

Your summary should focus on the technologies' effects on ease of use, flexibility, generality, performance, reliability; this idea is to explore the most-important technical challenges in doing the proposed rewrite.

ease of use
flexibility
generality
performance
reliability

focus on most important technical challenges of rewrite

F#: compiled

- functional language (also has imperative and object styles)
 - supports tail recursion
- part of CLI (Common Language Infrastructure)
 - support for garbage collection
 - support for exception handling
 - Common Language Runtime (VM for CLI)
- strongly typed
- statically typed (mostly)
- meant to work with .NET languages and libraries
 - .NET runs on virtual machine (CLI shit)
 - .NET framework designed to support interoperability
 - F# works well with others (if microsoft is telling the truth)
 - .NET primarily designed for windows
 - TensorFlowSharp, .NET bindings to TensorFlow library
 - Not completely finished yet ("pretty much done")
 - Incomplete API documentation (unstyled, harder to read)
- intended to reduce development time
 - data rich analytical services
- first class language (meaning?)
- member of ML language family
 - type inference
- native support for asynchronous programming
 - let! indicates callback
 - allows parallel execution of asynchronous blocks natively
 - Async.Parallel
 - also can turn asynchronous into blocking (Asnyc.RunSynchronously)
- metaprogramming:
 - >can compile F# to javascript and GPU code
- supports exception handling

Java: (byte code, run on JVM with JIT) (could compile Python into Java Byte code)
Native support by TensorFlow

"TensorFlow Java API is not currently covered by the TensorFlow API stability guarantees"

Support for asynchronous event loop lacking

- have to build it yourself
- JVM bytecode
 - JIT support, faster than Python
- garbage collected (generational?)

Throws exceptions for error handling

OCaml: (compiled)

"The tensorflow-ocaml project provides some OCaml bindings for TensorFlow."

- doesn't seem to support all the functionality for TensorFlow in OCaml
- also requires additional dependencies due to TensorFlow being built on C++
- Static types
 - constrains programming to careful planning
 - improves performance (would be faster than)
- type inference
- tail call recursion support (unclear if required by standard)
 - do have to restructure code with that in mind
- supports bytecode compiling
- native support for asynchronous programming
 - event loops using Async.Std
- multithreading is not multicore
 - downside if attempting to grow program
- supports exception handling
- garbage collection
 - possibly generational

references:

The OCaml system release 4.06 Documentation and user's manual <http://caml.inria.fr/pub/docs/manual-ocaml/>
<https://realworldocaml.org/v1/en/html/index.html> (possibly cite the book instead)
<https://github.com/LaurentMazare/tensorflow-ocaml/>

<https://docs.oracle.com/javase/8/docs/>

<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/java>

<https://github.com/migueldeicaza/TensorFlowSharp>

<https://fsharp.org/specs/language-spec/4.1/FSharpSpec-4.1-latest.pdf> //fsharp 4.1 language spec

.NET Framework.â€ Wikipedia, Wikimedia Foundation, 8 June 2018, en.wikipedia.org/wiki/.NET_Framework.

https://en.wikipedia.org/wiki/.NET_Framework

webber book (get reference for it)

benchmarks : <https://benchmarksgame-team.pages.debian.net/>

Introduction: Problems with Python:

Python is designed to be user friendly to learn, read, and write. It provides garbage collection memory management and a plethora of library support for every possibly need, increasing reliability and ease of use .The power of Python is its promotion of high level concern

over low level; the programmer can spend time thinking about their main problem rather than the minutiae of implementation. By default, Python programs are compiled/interpreted with many runtime checks that allow for errors and bugs to be caught quickly. This philosophy comes at a cost.

Interpretation and runtime checking add overhead to computation. While the cost of interpretation can be mitigated by compiling to machine code, Python's choice of dynamic typing requires runtime checks to be performed.

Although dynamic type checking makes programs more flexible, the cost of these runtime checks are not amortized by computation for this specific use case. Thus, Python's design choices become a bottleneck.

Alternative 1: Java

Java is a possible alternative for the Python implementation due to TensorFlow's native API support of Java (although it is not as stable as the Python API). Unlike Python, Java employs static strong typing. Not only does this indicate typing bugs at bytecode compile time, this also improves speed of the program due to the lack of runtime checks. Java is a partially compiled language that makes use of a virtual machine. Rather than employing conventional interpretation as Python does, Java compiles code to bytecode before running on a virtual machine. This means that Java can utilize the same runtime checks (eg. index in bounds) as Python but at a lesser cost. Parsing and running bytecode on a virtual machine is much less expensive than parsing text with an interpreter. To further improve performance, Java uses JIT (Just In Time) compiling to compile the bytecode of commonly used functions to machine code. As a result, Java performs far better in benchmarks than comparable Python code. Another benefit of bytecode is its portability to any machine that runs the JVM.

Java's memory management is based on garbage collection. What type of garbage collection is implementation dependent (possibly generational). Not only does this make Java an easier language for programmers to use, it also improves the reliability of programs. Dangling pointers and memory leaks are abstracted away by the use of garbage collection.

Additionally, Java utilizes exception handling, allowing programmers to properly recognize and handle errors/bugs without complete crash of programs.

A benefit of Java is its support of polymorphism. Functions need not be rewritten for each intended type through the use of generics. Although this does require the programmer to understand the meaning of generics, it saves programming time for rewriting of functions.

It is possible to create the equivalent event loop for asynchronous IO in Java 8 by using `java.util.concurrent` and `java.nio.channels`. However, programmers should approach with caution. The current tutorials and documentation are not as straight forward as Python's `asyncio`. While it is entirely possible to implement the abstractions provided by the `asyncio` library, the design process will be longer due to the necessity of programmers to understand lower level implementation details.

Additionally, the conventional Java approach is that of multithreading rather than single threaded event loops. So while it is possible to write the equivalent event loop in Java, convention dictates multithreading. While this can achieve better performance for computations than asynchronous event loops, it adds the complexity of race conditions. Again, lengthening the design process.

Alternative 2: OCaml

Despite its functional programming origins, OCaml supports imperative programming styles. This makes a conversion from more than Python feasible. Like Python, Java and F#, OCaml hides memory management from the user by use of garbage collection (what flavor is implementation dependent). Again, this makes conversion less complicated by avoiding memory leaks and dangling pointer problems.

OCaml utilizes static types and type inference at compile time. OCaml can be interpreted or converted to byte code, but its primary use is compilation to machine code. Provided compiler support for the CPU architecture in question, equivalent machine code will be at least as fast as bytecode on a virtual machine. While static types require more careful thought by the programmer, they allow for faster programs due to their lack of runtime checking. A major benefit of OCaml is its use of type inference. In simple cases, this means that the user need not specify `int` when using an integer type. In the more general case, users can define functions that take polymorphic types as argument. Provided valid use of types and function definitions, this generalizes functions to support many types without recompiling for each type used.

Although implementation will not be as simple as use of `asyncio`, OCaml has native support for asynchronous programming using the `Async` library. Thus, it is possible to write a comparable program in OCaml. However, it should be noted that OCaml prefers an object oriented, functional coding style. While imperative styles are supported,

they are not conventional in OCaml. This means that a programmer will need to be comfortable with recursion and utilization of tail calls in order to write efficient programs. Additionally, matching is a powerful convention used by OCaml not found in common Python or Java applications. Therefore, the logical conversion from Python to OCaml may not be as seamless as the conversion from Python to Java.

OCaml's support for multithreading is also a work in progress. Currently, multithreading on multiple CPU cores is not natively supported. While the current Python design does not employ multithreading, future scalability is a design consideration.

#####Maybe for conclusion?
the TensorFlow documentation
indicates the OCaml TensorFlow library only supports some of the bindings for the TensorFlow API.

Alternative 3: F#

As a descendent of ML and a relative of OCaml, F# has much in common with OCaml. Use of static typing and type inference provide the same benefits to F# as they do to OCaml. Garbage collection, native support for asynchronous programming, and tail call optimisations are also found in F#. F# also provides further support for object oriented and imperative programming.

The main distinction between F# and OCaml is F#'s .NET framework implementation. As a Common Language Infrastructure (CLI) language, F# is intended to be cross platform. This means in addition to compilation to machine language, F# can be run on the Common Language Runtime (CLR) virtual machine, improving portability between machines. Via the .NET framework and CLI, F# programs are maintainable and flexible.

Like other ML descendents, the functional nature of F# is intended to simplify coding and reduce development time. Due to its basis in the .NET framework, F# works well with other programs built on the .NET framework. Microsoft's intent in designing F# was for it to simplify both development and integration with other programs. Unfortunately, Microsoft's relation to F# dictates that it will not be as portable between operating systems as Java, Python, or OCaml implementations.

For future design considerations, F# natively supports parallel execution of asynchronous blocks. Unlike OCaml, F# can handle parallel execution on multiple cores, improving scalability of F#.

Which Language to use:

TensorFlow has native API support for both Python and Java while support for F# and OCaml

is done through binding. While this added code should add little overhead, it should be noted that the documentation and API for F# is not fully complete and documentation indicates that OCaml only has partial support.

In most cases, F# performs better in benchmarks than OCaml does. Additionally, F# has builtin integration with other programs via the .NET framework. Finally, F# has more has expanded support for object oriented and imperative programming conventions, making it a simpler language to learn coming from Python. Taking all of this into consideration, converting to F# is recommended over converting to OCaml.

Benchmarks show comparable performance for F# and Java. It should be considered that non-Windows operating systems may not have full .NET framework support, possibly limiting the usability of F# and the .NET binding of TensorFlow on certain machines. If this is the main concern, then Java should be used for conversion. Beyond that, the difference between F# and Java is a question of preference; both languages are capable of better performance for the desired use case than Python. If the programmer is more familiar with functional style, then F# is more desirable. If an imperative, object oriented style is favored (presumably similar to the current Python form of the program), then Java is the better choice.