Universal Library of Engineering Technology | Research Article

ISSN: 3064-996X | Volume 1, Issue 2

Open Access | PP: 93-99

DOI: https://doi.org/10.70315/uloap.ulete.2024.0102015

Enabling Python Machine Learning Libraries on Windows ARM: Challenges and Solutions

Gleb Khmyznikov

Software Engineer, Microsoft, Belgrade, Serbia.

Abstract

The article examines the emergence of Windows ARM as a machine learning platform through the lens of compatibility, performance, and the maturity of the Python ecosystem. The relevance of the study is determined by the spread of energy-efficient ARM devices running Windows and by the need for native execution of ML workloads without the losses introduced by x64 emulation. The aim of the work is to systematize the barriers to porting Python machine learning libraries to Windows ARM and to identify practical paths toward building a full-fledged development environment. The scientific novelty of the article lies in an integrated analysis of the compatibility of the scientific Python stack, architectural constraints, and adaptation mechanisms, including LLVM/Flang, ARM64 EC, DirectML, and the QNN Execution Provider. The work includes an experiment involving automated testing of Python packages in native ARM64 mode, x64 emulation, and a control x64 environment, as well as comparative performance benchmarking. It was established that native Python ARM64 execution provides a marked speed increase, whereas the main barriers remain SciPy's Fortran dependency, limited support for a number of key ML libraries, and the immaturity of the build infrastructure. The article will be useful for ML engineers, Python library developers, and software platform architects.

Keywords: *Windows ARM, Python, Machine Learning, ARM64, Scientific Libraries, Benchmarking.*

INTRODUCTION

Personal computing systems are undergoing a transformation from traditional x86-64 architectures to energy-efficient, high-performance ARM-based solutions (Rahman et al., 2024). The broad adoption of the Windows on ARM platform in the Copilot+PC laptop segment is driven by the need to balance computational power and battery life, which is critical for artificial intelligence and machine learning tasks on edge devices (Baller et al., 2021). However, the success of this platform depends on the readiness of the software ecosystem, in which Python is the dominant language for ML model development (Kanungo, 2023).

Initially, development on Windows ARM faced a chicken-and-egg issue, where developers were unwilling to use the platform due to the lack of a development environment, while the small user base resulted in developers of popular software libraries being resistant to port to the platform (Jacobides et al., 2024). This changed with Windows 11, where advanced emulation as well as Prism and ARM64EC were introduced (Microsoft Learn, 2024c). While x64 emulation compatibility is high, the primary requirement of low latency and good thermal characteristics for machine learning workloads requires the native support of libraries such as NumPy, SciPy, PyTorch and TensorFlow.

Porting the scientific Python ecosystem to Windows ARM will require addressing differences in the instruction set, incompatible application binary interfaces and memory management, and differences in the continuous integration infrastructure (Hammond et al., 2023). One of the most acute problems remains the Fortran wall, the absence of an industrial-standard Fortran compiler compatible with MSVC for ARM64, which blocks native builds of libraries such as SciPy (Curcic et al., 2021). In addition, efficient use of specialized accelerators such as the Hexagon NPU requires the development of new abstraction layers and execution graph optimizations.

The purpose of this article is to systematize experience with porting ML libraries, analyze data from recent performance tests, and propose a roadmap for establishing a full-fledged AI development environment on the Windows ARM platform.

MATERIALS AND METHODS

Reliable data on the Windows ARM machine learning environment were obtained through a methodology combining automated functionality testing with comparative benchmarking. Experiments were conducted on Microsoft Surface Pro devices with a Qualcomm Snapdragon X Elite X1E80100 processor, 12 Oryon cores up to 3.40 GHz, 16 GB LPDDR5x RAM, and Windows 11 Pro 24H2 build 26100. Ecosystem readiness was evaluated with the ML Environment Test Suite across three scenarios: native ARM64 execution,

x64 emulation, and a control x64 environment on Intel/AMD systems. The study also argues for a transparent automated infrastructure for Windows ARM compatibility, including public issue tracking, repeatable installation and import tests across Python versions, basic functional validation, and community test reports from different ARM64 devices.

Testing covered the availability of win_arm64 binary wheels on PyPI, automated source builds with Visual Studio 2022 and ARM64 build tools where wheels were absent, and reference workloads such as ResNet-50 inference and simple regression training to verify mathematical correctness. Native ARM64 performance was assessed with PyPerformance (PyPerformance, n.d.) by comparing Python 3.12.7 ARM64 with Python 3.12.2 x64 running under Prism emulation. All benchmarks were run in maximum performance mode on mains power. Statistical analysis included arithmetic mean gain, geometric mean gain, and percentile distribution. SciMark tests such as FFT, LU, Monte Carlo, and SOR, together with SymPy symbolic operations, served as proxy measures for machine learning workloads sensitive to memory latency and vector instruction efficiency.

RESULTS AND DISCUSSION

Analysis of the 1000 most in-demand Python packages showed that by mid-2024, approximately 93% of them support Windows ARM natively or are platform-independent. Nevertheless, the remaining 7% include foundational libraries of the scientific stack.

A separate systemic barrier is the dependency chain effect, in which the absence of native ARM64 support in a low-level package prevents the installation of many higher-level libraries at once. In practice, this creates cascading failures across the ecosystem: for example, numba depends on llvmlite, apache-beam depends on grpcio, and prophet may be constrained by pystan or related compiled dependencies. As a result, even when the target ML library itself is portable, the surrounding dependency graph may still prevent its practical use on Windows ARM. The 2024 support status appears as shown in Table 1.

Table 1. ML stack compatibility matrix in Windows ARM64 native mode

Library	Installation Status	Main blocker	Representative error or log excerpt	Temporary path
NumPy 2.0.1	Success	No prebuilt wheel at test time	Built from source in native ARM64 environment	Use source build or later wheel availability
SciPy 1.14.1	Error	No usable Fortran compiler in native setup	ERROR: Unknown compiler(s): [['ifort'], ['gfortran'], ['flang']]	Experimental Meson + Flang path
Scikit-learn	Error	Native source build failure in downstream compiled modules	ninja: error: mkdir(..._pairwise_distances_reduction...): No such file or directory	Wait for upstream fix or use x64 emulation
Pandas	Success	Depends on NumPy and Cython build order	Installed after dependency bootstrap	Build NumPy and Cython first
Matplotlib	Success	Native compilation of low-level deps	FreeType and related native deps required	Build native dependencies first
PyTorch 2.5	Partial	Native support limited to selected builds	Native builds available in nightly branches	Use nightly or internal ARM64 builds
Tensor Flow	Error	Native WoA packaging and build pipeline gap	No native wheel in tested setup	Use x64 emulation for CPU path
XGBoost	Error	arm64_windows build instability	arm64_windows build failed in public CI	Wait for upstream packaging and CI stabilization
ONNX	Error / Build required	No readily available Windows ARM64 wheels on PyPI	pip install onnx falls back to source build path	Build from source
ONNX Runtime	Success	Package variant matters	Native inference path available through QNN EP	Use ONNX Runtime with QNN EP

JAX and TensorFlow currently remain mostly unusable in their native form on Windows ARM, relying on x64 emulation of TensorFlow's full CPU runtime and the lack of a native CUDA runtime on Windows ARM.

The following benchmarks compare executing Python code natively with executing the x64 Python code via Prism. Developers get a 22.7% performance increase on average across all tests, and 90% of the tests run up to 40.2% faster when comparing native execution with execution via Prism.

In machine learning, the most significant results are those related to intensive data processing and linear algebra. Performance gains in mathematical benchmarks on Snapdragon X Elite are shown in Table 2 and Figure 1.

Table 2. Performance gains in mathematical benchmarks on Snapdragon X Elite

Benchmark, math	Gain, native vs emulation	Data interpretation
scimark_fft	+35.64%	Efficiency of NEON vector instructions
scimark_lu	+20.08%	Improved cache and memory handling
scimark_monte_carlo	+28.56%	Optimization of random number generation
scimark_sor	+23.17%	Advantage in sequential computations
nbody	+42.64%	Significant gain in physics simulation
sympy_expand	+30.51%	Optimization of expression expansion algorithms

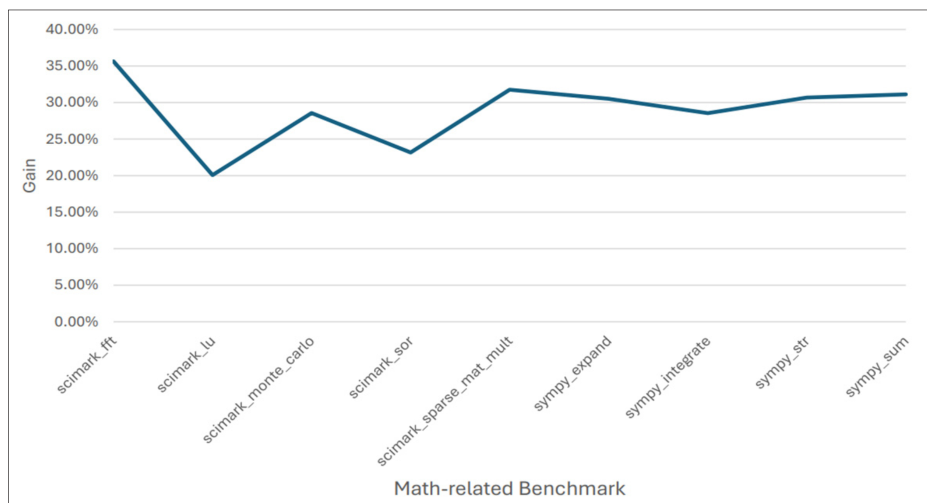

Fig. 1. Math Benchmark Performance Gain

Such gains of 20–35% in mathematical tasks demonstrate that emulation, despite its transparency, imposes a performance tax that becomes critical during neural network training or the processing of large arrays in Pandas.

The largest gap was recorded in Python interpreter startup operations. The python_startup test showed a gain of 95.71%, and python_startup_no_site showed a gain of 95.00%. This is explained by the fact that the Prism emulator requires time for JIT translation of x86 code blocks into ARM64 instructions during the first launch of a new module. For developers working in interactive environments or using microservice architecture, this means near-instantaneous response from the native system. The influence of execution modes on key system characteristics of Windows ARM is shown in Figure 2.

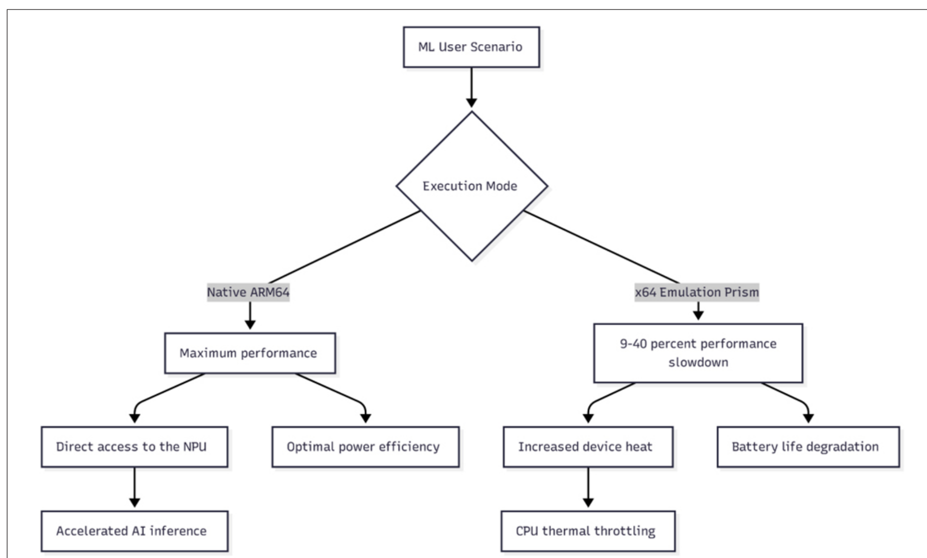

Fig. 2. The Impact of Execution Modes on Key System Characteristics of Windows ARM

Despite the advantages of the native environment, the process of its formation faces several fundamental obstacles. The lack of Fortran compilers in the MSVC toolchain for Windows ARM architectures (Geeson et al., 2024) is one of the main barriers to scientific Python in Windows ARM, since SciPy, the main Python package for scientific computing, uses Fortran for optimized linear algebra subroutines used in many ML algorithms.

On a native install of SciPy 1.14.1 the error ERROR: Unknown compiler(s): [['ifort'], ['gfortran'], ['flang']] is seen. Some progress has been made with the development of the LLVM Flang project which is expected to resolve the before mentioned error. The flang binary was included with LLVM 19.1.0, and maintainers demonstrated SciPy successfully building with Meson and flang (Brown, 2024). Nevertheless, this process has not been automated for the end user and requires manual environment configuration.

Scikit-learn provides a second representative case that illustrates how Windows ARM barriers extend beyond the Fortran toolchain. In a native Windows 11 ARM64 environment, Meson may complete the configuration stage and correctly detect the compiler, while the build still fails during the Ninja phase. One reported failure stops at the stage of creating an intermediate directory for _pairwise_distances_reduction, which prevents metadata generation and package installation. This case is important because it shows that successful builds of NumPy and SciPy in the same environment do not guarantee a successful build of downstream libraries whose source trees and generated artifacts place additional stress on the build system.

A different failure mode is observed in ONNX. The package has been identified as lacking readily available Windows ARM64 wheels on PyPI, which shifts the burden to source builds and raises the entry cost for users who need ONNX as a direct dependency or as part of a larger inference stack. This constraint has practical consequences for tools that depend on ONNX in conversion or deployment pipelines, since the absence of a native wheel converts a standard installation step into a toolchain and packaging problem. In the compatibility test suite, ONNX was recorded as a failed native build, which makes it a useful example of a packaging bottleneck rather than a compiler-specific defect.

XGBoost reflects a third category of failure that is tied to cross-platform build infrastructure and incomplete validation across target architectures. Public build reports associated with the vcpkg import effort show failures for arm64_windows alongside several other targets, while successful results were obtained on mainstream x64 Windows and selected Android and Linux configurations. This pattern indicates that Windows ARM support is influenced by the state of continuous integration, dependency packaging, and platform-specific build logic, even in projects with mature x64 support. In the present study, the native compilation attempt for XGBoost also failed, which supports its use as an example of ecosystem immaturity beyond the SciPy-centered dependency chain.

To support complex ML tools dependent on proprietary x64 plugins or third-party libraries, Microsoft introduced the ARM64EC architecture for applications, which is a special ABI allowing a mixture of native ARM64 and emulated x64 binaries to run in the same address space (Microsoft Learn, 2024a).

This is part of the ARM64EC ecosystem, which is a set of technologies that allow ARM and x64 components to run within the same process and interoperate with each other. Within this mechanism, ARM registers are projected onto equivalent x64 registers. For example, register x0 corresponds to RCX. At the same time, the use of some ARM registers is restricted. These include x13, x14, x23, x24, as well as v16–v31. This restriction is necessary in order that both architectural environments remain context compatible and interact with each other in a stable way.

Another important mechanism is thunks or adapters. The system automatically creates Entry Thunks to call ARM code from x64, and likewise creates Exit Thunks to call back to x64 code. Thunks make it easier to migrate the most compute-intensive Python modules, such as C++ extensions. Such an approach allows critical components to be adapted without waiting for the migration of the entire software stack.

Fast-Forward Sequences, or FFS, also play a substantial role. This technology is intended to reduce latency during function interception. Its application provides near-native call speed between different architectures. As a result, cross-architecture interaction becomes more efficient and incurs no visible overhead.

For ML libraries, this opens a path toward incremental porting, where the mathematical core runs natively while secondary dependencies run under emulation.

The key advantage of modern ARM processors is their dedicated neural processor. In Snapdragon X Elite, the Hexagon NPU provides up to 45 TOPS (Microsoft Learn, 2024b). However, access to this capability from Python code remains fragmented. The process of running ML model inference on Windows ARM using the NPU is shown in Figure 3.

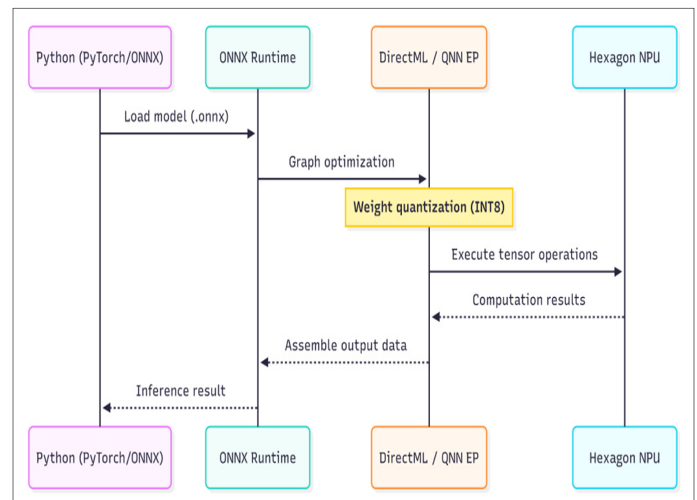

Fig. 3. The process of running ML model inference on Windows ARM using NPU

There are two main strategies for NPU integration. The first is associated with DirectML. This is a low-level Microsoft API integrated into DirectX 12. As of August 2024, DirectML supports the Snapdragon X Elite NPU (Pralle, 2023). Its advantages include independence from a specific hardware

vendor. At the same time, delays are currently observed in support for the latest transformer architectures used in LLMs.

The second strategy involves using the Qualcomm QNN SDK. This specialized toolkit is designed to achieve maximum performance by providing direct access to the Hexagon architecture. Integration with the QNN Execution Provider in ONNX Runtime is considered the most effective approach for Python applications (Microsoft Learn, 2024b).

An important finding during inference testing was that NPU use can paradoxically be slower than CPU execution if the software stack and drivers are not optimized for the specific graph architecture.

Windows ARM for machine learning faces the first key problem at the transition point from general compatibility to real performance. x64 emulation in Windows 11 makes many Python programs runnable, but for ML tasks, this is insufficient. As shown in the tests, native Python ARM64 provides an average performance gain of 22.7%, and in 90% of tests, the improvement reaches 40.2%. The solution lies in porting critical libraries to ARM64, since native execution reduces latency and better reveals processor capabilities.

The second problem is associated with the incomplete readiness of the scientific Python stack itself. Although about 93% of popular packages already support Windows ARM natively or are platform-independent, the remaining 7% include the most important libraries for ML. SciPy, Scikit-learn, TensorFlow, XGBoost, and JAX remain problematic. This makes the environment incomplete and constrains model development. The practical solution is to rely on components that already work, such as NumPy, Pandas, Matplotlib, and ONNX Runtime, while expanding native support for the remaining libraries in parallel.

The third problem is the Fortran wall. SciPy depends on Fortran subroutines, and on Windows ARM, there was, for a long period, no compatible compiler that could be integrated into the standard build chain. The next necessary step is to automate this procedure for the ordinary user. A practical route is to integrate LLVM Flang into the Meson-based SciPy build process through explicit compiler selection in the environment or in a Meson native file. This configuration turns the missing-Fortran problem into a defined build step and creates a reproducible path for native Windows ARM64 compilation.

The fourth problem concerns complex ML tools that depend on proprietary x64 libraries and plugins. Full migration of the entire stack under such conditions becomes too labor-intensive. For this purpose, Windows provides the ARM64EC mechanism, which allows ARM64 and x64 code to run in a single address space. The solution here consists of phased adaptation. The heaviest computational modules can be ported to ARM64 first, while secondary dependencies are temporarily left under emulation. Such an approach accelerates the transition and reduces technical risk.

The fifth problem concerns the use of the Hexagon NPU. The hardware accelerator delivers high computational performance, but Python access remains fragmented. The presence of an NPU does not yet guarantee inference acceleration, because even with weak driver and graph-structure optimization, the model may run slower than on the CPU. The solution is presented in two variants. DirectML is suitable as a universal integration layer. QNN SDK and the QNN Execution Provider in ONNX Runtime provide a path toward higher performance in Python applications.

The sixth problem lies in the immaturity of the build and package distribution infrastructure. Even when a library can be built, the user often has to manually configure Visual Studio 2022, ARM64 build tools, and dependency chains. This complicates the deployment of the environment and slows the platform's mass adoption. The solution consists of developing win_arm64 binary wheels for PyPI, standardizing build scenarios, and running automated tests on native ARM64, in x64 emulation, and in a control x64 environment. Such an approach makes the environment predictable and suitable for practical ML development.

Beyond technical incompatibilities, the pace of ecosystem enablement is also shaped by maintainer engagement and organizational constraints. In some cases, projects remain effectively blocked because maintainers do not review pull requests, lack access to Windows ARM hardware, or cannot justify the cost of adding ARM64 CI pipelines. Key challenges and their solution are illustrated in Table 3.

Table 3. Key challenges and their solution

Challenge	Why it matters on Windows ARM	Solution
Limits of x64 emulation for ML	Emulation slows ML workloads and increases latency.	Port key libraries to ARM64 and use native execution.
Incomplete scientific Python stack	Important ML libraries still have limited support.	Use the packages that already work and expand native support for the rest.
The Fortran wall in SciPy builds	Without a Fortran compiler, SciPy and dependent packages cannot be built.	Use LLVM Flang and automate Meson-based builds.
Dependence on x64 components	Full stack migration is difficult because some tools still rely on x64 libraries.	Use ARM64EC and port compute-heavy modules step by step.

Fragmented Python access to the NPU	Without optimization, the NPU can run slower than the CPU.	Use DirectML or QNN SDK with the QNN Execution Provider in ONNX Runtime.
Immature build and packaging infrastructure	Manual environment setup slows adoption.	Expand win_arm64 wheels, standardize build workflows, and improve testing.

From the perspective of CPython and JIT on ARM, a substantial contribution to platform development is made by the collaboration between Arm Holdings and Microsoft to optimize the CPython interpreter itself (Python, 2024). In Python 3.13, an experimental Just-in-Time compiler was introduced, for which Arm made a substantial contribution by optimizing code generation for the AArch64 architecture. The main achievements include reduction in the size of generated headers, increased branch efficiency in ARM machine code, and reduced memory overhead through the reuse of trampolines. These changes lay the foundation for long-term performance improvements across all Python applications on Windows ARM, making the interpreter more responsive to the execution characteristics of RISC architectures.

CONCLUSION

The research shows that the emergence of Windows ARM as a full-fledged machine learning platform is determined by the maturity of the Python ecosystem. The results confirm that native Python ARM64 execution provides a substantial gain compared to x64 emulation via Prism. This is especially noticeable in computation-intensive scenarios, where reducing overhead becomes critical for the practical use of ML libraries.

At the same time, the current state of the scientific stack on Windows ARM remains heterogeneous. With a high overall share of compatible packages, a number of key libraries, including SciPy, Scikit-learn, TensorFlow, XGBoost, and JAX, form a zone of systemic constraints. The central barrier remains the problem of building Fortran-dependent components, while additional complexity is created by the immaturity of the binary package infrastructure, the labor-intensive nature of manual environment configuration, and fragmented access to hardware accelerators, including the Hexagon NPU.

Thus, the development of a full-fledged ML environment on Windows ARM should be accompanied by several complementary directions. These include expanding native library support, automating builds with LLVM Flang, developing incremental porting approaches through ARM64EC, and deeper integration with ONNX Runtime, DirectML, and the QNN Execution Provider.

ACKNOWLEDGEMENTS

The author is an employee of Microsoft. The opinions and conclusions expressed herein are those of the author and do not necessarily represent the official policy, endorsement, or position of Microsoft.

REFERENCES

1. Baller, S. P., Jindal, A., Chadha, M., & Gerndt, M. (2021). DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices. *ArXiv*. https://doi.org/10.48550/arxiv.2108.09457
2. Brown, N. (2024). Fully integrating the Flang Fortran compiler with standard MLIR. *ArXiv*. https://doi.org/10.48550/arXiv.2409.18824
3. Curcic, M., Čertík, O., Richardson, B., Ehlert, S., Kedward, L., Markus, A., Pribec, I., & Vandenplas, J. (2021). *Toward Modern Fortran Tooling and a Thriving Developer Community*. ArXiv. https://arxiv.org/abs/2109.07382
4. Geeson, L., Brotherston, J., Dijkstra, W., Donaldson, A. F., Smith, L., Sorensen, T., & Wickerson, J. (2024). Mix Testing: Specifying and Testing ABI Compatibility of C/C++ Atomics Implementations. *Proceedings of the ACM on Programming Languages*, *8*, 442–467. https://doi.org/10.1145/3689727
5. Hammond, J., Dalcin, L., Schnetter, E., PéRache, M., Besnard, J.-B., Brown, J., Gadeschi, G. B., Byrne, S., Schuchart, J., & Zhou, H. (2023). MPI Application Binary Interface Standardization. *Proceedings of the 30th European MPI Users' Group Meeting*, 1–12. https://doi.org/10.1145/3615318.3615319
6. Jacobides, M. G., Cennamo, C., & Gawer, A. (2024). Externalities and complementarities in platforms and ecosystems: From structural solutions to endogenous failures. *Research Policy*, *53*(1), 104906. https://doi.org/10.1016/j.respol.2023.104906
7. Kanungo, P. (2023). Machine learning implementation in Python: Performance analysis of different libraries. *World Journal of Advanced Research and Reviews*, *20*(1), 1390–1398. https://doi.org/10.30574/wjarr.2023.20.1.2144
8. Microsoft Learn. (2024a, June 3). *Understanding Arm64EC ABI and assembly code*. Microsoft Learn. https://learn.microsoft.com/en-us/windows/arm/arm64ec-abi
9. Microsoft Learn. (2024b, June 12). *Copilot+ PCs Developer Guide*. Microsoft Learn. https://learn.microsoft.com/en-us/windows/ai/npu-devices/
10. Microsoft Learn. (2024c, June 18). *How emulation works on Arm*. Microsoft Learn. https://learn.microsoft.com/en-us/windows/arm/apps-on-arm-x86-emulation
11. Pralle, C. (2023, December 14). *DirectML: Accelerating AI on Windows, now with NPUs - DirectX Developer Blog*. DirectX Developers. https://devblogs.microsoft.com/directx/windows-directml-with-npus/

12. *Pyperformance*. (n.d.). GitHub. Retrieved December 3, 2024, from https://github.com/python/pyperformance
13. Python. (2024). *Python Release - Python 3.13.0*. Python. https://www.python.org/downloads/release/python-3130/
14. Rahman, T. N., Khan, N., & Zaman, Z. I. (2024). Redefining Computing: Rise of ARM from consumer to Cloud for energy efficiency. *World Journal of Advanced Research and Reviews*, *21*(1), 817–835. https://doi.org/10.30574/wjarr.2024.21.1.0017

Citation: Gleb Khmyznikov, "Enabling Python Machine Learning Libraries on Windows ARM: Challenges and Solutions", Universal Library of Engineering Technology, 2024; 1(2): 93-99. DOI: https://doi.org/10.70315/uloap.ulete.2024.0102015.

Copyright: © 2024 The Author(s). This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.