Chapter 1

TensorFlow.jl: An Idiomatic Julia Front End for TensorFlow

This paper is currently under review for the Journal of Open Source Software.

1.1 Summary

TensorFlow.jl is a Julia (Bezanson et al. 2014) client library for the TensorFlow deep-learning framework (Abadi et al. 2015; Abadi et al. 2016). It allows users to define TensorFlow graphs using Julia syntax, which are interchangable with the graphs produced by Google's first-party Python TensorFlow client and can be used to perform training or inference on machine-learning models.

Graphs are primarily defined by overloading native Julia functions to operate on a TensorFlow.jl `Tensor` type, which represents a node in a TensorFlow computational graph. This overloading is powered by Julia's powerful multiple-dispatch system, which in turn allows allows the vast majority of Julia's existing array-processing functionality to work as well on the new `Tensor` type as they do on native Julia arrays. User code is often unaware and thereby reusable with respect to whether its inputs are TensorFlow tensors or native Julia arrays by utilizing *duck-typing*.

TensorFlow.jl has an elegant, idiomatic Julia syntax. It allows all the usual infix operators such as `+`, `-`, `*` etc. It works seamlessly with Julia's broadcast syntax as well, such as the `.*` operator. Thus `*` can correspond to matrix multiplication while `.*` corresponds to element-wise multiplication, while Python clients needs distinct `@` (or `matmul`) and `*` (or `multiply`) functions. It also allows Julia-style indexing (e.g. `x[:, ii + end÷2]`), and concatenation (e.g. `[A B]`, `[x; y; 1]`). Its goal is to be idiomatic for Julia users while still preserving all the power and maturity of the TensorFlow computational engine.

TensorFlow.jl aims to carefully balance between matching the Python TensorFlow API and Julia conventions. In turn, the Python TensorFlow client is itself designed to closely mirror numpy. Some examples are shown in the table below.

Julia	**Python TensorFlow**	**TensorFlow.jl**
1-based indexing	0-based indexing	1-based indexing
Column Major	Row Major	Row Major
Explicit broadcasting	Implicit broadcasting	Implicit or explicit broadcasting
Last index at `end`, 2nd last in `end-1`	Last index at `-1`, second last in `-2`	last index at `end`, 2nd last in `end-1`
Operations in Julia ecosystem namespaces. (`SVD` in `LinearAlgebra`, `erfc` in `SpecialFunctions`, `cos` in `Base`)	All operations TensorFlow's namespaces (`SVD` in `tf.linalg`, `erfc` in `tf.math`, `cos` in `tf.math`, and all reexported from `tf`)	All hand imported Operations in the Julia ecosystems namespaces. (`SVD` in `LinearAlgebra`, `erfc` in `SpecialFunctions`, `cos` in `Base`) Ops that have no other place are in `TensorFlow`. Automatically generated ops are in `Ops`
Container types are parametrized by number of dimensions and element type	N/A: does not have a parametric type system	Tensors are parametrized by element type, enabling easy specialization of algorithms for different types.

Defining TensorFlow graphs in the Python TensorFlow client can be viewed as metaprogramming, in the sense that a host language (Python) is being used to generate code in a different embedded language (the TensorFlow computational graph) (Innes et al. 2017). This often comes with some awkwardness, as the syntax and the semantics of the embedded language by definition do not match the host language or there would be no need for two languages to begin with. Using TensorFlow.jl is similarly a form of meta-programming for the same reason. However, the flexibility and meta-programming facilities offered by Julia's macro system makes Julia especially well-suited as a host language, as macros implemented in TensorFlow.jl can syntactically transform idiomatic Julia code into Julia code that constructs TensorFlow graphs. This permits users to reuse their knowledge of Julia, while users of the Python TensorFlow client essentially need to learn both Python and TensorFlow.

One example of our ability to leverage the increased expressiveness of Julia is using `@tf` macro blocks implemented in TensorFlow.jl to automatically name nodes in the TensorFlow computational graph. Nodes in a TensorFlow graph have names; these correspond to variable names in a traditional programming language. Thus every operation, variable and placeholder takes a `name` parameter. In most TensorFlow bindings, these must be specified manually resulting in a lot of code that includes duplicate information such as `x = tf.placeholder(tf.float32, name="x")` or they are defaulted to an uninformative value such as `Placeholder_1`. In TensorFlow.jl, prefixing a lexical block (such as a `function` or a `begin` block) with the `@tf` macro will cause the `name` parameter on all operations occurring on the right-hand side of an assignment to be filled in using the left-hand side. For example, the TensorFlow.jl equivalent of the above example is `@tf x = placeholder(Float32)`. Note how `x` is named only once instead of twice, as is redundantly required in the Python example. Since all nodes in the computational graph can automatically be assigned the same name as the corresponding Julia variable with no additional labor from TensorFlow.jl users, users get for free more intuitive debugging and graph visualisation.

Another example of the use of Julia's metaprogramming is in the automatic generation of Julia code for each operation defined by the official TensorFlow C implementation (for example, convolutions of two TensorFlow tensors). The C API can be queried to return definitions of all operations as protocol buffer descriptions, which includes the

expected TensorFlow type and arity of its inputs and outputs, as well as documentation. This described the operations at a sufficient level to generate the Julia code to bind to the functions in the C API and automatically generate a useful docstring for the function,. One challenge in this is that such generated code must correct the indices to be 1-based instead of 0-based to accord with Julia convention. Various heuristics are employed by TensorFlow.jl to guess which input arguments represent indices and so should be converted.

TensorFlow.jl ships by default with bindings for most operations, but any operation can be dynamically imported at runtime using `@tfimport OperationName`, which will generate the binding and load it immediately. Additionally, for operations that correspond to native Julia operations (for example, `sin`), we overload the native Julia operation to call the proper binding.

We also use Julia's advanced parametric type system to enable elegant implementations of array operations not easily possible in other client libraries. TensorFlow.jl represents all nodes in the computational graph as parametric `Tensor` types which are parameterized by their element type, e.g. `Tensor{Int}`, `Tensor{Float64}` or `Tensor{Bool}`. This allows Julia's dispatch system to be used to simplify defining some bindings. For example, indexing a `Tensor` with an `Int`-like Tensor will ultimately create a node corresponding to a TensorFlow "gather" operation, and indexing with a `Bool`-like Tensor will correspond to a "boolean_mask" operation. It is also used to cast inputs in various functions to compatible shapes.

1.1.1 Challenges

The TensorFlow 1.0 C API primarily exposes low-level functionality for manually managing nodes in the computation graph. Gradient descent optimizers, RNNs functionality, and (until recently) shape-inference all required reimplementation on the Julia side. Most challengingly, the symbolic differentiation implemented in the `gradients` function is not available from the C API for all operations. To work around this, we currently use Julia's Python interop library to generate the gradient nodes using the Python client for those operations not supported by the C API. This requires serializing and deserializing TensorFlow graphs on both the Julia and Python side.

This has been improving over time, both due to Google moving more functionality from the Python TensorFlow client to the C API which can reused by Julia, and with more reimplementations of other aspects of the Python client from our own volunteer efforts. There nevertheless remains a large number of components from the upstream `contrib` submodule that remain unimplemented, including various efforts around probabilistic programming.

1.1.2 Other deep learning frameworks in Julia

Julia also has bespoke neural network packages such as Mocha (Zhang 2014), Knet (Yuret 2016) and Flux (Innes 2018), as well as bindings to other frameworks such as MxNet (Chen et al. 2015). While not having the full-capacity to directly leverage some of the benefits of the language and its ecosystem present in the pure Julia frameworks such as Flux, TensorFlow.jl provides an interface to one of the most mature and widely deployed deep learning environments. It naturally therefore supports TensorFlow visualization libraries like TensorBoard. It also gains the benefits from any optimisations made in the graph execution engine of the underlying TensorFlow C library, which includes extensive support for automatically distributing computations over multiple host machines which each have multiple GPUs.

1.1.3 Acknowledgements

- We gratefully acknowledge the 30 contributors to the TensorFlow.jl GitHub repository.

- We especially thank Katie Hyatt for contributing tests and documentation.
- We thank members of Julia Computing and the broader Julia Community for various discussions, especially Mike Innes and Keno Fischer.