Language Bindings for TensorFlow

Stefanie Shidoosh – University of California, Los Angeles

Abstract

This executive summary is a follow up on the assigned project, which implements application server proxy herd. We are to consider the downfalls of running an application server proxy herd on a large set of virtual machines. This application is said to use machine learning algorithms and depends on TensorFlow. TensorFlow is an "open source software library for high performance numerical computation" as a machine learning framework [1]. The prototype is said to use Python, which turns out to be a bottleneck when handling small queries that create and/or execute models. We are to consider other languages that would speed up performance, while maintaining the ability to prototype with Python and continue to use TensorFlow. This executive summary compare the suitability of three languages: Java, OCAML, and Crystal for a possible alternative approach to implementing this application.

1. Introduction

The application server herd, while very well equipped to handle a frequent, large amount of connections from mobile clients, can receive small queries with respect to each connection. Because the application in question heavily depends on TensorFlow and machine learning algorithms, poor handling of these small queries can create a costly bottleneck, hindering performance greatly in its attempt to create and/or executing models. This is further aggravated by the Python prototype, as in the said situation, Python is not as efficiency driven as other languages, such as the likes of those which we will further in this summary.

TensorFlow, in regards to this application, has two factors: the front-end Python which creates user-specified models, and the back-end C API that runs the corresponding computations. The creation of these models ends up being the culprit such that when benchmarks are run, the application spends a significant amount of time running the front-end Python code. This is greatly disappointing considering that the Python code is merely the set-up, the overhead that is needed for the actually substantial, computational work that the C API can execute. Therefore, we aim to consider another language for our application design to mitigate this bottleneck, considering factors such as compatibility, bindings, and performance while assessing the advantages and disadvantages of each language with respect to our desired functionality. We will first consider Python with its current prototype, and then investigate the alternatives.

2. Python

Python's was made with the intention of being a simple, minimalistic scripting language [2]. As a result of being a derivative of ABC, it utilizes white space indentations to signify code blocks, which makes for clean, visually pleasing code while maintaining a strong set of productivity features. These features include its typing capabilities, automated memory management, and its ability to support various programming styles, including (and probably most widely used among Python programmers) the imperative, object-oriented based style [3].

2.1 Advantages

As we have seen, Python is equipped with the modules asyncio and aiohttp that emphasize Python's ability to prototype for an application server herd. These modules together provide a functionality such that asynchronous connections can be made via TCP and HTTP (HTTP thanks to aiohttp). This is especially advantageous for the Python prototype, as handling of the server herd is quick and easy relative to other approaches, as demonstrated in the previous project. Equipped with this interface, combined with Python's dynamic typing, development time and effort is greatly facilitated. This interface is also well documented, with a solidified developer community and thus there is more aid available, than say with a less popular language.

Also, the maintenance of such an application will not have the sort of concerns say that a statically typed language would have to consider. The type of the models may vary depending on the application, and thus Python is better equipped to deal with this discrepancy, and thus less stress for programmers who do not want to worry over such details.

Moreover, the TensorFlow object type can be handling via duck typing, such that when Python is executed to generate the model, the object will be treated accordingly. If it looks like the type, and it acts like the type, then the object will be treated as that type. It should be noted here that the object in memory will be handled automatically, again alleviating the developers of the overhead that comes along with allocating memory on the heap manually.

2.2 Disadvantages

However, there is a significant tradeoff as a result of such typing. Python is an interpreted language with dynamic typing, and as such at run time must figure out the type of an object at runtime. This eats CPU time, and can be avoided is another design implementation was chosen, such that the language was statically time. In such a case, at compile time, the program expects a certain type and can act accordingly. This would require a bit more effort from the developer, but considering the functionality we are attempting to maintain, would be worth is as it would increase efficiency.

Also, there is a tradeoff of the automated memory management of Python, such that reference count based garbage collection must occur through the execution of the program to ensure that objects that are no longer in use can be freed. This also increases our execution time, which if we could explicitly free an object when we no longer need it, could have optimized this execution.

3. Java

Java was created with safety and reliability as its main priorities, but is also equipped with lucrative features for optimizing code, especially considering its optimal interface for concurrent programming. It is rooted for an object-oriented programming style, and thus may prove to be the next best thing relative to our Python prototype, as it handles the efficiency downfalls while maintaining the original prototype's approach, albeit with different supplementary support for compatibility reasons, which we will see is more or less negligible.

3.1 Advantages

As seen in our application server proxy herd project, Java has a convenient library, Node.js, that is more or less indistinguishable from asyncio in terms of functionality. Thus, the conversion to support a TensorFlow client from the Python prototype to a Java prototype would not be that demanding. The approach would be very similar, as asyncio and its implementation is analogous to the features that Node.js provides.

Moreover, Java could continue the object-oriented style equipped with libraries such as Node.js, and further introduce concurrency. Thus, Java benefits from the great developer community that Python also enjoys in terms of asynchronous connections, and as the cherry on top, also enjoys the availability of well documented libraries that support multithreading.

It should be noted that Python cannot be multithreaded. Considering this, because Java is compiled into bytecode, optimization can occur at this stage as well, thus further optimizing throughput for our server herd.

3.2 Disadvantages

As alluded to in discussing Python's advantages, static typing takes away from development time. Java, as it emphasizes reliability, is strongly statically typed and thus developers must consider the possible nuances of TensorFlow objects when creating the corresponding model [4].

4. OCAML

OCAML is a derivative of ML, supporting its origins functional style programming. However, it also supports flavors of imperative, object-oriented style of programming. Luckily for us, OCAML also has libraries that support an asynchronous server herd analogous to Python's asyncio and Java's Node.js.

4.1 Advantages

In terms of typing, OCAML is the middle ground between Python and Java. Its defining feature is type inference, similar to Python's dynamic typing. However, there is a significant difference, such that OCAML statically type checks at compile time. This handles the overhead that duck typing causes the Python prototype to bottleneck, such that once its compiled, the program does not have to decide a TensorFlow's object type. Since the type is known, this CPU time can be allocated to executing the computational work while the model is generated faster.

This is even more advantageous when considering the compilation is converted directly into machine code, like C/C++, say, instead of the middle step into bytecode that Java takes.

Furthermore, OCAML's functional programming style is preferred among machine learning developers (see: LISP, Scheme). Consider this in combination with OCAML's LWT library, which is analogous to asyncio and Node.js such that it provides supports for asynchronous server herd applications. Thus, for our implementation of our server heard in regards to generating models based on machine learning, OCAML binds these functionalities and provides a good platform for considering the said machine learning algorithms and support for the TensorFlow client-server herd [5].

4.2 Disadvantages

OCAML's type-checking system, though advantageous for reliability, takes a toll on development time. Speaking from personal experience (see: HW1, HW2), error-messages translated from French are frustrating, leaving developers with the feeling that they are debugging a debugger. In addition to this, debugging an asynchronous application is difficult enough, and OCAML's unfriendly interface for debugging further adds to this unnecessary strife. Thus, though we may up our performance, we run the risk of ridding ourselves of reliability in terms of functionality. This is extremely important if a TensorFlow model is to do just that, model, we ought to be sure that the model is what is desired before we continue with the back-end computation.

Furthermore, OCAML's compilation into machine code directly ultimately hinders portability, as the compiled program will be machine specific. Compare this to Java's bytecode that is mitigated by the JVM to avoid this very problem. Continuing with the comparison to Java, OCAML cannot support concurrency.

5. Crystal

Crystal is a statically-typed, imperative language with Ruby-like syntax. It is statically-typed specifying the type of variables or method arguments is not required, though it is not parallelizable. And, probably most advantageous for our purposes, is the language is able to call C code by writing bindings to it in Crystal [6].

5.1 Advantages

As an imperative language, Crystal comes with the advantages previously discussed with Python and Java in regards to object oriented programming. However, the driving point is that it has the capability of calling C code. This is especially important for our purposes as the back-end API responsible for computation is in C, and thus it is possible Crystal's compatibility with C can allow for routes of optimization that none of the previous languages discussed thus far can offer.

In addition, Crystal is sleek, inspired by Ruby syntax, and thus has the readability that Python offers. Crystal claims to be as efficient as C, with similar features like static type checking, which can optimize our TensorFlow model generation for the same reasons as previously discussed.

And, in terms of the asynchronous server herd, Crystal's runtime scheduler has a queue, and in it is an event loop that checks if "there is any async operation that is ready, and then executes the fiber waiting for that operation," whereas fiber is an execution unit [6]. This functionality is currently supported by libevent, which is "meant to replace the event loop found in event driven network servers," thus the language us supported by functionalities required for an asynchronous server herd [7].

5.2 Disadvantages

That being said, there is no one dedicated library for asynchronous functionalities like Python's asyncio, Java's Node.js, or OCAML's LWT. This would require the developer to create their own implementation, which is very demanding considering that there exist libraries that already do it, so it is hard to imagine why one would decide to do it themselves when they could just choose a language that supports the desired functionality.

This brings up a dire point – Crystal is in its infancy stages, and though it looks promising, it does not have the developer community that Python and Java do. It will take time and significant momentum for Crystal to come up to speed in this regard, so for immediate purposes, the language fails in terms of available documentation.

6. Conclusion

Considering all of the points made, I argue for the Java prototype. Its documentation, object-oriented style, and efficiency for type checking greatly improves performance, and it would not be too much a stretch for this front-end prototype to communicate with the C back-end as their both imperative. Though OCAML may perform better as it is functional language, the unfriendly error checking is not beneficial for development and does not ensure the same degree of reliability that Java does. And, Java can support multithreading, whereas none of the other languages support this feature. Notably, if Crystal did have a library for asynchronous I/O, I would choose it over the other for the sheer direct compatibility with C for consistency and more available optimization.

References

[1] https://www.tensorflow.org
[2]https://docs.python.org/2.0/ref/node92.html
[3]en.wikipedia.org/wiki/Python_(programming_lang
uage).
[4]https://docs.oracle.com/javase/specs/jls/se7/html/jl
s-4.html
[5] towardsdatascience.com/functional-
programming-for-deep-learning-bc7b80e347e9.
[6] https://crystal-lang.org/docs/
[7] http://libevent.org