DXNN Compiler (DX-COM) User Manual

2026/03/30 (v2.3.0)

DEEPX.ai

Copyright 2026 DEEPX All Rights Reserved.

Table of contents

1. Quick Start Guide 3
1.1 Install DX-COM 3
1.2 Compile Sample Models 3
1.3 Compile Your Own Model 3
1.4 Next Steps 4
2. DXNN SDK Overview 5
2.1 DEEPX SDK Architecture 5
3. Installation and Configuration 7
3.1 System Requirements of DX-COM 7
3.2 Installation of DX-COM 8
3.3 Source File Structure of DX-COM 10
3.4 ONNX File Configuration 11
3.5 JSON File Configuration 13
3.6 Execution of DX-COM 30
3.7 Common Use Cases 44
4. Supported ONNX Operators 53
4.1 Operator Support Details 53
5. Development Tools 59
5.1 Model Viewer_DX-TRON 59
6. Change Log 68
6.1 v2.3.0 (March 2026) 68
6.2 v2.2.1 (February 2026) 69
6.3 v2.2.0 (December 2025) 69
6.4 v2.1.0 (November 2025) 70
6.5 v2.0.0 (September 2025) 71
6.6 v1.60.1 (June 2025) 72

Copyright 2026 DEEPX All Rights Reserved.

1. Quick Start Guide

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

1.1 Install DX-COM

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

1.2 Compile Sample Models

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

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

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

Then run all sample compilations at once:

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

1.3 Compile Your Own Model

1.3.1 With the `dxcom` Command

For complete examples and options, see CLI Execution.

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


Copyright 2026 DEEPX All Rights Reserved.

1.3.2 With the `dx_com` Python Module

For complete examples and options, see dx_com Python Module Usage.

Using `dx_com.compile()`:

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

1.4 Next Steps

1. **Installation of DX-COM** → Installation of DX-COM
2. **Execution of DX-COM** → Execution of DX-COM
3. **JSON File Configuration** → JSON File Configuration
4. **Common Use Cases** → Common Use Cases

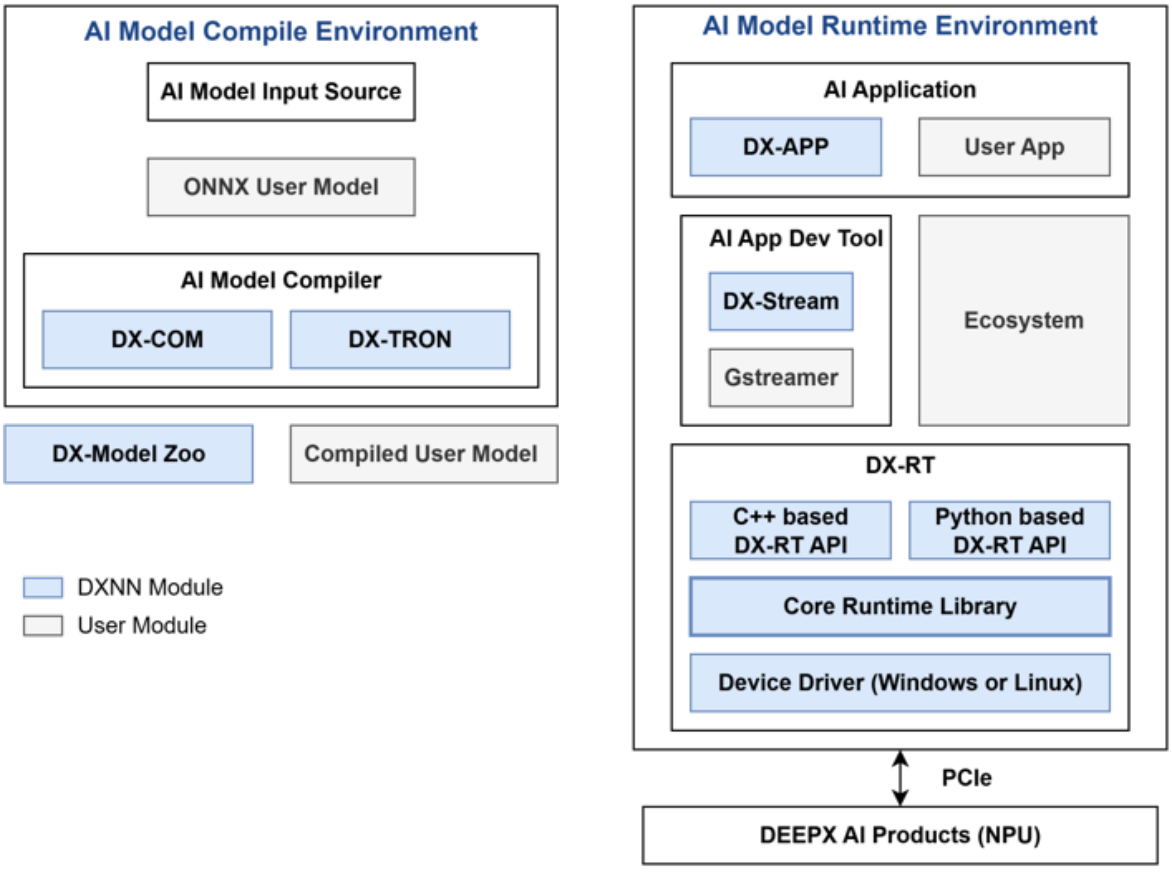
Happy compiling!

Copyright 2026 DEEPX All Rights Reserved.

2. DXNN SDK Overview

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

2.1 DEEPX SDK Architecture



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

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

Copyright 2026 DEEPX All Rights Reserved.

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

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

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

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

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

Copyright 2026 DEEPX All Rights Reserved.

3. Installation and Configuration

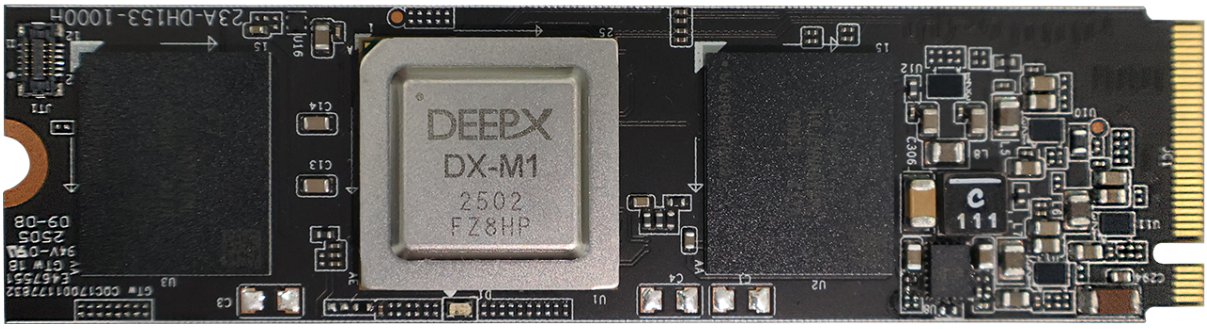
3.1 System Requirements of DX-COM

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

Hardware and Software Requirements

- **CPU:** amd64(x86_64)
  NOTE. aarch64(arm64) is **NOT** supported.
- **RAM:** ≥ 16 GB
- **Storage:** ≥ 8 GB available disk space
- **OS:** Ubuntu 20.04 / 22.04 / 24.04 (x86_64, primary supported environments), Fedora 42-45, Red Hat Enterprise Linux 9-10, and CentOS Stream 9-10 (validated in DX-COM v2.3.0).
  NOTE. Ubuntu 18.04 OS is **NOT** supported.
- **LDD:** ≥ 2.28

> **NOTE**
>
> To check your LDD 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.

- `libgl1-mesa-glx` : Provides OpenGL runtime support for graphical operations
- `libglib2.0-0` : Core utility library used by many GNOME and GTK applications

Run the following command to install the required libraries.

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

3.2.2 Installation

Supported Environments

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

Install the Wheel

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

```
pip install dx_com-<VERSION>-cp<VERSION>-cp<VERSION>-linux_x86_64.whl
```


Copyright 2026 DEEPX All Rights Reserved.

For example, for Python 3.11:

```
pip install dx_com-<VERSION>-cp311-cp311-linux_x86_64.whl
```

Verify the Installation

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

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

Copyright 2026 DEEPX All Rights Reserved.

3.3 Source File Structure of DX-COM

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

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

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

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

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

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

Copyright 2026 DEEPX All Rights Reserved.

3.4 ONNX File Configuration

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

PyTorch to ONNX Conversion Example

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

Example

```
import torch
import torch.nn as nn

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

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

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

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

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

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


Copyright 2026 DEEPX All Rights Reserved.

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

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

NOTE

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

Copyright 2026 DEEPX All Rights Reserved.

3.5 JSON File Configuration

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

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

3.5.1 Parameter Quick Reference

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

For complete examples, see Common Use Cases.

Copyright 2026 DEEPX All Rights Reserved.

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 Python Wheel Package Usage).
> - Input name **must** exactly match ONNX model definition.

Example

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

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

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

Calibration Method

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

Copyright 2026 DEEPX All Rights Reserved.

Available Methods

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

Example

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

Calibration Number

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

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

Example

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

Calibration Data Loading

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

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


Copyright 2026 DEEPX All Rights Reserved.

```
    }
}
```

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

For Multi-Input Models or Non-Image Data

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

3.5.3 Optional Parameters: Quantization Device

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

Available Values

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

Example

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

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

For practical examples, see Use Case 5: Enhanced Quantization (DXQ).

Copyright 2026 DEEPX All Rights Reserved.

3.5.4 Optional Parameters: Enhanced Quantization Scheme (DXQ)

Version Support

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

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

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

Recommendation

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

GPU Acceleration: DXQ schemes (especially P2-P5) are computationally intensive. Using `quantization_device="cuda"` can reduce compilation time by **2-5x** compared to CPU. See Use Case 5: Enhanced Quantization (DXQ) 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": {
```


Copyright 2026 DEEPX All Rights Reserved.

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

DXQ-P1

Enables a simple quantization improvement strategy.

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

DXQ-P2

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

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

DXQ-P3, DXQ-P4, DXQ-P5

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

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

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.

Copyright 2026 DEEPX All Rights Reserved.

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.

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.

Copyright 2026 DEEPX All Rights Reserved.

Performance Impact

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

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

Configuration Parameters

Type	Architecture	Supported Models
0	Anchor-Based	YOLOv3, YOLOv4, YOLOv5, YOLOv7
1	Anchor-Free	YOLOX, YOLOv8, YOLOv9, YOLOv10, YOLOv11, YOLOv12
2	DFL-Based Anchor-Free	YOLOv8, YOLOv9, YOLOv10, YOLOv11, YOLOv12 (optional CPU-side TopK optimization path; validated in DX-COM v2.3.0)

Type 0: Anchor-Based YOLO Models

```
{
  "ppu": {
    "type": 0,
    "conf_thres": 0.25,
    "activation": "Sigmoid",
    "num_classes": 80,
```


Copyright 2026 DEEPX All Rights Reserved.

```
    "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:

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


Copyright 2026 DEEPX All Rights Reserved.

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)

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

This option is intended for DFL-based YOLO families that expose separate per-scale `bbox` and `cls_conf` head nodes. It is an optional CPU-side optimization path in addition to the standard anchor-free PPU configuration. In internal validation for DX-COM v2.3.0, this path was compile-verified with YOLOv8, YOLOv9, YOLOv10, YOLOv11, and YOLOv12 variants using compatible head mappings.

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


Copyright 2026 DEEPX All Rights Reserved.

Parameters:

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

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

Implementation: Identifying ONNX Node Names

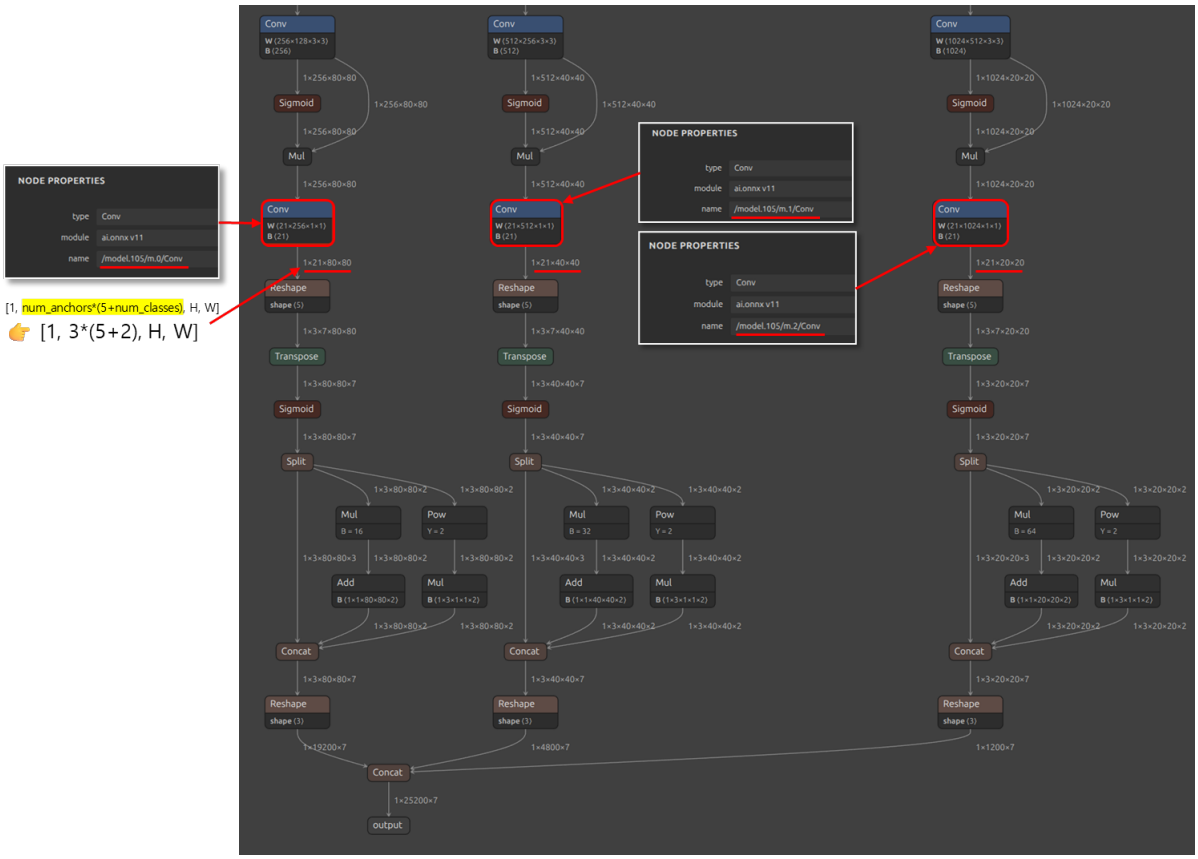
To configure PPU, you need to identify specific node names in your ONNX model:

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

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

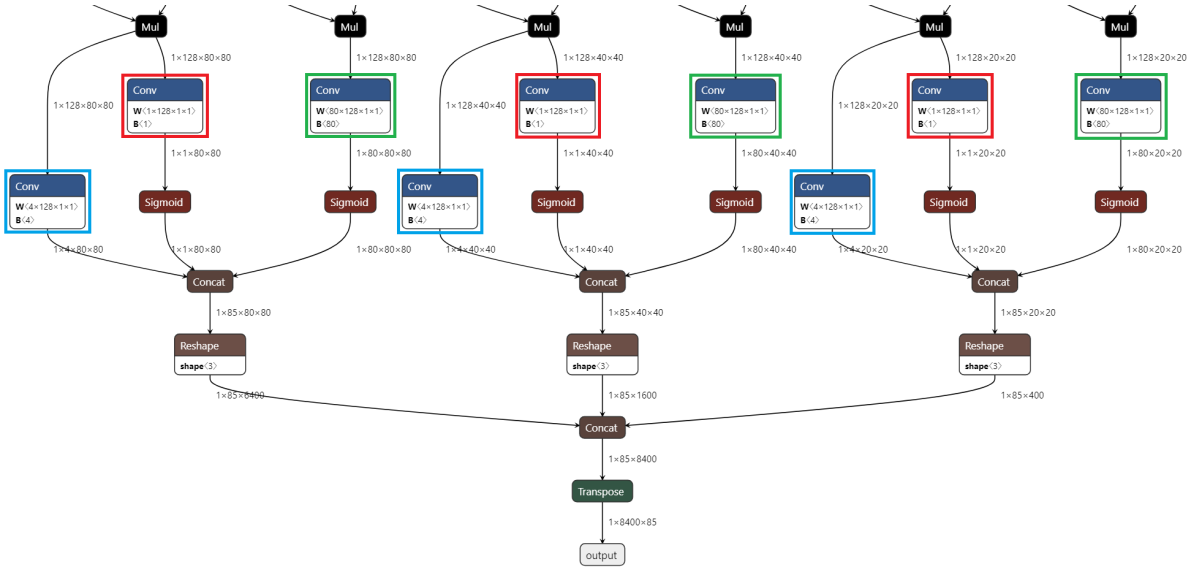
- Trace backwards from model outputs to locate detection head Conv layers
- These Conv layers output feature maps with shape `[1, num_anchors*(5+num_classes), H, W]`
- Example node names in YOLOv7: `/model.105/m.0/Conv`, `/model.105/m.1/Conv`, `/model.105/m.2/Conv`

Copyright 2026 DEEPX All Rights Reserved.



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

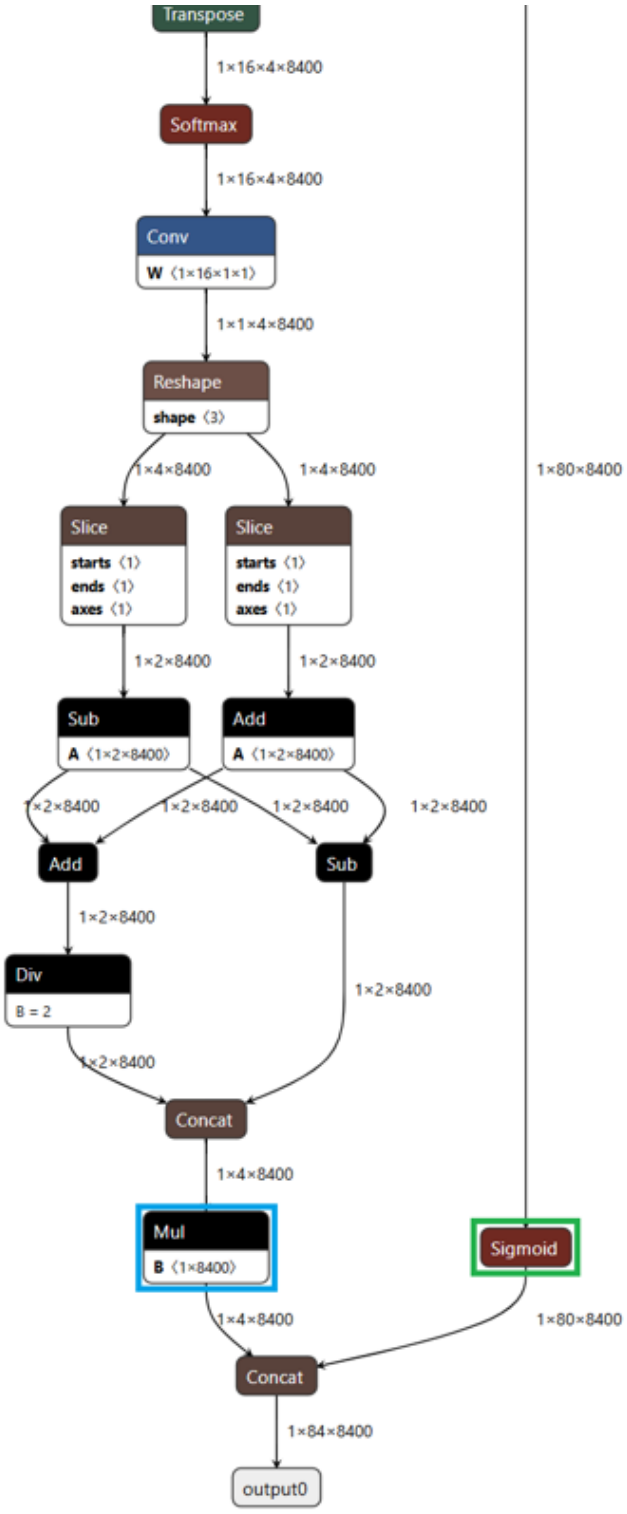
- Find three types of Conv nodes for each detection scale: `bbox`, `obj_conf`, `cls_conf`
- This results in 9 Conv layers total (3 scales × 3 branches)



Copyright 2026 DEEPX All Rights Reserved.

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

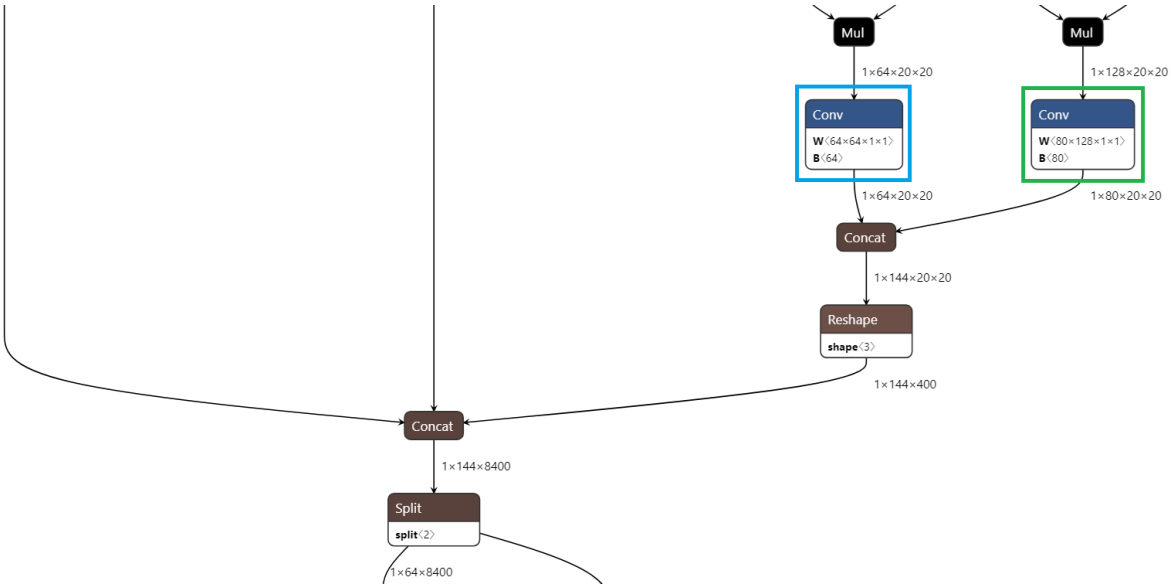
- Trace backwards from the final `Concat` operation
- Identify `bbox` (often from `Mul` or `Add`) and `cls_conf` (often from `Sigmoid`)
- Use the node names that match the `layer` format required by `type = 1`



Copyright 2026 DEEPX All Rights Reserved.

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

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



3.5.6 Appendix: Preprocessing Operations Reference

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

Copyright 2026 DEEPX All Rights Reserved.

Automatic Preprocessing Optimization

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

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

How to Verify and Optimize:

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

1. Identify Integrated Nodes:

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

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

2. Update Runtime Code:

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

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

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

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

convertColor

Copyright 2026 DEEPX All Rights Reserved.

Changes the color channel order of input images.

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

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

resize

Resizes input image to a specified target size.

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

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

centercrop

Crops the central region of the input image.

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

transpose

Rearranges tensor dimensions.

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

expandDim

Adds a new dimension at the specified axis.

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

normalize

Copyright 2026 DEEPX All Rights Reserved.

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

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

mul, add, subtract, div

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

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

3.5.7 Appendix: Custom Loader (Legacy)

> ⚠ **Deprecation Notice**
>
> The Custom Loader approach is **deprecated**. For new projects, use the Python wheel package with torch DataLoader instead, which provides more flexibility and better integration with Python workflows.

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

Copyright 2026 DEEPX All Rights Reserved.

3.6 Execution of DX-COM

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

DX-COM supports two execution methods:

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

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

3.6.1 Execution Prerequisites and Constraints

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

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

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

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

3.6.2 CLI Execution (Command-Line Interface)

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

Copyright 2026 DEEPX All Rights Reserved.

Execution Method

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

Basic Command

Using `dxcom`:

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

What you need:

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

Command Format

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

Required Arguments

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

Advanced Compilation Options

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

Copyright 2026 DEEPX All Rights Reserved.

PERFORMANCE AND RESOURCE CONTROL

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

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

Optimization Level Detail

The --opt_level option controls the optimization balance:

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

Aggressive Partitioning Detail (Experimental)

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

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

Copyright 2026 DEEPX All Rights Reserved.

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
`--version`	`-v`	Prints the compiler module version and exits

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

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

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

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

CLI Execution Examples

The following examples demonstrate common usage patterns for CLI compilation.

Basic Command

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

```
dxcom \
-m sample/MobilenetV1.onnx \
```


Copyright 2026 DEEPX All Rights Reserved.

```
-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
```

Version Information

This command prints the compiler module version and exits.

```
dxcom --version
```

Compile Sample Models (Script)

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

3.6.3 Python Wheel Package Usage

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

> **Examples and Guides**
>
> For practical code examples and step-by-step guides, see:
>
> - Quick Start Guide
> - Common Use Cases

Copyright 2026 DEEPX All Rights Reserved.

Overview

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

Function Signature

```
def compile(
    model: Union[str, onnx.ModelProto],
    output_dir: str,
    config: Optional[str] = None,
    dataloader: Optional[DataLoader] = None,
    calibration_method: str = "ema",
    calibration_num: int = 100,
    quantization_device: Optional[str] = None,
    opt_level: int = 1,
    aggressive_partitioning: bool = False,
    input_nodes: Optional[List[str]] = None,
    output_nodes: Optional[List[str]] = None,
    enhanced_scheme: Optional[Dict] = None,
    gen_log: bool = False,
    float64_calibration: bool = False,
) -> None
```

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):
        # Return single sample or tuple of samples for multi-input models
        return self.data[idx]

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


Copyright 2026 DEEPX All Rights Reserved.

Optional Parameters

`calibration_method`

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

`calibration_num`

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

`quantization_device`

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

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

`opt_level`

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

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.

```
    },
}
```

`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

---

Return Value

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

Usage Examples

Basic Compilation with Configuration File

```
import dx_com

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

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

Copyright 2026 DEEPX All Rights Reserved.

Important Considerations

Input Selection: Config vs DataLoader

Users must provide **either** a configuration file **or** a DataLoader. These inputs are mutually exclusive.

- **Config:** Recommended for static, file-based compilation workflows.
- **DataLoader:** Required for programmatic data provision and models with multiple inputs. When constructing a DataLoader for compilation, the **batch_size must be set to 1.**

Hardware Acceleration (CUDA)

To enable GPU-accelerated quantization (quantization_device="cuda"), ensure the following requirements are met:

- **System:** NVIDIA CUDA drivers and toolkit are installed.
- **Framework:** PyTorch is built with CUDA support (torch.cuda.is_available() is True).

Deprecation Notice: CustomLoader

The legacy CustomLoader for non-image data is **deprecated.**

- **New Standard:** Use the standard **PyTorch DataLoader** for all data modalities (Image, Tensor, etc.) to ensure long-term compatibility and performance.

Output Files

Upon successful compilation, the `output_dir` will contain:

- `[model_name].dxnn` : Compiled model binary for execution on DEEPX NPU hardware
- `compiler.log` (if `gen_log=True`): Detailed compilation logs

Copyright 2026 DEEPX All Rights Reserved.

3.6.4 Common Errors and Troubleshooting

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

Copyright 2026 DEEPX All Rights Reserved.

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

Copyright 2026 DEEPX All Rights Reserved.

No	Error Type	Description & Conditions
12	RequiredLibraryError	Missing essential system libraries. Examples: `libgl1-mesa-glx` is not installed
13	DataNotFoundError	No valid input data found in the specified dataset path. Examples: empty folder, wrong file extensions
14	OnnxFileNotFound	The ONNX model file cannot be found or does not exist at the specified location

Copyright 2026 DEEPX All Rights Reserved.

3.7 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
> ```

3.7.1 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.

Option A: `dxcom` Command

Configuration File (`resnet50_config.json`):

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


Copyright 2026 DEEPX All Rights Reserved.

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

Command:

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

Option B: `dx_com` Python Module

Complete Script (`compile_resnet50.py`):

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

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

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

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

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


Copyright 2026 DEEPX All Rights Reserved.

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

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

Run:

```
python3 compile_resnet50.py
```

3.7.2 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.

`dx_com` Python Module

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

class StereoDataset(Dataset):
    """Dataset providing stereo image pairs"""
    def __init__(self, left_dir, right_dir):
        self.left_dir = left_dir
        self.right_dir = right_dir

        # Get matching image pairs
        self.image_files = sorted([
            f for f in os.listdir(left_dir)
```


Copyright 2026 DEEPX All Rights Reserved.

```
            if f.endswith(('.jpg', '.png'))
        ])

        # Different preprocessing for each input
        self.transform_left = transforms.Compose([
            transforms.Resize((128, 128)),
            transforms.ToTensor(),
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

        self.transform_right = 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]

        # Load left image
        left_img = Image.open(
            os.path.join(self.left_dir, filename)
        ).convert('RGB')
        left_tensor = self.transform_left(left_img)

        # Load right image
        right_img = Image.open(
            os.path.join(self.right_dir, filename)
        ).convert('RGB')
        right_tensor = self.transform_right(right_img)

        # Return tuple of inputs (order must match model)
        return left_tensor, right_tensor

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

# Compile multi-input model
dx_com.compile(
    model="stereo_model.onnx",
    output_dir="output/stereo",
    dataloader=dataloader,
```


Copyright 2026 DEEPX All Rights Reserved.

```
    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:** The DataLoader must return a tuple of tensors (one per input).
- **Input Mapping:** The order of tensors in the tuple must align strictly with the ONNX model's input node order.
- **Heterogeneous Inputs:** Each input branch supports independent sizes and preprocessing configurations.

3.7.3 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 while maintaining short compilation times.

Configuration for Aggressive Partitioning

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


Copyright 2026 DEEPX All Rights Reserved.

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

Option A: `dxcom` Command

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

Option B: `dx_com` Python Module

```
import dx_com

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

Optimization Strategy: Aggressive Partitioning:

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

3.7.4 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.

Copyright 2026 DEEPX All Rights Reserved.

`dx_com` Python Module

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

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

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

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

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

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

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

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

3.7.5 Use Case 5: Enhanced Quantization (DXQ)

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

Copyright 2026 DEEPX All Rights Reserved.

Version Support

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

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

Option A: `dxcom` Command

Configuration File (`config.json`):

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

Command:

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

Option B: `dx_com` Python Module

```
import dx_com

dx_com.compile(
    model="large_model.onnx",
```


Copyright 2026 DEEPX All Rights Reserved.

```
    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, you can specify a device via `quantization_device` in the JSON config or the `dx_com.compile()` parameter (e.g., `"cuda:1"`). See Quantization Device for details.

For all available DXQ schemes (DXQ-P0 to DXQ-P5) and their parameters, see Enhanced Quantization Scheme (DXQ).

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.

Copyright 2026 DEEPX All Rights Reserved.

4.1.2 Normal Operations

Copyright 2026 DEEPX All Rights Reserved.

Operator	Supported Conditions
Add	Supported as: - Bias addition (e.g., as part of `Gemm` or `Conv`) - Element-wise addition - Used for input normalization - Constant scalar addition
ArgMax	Only supported if all of the following hold: - It is the final operation in the network - The preceding output is 2D or 4D - It operates along the channel dimension
AveragePool	- `kernel_shape` < 32 - `strides` < 32
BatchNormalization	No restrictions
Clip	Only supported as an activation function (e.g., ReLU6)
Concat	No restrictions
Constant	Only numeric constants are supported
ConstantOfShape	No restrictions
Conv	**Common constraints:** - `dilations` < 64 - `pads` < 64 - `strides` < 16 **Standard Conv:** - `kernel_shape` < 16 **Depth-wise Conv:** - `kernel_shape` ∈ {[3, 3], [5, 5]} - Only constant weights are supported
ConvTranspose	- `dilations` = [1, 1] - `output_padding` = [0, 0] - `pads` ≤ 14 - `strides` ∈ [2, 8] - `kernel_shape` < 16 - `group` = 1
Div	Supported as: - Constant scalar division - Input normalization

Copyright 2026 DEEPX All Rights Reserved.

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

Copyright 2026 DEEPX All Rights Reserved.

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

4.1.3 Activation Functions

Operator	Supported Conditions
HardSwish	No restrictions
HardSigmoid	No restrictions
LeakyRelu	No restrictions
Mish	No restrictions
PRelu	No restrictions
Relu	No restrictions
Sigmoid	No restrictions
Silu (Swish)	No restrictions
Softplus	No restrictions
Tanh	No restrictions

Copyright 2026 DEEPX All Rights Reserved.

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. Development Tools

5.1 Model Viewer_DX-TRON

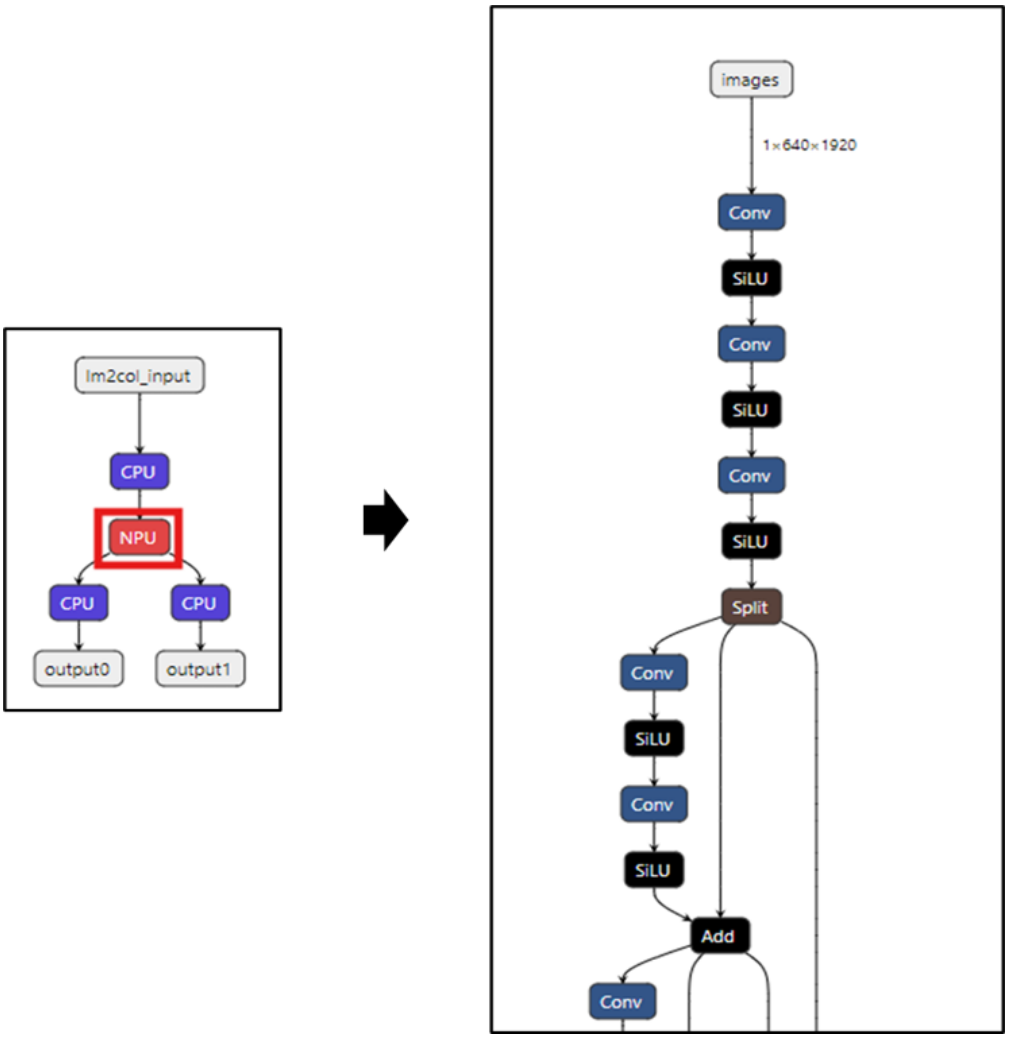
5.1.1 Overview

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

Key Features

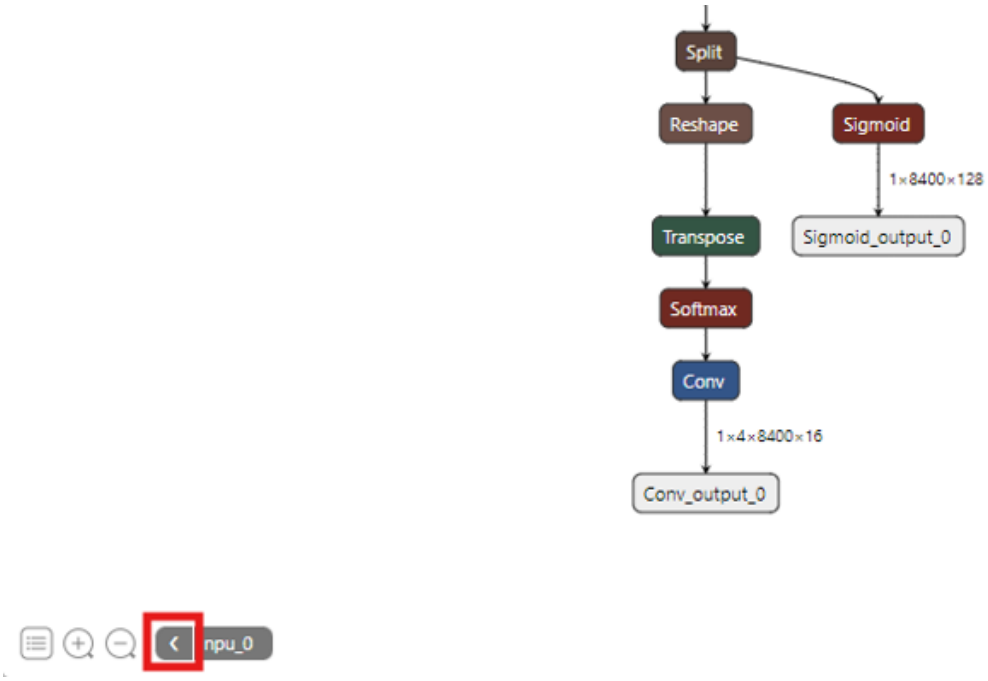
- **Support for .dxnn Files**: Load and visualize model files compiled with the DEEPX toolchain.
- **Visual Workload Representation**: Displays a color-coded breakdown of workload execution:
    - Red: Operations executed on the NPU
    - Purple: Operations executed on the CPU or host
- **Interactive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

Copyright 2026 DEEPX All Rights Reserved.



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

Copyright 2026 DEEPX All Rights Reserved.



5.1.2 Download

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

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

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

5.1.3 Installation on Linux

Supported Systems

Operating System	Versions	Architecture
Ubuntu	20.04, 22.04, 24.04	x86_64 (amd64), ARM64 (aarch64)
Debian	11, 12, 13	x86_64 (amd64), ARM64 (aarch64)

Copyright 2026 DEEPX All Rights Reserved.

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

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

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

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

Method 2: Manual DEB Package Installation

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

Step 1. Extract the Package

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

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

Step 2. Check Your System Architecture

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

Step 3. Install the DEB Package

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

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

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

Copyright 2026 DEEPX All Rights Reserved.

Method 3: AppImage (Portable)

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

Step 1. Install Required Packages

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

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

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

Step 2. Prepare the AppImage File

Make the AppImage executable:

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

Step 3. Run DX-TRON

Execute the AppImage:

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

Or use the convenience script:

```
./run_dxtron_appimage.sh
```


Copyright 2026 DEEPX All Rights Reserved.



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

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

5.1.4 Installation on Windows

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

Step 1. Installation File

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

Step 2. Installation Steps

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

Copyright 2026 DEEPX All Rights Reserved.

The default installation path is as follows.

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

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

5.1.5 Running DX-TRON

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

Desktop GUI Mode

After DEB Package Installation (Linux)

Run DX-TRON using the `dxtron` command:

```
dxtron
```

Using AppImage (Linux)

Execute the AppImage file directly:

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

Or use the convenience script:

```
./run_dxtron_appimage.sh
```

On Windows

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

Web Server Mode (Linux Only)

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

Copyright 2026 DEEPX All Rights Reserved.

Basic Usage

```
./run_dxtron_web.sh
```

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

Using a Different Port

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

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

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

For additional options, run:

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

5.1.6 Troubleshooting

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

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

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

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

Copyright 2026 DEEPX All Rights Reserved.

3) Reinstall DX-TRON

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

Copyright 2026 DEEPX All Rights Reserved.

6. Change Log

6.1 v2.3.0 (March 2026)

DX-Compiler Version

- dx_com : 2.3.0
- dx_tron : 2.0.1

Changed

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

Fixed

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

Added

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

Copyright 2026 DEEPX All Rights Reserved.

6.2 v2.2.1 (February 2026)

DX-Compiler Version

- dx_com : 2.2.1
- dx_tron : 2.0.1

Changed

- None

Fixed

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

Added

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

6.3 v2.2.0 (December 2025)

DX-Compiler Version

- dx_com : 2.2.0
- dx_tron : 2.0.1

Changed

- None

Copyright 2026 DEEPX All Rights Reserved.

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

6.4 v2.1.0 (November 2025)

DX-Compiler Version

- dx_com : 2.1.0
- dx_tron : 2.0.0

Changed

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

Copyright 2026 DEEPX All Rights Reserved.

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).

6.5 v2.0.0 (September 2025)

ONNX Support

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

Model Support

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

Copyright 2026 DEEPX All Rights Reserved.

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

---

6.6 v1.60.1 (June 2025)

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: CLI Execution

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.