

DXNN Compiler (DX-COM) User Manual

2026/10/23 (v2.5.0)

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 13
3.4 ONNX File Configuration 14
3.5 JSON File Configuration 16
3.6 Execution of DX-COM 36
4. Supported ONNX Operators 61
4.1 Operator Support Details 61
5. Quantization Tuning Workflow 68
5.1 Quantization Tuning Workflow 68
5.2 Q-Master (Quantization-Aware Training) 77
6. Pre-Optimize API 92
6.1 How It Works 92
6.2 Available Passes 93
6.3 API Reference 94
6.4 Configuration Schema 95
6.5 Usage 98
6.6 Identifying Conv Head Tensor Names 103
6.7 Performance Reference 109
6.8 Migration from PPU Type 2 110
7. Common Use Cases 113
7.1 Use Case 1: Simple Image Classification (ResNet / MobileNet) 113
7.2 Use Case 2: Multi-Input Models (Stereo Vision) 115
7.3 Use Case 3: Performance Optimization for Edge Devices 117
7.4 Use Case 4: Custom Data Type (Non-Image) 119
7.5 Use Case 5: Enhanced Quantization (DXQ) 120
7.6 Use Case 6: Diagnose and Re-quantize Without Recompile 123
7.7 Use Case 7: YOLO Post-Processing Optimization (CPU-Constrained Edge) 125

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7.8 Use Case 8: PPU Hardware Acceleration (YOLO) 127
8. Development Tools 131
8.1 Compilation Summary Report 131
9. DEEPX Agent-Driven Development for DX-COM - dx-agent-dev (Beta) 134
9.1 Introduction 134
9.2 Prerequisites 134
9.3 Architecture Overview 134
9.4 Available Agents and Skills 135
9.5 Supported AI Tools 137
9.6 User Scenarios 141
9.7 Config Auto-Inference 144
9.8 Output Isolation 144
9.9 Calibration Dataset Management 145
9.10 Sample Model Workflow 145
9.11 Session Sentinels 146
9.12 Troubleshooting 146
9.13 Mandatory Output Artifacts 147
9.14 TDD Verification Gate 148
10. Change Log 150
10.1 v2.5.0 (2026-10-16) 150
10.2 v2.4.1 (2026-08-14) 152
10.3 v2.4.0 (2026-07-22) 153
10.4 v2.3.1 (2026-05-06) 155
10.5 v2.3.0 (2026-04-10) 155
10.6 v2.2.1 (2026-02-09) 156
10.7 v2.2.0 (2026-01-16) 157
10.8 v2.1.0 (2025-11-28) 158
10.9 v2.0.0 (2025-09-08) 159
10.10 v1.0.0 (2025-07-23) 159

ⓒ 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
```

The compiled `.dxnn` files are saved to `dx_com/output/`.

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.

© Copyright 2026 DEEPX All Rights Reserved.

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

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()`:

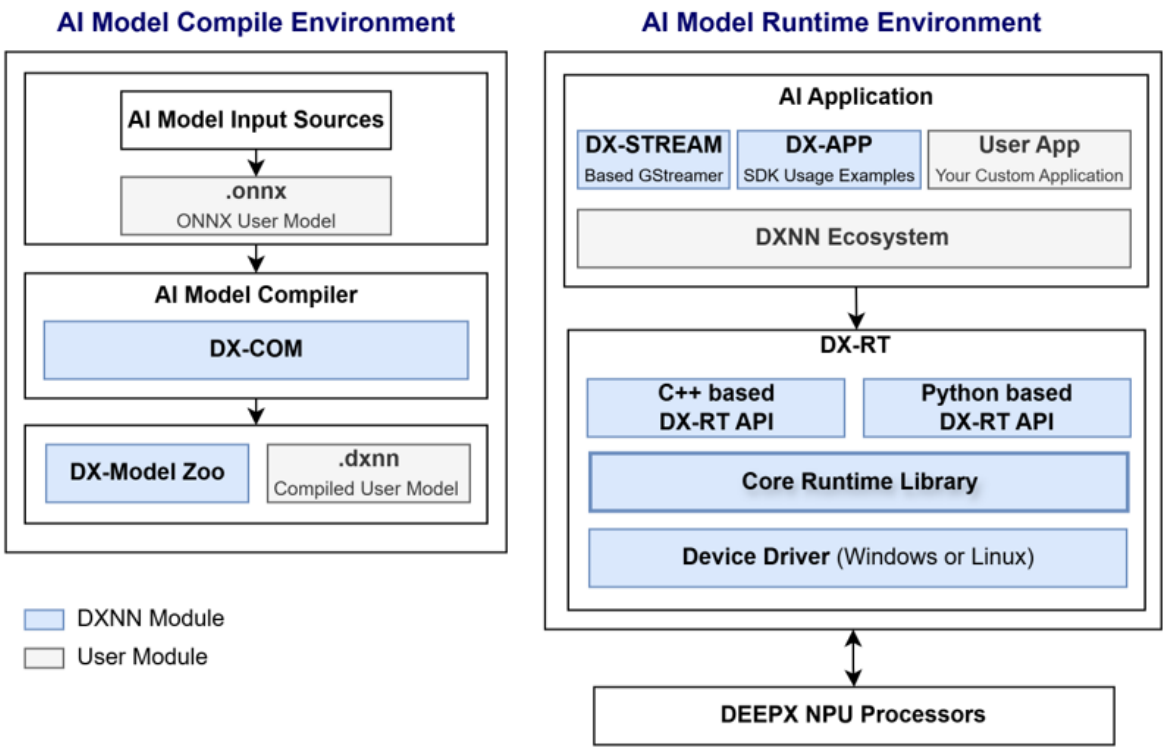
```
import dx_com
dx_com.compile(model="model.onnx", output_dir="output/", config="config.json")
```


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

© Copyright 2026 DEEPX All Rights Reserved.

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 pre-processing, inference, and post-processing, 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

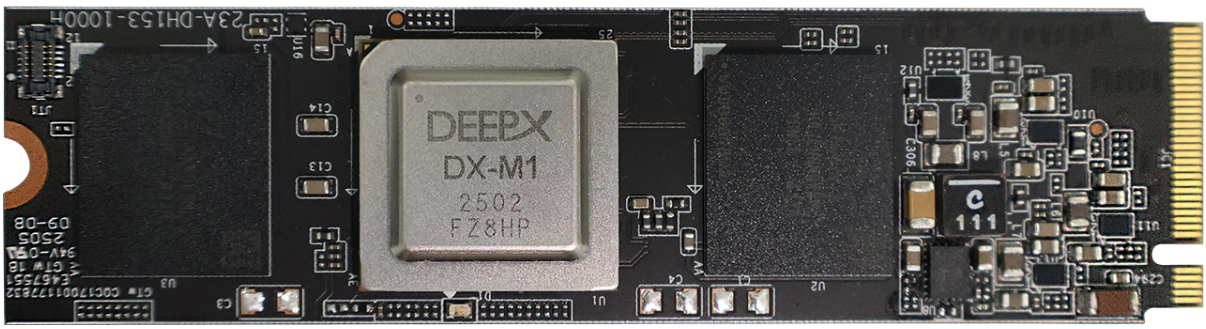
3.1 System Requirements of DX-COM

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 and later.
  NOTE. Ubuntu 18.04 OS is **NOT** supported.
- **glibc:** ≥ 2.31 (matches the `manylinux_2_31_x86_64` wheel platform tag)
- **Python:** 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14
- **Output format:** `.dxnn` file format v9. Models in this format require **DX-RT v3.5.0 or later**; DX-RT v3.4.2 and earlier cannot load them.

> **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`
  - 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 libglib2.0-0 make
```

Fedora

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

RHEL / CentOS (9, 10)

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

© Copyright 2026 DEEPX All Rights Reserved.

```
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
```

Wheel filename format

The wheel filename follows PEP 425 / PEP 600 conventions: `dx_com-<VERSION>-<PYTAG>-<ABITAG>-<PLATFORMTAG>.whl`. For DX-COM v2.5.0 on CPython 3.12, this looks like `dx_com-2.5.0-cp312-cp312-manylinux_2_31_x86_64.whl`.

© Copyright 2026 DEEPX All Rights Reserved.

Option 3: Install with uv (Fastest)

uv resolves and installs dependencies substantially faster than pip. `install.sh` uses uv automatically when it is already installed, so no flag is needed:

```
./install.sh --target=dx_com
```

If uv is not installed, `install.sh` uses pip and says so. Pass `--uv=true` to have it install uv first, or `--uv=false` to force the pip path:

```
./install.sh --target=dx_com --uv=true
./install.sh --target=dx_com --uv=false
```

To use uv directly against an existing virtual environment:

```
uv pip install dx-com
```

> **uv and pip coexist**
>
> When uv is used, `install.sh` creates the virtual environment with `uv venv --seed`, so `pip` remains available inside it. `--uv=false` keeps the original pip-only path unchanged. uv is only downloaded when you pass `--uv=true`; a plain `./install.sh` on a machine without uv behaves exactly as it did before, using pip.

> **Archive mode still uses pip**
>
> `--archive_mode=y` downloads wheels with `pip download`; uv has no equivalent command, so archive mode is unaffected by `--uv`.

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.

ⓒ 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/
├── sample_models/          # Sample model assets
│   ├── onnx/               # Sample ONNX models
│   └── json/               # Sample JSON files
├── calibration_dataset/    # Sample calibration dataset
└── output/                 # Compiled .dxnn files (created by 3-compile_sample_models.sh)
```

dx_com/

This directory stores the sample assets prepared during `install.sh`. The `dxcom` command-line interface and the `dx_com` Python module are installed into the virtual environment by `install.sh`; 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=17,       # ONNX opset version (11~21 supported, 17 recommended)
  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, 17 recommended)
- `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.
- `input shape` : Input shapes must be static. Do not use `dynamic_axes` ; dynamic shapes are not supported.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

3.5 JSON File Configuration

This chapter describes various parameters required for compiling an ONNX model using **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`	No	Calibration method: `"ema"` (default), `"minmax"`, or `"iqr"`
`calibration_num`	No	Number of calibration samples (default: 100)
`default_loader`	Yes	Preprocessing configuration with dataset loading
`quantization_device`	No	**Deprecated in v2.5.0 — ignored.** Select the device with `--use_gpu` / `use_gpu` and `CUDA_VISIBLE_DEVICES` (see **Quantization Device** below)
`enhanced_scheme`	No	DXQ-P0~P5 quantization enhancement for accuracy improvement (DX-COM v2.1.0+)
`qmaster`	No	Q-Master (QAT) hyperparameters. Presence of this block enables Q-Master (DX-COM v2.4.0+). See Q-Master (QAT)

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Required	Description
`ppu`	No	Object detection post-processing (YOLO models)
`pre_optimize`	No	ONNX-level post-processing optimization for YOLO/RTMDet models, written as a list `[{pass_name: config}]`. Mutually exclusive with `ppu`. See Pre-Optimize API

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]
  }
}
```


© Copyright 2026 DEEPX All Rights Reserved.

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.

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.
- `iqr` :
  Uses interquartile range (IQR) based on the 25th and 75th percentiles of activation values.
  More robust to outliers than `minmax`, particularly useful when calibration data contains extreme values.

Values are lowercase (`"EMA"` is rejected). Default: `"ema"`.

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.

© Copyright 2026 DEEPX All Rights Reserved.

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.

Default: `100`. Use a positive integer. If the dataset has fewer images than `calibration_num`, all of them are used without a warning.

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}},
   {"convertColor": {"form": "BGR2RGB"}},
   {"div": {"x": 255}},
   {"normalize": {"mean": [0.485, 0.456, 0.406], "std": [0.229, 0.224, 0.225]}},
   {"transpose": {"axis": [2, 0, 1]}},
   {"expandDim": {"axis": 0}}
  ]
 }
}
```


© Copyright 2026 DEEPX All Rights Reserved.

`default_loader` reads each image with OpenCV as **BGR, HWC, uint8 (0–255)** and applies only the listed `preprocessings`, in order. It never rescales, converts color, or transposes on its own:

- For an NCHW model (most PyTorch exports), end the list with `transpose [2, 0, 1]` and then `expandDim 0`. Without the transpose, calibration fails with `DataLoaderError: … The config specifies (1, H, W, C), but model requires (1, C, H, W)`.
- For a model trained on RGB input in [0, 1], put `convertColor BGR2RGB` and `div 255` before `normalize`. `normalize` alone does not rescale 0–255 values, and the compile still succeeds with wrongly scaled calibration data.
- `dataset_path`: a relative path is resolved from the current working directory, not from the config file. Prefer an absolute path.
- `file_extensions`: required. A list of extensions without the dot, matched case-sensitively (list `"JPG"` as well as `"jpg"` when both exist). Subfolders are searched too.

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

Quantization runs on GPU by default when a CUDA-capable GPU is available, and falls back to CPU otherwise. The device is selected outside the JSON file:

- `dxcom` CLI: `--use_gpu True` (default) or `--use_gpu False` to force CPU.
- Python: `dx_com.compile(..., use_gpu=True)` (default) or `use_gpu=False` to force CPU.
- A specific GPU: set the `CUDA_VISIBLE_DEVICES` environment variable (e.g., `CUDA_VISIBLE_DEVICES=1 dxcom ...`).

ⓒ Copyright 2026 DEEPX All Rights Reserved.

> **`quantization_device` is deprecated (v2.5.0)**
>
> From **DX-COM v2.5.0** the JSON `quantization_device` field and the Python `quantization_device` argument are **ignored**. DX-COM prints `UserWarning: quantization_device (JSON config) is deprecated and ignored` and selects the device as described above. Remove the field from existing configs.

> **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.

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

© Copyright 2026 DEEPX All Rights Reserved.

Name	Compilation Speed	Accuracy Improvement
DXQ-P1	Very Slow	Low-Middle
DXQ-P2	Very 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. Running quantization on GPU (the default when a GPU is available) 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.

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
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  }
}
```

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
  }
 }
}
```

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.

© Copyright 2026 DEEPX All Rights Reserved.

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).

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

© Copyright 2026 DEEPX All Rights Reserved.

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 (CPU TopK)	YOLOv8, YOLOv9, YOLOv11, YOLOv12

> ⚠ **Type 2 is deprecated**
>
> **PPU type 2** is deprecated as of DX-Compiler v2.4.0 and will be removed in a future release. Use the `dx_com.pre_optimize()` API instead for the same TopK-first optimization with additional features.

© Copyright 2026 DEEPX All Rights Reserved.

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": [
   {"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:

© Copyright 2026 DEEPX All Rights Reserved.

```
{
 "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

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, pose, YOLO26, and RTMDet). Compiling a model with PPU type 2 still works 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.

© Copyright 2026 DEEPX All Rights Reserved.

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"}
  ]
 }
}
```

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

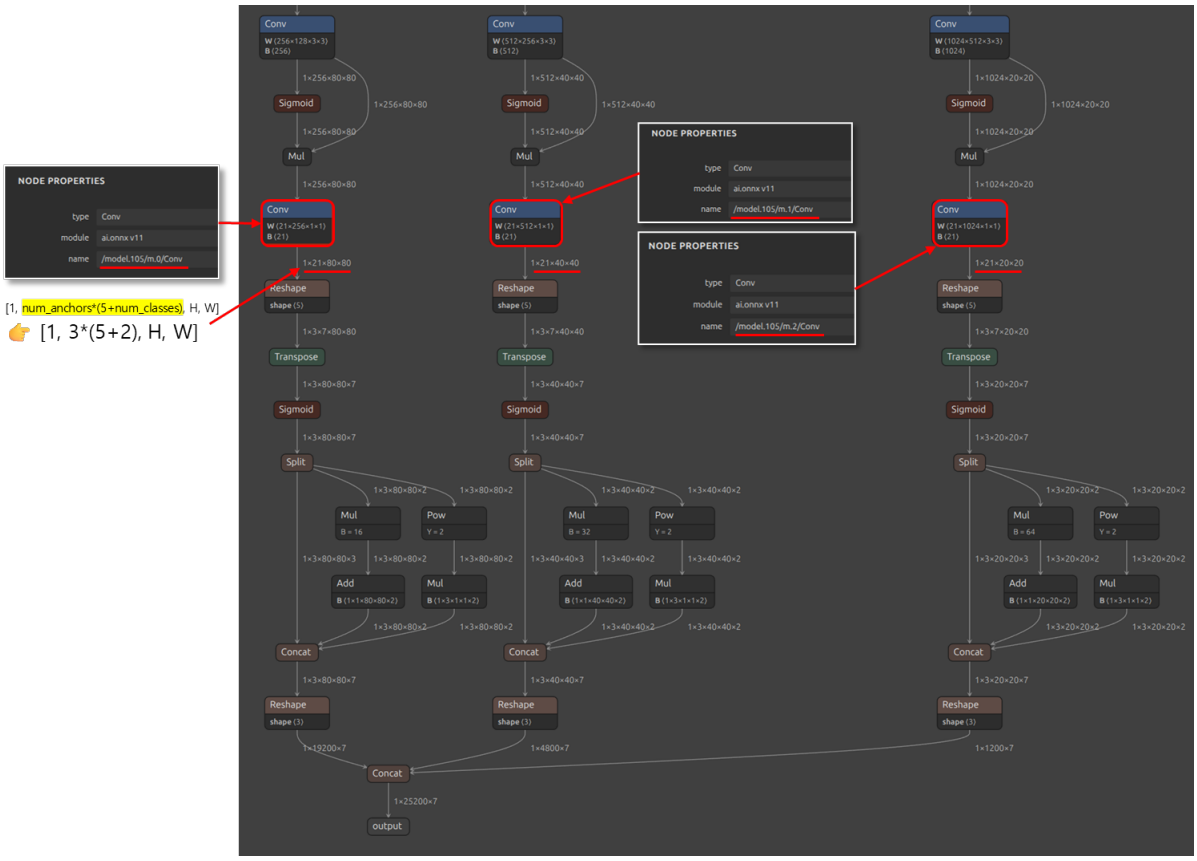
To configure PPU, you need to identify specific node names in your ONNX model:

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

© Copyright 2026 DEEPX All Rights Reserved.

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

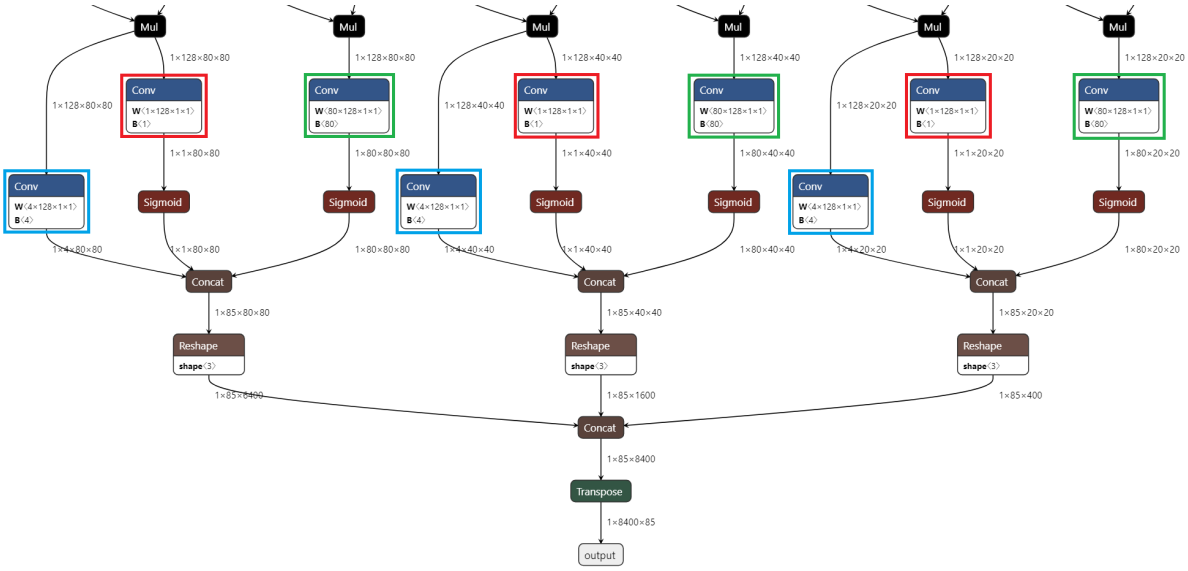
- Trace backwards from model outputs to locate detection head Conv layers
- These Conv layers output feature maps with shape `[1, num_anchors*(5+num_classes), H, W]`
- Example node names in YOLOv7: `/model.105/m.0/Conv`, `/model.105/m.1/Conv`, `/model.105/m.2/Conv`



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)

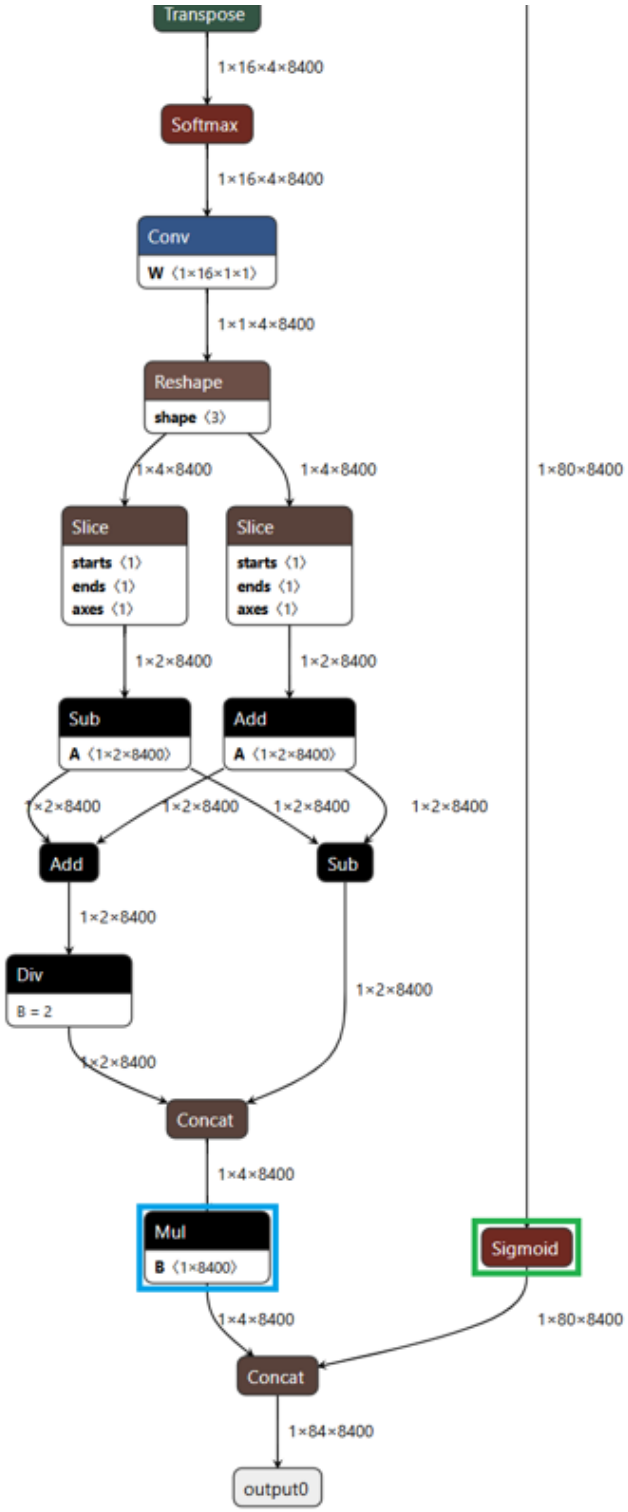
© Copyright 2026 DEEPX All Rights Reserved.



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`

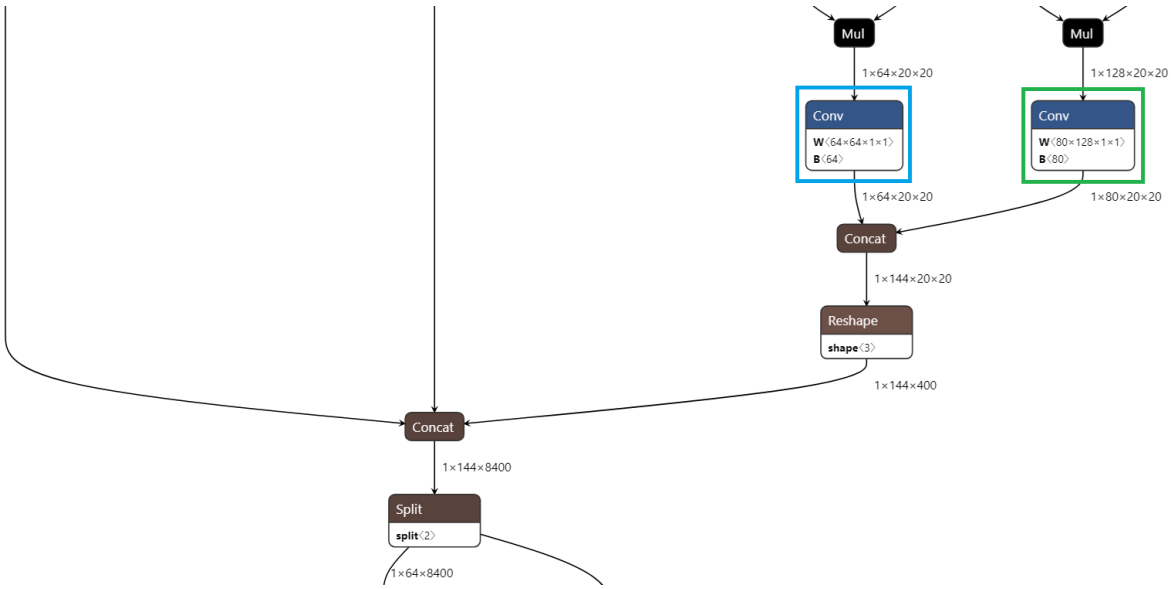
ⓒ Copyright 2026 DEEPX All Rights Reserved.



© 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)

```
# 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
```


© Copyright 2026 DEEPX All Rights Reserved.

convertColor

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`

`form` is case-sensitive. The `*2GRAY` forms output `(H, W, 1)`, so a 1-channel NCHW model still needs `transpose` `[2, 0, 1]`. `convertColor` is not integrated into the NPU graph (it is listed under `Skipped nodes` in `compiler.log`), so the inference code must apply the same conversion.

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`

Use the uppercase names from the row of the selected `mode`. Names from the other row (e.g. `BILINEAR` with `default`) and lowercase names are not accepted.

The target size can also be given with `size` instead of `width` and `height`:

```
{"resize": {"mode": "torchvision", "size": 256, "interpolation": "BILINEAR"}}
```

centercrop

Crops the central region of the input image.

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

transpose

Rearranges tensor dimensions.

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


© Copyright 2026 DEEPX All Rights Reserved.

`default_loader` images are HWC, so `[2, 0, 1]` converts them to CHW for an NCHW model. If the transpose comes after `expandDim`, use `[0, 3, 1, 2]` instead.

expandDim

Adds a new dimension at the specified axis.

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

normalize

Normalizes input data by mean and standard deviation. This operation may be integrated into the NPU graph during compilation (see the **Automatic Preprocessing Optimization** note 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 the **Automatic Preprocessing Optimization** note 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 Usage** 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

QXNN Resume mode

When `--checkpoint` is used, `-m` and `-c` are not required; only `-o` is required. See Quantization Tuning Workflow.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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.

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
`--use_gpu`	`{True,False}` (Default: `True`)	**(New in v2.5.0)** Run quantization on GPU when available (`True`) or force CPU (`False`). Select which physical GPU to use with the `CUDA_VISIBLE_DEVICES` environment variable
`--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.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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.

GPU / CPU Selection (`--use_gpu`) — New in v2.5.0
Quantization runs on GPU by default when a CUDA-capable GPU is available, and falls back to CPU otherwise. Pass `--use_gpu False` to force CPU. To pin a specific GPU on the command line, set the `CUDA_VISIBLE_DEVICES` environment variable (e.g. `CUDA_VISIBLE_DEVICES=1 dxcom ...`).

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

© Copyright 2026 DEEPX All Rights Reserved.

Option	Value/Default	Description
		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=1024`	**(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.

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.

© Copyright 2026 DEEPX All Rights Reserved.

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`) in the output directory after compilation. See Compilation Summary Report
`--verbose`	N/A	**(New in v2.5.0)** Expand masked error messages to include the origin type, the original message, and one bounded direct cause. Useful when a compilation error is otherwise reported in summarized form
`--version`	`-v`	Prints the compiler module version and exits

© Copyright 2026 DEEPX All Rights Reserved.

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 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

© Copyright 2026 DEEPX All Rights Reserved.

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.

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

© Copyright 2026 DEEPX All Rights Reserved.

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

```python
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,
    use_gpu: bool = True,  # New in v2.5.0
    opt_level: int = 1,
    aggressive_partitioning: bool = False,
    input_nodes: Optional[List[str]] = None,
    output_nodes: Optional[List[str]] = None,
    use_q_pro: bool = False,
    enhanced_scheme: Optional[Dict] = None,
    ppu_config: Optional[PPUConfig] = None,
    qmaster: Optional[QMasterConfig] = None,  # New in v2.5.0 (replaces qat_* args)
    gen_log: bool = False,
    float64_calibration: bool = False,
    export_html: bool = False,
    verbose: bool = False,  # New in v2.5.0
    **kwargs,
) -> None
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

Changed in v2.5.0 — entry point signature and QAT parameters

- `dx_com.compile()` now accepts keyword arguments (`**kwargs`); call it with keyword arguments as shown above.
- **QAT parameters were restructured.** The former `quantization_mode` , `qat_config` , `qat_skip_training` , and `qat_resume_from_checkpoint` arguments have been **replaced by a single `qmaster` argument** that takes a `QMasterConfig` object (import it from the top-level `dx_com` package). The old `qat_*` arguments are **no longer honored** when compiling from Python code. Compiling from a JSON config that contains a `qmaster` block is **unchanged** — QAT is still auto-selected (see Q-Master (QAT)).
- New arguments: `use_gpu` (GPU/CPU quantization toggle) and `verbose` (expanded error messages).

Additional Parameters

The signature above lists the most commonly used parameters. `dx_com.compile()` accepts further advanced/diagnostic parameters (e.g. `quant_diagnosis` , `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")
```


© Copyright 2026 DEEPX All Rights Reserved.

`output_dir`

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

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

`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]
```


© Copyright 2026 DEEPX All Rights Reserved.

```
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 `__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
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    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]
```

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). `resize2` and `resize3` also require `scale`; with `width`/`height` only they fail. Prefer `resize`	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]`)	

© Copyright 2026 DEEPX All Rights Reserved.

JSON operation	Arguments	DataLoader equivalent
		`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

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

© Copyright 2026 DEEPX All Rights Reserved.

calibration_method

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

calibration_num

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

quantization_device *(Deprecated in v2.5.0)*

- Ignored with a UserWarning. Select the device with use_gpu and the CUDA_VISIBLE_DEVICES environment variable instead.

use_gpu *(New in v2.5.0)*

- **Type**: bool
- **Default**: True — GPU when available, otherwise CPU (same as the CLI --use_gpu default).
- **Description**: True runs quantization on GPU when available, False forces CPU. To use a specific GPU, set the CUDA_VISIBLE_DEVICES environment variable (e.g., CUDA_VISIBLE_DEVICES=1 python compile.py). Pass only True or False : any other value, including None , makes DX-COM fall back to its legacy compile path, which skips a JSON pre_optimize block and ignores verbose .

```
use_gpu=True  # run quantization on GPU
use_gpu=False  # force CPU
```

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

© Copyright 2026 DEEPX All Rights Reserved.

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"]
```

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,
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    },
}
```

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.
- **See also**: the **Automatic Q-PRO (use_q_pro)** section above

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) — **deprecated since v2.4.0**	YOLOv8, v9, v11, v12

© Copyright 2026 DEEPX All Rights Reserved.

PPU type 2 is deprecated

`PPUTypes.YOLOV8` (type 2) is deprecated as of DX-Compiler v2.4.0. Use the `dx_com.pre_optimize()` API instead (see **Use Case 7** in Common Use Cases).

PPUTypes.YOLOV8 supports multiple YOLO versions

Despite the enum name, `PPUTypes.YOLOV8` supports YOLOv8, YOLOv9, YOLOv11, and YOLOv12 — any DFL-based YOLO model compatible with the CPU TopK optimization path.

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)
    .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")
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# 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.
  - **Type 1, YOLOX**: call once per detection scale, with `obj_conf=`.
  - **Type 1, YOLOv8–v12**: call once, naming the two nodes that feed the final output `Concat` — `bbox` (decoded boxes, usually a `Mul`) and `cls_conf` (a `Sigmoid`). They already cover every detection scale. Read the node names from the ONNX graph.
  - **Type 2**: call once per detection scale.

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

© Copyright 2026 DEEPX All Rights Reserved.

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.

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` (`str`): Observer override applied during re-calibration. One of `"minmax"`, `"ema"`, `"iqr"`.
  - `dataset_path` (`str`): Override the calibration dataset path embedded in the checkpoint.
  - `enhanced_scheme`: Manual Q-PRO scheme selection. Mutually exclusive with `use_q_pro`.

```
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
)
```

`qmaster` *(New in v2.5.0 — replaces the former `qat_*` arguments)*

- **Type**: `Optional[QMasterConfig]`
- **Default**: `None` (PTQ)
- **Description**: Quantization-Aware Training (QAT) configuration object. When omitted, `dx_com.compile()` runs normal PTQ (unless the JSON config supplies a `qmaster` block, which still auto-selects QAT). Import `QMasterConfig` from the top-level `dx_com` package.
- **Fields**:
  - `config` (`Optional[dict]`): QAT training hyperparameters — the same keys as the JSON `qmaster` block (see Q-Master (QAT)).
  - `skip_training` (`bool` , default `False`): Skip the training loop and run compilation only. Use with `resume_from_checkpoint`.
  - `resume_from_checkpoint` (`Optional[str]`): Path to a saved `qat_checkpoint.qxnn` to load trained weights before compilation.

```
from dx_com import QMasterConfig

# Train + compile
```


© Copyright 2026 DEEPX All Rights Reserved.

```
qmaster = QMasterConfig(config={"epochs": 30, "lr": 1e-5, "use_kd": True})

# Compile only, reusing a trained checkpoint
qmaster = QMasterConfig(
    skip_training=True,
    resume_from_checkpoint="output/qat_checkpoint/qat_checkpoint.qxnn",
)
```

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

verbose *(New in v2.5.0)*

- **Type**: `bool`
- **Default**: `False`
- **Description**: Expand masked error messages to include the origin type, the original message, and one bounded direct cause. The Python equivalent of the CLI `--verbose` flag.

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 run quantization on GPU (`use_gpu=True` , the default), 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

3.6.4 Common Errors and Troubleshooting

© Copyright 2026 DEEPX All Rights Reserved.

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: an OS outside the list in System Requirements of DX-COM
7	UbuntuVersionError	The installed Ubuntu version is outside the supported range
8	LDDVersionError	The installed `ldd` version is unsupported

© Copyright 2026 DEEPX All Rights Reserved.

No	Error Type	Description & Conditions
9	RamSizeError	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: `libGL.so.1` cannot be loaded (on Debian/Ubuntu, install `libgl1`)
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

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
ArgMax	Only supported if all of the following hold: - It is the final operation in the network - 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` < 16 - `dilations` = [1, 1], except a 3×3 kernel with `dilations` = [2, 2] - Only constant weights are supported **5D input (3 spatial dims):** additionally requires - `group` = 1 - `dilations` = [1, 1, 1] - `auto_pad` = `NOTSET` - Weight must be a compile-time constant Otherwise runs on CPU.
ConvTranspose	- `dilations` = [1, 1] - `output_padding` = [0, 0] - `pads` ≤ 14 - `strides` ∈ [2, 8] and symmetric (`strides[0]` =

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
	`strides[1]`) - `kernel_shape` < 16 - `group` = 1 **5D input (3 spatial dims):** additionally requires - `group` = 1 - `dilations` = [1, 1, 1] - `output_padding` = [0, 0, 0] - `auto_pad` = `NOTSET` - No `output_shape` attribute Otherwise runs on CPU.
Div	Supported as: - Constant scalar division - Input normalization - Part of `Softmax` - Part of `LayerNorm`
Dropout	Removed during inference
Einsum	Only supported for **two-input** equations that reduce to a (batched) matrix multiplication: - Exactly one contracted (summed) index shared between the two inputs - Each input has exactly one non-batch, non-contracted free index, and it must appear in the output - Any leading indices identical across both inputs and the output are treated as batch dimensions Examples: `ij,jk->ik` , `bij,bjk->bik` , `bik,bjk->bij` Single-input equations (e.g. diagonal, transpose, reduction) and equations with three or more inputs are not supported.
Erf	Only supported as part of `GELU`
Flatten	No restrictions
Gather	Supported when `indices` is a 0-D, 1-D, or 2-D tensor. Examples:

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
	- Scalar index: `indices = [0]` to select first element - 1-D index: `indices = [0, 2, 5]` to select multiple elements - 2-D index (**New in v2.5.0**): `indices = [[0, 2], [1, 3]]` to select a 2-D block of elements
Gemm	No restrictions
GlobalAveragePool	No restrictions
Identity	No restrictions
LayerNormalization	Only supported when normalizing over the last dimension: - `axis` = -1 (or the index of the last dimension) - `Scale` and `B` have shape [D], where D is the size of the last dimension (`B` is optional) - D < 4080; otherwise runs on CPU Not supported: - Normalizing over multiple dimensions (e.g., PyTorch `nn.LayerNorm([C, H, W])`) LayerNorm built from `ReduceMean`, `Sub`, `Pow`, `Sqrt`, `Div`, `Mul`, and `Add` is also supported when the mean and variance are computed over a single dimension (the last dimension, or the channel dimension of an NCHW tensor such as ConvNeXt `LayerNorm2d`). If they are computed over multiple dimensions, it runs on CPU.
MatMul	No restrictions
MaxPool	- `kernel_shape` < 16 - `strides` < 16
Mul	Supported as: - Element-wise multiplication - Constant scalar multiplication - Input normalization

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
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` } **Upsampling** - scale > 1 - `linear` - $s \in \mathbb{Z}, s \geq 2, s \notin \{p : p \text{ is prime}, p > 8\}$ - i.e. integer factors ≥ 2; primes > `8` (`11` , `13` , `17` , …) are not supported, while non-prime factors such as `9` , `10` are supported - `nearest` - $s \in \{2^k : k \geq 1\}$ - powers of two only (`2` , `4` , `8` , `16` , …) **Downsampling** - scale < 1 - `linear` - `s = 0.5` - only a scale factor of `0.5` is supported - `nearest` - not supported
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

© Copyright 2026 DEEPX All Rights Reserved.

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

© Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
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:

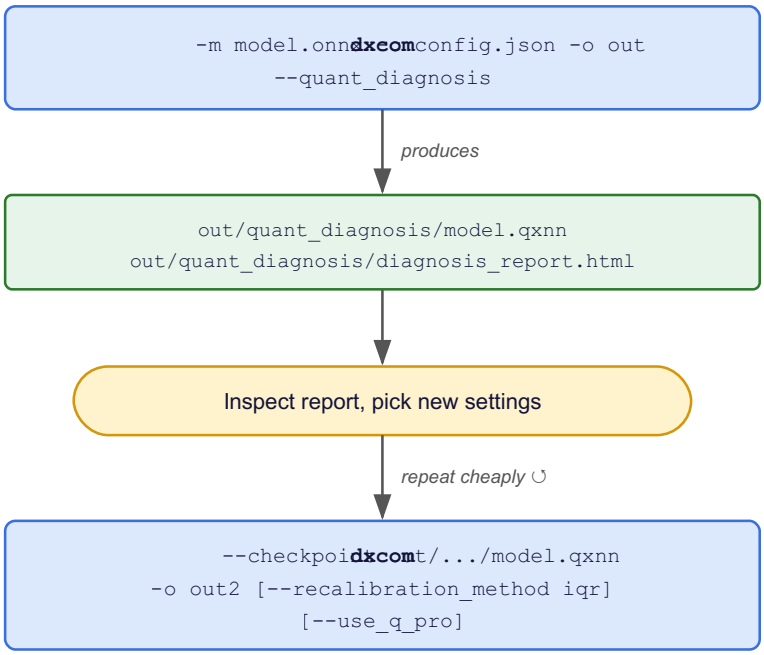
- **Quantization Diagnosis (`quant_diagnosis`)** — generates an HTML report of per-region quantization quality and a reusable `.qxnn` checkpoint.
- **QXNN Resume** — re-runs quantization from a `.qxnn` checkpoint, skipping the earlier compile phases.

> **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.

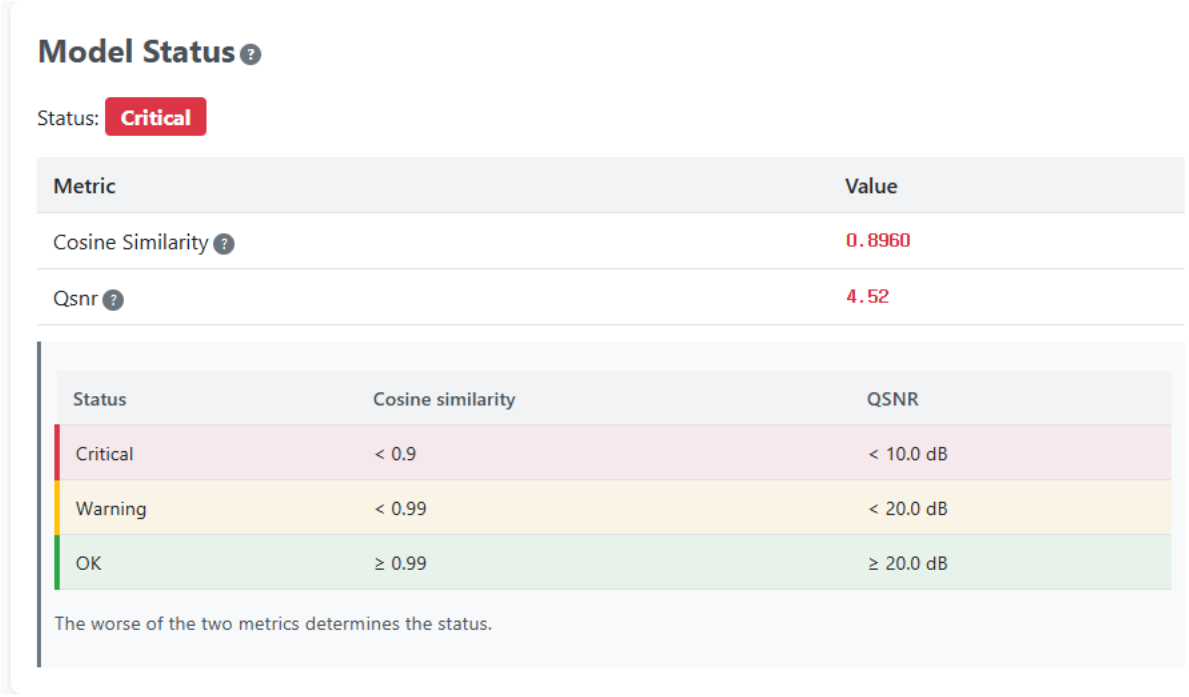
© Copyright 2026 DEEPX All Rights Reserved.

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

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

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)

© Copyright 2026 DEEPX All Rights Reserved.

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.

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

> **Not the QAT checkpoint**
>
> The `qat_checkpoint.qxnn` written by QAT is a different artifact. Pass it only through `QMasterConfig(resume_from_checkpoint=...)` (see Quantization-Aware Training (QAT)), not through `--checkpoint` / `checkpoint=` .

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(
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    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

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

© Copyright 2026 DEEPX All Rights Reserved.

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`).

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 Q-Master (Quantization-Aware Training)

Q-Master is the DX-COM feature that runs **Quantization-Aware Training (QAT)**. It fine-tunes the model while simulating INT8 quantization, so it can recover accuracy that post-training quantization (PTQ) alone cannot.

In short: add a `qmaster` block to your PTQ JSON, run `dxcom` as usual, and get a fine-tuned `.dxnn`. You do not need labels, a separate training script, or a different command.

Q-Master reuses the **same JSON configuration** as a normal compile. Add a `qmaster` block and DX-COM switches from PTQ to Q-Master automatically — no extra command-line flag is needed.

> **Version Support**
>
> Q-Master is available from **DX-COM v2.4.0**. From **v2.5.0** the Python API takes `qmaster=QMasterConfig(...)`, and Segmentation Q-Master is available. See Version History.

5.2.1 When to Use Q-Master

```
Compile with PTQ  ─▶  Accuracy acceptable?  ── yes ─▶  Done
                              │ no
                              ▼
        Tune PTQ first (02_07 Quantization Tuning Workflow)
                              │ still not enough
                              ▼
                    Use Q-Master (this chapter)
```

	PTQ	Q-Master (QAT)
Needs training data	Calibration images only (about 100)	Training images (hundreds to thousands)
Needs a GPU	No	**Yes** (CUDA GPU, default `cuda:0`)
Compile time	Minutes	Minutes to hours, depending on model and data size
Changes model weights	No	Yes (fine-tuned)

© Copyright 2026 DEEPX All Rights Reserved.

	PTQ	Q-Master (QAT)
Typical use	First attempt	Accuracy still below target after PTQ tuning

Checklist before you start

- [ ] The model already compiles with PTQ using the same JSON config.
- [ ] A CUDA GPU is available to PyTorch (`python -c "import torch; print(torch.cuda.is_available())"` prints `True`). To pick a specific GPU, set `CUDA_VISIBLE_DEVICES` and keep `device` at `"cuda:0"` .
- [ ] A folder of representative images is set in `default_loader.dataset_path` (the images are searched recursively by `file_extensions`). Use images similar to your real input, **not** the test set you evaluate on.
- [ ] You have a way to measure accuracy before and after (compare the PTQ `.dxnn` with the Q-Master `.dxnn` on the same evaluation set).

5.2.2 Quick Start

1. Add a `qmaster` block to your config

Start from the JSON you already use for PTQ and append a `qmaster` block. Example for an ImageNet classifier:

```
{
 "inputs": { "input": [1, 3, 224, 224] },
 "calibration_num": 100,
 "calibration_method": "ema",
 "default_loader": {
  "dataset_path": "/datasets/ILSVRC2012/train",
  "file_extensions": ["jpeg", "jpg", "png", "JPEG"],
  "preprocessings": [
   { "resize": { "mode": "default", "size": 256 } },
   { "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] } },
   { "transpose": { "axis": [2, 0, 1] } },
   { "expandDim": { "axis": 0 } }
  ]
 },
 "qmaster": {
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    "batch_size": 16,
    "train_limit": 2000,
    "val_limit": 200,
    "epochs": 10
  }
}
```

One preprocessing pipeline

`inputs`, `calibration_*`, and `default_loader` work exactly as in JSON File Configuration. `default_loader.preprocessings` is the single source of truth for both calibration and Q-Master training data — do **not** redefine it inside `qmaster`.

Apply `convertColor` again at inference

`convertColor` is not integrated into the NPU graph (it is listed under `Skipped nodes` in `compiler.log`). Your inference code must apply the same `BGR2RGB` conversion before feeding the compiled model; otherwise inference sees a different channel order from calibration and training, and accuracy drops without any error.

2. Compile

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

3. Check the result

- `output/<model>.dxnn` — the compiled model.
- `output/qat_checkpoint/qat_checkpoint.qxnn` — the best checkpoint (lowest validation loss).
- `compiler.log` — starts with a **Q-MASTER CONFIG** box listing the values actually used. **Always check it** (see Pitfalls).

4. What to expect

- Training time grows with `epochs × train_limit` and model size. Run the smoke test below first to estimate speed.
- Q-Master does not guarantee a gain. Compare accuracy against your PTQ result; if it does not improve, see Troubleshooting.
- The original ONNX file is never modified.

© Copyright 2026 DEEPX All Rights Reserved.

5. Smoke test first (recommended)

Before a long run, set `"fast_run": true` inside `qmaster`. It trains for 1 epoch × 1 batch to prove the pipeline works end to end. The resulting accuracy is **not meaningful**; remove `fast_run` for the real run.

5.2.3 How Q-Master Works

A Q-Master compile runs three steps:

1. **Calibration** — estimates initial quantization parameters (shared with PTQ).
2. **Training (Stage 1)** — fine-tunes the model with simulated quantization. The best checkpoint (lowest validation loss) is saved to `qat_checkpoint/qat_checkpoint.qxnn`. Stage 1 trains at the `batch_size` you set.
3. **Compilation (Stage 2)** — converts the trained weights into the NPU binary (`*.dxnn`), always at batch size 1.

Knowledge Distillation (KD)

By default Q-Master uses the original floating-point (FP32) model as a **teacher**, and trains the quantized model (the **student**) to reproduce the teacher's output. With the defaults (`kd_loss = "mse"`, `kd_alpha = 1.0`) training uses the teacher output only and **no ground-truth labels**, so unlabeled images work.

5.2.4 Recipes

Copy the closest recipe and change only what you need. All keys are described in the Parameter Reference.

A. Default (no labels needed)

```
"qmaster": { "batch_size": 16, "train_limit": 2000, "val_limit": 200, "epochs": 10 }
```

Teacher-only MSE distillation. Good first attempt for any model. Images need no labels and no particular folder layout. If `train_limit + val_limit` is larger than the number of images found, all available images are used.

B. Classification with labels (KD + cross-entropy)

© Copyright 2026 DEEPX All Rights Reserved.

```
"qmaster": {
  "batch_size": 16,
  "train_limit": 5000,
  "val_limit": 500,
  "criterion": "cross_entropy",
  "kd_loss": "kl_div",
  "kd_temperature": 2.0,
  "kd_alpha": 0.5
}
```

Blends the teacher signal (`kd_alpha`) with the label loss (`1 - kd_alpha`). Requires an ImageFolder layout (`<dataset_path>/<class_name>/<image>`).

> ⚠ **Labels come from folder names**
>
> Class indices are built from the class folders **found in the subset actually loaded** (`train_limit + val_limit`). If a class is missing from that subset, label indices shift and the label loss trains toward the wrong classes. Make sure every class is present, or use recipe A.

C. Embedding / encoder models (e.g. CLIP image encoder)

```
"qmaster": { "encoder_mode": true, "kd_alpha": 1.0 }
```

Tracks **cosine similarity** instead of classification accuracy.

D. Zero-shot classification (e.g. CLIP)

```
"qmaster": {
  "zeroshot_weights_path": "/path/to/text_embedding.npy",
  "use_kd": false
}
```

Zero-shot cannot be combined with `use_kd` or `encoder_mode` .

E. Segmentation (v2.5.0+)

```
"qmaster": {
  "task_type": "segmentation",
  "seg_root": "/path/to/dataset_root",
  "seg_pairs_train": "/path/to/dataset_root/lists/train.txt",
  "seg_pairs_val": "/path/to/dataset_root/lists/val.txt",
  "seg_ignore_index": 255,
  "seg_binary": true,
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  "use_kd": false
}
```

See Segmentation Parameters.

F. Out of GPU memory

```
"qmaster": { "batch_size": 4, "gradient_accumulation_steps": 4 }
```

Effective batch size stays `4 × 4 = 16`.

If CUDA runs out of memory at your `batch_size`, DX-COM automatically retries Stage 1 at batch size 1. This succeeds but is **much slower**, so lower `batch_size` yourself to stay fast.

G. Re-compile without re-training / continue training

Use `stage1_mode` — see Re-using a Checkpoint.

5.2.5 Parameter Reference

Every key is optional. Omitted keys use the defaults below.

Data and Device

Parameter	Default	Description
`batch_size`	`1`	Training batch size for Stage 1. Use `> 1` (e.g. `16`) for faster, more stable training. Models whose inputs have no batch axis of 1 (e.g. fixed-length text encoders) automatically fall back to `1`.
`num_workers`	`4`	DataLoader workers.
`train_limit`	`500`	Number of training images used. **The default is small (smoke-test scale); raise it for real training.**

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Default	Description
`val_limit`	`50`	Number of validation images, reserved **before** training images are taken from the same pool.
`train_drop_last` / `val_drop_last`	`true` / `false`	Drop the last incomplete batch.
`device`	`"cuda:0"`	Training device. Select the physical GPU with `CUDA_VISIBLE_DEVICES`.
`train_cpu_fp`	`false`	Also train the CPU (non-NPU) parts of the model and write the trained weights back into the compiled model. When `false`, CPU subgraph weights stay frozen.

Training Loop

Parameter	Default	Description
`epochs`	`30`	Maximum epochs.
`lr`	`1e-5`	Learning rate. Lower it (e.g. `5e-6`) if accuracy gets worse from the first epoch.
`optimizer`	`"adamw"`	`"adamw"` or `"sgd"`.
`weight_decay`	`1e-4`	Optimizer weight decay.
`scheduler`	`null`	`null`, `"cosine"`, or `"step"`.
`scheduler_step_size`	`max(1, (epochs - warmup_epochs) // 3)`	Epoch interval for `"step"` only.
`warmup_epochs`	`0`	Linear LR warmup epochs.
`max_grad_norm`	`1.0`	Gradient clipping.

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Default	Description
use_amp	true	Mixed-precision training.
gradient_accumulation_steps	1	Effective batch = batch_size × this.
early_stopping_patience	5	Stop after N epochs without improvement on validation loss. null disables. Requires validation data.
early_stopping_delta	1e-3	Minimum improvement counted as progress.
save_best_model	true	Save the checkpoint with the lowest validation loss.
freeze_bn_after	null	From this epoch on, freeze BatchNorm **and** stop updating activation quantization ranges. See Pitfalls.

Loss and Knowledge Distillation

The loss is kd_alpha × KD loss + (1 − kd_alpha) × task loss.

Parameter	Default	Description
use_kd	true	Use the FP teacher.
kd_loss	"mse"	"mse", "kl_div", "cosine", "bce", "smooth_l1", "ciou", "giou".
kd_alpha	1.0	KD weight in [0, 1]. At 1.0 the task loss is **not computed**.
kd_temperature	4.0	Softening temperature. Used only by "kl_div" and "bce".
kd_box_format	"xyxy"	

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Default	Description
		Box format for `"ciou"` / `"giou"`: `"xyxy"`, `"cxcywh"`, `"xywh"`.
`criterion`	`"mse"`	Task loss: `"mse"` or `"cross_entropy"` (`"seg_ce"` for segmentation). Used only when `kd_alpha < 1.0` or `use_kd = false`. `"cross_entropy"` switches to the labeled (ImageFolder) loader.
`label_smoothing`	`0.0`	Cross-entropy only.

Choosing the loss:

Goal	Settings
No labels available	defaults (`kd_alpha = 1.0`, `kd_loss = "mse"`)
Classification with labels	`criterion = "cross_entropy"`, `kd_loss = "kl_div"`, `kd_alpha = 0.5–0.7`
No teacher	`use_kd = false`, plus a `criterion`

Mode Switches

Parameter	Default	Description
`encoder_mode`	`false`	Track cosine similarity instead of accuracy (embedding/ encoder models). Does not disable the task loss by itself; keep `kd_alpha = 1.0`.
`zeroshot_weights_path`	—	`.npy` text-embedding file for zero-shot models.
`stage1_mode`	`"fresh"`	`"fresh"`, `"resume"`, `"skip"`. See Re-using a Checkpoint.

© Copyright 2026 DEEPX All Rights Reserved.

Parameter	Default	Description
`fast_run`	`false`	Smoke test: 1 epoch × 1 batch, 100 train / 10 val images. Not accuracy-meaningful.

Segmentation Parameters

Parameter	Default	Description
`task_type`	`"classification"`	Set `"segmentation"` for dense `(image, mask)` data.
`seg_root`	—	Root directory for the pair lists.
`seg_pairs_train` / `seg_pairs_val`	—	Text files, one `<image_path> <mask_path>` per line, relative to `seg_root`.
`seg_ignore_index`	`255`	Label value excluded from loss and metric.
`seg_binary`	`true`	Binary (foreground/background) segmentation. Set `false` for multi-class.

With `task_type = "segmentation"`, `criterion` defaults to `"seg_ce"`.
Masks are not passed through `preprocessings`; only the image is.

5.2.6 Re-using a Checkpoint

Training can take a long time. `stage1_mode` controls what happens when `output/qat_checkpoint/qat_checkpoint.qxnn` already exists:

`stage1_mode`	Behavior	Use when
`"fresh"` (default)	Train from the start and overwrite the checkpoint.	First run.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

`stage1_mode`	Behavior	Use when
`"resume"`	Load the checkpoint weights and continue training (optimizer and epoch counter restart). If no checkpoint exists, trains from scratch with a warning.	Fine-tune further with a lower `lr`.
`"skip"`	Skip training and run Stage 2 (compilation) only. The checkpoint **must** exist.	Re-generate the `.dxnn` (e.g. after a compiler update).

```
"qmaster": { "stage1_mode": "skip" }
```

To start from a checkpoint made elsewhere, copy it to `<output_dir>/qat_checkpoint/qat_checkpoint.qxnn` first.

5.2.7 Python API

```
import dx_com

# A qmaster block in the config JSON triggers Q-Master.
dx_com.compile(model="model.onnx", config="config_with_qmaster.json", output_dir="output/")
```

Configure Q-Master in the JSON

Set all Q-Master options in the JSON `qmaster` block, as in the Quick Start. To re-compile from a checkpoint, use `stage1_mode` in the JSON (see Re-using a Checkpoint).

Changed in v2.5.0

The flat arguments `quantization_mode`, `qat_config`, `qat_skip_training`, and `qat_resume_from_checkpoint` were replaced by the single `qmaster` argument. Do not pass any of the four removed arguments.

© Copyright 2026 DEEPX All Rights Reserved.

> **`fast_run` is a config key**
>
> Set `"fast_run": true` inside the `qmaster` block; it is not a `compile()` argument.

> **Not interchangeable with `quant_diagnosis`**
>
> The `.qxnn` from `quant_diagnosis` (see Quantization Tuning Workflow) is a different file from the Q-Master `qat_checkpoint.qxnn`.

5.2.8 Output Files

File	Description
`<output_dir>/*.dxnn`	Compiled NPU binary (the deliverable).
`<output_dir>/qat_checkpoint/qat_checkpoint.qxnn`	Best training checkpoint. Input for `stage1_mode = "resume"/"skip"`.
`<output_dir>/compiler.log`	Full log, including the **Q-MASTER CONFIG** box and per-epoch metrics.

5.2.9 Pitfalls

> **Unknown keys are silently ignored**
>
> A misspelled key (e.g. `learning_rate`) does **not** raise an error; the default is used instead. Open `compiler.log` and check the **Q-MASTER CONFIG** box to confirm the values that were actually applied.

> **Confirm Q-Master actually ran**
>
> After compiling, open `compiler.log` and check that the **Q-MASTER CONFIG** box and per-epoch training lines appear, and that no `falling back` message is present. If training did not run, the result is plain PTQ.

© Copyright 2026 DEEPX All Rights Reserved.

Dataset path falls back silently

If `default_loader.dataset_path` is missing or invalid, DX-COM does **not** fail. It falls back to a built-in default ImageNet location, so training may silently use unintended data. Verify the dataset path in the log.

Best checkpoint is chosen by validation loss

The saved checkpoint has the lowest **validation loss**, which includes the KD term. It can differ from the checkpoint with the best Top-1 accuracy.

BatchNorm / observer freezing

`freeze_bn_after` freezes BatchNorm **and** stops updating activation quantization ranges from the given epoch. `freeze_bn_after = 0` freezes from the very first epoch (after one warm-up pass), which can make the loss diverge to `nan`. Leave it `null` unless you need it, and then start from a later epoch.

Default `train_limit` is small

`train_limit = 500` is meant for quick checks. Use thousands of images for a real run.

5.2.10 Troubleshooting

Symptom	Likely cause	Action
CUDA out of memory	`batch_size` too large for the GPU	Lower `batch_size`, raise `gradient_accumulation_steps`. The automatic batch-1 fallback works but is slow.
Loss becomes `nan`	`freeze_bn_after = 0`, `lr` too high	Set `freeze_bn_after` to `null` or a later epoch; lower `lr`.
Validation loss is worst at later epochs; best epoch is 1	`lr` too high	Lower `lr`; try `stage1_mode = "resume"` with a smaller `lr`.
Training stops very early		Raise `patience`, raise `val_limit`.

© Copyright 2026 DEEPX All Rights Reserved.

Symptom	Likely cause	Action
	`early_stopping_patience` too small / validation too noisy	
Accuracy not better than PTQ	Too little data / loss mismatch	Raise `train_limit` ; for classification with labels use recipe B.
Label loss trains the wrong classes	A class is missing from the subset	Check class folders; raise `train_limit` ; or use recipe A.
Training used unexpected data	`dataset_path` invalid → silent fallback	Check the dataset path in `compiler.log` .
Settings seem to have no effect	Key misspelled or in the wrong block	Check the **Q-MASTER CONFIG** box in `compiler.log` .
Training metrics are good but `.dxnn` accuracy dropped	Loss in a compiler stage that training cannot see	Compare the `.dxnn` with the original model on the same evaluation set and report the gap to DEEPX.

5.2.11 FAQ

Do I need labeled data? No. The default recipe distills from the FP32 model and uses images only. Labels are needed only for recipe B.

Can I use my own dataset instead of ImageNet? Yes. Point `default_loader.dataset_path` at your images. Use the same preprocessing as in PTQ.

Can I use the calibration images for training? Yes, but more varied images generally give better results. Do not train on the images you use to report final accuracy.

Is the compiled `.dxnn` different from a PTQ one? Same format and runtime. Only the weights and quantization parameters are different.

Training finished but I want to try another setting. Do I need to start over? Use `stage1_mode = "resume"` to continue from the saved checkpoint, or `"fresh"` with a new `output_dir` to start clean.

I only want to rebuild the `.dxnn` (e.g. new DX-COM version). Use `stage1_mode = "skip"` with the existing checkpoint.

© Copyright 2026 DEEPX All Rights Reserved.

5.2.12 Version History

Version	Change
v2.4.0	Q-Master (QAT) added via the JSON `qmaster` block.
v2.5.0	Python API: `qmaster=QMasterConfig(...)`; `qat_*` arguments removed. Segmentation Q-Master (`task_type = "segmentation"`).

5.2.13 Related Pages

- JSON File Configuration — `inputs`, `calibration_*`, `default_loader`.
- Execution of DX-COM — CLI and Python API reference.
- Quantization Tuning Workflow — tune PTQ before using Q-Master.
- Change Log.

© Copyright 2026 DEEPX All Rights Reserved.

6. Pre-Optimize API

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

There are two ways to apply it:

- **Python API** — call `dx_com.pre_optimize()` on a loaded model, then pass the result to `dx_com.compile()`.
- **Config JSON** — add a `pre_optimize` block to the compile config, and the passes run automatically inside `dx_com.compile()`.

When to use

Use `pre_optimize()` when CPU post-processing (Sigmoid, DFL decoding, dist2bbox, NMS pre-filtering, etc.) dominates end-to-end latency. It is most effective for YOLO-family **detection / segmentation / pose** models and **RTMDet** 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). `pre_optimize` and `ppu` are **mutually exclusive** in one config — both rewrite detection post-processing, so supply only one. DX-COM 2.5.0 does not reject a config that has both: it compiles with `ppu` and drops `pre_optimize`, or fails later with an unrelated error. The only sign is a console line `falling back to legacy: ValueError: config 'pre_optimize' and 'ppu' are mutually exclusive`.

6.1 How It Works

The detection head of these models runs several operations on the host CPU after NPU inference: Sigmoid, Softmax, DFL decoding, dist2bbox, NMS pre-filtering. When these run

© Copyright 2026 DEEPX All Rights Reserved.

over the full anchor grid (for example, 8400 candidates at 640×640), CPU post-processing can dominate latency and leave the NPU idle.

`pre_optimize()` reorders the graph so that:

1. **TopK selection happens first**, narrowing candidates to `K` (default 300; 512 for RTMDet).
2. **Expensive operations run only on those `K` candidates**.

CPU post-processing cost then scales with `K` instead of the full anchor count.

6.2 Available Passes

Pass	Target models	Tasks	Output
`yolo_dfl_postprocess`	YOLOv8 / v9 / v11 / v12 / v13	detection · segmentation · pose	`[1, 4+C (+mask or +keypoints), K]`
`yolo_no_dfl_postprocess`	YOLO26 (NMS-free)	detection · segmentation · pose	`[1, K, 6 (+mask or +keypoints)]`
`rtmdet_postprocess`	RTMDet S / M / L / X	detection	NMS tensors (original NMS preserved)

Notes:

- For **segmentation**, the mask **prototype** output is preserved unchanged.
- Axis layout differs by family: `yolo_dfl_postprocess` puts channels on axis 1 and `K` on axis 2; `yolo_no_dfl_postprocess` puts `K` on axis 1.
- Box format also differs: `yolo_dfl_postprocess` outputs `xywh` (center x, center y, width, height) followed by per-class scores; `yolo_no_dfl_postprocess` outputs `xyxy` (top-left, bottom-right) followed by a single score and `class_id`.
- `rtmdet_postprocess` rebuilds only the pre-NMS pipeline; the original NMS and all post-NMS logic stay intact.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Changed in v2.5.0

Passes were **renamed**: `yolo_postprocess` → `yolo_dfl_postprocess`, `yolo26_postprocess` → `yolo_no_dfl_postprocess`. The old names now raise a migration hint.

YOLO passes require an explicit **`task`** key (`base` / `seg` / `pose`).

`input_height` / `input_width` are removed — the input size is derived from the model input. Supplying them raises `ValueError`.

New capabilities: **pose** (both YOLO passes), **segmentation on YOLO26** (`yolo_no_dfl_postprocess`), and the new **`rtmdet_postprocess`** pass.

6.3 API Reference

6.3.1 Signature

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

6.3.2 Parameters

Parameter	Type	Required	Description
`model`	`onnx.ModelProto`	**Yes**	Source ONNX model loaded via `onnx.load()`. Must be a **full model** — the network input is used to derive per-scale strides.
`passes`	`dict[str, dict]`	**Yes**	Mapping from pass name to its configuration. Passes are applied in the given order.

© Copyright 2026 DEEPX All Rights Reserved.

6.3.3 Returns

An optimized `onnx.ModelProto`. Pass it directly to `dx_com.compile()` via `model=`, or save it with `onnx.save()`.

6.3.4 Raises

- `KeyError` — unrecognized pass name (old names raise a migration hint), or a missing required configuration key.
- `ValueError` — a tensor name in `layers` cannot be resolved, or `task` validation fails (missing/unknown `task`, or per-layer keys that are required/forbidden for the chosen `task`).

6.4 Configuration Schema

6.4.1 YOLO passes

`yolo_dfl_postprocess` and `yolo_no_dfl_postprocess` target YOLO models exported from Ultralytics. Both take the fields below:

Field	Type	Required	Description
`task`	`str`	**Yes**	`base` (detection), `seg` (segmentation), or `pose` (keypoints). Drives head routing and per-layer key validation.
`layers`	`list[dict]`	**Yes**	One entry per detection scale, mapping tensor roles to Conv-head output tensor names. Entries **must be ordered by stride: 8, 16, 32** (80×80, 40×40, 20×20 at a 640×640 input).
`num_classes`	`int`	`yolo_dfl_postprocess`: **Yes**	

© Copyright 2026 DEEPX All Rights Reserved.

Field	Type	Required	Description
		yolo_no_dfl_postprocess : No (default 80)	Number of classification classes.
topk	int	No (default 300)	Candidates to keep after TopK selection.

Roles in each layers entry:

Role	Description
bbox	Output of the box-regression head Conv: the distances from each anchor point to the four box sides, as 16-bin distributions (yolo_dfl_postprocess) or as direct values (yolo_no_dfl_postprocess). e.g. /model.22/cv2.0/cv2.0.2/Conv_output_0 (YOLOv8n-seg)
cls_conf	Output of the classification head Conv: one score per class, as logits before Sigmoid. e.g. /model.22/cv3.0/cv3.0.2/Conv_output_0 (YOLOv8n-seg)
mask_coeff	Output of the mask-coefficient head Conv: 32 coefficients that combine the prototype masks into each instance mask. e.g. /model.22/cv4.0/cv4.0.2/Conv_output_0 (YOLOv8n-seg)
kpt	Output of the keypoint head Conv, with num_keypoints × values_per_keypoint channels (see kpt_shape). e.g. /model.22/cv4.0/cv4.0.2/Conv_output_0 (YOLOv8n-pose, 17 × 3 = 51 channels)
kpt_shape	Keypoint layout [num_keypoints, values_per_keypoint] . values_per_keypoint is 3 (x, y, visibility), or 2 (x, y) when the model has no visibility output. Ultralytics exports store it in the ONNX metadata as kpt_shape . e.g. [17, 3] (COCO keypoints)

© Copyright 2026 DEEPX All Rights Reserved.

Per-layer roles depend on `task`:

task	required per-layer roles	forbidden roles
`base`	`bbox`, `cls_conf`	`mask_coeff`, `kpt`
`seg`	`bbox`, `cls_conf`, `mask_coeff`	`kpt`
`pose`	`bbox`, `cls_conf`, `kpt` (+ `kpt_shape`)	`mask_coeff`

Validation runs before any graph edit and raises a message prefixed with the pass name. For `pose`, `kpt_shape` must be identical across scales, and `num_keypoints × values_per_keypoint` must equal the keypoint Conv channel count.

6.4.2 RTMDet pass

`rtmdet_postprocess` is detection-only — **no `task` key**. It targets RTMDet models exported from MMDetection.

Field	Type	Required	Description
`cls_layers`	`list[str]`	**Yes**	Classification Conv-head output tensor names, one per scale.
`reg_layers`	`list[str]`	**Yes**	Regression Conv-head output tensor names, one per scale.
`use_exp`	`bool`	**Yes**	`True` if the reg branch applies `Exp`; `False` otherwise.
`nms_boxes_tensor`	`str`	**Yes**	Boxes tensor feeding the original NMS.
`nms_scores_tensor`	`str`	**Yes**	Full per-class score tensor `[1, M, C]` (M = candidates kept by the model's own first `TopK`, e.g. 5000)

© Copyright 2026 DEEPX All Rights Reserved.

Field	Type	Required	Description
			feeding NMS (post-Sigmoid).
`nms_score_tensor`	`str`	**Yes**	Top-K score values tensor (TopK output).
`num_classes`	`int`	No (derived from `cls_layers`)	Number of classification classes.
`topk`	`int`	No (default `512`)	Candidates to keep after TopK selection.

6.5 Usage

6.5.1 Python API

Load the model, apply the pass, then compile. The tensor names below are the real Conv-head output names from models exported with Ultralytics (YOLO) and MMDetection (RTMDet). Find the names for your own model with Identifying Conv Head Tensor Names.

> ⚠ **Example tensor names**
>
> Auto-generated names such as `onnx::Concat_328` are specific to one export. They change with the exporter version, export options, and opset. Do not copy them. Look up the names in your own ONNX file with the script.

YOLOv8n Detection

```
import onnx
import dx_com

model = onnx.load("yolov8n.onnx")
optimized = dx_com.pre_optimize(model, passes={
   "yolo_dfl_postprocess": {
      "task": "base",
      "layers": [
         {"bbox": "onnx::Concat_328",
          "cls_conf": "onnx::Concat_335"},
         {"bbox": "onnx::Concat_343",
          "cls_conf": "onnx::Concat_350"},
```


© Copyright 2026 DEEPX All Rights Reserved.

```
         {"bbox": "onnx::Concat_358",
          "cls_conf": "onnx::Concat_365"},
      ],
      "num_classes": 80,   # required for the DFL pass
      "topk": 300,
   },
})
# Output: [1, 84, 300]   (4 bbox_xywh + 80 cls scores, K = 300)

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

YOLOv8n Segmentation

```
import onnx
import dx_com

model = onnx.load("yolov8n-seg.onnx")
optimized = dx_com.pre_optimize(model, passes={
   "yolo_dfl_postprocess": {
      "task": "seg",
      "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,
   },
})
# Output 0: [1, 116, 300]       (4 bbox + 80 cls + 32 mask_coeff, K = 300)
# Output 1: [1, 32, 160, 160]    (mask prototype, preserved unchanged)

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

YOLOv8n Pose

```
import onnx
import dx_com

model = onnx.load("yolov8n-pose.onnx")
optimized = dx_com.pre_optimize(model, passes={
   "yolo_dfl_postprocess": {
      "task": "pose",
```


ⓒ 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",
          "kpt": "/model.22/cv4.0/cv4.0.2/Conv_output_0", "kpt_shape": [17, 3]},
         {"bbox": "/model.22/cv2.1/cv2.1.2/Conv_output_0",
          "cls_conf": "/model.22/cv3.1/cv3.1.2/Conv_output_0",
          "kpt": "/model.22/cv4.1/cv4.1.2/Conv_output_0", "kpt_shape": [17, 3]},
         {"bbox": "/model.22/cv2.2/cv2.2.2/Conv_output_0",
          "cls_conf": "/model.22/cv3.2/cv3.2.2/Conv_output_0",
          "kpt": "/model.22/cv4.2/cv4.2.2/Conv_output_0", "kpt_shape": [17, 3]},
      ],
      "num_classes": 1,
      "topk": 300,
   },
})
# Output: [1, 56, 300]   (4 bbox + 1 cls + 17*3 kpt, K = 300)

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

YOLO26n Detection

```
import onnx
import dx_com

model = onnx.load("yolo26n.onnx")
optimized = dx_com.pre_optimize(model, passes={
   "yolo_no_dfl_postprocess": {
      "task": "base",
      "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,   # optional — defaults to 80 for the no-DFL pass
      "topk": 300,
   },
})
# Output: [1, 300, 6]   (bbox_xyxy + score + class_id, K = 300)

dx_com.compile(model=optimized, config="yolo26n.json", output_dir="./out")
```

YOLO26n Segmentation

```
import onnx
import dx_com
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
model = onnx.load("yolo26n-seg.onnx")
optimized = dx_com.pre_optimize(model, passes={
   "yolo_no_dfl_postprocess": {
      "task": "seg",
      "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",
          "mask_coeff": "/model.23/one2one_cv4.0/one2one_cv4.0.2/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",
          "mask_coeff": "/model.23/one2one_cv4.1/one2one_cv4.1.2/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",
          "mask_coeff": "/model.23/one2one_cv4.2/one2one_cv4.2.2/Conv_output_0"},
      ],
      "num_classes": 80,
      "topk": 300,
   },
})
# Output 0: [1, 300, 38]        (bbox_xyxy + score + class_id + 32 mask_coeff, K = 300)
# Output 1: [1, 32, 160, 160]    (mask prototype, preserved unchanged)

dx_com.compile(model=optimized, config="yolo26n-seg.json", output_dir="./out")
```

YOLO26n Pose

```
import onnx
import dx_com

model = onnx.load("yolo26n-pose.onnx")
optimized = dx_com.pre_optimize(model, passes={
   "yolo_no_dfl_postprocess": {
      "task": "pose",
      "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",
          "kpt": "/model.23/one2one_cv4_kpts.0/Conv_output_0", "kpt_shape": [17, 3]},
         {"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",
          "kpt": "/model.23/one2one_cv4_kpts.1/Conv_output_0", "kpt_shape": [17, 3]},
         {"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",
          "kpt": "/model.23/one2one_cv4_kpts.2/Conv_output_0", "kpt_shape": [17, 3]},
      ],
      "num_classes": 1,
      "topk": 300,
   },
})
# Output: [1, 300, 57]   (bbox_xyxy + score + class_id + 17*3 kpt, K = 300)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
dx_com.compile(model=optimized, config="yolo26n-pose.json", output_dir="./out")
```

RTMDet-S Detection

```
import onnx
import dx_com

model = onnx.load("rtmdet-s.onnx")
optimized = dx_com.pre_optimize(model, passes={
   "rtmdet_postprocess": {
      "cls_layers": [
         "/bbox_head/rtm_cls.0/Conv_output_0",
         "/bbox_head/rtm_cls.1/Conv_output_0",
         "/bbox_head/rtm_cls.2/Conv_output_0",
      ],
      "reg_layers": [
         "/bbox_head/rtm_reg.0/Conv_output_0",
         "/bbox_head/rtm_reg.1/Conv_output_0",
         "/bbox_head/rtm_reg.2/Conv_output_0",
      ],
      "use_exp": False,            # True for models whose reg branch has Exp (e.g. rtmdet-m)
      "nms_boxes_tensor": "/Gather_6_output_0",
      "nms_scores_tensor": "/Gather_7_output_0",
      "nms_score_tensor": "/TopK_output_0",
      "topk": 512,
      # num_classes optional — derived from cls layer channels if omitted
   },
})
# NMS and all post-NMS logic preserved; only the pre-NMS pipeline is rebuilt.

dx_com.compile(model=optimized, config="rtmdet-s.json", output_dir="./out")
```

6.5.2 Config JSON

Add a `pre_optimize` block to the compile config. It is a **list** of single-key `{pass_name: config}` dicts, applied in order — mirroring the Python API `passes={name: cfg}`. `dx_com.compile()` then applies the passes automatically to the raw ONNX before compilation. The pass config is identical to the Python API examples above.

```
{
 "inputs": { "images": [1, 3, 640, 640] },
 "calibration_num": 100,
 "calibration_method": "ema",
 "default_loader": {
  "dataset_path": "./calibration_dataset",
  "preprocessings": [
   {"resize": {"width": 640, "height": 640}},
```


© Copyright 2026 DEEPX All Rights Reserved.

```
      {"div": {"x": 255.0}},
      {"convertColor": {"form": "BGR2RGB"}},
      {"transpose": {"axis": [2, 0, 1]}},
      {"expandDim": {"axis": 0}}
    ]
  },
  "pre_optimize": [
    {
      "yolo_dfl_postprocess": {
        "task": "base",
        "layers": [
          {"bbox": "onnx::Concat_328",
           "cls_conf": "onnx::Concat_335"},
          {"bbox": "onnx::Concat_343",
           "cls_conf": "onnx::Concat_350"},
          {"bbox": "onnx::Concat_358",
           "cls_conf": "onnx::Concat_365"}
        ],
        "num_classes": 80,
        "topk": 300
      }
    }
  ]
}
```

For other tasks/models, swap the `pre_optimize` block for the matching pass config above (`seg` adds `mask_coeff`, `pose` adds `kpt` + `kpt_shape`, `rtmdet_postprocess` uses `cls_layers` / `reg_layers` / `use_exp` / `nms_*_tensor`). Then compile as usual — the pass runs inside compile:

```
dx_com.compile(model="yolov8n.onnx", config="config.json", output_dir="./out")
```

Output artifact

The optimized graph keeps the **source file name** — the artifact is `<stem>.dxnn`, the same as a normal compile. On the file-path entry the graph name is set to the source file stem; a directly-passed `ModelProto` is left unchanged. No extra ONNX dump is written.

6.6 Identifying Conv Head Tensor Names

Every YOLO pass needs a `layers` list. It has one entry per detection scale, and each entry maps a **role** to the **output tensor name of the last Conv** in that head branch. That Conv

© Copyright 2026 DEEPX All Rights Reserved.

output is the raw head output, before any Sigmoid, DFL, or box decoding. `pre_optimize` takes the graph apart at these tensors and rebuilds the post-processing after them.

Tensor names depend on the model and on the exporter version, so always look them up in your own model. Do not copy names from another model.

6.6.1 How to recognize a head Conv

A head Conv meets all of the following:

1. **It is the last Conv of its branch.** No activation (Sigmoid, SiLU, Mul) follows it. Its output feeds only `Concat`, `Reshape`, or `Transpose`.
2. **Its output is 4D, `[1, C, H/s, W/s]`, with stride `s` = 8, 16, or 32.**
3. **Its channel count `C` matches a role:**
   - `bbox`: `64` (4 sides × 16 DFL bins) for `yolo_dfl_postprocess`, `4` for `yolo_no_dfl_postprocess`
   - `cls_conf`: `num_classes`
   - `mask_coeff`: `32`
   - `kpt`: `num_keypoints × values_per_keypoint` (e.g. 17 × 3 = 51)

Two Convs look similar but are **not** heads, and the script skips them:

- The DFL Conv (1 channel, after a Softmax), which decodes the 64 bbox channels.
- The segmentation **prototype** Conv (32 channels at stride 4, e.g. 160×160). It has the same channel count as `mask_coeff` but a different resolution, and `pre_optimize` keeps it unchanged.

Notes:

- Use the YOLO26 **`one2one_*`** branches (the NMS-free head), not the `one2many` training branches, if both exist in the graph.
- If the model has no `kpt_shape` metadata, read it from the `Reshape` that follows the `Concat` of the three `kpt` outputs: its target shape is `[1, num_keypoints, values_per_keypoint, -1]`, e.g. `[1, 17, 3, -1]`.

6.6.2 Find them with a script (YOLO)

The script below scans the model and prints a ready-to-use `pre_optimize` config-JSON block. For Ultralytics exports it reads `task`, the class count, and `kpt_shape` from the ONNX metadata, so in most cases you only set `MODEL` and `PASS`.

```
import ast
import json
```


© Copyright 2026 DEEPX All Rights Reserved.

```
from collections import defaultdict

import onnx
from onnx import shape_inference

# ---- set these for your model ----
MODEL = "model.onnx"
PASS = "yolo_dfl_postprocess"  # YOLOv8/v9/v11/v12/v13; "yolo_no_dfl_postprocess" for YOLO26
# None = read from the Ultralytics ONNX metadata
TASK = None                  # "base" | "seg" | "pose"
NUM_CLASSES = None           # 80 = COCO, 1 = single-class (e.g. person pose)
KPT_SHAPE = None             # pose: [num_keypoints, values_per_keypoint], e.g. [17, 3]
MASK_CH = 32                 # seg: mask coefficient channels
# ---------------------------------


model = shape_inference.infer_shapes(onnx.load(MODEL))
g = model.graph
meta = {p.key: p.value for p in model.metadata_props}


def from_meta(key):
    assert key in meta, f"'{key}' is not in the ONNX metadata: set it manually above"
    return meta[key] if key == "task" else ast.literal_eval(meta[key])


if TASK is None:
    TASK = {"detect": "base", "segment": "seg", "pose": "pose"}.get(from_meta("task"))
    assert TASK, f"unsupported task '{meta['task']}'"
if NUM_CLASSES is None:
    NUM_CLASSES = len(from_meta("names"))
if TASK == "pose" and KPT_SHAPE is None:
    KPT_SHAPE = list(from_meta("kpt_shape"))

shape = {v.name: [d.dim_value for d in v.type.tensor_type.shape.dim]
         for v in list(g.input) + list(g.value_info) + list(g.output)}
consumers = defaultdict(set)
for n in g.node:
    for i in n.input:
        consumers[i].add(n.op_type)

# Channel count of each head Conv, per task
roles = {"bbox": 64 if PASS == "yolo_dfl_postprocess" else 4, "cls_conf": NUM_CLASSES}
if TASK == "seg":
    roles["mask_coeff"] = MASK_CH
elif TASK == "pose":
    roles["kpt"] = KPT_SHAPE[0] * KPT_SHAPE[1]
assert len(set(roles.values())) == len(roles), f"channel collision {roles}: pick heads in Netron"

# Head Conv = 4D output on an FPN scale (stride 8/16/32) with no activation after it
in_h = shape[g.input[0].name][2]
assert in_h, "input H is dynamic: re-export the model with a fixed input size"
heads = defaultdict(lambda: defaultdict(list))  # stride -> channels -> [tensor names]
```


ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
for n in g.node:
    out = n.output[0]
    d = shape.get(out)
    if n.op_type != "Conv" or not d or len(d) != 4 or not d[2] or in_h % d[2]:
        continue
    stride, ops = in_h // d[2], consumers[out]
    if stride in (8, 16, 32) and ops <= {"Concat", "Reshape", "Transpose"}:
        heads[stride][d[1]].append(out)
assert heads, "no head Conv found: check PASS and the model input"

layers = []
for stride in sorted(heads):  # layers must be in stride order (8 -> 16 -> 32)
    layer = {}
    for role, ch in roles.items():
        names = heads[stride].get(ch, [])
        assert len(names) == 1, f"stride {stride}: expected 1 {role} Conv ({ch}ch), got {names}"
        layer[role] = names[0]
    if TASK == "pose":
        layer["kpt_shape"] = KPT_SHAPE
    layers.append(layer)

cfg = {"task": TASK, "layers": layers, "num_classes": NUM_CLASSES}
print(json.dumps({"pre_optimize": [{PASS: cfg}]}, indent=2))
# Python API: dx_com.pre_optimize(onnx.load(MODEL), passes={PASS: cfg})
```

Settings for each model family and task are listed below. Set only `MODEL` and `PASS` when the model has Ultralytics metadata. Set the other values too when it does not (for example, a model re-exported by another tool).

YOLOv8 / v11 · det

```
PASS = "yolo_dfl_postprocess"
TASK = "base"
NUM_CLASSES = 80
```

Each layer gets `bbox` (64ch) and `cls_conf` (80ch).

YOLOv8 / v11 · seg

```
PASS = "yolo_dfl_postprocess"
TASK = "seg"
NUM_CLASSES = 80
```

Each layer gets `bbox` (64ch), `cls_conf` (80ch), and `mask_coeff` (32ch). The 32ch prototype Conv at stride 4 is skipped.

YOLOv8 / v11 · pose

ⓒ Copyright 2026 DEEPX All Rights Reserved.

```
PASS = "yolo_dfl_postprocess"
TASK = "pose"
NUM_CLASSES = 1
KPT_SHAPE = [17, 3]
```

Each layer gets `bbox` (64ch), `cls_conf` (1ch), `kpt` (51ch), and `kpt_shape`.

YOLO26 · det

```
PASS = "yolo_no_dfl_postprocess"
TASK = "base"
NUM_CLASSES = 80
```

Each layer gets `bbox` (4ch) and `cls_conf` (80ch).

YOLO26 · seg

```
PASS = "yolo_no_dfl_postprocess"
TASK = "seg"
NUM_CLASSES = 80
```

Each layer gets `bbox` (4ch), `cls_conf` (80ch), and `mask_coeff` (32ch). The 32ch prototype Conv at stride 4 is skipped.

YOLO26 · pose

```
PASS = "yolo_no_dfl_postprocess"
TASK = "pose"
NUM_CLASSES = 1
KPT_SHAPE = [17, 3]
```

Each layer gets `bbox` (4ch), `cls_conf` (1ch), `kpt` (51ch), and `kpt_shape`.

The output for the YOLOv8n detection model in this guide is:


```
{
 "pre_optimize": [
  {
   "yolo_dfl_postprocess": {
    "task": "base",
    "layers": [
     {"bbox": "onnx::Concat_328", "cls_conf": "onnx::Concat_335"},
     {"bbox": "onnx::Concat_343", "cls_conf": "onnx::Concat_350"},
     {"bbox": "onnx::Concat_358", "cls_conf": "onnx::Concat_365"}
    ],
    "num_classes": 80
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    }
   }
  ]
}
```

Paste it into the compile config as is, or pass `cfg` to `dx_com.pre_optimize()`. `topk` is not printed; add it to change the default of 300. If the script stops with an `AssertionError`, the message tells which value to check: usually `NUM_CLASSES` (no `cls_conf` Conv found) or `PASS` (no `bbox` Conv found).

6.6.3 Find them in Netron

1. Netron shows tensor shapes only if they are stored in the model. Save a copy with shapes first:

   ```
   import onnx
   from onnx import shape_inference
   onnx.save(shape_inference.infer_shapes(onnx.load("model.onnx")), "model_shapes.onnx")
   ```

2. Open `model_shapes.onnx` in Netron and scroll to the end of the graph, where the head branches merge into the output.
3. For each scale, follow the parallel branches upward from the `Concat` that joins them. The first `Conv` you reach on each branch is a head Conv. It has no Sigmoid or SiLU after it.
4. Click the Conv. The right panel shows its **output name** (copy this into `layers`) and its shape `[1, C, H, W]`. Use `C` to get the role (channel counts in step 3 of *How to recognize a head Conv*), and `H` to get the scale.

6.6.4 RTMDet (`rtmdet_postprocess`)

The script below finds all six keys of the RTMDet pass for an MMDetection RTMDet export with NMS in the graph:

```
import json
from collections import defaultdict

import onnx
from onnx import shape_inference

MODEL = "rtmdet.onnx"

model = shape_inference.infer_shapes(onnx.load(MODEL))
g = model.graph
shape = {v.name: [d.dim_value for d in v.type.tensor_type.shape.dim]
        for v in list(g.input) + list(g.value_info) + list(g.output)}
```


© Copyright 2026 DEEPX All Rights Reserved.

```
producer = {o: n for n in g.node for o in n.output}
consumers = defaultdict(list)
for n in g.node:
    for i in n.input:
        consumers[i].append(n.op_type)

in_h = shape[g.input[0].name][2]
assert in_h, "input H is dynamic: re-export the model with a fixed input size"
cls_layers, reg_layers = [], []
for n in g.node:
    out = n.output[0]
    d = shape.get(out)
    if n.op_type != "Conv" or not d or len(d) != 4 or not d[2]:
        continue
    if "rtm_cls" in out:
        cls_layers.append((in_h // d[2], out))
    elif "rtm_reg" in out:
        reg_layers.append((in_h // d[2], out))
assert cls_layers and len(cls_layers) == len(reg_layers), f"cls={cls_layers} reg={reg_layers}"

nms = [n for n in g.node if n.op_type == "NonMaxSuppression"]
assert nms, "no NonMaxSuppression in the graph: export the model with NMS"
scores_t = producer[nms[0].input[1]]
assert scores_t.op_type == "Transpose", f"unexpected NMS score producer: {scores_t.op_type}"
topk = next(n for n in g.node if n.op_type == "TopK")  # the first TopK, over all anchors

cfg = {
    "cls_layers": [o for _, o in sorted(cls_layers)],  # stride 8 -> 16 -> 32
    "reg_layers": [o for _, o in sorted(reg_layers)],
    "use_exp": "Exp" in consumers[reg_layers[0][1]],  # Exp (not Mul) after rtm_reg
    "nms_boxes_tensor": nms[0].input[0],
    "nms_scores_tensor": scores_t.input[0],
    "nms_score_tensor": topk.output[0],
}
print(json.dumps({"pre_optimize": [{"rtmdet_postprocess": cfg}]}, indent=2))
```

For RTMDet-S it prints the same values as the *RTMDet-S Detection* example above. RTMDet-M differs only in `"use_exp": true`.

6.7 Performance Reference

Throughput on DEEPX DX-M1 with an ARM Cortex-A53 host CPU:

Group	Model	Task	Before (FPS)	After (FPS)	Speed-up
`yolo_dfl_postprocess`	YOLOv8n	det	89.53	355.73	**4.0×**

© Copyright 2026 DEEPX All Rights Reserved.

Group	Model	Task	Before (FPS)	After (FPS)	Speed-up
	YOLOv8n	seg	63.81	209.69	**3.3×**
	YOLOv8n	pose	268.98	400.59	**1.5×**
`yolo_no_dfl_postprocess`	YOLO26n	det	184.89	315.73	**1.7×**
	YOLO26n	seg	130.82	195.05	**1.5×**
	YOLO26n	pose	242.77	304.38	**1.3×**
`rtmdet_postprocess`	RTMDet-S (score_threshold = 0.3)	det	156.56	290.16	**1.9×**

Measured with:

```
DXRT_DYNAMIC_CPU_THREAD=ON dxrun -m <model_path> -l 1000 --use-ort
```

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

6.8 Migration from PPU Type 2

If you previously used `ppu.type = 2` in the JSON config, switch to `pre_optimize` — either the API or the config-JSON block.

Previous (deprecated):

```
{
 "ppu": {
  "type": 2,
  "topk": 512,
  "num_classes": 80,
  "layer": [
   {"bbox": "onnx::Concat_328",
    "cls_conf": "onnx::Concat_335"},
   {"bbox": "onnx::Concat_343",
    "cls_conf": "onnx::Concat_350"},
   {"bbox": "onnx::Concat_358",
    "cls_conf": "onnx::Concat_365"}
  ]
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  }
}
```

New (`pre_optimize`):

```
{
 "pre_optimize": [
  {
   "yolo_dfl_postprocess": {
    "task": "base",
    "layers": [
     {"bbox": "onnx::Concat_328",
      "cls_conf": "onnx::Concat_335"},
     {"bbox": "onnx::Concat_343",
      "cls_conf": "onnx::Concat_350"},
     {"bbox": "onnx::Concat_358",
      "cls_conf": "onnx::Concat_365"}
    ],
    "num_classes": 80,
    "topk": 512
   }
  }
 ]
}
```

Reusing PPU layer names

A PPU `layer` entry can be reused as a `layers` entry only if both refer to the same ONNX file and each name is a raw head Conv output. If you re-export the model, the names can change. The `onnx::Concat_NNN` names above are examples. Look up the names again with the script.

Differences:

	PPU Type 2 (previous)	**`pre_optimize` (current)**
Configuration location	JSON config file	Python API **or** config-JSON `pre_optimize` block
Segmentation support	No	Yes
Pose support	No	Yes
YOLO26 support	No	Yes (`yolo_no_dfl_postprocess`)

© Copyright 2026 DEEPX All Rights Reserved.

	PPU Type 2 (previous)	`pre_optimize` (current)
RTMDet support	No	Yes (`rtmdet_postprocess`)
Pass names	(`ppu.type = 2`)	`yolo_dfl_postprocess` / `yolo_no_dfl_postprocess` / `rtmdet_postprocess`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

7. 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
```

7.1 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.

7.1.1 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_dataset",
    "file_extensions": ["jpeg", "jpg", "png"],
    "preprocessings": [
      {"resize": {"mode": "torchvision", "size": 256, "interpolation": "BILINEAR"}},
      {"centercrop": {"width": 224, "height": 224}},
```


© Copyright 2026 DEEPX All Rights Reserved.

```
      {"convertColor": {"form": "BGR2RGB"}},
      {"div": {"x": 255}},
      {"normalize": {"mean": [0.485, 0.456, 0.406], "std": [0.229, 0.224, 0.225]}},
      {"transpose": {"axis": [2, 0, 1]}},
      {"expandDim": {"axis": 0}}
    ]
  }
}
```

Command:

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

7.1.2 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(256),  # same preprocessing as Option A
            transforms.CenterCrop(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):
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        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_dataset', img_size=224)
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)

# 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
```

7.2 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.

7.2.1 `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"""
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    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'))
        ])

        # 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)
```


© Copyright 2026 DEEPX All Rights Reserved.

```
# 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
)

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.

7.3 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 and minimize host CPU load.

7.3.1 Configuration for Aggressive Partitioning

© Copyright 2026 DEEPX All Rights Reserved.

```
{
  "inputs": {
    "input": [1, 3, 224, 224]
  },
  "calibration_method": "ema",
  "calibration_num": 50,
  "default_loader": {
    "dataset_path": "./calibration_dataset",
    "file_extensions": ["jpeg", "jpg", "png"],
    "preprocessings": [
      {"resize": {"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]}},
      {"transpose": {"axis": [2, 0, 1]}},
      {"expandDim": {"axis": 0}}
    ]
  }
}
```

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

7.3.2 Option A: `dxcom` Command

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

7.3.3 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
)
```


© Copyright 2026 DEEPX All Rights Reserved.

Optimization Strategy: Aggressive Partitioning:

- **Pros**: Maximum NPU offloading and significantly reduced host CPU load.
- **Cons**: Compilation time and memory usage may increase. On systems with a powerful host CPU, the default partitioning may give better end-to-end performance.
- **Note**: `--opt_level 0` in the example above shortens compilation but may result in higher NPU latency than `opt_level 1`.

7.4 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.

7.4.1 `dx_com` Python Module

```python
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]))
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        # 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,
)
```

7.5 Use Case 5: Enhanced Quantization (DXQ)

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

> **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.

7.5.1 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}
  },
```


© Copyright 2026 DEEPX All Rights Reserved.

```
  "default_loader": {
    "dataset_path": "./calibration_dataset",
    "file_extensions": ["jpeg", "jpg", "png"],
    "preprocessings": [
      {"resize": {"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]}},
      {"transpose": {"axis": [2, 0, 1]}},
      {"expandDim": {"axis": 0}}
    ]
  }
}
```

Command:

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

7.5.2 Option B: `dx_com` Python Module

```
import dx_com

dx_com.compile(
    model="large_model.onnx",
    output_dir="output/large_model_dxq",
    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, select the GPU with the `CUDA_VISIBLE_DEVICES` environment variable (e.g., `CUDA_VISIBLE_DEVICES=1 dxcom ...`); `--use_gpu False` (Python: `use_gpu=False`) forces CPU. See the **Quantization Device** section in JSON File Configuration for details.

© Copyright 2026 DEEPX All Rights Reserved.

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

7.5.3 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

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,
)
```


© Copyright 2026 DEEPX All Rights Reserved.

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.

7.6 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.

7.6.1 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.

Option A: `dxcom` Command

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


© Copyright 2026 DEEPX All Rights Reserved.

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.

7.6.2 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
```

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",
```


© Copyright 2026 DEEPX All Rights Reserved.

```
    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.

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

Scenario: Deploying a YOLO-family detection, instance-segmentation, or pose model (or RTMDet) 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 `pre_optimize` before compilation. It rewrites the post-processing graph so that TopK selection happens first, and expensive operations run only on the `K` selected candidates (default 300; 512 for RTMDet) instead of the full anchor grid. Choose one of two equivalent entry points.

7.7.1 Option A — Python API

Call `dx_com.pre_optimize()`, then pass the result to `dx_com.compile()`.

```
import onnx
import dx_com

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


© Copyright 2026 DEEPX All Rights Reserved.

```
        "task": "base",
        "layers": [
            {"bbox": "onnx::Concat_328",
             "cls_conf": "onnx::Concat_335"},
            {"bbox": "onnx::Concat_343",
             "cls_conf": "onnx::Concat_350"},
            {"bbox": "onnx::Concat_358",
             "cls_conf": "onnx::Concat_365"},
        ],
        "num_classes": 80,
        "topk": 300,
    },
})

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

7.7.2 Option B — Config JSON

Add a `pre_optimize` block to the compile config; the pass runs automatically inside `dx_com.compile()`.

```
{
 "inputs": { "images": [1, 3, 640, 640] },
 "calibration_method": "ema",
 "calibration_num": 100,
 "default_loader": {
  "dataset_path": "./calibration_dataset",
  "file_extensions": ["jpeg", "jpg", "png"],
  "preprocessings": [
   {"convertColor": {"form": "BGR2RGB"}},
   {"resize": {"width": 640, "height": 640}},
   {"div": {"x": 255}},
   {"transpose": {"axis": [2, 0, 1]}},
   {"expandDim": {"axis": 0}}
  ]
 },
 "pre_optimize": [
  {
   "yolo_dfl_postprocess": {
    "task": "base",
    "layers": [
     {"bbox": "onnx::Concat_328",
      "cls_conf": "onnx::Concat_335"},
     {"bbox": "onnx::Concat_343",
      "cls_conf": "onnx::Concat_350"},
     {"bbox": "onnx::Concat_358",
```


© Copyright 2026 DEEPX All Rights Reserved.

```
        "cls_conf": "onnx::Concat_365"}
      ],
      "num_classes": 80,
      "topk": 300
    }
  }
 ]
}
```

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

For supported model families and tasks (YOLOv8 / v9 / v11 / v12 / v13 detection · segmentation · pose via `yolo_dfl_postprocess`; YOLO26 via `yolo_no_dfl_postprocess`; RTMDet via `rtmdet_postprocess`), output shapes, and the migration recipe from the deprecated `ppu.type = 2`, see Pre-Optimize API.

> **Do not combine with `ppu`**
>
> `pre_optimize` and `ppu` (Use Case 8) are **mutually exclusive** in one config — both rewrite detection post-processing, so supply only one. DX-COM 2.5.0 does not reject a config that has both: it compiles with `ppu` and drops `pre_optimize`, or fails later with an unrelated error. The only sign is a console line `falling back to legacy: ValueError: config 'pre_optimize' and 'ppu' are mutually exclusive`.

7.8 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.

> **Do not combine with `pre_optimize`**
>
> `ppu` and `pre_optimize` (Use Case 7) are **mutually exclusive** in one config — supply only one. DX-COM 2.5.0 does not reject a config that has both: it compiles with `ppu` and drops `pre_optimize`, or fails later with an unrelated error. The only sign is a console line `falling back to legacy: ValueError: config 'pre_optimize' and 'ppu' are mutually exclusive`.

7.8.1 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_dataset",
  "file_extensions": ["jpeg", "jpg", "png"],
  "preprocessings": [
   {"convertColor": {"form": "BGR2RGB"}},
   {"resize": {"width": 640, "height": 640}},
   {"div": {"x": 255}},
   {"transpose": {"axis": [2, 0, 1]}},
   {"expandDim": {"axis": 0}}
  ]
 },
 "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
```


© Copyright 2026 DEEPX All Rights Reserved.

7.8.2 Option B: `dx_com` Python Module

Here, the PPU is configured through `ppu_config`. Use a config file **without a `ppu` block** (for example, the Option A config with its `ppu` section removed), or pass a `dataloader` instead.

```
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)
    .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_base_config.json",  # no `ppu` block; 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},
    },
)
```


© Copyright 2026 DEEPX All Rights Reserved.

Selecting the PPU type

Match the `PPUTypes` value to your model architecture:

- Anchor-based (YOLOv3/v4/v5/v7) → `YOLO_BASE`
- Anchor-free (YOLOX, YOLOv8–v12) → `YOLO_ANCHORFREE`
- DFL-based with CPU TopK (YOLOv8/v9/v11/v12) → `YOLOV8` **(deprecated since v2.4.0; use Use Case 7 / `pre_optimize()` instead)**

See the **Configuration Parameters** table in the **PPU Configuration** section of JSON File Configuration and the **`ppu_config`** parameter in Execution of DX-COM.

© Copyright 2026 DEEPX All Rights Reserved.

8. Development Tools

8.1 Compilation Summary Report

8.1.1 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.

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.

8.1.2 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 \
```


© Copyright 2026 DEEPX All Rights Reserved.

```
-o output/mobilenetv1 \
--export_html
```

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.

8.1.3 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.

8.1.4 Requirements

© Copyright 2026 DEEPX All Rights Reserved.

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.

8.1.5 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.

9. 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.

9.1 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

9.2 Prerequisites

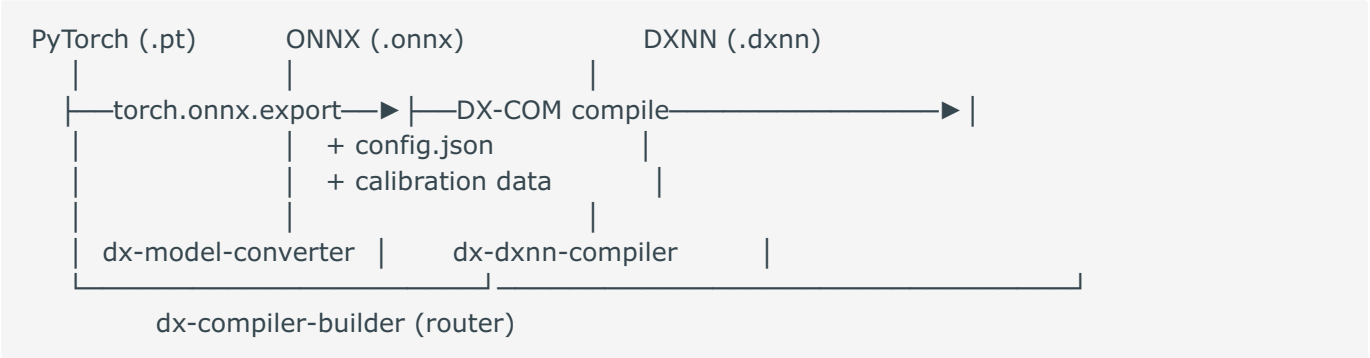
Requirement	Details
DX-COM	Installed via DX-COM installation guide (`pip install dx-com` or DEEPX package repository)
AI coding agent (one of)	Claude Code, GitHub Copilot (VS Code), Cursor, OpenCode, or Codex CLI
Python	3.8–3.14
ONNX	opset 11–21
OS	Debian Linux (Ubuntu 20.04/22.04/24.04/26.04), x86_64

9.3 Architecture Overview

© Copyright 2026 DEEPX All Rights Reserved.

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.

9.3.1 Compilation Pipeline



9.3.2 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.

9.4 Available Agents and Skills

9.4.1 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

9.4.2 Skills (All Platforms)

© Copyright 2026 DEEPX All Rights Reserved.

General SWE Process

Skill	Description
/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
/dx-agent-compiler-pre-optimize	Build a verified pre_optimize config (YOLO/ RTMDet head tensor names)
/dx-agent-brainstorm	DEEPX build brainstorming with model registry check and sub-project routing
/dx-agent-tdd	DEEPX build validation order — factory, pipeline, and integration checks
/dx-agent-verify	DEEPX build verification checklists — dx_app, dx_stream, and cross-project

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Harness

Skill	Description
`/dx-harness-writing-skills`	Create and edit skill files
`/dx-harness-validate`	Validate this repo's `.deepx/` integrity (bootstraps the suite harness when standalone)

9.5 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/` (20 files: `dx-compiler.mdc`, 3 agent `.mdc` files, 16 `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/*/`

ⓒ Copyright 2026 DEEPX All Rights Reserved.

Tool	Type	Auto-Load Mechanism	Agent Invocation
			`SKILL.md` directly when needed

9.5.1 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

# 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
```

9.5.2 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/` (20 files)	Cursor	Auto

© Copyright 2026 DEEPX All Rights Reserved.

File	Auto-loaded by	Loading
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
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 (16 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-agent-brainstorm	.deepx/skills/dx-agent-brainstorm/SKILL.md

© Copyright 2026 DEEPX All Rights Reserved.

Skill	File
/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
/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-agent-tdd	.deepx/skills/dx-agent-tdd/SKILL.md
/dx-agent-compiler-validate	.deepx/skills/dx-agent-compiler-validate/SKILL.md
/dx-agent-compiler-pre-optimize	.deepx/skills/dx-agent-compiler-pre-optimize/SKILL.md
/dx-agent-verify	.deepx/skills/dx-agent-verify/SKILL.md
/dx-swe-writing-plans	.deepx/skills/dx-swe-writing-plans/SKILL.md
/dx-harness-writing-skills	.deepx/skills/dx-harness-writing-skills/SKILL.md
/dx-harness-validate	.deepx/skills/dx-harness-validate/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/		

© Copyright 2026 DEEPX All Rights Reserved.

Directory	Files	Description
	`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/`	`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.

9.6 User Scenarios

9.6.1 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)

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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.

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+).

9.6.2 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.

© Copyright 2026 DEEPX All Rights Reserved.

Tool	How to Use
OpenCode	`/dx-agent-compiler-convert` or `@dx-model-converter` followed by the prompt.
Codex CLI	Open `dx-compiler/` and type the prompt directly.

9.6.3 Scenario 2: Compile ONNX to DXNN

Prompt:

```
"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.
Codex CLI	Open `dx-compiler/` and type the prompt directly, or `/dx-agent-compiler-compile` followed by the prompt.

9.6.4 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.

© Copyright 2026 DEEPX All Rights Reserved.

Tool	How to Use
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.
Codex CLI	Open `dx-compiler/` and type the prompt.

9.7 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)
`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

9.8 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>_<coding_model>_<target_model>_<task>` where `<agent>` is `claude` , `codex` , `copilot` , `cursor` , or `opencode` , `<coding_model>` is the model

© Copyright 2026 DEEPX All Rights Reserved.

the agent ran (e.g. `opus46`, `sonnet46`, `gpt55`), and `<target_model>` is the model being compiled (e.g. `yolo26n`)

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)
```

9.9 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)

9.10 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
```


© Copyright 2026 DEEPX All Rights Reserved.

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.

9.11 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.

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.

9.12 Troubleshooting

Symptom	Cause	Fix
`ValueError: Batch size must be 1`	Model has batch > 1	Re-export ONNX with batch=1 input shape

© Copyright 2026 DEEPX All Rights Reserved.

Symptom	Cause	Fix
`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()`. 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

9.13 Mandatory Output Artifacts

© Copyright 2026 DEEPX All Rights Reserved.

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.

9.14 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

© Copyright 2026 DEEPX All Rights Reserved.

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.

10. Change Log

10.1 v2.5.0 (2026-10-16)

DX-Compiler Version

- DX-COM: v2.5.0
- DX-TRON: Removed

> ⚠ **Removal Notice — DX-TRON**
>
> **DX-TRON** was deprecated in **DX-Compiler v2.4.0** and is **removed as of v2.5.0**. Use the DX-COM **Compilation Summary Report** (`--export_html`) for model inspection and visualization.

Changed

- **`.dxnn` File Format v9**: DX-COM v2.5.0 outputs `.dxnn` file format v9 only. Loading these models requires **DX-RT v3.5.0 or later**; DX-RT v3.4.2 and earlier cannot load them. See System Requirements of DX-COM.
- **Python Entry Point Signature**: `dx_com.compile()` now accepts keyword arguments (`**kwargs`); call it with keyword arguments as documented in Execution of DX-COM.
- **`quantization_device` Deprecated**: The JSON `quantization_device` field and the Python `quantization_device` argument are deprecated and ignored — DX-COM prints a `UserWarning` and selects the quantization device with `--use_gpu` / `use_gpu` (default `True`) and the `CUDA_VISIBLE_DEVICES` environment variable. Remove the field from existing configs.
- **QAT Parameters Restructured**: The Python arguments `quantization_mode` , `qat_config` , `qat_skip_training` , and `qat_resume_from_checkpoint` are **replaced by a single `qmaster` argument** that takes a `QMasterConfig` object (import it from `dx_com`). The old `qat_*` arguments are no longer honored from Python code. Triggering QAT from a JSON `qmaster` block is unchanged — QAT is still auto-selected.
- Removed DX-TRON support: `--target=dx_tron` is no longer accepted by `install.sh` or `uninstall.sh` , and the `run_dxtron_web.sh` / `run_dxtron_appimage.sh` launcher scripts were deleted. Valid targets are now `dx_com` and `all` ; in `uninstall.sh` both now do the same work.

© Copyright 2026 DEEPX All Rights Reserved.

- `compiler.properties` keeps only `COM_VERSION`; the DX-TRON version and download URL properties were removed.
- Removing an installed `dxtron` DEB package is now manual: `sudo apt-get remove dxtron`.
- Removed the *Model Viewer — DX-TRON* page from this manual and renumbered *Compilation Summary Report* to `04_Compilation_Summary_Report.md`.
- **`pre_optimize` Pass Renaming (Breaking)**: `yolo_postprocess` → `yolo_dfl_postprocess` and `yolo26_postprocess` → `yolo_no_dfl_postprocess`. The previous pass names are no longer accepted and raise an error.
- **`pre_optimize` Configuration Changes (Breaking)**: The YOLO passes (`yolo_dfl_postprocess`, `yolo_no_dfl_postprocess`) now require an explicit `task` key (`base` / `seg` / `pose`). The `input_height` / `input_width` keys are removed — the input size is derived from the model input, and supplying them raises `ValueError`. See Pre-Optimize API.
- **`Resize` Supported Scale Ranges Documented**: `nearest` upsampling supports powers of two only (`2`, `4`, `8`, `16`, …). `linear` upsampling supports integer factors ≥ 2, except primes greater than `8` (`11`, `13`, `17`, …). Downsampling is supported only for `linear` at scale `0.5`; `nearest` downsampling is not supported.

Fixed

- `uninstall.sh` cleans up a leftover `dx_tron/` directory or symlink from an earlier release, so upgrading does not orphan it.
- Without `--python_version`, `install.sh` now uses the OS default `python3` for the virtual environment instead of the newest `python3.X` on the host (e.g., a deadsnakes `python3.14` on Ubuntu 24.04). Newer interpreters are only considered when the OS default is below the minimum supported version.

Added

- **`--use_gpu {True,False}` (CLI)** and **`use_gpu` (Python)**: Toggle quantization between GPU (default, when available) and CPU. On the CLI, select a specific GPU with the `CUDA_VISIBLE_DEVICES` environment variable.
- **`--verbose` (CLI)** and **`verbose` (Python)**: Expand masked error messages to include the origin type, the original message, and one bounded direct cause.
- **`QMasterConfig`** public configuration object for QAT (exported from `dx_com`).
- **Segmentation QAT**: Quantization-Aware Training now supports dense segmentation models via an opt-in `task_type` block in the `qmaster` config (`seg_root`, `seg_pairs_train` / `seg_pairs_val`, `seg_ignore_index`, `seg_binary`, `seg_ce`).

© Copyright 2026 DEEPX All Rights Reserved.

- Passing `--target=dx_tron` now exits with a specific explanation instead of a generic "invalid target" error, pointing at `--target=dx_com` and the `--export_html` summary report.
- **One-line installation**: `oneline-install.sh` installs DX-COM from PyPI without cloning the repository (`curl -fsSL https://raw.githubusercontent.com/DEEPX-AI/dx-compiler/main/oneline-install.sh | sh`) into a dedicated virtualenv and links `dxcom` onto `PATH`. `oneline-uninstall.sh` removes it; `uninstall.sh` also removes a one-line install.
- **uv-based installation**: `install.sh` uses uv when it is available and falls back to pip otherwise. `--uv=true` installs uv first if missing; `--uv=false` forces pip. See Installation of DX-COM.
- **New `pre_optimize` Tasks**: The `task` key (`base` / `seg` / `pose`) adds pose (keypoint) and segmentation post-processing support.
- **`rtmdet_postprocess` Pass**: RTMDet detection post-processing.
- **`Gather` operator**: 2-D `indices` tensors are now supported (previously 0-D/1-D only). See Supported ONNX Operators.
- **`Einsum` operator**: Two-input `Einsum` equations that reduce to a (batched) matrix multiplication (e.g., `ij,jk->ik`, `bij,bjk->bik`) are now supported. See Supported ONNX Operators.
- **3D Convolution Support**: `Conv` and `ConvTranspose` with a 5D input (3 spatial dimensions) are now supported on the NPU (previously CPU-only), subject to constraints such as `group=1`, no dilation, and a compile-time constant weight. See Supported ONNX Operators for the full constraints.

10.2 v2.4.1 (2026-08-14)

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

- Added missing documentation for the `iqr` (Interquartile Range) calibration method.
- Improved PPU type configuration table formatting and clarified that `PPUTypes.YOLOV8` supports multiple YOLO versions (v8, v9, v11, v12).

© Copyright 2026 DEEPX All Rights Reserved.

- Added deprecation warning to the DX-TRON chapter directing users to the HTML graph viewer replacement.
- Added Codex CLI to Agent-Driven Development supported tools list and scenario tables.
- 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.

10.3 v2.4.0 (2026-07-22)

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.

> ⚠ **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.

© Copyright 2026 DEEPX All Rights Reserved.

- **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 Q-Master (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 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).

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **`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.

10.4 v2.3.1 (2026-05-06)

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`.

10.5 v2.3.0 (2026-04-10)

© Copyright 2026 DEEPX All Rights Reserved.

DX-Compiler Version

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

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, RHEL 9–10, and CentOS Stream 9–10.

10.6 v2.2.1 (2026-02-09)

DX-Compiler Version

- DX-COM: v2.2.1
- DX-TRON: v2.0.1

Changed

- None

ⓒ Copyright 2026 DEEPX All Rights Reserved.

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.

10.7 v2.2.0 (2026-01-16)

DX-Compiler Version

- DX-COM: v2.2.0
- DX-TRON: v2.0.1

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.
- **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.

ⓒ Copyright 2026 DEEPX All Rights Reserved.

- **DX-TRON**:
  - Debian package (.deb) installation support
  - Local web browser hosting (`dxtron` command)
  - Ubuntu 24.04 support

10.8 v2.1.0 (2025-11-28)

DX-Compiler Version

- DX-COM: v2.1.0
- DX-TRON: v2.0.0

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.
- **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).

© Copyright 2026 DEEPX All Rights Reserved.

10.9 v2.0.0 (2025-09-08)

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

---

10.10 v1.0.0 (2025-07-23)

Bug Fixes

- Internal bug fixes

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

© Copyright 2026 DEEPX All Rights Reserved.

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.