Université de Montréal

Software tools and processes for intelligent systems engineering

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

mars 2019

Sommaire

Summary

Table des matières

Liste des tableaux

Liste des figures

Introduction

Development environment

Software reproducibility

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 the same 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

[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/`.

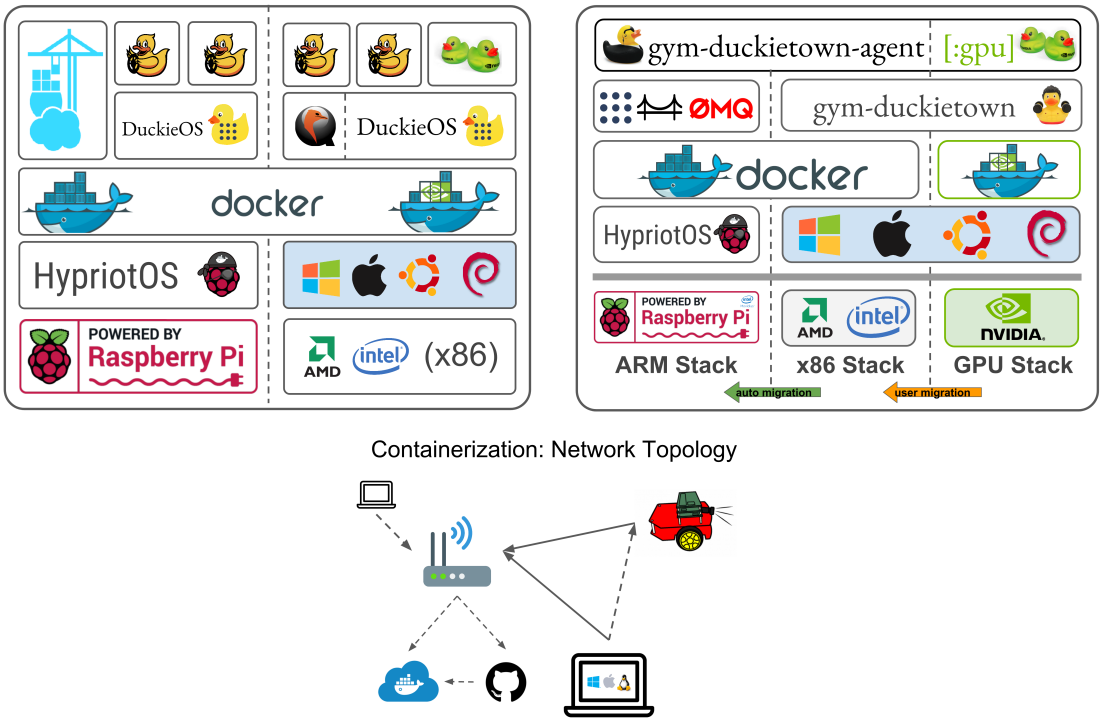


Fig. 0.1. AI-DO container infrastructure. Left: The ROS stack targets two primary architectures, x86 and ARM. To simplify the build process, we only build ARM artifacts, and emulate ARM on x86. Right: Reinforcement learning stack. Build artifacts are typically trained on a GPU, and transferred to CPU for evaluation. Deep learning models, depending on their specific architecture, may be run on an ARM device using an Intel NCS.

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, 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 considerably simplifies the process of building and deploying applications, it also introduces some additional complexity to the software development lifecycle. Docker, like most container platforms, uses a layered filesystem. This enables Docker to take an existing "image" and change it by installing new dependencies or modifying its functionality. Images 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 OS for build purposes. In this case, while submissions may be 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 also has the benefit of reproducibility and diagnostics. If a competitor requires assistance, troubleshooting becomes much easier when 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 environment periodically, copying sources into the image.

Languages and compilers

Prior work has shown it is possible to encode a deterministic context-free grammar as a *fluent interface* [**?**] in Java. This result was strengthened to prove Java's type system is Turing complete [**?**]. 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. We use *Kotlin*, whose type system is less expressive, but fully compatible with Java.

Differentiable programming has a rich history among dynamic languages like Python, Lua and JavaScript, with early implementations including projects like Theano, Torch, and TensorFlow. Similar ideas have been implemented in statically typed, functional languages, such as Haskell's Stalin∇ [**?**], DiffSharp in F# [**?**] and recently Swift [**?**]. 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 [**?**], Nexus [**?**] and DeepLearning.scala [**?**], 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 AD framework. In addition to type-safety and interoperability, Kotlin∇ primarily relies on the following language features:

[itemsep=-0.5ex]**Operator overloading and infix functions** allow a concise notation for defining arithmetic operations on tensor-algebraic structures, i.e. groups,

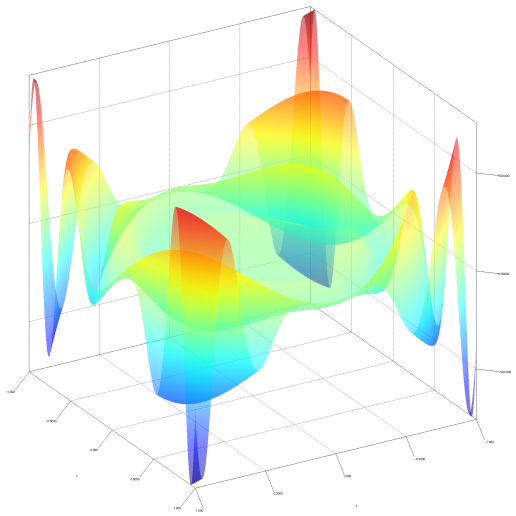
rings and fields. **λ-functions and coroutines** support backpropogation with lambdas and shift-reset continuations, following [?] and [?]. **Extension functions** support extending classes with new fields and methods and can be exposed to external callers without requiring sub-classing or inheritance.

0.1. Usage

Kotlin∇ allows users to implement differentiable programs by composing operations on fields to form algebraic expressions. Expressions are lazily evaluated inside a numerical context, which may imported on a per-file basis or lexically scoped for finer-grained control over the runtime behavior.

Fig. 0.2. A basic Kotlin∇ program with two inputs and one output.

Above, 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. To demonstrate, a display is required for animation purposes.



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

Fig. 0.3. Output generated by the program shown in Figure 0.2.

Kotlin∇ treats mathematical functions and programming functions with the same underlying abstraction. Expressions are composed recursively 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)`.

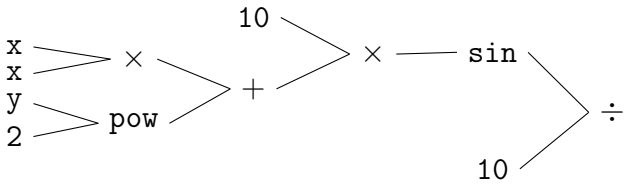


Fig. 0.4. Implicit DFG constructed by the original expression, `z`.

Kotlin∇ supports shape-shafe 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.

Fig. 0.5. Shape safe tensor addition for rank-1 tensors, $\forall L \leq 2$.

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. A similar pattern can be applied to matrices and higher rank 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 that is fully interoperable with Java, while also capable of compiling to JVM bytecode, JavaScript, and native code.

In future work, we intend to implement a full grammar of differentiable primitives including matrix convolution, control flow and recursion. While Kotlin∇ currently implements arithmetic manually, we plan to wrap a BLAS such as cuBLAS or native linear algebra library for performance.

Testing and validation

Neural networks and differentiable programming has given us a powerful new 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.

0.2. Background

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

$$P(x) = p_n \circ p_{n-1} \circ p_{n-2} \circ ... \circ p_1 \circ p_0 \tag{0.2.1}$$

From the chain rule of calculus, we know that:

$$\frac{dP}{dp_0} = \prod_{i=1}^{n} \frac{dp_i}{dp_{i-1}} \tag{0.2.2}$$

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

$$T : \forall x \in (0, 1), P(x) < C \tag{0.2.3}$$

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:

$$D_T : \{x^i \sim \mathbb{R}(0, 1) \mid P(x^i) \implies \neg T\}, maximize|D_T| \tag{0.2.4}$$

If we have no information about the program implementation or its input distribution, D_P, we can do no better than random search (cf. no free lunch). However, if we know something about the input distribution, we could re-parameterize the distribution to incorporate our knowledge. Assuming the program already works on most common inputs (e.g. from a training set), we might want to sample $x \sim \frac{1}{D_P}$ for inputs that are infrequent, although

this strategy is not guaranteed to yield any additional failures. If we knew how P were implemented, we might be able 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. Finally, we could train a neural network to predict inputs that were likely to cause a program to fail a given specification. As input, the network would take the function and test cases, and as its output, produce values that were likely to violate T.

Another strategy, independent of how candidate inputs are selected, is to use some form of gradient based optimization in the search procedure. For example in (3) we could have a loss function:

$$\mathcal{L}(P, x) = C - P(x) \tag{0.2.5}$$

The gradient of the loss w.r.t. x (assuming P is fixed[2]) is:

$$\nabla\mathcal{L}(P, x) = -\frac{dP}{dx} \tag{0.2.6}$$

Where the vanilla gradient update step is defined as:

$$x^i_{n+1} = x^i_n - \alpha\frac{dP}{dx} \tag{0.2.7}$$

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:

[2]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 G.D. steps C are reached before resampling x^{i+1} from the initial sampling strategy S to ensure each G.D. trajectory will terminate before exhausting our budget.

Case study: an application