

DXNN Compiler (DX-COM) User Manual

2026/06/16 (v2.4.0)

DEEPX.ai

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Table of contents

1. Quick Start Guide 4
1.1 Install DX-COM 4
1.2 Compile Sample Models 4
1.3 Compile Your Own Model 4
2. DXNN SDK Overview 6
2.1 DEEPX SDK Architecture 6
3. Installation and Configuration 8
3.1 System Requirements of DX-COM 8
3.2 Installation of DX-COM 9
3.3 Source File Structure of DX-COM 12
3.4 ONNX File Configuration 13
3.5 JSON File Configuration 15
3.6 Execution of DX-COM 33
4. Supported ONNX Operators 56
4.1 Operator Support Details 56
5. Quantization Tuning Workflow 61
5.1 Quantization Tuning Workflow 61
5.2 Quantization-Aware Training (QAT) 70
5.3 Pre-Optimize API 78
5.4 Common Use Cases 86
6. Development Tools 101
6.1 Model Viewer_DX-TRON 101
6.2 Compilation Summary Report 110
7. DEEPX Agent-Driven Development for DX-COM - dx-agent-dev (Beta) 113
7.1 Introduction 113
7.2 Prerequisites 113
7.3 Architecture Overview 114
7.4 Available Agents and Skills 114
7.5 Supported AI Tools 116
7.6 User Scenarios 120
7.7 Config Auto-Inference 122
7.8 Output Isolation 123
7.9 Calibration Dataset Management 124
7.10 Sample Model Workflow 124
7.11 Session Sentinels 124

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.12 Troubleshooting 125
7.13 Mandatory Output Artifacts 126
7.14 TDD Verification Gate 127
8. Change Log 128
8.1 v2.4.1 (2026-07-27) 128
8.2 v2.4.0 (2026-06-16) 128
8.3 v2.3.1 (May 2026) 130
8.4 v2.3.0 (March 2026) 131
8.5 v2.2.1 (February 2026) 131
8.6 v2.2.0 (December 2025) 132
8.7 v2.1.0 (November 2025) 133
8.8 v2.0.0 (September 2025) 134
8.9 v1.60.1 (June 2025) 134

ⓒ Copyright 2026 DEEPX All Rights Reserved.

1. Quick Start Guide

Welcome to **DX-COM** (DEEPX Compiler)! This guide will help you compile your first ONNX model in just a few minutes.

1.1 Install DX-COM

Follow Installation of DX-COM to install and verify DX-COM before continuing with this quick start.

1.2 Compile Sample Models

When DX-COM is installed from the package bundle, sample models and calibration data are automatically downloaded to `dx_com/` during `install.sh`. If they are not available, run the helper scripts below first:

```
./example/1-download_sample_models.sh
./example/2-download_sample_calibration_dataset.sh
```

These helper scripts populate the sample asset directories used by the bundle, such as `dx_com/sample_models/` and `dx_com/calibration_dataset/`.

Then run all sample compilations at once:

```
./example/3-compile_sample_models.sh
```

1.3 Compile Your Own Model

1.3.1 With the `dxcom` Command

For complete examples and options, see the **CLI Execution** section in Execution of DX-COM.

```
dxcom -m model.onnx -c config.json -o output/
```


© Copyright 2026 DEEPX All Rights Reserved.

1.3.2 With the `dx_com` Python Module

For complete examples and options, see the **Python Wheel Package Usage** section in Execution of DX-COM.

Using `dx_com.compile()`:

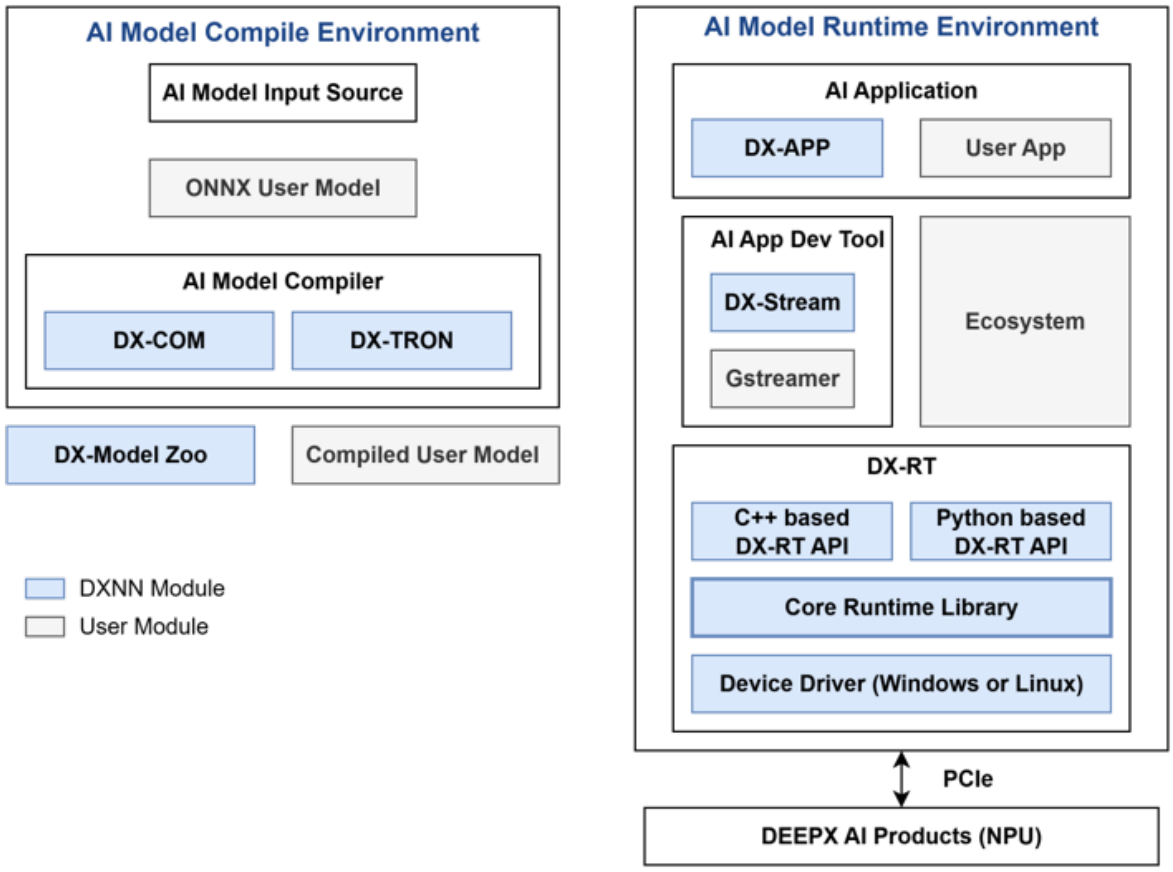
```
import dx_com
dx_com.compile(model="model.onnx", output_dir="output/", config="config.json")
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

2. DXNN SDK Overview

This chapter provides an overview of the DEEPX SDK architecture and explains each core component and its role in the AI development workflow.

2.1 DEEPX SDK Architecture



DEEPX SDK is an all-in-one software development platform that streamlines the process of compiling, optimizing, simulating, and deploying AI inference applications on DEEPX NPUs (Neural Processing Units). It provides a complete toolchain, from AI model creation to runtime deployment, optimized for edge and embedded systems, enabling developers to build high-performance AI applications with minimal effort.

DX-COM is the compiler in the DEEPX SDK that converts a pre-trained ONNX model and its associated configuration JSON file into a hardware-optimized .dxnn binary for DEEPX NPUs. The ONNX file contains the model structure and weights, while the JSON file defines pre/post-processing settings and

© Copyright 2026 DEEPX All Rights Reserved.

compilation parameters. DX-COM provides a fully compiled .dxnn file, optimized for low-latency and high-efficiency inference on DEEPX NPU.

DX-RT is the runtime software responsible for executing .dxnn models on DEEPX NPU hardware. DX-RT directly interacts with the DEEPX NPU through firmware and device drivers, using PCIe interface for high-speed data transfer between the host and the NPU, and provides C/C++ and Python APIs for application-level inference control. DX-RT offers a complete runtime environment, including model loading, I/O buffer management, inference execution, and real-time hardware monitoring.

DX ModelZoo is a curated collection of pre-trained neural network models optimized for DEEPX NPU, designed to simplify AI development for DEEPX users. It includes pre-trained ONNX models, configuration JSON files, and pre-compiled DXNN binaries, allowing developers to rapidly test and deploy applications. DX ModelZoo also provides benchmark tools for comparing the performance of quantized INT8 models on DEEPX NPU with full-precision FP32 models on CPU or GPU.

DX-Stream is a custom GStreamer plugin that enables real-time streaming data integration into AI inference applications on DEEPX NPU. It provides a modular pipeline framework with configurable elements for preprocessing, inference, and postprocessing, tailored to vision AI work. DX-Stream allows developers to build flexible, high-performance applications for use cases such as video analytics, smart cameras, and edge AI systems.

DX-APP is a sample application that demonstrates how to run compiled models on actual DEEPX NPU using DX-RT. It includes ready-to-use code for common vision tasks such as object detection, face recognition, and image classification. DX-APP helps developers quickly set up the runtime environment and serves as a template for building and customizing their own AI applications.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3. Installation and Configuration

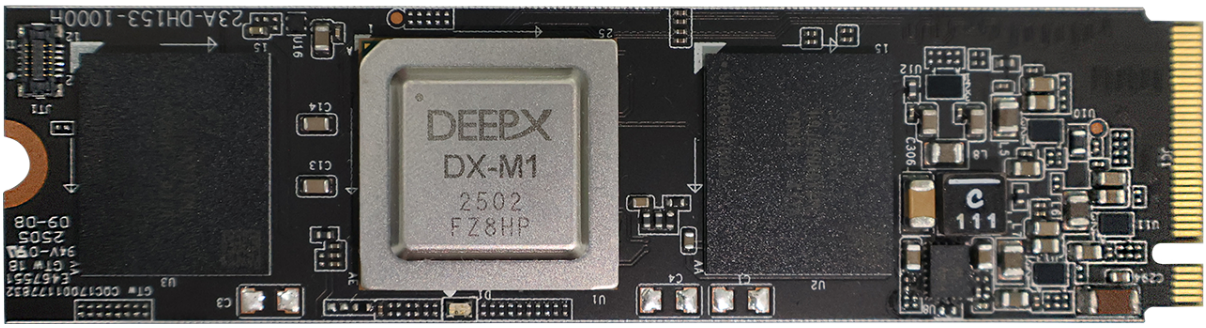
3.1 System Requirements of DX-COM

This section describes the hardware and software requirements for running **DX-COM**.

Hardware and Software Requirements

- **CPU:** amd64(x86_64)
  NOTE. aarch64(arm64) is **NOT** supported.
- **RAM:** ≥ 16 GB
- **Storage:** ≥ 8 GB available disk space
- **OS:** Ubuntu 20.04 / 22.04 / 24.04 / 26.04 (x86_64, primary supported environments; Ubuntu 26.04 validated in DX-COM v2.4.0). Fedora 42-45, Red Hat Enterprise Linux 9-10, and CentOS Stream 9-10 have been **validated since DX-COM v2.3.0** and continue to be supported in v2.4.0.
  NOTE. Ubuntu 18.04 OS is **NOT** supported.
- **glibc:** ≥ 2.31 (matches the `manylinux_2_31_x86_64` wheel platform tag)

> **NOTE**
>
> To check your glibc version, run `ldd --version` in the terminal.



© Copyright 2026 DEEPX All Rights Reserved.

3.2 Installation of DX-COM

This section provides instructions for installing **DX-COM** on supported Linux distributions and using it through both the `dxcom` command-line interface and the `dx_com` Python module.

> **Distribution Update**
>
> The standalone executable distribution is deprecated. This manual describes the wheel-based workflow only.

3.2.1 Pre-Installation Requirements

Before installing **DX-COM**, ensure the following libraries are installed.

- OpenGL runtime support for graphical operations
  - Debian/Ubuntu: `libgl1-mesa-glx`
  - Fedora/RHEL/CentOS: `mesa-libGL`
- Core utility library used by many GNOME and GTK applications
  - Debian/Ubuntu: `libglib2.0-0`
  - Fedora/RHEL/CentOS: `glib2`
- GNU `make`

Run the command that matches your distribution to install the required libraries.

Debian / Ubuntu

```
sudo apt-get install -y --no-install-recommends libgl1-mesa-glx libglib2.0-0 make
```

Fedora

```
sudo dnf install -y mesa-libGL glib2 make
```

RHEL / CentOS (9, 10)

© Copyright 2026 DEEPX All Rights Reserved.

`mesa-libGL` lives in the CodeReady Builder (CRB) repository on RHEL/CentOS, so enable it first:

```
sudo dnf install -y dnf-plugins-core
sudo dnf config-manager --set-enabled crb        # RHEL/CentOS 9, 10
sudo dnf install -y mesa-libGL glib2 make
```

3.2.2 Installation

Supported Environments

Python Version
Python 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14

Install the Wheel

DX-COM wheels are built with `auditwheel` and tagged with the `manylinux_2_31_x86_64` platform tag, which means they require **glibc ≥ 2.31** on the host (Ubuntu 20.04+, Debian 11+, RHEL/CentOS 9+, Fedora 42+).

Option 1: Install from PyPI (Recommended)

Install the latest version directly from PyPI:

```
pip install dx-com
```

Option 2: Install from Downloaded Wheel

Download the wheel file matching your Python version and install it using pip:

```
pip install dx_com-<VERSION>-<PYTAG>-<ABITAG>-manylinux_2_31_x86_64.whl
```

For example, for Python 3.11:

```
pip install dx_com-<VERSION>-cp311-cp311-manylinux_2_31_x86_64.whl
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Wheel filename format

The wheel filename follows PEP 425 / PEP 600 conventions: `dx_com-<VERSION>-<PYTAG>-<ABITAG>-<PLATFORMTAG>.whl`. For DX-COM v2.4.0 on CPython 3.12, this looks like `dx_com-2.4.0-cp312-cp312-manylinux_2_31_x86_64.whl`.

Verify the Installation

```
dxcom --version
python3 -c "import dx_com; print(dx_com.__version__)"
```

For detailed information on command-line usage, refer to the **CLI Execution** section in Execution of DX-COM. For the `dx_com` Python module, including the `compile()` function signature, parameters, and examples, refer to the **Python Wheel Package Usage** section in Execution of DX-COM.

© Copyright 2026 DEEPX All Rights Reserved.

3.3 Source File Structure of DX-COM

After running `install.sh`, the prepared `dx_com/` directory is organized as follows.

```
dx_com/
├── dx_com-*.whl              # DX-COM wheel package
├── sample_models/            # Sample model assets
│  ├── onnx/                 # Sample ONNX models
│  └── json/                 # Sample JSON files
└── calibration_dataset/      # Sample calibration dataset
```

dx_com/
This directory stores the DX-COM wheel file and the sample assets prepared during `install.sh`.

dx_com-*.whl
This Python wheel package contains the `dxcom` command-line interface and the `dx_com` Python module. Install it using `pip install dx_com-*.whl`. Refer to Installation of DX-COM for installation instructions.

sample_models
This directory contains the downloaded sample ONNX models and JSON configuration files used by the sample compilation workflow.

- **ONNX Files** (`dx_com/sample_models/onnx/`): Sample ONNX models used as compiler inputs
- **JSON Files** (`dx_com/sample_models/json/`): Sample configuration files used with the sample models. Refer to **JSON File Configuration**.

© Copyright 2026 DEEPX All Rights Reserved.

3.4 ONNX File Configuration

This section describes how to convert a PyTorch model to the ONNX format using the `torch.onnx.export()` function.

PyTorch to ONNX Conversion Example

You can export a PyTorch model to ONNX format as follows.

Example

```
import torch
import torch.nn as nn

# 1. Define or load the PyTorch model
class SimpleModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear = nn.Linear(10, 5)    # Input 10, Output 5

  def forward(self, x):
    return self.linear(x)

model = SimpleModel()
model.eval()                # Set to inference mode (Affect Dropout, BatchNorm,
etc.)

# 2. Create a dummy input tensor with the same shape and type as the model input
# This is used to trace the model's computational graph, not for the actual
inference
batch_size = 1              # batch size must be 1
dummy_input = torch.randn(batch_size, 10)  # Create input matching the model's input
shape (Batch, Features)

# 3. Export the model to ONNX format
onnx_file_path = "simple_model.onnx"

torch.onnx.export(
  model,                  # PyTorch model object to export
  dummy_input,            # Dummy input used for tracing (tuple is possible)
  onnx_file_path,         # Output ONNX file path
  export_params=True,     # If True, saves model parameter (weight) into the ONNX
file
  opset_version=11,       # ONNX opset version (11~21 supported)
  input_names=['input'],  # Name of the ONNX model input tensor
  output_names=['output']  # Name of the ONNX model output tensor
)
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Key Parameter of `torch.onnx.export()`

- `model` : PyTorch model object to export
- `dummy_input` : Input values to model's `forward()` method
- `onnx_file_path` : Output ONNX file path
- `export_params` : If True, includes weights in the ONNX file
- `opset_version` : ONNX opset version (11~21 supported)
- `input_names` : Name of the input tensor(s)
- `output_names` : Name of the output tensor(s)

NOTE

- `model.eval()` : Set the model to "eval()" mode before exporting.
- `batch size` : Batch size **must** be 1.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3.5 JSON File Configuration

This chapter describes various parameters required for compiling an ONNX model using the **DX_COM**. It includes input specifications, calibration methods, data preprocessing settings, and optional parameters for advanced compilation schemes.

These parameters are defined in a JSON file, which serves as a blueprint for how the compiler interprets and processes the input model.

3.5.1 Parameter Quick Reference

Parameter	Required	Description
`inputs`	Yes	Model input name and shape: `{"name": [1,C,H,W]}`
`calibration_method`	Yes	Calibration method: `"ema"` or `"minmax"`
`calibration_num`	Yes	Number of calibration samples (e.g., 100)
`default_loader`	Yes	Preprocessing configuration with dataset loading
`quantization_device`	No	Target device for quantization computation. GPU auto-selected if available
`enhanced_scheme`	No	DXQ-P0~P5 quantization enhancement for accuracy improvement (DX-COM v2.1.0+)
`qmaster`	No	Quantization-Aware Training (QAT) hyperparameters. Presence of this block enables QAT (DX-COM v2.4.0+). See Quantization-Aware Training (QAT)
`ppu`	No	Object detection post-processing (YOLO models)

© Copyright 2026 DEEPX All Rights Reserved.

For complete examples, see Common Use Cases.

3.5.2 Required Parameters

Inputs

Defines the input name and shape of the ONNX model.

> **Model Input Restrictions**
>
> - The batch size **must** be fixed to 1.
> - **Only** a single input is supported when compiling from a JSON configuration with the `dxcom` command. For multi-input models, use the `dx_com` Python module with DataLoader (see the **Python Wheel Package Usage** section in Execution of DX-COM).
> - Input name **must** exactly match ONNX model definition.

Example

```
{
  "inputs": {
    "input.1": [1, 3, 512, 512]
  }
}
```

In this example, `"input.1"` is the name of the input tensor and its shape is `[1, 3, 512, 512]`, where:

- `1` : batch size (**must be 1**)
- `3` : number of channels (e.g., RGB)
- `512 x 512` : image height and width

Calibration Method

Defines the calibration method used during quantization. It is essential for maintaining model accuracy after compilation by determining appropriate activation ranges.

© Copyright 2026 DEEPX All Rights Reserved.

Available Methods

- `ema`:
  Uses exponential moving average of activation values.
  Recommended for improved post-quantization accuracy.
- `minmax`:
  Uses the minimum and maximum activation values to determine quantization range.

Example

```
{
  "calibration_method": "ema"
}
```

Calibration Number

Defines the number of steps used during calibration. A higher number may improve quantization accuracy by better estimating activation ranges, but may also increase compile time.

To minimize the accuracy degradation, it is recommended to try different values, such as 1, 5, 10, 100, or 1000, and determine the value that yields the best accuracy for your model.

Example

```
{
  "calibration_num": 100
}
```

Calibration Data Loading

You must provide calibration data using the `default_loader` parameter, which specifies the dataset location, accepted file types, and preprocessing steps:

```
{
  "default_loader": {
    "dataset_path": "/datasets/ILSVRC2012",
    "file_extensions": ["jpeg", "png", "jpg", "JPEG"],
    "preprocessings": [
      {"resize": {"width": 224, "height": 224}},
      {"normalize": {"mean": [0.485, 0.456, 0.406], "std": [0.229, 0.224, 0.225]}}
    ]
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    }
}
```

For all available preprocessing operations, see Appendix: Preprocessing Operations Reference.

For Multi-Input Models or Non-Image Data

If you need to compile multi-input models or provide non-image data, use the `dx_com` Python module with torch DataLoader instead of JSON configuration. See the **Python Wheel Package Usage** section in Execution of DX-COM.

3.5.3 Optional Parameters: Quantization Device

Specifies the device used for quantization during compilation. By default, DX-COM automatically detects available hardware and uses GPU if a CUDA-compatible GPU is available, otherwise falls back to CPU. You can override this behavior by explicitly setting the device.

Available Values

- Not specified (Default): Auto-detect. Uses GPU if available, otherwise CPU.
- `"cpu"`: Force CPU quantization.
- `"cuda"`: Use the default CUDA GPU.
- `"cuda:N"`: Use a specific GPU (e.g., `"cuda:0"`, `"cuda:1"`).

Example

```
{
  "quantization_device": "cuda:0"
}
```

> ⚠ **Hardware Requirements**
>
> - **GPU**: NVIDIA GPU with CUDA support.
> - **Framework**: PyTorch built with CUDA support (`torch.cuda.is_available()` must return `True`).

For practical examples, see the **Use Case 5: Enhanced Quantization (DXQ)** section in Common Use Cases.

© Copyright 2026 DEEPX All Rights Reserved.

3.5.4 Optional Parameters: Enhanced Quantization Scheme (DXQ)

Version Support

DXQ (`enhanced_scheme`) is supported in **DX-COM v2.1.0 and later**.

When quantizing a model, accuracy degradation may occur compared to the original model. To mitigate this, **Q-PRO options** (DXQ-P0 to DXQ-P5) can be used to enhance quantization performance.

Prefer automatic selection?

The DXQ schemes below are the **manual** Q-PRO interface. To let DX-COM pick the DXQ combination for you, use the compile-time `--use_q_pro` flag instead — see the **Automatic Q-PRO (`use_q_pro`)** section in Execution of DX-COM. Automatic Q-PRO and a manual `enhanced_scheme` are mutually exclusive.

Name	Compilation Speed	Accuracy Improvement
DXQ-P0	Very Fast	Low
DXQ-P1	Very Slow	Low-Middle
DXQ-P2	Quite Slow	Middle-High
DXQ-P3	Very Slow	High
DXQ-P4	Very Slow	High
DXQ-P5	Very Slow	High

Recommendation

Best Accuracy: DXQ-P3 and DXQ-P4 generally offer better accuracy, so try them first.

GPU Acceleration: DXQ schemes (especially P1~P5) are computationally intensive. Using `quantization_device="cuda"` can reduce compilation time by **2-5x** compared to CPU. See the **Use Case 5: Enhanced Quantization (DXQ)** section in Common Use Cases for examples.

© Copyright 2026 DEEPX All Rights Reserved.

> **Limitations**
>
> Results are **not guaranteed** to improve accuracy. Results may vary depending on the model and dataset.

DXQ-P0

Uses an alpha value to control the strength of the quantization enhancement.

```
{
  "enhanced_scheme": {
    "DXQ-P0": {
      "alpha": 0.5
    }
  }
}
```

DXQ-P1

Enables a simple quantization improvement strategy.

```
{
  "enhanced_scheme": {
    "DXQ-P1": true
  }
}
```

DXQ-P2

Uses a more refined strategy with `alpha`, `beta`, and `cosim_num` parameters.

```
{
  "enhanced_scheme": {
    "DXQ-P2": {
      "alpha": 0.1,
      "beta": 1.0,
      "cosim_num": 2
    }
  }
}
```

DXQ-P3, DXQ-P4, DXQ-P5

Support the optional `num_samples` parameter. Recommended value: 1024.

```
{
  "enhanced_scheme": {
    "DXQ-P3": {
      "num_samples": 1024
    }
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    }
}
```

3.5.5 Optional Parameters: PPU Configuration

The Post-Processing Unit (PPU) is a hardware module inside the NPU for accelerating selected post-processing steps in supported object detection models. The `ppu` configuration section exists to enable this hardware path and provide the parameters required to use it.

In this section, `type = 0` and `type = 1` still configure PPU hardware paths. `type = 2` is different: it does **not** use the PPU hardware. Instead, it enables an additional CPU-side TopK-based optimization flow for compatible DFL-based YOLO models. Note that `ppu.type = 2` is deprecated in v2.4.0 — see the Type 2 section below.

These configurations can automate or optimize the following operations:

- **Confidence Filtering (`type = 0`, `type = 1`)**: Removal of detection candidates below a specified threshold through the PPU hardware path.
- **Class Prediction (`type = 0`, `type = 1`)**: Execution of Argmax-style class selection per detection through the PPU hardware path.
- **TopK Candidate Reduction (`type = 2`)**: CPU-side selection of the highest-scoring candidates to reduce the work required in later CPU-side decoding and filtering stages for supported DFL-based YOLO models.

> ⚠ **IMPORTANT**
>
> **NMS** (Non-Maximum Suppression) is not supported by the PPU hardware and must still be executed on the host CPU after compilation using the filtered model outputs.

Operational Guidelines

Before using the `ppu` configuration, verify that your model architecture and system requirements align with the following criteria:

Recommended Use Cases (Enable)

- **Model Compatibility:** Optimized for the YOLO family and similar object detection architectures.
- **Hardware Constraints:** Highly recommended for edge devices with limited CPU overhead.
- **Performance Goals:** Necessary for real-time applications requiring high FPS (Frames Per Second).

© Copyright 2026 DEEPX All Rights Reserved.

Non-Supported Scenarios (Disable)

- **Classification Models:** (e.g., ResNet, MobileNet) These models do not require the filtering logic provided by the PPU.
- **Segmentation Models:** (e.g., DeepLab, U-Net) These rely on pixel-level mask processing, which is incompatible with PPU logic.
- **Custom Architectures:** Any model utilizing a detection head that does not align with standard YOLO-style output tensors.

Performance Impact

The performance effect depends on the selected PPU type. `type = 0` and `type = 1` reduce CPU-side detection work through the PPU hardware path, while `type = 2` improves CPU-side post-processing efficiency by shrinking the candidate set before the heavier CPU-side DFL-based decoding and later filtering stages.

PPU Mode	Primary Effect	Expected Benefit
Disabled	All post-processing remains on the host CPU	Higher CPU workload
`type = 0` / `type = 1`	The compiled model uses the PPU path for filtering and class prediction	Lower CPU utilization and lower end-to-end latency
`type = 2`	TopK and the later decoding/ filtering stages remain on the host CPU; TopK reduces the number of candidates processed later	Reduced CPU post-processing complexity and improved runtime efficiency

---

Configuration Parameters

Type	Architecture	Supported Models
0	Anchor-Based	YOLOv3, YOLOv4, YOLOv5, YOLOv7
1	Anchor-Free	YOLOX, YOLOv8, YOLOv9, YOLOv10, YOLOv11, YOLOv12
2	DFL-Based Anchor-Free	

© Copyright 2026 DEEPX All Rights Reserved.

Type	Architecture	Supported Models
		YOLOv8, YOLOv9, YOLOv11, YOLOv12 (optional CPU-side TopK optimization path; **deprecated in v2.4.0** — use `dx_com.pre_optimize()` instead)

Type 0: Anchor-Based YOLO Models

```
{
  "ppu": {
    "type": 0,
    "conf_thres": 0.25,
    "activation": "Sigmoid",
    "num_classes": 80,
    "layer": {
      "Conv_245": {"num_anchors": 3},
      "Conv_294": {"num_anchors": 3},
      "Conv_343": {"num_anchors": 3}
    }
  }
}
```

Parameters:

- `type`: Set to `0` for anchor-based models
- `conf_thres`: Confidence threshold (fixed at compile time)
- `num_classes`: Number of detection classes
- `activation`: Typically `"Sigmoid"`
- `layer`: Dictionary mapping Conv node names to anchor configurations

Type 1: Anchor-Free Models

YOLOX Example:

```
{
  "ppu": {
    "type": 1,
    "conf_thres": 0.25,
    "num_classes": 80,
    "layer": [
```


© Copyright 2026 DEEPX All Rights Reserved.

```
      {"bbox": "output_bbox_1", "obj_conf": "output_obj_1", "cls_conf": "output_cls_1
"},
      {"bbox": "output_bbox_2", "obj_conf": "output_obj_2", "cls_conf": "output_cls_2
"},
      {"bbox": "output_bbox_3", "obj_conf": "output_obj_3", "cls_conf": "output_cls_3
"}
    ]
  }
}
```

YOLOv8/v9/v10/v11/v12 Example:

```
{
  "ppu": {
    "type": 1,
    "conf_thres": 0.25,
    "num_classes": 80,
    "layer": [
      {"bbox": "Mul_441", "cls_conf": "Sigmoid_442"}
    ]
  }
}
```

Parameters:

- `type`: Set to `1` for the standard anchor-free PPU path
- `conf_thres`: Confidence threshold (fixed at compile time)
- `num_classes`: Number of detection classes
- `layer`: List of layer configurations
  - `bbox`: Bounding box output layer
  - `obj_conf`: Object confidence layer
  - `cls_conf`: Class confidence layer

© Copyright 2026 DEEPX All Rights Reserved.

Type 2: DFL-Based Anchor-Free Models (CPU-Side TopK Optimization)

> ⚠ **Deprecated in v2.4.0**
>
> `ppu.type = 2` is **deprecated** as of DX-Compiler v2.4.0 and will be removed in a future release. New projects should use the `dx_com.pre_optimize()` API, which provides the same TopK-first optimization (and adds support for instance segmentation and YOLO26). Compiling a model with PPU type 2 still works in v2.4.0 but emits a deprecation warning. See the **Migration from PPU Type 2** section in Pre-Optimize API for the replacement recipe.

`ppu.type = 2` is an optional CPU-side TopK optimization path for compatible DFL-based anchor-free YOLO models. It does **not** use the PPU hardware. Post-processing still runs on the CPU, and TopK is also performed on the CPU. By reducing the number of candidate boxes that continue into later CPU-side DFL-based decoding and filtering stages, this option reduces post-processing complexity and improves runtime efficiency.

This option is intended for DFL-based YOLO families that expose separate per-scale `bbox` and `cls_conf` head nodes. It is an optional CPU-side optimization path in addition to the standard anchor-free PPU configuration. In internal validation for DX-COM v2.3.0, this path was compile-verified with YOLOv8, YOLOv9, YOLOv11, and YOLOv12 variants using compatible head mappings.

```
{
  "ppu": {
    "type": 2,
    "topk": 512,
    "num_classes": 80,
    "layer": [
      {"bbox": "bbox_head_p3", "cls_conf": "cls_head_p3"},
      {"bbox": "bbox_head_p4", "cls_conf": "cls_head_p4"},
      {"bbox": "bbox_head_p5", "cls_conf": "cls_head_p5"}
    ]
  }
}
```


© Copyright 2026 DEEPX All Rights Reserved.

Parameters:

- `type` : Set to `2` to compile the CPU-side TopK-optimized flow
- `topk` : Number of candidates kept before later decoding stages
  - If omitted, DX-COM uses the internal default value
  - `512` is a practical starting point for validated YOLO examples
- `num_classes` : Number of detection classes
- `layer` : List of per-scale detection head mappings
  - `bbox` : Final bbox head node name for a detection scale
  - `cls_conf` : Final class-confidence head node name for the same scale

Use this option only when the model head structure is compatible with this workflow. A DFL-based output is important, but it is not the only requirement. You must also provide correct per-scale `bbox` and `cls_conf` node mappings in the JSON configuration.

Implementation: Identifying ONNX Node Names

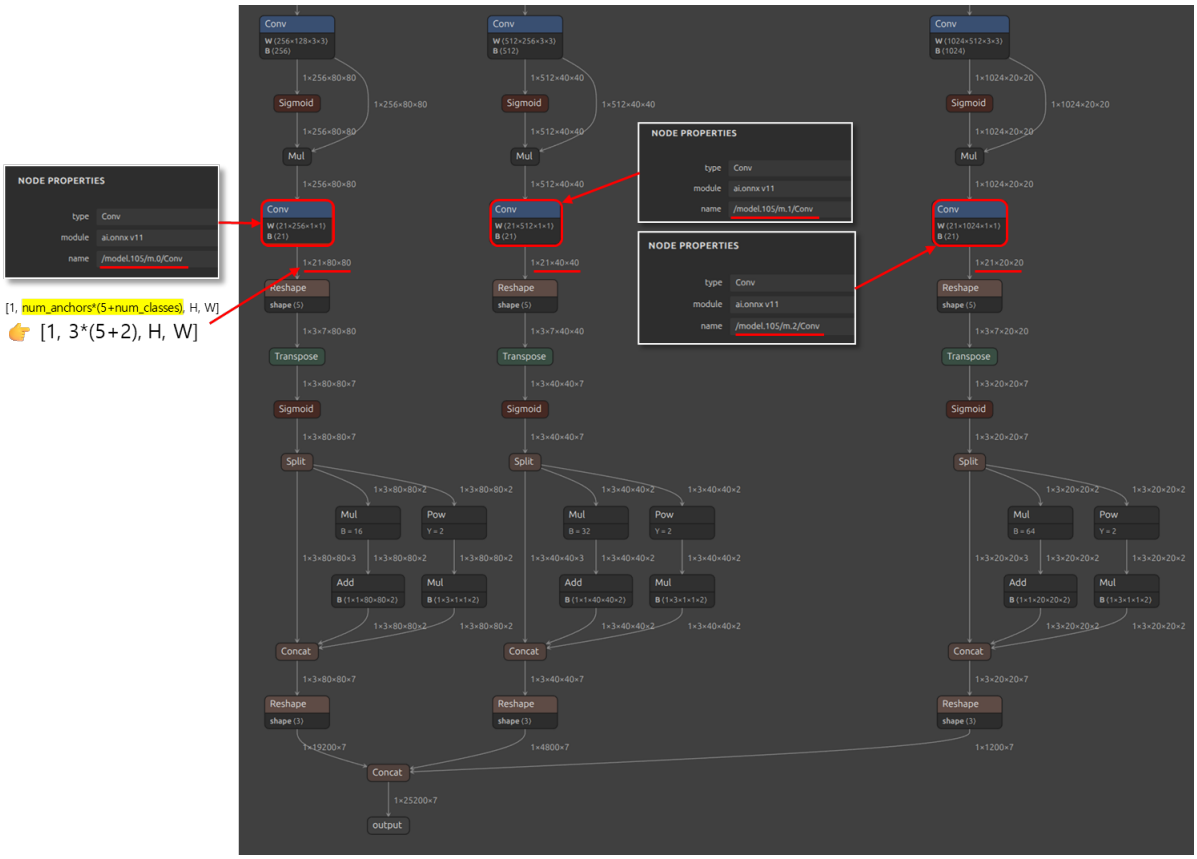
To configure PPU, you need to identify specific node names in your ONNX model:

Step 1. Open your model in Netron to visualize the ONNX graph

Step 2-1. For Type 0 (Anchor-based YOLO):

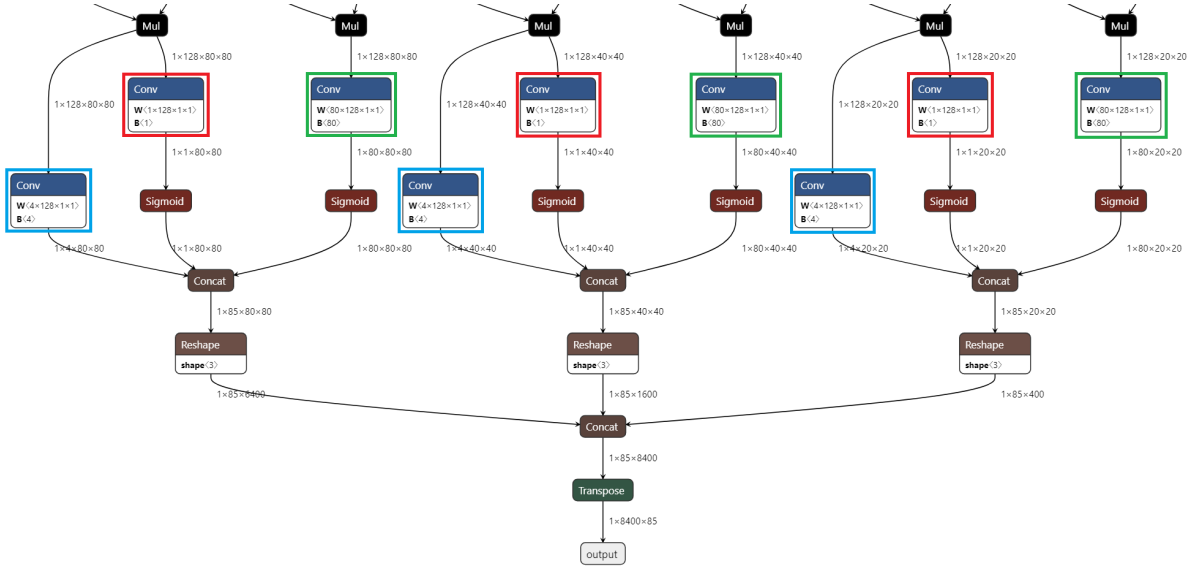
- Trace backwards from model outputs to locate detection head Conv layers
- These Conv layers output feature maps with shape `[1, num_anchors*(5+num_classes), H, W]`
- Example node names in YOLOv7: `/model.105/m.0/Conv`, `/model.105/m.1/Conv`, `/model.105/m.2/Conv`

ⓒ Copyright 2026 DEEPX All Rights Reserved.



Step 2-2. For Type 1 (standard anchor-free PPU path, YOLOX example):

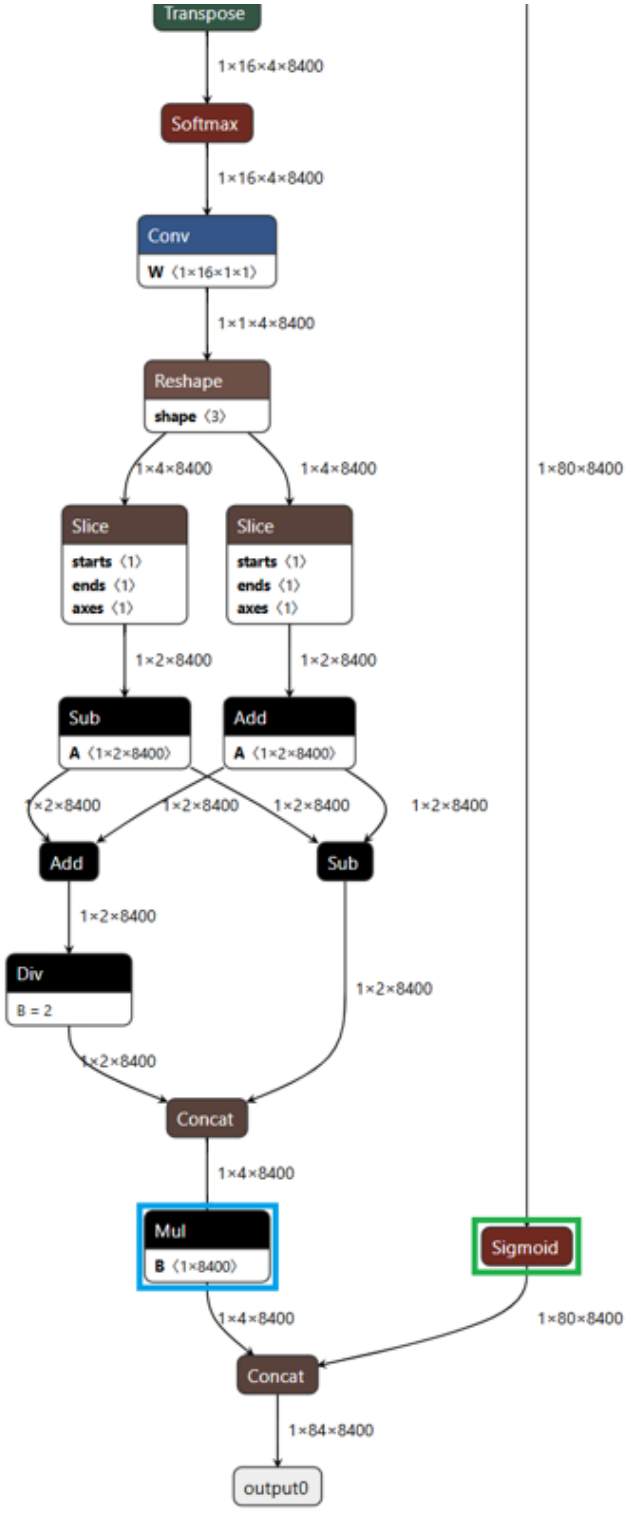
- Find three types of Conv nodes for each detection scale: `bbox`, `obj_conf`, `cls_conf`
- This results in 9 Conv layers total (3 scales × 3 branches)



© Copyright 2026 DEEPX All Rights Reserved.

Step 2-3. For Type 1 (YOLOv8 and later):

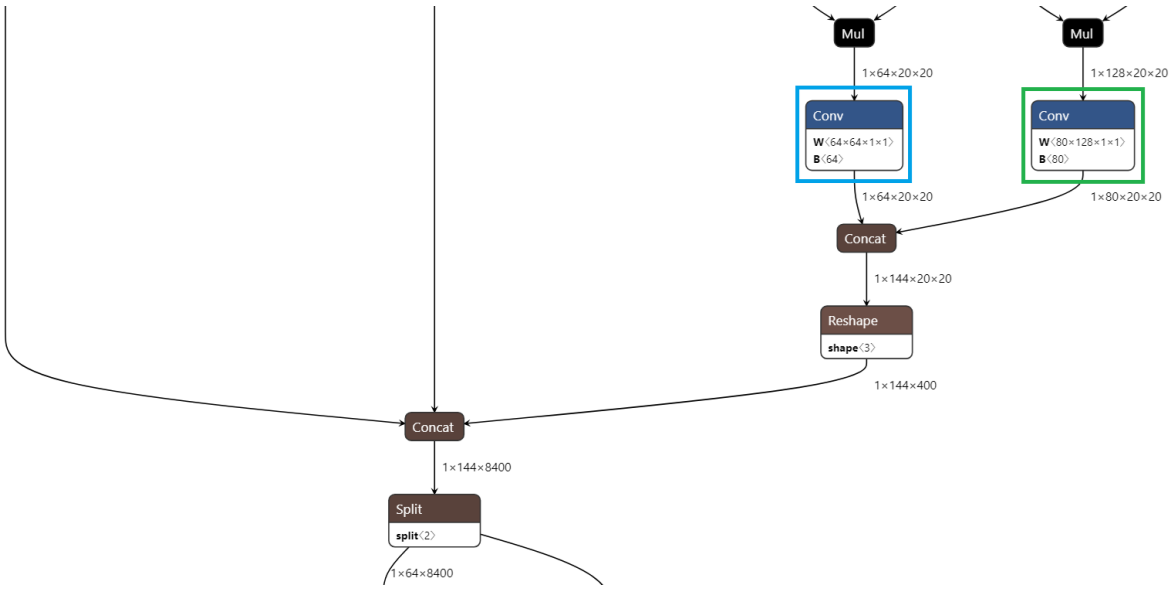
- Trace backwards from the final `Concat` operation
- Identify `bbox` (often from `Mul` or `Add`) and `cls_conf` (often from `Sigmoid`)
- Use the node names that match the `layer` format required by `type = 1`



© Copyright 2026 DEEPX All Rights Reserved.

Step 2-4. For Type 2 (CPU-side TopK optimization path):

- Inspect the ONNX tail and locate the final per-scale bbox and class head nodes
- Identify three detection scales and pair the matching `bbox` and `cls_conf` nodes for each scale
- For validated examples, the correct nodes were the final Conv nodes in the decoupled head branches
- Some models may already include TopK-style operations in the ONNX tail, but you should still map the compatible per-scale `bbox` / `cls_conf` nodes for the `type = 2` configuration



3.5.6 Appendix: Preprocessing Operations Reference

The following preprocessing operations can be applied to input data when using `default_loader`. These operations help standardize input formats and ensure consistency between calibration and deployment.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

⚠ Automatic Preprocessing Optimization

To maximize performance, the compiler may automatically integrate specific preprocessing operations directly into the NPU execution graph. This hardware-level integration reduces host CPU load and minimizes data transfer latency.

Optimization depends on input data types and model architecture; therefore, integration **is not guaranteed** for all operations.

How to Verify and Optimize:

After compilation, inspect the compilation logs for `[INFO] - Added nodes: messages.` This indicates which operations have been offloaded to the NPU.

1. Identify Integrated Nodes:

Check your log for entries similar to the following: `[INFO] - Added nodes: ['Normalize', 'Cast', 'Transpose']`

```
[INFO] - Added nodes: Div(x=255) -> Normalize(mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])
```

2. Update Runtime Code:

If an operation is listed in the "Added nodes" log, it is now handled by the NPU hardware. You must remove these operations from your host-side runtime code to prevent redundant processing and performance degradation.

Status in Log	Action Required	Result
Node Added	Remove from Host Code	Hardware Acceleration Enabled
Node Not Added	Keep in Host Code	Software Preprocessing (CPU)

```python
# If preprocessing is NOT integrated into NPU:
img = img / 255.0
normalized = (img - mean) / std
output = model.run(normalized)

# If preprocessing IS integrated into NPU (log shows "Added nodes"):
output = model.run(img)  # Skip preprocessing - NPU handles it automatically
```

convertColor

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Changes the color channel order of input images.

```
{"convertColor": {"form": "BGR2RGB"}}
```

Supported values: `RGB2BGR`, `BGR2RGB`, `RGB2GRAY`, `BGR2GRAY`, `RGB2YCrCb`, `BGR2YCrCb`, `RGB2YUV`, `BGR2YUV`, `RGB2HSV`, `BGR2HSV`, `RGB2LAB`, `BGR2LAB`

resize

Resizes input image to a specified target size.

```
{"resize": {"mode": "default", "width": 320, "height": 320, "interpolation":
"LINEAR"}}
```

Mode	Supported Interpolation Methods
`default` (OpenCV)	`LINEAR`, `NEAREST`, `CUBIC`, `AREA`, `LANCZOS4`
`torchvision` (PIL)	`BILINEAR`, `NEAREST`, `BICUBIC`, `LANCZOS`

centercrop

Crops the central region of the input image.

```
{"centercrop": {"width": 224, "height": 224}}
```

transpose

Rearranges tensor dimensions.

```
{"transpose": {"axis": [0, 2, 3, 1]}}
```

expandDim

Adds a new dimension at the specified axis.

```
{"expandDim": {"axis": 0}}
```

normalize

© Copyright 2026 DEEPX All Rights Reserved.

Normalizes input data by mean and standard deviation. This operation may be integrated into the NPU graph during compilation (see NPU Integration above).

```
{"normalize": {"mean": [0.485, 0.456, 0.406], "std": [0.229, 0.224, 0.225]}}
```

mul, add, subtract, div

Arithmetic operations on input data. The `subtract` and `div` operations may be integrated into the NPU graph during compilation (see NPU Integration above).

```
{"mul": {"x": 255}}
{"add": {"x": 128}}
{"subtract": {"x": 127}}
{"div": {"x": 255}}
```

3.5.7 Appendix: Custom Loader (Legacy)

> ⚠ **Deprecation Notice**
>
> The Custom Loader approach is **deprecated**. For new projects, use the **Python Wheel Package with torch DataLoader** section in Execution of DX-COM instead, which provides more flexibility and better integration with Python workflows.

For legacy projects that still require Custom Loader, download the Custom Dataloader Guide for detailed instructions.

© Copyright 2026 DEEPX All Rights Reserved.

3.6 Execution of DX-COM

This section details the entire process for executing the DXNN Compiler, which converts the prepared ONNX model (`*.onnx`) and configuration JSON file (`*.json`) into the optimized `.dxnn` output file. You can use DX-COM through both the `dxcom` command-line interface and the `dx_com` Python module.

DX-COM supports two execution methods:

- **CLI Execution**: Execute compilation using the `dxcom` command with configuration files
- **dx_com Python Module**: Programmatic compilation using the `dx_com` Python module with torch DataLoader

Choose the execution method that best fits your workflow and model requirements.

3.6.1 Execution Prerequisites and Constraints

Calibration Data Requirements
The data used for model calibration must adhere to the following specifications:

- **Default Data Type**: By default, the Calibration Data must consist of image files (e.g., JPEG, PNG).
- **Custom Data**: If the use of non-image data types is required, use the `dx_com` Python module with a custom torch DataLoader.

Multi-Input Model Support
Multi-input models are now supported through the `dx_com` Python module using torch DataLoader. For command-line execution, only single-input models are supported.

Non-Deterministic Output Notice
The compiled results may exhibit variation dependent on the underlying system environment, including CPU architecture, OS, and other specific hardware factors.

3.6.2 CLI Execution (Command-Line Interface)

The compiler can be executed via the `dxcom` command, requiring the model, configuration, and desired output directory to generate the final `.dxnn` output file.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Execution Method

Use the `dxcom` command for command-line compilation.

Basic Command

Using `dxcom`:

```
dxcom -m model.onnx -c config.json -o output/
```

What you need:

- `model.onnx` - Your pre-trained model
- `config.json` - Configuration file (see JSON File Configuration)
- `calibration_dataset/` - Folder with calibration images (referenced in config.json)

Command Format

```
dxcom -m <MODEL_PATH> -c <CONFIG_PATH> -o <OUTPUT_DIR> [OPTIONS]
```

Required Arguments

Argument	Shorthand	Description
`--model_path MODEL_PATH`	`-m`	Path to the ONNX Model file (`*.onnx`)
`--config_path CONFIG_PATH`	`-c`	Path to the Model Configuration JSON file (`*.json`)
`--output_dir OUTPUT_DIR`	`-o`	Directory to save the compiled model data

Advanced Compilation Options

The following optional arguments (`[OPTIONS]`) provide fine-grained control over the DXNN compilation process, allowing for performance tuning, resource management, and specialized debugging.

© Copyright 2026 DEEPX All Rights Reserved.

PERFORMANCE AND RESOURCE CONTROL

These options manage the balance between compilation time, NPU execution latency, and host CPU resource utilization.

Option	Value/Default	Description
`--opt_level`	`{0,1}` (Default: `1`)	Controls the model optimization level during compilation
`--aggressive_partitioning`	Flag	**(Experimental)** Enables partitioning designed to maximize operations executed on the NPU
`--float64_calibration`	Flag	Use float64 precision during calibration and offset calculations for cross-CPU determinism

Optimization Level Detail

The --opt_level option controls the optimization balance:

- `0`: Fast compilation with basic optimizations. Reduces compilation time but may result in higher NPU latency.
- `1` (Default): Full optimization for best performance. Compilation takes longer but provides optimal (lowest) NPU latency.

Aggressive Partitioning Detail (Experimental)

Enabling `--aggressive_partitioning` maximizes operations executed on the NPU. This feature is currently experimental and may produce unexpected results for some models.

- **Benefit**: This is particularly advantageous in environments with limited host CPU performance (e.g., embedded systems, edge devices), as it significantly improves overall performance by minimizing CPU workload.
- **Consideration**: In systems with powerful host CPUs, the compiler's default partitioning strategy might yield better end-to-end performance. Note that using this option may increase compilation time and memory usage.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

QUANTIZATION QUALITY AND TUNING

These options control quantization accuracy enhancement and the diagnose → re-quantize tuning loop. Available in **DX-COM v2.4.0 and later**.

Option	Value/Default	Description
`--use_q_pro`	Flag	Enable the automatic Q-PRO quantization pipeline (ONNX compile path only). Mutually exclusive with a manual `enhanced_scheme`.
`--quant_diagnosis`	Flag	Produce a per-region quantization diagnosis report (`quant_diagnosis/diagnosis_report.html`) and a reusable resume checkpoint (`quant_diagnosis/{model}.qxnn`).
`--checkpoint`	`<path>.qxnn`	Path to a `.qxnn` resume artifact. Selects **QXNN resume** mode (re-quantize without recompile). Mutually exclusive with `-m/--model_path`.
`--recalibration_method`	`{minmax,ema,iqr}`	**(Resume-only)** Observer override applied during re-calibration.
`--enhanced_scheme`	e.g. `P3:num_samples=2048`	**(Resume-only)** Manual Q-PRO scheme selection. Mutually exclusive with `--use_q_pro`.
`--dataset_path`	Path	**(Resume-only)** Override the calibration dataset path embedded in the checkpoint.

For automatic Q-PRO details see Automatic Q-PRO (`use_q_pro`) below. For the full diagnose → resume workflow, see Quantization Tuning Workflow.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Automatic Q-PRO (`use_q_pro`)

When quantization accuracy degrades, Q-PRO enhancement schemes (DXQ-P0 to DXQ-P5) can improve it. The `enhanced_scheme` JSON field exposes these schemes for **manual** selection (see the **Enhanced Quantization Scheme (DXQ)** section in JSON File Configuration). The `--use_q_pro` flag is the **automatic** alternative: DX-COM generates DXQ combinations and selects the optimal enhancement stages based on model structure and compile-time metrics — no manual tuning required.

```
# dxcom CLI
dxcom -m model.onnx -c config.json -o ./output --use_q_pro
```

```
# dx_com Python module
import dx_com
dx_com.compile(model="model.onnx", output_dir="./output", config="config.json", use_q
_pro=True)
```

- **Mutually exclusive** with a manual `enhanced_scheme` — choose one, not both.
- Can also be enabled while re-quantizing via the **QXNN Resume** section in Quantization Tuning Workflow.

> **Automatic vs Manual**
>
> Prefer `--use_q_pro` for the easiest path to higher-accuracy quantization. Drop down to a manual `enhanced_scheme` only when you need to pin a specific DXQ scheme.

DEBUGGING AND LOGGING

These options are vital for troubleshooting, logging, and targeting specific sections of the model.

Option	Shorthand	Description
`--gen_log`	N/A	When enabled, the compiler collects all compilation logs into a `compiler.log` file in the specified output directory. Useful for debugging or analyzing the compilation process
`--export_html`	N/A	Generate a self-contained HTML summary report (`<model_name>_summary.html`)

© Copyright 2026 DEEPX All Rights Reserved.

Option	Shorthand	Description
		in the output directory after compilation. See Compilation Summary Report
`--version`	`-v`	Prints the compiler module version and exits

Partial Compilation (`--compile_input_nodes` , `--compile_output_nodes`)
These advanced options allow compiling only a specific subgraph of the ONNX model by defining starting and/or ending nodes.

- `--compile_input_nodes` : Comma-separated list of node names where compilation should begin.
- `--compile_output_nodes` : Comma-separated list of node names where compilation should end (compile up to).

Use Cases: Debugging specific model sections, isolating problematic operations, and testing partial model compilation.

> ⚠ **Crucial Naming Requirement**
>
> You **must** specify the ONNX Operator Node names (the operations/boxes in visualization tools like Netron), not the tensor/edge names (the lines connecting them).

CLI Execution Examples

The following examples demonstrate common usage patterns for CLI compilation.

Basic Command

This command compiles the model using the required model path (`-m`), config file (`-c`), and output directory (`-o`).

```
dxcom \
-m sample/MobilenetV1.onnx \
-c sample/MobilenetV1.json \
-o output/mobilenetv1
```

With Log Generation

This command uses the `--gen_log` flag to collect all compilation logs into `compiler.log` in the output

© Copyright 2026 DEEPX All Rights Reserved.

directory.

```
dxcom \
-m sample/MobilenetV1.onnx \
-c sample/MobilenetV1.json \
-o output/mobilenetv1 \
--gen_log
```

With HTML Summary Report

This command adds the `--export_html` flag to also generate `<model_name>_summary.html` in the output directory.

```
dxcom \
-m sample/MobilenetV1.onnx \
-c sample/MobilenetV1.json \
-o output/mobilenetv1 \
--export_html
```

Version Information

This command prints the compiler module version and exits.

```
dxcom --version
```

With Quantization Diagnosis

This command enables `--quant_diagnosis` to produce a per-region diagnosis report and a `.qxnn` resume checkpoint under `quant_diagnosis/` in the output directory.

```
dxcom \
-m large_model.onnx \
-c config.json \
-o output/large_model \
--quant_diagnosis
```

Re-quantize from a Checkpoint (QXNN Resume)

This command re-runs quantization from a `.qxnn` checkpoint with a different calibration observer, skipping the earlier compile phases. No `-m` / `-c` is required.

```
dxcom \
--checkpoint output/large_model/quant_diagnosis/large_model.qxnn \
-o output/large_model_iqr \
--recalibration_method iqr
```

See Quantization Tuning Workflow for the full diagnose → resume loop.

© Copyright 2026 DEEPX All Rights Reserved.

Compile Sample Models (Script)
For the end-to-end sample workflow, see the **Compile Sample Models** section in Quick Start Guide. The `./example/3-compile_sample_models.sh` helper compiles `YOLOV5S-1`, `YOLOV5S_Face-1`, and `MobileNetV2-1` with `dxcom`, using assets prepared under `dx_com/`. If `dxcom` is not available in the current shell, the script first tries to activate the DX-COM virtual environment.

3.6.3 Python Wheel Package Usage

The Python wheel package also provides a programmatic interface for model compilation directly from Python code. This approach is particularly useful for automated workflows, multi-input models, and integration with existing Python pipelines.

Examples and Guides

For practical code examples and step-by-step guides, see:

- Quick Start Guide
- Common Use Cases
- Pre-Optimize API for `dx_com.pre_optimize()`, an ONNX-level transform that reduces CPU-side post-processing for YOLO-family models before compilation.

Overview

The `dx_com.compile()` function is the main entry point for compilation. It performs quantization, optimization, partitioning, and generates compiled artifacts including the `.dxnn` file.

Function Signature

```
def compile(
    model: Union[str, onnx.ModelProto],
    output_dir: str,
    config: Optional[str] = None,
    dataloader: Optional[DataLoader] = None,
    calibration_method: str = "ema",
    calibration_num: int = 100,
    quantization_device: Optional[str] = None,
    opt_level: int = 1,
    aggressive_partitioning: bool = False,
    input_nodes: Optional[List[str]] = None,
    output_nodes: Optional[List[str]] = None,
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    use_q_pro: bool = False,
    enhanced_scheme: Optional[Dict] = None,
    ppu_config: Optional[PPUConfig] = None,
    gen_log: bool = False,
    float64_calibration: bool = False,
    export_html: bool = False,
    quantization_mode: str = "ptq",
    qat_config: Optional[Dict] = None,
    qat_skip_training: bool = False,
    qat_resume_from_checkpoint: Optional[str] = None,
) -> None
```

Additional Parameters

The signature above lists the most commonly used parameters. `dx_com.compile()` accepts further advanced/diagnostic parameters (e.g. `quant_diagnosis`, `super_debug`, `checkpoint` for QXNN resume). See the function docstring (`help(dx_com.compile)`) for the complete list.

Required Parameters

`model`

- **Type**: `Union[str, onnx.ModelProto]`
- **Description**: The ONNX model to compile
  - Can be a file path string to an ONNX model file
  - Or a pre-loaded `onnx.ModelProto` object

```
# Using file path
model="path/to/model.onnx"

# Using ModelProto object
import onnx
model = onnx.load("path/to/model.onnx")
```

`output_dir`

- **Type**: `str`
- **Description**: Directory where compiled artifacts will be saved (e.g., `.dxnn` file)

```
output_dir="./compiled-model"
```


© Copyright 2026 DEEPX All Rights Reserved.

`config` **or** `dataloader` (one must be provided)

`config`

- **Type**: `Optional[str]`
- **Default**: `None`
- **Description**: Path to JSON configuration file containing calibration and compilation settings
- **Mutually exclusive**: Cannot be used together with `dataloader`

```
config="path/to/config.json"
```

`dataloader`

- **Type**: `Optional[DataLoader]`
- **Default**: `None`
- **Description**: PyTorch DataLoader providing calibration data
- **Use Case**: Useful for multi-input models and programmatic data provision
- **Requirement**: `batch_size` must be set to 1
- **Mutually exclusive**: Cannot be used together with `config`

```
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    def __init__(self):
        # Initialize your dataset
        pass

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        # Single-input: return one tensor.
        # Multi-input: return a dict keyed by ONNX input node name (recommended).
        return self.data[idx]

dataset = CustomDataset()
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
```

DEFINING PREPROCESSING TRANSFORMS IN THE DATALOADER

When you compile with a `dataloader`, the JSON `default_loader.preprocessings` block is **not** used. All preprocessing (resize, color conversion, normalization, layout) must be applied **inside the Dataset's**

© Copyright 2026 DEEPX All Rights Reserved.

`__getitem__`, so that each tensor the DataLoader yields is already in the exact shape and value range the ONNX model expects.

> ⚠ **Calibration must match deployment preprocessing**
>
> The transform applied here **must match the preprocessing used at inference time**. A mismatch (e.g. different mean/std, wrong channel order, missing `/255`) degrades calibration quality and post-quantization accuracy.

Two common ways to define transforms:

1. `torchvision.transforms` (PIL-based, recommended for image models):

```
from torchvision import transforms
from PIL import Image

transform = transforms.Compose([
    transforms.Resize((224, 224)),                # resize
    transforms.ToTensor(),                        # HWC uint8 -> CHW float in [0,1]
(implicit /255)
    transforms.Normalize(                         # normalize
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225],
    ),
])

def __getitem__(self, idx):
    image = Image.open(self.image_files[idx]).convert("RGB")  # RGB channel order
    return transform(image)                                   # shape [3, 224, 224]
```

2. OpenCV + manual NumPy/torch (when you need exact control):

```
import cv2
import numpy as np
import torch

def __getitem__(self, idx):
    img = cv2.imread(self.image_files[idx])           # BGR, HWC
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)        # convertColor BGR2RGB
    img = cv2.resize(img, (224, 224))                 # resize
    img = img.astype(np.float32) / 255.0              # div: x=255
    mean = np.array([0.485, 0.456, 0.406], np.float32)
    std = np.array([0.229, 0.224, 0.225], np.float32)
    img = (img - mean) / std                          # normalize
    img = np.transpose(img, (2, 0, 1))                # HWC -> CHW (transpose)
    return torch.from_numpy(img)                      # shape [3, 224, 224]
```


© Copyright 2026 DEEPX All Rights Reserved.

Mapping JSON `preprocessings` to DataLoader transforms:

The `default_loader` reads each image with `cv2.imread` (**BGR**, `HWC`, `uint8`) and applies the operations below in order. When you write a custom `dataloader` you must reproduce the same chain by hand. The table lists every operation in the preprocessing registry and its `key: {args}` form.

JSON operation	Arguments	DataLoader equivalent
`resize`	`width`, `height` (or `size`, `mode`)	`cv2.resize(...)` / `transforms.Resize(...)`
`resize2` / `resize3` / `resize_tv`	resize variants (`mlcommons` / `scale` / `torchvision` mode)	matching resize logic
`centercrop` / `centercrop2`	`width`, `height`	`img[top:top+h, left:left+w]` / `transforms.CenterCrop(...)`
`convertColor`	`form` (e.g. `BGR2RGB`)	`cv2.cvtColor(...)` / `Image.convert("RGB")`
`div`	`x` (scalar or per-channel list)	`img / x` (`div:{x:255}` ≈ `transforms.ToTensor()`)
`mul`	`x`	`img * x`
`subtract`	`x`	`img - x`
`add`	`x`	`img + x`
`normalize`	`mean`, `std` (lists)	`(img - mean) / std` / `transforms.Normalize(...)`
`transpose`	`axis` (e.g. `[2,0,1]`)	`np.transpose(img, axis)` / `tensor.permute(...)`
`expandDim`	`axis`	`np.expand_dims(...)` / `tensor.unsqueeze(axis)`
`squeeze`	`axis`	`np.squeeze(...)` / `tensor.squeeze(axis)`
`slice`	`channel`	`img[..., channel]`
`dtype`	`t` (numpy dtype)	`img.astype(t)`
`pil_2_cv`	—	PIL→numpy BGR conversion

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Output shape, dtype, and batch size

The compiler runs the verifier on `next(iter(dataloader))` and requires the **batched** sample shape to match the ONNX input shape **exactly**, including the batch dimension. So each `__getitem__` item must be `model_input_shape` without the leading batch dim (e.g. `[3, 224, 224]` for input `[1, 3, 224, 224]`), and `batch_size` must equal the model's input batch (normally `1`). Tensors should be `float32`.

Supported return types

Each `__getitem__` may return: a single `torch.Tensor` (single-input models), a **`dict[str, torch.Tensor]`** keyed by ONNX input node name (**recommended for multi-input** — mapped by name), or a **list/tuple of tensors** (mapped by the model's internal input-node order, which may differ from your return order). All elements must be tensors. See the **Use Case 2: Multi-Input Models** section in Common Use Cases.

Optional Parameters

`calibration_method`

- **Type**: `str`
- **Default**: `"ema"`
- **Description**: Calibration method for quantization
- **Supported Values**: `"ema"` (Exponential Moving Average), `"minmax"` (Min-Max method)

`calibration_num`

- **Type**: `int`
- **Default**: `100`
- **Description**: Number of calibration samples to use for quantization

© Copyright 2026 DEEPX All Rights Reserved.

`quantization_device`

- **Type**: `Optional[str]`
- **Default**: `None` (auto-detect: uses GPU if available, otherwise CPU)
- **Description**: Device for quantization computation
- **Supported Values**: `None` (auto-detect), `"cpu"`, `"cuda"`, `"cuda:0"`, `"cuda:1"`, etc.

```
quantization_device="cuda"  # Use GPU
quantization_device="cuda:1"  # Use specific GPU
```

`opt_level`

- **Type**: `int`
- **Default**: `1`
- **Description**: Optimization level
- **Supported Values**:
  - `0`: Fast compilation with basic optimizations
  - `1`: Full optimization (recommended) - provides best performance but takes longer

`aggressive_partitioning`

- **Type**: `bool`
- **Default**: `False`
- **Description**: **(Experimental)** Enable aggressive partitioning to maximize operations on NPU. This feature is currently experimental and may produce unexpected results for some models.
- **Use Case**: Beneficial for systems with limited host CPU performance

`input_nodes`

- **Type**: `Optional[List[str]]`
- **Default**: `None`
- **Description**: List of entry node names for subgraph compilation
- **Note**: Must specify ONNX operator node names (not tensor names)

```
input_nodes=["Conv12", "Conv13"]
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

`output_nodes`

- **Type**: `Optional[List[str]]`
- **Default**: `None`
- **Description**: List of exit node names for subgraph compilation
- **Note**: Must specify ONNX operator node names (not tensor names)

```
output_nodes=["Conv123", "Conv124"]
```

`enhanced_scheme`

- **Type**: `Optional[Dict]`
- **Default**: `None`
- **Description**: Advanced quantization scheme for improved accuracy
- **Limitation**: Not supported for multi-input models
- **Supported Schemes**: `"DXQ-P0"` through `"DXQ-P5"`

```
enhanced_scheme={
    "DXQ-P0": {"alpha": 0.5},
    "DXQ-P2": {
        "alpha": 0.1,
        "beta": 1.0,
        "cosim_num": 2,
    },
}
```

`use_q_pro`

- **Type**: `bool`
- **Default**: `False`
- **Description**: Enable the automatic Q-PRO quantization optimization pipeline. The compiler auto-selects and applies the optimal combination of enhancement stages based on model structure.
- **Limitation**: Mutually exclusive with `enhanced_scheme` (set only one). ONNX compile path only.

© Copyright 2026 DEEPX All Rights Reserved.

`ppu_config`

- **Type**: `Optional[PPUConfig]`
- **Default**: `None` (PPU disabled)
- **Description**: PPU (Post-Processing Unit) configuration object that enables hardware-accelerated post-processing for YOLO-family object detection models. It is the Python-module equivalent of the JSON `ppu` section (see the **PPU Configuration** section in JSON File Configuration).
- **Requirement**: Must be a `PPUConfig` instance. `compile()` calls `ppu_config.validate()` and raises if required fields are missing.

Import `PPUConfig` and `PPUTypes` from the top-level `dx_com` package:

```
from dx_com import PPUConfig, PPUTypes
```

`PPUTypes` maps to the JSON `type` field:

PPUTypes	Value	Architecture	Models
`PPUTypes.YOLO_BASE`	0	Anchor-Based	YOLOv3/v4/v5/v7
`PPUTypes.YOLO_ANCHORFREE`	1	Anchor-Free	YOLOX, YOLOv8–v12
`PPUTypes.YOLOV8`	2	DFL-Based (CPU TopK)	YOLOv8–v12

Construction patterns — full init or incremental builder (chainable setters):

```
# Type 0 (anchor-based) — full init
cfg = PPUConfig(
    type=PPUTypes.YOLO_BASE,
    conf_thres=0.25,
    activation="Sigmoid",
    num_classes=80,
    layer={
        "Conv_245": {"num_anchors": 3},
        "Conv_294": {"num_anchors": 3},
        "Conv_343": {"num_anchors": 3},
    },
)

# Type 0 (anchor-based) — incremental builder
cfg = (
    PPUConfig()
    .set_type(PPUTypes.YOLO_BASE)
    .set_num_classes(80)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    .set_activation("Sigmoid")
    .set_conf_thres(0.25)
)
cfg.add_layer("Conv_245", num_anchors=3)
cfg.add_layer("Conv_294", num_anchors=3)
cfg.add_layer("Conv_343", num_anchors=3)


# Type 1 (anchor-free) — incremental builder
cfg = (
    PPUConfig()
    .set_type(PPUTypes.YOLO_ANCHORFREE)
    .set_num_classes(80)
    .set_conf_thres(0.25)
)
cfg.add_layer(bbox="Mul_441", cls_conf="Sigmoid_442")

# Type 2 (DFL-based, CPU-side TopK)
cfg = PPUConfig(type=PPUTypes.YOLOV8, num_classes=80, topk=512)
cfg.add_layer(bbox="bbox_head_p3", cls_conf="cls_head_p3")
cfg.add_layer(bbox="bbox_head_p4", cls_conf="cls_head_p4")
cfg.add_layer(bbox="bbox_head_p5", cls_conf="cls_head_p5")
```

`PPUConfig` builder methods:

Method	Purpose
`set_type(PPUTypes.*)`	Set PPU type (resets `layer`)
`set_num_classes(int)`	Set number of detection classes
`set_conf_thres(float)`	Set confidence threshold (type 0/1)
`set_activation(str)`	Set activation, e.g. `"Sigmoid"` (type 0)
`set_topk(int)`	Set TopK candidate count (type 2)
`add_layer(...)`	Add a detection head; signature depends on type
`validate()`	Validate required fields (called by `compile()`)

`add_layer` signature by type

- **Type 0**: `add_layer("Conv_245", num_anchors=3)` — `layer` is a dict.
- **Type 1 / 2**: `add_layer(bbox="...", cls_conf="...")` (optional `obj_conf=`) — `layer` is a list. Call once per detection scale.

© Copyright 2026 DEEPX All Rights Reserved.

See the **Use Case 8: PPU Hardware Acceleration** section in Common Use Cases for a complete script.

`gen_log`

- **Type**: `bool`
- **Default**: `False`
- **Description**: Enable detailed logging for debugging

`float64_calibration`

- **Type**: `bool`
- **Default**: `False`
- **Description**: Use float64 precision during calibration and offset calculations for cross-CPU determinism

`export_html`

- **Type**: `bool`
- **Default**: `False`
- **Description**: Generate a self-contained HTML summary report (`<model_name>_summary.html`) in the output directory after compilation. See Compilation Summary Report for details.

`use_q_pro`

- **Type**: `bool`
- **Default**: `False`
- **Description**: Enable the automatic Q-PRO quantization pipeline. DX-COM automatically selects and applies the optimal DXQ enhancement stages.
- **Limitation**: Mutually exclusive with `enhanced_scheme` .
- **See also**: the **Automatic Q-PRO (`use_q_pro`)** section above

`quant_diagnosis`

- **Type**: `bool`
- **Default**: `False`
- **Description**: Generate an HTML quantization diagnosis report. Produces `{output_dir}/quant_diagnosis/{model}.qxnn` (resume checkpoint) and `{output_dir}/quant_diagnosis/diagnosis_report.html` .
- **See also**: Quantization Tuning Workflow

© Copyright 2026 DEEPX All Rights Reserved.

`checkpoint` *(QXNN Resume)*

- **Type**: `Optional[str]`
- **Default**: `None`
- **Description**: Path to a `.qxnn` resume artifact. When provided, selects the **QXNN resume** path, which re-runs quantization without recompiling. The `model` / `config` arguments are not required in this mode.
- **Related parameters** (resume-only): `recalibration_method` (`"minmax"` / `"ema"` / `"iqr"`), `dataset_path`, and `enhanced_scheme`.

```
import dx_com

dx_com.compile(
    checkpoint="output/large_model/quant_diagnosis/large_model.qxnn",
    output_dir="output/large_model_iqr",
    recalibration_method="iqr",   # or: use_q_pro=True
)
```

`quantization_mode`

- **Type**: `str`
- **Default**: `"ptq"`
- **Description**: Quantization mode. The default `"ptq"` runs Post-Training Quantization. When the config JSON contains a `qmaster` block, QAT is **auto-selected** — you do not need to pass this argument. Set to `"qat"` explicitly only when supplying `qat_config` directly in Python; doing so bypasses `qmaster` auto-detection. When set to `"qat"`, you must provide either `qat_config` (for training) or `qat_skip_training=True` (for compile-only/resume).
- **Supported Values**: `"ptq"`, `"qat"`

`qat_config`

- **Type**: `Optional[Dict]`
- **Default**: `None`
- **Description**: QAT training hyperparameters. Usually supplied via the `qmaster` block in the config JSON instead of this argument.

`qat_skip_training`

- **Type**: `bool`
- **Default**: `False`
- **Description**: Skip the QAT training loop and run compilation only. Use together with `qat_resume_from_checkpoint`.

© Copyright 2026 DEEPX All Rights Reserved.

`qat_resume_from_checkpoint`

- **Type**: `Optional[str]`
- **Default**: `None`
- **Description**: Path to a saved `qat_checkpoint.qxnn` to load trained weights before compilation.

> **QAT Details**
>
> For the full QAT workflow, the `qmaster` block, and all training hyperparameters, see Quantization-Aware Training (QAT).

Return Value

- **Type**: `None`
- **Behavior**: Compiled artifacts are saved to the specified `output_dir`

Usage Examples

Basic Compilation with Configuration File

```
import dx_com

dx_com.compile(
    model="model.onnx",
    output_dir="./compiled",
    config="config.json",
)
```

For more detailed examples — including DataLoader usage, multi-input models, edge device optimization, and advanced quantization — see Common Use Cases.

© Copyright 2026 DEEPX All Rights Reserved.

Important Considerations

> **Input Selection: Config vs DataLoader**
>
> Users must provide **either** a configuration file **or** a DataLoader. These inputs are mutually exclusive.
>
> - **Config:** Recommended for static, file-based compilation workflows.
> - **DataLoader:** Required for programmatic data provision and models with multiple inputs. When constructing a DataLoader for compilation, the **batch_size must be set to 1.**

> **Hardware Acceleration (CUDA)**
>
> To enable GPU-accelerated quantization (quantization_device="cuda"), ensure the following requirements are met:
>
> - **System:** NVIDIA CUDA drivers and toolkit are installed.
> - **Framework:** PyTorch is built with CUDA support (torch.cuda.is_available() is True).

> **Deprecation Notice: CustomLoader**
>
> The legacy CustomLoader for non-image data is **deprecated.**
>
> - **New Standard:** Use the standard **PyTorch DataLoader** for all data modalities (Image, Tensor, etc.) to ensure long-term compatibility and performance.

Output Files

Upon successful compilation, the `output_dir` will contain:

- `[model_name].dxnn` : Compiled model binary for execution on DEEPX NPU hardware
- `compiler.log` (if `gen_log=True`): Detailed compilation logs
- `[model_name]_summary.html` (if `--export_html` / `export_html=True`): Self-contained HTML compilation summary report

---

© Copyright 2026 DEEPX All Rights Reserved.

3.6.4 Common Errors and Troubleshooting

The following error types may occur during the compilation process using either the `dxcom` command or the `dx_com` Python module. Understanding these errors will help you troubleshoot issues regardless of which interface you choose.

No	Error Type	Description & Conditions
1	NotSupportError	Triggered when using features unsupported by the compiler. Examples: multi-input models with the `dxcom` command, dynamic input shape, cubic resize
2	ConfigFileError	Invalid or missing JSON configuration file. Examples: incorrect file path, malformed JSON syntax
3	ConfigInputError	Input definitions in the config file do not match the ONNX model. Examples: mismatched input name or shape
4	DatasetPathError	The dataset path specified in the configuration is invalid. Examples: path does not exist, or is not a directory
5	NodeNotFoundError	The ONNX model contains a node that is unsupported by the compiler
6	OSError	The operating system is unsupported. Examples: OS is not Ubuntu
7	UbuntuVersionError	The installed Ubuntu version is outside the supported range
8	LDDVersionError	The installed `ldd` version is unsupported
9	RamSizeError	

© Copyright 2026 DEEPX All Rights Reserved.

No	Error Type	Description & Conditions
		The system does not meet the minimum RAM requirements
10	DiskSizeError	Available disk space is insufficient for compilation
11	NotsupportedPaddingError	Padding configuration is unsupported. Examples: asymmetric padding in width and height
12	RequiredLibraryError	Missing essential system libraries. Examples: `libgl1-mesa-glx` is not installed
13	DataNotFoundError	No valid input data found in the specified dataset path. Examples: empty folder, wrong file extensions
14	OnnxFileNotFound	The ONNX model file cannot be found or does not exist at the specified location

ⓒ Copyright 2026 DEEPX All Rights Reserved.

4. Supported ONNX Operators

This chapter describes the ONNX operations currently supported by DX-COM. When you build or export models to ONNX format, you **must** use only the supported operations to ensure successful compilation and optimal performance on our NPU.

4.1 Operator Support Details

The following ONNX operators are supported by the compiler.

4.1.1 Common Conditions (Applicable to All Operation Types)

Tensor Shape Limitations

- **Width, height:** < 8,192
- **Channels:** < 32,768
- Dynamic shapes are not supported.

Broadcasting Restrictions

- In element-wise operations like Add, Div, Mul, and Sub, **channel-wise broadcasting** is not supported when the channel dimension size is greater than **1**.
- **Example:** A tensor with shape 1x24x24x1 (NHWC) cannot be broadcast to shape 1x24x24x32.

4.1.2 Normal Operations

Operator	Supported Conditions
Add	Supported as: - Bias addition (e.g., as part of `Gemm` or `Conv`) - Element-wise addition - Used for input normalization - Constant scalar addition
ArgMax	Only supported if all of the following hold: - It is the final operation in the network

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
	- The preceding output is 2D or 4D - It operates along the channel dimension
AveragePool	- `kernel_shape` < 32 - `strides` < 32
BatchNormalization	No restrictions
Clip	Only supported as an activation function (e.g., ReLU6)
Concat	No restrictions
Constant	Only numeric constants are supported
ConstantOfShape	No restrictions
Conv	**Common constraints:** - `dilations` < 64 - `pads` < 64 - `strides` < 16 **Standard Conv:** - `kernel_shape` < 16 **Depth-wise Conv:** - `kernel_shape` ∈ {[3, 3], [5, 5]} - Only constant weights are supported
ConvTranspose	- `dilations` = [1, 1] - `output_padding` = [0, 0] - `pads` ≤ 14 - `strides` ∈ [2, 8] and symmetric (`strides[0]` = `strides[1]`) - `kernel_shape` < 16 - `group` = 1
Div	Supported as: - Constant scalar division - Input normalization - Part of `Softmax` - Part of `LayerNorm`
Dropout	Removed during inference
Erf	Only supported as part of `GELU`

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
Flatten	No restrictions
Gather	Supported when `indices` is a 0-D or 1-D tensor. Examples: - Scalar index: `indices = [0]` to select first element - 1-D index: `indices = [0, 2, 5]` to select multiple elements
Gemm	No restrictions
GlobalAveragePool	No restrictions
Identity	No restrictions
MatMul	No restrictions
MaxPool	- `kernel_shape` < 16 - `strides` < 16
Mul	Supported as: - Element-wise multiplication - Constant scalar multiplication - Input normalization
Pad	- Only `mode=constant` is supported - Must precede a `Pool` or `Conv` operation
ReduceMean	Only supported when reducing along: - Channel dimension - (Width, Height) dimensions
ReduceSum	Only supported when reducing along the channel dimension
Reshape	No restrictions
Resize	Only supported with the following attributes: - `coordinate_transformation_mode` = `pytorch_half_pixel` - `mode` $\in$ {`nearest`, `linear`} - Scale values $\in \mathbb{Z}$ (integers)

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
Shape	Cannot be used as a model output
Slice	No restrictions
Softmax	Only supported if the size of the input along the specified `axis` is ≤ 4080
Split	No restrictions
Squeeze	No restrictions
Sub	Supported as: - Element-wise subtraction - Constant scalar subtraction - Input normalization
Transpose	No restrictions

4.1.3 Activation Functions

Operator	Supported Conditions
Elu	No restrictions
Gelu	No restrictions
HardSwish	No restrictions
HardSigmoid	No restrictions
LeakyRelu	No restrictions
Mish	No restrictions
PRelu	No restrictions on functionality. **Known issue (v2.4.0):** models using `PRelu` as an activation function exhibit significant FPS degradation on the NPU runtime. Use `LeakyRelu` or `Relu` where possible until this is resolved in a future release. See the **Known Issues** section in Change Log for details.
Relu	No restrictions

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
Selu	No restrictions
Sigmoid	No restrictions
Silu (Swish)	No restrictions
Softplus	No restrictions
Tanh	No restrictions

4.1.4 Deprecated Operations

The following operations are deprecated in ONNX and maintained here only for backward compatibility.
Their usage is discouraged in new models and may be removed in future versions.
Please use alternative operators where possible.

Operator	Supported Conditions
Upsample	Only supported when scale values in the N and C dimensions are 1

NOTE

The operator support may vary depending on how operations are combined within a model. This document is intended as a general guideline. For validation of specific use cases, please contact our technical support team.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

5. Quantization Tuning Workflow

5.1 Quantization Tuning Workflow

When a model's quantized accuracy is lower than expected, DX-COM provides a tuning loop that lets you **diagnose** which regions are degrading accuracy and **re-quantize** with new settings **without repeating the full compilation**. This chapter covers the two features that make up that loop:

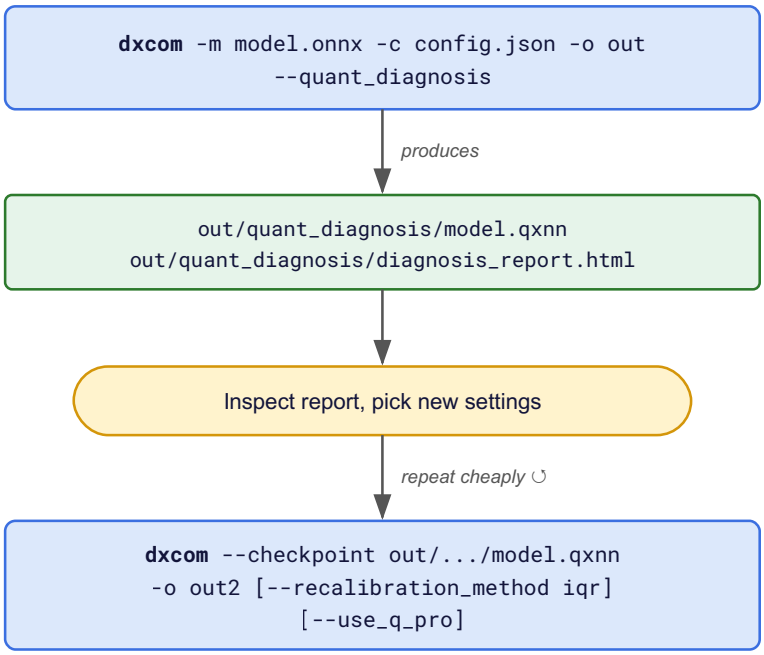
- **Quantization Diagnosis (`quant_diagnosis`)** — generates an HTML report of per-region quantization quality and a reusable `.qxnn` checkpoint.
- **QXNN Resume** — re-runs quantization from a `.qxnn` checkpoint, skipping the earlier compile phases.

> **Version Support**
>
> `quant_diagnosis` and QXNN Resume are available in **DX-COM v2.4.0 and later**. Both are accessible from the `dxcom` CLI and the `dx_com` Python API.

5.1.1 The Diagnose → Resume Loop

Standard quantization tuning requires recompiling the model from ONNX every time you change a setting, which is slow. The diagnose → resume loop removes that cost:

ⓒ Copyright 2026 DEEPX All Rights Reserved.



Because the resume path reuses everything that does not depend on quantization, you can iterate over several calibration methods or Q-PRO settings quickly.

5.1.2 Quantization Diagnosis (`quant_diagnosis`)

Enabling `quant_diagnosis` during compilation produces an HTML report that visualizes per-region quantization quality, flags high-severity regions, and includes ready-to-paste compile snippets for retrying with recommended settings. It also emits a `.qxnn` resume artifact so you can act on those recommendations without recompiling.

The option is self-contained — it does not require any other debug flag.

> **What is a *region*?**
>
> A **region** is a quantization block — a group of operations that are quantized and evaluated together as one unit during diagnosis. The report grades each region into a severity tier (**OK / Warning / Critical**) and reports region-level metrics (Cosine Similarity, QSNR) measured in isolation, which may not fully reflect deployed whole-model behavior.

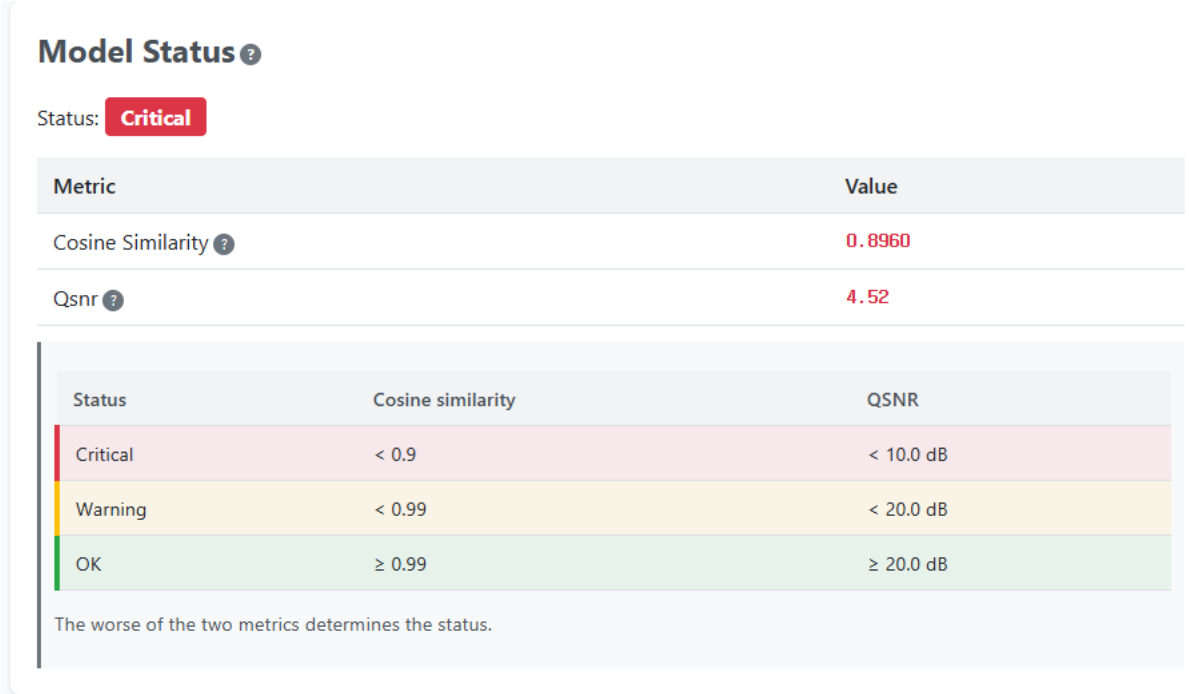
© Copyright 2026 DEEPX All Rights Reserved.

The report contains three parts:

- **Model Status** — overall judgment (OK / Warning / Critical) with whole-model Cosine Similarity and QSNR.
- **Candidate Recompile Intents** — suggested next steps (e.g., a DXQ-P* scheme or a different calibration method), each with ready-to-paste Python API and `dxcom` CLI snippets.
- **Evidence Summary** — region severity distribution and weight/activation quantization details.

Reading the Report

Model Status gives the at-a-glance verdict. The status badge (here, *Critical*) is derived from the worse of the whole-model Cosine Similarity and QSNR, graded against the thresholds shown below it.



Candidate Recompile Intents lists the recommended next steps in priority order. Each intent comes with ready-to-paste **Python API** and **`dxcom` CLI** snippets that target the `.qxnn` checkpoint, so you can apply the suggestion directly via QXNN Resume. The candidate capabilities (e.g., `DXQ-P3` / `P4` / `P5`) are mutually-exclusive alternatives — pick one.

© Copyright 2026 DEEPX All Rights Reserved.

Candidate Recompile Intents

Intent	Category	Candidate Capabilities	Confidence	Rationale
Optimize Quantization	Weight Optimization	DXQ-P3, DXQ-P4, DXQ-P5	medium	Weight quantization sensitivity detected. Candidate schemes are mutually-exclusive alternatives; the compiler allows only one weight optimization scheme per pipeline.

Choose ONE of: DXQ-P3, DXQ-P4, DXQ-P5 (example below uses DXQ-P3)

Python API

```
compile(checkpoint="<your_model>.qxnn", ..., enhanced_scheme={"DXQ-P3": {"num_samples": 1024}})
```

CLI (dxcom)

```
dxcom --checkpoint <your_model>.qxnn --enhanced_scheme P3:num_samples=1024
```

→ compare QSNR/cosine

Intent	Category	Candidate Capabilities	Confidence	Rationale
Adjust Activation Range Modeling	Calibration Method	iqr	medium	Activation distribution outlier pressure detected. Switching the calibration method is a low-cost first step; iqr is robust to outliers (default is ema).

Python API

```
compile(checkpoint="<your_model>.qxnn", ..., recalibration_method="iqr")
```

CLI (dxcom)

```
dxcom --checkpoint <your_model>.qxnn --recalibration_method iqr
```

→ compare QSNR/cosine

Intent	Category	Candidate Capabilities	Confidence	Rationale
Optimize Quantization	Activation Outlier Smoothing	DXQ-P0	exploratory	If activation range adjustment is insufficient, DXQ-P0 (SmoothQuant) applies pre-calibration outlier smoothing. Note: DXQ-P0 is primarily effective on ViT-family (transformer/attention) models and is generally not effective on CNN-family models.

Python API

```
compile(checkpoint="<your_model>.qxnn", ..., enhanced_scheme={"DXQ-P0": {"alpha": 0.5}})
```

CLI (dxcom)

```
dxcom --checkpoint <your_model>.qxnn --enhanced_scheme P0:alpha=0.5
```

→ compare QSNR/cosine

Evidence Summary (collapsible) backs the recommendations with the per-region severity distribution and weight/activation quantization details that the diagnosis is based on.

© Copyright 2026 DEEPX All Rights Reserved.

▾ Evidence Summary

Region Severity Distribution

A region is a quantization block — a group of operations that are quantized and evaluated together as one unit during diagnosis. Region-level metrics are measured in isolation and may not reflect the deployed whole-model behavior.

Total analyzed regions: **10**

- CRITICAL ?: 5
- WARNING ?: 5
- OK ?: 0

Activation Quantization Summary ?

(measured per analyzed region, in isolation)

Detected: **No**

Worst analyzed-region ? QSNR: **22.77 dB**

Worst analyzed-region ? Cosine Similarity: **0.9974**

Activation distribution risk level ?: **High**

Weight Quantization Summary ?

(measured per analyzed region, in isolation)

Detected: **Yes**

Worst analyzed-region ? QSNR: **9.47 dB**

Worst analyzed-region ? Cosine Similarity: **0.9675**

Quantization Interaction ?

Interaction level: **Additive**

Level	Gap (vs additive)	Meaning
Additive ◀ current	**≤ 2 dB**	**Combined error ≈ sum of independent errors; no significant interaction**
Mild interaction	2–6 dB	Combined error moderately exceeds the independent prediction
Interaction-dominated	> 6 dB	Combined error far exceeds independent prediction

Gap ?: -0.9 dB worse than independent-error (additive) prediction

Whole-model weight-only QSNR ?: **3.9 dB**

Whole-model weight-only Cosine ?: **0.9089**

Whole-model activation-only QSNR ?: **15.6 dB**

Whole-model activation-only Cosine ?: **0.9864**

Dominant axis (whole-model) ?: weight

Per-axis (weight-only / activation-only) metrics are measured in isolation and do not capture this joint effect; only the whole-model metric reflects the deployed model.

© Copyright 2026 DEEPX All Rights Reserved.

Usage

`dxcom` Command

```
dxcom \
  -m large_model.onnx \
  -c config.json \
  -o output/large_model \
  --quant_diagnosis
```

`dx_com` Python Module

```
import dx_com

dx_com.compile(
    model="large_model.onnx",
    output_dir="output/large_model",
    config="config.json",
    quant_diagnosis=True,
)
```

Output Files

`quant_diagnosis` creates a `quant_diagnosis/` subdirectory under `output_dir`:

File	Description
`quant_diagnosis/{model}.qxnn`	Resume checkpoint consumed by QXNN Resume
`quant_diagnosis/diagnosis_report.html`	Per-region quantization quality report with retry snippets

> **Note**
>
> The `.qxnn` checkpoint is always written when diagnosis runs; the HTML report is written on diagnosis success.

5.1.3 QXNN Resume (Re-quantization Without Recompile)

QXNN Resume takes a `.qxnn` checkpoint and re-runs only the quantization-dependent stages (`Quantization → Optimize → Codegen → SingleArtifact`). The original ONNX model and config are **not** needed — the calibration settings are embedded in the checkpoint.

© Copyright 2026 DEEPX All Rights Reserved.

A `.qxnn` checkpoint is produced by `quant_diagnosis` (see above).

Selecting Resume Mode

QXNN resume mode is selected by passing a `.qxnn` checkpoint:

- **CLI**: `--checkpoint <path>.qxnn`
- **Python**: `checkpoint="<path>.qxnn"`

The checkpoint file extension must be `.qxnn` (case-insensitive); any other extension raises `ValueError`.

Usage

Re-calibrate with a different observer:

```
dxcom \
  --checkpoint output/large_model/quant_diagnosis/large_model.qxnn \
  -o output/large_model_iqr \
  --recalibration_method iqr
```

Enable automatic Q-PRO on resume:

```
dxcom \
  --checkpoint output/large_model/quant_diagnosis/large_model.qxnn \
  -o output/large_model_qpro \
  --use_q_pro
```

`dx_com` Python Module:

```
import dx_com

dx_com.compile(
    checkpoint="output/large_model/quant_diagnosis/large_model.qxnn",
    output_dir="output/large_model_iqr",
    recalibration_method="iqr",   # or: use_q_pro=True
)
```


© Copyright 2026 DEEPX All Rights Reserved.

Resume-Only Options

The following arguments apply **only** in QXNN resume mode (i.e., together with `--checkpoint`). Using them in standard compile mode raises an error.

CLI Option	Python Parameter	Description
`--checkpoint`	`checkpoint`	Path to the `.qxnn` resume artifact. Selects resume mode.
`--recalibration_method`	`recalibration_method`	Observer override for re-calibration. One of `minmax`, `ema`, `iqr`.
`--enhanced_scheme`	`enhanced_scheme`	Manual Q-PRO scheme selection (e.g., `P3:num_samples=1024`). Mutually exclusive with `--use_q_pro`.
`--dataset_path`	`dataset_path`	Override the calibration dataset path embedded in the checkpoint.

Constraints

- `--checkpoint` and `-m/--model_path` are **mutually exclusive**.
- `--output_dir` (`-o`) is **required** for resume.
- `--use_q_pro` and `--enhanced_scheme` are **mutually exclusive**.
- `-c/--config_path` is **not required** — on resume the calibration DataLoader is auto-built from the config embedded in the `.qxnn` (default loader only). Use `--dataset_path` to point at a different dataset.

Output Files

QXNN Resume writes the re-quantized model binary to `output_dir`:

- `{checkpoint_stem}.dxnn` — re-quantized model binary (filename defaults to the checkpoint stem; override with `output_name`).

© Copyright 2026 DEEPX All Rights Reserved.

5.1.4 End-to-End Example

For a complete, step-by-step walkthrough of diagnosing a model and iterating on quantization settings, see the **Use Case 6: Diagnose and Re-quantize Without Recompile** section in Common Use Cases.

For the automatic Q-PRO quantization pipeline used during resume, see the **Automatic Q-PRO (`use_q_pro`)** section in Execution of DX-COM.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

5.2 Quantization-Aware Training (QAT)

This chapter describes how to compile a model with **Quantization-Aware Training (QAT)** using **DX-COM**. QAT fine-tunes the model while simulating quantization, which can recover accuracy that is otherwise lost by post-training quantization (PTQ) alone.

QAT reuses the **same JSON configuration** as a normal (PTQ) compile. When the configuration file contains a `qmaster` block, `dx_com.compile()` automatically switches to QAT mode and runs the training pipeline using the same dataset and preprocessing settings as PTQ calibration.

> **Version Support**
>
> QAT (the `qmaster` block) is supported in **DX-COM v2.4.0 and later**.

5.2.1 How QAT Is Triggered

QAT is enabled by **adding a `qmaster` block** to the JSON configuration. No separate command-line flag is required.

- If the config contains a `qmaster` block → DX-COM runs **QAT** (Calibration → Training → Compilation).
- If the config has **no** `qmaster` block → DX-COM runs the normal **PTQ** compile.

```
{
  "inputs": { "input.1": [1, 3, 224, 224] },
  "calibration_num": 100,
  "calibration_method": "ema",
  "default_loader": { "dataset_path": "/datasets/ILSVRC2012/train",
"file_extensions": ["jpeg","jpg","png","JPEG"], "preprocessings": [ "..." ] },
  "qmaster": { "epochs": 30, "lr": 1e-5, "use_kd": true }
}
```

> **Reuses Your Existing Config**
>
> `inputs`, `calibration_*`, and `default_loader` work exactly as described in JSON File Configuration. The image preprocessing pipeline (`default_loader.preprocessings`) is the single source of truth for both calibration and QAT training data — you do **not** redefine it inside `qmaster`.

© Copyright 2026 DEEPX All Rights Reserved.

5.2.2 Compilation Stages

A QAT compile runs three stages automatically:

1. **Calibration** — estimates initial quantization parameters (shared with PTQ).
2. **Training (Stage 1)** — runs the QAT fine-tuning loop. The best checkpoint (lowest validation loss) is saved to `qat_checkpoint/qat_checkpoint.qxnn`.
3. **Compilation (Stage 2)** — converts the trained weights into the NPU binary (`*.dxnn`).

5.2.3 The `qmaster` Block

The `qmaster` block holds **training hyperparameters only**. Every key is optional; omitted keys fall back to the defaults below.

Parameter Quick Reference

Parameter	Default	Description
`batch_size`	`1`	Training batch size. Also selects the internal compile path (see note below). Default `1` uses the single-shot path; set to `> 1` (e.g. `16`) to use the staged path recommended for most models.
`num_workers`	`4`	Number of DataLoader workers.
`train_limit` / `val_limit`	`500` / `50`	Number of training / validation samples (subset).
`device`	`"cuda:0"`	Training device.
`epochs`	`30`	Maximum number of training epochs.
`lr`	`1e-5`	Learning rate.
`optimizer`	`"adamw"`	Optimizer: `"adamw"` or `"sgd"`.
`criterion`	`mse`	Task loss: `mse` or `cross_entropy`. Applied when use_kd = false, or when use_kd =

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Default	Description
		true and kd_alpha < 1.0 (task loss weight = 1 - kd_alpha). With the default use_kd = true and kd_alpha = 1.0, task loss is disabled and this value has no effect.
`scheduler`	`null`	LR scheduler: `null`, `"cosine"`, or `"step"`.
`scheduler_step_size`	`max(1, (epochs - warmup_epochs) // 3)`	Step interval (in epochs) for the `"step"` scheduler. Only used when `scheduler = "step"`.
`weight_decay`	`1e-4`	Optimizer weight decay.
`max_grad_norm`	`1.0`	Gradient-clipping max norm.
`use_amp`	`true`	Mixed-precision (AMP) training.
`warmup_epochs`	`0`	Scheduler warmup epochs.
`gradient_accumulation_steps`	`1`	Effective batch = `batch_size × this value`.
`early_stopping_patience`	`5`	Stop early after N epochs with no improvement.
`early_stopping_delta`	`1e-3`	Minimum improvement counted as progress.
`save_best_model`	`true`	Save the best (lowest val-loss) checkpoint.
`use_kd`	`true`	Enable Knowledge Distillation (FP teacher → quantized student).
`kd_loss`	`"mse"`	KD loss type.
`kd_alpha`	`1.0`	KD loss weight. The task loss is weighted by `1 - kd_alpha`, so the default `1.0` trains with **KD only** (task loss disabled).
`kd_temperature`	`4.0`	KD softening temperature.

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Default	Description
`encoder_mode`	`false`	Track accuracy as **cosine similarity** instead of classification accuracy (for embedding/encoder-style models). Does not by itself disable the task loss — combine with `kd_alpha = 1.0` for pure KD training.
`freeze_bn_after`	`null`	Freeze BatchNorm from the given epoch. See warning below.
`train_cpu_fp`	`false`	Keep the FP teacher on CPU to save GPU memory (slightly slower).
`fast_run`	`false`	Quick smoke test: 1 epoch × 1 batch. Result is **not** accuracy-meaningful.

Batch Size and the Internal Compile Path

`batch_size > 1` runs the **staged** path (recommended for most models); `batch_size = 1` runs the **single-shot** path. If a batched run is not possible for a given model, DX-COM automatically falls back to single-shot at batch size 1.

Data and Device

```
"qmaster": {
  "batch_size": 16,
  "num_workers": 4,
  "train_limit": 500,
  "val_limit": 50,
  "device": "cuda:0"
}
```

Training Loop

```
"qmaster": {
  "epochs": 30,
  "lr": 1e-5,
  "optimizer": "adamw",
  "criterion": "mse",
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  "scheduler": "cosine",
  "warmup_epochs": 1,
  "weight_decay": 1e-4,
  "max_grad_norm": 1.0,
  "use_amp": true,
  "early_stopping_patience": 5
}
```

Knowledge Distillation (KD)

KD uses the original floating-point model as a teacher to guide the quantized student. It is enabled by default and generally improves accuracy.

```
"qmaster": {
  "use_kd": true,
  "kd_loss": "mse",
  "kd_alpha": 1.0,
  "kd_temperature": 4.0
}
```

KD Loss Weighting

`kd_alpha` weights the KD term; the task loss is weighted by `1 - kd_alpha`. With the default `kd_alpha = 1.0` the model trains with **KD only**. Lower it (e.g. `0.7`) to blend KD with the task loss.

Encoder Mode

For embedding/encoder-style models (e.g. CLIP image encoders), set `"encoder_mode": true`. This switches the tracked metric to **cosine similarity** (instead of classification accuracy). It does not disable the task loss on its own; keep `kd_alpha = 1.0` (the default) for pure KD training.

5.2.4 Control Parameters (Python API)

The following parameters control **training vs. compilation** behavior. They are available through the Python API (`dx_com.compile()`); the `dxcom` CLI runs the full QAT pipeline directly from the `qmaster` block.

Parameter	Type	Default	Description
`quantization_mode`	`str`	`"ptq"`	

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Type	Default	Description
			Keep as `"ptq"` (default) to let QAT be auto-selected when a `qmaster` block is present. Set to `"qat"` only when supplying `qat_config` directly in Python — doing so bypasses `qmaster` auto-detection. When set to `"qat"`, you must provide either `qat_config` (for training) or `qat_skip_training=True` (for compile-only/ resume).
`qat_config`	`Optional[Dict]`	`None`	QAT training hyperparameters. Normally supplied via the `qmaster` block in the config JSON; this argument is an alternative for callers that build the config in code.
`qat_skip_training`	`bool`	`False`	Skip the training loop and run compilation only (Stage 2). Use with `qat_resume_from_checkpoint`.
`qat_resume_from_checkpoint`	`Optional[str]`	`None`	Path to a `qat_checkpoint.qxnn`. Loads trained weights, then compiles (or continues).

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Re-running Compilation Only

Training can take a long time. After a successful run you can regenerate the `.dxnn` without re-training by passing the saved checkpoint: `qat_skip_training=True` together with `qat_resume_from_checkpoint="<path>.qxnn"`.

`fast_run` Is a Config Key

To run a quick smoke test, set `"fast_run": true` **inside the `qmaster` block** of the JSON config (it is not a `compile()` argument).

5.2.5 Usage

CLI (`dxcom`)

Add a `qmaster` block to your config and compile as usual — QAT runs automatically.

```
dxcom -m model.onnx -c config_with_qmaster.json -o output/
```

Python API

```
import dx_com

# Basic QAT (training + compilation). qmaster block in config triggers QAT
automatically.
dx_com.compile(
    model="model.onnx",
    config="config_with_qmaster.json",
    output_dir="output/",
)

# Compilation only, reusing a previously trained checkpoint.
dx_com.compile(
    model="model.onnx",
    config="config_with_qmaster.json",
    output_dir="output/",
    quantization_mode="qat",
    qat_skip_training=True,
    qat_resume_from_checkpoint="output/qat_checkpoint/qat_checkpoint.qxnn",
)
```


© Copyright 2026 DEEPX All Rights Reserved.

5.2.6 Output Files

File	Description
`<output_dir>/*.dxnn`	Compiled NPU binary (the deliverable).
`<output_dir>/qat_checkpoint/qat_checkpoint.qxnn`	Best training checkpoint, for `qat_resume_from_checkpoint`.

5.2.7 Notes and Recommendations

BatchNorm Freezing

`freeze_bn_after = 0` freezes BatchNorm from the very first epoch, which can cause the loss to diverge (`nan`). Leave it as `null` unless you specifically need BN freezing, and in that case start from a later epoch.

Dataset Path

If `default_loader.dataset_path` is missing or invalid, DX-COM falls back to a default ImageNet location instead of failing. Verify the dataset path in the log to make sure training used the data you intended.

Reducing GPU Memory

If you hit out-of-memory errors, lower `batch_size` and/or raise `gradient_accumulation_steps` to keep the effective batch size constant. Setting `"train_cpu_fp": true` moves the FP teacher to CPU to save GPU memory (slightly slower).

5.2.8 Related Pages

- JSON File Configuration — base config (`inputs`, `calibration_*`, `default_loader`).
- Execution of DX-COM — CLI and Python API reference.
- Change Log — QAT was added in v2.4.0.

© Copyright 2026 DEEPX All Rights Reserved.

5.3 Pre-Optimize API

`dx_com.pre_optimize()` is an ONNX-level transformation API that restructures the CPU-side post-processing of a model so that expensive operations are applied only to a small set of TopK candidates. It is intended for edge environments where CPU performance is limited (for example, ARM cores) and CPU post-processing becomes a bottleneck for end-to-end throughput.

When to use

Use `pre_optimize()` when CPU post-processing (Sigmoid, DFL decoding, dist2bbox, NMS pre-filtering, etc.) becomes a bottleneck for end-to-end inference throughput. It is most effective for YOLO-family detection / instance-segmentation models on CPU-constrained hosts.

Replaces PPU Type 2

`pre_optimize()` is the recommended replacement for the deprecated **PPU Type 2** post-processing mode. See Migration from PPU Type 2 below. PPU Type 2 will be removed in a future release.

5.3.1 Overview

The detection head of YOLO-family models contains operations that run on the host CPU after the NPU has finished inference: Sigmoid, Softmax, DFL decoding, dist2bbox, and so on. When these operations run over the full set of anchor candidates (for example, 8400 candidates for a 640×640 input), CPU post-processing can dominate the end-to-end latency and leave the NPU idle while waiting for the next frame.

`pre_optimize()` reorders the post-processing graph so that:

1. **TopK selection happens first**, narrowing the candidate set to `K` (default 300).
2. **Expensive operations run only on those `K` candidates**, instead of the full anchor grid.

The result is a model whose CPU post-processing cost is roughly proportional to `K` rather than to the full anchor count.

© Copyright 2026 DEEPX All Rights Reserved.

5.3.2 API Reference

Signature

```
dx_com.pre_optimize(
    model: onnx.ModelProto,
    passes: dict[str, dict],
) -> onnx.ModelProto
```

Parameters

Parameter	Type	Required	Description
`model`	`onnx.ModelProto`	Yes	Source ONNX model loaded via `onnx.load()`.
`passes`	`dict[str, dict]`	Yes	Mapping from pass name to its configuration. See Available Passes. Specify exactly one pass per call.

Returns

An optimized `onnx.ModelProto` with the original CPU post-processing replaced by the TopK-first equivalent. The returned model can be passed directly to `dx_com.compile()` via the `model` argument, or saved with `onnx.save()`.

Raises

- `KeyError` — if the pass name is not recognized, or if a required pass configuration key is missing. See the schema below for which keys are required vs. optional.
- `ValueError` — if a tensor name in `layers` cannot be resolved in the model graph.

`passes` Configuration Schema

Field	Type	Required	Description
`layers`	`list[dict]`	Yes	One entry per detection scale. Each entry maps tensor roles (`bbox`, `cls_conf`,

© Copyright 2026 DEEPX All Rights Reserved.

Field	Type	Required	Description
			`mask_coeff`) to the Conv-head output tensor name.
`num_classes`	`int`	Yes	Number of classification classes.
`topk`	`int`	No (default `300`)	Number of candidates to keep after TopK selection.
`input_height`	`int`	Yes	Input image height (used to derive per-scale stride).
`input_width`	`int`	Yes	Input image width (used to derive per-scale stride).

5.3.3 Available Passes

`yolo_postprocess`

Field	Value
Target models	YOLOv8, YOLOv9, YOLOv11, YOLOv12, YOLOv13
Head structure	64-channel bbox + `C`-channel classification, optional mask coefficient
Supported tasks	Detection, Instance Segmentation
Output (det)	`[N, 4 + C, K]` — bbox (xywh) + class scores
Output (seg)	`[N, 4 + C + M, K]` + mask prototype (preserved from the original model)

`yolo26_postprocess`

Field	Value
Target models	YOLO26 (models with `one2one` decoupled head)
Head structure	4-channel bbox (direct ltrb) + `C`-channel classification

© Copyright 2026 DEEPX All Rights Reserved.

Field	Value
Supported tasks	Detection
Output	`[N, K, 6]` — bbox (xyxy) + score + class_id *(note: K is the second axis, unlike `yolo_postprocess` where channels are the second axis)*

5.3.4 Identifying Conv Head Tensor Names

The `layers` field expects the **output tensor name** of each Conv head (one per detection scale). You can locate these names with Netron or programmatically:

```
import onnx
from onnx import shape_inference

model = onnx.load("model.onnx")
model = shape_inference.infer_shapes(model)

# Set this to your model's classification class count.
# Common defaults: 80 (COCO), 1 (single-class), 20 (VOC), etc.
num_classes = 80

for node in model.graph.node:
    if node.op_type == "Conv":
        out = node.output[0]
        for vi in model.graph.value_info:
            if vi.name == out:
                dims = [d.dim_value for d in vi.type.tensor_type.shape.dim]
                # 64ch = bbox (DFL), 4ch = bbox (direct),
                # num_classes-ch = cls, 32ch = mask coefficient
                if len(dims) == 4 and dims[1] in {4, 32, 64, num_classes}:
                    print(f"{node.name} -> {out}: {dims}")
```

5.3.5 Examples

YOLOv8n Detection (ultralytics export)

```
import onnx
import dx_com

model = onnx.load("yolov8n.onnx")
optimized = dx_com.pre_optimize(model, passes={
    "yolo_postprocess": {
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        "layers": [
            {
                "bbox": "/model.22/cv2.0/cv2.0.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.0/cv3.0.2/Conv_output_0",
            },
            {
                "bbox": "/model.22/cv2.1/cv2.1.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.1/cv3.1.2/Conv_output_0",
            },
            {
                "bbox": "/model.22/cv2.2/cv2.2.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.2/cv3.2.2/Conv_output_0",
            },
        ],
        "num_classes": 80,
        "topk": 300,
        "input_height": 640,
        "input_width": 640,
    },
})
# Output: [1, 84, 300]  (4 bbox_xywh + 80 cls scores, K = 300)
```

YOLOv8n Instance Segmentation (ultralytics export)

```
import onnx
import dx_com

model = onnx.load("yolov8n-seg.onnx")
optimized = dx_com.pre_optimize(model, passes={
    "yolo_postprocess": {
        "layers": [
            {
                "bbox": "/model.22/cv2.0/cv2.0.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.0/cv3.0.2/Conv_output_0",
                "mask_coeff": "/model.22/cv4.0/cv4.0.2/Conv_output_0",
            },
            {
                "bbox": "/model.22/cv2.1/cv2.1.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.1/cv3.1.2/Conv_output_0",
                "mask_coeff": "/model.22/cv4.1/cv4.1.2/Conv_output_0",
            },
            {
                "bbox": "/model.22/cv2.2/cv2.2.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.2/cv3.2.2/Conv_output_0",
                "mask_coeff": "/model.22/cv4.2/cv4.2.2/Conv_output_0",
            },
        ],
        "num_classes": 80,
        "topk": 300,
        "input_height": 640,
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        "input_width": 640,
    },
})
# Output 1: [1, 116, 300]  (4 bbox + 80 cls + 32 mask_coeff, K = 300)
# Output 2: [1, 32, 160, 160]  (mask prototype, unchanged)

dx_com.compile(
    model=optimized,
    config="yolov8n-seg.json",
    output_dir="./yolov8n_seg_optimized",
)
```

YOLO26n Detection (ultralytics export)

```
import onnx
import dx_com

model = onnx.load("yolo26n.onnx")
optimized = dx_com.pre_optimize(model, passes={
    "yolo26_postprocess": {
        "layers": [
            {
                "bbox": "/model.23/one2one_cv2.0/one2one_cv2.0.2/Conv_output_0",
                "cls_conf": "/model.23/one2one_cv3.0/one2one_cv3.0.2/Conv_output_0",
            },
            {
                "bbox": "/model.23/one2one_cv2.1/one2one_cv2.1.2/Conv_output_0",
                "cls_conf": "/model.23/one2one_cv3.1/one2one_cv3.1.2/Conv_output_0",
            },
            {
                "bbox": "/model.23/one2one_cv2.2/one2one_cv2.2.2/Conv_output_0",
                "cls_conf": "/model.23/one2one_cv3.2/one2one_cv3.2.2/Conv_output_0",
            },
        ],
        "num_classes": 80,
        "topk": 300,
        "input_height": 640,
        "input_width": 640,
    },
})
# Output: [1, 300, 6]  (bbox_xyxy + score + class_id, K = 300)
```

5.3.6 Performance Reference (DeepX DX-M1, ARM Cortex-A53)

Model	Before (FPS)	After (FPS)	Speed-up
YOLOv8n det	61	348	**5.7×**

© Copyright 2026 DEEPX All Rights Reserved.

Model	Before (FPS)	After (FPS)	Speed-up
YOLOv8n seg	59	187	**3.2×**
YOLO26n	176	315	**1.8×**

End-to-end throughput improves significantly because CPU post-processing is no longer the bottleneck on CPU-constrained hosts.

5.3.7 Migration from PPU Type 2

If you previously used `ppu.type = 2` in the JSON configuration file, switch to `dx_com.pre_optimize()`.

Previous (deprecated):

```
{
  "ppu": {
    "type": 2,
    "topk": 512,
    "num_classes": 80,
    "layer": [
      {"bbox": "bbox_head_p3", "cls_conf": "cls_head_p3"},
      {"bbox": "bbox_head_p4", "cls_conf": "cls_head_p4"},
      {"bbox": "bbox_head_p5", "cls_conf": "cls_head_p5"}
    ]
  }
}
```

New (`pre_optimize` API):

```
import onnx
import dx_com

model = onnx.load("model.onnx")
optimized = dx_com.pre_optimize(model, passes={
    "yolo_postprocess": {
        "layers": [
            {"bbox": "bbox_head_p3", "cls_conf": "cls_head_p3"},
            {"bbox": "bbox_head_p4", "cls_conf": "cls_head_p4"},
            {"bbox": "bbox_head_p5", "cls_conf": "cls_head_p5"},
        ],
        "num_classes": 80,
        "topk": 512,
        "input_height": 640,
        "input_width": 640,
    },
```


© Copyright 2026 DEEPX All Rights Reserved.

```
})
dx_com.compile(model=optimized, config="config.json", output_dir="./output")
```

Differences:

	PPU Type 2 (previous)	**`pre_optimize` (current)**
Configuration location	JSON config file	Python API
Segmentation support	❌	☑
YOLO26 support	❌	☑ (`yolo26_postprocess`)
Required parameters	(none beyond JSON fields)	`input_height`, `input_width` (per-scale stride derivation)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

5.4 Common Use Cases

This chapter provides practical, real-world scenarios with ready-to-use examples. Each use case demonstrates the implementation using the `dxcom` command and the `dx_com` Python module where applicable.

> **Additional Dependencies**
>
> Some examples in this chapter use `torchvision` for image preprocessing. Install it before running these examples:
>
> ```
> pip install torchvision
> ```

5.4.1 Use Case 1: Simple Image Classification (ResNet / MobileNet)

Scenario: Compiling a pre-trained ResNet50 or MobileNetV1 model using standard image preprocessing.

- **Option A: `dxcom` Command** – Best for quick, standard builds.
- **Option B: `dx_com` Python Module** – Best for integration into automated scripts.

Option A: `dxcom` Command

Configuration File (`resnet50_config.json`):

```
{
  "inputs": {
    "input": [1, 3, 224, 224]
  },
  "calibration_method": "ema",
  "calibration_num": 100,
  "default_loader": {
    "dataset_path": "./calibration_images",
    "file_extensions": ["jpeg", "jpg", "png"],
    "preprocessings": [
      {"resize": {"mode": "torchvision", "size": 256, "interpolation": "BILINEAR"}},
      {"centercrop": {"width": 224, "height": 224}},
      {"convertColor": {"form": "BGR2RGB"}},
      {"div": {"x": 255}},
      {"normalize": {"mean": [0.485, 0.456, 0.406], "std": [0.229, 0.224, 0.225]}}
    ]
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  }
}
```

Command:

```
dxcom \
  -m ResNet50_sim.onnx \
  -c resnet50_config.json \
  -o output/resnet50 \
  --opt_level 1
```

Option B: `dx_com` Python Module

Complete Script (`compile_resnet50.py`):

```
import dx_com
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import os

class ImageNetDataset(Dataset):
    """Standard ImageNet-style dataset"""
    def __init__(self, image_dir, img_size=224):
        self.image_dir = image_dir
        self.image_files = sorted([
            f for f in os.listdir(image_dir)
            if f.endswith(('.jpg', '.png', '.jpeg'))
        ])
        self.transform = transforms.Compose([
            transforms.Resize((img_size, img_size)),
            transforms.ToTensor(),
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        img_path = os.path.join(self.image_dir, self.image_files[idx])
        image = Image.open(img_path).convert('RGB')
        return self.transform(image)

# Setup
dataset = ImageNetDataset('./calibration_images', img_size=224)
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
# Compile
dx_com.compile(
    model="ResNet50_sim.onnx",
    output_dir="output/resnet50",
    dataloader=dataloader,
    calibration_method="ema",
    calibration_num=100,
    opt_level=1
)

print("ResNet50 compilation complete!")
```

Run:

```
python3 compile_resnet50.py
```

5.4.2 Use Case 2: Multi-Input Models (Stereo Vision)

Scenario: A stereo camera system requiring two image inputs with different dimensions.

> **dx_com Python Module Only**
>
> Multi-input models are **only supported via the `dx_com` Python module**. The `dxcom` command does not support multiple inputs.

`dx_com` Python Module

```
import dx_com
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import os

class StereoDataset(Dataset):
    """Dataset providing stereo image pairs"""
    def __init__(self, left_dir, right_dir):
        self.dirs = {"left_input": left_dir, "right_input": right_dir}

        # Get matching image pairs
        self.image_files = sorted([
            f for f in os.listdir(left_dir)
            if f.endswith(('.jpg', '.png'))
        ])
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        # Per-input preprocessing, keyed by ONNX input node name.
        # Keys MUST match the model's input node names exactly.
        self.transform = {
            "left_input": transforms.Compose([
                transforms.Resize((128, 128)),
                transforms.ToTensor(),
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406],
                    std=[0.229, 0.224, 0.225]
                )
            ]),
            "right_input": transforms.Compose([
                transforms.Resize((256, 256)),
                transforms.ToTensor(),
                transforms.Normalize(
                    mean=[0.485, 0.456, 0.406],
                    std=[0.229, 0.224, 0.225]
                )
            ]),
        }

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        filename = self.image_files[idx]

        # Build a dict keyed by ONNX input node names.
        sample = {}
        for input_name, dir_path in self.dirs.items():
            image = Image.open(
                os.path.join(dir_path, filename)
            ).convert('RGB')
            sample[input_name] = self.transform[input_name](image)
        return sample

# Setup
dataset = StereoDataset(
    left_dir='./calibration_left',
    right_dir='./calibration_right'
)
dataloader = DataLoader(dataset, batch_size=1)

# Compile multi-input model
dx_com.compile(
    model="stereo_model.onnx",
    output_dir="output/stereo",
    dataloader=dataloader,
    calibration_method="ema",
    calibration_num=50,  # Fewer samples needed for smaller models
    opt_level=1
```


© Copyright 2026 DEEPX All Rights Reserved.

```
)

print("Stereo model compilation complete!")
```

Key Technical Requirements:

- **DataLoader Output:** For multi-input models, return a `dict[str, torch.Tensor]` keyed by the ONNX input node names. The calibrator maps a dict by **name** (`run_session` matches each key to a model input), so the keys must match the input node names exactly.
- **Why dict (not tuple):** A tuple/list is also accepted, but it is mapped to inputs **by position** in the model's internal input-node order — not the order you return them. A name-keyed dict removes this ambiguity and is the recommended form for multi-input models.
- **Shape & dtype:** Each per-input tensor must match its ONNX input shape without the batch dim, and `batch_size` must equal the model's input batch (normally `1`). Tensors should be `float32`. See the **Defining Preprocessing Transforms in the DataLoader** section in Execution of DX-COM.
- **Heterogeneous Inputs:** Each input branch supports independent sizes and preprocessing configurations.

5.4.3 Use Case 3: Performance Optimization for Edge Devices

> ⚠ **Experimental Feature**
>
> `aggressive_partitioning` is currently experimental and may produce unexpected results for some models.

Scenario: Deploying on embedded systems with restricted CPU resources. The goal is to maximize NPU offloading while maintaining short compilation times.

Configuration for Aggressive Partitioning

```
{
  "inputs": {
    "input": [1, 3, 224, 224]
  },
  "calibration_method": "ema",
  "calibration_num": 50,
  "default_loader": {
    "dataset_path": "./calibration_images",
    "file_extensions": ["jpeg", "jpg", "png"],
    "preprocessings": [
      {"resize": {"width": 224, "height": 224}},
      {"normalize": {"mean": [0.485, 0.456, 0.406], "std": [0.229, 0.224, 0.225]}}
```


© Copyright 2026 DEEPX All Rights Reserved.

```
      ]
    }
}
```

You can compile using either the `dxcom` **command** or the `dx_com` **Python module**. Choose one:

Option A: `dxcom` Command

```
dxcom \
  -m efficient_model.onnx \
  -c config.json \
  -o output/efficient \
  --aggressive_partitioning \
  --opt_level 0
```

Option B: `dx_com` Python Module

```
import dx_com

# Maximize NPU offloading with aggressive partitioning
dx_com.compile(
    model="efficient_model.onnx",
    output_dir="output/efficient",
    config="config.json",
    aggressive_partitioning=True,  # Maximize NPU usage
    calibration_num=50  # Fewer samples = faster calibration
)
```

Optimization Strategy: Aggressive Partitioning:

- **Pros**: Maximum NPU offloading, significantly reduced host CPU load and faster compilation cycles.
- **Cons**: Potential for slightly higher latency compared to `opt_level 1` and increased output binary size.

5.4.4 Use Case 4: Custom Data Type (Non-Image)

Scenario: Processing non-visual data such as audio spectrograms, time-series data, or 3D point clouds.

> **dx_com Python Module Only**
>
> Non-image data types are **only supported via the `dx_com` Python module**. The `dxcom` command only supports image data.

© Copyright 2026 DEEPX All Rights Reserved.

`dx_com` Python Module

```
import dx_com
import torch
from torch.utils.data import Dataset, DataLoader
import numpy as np
import os

class CustomDataDataset(Dataset):
    """Example: Audio spectrogram dataset"""
    def __init__(self, data_dir, input_shape=(1, 64, 128)):
        self.data_files = sorted([
            f for f in os.listdir(data_dir)
            if f.endswith('.npy')
        ])
        self.data_dir = data_dir
        self.input_shape = input_shape

    def __len__(self):
        return len(self.data_files)

    def __getitem__(self, idx):
        # Load numpy array (e.g., audio spectrogram)
        data = np.load(os.path.join(self.data_dir, self.data_files[idx]))

        # Normalize to [0, 1]
        data = (data - data.min()) / (data.max() - data.min() + 1e-8)

        # Convert to tensor with correct shape
        return torch.from_numpy(data.astype(np.float32)).unsqueeze(0)

# Setup
dataset = CustomDataDataset('./spectrogram_data', input_shape=(1, 64, 128))
dataloader = DataLoader(dataset, batch_size=1)

# Compile
dx_com.compile(
    model="audio_model.onnx",
    output_dir="output/audio_model",
    dataloader=dataloader,
    calibration_method="minmax",  # minmax better for non-image data
    calibration_num=100,
)
```

5.4.5 Use Case 5: Enhanced Quantization (DXQ)

Scenario: Improving quantization accuracy for models where standard quantization causes unacceptable accuracy degradation.

© Copyright 2026 DEEPX All Rights Reserved.

Version Support

DXQ (`enhanced_scheme`) is supported in **DX-COM v2.1.0 and later**.

- **Option A: `dxcom` Command** – Set `enhanced_scheme` in the JSON config file.
- **Option B: `dx_com` Python Module** – Pass `enhanced_scheme` as a parameter directly.

Option A: `dxcom` Command

Configuration File (`config.json`):

```
{
  "inputs": {
    "input": [1, 3, 224, 224]
  },
  "calibration_method": "ema",
  "calibration_num": 100,
  "enhanced_scheme": {
    "DXQ-P3": {"num_samples": 1024}
  },
  "default_loader": {
    "dataset_path": "./calibration_images",
    "file_extensions": ["jpeg", "jpg", "png"],
    "preprocessings": [
      {"resize": {"width": 224, "height": 224}},
      {"normalize": {"mean": [0.485, 0.456, 0.406], "std": [0.229, 0.224, 0.225]}}
    ]
  }
}
```

Command:

```
dxcom \
  -m large_model.onnx \
  -c config.json \
  -o output/large_model_dxq \
  --opt_level 1
```

Option B: `dx_com` Python Module

```
import dx_com

dx_com.compile(
    model="large_model.onnx",
    output_dir="output/large_model_dxq",
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    config="config.json",
    enhanced_scheme={
        "DXQ-P3": {"num_samples": 1024}
    },
    opt_level=1,
    calibration_num=100
)
```

> **GPU Device Selection**
>
> By default, DX-COM automatically uses GPU if available. In multi-GPU environments, you can specify a device via `quantization_device` in the JSON config or the `dx_com.compile()` parameter (e.g., `"cuda:1"`). See the **Quantization Device** section in JSON File Configuration for details.

For all available DXQ schemes (DXQ-P0 to DXQ-P5) and their parameters, see the **Enhanced Quantization Scheme (DXQ)** section in JSON File Configuration.

Automatic Q-PRO (`use_q_pro`)

If you don't want to hand-pick a DXQ scheme, let DX-COM choose for you. Setting `use_q_pro` enables the **automatic Q-PRO pipeline**: the compiler generates DXQ combinations and applies the optimal quantization enhancement stages based on model structure and compile-time metrics — no manual `enhanced_scheme` tuning required.

> **Version Support**
>
> Automatic Q-PRO (`use_q_pro`) is available in **DX-COM v2.4.0 and later**. It is **mutually exclusive** with manual `enhanced_scheme` .

Option A: `dxcom` Command

```
dxcom \
  -m large_model.onnx \
  -c config.json \
  -o output/large_model_qpro \
  --use_q_pro \
  --opt_level 1
```

Option B: `dx_com` Python Module

```
import dx_com
```


© Copyright 2026 DEEPX All Rights Reserved.

```
dx_com.compile(
    model="large_model.onnx",
    output_dir="output/large_model_qpro",
    config="config.json",
    use_q_pro=True,   # automatic DXQ selection (do not combine with enhanced_scheme)
    opt_level=1,
    calibration_num=100,
)
```

Automatic vs Manual

Start with `use_q_pro=True` for the easiest path to higher-accuracy quantization. Switch to a manual `enhanced_scheme` (e.g., `{"DXQ-P3": {...}}`) only when you need precise control over a specific DXQ scheme. See the **Automatic Q-PRO (`use_q_pro`)** section in Execution of DX-COM.

5.4.6 Use Case 6: Diagnose and Re-quantize Without Recompile

Scenario: A model compiled successfully, but quantization accuracy is lower than expected. You want to (1) find out *which* regions are responsible and (2) iterate on quantization settings **without paying the full compile cost each time**.

This use case chains three v2.4.0 features into one tuning loop:

1. **`quant_diagnosis`** — produces an HTML report of per-region quantization quality **plus** a reusable `.qxnn` checkpoint.
2. **QXNN Resume** — re-runs quantization from that `.qxnn` checkpoint, skipping the earlier compile phases.
3. **Re-calibration / Q-PRO** — applies a different calibration method or enables Q-PRO during the resume to improve accuracy.

Version Support

`quant_diagnosis` and QXNN Resume are available in **DX-COM v2.4.0 and later**. For the full workflow reference, see Quantization Tuning Workflow.

Step 1 — Compile with diagnosis enabled

Enabling `quant_diagnosis` writes both the HTML report and the `.qxnn` resume artifact under `quant_diagnosis/` in the output directory.

© Copyright 2026 DEEPX All Rights Reserved.

Option A: `dxcom` Command

```
dxcom \
  -m large_model.onnx \
  -c config.json \
  -o output/large_model \
  --quant_diagnosis
```

Option B: `dx_com` Python Module

```
import dx_com

dx_com.compile(
    model="large_model.onnx",
    output_dir="output/large_model",
    config="config.json",
    quant_diagnosis=True,
)
```

This produces:

- `output/large_model/quant_diagnosis/large_model.qxnn` — resume checkpoint
- `output/large_model/quant_diagnosis/diagnosis_report.html` — per-region report with ready-to-paste retry snippets

Open the HTML report to identify high-severity regions and the recommended settings.

Step 2 — Resume from the checkpoint with new settings

Instead of recompiling from the ONNX model, point `dxcom` / `dx_com` at the `.qxnn` checkpoint. The earlier compile phases are skipped and only quantization re-runs.

Re-calibrate with a different observer:

```
dxcom \
  --checkpoint output/large_model/quant_diagnosis/large_model.qxnn \
  -o output/large_model_iqr \
  --recalibration_method iqr
```

Or enable automatic Q-PRO during the resume:

```
dxcom \
  --checkpoint output/large_model/quant_diagnosis/large_model.qxnn \
  -o output/large_model_qpro \
  --use_q_pro
```


© Copyright 2026 DEEPX All Rights Reserved.

Python equivalent:

```
import dx_com

# Re-quantize from the checkpoint — no recompile, no model/config needed
dx_com.compile(
    checkpoint="output/large_model/quant_diagnosis/large_model.qxnn",
    output_dir="output/large_model_iqr",
    recalibration_method="iqr",    # or: use_q_pro=True
)
```

Resume-only options

`--recalibration_method`, `--enhanced_scheme`, and `--dataset_path` are valid **only** in QXNN resume mode (i.e., with `--checkpoint`). `--checkpoint` and `-m/--model_path` are mutually exclusive. On resume, the calibration DataLoader is auto-built from the config embedded in the `.qxnn` — no `-c/--config_path` is required.

Why this matters

Because QXNN Resume skips the compile phases that don't depend on quantization, you can try several calibration methods or Q-PRO settings in quick succession, dramatically shortening the accuracy-tuning loop.

5.4.7 Use Case 7: YOLO Post-Processing Optimization (CPU-Constrained Edge)

Scenario: Deploying a YOLO-family detection or instance-segmentation model on a CPU-constrained edge host (for example, an ARM Cortex-A53) where CPU-side post-processing (Sigmoid, DFL decoding, dist2bbox over the full anchor grid) becomes the end-to-end throughput bottleneck.

Approach: Apply `dx_com.pre_optimize()` before `dx_com.compile()`. The API rewrites the post-processing graph so that TopK selection happens first, and expensive operations are applied only to the `K` selected candidates (default 300) instead of the full anchor grid.

```
import onnx
import dx_com

model = onnx.load("yolov8n.onnx")
optimized = dx_com.pre_optimize(model, passes={
    "yolo_postprocess": {
        "layers": [
```


© Copyright 2026 DEEPX All Rights Reserved.

```
            {
                "bbox": "/model.22/cv2.0/cv2.0.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.0/cv3.0.2/Conv_output_0",
            },
            {
                "bbox": "/model.22/cv2.1/cv2.1.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.1/cv3.1.2/Conv_output_0",
            },
            {
                "bbox": "/model.22/cv2.2/cv2.2.2/Conv_output_0",
                "cls_conf": "/model.22/cv3.2/cv3.2.2/Conv_output_0",
            },
        ],
        "num_classes": 80,
        "topk": 300,
        "input_height": 640,
        "input_width": 640,
    },
})

dx_com.compile(
    model=optimized,
    config="yolov8n.json",
    output_dir="./yolov8n_optimized",
)
```

For supported model families (YOLOv8 / YOLOv9 / YOLOv11 / YOLOv12 / YOLOv13 via the `yolo_postprocess` pass, and YOLO26 via the `yolo26_postprocess` pass), output shapes, instance-segmentation usage, and the migration recipe from the deprecated `ppu.type = 2`, see Pre-Optimize API.

5.4.8 Use Case 8: PPU Hardware Acceleration (YOLO)

Scenario: Offloading object-detection post-processing (confidence filtering, class prediction) to the on-NPU Post-Processing Unit for YOLO-family models, reducing host CPU load.

- **Option A: `dxcom` Command** – Set the `ppu` block in the JSON config file.
- **Option B: `dx_com` Python Module** – Pass a `PPUConfig` object via `ppu_config`.

Both paths configure the same hardware; the Python `PPUConfig` is the programmatic equivalent of the JSON `ppu` section. For node-name identification and the full parameter reference, see the **PPU Configuration** section in JSON File Configuration.

© Copyright 2026 DEEPX All Rights Reserved.

> ⚠ **NMS still runs on the host CPU**
>
> The PPU accelerates filtering and class prediction only. Non-Maximum Suppression (NMS) must still be executed on the host CPU using the filtered outputs.

Option A: `dxcom` Command

Configuration File (`yolov8_config.json`):

```
{
  "inputs": {
    "images": [1, 3, 640, 640]
  },
  "calibration_method": "ema",
  "calibration_num": 100,
  "default_loader": {
    "dataset_path": "./calibration_images",
    "file_extensions": ["jpeg", "jpg", "png"],
    "preprocessings": [
      {"convertColor": {"form": "BGR2RGB"}},
      {"resize": {"width": 640, "height": 640}},
      {"div": {"x": 255}}
    ]
  },
  "ppu": {
    "type": 1,
    "conf_thres": 0.25,
    "num_classes": 80,
    "layer": [
      {"bbox": "Mul_441", "cls_conf": "Sigmoid_442"}
    ]
  }
}
```

```
dxcom -m yolov8.onnx -c yolov8_config.json -o output/yolov8
```

Option B: `dx_com` Python Module

```
import dx_com
from dx_com import PPUConfig, PPUTypes

# Anchor-free YOLOv8: build the PPU configuration
ppu_config = (
    PPUConfig()
    .set_type(PPUTypes.YOLO_ANCHORFREE)
    .set_num_classes(80)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    .set_conf_thres(0.25)
)
ppu_config.add_layer(bbox="Mul_441", cls_conf="Sigmoid_442")

dx_com.compile(
    model="yolov8.onnx",
    output_dir="output/yolov8",
    config="yolov8_config.json",  # or dataloader=...
    ppu_config=ppu_config,
)

print("YOLOv8 PPU-accelerated compilation complete!")
```

Anchor-based (YOLOv5/v7) example — type 0 uses a dict-style `layer` with `num_anchors` and requires `activation`:

```
from dx_com import PPUConfig, PPUTypes

ppu_config = PPUConfig(
    type=PPUTypes.YOLO_BASE,
    conf_thres=0.25,
    activation="Sigmoid",
    num_classes=80,
    layer={
        "Conv_245": {"num_anchors": 3},
        "Conv_294": {"num_anchors": 3},
        "Conv_343": {"num_anchors": 3},
    },
)
```

Selecting the PPU type

Match the `PPUTypes` value to your model architecture (anchor-based → `YOLO_BASE`, anchor-free → `YOLO_ANCHORFREE`, DFL-based CPU TopK → `YOLOV8`). See the **type/model table** section in JSON File Configuration and the **PPUConfig API reference** section in Execution of DX-COM.

© Copyright 2026 DEEPX All Rights Reserved.

6. Development Tools

6.1 Model Viewer_DX-TRON

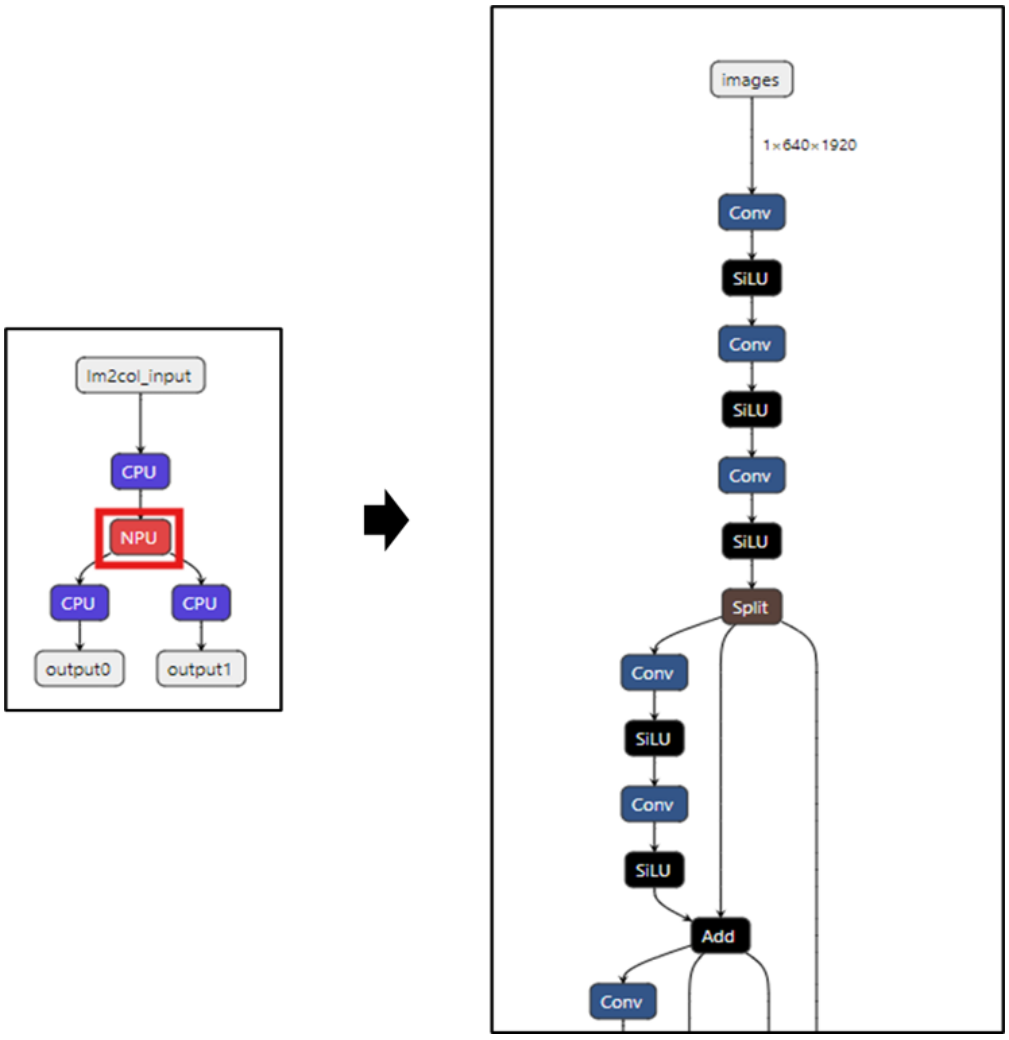
6.1.1 Overview

DX-TRON is a graphical visualization tool for exploring .dxnn model files compiled with the DEEPX toolchain.
It allows users to load and inspect model structures, view workload distribution between NPU and CPU through color-coded graphs. With **DX-TRON**, users can better understand model execution flow and improve overall performance.

Key Features

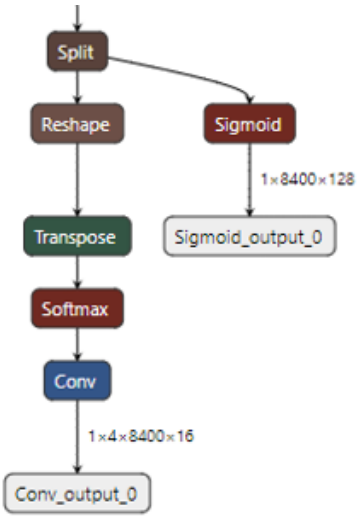
- **Support for .dxnn Files**: Load and visualize model files compiled with the DEEPX toolchain.
- **Visual Workload Representation**: Displays a color-coded breakdown of workload execution:
  - Red: Operations executed on the NPU
  - Purple: Operations executed on the CPU or host
- **Interactive Node Inspection** : Double-click any node within the graph to view detailed information about the associated operations.

ⓒ Copyright 2026 DEEPX All Rights Reserved.



- **Model Navigation Controls**: Use the backward arrow in the bottom-left corner to return to the model overview screen at any time.

ⓒ Copyright 2026 DEEPX All Rights Reserved.





6.1.2 Download

DX-TRON can be downloaded from the **DEEPX Developers' Portal**:
https://developer.deepx.ai

> **NOTE**
>
> Account registration and authentication are required to download DX-TRON packages.

For detailed installation instructions using the automated installation script, refer to the dx-all-suite installation guide.

6.1.3 Installation on Linux

Supported Systems

Operating System	Versions	Architecture	Supported Mode
Ubuntu	20.04, 22.04, 24.04, 26.04	x86_64 (amd64), ARM64 (aarch64)	CLI/Desktop + Web
Debian	11, 12, 13		CLI/Desktop + Web

© Copyright 2026 DEEPX All Rights Reserved.

Operating System	Versions	Architecture	Supported Mode
		x86_64 (amd64), ARM64 (aarch64)	
Fedora	42, 43, 44, 45	x86_64 (amd64), ARM64 (aarch64)	Web only
RHEL	9, 10	x86_64 (amd64), ARM64 (aarch64)	Web only
CentOS	9, 10	x86_64 (amd64), ARM64 (aarch64)	Web only

> **Note:** On the Red Hat family (Fedora / RHEL / CentOS), only the DX-TRON web variant is installed. The `dxtron` CLI/desktop (AppImage) binary is supported on the Debian/Ubuntu family only. Launch the web server with `./run_dxtron_web.sh --port=8080`.

DX-TRON can be installed using one of the following methods:

Method 1: Automated Installation using install.sh (Recommended)

The easiest way to install DX-TRON is using the automated installation script. This script will download the package from DEEPX Portal, extract files to the `dx_tron/` directory, and install the Debian package automatically.

```
./install.sh --target=dx_tron --username=<your_email> --password=<your_password>
```

Method 2: Manual DEB Package Installation

If you have already downloaded the DX-TRON package, you can manually install the Debian package.

Step 1. Extract the Package

```
tar xfz dxtron_x.y.z.tar.gz
```

This will extract the following files: - `dxtron_x.y.z_amd64.deb` / `dxtron_x.y.z_arm64.deb` - `dxtron_x.y.z_web/` - `dxtron_x.y.z_amd64.AppImage` - `dxtron_x.y.z_setup.exe`

© Copyright 2026 DEEPX All Rights Reserved.

Step 2. Check Your System Architecture

```
dpkg --print-architecture  # Returns: amd64 or arm64
```

Step 3. Install the DEB Package

```
# For x86_64 (amd64)
sudo apt-get install ./dxtron_x.y.z_amd64.deb

# For ARM64 (aarch64)
sudo apt-get install ./dxtron_x.y.z_arm64.deb
```

> **NOTE**
>
> The DEB package automatically handles all required dependencies. No additional package installation is needed.

Method 3: AppImage (Portable)

AppImage is a portable executable that does not require system installation.

Step 1. Install Required Packages

DX-TRON AppImage requires several system libraries. Run the following command to install dependencies:

```
./scripts/install_prerequisites.sh
```

> **NOTE**
>
> `libfuse2` is mandatory for AppImage execution and may not be installed by default on Ubuntu 22.04 or later.

Step 2. Prepare the AppImage File

Make the AppImage executable:

```
chmod +x dxtron_x.y.z_amd64.AppImage
```


© Copyright 2026 DEEPX All Rights Reserved.

Step 3. Run DX-TRON

Execute the AppImage:

```
./dxtron_x.y.z_amd64.AppImage
```

Or use the convenience script:

```
./run_dxtron_appimage.sh
```



Once launched, a GUI window will appear, enabling DXNN model visualization.

> ⚠ **Warning**
>
> Running DX-TRON with `sudo` may require the `--no-sandbox` flag, but this is **not recommended** for security reasons.

6.1.4 Installation on Windows

This section explains how to install and launch DX-TRON on Windows using the provided setup file.

Step 1. Installation File

- File Name: `DXTron Setup x.y.z.exe`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Step 2. Installation Steps
1) Save the provided .exe file to your PC.
2) Double-click the file to launch the installer.
3) When the User Account Control (UAC) prompt appears, click **Yes**.
4) Follow the on-screen instructions in the installation wizard.
5) After installation, a DX-TRON shortcut is created in the Start Menu and optionally on the Desktop.

The default installation path is as follows.

```
C:\Users\YourUsername\AppData\Local\Programs\DXTron\
```

Step 3. Launching DX-TRON
- Search for DX-TRON in the Start Menu and run it, or
- Double-click the Desktop shortcut (if created).

6.1.5 Running DX-TRON

DX-TRON can be run in two modes: **Desktop GUI Mode** and **Web Server Mode**.

Desktop GUI Mode

After DEB Package Installation (Linux)

Run DX-TRON using the `dxtron` command:

```
dxtron
```

Using AppImage (Linux)

Execute the AppImage file directly:

```
./dxtron_x.y.z_amd64.AppImage
```

Or use the convenience script:

```
./run_dxtron_appimage.sh
```

On Windows

Launch DX-TRON from the Start Menu or Desktop shortcut.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Web Server Mode (Linux Only)

DX-TRON can be run as a local web server, allowing you to access the model viewer through a web browser.

Basic Usage

```
./run_dxtron_web.sh
```

Then open your browser and navigate to: `http://localhost:8080`

Using a Different Port

```
./run_dxtron_web.sh --port=3000
```

Then access: `http://localhost:3000`

> **NOTE**
>
> The web server mode requires the `dxtron_x.y.z_web/` directory, which is created after running `install.sh` or extracting the downloaded package.

For additional options, run:

```
./run_dxtron_web.sh --help
```

6.1.6 Troubleshooting

Step 1. Security Warnings on Windows
If Microsoft Defender SmartScreen blocks the installer:
1) Click More Info.
2) Select Run Anyway to continue installation.

Step 2. Program Fails to Launch
Perform the following checks in order.
1) Check AppImage Permission (Linux)
- Ensure the file is executable.

```
chmod +x dxtron_x.y.z_amd64.AppImage
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

2) Verify Required Libraries (Linux)
- Confirm all dependencies are installed.

3) Reinstall DX-TRON
- Uninstall and reinstall DX-TRON to ensure all files are properly installed.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

6.2 Compilation Summary Report

6.2.1 Overview

The **Compilation Summary Report** is a self-contained HTML report that summarizes the result of a DX-COM compilation. When enabled, DX-COM writes a single `<model_name>_summary.html` file into the output directory after the `.dxnn` file is generated.

Because the report embeds all of its assets, it can be opened directly in any web browser and shared as a single file — without copying the rest of the output directory. It provides a quick, install-free view of the compiled model, similar to the DX-TRON Model Viewer.

Key Features

- **Self-contained HTML**: A single file with all styles and scripts embedded; open it in any browser.
- **Model overview**: Model name, original file name, total parameter count, and original ONNX vs compiled `.dxnn` file sizes.
- **Input/Output tensors**: Names and shapes of the model inputs and outputs (using the compiled graph's static shapes when available).
- **Compilation settings**: Optimization level, aggressive partitioning, calibration method and sample count, quantization scheme, target NPU device, and compiler version.
- **Interactive graph viewer**: Pan/zoom visualization of the compiled graph with NPU vs CPU workload distribution and per-partition CPU-fallback reasons.

6.2.2 Generating the Report

The report is generated as part of compilation by adding the `--export_html` option (CLI) or `export_html=True` (Python API). It is disabled by default.

CLI

```
dxcom \
-m sample/MobilenetV1.onnx \
-c sample/MobilenetV1.json \
-o output/mobilenetv1 \
--export_html
```


© Copyright 2026 DEEPX All Rights Reserved.

Python Module

```
import dx_com

dx_com.compile(
    model="model.onnx",
    output_dir="./compiled",
    config="config.json",
    export_html=True,
)
```

> **NOTE**
>
> `--export_html` / `export_html=True` is also documented as a compilation option in Execution of DX-COM.

6.2.3 Output

On success, the report is written to the output directory and the compiler prints a confirmation line:

```
[dx_com] Summary HTML saved to: output/mobilenetv1/MobilenetV1_summary.html
```

Item	Detail
File name	`<model_name>_summary.html` (derived from the input ONNX file name)
Location	The compilation `output_dir`
Format	Self-contained HTML (no external files required)

Open the file in any web browser to view the report.

6.2.4 Requirements

The report renderer relies on the `jinja2` library, which is included as a built-in dependency of DX-COM and is normally already available. No additional setup is required for a standard installation.

© Copyright 2026 DEEPX All Rights Reserved.

6.2.5 Troubleshooting

> **Compilation is never blocked by the report**
>
> Generating the HTML report is a best-effort, post-compilation step. If the report cannot be produced, DX-COM emits a warning and the compilation still completes successfully — the `.dxnn` output is unaffected.

- **No HTML file is produced**: Confirm that `--export_html` (or `export_html=True`) was specified, and check the compilation logs for a warning indicating the report was skipped or failed.
- **Report opens but the graph is empty**: Ensure the compilation produced a valid `.dxnn` file in the same `output_dir`; the graph viewer uses the compiled graph.

© Copyright 2026 DEEPX All Rights Reserved.

7. DEEPX Agent-Driven Development for DX-COM - dx-agent-dev (Beta)

> **Beta Feature** — Agent-Driven development support is under active development. Skill definitions and routing behavior may change between releases.

7.1 Introduction

Compile ONNX models into DXNN format using natural language instructions. AI coding agents understand the DX-COM compilation pipeline — config.json generation, calibration data preparation, quantization strategies, and PPU configuration — so you can describe *what* you want and let the agent handle the implementation details.

Supported workflows include:

- PyTorch model export to ONNX with automatic opset and shape configuration
- ONNX to DXNN compilation with INT8 quantization via DX-COM CLI or Python API
- End-to-end pipeline: PT → ONNX → DXNN with calibration and validation

7.2 Prerequisites

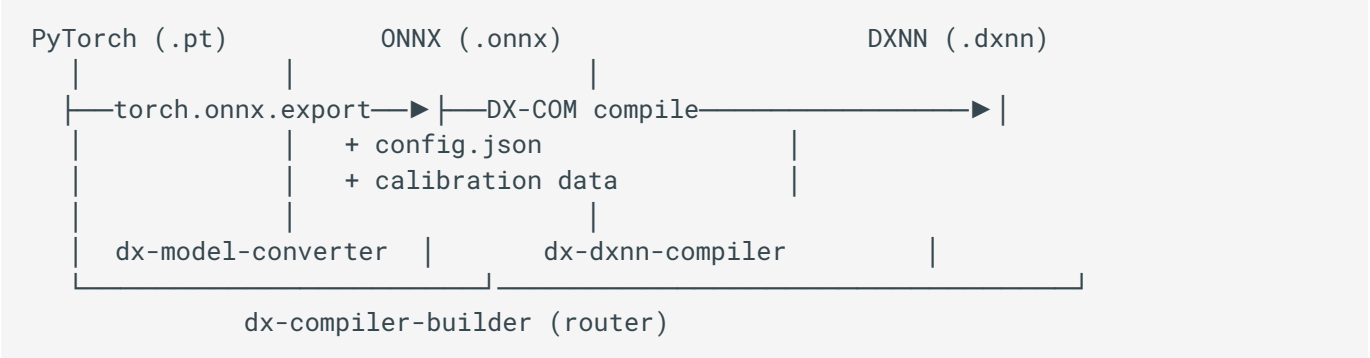
Requirement	Details
DX-COM	Installed via DX-COM installation guide (`pip install dx-com` or DEEPX package repository)
AI coding agent (one of)	Claude Code, GitHub Copilot (VS Code), Cursor, or OpenCode
Python	3.8–3.14
ONNX	opset 11–21
OS	Debian Linux (Ubuntu 20.04/22.04/24.04/26.04), x86_64

© Copyright 2026 DEEPX All Rights Reserved.

7.3 Architecture Overview

The agent-driven knowledge base for dx-compiler is organized in a single `.deepx/` directory containing agents, skills, instructions, toolsets, and memory files that the agent reads at task time.

7.3.1 Compilation Pipeline



7.3.2 Key Constraints

- Batch size must be 1
- Input shapes must be static (no dynamic axes)
- ONNX opset: 11–21
- Multi-input models: Python API only (not CLI)

> **Note:** Target device is always DX-M1 — agents should not ask about hardware selection.

7.4 Available Agents and Skills

7.4.1 Agents

Agent	Description
`@dx-compiler-builder`	Master router — classifies tasks and routes to specialist agents
`@dx-model-converter`	Converts PyTorch models to ONNX format
`@dx-dxnn-compiler`	Compiles ONNX models to DXNN using DX-COM

© Copyright 2026 DEEPX All Rights Reserved.

7.4.2 Skills (All Platforms)

General SWE Process

Skill	Description
`/dx-swe-brainstorm`	Brainstorm and plan before any compilation task
`/dx-swe-tdd`	Test-driven development — validate each step incrementally
`/dx-swe-verify`	Verify before claiming completion — evidence before assertions
`/dx-swe-writing-plans`	Write implementation plans from specs or requirements
`/dx-swe-executing-plans`	Execute implementation plans with review checkpoints
`/dx-swe-debugging`	Systematic debugging before proposing fixes
`/dx-swe-parallel-agents`	Dispatch independent tasks to parallel agents
`/dx-swe-subagent-dev`	Execute plans with independent sub-agents
`/dx-swe-receiving-review`	Receive and process code review feedback
`/dx-swe-requesting-review`	Request code review before merging
`/dx-skill-router`	Route tasks to appropriate skills

DEEPX Build

Skill	Description
`/dx-agent-compiler-compile`	Step-by-step ONNX to DXNN compilation workflow
`/dx-agent-compiler-convert`	Step-by-step PyTorch to ONNX conversion workflow
`/dx-agent-compiler-validate`	Validate compiled .dxnn model output

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.5 Supported AI Tools

Agent-Driven development works with five AI coding tools. Each tool auto-loads the `.deepx/` knowledge base through its own configuration mechanism.

Tool	Type	Auto-Load Mechanism	Agent Invocation
Claude Code	CLI	`CLAUDE.md` at project root	Free-form conversation; Context Routing Table dispatches automatically
GitHub Copilot	VS Code	`.github/copilot-instructions.md`	`@dx-compiler-builder "prompt"` in Copilot Chat
Cursor	IDE	`.cursor/rules/` (19 files: `dx-compiler.mdc`, 3 agent `.mdc` files, 15 `skill-*.mdc` files)	Free-form conversation; rules loaded by `alwaysApply`
OpenCode	CLI	`AGENTS.md` + `opencode.json`	`@dx-compiler-builder "prompt"` or `/dx-agent-compiler-compile`
Codex CLI	CLI	`AGENTS.md` + `.codex/skills/dx-codex-identity/SKILL.md`	Free-form conversation via `codex` / `codex exec`; inspect `.deepx/skills/*/SKILL.md` directly when needed

7.5.1 First-Time Setup

No additional configuration is needed. Open the `dx-compiler/` directory in your preferred tool and the configuration files are loaded automatically:

```
# Claude Code
cd dx-all-suite/dx-compiler
claude

# OpenCode
cd dx-all-suite/dx-compiler
opencode
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# Codex CLI
cd dx-all-suite/dx-compiler
codex

# GitHub Copilot — open folder in VS Code
code dx-all-suite/dx-compiler

# Cursor — open folder in Cursor
cursor dx-all-suite/dx-compiler
```

7.5.2 Platform File Loading Reference

Each AI coding agent auto-loads different configuration files at the dx-compiler level.

Auto-Loaded Files

File	Auto-loaded by	Loading
`.github/copilot-instructions.md`	Copilot Chat/CLI	Auto
`CLAUDE.md`	Claude Code	Auto
`AGENTS.md` + `opencode.json`	OpenCode	Auto
`.cursor/rules/` (19 files)	Cursor	Auto
`AGENTS.md` + `.codex/skills/dx-codex-identity/SKILL.md`	Codex CLI	Auto

Agent Files (Manual @mention)

Agent	Copilot (`@mention`)	Claude Code (`@mention`)	OpenCode (`@mention`)
`dx-compiler-builder`	`.github/agents/dx-compiler-builder.agent.md`	`.claude/agents/dx-compiler-builder.md`	`.opencode/agents/dx-compiler-builder.md`
`dx-dxnn-compiler`	`.github/agents/dx-dxnn-compiler.agent.md`	`.claude/agents/dx-dxnn-compiler.md`	`.opencode/agents/dx-dxnn-compiler.md`

© Copyright 2026 DEEPX All Rights Reserved.

Agent	Copilot (`@mention`)	Claude Code (`@mention`)	OpenCode (`@mention`)
`dx-model-converter`	`.github/agents/dx-model-converter.agent.md`	`.claude/agents/dx-model-converter.md`	`.opencode/agents/dx-model-converter.md`

> **Codex CLI note:** Codex does not use `@mention` agent wrappers here. It reads `AGENTS.md` automatically and then consults `.deepx/agents/*.md` or `.deepx/skills/*/SKILL.md` directly for deeper grounding.

Skill Files (All Platforms)

Skills exist across all platforms:

- `.deepx/skills/` — canonical definitions (15 skills)
- `.github/skills/` — Copilot inline copies
- `.claude/skills/` — Claude thin wrappers
- `.opencode/agents/` — OpenCode, via skill references
- `.cursor/rules/skill-*.mdc` — Cursor rules
- `.codex/skills/dx-codex-identity/` — Codex auto-loaded identity wrapper; task-specific skills are read from `.deepx/skills/` directly

Skill	File
`/dx-swe-brainstorm`	`.deepx/skills/dx-swe-brainstorm/SKILL.md`
`/dx-agent-compiler-compile`	`.deepx/skills/dx-agent-compiler-compile/SKILL.md`
`/dx-agent-compiler-convert`	`.deepx/skills/dx-agent-compiler-convert/SKILL.md`
`/dx-swe-parallel-agents`	`.deepx/skills/dx-swe-parallel-agents/SKILL.md`
`/dx-swe-executing-plans`	`.deepx/skills/dx-swe-executing-plans/SKILL.md`
`/dx-swe-receiving-review`	`.deepx/skills/dx-swe-receiving-review/SKILL.md`

© Copyright 2026 DEEPX All Rights Reserved.

Skill	File
`/dx-swe-requesting-review`	`.deepx/skills/dx-swe-requesting-review/SKILL.md`
`/dx-skill-router`	`.deepx/skills/dx-skill-router/SKILL.md`
`/dx-swe-subagent-dev`	`.deepx/skills/dx-swe-subagent-dev/SKILL.md`
`/dx-swe-debugging`	`.deepx/skills/dx-swe-debugging/SKILL.md`
`/dx-swe-tdd`	`.deepx/skills/dx-swe-tdd/SKILL.md`
`/dx-agent-compiler-validate`	`.deepx/skills/dx-agent-compiler-validate/SKILL.md`
`/dx-swe-verify`	`.deepx/skills/dx-swe-verify/SKILL.md`
`/dx-swe-writing-plans`	`.deepx/skills/dx-swe-writing-plans/SKILL.md`

Shared Knowledge Base (`.deepx/`)

The `.deepx/` directory is the **canonical source** for all platform-specific files. The `dx-agent-gen` generator transforms agents, skills, and templates into platform-specific files for Copilot (`.github/`), Claude Code (`.claude/`), OpenCode (`.opencode/`), and Cursor (`.cursor/rules/`). Runtime knowledge files (memory, instructions, toolsets) are read on demand by agents during task execution.

Directory	Files	Description
`.deepx/agents/`	`dx-compiler-builder.md` , `dx-dxnn-compiler.md` , `dx-model-converter.md`	Authoritative agent definitions — `dx-agent-gen` generates platform copies to `.github/agents/` , `.claude/agents/` , `.opencode/agents/` , and `.cursor/rules/`
`.deepx/templates/`	`{en,ko}/*.tmpl`	Instruction file templates (fragments via parent traversal to suite root)
`.deepx/toolsets/`	`dxcom-api.md` , `dxcom-cli.md` , `config-schema.md`	API and CLI reference
`.deepx/instructions/`		

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Directory	Files	Description
	`coding-standards.md`, `compilation-workflow.md`	Coding conventions and workflow rules
`.deepx/memory/`	`common_pitfalls.md`, `MEMORY.md`	Persistent knowledge — pitfalls and session memory

Generation Pipeline

All platform-specific files (`.github/` , `.claude/` , `.opencode/` , `.cursor/rules/`) are generated from `.deepx/` by `dx-agent-gen` :

```
dx-agent-gen generate --repo dx-compiler
```

A pre-commit hook enforces no drift between `.deepx/` sources and generated platform files. **Platform files should never be edited directly** — always edit the canonical `.deepx/` source and re-run the generator.

7.6 User Scenarios

7.6.1 Mandatory Brainstorming Questions

Before any compilation task, the agent asks three mandatory questions to ensure correct configuration. These questions cannot be skipped.

Q1: NMS-Free Model Detection (PT → ONNX tasks)

For YOLO models, the agent auto-detects NMS-free capability and presents a YOLO version characteristics table showing anchor type, NMS-free support, and PPU type for each version (v3–v26). The recommended export mode depends on the model's NMS-free architecture:

- **NMS-free models** (YOLOv10, YOLO26): **end2end=True** (recommended) — native NMS-free output `[1, 300, 6]` , no postprocessing needed. These models use one-to-one matching natively.
- **Optional NMS-free models** (YOLOv8, v9, v11, v12): **end2end=False** (recommended, default) — fused output `[1, 84, 8400]` , requires NMS postprocessing but gives full control over NMS parameters.

© Copyright 2026 DEEPX All Rights Reserved.

Q2: ONNX Simplification

Default is OFF. The agent presents pros (graph cleanup, reduced model size) and cons (numerical precision loss, debugging difficulty, model breakage risk, input name changes). The user confirms whether to run `onnx-simplifier` after export.

Q3: PPU Compilation Support (ONNX → DXNN tasks)

For detection models, the agent auto-detects PPU eligibility and presents the trade-off:

- **Without PPU** (default) — full control over NMS parameters at inference time
- **With PPU** — post-processing runs on hardware, simpler deployment

If PPU is enabled, the agent auto-infers PPU type from the model family (type 0 for anchor-based YOLOv3–v7, type 1 for anchor-free YOLOv8+).

7.6.2 Scenario 1: Convert PyTorch Model to ONNX

Prompt:

```
"Convert my yolo26x-custom.pt to ONNX with opset 17 and input shape [1, 3, 640, 640]"
```

Tool	How to Use
Claude Code	Open `dx-compiler/` and type the prompt directly.
GitHub Copilot	`@dx-model-converter` followed by the prompt.
Cursor	Open `dx-compiler/` and type the prompt.
OpenCode	`/dx-agent-compiler-convert` or `@dx-model-converter` followed by the prompt.

7.6.3 Scenario 2: Compile ONNX to DXNN

Prompt:

ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
"Compile model.onnx to DXNN with INT8 quantization using EMA calibration with 200
samples from ./calibration_images/"
```

Tool	How to Use
Claude Code	Open `dx-compiler/` and type the prompt directly.
GitHub Copilot	`@dx-dxnn-compiler` followed by the prompt.
Cursor	Open `dx-compiler/` and type the prompt.
OpenCode	`/dx-agent-compiler-compile` or `@dx-dxnn-compiler` followed by the prompt.

7.6.4 Scenario 3: Full Pipeline PT → DXNN

Prompt:

```
"Convert my yolo26x-custom.pt to DXNN for DX-M1"
```

Tool	How to Use
Claude Code	Open `dx-compiler/` and type the prompt. The router agent orchestrates both conversion and compilation.
GitHub Copilot	`@dx-compiler-builder` followed by the prompt.
Cursor	Open `dx-compiler/` and type the prompt.
OpenCode	`@dx-compiler-builder` followed by the prompt.

7.7 Config Auto-Inference

When you provide a model and calibration data, the agent automatically infers:

Field	Auto-Inferred From
`inputs` shape	ONNX model metadata (`onnx.load()` → `graph.input`)
`calibration_method`	Default `ema` (recommended for most models)

© Copyright 2026 DEEPX All Rights Reserved.

Field	Auto-Inferred From
`preprocessings.resize`	Input shape dimensions (H, W)
`preprocessings.normalize`	Model family (ImageNet defaults for classification, [0,1] for YOLO)
`ppu.type`	Model architecture (0 for anchor-based, 1 for anchor-free)
`ppu.num_classes`	ONNX output shape analysis

7.8 Output Isolation

All compilation artifacts go to `dx-agent-dev/<session_id>/` by default. This keeps each compilation session self-contained and reproducible.

Session ID format: `YYYYMMDD-HHMMSS_<agent>_<model>_<task>` where `<agent>` is `claude`, `codex`, `copilot`, `cursor`, or `opencode`

Output Type	Path	When
Default (isolated)	`dx-agent-dev/<session_id>/`	Always, unless user says otherwise
Custom	User-specified path via `-o`	When explicitly requested

Working directory contents after compilation:

```
dx-agent-dev/<session_id>/
├── calibration_dataset   → ../../dx_com/calibration_dataset/ (symlink)
├── config.json           (auto-generated)
├── model.onnx            (input or converted)
├── model.dxnn        (compiled output)
├── compiler.log          (compilation log)
├── detect_model.py       (inference application)
├── verify.py             (ONNX vs DXNN verification)
├── setup.sh              (environment setup)
├── run.sh                (inference launcher)
└── README.md             (session report with file list)
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.9 Calibration Dataset Management

The agent automatically manages calibration data:

1. Checks if `dx_com/calibration_dataset/` exists (100 JPEG images)
2. If missing, runs `example/2-download_sample_calibration_dataset.sh`
3. Creates a symlink in the session working directory
4. Uses relative path `./calibration_dataset` in config.json (never absolute)

7.10 Sample Model Workflow

The `example/` directory provides a complete 3-step workflow for testing the compilation pipeline with pre-built sample models:

```
cd dx-compiler
./example/1-download_sample_models.sh      # Download ONNX + JSON configs
./example/2-download_sample_calibration_dataset.sh  # Download calibration dataset
./example/3-compile_sample_models.sh       # Compile all sample models to .dxnn
```

Available sample models: YOLOV5S-1, YOLOV5S_Face-1, MobileNetV2-1

The downloaded JSON config files serve as canonical references for agents generating config.json for new models — they demonstrate proper input naming, preprocessing parameters, calibration settings, and PPU configuration.

7.11 Session Sentinels

Agents output fixed markers at the start and end of each task for automated testing:

Marker	When
`[DX-AGENT-DEV: START]`	First line of the agent's response
`[DX-AGENT-DEV: DONE (output-dir: <relative_path>)]`	Last line after all work is complete. `<relative_path>` is the session output directory relative to the project root. If no files were generated, omit the `(output-dir: ...)` part.

© Copyright 2026 DEEPX All Rights Reserved.

Rules:

1. **CRITICAL** — Output `[DX-AGENT-DEV: START]` as the absolute first line of your first response, before any other text. This is non-negotiable even if the user says to proceed autonomously.
2. Output `[DX-AGENT-DEV: DONE (output-dir: <path>)]` as the very last line after all work is complete.
3. Sub-agents invoked via handoff do not output sentinels — only the top-level agent does.
4. If the user sends multiple prompts in a session, output START/DONE for each prompt.
5. The `output-dir` in DONE must be the relative path from the project root to the session output directory.
6. **Never output DONE after only producing planning artifacts** (specs, plans, design documents). DONE means all deliverables are produced — implementation code, scripts, configs, and validation results.

7.12 Troubleshooting

Symptom	Cause	Fix
`ValueError: Batch size must be 1`	Model has batch > 1	Re-export ONNX with batch=1 input shape
`KeyError` in config.json inputs	Input name doesn't match ONNX model	Use `onnx.load(model).graph.input[0].name` to get correct name
Compilation hangs at calibration	Not enough calibration images or wrong file extensions	Verify `dataset_path` exists and contains images matching `file_extensions`
Low accuracy after compilation	Calibration data not representative	Use real-world inference images, increase `calibration_num` to 200+
`ONNX opset not supported`	Opset < 11 or > 21	Re-export with `opset_version=17` (recommended)
PPU output format mismatch	Wrong PPU type for model architecture	Use type=0 for YOLOv3/v4/v5/v7, type=1 for YOLOX/v8/v9/v10/v11/v12
ONNX has 6 outputs instead of 1	Ultralytics YOLO `Detect.export` flag not set	Use `model.export(format="onnx")` or set `Detect.export=True` before `torch.onnx.export()`.

© Copyright 2026 DEEPX All Rights Reserved.

Symptom	Cause	Fix
		See Pitfall #10 in `.deepx/memory/common_pitfalls.md`
Inference gives wrong class labels	Postprocessing class index mismatch	Run `verify.py` to compare ONNX vs DXNN output. Check 0-indexed vs 1-indexed COCO classes
No detections from compiled model	Postprocessing bug in generated app	Run `verify.py`. Check output tensor shape parsing and confidence threshold

7.13 Mandatory Output Artifacts

Every compilation session that generates an inference application MUST also produce these deployment artifacts in the session directory:

Artifact	Purpose
`setup.sh`	Checks dx-runtime installation via `sanity_check.sh`, installs missing components via `install.sh`; checks dxcom availability, installs via `dx-compiler/install.sh` if missing; creates venv, installs dx_engine, opencv-python, numpy, onnxruntime
`run.sh`	One-command inference launcher with task-aware sample image paths
`README.md`	Session summary: pipeline, generated files, quick start, environment info
`verify.py`	ONNX vs DXNN inference comparison — catches postprocessing bugs

The user should be able to run `bash setup.sh && bash run.sh` immediately after compilation with zero manual setup.

© Copyright 2026 DEEPX All Rights Reserved.

7.14 TDD Verification Gate

Before presenting the final compilation report, the agent runs `verify.py` to compare ONNX inference (ground truth) against DXNN inference output:

- Uses sample images from `dx-runtime/dx_app/sample/` — selected based on model task:
- Object Detection: `img/sample_dog.jpg`, `img/sample_horse.jpg`
- Face Detection: `img/sample_face.jpg`, `img/sample_crowd.jpg`
- Pose: `img/sample_people.jpg` | Hand: `img/sample_hand.jpg`
- OBB: `dota8_test/P0177.png` | Segmentation: `img/sample_street.jpg`
- Classification: `ILSVRC2012/0.jpeg` | Super Resolution: `img/sample_superresolution.png`
- Low-light: `img/sample_lowlight.jpg` | Denoising: `img/sample_denoising.jpg`
- Compares detection count (within 20%), class labels (top-K match), bbox IoU (avg > 0.5)
- **PASS** → compilation and inference app are correct
- **FAIL** → postprocessing bugs exist; agent must debug and fix before reporting success

This gate was introduced after real-world testing revealed that compiled models can benchmark correctly (e.g., 139 FPS) while the generated inference application produces wrong results due to postprocessing bugs (wrong class mapping, incorrect bbox decoding).

ⓒ Copyright 2026 DEEPX All Rights Reserved.

8. Change Log

8.1 v2.4.1 (2026-07-27)

DX-Compiler Version

- DX-COM: v2.4.0
- DX-TRON: v2.0.1 (Deprecated)

Documentation-only hotfix; no changes to DX-COM or DX-TRON binaries.

Fixed

- Fixed broken internal links and reorganized several pages in the user manual for improved navigation.
- Corrected the `dx_com.pre_optimize()` supported-model listings in the user manual: YOLOv10 is not a valid target and has been removed from the `yolo_postprocess` and PPU-type-2 migration guidance.

8.2 v2.4.0 (2026-06-16)

DX-Compiler Version

- DX-COM: v2.4.0
- DX-TRON: v2.0.1 (Deprecated)

> ⚠ **Deprecation Notice — DX-TRON**
>
> Starting with DX-Compiler **v2.4.0**, **DX-TRON is deprecated** and will be removed in a future release. No further feature updates or bug fixes are planned for DX-TRON. We recommend migrating to the new **HTML graph viewer** bundled with DX-COM (see *Interactive HTML Graph Viewer* below) for model inspection and visualization.

© Copyright 2026 DEEPX All Rights Reserved.

> ⚠ **Deprecation Notice — PPU Type 2**
>
> Starting with DX-Compiler **v2.4.0**, the legacy **PPU type 2** post-processing mode (used for YOLOv8-family DFL post-processing) is **deprecated** and will be removed in a future release. Compiling a model that uses PPU type 2 now emits a deprecation warning. Please migrate to the new `dx_com.pre_optimize()` **API** with the built-in YOLO post-processing passes (see *Added* below).

Changed

- **PPU Type 2 Deprecated**: The legacy PPU type 2 post-processing mode is now deprecated and emits a deprecation warning when used. See the Deprecation Notice above for the migration path.
- **Faster, Lighter Compilation**: Reduced memory usage and improved compile time, especially on large models.
- **Expanded Python Version Support**: In addition to Python 3.8–3.12, DX-COM now supports **Python 3.13 and 3.14**.

Fixed

- Fixed a Python API issue where models expecting integer inputs were sometimes fed float data, causing accuracy degradation.
- Fixed several Q-PRO / DXQ quantization crashes and stability issues observed on real models.
- Fixed multiple compilation errors and runtime issues caused by tiling, partitioning, and memory allocation in models containing `Split`, `Concat`, `Reshape`, `Bilinear Resize`, `Clip`, or odd spatial dimensions.
- Fixed compatibility issues with **NumPy 2.4+** and **onnxruntime ≥ 1.25.0**.

Added

- **Automated Q-PRO Configuration**: Q-PRO quantization (formerly available only by hand-picking DXQ combinations) is now much easier to use. DX-COM can now **automatically generate DXQ combinations** for you and run Q-PRO under the hood, removing the need to manually tune the many DXQ knobs to get higher-accuracy quantization.
- **Quantization-Aware Training (QAT)**: Added end-to-end **QAT** support directly through `dx_com.compile()`. When the supplied config JSON includes a `qmaster` block, `dx_com.compile()` automatically switches to QAT mode and runs the training pipeline using the same dataset settings as PTQ calibration. Available from both the `dxcom` CLI and the Python API. A new `fast_run` flag is also available for quick QAT smoke tests. See Quantization-Aware Training (QAT).
- **QXNN Resume (Re-quantization without Recompile)**: Added a checkpoint-based **QXNN resume** flow available from both the `dxcom` CLI and the Python API. Once a model has been compiled, users can re-run

© Copyright 2026 DEEPX All Rights Reserved.

quantization with different settings (e.g., a different calibration method) without repeating the earlier compile phases, dramatically shortening the iteration loop when tuning quantization quality.

- **Quantization Diagnosis Report (HTML)**: Added an HTML report that visualizes per-region quantization quality, flags high-severity regions, and includes ready-to-paste compile snippets to retry compilation with recommended settings. Enabled via the new `quant_diagnosis` option, available from both the `dxcom` CLI and `dx_com.compile()`.
- **Interactive HTML Graph Viewer (replaces DX-TRON)**: DX-COM now produces a standalone HTML viewer for inspecting compiled models, including parameter shapes, CPU/NPU partition reasons, and cross-subgraph connections. This replaces the DX-TRON workflow (see Deprecation Notice above).
- **`dx_com.pre_optimize()` API**: Added a new top-level `dx_com.pre_optimize()` API for applying ONNX-level pre-processing transforms before compilation, with built-in support for **YOLO post-processing** integration (detection and segmentation modes). See the Pre-Optimize API chapter.
- **Ubuntu 26.04 Validation**: DX-COM is now validated on **Ubuntu 26.04**, in addition to the previously supported Linux distributions.

Known Issues

- Significant FPS degradation has been observed in models using PReLU as an activation function.

8.3 v2.3.1 (May 2026)

DX-Compiler Version

- DX-COM: v2.3.0
- DX-TRON: v2.0.1

Fixed

- Fixed `uninstall.sh` not removing installed packages and extracted module directories.
  - `dx_com` Python package is now properly uninstalled via `pip3 uninstall` before the virtual environment is removed.
  - `dxtron` Debian package is now properly removed via `apt-get remove`.
  - `dx_com/` and `dx_tron/` directories are now deleted on uninstall.

Added

- Added `--target=<dx_com|dx_tron|all>` option to `uninstall.sh` (default: `all`), consistent with `install.sh`.

© Copyright 2026 DEEPX All Rights Reserved.

8.4 v2.3.0 (March 2026)

DX-Compiler Version

- dx_com : 2.3.0
- dx_tron : 2.0.1

Changed

- **Python Packaging**: Relaxed Python package dependency constraints to improve installation flexibility.
- **Distribution**: The standalone DX-COM executable distribution is deprecated. The user manual now documents the wheel-based workflow only.
- **Performance**:
  - Reduced NPU inference latency for most models.
  - Improved compiler performance to reduce compilation time.

Fixed

- **Compiler Stability**: Fixed various DX-COM compiler stability issues.

Added

- **TopK-Optimized Post-Processing Pipeline**: Added an optimized post-processing pipeline for supported DFL-based YOLO models that applies TopK filtering before bounding box decoding, reducing CPU post-processing workload and improving runtime efficiency.
- **Batched Convolution Support**: Added support for batched convolution.
- **Linux Distribution Validation**: In addition to Ubuntu 20.04, 22.04, and 24.04, DX-COM v2.3.0 was also validated on Fedora 42-45, Red Hat 9-10, and CentOS Stream 9-10.

8.5 v2.2.1 (February 2026)

DX-Compiler Version

- dx_com : 2.2.1
- dx_tron : 2.0.1

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Changed

- None

Fixed

- **DXQ Quantization**: Fixed DXQ enhanced quantization option bugs.
- **Python Wheel Package**: Fixed PPU compilation bug for Python 3.8, 3.9, and 3.10.
- **Input Validation**: Fixed an issue where compilation proceeded without error when invalid model input names were specified.

Added

- **GPU Quantization (JSON Config)**: Added `quantization_device` support in JSON configuration file, enabling GPU-accelerated quantization via CLI (`dxcom`) in addition to the `dx_com` Python module. Available only with the Python wheel package installation.
- **GPU Auto-Detection**: When `quantization_device` is not specified, DX-COM now automatically uses GPU if a CUDA-compatible GPU is available, otherwise falls back to CPU.

8.6 v2.2.0 (December 2025)

DX-Compiler Version

- dx_com : 2.2.0
- dx_tron : 2.0.1

Changed

- None

Fixed

- **Model Accuracy**: Resolved accuracy degradation issue in the `DeepLabV3PlusMobilenet-1` model from DX ModelZoo.

Added

- **PPU Support**: Extended PPU support to YOLOv8, YOLOv9, YOLOv10, YOLOv11, and YOLOv12.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **Python Wheel Package**: DX-COM is now available as a Python wheel package (in addition to the existing executable file), supporting Python 3.8, 3.9, 3.10, 3.11, and 3.12, enabling programmatic compilation using torch DataLoader without configuration files.
- **Multi-Input Model Support**: Added support for multi-input models through torch DataLoader in the Python wheel package.
- **DX-TRON**:
  - Debian package (.deb) installation support
  - Local web browser hosting (`dxtron` command)
  - Ubuntu 24.04 support

8.7 v2.1.0 (November 2025)

DX-Compiler Version

- dx_com : 2.1.0
- dx_tron : 2.0.0

Changed

- **Command-Line Interface**: Removed deprecated command-line options: `--jobs` , `--shrink` , `--info` (or `-i`).
- **ONNX Support**:
  - Removed restrictions on `Split` , `Transpose` , `Reshape` , `Flatten` , and `Slice` operators.
  - Clarified ONNX opset version support (versions 11-21 are supported; version 22 and above are not supported).

Added

- **Command-Line Interface**:
  - `--aggressive_partitioning` : Enables aggressive partitioning to maximize operations executed on NPU.
  - `--opt_level {0,1}` : Controls optimization level (default: 1).
  - `--compile_input_nodes` and `--compile_output_nodes` : Support for partial compilation.
- **ONNX Support**: Added support for `Gather` operator.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **Quantization**: Reintroduced the DXQ enhanced quantization option (`enhanced_scheme` , DXQ-P0 to DXQ-P5), previously removed in dx_com v2.0.0.
- **PPU (Post-Processing Unit)**: Reinstated PPU support.
  - Supported models: YOLOv3, YOLOv4, YOLOv5, YOLOv7 (anchor-based), YOLOX (anchor-free).

8.8 v2.0.0 (September 2025)

ONNX Support

- Re-enabled support for the following operators:
  - `Softmax`
  - `Slice`
- Newly added support for the following operator:
  - `ConvTranspose`

Model Support

- Partial support for Vision Transformer (ViT) models
- Verified with the following OpenCLIP models:
  - ViT-L-14, ViT-L-14-336, ViT-L-14-quickgelu
  - RN50x64, RN50x16
  - ViT-B-16, ViT-B-32-256, ViT-B-16-quickgelu

Compatibility and Deprecations

- Compatibility with DX-RT versions earlier than v3.0.0 is not guaranteed
- The DXQ enhanced quantization option (`enhanced_scheme`) was removed in dx_com v2.0.0 and reintroduced in dx_com v2.1.0
- `PPU (Post-Processing Unit)` is no longer supported, and there are no current plans to reinstate it

8.9 v1.60.1 (June 2025)

Bug Fixes

- Internal bug fixes

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Command-Line Interface Updates

- Added support for:
  - `-v` option: Displays **DX-COM module version**
  - `-i` option: Displays **internal module information**
    → For usage, see: the **CLI Execution** section in Execution of DX-COM

ONNX Support

- The following operators were deprecated and are scheduled to be re-supported in a future release:
  - `Softmax`
  - `Slice`

ⓒ Copyright 2026 DEEPX All Rights Reserved.