ChatGPT

AI Frameworks on Windows: Full-Stack Overview

Windows supports a rich ecosystem of AI frameworks, from high-level model libraries down to low-level GPU kernels. This overview is organized by **stack layer**, covering each layer's key frameworks, hardware support (NVIDIA GPUs via CUDA, AMD CPUs/GPUs, etc.), programming language support (especially Python, C++, C#), and Windows-specific tools. Both **training** and **inference** use cases are addressed. Each section includes tables and brief explanations for clarity.

High-Level Model Libraries (Transformers & Domain-Specific APIs)

These libraries provide ready-to-use models and high-level APIs on top of core frameworks. They simplify working with state-of-the-art models (like Transformers) by abstracting training/inference details. Typically, they are **Python** libraries that leverage underlying frameworks (e.g. PyTorch or TensorFlow):

- **Hugging Face Transformers** – A popular library with thousands of pre-trained models for NLP, vision, audio, etc. It supports **PyTorch**, **TensorFlow**, and **JAX** backends for model execution [1]. Developers can train or infer transformer models with just a few lines of code. On Windows, Hugging Face Transformers works if the chosen backend framework (PyTorch/TF/JAX) is properly installed. It primarily offers a **Python API**, but models can be exported (e.g. to ONNX) for use in C++ or C# inference. Hugging Face's library is actively maintained and widely used for **Transformer-based** applications [1].

- **Keras** – Originally an independent high-level API, Keras is now the **high-level API of TensorFlow 2** [2]. It provides an intuitive interface to build and train neural networks (covering layers, training loops, etc.) with minimal code. Keras runs on top of TensorFlow (and recently introduced experimental support for other backends like JAX or PyTorch via KerasCV/KerasCore). On Windows, **tf.keras** is included with TensorFlow and supports **NVIDIA GPUs** (via CUDA) or CPU. Keras is **Python-only**, but models can be saved and loaded in other environments (e.g. via TensorFlow's C++ API or conversion to ONNX for C#). It is well-suited for quick prototyping and **fast experimentation** [2].

- **fast.ai** – A high-level library built on PyTorch that simplifies common deep learning tasks (especially in computer vision, text, tabular data). It provides concise APIs for training models with state-of-the-art techniques. fast.ai is **Python**-based and fully compatible with Windows (leveraging PyTorch under the hood). It is primarily used for training (with built-in best practices) and can utilize **NVIDIA GPUs** on Windows via PyTorch's CUDA support.

- **PyTorch Lightning** (Lightning AI) – This is a higher-level organizational framework on top of PyTorch. It decouples the science code from engineering, making it easier to write training loops, handle checkpointing, etc. It's popular for research and uses **Python**. Lightning runs on Windows (using PyTorch), supporting NVIDIA GPUs. (It's mainly a training tool; for inference one would typically export the trained model.)

Table 1: High-Level Model Libraries (Transformer & API Layer)

Library	Underlying Frameworks	Language Support	Windows GPU Support	Use Cases / Notes
Hugging Face Transformers [1]	PyTorch, TensorFlow, JAX	Python (primary)	NVIDIA GPUs (CUDA); AMD GPUs via underlying framework (limited – e.g. PyTorch AMD not on Win [3]); CPU (Intel/AMD)	Pre-trained Transformers for NLP, vision, etc. Active community.
Keras (tf.keras) [2]	TensorFlow 2.x	Python (primary)	NVIDIA GPUs (CUDA); AMD GPU not natively (requires DirectML fork); CPU (oneDNN optimized)	High-level neural network API in TensorFlow. Easy model building and training.
fast.ai	PyTorch	Python	NVIDIA GPUs (via PyTorch CUDA); AMD GPU (no native support on Win); CPU	Simplified training for vision/NLP; uses PyTorch underneath.
PyTorch Lightning	PyTorch	Python	NVIDIA GPUs (CUDA); AMD GPU (no native Win support); CPU	Training loop orchestration and research prototyping on PyTorch.

Note: AMD GPU support in high-level libraries depends on backend framework support. (As of 2024, PyTorch and TF do **not** support AMD GPUs on native Windows; see below for alternatives.) AMD CPUs are fully supported by these libraries via underlying CPU kernels, though vendor-specific optimizations (oneDNN, ZenDNN) may be in use.

Core Deep Learning Frameworks (Training & Inference)

Core frameworks provide the **building blocks to develop, train, and deploy** deep learning models. They include features for tensor operations, automatic differentiation, and hardware acceleration. The major frameworks and their Windows compatibility:

- **PyTorch** – An open-source deep learning framework known for its dynamic computation graph ("eager execution") and pythonic style. PyTorch has **first-class support on Windows** including GPU acceleration [4] [5]. Key points:
- *Languages:* Primarily **Python**, but also offers a C++ API (**LibTorch**) for deploying models in C++ applications. There are community bindings for .NET (e.g. **TorchSharp** for C#) and even experimental Java support [6]. Python remains the most feature-complete interface.
- *NVIDIA GPU:* Full support via CUDA on Windows. Official binaries are provided for various CUDA versions [7]. PyTorch uses NVIDIA's CUDA libraries (cuDNN, cuBLAS, NCCL, etc.) to accelerate training/inference on GeForce/RTX/Datacenter GPUs.
- *AMD GPU:* **Not supported on Windows** in stable releases as of 2024 – PyTorch's AMD GPU support relies on ROCm, which is Linux-only currently [3]. (AMD is working on ROCm for Windows – a HIP SDK was introduced in ROCm 5.5/5.6, but PyTorch Windows support for it is not yet available [3] [8].) On Windows, if using an AMD Radeon GPU, one must use alternatives like DirectML or run PyTorch in WSL2 with ROCm.

- *CPU:* PyTorch runs on CPU (both Intel & AMD). It is often compiled with **oneDNN** (Intel's Deep Neural Network Library) to optimize x86 performance. Recent PyTorch releases integrate vendor optimizations (e.g. Intel oneAPI extensions, and AMD's ZenDNN plugin on Linux for EPYC CPUs) for faster CPU inference [9] [10]. On Windows, PyTorch uses oneDNN and can utilize AMD CPUs (Zen) effectively, though AMD-specific ZenDNN is available only on Linux builds.

- *Training vs Inference:* PyTorch is used heavily for research and training. It also supports model export (to TorchScript or ONNX) for inference. PyTorch models can be served on Windows using **TorchServe** or converted to ONNX and run with ONNX Runtime for production.

- **TensorFlow** – A widely-used framework with a comprehensive ecosystem. TensorFlow 2.x emphasizes eager execution with the Keras API, while still allowing graph compilation (tf.function) for performance. Windows support is **mature**: official TensorFlow releases include Windows wheels for CPU and GPU. Key points:

- *Languages:* **Python** is primary (with high-level Keras API). TensorFlow also provides a C++ API (for advanced use cases, one can link against the TensorFlow C++ library to run graphs, although building it on Windows can be complex). There are **Java bindings** (TensorFlow Java for inference) and even community .NET bindings (e.g. SciSharp's **TensorFlow.NET** which ML.NET uses [11]). However, most users interact via Python.
- *NVIDIA GPU:* Fully supported on Windows via CUDA. You install `tensorflow` with GPU support, and it uses CUDA and cuDNN (one must have a matching NVIDIA driver and CUDA toolkit installed). It supports training and inference on NVIDIA GPUs out-of-the-box.
- *AMD GPU:* **Not officially supported on Windows** by Google. AMD provides ROCm-enabled TensorFlow for Linux, but not for Windows. Microsoft and AMD did collaborate on a special TensorFlow-DirectML package (a plugin that let TensorFlow run on any DirectX 12 GPU, including AMD) [12]. This allowed training on AMD GPUs via the DirectML layer. However, that project was discontinued in early 2025 [12], indicating limited long-term support. The current practical solution for AMD GPUs on Windows is using **DirectML** (see *Inference/Acceleration* section) or switching to Linux/WSL2 for ROCm support.
- *CPU:* TensorFlow runs on CPU using optimizations from **oneDNN** by default (since TF 2.6+). It is optimized for Intel architectures (AVX, etc.), and also runs on AMD CPUs (though older TensorFlow used Intel's MKL which sometimes underutilized AMD chips). AMD has introduced **ZenDNN** – a library to accelerate TensorFlow on AMD EPYC CPUs – via a plugin in TensorFlow 2.12+ [13] [14]. (ZenDNN is Linux-only, but it shows that AMD CPUs are also being specifically tuned for inference workloads.) On Windows, the default oneDNN execution is used for AMD CPUs as well (oneDNN is architecture-aware and includes some AMD Zen optimizations, albeit not as many as ZenDNN).

- *Training vs Inference:* TensorFlow is used in both scenarios. For training, it scales from single Windows workstations to multi-GPU servers (though multi-GPU on Windows may have limitations due to NCCL, as discussed later). For inference, TensorFlow models (SavedModel format) can be deployed via TensorFlow Serving (Linux only) or converted to TensorFlow Lite or ONNX for lightweight deployment.

- **JAX** – A newer framework from Google, combining NumPy-like syntax with **XLA** (Accelerated Linear Algebra) compilation for high performance. JAX is popular in cutting-edge research (e.g. large-scale TPU training, advanced math). However, JAX's Windows support is **limited**: official JAX wheels are available for Linux and macOS; Windows is not officially supported as of 2025. Users on Windows typically run JAX in a WSL2 environment (Linux subsystem) if they need GPU support. Under WSL2 (with a NVIDIA GPU), JAX can utilize CUDA drivers via the Windows host

(NVIDIA's CUDA on WSL). There are community-contributed ways to install JAX on Windows CPU, but GPU acceleration natively is not yet present. In summary, JAX is Python-only and for Windows developers it's recommended to use WSL2 or a Linux environment to leverage it. (For this reason, JAX is less common in Windows-centric workflows.) When used, JAX can compile models via XLA for NVIDIA GPUs and even AMD GPUs (if using ROCm on Linux), delivering very high performance for both training and inference.

- **MXNet (Apache MXNet)** – An open-source deep learning framework that supports multiple languages (Python, C++, R, Scala, etc.) and was once favored for its scalability. MXNet does have Windows support (it provides Windows Python wheels and can be compiled for Windows). It supports NVIDIA GPUs via CUDA. Historically, MXNet had some experimental support for AMD GPUs via OpenCL or Vulkan (through projects like Apache TVM or PlaidML integration), but those are not mainstream. MXNet is incubated at Apache and not as actively developed as PyTorch/TF. It's still used in some projects (and is the engine behind Amazon's Gluon API and some AWS services). For Windows developers, MXNet is an option if a multi-language, portable framework is needed, but the community and support are smaller. **Python and C++** are well-supported (it even allows writing custom C++ ops). Given the stronger popularity of PyTorch/TF, MXNet's role on Windows is now minor.

- **MindSpore and PaddlePaddle** – These are frameworks from Huawei and Baidu respectively. They are less common outside their origin countries. MindSpore has limited Windows support (primarily focused on Linux/Ascend hardware). PaddlePaddle (Baudu) does offer Windows packages with CUDA support. However, neither is a dominant choice in Windows environments, so we note them only for completeness.

Table 2: Core Deep Learning Frameworks

Framework	Windows Support & Languages	NVIDIA GPU (CUDA)	AMD GPU (ROCm/ DirectML)	CPU (Intel/ AMD)	Notable Features
PyTorch [15] [7]	**Yes** – official support on Windows 7/10+; Python API (best), C++ API (LibTorch), .NET via TorchSharp.	Yes – native CUDA support (pip wheels with CUDA 11/12) [15] [7].	*No native support* on Windows (ROCm Linux-only) [3]. *Workaround:* use WSL2 or DirectML for AMD GPUs.	Yes – optimized via oneDNN; uses MKL/ OpenBLAS for BLAS. AMD Zen supported (no ZenDNN on Win).	Dynamic graph, eager execution. Strong community. Good C++ deployment story (LibTorch).

Framework	Windows Support & Languages	NVIDIA GPU (CUDA)	AMD GPU (ROCm/ DirectML)	CPU (Intel/ AMD)	Notable Features
TensorFlow 2.x	**Yes** – official Windows pip packages; Python API (with tf.keras). C++ API available (manual build). Java and Go bindings for inference.	Yes – native CUDA support (must install matching CUDA/ cuDNN).	*No official support* (Windows). AMD/Microsoft provided TensorFlow-DirectML (now discontinued) [12] for DX12 GPUs. Otherwise, use ONNX or WSL2 for AMD GPUs.	Yes – oneDNN optimized. Uses AVX/ VNNI on Intel; runs on AMD (ZenDNN plugin on Linux for AMD).	Mature ecosystem, includes high-level Keras API. Graph (XLA) or eager modes. Vast tooling (TensorBoard, etc.).
JAX	**Partial** – No native Windows binaries; Python-only. Runs via WSL2 or in Docker.	Yes (via WSL2 Linux) – uses CUDA and cuDNN through XLA.	Not yet (ROCm support on Linux only; no Windows).	Yes (CPU JAX can run on Windows Python, but without official wheel).	XLA just-in-time compiler for high performance. Used in research (Flax, etc.).
MXNet	**Yes** – Windows support present. APIs in Python, C++, Java, Scala, R.	Yes – CUDA support for Nvidia GPUs.	Experimental (OpenCL/ Vulkan backends existed, not mainstream).	Yes – Uses MKL or OpenBLAS for CPU ops.	Hybrid static/ dynamic graph (Gluon API). Less active community now.
Others	**MindSpore, PaddlePaddle** (China-origin): Windows support limited (Paddle has CUDA Windows binaries; MindSpore mostly Linux). **CNTK** (Microsoft Cognitive Toolkit, C++/.NET) was Windows-friendly but is *discontinued*.	N/A	N/A	N/A	N/A

Note: Multi-GPU training on Windows is possible for frameworks like PyTorch/TF, but there are caveats. NVIDIA's NCCL (collective communication library) is **not supported on Windows**, which means distributed data-parallel training might fall back to slower alternatives (e.g. Gloo or MPI) [16]. Single-machine multi-GPU can still be utilized (PyTorch, for example, will use multiple processes with CPU binding for gradients if NCCL is unavailable). For multi-node clusters or optimal multi-GPU throughput, Linux is often preferred. Windows users needing distributed training may use WSL2 or containers to leverage full NCCL support [17].

Training Optimization & Distributed Frameworks

For large-scale training (especially of transformer models or other huge networks), additional frameworks and libraries are used on top of core frameworks to optimize performance and memory usage:

- **DeepSpeed** – A deep learning optimization library by Microsoft, geared towards efficient **distributed training** of large models. It provides features like ZeRO for memory optimization (allowing training of models larger than GPU memory), pipeline and tensor parallelism, etc. DeepSpeed integrates with PyTorch (you wrap your model with it). **Windows support:** Many DeepSpeed features are supported on Windows (as of v0.9+), though some advanced components (like NVMe offloading or NCCL-based ops) may be limited [18]. Microsoft provides precompiled Windows pip wheels for DeepSpeed. It can be used for **training** on multi-GPU Windows machines, but for multi-node training or maximum performance, Linux is still the primary environment. For example, on Windows, DeepSpeed will use TCP/Gloo backend for communication (since NCCL isn't available). Still, for a single high-end Windows workstation with multiple GPUs, DeepSpeed can significantly speed up training and handle large model sizes.

- **Horovod** – An Uber-contributed distributed training framework (for TensorFlow, PyTorch, etc.) using MPI/all-reduce. **Not supported on Windows** (officially) [19]. Users must use WSL2 or Linux containers for Horovod. Windows developers typically opt for DeepSpeed or PyTorch's native `DistributedDataParallel` for multi-GPU.

- **Distributed TensorFlow** – TensorFlow's built-in distribution strategies (MirroredStrategy, etc.) can be used on Windows for multi-GPU training on one machine. For multi-node, TensorFlow on Windows is rarely used (one would run on Linux servers). Azure and other cloud services provide Linux-based TF clusters for that need.

- **Lightning Fabric / Ray** – Higher-level orchestration like Lightning's Fabric or Ray Tune can manage distributed training or hyperparameter search across nodes. These can run on Windows for simple cases, but are more commonly run on Linux.

In summary: Windows can be used for training models on single machines (even with multiple GPUs), and libraries like DeepSpeed are making this more efficient. For distributed multi-machine training, Windows is uncommon – WSL2 or cloud Linux instances are typically leveraged.

Inference and Deployment Runtimes

When deploying AI models (for predictions in production), specialized runtimes can provide optimized performance and easy integration. These focus on **inference** (not training) and often use model formats like ONNX or saved models. Key frameworks/tools:

- **ONNX Runtime (ORT)** – A **highly optimized inference engine** for ONNX (Open Neural Network Exchange) models, developed by Microsoft. ONNX is an open standard format that many frameworks can export to [20]. ONNX Runtime is cross-platform and particularly well-supported on Windows (it's even built into Windows 10/11 as "Windows ML"). Highlights:
- *Hardware support:* ORT supports **CPU (Intel/AMD)** with optimizations (it has a built-in CPU execution provider using oneDNN and other accelerations), **NVIDIA GPUs** via a CUDA execution provider (requires CUDA and cuDNN installed) [21], **Intel GPUs** via an OpenVINO or oneAPI execution provider, and **AMD GPUs** via the **DirectML** execution provider [22]. In fact, ONNX Runtime's DirectML EP allows running on **any DirectX12-capable GPU** (including AMD Radeon and even Qualcomm Adreno or Intel integrated GPUs) [22]. This makes it unique for Windows: ORT can use *whatever GPU is in the machine* using DML, albeit with performance depending on the quality of the GPU's DX12 driver ML support.
- *Language support:* ONNX Runtime offers APIs in many languages – Python, C++, C#, Java, even JavaScript for web via WebAssembly. On Windows, a NuGet package (`Microsoft.ML.OnnxRuntime`) allows easy use from C#/.NET applications [23] [24]. It's also accessible via WinRT (Windows.AI.MachineLearning namespace) for C++ UWP apps [25].
- *Performance:* ORT is optimized with graph optimizations, operator fusions, and uses different backends (TensorRT, OpenVINO, DML, etc.) to maximize throughput. It's actively maintained and often updated to support new accelerators (including NPUs). Microsoft recommends ONNX Runtime as the best way to integrate AI into Windows apps for local inference [26] [22].

- *Use cases:* Any scenario where you have a trained model (from PyTorch, TF, scikit-learn, etc.) and want to deploy it on Windows in a lean, fast runtime. Export the model to ONNX, then load it in ORT for inference. This works for classical ML and deep learning models alike.

- **NVIDIA TensorRT** – A proprietary high-performance inference engine by NVIDIA. It takes trained neural network models (often via ONNX or PyTorch direct export) and performs optimizations like FP16/INT8 quantization, layer fusion, and Tensor Core optimization, producing a deployable **engine** that runs extremely fast on NVIDIA GPUs. On Windows, TensorRT is available as part of the NVIDIA TensorRT SDK (compatible with Windows; requires an NVIDIA GPU). Developers can use TensorRT via a C++ API or a Python API. It's primarily used in scenarios where maximum inference throughput or lowest latency is needed (e.g., real-time inferencing, high-QPS servers) on NVIDIA hardware. TensorRT only works with NVIDIA GPUs (it's tied to CUDA drivers). Windows support is actively maintained – many enterprises deploy TensorRT on Windows servers or edge devices with GPUs. For example, one might export a TensorFlow model to ONNX, then use TensorRT Python API on Windows to load the ONNX, build an engine, and run inference, achieving much higher FPS than raw TensorFlow or PyTorch. (TensorRT can also integrate with ONNX Runtime as an execution provider – so ORT on Windows can hand off subgraphs to TensorRT for acceleration if a NVIDIA GPU is present.)

- **Intel OpenVINO** – An inference toolkit from Intel, geared towards Intel hardware (CPU, iGPU, VPU). OpenVINO includes a model optimizer and runtime. It can take models from TF, ONNX, PyTorch (via ONNX) and optimize them for Intel's CPUs (leveraging AVX512, etc.) and integrated GPUs (via oneAPI Level Zero or OpenCL). On Windows, OpenVINO is well-supported – Intel provides installers and it's used in many Windows computer vision applications that run on CPU

(for example, accelerating inference on Intel Core CPUs by ~x2-x3 vs unoptimized frameworks [27] [14]). Language support is typically C++ (for integration into native apps) and Python (they provide Python bindings for ease of use in prototyping). OpenVINO's runtime can serve as an execution provider in ONNX Runtime as well, or be called directly. **AMD CPU/GPU:** OpenVINO mainly focuses on Intel devices; it won't optimize for AMD-specific features (though standard ONNX Runtime or oneDNN might cover those). For Windows developers using Intel CPUs/GPU, OpenVINO is a powerful tool for **inference optimization**.

- **Windows ML (WinML)** – This is the Windows 10/11 built-in ML inference API (WinRT API). Under the hood, it is essentially ONNX Runtime packaged with the OS [25]. WinML allows developers to load ONNX models and evaluate them, with the OS handling hardware selection (using CPU or DirectML for GPU/NPU). While one could use WinML APIs directly in C++/C# UWP apps, most developers now use the ONNX Runtime NuGet for more flexibility (as it's the same core engine with more control). WinML is notable for being *in-box* – no additional installs needed for basic model scoring on modern Windows. It supports passing `VideoFrame` objects directly for vision models [28], making it convenient for camera or media apps.

- **TensorFlow Lite (TFLite)** – A lightweight inference runtime primarily for mobile/IoT. On Windows, TFLite can be used (there are C++ libraries you can compile, and even a WinML plugin to run TFLite delegates). However, it's not commonly used on Windows since ONNX Runtime and full TensorFlow are available. If someone is targeting low-power devices or want to use TFLite models, they might run them on Windows for testing. Generally, not a first choice for Windows deployment (except perhaps on ARM64 Windows devices where TFLite might be lighter weight).

- **Others:** There are other deployment solutions like **NVIDIA Triton Inference Server** (which can run on Windows but is more often on Linux for serving multiple models with HTTP/GRPC endpoints), and **Microsoft Azure ML** local endpoints (containerized deployments). For game development and Unity, there's **Unity Barracuda** (a C# inference engine using ONNX/TFLite under the hood) which supports Windows for running ML in Unity games. In .NET, besides ONNX Runtime, one can also use **ML.NET** for certain scenarios (discussed below).

Table 3: Inference/Deployment Frameworks

Framework / Runtime	Supported Hardware (Windows)	Language APIs	Notes on Windows Optimization
ONNX Runtime [22] [29]	CPU (Intel & AMD, optimized), NVIDIA GPU (CUDA), AMD/ Intel/Qualcomm GPU (via DirectML) [22], Intel iGPU/VPU (via OpenVINO EP), ARM CPUs (on Windows ARM). Also supports NPUs (via DirectML or specific EPs).	Python, C++, C# (.NET) [23] [24], Java, JavaScript	Microsoft's universal high-performance inference engine. Highly recommended for Windows apps [26]. Integrates with WinML, DirectML.

Framework / Runtime	Supported Hardware (Windows)	Language APIs	Notes on Windows Optimization
NVIDIA TensorRT	NVIDIA GPUs only (requires CUDA-capable GPU and NVIDIA drivers).	C++ (primary), Python (bindings)	Maximizes throughput on NVIDIA hardware via model-specific optimizations. Windows is fully supported (part of NVIDIA SDK). Great for low-latency apps on RTX/Tesla GPUs.
Intel OpenVINO	Intel CPUs (SSE/AVX/VNNI optimizations), Intel iGPUs (Intel HD/UHD, Iris Xe via oneAPI drivers), Intel Movidius VPUs.	C++ (primary), Python (API bindings)	Accelerates inference on Intel hardware. Windows support is strong; often yields significant speedup on Intel CPU vs generic runtime.
Windows ML (WinML)	DirectX12 GPUs (through DirectML) and CPU – uses ONNX Runtime internally. Also can leverage dedicated NPUs on "Copilot" PCs [30] [31].	C++ (WinRT API), C# (via Windows.AI.MachineLearning WinRT interop)	Built-in Windows inference engine for ONNX models [25]. Convenient for UWP/Packaged apps; for most .NET apps, using ONNX Runtime NuGet (which uses same core) is preferred.
ML.NET (ONNX support)	CPU, GPU via ONNX Runtime (internally)	C# (.NET)	ML.NET can consume ONNX models in .NET apps, calling into ORT behind the scenes [32]. Simpler for developers who are already using ML.NET for other ML tasks.

Framework / Runtime	Supported Hardware (Windows)	Language APIs	Notes on Windows Optimization
TensorFlow (Serving or C++ API)	CPU, NVIDIA GPU (CUDA)	Python (for tf.keras models in-process), C++ (for using TF C API), or use TF-Serving (C++ server)	One can deploy a TensorFlow model on Windows by loading it in Python and serving via Flask, etc., but this is heavier. TensorFlow Serving doesn't officially target Windows. Usually prefer converting to ONNX/ORT on Windows.

Windows **NPUs** (like the Qualcomm Hexagon DSP in Snapdragon or the **AMD Ryzen™ AI Engine** in new Ryzen 7040 chips) are also supported via the above inference stack. DirectML can target these NPUs when available [30] [31]. For example, an "NPU-enabled" Windows 11 device will offload ONNX Runtime inference to the NPU if possible, freeing CPU/GPU [31]. This is an emerging area – as of 2025, frameworks will increasingly integrate NPU support (ONNX Runtime already has experimental NPU execution providers). AMD's Ryzen AI and Intel's upcoming AI accelerators come with SDKs (like AMD's **Ryzen AI SDK** and tools to convert models to target their XDNA NPU).

Compiler and Optimization Layers (XLA, JITs, etc.)

This layer sits within or atop frameworks to **optimize model execution** via code generation and graph-level transformations. Key components include:

- **XLA (Accelerated Linear Algebra)** – A compiler for linear algebra that **TensorFlow and JAX** use to optimize computations. XLA can compile a compute graph into efficient machine code targeted at CPU or GPU, improving speed. On Windows, XLA is bundled in TensorFlow (and used when you enable `tf.function` with XLA or run tf.Graph mode). For JAX, XLA is essential but since JAX isn't native on Windows, XLA's use is mostly behind the scenes (in WSL2). XLA's impact is seen in faster training/inference when using fused kernels. Google's **OpenXLA** initiative (open-sourcing XLA) means more frameworks might leverage it across platforms.

- **TorchScript & TorchInductor (PyTorch 2.x)** – PyTorch traditionally executed models eagerly, but it provides **TorchScript** to serialize models and run them in a C++ environment, and **Torch Compile (Inductor)** to generate optimized code. TorchInductor in PyTorch 2 uses a Python CUDA compiler called **Triton** to generate specialized GPU kernels, and uses LLVM to optimize CPU code [33]. This JIT compilation can give significant speedups. On Windows, TorchScript works (you can save a `scriptModule.pt` and load in C++ LibTorch). TorchInductor is also supported on Windows for CPU and NVIDIA GPUs (it falls back to eager if a kernel can't be compiled). These compiler technologies make PyTorch more competitive in inference performance without user intervention.

- **ONNX Runtime Graph Optimizer** – Before executing a model, ORT performs graph-level optimizations (constant folding, operator fusion, etc.), and can even apply **kernel fusion** for CPU

or call external libraries like oneDNN Graph [34]. ORT's use of providers like TensorRT means it's leveraging those compilers too.

- **TVM** – Apache TVM is an ML compiler framework that can compile models from various frontends (TensorFlow, PyTorch, ONNX) to optimized code for CPU, GPU, or even specialized accelerators. TVM does support Windows (to some extent, via LLVM backend). It's more of a tool for researchers or performance engineers than everyday use, but it's part of the stack for those who want to generate highly optimized inference code targeting specific hardware or even to create CUDA kernels for Nvidia, or HLS for FPGAs. A Windows developer could use TVM in Python to auto-tune and compile a model for, say, an AMD CPU or an NVIDIA GPU, but this is advanced usage.

- **Intel oneAPI AI Analytics** – Intel provides the Intel® oneDNN and related tools, plus Intel Extension for PyTorch and TensorFlow, which can be seen as part of the compiler/optimization layer. For instance, **Intel Extension for PyTorch** can transparently swap in optimized kernels or use graph mode to accelerate inference on Intel GPUs and CPUs [35]. On Windows, Intel's extensions are available (oneAPI toolkit supports Windows). This means if you run PyTorch on a new Intel GPU (Arc or integrated Xe), you can install `intel_extension_for_pytorch` and get improvements [34] [36]. Similarly, for TensorFlow, Intel's oneDNN and oneDAL optimizations are built-in.

- **Quantization and Pruning tools** – These reduce model precision or size to run faster. PyTorch has quantization-aware training and dynamic quantization utilities; TensorFlow has the `tf.quantization` and TFLite tooling. ONNX Runtime has a Post-Training Quantization tool as well. These aren't frameworks themselves but techniques supported across the stack to optimize models for inference (especially on Windows devices where you might trade off some accuracy to run on CPU or smaller GPU).

In summary, the compiler layer ensures that whatever framework you use, the computations are optimized – either by fusing many operations into one GPU kernel (reducing overhead) or by leveraging vector instructions on CPU. Windows users benefit from these just as on Linux. For instance, the AMD ZenDNN plugin essentially serves as an optimized kernel library for AMD CPUs, integrated via the plugin mechanism in TF/PyTorch [9] [10]. Similarly, DirectML itself can be seen as a "compiler" that translates high-level ML ops into DirectX GPU shader programs, so that any GPU can execute them.

Hardware Acceleration Libraries & Drivers

At the lowest level of the stack, we have vendor-specific libraries and drivers that do the heavy lifting:

- **CUDA Toolkit (NVIDIA)** – This includes the CUDA driver and runtime, and libraries like **cuDNN** (CUDA Deep Neural Network library for fast conv/pooling ops), **cuBLAS** (BLAS for GPU), **TensorRT** (for inference), and **NCCL** (for multi-GPU comms, though NCCL is Linux-only). On Windows, developers install the NVIDIA driver and CUDA Toolkit to enable GPU acceleration. Frameworks like PyTorch/TF bundle the needed CUDA libraries (e.g., PyTorch's pip wheel comes with cuDNN, etc.), but you still need a system driver. The CUDA toolkit integrates with **Visual Studio** for C++ development – NVCC (the CUDA compiler) uses MSVC on Windows to build CUDA code into device kernels [37]. For any custom CUDA development or compiling frameworks from source on Windows, Visual Studio with the appropriate CUDA toolkit is required [37].

- **ROCm (AMD)** – AMD's counterpart to CUDA. It includes the **HIP** runtime (CUDA-like platform), **MIOpen** (equivalent to cuDNN for AMD GPUs), and **rocBLAS**, etc. Historically ROCm was Linux-only, but in 2023 AMD began releasing ROCm for Windows (alpha support for some Radeon GPUs) [8] [38]. This means AMD GPUs on Windows might soon be able to run frameworks via ROCm/HIP. Currently, however, mainstream frameworks do not yet provide Windows ROCm builds. The **HIP SDK 5.5+** is available on Windows for developers who want to compile HIP code or potentially build PyTorch from source with AMD support (as some enthusiasts have attempted) [39] [40]. It's an evolving area – for now, using AMD GPUs on Windows for AI relies on DirectML or simply using the GPU for display and doing compute on CPU. But the groundwork for ROCm on Windows is being laid by AMD [8].

- **oneDNN / MKL-DNN (Intel)** – This is an open-source library of optimized CPU primitives for deep learning (maintained by Intel, part of oneAPI). It's used by TensorFlow, PyTorch, ONNX Runtime, and others to accelerate operations like convolutions, LSTMs, etc., on CPUs. oneDNN is optimized for Intel architectures (taking advantage of AVX2, AVX-512, VNNI, etc.), but it also runs on AMD CPUs (since AMD CPUs support many of the same instructions). In recent versions, oneDNN has become pretty neutral and often benefits AMD Zen as well [41]. On Windows, oneDNN is included within these frameworks; developers typically don't use it standalone (though you could if writing a custom engine). Intel's **MKL** (Math Kernel Library) is another low-level library used for BLAS routines. Many Python packages (NumPy, etc.) use MKL on Windows. MKL is tuned for Intel but also runs on AMD (with some performance penalty historically, though that has lessened). AMD's alternative BLAS is **BLIS/AOCL** – which could be used if building a framework from source to optimize for AMD CPU. AMD's AOCL includes FFTs, BLAS, etc., and is optimized for Zen architecture [42] [43].

- **ZenDNN (AMD)** – Mentioned earlier, ZenDNN is AMD's analog to oneDNN, tailored for AMD EPYC/Zen CPUs [9]. It reuses some oneDNN infrastructure but adds new optimizations for AMD. Currently, ZenDNN is officially supported on Linux (Ubuntu/RHEL) and integrated as a plugin for TensorFlow and PyTorch there [10]. On Windows, ZenDNN is not available (since it's delivered as Linux .so libraries). However, AMD's CPU optimizations in general (like leveraging AVX2 on older Zen and AVX-512/VNNI on Zen4) will still benefit frameworks on Windows via oneDNN. We might see AMD release ZenDNN for Windows in the future, given their push for better Windows AI performance (especially as Windows gets more AMD-based AI silicon).

- **DirectML** – A Windows-specific hardware abstraction for machine learning. It is part of DirectX 12. DirectML provides a set of ML operators implemented on shaders or compute kernels that can run on any DirectX12 GPU (including integrated graphics, Xbox, etc.). On Windows, DirectML is used under the hood by ONNX Runtime's DML execution provider and by WinML [22]. It means developers normally don't call DirectML directly (unless writing a custom engine using D3D12). But it's an important layer enabling GPU support on non-CUDA GPUs. For example, an AMD Radeon RX 6800 can't run CUDA, but it can execute DirectML. TensorFlow-DirectML (when it was active) translated TF ops to DirectML calls. Performance-wise, DirectML is improving but might not match highly optimized CUDA/cuDNN for Nvidia or ROCm on AMD; still, it enables functionality and hardware acceleration where otherwise only the CPU would be used.

- **Visual Studio & MSVC** – While not a framework, it's worth noting the role of development tools. On Windows, compiling AI libraries from source (or custom ops) requires **Visual C++** compilers. PyTorch, for instance, uses MSVC toolchain for Windows builds [37]. Visual Studio 2022 is commonly used to compile CUDA code (with the proper toolset integration). Windows AI developers often use **Visual Studio Code** as an editor (with Python or C++ extensions) for convenience, but the actual builds rely on MSVC, CMake, etc. Visual Studio also provides profiling

tools (GPU Usage tool, CPU profilers) which can be used to analyze performance of AI workloads on Windows.

- **WSL2 (Windows Subsystem for Linux)** – This isn't hardware, but a compatibility layer. WSL2 allows running a Linux environment on Windows with near-native performance, including GPU acceleration (WSL2 can interface with the Windows host's GPU drivers). This means if something isn't supported natively on Windows (say JAX or a specific version of ROCm), a developer can install Ubuntu in WSL2 and run it there, using the Windows GPU driver (NVIDIA's driver has WSL support for CUDA; AMD has begun offering preview support for GPUs in WSL as well via DirectML). WSL2 has become a powerful tool for Windows AI developers to access the Linux ecosystem without dual-booting. For instance, one could run Docker containers with Linux-only AI software on a Windows machine via WSL2.

Finally, it's important to note that **actively maintained and optimized frameworks** on Windows are those we've focused on: PyTorch, TensorFlow, ONNX Runtime, etc., all have large development communities and regular updates. Some older or niche projects (Theano, CNTK, Caffe, etc.) are no longer active – modern Windows AI development gravitates towards the frameworks listed above. Microsoft's own efforts (DirectML, ONNX Runtime, ML.NET) ensure that Windows remains a first-class environment for AI, even as much AI development happens on Linux.

.NET and Windows-Specific AI Tools

For developers using **C#/.NET** or integrating into Windows apps, there are additional frameworks to mention:

- **ML.NET** – Microsoft's machine learning library for .NET. It is primarily for classical ML (regressions, classification, AutoML, etc.), but in recent versions it expanded to deep learning. ML.NET 3.0 (Nov 2023) introduced image classification, object detection, and NLP tasks powered by deep learning [44] [45]. Under the hood, ML.NET uses **TorchSharp** and **TensorFlow.NET** for these tasks [11]. TorchSharp is a .NET wrapper around libTorch (PyTorch's C++ backend), and TensorFlow.NET is a .NET binding to TensorFlow's C API. This means a .NET developer can train and inference neural networks using the familiar ML.NET API, and it will actually leverage PyTorch or TensorFlow engines internally. ML.NET also makes it easy to consume ONNX models (via the `ApplyOnnxModel` transform) [32]. In short, ML.NET provides a high-level, **C#-friendly** experience and abstracts the complexity of dealing with TensorFlow/PyTorch directly. It's well-supported on Windows (and cross-platform with .NET Core). For instance, one can use Model Builder in Visual Studio to train an image classification model (which uses TorchSharp under the hood) and then use it in a WPF application.

- **TorchSharp** – This is worth highlighting itself: TorchSharp is essentially the .NET library for PyTorch, maintained under the .NET Foundation. It exposes tensor operations, autograd, and neural network modules in C#. Microsoft uses it to enable deep learning in ML.NET [46] [45]. A developer can use TorchSharp directly for full control (writing training loops in C# similar to how one would in Python with PyTorch). TorchSharp supports CPU and GPU (CUDA) on Windows, since it leverages libTorch. The experience is lower-level than ML.NET but more flexible. It's actively maintained alongside PyTorch releases.

- **TensorFlow.NET** – A third-party open-source .NET binding for TensorFlow (by SciSharp). This allows constructing and training TensorFlow graphs in C# (or loading existing models). It's used

less frequently now (since most .NET folks use ML.NET or ONNX Runtime), but it exists for scenarios where TensorFlow might be needed in a .NET context.

- **DirectML and WinML APIs** – For C++ developers writing Windows applications (especially UWP or WinUI apps), the Windows AI APIs allow using the system's ML capabilities. For example, a developer can load an ONNX file using `LearningModel.LoadFromFilePath()` (WinRT API) and bind inputs/outputs to evaluate it. This is essentially calling into ONNX Runtime + DirectML (if a GPU is available) [25]. The advantage is simplicity and the fact it's supported by the OS with no additional dependencies. The disadvantage is less flexibility (one can't easily choose which EP to use beyond what the OS picks, and for new ops one might need to wait for OS updates). Nonetheless, for many Windows apps that just need to run a standard model (like a RESNET50 or a BERT classifier in ONNX format), WinML is a convenient solution.

- **Visual Studio Tools** – Microsoft has been integrating AI into their developer tools. For instance, **AI Toolkit for VS Code** is an extension to streamline model development and deployment [31]. Visual Studio has templates and debugging support for machine learning scenarios. While not frameworks, these tools improve the developer experience on Windows.

- **Community and Support** – Windows AI developers benefit from both Microsoft's documentation (e.g., "*FAQ about AI in Windows apps*" which recommends ONNX Runtime and explains DirectML [26] [22]) and the broader framework communities (PyTorch forums have Windows-specific threads, etc.). Active maintenance is evident: PyTorch regularly tests Windows builds; TensorFlow does CI for Windows; ONNX Runtime is developed by Microsoft primarily on Windows as well. This means bug fixes and performance improvements land on Windows consistently.

Conclusion

Windows is a capable platform for AI development across the full stack – from prototyping models in Python to deploying high-performance inference in C# applications. High-level libraries like Hugging Face Transformers run smoothly on Windows (leveraging PyTorch/TF), core frameworks like PyTorch and TensorFlow have native Windows GPU support (for NVIDIA, with AMD GPU support catching up via DirectML/ROCm), and inference engines like ONNX Runtime are optimized for Windows and even baked into the OS [25].

When working on Windows, developers should be mindful of hardware compatibility: NVIDIA GPUs offer the most seamless experience via CUDA, while AMD GPUs may require using DirectML or waiting for ROCm improvements. AMD and Intel CPUs are well-supported; in fact, frameworks use optimized kernels (oneDNN, etc.) to utilize advanced CPU instructions on Windows, and plugins like ZenDNN on Linux show the direction for vendor-specific boosts [47] [9].

In terms of programming languages, **Python** remains the dominant language for model development on Windows (with tools like Jupyter, VS Code, etc.), but **C++** and **C#** are first-class for deployment: C++ via libraries like libTorch, TensorRT, OpenVINO for integrating AI into native apps or game engines, and C# via ML.NET and ONNX Runtime for enterprise and desktop applications. Each framework highlighted above notes the language support – for example, ONNX Runtime's NuGet makes C# inference trivial, and ML.NET's use of TorchSharp allows training a transformer in pure C# [11].

Lastly, Windows-specific tools like **Visual Studio**, **WSL2**, and **DirectML** provide additional flexibility – Visual Studio for compiling and debugging C++ CUDA code (if you venture into custom kernels or build

frameworks from source) [37], and WSL2 for interoperability with the Linux AI world (letting you run anything not yet available on Windows). With the advent of NPUs and Microsoft's focus on AI (Windows 11's "Copilot" features, etc.), we can expect Windows to continue being a well-supported environment for AI frameworks, bridging cloud and edge scenarios.

---

[1] Using transformers at Hugging Face
https://huggingface.co/docs/hub/en/transformers

[2] Keras: The high-level API for TensorFlow | TensorFlow Core
https://www.tensorflow.org/guide/keras

[3] machine learning - Installing Pytorch for Windows 11 and AMD GPU - Stack Overflow
https://stackoverflow.com/questions/78439640/installing-pytorch-for-windows-11-and-amd-gpu

[4] [5] [6] [7] [15] [37] Get Started
https://pytorch.org/get-started/locally/

[8] [38] AMD ROCm Comes To Windows On Consumer GPUs | Tom's Hardware
https://www.tomshardware.com/news/amd-rocm-comes-to-windows-on-consumer-gpus

[9] [10] GitHub - amd/ZenDNN
https://github.com/amd/ZenDNN

[11] [32] Deep Learning overview - ML.NET | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/machine-learning/deep-learning-overview

[12] DirectML Plugin for TensorFlow 2 | Microsoft Learn
https://learn.microsoft.com/en-us/windows/ai/directml/gpu-tensorflow-plugin

[13] [14] [27] [47] Enabling Optimal Inference Performance on AMD EPYC™ Processors with the ZenDNN Library — The TensorFlow Blog
https://blog.tensorflow.org/2023/03/enabling-optimal-inference-performance-on-amd-epyc-processors-with-the-zendnn-library.html

[16] Multi-GPU and Multi-Node Training — Isaac Lab Documentation
https://isaac-sim.github.io/IsaacLab/main/source/features/multi_gpu.html

[17] Batch sizes / 2 GPUs + Windows 10 = 1 GPU? - Hugging Face Forums
https://discuss.huggingface.co/t/batch-sizes-2-gpus-windows-10-1-gpu/9349

[18] DeepSpeed is a deep learning optimization library that ... - GitHub
https://github.com/deepspeedai/DeepSpeed

[19] Horovod Installation Guide
https://horovod.readthedocs.io/en/latest/install_include.html

[20] [22] [26] [29] [30] [31] FAQs about using AI in Windows apps | Microsoft Learn
https://learn.microsoft.com/en-us/windows/ai/faq

[21] Install ONNX Runtime | onnxruntime
https://onnxruntime.ai/docs/install/

[23] [24] [25] [28] Windows | onnxruntime
https://onnxruntime.ai/docs/get-started/with-windows.html

[33] The Benefits of Accelerating CPU Inference with PyTorch Inductor
https://www.intel.com/content/www/us/en/developer/articles/technical/accelerated-cpu-inference-with-pytorch-inductor.html

34 Accelerating Inference on x86-64 Machines with oneDNN Graph
https://pytorch.org/blog/accelerating-inference/

35 36 Intel GPU Support Now Available in PyTorch 2.5
https://pytorch.org/blog/intel-gpu-support-pytorch-2-5/

39 40 About ROCm 5.6.0 on Windows - windows - PyTorch Forums
https://discuss.pytorch.org/t/about-rocm-5-6-0-on-windows/177490

41 Intel's “cripple AMD” function (2019) | Hacker News
https://news.ycombinator.com/item?id=24307596

42 AMD Optimizing CPU Libraries (AOCL)
https://www.amd.com/en/developer/aocl.html

43 AMD Optimizing CPU Libraries (AOCL) - HECC Knowledge Base
https://www.nas.nasa.gov/hecc/support/kb/amd-optimizing-cpu-libraries-(aocl)_690.html

44 45 46 ML.NET 3.0 Boosts Deep Learning, Data Processing for .NET-Based AI Apps -- Visual Studio Magazine
https://visualstudiomagazine.com/articles/2023/11/28/ml-net-3-0.aspx