# Language Binding Considerations for an Application Server Proxy Herd Running TensorFlow Machine Learning Algorithms

Naim Ayat, *UCLA Department of Computer Science*

## Abstract

A Python application server proxy herd runs on a large set of virtual machines and employs machine learning algorithms utilizing TensorFlow. Following a series of performance assessments, it is discovered that the Python implementation is suboptimal in that its bottleneck is the Python language itself rather than TensorFlow's machine learning framework. Hence, three alternative languages are studied and evaluated as potential server herd foundations: (1) Java, (2) OCaml, (3) Julia. It is ultimately determined that Julia yields the most cost-efficient performance and dependability benefits, as well as the superior TensorFlow API.

## 1. Introduction

The objective is to design and operate an efficient application server proxy herd on a large set of virtual machines. The application in question utilizes machine learning algorithms based in the TensorFlow 1.6.0 dataflow programming library. A prototype application has been developed in Python 3.6.4; it runs well on large queries. However, the prototype is unusual in that it handles many small queries – each of these creates and executes models which are relatively minute by machine learning standards.

Typically, TensorFlow applications are bottlenecked inside C++ or CUDA code. However, benchmark tests reveal that the prototype spends the majority of its process time executing Python code to initialize machine learning models. Hence, three alternative languages will be considered for a new prototype in an attempt to optimize performance of the application server proxy herd:

1. Java 8.161
2. OCaml 4.06.0
3. Julia 0.6.2.

## 2. Remarks on TensorFlow

### 2.2. Architecture

The TensorFlow architecture is built for large-scale distributed machine learning. Its runtime is a cross-platform library; a C API is used to separate user-level code in different languages from the core runtime. Users of TensorFlow write the client –a program that builds the *computation graph*, TensorFlow's data structure for semi-supervised machine learning.

As stated in Section (1), TensorFlow supports multiple client languages, though it is worth noting that "Python and C++ [are prioritized]." Most of its libraries are exclusively available in Python; they are ported to C++ as they "become more established."[12]

The TensorFlow runtime contains over two hundred standard operations, many of which have kernel implementations which utilize `Eigen::Tensor`, a C++ module that serves to generate efficient concurrent programs for multicore systems. However, when possible, the NVIDIA CUDA Deep Neural Network library (cuDNN) is used in its place to achieve higher efficiency. Assuming user-level code is optimal, these kernel implementations are expected to be the bottleneck of a TensorFlow-powered application server proxy herd due to the comparatively high overhead of `Eigen::Tensor` and cuDNN functions.[6]

### 2.2. Support for Multiple Languages

TensorFlow originally supported Python alone. It has now expanded, but Python remains as the language that offers the most features. However, these features are gradually being relayed into TensorFlow's C++ core and exposed via the C API. Programmers may use a language's foreign function interface (and in Java, the Java Native Interface) to interact with this API.

As new language support must be built on the C API, some features of TensorFlow are not available to users who opt for a language other than Python. For example, the *gradient, function, control flow,* and *neural network* library features of the machine learning framework are not yet present in the TensorFlow core and therefore unusable in non-Python applications.[9]

Nevertheless, the machine learning framework is completely open-source; it is theoretically possible for a programmer to add such functionalities on his or her own initiative. However, this clearly would require much further research on the TensorFlow codebase than otherwise necessary. Therefore, at face-value, it is ideal to use Python when working on projects that necessitate the use of extensive TensorFlow features. Assuming there is no time constraint or need for the large majority of TensorFlow's functionality, it becomes logical for alternate languages to be considered.

## 3. Java Implementation

### 3.1. General Remarks on Performance

Because the application server proxy herd runs on a large set of virtual machines, each server should ideally be able to handle concurrent tasks, such as getting and transmitting data. The most efficient implementation of a horizontally-scalable server configuration is, therefore, one that makes use of multithreading – for which the Java interface offers the most extensive interface.[8]

The introduction of multithreading, however, introduces issues such as race conditions, deadlocks, and starvation. These complications make designing an application server herd in Java inherently more demanding than doing so with Python's `asyncio` module, though the performance gain over a large configuration of multicore machines will be significant.

A single-threaded, event-driven approach in Java would likely involve the `Node.js` framework, which is slower than `asyncio` regardless of platform.[4] Also, note that Java is compiled while Python is interpreted. Because interpreted programs must be converted to machine code at runtime, they are generally slower than compiled programs during their first execution. However, since this is only an issue in the first execution alone, there is no clear performance benefit for a Java application server proxy herd implementation.

### 3.2. Static Type Checking

While Python is easier to implement, Java's static type checking makes it the more reliable language for production server code. Since types are addressed at compile time, the risk of undefined behavior (or complete server crashes) is significantly mitigated. However, statically-typed programs are fundamentally much more verbose than their dynamically-typed counterparts. Java applications, as a result, typically take much longer to develop than Python applications that accomplish the same purpose.[1]

### 3.3. TensorFlow Java API

Clearly, there is a tradeoff – Java provides dependability at the cost of Python's ease-of-use and flexibility. In regard to TensorFlow, it is certainly more straightforward to use Python due to it being the natively supported language.

Still, TensorFlow officially recognizes an experimental Java API. Its documentation is noticeably less rigorous than that of the original framework and backwards-compatibility with future TensorFlow releases is not guaranteed. However, it offers a thread-safe `Graph` class for concurrent TensorFlow computation. In a multithreaded application server proxy herd, this is quite useful.

Due to the absence of *gradient*, *function*, *control flow*, and *neural network* features from TensorFlow's C++ core, these functions are not available in the Java API. However, they are not strictly necessary for the development of an application server herd. Hence, Java is the ideal language for server dependability; although a Java rewrite of the Python prototype would undoubtedly be time-inefficient and unlikely to yield any significant performance gain.[13]

## 4. OCaml Implementation

### 4.1. OCaml Type System

One of OCaml's greatest advantages is its unique type system. Like Java, it is statically typed, meaning type errors are detected at compile time. Once more, like Java, OCaml is strongly typed, hence the compiler conveniently detects type mismatches before execution.

Where the language truly shines is in its type inference system. Despite being strongly and statically typed, it does not require the programmer to repeatedly or explicitly declare expression types. Rather, types in OCaml are completely inferred; thus, code is much more succinct when compared to Java.[5]

### 4.2. Programming Style

OCaml is heavily reliant on recursion – iterative loops are not frequently employed as in Java or Python. This allows the language to be quite powerful and even more compact than previously acknowledged. As a result, OCaml programs are incredibly lightweight and memory efficient when compared to both Java and Python.

By nature of recursive code, however, OCaml disappoints in its readability. In the context of this study, readability is not a crucial issue (since the application server proxy herd is being developed by a single programmer). If the code is to be revisited or collaborated on, however, the choice of OCaml as the server language will likely introduce a learning curve that would otherwise not be present with Java or Python implementation.

### 4.3. TensorFlow OCaml Bindings

The most popular open-source TensorFlow OCaml bindings are not officially recognized, however, they offer the same functionality as the Java API and closely follow the original TensorFlow API. Hence, their lack of rigorous documentation is not as glaring an issue as with the Java API, although learning resources are still limited when compared to Python.

The OCaml bindings offer the FNN API, a layer-based API for easily building neural networks. Artificial intelligence constructs such as these are naturally simpler to implement in functional languages such as OCaml, so the machine learning component of the application server herd can be optimal in terms of brevity.

It is worth noting that the developer of the OCaml bindings warns that the API is susceptible to segmentation faults, so the choice of OCaml is not unconditionally reliable.[11] However, it is the most concise and memory efficient.

## 5. Julia Implementation

### 5.1. Julia Type System

Julia types are a property of values, not expressions; the language does not assign types to expressions before evaluating them. Hence, it is a dynamically typed language by definition. However, its type system syntax is relatively concise when compared to that of Java. Moreover, it is weakly-typed, meaning that type definitions are not strictly necessary.[3] Writing type-stable code sometimes requires careful deliberation on behalf of the programmer, but it yields exceptional performance benefits. Type-stable Julia code reaches speeds near that of statically-compiled languages, making it marginally faster than Python and similarly concise.

### 5.2. Unique Features and TensorFlow Compatibility

Julia offers the *PyCall* package, which provides the ability to directly call and fully interoperate with Python. Python modules may be imported from Julia, and Python functions can be called with automatic conversion of types.[7] Similarly, Julia's "no boilerplate" philosophy allows it to call C functions without the need for wrappers or special APIs. This is easily accomplished with the `ccall` syntax; making development of multiple-language projects incredibly simple.[2]

Further, Julia is compiled rather than interpreted, giving it another slight edge over Python in its initial execution and matching Java in terms of dependability.

The most popular TensorFlow Julia API is `TensorFlow.jl`, which is nearly identical to the Python API. Almost all functionality of TensorFlow is exposed, barring *distributed graph execution*, *control flow*, and *graph visualization*. However, Julia's unique ability to work effortlessly with Python and C make it possible to communicate with TensorFlow directly – essentially in the same manner that Python allows.[10]

## 6. Conclusion

Although Java offers a similar reliability and performance boost to Julia, its type system makes development of an application server proxy herd needlessly verbose. OCaml, on the other hand, is brief yet powerful – unfortunately, its TensorFlow API is unreliable.

Therefore, it is strongly recommended that the application server proxy herd be rewritten in Julia using `TensorFlow.jl`. The Julia language is nearly functionally identical to Python in terms of its utilization of the TensorFlow API; therefore, redevelopment should not be exceptionally labor-intensive. Moreover, Julia's dynamically and weakly-typed approach to type checking guarantee it to operate faster than Python in the context of a well-written, type-stable event-driven server.

## 7. References

[1] "Dynamic Typing vs. Static Typing." *Oracle Docs*. N.p., n.d. Web. 13 Mar. 2018.

[2] Ekre, Fredrik. "The Julia Language." *Calling C and Fortran Code · The Julia Language*. N.p., 2 June 2017. Web. 13 Mar. 2018.

[3] Ekre, Fredrik. "The Julia Language." *Performance Tips · The Julia Language*. N.p., 2 June 2017. Web. 09 Mar. 2018.

[4] Flaxman, Michael. "Python 3's Killer Feature: Asyncio." *PAXOS*. N.p., 21 June 2017. Web. 10 Mar. 2018.

[5] Furr, Michael, and Jeffrey S. Foster. "Checking type safety of foreign function calls." *ACM SIGPLAN Notices*. Vol. 40. No. 6. ACM, 2005.

[6] Hwang, Kai. "TensorFlow, Keras, DeepMind, and Graph Analytics." *Cloud Computing for Machine Learning and Cognitive Applications*. N.p.: MIT, 2017. 479-80. Print.

[7] Johnson, Steven G. "JuliaPy/PyCall.jl." *GitHub*. N.p., 08 Mar. 2018. Web. 13 Mar. 2018.

[8] "Lesson: Concurrency." *The Java Tutorials Essential Classes*. N.p., n.d. Web. 11 Mar. 2018.

[9] Malakoff, Kevin. "Gradients in C and C++ APIs · Issue #6268 · Tensorflow/tensorflow." *GitHub*. N.p., 12 Dec. 2016. Web. 13 Mar. 2018.

[10] Malmaud, Jon. "Malmaud/TensorFlow.jl." *GitHub*. N.p., 20 Jan. 2018. Web. 13 Mar. 2018.

[11] Mazare, Laurent. "LaurentMazare/tensorflow-ocaml." *GitHub*. N.p., 18 Feb. 2018. Web. 10 Mar. 2018.

[12] "TensorFlow Architecture." *TensorFlow Documentation*, 2 Mar. 2018, www.tensorflow.org/extend/architecture.

[13] TensorFlow for Java. "Tensorflow/tensorflow." *GitHub*. N.p., 10 Mar. 2018. Web. 13 Mar. 2018.