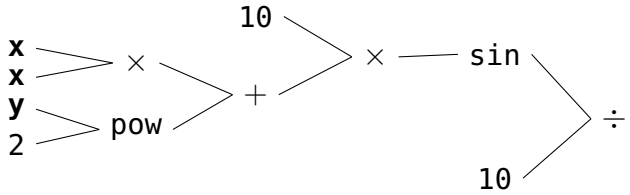

Fig. 3.4. Implicit DFG constructed by the original expression, `z`.

Kotlin∇ supports shape-safe tensor operations by encoding tensor rank as a parameter of the operand's type signature. By enumerating type-level integer literals, we can define tensor operations just once using the highest literal, and rely on Liskov substitution to preserve shape safety for subtypes.

Listing 3.3. Shape safe tensor addition for rank-1 tensors, $\forall L \leq 2$.

```
1 // Literals have reified values for runtime comparison
2 open class ‘0‘(override val value: Int = 0): ‘1‘(0)
3 open class ‘1‘(override val value: Int = 1): ‘2‘(1)
4 class ‘2‘(open val value: Int = 2) // Greatest literal
```

```
// <L: '2'> will accept L <= 2 via Liskov substitution
class Vec<E, L: '2'>(len: L, cts: List<E> = listOf())
// Define addition for two vectors of type Vec<Int, L>
operator fun <L: '2', V: Vec<Int, L>> V.plus(v: V) =
Vec<Int, L>(len, cts.zip(v.cts).map { it.l + it.r })
// Type-checked vector addition with shape inference
val Y = Vec('2', listOf(1, 2)) + Vec('2', listOf(3, 4))
val X = Vec('1', listOf(1, 2)) + Vec('3') // Undefined!
```

It is possible to enforce shape-safe vector construction as well as checked vector arithmetic up to a fixed `L`, but the full implementation is omitted for brevity. This pattern can also be applied to matrices and tensors, where the type signature encodes the shape of the operand at runtime.

With these basic ingredients, we have almost all the features necessary to build an expressive shape-safe AD, but unlike prior implementations using Scala or Haskell, in a language fully interoperable with Java, and which compiles to JVM bytecode, JavaScript, and native code.

3.8. Type system

Describing the Kotlin∇ type system (formally).

3.9. Testing

Kotlin∇ claims to eliminate certain runtime errors, but how do we know the proposed implementation is not incorrect? One method, borrowed from the Haskell community, is called property-based testing (PBT), closely related to metamorphic testing. Notable implementations include QuickCheck, Hypothesis and ScalaTest (ported to Kotlin in KotlinTest). 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) **Symbolic differentiation**: manually differentiate and compare the values returned on a subset of the domain with AD.

(2) **Finite difference approximation**: sample space of symbolic (differentiable) functions, comparing results of AD to FD.

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

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

For example, the following test checks whether the manual derivative and the automatic derivative, when evaluated at a given point, 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)
        // Should pass iff |adEval - manualEval| < eps
        autoEval shouldBeApproximately 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 finite differences:

```
"d(sin x)/dx should be equal to (sin(x + dx) - sin(x)) / dx" {
    NumericalGenerator.assertAll { x0 ->
        val f = sin(x)
        val df_dx = d(f) / d(x)
        val adEval = df_dx(x0)
        val dx = 1E-8
        val fdEval = (sin(x0 + dx) - sin(x0)) / dx
        adEval shouldBeApproximately fdEval
    }
}
```

There are many other ways to independently verify the numerical gradient, such as dual numbers or the complex step derivative. Another method is to compare the numerical output against a well-known implementation, such as TensorFlow. We plan to conduct a more thorough comparison of numerical accuracy and performance.

3.10. Shape safety

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

(1) Hide the error somehow by implicitly reshaping or broadcasting arrays

(2) Announce the error at runtime, with a relevant message, e.g. `InvalidArgumentError`

(3) Do not allow programs which can result in a shape error to compile

In Kotlin∇, we use the third strategy. Consider the following program:

```
val a = Vec(1.0, 2.0) // Inferred type: Vec<Int, '2'>
val b = Vec(1.0, 2.0, 3.0) // Inferred type: Vec<Int, '3'>
val c = b + b
```

```
val d = a + b // Does not compile, shape mismatch
```

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

```
val a = Mat(‘1‘, ‘4‘, 1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, ‘1‘, ‘4‘>
val b = Mat(‘4‘, ‘1‘, 1.0, 2.0, 3.0, 4.0) // Inferred type: Mat<Double, ‘4‘, ‘1‘>
val c = a * b
val d = a * a // Does not compile, inner dimension mismatch
```

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

```
val a = Mat(‘2‘, ‘4‘,
            1.0, 2.0, 3.0, 4.0,
            5.0, 6.0, 7.0, 8.0)
val b = Mat(‘4‘, ‘2‘,
            1.0, 2.0,
            3.0, 4.0,
            5.0, 6.0,
            7.0, 8.0)
val c: Mat<Double, ‘2‘, ‘2‘> = a * b // Types are optional, but encouraged
val d = Mat(‘2‘, ‘1‘, 1.0, 2.0)
val e = c * d
val f = Mat(‘3‘, ‘1‘, 1.0, 2.0, 3.0)
val g = e * f // Does not compile, inner dimension mismatch
```

Explicit types are optional but encouraged. Type inference helps preserve shape information over long programs.

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

When writing a function, it is mandatory to declare the input type(s), but the return type may be omitted. Shape safety is currently supported up to rank-2 tensors, i.e. matrices.

First class dependent types are useful for ensuring arbitrary shape safety (e.g. when concatenating and reshaping matrices), but they are unnecessary for simple equality checking (such as when multiplying two matrices)[4]. When the shape of a tensor is known at compile time, it is possible to encode this information using a less powerful type system, as long

[4]Many less powerful 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 [**34**]. It may be possible to

as it supports subtyping and parametric polymorphism. In practice, we can implement a shape-checked tensor arithmetic in languages like Java, Kotlin, C++, C# or Typescript, which accept generic type parameters. In Kotlin, whose type system is less expressive than Java, we use the following strategy.

First, we enumerate a list of integer type literals as a chain of subtypes, so that $0 <: 1 <: 2 <: 3 <: \cdots <: C$, where C is the largest fixed-length dimension. Using this encoding, we are guaranteed linear growth in space and time for subtype checking. C can be specified by the user, but in order to do so the code will need to be regenerated.

```
open class `0`(override val i: Int = 0): `1`(i) { companion object: `0`(), Nat<`0`> }
open class `1`(override val i: Int = 1): `2`(i) { companion object: `1`(), Nat<`1`> }
open class `2`(override val i: Int = 2): `3`(i) { companion object: `2`(), Nat<`2`> }
open class `3`(override val i: Int = 3): `4`(i) { companion object: `3`(), Nat<`3`> }
//...This is generated
sealed class `100`(open val i: Int = 100) { companion object: `100`(), Nat<`100`> }
interface Nat<T: `100`> { val i: Int } // Used for certain type bounds
```

Kotlin∇ supports shape-safe tensor operations by encoding tensor rank as a parameter of the operand's type signature. 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 to preserve shape safety for all subtypes. Let us consider the rank-1 tensor (i.e. vector) case:

```
infix operator fun <C: `100`, V: Vec<Float, C>> V.plus(v: V): Vec<Float, C> =
    Vec(length, contents.zip(v.contents).map { it.first + it.second })
```

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, MaxLen: `100`> constructor(val len: Nat<MaxLen>, 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))
        //...
    }
```

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

```
}
```

Dynamic length construction is also possible, although this 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(‘2‘, 1, 2, 3)                       // Does not compile
val sum = Vec(‘2‘, 1, 2) + add                    // Does not compile
```

A similar syntax is used for matrices and tensors. For example, Kotlin∇ can infer the shape of matrix multiplication, and will not compile if their inner dimensions disagree:

```
val l = 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 m = Mat(‘4‘, ‘3‘, // Inferred type: Mat<Int, ‘4‘, ‘3‘>
            1, 1, 1,
            2, 2, 2,
            3, 3, 3,
            4, 4, 4)
val lm = l * m // Inferred type: Mat<Int, ‘4‘, ‘3‘>
val mm = m * m // Does not compile
```

This technique originates in Haskell, which supports more powerful forms of type-level computation, *type arithmetic* [**43**]. Type arithmetic makes it easy to express convolutional arithmetic [**24**] and other arithmetical operations on shape variables, which is currently not possible in Kotlin∇, or would require enumerating every possible combination of type literals.

3.11. Operator overloading

Operator overloading enables 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 [**14**] to ensure that operations return the concrete type variable T, rather than something more generic like `Group` (effectively, `T` is a `self` type). Imagine a class `Expr` which has implemented `Group`. It can be used as follows:

```
fun <T: Group<T>> cubed(t: T): T = t * t * t
fun <E: Expr<E>> twiceExprCubed(e: E): E = 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 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 (typed FP). A number of recent papers [**60, 74**] have demonstrated the expressiveness of this paradigm for automatic differentiation.

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<Double> = Var(1.0)
val f = makePoly(x, y)
val z = f(1.0, 2.0) // Returns a value
println(z) // Prints: 7
```

3.13. Coroutines

Coroutines are a generalization of subroutines for non-preemptive multitasking, typically implemented using continuations. 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 Wang et al. in Shift/Reset the Penultimate Backpropagator [**75**] and later in Backpropagation with Continuation Callbacks [**74**]. Delimited continuations can be implemented using Kotlin coroutines and they are currently a work in progress.

3.14. Extension Functions

Extension functions augment external classes with new fields and methods. Via context oriented programming, Kotlin∇ can expose its custom extensions (e.g. in `DoublePrecision`) to consumers without requiring subclasses or inheritance.

Listing 3.4. Using extension functions we can provide numerical conversions for common data types, wrapped by a Context.

```
data class Const<T: Group<T>>(val number: Double) : Expr()
data class Sum<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()
data class Prod<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()

class Expr<T: Group<T>>: Group<Expr<T>> {
    operator fun plus(addend: Expr<T>) = Sum(this, addend)
    operator fun times(multiplicand: Expr<T>) = Prod(this, multiplicand)
}

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

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

```
fun Expr<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 [**57**], another mathematical library for Kotlin.

3.15. Algebraic data types

Algebraic data types (ADTs) in the form of sealed classes (a.k.a. sum types) allows creating a closed set of internal subclasses to guarantee an exhaustive control flow over the concrete types of an abstract class. At runtime, we can branch on the concrete type of the abstract class. For example, suppose we have the following classes:

```
class Const<T: Group<T>>(val number: Double) : Expr()
class Sum<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()
class Prod<T: Group<T>>(val e1: Expr, val e2: Expr) : Expr()
class Var<T: Group<T>>: Expr()
class Zero<T: Group<T>>: Const<T>
class One<T: Group<T>>: Const<T>
```

Listing 3.5. Users must handle all subclasses when branching on the type of a sealed class, as incomplete control flow will not compile (instead of failing silently at runtime).

```
sealed class Expr<T: Group<T>>: Group<Expr<T>> {
    fun diff() = when(expr) {
       is Const -> Zero
       // Smart casting allows us to access members of a checked typed without explicit casting
       is Sum -> e1.diff() + e2.diff()
       // Product rule: d(u*v)/dx = du/dx * v + u * dv/dx
       is Prod -> e1.diff() * e2 + e1 * e2.diff()
       is Var -> One
       // Since the subclasses of Expr are a closed set, no 'else -> ...' is required.
    }
    operator fun plus(addend: Expr<T>) = Sum(this, addend)
    operator fun times(multiplicand: Expr<T>) = Prod(this, multiplicand)
}
```

Smart-casting allows us to treat the abstract type `Expr` as a concrete type, e.g. `Sum` after performing an `is Sum` check. Otherwise, we would need to write `(expr as Sum).e1` in order to access its field, `e1`. If the type were incorrect, performing a cast without checking would throw a `ClassCastException`, which sealed classes prevent.

3.16. Multiple Dispatch

In conjunction with ADTs, Kotlin∇ also uses multiple dispatch to instantiate the most specific result type of applying an operator based on the type of its operands. While multiple dispatch is not an explicit language feature, it can be emulated using inheritance.

Building on the previous example, suppose we would like to perform algebraic simplification. This can be useful for reducing expression swell and improving numerical stability. We can use multiple dispatch to branch on a the type of a subexpression at runtime. *Smart casting* lets us access class members after first checking their type, as it if were casted:

Listing 3.6. 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.

```
override fun times(multiplicand: Expr<X>): Expr<X> =
    when {
        this == zero -> this
        this == one -> multiplicand
        multiplicand == one -> this
        multiplicand == zero -> multiplicand
        this == multiplicand -> pow(two)
        // Without Smart Casting: const((this as Const).number * (multiplicand as Const).number)
        this is Const && multiplicand is Const -> const(number * multiplicand.number) // With SC
        // 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))
```

3.17. Numeric Tower

Kotlin∇ uses a numeric tower [**69**]. First pioneered in Scheme[**68**], this strategy is highly-suited to object oriented programming [**55, 56**] and widely used in other mathematical libraries such as KMath [**57**] and Apache Commons Math [**22**]. By doing so, we are able to define common functionality on the supertype.

Listing 3.7. Many common mathematical operations can be defined in simpler terms.

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

Furthermore, if we need to later define a field over the quaternions (suppose for backpropogating through a quaternion neural network [**38**]), these abstractions are easy to extend, since we only need to implement a very small set of primitive operations.

3.18. Symbolic and Automatic Differentiation

It has long been claimed that automatic differentiation is not symbolic differentiation [**6**]. Many, including the author, have suspected this claim to be misleading. Recent literature has questioned [**75**] and refuted [**45**] this claim. It is our view the distinction is a minor semantic one. While it may be true that certain implementations of automatic differentiation do interleave numerical and symbolic differentiation during a program's execution, it is certainly not a prerequisite for a library to be considered automatic differentiation, especially considering that prior AD implementations, including Theano, have taken a different approach. Furthermore, we consider symbolic differentiation to be a type of automatic differentiation.

3.19. Comparison

Kotlin∇ is a symbolic graph-based autograd.

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	✗	✓	✓	✗	✗	✗	✗

Chapitre 4

Verification 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 Weiner, *Some moral and technical consequences of automation* **[82]**

Roughly speaking, in backpropogation we perform gradient descent on the parameters of a program for a given input. In adversarial testing, we do gradient ascent on the input, at a given parameter setting.

Neural networks and differentiable programming have provided a powerful set of optimization tools for training learning algorithms. However, these methods are often brittle to small variations in the input space, and have difficulty with generalization. In contrast, these same techniques used for probing the failure modes of neural networks can be applied to adversarial test case generation for traditional programs.

4.1. Background

Suppose we have a program $P : \mathbb{R} \to \mathbb{R}$ where:

$$P(p_0) = p_n \circ p_{n-1} \circ p_{n-2} \circ ... \circ p_1 \circ p_0 \circ p_0 \tag{4.1.1}$$

From the chain rule of calculus, we know that:

$$\nabla p_i(p_0) = \frac{dp_{i+1}}{dp_i} \cdot \frac{dp_i}{dp_{i-1}}, \forall i \in (0, n) \tag{4.1.2}$$

More broadly, if $P : \mathbb{R}^n \to \mathbb{R}^k$, and $p_i : \mathbb{R}^{dim_{\mathbb{R}}(p_{i-1})} \to \mathbb{R}^{dim_{\mathbb{R}}(p_{i+1})} \forall i \in (0, n)$:

$$\nabla p_i(p_0) = \nabla p_i(p_{i-1}(p_0)) \cdot \nabla p_{i-1}(p_0) \tag{4.1.3}$$

Imagine a single test $T : \mathbb{R} \to \mathbb{B}$. Consider the following example:

$$T(P) = \forall x \in (0, 1), P(x) < C \tag{4.1.4}$$

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:

$$D_T : \{x^i \sim \mathbb{R}(0, 1) \mid P(x^i) \implies \neg T\}, maximize|D_T| \tag{4.1.5}$$

If we have no information about the program implementation or its input distribution, D_P, we can do no better than random search **[83]**. However, if we know something about the input distribution, we could re-parameterize the distribution to incorporate our knowledge. Assuming the program has been tested on common inputs, we might consider sampling $x \sim \frac{1}{D_P}$ for inputs that are infrequent. If we knew how P were implemented, we could prioritize our search in input regions leading towards internal discontinuities (e.g. edge cases in software testing). However for functions that are continuous and differentiable, these heuristics are almost certainly insufficient.

Another strategy, independent of how candidate inputs are selected, is to use some form of gradient based optimization in the search procedure. The gradient of P's loss with respect to x (assuming θ is fixed[1]) is $\nabla_x \mathcal{L}(P(x; \theta))$. Instead of minimizing the loss, we want to maximize it. So the vanilla gradient update step 3.1.10 becomes:

$$x^i_{n+1} = x^i_n + \alpha \nabla_{x_n} \mathcal{L}(P(x_n; \theta)) \tag{4.1.6}$$

We hypothesize that if the implementation of P were flawed and a counterexample to (3) existed, as sample size increased, a subset of gradient descent trajectories would fail to converge, a portion would converge to local minima, and a subset of trajectories would discover inputs violating the program specification. How would such a search procedure look in practice? Consider the following algorithm:

4.2. Regression testing and forgetting

An endemic problem in modern deep learning is the problem of forgetting. In order to combat this issue, we turn to a classic software testing tool: regression testing.

[1] In contrast with backpropogation, where the parameters are updated.

Input : Program P, specification T, evaluation budget $Budget$

Output: D_T, the set of inputs which cause P to fail on T

$D_T = []$;

evalCount = 0;

while *evalCount* $\leq$*Budget* **do**

  sample candidate input x^i *according to selection strategy S;*

  if $P(x^i) \implies \neg T$ **then**

    append x^i *to* D_T

  else

    $n = 0$;

    $x_n^i = x^i$;

    while $n \leq C \wedge evalCount \leq Budget \wedge \neg converged$ **do**

      $n{+}{+}$;

      $x_n^i = x_{n-1}^i - \alpha \frac{dP}{dx}$;

      if $P(x_n^i) \implies T$ **then**

        append x_n^i *to* D_T

        break;

      end

      evalCount++;

    end

  end

  evalCount++; i++;

end

Algorithm 1: Algorithm for finding test failures. First select a candidate input x^i according to sampling strategy S (e.g. uniform random, or a neural network which takes P and T as input). If $P(x^i)$ violates T, we can append x^i to D_T and repeat. Otherwise, we follow the gradient of $\mathcal{L}(P, x)$ with respect to x and repeat until test failure, gradient descent convergence, or a fixed number of steps C are reached before resampling x^{i+1} from the initial sampling strategy S to ensure each gradient descent trajectory will terminate before exhausting our budget.

Regular regression testing gives clear diagnostics about the behavior of intelligent systems.

Chapitre 5

Software Maintenance and Reproducibility

In this chapter, we will discuss the challenges of software reproducibility and how best practices in software engineering like continuous integration and delivery can help researchers mitigate the variability associated with building and running software. Docker [**54**] is one technical solution we will discuss. Our work is roughly related to computational determinism, and does not consider the variability of distributional shift or related statistical notions of variation.

5.1. Operating systems and virtualization

In 2006, Linux began introducing a variety of new kernel features for controlling groups of processes, under the aegis of **cgroups** [**53**]. Collectively, these features supported a kind of lightweight virtualization, where a fully virtual machine was no longer necessary to get many of the benefits of VMs, primarily resource control and namespace isolation. 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 the host OS and sibling containers. While the overhead of running VMs often limits their deployment to server-class hardware, containers are comparatively lightweight, which enables them to run on a far broader class of mobile and embedded platforms.

5.2. Dependency management

One common source of software variability are software dependencies and the dependency solving problem. In fact, operating systems have struggled with this issue for a number of years before it was realized that dependency resolution was NP-complete [**2**].

5.3. Containerization

One of the challenges of distributed software development across heterogeneous platforms is the problem of variability. With the 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 a robotics setting where contributors are frequently joining and leaving the project. Together, these challenges present significant obstacles to experimental reproducibility and scientific collaboration.

In order to address the issue of software reproducibility, we developed 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 participants and minimize variability across platforms (e.g. simulators, robotariums, Duckiebots), we provide a state-of-the-art container infrastructure based on Docker, a popular container engine. Docker allows us to construct versioned deployment artifacts that represent the entire filesystem and to manage resource constraints via a sandboxed runtime environment.

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 all target architectures. Runtime emulation of foreign artifacts is also possible, using a similar technique.[1] For performance and simplicity, we only build ARM artifacts and use emulation where necessary (e.g., on x86 devices). On ARM-native, the base operating system is HypriotOS, a lightweight Debian distribution with built-in 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.

Docker containers are sandboxed runtime environments that are portable, reproducible and version controlled. Each environment contains all the software dependencies necessary to run the packaged application(s), but remains isolated from the host OS and file system. Docker provides a mechanism to control the resources each container is permitted to access,

[1]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/`.

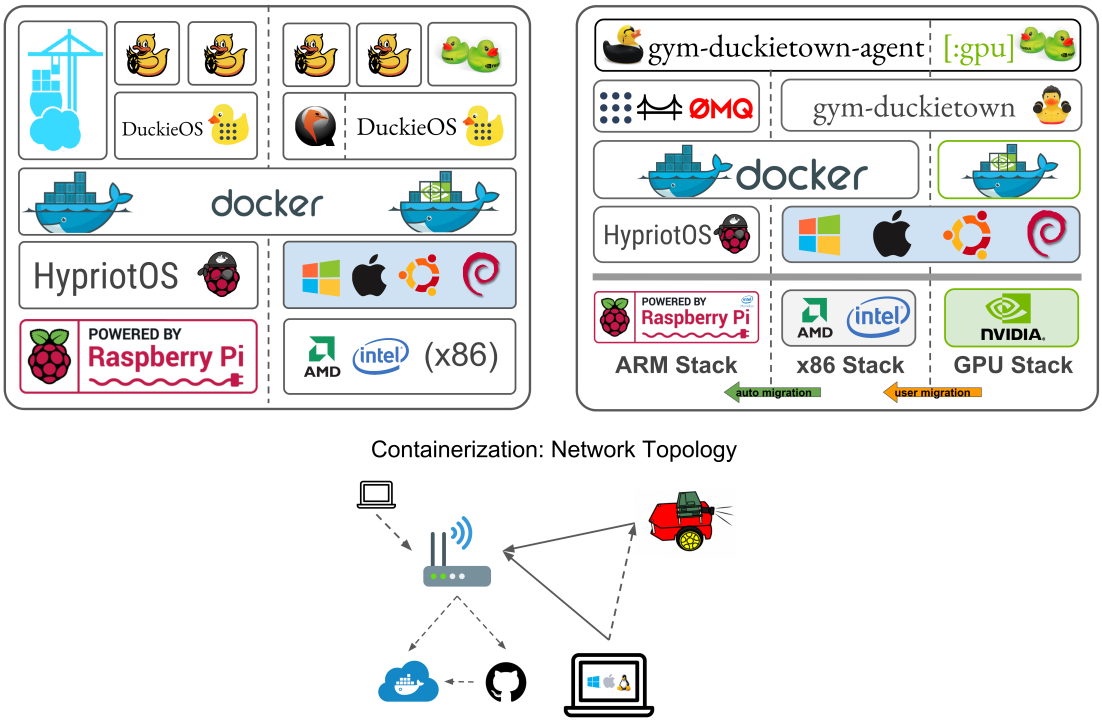

Fig. 5.1. AI-DO container infrastructure. Left: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we only build ARM artifacts, and emulate ARM on x86. Right: Reinforcement learning stack. Build artifacts are typically trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their specific architecture, may be run on an ARM device using an Intel NCS.

and a separate Linux namespace for each container, isolating the network, users, and file system mounts. Unlike virtual machines, container-based virtualization like Docker only requires a lightweight kernel, and can support running many simultaneous containers with close to zero overhead. A single Raspberry Pi is capable of supporting hundreds of running containers.

While containerization simplifies the process of building and deploying applications considerably, 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 may be based on a number of lower layers, which must periodically be updated.

Care must be taken when designing the development pipeline to ensure that such updates do not silently break a subsequent layer as described earlier in Sec. **??**.

One issue encountered is the matter of whether to package source code directly inside the container, or to store it separately. If source code is stored separately, a developer can use a shared volume on the host to build the artifacts. In this case, while build artifacts images may be standalone reproducible, they are not easily modified or inspected. The second method is to ship code directly inside the container, where any changes to the source code will trigger a subsequent rebuild, effectively tying the sources and the build artifacts together. Including source code alongside build artifacts has the benefit of improved reproducibility and diagnostics. If a competitor requires assistance, troubleshooting becomes much easier when the source code is directly accessible. However doing so adds some friction during development, which has caused competitors to struggle with environment setup. One solution is to store all sources on the local development environment and rebuild the Docker image periodically, copying sources into the image.

5.4. Docker and ROS

Prior work has explored the Dockerization of ROS containers [**81**]. This work forms the basis for our own, which is extended specifically for the Duckietown platform [**59**].

Chapitre 6

Case study: application for autonomous robotics

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. Verification 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 naively, 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 naive 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 overall trend in software and systems engineering is the transition away from long development cycles towards continuous integration and deployment. Development teams in 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 code on an even shorter basis.

Incremental updates will grow increasingly smaller, until the program starts to alter itself after every input it processes.

We need tools to more effectively harness the stochasticity of these learning systems.

7.1.3. Developers, Operations, and the DevOps toolchain

Software engineers have begun to realize the value of bespoke tools that facilitate the process of shipping software, in addition to the software itself.

Teams building software are cybernetic systems, and require meta-programs for building code and organizational processes which enable them to ship code more efficiently.

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015.

[7] Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Diffsharp: Automatic differentiation library. *CoRR*, abs/1511.07727, 2015.

[8] 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.

[9] 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.

[10] 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.

[11] Yang Bo. Deep L earning.scala: A simple library for creating complex neural networks. 2018.

[12] 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.

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

[14] 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.

[15] Tongfei Chen. Typesafe abstractions for tensor operations (short paper). pages 45–50, 2017.

[16] Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1985.

[17] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

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

[19] 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.

[20] 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.

[21] Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis Wyffels. A differentiable physics engine for deep learning in robotics. *CoRR*, abs/1611.01652, 2016.

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

[23] 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.

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

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

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

[27] 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.

[28] M. Fowler. Fluent interface, 2005.

[29] 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.

[30] Yossi Gil and Tomer Levy. Formal language recognition with the java type checker. 56, 2016.

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

[32] Andreas Griewank. Some bounds on the complexity of gradients, jacobians, and hessians. In *Complexity in numerical optimization*, pages 128–162. World Scientific, 1993.

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

[34] Radu Grigore. Java generics are T uring C omplete. pages 73–85, 2017.

[35] Martin Guenther. Are serious things done with ros in python? - discourse.ros.org. `https://discourse.ros.org/t/are-serious-things-done-with-ros-in-python/4359/6`. (Accessed on 04/12/2019).

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

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

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

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

[44] Chris Lattner and Richard Wei. Swift for tensorflow. 2018.

[45] Soeren Laue. On the equivalence of forward mode automatic differentiation and symbolic differentiation. *arXiv preprint arXiv:1904.02990*, 2019.

[46] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

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

[48] 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.

[49] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

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

[51] 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.

[52] 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.

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

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

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

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

[57] 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.

[58] 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.

[59] 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.

[60] 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, 2008.

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

[62] 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.

[63] 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.

[64] 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.

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

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

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

[68] 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.

[69] 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.

[70] 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.

[71] Ross Tate. Mixed-site variance. FOOL, 2013.

[72] 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.

[73] 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.

[74] 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, 2018.

[75] Fei Wang, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *arXiv preprint arXiv:1803.10228*, 2018.

[76] Fei Wang, Xilun Wu, Gr é gory M. Essertel, James M. Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018.

[77] 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, 2018.

[78] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

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

[80] 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.

[81] Ruffin White and Henrik Christensen. Ros and docker. In *Robot Operating System (ROS)*, pages 285–307. Springer, 2017.

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

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