Fashionable Modelling With Flux

Mike Innes[1], Elliot Saba[1], Keno Fischer[1], Dhairya Gandhi[1], Marco Concetto Rudilosso[2], Neethu Mariya Joy[3], Tejan Karmali[4], Avik Pal[5], and Viral B. Shah[1]

[1]Julia Computing, Inc.
[2]University College London
[3]Birla Institute of Technology and Science Pilani, India
[4]Indian Institute of Science, Bangalore, India
[5]Indian Institute of Technology Kanpur, India

ABSTRACT

Machine learning as a discipline has seen an incredible surge of interest in recent years due in large part to a perfect storm of new theory, superior tooling, renewed interest in its capabilities. We present in this paper a framework named `Flux` that shows how further refinement of the core ideas of machine learning, built upon the foundation of the Julia programming language, can yield an environment that is simple, easily modifiable, and performant. We detail the fundamental principles of `Flux` as a framework for differentiable programming, give examples of models that are implemented within `Flux` to display many of the language and framework-level features that contribute to its ease of use and high productivity, display internal compiler techniques used to enable the acceleration and performance that lies at the heart of `Flux`, and finally give an overview of the larger ecosystem that `Flux` fits inside of.

Proceedings of JuliaCon

Keywords

Julia, Optimization, Machine Learning, Compiler, Differentiable Programming

1. Introduction

`Flux` is a new machine learning (ML) stack. Only a year old and developed by a very small team, it nonetheless has inspired an enthusiastic and rapidly growing community of researchers and industry users implementing novel kinds of models and acceleration techniques.

ML engineering is fundamentally a programming languages problem; machine learning models are growing ever more complex, incorporating control flow, state and data structures, and borrowing techniques from research areas throughout the field of computer science from software engineering to scientific simulation to statistical inference. These models are best viewed as *differentiable algorithms*; and to express algorithms we use programming languages [16].

`Flux` takes Julia [6] to be this language. Julia is recent but, being designed from the ground up for mathematical and numerical computing, unusually well-suited for expressing ML programs. Its mix of modern design and novel techniques in the compiler—which we have significantly extended to further suit differentiable algorithms and accelerators—makes it easier to address the high performance requirements needed for applying machine learning to large models and data sets.

There are three pillars that set `Flux` apart among ML systems: simplicity, hackability, and underlying compiler technology.

1.1 Simplicity

The word “simple" is typically not thrown around in the ML systems world. However, simplifying solutions is a crucial part of approaching more complex problems. Where typical ML frameworks are written in many hundreds of thousands of lines of C++ [1], `Flux` is only a thousand lines of straightforward Julia code.

Several factors compound to enable this. First, the ability to write everything from layer definitions, to algorithmic differentiation, to CUDA kernels, in high-level and *fast* Julia code enables high developer productivity; code can typically be written once and forgotten about rather than being written, rewritten and optimized in C over years. Fast high-level code also enables something much more powerful than convenience: abstraction. This means infrastructure like higher-order kernels for `map(f,xs)` and `broadcast`, which can be written once and generalize to any user-defined `f` or input type with no extra effort or undue performance overhead. These high-level features in other languages and frameworks are often available, but typically incur performance penalties such that all “real work” must be done in the lower-level language components of the ML systems.

This extends to integration with other tools in the ecosystem. Writing an image pipeline really means loading the `Flux` and `Images` [13] packages, and using them as normal. In fact, even GPU support is handled this way; rather than being provided as part of `Flux`, one loads a generic GPU arrays package such as `CuArrays` [23], and passing those arrays into a `Flux` model just works; transferring data and computational kernels off to the GPU accelerator then bringing the results back to the CPU without any special handling within `Flux` itself. This leverages Julia's heavy use of specialization, which effectively generates custom code for the GPU, and even for custom floating-point types that would otherwise need to be written by hand. Far from becoming an all-encompassing monolith, `Flux` remains a lean "glue" package, bringing together a set of underlying abstractions (e.g. gradients and GPU support) and combining them to create an ML framework.

1.2 Hackability

A core tenet of `Flux`, and Julia more generally, is that library code is just user code that happens to have been loaded from an external file. Unlike previous ML frameworks (including those written in Julia), `Flux` does not pick a certain level of abstraction (such as mathematical graphs or layer stacking) that all models must use. Instead, careful design of the underlying automatic differentiation (Section 3.2) allows freely mixing mathematical expressions, built-in and custom layers and algorithms with control flow in one model. This makes `Flux` unusually easy to extend to new problems.

`Flux` users regularly inject custom CUDA kernels, write down new mathematical functions and layers, hook in custom gradient definitions and even custom parallel training algorithms, all with the same performance as built-in features. Hooking in custom functionality is often a one-line change that happens in the same Julia script as model code. Because there is no difference between `Flux` code and other general Julia code, it is possible to break out of the typical deep learning paradigm and experiment with new concepts such as interleaving gradient descent training with MCMC. Just write down your own training loop; the obvious mathematical code will work, and be fast.

This extends to integrating other packages with `Flux` models, such as Julia's state-of-the-art tools for mathematical optimization [8], differential equations [30] and probabilistic programming [9]. This goes deeper than just using neural nets alongside these other techniques, as one can even incorporate them directly into models. For example, Julia's differential equation solvers while not explicitly adapted either for AD or GPUs can seamlessly be used with both. This means that one can use a physical simulation to enhance the predictions of the model, then backpropagate through the entire model *including* the simulation to get gradients. Bringing tools like physics simulators into models is where deep learning truly becomes differentiable programming.

1.3 Compiler Technology

`Flux` is committed to providing a dynamic (or "define-by-run") interface, and takes a hard line against any kind of graph building or performance annotations [29]. We support all of Julia's language features, from control flow and data structures to macros. Users can code interactively in Jupyter notebooks and combine high-performance numerics with convenient plotting and visualization. But we also want to get the benefits traditionally held by "static graph" frameworks - zero-overhead source-to-source AD, operator fusion, multi-GPU/distributed training, and single-executable deployment.

Doing this effectively requires extracting and analyzing "static graphs" directly from written Julia syntax, a common task within the field of programming language theory. Most ML systems problems are, in fact, standard and well-studied compiler problems, viewed through the right lens. Using a compiled language is enough to solve many issues, and extending that compiler is the best way to solve many more.

Several illustrative examples, covering differentiation (Section 3.2), GPU and TPU compilation (Sections 3.1 and 3.3) and SPMD / batching (Section 3.4) are covered briefly in this paper.

2. Fashionable Modelling

The inherent design of `Flux`, built on top of the technical foundation of Julia's multiple dispatch semantics, unlocks the ability for `Flux` to act as an extremely flexible "glue" that can bring together disparate packages into a single, differentiable and high-performance program. Multiple dispatch allows the programmer to write a function definition in terms of "verbs", (e.g. *sum*, *relu*, *multiplication*) without needing to create a different version for each separate type of the data being operated upon. An example of what such a function looks like is given in listing 1, where the definition of the `leakyrelu()` activation function is given.

```julia
function leakyrelu(x, a=oftype(x/1, 0.01))
    return max(a*x, x/1)
end
```

Listing 1: `leakyrelu()` activation function definition

This code listing demonstrates the three tenets of the `Flux` paradigm: Its simplicity is evident, and the structure of the code follows very closely to the mathematical definition of Leaky ReLU as given in the original paper [21]. It is hackable, in that these lines of code in a `Flux` program are all that is needed to add a new activation function to the machine learning library, setting an extremely low barrier to entry for definitions of new layers, activation functions, etc... Finally, in a nod toward the realities of high performance computing, Julia (and by extension, `Flux`) makes it easy to provide to the compiler the kind of information necessary to generate high-performance code. In this case, the `leakyrelu()` function ensures that the default `a` parameter is of the same type as whatever type would result from division involving the `x` parameter. This allows for intelligent (and type-stable) code to be generated for double-precision arrays on the CPU, GPU arrays with reduced precision, or even TPU arrays with further reduced precision, all from a single, readable, function definition.

Comparing the definition given in listing 1 with the equivalent definition in a graph-construction based framework without the benefit of Julia's type system emphasizes the ease with which new definitions and new fundamental operations can be added. Listing 2 shows the equivalent operation performed within TensorFlow [1]. The fundamental operations being used within the Python `leakyrelu()` function are defined within TensorFlow and are independent of the typical mathematical functions one would use in a normal Python program. This disconnect causes a severe lack of composability; one cannot mix and match functionality from different parts of the language ecosystem because all fundamental operations are built upon the foundations of the TensorFlow runtime.

```
def leakyrelu(x, a=0.2, name=None):
  x = ops.convert_to_tensor(x)
  if x.dtype.is_integer:
    x = math_ops.to_float(x)
  a = ops.convert_to_tensor(a, dtype=x.dtype)
  return math_ops.maximum(a * x, x)
```

Listing 2: `leakyrelu()` activation function definition

In the following sub-sections we will demonstrate how the three pillars of `Flux` (*Simplicity*, *Hackability* and *Compiler Technology*) combine to create an attractive and productive environment for machine learning researchers to build effective, performant models across a variety of applications. We do so by picking a few well-known models from the machine learning community and displaying salient portions of their implementations in `Flux`.

2.1 Resnet

```
1  struct ResidualBlock
2      conv::Tuple
3      norm::Tuple
4      shortcut
5  end
6
7  function (B::ResidualBlock)(input)
8      x = B.norm[1](B.conv[1](input))
9      for i in 2:length(B.conv)
10         x = B.norm[i](B.conv[i](relu.(x)))
11     end
12     return relu.(x + B.shortcut(input))
13 end
```

Listing 3: A resnet block in `Flux`

Resnet [12], a very popular computer vision model making extensive use of "residual connections" to combat the vanishing-gradients problem, yields a succinct example of many of the advantages that `Flux` maintains over competing deep learning frameworks. Listing 3, shows first a definition of a data type called `ResidualBlock` that contains three members, two of which are constrained to be of a certain type but the last of which is not. By explicitly constraining types, the general Julia compiler can generate optimally packed memory representations, however we can also leave fields unspecified, giving us the ability to insert arbitrary user-defined types. This is used within the `ResidualBlock` example to allow for the `shortcut` (which represents the residual connection within a single convolutional block) to be the identity function, a convolutional function, or any other function of the right shape.

This underscores the hackability and simplicity of defining models in `Flux`. The only explicit constraints we are putting upon this residual block is that the `conv` and `norm` fields are `Tuples` (e.g. they are collections of other objects), and the only implicit constraints we are imposing are that the elements of those `Tuple` fields and the `shortcut` field are callable. In Julia, objects are made callable by defining a function with the object's type as its name which we demonstrate on line 8 of the listing. This further shows how simple it is to use the basic building blocks of Julia within `Flux`, sacrificing neither expressiveness nor speed while still attaining clarity in the implementation.

2.2 Discriminative Adversarial Networks

Nonstandard gradient control is a growing need in the machine learning community as model architectures are stretched to complete more and more complex tasks. A significant recent development has been the emergence of adversarial networks [10], where two networks are arranged in opposition to each other, one attempting to learn a task and the other inventing new inputs that the first can learn from. Extensions of this idea include work done in [32] using `Flux`, where adversarial networks were used to decrease bias induced by dataset imbalance.

The fundamental problem was strong correlation between classification label (in this case, *tuberculosis* versus *non-tuberculosis* coughs) and originating dataset (in this case, *clinical* versus *non-clinical*). The machine learning model naturally learned to key off of the differences in dataset, rather than the differences in the cough sounds themselves, because the differences between clinical and non-clinical recordings were larger than the differences between tuberculosis and non-tuberculosis coughs within a single dataset. Due to the fact that the overwhelming majority of non-tuberculosis coughs were from the non-clinical dataset, the classification algorithm was trapped in a local minimum simply predicting "tuberculosis" for every sample from the clinical dataset.

In order to solve this, a Discriminative Adversarial Network (DAN) was employed, building a second network explicitly designed to determine dataset provenance. This network is then used to "penalize" a shared set of weights (the convolutional block shown in red in Figure 1) for extracting information that can be used to determine which dataset a sample originated from.

```
for x, y_c, y_d in training_set
    y_c_hat, y_dan_hat = model(x)

    c_loss = loss(y_c_hat, y) +
             lambda*loss(y_dan_hat, 1 - y_d)
    d_loss = loss(y_dan_hat, y_d)

    back!(c_loss)
    back!(d_loss)

    opt()
end
```

Listing 4: DAN training loop

The code necessary to perform this nonstandard optimization task is surprisingly simple; shown in listing 4, it simple calculates a forward pass through the model, returning the outputs from the two branches of the model as `y_c_hat` and `y_dan_hat`, calculates the classifier loss (`c_loss`) and the DAN loss (`d_loss`), backpropagates the losses onto the network, then takes an optimizer step (`opt()`). As can be seen, the optimization framework of `Flux` follows very naturally from the basic mathematical principles of optimization and provides for very flexible training and evaluation paradigms.

2.3 Alpha Go

An implementation of Google DeepMind's AlphaGo Zero [33] was recently built in `Flux` [17]. We give in listing 5 the model construction as a demonstration of what a larger-scale model looks like within `Flux`.

The code in listing 5 demonstrates many convenient features of both Julia and `Flux`. `Chain()` provides a sequential model container, passing the output of one layer as the input to the next, and doing the reverse with gradients.

```
tower = [ResidualBlock(
    [256,256,256],
    [3,3],
    [1,1],
    [1,1],
) for i in 1:tower_height]

base_net = Chain(
    Conv((3,3), 2*planes+1 => 256, pad=1),
    BatchNorm(256, relu; momentum = 0.95),
    tower...,
) |> gpu

value = Chain(
    Conv((1,1), 256 => 1),
    BatchNorm(1, relu; momentum = 0.95),
    x -> reshape(x, :, size(x, 4)),
    Dense(N*N, 256, relu),
    Dense(256, 1, tanh),
) |> gpu

policy = Chain(
    Conv((1,1), 256 => 2),
    BatchNorm(1, relu; momentum = 0.95),
    x -> reshape(x, :, size(x, 4)),
    Dense(2*N*N, action_space),
    x -> softmax(x),
) |> gpu
```

Listing 5: AlphaGo Zero model in `Flux`, parameterized by the values of `tower_height`, `planes`, `N` and `action_space`

The `tower...` syntax denotes "splatting" the `tower` object into the `Chain` function call, unpacking the array of `ResidualBlock` objects as separate arguments into the `Chain` call. The `|> gpu` syntax denotes the pipe operator, passing the output of one expression as the input to the next, in this case wrapping the entire expression in a call to `gpu()` which transparently converts the expression to use GPU datatypes and to compile GPU code when run. The `x -> reshape(x, :, size(x, 4))` syntax shows the creation of an anonymous function that dynamically reshapes its inputs to be 2-dimensional, keeping the last axis (which represents batches) but flattening all other axes together.

These listings show, through a combination of Julia language features and intuitive `Flux` abstractions, a remarkably simple yet powerful language for defining models and building differentiable programs.

3. Extending Julia's Compiler

3.1 Compiling Julia for GPUs

Julia supports the basic CUDA programming model for writing GPU kernels [5]. A simple vector addition kernel looks similar to the CUDA C equivalent.

```
function kernel_vadd(a, b, c)
    i = (blockIdx().x-1) * blockDim().x +
        threadIdx().x
    c[i] = a[i] + b[i]
end
```

Listing 6: Basic CUDA programming model expressed in Julia

However, Julia's type specialization enables a powerful set of additional abstractions on the GPU. For example, the code above is not restricted to dense arrays of floats, and could instead be given sparse arrays of complex numbers; Julia's normal specialization mechanisms would generate a new set of PTX instructions for that case. We can even abstract this code further into a "higher-order kernel" that accepts the `+` function (or `*`, or arbitrary user-defined `f`) and thus create a whole family of functions `map(f, x, y)` in four lines of code [14].

Since this works for user-defined types, we can additionally make use of dual numbers for forward-mode differentiation [31]. Compared to reverse mode techniques, dual numbers have the significant advantage of being stack-allocated and interleaving primal and tangent computation, and thus in memory bound situations (such as GPUs) the derivatives are effectively free. This is particularly valuable in functions that contain control flow.

In the case of element-wise operations, such as `c = tanh.(a .+ b)` with vectors of length N, we can see this single $\mathbb{R}^{2N} \to \mathbb{R}^N$ computation as N independent $\mathbb{R}^2 \to \mathbb{R}$ ones, for which forward mode is very efficient. In other words, by running the same computation with dual numbers we efficiently compute the (very sparse) Jacobians $\partial c/\partial a$ and $\partial c/\partial b$ fused with the original operation, which can then be used within the reverse-mode sweep to get gradients. In the base case this allows us to fully fuse elementwise operations like the above, but it also generalizes to complex user-defined functions including control flow, and can be much faster than the equivalent series of vector-level operations that would have to be used otherwise.

3.2 Algorithmic Differentiation

Pushing the limits of reverse-mode differentiation [35], we have also come to see this as a language-level problem. Differentiation is a symbolic transformation that works on programs or symbolic expressions, and this is the domain of compilers. Existing ML frameworks achieve this transformation by *tracing* (or *partial evaluation*); a new tensor type is introduced which records all the basic mathematical operations performed, yielding a graph with the control flow and data structures of the host language elided. This graph, equivalent to a Wengert list [3], is more easily differentiated than the original program.

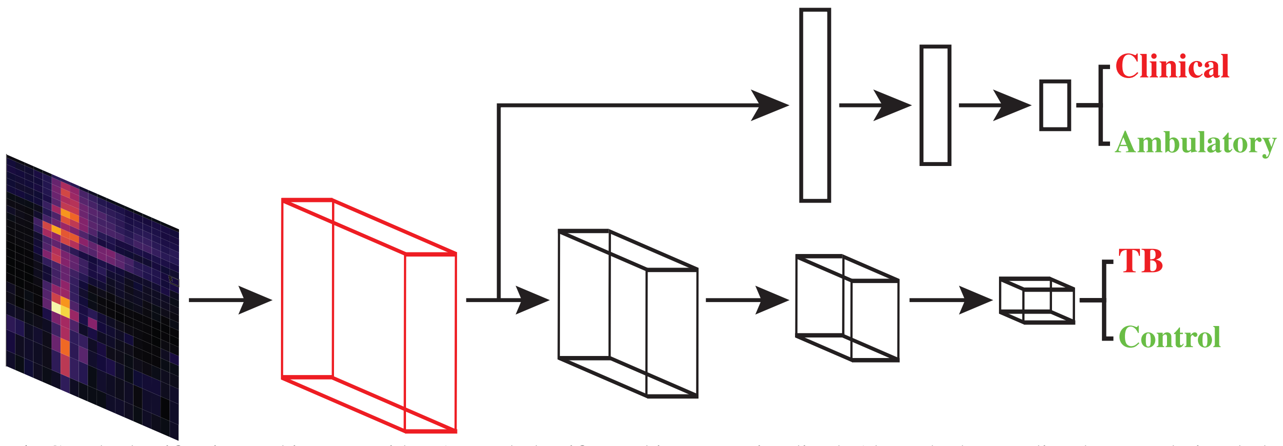

Fig. 1: Cough classification architecture with DAN and classifier architectures visualized. Along the bottom lies the convolutional classification network that classifies cough type, with the outputs from the first layer feeding into a multilayer perceptron that classifies dataset source.

Frameworks are thus divided into the "dynamic" and "static" approaches [24] depending on whether they interpret [22] or compile [4] the recorded graph. However, this presents to us a false dichotomy: we either accept the overhead of an interpreter or freeze user control flow and limit the kinds of models that can be built (perhaps providing limited additional primitives to place control flow into the graph).

We reject these constraints and assert that the "graph" can instead simply be the Julia syntax tree. Extending previous work on differentiable languages [27] we have built `Zygote`, a source-to-source auto-differentiator in Julia, which works directly on SSA-form IR and supports language features like control flow, recursion, data structures and macros, resolving the AD trade-off. By putting the generated SSA-form adjoint code through a traditional compiler such as LLVM [19], we get all the benefits of traditional compiler optimization applied to both our forward and backwards passes. In addition, it opens up the possibility of extending that compiler infrastructure with more advanced and domain-specific optimizations, such as kernel fusion and compilation to accelerators such as TPUs.

3.3 Compiling Julia for TPUs

Google recently introduced a new API that allows users of their Cloud TPU offering to directly generate IR for the *XLA* ("Accelerated Linear Algebra") compiler. This IR is essentially a general purpose IR and optimizing compiler for expressing arbitrary computations of linear algebra primitives and thus provides a good foundation for targeting TPUs by non-Tensorflow users as well as for non-machine learning workloads. We take advantage of Julia's dynamic type system and extendable compiler to build the capability to compile arbitrary Julia code to XLA, and then to run it on TPUs. This allows users to take advantage of the full expressiveness of the Julia programming language in writing their models, including multiple dispatch, higher order functions and existing libraries such as those for differential equation solvers and generic linear algebra routines, while reaping the benefits of the high-performance systolic array engine within the TPU.

The basic workflow of the Julia compiler is to analyze chunks of a Julia program, identify static sub-segments, compile those sub-segments, and run them, returning to a dynamic environment only when necessary to compute, at runtime, the next static sub-segment to be compiled/run. By hooking into this compilation infrastructure, we are able to define custom compiler passes that identify static sub-segments of Julia code and compile them directly to blocks of static XLA IR, which can be run directly on TPUs through the newly exposed API offered by Google.

While many dynamic languages are capable of limited amounts of static analysis, it is worth pointing out that the "static sub-segments" being referred to here are often quite large due to the Julia compiler's aggressive type specialization, type inference and constant propagation. As an illustrative example, we are able to compile the VGG19 [34] machine learning model's forward pass, backward pass and optimization step (that is to say, the entire training loop) into a single chunk of XLA code that can be run on the TPU without breaking out into Julia at all. This technical capability is critical in a heterogeneous computing system such as the TPU, as communication latency between the Julia host process and the TPU accelerator is significant, and must be avoided whenever possible.

3.4 Automatic Batching

The naturally data-parallel structure of many ML models makes them an excellent fit for massively parallel processors such as GPUs [26] and TPUs. To get the most from these accelerators—which can have significant constant overheads per kernel launch, but scale very well over input size—it is common to *batch* them, applying the forwards and backwards passes to multiple training examples at once. In simple models this can be achieved by stacking a set of images or samples along an additional batch dimension, and primitives such as matrix multiply, convolution and broadcasting naturally handle this extra dimension as if all samples were independent.

However, this approach assumes that any control flow in the model follows the same path for each sample in the batch. This is not valid when dealing with more dynamic inputs, such as trees

[36] or graphs [18], or any kind of value-dependent control flow. To see the issue, consider the following scalar function, which we may wish to apply to a batch of scalars at once.

```
function tozero(x)
    if x < 0
        x += 1
    else
        x -= 1
    end
    return x
end
```

Listing 7: `tozero()` is a simple scalar function

To batch this is much more complex than feeding in a vector x, since the condition $x < 0$ is no longer meaningful. Instead we must significantly rework the code to construct an element-wise *mask* `x .< 0`, evaluate both branches, and merge the results of each branch together.

```
function tozero(x)
    cond = x .< 0
    x1 = x .+ 1
    x2 = x .- 1
    x = select(cond, x1, x2)
    return x
end
```

Listing 8: `spmd()` is a version of `notspmd()` that works on batches of inputs

Most researchers address this by taking on the significant burden of batching code by hand. Different solutions have been proposed for different frameworks [25] [20], which heuristically try to batch some high level operations together when possible, using different policies to choose what to batch and when, but these have their own usability issues and typically do not achieve the performance of hand-written code.

We propose that this problem is identical to that of Single Program Multiple Data (SPMD) programming, which has been well-studied by the language and compiler community for decades [7, 2]. Indeed, it is very similar to the model of parallelism used by GPUs internally, and has been implemented as a compiler transform for the SIMD units of CPUs [28]. Taking inspiration from this work, we are implementing the same transform [11] in Julia to provide SPMD programming both for scalar SIMD units and for model-level batching.

By doing so, we are able to take models written for individual samples and transform the IR so that low level operations such as matrix multiplies are run in parallel across the batch. Furthermore we modify the IR so that control flow is handled correctly. Finally, we extend the input types that the original function takes in order to have one extra dimensions, as is currently done when batching images. The case of batching simple models is then a special case of a more general transformation.

This work is ongoing, and will be packaged as a normal Julia library that will give users of `Flux` the ability to painlessly transform their model in such a way that they will be able to train them with mini-batches and fully utilize the power of their parallel hardware.

3.5 JavaScript Compilation

In an approach similar to that of the GPU and TPU compilation efforts, the `FluxJS` package [15] supports compiling Julia models to Javascript, using an approach based on partial evaluation of model code. By defining a small number of fundamental operator equivalencies between Julia and Javascript, this package is able to parse the Julia syntax tree to create a function call graph, including control flow such as in RNNs, and makes use of the `tensorflow.js` [37] package in order to accelerate specific machine learning operations.

This encompasses one of the best ways to showcase a trained `Flux` model, as the web browser platform is one of the most widely available and compatible software platforms in existence. The work of adding new fundamental operations to the Julia to Javascript transpiler is minimal, and thanks to the introspective nature of Julia's code representations, compositions of fundamental operations are themselves transpileable. This hijacking of multiple dispatch within the Julia compiler is flexible, and powerful; as an example, `FluxJS` specializes the `*(x::AbstractArray, y::AbstractArray)` method to convert matrix-matrix and matrix-vector multiplication operations into the `tensorflow.js` equivalents, yielding high-performance dense and LSTM layers.

4. A Library Ecosystem for ML

The inherent design of `Flux`, built on top of the technical foundation of Julia's multiple dispatch semantics, unlocks the ability for `Flux` to act as an extremely flexible "glue" that can bring together disparate packages into a single, differentiable and high-performance program.

4.1 Metalhead

The composable approach to programming that `Flux` offers has yielded great results in the ability for the community to merge disparate packages together into new and exciting combinations. *Metalhead*, a package for standard computer vision models built on top of `Flux`, provides example models such as VGG19 [34], or ResNet [12]. In order to load images from the various computer vision datasets, Metalhead does not define its own image loading code but instead simply uses the `Images` package which yields sufficient information about the loaded images' data layout through the type system so as to enable efficient data loading and transformations. Metalhead provides the basic models and building blocks necessary for modern computer vision research in one convenient location.

4.2 Package management

Integrating composable pieces of code at the unit of "packages" brings about another useful benefit; the Julia packaging system (referred to as Pkg) allows for a researcher to install a set of packages, complete their experiments, then record a manifest file which contains the exact versions and hashes of every package used within the entire project, including the code internal to Julia itself. This allows a second researcher to, months later, load that manifest file and obtain the exact versions of the software used to run the experiments the first time around. The balance of generating reproducible science while maintaining a rapid pace of innovation is a difficult one to achieve, however with the proper tools we believe it is within our grasp.

5. Conclusion

In conclusion, we have presented the reasons why Flux in particular and Julia in general provide an excellent environment for high performance, simple and hackable machine learning. We gave examples of complex models and functions defined in clearly readable, mathematically recognizable forms and detailed many of the techniques used to ensure that the generated code is not only performant on traditional CPUs, but also the accelerators that are increasingly critical to applied machine learning today.

6. References

[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 *OSDI*, volume 16, pages 265–283, 2016.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[3] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1-2):171–190, 2000.

[4] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011.

[5] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing julia on gpus. *arXiv preprint arXiv:1712.03112*, 2017.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[7] Guy E Blelloch. *Vector models for data-parallel computing*, volume 75. MIT press Cambridge, 1990.

[8] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[9] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690, 2018.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[11] Ralf Karrenberg Sebastian Hack, Reinhard Wilhelm, Sandra Neumann, Reinhard Spurk, Johannes Doerfert, Michael Jacobs, Tina Jung, Simon Moll, Fabian Ritter, Roland Leißa, et al. Whole-function vectorization. 2011.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[13] Tim Holy and Contributors. Images.jl: An image library for julia, 2016.

[14] Michael J Innes. Generic gpu kernels, 2017.

[15] Mike Innes. Fluxjs.jl: Flux to javascript compiler, 2018.

[16] Mike Innes, Stefan Karpinski, Viral Shah, David Barber, Pontus Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, et al. On machine learning and programming languages, 2018.

[17] Tejan Karmali. Alphago.jl: Alpha go zero implementation in flux.jl, 2018.

[18] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[19] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[20] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[21] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. *International Conference on Machine Learning*, page 6, 2013.

[22] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[23] Tim Besard Mike Innes and Contributors. Cuarrays.jl: A curious cumulation of cuda cuisine, 2018.

[24] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[25] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3971–3981, 2017.

[26] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[27] 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.

[28] Matt Pharr and William R. Mark. ispc: A spmd compiler for high-performance cpu programming. *2012 Innovative Parallel Computing (InPar)*, pages 1–13, 2012.

[29] PyTorch Team. The road to 1.0: production ready PyTorch. `https://pytorch.org/blog/a-year-in/`, 2018. Accessed: 2018-09-22.

[30] Christopher Rackauckas and Qing Nie. Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, 5(1), 2017.

[31] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

[32] Elliot Saba. *Techniques for Cough Sound Analysis*. PhD thesis, University of Washington, 2018.

[33] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[35] Bert Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980.

[36] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.

[37] Tensorflow Team. Introducing tensorflow.js: Machine learning in javascript, 2018.