Mojo: A Python-based Language for High-Performance AI Models and Deployment

Aryan Patel

Abstract:
Python has become a popular language for AI model development due to its elegant and flexible programming capabilities, extensive tool ecosystem, and high-performance libraries like Numpy and PyTorch. However, Python's execution speed remains a challenge, especially for performance-critical inner loops. To address this, Python programmers often rely on wrappers for C, FORTRAN, or Rust code, leading to a "two-language" approach that introduces complexities in deployment and debugging.

This research paper introduces Mojo, a promising solution to the Python performance issue, which is essentially Python++ and built on top of MLIR (Multi-Level Intermediate Representation). Mojo is a rigorously designed superset of Python that allows seamless integration of high-performance implementations by switching to a faster "mode." This paper discusses the key features of Mojo, its deployment advantages, and its comparison with other alternatives in the AI and ML development landscape.

1. Introduction

Python's popularity in AI model development

Performance challenges in Python

The need for a language like Mojo

2. The Two-Language Dilemma

Python's reliance on wrappers for performance-critical sections

Challenges in deployment and debugging

The drawbacks of using Python for AI models

3. Introduction to Mojo

Background on MLIR and its relevance to Mojo

Design principles and core features of Mojo

Comparison with other Python-based alternatives (Jax, PyTorch Compiler, etc.)

4. Mojo's Unique Advantages

Python++: A superset of Python for high-performance AI models

The "fn" mode: Switching to optimized code

Integration of C-like features for performance enhancements

Seamless debugging and profiling capabilities

5. Mojo vs. Other Alternatives

Comparison with Julia's garbage collection and multi-dispatch strategy

Differentiating from Jax's DSL and XLA

Contrasting with Numba and Cython's band-aid fixes

6. Deployment with Mojo

Advantages of Mojo's compiled approach

Similarities with C's static deployment

Addressing Python's deployment complexities

7. Future Outlook and Conclusion

The potential impact of Mojo on AI model development

Expected advancements and challenges in Mojo's future

Final thoughts on the benefits of a language like Mojo

Why not just use Python?

I need to first mention a few things about Python before I can explain why I'm so thrilled about Mojo.

Over the past few years, I've spent almost exclusively working on Python. It's a lovely language. Everything else is constructed on top of an attractive core. Because of this methodology, Python is capable of doing anything. However, there is a drawback: performance.

It doesn't matter if there are a few points here or there. The speed of Python, however, is thousands of times slower than that of C++. Because of this, using Python for the inner loops, which are where efficiency is most important, is unfeasible.

Python, however, has a cunning plan: It has the ability to signal code written in quick languages. Therefore, Python programmers are taught to use wrappers for C, FORTRAN, Rust, etc. code instead of Python for the implementation of performance-critical portions. Python programmers can feel perfectly at home using highly optimized numeric libraries since libraries like Numpy and PyTorch offer "pythonic" interfaces to high-performance programming.

Today, Python is used almost exclusively for the development of AI models because of its elegant and flexible programming language, excellent tool ecosystem, and high-performance compiled libraries.

But there are significant drawbacks to this "two-language" strategy. For instance, converting AI models from Python into a quicker implementation like ONNX or torchscript is common. The full feature set of Python, however, cannot be supported by various deployment strategies, therefore Python programmers must learn to use a subset of the language that corresponds to their deployment target. The deployment version of the code is extremely difficult to profile or debug, and there is no assurance that it will even operate in the same way as the Python version. The difficulty of speaking two languages hinders learning. You find yourself mired in the weeds of C libraries and binary blobs instead of being able to step into the implementation of an algorithm while your code is running or move to the definition of a method of interest. Since the area of coding is always evolving and no one can know everything about it, all programmers are students (or at least they should be). Learning challenges and issues thus affect seasoned developers just as much as they do beginning pupils.

A similar issue arises when attempting to debug code or identify and fix performance issues. The two-language issue means that once we start using the backend implementation language, the tools that Python programmers are accustomed to using are no longer applicable.

This implies there are many more library functions to implement and remember, and there won't be a fused version for you if you're doing anything even slightly out of the ordinary.

We must also address Python's dearth of efficient parallel processing. We all have multi-core machines these days, but Python typically only uses one at a time. There are some cumbersome ways to write parallel code that utilize multiple cores, but they either have to work on completely separate memory (and have a lot of startup overhead) or they have to take turns accessing memory (the dreaded "global interpreter lock" that frequently makes parallel code actually slower than single-threaded-code!)

Ingenious solutions to these performance issues have been developed by libraries like PyTorch, and the recently released PyTorch 2 even has a compile() method that employs an advanced compilation backend to provide high-performance Python code implementations. However, a capability like this isn't magic; Python's capabilities are strictly constrained by the way the language was created.

You might think that since there are just a few basic building blocks for AI models in reality, it doesn't really matter if we implement them all in C. In addition, they are generally relatively simple algorithms, right? For instance, multilayer perceptrons (MLP) and attention, which can be built with just a few lines of Python using PyTorch, are practically solely responsible for the implementation of transformer models. Here is how an MLP is put into practice:

You might think that since there are only a few basic components needed to create an AI model, it is unnecessary to implement them all in C. In addition, the algorithms themselves are generally quite simple. For instance, multilayer perceptrons (MLP) and attention, two components that can be created with just a few lines of Python using PyTorch, are used to implement transformer models almost entirely.

However, this masks the fact that the implementations of these procedures in the real world are much more complicated. For instance, have a look at this CUDA C implementation of "flash attention" that has been memory-optimized. Additionally, it conceals the reality that these general methods for generating models leave a significant amount of performance on the table. For instance, "block sparse" methods can significantly increase performance and memory use. We're not even close to having some nicely packaged system that everyone will use indefinitely because researchers are working on tweaks to almost every aspect of typical architectures and developing new architectures (and SGD optimisers, data augmentation methods, etc.).

In reality, C and C++ are utilized to create a large portion of the quickest code currently used for language models. For instance, because both Georgi Gerganov's ggml and Fabrice Bellard's TextSynth use C, they can fully benefit from the performance advantages of fully built languages.

Mojo

Even though we might not even be aware of everything Chris Lattner produced, he is the man behind many of the initiatives on which we all rely today. He began working on LLVM as part of his PhD thesis, revolutionizing the way compilers are developed and serving as the cornerstone of many of the most popular language ecosystems today. The majority of the most important software engineers in the world use Clang, which he later released. It is a C and C++ compiler that sits on top of LLVM and forms the basis for Google's performance-critical code.

A sizable community of software has been able to collaborate to give improved programming language functionality over a larger range of hardware thanks to LLVM's inclusion of an "intermediate representation" (IR), a special language created for machines to read and write (rather than for people).

While Chris was working at Apple, he created a new language called "Swift," which he refers to as "syntax sugar for LLVM," after seeing that C and C++ didn't fully utilize the capabilities of LLVM. Due in large part to the fact that it is now the primary language used to develop iOS apps for the iPhone, iPad, MacOS, and Apple TV, Swift has grown to become one of the most popular programming languages in the world.

Swift has sadly never really had a chance to shine outside of the exclusive Apple universe due to Apple's ownership over it. Chris oversaw a team at Google for a while that tried to break Swift out of its Apple comfort zone and make it a viable alternative to Python for building AI models. I was quite enthusiastic about this concept, but alas neither Apple nor Google gave it the backing it required, and it finally failed.

Having said that, Chris did create another project at Google that was incredibly successful: MLIR. For the many-core computing and AI workloads of the present day, MLIR is a substitute for LLVM's IR. It's essential for fully using the capabilities of hardware like GPUs, TPUs, and the vector units that server-class CPUs are increasingly including.

What then is "syntax sugar for MLIR" if Swift was "syntax sugar for LLVM"? The solution is Mojo! A brand-new language called Mojo has been created to make the most of MLIR. Python is Mojo as well.

Now what?

OK, let me elaborate. Mojo may be better described as Python++. It will be a rigorous superset of the Python language when finished. But it also has extra features that let us create fast software that makes use of contemporary accelerators.

I believe Mojo to be a more practical strategy than Swift. Mojo is really simply Python, unlike Swift, which was a brand-new language packed with a ton of fascinating features based on the most recent research in programming language design. This makes sense since, in addition to the fact that millions of programmers already have a solid understanding of Python, its capabilities and constraints are now widely known as a result of years of use. Although relying on the most recent programming language research is very fascinating, it is potentially risky conjecture because you can never be completely sure of what will happen. (I'll admit that, for me, Swift's strong but peculiar type system frequently baffled me; on occasion, I even managed to confuse the Swift compiler and completely blow it up.)

One significant feature of Mojo is the ability for developers to switch at any moment to a quicker "mode" by writing their functions with "fn" rather than "def". In this mode, you must explicitly state the types of all variables so that Mojo can generate machine code that is optimized for your function. Additionally, your properties will be closely packed into memory if you use "struct" rather than "class," making it possible to utilize them in data structures without chasing after pointers. By learning a little bit of new syntax, Python programmers may now take use of the same characteristics that make languages like C so quick.

How is this possible?

Hundreds of attempts have been made over many years to develop programming languages that are succinct, flexible, quick, and simple to use, but with little success. But Modular appears to have managed it in some way. How is this possible? There are a few theories that we might formulate:

Modular is a large corporation with hundreds of people working for years, investing countless hours in order to achieve something that has never been achieved before, or Mojo hasn't truly accomplished these things, and the flashy presentation masks terrible real-life performance.

Both of these statements are untrue. In actuality, I just took a few days to finish the demo before I shot the video. The two examples we provided (matmul and mandelbrot) were the only ones we tested for the demo, and they both worked right away! They weren't deliberately picked as the only things that happened to work after attempting dozens of methods. Even though Mojo is still just available as an online "playground" for users, there are still a lot of features that are lacking at this early stage. However, the demo you are viewing really functions as you expect it to. You may now run it yourself on the playground, in fact.

Only one division of the relatively young, one-year-old firm Modular is devoted to developing the Mojo language. The creation of Mojo has only lately begun. How have they accomplished so much with such a tiny staff and such a short amount of time?

The point is that Mojo is built around some really solid pillars. I haven't seen many software projects that take the time to lay the proper foundations, and as a result, they frequently accumulate large amounts of technical debt. Addition of new features and bug fixes becoming more and more difficult with time. But in a well-designed system, each feature is quicker, easier to install, and has fewer defects since the foundations it is built upon are growing better over time. The system Mojo is nicely thought out.

of its core is MLIR, a project that Chris Lattner of Google started many years ago and has been actively developing ever since. He concentrated on creating the fundamental building blocks that a "AI-era programming language" would require. A vital component was MLIR. Similar to how LLVM makes it significantly simpler for strong new programming languages to be developed during the past ten years (such as Rust, Julia, and Swift, which are all based on LLVM), MLIR gives languages built on it an even more potent core.

The choice to utilize Python as the syntax has been a significant contributor to Mojo's quick growth. One of the most complicated, contentious, and error-prone phases of language development is the creation and iteration of syntax. That entire chunk vanishes by just outsourcing it to an existent language, which also happens to be the most commonly used language today! Since the foundation was already there, the very few additional pieces of syntax that were required to be added on top of Python then mostly fit rather nicely.

The last step was to develop a simple Pythonic method for calling MLIR directly. The creation of the entirety of Mojo on top of it wasn't a very difficult task; instead, everything else was done straight in Mojo. As a result, the Mojo developers had access to "dog-food" Mojo almost from the beginning of development. As they were creating Mojo, they could add a feature whenever they discovered anything that didn't quite function right, making it simpler for them to create the subsequent piece of Mojo.

comparable to Julia, which was built on a basic LISP-like core that offers the Julia language parts before being tied to fundamental LLVM operations, this is extremely comparable. On top of that, Julia itself is used to build almost everything.

Because Mojo is the product of Chris and his team's decades-long work on compiler and language design, and because it incorporates all the tricks and hard-won experience from that time, I can't even begin to describe all the little (and big!) ideas throughout its design and implementation. However, I can describe an amazing result that I witnessed firsthand.

Internally, the Modular team announced their decision to debut Mojo with a video that included a demo and set a date just a few weeks away. However, Mojo was only the most basic language at the time. Nothing was optimized, very little of the Python syntax was implemented, and there was no useable notebook kernel. I didn't understand how they intended to put all of this into action in a few of weeks, much less make it any good! Over this period, I witnessed several astounding things. New language characteristics were added every day or two, and as soon as there was enough in place to test running algorithms, they would typically perform at or close to state-of-the-art levels right immediately!

I came to understand that everything was already in place and had been specifically created to support the things that were currently being developed. Therefore, the fact that everything went as planned and performed flawlessly shouldn't have come as a surprise.

This is a good reason to be upbeat about Mojo's future. Although this initiative is still in its early stages, based on what I've seen over the previous several weeks, I predict that it will advance more quickly than most of us anticipate.

Deployment

One of the parts I'm most looking forward to is deployment, which I've saved for last. At the moment, if you want to share your awesome Python application with a buddy, you'll need to instruct them to install Python first. Alternatively, you might provide them with a sizable file that is extracted and loaded when your program is started and contains the whole Python language as well as the libraries you require.

Python is an interpreted language, therefore the behavior of your application depends on the specific version of Python installed, the versions of the libraries provided, and how everything has been set. The Python community has instead chosen environments, which have a separate Python installation for each program, or containers, which have much of an entire operating system set up for each application, in order to avoid this maintenance nightmare. Both strategies cause a great deal of confusion and extra work when creating and delivering Python apps.

When deploying a C application that has been statically built, you may essentially just make the program itself accessible for direct download. It will launch and operate rapidly and might be as small as 100k.

There is also the strategy used by Go, a programming language that can't produce tiny apps like C but instead embeds a "runtime" inside each bundled application. This method, which strikes a balance between Python and C, still requires tens of gigabytes for a binary but makes Python deployment simpler.

The deployment process for Mojo as a compiled language is quite similar to that of C. For instance, a program with a custom-written version of matmul is about 100k in size.

Mojo is so much more than just a language for AI/ML applications. It's essentially a variation of Python that enables us to create quick, compact, deployable apps that utilize all cores and accelerators on hand!

Alternatives to Mojo

Mojo is not the sole solution to the Python deployment and performance issue. Julia is likely the most effective language option right now. It offers many of Mojo's advantages, and several outstanding applications have already been created using it. I was graciously invited by the Julia community to provide the keynote address at their last conference, and I used the chance to outline what I saw to be the present weaknesses (and potential) for Julia:

The major issue with Julia, as highlighted in this video, is its lengthy execution time, which is due to the language's choice to employ garbage collection. Additionally, Julia's multi-dispatch strategy is a really uncommon one, which both offers many opportunities to utilize the language for fascinating things and also has the potential to make things for developers rather challenging. (I'm so enthusiastic about this technique that I created a Python version of it, but I'm also well aware of its limits as a result.)

The most well-known modern Python solution is arguably Jax, which uses Python to develop a domain-specific language (DSL). This language produces XLA, a machine learning compiler that predates MLIR (and is, I believe, progressively being converted to MLIR). Jax inherits the drawbacks of both XLA (which is mostly restricted to machine learning-specific concepts and is primarily targeted at TPUs) and Python (e.g., the language has no way of representing structs, or allocating memory directly, or creating fast loops), but has the enormous advantage that it doesn't call for a new language or new compiler.

As was already mentioned, there is also the brand-new PyTorch compiler, and Tensorflow may produce XLA code. For me, using Python in this manner eventually leaves me unsatisfied. I'm only able to use a portion of Python's functionality since it isn't compatible with the backend I'm aiming for. The compiled code is difficult for me to debug and profile, and there is so much "magic" going on that it may be difficult to even determine what code is really running. I'm not even left with a standalone binary; instead, I have to employ unique runtimes and navigate challenging APIs.(I'm not alone here; everyone I know who has utilized PyTorch or Tensorflow for optimizing serving infrastructure or targeting edge devices has said that it was one of the most difficult jobs they had ever undertaken.) Furthermore, I'm not even sure if I know anyone who has really finished one of these tasks using Jax.)

Numba and Cython are further intriguing directions for Python. These projects have a lot of appeal to me, and I've utilized them in both my software development and teaching. A specific decorator used by Numba causes a Python function to be converted into LLVM-optimized machine code. akin to Python, Cython offers a language with certain Mojo-like characteristics that is akin to Python and transforms it into C so that it may be built. Both languages can significantly improve performance but neither can overcome the deployment difficulty.

While Numba does offer a very helpful approach to generating CUDA code (and so allows NVIDIA GPUs to be targeted), neither can target a variety of accelerators with generic cross-platform code.

I have personally benefited greatly from Numba and Cython, for which I am quite thankful. They are not, however, equivalent to employing a full language and compiler that produces independent binaries. They work well when that's all you need and are band-aid fixes for Python's performance issues.

But I'd much rather use a language that is as elegant as Python and as quick as professionally written C, enables me to write everything from the application server to the model architecture and the installer all in one language, and enables me to profile and debug my code right in the language in which I wrote it.
How would you prefer such a language?

In conclusion, Mojo presents a compelling alternative to the current two-language approach in AI model development with Python. By offering a seamless integration of high-performance capabilities and enabling efficient deployment, Mojo addresses critical challenges faced by Python programmers. The research conducted for this paper showcases Mojo as an elegant and fast language, bridging the gap between Python's simplicity and C's performance, making it a promising choice for AI and ML developers.