MIT 18.337 Final Project Report

Composable Function Transformations for Machine Learning in the Julia Programming Language

Marcel Rød

May 16, 2023

1 Introduction

Over the past decade, the field of machine learning (ML) has significantly expanded, with a notable increase in the last few years. As such, the selection of a programming language and the corresponding development tools has become increasingly important in order to optimize productivity in ML software development. A key factor to consider is the performance of the resulting code. In the context of ML, any inefficiency in the speed of either the training phase or model inference can greatly reduce the utility of the end product. Additionally, ML models often have wide-ranging hardware requirements, from small embedded systems to large-scale GPU clusters.

Julia [1] is a relatively new programming language designed with a focus on performance, possessing several features that make it suitable for ML applications. However, being a newer language, Julia does not yet have the comprehensive library ecosystem that established languages like Python offer. This is particularly evident in the ML domain where Python has become the standard. Despite these challenges, Julia exhibits potential to be a strong contender in the ML space due to its blend of performance and user accessibility.

Nevertheless, there are areas where Julia can improve, especially in terms of ML. This project seeks to address some of these gaps, borrowing from successful strategies that have contributed to Python's dominance in the ML landscape.

2 Background

2.1 Machine Learning

Machine Learning (ML), a branch of computer science, concerns the creation of algorithms that derive insights from data. It aims to construct algorithms that can execute tasks usually challenging to code explicitly, by defining behaviors through parameters that can be learned. This is achieved by training a model using a data set and then employing the trained model for inference. Generally, the model is a function that accepts certain inputs to generate outputs.

The model's training involves the minimization of a loss function, which quantifies the discrepancy between the model's output and the target output. To optimize this loss function, its parameters are iteratively updated using gradient descent. This optimization process necessitates the computation of the gradient of the loss function concerning the model parameters. The backpropagation algorithm [2] efficiently accomplishes this, computing the gradient of a function composed of other functions.

ML has seen a surge in popularity recently and found diverse applications, from image classification to natural language processing. This expansion is largely attributed to the advancement of deep learning, a set of methods employing multi-layered neural networks. Deep learning has thrived primarily due to the escalation of computational power, enabling the enlargement of both our models and our datasets. The shift from single process CPUs to large-scale multi-core and multi-GPU systems has facilitated the development of sophisticated systems that were unthinkable a few years ago.

2.2 Language and Framework Performance

The increase in computational power has led to the need for improved software tools. These tools should not only handle important algorithms, such as matrix multiplications, but also manage tasks, compute gradients, and distribute work across multiple devices.

Python is currently the most used language for ML, mainly due to its large array of libraries providing these tools. Although Python itself is not very fast, its capability to connect with C++ code offers a user-friendly interface while also benefiting from the speed of C++.

Two main Python frameworks, PyTorch [3] and Jax [4], are in competition right now, both presenting this high-level user interface. PyTorch is popular because it's easy to use, simple to experiment with, and convenient for debugging. These attributes have made PyTorch a favorite among ML researchers and practitioners for

years.

On the other hand, Jax prioritizes performance, composability, and simplicity, utilizing the principles of functional programming.

2.3 The Julia Programming Language

Introduced relatively recently, the Julia programming language [1] has quickly gained recognition for its ease of use and speed, particularly within the scientific computing community. Julia's standout feature is its competitive performance, comparable to that of C and Fortran, despite being a high-level language equipped with dynamic runtime features reminiscent of Python and Matlab.

One hurdle for Julia's adoption is its somewhat disjointed ecosystem of libraries, whereas Python's concensus on core libraries a key factor contributing to its dominance in ML. Nevertheless, Julia's development has accelerated in recent years, showing promising progress.

Currently, ML applications in Julia primarily use `Flux.jl` [5], the most popular framework for constructing neural networks within the language.

Julia features a robust type system and supports multiple dispatch, allowing for the creation of generic code that can be tailored for various types.

A distinctive strength of Julia, setting it apart from other languages, is its built-in JIT (Just-In-Time) compiler [6]. This JIT compiler enables dynamically typed code to be specialized into statically typed Julia code at runtime, which can subsequently be compiled into efficient, native machine code.

3 Method

Although much of this section describes problems and potential solutions in Python, we will rewrite them in Julia, and use the column-major convention for matrices for consistency with Julia. This means that standard batching dimensions come last, and that indexing conventions require each index to be specified, i.e. for an $n \times m$ array, `array[1]` does not represent a vector, and we instead need to use `array[1, :]` to get the first m-dimensional row vector.

3.1 Batching

Batching is the process of taking a function that operates on a single element, and transforming it into a function that operates on a set of elements independently. Batching is not a trivial process for several reasons. Firstly, batching exhibits a

```julia
function addfirstlast(x)
    # Return the sum of the first and last elements of a vector
    return x[1] + x[end]
end

data = [1, 2, 3, 4, 5]  # 5-element Vector
addfirstlast(data)  # 6

batched_data = [data data data]  # 5×3 Matrix

addfirstlast(batched_data)  # also 6?
```

Listing 1: Batching done wrong.

trade-off between memory usage parallelism. If we batch a function using a purely sequential loop, the memory allocated by that function can be reused for each iteration of the loop, but in this case, we cannot parallelize the function. If we instead use a parallel loop, we can perform work in parallel, but since each thread does allocations for their own work, we end up needing to allocate a corresponding amount of additional memory. In PyTorch, the process of batching a function is usually one of rewriting a function entirely, and even with this rewrite, managing parallelism is tricky. Instead of this, then, what we want is a way to batch functions that can help us reinterpret the meaning of a program in the light of the batching context of our input data, and in which we can determine the level of parallelism at the latest possible stage.

3.2 Function Transformations

Function transformations are higher order functions that take a function as input, and return a new function as output. While deceptively simple in concept, they lead to powerful abstractions of the operations we often do imperatively, and allow for the creation of complex programs from simple building blocks – essentially decoupling the meaning of the function being called from the structure it is acting on.

A naïve attempt at batching a Julia

The code in Listing 1 is incorrect, but does produce a result, and shows a weakness of the flexibility of Julia. When applied to a matrix, a data structure that was not considered while writing the function `addfirstlast`, Julia adds the first and last

elements of the matrix together, meaning element `1,1` and `end,end`. In order to

We solve these issues by introducing a function called `vmap`. `vmap` is similar to the well known `map`, but suited uniquely for array programming. The arguments to `vmap` are a function (to be batched), a tuple representing the axes to be batched over for all input arguments, and a tuple representing the axes to be batched over for all output arguments.

In order to generalize batching of functions to allow for arbitrarily nested structs of arrays, we not only allow for arguments to be arrays, but also arbitrary nested tuples of arrays. This functionality is facilitated by a project by the *JuliaCollections* project, named `AbstractTrees.jl`.

3.3 Composability

Batching using functional transformations leads to a very compositional way of writing code. One can write a function based on its minimal functionality, and then transform and compose this function to act on data of any format. For instance, consider this way of building a matrix multiplication method:

1. Define a function `multiply` that multiplies two elements. This function knows nothing about batching, and is defined only for floating point numbers.
2. Batch the `multiply` function and compose it with `sum` to create a `dot` function.
3. Batch `dot` to create a matrix-vector multiplication.
4. Batch the matrix-vector multiplication to create a matrix-matrix multiplication.

This approach is implemented in Listing 2. One can further batch this function to construct any variety on the "batched matrix multiplication," which is often specifically implemented by most frameworks, called `torch.bmm` in PyTorch [7]. A great advantage of doing this using a function like sum instead of writing a for loop is that it is made clear to the compiler that reordering the summation does not change the result, and so the compiler is free to reorder the summation to optimize cache usage. Note that this benefit is also present in the common Julia method of writing matrix multiplications, but is often not standard when doing sums in the later stage of an algorithm.

```
# (), () -> ()
function multiply(x, y)
    return x * y
end

# (n,), (n,) -> ()
dot = sum ∘ @vmap(multiply)

# (m, n), (n,) -> (m,)
matrix_vector_multiply = @vmap(dot, (1, nothing))

# (m, n), (n, p) -> (m, p)
matrix_matrix_multiply = @vmap(matrix_vector_multiply, (nothing, :last))

# (b, m, n), (b, n, p) -> (b, m, p)
batched_matrix_matrix_multiply = @vmap(matrix_matrix_multiply, (:last, :last))
```

Listing 2: Matmul implemented with only sum and multiply

3.4 Performance and Reconfigurability

Functional transformations allow both library developers and users to write code that is both performant and reconfigurable. A library developer is then free to write only the most basic functions, and then let the user compose them into more complex variations with batching. A good example is that of the dot product implemented in Listing 2 – any version of matrix multiplication breaks down into a batched set of dot products, so a library implementation of these could supply just a dot function, composed of a sum and a batched multiplication. This is not currently possible with the dominant deep learning frameworks. It turns out that most array operations are very simple to express using our vmap construct, and with the addition of a gradient transform one can write compact, fast, readable, and reconfigurable code with little effort.

Consider computing the gradients of the loss function in a neural network with respect to one of the parameters, which is one of the most common operations in deep learning. In PyTorch, one would compute gradients by performing a forward pass through the neural network, and then backpropagating the gradients through the network, however this can only be done cumulatively (adding up gradients over elements in a batch), or individually. Thus, in order to compute gradients for multiple

```julia
function softmax(x)
    exp_x = exp.(x)
    exp_x ./ sum(exp_x)
end

# First batch then gradient
# (n, b) -> (n, b, n, b)
softmax_batched_grad = @grad @vmap softmax

# First gradient then batch
# (n, b) -> (n, n, b)
softmax_grad_batched = @vmap @grad softmax
```

Listing 3: Computing gradients for multiple elements using `vmap`.

elements, we have to trace the forward pass multiple times, and then backpropagate each of these traces, which is very inefficient. In Julia one would also have to repeat the forward and backward passes once for each of these gradient samples.

With functional transformations, we can write a function that computes the gradients of a single element, and then batch this function to compute the gradients of multiple elements. This is fundamentally a different computational task than the one PyTorch is performing, and is much more efficient.

In PyTorch's current prototype implementation of functional transformations they show that even for small neural networks, batching the gradient computation leads to a roughly 9x faster computation of gradients [8]. There is no other viable approach in PyTorch. Listing 3 shows the implementation of this operation in Julia using `vmap`.

Reconfiguring the computational mapping of the program without changing the program itself is also incredibly powerful for an end user.

3.5 Batching and Multi-Device Parallelism

Batching is not only a way to generalize code to work on multiple elements, but it is also a great way of expressing independent compute, and thus to determine parallelism. Batched operations are almost by definition embarrassingly parallel, as each element in the batch manipulated independently from the others. Batching along the outermost (first in column-major convention) dimension of an array is also

```
# (n, ) -> (n, )
function softmax(arr)
    exp_arr = exp.(arr)
    return exp_arr ./ sum(exp_arr)
end

# (n, b) -> (n, b)
# Name the axis to be batched over
softmax_batched = @vmap(softmax, (:last,), name=:softmax_axis)

# (n, b) = (10, 128)
array = randn(10, 128)

# I want 8 parallel threads each doing 16 sequential batches
# 8 * 16 = 128
result = nothing
run_context(;softmax_axis=(parallel=8, sequential=16)) do
    result = softmax_batched(array)
end

# I want 2 devices (GPUs) to run 32 parallel threads with 2 sequential batches
# 2 * 32 * 2 = 128
result = nothing
run_context(;softmax_axis=(device=2, parallel=32, sequential=2)) do
    result = softmax_batched(array)
end
```

Listing 4: A batched function can be run with various setups for parallelism. `run_context` determines parallelism for the supplied function or closure.

a good way of coaxing SIMD-vectorization out of a compiler, as it guarantees that the data is contiguous in memory.

Consider the example given in Listing 5. We have a set of neural networks that all share the same prefix, before branching off into different heads. Our code is already batched with respect to the data, so a single batched data input is fed into the network. Because of this structure, we can compute the common prefix between these networks once, and then branch off into different heads when necessary. This is a common pattern when working with partially frozen neural networks or ensemble methods. If we have our neural network in the form of an `AbstractTree` or just a `NamedTuple` of parameters, we can achieve this with a single application of `vmap`.

3.6 Implementation

We implement `vmap` as a typed final interpreter [9] by wrapping arrays in a `vmap` tracing type called `BatchedArray` containing the array and the batch index.

We define batching rules for operations on `BatchedArrays` based on if they are simple unary operations, binary operations, or reductions. These rules are very simple due to the flexibility of Julia's type system, multiple dispatch and the broadcasting system.

Now, when a `vmap`-ed function is applied to an array, the array gets wrapped in a `BatchedArray`, which will be passed into applied function. When operations are performed on the `BatchedArray`, they respect the surrounding batching dimensions, and permute the dimensions of the resulting arrays to match the vmap output dimensions.

For the `run_context`, we set global context flags that can be seen by the `vmap` function, and which determine looping shapes and parallelism. An interesting consequence of this implementation is that Julia can JIT compile these shapes into the same binaries as those produced by an ahead-of-time compiler using static settings for loop sizes and parallelism. The `device` option is not yet implemented, since multi-gpu support is currently a bit tricky in Julia.

4 Results

4.1 Simplification of Code and Expressivity

The difference in brevity, readability and flexibility between batching code by standard means of looping assignments and the `vmap` functional transformation is already

```julia
function linear(x; W, b)
    W * x .+ b
end

function model(x; params)
    linear(x; params.W, params.b)
end

# Change this function independent from data layout
loss_pair = abs2

function loss(x, y; params)
    ŷ = model(x; params)
    # Batching is handled when needed
    (sum ∘ @vmap(loss_pair))(y, ŷ)
end

# A single model
params = (W = randn(10, 5), b = randn(10))
loss(x, y; params)

# An ensemble of 20 models, only differing in b
ensemble_params = (W = randn(10, 5), b = randn(10, 20))

# Compute the losses for the models of the ensemble
# Returns a 20-element vector
(@vmap loss (nothing, nothing, (nothing, :last)))(x, y; params=ensemble_params)
```

Listing 5: Batching of a neural network using `vmap`.

quite clear. There is however additional utility in the simplification of APIs by defining smaller functions with more generic functionality, and composing these functions together. When composing a program out of `vmap`s, we can very simply determine the level of parallelism and thus also memory usage without tampering with the logic of the program itself.

4.2 Performance

In subsection 3.4, we saw how PyTorch has vastly improved the performance of a common and fundamental operation with their initial implementation of functional transformations. In Julia most of these problems are solved by the compiler and the approaches Julia's ecosystem has taken to differentiable programming. However, the advantages of functional transformations are still present in this context through the savings made by avoiding prefix computation and other features like the per-sample gradients seen in Listing 3. Performance is also gained through simple configurability of the parallelism and memory usage of a program, as seen in Listing 4. The result reusable code that can be tuned to any individual use-case without making any sacrifices in performance.

5 Discussion

5.1 Future Work

There are many possible avenues for future work in using compositional functional transformations in Julia. I would firstly like to see a complete implementation of a gradient transform that can take into account shapes to determine the best possible combination of forward and backward passes for gradient computation. Further, I would like to see `vmap` implement all of its functionality entirely at JIT compile time. Currently, determining the shapes to use for parallelism and memory allocation is done at runtime, but there is no reason other than implementation complexity that this cannot be done at compile time. I also think it would be possible to determine the optimal parallelism schedule automatically given a whole function composed out of `vmap`s. In this case the user could supply a memory budget and ask for the fastest possible execution within that budget. I think these are all solvable problems and would like to pursue them further in the future.

5.2 Conclusion

Batching is the process of taking a function that operates on an array and transforming it into a function that operates on a batch of arrays. Standard approaches of batching make hidden assumptions and impose restrictions on code that makes it hard to compose, and hard to optimize. Function transformations, like `vmap`, are a way of automatically deriving functions from other functions, and are a great way to increase code reuse, composability and performance. We have shown how functional transformations can be used to simplify batching behaviors, and move decisions about layout and computational mapping to the last possible moment, enabling faster development, more readable code, better performance, and multi-device parallelism with little effort required from the user.

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, ISSN: 0036-1445. DOI: 10.1137/141000671. [Online]. Available: https://epubs.siam.org/doi/10.1137/141000671.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 6088, ISSN: 1476-4687. DOI: 10.1038/323533a0. [Online]. Available: https://www.nature.com/articles/323533a0.

[3] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[4] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.2.5. [Online]. Available: http://github.com/google/jax.

[5] M. Innes, E. Saba, K. Fischer, *et al.*, "Fashionable modelling with flux," *CoRR*, vol. abs/1811.01457, arXiv: 1811.01457. [Online]. Available: https://arxiv.org/abs/1811.01457.

[6] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, ISSN: 0360-0300. DOI: 10.1145/857076.857077. [Online]. Available: https://dl.acm.org/doi/10.1145/857076.857077.

[7] "Torch.bmm — PyTorch 2.0 documentation." (), [Online]. Available: https://pytorch.org/docs/stable/generated/torch.bmm.html.

[8] "Per-sample-gradients — PyTorch Tutorials 2.0.1+cu117 documentation." (), [Online]. Available: https://pytorch.org/tutorials/intermediate/per_sample_grads.html.

[9] O. Kiselyov, "Typed Tagless Final Interpreters," in *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, ser. Lecture Notes in Computer Science, J. Gibbons, Ed., Berlin, Heidelberg: Springer, pp. 130–174, ISBN: 978-3-642-32202-0. DOI: 10.1007/978-3-642-32202-0_3. [Online]. Available: https://doi.org/10.1007/978-3-642-32202-0_3.