Analyzing the Alternatives to Python for TensorFlow

Abstract

Following up with implementing a server herd to store and deal with client requests, we want to create a server heard to work with an application that uses machine learning algorithms from the TensorFlow library. However, because Python, the original language the server herd is implemented with, is too slow, we want to rewrite the server application into another language keeping the same functionality while keeping TensorFlow a part of the new translated application.

1. TensorFlow Details

Generally, TensorFlow can be described as an open-source library with bindings to multiple languages that provides the means for numeric computation and functionality specific to machine learning. Typically, the client would "write the client TensorFlow program that builds the computation graph" [1]. In other words, assuming that the "client" and "distributed master" layers are separate over a network, we would be having our application server take TensorFlow requests from clients and parse the requests by setting up and parsing models, then returning the data from the models.

TensorFlow was originally designed for Python and run inside C++ and CUDA code, meaning that it supports all of TensorFlow's functionality. For other languages, they are to call the FFI (foreign function interface) in order to access the functionality through the C API layer that separates user-level code and the TensorFlow runtime.

1.1. Problems with Python

In our specific case, Python is said to spend too much time creating the models for the client requests. In particular, if the client requests is small, the overhead from creating each model to satisfy the client's request would be too significant, making up a large part of the runtime's response.

There are some possible explanations to why this is the case in Python. First of all, because Python is dynamically-typed, the program on the server must check the type of every variable during runtime. However, by doing so, execution speed would be lower, and would be a major hit on the application's performance. For memory management, Python makes use of both reference counting and mark-and-sweep operations. With reference counting, there would be an overhead every time the application were to create a new object, or delete a reference to object. Because of both dynamic type checking and reference counting, instantiating a TensorFlow model would of have a higher overhead, meaning that for smaller requests, it would be more efficient to skip doing so if possible. In addition, Python also does not support parallelism with multiple threads (besides multi-processing) (due to the GIL [global interpreter lock] in CPython and other builds), so it would be difficult to make use of shared memory while making use of multiple CPUs/cores.

2. Alternatives: Java, OCaml, and Kotlin

So, because of the downsides of Python, we would look at other languages to write our application with. In our case, because we want to preserve the TensorFlow functionality in our application, we would look at other languages that TensorFlow supports. Here, we would look at three potential other languages: Java, Ocaml, and Kotlin.

2.1. Java

For Java, performance-wise, the language is a much better alternative to Python. There are three reasons why Java runs more efficiently than Python: it is statically-typed, it's garbage collector is more efficient, and it supports better parallelism.

For being statically-typed, Java doesn't need to always check the types of its variables during runtime. Just with this, the runtime execution speed of Java is already faster and less error prone than that of Python. Although Java is strongly typed like Python, it requires that the programmer define the type of variable explicitly, increasing user readability. In addition, when the Java compiler compiles the source code into bytecode, the compiler can check each type and catch them before runtime, meaning that there would be less runtime errors and overhead from type checking.

For memory management, Java uses a mark-sweep-compact garbage collection system that not only has less overhead than Pythons reference counting, but the compacting aspect also allows for better caching of values in heap. Both of these factors are already a huge boon for Java's performance in comparison to Python.

Finally, unlike Python, Java does support true parallel multithreading, meaning that it can make use of parallelism using multiple CPUs/cores while taking advantage of the shared memory that comes with multithreading.

Another major thing to consider not regarding the language itself is TensorFlow's Java support. Unlike Python, Java support isn't as great, as "the TensorFlow Java API is not covered by the TensorFlow API stability guarantees" [3]. In other words, the library wouldn't perform as robustly as it would in Python.

2.2. OCaml

OCaml is a statically typed language, meaning that, like Java, it's runtime wouldn't be hampered by dynamic type checking like in Python. For runtime, there would no error regarding the variable types, so there would be less errors in general.

OCaml's memory mamangement is unique in that is uses mark-and-sweep, but it keeps short-lived data and long-living data in two separate heaps that run the garbage collector at different frequencies. By doing so, OCaml's garbage collection is somewhat more efficient that it would be in Python (with no reference counting), but it doesn't match up to Java due to no compaction.

In addition, like Python, OCaml has a GIL, meaning that parallelism on multiple CPUs/cores is not supported.

Another important thing to note is the language style. Unlike Python and Java, which are both imperative languages, OCaml is mostly functional. The difference in programming style can make the

For TensorFlow OCaml support, it is even less supported than the support for Java. The bindings are still early in development (as of Dec 2018), and some operators are not supported [7].

2.3 Kotlin

Now we can look at the programming language our project is written in: Kotlin. Kotlin is a language developed by Jetbrains that is explicitly meant to be a strictly better descendent of Java. Kotlin is able to be compiled into JVM and Javascript bytecode, so it is more scalable than Java, Python, and Ocaml.

Because Kotlin can be converted into JVM bytecode and run on the JVM, we can say that it basically has the same performance as its Java counterpart. For static typing, garbage collection, and multithreading, it is all supported just like it is in Java. In addition, Kotlin also has access to all of Java's and C's libraries, including TensorFlow.

2. Conclusion

If we are to compare all four of the languages together, Kotlin would be the best choice, with Java being a close second.

References

[1] "Garbage Collection." *OCaml*. https://ocaml.org/learn/tutorials/garbage_collection.html

[2] Gill, Navdeep S."Overview of Kotlin and Comparison Between Kotlin and Java." 17 May 2017. https://www.xenonstack.com/blog/overview-kotlin-comparison-kotlin-java/

[3] "Install TensorFlow for Java." *TensorFlow*. https://www.tensorflow.org/install/lang_java

[4] Kunze, Julius. "TensorFlow in Kotlin/Native." 24 Jan, 2018. https://juliuskunze.com/tensorflow-in-kotlin-native.html

[5] "Multicore OCaml." *OCaml Labs*. http://ocamllabs.io/doc/multicore.html

[6] "TensorFlow Architecture." *TensorFlow*. https://www.tensorflow.org/guide/extend/architecture

[7] "TensorFlow bindings for OCaml." Opam. 12 Dec, 2018. https://opam.ocaml.org/packages/tensorflow/

[8] "TensorFlow in Other Languages."*TensorFlow*. https:// www.tensorflow.org/guide/extend/bindings

[9] "TensorFlow Version Compatability." *TensorFlow*. https://www.tensorflow.org/guide/version_compat